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.

**A SYSTOLIC ARRAY APPLICATION TO
THE DECODING OF CONVOLUTIONAL CODES**

by

Lance Graham Wright, P.Eng.

**A thesis submitted to the Faculty of Graduate
Studies and Research in partial fulfilment
of the requirements for the degree of**

Master of Engineering

**Department of Electronics
Faculty of Engineering
Carleton University**

May 13, 1991

© copyright

MCMXCI , Lance Graham Wright

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-68912-9

Canada

**The undersigned recommend to the Faculty of Graduate Studies
and Research the acceptance of the thesis**

**"A Systolic Array Application to
the Decoding of Convolutional Codes"**

**submitted by Lance G. Wright
in partial fulfilment of the requirements
for the degree of Master of Engineering**

Ted Kwasniewski

**Prof. T.A. Kwasniewski
Thesis supervisor**

J.S. Wight

**Prof. J.S. Wight
Chairman,
Department of Electronics**

Carleton University

May 13, 1991

ABSTRACT

Advances in VLSI technology can potentially provide the means of implementing complex communications algorithms on single substrates. However, such algorithms, in their current form, may not be capable of a "brute force" VLSI implementation, due to their complexity. Through careful investigation and modification, however, these algorithms could be adapted for efficient VLSI design. This thesis is applied to complex communications problems, such as the Viterbi Algorithm, to produce a design methodology which integrates the constraints of the technology with the requirements of the algorithm.

To demonstrate the effectiveness of this approach, the VA has been modified and a vertically integrated design methodology has been developed to create a prototype design in a candidate sub-micron CMOS process. The resultant prototype is a single-chip, $r=1/2$, $K=7$, soft-decision, decoder, which provides over 5.2 dB of coding gain at 10^{-5} , and is capable of a baseband throughput rate of approximately 10 Mbps.

ACKNOWLEDGEMENTS

I would like to thank Professor Tad Kwasniewski for his sage advice and confidence in my abilities. I would also like to thank Professor John Knight for (inadvertently) giving me a suitable topic for my thesis.

I would like to thank my wife, Elissa, for her patience, understanding and support throughout most of my university education. I would also like to thank my parents, Graham and Shirley, for taking such pride in my accomplishments.

Partial financial assistance, in the form of tuition only, has been provided by Canadian Marconi Company (CMC), Computing Devices Company (CDC) and Bell-Northern Research (BNR).

The usage of various software and hardware facilities has been provided by the following organizations: the Communications Research Centre (CRC), Bell-Northern Research (BNR), and the Canadian Microelectronics Corporation (CMC).

TABLE OF CONTENTS

ABSTRACT	iii
ACKNOWLEDGEMENTS	iv
TABLE OF CONTENTS	v
LIST OF TABLES	ix
LIST OF ILLUSTRATIONS	x
LIST OF APPENDICES	xii
GLOSSARY OF TERMS	xiii
CHAPTER 1 INTRODUCTION	
1.1 Thesis Objective	1
1.2 Thesis Motivation	2
1.3 Thesis Format	4
CHAPTER 2 CONVOLUTIONAL CODE APPLICATIONS	
2.1 Introduction	7
2.2 Overview of Digital Communications Systems	7
2.3 Overview of Convolutional Codes	9
2.4 Encoding by Use of Convolutional Codes	9
2.5 Decoding Convolutional Codes	18
2.5.1 The Viterbi Algorithm	18
2.5.2 Simplifying the Viterbi Algorithm	22
2.5.3 Implementation of the Viterbi Algorithm	23
2.5.3.1 Path Memory Truncation	24
2.5.3.2 Metric Quantization	24
2.5.4 Performance of the Viterbi Algorithm	25
2.6 Summary	26
CHAPTER 3 VLSI AND SYSTOLIC ARRAY APPLICATIONS	
3.1 Introduction	31
3.2 Existing VLSI Designs	31
3.2.1 Viterbi Decoder Designs	32
3.2.2 Systolic Array Designs	34
3.3 Systolic Array Architectures	35
3.3.1 Systolic Array Definition	36
3.3.2 Systolic Array Example	37
3.4 CMOS VLSI Technology Considerations	39
3.4.1 CMOS VLSI Speed Considerations	40
3.4.1.1 Systolic Array Solution	41
3.4.2 CMOS VLSI Layout Considerations	41
3.4.2.1 Systolic Array Solution	42
3.5 Viterbi Decoding by Systolic Array	43
3.5.1 Potential Mapping of the VA into Systolic Arrays	43

