1

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS
STANDARD REFERENCE MATERIAL 1010a
(ANSI and ISO TEST CHART No. 2)

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

**The Architecture Development and Design
Considerations of a High-Speed,
Single Chip, CMOS Convolutional Decoder
Based on a Modification of the Viterbi Algorithm**

D. Peter Noel, B.Eng.

A thesis submitted in partial fulfillment of the requirements for
the degree of Master of Engineering at Carleton University

Ottawa - Carleton Institute of Electrical Engineering

Department of Electronics

Faculty of Engineering

Carleton University

Ottawa, Canada

September, 1990

(C) Copyright D.P. Noel 1990

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

ISBN 0-315-60456-5

The undersigned recommend to the faculty of Graduate Studies and Research
the acceptance of the thesis:

**“The Architecture Development and Design Considerations,
of a High-Speed, Single Chip, CMOS
Convolutional Decoder Based on a
Modification of the Viterbi Algorithm”**

submitted by David Peter Noel in partial fulfillment of the requirements for the
degree of Master of Engineering.

Thesis Supervisor

Chairman, Dept. of Electronics

Carleton University

September 1990

Abstract

The rapidly increasing requirements for high speed data communications in satellite and mobile applications are straining existing devices to their limits. The limitations imposed on the developed systems dictate the design goals for future implementations. Radio channel environments associated with the transmission of high speed data often obscure the unprotected signal. A cost-effective means to removing the obscurity from a received signal is to provide forward error correction through the use of an optimal encoding-decoding scheme capable of relatively high speed operation.

A solution providing a high speed encoder-decoder device for reliable data communications is outlined in this thesis. The development, design and simulation of a high speed, single-chip, CMOS Convolutional Decoder based on a derivation of the Viterbi Algorithm is presented in detail. A complementary rate = $1/2$, constraint length 5 convolutional encoder is incorporated into the design to provide a complete forward error correcting codec. Several of the general concepts are as found in several classical Viterbi Decoder implementations but modified to produce optimal performance while operating at frequencies higher than those currently utilized. Other simplifications to accommodate hardware constraints cause a deviation from the original Viterbi Algorithm. To maintain minimum area requirements while achieving the desired operating speeds in excess of 100 MHz, the 1.2 micron CMOS processing technology is the targeted fabrication process.

Acknowledgments

The development of this thesis and the research behind it has been a rewarding and fruitful opportunity. The many friendships formed and experiences gained has certainly created an optimum environment for professional, academic and social advancement. There are many people involved in my inspirations and encouragements.

My greatest foundation has been developed by my wonderful parents, Clyde and Norma. The determination and driving thrust motivating my many long hours and late evenings has been instilled in me by my parents from their endless support and confidence in my abilities. Those evenings of whale-watching and listening to sound advice, the many late night long-distance telephone conversations and endless tokens of their support are very greatly appreciated. I am forever in their debt.

Added to the encouragement of my parents has been the never-ending support from my sister Karen and the many other relatives and friends too numerous to mention. It has been such support that has nurtured the kindling to keep the flame burning these past several years.

Academically, I owe great appreciation to the faculty, staff and students of the Department of Electronics at Carleton University. Dr. Robert Harrison took me under his wing as Faculty Advisor in my early semesters followed by Dr. Tad Kwasniewski who later became my research and thesis supervisor. Dr. Kwasniewski deserves a special note of appreciation and acknowledgment for his many guiding thoughts, lively discussions, Polish History lessons and wonderful friendship throughout my Carleton academic career.

Professionally, I would like to thank the people of Telesat Canada, Telesat Mobile Incorpo-

rated, and Gandalf Technologies Incorporated for making my periods of employment with the respective firms so valuable to my academic, social and personal development. Deserving special acknowledgment are Dr. Sundara Murthy of Telesat Canada, Mr. Arthur Guibord of Telesat Mobile, and Karl Mann of Gandalf. Each of these people have spent time discussing the various technical details of my research and have aided myself in determining the best path of action to take in several instances.

I could not have fulfilled my vision of completing my research and thesis without the understanding of many people. It is said that appreciation is the best reward for services. All of those people that have helped me through my tasks, my sincere appreciation is extended to all of you.

As a special tribute, I would like to dedicate this thesis to the first Engineer in my family, the Late Mr. Lewellin Noel, who studied to become a Civil Engineer between the daily commercial fishing trips. The torch has not fallen nor the flame extinguished. Ours the Task Eternal.

Table of Contents

	<u>Page</u>
Abstract	ii
Acknowledgments	iii
List of Illustrations	viii
List of Tables	ix
List of Abbreviations	x
Chapter 1: Introduction	1
1.1 Introduction to Data Communication Theory	1
1.2 Thesis Motivation and Outline	2
Chapter 2: An Introduction to Coding in Mobile Communications	4
2.1 Mobile Data Communications	4
2.2 Error Control Coding	9
2.3 Previous Implementations of the Viterbi Decoder	21
2.3.1 The Purpose of a Literature Review	21
2.3.2 Existing Implementations	22
2.3.2.1 Implementation A	22
2.3.2.2 Implementation B	24
2.3.2.3 Implementation C	29
2.3.2.4 Implementation D	30
2.3.2.5 Implementation E	33
2.3.2.6 Implementation F	38
2.3.2.7 Implementation G	39
2.3.2.8 Implementation H	42
2.3.2.9 Implementation I	45
2.3.2.10 Implementation J	46
2.3.2.11 Implementation K	48

Table of Contents (Continued)	<u>Page</u>
2.4 Section Summary	49
Chapter 3: The Development of a High Speed Convolutional Decoder	51
3.1 The Requirement for High Speed Decoding	51
3.2 Review of Selected Concepts from Previous Implementations	51
3.3 The Functional Design of a High Speed Decoder	53
3.3.1 Quantizer Design	55
3.3.2 The Decoder Front End	56
3.3.3 The Branch Metric Calculator	60
3.3.4 The Add - Compare - Select Circuit	61
3.3.5 Designing to Prevent Overflow	64
3.3.6 Path Memory	64
3.3.7 The Output Decision Unit	65
3.3.8 The Convolutional Encoder	68
3.4 High Level Simulation	70
3.4.1 A Simple High Level Simulator	70
3.4.2 Simulation Results	73
3.4.3 Comparative Analysis	78
Chapter 4: The Physical Implementation of a High-Speed, Single Chip CMOS	
Convolutional Decoder	81
4.1 Hardware Implementation	81
4.1.1 Introduction to Hardware	81
4.1.2 Required Off Chip Circuitry	82
4.1.3 Decoder Front End	82
4.1.4 The Branch Metric Calculator	85
4.1.5 The Add - Compare - Select (ACS) Circuitry	85
4.1.6 Overflow Detection and State Metric Normalization	86

Table of Contents (Continued)	<u>Page</u>
4.1.7 The Path Memory	87
4.1.8 The Output Decision Unit	88
4.2 Hardware Simulation	89
4.2.1 SPLICE Representation of the Schematic	89
4.2.2 Simulation Results	91
4.3 CMOS Layout	92
4.4 Comparison of a 2 Micron to 1.2 Micron Implementation	94
4.5 Hardware Design Conclusions	95
Chapter 5: Conclusions and Future Work	97
5.1 Thesis Conclusions	97
5.2 Future Work	99
References	101
Appendix A	104
Appendix B	118
Appendix C	146
Appendix D	167
Appendix E	188

List of Illustrations

- Figure 1: Encoder for a Rate = $1/2$, $K=3$ Convolutional Code.
- Figure 2: The Transition Diagram for the Rate = $1/2$, $K=3$ Encoder.
- Figure 3: Trellis Diagram for the Rate = $1/2$, $K=3$ Encoder.
- Figure 4: A Block Diagram of NASA's High Speed Decoding Implementation.
- Figure 5: Viterbi Decoder Block Diagram for Implementation A.
- Figure 6: Block Diagram of the Viterbi Decoder found in Implementation B.
- Figure 7: Correlator Chip Architecture.
- Figure 8: State Metric Chip Architecture.
- Figure 9: Trellis Modification (a) Planar Graph (b) Modified Planar Graph (c) Equivalent Grid Model
- Figure 10: Equivalent Necklace Diagram Derived From Figure 9.
- Figure 11: Deriving a Cube-Connected Cycles Graph. (a) Two Stage Trellis with Recirculation, (b) Restructured Trellis from (a), (c) Grid Model for an 8 Node 2-CCC Graph.
- Figure 12: A Pipelined Cascade Layout of Implementation E.
- Figure 13: Linearly Connected Processor Layout. (a) Functional Block Diagram. (b) Linear Array of Regular Building Blocks.
- Figure 14: The Six State Trellis Diagram for the Code of Implementation G.
- Figure 15: The Basic Decoder Structure for Implementation G.
- Figure 16: Path Memory Layout for Implementation G.
- Figure 17: Viterbi Decoder Block Diagram for Implementation H.
- Figure 18: A Block Diagram of the UCR Viterbi Decoder from Implementation J.
- Figure 19: Block Diagram of the Proposed Viterbi Decoder.
- Figure 20: Quantized Output Channel Representations.
- Figure 21: Block Diagram of the Front End Processor.

List of Illustrations (Continued)

Figure 22: Trellis Diagram for a Rate = $1/2$, $K=5$ Convolutional Code.

Figure 23: Proposed Path Memory Unit Block Diagram.

Figure 24: Proposed Tree Layout of the Output decision Unit.

Figure 25: Proposed Convolutional Encoder Used in the Viterbi Codec.

Figure 26: Flow Chart for the Noiseless Simulation Algorithm.

Figure 27: System Diagram of the Simulated Communication System with Noise.

Figure 28: BER Curves for the Proposed Decoder using the Simple Channel Simulator.

Figure 29: BER Curves for the CRC Proposed Decoder Simulators.

Figure 30: Block Diagram of a Typical Three Bit Quantizer.

Figure 31: Layout Floor Plan for the Single Chip High-Speed Viterbi Decoder.

List of Tables

Table 1: Transition Table for the Encoder of Figure 1.

Table 2: Set of Symbol Metrics for Eight-Level Quantized Data.

Table 3: Transition Table for the Trellis of Figure 20.

Table 4: 2 Micron CU-SPLICE Simulation Highlights.

Table 5: 1.2 Micron CU-SPLICE Simulation Highlights.

List of Abbreviations

ACS - Add - Compare - Select operation.

a_k - the estimated received data sequence.

alog - real valued natural logarithm function.

AND - Logical AND function.

ARQ - Automatic Retransmission Request.

AWGN - Additive White Gaussian Noise.

BER - Bit Error Ratio.

BiCMOS - Bipolar Complementary Metal Oxide Semiconductor.

BPSK - Binary Phase Shift Keying.

BSC - Binary Symmetric Channel.

c_k - Weights or templates associated with decoding probabilities.

CCC - Cube Connected Cycle(s).

CMC - Canadian Microelectronics Corporation.

CMOS - Complementary Metal Oxide Semiconductor.

CMOS/SOS - Complementary Metal Oxide Semiconductor fabricated on a Silicon on Sapphire wafer.

CMOS4 - Complementary Metal Oxide Semiconductor based devices with 1.2 micron channel widths.

cos() - Arithmetic cosine function.

CRC - Cyclic Redundancy Check as used in data communications.

CRC - Communications Research Center.

C/N - Ratio of Carrier Signal Power to Noise Signal Power.

CU-SPLICE - Carleton University Customized SPLICE Simulation Package.

dB - Decibels.

DMC - Discrete Memoryless Channel.

List of Abbreviations (Continued)

E_b - Average Energy per Received Data Bit.

E_b/N₀ - Ratio of Average Received Bit Energy to Average Received Noise Power.

FDMA - Frequency Division Multiple Access.

FEC - Forward Error Correction.

FIFO - First In First Out.

GHz - Gigahertz.

HF - High Frequency.

IF - Intermediate Frequency.

LSI - Large Scale Integration.

MBPS - Megabits per second.

MHz - Megahertz.

ML - Maximum Likelihood.

MSPS - Megasymbols per second.

NAND - Logical NAND function.

NOR - Logical NOR function.

NTT - Nippon Telephone and Telecommunications.

OR - Logical OR function.

phi1 - First phase of a two phase logical clock signal.

phi2 - Second phase of a two phase logical clock signal.

PI - Symbol representing the mathematical value of 3.1414....

QPSK - Quadrature Phase Shift Keying.

RAM - Random Access Memory.

RATFOR - Rational FORTRAN programming language.

RF - Radio Frequency.

ROM - Random Access Memory.

List of Abbreviations (Continued)

SE - Shuffle Exchange.

SPLICE - A circuit level logical simulation tool.

sqrt - The square root function.

SST - Scarce State Transition.

TDMA - Time Division Multiple Access.

TI - Texas Instruments.

TTL - Transistor - Transistor Logic.

UCR - Universal Coding Rate.

VHF - Very High Frequency.

VLSI - Very Large Scale Integration.

z_k - stored data representation.

2-SE - Two dimensional Shuffle Exchange.

Chapter 1:

Introduction

1.1 Introduction to Data Communication Theory

Early data communications systems were of very low speed and quite lightly loaded by present standards. Due to the lower speeds of transmission, the bit and even packet error probabilities were relatively insignificant. If the example of a satellite data transmission link is examined, the lower bit rates usually required only relatively low signal power to achieve a suitable Bit Error Ratio (BER). In addition, the lightly loaded systems caused very little concern for the consumption of the radio frequency (RF) bandwidth or of the amount of the channel actually allocated for the purposes of retransmitting any errored bits or more generally errored packets. As the world of data communications began to expand its horizons in terms of services and applications, a growing need became apparent for more efficient communications services.

End users and service providers alike were no longer were prepared to pay for the lower rates of data transmission which were associated with the out-dated retransmission requirements. Much thought and experimentation has gone into the improvement of data links and error recovery techniques. Access techniques such as Time Division Multiple Access (TDMA) and Frequency Division Multiple Access (FDMA) provided improvements in the shared allocation of the transmission channel. Forward Error Correction (FEC) schemes through the use of encoders in the transmitters and decoders in the receivers were increasing the probability of error recovery at the receiver without the need for numerous retransmissions. Through the use of FEC techniques, the effective carrier signal strength to noise ratio (C/N) of the transmission link could be increased by an amount known as the coding

gain. Such a gain in C/N would offset the uncoded increase in bit errors at higher transmission rates.

One highly efficient method of decoding the received Convolutionally encoded data streams was devised by Dr. Andrew J. Viterbi and today bears his name. An intricate method of stepping through a trellis structure and determining the maximum likelihood (ML) path through that structure to reveal the most probable reconstruction of data sequences containing bursty errors has become known as the Viterbi Decoding Algorithm.

Upon its conception, the Viterbi Algorithm was used in operational research but other researchers soon found it applicable to increasing the reliability of data communications. The original simplex Viterbi Decoders were easily implemented in software and in several cases in hardware utilizing the advanced technologies of the time. Such designs were sufficient for the lower data transmission speeds. Since the original use of the Viterbi Decoding Algorithm for Forward Error Correction, many variations of hardware and software implementations have been presented. In most cases successive decoder designs have shown either some increase in performance and/or decoding rates. The demand to transmit data over a medium at increased rates with improved performance has required current decoder designs to operate at speeds of up to 37 MHz in rare cases. More generally, speeds of up to 10 MHz have been made commercially available.

1.2 Thesis Motivation and Outline

The objective of this report is to give the reader a fair understanding of some of the transmission problems facing communications systems designers of today and to present an encoder-decoder design concept utilizing a high-speed, single chip CMOS Convolutional Decoder. An introduction to data communications systems is given in Chapter 2 followed

by a discussion of interference problems in satellite and mobile radio data communications. Several alternatives are suggested for the correction or reduction of these introduced problems but the application of the Viterbi Decoding Algorithm in Forward Error Correction is the only pursued topic. The details of the operation of the classical Convolutional Encoder and Viterbi Decoder are presented followed by the results of a literature search on currently developed Viterbi Decoders. The designs as presented by several authors are reviewed and noted for their usefulness in contributing to the high-speed decoder design. The actual systematic design of a high-speed Convolutional Decoder derived from the Viterbi Algorithm and the simulation of it are presented in Chapter 3. Results of the simulation are discussed and a comparison is made between the decoder design simulated results and those of a simulation of a previously designed classical Viterbi Decoder. In such a comparison, both decoders are simulated under the same environments such that the proposed design may be evaluated under known controlled conditions. The actual hardware design and its simulation using a customized digital simulator, SPLICE, are presented in Chapter 4. The implementation using a 2 micrometer CMOS technology and repeated using a 1.2 micrometer CMOS technology proceeds the simulation presentation to be followed by a brief outline of a proposed optimal layout. Chapter 5 discusses the Thesis conclusions and possible future work to be performed on the implementation.

Chapter 2:

An Introduction to Coding in Mobile Communications

2.1 Mobile Data Communications

With the growth of the data communications environment, the introduction of industrial and commercial requirements has warranted the creation of effective and acceptable communication standards. Such standards are often not easily achieved in various physical environments yet without their enforcement, many communications systems would not be designed for integration and compatibility. One such standard exists for enforcing an upper bound on the quantity of errors permitted in a block of transmitted data. A major goal in designing an improved data communications system is ensuring that incorporated error recovery techniques will meet or exceed the standard of near error free communications as defined by the application. A relatively low error count may be acceptable in non-critical transactions but in more secure systems the margin allotted to error is narrow.

Data communications has greatly evolved since its conception with the crude system used by Marconi in the 1901 reception of the first trans-Atlantic data message. Following the Marconi experiments the use of the wireless radio gained popularity through use as the original telegraph systems which spanned the civilized regions of the globe. An evolved version of the older telegraph system in existence at present is the continuous wave amplitude modulation system of amateur radio employing Morse Code. The characterization of the data pattern with the telegraph system depends on the ability and technique of the sender. Each character of the Morse Code system is sent by transmitting a series of short and long pulses commonly referred to as dots and dashes. Individual operators have a unique rhythm in sending these coded characters depending on the speed of transmission

and the steadiness of his hand. An operator receiving an audible message would copy the received data by a mental conversion of the dots and dashes to actual characters. Any receiving operator would have to adapt to the style of the transmitting operator and estimate the most likely characters if errors had occurred in the transmission. If the reception quality was so low that estimation of the transmitted sequence was not possible the operator would have to manually send a request for a retransmission. The telegraph systems in use now and in the past used the high frequency (HF) spectrum which is highly susceptible to atmospheric conditions. As a result, solar noise and adverse weather conditions played havoc with transmissions. With time, experimentation led to improvements in the communications systems for dealing with some of the problems of operating environments. Through the use of higher quality material and more refined design/implementation techniques, various system elements such as power amplifiers, filters and antennas may be improved in terms of efficiency and noise reduction. Such improvements permit the integration and implementation of ideas not possible with system components of lesser quality. With the progression of technology, the early telegraph transceiver systems gained the technology permitting automatic detection and correction of errors as well as the use of other transmission techniques and refined spectrum efficiency. Such an evolution of transceivers has enabled the development of the sophisticated radio modems presently in existence or in the design phase.

Generally, a fixed data communications system is equipped with either an omnidirectional antenna (VHF radio systems) or a high gain, directional antenna (satellite or microwave systems). While such antennas would be relatively expensive, the cost of the remaining communications system could be reduced as a result of a reduction in required power and sensitivity due to the increased gain at the antenna. Even with such high quality systems the fact still remains that to get data from point A to point B, transmission through some usually disruptive medium is necessary. In the case of radio communication, this medium will in-

clude the earth's atmosphere and possibly a space environment in the case of communicating with space vehicles or via satellite.

Typically a fixed communication station will communicate with other fixed stations or with mobile stations. The channel used for the transmission of the data will be susceptible to various forms of interference including the inherent problem of transceiver feedback. In a satellite communication system the signal must propagate to and from a satellite in space. Current generations of satellites are generally power limited. This power limitation produces a requirement for more sophisticated equipment utilizing advanced transmission techniques in both the satellite and terrestrial based transceivers to avoid increased signal distortion. An increased user population causes more concern of interference due to the efficient frequency spectrum utilization and the increased power limits. Future generations of satellites will be more bandwidth limited and less power limited than their predecessors thereby changing the emphasis on interference sources. The problems of adjacent channel interference and intersymbol interference will be added to the continuously present problem of noise.

In most communications environments, even under ideal conditions, there is the presence of additive white Gaussian noise (AWGN). This form of noise is purely random in nature, contains noise components of all frequencies and interferes destructively with the transmitted signal. Additive white Gaussian noise is simply added to the signal as given by equation 1.

$$z_i = r_i + n_i \quad (1)$$

The term r_i represents the data signal, n_i represents the additive noise and z_i the summation of signal and noise. Such noise is generated by the thermal agitation of electrons within the components used in the communications systems [1]. The strength of such noise depends on the quality of the material used in manufacturing the components. Based on the quality

of the components, the average noise signal present in a transceiver will exist at a certain level known as the noise floor. The transmitted signal power must be greater than this average noise power if the information is to be successfully detected by the receiver. Another form of noise occurs when a sudden rapid, fluctuation of energy is observed. This is commonly known as impulse or shot noise and is a cause of bursty errors (both long and short) [1]. Such noise may be caused by electrical disturbances in the atmosphere through which the signal propagates or by electrostatic discharge within the receiver. Other interferences result from electronic component non-linearities and imperfect transfer characteristics of certain elements within the transmitter and/or receiver. Such interferences occur as phase and amplitude distortion and possibly frequency offsets or as phase jitter. The addition of most interference sources would create a realistic model but usually an ideal channel model is assumed in test experiments.

A more harsh environment exists in mobile communications. Not only are the problems of the fixed communications link present but several others occur as a result of the movement of the station during the communications process. Minor problems which occur, such as spurious emissions and distortion, as a result of the actual vibration of the transceiver are dealt with by using higher quality components in the transceiver and improved manufacturing processes. There are basically two major problems associated with the mobile communications channel. First the signal to noise ratio for a signal transmitted from a mobile station would be low due to power limitation of the vehicle and to the use of less efficient, low gain antennas. Second would be the problem of fading due to multipath transmissions. Phase and amplitude distortion due to transmitter non-linearities are also present but could be equalized by carefully designing the RF and IF portions of the receiver. A low signal to noise ratio due to the power constraints, combined with the problem of fading requires more intricate detection and decoding schemes at the baseband level.

The reduced signal to noise ratio of the communications system is due to a decrease in the overall gain of the link. To counteract this loss of gain, a method of increasing it without increasing the system power input or quality of system radio frequency components is essential. The main option available for increasing the gain is the use of an intricate coding scheme.

The multipath fading results from the antenna simultaneously receiving the main signal and several time delayed (on the order of nanoseconds) versions of earlier transmitted signals. The time delayed versions are reflections of the main signal from nearby objects such as trees, buildings, hills, etc. The signal detected by the receiver will consist of the sum of the main signal and all other delayed versions of it. Depending on the phase of the delayed signals the addition will be either constructive or destructive. If the signals are in phase (add constructively in terms of waveform amplitude) the detected signal will be distorted in amplitude and possibly somewhat distorted in phase. If the signals are not in phase, the amplitude distortion increases and the already relatively low received signal strength may be reduced to unusable levels. As a vehicle equipped with a mobile data transceiver travels along a roadway, the reflecting objects change thus changing the extent of the fade. The actual velocity of the vehicle will determine the frequency and duration of the fades. If the vehicle is traveling slowly, the fades would last longer thus having a greater effect on the received data. In this case the occurrence of fades is less frequent. At higher vehicle speeds the fades occur more quickly therefore having less of an effect on the data but there will be more fades to contend with.

As the fades will vary in duration from being barely noticeable for low speed data to several bit periods for high speed data some method of adapting the recovery period of the signal or information during these fades is essential. If the information colliding with noise bursts is greater in duration than the noise and the signal is sampled appropriately, then it is highly

probable that the data may be recovered correctly. A means of detecting errors in the received sequence is in providing redundancy of the data at the transmitter. Such a method of adding redundancy to the data stream is encoding. At the receiver the redundancy is removed by decoding. During the decoding process most errors will be detected and possibly corrected. Usually a cyclic redundancy check (CRC) is used to detect uncorrected or undetected block errors. In a data system which does not tolerate errors a CRC is appended to a block of data. The value of the CRC depends on the data within the block to be transmitted. In the receiver, once the decoding has been performed on the block of data and the received CRC, a second CRC value is calculated based on the decoded data. If the received and decoded CRC is not equivalent to the newly calculated CRC then one or more bits within the block are incorrect. Such blocks can only then be correctly recovered by requesting a retransmission.

2.2 Error Control Coding

The idea of providing redundancy of the transmitted bits to overcome the fading problem has become the basis of error control coding. The actual type of coding and decoding may take several forms. Initially the provision of redundancy was provided via a retransmission method. If a block of data was found to have been received in error the receiver would request the transmitter to retransmit the received errored sequence. Originally the transmission was requested manually as in the historic telegraph system but as systems moved to automation, less efficient user intervention in such tasks became obsolete. There are several schemes used for providing automatic retransmission requests (ARQ) [2] but these will not be covered here. The other alternative or complement to the ARQ strategy is that of forward error correction (FEC) [3]. FEC involves converting the data at the transmitter to a certain set of predetermined output symbols which are dependent on the input data sequence. These symbols then provide the redundancy allowing the receiver to detect errors

as the data are received and decoded. With an intelligent decoding scheme the receiver may also correct errored symbols to result in near error free transmission.

Forward error correction has been historically separated into two types of coding [4]. The two types are block codes and convolutional codes. Block codes involve adding some quantity of parity bits to a block of data. At the receiver the data is checked against these parity bits to detect errors. Once an error is detected, an attempt to correct it is made based on the received parity. If the correction attempt is not successful an ARQ is implemented. Block codes will not be discussed further since they are not the main topic of interest in this report and have only been briefly discussed as a matter of completion. The use of convolutional codes is important for this discussion as the method of decoding requires processing the symbols generated by such a code. Convolutional codes use shift-register sequences of data bits at the transmitter to generate the associated symbol sets for transmission. The data used for symbol generation are stored in a K -stage shift register. Shifting a fresh bit into the register will cause the generation of several new output symbols. The generation of a symbol at a particular time stage is dependent on the freshest data bit and the more recent $K-1$ data bits. K is a value denoting the constraint length of the convolutional code being used. The particular transmitted sequences of symbols may only be generated by specific shift register data sequences. Thus, at the receiver the decoding process requires that the detected symbol pairs must reproduce these specific data sequences. To ensure this, the receiver should detect and correct any errors in the decoded data stream if the signal quality has not deteriorated to a level which inhibits the correct detection of data. This detection and correction process is made easier if the number of symbol pairs possible following the last correctly received symbol pair is reduced. This characteristic is available with the use of convolutional codes.

FEC does require more capacity and in general wider bandwidths on the channel and longer

bursts in TDMA schemes (or a higher chip rate in spread spectrum) to achieve the processing gain of a data signal with little or no errors [4]. The main advantage of introducing an FEC scheme is the increase in coding gain achieved but at the expense of increasing the channel bandwidth. With the use of coding, weaker signals can be detected and utilized. Such coding utilization was first demonstrated by Shannon in 1948 [5]. Shannon proved mathematically that error free communication could be achieved with the use of optimum codes, provided the channel interference levels permitted values of E_b/N_0 of -1.6 dB and higher. The ratio of E_b/N_0 is a measure of the ratio of the energy allocated per bit of information to the average noise power spectral density. As Shannon's limit is based on theoretical calculations the best codes to date require an approximate ratio of E_b/N_0 of 11.2 dB higher than this lower limit.

The concept of coding gain is important in FEC. It is defined as the difference in values of E_b/N_0 required to obtain a particular error rate without coding and with coding. Generally an increase in the coding gain requires an increase in the channel bandwidth. The coding gain is also only useful when an increase in performance without coding can be achieved by increasing the power. In most systems a threshold exists at which performance cannot simply be increased by an increase in power. The addition of coding may cause a reduction or elimination of this threshold [4].

With the use of convolutional codes comes the choice of a suitable decoder algorithm in the receiver. In 1966, Dr. Andrew J. Viterbi originated what he termed an "Asymptotically Optimum Decoding Algorithm" [6]. This algorithm was used to obtain the optimal decoding sequence of a Convolutional Code. Several years later Forney described this optimum algorithm [7] with a name that bore that of its originator. In addition to giving a tutorial on the use of the Viterbi Algorithm and a list of its applications, Forney explains that the "Viterbi Algorithm is a recursive optimal solution to the problem of estimating the state se-

quence of a discrete-time finite-state Markov process observed in memoryless noise."

Theoretically the Viterbi Algorithm will optimally detect and correct single bit errors given that the appropriate code is used.

Generally defined, the Viterbi Algorithm operates on sets of received data sequences or symbols to determine the most probable path through a trellis diagram. The received symbols are correlated against certain predefined sets of binary digits representing transitions from one state to another. Such correlations are performed over a certain quantity of continuous branches forming paths the length or depth of the trellis structure. Once the end of the truncated trellis is reached, all paths through this trellis are compared and the most probable one selected. Meanwhile as the final path selection is proceeding, new symbols are being processed by the decoding algorithm to permit continuous operation of the entire process. A Viterbi Decoder will perform the optimum decoding of the symbols on several types of channels [4]. The Viterbi Decoder does not, however, perform well on a bursty channel but when used with an interleaved convolutional code its performance optimality returns. The interleave/deinterleave functions serve to disperse bursts of errors to become several single bit errors.