3.5.2	Recent Work on Viterbi Decoding Using Systolic Arrays ..	44
3.5.2.1	Research by Chang and Yao (1985)	44
3.5.2.2	Research by Gulak and Shwedyk (1986)	46
3.5.2.3	Research by Provence and Gupta (1987-88)	46
3.5.2.4	Research by Wen, Wang, Lee and Lin (1988)	47
3.5.2.5	Research by Gulak and Kailath (1988)	48
3.5.2.6	Research by Fettweis and Meyr (1990)	49
3.6	Summary	50
 CHAPTER 4 THE PROPOSED DECODER ARCHITECTURE		
4.1	Introduction	51
4.2	The Systolic-Array Decoder Overview	51
4.3	The Correlation-Array Overview	54
4.3.1	Correlation-Array Background	54
4.3.2	Correlation-Array Idea Development	57
4.3.2.1	Algorithmic Calculation Requirements	57
4.3.2.2	Inter-Processor Coordination	60
4.3.2.3	Operational Data Flow	61
4.3.3	Correlation-Array Processor Utilization Analysis	66
4.4	Required Deviations from the Viterbi Algorithm	66
4.5	Comparison to Alternative Methods	67
4.5.1	Comparison to Uniprocessor Implementation	68
4.5.2	Comparison to Work by Chang and Yao (1985)	68
4.5.3	Comparison to Work by Provence and Gupta (1988)	69
4.5.4	Comparison to Work by Fettweis and Meyr (1990)	69
4.6	Summary	70
 CHAPTER 5 THE SYSDEC DESIGN		
5.1	Introduction	71
5.2	The Systolic-Array Decoder Detailed Design	71
5.2.1	The Systolic-Array Decoder Interface	72
5.2.2	The Systolic-Array Decoder Architecture	73
5.2.2.1	The Decoder-Stage Architecture and Operation	75
5.3	The Correlation-Array Design	79
5.3.1	The Correlation-Array Interface	79
5.3.2	The Correlation-Array Architecture	81
5.3.3	The Correlation-Array Operation	81
5.3.4	The Correlation-Array Processing Element Design	83
5.3.4.1	The Correlation-Array PE Interface	83
5.3.4.2	The Correlation-Array PE Architecture	85
5.3.4.3	The Correlation-Array PE Operation	86
5.4	The Comparison-Array Design	87
5.4.1	The Comparison-Array Interface	87
5.4.2	The Comparison-Array Architecture	89
5.4.3	The Comparison-Array Operation	90
5.4.4	The Comparison-Array Processing Element Design	91
5.4.4.1	The Comparison-Array PE Interface	91
5.4.4.2	The Comparison-Array PE Architecture	92
5.4.4.3	The Comparison-Array PE Operation	93
5.5	The Metric-Latch Array Design	93
5.5.1	The Metric-Latch Array Interface	94