To understand how a Viterbi Decoder operates, it is first important to understand the operation of a convolutional encoder. To aid in the explanation, a diagram of a convolutional encoder is shown in Figure 1. A rate n/m convolutional encoder creates m output symbols for each n input data bits and requires generally m/n times the bandwidth of the uncoded data. For the encoder illustrated, n has a value of 1 and m has the value of 2. As a bit enters the encoder, its introduction to the data stream causes the creation of a specific number of symbols representing it to be generated at the encoder output. The inputs are shifted through a K (code constraint length) state shift register and until shifted out of the encoder will influence the symbols generated for the proceeding $K-1$ input bits. The symbols are

generated by a modulo-2 addition process. Taps are taken from selected shift register stages and input to the assigned modulo-2 adders. Each modulo-2 adder will create one output symbol. These symbols are then transferred to the transmitter in a specific order and at a rate sufficient to produce m outputs for each n inputs. The movement of data bits into and through the shift register represents the transition from one state to another. After k shifts through the encoder the transitions become repetitive such as to allow the transition diagram to fold onto itself to form a trellis diagram. A list of shift register contents and the associated states for the convolutional encoder of Figure 1 is produced in Table 1. The resultant transition diagram for this encoder is given in Figure 2 with the corresponding trellis diagram shown in Figure 3.

S2	S1	$T_1(X)$	$T_2(X)$	STATE
0	1	1	1	3
1	0	0	1	2
0	1	1	1	3
1	0	0	1	2
0	1	1	1	3
1	0	0	1	2
0	0	0	0	0

TABLE 1: TRANSITION TABLE FOR THE ENCODER

The organization of the trellis diagram is such that transitions from one state to another are characterized by a set of bits. As the destination state depends on the input bit and the most recent $K-1$ shift register bits, the generated symbols, representing a transition through the trellis diagram, are restricted in the number of possible combinations they may assume. Usually the number of bits characterizing a branch is the same as the number of symbols generated by the encoder per input data bit. The encoded symbols are transmitted to the re-

FIGURE 1: ENCODER FOR A RATE=1/2, K=3 CONVOLUTIONAL CODE

generated by a modulo-2 addition process. Taps are taken from selected shift register stages and input to the assigned modulo-2 adders. Each modulo-2 adder will create one output symbol. These symbols are then transferred to the transmitter in a specific order and at a rate sufficient to produce m outputs for each n inputs. The movement of data bits into and through the shift register represents the transition from one state to another. After k shifts through the encoder the transitions become repetitive such as to allow the transition diagram to fold onto itself to form a trellis diagram. A list of shift register contents and the associated states for the convolutional encoder of Figure 1 is produced in Table 1. The resultant transition diagram for this encoder is given in Figure 2 with the corresponding trellis diagram shown in Figure 3.

S2	S1	$T_1(X)$	$T_2(X)$	STATE
0	1	1	1	3
1	0	0	1	2
0	1	1	1	3
1	0	0	1	2
0	1	1	1	3
1	0	0	1	2
0	0	0	0	0

TABLE 1: TRANSITION TABLE FOR THE ENCODER

The organization of the trellis diagram is such that transitions from one state to another are characterized by a set of bits. As the destination state depends on the input bit and the most recent $K-1$ shift register bits, the generated symbols, representing a transition through the trellis diagram, are restricted in the number of possible combinations they may assume. Usually the number of bits characterizing a branch is the same as the number of symbols generated by the encoder per input data bit. The encoded symbols are transmitted to the re-

FIGURE 2: THE TRANSITION DIAGRAM FOR THE RATE 1/2, K=3 ENCODER

FIGURE 3: TRELLIS DIAGRAM FOR THE RATE 1/2, K=3 ENCODER

ceiver at which the decoding process is undertaken. The signal detected at the receiver is demodulated and signal processed to recover the data symbol stream. Once sets of symbols are recovered and routed to the Viterbi decoder, they are compared to the sets of bits representing the various branches of the trellis.

For a Binary Symmetric Channel (BSC) [8], [9] the comparison is equivalent to a calculation of the Hamming distance between the received set and the branch identification label. For a Discrete Memoryless Channel (DMC) with soft quantization, a maximum likelihood approach is taken using a set of probabilities in determining the received sequence to branch label comparison [10], [25]. Although the measure of closeness of the received sequence to the transmitted sequence presented in the original Viterbi Algorithm is determined through Hamming distance calculations, Omura [11] showed that the maximum likelihood calculation (log probability) provides an equivalent representation. This minimum distance of maximum likelihood calculation produces a value representing the node to node transition called the branch metric. A branch metric is calculated for each transition within the trellis. The appropriate branch metrics are then added at each node to a cumulative value representing the old state metric of the node. This updated value represents the metric of the state to which the branch leads. Such a process is repeated on successive sets of received symbols until a path through the truncated trellis diagram has been completed. At this point the path restarts its processing at the beginning of the trellis (wrap around).

Each node has several branches leading to it. In the case of two branches ($n=1$) merging at each node or state, two updated state metrics are determined for that node. A comparison of those state metrics is performed and a decision made to keep the best metric and the path by which it reached this destination node. Each node of the current time stage is now represented by a single state metric value. The paths linking previous stages with each state of the current stage are stored at this point to be used later to determine the most likely path

that the data follows through the trellis diagram. Branches that represent an upper trellis transition select the estimate of the received data as being a logic zero otherwise a lower branch is used in the transition and a logic one is the estimate. This method of tracing a path through a predetermined trellis allows the detection and effective correction of errored symbols. The trace-back process results in the correct data output provided the signal to noise ratio has not been decreased below an error performance threshold.

The algorithm described previously has become one of the most widely used forward-error correction schemes. The simplicity of implementation and the relatively large coding gains results from the ease with which a Viterbi Decoder may utilize demodulator soft decisions to provide approximately 2 dB more gain than a corresponding hard decision Viterbi Decoder [12],[13]. The most significant problem in implementing a Viterbi Decoder is that the complexity increases exponentially with the constraint length K to give 2^{K-1} states. Once the constraint length has been fixed the code may be optimized. Although the increase in complexity with constraint length is a stated disadvantage, it is an advantage over many other codes in that the number of nodes does not increase with the number of input bits.

With the use of new integrated circuit and radio frequency (RF) technologies, the rate at which coded data may be transmitted and received has increased. The increased rate of data transmission, without an increase in transmit power, causes a reduction in the bit energy E_b . This resulting decrease in E_b/N_0 has its effect on the overall bit error ratio. To compensate for the increase in bit errors coding may be incorporated into the system. This will create an increase in effective system gain due to the coding gain achieved. To meet the requirements an encoder/decoder pair capable of operating at high rates is essential. Unfortunately the gap between the rates at which individual decoders operate efficiently and the rates at which data is transmitted has also increased. Much of this has had to do with the technology with which decoders have been implemented.

Using discrete components as opposed to VLSI devices has permitted the achievement of relatively high decoding rates but at the expense of occupied component area. Usually such a non-VLSI implementation has required a high percentage of area in communications systems [14]. In other cases several low speed LSI decoders have been implemented in parallel with the appropriate sampling and switching circuitry to achieve the desired high data decoding speed. Such implementations using LSI decoders have also required a relatively high percentage of available area. To reduce the required implementation area of a high speed decoder and its associated circuitry, it is desired to place as much of the decoding process on a single silicon die. To optimize speed and the chip area used, the utilization of the smallest scale CMOS technology physically and economically available is desirable. The introduction of reduced complexity methods [24] would also aid in the reduction in area and of the decoding delay. The optimum solution to closing the gap between increasing data transmission rates and the actual decoding rates would be the design and implementation of a single chip high speed CMOS Convolutional Decoder (based on a derivation of the Viterbi Algorithm) utilizing reduced complexity techniques. The design objective in decoding rates is 120 megasymbols per second (MSPS). To complement the high speed decoder a high speed convolutional encoder must be included on the silicon die.

A previous application of a Viterbi Decoder implementation which inspired the criterion of high decoding rates in the neighborhood of 120 MSPS was presented by McCallister and Crawford [15]. The National Aeronautic and Space Administration (NASA) required forward error correction in its 30/20 Gigahertz (GHz) program to operate with an input capable handling a data rate of 750 MSPS. The program involved placing a geosynchronous satellite in orbit capable of handling TDMA information bursts with a throughput in excess of 4 GHz. The FEC scheme used a Viterbi Decoder arrangement. At the time of the system design, decoders were available at relatively low decoding throughputs. A block diagram in Figure 4 illustrates how the 750 MSPS data is demultiplexed into three streams followed

FIGURE 4: A BLOCK DIAGRAM OF NASA'S HIGH SPEED DECODING IMPLEMENTATION

by each stream undergoing a serial to parallel conversion (16 bits) before the decoding section. Each of the 48 (in total) parallel outputs are routed to corresponding parallel Viterbi Decoders. The decoded data (48 bits) are then parallel to serial converted back to three streams and multiplexed into one data output stream. The decoders involved receive data at 15.625 MSPS thus resulting in the system requiring 48 decoders. If this decoding speed could be increased significantly the required area and complexity would be reduced. Achieving a decoding rate of 125 MSPS for example, would reduce the number of decoders required from forty-eight to six. Other examples [14] exist to warrant the design of a decoder, based on a deviation from that of the classical Viterbi Decoder, with a decoding rate in the range of 100 MSPS but are not reviewed here.

2.3 Previous Implementations of the Viterbi Decoder

2.3.1 The Purpose of a Literature Review

As an aid to the design of an optimal Viterbi Decoder, several previous implementations have been reviewed. Most of those examined have been implemented over the last several years using either CMOS or fast TTL technologies. Experience has shown that several optimum designs have resulted from reviewing previous methodologies and being aware of the advantages and short-comings of existing devices. The design of a near optimal Convolutional Decoder derived from classical Viterbi Decoders is no exception and has benefited from a culmination of modified ideas as may be gathered from the literature review. Over the next several pages, several currently existing Viterbi Decoder implementations are reviewed in detail. Ideas and concepts are collected throughout the review to be analyzed at the end of this chapter and again in further detail in Chapter 3 for possible contributions to an improved decoder design.

2.3.2 Existing Implementations

2.3.2.1 Implementation A

Cain and Kriete [16] describe the design of a rate=1/2 constraint length $K=7$ Viterbi Decoder. The block diagram of this implementation is illustrated in Figure 5 for reference during the discussion of it. Initially an input formatter section determines the beginning of a branch in the received signal stream and resolves symbol polarity. The formatter monitors the merging properties of the trellis. When the decoder is out of synchronization the surviving paths of the trellis merge at a slower rate than when the decoder is synchronized. Based on an estimate of merging paths for the utilized code, the input formatter will attempt to resynchronize the data stream if a retarded merge rate is detected.

In the next step, the branch metric computer determines a new set of metric values for each of the different branches in the trellis as each new pair of data is received. Following this, the survivor metric calculator function represents the add-compare-select (ACS) unit. It contains two adders followed by a comparator and a storage element. The ACS adds newly determined branch metrics to old state metrics then selects and stores the better of the two metrics. The path storage and output sequence selection are implemented by a "trace back method". With this method the actual information sequences are not stored but instead the comparison results from the ACS units are saved. After several branches have been processed the stored trellis connections are recalled in reverse order and the "most likely" path traced back through the trellis diagram. The "most likely" path will represent the estimated received data. The implementation as described has achieved a 10 MBPS decoding throughput.

FIGURE 5: VITERBI DECODER BLOCK DIAGRAM FOR IMPLEMENTATION A

2.3.2.2 Implementation B

Frenette [12] has implemented a Viterbi Processor for a time-dispersive channel within a digital communications system. Here the Viterbi Algorithm was implemented using a “brute force” method with 4 to 5 micrometer CMOS. The achieved data rate for this implementation was 2400 bits per second. The Viterbi processor was divided into three sub-processors under the partitions of correlation, state metric evaluation and state decision making. These partitions are outlined in the block diagram of this processor illustrated in Figure 6. Due to the implementation area required with the CMOS technology used, each of these sub-processors were placed on a separate silicon die.

The correlator is responsible for calculating the branch metrics. It accepts the quantized matched filter output (input data) and nine channel estimator coefficients as well as the label of the state being considered. The output of the correlator is a twelve bit transition metric corresponding to the input state.

Present state metrics are computed by state metric evaluation using the calculated branch metrics and old state metrics. This is performed by the second circuit which receives the twelve bit transition metric from the correlator. This input is added to the old state metric in the updating process. The state metrics for time $t-1$ (the old state metrics) are stored as sixteen bits to provide for accuracy and overflow as it is cumulative over the period of operation. The outputs for the state metric block consist of a one bit signal to specify the chosen path and the associated updated sixteen bit state metric. Finally the state decision unit determines the correct output. The inputs to the decision unit are the one bit signals and the best sixteen bit state metrics of the current stage. The output is the estimated transmitted sequence based on the state metric inputs. All data preparation such as filtering, quantizations, clock and bit recovery for the three chip Viterbi Decoder are performed off-chip.

FIGURE 6: BLOCK DIAGRAM OF THE VITERBI DECODER FOUND IN IMPLEMENTATION B

The correlator and branch metric evaluation function will now be covered in greater detail. For this discussion of the correlator chip, reference is made to Figure 7. The channel weights and the data are transferred serially to an eight bit wide ten-word long shift register. Once shifting is complete, each cell of the shift register becomes memory storage. The stored data z_k and weights c_k are fed to ten complementers controlled by a_k (the estimated received sequence). With each cycle a new estimated sequence is fed to the correlator thereby creating a new set of inputs to the adders. The inputs are summed in pairs, the result of which are summed in subsequent adder levels until a final total is obtained. This total is multiplied by a_k using a twelve bit complementer. Two non-overlapping clock phases ϕ_1 and ϕ_2 are used to control the pipelining operation of the binary correlator. While a set of conditionally complemented inputs are being added in the first level of adders, a new set of a_k 's are fed to the complementers. The gating of the connections between adder levels by the two phase clocks enable one level to sum the previous subtotals while the previous level completes a new subtotal. Once all levels are filled with data the correlator outputs one transition metric per clock. Sign extension is used throughout the levels of adders to prevent overflow.

A simplified block diagram of the state metric chip is shown in Figure 8 for reference in the following discussion. On each clock cycle, a transition metric and a state metric are read into the demultiplexer. Using the first half of the clock cycle, a transition metric from the correlator chip is read into the demultiplexer. During the second half of the cycle, a state metric is read from memory through the demultiplexer into the adder. Also during this half cycle the twelve bit transition metric is sign extended to sixteen bits and sent from the demultiplexer to the adder. This sum is labeled Path A and stored in a register. This operation is repeated for the second transition metric leading to the desired state. This result is labeled Path B and is stored in a second register. These two results are fed into a comparator which performs the function $\text{Path A} - \text{Path B}$. The output of the comparator is used as a control

FIGURE 7: CORRELATOR CHIP ARCHITECTURE

FIGURE 8: STATE METRIC CHIP ARCHITECTURE

signal to select the surviving path. The smallest path is the survivor and is sent on to the Adjust operator. The Adjust operator protects against overflow of the state metrics. After the Adjust operation the resultant state metric is written to memory and passed to the output decision chip. This chip was not discussed in the literature reviewed as problems had arisen in its CMOS implementation.

2.3.2.3 Implementation C

Two alternatives to the "brute force" approach of implementing a Viterbi Decoder were proposed by Gulak [13]. These alternatives were based upon what has been termed the VLSI Grid Model. This model may be compared to a computation graph which has vertices as nodes and edges as wires. The nodes represent the processors for computing Boolean Functions and the wires represent the electrical connections responsible for the transfer of information, power and timing. Such models attempt to achieve a lower bound on the product of (chip area) * (baud rate)⁻² and on the energy consumption of a VLSI implementation.

The Viterbi Algorithm is not demanding as it is considered complex in the conceptual sense only. It is a simple procedure of applying add, compare and select operations to a large number of states. Various systolic array implementations are compared by the means of the VLSI grid model which determines the relative area required to implement each concept. The use of established theoretical routing methods such as the shuffle exchange and cube connected cycles require a large wire area when implementing a Viterbi Algorithm using systolic arrays. Such arrays with the appropriate routing permits relatively high decoding rates while trading off die size. Shuffle exchange (SE) graphs were originally used for parallel computation. It has been shown that a single stage of a trellis diagram will map naturally onto the SE graph. The shuffle exchange contains 2^v nodes corresponding to the nodes of a trellis diagram. The v term represents a value of one less than the constraint length.

The nodes of the trellis diagram are given a binary representation according to their position. The shifting of the originator's binary sequence left or right by one cycle results in the label of the node above or below it. Two nodes will be linked by a shuffle edge if the transition from one to the other requires a left or right cycle shift of the originator's binary sequence. The two nodes may be linked by an exchange edge if their representations differ only in the first bit. A stage of the trellis diagram may map into a SE graph in which each initial state would map into two states. The transition from an initial state to the two final states would consist of a perfect shuffle operation followed by an exchange operation. The cycle of shuffle edges is called a necklace. By cleaving the architecture into such necklaces the layout of a VLSI Viterbi Decoder may be simplified.

To illustrate the idea of necklace formation refer to Figure 9. A single stage trellis with recirculation edges may be rearranged as a planar graph as seen in Figure 9a. The nodes serving as only wiring distribution points can be split to form the modified graph of Figure 9b. An equivalent grid model is created in Figure 9c by drawing heavy lines on the bidirectional edges containing the nodes. The dashed line represents the necklace and the node represents the ACS units. Figure 10 is an illustration of a binary SE or a 2-SE graph derived from the necklace formation. It can be modified to accommodate an M symbol alphabet. Knowing the algorithm memory length and alphabet size, an efficient layout may be created based on the above discussion.

2.3.2.4 Implementation D

The second architecture, proposed by Gulak et al [13], consisting of the cube connected cycles (CCC) structure, uses even more wire area than the shuffle exchange. The CCC network consists of several sets of pipelined processors. Outputs from the last set of enabled processors are fed back to the first set of processors. Each row of processors simulates one

FIGURE 9: TRELLIS MODIFICATION (a) PLANAR GRAPH (b) MODIFIED PLANAR GRAPH (c) EQUIVALENT GRID MODEL

FIGURE 10: EQUIVALENT NECKLACE DIAGRAM DERIVED FROM FIGURE 9

of the stages in a trellis diagram. Only one row of processors are active during a symbol interval. A processor is basically responsible for all computation associated with its state. With this architecture, the wiring scheme is fairly simple and regular while allowing the processor to deliver information at the next baud period to the required processor in the next stage. If connectivity is not correct, state metrics would require extra cycles to shift through the system thereby degrading system performance.

The CCC structure with its $v * 2^v$ nodes implements v stages of the trellis diagram including recirculation. This is shown in Figure 11 for $v=2$. By rearranging the location of processors (nodes) a series of cube connections and cycle connections may be performed. A cycle connection results in a transition from a state to a second state and returning to the original state. Such a connection constitutes a cube and connections between cubes result in cube connections. The described CCC structures have optimal area properties. Overflow control is achieved in these CCC implementations by subtracting a constant from the state metric on every (or alternating) baud periods.

2.3.2.5 Implementation E

Several approaches to implementing the VLSI architecture for a Viterbi Decoder by Gulak and Kailath [17] states that "to efficiently utilize the full capabilities of VLSI, efforts are required to devise algorithms capable of doing so". In an architecture presented, a parallel processing approach known as a perfect shuffle was taken. By defining the state sequence of a shift register process by a de Bruijn graph, an estimation of it using the Viterbi Algorithm may be performed using a parallel processor. This processor has an interconnection network defined by a shuffle exchange (SE) graph. Using this SE approach a Cascade Layout and a Linearly Connected Layout are presented.

(a)

(b)

(c)

FIGURE 11: DERIVING A CUBE-CONNECTED CYCLES GRAPH. (a) TWO STAGE TRELLIS WITH RECIRCULATION, (b) RESTRUCTURED TRELLIS FROM (a), (c) GRID MODEL FOR AN 8 NODE 2-CCC GRAPH

The first architecture discussed was that of the cascade layouts of which an example is shown in Figure 12. The Viterbi Algorithm can be adapted to run on $\log_2 N$ processors (N is the number of states in the trellis) each associated with a local memory. Each processor contains two ACS circuits arranged pairwise in a butterfly configuration. With each processor are two associated auxiliary first-in first-out (FIFO) queues and a programmable switch. The total FIFO memory in the system is proportional to N . Each word of this FIFO stores a state metric and the respective survivor sequence. The switches, capable of two modes of operation coordinate unidirectional information transfers between processors and FIFOs. One mode will permit data to pass straight through the switch while the other mode will criss-cross the operands.

With each symbol interval the circuit accepts a channel output symbol as an input into the FIFO queue. The operands are routed from this queue to the appropriate branch metric operator. Each branch metric generator provides a set of four branch metrics during each valid processing cycle. Normalization of state metrics by subtracting the first path metric generated in a group of 2^{v-1} processing cycles ($v = K =$ the code constrain length) control register overflow. Maximum likelihood estimates of the transmitted data sequence are available from a truncated survivor sequence.

Cascade Layouts [17], being similar to a serial uniprocessor scheme, are determined to be inferior to parallel layouts. Linearly Connected Layout patterns have a speed advantage over the Cascade Layouts but require more die area. The linear connected scheme shown in Figure 13 consists of three rows of processing cells with each element connected in parallel to its neighbor. The center row of cells form the backbone of the decoder with the top row being the sequence control generators creating one control signal for each processor in the center row. The bottom row accepts the symbol inputs and generates the branch metrics destined for the center row. Each cell in the center row contains routing logic circuitry, a

FIGURE 12: A PIPELINED CASCADE LAYOUT OF IMPLEMENTATION E

FIGURE 13: LINEARLY CONNECTED PROCESSOR LAYOUT. (a) FUNCTIONAL BLOCK DIAGRAM. (b) LINEAR ARRAY OF REGULAR BUILDING BLOCKS.

path metric register, survivor sequence and add-compare-select circuitry. The center row uses the branch metrics to compute state metrics ultimately and choose the best possible solution for the output. The linear architecture proposed rely on either systolic arrays or perfect shuffles to perform the Viterbi Algorithm.

2.3.2.6 Implementation F

Clark and Clayden [18] describe work done in reducing the number of operations per data symbol and storage area required without losing the highly structured form of the original Viterbi Detector. Maximum likelihood detectors operating with multilevel data symbols in the presence of additive white Gaussian noise detect data symbol values that are the same or close to those given by a sub-optimum detector such as a linear or nonlinear equalizer. Based on theoretical and experimental studies, the authors claim that “An equalizer may therefore be used as a part of a near-maximum likelihood detector. This detector could be used to give an initial sequence of detected symbols from which values could be determined for the use of the Viterbi Detector.” Such an arrangement would greatly reduce the number of combinations of data symbols which leads to a considerable reduction in the complexity of the Viterbi Detector. To exploit the reduced complexity the detector first recodes the received data symbols from 16 quantized levels to binary and then operates on the recoded symbols as would a conventional Viterbi Detector.

The symbols could be reduced to 3 or 4 level symbols but such reduction would not permit as great a reduction in decoder complexity as would reduction to 2 levels. As with the conventional Viterbi Algorithm, a metric or cost is associated with each transition resulting from new data received. The principle behind a detection process based on or derived from the Viterbi Algorithm is that prior to the receipt of a data symbol the detector has stored m different vectors and the associated costs. Upon receipt of the new data symbol each stored

vector is expanded into 16 vectors (the first $k-1$ components are as in the original unexpanded vector and the last component takes on one of its 16 possible values). The cost of each vector is evaluated giving $16 \cdot m$ vectors and costs. m vectors are selected from these $16 \cdot m$ vectors according to the provided selection criterion. At the end of the received signal message the vector with the lowest cost is taken as the detected value to be the decoder output. This process compares to the path metric memory and output decision units of a conventional Viterbi Decoder.

2.3.2.7 Implementation G

Stahl [19] discusses a somewhat conventional method of implementing a rate= $2/3$, 12 MBPS Viterbi Decoder. The code used creates a trellis with six states as shown in Figure 14. According to Stahl et al, such a code achieves a coding gain of 3 dB and creates better acquisition performance for the signal phase than would the equivalent Ungerboeck [20] code. This Viterbi Decoder is divided into three parts, shown in Figure 15, representing the transition (branch) metric computation, the add-compare-select (ACS) unit and the survivor memory. The units are pipelined as much as possible to allow the fastest possible decoding speed for the technology (TTL) used. The transition metric unit calculates the transition value of each symbol through each branch of a trellis diagram. This can be done in eight parallel working units as there are only eight different transition values for the entire trellis. Following this, each ACS unit performs two additions (two transition metrics are added separately to the state metric from the previous stage) and a single comparison (of the two additions) followed by a selection of the best state metric. In a fully parallel operation as incorporated in this implementation, a single ACS unit is required for each state of the trellis resulting in six ACS units.

Overflow protection of the state metrics is provided by computing the largest transition

FIGURE 14: THE SIX STATE TRELLIS DIAGRAM FOR THE CODE OF IMPLEMENTATION G.

FIGURE 15: THE BASIC DECODER STRUCTURE FOR IMPLEMENTATION G.

metric and subtracting this value from all state metrics. The operation has to be performed in one symbol cycle to restrict the range of the state metrics. To avoid slowing the decoding speed, only the highest bits of the state metrics are monitored. Once these bits exceed a certain value the correction value is subtracted from all state metrics.

The normalized state metrics are transferred to the path metric survivor memory. This memory will store the surviving state metric sequences for a path of up to a length N . The value of N depends on the structure of the code being used. A specially arranged memory consisting of cells as in Figure 16 is used to avoid excessive time delays or use of RAM. The oldest metrics are then recalled to be used in deciding an output.

2.3.2.8 Implementation H

Orndorff [21] of Rockwell International implemented a VLSI Viterbi Decoder capable of achieving a BER of 10^{-5} at 4.3 dB net coding gain capable rates of up to 37 MBPS. The scheme uses a rate = $1/2$, constraint length $K = 5$ convolutional code. The implementation was similar to that of Stahl [15] but used small geometry (2 micrometer) CMOS/SOS technology incorporating 33,109 transistors. A block diagram of the implementation is given in Figure 17. Received data first passes through four 3 bit adders in the branch metric circuitry. The 4 bit sums are then distributed to the 16 ACS circuits (one for each state). Each ACS consists of two 8 bit adders, an 8 bit magnitude comparator and a 9 bit selectable gate register. The outputs of the ACS units form a closed loop as they are fed back to the ACS inputs. The magnitude comparator of each ACS passes its decision to 16 path memory registers interconnected in a trellis manner where the maximum likelihood voting occurs. Each path memory register is 25 bits long providing the maximum likelihood of error correction. Final voting at the output of the path memory determines the output data bit.

FIGURE 16: PATH MEMORY LAYOUT FOR IMPLEMENTATION G.

FIGURE 17: VITERBI DECODER BLOCK DIAGRAM FOR IMPLEMENTATION H

2.3.2.9 Implementation I

A design group at NTT developed a VLSI scarce-state-transition (SST) Viterbi Decoder [22] for a rate = $1/2$, constraint length $K = 7$ convolutional code. The design achieved a throughput of 23 MBPS with a coding gain of 4.4 dB at a BER of 10^{-4} . The design has been implemented in 1.5 micrometer CMOS requiring a die size of 9.52 X 10.0 square millimeters. The SST Viterbi Decoder uses an SST block consisting of a pre-decoder, a re-encoder and modulo-2 adders at the input of a conventional Viterbi Decoder. The SST receives encoded data (including errors) which are decoded by the simple pre-decoder. These pre-decoded symbols are re-encoded using the same type of encoder as found in the transmitter. The re-encoded data are then compared to the received data in the modulo-2 adders and the output of these adders/comparators is transferred to the Viterbi Decoder.

The pre-decoding and re-encoding operations in the SST block ensure that the signals which are input into the narrow-sense Viterbi Decoder are composed of only errors. If the channel is error free only logical zero signals are input to the Viterbi Decoder. As the SST Viterbi Decoder requires only the final bit of path memory corresponding to the zero state the maximum likelihood-decision circuit of the conventional Viterbi Decoder and the section of path memory cells having no connection with the final output bit may be eliminated. This leads to a significant hardware reduction at the cost of adding the SST block.

The ACS block, being the path metric computer, consists of a branch metric computer and sixty-four state metric computers. The branch metric computer is made of four three-bit adders. The state metric computers contain two six-bit adders, a comparator and a selector as in the conventional Viterbi ACS circuits. The path memory block renews and stores survived paths based on computed state metrics. The memory circuit consists of 2^{K-1} (number of states) by T path memory cells. T is the path memory length, being 40 in this implemen-

tation. As several path memory cells may be eliminated in the SST Viterbi Decoder, an effective path memory length of twenty-nine would achieve performance equivalent to a value of forty for T.

2.3.2.10 Implementation J

A second design by the NTT group was of a universal coding rate (UCR) Viterbi Decoder [23]. This decoder consisted of a Branch Metric Calculator (implemented in ROM), a speed converter and a conventional low coding rate Viterbi Decoder. A block diagram of the decoder is shown in Figure 18. A high speed convolutional encoder is used at the transmitter to create the data symbols. Received data symbols are fed to the UCR Viterbi Decoder. These symbols are routed to the ROM based branch metric calculator. The newly calculated branch metrics from ROM for these symbols are passed to a memory based fast-in fast-out speed converter. The channel rate branch metrics are converted to source signal rate metrics which are passed to the ACS units of the UCR Viterbi Decoder. Once the ACS processing has been performed the selected signals from the ACS units are conveyed to the path memory circuits. The decoder output is chosen by selecting the best bit from the last bit in each path of the memory.

These ACS and path memory functions, requiring relatively large areas were implemented in separate packages. As mentioned previously, the branch metric calculator and the speed converter were implemented in off-chip memory thus requiring that the system have extra components. The UCR Viterbi Decoder could be applied to all rate $r=(n-1)/n$ convolutional codes due to the flexibility warranted by off-chip components. By changing the branch metric calculation table and the speed conversion rate of the FIFO memories the adaptation process is completed.

FIGURE 18: A BLOCK DIAGRAM OF THE UCR VITERBI DECODER FROM IMPLEMENTATION J.