5.5.3	The Metric-Latch Array Operation	96
5.5.4	The Metric-Latch Array Processing Element Design	96
5.5.4.1	The Metric-Latch Array PE Interface	97
5.5.4.2	The Metric-Latch Array PE Architecture	99
5.5.4.3	The Metric-Latch Array PE Operation	99
5.6	Summary	100
CHAPTER 6 THE VERIFICATION ENVIRONMENT		
6.1	Introduction	101
6.2	Digital Architecture Evaluation Environment	101
6.2.1	Introduction to VHDL	102
6.2.2	VHDL Tool Syntax	103
6.2.3	Mapping the Design into VHDL	103
6.3	Coding-Gain Evaluation Environment	104
6.3.1	Introduction to LINKSIM	106
6.3.2	LINKSIM Usage	107
6.3.3	Integration with the LINKSIM Package	108
6.4	Summary	108
CHAPTER 7 VERIFICATION RESULTS		
7.1	Introduction	111
7.2	Results of LINKSIM Simulations	111
7.2.1	BER Results	112
7.2.2	Coding Gain Results	117
7.3	Results of VHDL Simulations	120
7.4	Assessment of Technology Requirements and Constraints	121
7.4.1	SYSDEC Design Complexity	122
7.4.2	Estimated Technology Requirements	125
7.4.2.1	Complete Decoder Implementation	125
7.4.2.2	Single-Stage Implementation	126
7.4.3	Estimated Circuit Speed	127
7.4.4	Estimated Power Requirements	127
7.4.5	Summary of Technology Requirements	128
7.5	Summary	128
CHAPTER 8 RECOMMENDATIONS AND CONCLUSIONS		
8.1	Overview of Achievements and Contributions	130
8.1.1	The Systolic-Array Decoder Simulation Summary	130
8.1.1.1	Coding Gain Performance	130
8.1.1.2	VLSI Implementation Potential	131
8.1.1.3	Throughput Rate	132
8.1.1.4	Alternative Methods	132
8.1.2	Milestones vs Objectives	133
8.1.3	New Ideas	134
8.2	Recommendations	135
8.2.1	Potential Improvements on the Decoder Design	135
8.2.1.1	Physical Improvements	136
8.2.1.2	Additional Operational Features	138
8.2.2	Topics for Future Consideration	139
8.3	Conclusions	140

REFERENCES	141
APPENDIX A : SYSDEC Design Verification History	A-1
A.1 The Symbolic-Array Decoder Detailed Design	A-2
A.2 Design Maturation	A-2
A.3 Design Potential	A-4
APPENDIX B : Sample VHDL Control Files	B-1
B.1 Analyze, Generate and Build Commands	B-2
B.2 Simulation and Report Generation Commands	B-3
B.3 Simulation Trace File	B-3
B.4 Report Generation Control File	B-4
APPENDIX C : Sample VHDL Simulation Output	C-1
C.1 Sample Simulation Execution Output	C-2
C.2 Sample Signal-Trace Report	C-3
APPENDIX D : SYSDEC VHDL Source Listings	D-1
D.1 The Comparison Array	D-3
D.2 The Correlation Array	D-5
D.3 One Row of the Correlation Array	D-7
D.4 The Data-Extraction Unit	D-10
D.5 One Complete Decoder Stage	D-11
D.6 The Trellis-Direction Counter	D-14
D.7 The Metric-Latch Array	D-16
D.8 The Normalization Detector	D-18
D.9 The Comparison Array Processing Element	D-19
D.10 The Correlation Array Processing Element	D-20
D.11 The Metric-Latch Array Processing Element	D-22
D.12 The Symbol-Shifting Unit	D-23
D.13 The SYSDEC Package	D-25
D.14 The Test Bench	D-26
D.15 The Timing-Control Unit	D-32
APPENDIX E : Sample LINKSIM Parameter File	E-1
APPENDIX F : Sample LINKSIM Results File	F-1
APPENDIX G : FORTRAN Code for SYSDEC LINKSIM Simulation	G-1
G.1 FORTRAN Listing for SYSDECIN.F	G-2
G.2 FORTRAN Listing for SYSDEC.F	G-5

LIST OF TABLES

Table Ia.	The State Transitions for a K=7 Encoder	13
Table Ib.	The State Transitions for a K=7 Encoder (continued)	14
Table Ic.	The State Transitions for a K=7 Encoder (continued)	15
Table Id.	The State Transitions for a K=7 Encoder (continued)	16
Table II.	The State Transitions for a K=3 Encoder	17
Table III.	LINKSIM Simulation Results for Hard-Decision, K=7	113
Table IV.	LINKSIM Simulation Results for Soft-Decision, K=7	113
Table V.	LINKSIM Simulation Results for Hard-Decision, K=3	115
Table VI.	LINKSIM Simulation Results for Soft-Decision, K=3	115
Table VII.	Hard-Decision Coding Gains for Constraint Length K=7	118
Table VIII.	Soft-Decision Coding Gains for Constraint Length K=7	118
Table IX.	Hard-Decision Coding Gains for Constraint Length K=3	119
Table X.	Soft-Decision Coding Gains for Constraint Length K=3	119
Table XI.	SYSDEC Parts Count for a Single K=7 Decoder Stage	123
Table XII.	Transistor Requirements for a Single K=7 Decoder Stage	123
Table XIII.	SYSDEC Parts Count for a Single K=3 Decoder Stage	124
Table XIV.	Transistor Requirements for a Single K=3 Decoder Stage	124
Table XV.	Summary of VLSI Parameters vs Technology	129

LIST OF ILLUSTRATIONS

Figure 1. A Communications System Block Diagram	8
Figure 2. A Convolutional Encoder for Constraint-Length $K=7$ Codes	10
Figure 3. A Convolutional Encoder for Constraint-Length $K=3$ Codes	10
Figure 4. Two Stages of a Four-State Trellis Diagram	17
Figure 5. Viterbi Algorithm Performance for Hard-Decision Decoding	27
Figure 6. Viterbi Algorithm Performance for Soft-Decision Decoding	28
Figure 7. Viterbi Algorithm Performance for Constraint-Length $K=7$	29
Figure 8. Viterbi Algorithm Performance for Constraint-Length $K=3$	30
Figure 9. Systolic Array Example of String Pattern Matching	38
Figure 10. Demonstration of Trellis Compaction for $K=3$	45
Figure 11. Two-Dimensional Design of Provence and Gupta	47
Figure 12. The Systolic-Array Decoder Architecture (Single Stage)	53
Figure 13. Two-Dimensional Compacted Trellis Mapping of the VA	56
Figure 14. Single Row of a $K=7$ Correlation-Array	59
Figure 15a. Correlation-Array Data-Flow Behaviour : Time $t = 0$	64
Figure 15b. Correlation-Array Data-Flow Behaviour : Time $t = 1$	64
Figure 15c. Correlation-Array Data-Flow Behaviour : Time $t = 2$	64
Figure 16. The Systolic-Array Decoder Interface	73
Figure 17. The Systolic-Array Decoder Architecture	74
Figure 18. The Systolic-Array Decoder Architecture (Single Stage)	76
Figure 19. The Correlation-Array Interface	80
Figure 20. The Correlation-Array Architecture	82
Figure 21. The Correlation-Array Processing Element Interface	84
Figure 22. The Correlation-Array Processing Element Architecture	86

Figure 23. The Comparison-Array Interface	88
Figure 24. The Comparison-Array Architecture	90
Figure 25. The Comparison-Array Processing Element Interface	92
Figure 26. The Comparison-Array Processing Element Architecture	93
Figure 27. The Metric-Latch Array Interface	95
Figure 28. The Metric-Latch Array Architecture	96
Figure 29. The Metric-Latch Array Processing Element Interface	97
Figure 30. The Metric-Latch Array Processing Element Architecture	100
Figure 31. Hierarchy of the SYSDEC VHDL Implementation	105
Figure 32. LINKSIM Benchmark Results for K=7	109
Figure 33. LINKSIM Benchmark Results for K=3	110
Figure 34. LINKSIM Simulation Results for Constraint-Length K=7	114
Figure 35. LINKSIM Simulation Results for Constraint-Length K=3	116

LIST OF APPENDICES

APPENDIX A :	SYSDEC Design Verification History	A-1
APPENDIX B :	Sample VHDL Control Files	B-1
APPENDIX C :	Sample VHDL Simulation Output	C-1
APPENDIX D :	SYSDEC VHDL Source Listings	D-1
APPENDIX E :	Sample LINKSIM Parameter File	E-1
APPENDIX F :	Sample LINKSIM Results File	F-1
APPENDIX G :	FORTRAN Code for SYSDEC LINKSIM Simulation	G-1

GLOSSARY OF TERMS

ACS	Add-Compare-Select (circuit)
ASIC	Application Specific Integrated Circuit
baud-rate	rate of symbols per second
BER	Bit Error Rate
BPSK	Binary Phase-Shift Keying
bps	bits per second
branch-metric	the error quality associated with a single transition within a trellis structure
BSC	Binary Symmetric Channel
CC	Convolutional Code
CCITT	Consultative Committee on International Telegraphy and Telephony
CMP	Comparator
CMOS	Complimentary Metal Oxide Semiconductor
CPM	Continuous Phase Modulation
CPU	Central Processing Unit
CPSK	Continuous Phase-Shift Keying
Codec	Coder/Decoder
CRC	Communications Research Centre
dB	decibels
DLM	Double Level Metal
DMC	Discrete Memoryless Channel
DoD	(United States) Department of Defense
DPSK	Differential Phase-Shift Keying
DSP	Digital Signal Processing