2.3.2.11 Implementation K

Research by Simmons [24] lead to a proposal to reduce the complexity associated with the Viterbi Algorithm. A conventional Viterbi Decoding Algorithm has a fixed level of computation which is designed to deal with worst case channel conditions. By adapting the decoding effort to that actually required during periods of less severe conditions, the decoding speed of the device may be increased at the expense of a modest increase in buffer size and decoding delays.

An optimum conventional Viterbi Decoder chooses a path through the trellis from an initial state which has the best metric. The proposed algorithm avoids an exhaustive inspection of all paths by discarding the paths at any depth K in the trellis that have a relatively high metric. Path metrics accumulate with time and comparing them to a biased threshold tied to the depth in the trellis is a possibility for reducing the complexity of the algorithm. A better alternative is to sample the noise conditions by determining the best metric for any particular time period. The discard rule then may use this best metric as a basis for elimination of less likely choices. A path is discarded if the difference between its metric and the chosen best metric is greater than some fixed threshold.

The decoder output path is derived from the common portion of the path histories. At the path length L , if all paths have not merged, the best strategy for deciding the decoder output is to take the best decision path branch to be that corresponding to the path in the storage with the lowest metric. All unmerged paths at this point may be rejected.

The advantage to this algorithm is the reduced number of computations required due to an early elimination of selected paths. The implementation of this would physically occur within the ACS circuitry to permit the immediate removal of these expired paths from any

successive processing. While Simmons does not discuss a hardware implementation, the algorithm would easily be designed into hardware.

2.4 Section Summary

The knowledge gained through the literary search of the Viterbi Decoder implementations has provided the required insight for designing a high speed optimal decoder while avoiding the pitfalls of rate determining processes within the hardware. While the review is not exhaustive a representative cross-section of implementation methods have been included. Several attributes from the above decoders are possible candidates for inclusion in the design process to be pursued. All reviewed implementations discuss the basic architecture of the Viterbi decoder being the metric computation, add-compare-select, survivor memory and output decision. Many designers have different methods for implementing each unit and in formatting the input data. Data and frame synchronization methods are design dependent and will be original. Other functions are borrowed from the review and modified to increase the operability of the near optimum Convolutional decoder.

There appear to be several contenders for implementations optimizing speed, area and power consumption. The better implementations for area minimization would be one of the systolic array approaches (either the cube connected cycles or shuffle exchanges). These systolic implementations are not noted for high throughput and to gain a processing speed advantage the more "brute force" approach is necessary. The term "brute force" refers to a sequential/parallel layout in which the processing units (metric calculator, ACS, memory, etc.) follow each other sequentially with as many data streams in parallel as required to meet the throughput requirements. The correlator approach of [12] to branch metric generation, though a novel implementation, would present a bottleneck to further processing. Corresponding state metric generation hardware, however, lends itself, with modification

to avoiding such throughput impedances at this stage. The complexity reduction of [18] would not be pursued in an optimal design as reducing the quantization from sixteen levels to two levels results in a loss of coding gain. Stahl [19] has a simple yet efficient approach to the brute force implementation. Many ideas were generated by his research presented in the paper. The decoder discussed by Stahl is for a rate = $2/3$ convolutional code as opposed to a rate = $1/2$ code. A rate = $1/2$ convolutional code and the appropriate decoder is more desirable due to the simplicity advantage of a rate = $1/2$ code over a rate = $2/3$ code. Stahl's methods of pipelining, modular circuit grouping and trellis fashioned memory arrangement are highly efficient and shall be favorably considered. Orndorff's review [21] represents the rate = $1/2$ decoder and has contributed much to the implementation of the building blocks for the single chip CMOS decoder implementation. The designs by NTT [22],[23] discuss the use of several off-chip components such as RAM and ROM based path memory and state metric generation. Such external components are not desirable as these would, while reducing the silicon real estate requirements, increase the overall system area requirements and reduce the operational speed of the design. The principle of the NTT scarce state transition decoder design could experience potential timing problems in addition to increasing system complexity and silicon die size. It will not influence the design significantly.

While the above discussion merely eliminates several possible ideas in choosing the more suitable methods for high speed operation, the optimal properties are not listed. Further detailed analysis of the final selection of the previous implementations appear in the following chapter. Concepts and methodologies considered acceptable must be modified in implementation to permit the high speed advantage required once the CMOS fabrication of the entire circuit is considered. Such modification will generate a portion of the speed increment but the overall rate will be determined by the CMOS technology of fabrication. Ideally the CMOS technology yielding the lower device propagation delays would be desired for use in fabrication.

Chapter 3:

The Development of a High Speed Convolutional Decoder

3.1 The Requirement for High Speed Decoding

Recognized in several of the Viterbi Decoder implementations discussed in the previous chapter is the requirement of decoding speeds in excess of 100 MSPS. Such requirements exist in areas of space communications involving data collection and relaying of such data either via or from a space based platform. The large quantities of data to be transmitted over a short time frame would delegate a high transmission speed. To date, to achieve such transmission rates, several individual decoders have been arranged in parallel with the received data being demultiplexed and transferred to the appropriate decoder. Each decoder of the parallel arrangement operates on received data to generate most likely bit estimates which are then multiplexed into the correct output order. Such parallel implementations generally utilized much area and if placed on a satellite or in a portable receiver, the weight was of concern. To meet some if not all of the requirements of such data communications systems a high speed Viterbi Decoder implemented with minimum complexity and minimized area while producing near optimum performance is desired.

3.2 Review of Selected Concepts from Previous Implementations

In comparing the reviewed literature for deciding upon the best method of implementation, several points were noted. Implementation by systolic arrays would not provide the required speed of operation nor be easily implemented for optimum performance at minimum area. The systolic array implementation for near optimum performance would require a rather large relative area and is not deemed to be acceptable for the proposed criterion. The

“brute force” method of creating branch metrics, state metrics and using somewhat traditional ACS (add-compare-select) circuitry would allow improvements in reducing the complexity while reducing required implementation area and increasing processing speed. Serial implementations tend to require less processing elements but more processing clock cycles to perform certain functions. Such a configuration would meet the established criterion in terms of reduced implementation area but would fail in meeting the decoding speed requirements. Parallel arrangements would be better suited to the application of Viterbi Decoding which contain inherent parallelism by nature. The parallel configuration will permit the same operations to be performed simultaneously on sequential data. This would require slightly increased area while permitting high speed operation. In view of the literary research, a parallel type “brute force” method is the better method for implementation to achieve the design goals.

In designing a high speed Convolutional Decoder based on the Viterbi Algorithm but with deviations from it, several ideas from the implementations reviewed in Chapter 2 have been utilized. Branch metric generators similar to that as used by [12] would create the branch metric values at a rate as required in the high speed design. The hardware operation is considered while a modification to the maximum likelihood process is performed. That particular implementation is relatively simple thus maintaining the minimum area constraint. The add-compare-select (ACS) designs of most of the non-systolic array implementations are quite simple but yet appear efficient in area utilization and operational speed. The adaptation of modified, existing ACS designs to the proposed decoder would prove optimal. When dealing with the path memory, the problems of speed and interconnection complexity are of importance. The memory cells must be constructed such that the estimated data will pass through each stage and out of the final memory stage of the path at the same rate as the generation of the estimated data. The interconnection of such cells must be possible in the proposed CMOS process. The memory cell design of [21] is superior to the other de-

signs reviewed as this design exhibits the best speed advantage due to the avoidance of over-processing of data to obtain the correct path format. The individual memory cells of this design are interconnected in a trellis manner (as is the branch metric generator to the ACS units). This will permit simplified nearest neighbor routing between the cells thus improving the area minimization while reducing the interconnection propagation delay. Reduced complexity methods must be incorporated to reduce the processing to a minimum while not degrading the performance significantly. Several other sections of the design such as the input formatter and the output decision unit are specific to this design and have not used ideas from the references. One exception to this previous statement is the use of a tree structure in the output decision unit.

Again the importance of adapting all acceptable ideas to formats required to permit fabrication in advanced small geometry CMOS process must be stressed. Failing such adaptation would result in not achieving the stated design goals of minimum area and power consumption while permitting optimal functionality.

3.3 The Functional Design of a High Speed Decoder

As discussed in the literature analysis at the end of the previous chapter several ideas used in previous implementations have contributed to the design of a high speed Viterbi-like Convolutional Decoder. The adjective "Viterbi-like" is used to indicate that while the Viterbi Algorithm is followed in the design there are several salient deviations from it which reduce the optimality of its performance. The system level design is considered first as represented in the block diagram of Figure 19. A decision has been made to utilize a code generated by a rate = $1/2$, $K=5$ Convolutional encoder. This encoder is described at the end of this section. The data synchronization and clock recovery circuitry involves operating on the received demodulated data signal. The clocking signal is recovered by means of level

FIGURE 19: BLOCK DIAGRAM OF THE PROPOSED VITERBI DECODER

detection (i.e. zero crossing, peak signal, etc.) of the demodulated data signal. The detected data is quantized to eight levels and transferred to the input of the Viterbi Decoder. In addition to the clock signal and quantized data, the decoder utilizes a framing pulse as the received data symbols are required to be grouped in sets for the decoding process. In the case of symbols being grouped into 2 symbols per set as with the rate=1/2 convolutional code being used, the framing pulse must come at the beginning of the pair. Such a framing pulse is required at distant intervals to ensure that the decoder has not lost synchronization. If the framing pulse occurs at the beginning of any bit in the set other than the first bit, the synchronization of the sets will be incorrect thus not permitting the proper decoding of the received data. With the correct bit timing, correct set synchronization and data quantization, the decoding process should perform optimally.

3.3.1 Quantizer Design

To determine the number of levels chosen for quantization of the data, a graph of achieved Bit Error Ratio (BER) versus E_b/N_0 for a theoretical decoder operating with various numbers of quantization levels is closely examined. An immediate improvement of approximately 3 dB is seen in increasing to four levels of quantization [26] from two (two level quantization is commonly referred to as hard quantization). Another increase of approximately 1.2 dB is achieved in a further increase to eight levels but increasing the number of quantization levels above this does not achieve a significant improvement in gain. From this examination, eight levels of quantization will be considered acceptable. To accommodate the eight levels of quantization, three single data bit lines are required leading from the quantizer to the decoder data input.

3.3.2 The Decoder Front End

The main purpose of the front end processor is to capture a quantized channel output sample and to perform a table look-up to determine the closeness of this sample to the two possible channel inputs (logic 0 and logic 1). From Figure 20 below showing the eight level quantization structure, eight channel output representations are possible due to noise added to the data signal by noise. In determining the closeness of the channel output to the channel input two sets (of eight) likelihood values are utilized. These likelihood values are proportional to the maximum likelihood (ML) calculations as described by equation 2 below.

$$ML = -\log(\text{Prob.}(y_i|x_i)) \quad (2)$$

where $x_i = (0 \text{ or } 1)$.

As there are eight possible quantized channel output symbols, each sample must be compared to the eight possible channel output representations to effectively perform the table look-up of the correct likelihood values. With the symbol rate of the decoder being near the limit of the CMOS technology being used the comparison process utilizes the same high speed clock as the data capture process. But as the captured sample must be compared to the possible channel output representations (one comparison per high speed clock cycle), each sample must be held for eight of these clock periods. To avoid slowing the incoming symbol rate to accommodate this comparison, eight channel output samples are captured sequentially and the comparisons performed on all eight in parallel.

Figure 20: Quantized Output Channel Representations.

The front end processor is configured to synchronize quantized channel output sets and to compare them to various predetermined templates as given in Table 2 (somewhat later in this section). This comparison is to determine whether the received pair of symbols most (least) likely represents a one (zero) or a zero (one). The templates to which the samples are compared are representative of the eight values that may be generated by the quantizer. Due to the quantization process the input data symbols are a values in the range of 0 (000 binary) to 7 (111 binary), therefore the templates are of the same representation. Associated with each template are two three-bit likelihood values of the same integer range of values.

As the data is quantized to eight levels as represented by the three bits, it would be convenient use a method of state metric generation as stated by Bhargava et al [9]. Due to the quantization and the independence of individual symbols, the communication channel is assumed to be discrete memoryless. With a DMC configuration, the state metrics will be generated by a maximum likelihood log probability function. The literature states that the metrics are desired in integer form as this would allow them to be easily implemented in a digital hardware configuration. With this normalization of the logarithmic probability densities to integers, the decoding algorithm becomes slightly suboptimum. Jacobs and Heller [26] have shown that for satellite communication channels, the decoder performance is quite insensitive to a particular choice of a set of integer metrics. For a three bit quantized DMC using the metrics of Table 2 results in negligible performance degradation. Based on the description in [9], each three bit quantized data symbol may be compared to three bit templates as represented by the possible channel inputs of Table 2.

Template values:	0	1	2	3	4	5	6	7
1 Likelihood Value	7	6	5	4	3	2	1	0
0 Likelihood Value	0	1	2	3	4	5	6	7

Table 2: Set of Symbol Metrics for Eight Level Quantized Channel [9].

In a maximum likelihood decoder, the values representing these likelihood values are normally computed by calculating the probability of a received sequence representing a particular transmitted sequence. In the present case, the probability calculation process would be performed sixteen times (twice for each possible eight level quantized channel output because of the two possible channel inputs of 0 and 1). Eight calculated likelihoods represent the possibility of the received symbols representing a "1" having been transmitted and the second set of eight likelihoods representing a "0" having been transmitted. According to [25] most maximum likelihood probability calculations result in the values initially computed being of non-integer type requiring that the eight values be normalized to integer values. Such normalization produces a relatively large range of integers (from 0 to 127 in the worst case). Keeping in mind the results of [9], these non-integer values may be replaced with eight consecutive integers in the range of 0 to 7 with only minor degradation to the decoding process. Such an integer assignment to the metrics and templates would greatly simplify the required implementation of digital hardware.

In understanding the front end processing section the diagram of Figure 21 and the following description would be useful. Block A of Figure 21 captures eight consecutive quantized channel output samples (one per master clock cycle for a duration of eight clock cycles). On the next clock cycle following the latching of the eighth sample, these eight samples are transferred in parallel to Block B. This block is responsible for the data - template comparison. These templates are as found in the top row of Table 2. At the master clock frequency, these templates are, in a manner, read consecutively from a look-up table and compared to the eight parallel channel output samples. With each template, two corresponding likelihood values are also read from the look-up table. Simply stated, each column of Table 2 is read consecutively from the table. Upon a template matching the quantized channel output sample to which it is compared, the two associated likelihood values are saved. As eight comparison processes are conducted in parallel, sixteen likelihood values are saved by the

FIGURE 21: BLOCK DIAGRAM OF THE FRON END PROCESSOR.

end of the eight master clock cycles required for the process. These sixteen likelihood values are then transferred from Block B to the branch metric calculator.

A special timing scheme is required to allow each three bits of data to be compared to each of the eight templates. The quantized data will enter the decoder front end at a desired data rate of 120 Megasymbols per second (MSPS). This data rate is very near the upper bound of the speed available when using CMOS4 (1.2 micrometer channel width). The template retrieval rate is set at the incoming data rate as a result of this bound. To permit a single symbol (three bit representation) to be compared to each of the eight templates it must be held for exactly eight symbol periods. A convenient way of performing the template/data comparison is, as mentioned above, to sequentially latch the eight data symbols and synchronously move all eight to the template comparison stage. The eight symbols would be individually compared to each template until a match is found. Once an individual match is found, its two likelihood representations are selected and moved to the branch metric calculator.

3.3.3 The Branch Metric Calculator

The branch metric computation stage consists of retrieving the trellis branch (transition) label for a particular path leading from one node to the next and summing the individual likelihood values of the data pair representing that transition label. As an example, imagine the transition is labeled with 01. The four likelihood values generated by the front end processor for the set of two data symbols (01) are in a specific order. The first of the likelihood values represent the possibility that the first symbol is a logic 1, the second that the first symbol is a logic 0, etc. As the first digit of the label is a logic 0, the likelihood for that first symbol being a logic 0 becomes the first of two operands in an addition process. The second digit being a logic 1, selects the likelihood of the second symbol being a logic 1 as the sec-

ond operand of the addition. These two selected likelihood values are summed to create the branch metric for that transition. The branch metrics are then transferred to the appropriate add-compare-select (ACS) unit to be used to update the state metric of the node represented by this ACS. The process of selecting the appropriate likelihood values and summing the pairs of values to create branch metrics is performed for each transition (branch) within a stage of the trellis diagram.

3.3.4 The Add - Compare - Select Circuit

The ACS circuits to which branch metrics are routed are fixed by the formation of the trellis diagram from the convolutional encoder input/output sequence. Within a parallel oriented design there must be a single ACS cell for each node found in each stage of the trellis. For example, for a sixteen state trellis there are sixteen nodes per stage resulting in a requirement of sixteen ACS units within the decoder.

Within each ACS sub-circuit or cell, the transferred branch metrics are added to the selected state metric of the previous time stage. Each node has two branches leading to it from the previous stage therefore two adding processes must occur. The results of these two additions represent the two possible updated state metrics from which the best must be selected. The best state metric is transferred to the output decision unit and two single bits representing the estimated received data and a path memory control bit resulting from the state metric selection are passed to the path memory.

The bits estimating the received data are dependent upon the state from which they originate. Examination of the trellis diagram of Figure 22 and transition labels of Table 3 for the proposed convolutional encoding scheme reveals that for even numbered states (for a scheme with the first state labeled as 0) the branches leading to that state constitute a logic

FIGURE 22: TRELLIS DIAGRAM FOR A RATE 1/2, K=5 CONVOLUTIONAL CODE

ORIGINATION NODE:	DESTINATION NODE:	TRANSITION LABEL:	DATA:
0	0	00	0
	1	11	1
1	2	10	0
	3	01	1
2	4	10	0
	5	01	1
3	6	00	0
	7	11	1
4	8	01	0
	9	10	1
5	10	11	0
	11	00	1
6	12	11	0
	13	00	1
7	14	01	0
	15	10	1
8	0	11	0
	1	00	1
9	2	01	0
	3	10	1
10	4	01	0
	5	10	1
11	6	11	0
	7	00	1
12	8	10	0
	9	01	1
13	10	00	0
	11	11	1
14	12	00	0
	13	11	1
15	14	10	0
	15	01	1

TABLE 3: TRANSITION TABLE FOR THE TRELLIS OF FIGURE 20.

0 being the estimated data. The estimated data is a logic 1 when the branches are leading to one of the odd numbered states. As with encoding schemes the upper branch emanating from a node constitutes a logic 0 and the lower branch a logic 1. In addition to these signals, a control signal is sent to the path memory from each ACS to direct each path through the memory configuration.

3.3.5 Designing to Prevent Overflow

With the continuous operation of adding an increasing state metric to newly calculated branch metrics within the ACS, the adders will at some point overflow. Such an overflow would eventually cause erroneous estimations of the received sequence to result. The possibility of an overflow occurring must be detected before it actually occurs. There are several methods for detecting overflow and correcting for it by normalization of the state metrics. One method is to normalize after each addition-comparison-selection operation. Other methods include detecting a threshold before overflow occurs and then by dividing all state metrics in half (a deviation from the Viterbi Algorithm which will be discussed in detail later in this Thesis) or by subtracting some predetermined value from each metric. Overflow prevention will be studied later with the use of a functional simulator to determine the optimal method of its detection.

3.3.6 Path Memory

From each ACS unit three values are transferred to other units for further processing. The bits representing the estimated data sequence are routed to the data input of each path memory cell of depth one. The bits representing the control signals from each ACS are sent to all path memory cells in the memory block which are of the same state as the originating

ACS cell. A diagram of a path memory cell and its interconnection is shown in Figure 23. Each ACS cell directly transfers its bit representing its estimation of the received data to the first stage memory cell corresponding to its state. The path memory cell data interconnections are routed in an arrangement similar to that of the trellis layout thus allowing the data paths to be traced easily. The control signal from each ACS cell will be transferred to the respective path memory cell corresponding to the originating ACS. As an ACS cell processes new state metric calculations, the estimated data signals are transferred from the first memory cell in each path, using ACS supplied control signals, to the succeeding memory cells. Upon receiving the control signal, each path memory cell will select one of two previous ACS data estimation values to store as its current value. As the memory interconnections are configured as in the trellis structure, a control signal from an ACS cell will select and store only one of two hard wired signal values from the previous path memory stage. Such movement from stage to stage will produce, after N clock cycles for a path memory of length N , the optimum estimations (one per final stage path memory cell) of the data delayed by N master clock periods. Generally, according to [26] for a Viterbi decoder, the path memory length may be truncated to 4 times the constraint length of the encoder (resulting in $N=20$ for $K=5$) while retaining the optimality of the decoding structure.

3.3.7 The Output Decision Unit

The third of the three sets of signals transferred from the ACS cells would be the updated state metrics. These metrics are passed to the output decision unit along with the outputs of the final stage of the path memory unit. The state metrics of each ACS are paired with the path memory output bit from the memory cell of the same stage as the ACS. Two consecutive pairs of state metrics and estimated data sequences (pairs 0+1, 2+3,... etc.) are transferred to an output decision unit cell. The first stage of the output decision unit contains eight such cells as shown in Figure 24. From the two pairs, an output decision cell deter-

FIGURE 23: PROPOSED PATH MEMORY CELL BLOCK DIAGRAM

FIGURE 24: PROPOSED TREE LAYOUT OF THE OUTPUT DECISION UNIT.

mines the best of the two state metrics and selects this as its output with the associated data estimate bit. The comparison and selection here are similar to that which occurs within each ACS unit. The first stage will result in the selection of the eight of sixteen best metrics and data estimates requiring four decision cells in the next stage. Similarly the third stage requires two decision cells and the fourth and final stage uses just one decision cell to determine the best state metric and the actual decoder output data. This state metric would not necessarily be made available on the output of the last decision cell except if needed for test requirements. Effectively the output decision unit will use current state metric values to choose the optimum estimated data bit from delayed (by twenty clock periods) path memory data bits.

3.3.8 The Convolutional Encoder

To create a complete codec design, a convolutional encoder is required to complement the decoder presented. This encoder is of rate = $1/2$, constraint length $K = 5$ as seen in the diagram of Figure 25. The generator polynomials for this encoder are $G_1(D) = 1 + D_2 + D_3 + D_5$ and $G_2(D) = 1 + D_4 + D_5$. The resulting transition table has been previously shown in Table 2 and the trellis diagram for the encoder was given in Figure 22. As an aid to the design and understanding of the connectivity of the ACS units and the path memory unit a column in Table 3 illustrates the estimated data as a result of a branch leading to the node in question. In the table, the state of the encoder is determined by using the first four bits of the encoder shift register and reading the bits from the oldest to the more recent. The transition from one state to another is created by shifting a data bit into the encoder. The table is arranged such that the transitions created are shown by an increasing binary shift register representation. The output will be a two bit symbol representing the input data bit. The two bit symbol also represents the label on the branch connecting two states.

FIGURE 25: PROPOSED CONVOLUTIONAL ENCODER USED IN THE VITERBI CODEC

3.4 High Level Simulation

3.4.1 A Simple High Level Simulator

The previous pages have described the design of a high speed Convolutional Decoder and a Convolutional Encoder at a functional level. To prove that such a design is indeed operational and to determine its standing in terms of performance for the purposes of forward error correction, the generation of a high level simulator is required. Successfully running the simulator of the decoder/encoder assuming a perfect channel between the two will prove the operational functionality of the design. By adding noise generators, both impulse noise and additive white Gaussian noise generators, and running the simulator over durations sufficient to allow simulation of the transmission, reception and decoding of large quantities of data bits will provide insight to the decoding ability. A final comparison of the proposed decoder with a classical Viterbi Decoder from a previous, third party implementation under the same environment will allow its relative functionality to be assessed.

Initially the simulator was written containing only an encoder and a decoder. A flow chart of the simulator software for this stage is shown in Figure 26. The encoder followed the design described previously and for each data bit input, it generated two outputs corresponding to the data symbol. The input bits were generated by a random number generator creating either a "1" or a "0". Instead of transmitting the expected "1" or "0" data bit representations, the output designation of +1 for a "1" and a -1 for a "0" was used. These symbol bits were then transferred directly to the decoder data quantizer.

To keep the number of variables to a minimum for initial operational testing the eight level quantizer generated the output "111" upon receipt of a signal representing +1 (amplitude greater than 0) and "000" for a signal representing -1 (amplitude less than 0). As noise was not added at this step the received levels were at the extremes representing the best possible

FIGURE 26: FLOW CHART FOR THE NOISELESS SIMULATION ALGORITHM

received signal. The three quantized bits were then used to generate the probabilities of a "1" being the received data bit or of a "0" being the received bit. The received bits arrive at the receiver in pairs, therefore on each pass through the simulator a bit was input to the encoder, two symbol bits were generated by the encoder and received for selecting the four associated likelihood values.

Using the four generated likelihood values for each received data pair, a branch metric is created for each trellis transition. By choosing the best state metric for a node, the optimal estimation of the received data sequence is chosen. Similar to that noted in the design, the estimated value originating from a node will be a "0" if the node label is odd (in the simulator the label numbers begin at one instead of zero) and a one if the node label is even. The path metric control bit is generated by the simulated ACS by selecting a "0" as the output if the upper branch leading to a node is the best or a "1" if the lower branch is chosen. The ACS state metrics are then stored for use by the output decision unit and both the estimated data and control data are stored for use by the path memory. The path memory will shift the data appropriately through it by one step each time the simulation counter is incremented. The output decision unit will make a series of fifteen comparisons to determine which state metric and therefore which estimated data bit is correct.

As the path memory length is 20 (four times the encoder constraint length) the data bit out of the decoder is compared with a twenty program cycle delayed encoder input data bit. If these bits are the same, the decoder output is correct but if they are not the same the output is incorrect and the error output count is incremented by one. The bit error rate (BER) is computed by dividing the error output count by the total number of bits transmitted. The count of the output bit errors is the important quantity to be determined for the decoder under test. Usually the bit error count is determined for several operating environments.

3.4.2 Simulation Results

Running the simulator as it is described above determined that the branch metrics, ACS, path memory and output decision units functioned properly. It also served to confirm that the labels on the branch transitions between nodes were generated and placed correctly. Experimentation at this stage did yield some interesting results. One such result was such that, despite the delay in the output bits with respect to the state metrics being used to make a decision, errors would result on the output if the state metrics were not fed back to the ACS adders before the beginning of the subsequent bit period. The presence of errors in the absence of noise indicated a problem in the method of implementation. Delays of one through five bit periods were introduced in the state metric feedback path. The quantity of errors generated at the output for a fixed quantity of input bits increased with the increased delay in the state metric feedback path. From this experiment, it is quite important that the updated state metric be fed back to the adders of the ACS circuits before the next addition occurs.

A second result was the selection of the better method of detecting and preventing overflow of the state metrics. Detecting an eminent overflow by monitoring the level of each state metric for exceeding a threshold and modifying the state metrics by dividing each by two gave the better error performance. Such a state metric normalization does decrease the optimality of the metric under certain conditions as will later be shown. In the majority of cases, this method performs better than the others. The observed irregularities very likely occur a result of this normalization process being a deviation from the known method of state metric normalization within the Viterbi Algorithm. It must also be noted that the branch metric values are not affected by this normalization. The divide by two method of normalization was compared to two others. The first method involved subtracting a fixed margin from each metric to compensate for the overflow once the threshold had been exceeded. The second method normalized (by means of division by two) the state metrics

upon each cycle of the decoder. Error counts achieved with the latter two methods were not as acceptable as that of the initial normalization method. The subtraction method is, however, a documented near-optimal method for providing state metric overflow [12],[15],[19] and [21].

As the simulation indicates that the better error recovery does occur with the divide by two method, closer examination of its implementation is desired. In terms of hardware, the divide by two method is much easier to implement (as discussed in the appropriate section of Chapter 4). In dividing all state metrics by two, upon the detection of an overflow, the ordering of the nodes from the lowest to the largest state metric will not change. On occasion, adjacent state metrics that differ by a value of one will be normalized to the same value. Such an occurrence should rarely affect the selection of the best state metric in the output decision unit. As the overall results of the simulation appear to reflect that under low noise conditions, the integrity of the system is maintained despite the assignment of normalized weights of the surviving paths. This adopted normalization method is noted as a deviation from the original Viterbi Algorithm. Further simulation under varying noise conditions will determine its final acceptance.

Following the initial simulation experiments for debugging and functionality testing of the high level Viterbi Decoder design, additional code was added to simulate an ideal communications channel between the encoder and the decoder. The updated source code may be found in Appendix A.1 for reference. An ideal channel by definition, will pass the signal without distortion except for the introduction of additive white Gaussian noise. This produces a simulated communications system as shown in the block diagram of Figure 27. The Gaussian noise source is placed just before the quantizer in the receive path. The quantizer was made fully operational over the amplitudes of interest so that the effects of the noise may be determined. The noise was simulated by generating two random variables, x_1 and

FIGURE 27: SYSTEM DIAGRAM OF THE SIMULATED COMMUNICATION SYSTEM WITH NOISE.

x_2 , and using these variables as the sample values x in the following transcendental function of Equation 3 and Equation 4 given below [27].

$$p_1 = \sqrt{\log(1/(1-x_1))} \quad (3)$$

where $0 \leq x_1 < 1$

$$p_2 = 2 * \pi * x_2 \quad (4)$$

where $0 \leq x_2 < 1$

The noise amplitude value is determined by substituting the results for p_1 and p_2 into equation 5.

$$n_i = g_n * p_1 * \cos(p_2) \quad (5)$$

where n_i = the resulting noise amplitude

and g_n = the equivalent noise gain.

The noise amplitude may range from 0 representing high values of E_b/N_0 to relatively very large for low or negative E_b/N_0 values. This noise amplitude may take on a positive or negative value and is added directly to the amplitude of the data (either +1 or -1).