E	Energy per Bit (W·s)
E_b/N_0	band-limited signal-to-noise ratio (= $10 \text{ Log}_{10} [E/\eta]$ dB)
Erfc	The Complimentary Error Function
FA	Full Adder
FECC	Forward Error Correcting Codes
fF	femtofarad (1 fF = 10^{-15} Farads)
F/F	Flip-Flop
FIFO	First-In, First-Out memory
frame	complete set of k associated symbols from the encoder
FSK	Frequency-Shift Keying
FSM	Finite-State Machine
Gbps	10^9 bits per second
GHz	10^9 Hertz (cycles per second)
HA	Half Adder
IC	Integrated Circuit
I/O	Input/Output
ISI	Inter-Symbol Interference
k	the number of convolutional-encoder input bits
K	the constraint-length of the decoder
LINKSIM	communications Link Simulation package
MAP	Maximum <i>a posteriori</i> Probability
m	the numerical radix of the symbols (for a binary code, $m=2$)
M	the number of states in the convolutional-code encoder
Mbps	10^6 bits per second
MCS	Miller Communications Systems
mil	one-thousandth of one inch (1 mil = $25.4 \mu\text{m}$)

milliwatt	one-thousandth of one Watt
MHz	10^6 Hertz (cycles per second)
mm	millimetre (1 mm = 10^{-3} metres)
MSK	Minimum-Shift Keying
MUX	Multiplexer
n	the number of convolutional-encoder output bits
ns	nanosecond (1 ns = 10^{-9} seconds)
OQPSK	Orthogonal Quaternary-Phase Shift Keying
path-metric	the error quality measure over a complete trellis path
PE	Processing Element
P_e	probability of error
pF	picofarad (1 pF = 10^{-12} Farads)
PM	Path Metric
ps	picosecond (1 ps = 10^{-12} seconds)
PSK	Phase-Shift Keying
QAM	Quaternary Amplitude Modulation
QPSK	Quaternary Phase-Shift Keying
r	the rate of a convolutional code ($r = k/n$)
SA	Systolic Array
SNR	Signal to Noise Ratio (dB)
state-metric	the error quality measure at a given state in the trellis
SYSDEC	mnemonic for this Systolic array Decoder design
T	the number of elements, in a code block of arbitrary length
t	time
TLM	Triple Level Metal
TP	throughput per unit area metric for VLSI (gate-Hz/mm²)

u	sequence of encoder input values
v	the order of the Markov process ($v = K-1$)
VA	Viterbi Algorithm
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLSI	Very Large Scale Integration
X	survivor state sequence
x	encoder state sequence
XOR	Exclusive Or gate
y	sequence of encoded symbols input to the channel
z	sequence of symbols received at the output of the channel
Γ	survivor metric
η	Noise spectral density (W/Hz)
$\lambda(\xi_t)$	branch-metric for transition ξ_t
$\lambda(\xi_0^t)$	path-metric from time 0 to time t for node x_t
μm	one micron ($1 \mu\text{m} = 10^{-6}$ metres)
Ξ	the set of all possible transitions in one stage of a convolutional code trellis
ξ_t	state transition from x_t to x_{t+1}

CHAPTER 1 INTRODUCTION

1.1 Thesis Objective

This thesis will demonstrate, by way of new algorithms and a vertically integrated design methodology for a convolutional code decoder, the potential for using systolic-array architectures in the design of single-chip, very large scale integrated (VLSI) circuits for communications systems. Conclusions will be drawn from this investigation based upon the suitability, theoretical performance and VLSI potential of a design which relies primarily on one- and two-dimensional systolic arrays to implement a modified version of the Viterbi Algorithm. The use of *de facto* standards^a will be applied to this analysis so that the results may be compared against readily available published performance results and future VLSI technology^b availability.

This study is not an actual integrated circuit (IC) design, however, due to the level of detail of this investigation, there exists a high degree of correlation between the information presented in this document and a potential CMOS VLSI design.