The data signal with the added noise is then input to the eight level quantizer. The range of amplitudes from +1 to -1 are divided into eight regions and labeled accordingly. Such a scheme would label amplitudes with values less than or equal to -0.75 as "000", amplitudes with values within increments of 0.25, increase the binary label by 1 per increment and ending with amplitudes greater than 0.75 labeled as "111". These quantized values are then transferred to the decoder front end.

Simulations were performed with AWGN inserted into the channel. The noise levels were varied creating varying E_b/N_0 ratios. The respective bit error ratios (BER values) were determined at various E_b/N_0 settings for simulations involving uncoded data and coded data which have been quantized to two levels and eight levels. The measured results are illustrated in the plot of Figure 28. From this graph of results illustrates that a coding gain of 3

Figure 28: BER curves for the Proposed Decoder using the simple channel simulator

dB or more is achieved over the uncoded data by using the soft decision decoding. Similar results were achieved for simulations involving the introduction of impulse noise into the system. With impulse noise, however, the BER value for a specific value of E_b/N_0 was higher.

3.4.3 Comparative Analysis

A further check on the validity of the described Convolutional Decoder design and the simulation of it was to run it in an environment in which a proven Viterbi Decoder had been tested. The previous version of a simple channel simulator is abandoned in favor of an elaborate simulator. A communications channel link and decoder simulator [27] designed by the Communications Research Center (CRC) in Ottawa was used for this purpose. Identical Convolutional codes ($r=1/2$, $K=5$) were used in the simulations involving the CRC channel simulator. The environment simulated was a DMC without the effects of fading. First the CRC channel simulator had been executed containing the CRC Viterbi Decoder for both two and eight level quantized data. The results obtained (BER values) by varying the E_b/N_0 of the system were recorded. The CRC decoder was then replaced by a version of the described Convolutional Decoder simulator modified to operate within the CRC Linksim simulator (Modified source code is presented in Appendix A.2) and the same tests executed. The results of this simulation were recorded. Both sets of data are plotted on a single graph as shown in Figure 29. A comparison of this plot illustrates that the proposed Viterbi Decoder architecture performs as well as the proven CRC simulated Viterbi Decoder (to within 0.5 dB). Such validation of the simulated architecture against the proven design of the CRC decoder justifies acceptance of the proposed high speed Viterbi-like Convolutional Decoder as designed. Chapter 4 will further describe the actual implementation and optimization of the hardware.

Figure 29: BER curves for the CRC and Proposed Decoder Simulators

There are several points in the plot c. Figure 29 corresponding to results of the simulation of the proposed Convolutional Decoder that deviate from a smooth pattern. These points, in particular, are worthy of mention. At E_b/N_0 of 3 dB, the BER for the proposed decoder using hard decisions and at E_b/N_0 of 5 dB, the BER for soft decisions exhibit some irregularity. Such irregularity is very likely caused by the choice of the metric values as described in section 3.3.2 and of the proposed deviation of the state metric normalization as discussed in section 3.4.2. Based on the good quality of the remaining portions of the plots and simplicity by which it may be implemented in hardware, the divide by two method of normalization will be the implemented method.

Chapter 4:

The Physical Implementation of a High-Speed, Single Chip CMOS Convolutional Decoder

4.1 Hardware Implementation

4.1.1 Introduction to Hardware

With the functional verification, by simulation, of the Viterbi-like Convolutional Decoder design presented in Chapter 3, the hardware implementation of it commenced. The design was separated into two sections with one portion to be implemented as the core on-chip processor and the second consisting of required off-chip support circuitry. The first section is described and will be implemented if conditions warrant its fabrication, whereas the off-chip portion is only described here to provide a complete discussion. Off-chip circuitry will consist of the clock recovery circuitry, framing pulse generator and the quantizer which require clock speeds in excess of that available on the decoder chip. As the desired decoder operation speed reaches the upper bound of the available technology, attempting to implement these functions on the chip would require a slowing of the decoding rate. Only the design of these off-chip circuits, not the implementation, will be discussed. The portion of the design to be implemented will be discussed in detail with respect to the hardware involved. Schematics of the hardware are found in Appendix B. These drawings should be used in conjunction with the following hardware description to obtain a detailed working knowledge of the design.

2

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS
STANDARD REFERENCE MATERIAL 1010a
(ANSI and ISO TEST CHART No. 2)

4.1.2 Required Off Chip Circuitry

The clock recovery scheme is based on an analysis of the received filtered demodulated data signal. Depending on the type of modulation used in the transmitter (i.e.: BPSK, QPSK, etc.), some means of detecting a zero crossing of the Nyquist pulses or instantaneous phase shifting of the signal permits the recovery of the clock signal and of the data bits. The frame pulse is then derived from the recovered clock and data. Such a frame pulse generator may require bit sequencing in the headers of transmitted packets.

The quantizer required generates a three bit parallel sequence representing the level (amplitude) of the sampled received data signal. The three bits result in eight levels of quantization requiring the quantizer to operate at eight times the speed of the incoming data rate. The thresholds of the amplitudes measured by the quantizer are set and a data sample is input. The eight quantizer thresholds are compared to the input signal. When this signal level is within the range of two consecutive thresholds a three bit binary value representing this range is latched and output as the quantized data. A block diagram of a typical quantizer is shown in Figure 30. Note the circuitry requiring the increased clock rate.

4.1.3 Decoder Front End

From the off-chip circuitry, signals are routed to the input of the decoder for clocking, frame sequencing, data and resetting the on-chip circuitry. The decoder front end is used to capture data, organize it and determine the values to be passed to the Branch Metric Calculator. To capture the data in the proper sequence, symbols must be latched at the appropriate instance in time. This synchronization is provided by a periodic framing pulse generated at intervals throughout the data stream. The master clock input signal is used to directly drive a three-bit binary counter. This three-bit counter repeatedly counts from 0 (binary 000) to

FIGURE 30: BLOCK DIAGRAM OF A TYPICAL THREE BIT QUANTIZER. NOTE THE CLOCK MUST BE A FACTOR OF EIGHT GREATER THAN THE ANALOG SIGNAL DATA RATE.

7 (binary 111). This counter output is transferred to a permanently enabled 3-to-8 decoder. The eight active low outputs of the decoder act as individual clocks for eight data latches. As the counter proceeds through its repetitive sequence, the 3-to-8 decoder output signals latch quantized data samples. With the aid of the framing pulse, the first relative data sample will be latched into the first latch. Each increment of the counter will latch a consecutive data sample into the next latch clocked by the 3-to-8 decoder. Upon the eighth latch being clocked, the proceeding clock cycle transfers these eight collected samples into a second series of three-bit data latches and restarts the process of capturing eight consecutive data symbols with the first series of latches. The sequential capture of eight consecutive data samples commences the parallel operation of the decoder. These eight symbols are routed in parallel to the proceeding sequential functions to be discussed in this chapter. The parallel structure exists in all sequential blocks of the decoder (refer to Figure 19 of Chapter 3).

The outputs of each latch of the second series are then compared, using eight three-bit comparators, to eight three-bit templates. The templates are generated by a second three-bit counter also clocked by a delayed version of input master clock. The templates range in value from 0 (binary 000) to 7 (binary 111) with only one template matching each latched data sample in a comparison cycle. In the comparison process, an identity match signal from each comparator will clock a probability latch when a data-template match is found. Associated with each template are two three-bit probability values. One three-bit value represents the probability that the latched data represents a logic 1 and the second value represents the probability that the data represents a logic 0. The two complementary groups of probabilities are also generated by two three-bit binary counters clocked by variations of the input master clock. By the end of a comparison cycle each three-bit data symbol has been used to select two probabilities creating a total of sixteen selected probabilities. These sixteen three-bit probabilities are transferred, in parallel, to the next stage in which the branch metrics are determined.

4.1.4 The Branch Metric Calculator

As the convolutional code used is rate $1/2$, each trellis transition is represented by two symbols requiring data samples to be grouped into pairs of two. Therefore, probabilities are required in pairs of four (two per data symbol). A 4-to-1 multiplexer is used to select the probabilities at a rate of four per cycle. As the rate of generation of probabilities by the comparison stage was slowed by a factor of eight, the rate of the 4-to-1 multiplexer is increased from this slower rate to operate at four times the probability generation rate. This permits sufficient time to be allotted to each branch metric generation. From the sets of four probabilities passed to the branch metric calculator, two probabilities are selected (one for the first data symbol and one for the second) based on the digits comprising the transition label. These two selected probabilities are summed to produce the branch metric for the represented trellis transition. As there are four different possible labels for the stage to stage transitions only four branch metrics need be calculated. These four metrics are computed simultaneously by four parallel streams. Each branch metric will be used by eight different ACS circuits in the state metric generation.

4.1.5 The Add-Compare-Select (ACS) Circuitry

The four different branch metrics are transferred to the appropriate ACS cells as determined by the connectivity of the trellis diagram. As previously mentioned, there are sixteen states in the trellis therefore sixteen ACS cells operate in parallel. Each ACS cell, representing a node or state of the trellis diagram, has two branch metrics transferred to it per clock cycle. These two branch metrics are individually added to the old state metric for the node. The result of the additions are two updated state metrics in each ACS sub-circuit from which the best metric must be selected. To select the best metric generated by an ACS, both updated state metrics are compared and the lower value selected. In the case of two equal state

metrics the value associated with the upper branch is selected. The two metrics are compared by a comparator from which the output will be the selection signal for a 2-to-1 multiplexer. Both state metrics are routed to this multiplexer and the selection signal from the comparator transfers one metric value to the multiplexer output. This output represents the best ACS state metric of this particular unit. This best metric is also fed back to the initial ACS adders as the old state metric value for the next cycle. It is also latched for use by the output decision unit. The selection signal generated by the comparator is latched for use by the path memory unit as a control signal. These control signals create the estimated data paths through the path memory.

4.1.6 Overflow Detection and State Metric Normalization

From the BER simulations of Chapter 3, the better performance of the decoder occurs when the divide by two state metric normalization method is utilized. This indicates that for the proposed configuration, a deviation from the original Viterbi Algorithm is incorporated into this design.

With the worst case of state metric updating, the maximum branch metric value is added to the previous state metric each cycle. The maximum increment of the state metric under worst case conditions would be fourteen. As the state metric consists of eight bits, allowing an unsigned maximum value of two hundred-fifty-five, the state metric would overflow its registers after nineteen master clock cycles. To prevent overflow each ACS unit is equipped with circuitry for detecting the possibility of a pending overflow and to normalize the state metrics upon such detection. The seventh of eight bits of each selected state metric is monitored. If the seventh bit of any ACS circuit becomes set (logic 1) then all state metrics will be divided by two and rounded down to the nearest integer. To monitor these bits, each of the seventh bits are ORed. If one of these bits become set to a logic 1, a flag is set. This will

act as a second selection bit to the multiplexers of each ACS unit. All selected state metrics in the succeeding cycle (and only that cycle) will be shifted towards the least significant bit by one position. This effectively performs the dividing and signal routing required. Once this shift has been performed, the seventh bit and the overflow flag which had been set to logic 1 will be reset to logic 0. Normal operation of the ACS proceeds until a succeeding overflow is pending.

4.1.7 The Path Memory

The estimated received data sequence as selected by each ACS cell, as noted previously in the functional description of Chapter 3, is a function of the state which the ACS unit represents. If a state has an even numbered label (0,2,...., etc. state labeling starting with 0) the data estimate selected by the ACS representing that state will be a logic 0. Otherwise, if the state label is odd the estimated data selected will be a logic 1. To maintain the minimal amount of routing, the first path memory cell of each of the sixteen paths will not perform a selection but will pass its estimated data bit to the two respective receiving cells in the second stage of the path memory. As a result, hardware is not required for the first stage of the path memory and the inputs of the selector units of the second stage may be hard-wired to a logic 1 or logic 0 signal according to the label of the ACS from which it should originate.

Each path memory cell contains a single latched output 2-to-1 multiplexer. The two signals from two separate path memory cells in the previous stage are the data inputs to the multiplexer. The output data will be selected from one of the two inputs by the control signal from the appropriate ACS cell. A clocking signal common to all path memory cells is used to latch the selected data on the output.

The required depth of each path is, as stated, four times the encoder state memory length resulting in twenty. In physical hardware the first cell in each path is eliminated thus resulting in a path length of nineteen. The output of the nineteenth cell of each path represents the estimated data bit for that path and is transferred to the output decision unit to be used in the optimum selection process. As sixteen nodes exist in the trellis, there must be a number of path memory cells sufficient to accommodate the sixteen possible surviving paths. This statement reflects that 16×19 cells are required to satisfy the minimum path memory requirements.

4.1.8 The Output Decision Unit

The output decision unit consists of a series of fifteen decision cells arranged in a tree structure. It is used to compare sixteen groups of values (state metrics) and to select one best output per group of sixteen inputs. Initially the sixteen eight-bit values are compared by eight decision cells resulting in eight best selections. These eight selected values are compared in pairs by four cells resulting in two remaining values from which one final comparison will produce the best of the sixteen values.

The values being compared by the output decision cells are the selected state metrics from the sixteen ACS cells. Associated with each state metric is an estimated received data bit delayed by twenty bit periods due to its transition through the path memory. As each comparison and selection of the current state metrics occur, an estimate of the received data is selected as well. The final selection stage will result in the best state metric deciding the output of the decoder from the narrowed choice of estimates. Due to the length of the path memory, the paths should merge to a common state thus enhancing one of the state metrics and reducing the risk of choosing the incorrect data sequence.

Each output decision cell is structured similar to the ACS cells. The two input eight-bit state metrics are compared by an eight-bit magnitude comparator. If the first metric is less than or equal to the second, the comparator output is a logic 0 otherwise it is a logic 1. The state metrics become the inputs of an eight bit 2-to-1 multiplexer. Also, in parallel, two corresponding data bit estimations become the inputs of a single 2-to-1 multiplexer. The output of the comparator performs the selection. The selected outputs are transferred to the next stage of output decision cells unless the output is from the last output decision stage. The last stage will output only the estimated data sequence representing the Convolutional Decoder output. The output data rate will be one half of the decoder input sample rate.

4.2 Hardware Simulation

4.2.1 SPLICE Representation of the Schematic

To simulate the hardware once the schematic had been entered using the Cadence Edge Schematic Entry package [28], a decision was made to use a customized version of the digital simulator SPLICE [29]. For fast processing speed the perceived CMOS implementation technology to be utilized for the design was the CMOS4 (1.2 micrometer channel width) process. At the time of entry, the only standard cell library available on the computer system for use was the 3 micrometer CMOS therefore initial schematics consisted of 3 micrometer based devices. Although Cadence Edge did contain its own hardware simulator, SPLICE provided more user flexibility permitting the modelling and entry of user specific components. In addition to the adaptability to user models, it also permitted the use of user dictated device characteristics. Standard cell characterizations were not originally available for the proposed 1.2 micrometer CMOS therefore initial simulations used the characteristics of the Texas Instruments 2 micron Standard Cells [30]. The simulations using the TI 2 micron standard cell characteristics were treated as a bench mark to provide proof of the operability

of the configured hardware and of the achievable operating speed of the decoder design. Shortly after the simulations had been completed the characteristics of the TI 1.2 micron standard cell reference library [31] had become available. A comparison was made between the 2 micron and 1.2 micron to reinforce the operability of the design. Eventually the customized SPLICE simulation file was modified to operate with the TI 1.2 micron standard cell references.

The non-customized version of SPLICE contains the basic logic gates such as AND, OR, NAND, NOR, etc. but not the larger blocks required such as counters, comparators, eight bit latches, etc. To provide the larger logic blocks as required without having to build these with the simulator, SPLICE was customized. To customize SPLICE the required blocks were functionally created by writing the models in a RATFOR program and compiling with the original SPLICE to create an executable file labelled CU-SPLICE (the user customized SPLICE) [32]. Through such customization, eleven different devices were added to the existing SPLICE models. A detailed list of these additions and the RATFOR source code (filename.MOD) are included in Appendix C.

The hardware simulator was written according to the SPLICE implementation instructions. The layout of the schematic was followed in the flow of writing the CUSPLICE simulator to ensure that required signals were generated with the proper timing. Basically a logic block is implemented within the CUSPLICE file by first giving it a local name such as ORG1 for an OR gate. Following this on the same line but separated by a delimiter are the output signal names and the input signal names. The final label will be the actual logic block reference for CUSPLICE. For a two input OR gate, as discussed, this could be OR2G. Complete details of the format of the SPLICE structure are given in the SPLICE User's Guide. Each required logic block is implemented in a similar fashion. These code calls to logic blocks represent the various discrete elements in the schematic. Input signals to the

simulated circuit elements are entered in the SPLICE description code following the logic block implementation section discussed above. For the Viterbi Decoder simulator the inputs required were the clocking signals, the reset signal and a single data stream. The inputs and outputs of any modelled device may be plotted, as a standard SPLICE output, for verification purposes.

The final entries in the CUSPLICE deck are the references to the required logic blocks. For the two input OR gate mentioned above such a reference would be ORI2. In the case of referencing one of the blocks introduced by customized SPLICE RATFOR code, a reference to the MODEL number is required in the SPLICE deck. This MODEL number is assigned to the functional block in the RATFOR *.MOD file. These reference statements also contain the propagation time constants such as signal rise and fall times as determined from the characterization of the standard cells. The SPLICE deck source as used for the simulations is presented in Appendix D.

4.2.2 Simulation Results

Initially the simulation was furnished with typical TI 2 micron values of device rise and fall times for use in propagation delay effects. Under such a simulation, the encoder was given long sequences of logic 0's, sequences of logic 1's, and alternating patterns of each (i.e. 0101...). The results of such simulations gave outputs matching the inputs as may be seen in the SPLICE plots of Appendix E. As all devices will have propagation delays inherent dependent on the processing technology used, there is a range of delay values associated with each technology. Each device, depending on processing flaws, position on the die, micro-climatic effects, thermal effects, etc. will have delays associated with it that may range from the minimum achievable in a certain technology to the maximum. Two similar devices, having the same functionality (i.e.: two AND gates) probably would have different

propagation delays associated with them. To attempt to test the simulator under varying delay conditions the delay values (rise and fall times) were changed to the minimum values stated in the TI 2 micron library. The simulation conditions as used with the typical delay value tests were executed and results plotted. Similarly, the propagation delay values were changed to the maximum and the simulation conditions re-executed. Plotted results for the maximum delays were recorded and also appear with those of the typical and minimum delays in Appendix E.

The total decoder bit delay is of interest at this point. Another interesting result of the simulations would be the operating speed of the decoder under the three different delay conditions stated above. Table 4 listed below illustrates the average measured results for the three 2 micron test conditions.

MEASUREMENT	MINIMUM	NOMINAL	MAXIMUM
OPERATING FREQUENCY	50 MHz	50 MHz	25 MHz
DECODING DELAY	700 ns	900 ns	1400 ns

TABLE 4: 2 MICRON CU-SPLICE SIMULATION HIGHLIGHTS

4.3 CMOS Layout

The generation of a CMOS transistor level layout may be accomplished automatically through the use of Cadence Edge. The layout floor pattern required to give near optimal area utility and minimized routing is shown in Figure 29. Implementation estimates using the TI 2 micron CMOS area figures results in the requirement of approximately 11,700 gates. Each gate is approximately 2500 square microns thus resulting in an area of approx-

FIGURE 31: LAYOUT FLOOR PLAN FOR THE SINGLE CHIP HIGH-SPEED VITERBI DECODER

imately 29.25 square millimeters. If this area is doubled to allow area for routing and the addition of I/O pads the total then becomes approximately 58.5 square millimeters. Such an implementation requires a silicon die of approximately 7.6 millimeters on a side will not likely fit on a standard Canadian Microelectronics Center (CMC) A-frame die [33]. An oversized die would be required for fabrication using the TI 2 micron process.

4.4 Comparison of a 2 Micron to a 1.2 Micron Implementation

By obtaining the TI 1.2 micron standard cell library the results of the simulations and layouts may be scaled to the 1.2 micron level from 2 micron. Generally the propagation delay values for the 1.2 micron cells are sixty percent of that for the 2 micron cells. This would effectively reduce the propagation delays and increase the operating speed of the decoder. Further simulations with the 1.2 micron propagation delay characteristics has generated results shown in Table 5 below.

MEASUREMENT	MINIMUM	NOMINAL	MAXIMUM
OPERATING FREQUENCY	NOT AVAILABLE	100 MHz	40 MHz
DECODING DELAY	NOT AVAILABLE	700 ns	800 ns

TABLE 5: 1.2 MICRON CU-SPLICE SIMULATION HIGHLIGHTS

With the availability of the TI 1.2 micron standard cell propagation delay characteristics, a second series of simulations were performed using the nominal gate delays for this technology. Such simulations increase the confidence level of the proposed decoder being operational at the symbol rate of 100 MSPS (clock speed 100 MHz). The decoder throughput delay with the simulation is approximately 0.7 microseconds. A complete set of timing di-

agrams for such a simulation are produced for reference in Appendix E.

The relative area decrease in shrinking to 1.2 micron from 2 micron technology is approximately forty percent. This would produce an overall area requirement of approximately 36.8 square millimeters. Such a die may be realized by a square with sides of length of approximately 6 millimeters. This implementation will fit on the CMC standard A-frame silicon die.

4.5 Hardware Design Conclusions

The successful development of circuit level and high level system simulators has greatly increased confidence in the functionality and operability of the final design. Based on such confidence, the fabrication of the proposed Viterbi Decoder may be undertaken providing the appropriate standard cell library is used to generate a layout. The simulation benchmarks have used the Texas Instruments standard cell design guidelines but fabrication through the Canadian Microelectronics Corporation would require a layout using the CMC CMOS4S standard cells. The characteristics of these cells must be studied before functionality and operability of the proposed Viterbi Decoder may be confirmed using the CMC 1.2 micron standard cells. Ideally a firm running a CMOS process utilizing the TI 1.2 micron standard cells or an equivalent would be the desired target for fabricating the Viterbi Decoder. Based on the research performed in generating this thesis, the proposed decoder design is optimized for fabrication utilizing the noted TI 1.2 micrometer CMOS technology.

In-circuit testability of specific sections of the design would be essential before an implementation could proceed. Although not dealt with in this Thesis, the provision for on-chip circuitry for testing the high speed critical path sections of the design warrants much effort. Specifically the high speed front end and the branch metric generator should contain the

means for post-fabrication on-chip testing. Only these sections of the design operate at the master clock frequency and are vital to the successful operation of the decoder.

One of the design objectives has been to achieve a decoder that consumes low power. It has been assumed that such an objective will be met through the use of a CMOS technology. Estimates have not been performed to determine the power consumption of the proposed circuit. Such a detail should be performed before a fabrication is initiated.

Chapter 5:

Conclusions and Future Work

5.1 Thesis Conclusions

The scope of data communications has become quite broad in recent years. Transferring data electronically may range from routing files amongst personal computers in a single room to transmitting digitized photographs from deep space probes to earth. A niche within the digital communications sector which has been expanding rapidly in recent periods is mobile data communications. This expansion has been introduced with the public acceptance of cellular data networks and the more recent mobile satellite data systems. Inherent in this development is the requirement for increasing the data transmission rate while maintaining spectrum efficiency. Devices capable of achieving the operating requirements and providing acceptable level of error free data are essential to advanced development.

In this thesis such a high speed device has been proposed. Within a high speed data system, particularly a system which transmits its data through noise contaminated environments, a device for implementing a Forward Error Correction scheme is crucial to providing relatively error free data communications. A known optimum algorithm for use in Forward Error Correction is the Viterbi Algorithm. The proposed device consists of an optimized Convolutional Decoder based on the Viterbi Algorithm but with slight discrepancies and a complementary rate = $1/2$, constraint length = 5 Convolutional Encoder forming a high speed, single chip CMOS Convolutional Codec. Development methodologies utilize state of the art concepts from various existing classical Viterbi based devices. These structures have been modified and configured in parallel/sequential processes to provide the greatest possible throughput. Current bottleneck problems are avoided with eight such parallel se-

quential streams. The overall throughput of the device is dependant upon the technology with which it is fabricated. The codec may easily be modified to operate with rate = $1/2$, constraint length $K = 7$ Convolutional Code with the availability of either larger die sizes or an even greater reduced scale processing technology.

System and hardware verification give high confidence that a chip fabricated will be functional and will have performance as predicted. In optimizing the design and verifying it through simulation several technologies were compared and utilized in the experiments. Originally, the TI 2 micron technology characteristics were incorporated to gain knowledge of the relative operational frequencies achievable and the required corresponding implementation areas. Latter availability of 1.2 micron CMOS characteristics, again from the TI library, permitted a transition of the simulations to a more advanced technology. Results utilizing these characteristics revealed that using the TI 1.2 micron CMOS standard cell library, the proposed Viterbi-like Convolutional Codec may be placed on a standard CMC A-frame silicon die. Nominal operating rates in excess of 100 MHz (100 MSPS) are easily achievable as indicated by simulation.

Independent of processing technology is the error detection and correction ability of the device. A high level functional simulator of the proposed Decoder was created and interfaced to a widely used mobile data communications channel simulator. Bit error ratio measurements were performed for various values of average bit energy to noise ratios. This characterization of the proposed decoder were compared to known, proved results of an existing, low speed Viterbi Decoder. The comparison indicated that the high speed decoder performed as well as the low speed decoder under similar conditions.

The proposed Codec, based on requirements for use with mobile data communications systems, is capable of being adapted to many systems. The codec is independent of the mod-

ulation type utilized for the data transmission. The use of static logic permits operation at a range of frequencies extending from very low data rate to the upper bound. Such a configuration may be used in an adaptive signal processing environment given the appropriate selection of the master clock and framing pulse are available. The internal logic uses return to zero binary logic with the internal nodes cleared to the logic zero value upon system reset. Internal symbol and data template signals consist of three parallel bits with the state metrics consisting of eight parallel bits. State metric overflow prevention through an aperiodic normalization process ensures optimal selection of the most likely data estimation. Based on the presented design structure and simulation results, the proposed Codec would be an efficient signal processing device for conditioning received data corrupted by noise.

Past Viterbi Algorithm based codecs, when used in low speed data communication systems, have achieved optimal error correcting limits. Such implementations were often cumbersome in size. The design of an optimal Viterbi-like, Convolutional based codec using modified structures of these previous low speed systems, enhanced by implementation using reduced scale geometry CMOS technology, permits optimal error correction to be performed in high speed data communication systems. The improved speed advantage allows successful error correction at rates in excess of 100 MHz in comparison to the slower speeds in the 10 MHz range.

5.2 Future Work

With the development of the proposed high speed single chip CMOS Convolutional Codec, the concept of increasing the capabilities and processing power of the device as an adaptive digital signal processing tool became evident. The adaptive properties of the Viterbi Algorithm permit creative methods of signal processing and conditioning to be incorporated in many single chip processing applications. Effecting any of the following would advance

the acceptability of the proposed design. Through the use of a suitable CMOS processing technology, the clock and bit recovery, framing pulse generator and quantizer may be placed on a single silicon die with the Codec. Using dynamic digital logic would permit a further processing technology independent increase in the overall Codec operating frequency. With the addition of the appropriate programmable logic units, such as templates and branch metrics would permit in-circuit adaptability of the decoding algorithm.

Before researching the applicability of the high speed Viterbi Algorithm concept to adaptive digital signal processing, a fabrication of the implementation contained in this Thesis would confirm simulation results obtained in this development. In preparing for fabrication, the incorporation of in-circuit testability should be given priority. Such inclusion would be beneficial in device testing.

References

- [1] J.W. Harris and T.J. Ledwidge, "Introduction to Noise Analysis", Pion Limited, 207 Brondesbury Park, London, England NW2 5JN.
- [2] R.W. Lucky, J. Salz and E.J. Weldon, Bell Telephone Laboratories, Inc., "Principles of Data Communication", McGraw Hill Book Company, New York 1968.
- [3] G.C. Clarke, Jr. and J.B. Cain, "Forward Error Correction Coding for Digital Communications", New York Plenum Press, 1981.
- [4] Vijay K. Bhargava, "Forward Error Correction Schemes for Digital Communications", IEEE Communications Magazine, January 1983.
- [5] C.E. Shannon, "A Mathematical Theory of Communication", Bell System Technical Journal, Vol. 38, pp. 651-656.
- [6] A.J. Viterbi, "Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm", IEEE Trans. Info. Theory, Vol. IT-131, pp. 260-269, April 1967.
- [7] G.D. Forney, Jr., "The Viterbi Algorithm", Proceedings of the IEEE, Vol. 61, No. 3, March 1973.
- [8] B. Sklar, "Digital Communications: Fundamentals and Applications", Prentice Hall, 1988.
- [9] V.K. Bhargava, D. Haccoun, R. Matyas, and P. Nuspl, "Digital Communications by Satellite: Modulation, Access and Coding", John Wiley and sons, 1981.
- [10] G.C. Clark, Jr. and J.B. Cain, "Error Control Coding for Digital Communications", Harris Corporation, Melbourne, Florida, Plenum Press, New, York, 1981.
- [11] J.K. Omura, "On the Viterbi Algorithm", IEEE Transactions on Information Theory, Vol. IT15, January 1969, pp.177-179.
- [12] N.J.P. Frenette, et al., "Implementation of a Viterbi Processor for a Digital Communications System with a Time-Dispersive Channel", IEEE Journal on Selected Areas in Communications, Vol. SAC-4, No. 1, Jan. 1986.
- [13] P.G. Gulak and E. Shwedyk, "VLSI Structures for Viterbi Receivers: Part I - General Theory and Applications", IEEE Journal on Selected Areas in Communications, Vol. SAC-4, No. 1, Jan. 1986.
- [14] T. Fujino, et al., "A 120 Mbit/s Coded 8PSK Modem with Soft-Decision Viterbi Decoder", IEEE International Conference on Communications, 1986.

- [15] R.D. McCallister and J J. Crawford, "A Low-Power, High-Throughput Maximum-Likelihood Convolutional Decoder Chip for NASA's 30/20 GHz Program", IEEE International Conference on Communications, 1981.
- [16] J.B. Cain and R.A. Kriete, "A VLSI $R=1/2$, $K=7$ Viterbi Decoder", IEEE NAE COM. 1984.
- [17] P.G. Gulak and T. Kailath, "Locally Connected VLSI Architectures for the Viterbi Algorithm", IEEE Journal on Selected Areas in Communications, Vol. 6, No. 3, April 1988.
- [18] A.P. Clarke and M. Clayden, "Pseudobinary Viterbi Detector", IEE Proceedings, Vol. 131, Part F, No.2, April 1984.
- [19] J. Stahl, M. Meyr, and M. Oerder, "Implementation of a High Speed Viterbi Decoder", Signal Processing III: Theories and Applications, Elsevier Science Publishers B.V. (North-Holland), 1986.
- [20] G. Ungerboeck, "Channel Coding with Multilevel/Phase Signals", IEEE Transactions on Information Theory, Vol. IT28, No. 1, January 1982, pp.56-67.
- [21] R.M. Orndorff, et al., "Viterbi Decoder VLSI Integrated Circuit for Bit Error Correction", IEEE International Conference on Communications, 1983.
- [22] T. Ishitani, et al., "A Scarce-State-Transition Viterbi Decoder VLSI for Bit Error Correction", IEEE Journal of Solid-State Circuits, Vol. SC-22, No. 4, August 1987.
- [23] T. Ishitani, et al., "A Proposal of Universal-Coding-Rate Viterbi Decoder - A Novel Encoding and Decoding Method for High-Coding-Rate FEC", IEEE International Conference on Communications, 1987.
- [24] S.J. Simmons, "A Reduced-Computation Trellis Decoder with Inherent Parallelism", IEEE Conference on Global Communications, 1987.
- [25] D.J. Costello and Shu Lin, "Error Control Coding; Fundamentals and Applications", Prentice-Hall Inc., Englewood Cliffs, New Jersey, 07632, 1988.
- [26] J.A. Heller and I.M. Jacobs, "Viterbi Decoding for Satellite and Space Communication", IEEE Transactions on Communications Technology, Vol. COM-19, No. 5, October 1971.
- [27] "Linksim Simulation Program User's Guide", Miller Communications Systems Ltd., 300 Legget Drive, Kanata, Ontario, K2Y 1Y5
- [28] "SDA Physical Design and Reference Manual", Version 2.0, SDA Systems Inc. February 29, 1988.

[29] R.A. Saleh, J.E. Kleckner and A.R. Newton, "SPLICE Version 1.7 User's Guide", Department of Electrical Engineering and Computer Science, University of California, Berkeley, Ca. 94720.

[30] "Texas Instruments 2 um Standard Cell Library", Texas Instruments, Houston, Texas, USA.

[31] "Texas Instruments TSG500 Series Standard Cell Manual", Texas Instruments, Houston, Texas, USA.

[32] "Carleton University SPLICE V1.7.87.10/MAC User Guide", Department of Electronics. Carleton University, Ottawa, Canada.

[33] "Guide to the Implementation Services of the Canadian Microelectronics Corporation", Canadian Microelectronics Corporation, Carruthers Hall, Queen's University, Kingston, Ontario, Canada K7L 3N6

Appendix A.1

The source listing for a simple simulation appears in this Appendix. Data is randomly generated which is used as a source for a rate = $1/2$, constraint length = 5 convolutional encoder. The symbols then generated by the encoder are transmitted through the channel to the receiver. Enroute to the receiver additive white Gaussian noise or Impulse noise may be added to the data signal. The received data signal is quantized by the receiver and the quantized symbols routed to the decoder. The decoder output and encoder input are compared to determine the bit error ratio for each noise level for which the simulator is executed. This is a stand alone, dedicated simulator for the Viterbi Decoder design as presented in Chapter 4 of this Thesis.

Appendix A.2

The source listing for a modified version of the above simulator is listed in this Appendix. The above simulator has been stripped of the data generator, encoder and noise simulators to allow mating with the CRC Linksim Channel simulator. The appropriate additions have been made to permit this integration.

Appendix A.1

program channel

```
c This program is a simple model of a convolutional encoder, an ideal
c channel (adds AWGN only) and a Viterbi-like decoder.
c Variable Definition:
c All undefined variables will be integers.
  implicit integer (a - z)
c The randomly generated data:
  integer data
c The state register bits:
  integer br(5)
c The convolutionally encoded symbols:
  integer sy(2)
c
  integer v(31)
c
  integer rb(32)
c
  integer cu(16)
c
  integer dm(16,32)
c
  integer lb(2)
c
  integer ub(2)
c
  integer us(16)
c
  integer is(16)
c State metrics [1..16 >> ACS], [17..31 >> ODU].
  integer nn(31)
c Branch metric temporary values:
  integer lt(2,2)
c
  real p1,p2
c Noise generation random value:
  real sm
c
  real s=ed
c Noise amplitude relative to 1V.
  real pb
c
  real pcn
c
  real smn
```

```

c      real gn
c
c      common /metric/ lt(2,2),bar
c
c      open (21,file='vits.dat')
c      Perform BER simulations for a range of Eb/No values.
c      do 200 gn=0.1,2.0,0.1
c      Initialize variables'
c      seed = 0.5
c      ov = 0
c      bov = 0
c      pcn = 0.0
c      ce = 0
c      xr = 0
c      bar = 1
c      ec = 0
c      er = 0
c
c      do 10 p=1,31
c      nn(p) = 0
10      v(p) = 0
c      i = 0
c      do 20 t=1,16
c      cu(t) = 0
c      do 15 g=1,32
15      dmuh(t,g) = 0
20      continue
c      do 25 h=1,32
25      rb(h) = 0
c      do 30 h=1,5
30      bt(h) = 0
c      Shift all encoder bits one farther into the shift register.
100      do 35 h=5,2,-1
35      bt(h) = bt(h-1)
c      Shift the saved random data bits one position farther through memory.
c      do 40 z=32,2,-1
40      rb(z) = rb(z-1)
c      Generate a count of generated data bits.
c      i = i+1
c      Generate the first bit of the shift register.
c      if (i.le.5) then
c      bt(1) = 0
c      else
c      bt(1) = data(seed)
c      endif

```

```

c      Save the data in the first shift register position for later comparison to
c      decoded data. This will generate the BER count.
      rb(1) = bt(1)
c      Generate the encoded data.
      sy(1) = mod((bt(1) + bt(2) + bt(3) + bt(5)),2)
      sy(2) = mod((bt(1) + bt(4) + bt(5)),2)
      if (sy(1) .eq. 0) sy(1) = -1
      if (sy(2) .eq. 0) sy(2) = -1
c      Generate the noise to be added to the signal.
      do 50 l=1,2
      sm = abs(float(irand(0))/2147483648)
      p1 = sqrt(alog(1/(1-sm)))
      sm = abs(float(irand(0))/2147483648)
      p2 = 2*3.141592654*sm
      pb = gn*p1*cos(p2)
      pcn = pcn + pb*pb
c
      smn = sy(l) + pb
c      Quantize the additive noise/signal waveform.
      if (abs(smn).le.0.25) qn=4
      if((abs(smn).le.0.5).and.(abs(smn).gt.0.25)) qn=5
      if((abs(smn).le.0.75).and.(abs(smn).gt.0.5)) qn=6
      if(abs(smn).gt.0.75) qn=7
c      A check for high noise causing destructive interference.
      if ((sy(1).eq.1).and.(smn.le.0.0)) xr=xr+1
c      A check for high noise causing destructive interference.
      if ((sy(1).eq.-1).and.(smn.gt.0.0)) xr=xr+1
c      Generate the partial branch metrics.
      if (smn.le.0.0) then
      lb(1) = 7-qn
      ub(1) = qn
      else
      lb(1) = qn
      ub(1) = 7-qn
      endif
c      Repeat for the second symbol.
50      continue
c      Transfer partial branch metrics to temporary variable.
      lt(2,1) = lb(2)
      lt(2,2) = ub(2)
      lt(1,1) = lb(1)
      lt(1,2) = ub(1)
c      Calculate the branch and state metrics for each node.
      call branch(1,0,0,1,1,us(1),ls(1),nn(1))
      call branch(2,1,0,0,1,us(2),ls(2),nn(2))
      call branch(3,1,0,0,1,us(3),ls(3),nn(3))

```

```

call branch(4,0,0,1,1,us(4),ls(4),nn(4))
call branch(5,0,1,1,0,us(5),ls(5),nn(5))
call branch(6,1,1,0,0,us(6),ls(6),nn(6))
call branch(7,1,1,0,0,us(7),ls(7),nn(7))
call branch(8,0,1,1,0,us(8),ls(8),nn(8))
call branch(9,1,1,0,0,us(9),ls(9),nn(9))
call branch(10,0,1,1,0,us(10),ls(10),nn(10))
call branch(11,0,1,1,0,us(11),ls(11),nn(11))
call branch(12,1,1,0,0,us(12),ls(12),nn(12))
call branch(13,1,0,0,1,us(13),ls(13),nn(13))
call branch(14,0,0,1,1,us(14),ls(14),nn(14))
call branch(15,0,0,1,1,us(15),ls(15),nn(15))
call branch(16,1,0,0,1,us(16),ls(16),nn(16))
bar = 1

```

c Perform the Add-Compare-Select functions.

```

do 60 as=1,16
  d1 = int(real(as)/2 + 0.5)
  d2 = d1+8
  if (mod(as,2).eq.0) then
    a1 = ls(d1)
    a2 = ls(d2)
    v(as) = 1
  else
    a1 = us(d1)
    a2 = us(d2)
    v(as) = 0
  endif
  if (a2.ge.a1) then
    nn(as) = a1
    cu(as) = 0
  else
    nn(as) = a2
    cu(as) = 1
  endif

```

c State metric normalization flag set.

```

  if (nn(as).ge.64) then
    bov = bov + 1
    bar = 2
  endif

```

60 continue

c Transfer path memory data one step further into the array.

c Place the new ACS generated data into the first stage of each path.

```

do 62 mm=32,2,-1
  do 55 k=1,16
    if (cu(k).eq.1) then
      dmu(k,mm) = dmu((int(real(k)/2+0.5)+8),mm-1)
    
```

```

else
  dmu(k,mm) = dmu((int(real(k)/2+0.5)),mm-1)
endif
55  continue
62  continue
c    Perform the output decision unit tasks.
    do 65 k=1,16
65   dmu(k,1) = v(k)
    do 70 al=1,15
      f2 = 2*al
      f1 = f2-1
      if (al.le.8) then
        b1 = dmu(f1,32)
        b2 = dmu(f2,32)
      else
        b1 = v(f1)
        b2 = v(f2)
      endif
      a1 = nn(f1)
      a2 = nn(f2)
      if (a2.lt.a1) then
        nn(al+16) = a2
        v(al+16) = b2
      else
        nn(al+16) = a1
        v(al+16) = b1
      endif
70   continue
c    Compare the ODU data estimate of the decoder output to the delayed
c    encoder input. If different increment the BER count.
      if (v(31).ne.rb(32)) er=er+1
      br = er/i * 100
c    Continue simulation until maximum bit count exceeded.
      if (i.lt. 1e4) goto 100
c    Write results of simulation (once finished) to data file.
      write (21,*) gn,i,'Ein=',xr,'Eout=',er,' Eb/No =',10*log10(i/pcn)
200   continue
c    Close the data file.
      close (21)
      end

      subroutine branch(numd,da,db,dc,dd,upp,lww,smm)

c    This subroutine uses the partial branch metrics (probabilities for
c    each label digit) to calculate the branch metrics and path metrics.
      implicit integer (a - z)

```

```

common /metric/ lt(2,2),bar
c   Change 0 to 1 and 1 to 2 to permit correct array access.
    ua = da + 1
    ub = db + 1
    lc = dc + 1
    ld = dd + 1
c   Generate the state metric using the upper branch metric.
    upp = lt(1,ua) + lt(2,ub) + int(real(smm)/bar)
c   Generate the state metric using the lower branch metric.
    lww = lt(1,lc) + lt(2,ld) + int(real(smm)/bar)

    return
    end

    real function synb(seed)
c   This function is used to randomly generate values between 0 and 1.
    integer seed,m,const1
    real synb, const2
    parameter (const1 = 2147483647, const2 = 0.4656613e-9)
    save
    data m /0/

    if (m.eq.0) m =seed
    m = m* 65539
    if (m .lt. 0) m = (m+1) + const1
    synb = m*const2
    seed = synb
    end

    integer function data(seed)
c
c   the data stream.
    implicit integer (a - z)
    real seed,test,synb

    test = synb(seed)
    data = 1
    if (test.lt. 0.5) data = 0

    end

```


Appendix A.2

```
c-----
c
c SUBROUTINE FASTVIT is a high speed conventional simulator
c for the Viterbi Decoding Algorithm.
c
c INPUTS:
c
c BITBLK - Quantity of estimated data symbols.
c DECISI - Real representation of the symbols in
c analog form.
c DATINT - Interleave Factor.
c WINDOW - Bit delay through the decoder (31 in this case).
c INCODE - Type of input code (rate 1/2 here).
c BITDEL - Channel bit delay.
c SYSINIT - Initialization flag.
c
c OUTPUT:
c
c RXBITS - The decoded data bits.
c
c-----
c
c Author : D. Peter Noel
c
c Created: June 7,1990 (Basic version on TRS-80 Sept. 1989.)
c
c Revised:
c
c (C) Copyright 1990, D. Peter Noel
c
c-----
c          subroutine fastvit(bitblk, decisi, datint, window, incode,
c          & bitdel, sysinit, rxbits)
c
c Variable declarations.
c
c          implicit integer (a - z)
c
c The preliminary estimated data from the ACS and output decision
c units.
c
c          integer v(31)
c
c The path memory selection control bits.
c
```

```

integer cu(16)
c
c The path memory cell contents.
c
integer dmu(16,32)
c
c Symbol probabilities: probability of symbol being 1 and 0.
c
integer lt(2,2)
c
c The probability of symbols being zero.
c
integer lb(2)
c
c The probability of symbols being one.
c
integer ub(2)
c
c The upper branch metrics.
c
integer us(16)
c
c The lower branch metrics.
c
integer ls(16)
c
c The resultant node state metrics.
c
integer nn(31)
c
c The maximum size of a received data block.
c
integer maxblk
parameter (maxblk = 1024)
c
c The quantity of symbols transferred per block.
c
integer bitblk
c
c The interleaving factor.
c
integer datint
c
c The bit delay through the decoder.
c
integer window

```

```

c
c The type of data encoding utilized.
c
      character*20 incode
c
c The bit delay through the channel simulator.
c
      integer bitdel
c
c The input samples.
c
      real decisi(*)
c
c The decoded bits.
c
      logical rxbits(1024)
c
c Other local variables used.
c
      real smn
c
c The system initialization flag.
c
      logical sysinit

      common /fstvit/ lt(2,2)

      cave bar,nn,v,cu,dmu,oldbit,symvlp
      data symvlp /0/
c
c Perform sanity checks for constraint length
c and interleaving.
c
      if (incode(7:7).ne.'5') then
        write (*,*) 'FASTVIT - Constraint length must be 5.'
        stop
      endif
      if (datint .ne. 1) then
        write (*,*) 'FASTVIT - Interleave factor must be 1.'
        stop
      endif
c
c Initial the variable on the first pass only.
c
      if (sysinit) then
        bar = 1

```

```

do 10 p=1,31
nn(p) = 0
10    v(p) = 0
do 20 t=1,16
cu(t) = 0
do 15 g=1,32
15    dmu(t,g) = 0
20    continue
oldbit = bitdel
sysinit = .false.
endif

c
c Calculate the updated bit delay with addition of the encoder.
c
    bitdel = ((bitdel+1)/2) + (window * datint)
c
c Handle the odd overflow problem.
c
    if (mod(oldbit,2).eq.1) then
        ist = 0
        decisi(0) = symvlp
    else
        ist = 1
    endif
    rj = 0

c
c Perform the actual decoding.
c
    do 100 i=ist,bitblk-1,2
        rj = rj + 1
        p = 0

c
c Get two symbols and use them in the decoding process.
c
        do 50 l=i,i+1
            p = p + 1
            smn = decisi(l)

c
c Quantize the samples.
c
            if (abs(smn).le.0.15) qn=4
            if((abs(smn).le.0.43).and.(abs(smn).gt.0.15)) qn=5
            if((abs(smn).le.0.72).and.(abs(smn).gt.0.43)) qn=6
            if(abs(smn).gt.0.72) qn=7

c
c Determine the one and zero probabilities.

```

```

c      if (smn.le.0.0) then
        lb(p) = 7-qn
        ub(p) = qn
      else
        lb(p) = qn
        ub(p) = 7-qn
      endif
50    continue
      lt(2,1) = lb(2)
      lt(2,2) = ub(2)
      lt(1,1) = lb(1)
      lt(1,2) = ub(1)

c
c Calculate the branch metrics.
c
      call branch(bar,0,0,1,1,us(1),ls(1),nn(1))
      call branch(bar,1,0,0,1,us(2),ls(2),nn(2))
      call branch(bar,1,0,0,1,us(3),ls(3),nn(3))
      call branch(bar,0,0,1,1,us(4),ls(4),nn(4))
      call branch(bar,0,1,1,0,us(5),ls(5),nn(5))
      call branch(bar,1,1,0,0,us(6),ls(6),nn(6))
      call branch(bar,1,1,0,0,us(7),ls(7),nn(7))
      call branch(bar,0,1,1,0,us(8),ls(8),nn(8))
      call branch(bar,1,1,0,0,us(9),ls(9),nn(9))
      call branch(bar,0,1,1,0,us(10),ls(10),nn(10))
      call branch(bar,0,1,1,0,us(11),ls(11),nn(11))
      call branch(bar,1,1,0,0,us(12),ls(12),nn(12))
      call branch(bar,1,0,0,1,us(13),ls(13),nn(13))
      call branch(bar,0,0,1,1,us(14),ls(14),nn(14))
      call branch(bar,0,0,1,1,us(15),ls(15),nn(15))
      call branch(bar,1,0,0,1,us(16),ls(16),nn(16))

c
c Reset the overflow flag.
      bar = 1

c
c Perform the add-compare-select calculations.
c
      do 60 as=1,16
        d1 = int(real(as)/2 + 0.5)
        d2 = d1+8
        if (mod(as,2).eq.0) then
          a1 = ls(d1)
          a2 = ls(d2)
          v(as) = 1
        else

```

```

a1 = us(d1)
a2 = us(d2)
v(as) = 0
endif
if (a2.ge.a1) then
nn(as) = a1 cu(as) = 0
else
nn(as) = a2
cu(as) = 1
endif
if (nn(as).ge.64) then
bov = bov + 1
bzi = 2
endif
60      continue
c
c Pass the estimated survivors through the state memory.
c
do 62 mm=32,2,-1
do 55 k=1,16
if (cu(k).eq.1) then
dmu(k,mm) = dmu((int(real(k)/2+0.5)+8),mm-1)
else
dmu(k,mm) = dmu((int(real(k)/2+0.5)),mm-1)
endif
55      continue
62      continue
c
c Use the most recent ACS estimates as the new memory input.
c
do 65 k=1,16
65      dmu(k,1) = v(k)
c
c Use the output decision unit to select the best survivor.
c
do 70 al=1,15
f2 = 2*al
f1 = f2-1
if (al.le.8) then
b1 = dmu(f1,32)
b2 = dmu(f2,32)
else
b1 = v(f1)
b2 = v(f2)
endif
al = nn(f1)

```

```

        a2 = nn(f2)
        if (a2.lt.a1) then
            nn(al+16) = a2
            v(al+16) = b2
        else
            nn(al+16) = a1
            v(al+16) = b1
        endif
70      continue
c
c Assign the logical values to the decoded data.
c
        rxbits(rj) = (v(31).eq.1)
100     continue
c
c Save the last symbol for the next run if the delay is odd.
c
        if (mod(oldbit,2) .eq. 1) then
            symvlp = decisi(bitblk)
        endif
c
c return the block size to its original uncoded value.
        bitblk = bitblk/2
c
c Done decoding.
c
return
        end

        subroutine branch(bar,da,db,dc,dd,upp,lww,smm)
c
c This is a subroutine to calculate the branch metrics.
c
        implicit integer (a - z)
        integer lt(2,2)
        common /fstvit/ lt(2,2)

        ua = da + 1
        ub = db + 1
        lc = dc + 1
        ld = dd + 1
        upp = lt(1,ua) + lt(2,ub) + int(real(smm)/bar)
        lww = lt(1,lc) + lt(2,ld) + int(real(smm)/bar)

        return
        end

```

Appendix B:

The Cadence Edge Schematic Editor generated schematics for the Viterbi Decoder described in Chapter 4 appear in this Appendix. The first eleven fold-out pages detail the actual schematic. The proceeding pages of this Appendix contain schematic of the components or building blocks of the main schematic. The basic building blocks of all the schematics were taken from the Texas Instruments 2 micron Standard Cell Library except the schematics of Drawings 1 - 11 and Drawings 20 and 21. A list of all schematics appear below:

<u>Description.</u>	<u>Drawing</u>
The Viterbi Decoder Front End / Convolutional Encoder.	1
The Branch Metric Calculator	2
The State Metric Calculator	3
The State Metric Calculator	4
Add - Compare - Select Circuitry	5
Add - Compare - Select / Output Decision Circuitry	6
Output Decision Circuitry	7
Path Memory - Part 1	8
Path Memory - Part 2	9
Path Memory - Part 3	10
Path Memory - Part 4	11
Two Bit Adder	12
Two - to - One Multiplexor	13
Latched Two - to - One Multiplexor	14
Two Bit Two - to - One Multiplexor	15
Three - to - Eight Decoder	16
Four Bit Adder	17

Four - to - One Multiplexor

Eight Bit Latch

Output Selection Cell (Acell)

Add - Compare - Select Cell (AC)

Four Bit Counter

Four Bit Identity Comparitor

Eight Bit Magnitude Comparitor

J - K Flip - Flop

Three Bit Latch

18
19
20
21
22
23
24
25
26

APPROVALS		DATE	
DESIGN	DPN	09/90	
CHECKED			
CHECKED			
CHECKED			
REVISION			
C		SCALE	1 OF 11

2

1

REVISIONS				
ZONE	REV	DESCRIPTION	DATE	APPROVED

D

C

2

A

→ esph 11@3D4, 4D4

→ esph 12@3C4, 4D4

→ esph 13@3C4, 4C4

→ esph 14@3C4, 4C

→ esph 15@3B4, 4B4

→ esph 16@3B4, 4B4

→ esph 17@3A4, 4A4

→ esph 18@3A4, 4A4

APPROVALS		DATE	Branch Metric Calc.		
DESIGNED	DPN	09/90			
CHECKED			C		
DESIGNED					
CHECKED					
DESIGNED					

2

1

4

3

APPROVALS	DATE	State Metric Calc.
DRAWN DPN	09/90	
CHECKED		
CHECKED		
CHECKED		
REV	SCALE	3 OF 11

FIG. NO.	3
PAGE NO.	3

APPROVALS		DATE
DESIGN	DPN	09/10
CHECKED		
DESIGNED		
REVIEWED		
DATE		

ADD-Compare_Select

REV C

ISS. FROM NO.

DATE NO.

REV

6 OF 11

ZONE		REV		DESCRIPTION		DATE	APPROVED

APPROVALS		DATE	Output Decision Unit
DESIGNED	DPN	09/90	
CHECKED			
ISSUED			
SCALE		7 OF 11

REVISIONS				
ZONE	REV	DESCRIPTION	DATE	APPROVED

APPROVALS	DATE	Path Memory Part I
DRAWN DPN	09/90	
CHECKED		
CHECKED		
CHECKED		
ISSUED		SCALE C 8 OF 11

D
 C
 A
 8
 8 OF 11

Single_2:1_Mux

APPROVALS		DATE
DESIGN	BPN	09/90
CHECKED		
CHECKED		
CHECKED		
ISSUED		
SHEET NO.		REV
A		
DRAW NO.		
SHEET		OF
4

APPROVALS		DATE	3_to_8_Decoder	
DESIGN	J.P.N.	09/10		
CHECKED				
CHECKED				
CHECKED				
ISSUED				
SIZE		FORM NO.	ENC NO.	REV
A				
SCALE		SHEET OF

2

1

REVISION				
DATE	BY	REVISION	DATE	APPROVED

D

4

APPROVAL	DATE	Output Decision Cell
SPN 09/90		
		1 OF 1

A

2

1

REVISED		REVISED		REVISED	
ZONE	REV	DESCRIPTION	DATE	APPROVED

APPROVALS	DATE	Identity Comp.	
DRAWN DPM	09/50	SIZE	FROM INCH
CHECKED		DATE	REV
CHECKED		DATE	REV
CHECKED		DATE	REV
REUSE		SHEET	OF

APPROVALS		DATE	REVISIONS	
DESIGN	DPN	07/96	ZONE	REV
CHECKED			DESCRIPTION	DATE
CHECKED				APPROVED
CHECKED				
MAGNITUDE COMPARATOR				
SCALE	FROM NO.	TO NO.		
SCALE	2	OF 2		

REVISIONS			
ZONE	REV	DESCRIPTION	DATE

DESIGN	APPROVALS	DATE
000000	DPN	09/19/00
000000		
000000		
000000		
000000		

3_Bit_Latch			
REV	FROM REV	DATE	REV
1	0		
2	1		
3	2		
4	3		
5	4		
6	5		
7	6		
8	7		
9	8		
10	9		
11	10		
12	11		
13	12		
14	13		
15	14		
16	15		
17	16		
18	17		
19	18		
20	19		
21	20		
22	21		
23	22		
24	23		
25	24		
26	25		
27	26		
28	27		
29	28		
30	29		
31	30		
32	31		
33	32		
34	33		
35	34		
36	35		
37	36		
38	37		
39	38		
40	39		
41	40		
42	41		
43	42		
44	43		
45	44		
46	45		
47	46		
48	47		
49	48		
50	49		
51	50		
52	51		
53	52		
54	53		
55	54		
56	55		
57	56		
58	57		
59	58		
60	59		
61	60		
62	61		
63	62		
64	63		
65	64		
66	65		
67	66		
68	67		
69	68		
70	69		
71	70		
72	71		
73	72		
74	73		
75	74		
76	75		
77	76		
78	77		
79	78		
80	79		
81	80		
82	81		
83	82		
84	83		
85	84		
86	85		
87	86		
88	87		
89	88		
90	89		
91	90		
92	91		
93	92		
94	93		
95	94		
96	95		
97	96		
98	97		
99	98		
100	99		

REVISED		DATE		APPROVED	
ZONE	REV	DESCRIPTION

APPROVALS	DATE		
DESIGN DPN	09/90		
OR CDD			
OR CDD			
OR CDD			
BSUB			

J-K_Flip_Flop	
SET FROM NO.	ORC NO.
SET A	REV
SET	OF

Appendix C:

The RATFOR source code for the created SPLICE models is given in this Appendix. A list of the models and their associated model numbers are given below:

<u>Model Description</u>	<u>Model Number</u>
Four Bit Counter	1
Three - to - Eight Decoder	2
J - $\bar{K}$ Flip - Flop	3
Eight Bit Latch	4
Four Bit Adder	5
Two - to - One Multiplexor	6
Eight Bit Magnitude Comparitor	7
Four Bit Latch	8
Four Bit Identity Comparitor	9
Four - to - One Multiplexor	10
Two Bit Adder	11
Two Bit Two - to - One Multiplexor	12
Latched Two - to - One Multiplexor	13

Appendix C

```
.define lgmod1
# Model for a four bit counter
.numout 3
.numinp 2
.nummem 4
.*begin
if (lastim == -1) {
    ivar1 = zero
    ivar2 = zero
    ivar3 = zero
    ivar4 = one
}
if (ivar4 != one & lvin1 == one) {
    if (lvin2 == zero) {
        lvop1 = zero
        lvop2 = zero
        lvop3 = zero
    }
    else {
        if (land(ivar1,land(ivar2,ivar3)).ne.zero) {
            lvop1 = zero
            lvop2 = zero
            lvop3 = zero
        }
        else {
            if (land(lnot(ivar1),land(lnot(ivar2),lnot(ivar3))).ne.zero) {
                lvop1 = one
                lvop2 = zero
                lvop3 = zero
            }
            else
                if (zero.ne.land(ivar1,land(lnot(ivar2),lnot(ivar3)))) {
                    lvop1 = zero
                    lvop2 = one
                    lvop3 = zero
                }
            else
                if (zero.ne.land(lnot(ivar1),land(ivar2,lnot(ivar3)))) {
                    lvop1 = one
                    lvop2 = one
                    lvop3 = zero
                }
            else
                if (zero.ne.land(ivar1,land(ivar2,lnot(ivar3)))) {
```

```

    lvop1 = zero
    lvop2 = zero
    lvop3 = one
}
else
    if (zero.ne.land(lnot(ivar1),land(lnot(ivar2),ivar3))) {
        lvop1 = one
        lvop2 = zero
        lvop3 = on
    }
    else
        if (zero.ne.land(ivar1,land(lnot(ivar2),ivar3))) {
            lvop1 = zero
            lvop2 = one
            lvop3 = one
        }
        else
            if (zero.ne.land(lnot(ivar1),land(ivar2,ivar3))) {
                lvop1 = one
                lvop2 = one
                lvop3 = one
            }
        }
    else {
        lvop1 = ivar1
        lvop2 = ivar2
        lvop3 = ivar3
    }
    ivar4 = lvin1
    ivar1 = lvop1
    ivar2 = lvop2
    ivar3 = lvop3
.end