-
- ^a The use of rate 1/2, constraint-length K=7 codes will be the primary datum from which comparisons will be made. References will also be made to constraint-length K=3 codes for explanation and comparison purposes. Both hard-decision and eight-level soft-decision will be used. The physical consideration of the study will be targeted for a sub-micron CMOS process.
 - ^b Current industrial (gate-array) CMOS VLSI technology is capable of a 0.8 μm feature size, approximately 180 ps unloaded gate delay, approximately 250,000 raw gates per die (with 60% to 75% utilization), and silicon die sizes approximately 550 mils (14 mm) per side. Note : specific references cannot be quoted due to the preliminary, and confidential, nature of this information.

1.2 Thesis Motivation

The Viterbi Algorithm (VA) [1][2][3][4][5][6] has long been recognized as an effective solution to the decoding of convolutional codes (CC). Early implementations of the VA were performed on conventional von Neumann type uniprocessors. However, due to the high number of calculations required, the throughput was limited. As the need for longer constraint-lengths and greater data rates evolved, it became necessary to create systems composed of parallel processors. This approach solved the throughput problems, but created large decoder structures which were difficult, and expensive, to design.

With the advent of VLSI, it became possible to implement the parallel processors on a single chip. Traditional VLSI implementations of the VA [7][8][9][10], however, have had performance, layout and design difficulties which were caused by a literal interpretation of the CC trellis structure when mapped into a VLSI architecture. An important example of this interpretation is the architectural requirement for a path-memory^c, causing a reduction in throughput and an increase in area due to limited memory access times and increased number of interconnections. In addition, the verbatim layout of the CC trellis relied heavily on global broadcasts and non-local interconnections. This trellis nature created a requirement to connect processor number p (at time t) to processors number $2p$ and $2p+1$ (at time $t+1$, modulo 2^{K-1}). This was acceptable for low memory (encoder or channel) values ($K-1$), but as the memory length increased linearly, the number, and length, of interconnections increased exponentially (2^{K-1}). This not only increased the required silicon area but, due to excessive wire-load capacitance, it also reduced the speed at which the designs could operate.

^c The recommended size of the path memory is 4 or 5 times the constraint-length in depth [2], with a width equal to the number of states in the trellis.

Systolic arrays [11][12][13][14] are known to be area-efficient with respect to interconnection and, when suitably designed, have a high utilization of circuitry. This investigation will demonstrate that such architectures can be used for Viterbi decoding, resulting in a reduction of the number, and length, of interconnections and hence, higher throughput rates.

Work performed, during the fulfilment of this thesis, has resulted in contributions to the following three fields :

Systems Engineering and Design Methodology

The use of a vertically integrated design methodology which traverses the levels of hierarchy (in both directions) from the algorithm and system definition to the transistor-level design.

Communications Theory

The creation of a new algorithm, based on the Viterbi Algorithm, for decoding convolutional codes within systolic arrays, without the requirement for path memory.

VLSI Implementation

The detailed implementation and analysis of a truly systolic maximum likelihood sequence estimator design.

Hence, the proposed approach will address three problems : a) difficulties in designing using only a top-down (or bottom-up) approach, b) design "cost" based on architectural complexity, and c) reduced chip speed due to excessive interconnection capacitance.

1.3 Thesis Format

This thesis is a treatise on the investigation, design, integration and performance of convolutional-code decoders which employ systolic-array architecture(s). The following paragraphs provide a simplified overview of where to find pertinent information.

Chapter 1 contains an introduction to this thesis. It delivers the objective and motivation behind the thesis and an outline of the thesis format.

Chapter 2 reviews the necessary background on communication systems and convolutional codes. It serves as a reference for the performance requirements of a convolutional code decoder. All material presented in Chapter 2 is common knowledge in the fields of communications and information theory.

Chapter 3 reviews the necessary background on VLSI technology and systolic arrays. It also contains a synopsis of recent research on systolic-array implementations with selected applications towards the Viterbi Algorithm. All material presented in Chapter 3 has been extracted from the works of other authors.

Chapter 4 contains a discussion on the specific application of systolic arrays as it pertains to this thesis. It covers the basic premise of a convolutional-code decoder based on a two-dimensional systolic array engine. The detailed design for the systolic arrays, as proposed in Chapter 4, is specified in Chapter 5. All information proposed/presented in Chapter 4 is unique to the author.