```

```

.define lgmod2
# model for a 3:8 decoder
.numout 8
.numinp 3
.*begin
if (lastim == -1) {
    lvin1 = zero
    lvin2 = zero
    lvin3 = zero
    lvop1 = zero
    lvop2 = zero
    lvop3 = zero
    lvop4 = zero

```

```

lvop5 = zero
lvop6 = zero
lvop7 = zero
lvop8 = zero
}
else {
if (zero.ne.land(lnot(lvin1),land(lnot(lvin2),lnot(lvin3)))) {
lvop1 = one
lvop2 = zero
lvop3 = zero
lvop4 = zero
lvop5 = zero
lvop6 = zero
lvop7 = zero
lvop8 = zero
}
else
if (zero.ne.land(lvin1,land(lnot(lvin2),lnot(lvin3)))) {
lvop1 = zero
lvop2 = one
lvop3 = zero
lvop4 = zero
lvop5 = zero
lvop6 = zero
lvop7 = zero
lvop8 = zero
}
else
if (zero.ne.land(lnot(lvin1),land(lvin2,lnot(lvin3)))) {
lvop1 = zero
lvop2 = zero
lvop3 = one
lvop4 = zero
lvop5 = zero
lvop6 = zero
lvop7 = zero
lvop8 = zero
}
else
if (zero.ne.land(lvin1,land(lvin2,lnot(lvin3)))) {
lvop1 = zero
lvop2 = zero
lvop3 = zero
lvop4 = one
lvop5 = zero
lvop6 = zero

```



```

lvop7 = zero
lvop8 = zero
}
else
if (zero.ne.land(lnot(lvin1),land(lnot(lvin2),lvin3))) {
lvop1 = zero
lvop2 = zero
lvop3 = zero
lvop4 = zero
lvop5 = one
lvop6 = zero
lvop7 = zero
lvop8 = zero
}
else
if (zero.ne.land(lvin1,land(lnot(lvin2),lvin3))) {
lvop1 = zero
lvop2 = zero
lvop3 = zero
lvop4 = zero
lvop5 = zero
lvop6 = one
lvop7 = zero
lvop8 = zero
}
else
if (zero.ne.land(lnot(lvin1),land(lvin2,lvin3))) {
lvop1 = zero
lvop2 = zero
lvop3 = zero
lvop4 = zero
lvop5 = zero
lvop6 = zero
lvop7 = one
lvop8 = zero
}
else
if (zero.ne.land(lvin1,land(lvin2,lvin3))) {
lvop1 = zero
lvop2 = zero
lvop3 = zero
lvop4 = zero
lvop5 = zero
lvop6 = zero
lvop7 = zero
lvop8 = one

```

```

}}

.end

.define lgmod3
# model for a j-kbar flip flop
.numout 2
.numinp 2
.nummem 2
.*begin
if (lastim == -1) {
    lvop1 = zero
    lvop2 = zero
}
if (zero.ne.lnot(lvin2)) {
    lvop1 = zero
    lvop2 = one
}
else
    if (ivar1 == zero & lvin1 == one) {
        lvop1 = lnot(ivar2)
        lvop2 = ivar2
    }
    else {
        lvop1 = ivar2
        lvop2 = lnot(ivar2)
    }
ivar1 = lvin1
ivar2 = lvop1
.end

.define lgmod4
# model for a 8-bit latch
.numout 8
.numinp 10
.nummem 9
.*begin

if (lastim == -1) {
    lvop1 = zero
    lvop2 = zero
    lvop3 = zero
    lvop4 = zero
    lvop5 = zero
    lvop6 = zero
    lvop7 = zero
    lvop8 = zero

```

```

    }
    else {
        if (lvin10 == zero) {
            lvop1 = zero
            lvop2 = zero
            lvop3 = zero
            lvop4 = zero
            lvop5 = zero
            lvop6 = zero
            lvop7 = zero
            lvop8 = zero
        }
        else {
            if (ivar9 != one & lvin9 == one) {
                lvop1 = lvin1
                lvop2 = lvin2
                lvop3 = lvin3
                lvop4 = lvin4
                lvop5 = lvin5
                lvop6 = lvin6
                lvop7 = lvin7
                lvop8 = lvin8
            }
            else

                lvop1 = ivar1
                lvop2 = ivar2
                lvop3 = ivar3
                lvop4 = ivar4
                lvop5 = ivar5
                lvop6 = ivar6
                lvop7 = ivar7
                lvop8 = ivar8
            }
        }
        ivar1 = lvop1
        ivar2 = lvop2
        ivar3 = lvop3
        ivar4 = lvop4
        ivar5 = lvop5
        ivar6 = lvop6
        ivar7 = lvop7
        ivar8 = lvop8
        ivar9 = lvin9
    end
    .define lgmod5
    # Model for a eight bit adder

```

```

.numout 9
.numinp 18
.nummem 10
.*begin

if (lastim == -1) {
  lvop1 = zero
  lvop2 = zero
  lvop3 = zero
  lvop4 = zero
  lvop5 = zero
  lvop6 = zero
  lvop7 = zero
  lvop8 = zero
  lvop9 = zero
  ivar10 = zero
}
else {
  if (lvin18 != one) {
    lvop1 = zero
    lvop2 = zero
    lvop3 = zero
    lvop4 = zero
    lvop5 = zero
    lvop6 = zero
    lvop7 = zero
    lvop8 = zero
    lvop9 = zero
  }
  else {
    if (ivar9 != one & lvin9 == one) {
      ivar10 = zero
      if (one.ne.land(lvin1,lvin10)) {
        if (one.ne.lor(lvin1,lvin10)) {
          lvop1 = ivar10
          ivar10 = lor(lvin1,lvin10)
        }
      }
      else
      {
        lvop1 = lnot(ivar10)
      }
    }
    else
    {
      lvop1 = ivar10
      ivar10 = land(lvin1,lvin10)
    }
  }
}

```

```

    }
    if (one.ne.land(lvin2,lvin11)) {
        if (one.ne.lor(lvin2,lvin11)) {
            lvop2 = ivar10
            ivar10 = lor(lvin2,lvin11)
        }
        else
        {
            lvop2 = lnot(ivar10)
        }
    }
    else
    {
        lvop2 = ivar10
        ivar10 = land(lvin2,lvin11)
    }
    if (one.ne.land(lvin3,lvin12)) {
        if (one.ne.lor(lvin3,lvin12)) {
            lvop3 = ivar10
            ivar10 = lor(lvin3,lvin12)
        }
        else
        {
            lvop3 = lnot(ivar10)
        }
    }
    else
    {
        lvop3 = ivar10
        ivar10 = land(lvin3,lvin12)
    }
    if (one.ne.land(lvin4,lvin13)) {
        if (one.ne.lor(lvin4,lvin13)) {
            lvop4 = ivar10
            ivar10 = lor(lvin4,lvin13)
        }
        else
        {
            lvop4 = lnot(ivar10)
        }
    }
    else
    {
        lvop4 = ivar10
        ivar10 = land(lvin4,lvin13)
    }

```

```

if (one.ne.land(lvin5,lvin14)) {
  if (one.ne.lor(lvin5,lvin14)) {
    lvop5 = ivar10
    ivar10 = lor(lvin5,lvin14)
  }
  else
  {
    lvop5 = lnot(ivar10)
  }
}
else
{
  lvop5 = ivar10
  ivar10 = land(lvin5,lvin14)
}
if (one.ne.land(lvin6,lvin15)) {
  if (one.ne.lor(lvin6,lvin15)) {
    lvop6 = ivar10
    ivar10 = lor(lvin6,lvin15)
  }
  else
  {
    lvop6 = lnot(ivar10)
  }
}
else
{
  lvop6 = ivar10
  ivar10 = land(lvin6,lvin15)
}
if (one.ne.land(lvin7,lvin16)) {
  if (one.ne.lor(lvin7,lvin16)) {
    lvop7 = ivar10
    ivar10 = lor(lvin7,lvin16)
  }
  else
  {
    lvop7 = lnot(ivar10)
  }
}
else
{
  lvop7 = ivar10
  ivar10 = land(lvin7,lvin16)
}
if (one.ne.land(lvin8,lvin17)) {

```

```

    if (one.ne.lor(lvin8,lvin17)) {
        lvop8 = ivar10
        lvop9 = lor(lvin8,lvin17)
    }
    else
    {
        lvop8 = lnot(ivar10)
        lvop9 = ivar10
    }
}
else
{
    lvop8 = ivar10
    lvar10 = land(lvin8,lvin17) }
}
else {
    lvop1 = ivar1
    lvop2 = ivar2
    lvop3 = ivar3
    lvop4 = ivar4
    lvop5 = ivar5
    lvop6 = ivar6
    lvop7 = ivar7
    lvop8 = ivar8
    lvop9 = ivar10
}}}
ivar9 = lvin9
ivar1 = lvop1
ivar2 = lvop2
ivar3 = lvop3
ivar4 = lvop4
ivar5 = lvop5
ivar6 = lvop6
ivar7 = lvop7
ivar8 = lvop8
ivar10 = lvop9

.end

.define lgmod6
# model for 2:1 mux
.numout 8
.numinp 17
.*begin

if (lastim == -1) {

```

```

lvop1 = zero
lvop2 = zero
lvop3 = zero
lvop4 = zero
lvop5 = zero
lvop6 = zero
lvop7 = zero
lvop8 = zero
}
if (lvin9 == zero) {
lvop1 = lvin1
lvop2 = lvin2
lvop3 = lvin3
lvop4 = lvin4
lvop5 = lvin5
lvop6 = lvin6
lvop7 = lvin7
lvop8 = lvin8
}
else
if (lvin9 == one) {
lvop1 = lvin10
lvop2 = lvin11
lvop3 = lvin12
lvop4 = lvin13
lvop5 = lvin14
lvop6 = lvin15
lvop7 = lvin16
lvop8 = lvin17
}

.end

.define lgmod7
# Model for a 8 bit comparitor.
.numout 3
.numinp 17
.nummem 4
.*begin

integer cnt1,cnt2
if (lastim == -1) {
lvop1 = zero
lvop2 = zero
lvop3 = zero
ivar1 = zero

```



```

    }
    else {
        cnt1 = 0
        cnt2 = 0
        if (ivar1 != one & lvin9 == one){
            if (lvin1 == one) {
                cnt1 = cnt1 + 1
            }
            if (lvin2 == one) {
                cnt1 = cnt1 + 2
            }
            if (lvin3 == one) {
                cnt1 = cnt1 + 4
            }
            if (lvin4 == one) {
                cnt1 = cnt1 + 8
            }
            if (lvin5 == one) {
                cnt1 = cnt1 + 16
            }
            if (lvin6 == one) {
                cnt1 = cnt1 + 32
            }
            if (lvin7 == one) {
                cnt1 = cnt1 + 64
            }
            if (lvin8 == one) {
                cnt1 = cnt1 + 128
            }
            if (lvin10 == one) {
                cnt2 = cnt2 + 1
            }
            if (lvin11 == one) {
                cnt2 = cnt2 + 2
            }
            if (lvin12 == one) {
                cnt2 = cnt2 + 4
            }
            if (lvin13 == one) {
                cnt2 = cnt2 + 8
            }
            if (lvin14 == one) {
                cnt2 = cnt2 + 16
            }
            if (lvin15 == one) {
                cnt2 = cnt2 + 32
            }

```

```

    }
    if (lvin16 == one) {
        cnt2 = cnt2 + 64
    }
    if (lvin17 == one) {
        cnt2 = cnt2 + 128
    }
    if (cnt1 > cnt2) {
        lvop1 = zero
        lvop2 = one
        lvop3 = one
    }
    if (cnt1 == cnt2) {
        lvop1 = one
        lvop2 = zero
        lvop3 = one
    }
    if (cnt1 < cnt2) {
        lvop1 = one
        lvop2 = one
        lvop3 = zero
    }
}
else
{
    lvop1 = ivar2
    lvop2 = ivar3
    lvop3 = ivar4
}
}
ivar1 = lvin9
ivar2 = lvop1
ivar3 = lvop2
ivar4 = lvop3
.end

.define lgmod8
# model for a 4-bit latch
.numinp 5
.numout 4
.nummem 5
.*begin

if (lastim == -1) {
    lvop1 = zero
    lvop2 = zero
    lvop3 = zero

```

```

    lvop4 = zero
}
else {
if (ivar5 != one & lvin5 == one) {
    lvop1 = lvin1
    lvop2 = lvin2
    lvop3 = lvin3
    lvop4 = lvin4
}
else
{
    lvop1 = ivar1
    lvop2 = ivar2
    lvop3 = ivar3
    lvop4 = ivar4
}}
ivar1 = lvop1
ivar2 = lvop2
ivar3 = lvop3
ivar4 = lvop4
ivar5 = lvin5

```

.end

```

.define lgmod9
# Model for a 4 bit comparitor.
.numout 3
.numinp 9
.nummem 4
.*begin

```

```

integer cnt1,cnt2
u' (lastim == -1) {
    lvop1 = zero
    lvop2 = zero
    lvop3 = zero
    ivar1 = zero
}
else {
cnt1 = 0
cnt2 = 0

if (ivar1 != one & lvin5 == one){
    if (lvin1 == one) {
        cnt1 = cnt1 + 1
    }
}

```

```

if (lvin2 == one) {
    cnt1 = cnt1 + 2
}
if (lvin3 == one) {
    cnt1 = cnt1 + 4
}
if (lvin4 == one) {
    cnt1 = cnt1 + 8
}
if (lvin6 == one) {
    cnt2 = cnt2 + 1
}
if (lvin7 == one) {
    cnt2 = cnt2 + 2
}
if (lvin8 == one) {
    cnt2 = cnt2 + 4
}
if (lvin9 == one) {
    cnt2 = cnt2 + 8
}
if (cnt1 > cnt2) {
    lvop1 = zero
    lvop2 = one
    lvop3 = one
}
if (cnt1 == cnt2) {
    lvop1 = one
    lvop2 = zero
    lvop3 = one
}
if (cnt1 < cnt2) {
    lvop1 = one
    lvop2 = one
    lvop3 = zero
}
}
else
{
    lvop1 = ivar2
    lvop2 = ivar3
    lvop3 = ivar4}}
ivar1 = lvin5
ivar2 = lvop1
ivar3 = lvop2
ivar4 = lvop3

```

3

of/de

3

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS
STANDARD REFERENCE MATERIAL 1010a
(ANSI and ISO TEST CHART No. 2)

```

.end

.define lgmod10
# model for 4:1 mux
.numout 4
.numinp 18
.*begin

if (lastim == -1) {
  lvop1 = zero
  lvop2 = zero
  lvop3 = zero
  lvop4 = zero
}
else {
  if (land(!not(lvin1),lnot(lvin2)).ne.zero) {
    lvop1 = lvin3
    lvop2 = lvin4
    lvop3 = lvin5
    lvop4 = lvin6
  }
  else {
    if (land(lvin1,lnot(lvin2)).ne.zero) {
      lvop1 = lvin7
      lvop2 = lvin8
      lvop3 = lvin9
      lvop4 = lvin10
    }
    else {
      if (land(lnot(lvin1),lvin2).ne.zero) {
        lvop1 = lvin11
        lvop2 = lvin12
        lvop3 = lvin13
        lvop4 = lvin14
      }
      else {
        if (land(lvin1,lvin2).ne.zero) {
          lvop1 = lvin15
          lvop2 = lvin16
          lvop3 = lvin17
          lvop4 = lvin18
        } } } }
    }
  }
}

.end

```

```

.define lgmod11
# Model for a two bit adder
.numout 3
.numinp 6
.nummem 5
.*begin

if (lastim == -1) {
  lvop1 = zero
  lvop2 = zero
  lvop3 = zero
  ivar5 = zero
}
else {
  if (lvin6 != one) {
    lvop1 = zero
    lvop2 = zero
    lvop3 = zero
  }
  else {
    if (ivar3 != one & lvin3 == one) {
      ivar5 = zero
      if (one.ne.land(lvin1,lvin4)) {
        if (one.ne.lor(lvin1,lvin4)) {
          lvopi = zero
          ivar5 = zero
        }
      }
      else
      {
        {
          lvop1 = one
          ivar5 = zero
        }
      }
    }
    else
    {
      {
        lvop1 = zero
        ivar5 = one
      }
    }
    if (one.ne.land(lvin2,lvin5)) {
      if (one.ne.lor(lvin2,lvin5)) {
        lvop2 = ivar5
        ivar5 = zero
      }
    }
    else
    {

```

```

    lvop2 = lnot(ivar5)
  }
}
else
{
  lvop1 = ivar5
  lvop3 = one
}
}
else {
  lvop1 = ivar1
  lvop2 = ivar2
  lvop3 = ivar4
}
}
ivar3 = lvin3
ivar1 = lvop1
ivar2 = lvop2
ivar4 = lvop3

.end

.define lgmod12
# model for a single 2:1 mux
.numout 1
.numinp 3
.*begin

if (lastim == -1) {
  lvop1 = zero
}
else {
  if (lvin3 == zero) {
    lvop1 = lvin1
  }
  else
  if (lvin3 == one) {
    lvop1 = lvin2
  }
}

.end

.define lgmod13
# model for a latched 2:1 mux
.numout 8
.numinp 18
.nummem 9

```



```

.*begin

if (lastim == -1) {
    lvop1 = zero
    lvop2 = zero
    lvop3 = zero
    lvop4 = zero
    lvop5 = zero
    lvop6 = zero
    lvop7 = zero
    lvop8 = zero
}
if (lvin18 == one & ivar9 != one) {
    if (lvin9 == zero) {
        lvop1 = lvin1
        lvop2 = lvin2
        lvop3 = lvin3
        lvop4 = lvin4
        lvop5 = lvin5
        lvop6 = lvin6
        lvop7 = lvin7
        lvop8 = lvin8
    }
    else
        if (lvin9 == one) {
            lvop1 = lvin10
            lvop2 = lvin11
            lvop3 = lvin12
            lvop4 = lvin13
            lvop5 = lvin14
            lvop6 = lvin15
            lvop7 = lvin16
            lvop8 = lvin17
        }
    else {
        lvop1 = ivar1
        lvop2 = ivar2
        lvop3 = ivar3
        lvop4 = ivar4
        lvop5 = ivar5
        lvop6 = ivar6
        lvop7 = ivar7
        lvop8 = ivar8
        ivar9 = lvin18
    }
    ivar1 = lvop1

```

```
ivar2 = lvop2  
ivar3 = lvop3  
ivar4 = lvop4  
ivar5 = lvop5  
ivar6 = lvop6  
ivar7 = lvop7  
ivar8 = lvop8  
ivar9 = lvin18
```

```
.end
```

```
.finish
```

Appendix D:

The SPLICE simulation description source code for execution with the CU-SPLICE simulator is given in this Appendix. In the simulator, a data signal is specified which is transferred to the input of a rate = $1/2$, constraint length = 5 convolutional encoder. The symbols generated are sent to the decoder input. As the encoder expects three-bit quantized data symbols, the single bit data symbol is passed to each of the three symbol inputs. If the encoded symbol was a "0" the decoder receives a "000" and similarly for a single "1" the encoder receives a "111". This data is then decoded and the resultant decoded data is available, with the state metric associated with it, at the decoder output.

Any node within the simulation code may be output for plotting. Such a useful feature is made available by specifying the node names in a "wplot" statement. All inputs must be specified by the user through the "input" statement and the ".model" statement. The SPLICE and CU-SPLICE models are referenced by also using the ".model" statement.

Appendix D

splice
viterbi decoder

cmos

;device	r.nodes	model name
count1	aout,bout,cout,clk120,clrz	counter
decod1	ck1,ck2,ck3,ck4,ck5,ck6,ck7,ck8,aout,bout,cout	decoder
notib1	ztlu1,aout	inv1
notc1	zbout,bout	inv1
nottd1	zcout,cout	inv1
nota1	tlu1,ztlu1	inv1
nottc1	tlu2,zbout	inv1
nottc2	tlu3,zcout	inv1
;		
notb1	tup1,aout	inv1
xora1	ztup2,aout,bout	xori2
nott2	tup2,ztup2	inv1
anda1	ck20m,aout,zbout	andi2
jk1	tlw3,ck20,ck20m,clrz	jkflop
notb2	tlw1,tup1	inv1
notc2	tlw2,tup2	inv1
notd2	tup3,tlw3	inv1
;		
not1	zclk120,clk120	inv1
or41	selack,ck2,ck4,ck6,ck8	ori4
nots1	selbck,selack	inv1
nots2	selck,selbck	inv1
or42	cka60,ck1,ck3,ck5,ck7	ori4
notk1	ckb60,cka60	inv1
notk2	ckc60,ckb60	inv1
notk3	ckd60,ckc60	inv1
notk4	cke60,ckd60	inv1
count2	s1,s2,s3,selck,clrz	counter
;		
; The Encoder		
latched	cs1,cs2,cs3,cs4,cs5,cs6,cs7,cs8,din, \$	
	cs1,cs2,cs3,cs4,l1,l1,l1,selck,clrz	latch8

```

addcd1    pt5,cs1,cs2,cs3,cs5          xori4
addcd2    pt6,cs1,cs4,cs5             xori3
muxcd     datain,pt5,pt6,ck60,    mux2
;
;
; The Decoder
latch11   ll11,ll12,ll13,ll14,datain,datain,datain,ll,ck1...latch4
latch12   ll21,ll22,ll23,ll24,datain,datain,datain,ll,ck2 latch4
latch13   ll31,ll32,ll33,ll34,datain,datain,datain,ll,ck3 latch4
latch14   ll41,ll42,ll43,ll44,datain,datain,datain,ll,ck4 latch4
latch15   ll51,ll52,ll53,ll54,datain,datain,datain,ll,ck5 latch4
latch16   ll61,ll62,ll63,ll64,datain,datain,datain,ll,ck6 latch4
latch17   ll71,ll72,ll73,ll74,datain,datain,datain,ll,ck7 latch4
latch18   ll81,ll82,ll83,ll84,datain,datain,datain,ll,ck latch4
;
latch21   lc11,lc12,lc13,lc14,ll11,ll12,ll13,ll14,ck1 latch4
latch22   lc21,lc22,lc23,lc24,ll21,ll22,ll23,ll24,ck1 latch4
latch23   lc31,lc32,lc33,lc34,ll31,ll32,ll33,ll34,ck1 latch4
latch24   lc41,lc42,lc43,lc44,ll41,ll42,ll43,ll44,ck1 latch4
latch25   lc51,lc52,lc53,lc54,ll51,ll52,ll53,ll54,ck1 latch4
latch26   lc61,lc62,lc63,lc64,ll61,ll62,ll63,ll64,ck1 latch4
latch27   lc71,lc72,lc73,lc74,ll71,ll72,ll73,ll74,ck1 latch4
latch28   lc81,lc82,lc83,lc84,ll81,ll82,ll83,ll84,ck1 latch4
;
comp11 agtb11,zaeqb11,altb11,lc11,lc12,lc13,lc14,zclk120, $
tlu1,tlu2,tlu3,ll compar4
comp12 agtb12,zaeqb12,altb12,lc21,lc22,lc23,lc24,zclk120, $
tlu1,tlu2,tlu3,ll compar4
comp13 agtb13,zaeqb13,altb13,lc31,lc32,lc33,lc34,zclk120, $
tlu1,tlu2,tlu3,ll compar4
comp14 agtb14,zaeqb14,altb14,lc41,lc42,lc43,lc44,zclk120, $
tlu1,tlu2,tlu3,ll compar4
comp15 agtb15,zaeqb15,altb15,lc51,lc52,lc53,lc54,zclk120, $
tlu1,tlu2,tlu3,ll compar4
comp16 agtb16,zaeqb16,altb16,lc61,lc62,lc63,lc64,zclk120, $
tlu1,tlu2,tlu3,ll compar4
comp17 agtb17,zaeqb17,altb17,lc71,lc72,lc73,lc74,zclk120, $
tlu1,tlu2,tlu3,ll compar4
comp18 agtb18,zaeqb18,altb18,lc81,lc82,lc83,lc84,zclk120, $
tlu1,tlu2,tlu3,ll compar4

```

```

;
ntcp1      aeqb1,zaeqb11      inv1
ntcp2      aeqb2,zaeqb12      inv1
ntcp3      aeqb3,zaeqb13      inv1
ntcp4      aeqb4,zaeqb14      inv1
ntcp5      aeqb5,zaeqb15      inv1
ntcp6      aeqb6,zaeqb16      inv1
ntcp7      aeqb7,zaeqb17      inv1
ntcp8      aeqb8,zaeqb18      inv1
;
ncp1       aeqb11,clk120,aeqb1      andi2
ncp2       aeqb12,clk120,aeqb2      andi2
ncp3       aeqb13,clk120,aeqb3      andi2
ncp4       aeqb14,clk120,aeqb4      andi2
ncp5       aeqb15,clk120,aeqb5      andi2
ncp6       aeqb16,clk120,aeqb6      andi2
ncp7       aeqb17,clk120,aeqb7      andi2
ncp8       aeqb18,clk120,aeqb8      andi2
;
lth41a     lt11,lt12,lt13,lt14,tup1,tup2,tup3,l1,aeqb11      latch4
lth42a     lt21,lt22,lt23,lt24,tup1,tup2,tup3,l1,aeqb12      latch4
lth43a     lt31,lt32,lt33,lt34,tup1,tup2,tup3,l1,aeqb13      latch4
lth44a     lt41,lt42,lt43,lt44,tup1,tup2,tup3,l1,aeqb14      latch4
lth45a     lt51,lt52,lt53,lt54,tup1,tup2,tup3,l1,aeqb15      latch4
lth46a     lt61,lt62,lt63,lt64,tup1,tup2,tup3,l1,aeqb16      latch4
lth47a     lt71,lt72,lt73,lt74,tup1,tup2,tup3,l1,aeqb17      latch4
lth48a     lt81,lt82,lt83,lt84,tup1,tup2,tup3,l1,aeqb18      latch4
;
lth41b     lw11,lw12,lw13,lw14,tlw1,tlw2,tlw3,l1,aeqb11      latch4
lth42b     lw21,lw22,lw23,lw24,tlw1,tlw2,tlw3,l1,aeqb12      latch4
lth43b     lw31,lw32,lw33,lw34,tlw1,tlw2,tlw3,l1,aeqb13      latch4
lth44b     lw41,lw42,lw43,lw44,tlw1,tlw2,tlw3,l1,aeqb14      latch4
lth45b     lw51,lw52,lw53,lw54,tlw1,tlw2,tlw3,l1,aeqb15      latch4
lth46b     lw61,lw62,lw63,lw64,tlw1,tlw2,tlw3,l1,aeqb16      latch4
lth47b     lw71,lw72,lw73,lw74,tlw1,tlw2,tlw3,l1,aeqb17      latch4
lth48b     lw81,lw82,lw83,lw84,tlw1,tlw2,tlw3,l1,aeqb18      latch4
;
lth51a     mx11,mx12,mx13,mx14,lt11,lt12,lt13,lt14,ck2      latch4
lth52a     mx51,mx52,mx53,mx54,lt21,lt22,lt23,lt24,ck2      latch4
lth53a     mx15,mx16,mx17,mx18,lt31,lt32,lt33,lt34,ck2      latch4

```

```

lth54a    mx55,mx56,mx57,mx58,lt41,lt42,lt43,lt44,ck2    latch4
lth55a    mx19,mx110,mx111,mx112,lt51,lt52,lt53,lt54,ck2    latch4
lth56a    mx59,mx510,mx511,mx512,lt61,lt62,lt63,lt64,ck2    latch4
lth57a    mx113,mx114,mx115,mx116,lt71,lt72,lt73,lt74,ck2    latch4
lth58a    mx513,mx514,mx515,mx516,lt81,lt82,lt83,lt84,ck2    latch4
;
lth51b    mx31,mx32,mx33,mx34,lw11,lw12,lw13,lw14,ck2    latch4
lth52b    mx71,mx72,mx73,mx74,lw21,lw22,lw23,lw24,ck2    latch4
lth53b    mx35,mx36,mx37,mx38,lw31,lw32,lw33,lw34,ck2    latch4
lth54b    mx75,mx76,mx77,mx78,lw41,lw42,lw43,lw44,ck2    latch4
lth55b    mx39,mx310,mx311,mx312,lw51,lw52,lw53,lw54,ck2    latch4
lth56b    mx79,mx710,mx711,mx712,lw61,lw62,lw63,lw64,ck2    latch4
lth57b    mx313,mx314,mx315,mx316,lw71,lw72,lw73,lw74,ck2    latch4
lth58b    mx713,mx714,mx715,mx716,lw81,lw82,lw83,lw84,ck2    latch4
;
mux61     up11,up12,up13,up14,s1,s2,mx11,mx12,mx13,mx14,mx15, $
mx16,mx17,mx18,mx19,mx110,mx111,mx112,mx113,mx114, $
mx115,mx116     mux4
mux63     low11,low12,low13,low14,s1,s2,mx31,mx32,mx33,mx34,mx35, $
mx36,mx37,mx38,mx39,mx310,mx311,mx312,mx313,mx314, $
mx315,mx316     mux4
mux65     up21,up22,up23,up24,s1,s2,mx51,mx52,mx53,mx54,mx55, $
mx56,mx57,mx58,mx59,mx510,mx511,mx512,mx513,mx514, $
mx515,mx516     mux4
mux67     low21,low22,low23,low24,s1,s2,mx71,mx72,mx73,mx74,mx75, $
mx76,mx77,mx78,mx79,mx710,mx711,mx712,mx713,mx714, $
mx715,mx716     mux4
; Next the ACS circuit.
; The branch metrics.
addac1a    au01,au02,au03,au04,au05,au06,au07,au08,car1,up11,up12,up13, $
up14,l1,l1,l1,l1,l1,ck60,up21,up22,up23,up24,l1,l1,l1,l1,ltclr    adder8
addac1b    al01,al02,al03,al04,al05,al06,al07,al08,car1,low11,low12, $
low13,low14,l1,l1,l1,l1,ck60,low21,low22,low23,low24,l1,l1,l1,l1,ltclr    adder8
addac2a    au11,au12,au13,au14,au15,au16,au17,au18,car1,low11,low12, $
low13,low14,l1,l1,l1,l1,ck60,up21,up22,up23,up24,l1,l1,l1,l1,ltclr    adder8
addac2b    al11,al12,al13,al14,al15,al16,al17,al18,car1,up11,up12,up13, $
up14,l1,l1,l1,l1,ck60,low21,low22,low23,low24,l1,l1,l1,l1,ltclr    adder8
addac3a    au21,au22,au23,au24,au25,au26,au27,au28,car1,up11,up12,up13, $
up14,l1,l1,l1,l1,ck60,low21,low22,low23,low24,l1,l1,l1,l1,ltclr    adder8
addac3b    al21,al22,al23,al24,al25,al26,al27,al28,car1,low11,low12, $

```

low13,low14,l1,l1,l1,l1,ck60,up21,up22,up23,up24,l1,l1,l1,l1,ltclr adder8
 addac4a au31,au32,au33,au34,au35,au36,au37,au38,car1,low11,low12, \$
 low13,low14,l1,l1,l1,l1,ck60,low21,low22,low23,low24,l1,l1,l1,l1,ltclr adder8
 addac4b al31,al32,al33,al34,al35,al36,al37,al38,car1,up11,up12,up13, \$
 up14,l1,l1,l1,l1,ck60,up21,up22,up23,up24,l1,l1,l1,l1,ltclr adder8
 ; The state metrics.
 add701a pu01,pu02,pu03,pu04,pu05,pu06,pu07,pu08,car2,au01,au02,au03, \$
 au04,au05,au06,au07,au08,ck60,ns01,ns02,ns03,ns04,ns05,ns06,ns07,ns08,ltclr adder8
 add701b pl01,pl02,pl03,pl04,pl05,pl06,pl07,pl08,car2,al01,al02,al03, \$
 al04,al05,al06,al07,al08,ck60,ns01,ns02,ns03,ns04,ns05,ns06,ns07,ns08,ltclr adder8
 add702a pu11,pu12,pu13,pu14,pu15,pu16,pu17,pu18,car2,au11,au12,au13, \$
 au14,au15,au16,au17,au18,ck60,ns11,ns12,ns13,ns14,ns15,ns16,ns17,ns18,ltclr adder8
 add702b pl11,pl12,pl13,pl14,pl15,pl16,pl17,pl18,car2,al11,al12,al13, \$
 al14,al15,al16,al17,al18,ck60,ns11,ns12,ns13,ns14,ns15,ns16,ns17,ns18,ltclr adder8
 add703a pu21,pu22,pu23,pu24,pu25,pu26,pu27,pu28,car2,au11,au12,au13, \$
 au14,au15,au16,au17,au18,ck60,ns21,ns22,ns23,ns24,ns25,ns26,ns27,ns28,ltclr adder8
 add703b pl21,pl22,pl23,pl24,pl25,pl26,pl27,pl28,car2,al11,al12,al13, \$
 al14,al15,al16,al17,al18,ck60,ns21,ns22,ns23,ns24,ns25,ns26,ns27,ns28,ltclr adder8
 add704a pu31,pu32,pu33,pu34,pu35,pu36,pu37,pu38,car2,au01,au02,au03, \$
 au04,au05,au06,au07,au08,ck60,ns31,ns32,ns33,ns34,ns35,ns36,ns37,ns38,ltclr adder8
 add704b pl31,pl32,pl33,pl34,pl35,pl36,pl37,pl38,car2,al01,al02,al03, \$
 al04,al05,al06,al07,al08,ck60,ns31,ns32,ns33,ns34,ns35,ns36,ns37,ns38,ltclr adder8
 add705a pu41,pu42,pu43,pu44,pu45,pu46,pu47,pu48,car2,au21,au22,au23, \$
 au24,au25,au26,au27,au28,ck60,ns41,ns42,ns43,ns44,ns45,ns46,ns47,ns48,ltclr adder8
 add705b pl41,pl42,pl43,pl44,pl45,pl46,pl47,pl48,car2,al21,al22,al23, \$
 al24,al25,al26,al27,al28,ck60,ns41,ns42,ns43,ns44,ns45,ns46,ns47,ns48,ltclr adder8
 add706a pu51,pu52,pu53,pu54,pu55,pu56,pu57,pu58,car2,au31,au32,au33, \$
 au34,au35,au36,au37,au38,ck60,ns51,ns52,ns53,ns54,ns55,ns56,ns57,ns58,ltclr adder8
 add706b pl51,pl52,pl53,pl54,pl55,pl56,pl57,pl58,car2,al31,al32,al33, \$
 al34,al35,al36,al37,al38,ck60,ns51,ns52,ns53,ns54,ns55,ns56,ns57,ns58,ltclr adder8
 add707a pu61,pu62,pu63,pu64,pu65,pu66,pu67,pu68,car2,au31,au32,au33, \$
 au34,au35,au36,au37,au38,ck60,ns61,ns62,ns63,ns64,ns65,ns66,ns67,ns68,ltclr adder8
 add707b pl61,pl62,pl63,pl64,pl65,pl66,pl67,pl68,car2,al31,al32,al33, \$
 al34,al35,al36,al37,al38,ck60,ns61,ns62,ns63,ns64,ns65,ns66,ns67,ns68,ltclr adder8
 add708a pu71,pu72,pu73,pu74,pu75,pu76,pu77,pu78,car2,au21,au22,au23, \$
 au24,au25,au26,au27,au28,ck60,ns71,ns72,ns73,ns74,ns75,ns76,ns77,ns78,ltclr adder8
 add708b pl71,pl72,pl73,pl74,pl75,pl76,pl77,pl78,car2,al21,al22,al23, \$
 al24,al25,al26,al27,al28,ck60,ns71,ns72,ns73,ns74,ns75,ns76,ns77,ns78,ltclr adder8
 add709a pu81,pu82,pu83,pu84,pu85,pu86,pu87,pu88,car2,au31,au32,au33, \$
 au34,au35,au36,au37,au38,ck60,ns81,ns82,ns83,ns84,ns85,ns86,ns87,ns88,ltclr adder8

add709b pl81,pl82,pl83,pl84,pl85,pl86,pl87,pl88,car2,al31,al32,al33, \$
 al34,al35,al36,al37,al38,ck60,ns81,ns82,ns83,ns84,ns85,ns86,ns87,ns88,ltclr adder8
 add710a pu91,pu92,pu93,pu94,pu95,pu96,pu97,pu98,car2,au21,au22,au23, \$
 au24,au25,au26,au27,au28,ck60,ns91,ns92,ns93,ns94,ns95,ns96,ns97,ns98,ltclr adder8
 add710b pl91,pl92,pl93,pl94,pl95,pl96,pl97,pl98,car2,al21,al22,al23, \$
 al24,al25,al26,al27,al28,ck60,ns91,ns92,ns93,ns94,ns95,ns96,ns97,ns98,ltclr adder8
 add711a pu101,pu102,pu103,pu104,pu105,pu106,pu107,pu108,car2,au21, \$
 au22,au23,au24,au25,au26,au27,au28,ck60,ns101,ns102,ns103,ns104,ns105, \$
 ns106,ns107,ns108,ltclr adder8
 add711b pl101,pl102,pl103,pl104,pl105,pl106,pl107,pl108,car2,al21, \$
 al22,al23,al24,al25,al26,al27,al28,ck60,ns101,ns102,ns103,ns104,ns105, \$
 ns106,ns107,ns108,ltclr adder8
 add712a pu111,pu112,pu113,pu114,pu115,pu116,pu117,pu118,car2,au31, \$
 au32,au33,au34,au35,au36,au37,au38,ck60,ns111,ns112,ns113,ns114,ns115, \$
 ns116,ns117,ns118,ltclr adder8
 add712b pl111,pl112,pl113,pl114,pl115,pl116,pl117,pl118,car2,al31, \$
 al32,al33,al34,al35,al36,al37,al38,ck60,ns111,ns112,ns113, \$
 ns114,ns115,ns116,ns117,ns118,ltclr adder8
 add713a pu121,pu122,pu123,pu124,pu125,pu126,pu127,pu128,car2,au11, \$
 au12,au13,au14,au15,au16,au17,au18,ck60,ns121,ns122,ns123, \$
 ns124,ns125,ns126,ns127,ns128,ltclr adder8
 add713b pl121,pl122,pl123,pl124,pl125,pl126,pl127,pl128,car2,al11, \$
 al12,al13,al14,al15,al16,al17,al18,ck60,ns121,ns122,ns123, \$
 ns124,ns125,ns126,ns127,ns128,ltclr adder8
 add714a pu131,pu132,pu133,pu134,pu135,pu136,pu137,pu138,car2,au01, \$
 au02,au03,au04,au05,au06,au07,au08,ck60,ns131,ns132,ns133, \$
 ns134,ns135,ns136,ns137,ns138,ltclr adder8
 add714b pl131,pl132,pl133,pl134,pl135,pl136,pl137,pl138,car2,al01, \$
 al02,al03,al04,al05,al06,al07,al08,ck60,ns131,ns132,ns133, \$
 ns134,ns135,ns136,ns137,ns138,ltclr adder8
 add715a pu141,pu142,pu143,pu144,pu145,pu146,pu147,pu148,car2,au01, \$
 au02,au03,au04,au05,au06,au07,au08,ck60,ns141,ns142,ns143, \$
 ns144,ns145,ns146,ns147,ns148,ltclr adder8
 add715b pl141,pl142,pl143,pl144,pl145,pl146,pl147,pl148,car2,al01, \$
 al02,al03,al04,al05,al06,al07,al08,ck60,ns141,ns142,ns143, \$
 ns144,ns145,ns146,ns147,ns148,ltclr adder8
 add716a pu151,pu152,pu153,pu154,pu155,pu156,pu157,pu158,car2,au11, \$
 au12,au13,au14,au15,au16,au17,au18,ck60,ns151,ns152,ns153, \$
 ns154,ns155,ns156,ns157,ns158,ltclr adder8
 add716b pl151,pl152,pl153,pl154,pl155,pl156,pl157,pl158,car2,al 1, \$

```

al12,al13,al14,al15,al16,al17,al18,ck60,ns151,ns152,ns153, $
ns154,ns155,ns156,ns157,ns158,ltclr      adder8
;ACS Unit for State 0
cmp701a agtb701a,aeqb701a,altb701a,pu01,pu02,pu03,pu04,pu05,pu06, $
pu07,pu08,selck,pu81,pu82,pu83,pu84,pu85,pu86,pu87,pu88      compar
mx701a   ns01,ns02,ns03,ns04,agtb701a,bt7,pu81,pu82,pu83,pu84,pu01, $
pu02,pu03,pu04,pu82,pu83,pu84,pu85,pu02,pu03,pu04,pu05      mux4
mx701b   ns05,ns06,ns07,ns08,agtb701a,bt7,pu85,pu86,pu87,pu88,pu05, $
pu06,pu07,pu08,pu86,pu87,pu88,l1,pu06,pu07,pu08,l1      mux4
ltch01b   nn01,nn02,nn03,nn04,nn05,nn06,nn07,nn08,ns01,ns02,ns03,ns04, $
ns05,ns06,ns07,ns08,ck60,ltclr      latch8
;ACS Unit For State 1
cmp702a   agtb702a,aeqb702a,altb702a,pl01,pl02,pl03,pl04,pl05,pl06, $
pl07,pl08,selck,pl81,pl82,pl83,pl84,pl85,pl86,pl87,pl88      compar
mx702a   ns11,ns12,ns13,ns14,agtb702a,bt7,pl81,pl82,pl83,pl84,pl01, $
pl02,pl03,pl04,pl82,pl83,pl84,pl85,pl02,pl03,pl04,pl05      mux4
mx702b   ns15,ns16,ns17,ns18,agtb702a,bt7,pl85,pl86,pl87,pl88,pl05, $
pl06,pl07,pl08,pl86,pl87,pl88,l1,pl06,pl07,pl08,l1      mux4
ltch02b   nn11,nn12,nn13,nn14,nn15,nn16,nn17,nn18,ns11,ns12,ns13,ns14, $
ns15,ns16,ns17,ns18,ck60,ltclr      latch8
;ACS Unit For State 2
cmp703a   agtb703a,aeqb703a,altb703a,pu11,pu12,pu13,pu14,pu15,pu16, $
pu17,pu18,selck,pu91,pu92,pu93,pu94,pu95,pu96,pu97,pu98      compar
mx703a   ns21,ns22,ns23,ns24,agtb703a,bt7,pu91,pu92,pu93,pu94,pu11, $
pu12,pu13,pu14,pu92,pu93,pu94,pu95,pu12,pu13,pu14,pu15      mux4
mx703b   ns25,ns26,ns27,ns28,agtb703a,bt7,pu95,pu96,pu97,pu98,pu15, $
pu16,pu17,pu18,pu96,pu97,pu98,l1,pu16,pu17,pu18,l1      mux4
ltch03b   nn21,nn22,nn23,nn24,nn25,nn26,nn27,nn28,ns21,ns22,ns23,ns24, $
ns25,ns26,ns27,ns28,ck60,ltclr      latch8
;ACS Unit For State 3
cmp704a   agtb704a,aeqb704a,altb704a,pl11,pl12,pl13,pl14,pl15,pl16, $
pl17,pl18,selck,pl91,pl92,pl93,pl94,pl95,pl96,pl97,pl98      compar
mx704a   ns31,ns32,ns33,ns34,agtb704a,bt7,pl91,pl92,pl93,pl94,pl11, $
pl12,pl13,pl14,pl92,pl93,pl94,pl95,pl12,pl13,pl14,pl15      mux4
mx704b   ns35,ns36,ns37,ns38,agtb704a,bt7,pl95,pl96,pl97,pl98,pl15, $
pl16,pl17,pl18,pl96,pl97,pl98,l1,pl16,pl17,pl18,l1      mux4
ltch04b   nn31,nn32,nn33,nn34,nn35,nn36,nn37,nn38,ns31,ns32,ns33,ns34, $
ns35,ns36,ns37,ns38,ck60,ltclr      latch8
;ACS Unit For State 4
cmp705a   agtb705a,aeqb705a,altb705a,pu21,pu22,pu23,pu24,pu25,pu26, $

```

```

pu27,pu28,selck,pu101,pu102,pu103,pu104,pu105,pu106,pu107,pu108    compar
mx705a    ns41,ns42,ns43,ns44,agtb705a,bt7,pu101,pu102,pu103,pu104,pu21,$
pu22,pu23,pu24,pu102,pu103,pu104,pu105,pu22,pu23,pu24,pu25    mux4
mx705b    ns45,ns46,ns47,ns48,agtb705a,bt7,pu105,pu106,pu107,pu108,pu25,$
pu26,pu27,pu28,pu106,pu107,pu108,l1,pu26,pu27,pu28,l1    mux4
ltch05b    nn41,nn42,nn43,nn44,nn45,nn46,nn47,nn48,ns41,ns42,ns43,ns44, $
ns45,ns46,ns47,ns48,ck60,ltclr    latch8
;ACS Unit For State 5
cmp706a    agtb706a,aeqb706a,altb706a,pl21,pl22,pl23,pl24,pl25,pl26, $
pl27,pl28,selck,pl101,pl102,pl103,pl104,pl105,pl106,pl107,pl108    compar
rx706a    ns51,ns52,ns53,ns54,agtb706a,bt7,pl101,pl102,pl103,pl104,pl21,$
pl22,pl23,pl24,pl102,pl103,pl104,pl105,pl22,pl23,pl24,pl25    mux4
mx706b    ns55,ns56,ns57,ns58,agtb706a,bt7,pl105,pl106,pl107,pl108,pl25,$
pl26,pl27,pl28,pl106,pl107,pl108,l1,pl26,pl27,pl28,l1    mux4
ltch06b    nn51,nn52,nn53,nn54,nn55,nn56,nn57,nn58,ns51,ns52,ns53,ns54, $
ns55,ns56,ns57,ns58,ck60,ltclr    latch8
;ACS Unit For State 6
cmp707a    agtb707a,aeqb707a,altb707a,pu31,pu32,pu33,pu34,pu35,pu36, $
pu37,pu38,selck,pu111,pu112,pu113,pu114,pu115,pu116,pu117,pu118    compar
mx707a    ns61,ns62,ns63,ns64,agtb707a,bt7,pu111,pu112,pu113,pu114,pu31,$
pu32,pu33,pu34,pu112,pu113,pu114,pu115,pu32,pu33,pu34,pu35    mux4
mx707b    ns65,ns66,ns67,ns68,agtb707a,bt7,pu115,pu116,pu117,pu118,pu35,$
pu36,pu37,pu38,pu116,pu117,pu118,l1,pu36,pu37,pu38,l1    mux4
ltch07b    nn61,nn62,nn63,nn64,nn65,nn66,nn67,nn68,ns61,ns62,ns63,ns64, $
ns65,ns66,ns67,ns68,ck60,ltclr    latch8
;ACS Unit For State 7
cmp708a    agtb708a,aeqb708a,altb708a,pl31,pl32,pl33,pl34,pl35,pl36, $
pl37,pl38,selck,pl111,pl112,pl113,pl114,pl115,pl116,pl117,pl118    compar
mx708a    ns71,ns72,ns73,ns74,agtb708a,bt7,pl111,pl112,pl113,pl114,pl31,$
pl32,pl33,pl34,pl112,pl113,pl114,pl115,pl32,pl33,pl34,pl35    mux4
mx708b    ns75,ns76,ns77,ns78,agtb708a,bt7,pl115,pl116,pl117,pl118,pl35,$
pl36,pl37,pl38,pl116,pl117,pl118,l1,pl36,pl37,pl38,l1    mux4
ltch08b    nn71,nn72,nn73,nn74,nn75,nn76,nn77,nn78,ns71,ns72,ns73,ns74, $
ns75,ns76,ns77,ns78,ck60,ltclr    latch8
;ACS Unit For State 8
cmp709a    agtb709a,aeqb709a,altb709a,pu41,pu42,pu43,pu44,pu45,pu46, $
pu47,pu48,selck,pu121,pu122,pu123,pu124,pu125,pu126,pu127,pu128    compar
mx709a    ns81,ns82,ns83,ns84,agtb709a,bt7,pu121,pu122,pu123,pu124,pu41,$
pu42,pu43,pu44,pu122,pu123,pu124,pu125,pu42,pu43,pu44,pu45    mux4
mx709b    ns85,ns86,ns87,ns88,agtb709a,bt7,pu125,pu126,pu127,pu128,pu45,$

```

```

pu46,pu47,pu48,pu126,pu127,pu128,l1,pu46,pu47,pu48,l1      mux4
ltch09b      nn81,nn82,nn83,nn84,nn85,nn86,nn87,nn88,ns81,ns82,ns83,ns84, $
ns85,ns86,ns87,ns88,ck60,ltclr      latch8
;ACS Unit For State 9
cmp710a      agtb710a,aeqb710a,altb710a,pl41,pl42,pl43,pl44,pl45,pl46, $
pl47,pl48,selck,pl121,pl122,pl123,pl124,pl125,pl126,pl127,pl128      compar
mx710a      ns91,ns92,ns93,ns94,agtb710a,bt7,pl121,pl122,pl123,pl124,pl41,$
pl42,pl43,pl44,pl122,pl123,pl124,pl125,pl42,pl43,pl44,pl45      mux4
mx710b      ns95,ns96,ns97,ns98,agtb710a,bt7,pl125,pl126,pl127,pl128,pl45,$
pl46,pl47,pl48,pl126,pl127,pl128,l1,pl46,pl47,pl48,l1      mux4
ltch10b      nn91,nn92,nn93,nn94,nn95,nn96,nn97,nn98,ns91,ns92,ns93,ns94, $
ns95,ns96,ns97,ns98,ck60,ltclr      latch8
;ACS Unit for State 10
cmp711a      agtb711a,aeqb711a,altb711a,pu51,pu52,pu53,pu54,pu55, $
pu56,pu57,pu58,selck,pu131,pu132,pu133,pu134,pu135,pu136,pu137,pu138      compar
mx711a      ns101,ns102,ns103,ns104,agtb711a,bt7,pu131,pu132,pu133,pu134, $
pu51,pu52,pu53,pu54,pu132,pu133,pu134,pu135,pu52,pu53,pu54,pu55      mux4
mx711b      ns105,ns106,ns107,ns108,agtb711a,bt7,pu135,pu136,pu137,pu138, $
pu55,pu56,pu57,pu58,pu136,pu137,pu138,l1,pu56,pu57,pu58,l1      mux4
ltch11b      nn101,nn102,nn103,nn104,nn105,nn106,nn107,nn108,ns101,ns102, $
ns103,ns104,ns105,ns106,ns107,ns108,ck60,ltclr      latch8
;ACS Unit for State 11
cmp712a      agtb712a,aeqb712a,altb712a,pl51,pl52,pl53,pl54,pl55,pl56, $
pl57,pl58,selck,pl131,pl132,pl133,pl134,pl135,pl136,pl137,pl138      compar
mx712a      ns111,ns112,ns113,ns114,agtb712a,bt7,pl131,pl132,pl133,pl134,$
pl51,pl52,pl53,pl54,pl132,pl133,pl134,pl135,pl52,pl53,pl54,pl55      mux4
mx712b      ns115,ns116,ns117,ns118,agtb712a,bt7,pl135,pl136,pl137,pl138,$
pl55,pl56,pl57,pl58,pl136,pl137,pl138,l1,pl56,pl57,pl58,l1      mux4
ltch12b      nn111,nn112,nn113,nn114,nn115,nn116,nn117,nn118,ns111,ns112, $
ns113,ns114,ns115,ns116,ns117,ns118,ck60,ltclr      latch8
;ACS Unit for State 12
cmp713a      agtb713a,aeqb713a,altb713a,pu61,pu62,pu63,pu64,pu65, $
pu66,pu67,pu68,selck,pu141,pu142,pu143,pu144,pu145,pu146,pu147,pu148      compar
mx713a      ns121,ns122,ns123,ns124,agtb713a,bt7,pu141,pu142,pu143,pu144, $
pu61,pu62,pu63,pu64,pu142,pu143,pu144,pu145,pu62,pu63,pu64,pu65      mux4
mx713b      ns125,ns126,ns127,ns128,agtb713a,bt7,pu145,pu146,pu147,pu148, $
pu65,pu66,pu67,pu68,pu146,pu147,pu148,l1,pu66,pu67,pu68,l1      mux4
ltch13b      nn121,nn122,nn123,nn124,nn125,nn126,nn127,nn128,ns121,ns122, $
ns123,ns124,ns125,ns126,ns127,ns128,ck60,ltclr      latch8
;ACS Unit for State 13

```

```

cmp714a      agtb714a,aeqb714a,altb714a,pl61,pl62,pl63,pl64,pl65,pl66, $
pl67,pl68,selck,pl141,pl142,pl143,pl144,pl145,pl146,pl147,pl148      compar
mx714a      ns131,ns132,ns133,ns134,agtb714a,bt7,pl141,pl142,pl143,pl144,$
pl61,pl62,pl63,pl64,pl142,pl143,pl144,pl145,pl62,pl63,pl64,pl65      mux4
mx714b      ns135,ns136,ns137,ns138,agtb714a,bt7,pl145,pl146,pl147,pl148,$
pl65,pl66,pl67,pl68,pl146,pl147,pl148,l1,pl66,pl67,pl68,l1      mux4
ltch14b      nn131,nn132,nn133,nn134,nn135,nn136,nn137,nn138,ns131,ns132, $
ns133,ns134,ns135,ns136,ns137,ns138,ck60,ltclr      latch8
;ACS Unit for State 14
cmp715a      agtb715a,aeqb715a,altb715a,pu71,pu72,pu73,pu74,pu75,pu76, $
pu77,pu78,selck,pu151,pu152,pu153,pu154,pu155,pu156,pu157,pu158      compar
mx715a      ns141,ns142,ns143,ns144,agtb715a,bt7,pu151,pu152,pu153,pu154, $
pu71,pu72,pu73,pu74,pu152,pu153,pu154,pu175,pu72,pu73,pu74,pu75      mux4
mx715b      ns145,ns146,ns147,ns148,agtb715a,bt7,pu155,pu156,pu157,pu158, $
pu75,pu76,pu77,pu78,pu156,pu157,pu158,l1,pu76,pu77,pu78,l1      mux4
ltch15b      nn141,nn142,nn143,nn144,nn145,nn146,nn147,nn148,ns141,ns142, $
ns143,ns144,ns145,ns146,ns147,ns148,ck60,ltclr      latch8
;ACS Unit for State 15
cmp716a      agtb716a,aeqb716a,altb716a,pl71,pl72,pl73,pl74,pl75,pl76, $
pl77,pl78,selck,pl151,pl152,pl153,pl154,pl155,pl156,pl157,pl158      compar
mx716a      ns151,ns152,ns153,ns154,agtb716a,bt7,pl151,pl152,pl153,pl154,$
pl71,pl72,pl73,pl74,pl152,pl153,pl154,pl155,pl72,pl73,pl74,pl75      mux4
mx716b      ns155,ns156,ns157,ns158,agtb716a,bt7,pl155,pl156,pl157,pl158,$
pl75,pl76,pl77,pl78,pl156,pl157,pl158,l1,pl76,pl77,pl78,l1      mux4
ltch16b      nn151,nn152,nn153,nn154,nn155,nn156,nn157,nn158,ns151,ns152, $
ns153,ns154,ns155,ns156,ns157,ns158,ck60,ltclr      latch8
; Overflow detection
;
ovf1      pov1,nn07,nn17,nn27,nn37      ori4
ovf2      pov2,nn47,nn57,nn67,nn77      ori4
ovf3      pov3,nn87,nn97,nn107,nn117      ori4
ovf4      pov4,nn127,nn137,nn147,nn157      ori4
ovf5      pov5,pov1,pov2,pov3,pov4      ori4
ovf6      bti7,xv,xv,xv,pov5,l1,l1,l1,sel      latch4
noti71     btj7,bti7      inv1
noti72     bt7,btj7      inv1
; Next is the PATH MEMORY:
;
; Path 0
ntpm1      pmc1,agtb701a      inv1

```

```

mxpm1a    pmd01,pmd02,pmd03,pmd04,pmd05,pmd06,pmd07,pmd08,l1,pmd01, $
pmd02,pmd03,pmd04,pmd05,pmd06,pmd07,pmc1,l1,pmd81,pmd82, $
pmd83,pmd84,pmd85,pmd86,pmd87,selck      cmux
mxpm1b    pmd09,pmd010,pmd011,pmd012,pmd013,pmd014,pmd015,pmd016, $
pmd08,pmd09,pmd010,pmd011,pmd012,pmd013,pmd014,pmd015,pmc1,$
pmd88,pmd89,pmd810,pmd811,pmd812,pmd813,pmd814,pmd815,selck      cmux
mxpm1c    pmd016,pmd017,pmd018,pmd019,odu1,p0,p0,p0, $
pmd015,pmd016,pmd017,pmd018,pmd019,l1,l1,l1,pmc1,$
pmd815,pmd816,pmd817,pmd818,pmd819,l1,l1,l1,selck      cmux
; Path 1
ntpm2     pmc2,agtb702      invl
mxpm2a    pmd11,pmd12,pmd13,pmd14,pmd15,pmd16,pmd17,pmd18,h1,pmd01, $
pmd02,pmd03,pmd04,pmd05,pmd06,pmd07,pmc2,h1,pmd81,pmd82, $
pmd83,pmd84,pmd85,pmd86,pmd87,selck      cmux
mxpm2b    pmd19,pmd110,pmd111,pmd112,pmd113,pmd114,pmd115,pmd116, $
pmd08,pmd09,pmd010,pmd011,pmd012,pmd013,pmd014,pmd015,pmc2,$
pmd88,pmd89,pmd810,pmd811,pmd812,pmd813,pmd814,pmd815,selck      cmux
mxpm2c    pmd116,pmd117,pmd118,pmd119,odu2,p0,p0,p0, $
pmd015,pmd016,pmd017,pmd018,pmd019,l1,l1,l1,pmc2,$
pmd815,pmd816,pmd817,pmd818,pmd819,l1,l1,l1,selck      cmux
; Path 2
ntpm3     pmc3,agtb703      invl
mxpm3a    pmd21,pmd22,pmd23,pmd24,pmd25,pmd26,pmd27,pmd28,l1,pmd11, $
pmd12,pmd13,pmd14,pmd15,pmd16,pmd17,pmc3,l1,pmd91,pmd92, $
pmd93,pmd94,pmd95,pmd96,pmd97,selck      cmux
mxpm3b    pmd29,pmd210,pmd211,pmd212,pmd213,pmd214,pmd215,pmd216, $
pmd18,pmd19,pmd110,pmd111,pmd112,pmd113,pmd114,pmd115,pmc3,$
pmd98,pmd99,pmd910,pmd911,pmd912,pmd913,pmd914,pmd915,selck      cmux
mxpm3c    pmd216,pmd217,pmd218,pmd219,odu3,p0,p0,p0, $
pmd115,pmd116,pmd117,pmd118,pmd119,l1,l1,l1,pmc3,$
pmd915,pmd916,pmd917,pmd918,pmd919,l1,l1,l1,selck      cmux
; Path 3
ntpm4     pmc4,agtb704      invl
mxpm4a    pmd31,pmd32,pmd33,pmd34,pmd35,pmd36,pmd37,pmd38,h1,pmd11, $
pmd12,pmd13,pmd14,pmd15,pmd16,pmd17,pmc4,h1,pmd91,pmd92, $
pmd93,pmd94,pmd95,pmd96,pmd97,selck      cmux
mxpm4b    pmd39,pmd310,pmd311,pmd312,pmd313,pmd314,pmd315,pmd316, $
pmd18,pmd19,pmd110,pmd111,pmd112,pmd113,pmd114,pmd115,pmc4,$
pmd98,pmd99,pmd910,pmd911,pmd912,pmd913,pmd914,pmd915,selck      cmux
mxpm4c    pmd316,pmd317,pmd318,pmd319,odu4,p0,p0,p0, $

```

```

pmd115,pmd116,pmd117,pmd118,pmd119,l1,l1,l1,pmc4,$
pmd915,pmd916,pmd917,pmd918,pmd919,l1,l1,l1,selck      cmux
; Path 4
ntpm5      pmc5,agtb705      inv1
mxpm5a      pmd41,pmd42,pmd43,pmd44,pmd45,pmd46,pmd47,pmd48,l1,pmd21, $
pmd22,pmd23,pmd24,pmd25,pmd26,pmd27,pmc5,l1,pmd101,pmd102, $
pmd103,pmd104,pmd105,pmd106,pmd107,selck      cmux
mxpm5b      pmd49,pmd410,pmd411,pmd412,pmd413,pmd414,pmd415,pmd416, $
pmd28,pmd29,pmd210,pmd211,pmd212,pmd213,pmd214,pmd215,pmc5,$
pmd108,pmd109,pmd1010,pmd1011,pmd1012,pmd1013,pmd1014, $
pmd1015,selck      cmux
mxpm5c      pmd416,pmd417,pmd418,pmd419,odu5,p0,p0,p0, $
pmd215,pmd216,pmd217,pmd218,pmd219,l1,l1,l1,pmc5,$
pmd1015,pmd1016,pmd1017,pmd1018,pmd1019,l1,l1,l1,selck      cmux
; Path 5
ntpm6      pmc6,agtb706      inv1
mxpm6a      pmd51,pmd52,pmd53,pmd54,pmd55,pmd56,pmd57,pmd58,h1,pmd21, $
pmd22,pmd23,pmd24,pmd25,pmd26,pmd27,pmc6,h1,pmd101,pmd102, $
pmd103,pmd104,pmd105,pmd106,pmd107,selck      cmux
mxpm6b      pmd59,pmd510,pmd511,pmd512,pmd513,pmd514,pmd515,pmd516, $
pmd28,pmd29,pmd210,pmd211,pmd212,pmd213,pmd214,pmd215,pmc6,$
pmd108,pmd109,pmd1010,pmd1011,pmd1012,pmd1013,pmd1014, $
pmd1015,selck      cmux
mxpm6c      pmd516,pmd517,pmd518,pmd519,odu6,p0,p0,p0, $
pmd215,pmd216,pmd217,pmd218,pmd219,l1,l1,l1,pmc6,$
pmd1015,pmd1016,pmd1017,pmd1018,pmd1019,l1,l1,l1,selck      cmux
; Path 6
ntpm7      pmc7,agtb707      inv1
mxpm7a      pmd61,pmd62,pmd63,pmd64,pmd65,pmd66,pmd67,pmd68,l1,pmd31, $
pmd32,pmd33,pmd34,pmd35,pmd36,pmd37,pmc7,l1,pmd1101, $
pmd1102,pmd1103,pmd1104,pmd1105,pmd1106,pmd1107,selck      cmux
mxpm7b      pmd69,pmd610,pmd611,pmd612,pmd613,pmd614,pmd615,pmd616, $
pmd38,pmd39,pmd310,pmd311,pmd312,pmd313,pmd314,pmd315,pmc7,$
pmd1108,pmd1109,pmd1110,pmd1111,pmd1112,pmd1113,pmd1114, $
pmd1115,selck      cmux
mxpm7c      pmd616,pmd617,pmd618,pmd619,odu7,p0,p0,p0, $
pmd315,pmd316,pmd317,pmd318,pmd319,l1,l1,l1,pmc7,$
pmd1115,pmd1116,pmd1117,pmd1118,pmd1119,l1,l1,l1,selck      cmux
; Path 7
ntpm8      pmc8,agtb708      inv1

```

```

mxpm8a    pmd71,pmd72,pmd73,pmd74,pmd75,pmd76,pmd77,pmd78,h1,pmd31, $
pmd32,pmd33,pmd34,pmd35,pmd36,pmd37,pmc8,h1,pmd1101, $
pmd1102,pmd1103,pmd1104,pmd1105,pmd1106,pmd1107,selck    cmux
mxpm8b    pmd79,pmd710,pmd711,pmd712,pmd713,pmd714,pmd715,pmd716, $
pmd38,pmd39,pmd310,pmd311,pmd312,pmd313,pmd314,pmd315,pmc8,$
pmd1108,pmd1109,pmd1110,pmd1111,pmd1112,pmd1113,pmd1114, $
pmd1115,selck    cmux
mxpm8c    pmd716,pmd717,pmd718,pmd719,odu8,p0,p0,p0, $
pmd315,pmd316,pmd317,pmd318,pmd319,l1,l1,l1,pmc8,$
pmd1115,pmd1116,pmd1117,pmd1118,pmd1119,l1,l1,l1,selck    cmux
; Path 8
ntpm9     pmc9,agtb709    inv1
mxpm9a    pmd81,pmd82,pmd83,pmd84,pmd85,pmd86,pmd87,pmd88,l1,pmd41, $
pmd42,pmd43,pmd44,pmd45,pmd46,pmd47,pmc9,l1,pmd121, $
pmd122,pmd123,pmd124,pmd125,pmd126,pmd127,selck    cmux
mxpm9b    pmd89,pmd810,pmd811,pmd812,pmd813,pmd814,pmd815,pmd816, $
pmd48,pmd49,pmd410,pmd411,pmd412,pmd413,pmd414,pmd415,pmc9,$
pmd128,pmd129,pmd1210,pmd1211,pmd1212,pmd1213,pmd1214, $
pmd1215,selck    cmux
mxpm9c    pmd816,pmd817,pmd818,pmd819,odu9,p0,p0,p0, $
pmd415,pmd416,pmd417,pmd418,pmd419,l1,l1,l1,pmc9,$
pmd1215,pmd1216,pmd1217,pmd1218,pmd1219,l1,l1,l1,selck    cmux
; Path 9
ntpm10    pmc10,agtb710    inv1
mxpm10a    pmd91,pmd92,pmd93,pmd94,pmd95,pmd96,pmd97,pmd98,h1,pmd41, $
pmd42,pmd43,pmd44,pmd45,pmd46,pmd47,pmc10,h1,pmd121, $
pmd122,pmd123,pmd124,pmd125,pmd126,pmd127,selck    cmux
mxpm10b    pmd99,pmd910,pmd911,pmd912,pmd913,pmd914,pmd915,pmd916, $
pmd48,pmd49,pmd410,pmd411,pmd412,pmd413,pmd414,pmd415,$
pmc10,pmd128,pmd129,pmd1210,pmd1211,pmd1212,pmd1213, $
pmd1214,pmd1215,selck    cmux
mxpm10c    pmd916,pmd917,pmd918,pmd919,odu10,p0,p0,p0, $
pmd415,pmd416,pmd417,pmd418,pmd419,l1,l1,l1,pmc10,$
pmd1215,pmd1216,pmd1217,pmd1218,pmd1219,l1,l1,l1,selck    cmux
; Path 10
ntpm11    pmc11,agtb711    inv1
mxpm11a    pmd101,pmd102,pmd103,pmd104,pmd105,pmd106,pmd107,pmd108,l1,$
pmd51,pmd52,pmd53,pmd54,pmd55,pmd56,pmd57,pmc11,l1,pmd131, $
pmd132,pmd133,pmd134,pmd135,pmd136,pmd137,selck    cmux
mxpm11b    pmd109,pmd1010,pmd1011,pmd1012,pmd1013,pmd1014,pmd1015, $

```



```

pmd1016,pmd58,pmd59,pmd510,pmd511,pmd512,pmd513,pmd514, $
pmd515,pmc11,pmd138,pmd139,pmd1310,pmd1311,pmd1312,pmd1313,$
pmd1314,pmd1315,selck      cmux
mxpml1c      pmd1016,pmd1017,pmd1018,pmd1019,odu11,p0,p0,p0, $
pmd515,pmd516,pmd517,pmd518,pmd519,11,11,11,pmc11,$
pmd1315,pmd1316,pmd1317,pmd1318,pmd1319,11,11,11,selck      cmux
; Path 11
ntpm12      pmc12,agtb712      inv1
mxpml2a      pmd1101,pmd1102,pmd1103,pmd1104,pmd1105,pmd1106,pmd1107, $
pmd1108,h1,pmd51,pmd52,pmd53,pmd54,pmd55,pmd56,pmd57,pmc12,$
h1,pmd131,pmd132,pmd133,pmd134,pmd135,pmd136,pmd137,selck      cmux
mxpml2b      pmd1109,pmd1110,pmd1111,pmd1112,pmd1113,pmd1114,pmd1115, $
pmd1116,pmd58,pmd59,pmd510,pmd511,pmd512,pmd513,pmd514, $
pmd515,pmc12,pmd138,pmd139,pmd1310,pmd1311,pmd1312,pmd1313,$
pmd1314,pmd1315,selck      cmux
mxpml2c      pmd1116,pmd1117,pmd1118,pmd1119,odu12,p0,p0,p0, $
pmd515,pmd516,pmd517,pmd518,pmd519,11,11,11,pmc12,$
pmd1315,pmd1316,pmd1317,pmd1318,pmd1319,11,11,11,selck      cmux
; Path 12
ntpm13      pmc13,agtb713      inv1
mxpml3a      pmd121,pmd122,pmd123,pmd124,pmd125,pmd126,pmd127, $
pmd128,11,pmd61,pmd62,pmd63,pmd64,pmd65,pmd66,pmd67,pmc13,$
11,pmd141,pmd142,pmd143,pmd144,pmd145,pmd146,pmd147,selck      cmux
mxpml3b      pmd129,pmd1210,pmd1211,pmd1212,pmd1213,pmd1214,pmd1215, $
pmd1216,pmd68,pmd69,pmd610,pmd611,pmd612,pmd613,pmd614, $
pmd615,pmc13,pmd148,pmd149,pmd1410,pmd1411,pmd1412,pmd1413,$
pmd1414,pmd1415,selck      cmux
mxpml3c      pmd1216,pmd1217,pmd1218,pmd1219,odu13,p0,p0,p0, $
pmd615,pmd616,pmd617,pmd618,pmd619,11,11,11,pmc13,$
pmd1415,pmd1416,pmd1417,pmd1418,pmd1419,11,11,11,selck      cmux
; Path 13
ntpm14      pmc14,agtb714      inv1
mxpml4a      pmd131,pmd132,pmd133,pmd134,pmd135,pmd136,pmd137,pmd138, $
h1,pmd61,pmd62,pmd63,pmd64,pmd65,pmd66,pmd67,pmc14,$
h1,pmd141,pmd142,pmd143,pmd144,pmd145,pmd146,pmd147,selck      cmux
mxpml4b      pmd139,pmd1310,pmd1311,pmd1312,pmd1313,pmd1314,pmd1315, $
pmd1316,pmd68,pmd69,pmd610,pmd611,pmd612,pmd613,pmd614, $
pmd615,pmc14,pmd148,pmd149,pmd1410,pmd1411,pmd1412,pmd1413,$
pmd1414,pmd1415,selck      cmux
mxpml4c      pmd1316,pmd1317,pmd1318,pmd1319,odu14,p0,p0,p0, $

```

```

pmd615,pmd616,pmd617,pmd618,pmd619,11,11,11,pmc14,$
pmd1415,pmd1416,pmd1417,pmd1418,pmd1419,11,11,11,selck      cmux
; Path 14
ntpm15      prnc15,agtb715      inv1
mxpml5a      pmd141,pmd142,pmd143,pmd144,pmd145,pmd146,pmd147,pmd148,$
11,pmd71,pmd72,pmd73,pmd74,pmd75,pmd76,pmd77,pmc15,$
11,pmd151,pmd152,pmd153,pmd154,pmd155,pmd156,pmd157,selck      cmux
mxpml5b      pmd149,pmd1410,pmd1411,pmd1412,pmd1413,pmd1414,pmd1415,$
pmd1416,pmd78,pmd79,pmd710,pmd711,pmd712,pmd713,pmd714,$
pmd715,pmc15,pmd158,pmd159,pmd1510,pmd1511,pmd1512,pmd1513,$
pmd1514,pmd1515,selck      cmux
mxpml5c      pmd1416,pmd1417,pmd1418,pmd1419,odu15,p0,p0,p0,$
pmd715,pmd716,pmd717,pmd718,pmd719,11,11,11,pmc15,$
pmd1515,pmd1516,pmd1517,pmd1518,pmd1519,11,11,11,selck      cmux
; Path 15
ntpm16      pmc16,agtb716      inv1
mxpml6a      pmd151,pmd152,pmd153,pmd154,pmd155,pmd156,pmd157,pmd158,$
h1,pmd71,pmd72,pmd73,pmd74,pmd75,pmd76,pmd77,pmc16,$
h1,pmd151,pmd152,pmd153,pmd154,pmd155,pmd156,pmd157,selck      cmux
mxpml6b      pmd159,pmd1510,pmd1511,pmd1512,pmd1513,pmd1514,pmd1515,$
pmd1516,pmd78,pmd79,pmd710,pmd711,pmd712,pmd713,pmd714,$
pmd715,pmc16,pmd158,pmd159,pmd1510,pmd1511,pmd1512,pmd1513,$
pmd1514,pmd1515,selck      cmux
mxpml6c      pmd1516,pmd1517,pmd1518,pmd1519,odu16,p0,p0,p0,$
pmd715,pmd716,pmd717,pmd718,pmd719,11,11,11,pmc16,$
pmd1515,pmd1516,pmd1517,pmd1518,pmd1519,11,11,11,selck      cmux
; The Output Decision ("A") cells are next
;"A" cell 1.
cmpu1      agtba01,aeqba01,altba01,nn01,nn02,nn03,nn04,nn05,nn06,$
nn07,nn08,selck,nn11,nn12,nn13,nn14,nn15,nn16,nn17,nn18      compar
mxa01a      na91,na92,na93,na94,na95,na96,na97,na98,nn11,nn12,nn13,nn14,$
nn15,nn16,nn17,nn18,agtba01,nn01,nn02,nn03,nn04,nn05,nn06,nn07,nn08      mux
mxa01b      vbi9,odu2,odu1,agtba01      mux2
lchao1      no201,no202,no203,no204,no205,no206,no207,no208,na91,na92,$
na93,na94,na95,na96,na97,na98,ck60,ltclr      latch8
;"A" cell 2.
cmpa2      agtba02,aeqba02,altba02,nn21,nn22,nn23,nn24,nn25,nn26,$
nn27,nn28,selck,nn31,nn32,nn33,nn34,nn35,nn36,nn37,nn38      compar
mxa02a      nb91,nb92,nb93,nb94,nb95,nb96,nb97,nb98,nn31,nn32,nn33,nn34,$
nn35,nn36,nn37,nn38,agtba02,nn21,nn22,nn23,nn24,nn25,nn26,nn27,nn28      mux

```

mxa02b vci9,odu4,odu3,agtba02 mux2
 lchao2 no211,no212,no213,no214,no215,no216,no217,no218,nb91,nb92, \$
 nb93,nb94,nb95,nb96,nb97,nb98,ck60,ltclr latch8
 ;"A" cell 3.
 cmpa3 agtba03,aeqba03,altba03,nn41,nn42,nn43,nn44,nn45,nn46, \$
 nn47,nn48,selck,nn51,nn52,nn53,nn54,nn55,nn56,nn57,nn58 compar
 mxa03a na101,na102,na103,na104,na105,na106,na107,na108, \$
 nn51,nn52,nn53,nn54,nn55,nn56,nn57,nn58,agtba03, \$
 nn41,nn42,nn43,nn44,nn45,nn46,nn47,nn48 mux
 mxa03b vbi10,odu6,odu5,agtba03 mux2
 lchao3 no221,no222,no223,no224,no225,no226,no227,no228,na101, \$
 na102,na103,na104,na105,na106,na107,na108,ck60,ltclr latch8
 ;"A" cell 4.
 cmpa4 agtba04,aeqba04,altba04,nn61,nn62,nn63,nn64,nn65,nn66, \$
 nn67,nn68,selck,nn71,nn72,nn73,nn74,nn75,nn76,nn77,nn78 compar
 mxa04a nb101,nb102,nb103,nb104,nb105,nb106,nb107,nb108, \$
 nn71,nn72,nn73,nn74,nn75,nn76,nn77,nn78,agtba04, \$
 nn61,nn62,nn63,nn64,nn65,nn66,nn67,nn68 mux
 mxa04b vci10,odu8,odu7,agtba04 mux2
 lchao4 no231,no232,no233,no234,no235,no236,no237,no238,nb101,nb102, \$
 nb103,nb104,nb105,nb106,nb107,nb108,ck60,ltclr latch8
 ltchal vt9,vc9,vb10,vc10,vbi9,vci9,vbi10,vci10,ck60 latch4
 ;"A" cell 5.
 cmpa5 agtba05,aeqba05,altba05,nn81,nn82,nn83,nn84,nn85,nn86, \$
 nn87,nn88,selck,nn91,nn92,nn93,nn94,nn95,nn96,nn97,nn98 compar
 mxa05a na111,na112,na113,na114,na115,na116,na117,na118, \$
 nn91,nn92,nn93,nn94,nn95,nn96,nn97,nn98,agtba05, \$
 nn81,nn82,nn83,nn84,nn85,nn86,nn87,nn88 mux
 mxa05b vbi11,odu10,odu9,agtba05 mux2
 lchao5 no241,no242,no243,no244,no245,no246,no247,no248,na111,na112, \$
 na113,na114,na115,na116,na117,na118,ck60,ltclr latch8
 ;"A" cell 6.
 cmpa6 agtba06,aeqba06,altba06,nn101,nn102,nn103,nn104,nn105,nn106, \$
 nn107,nn108,selck,nn111,nn112,nn113,nn114,nn115,nn116,nn117,nn118 compar
 mxa06a nb111,nb112,nb113,nb114,nb115,nb116,nb117,nb118, \$
 nn111,nn112,nn113,nn114,nn115,nn116,nn117,nn118,agtba06, \$
 nn101,nn102,nn103,nn104,nn105,nn106,nn107,nn108 mux
 mxa06b vci11,odu11,agtba06,odu12 mux2
 lchao6 no251,no252,no253,no254,no255,no256,no257,no258,nb111,nb112, \$
 nb113,nb114,nb115,nb116,nb117,nb118,ck60,ltclr latch8

;"A" cell 7.

cmpa7 agtba07,aeqba07,altba07,nn121,nn122,nn123,nn124,nn125,nn126, \$
nn127,nn128,selck,nn131,nn132,nn133,nn134,nn135,nn136,nn137,nn138 compar
mxa07a na121,na122,na123,na124,na125,na126,na127,na128, \$
nn131,nn132,nn133,nn134,nn135,nn136,nn137,nn138,agtba07, \$
nn121,nn122,nn123,nn124,nn125,nn126,nn127,nn128 mux
mxa07b vbi12,odu14,odu13,agtba07 mux2
lchao7 no261,no262,no263,no264,no265,no266,no267,no268,na121,na122, \$
na123,na124,na125,na126,na127,na128,ck60,ltclr latch8

;"A" cell 8.

cmpa8 agtba08,aeqba08,altba08,nn141,nn142,nn143,nn144,nn145,nn146, \$
nn147,nn148,selck,nn151,nn152,nn153,nn154,nn155,nn156,nn157,nn158 compar
mxa08a nb121,nb122,nb123,nb124,nb125,nb126,nb127,nb128, \$
nn151,nn152,nn153,nn154,nn155,nn156,nn157,nn158,agtba08, \$
nn141,nn142,nn143,nn144,nn145,nn146,nn147,nn148 mux
mxa08b vci12,odu16,odu15,agtba08 mux2
lchao8 no271,no272,no273,no274,no275,no276,no277,no278,nb121,nb122, \$
nb123,nb124,nb125,nb126,nb127,nb128,ck60,ltclr latch8
ltcha2 vb11,vc11,vb12,vc12,,vbi11,vci11,vbi12,vci12,ck60 latch4

;"A" cell 9.

cmpa9 agtba09,aeqba09,altba09, \$
no201,no202,no203,no204,no205,no206,no207,no208,selck, \$
no211,no212,no213,no214,no215,no216,no217,no218 compar
mxa09a na131,na132,na133,na134,na135,na136,na137,na138, \$
no211,no212,no213,no214,no215,no216,no217,no218,agtba09, \$
no201,no202,no203,no204,no205,no206,no207,no208 mux
mxa09b vbi13,vc9,vb9,agtba09 mux2
lchao9 no301,no302,no303,no304,no305,no306,no307,no308,na131,na132, \$
na133,na134,na135,na136,na137,na138,ck60,ltclr latch8

;"A" cell 10.

cmpa10 agtba10,aeqba10,altba10,no221,no222,no223,no224,no225,no226, \$
no227,no228,selck,no231,no232,no233,no234,no235,no236, \$
no237,no238 compar
mxa10a nb131,nb132,nb133,nb134,nb135,nb136,nb137,nb138, \$
no231,no232,no233,no234,no235,no236,no237,no238,agtba10, \$
no221,no222,no223,no224,no225,no226,no227,no228 mux
mxa10b vci13,vc10,vb10,agtba10 mux2
lchao10 no311,no312,no313,no314,no315,no316,no317,no318,nb131,nb132, \$
nb133,nb134,nb135,nb136,nb137,nb138,ck60,ltclr latch8

;"A" cell 11.

cmpa11 agtba11,aeqba11,altba11,no241,no242,no243,no244,no245,no246, \$
 no247,no248,selck,no251,no252,no253,no254,no255,no256,no257,no258 compar
 mxa11a na141,na142,na143,na144,na145,na146,na147,na148, \$
 no251,no252,no253,no254,no255,no256,no257,no258,agtba11, \$
 no241,no242,no243,no244,no245,no246,no247,no248 mux
 mxa11b vbi14,vcl1,vb11,agtba11 mux2
 lchao11 no321,no322,no323,no324,no325,no326,no327,no328,na141,na142, \$
 na143,na144,na145,na146,na147,na148,ck60,ltclr latch8
 ;"A" cell 12.

cmpa12 agtba12,aeqba12,altba12,no261,no262,no263,no264,no265,no266, \$
 no267,no268,selck,no271,no272,no273,no274,no275,no276,no277,no278 compar
 mxa12a nb141,nb142,nb143,nb144,nb145,nb146,nb147,nb148, \$
 no271,no272,no273,no274,no275,no276,no277,no278,agtba12, \$
 no261,no262,no263,no264,no265,no266,no267,no268 mux
 mxa12b vci14,vcl2,agtba12,vb12 mux2
 lchao12 no331,no332,no333,no334,no335,no336,no337,no338,nb141,nb142, \$
 nb143,nb144,nb145,nb146,nb147,nb148,ck60,ltclr latch8
 ltchb3 vb13,vcl3,vb14,vcl4,vbi13,vci13,vbi14,vci14,ck60 latch4
 ;"A" cell 13.

cmpa13 agtba13,aeqba13,altba13,no301,no302,no303,no304,no305,no306, \$
 no307,no308,selck,no311,no312,no313,no314,no315,no316,no317,no318 compar
 mxa13a na151,na152,na153,na154,na155,na156,na157,na158, \$
 no311,no312,no313,no314,no315,no316,no317,no318,agtba13, \$
 no301,no302,no303,no304,no305,no306,no307,no308 mux
 mxa13b vbi15,vcl3,vb13,agtba13 mux2
 lchao13 no401,no402,no403,no404,no405,no406,no407,no408,na151,na152, \$
 na153,na154,na155,na156,na157,na158,ck60,ltclr latch8
 ;"A" cell 14.

cmpa14 agtba14,aeqba14,altba14,no321,no322,no323,no324,no325,no326, \$
 no327,no328,selck,no331,no332,no333,no334,no335,no336,no337,no338 compar
 mxa14a nb151,nb152,nb153,nb154,nb155,nb156,nb157,nb158, \$
 no331,no332,no333,no334,no335,no336,no337,no338,agtba14, \$
 no321,no322,no323,no324,no325,no326,no327,no328 mux
 mxa14b vci15,vcl4,vb14,agtba14 mux2
 lchao14 no411,no412,no413,no414,no415,no416,no417,no418,nb151,nb152, \$
 nb153,nb154,nb155,nb156,nb157,nb158,ck60,ltclr latch8
 ltchb4 vb15,vcl5,vb16,vcl6,vbi15,vci15,ltclr latch4
 ;"A" cell 15.

cmpa15 agtba15,aeqba15,altba15,no401,no402,no403,no404,no405,no406, \$
 no407,no408,selck,no411,no412,no413,no414,no415,no416,no417,no418 compar

```

mxal5a    nat1,nat2,nat3,nat4,nat5,nat6,nat7,nat8, $
no411,no412,no413,no414,no415,no416,no417,no418,agtba15, $
no401,no402,no403,no404,no405,no406,no407,no408    mux
mxal5b    viout,vc15,vb15,agtba15    mux2
ltchb15    xv,xv,xe,xf,xe1,xe2,xe3,vout,l1,l1,l1,l1,l1,l1,viout,ck60,ltout    latch8
lchao15    f1,f2,f3,f4,f5,f6,f7,f8,nat1,nat2, $
nat3,nat4,nat5,nat6,nat7,nat8,ck60,ltout    latch8

```

```

;inputs

```

```

input1    clk120    inputa
input2    din    inputb
input3    h1    inputh
input4    l1    inputl
input5    clrz    inputz
input6    ltclr    inputlc
input7    l1clr    inputlx
input8    ltout    inputlz

```

```

;analysis

```

```

time 333ps 2us

```

```

;
wplot f1,f2,f3,f4,f5,f6,f7,f8,din,vout

```

```

;model cards

```

```

;.model inputa lsrc: 0 1 0 20n 0 9.75n 10n 19.75n 20n
;.model inputa lsrc: 0 1 0 10n 0 4.75n 5.25n 9.75n 10n
;.model inputb lsrc: 0 1 75ns 80n 0 39.5n 40ns 79.5n 80n
;.model inputb lsrc: 0 1 10ns 40n 0 19.5n 20ns 39.5n 40n
;.model inputh lsrc: 0 1 0 2000n 0 1.5n 2n 2000n
;.model inputl lsrc: 0 1 0 2000n 0 2000n
;.model inputz lsrc: 0 1 0 2000n 0 40ns 40.5ns 2000ns
;.model inputlc lsrc: 0 1 0 2000ns 0 500ns 500.5ns 2000ns
;.model inputlx lsrc: 0 1 0 2000ns 0 70ns 70.5ns 2000ns
;.model inputlz lsrc: 0 1 0 2000ns 0 724ns 730.5ns 2000ns

```

```

;device models

```

```

;.model counter lgmod : model=1 tr=5.26ns,tf=4.96ns
;.model counter lgmod : model=1 tr=5.26ns,tf=5.26ns

```

```

.model nandi3 nand: tr=0.86ns,tf=0.84ns
.model andi2 and: tr=1.1ns,tf=0.94ns
.model inv1 inverter: tr=0.49ns,tf=0.44ns
.model decoder lgmod : model=2 tr=2.28ns,tf=3.08ns
.model orgt1 or: tr=0.82ns,tf=0.92ns
.model jkflop lgmod : model=3 tr=2.36ns tf=2.24ns
.model latch8 lgmod : model=4 tr=3.8ns,tf=3.08ns
.model adder8 lgmod : model=5 tr=3.52ns,tf=3.7ns
.model mux lgmod : model=6 tr=2.46ns,tf=2.38ns
.model compar lgmod : model=7 tr=3.58ns,tf=3.68ns
.model latch4 lgmod : model=8 tr=3.8ns,tf=3.08ns
.model compar4 lgmod : model=9 tr=3.08ns,tf=3.28ns
.model ori4 or: tr=1.07ns,tf=1.47ns
.model mux4 lgmod : model=10 tr=1.79ns,tf=1.54ns
.model adder2 lgmod : model=11 tr=3.52ns,tf=3.7ns
.model mux2 lgmod : model=12 tr=2.46ns,tf=2.38ns
.model cmux lgmod : model=13 tr=2.96ns,tf=2.88ns
.model xori2 xor: tr=1.07ns,tf=1.47ns
.model xori3 xor: tr=1.07ns,tf=1.47ns
.model xori4 xor: tr=1.07ns,tf=1.47ns

```

```

go
end

```

Appendix E:

The results of several CU-SPLICE simulations are contained in this Appendix. The first three plots represent simulation of the Viterbi Decoder of Chapter 4 using TI 2 micron standard cell characteristics. The remaining two plots exhibit the results of simulating utilizing the TI1.2 micron standard cell characteristics. A list of the plots contained in this Appendix is given below.

Alternating Data

- E.1: Simulation using maximum 2 micron propagation delay values.
- E.2: Simulation using nominal 2 micron propagation delay values.
- E.3: Simulation using minimum 2 micron propagation delay values.
- E.4: Simulation using maximum 1.2 micron propagation delay values.
- E.5: Simulation using nominal 1.2 micron propagation delay values.

Constant Zeros

- E.6: Simulation using constant logic zero input.

Constant Ones

- E.7: Simulation using constant logic one input.

Appendix E.1

splice v1.7 viterbi decoder, (file maximum)

Appendix E.2

splice v1.7 viterb1 decoder, (file oneszero)

Appendix E.3

splice v1 7 viterbi decoder, (file minimum)

Appendix E.4

splice v1 7 viterbi_decoder .file viterbi2h

Appendix E.5

splice v1 7 viterbi decoder (ifile 00012)

Appendix E.6

splice v1.7, viterbi decoder, (file: allzerus)

afline v1.7 viterbi_decoder: ifile trell12/

Appendix E.7

END

1 2 0 4 9 1

FIN