Chapter 5 contains all the design information specific to a unique convolutional-code decoder based on a multiple systolic-array architecture. This design will serve as a reference point for evaluating the use of systolic arrays in implementing the Viterbi Algorithm. All information presented in Chapter 5 is unique to the author.

Chapter 6 describes the simulation environments which will be used in the design verification process. Specifically, it will contain introductions to the VHDL tools and the LINKSIM communications link simulation package. Simulation tools presented in Chapter 6 have been obtained for evaluation purposes only. All source code (Appendices D and G), describing the behaviour of the proposed decoder architecture, has been written by the author. Software integration and adaptation of the LINKSIM package has also been performed by the author.

Chapter 7 contains the simulation results for the proposed decoder. It will verify, through statistical simulations, that the coding gain of the design is comparable to acceptable published simulation results of the Viterbi Algorithm. It will also verify, through VHDL simulations, that the architecture may be physically mapped into a digital hardware implementation without excessive degradation in operational speed and coding gain performance.

Chapter 7 also contains the physical-level accounting of the design specified in Chapter 5. It contains sufficient information to verify the potential to implement the design in VLSI. This is accomplished through calculated estimates of the decoder complexity, throughput rate and technology requirements. All results and comparisons presented in Chapter 7 are unique to the author's investigation.

Chapter 8 is a discussion of the information contained throughout this document. This chapter will reiterate the thesis objectives and the results obtained from the analysis of the proposed architecture, and highlight the major contributions made by this thesis. Chapter 8 will also provide useful insight into potential design improvements and areas for future investigation in this field.

Appendix A contains a history of the iterative design verification process which produced the final prototype of the SYSDEC test case.

Appendices D and G contain all the source-code created for this investigation. Appendices B and E contain sample stimulus files used in the simulation environments. Appendices C and F contain sample simulation results which were generated using the stimulus files of Appendices B and E.

CHAPTER 2 CONVOLUTIONAL CODE APPLICATIONS

2.1 Introduction

This chapter provides a brief overview of digital communications systems with emphasis on forward error-correction coding using convolutional codes. It will also outline typical applications for convolutional codes, and provide graphical representations of accepted coding-gain performance parameters. All information presented in this chapter is common knowledge in the fields of communications and information theory.

2.2 Overview of Digital Communications Systems

Figure 1 contains a block diagram of a generic communications system. The system consists of a data source/sink, an interleaver/deinterleaver⁴, a modulator/demodulator, a transmitter/receiver pair and the communications medium (channel). For the purposes of this investigation, the data source/sink is arbitrary, the use of the interleaver/deinterleaver is optional (conditional on the bursty nature of the channel), the modulator/demodulator will use binary phase-shift keying (BPSK), and the receiver will use coherent detection.

The selection and design of a coder/decoder (codec) is heavily dependent on the intended application. Convolutional codes are not very effective on high error-rate links (bit error rates approaching 10^{-1}), nor are they useful for communication links which have

⁴ An interleaver is used to pseudo-randomly shuffle the data ordering such that, when de-interleaved, any bursts of errors will be transformed to discretely spaced "random" errors.

a bursty error distribution (unless data-interleaving is used). In both these cases, the occurrence of multiple localized errors in the decoder path will cause incorrect decisions to be made, and hence, increase the effective bit error rate (or alternatively, a higher energy-per-bit ratio will be required to achieve a comparable bit error rate).

Figure 1. A Communications System Block Diagram

To facilitate the decoder architecture evaluation, this investigation will concentrate on a binary-symmetric^{*} channel (BSC) with additive white Gaussian noise resulting in a BER of 10^{-2} or less. These assumptions are applicable to a variety of classes of digital systems including fibre optics, mobile radio, satellite and deep-space communications. For the purposes of this investigation, a high bit-rate channel (such as in fibre optic links) will be considered as the target application. Inherent in the target application will be system issues regarding complexity, power requirements, reliability, manufacturability and cost. The target application, for example, will require low complexity, high reliability, ease of manufacture and, above all, low cost (due to the number of installations).

* The binary symmetric channel is a special case of the discrete memoryless channel (DMC). It exhibits equal probabilities for binary transpositions, and is not subject to inter-symbol interference (ISI). It has been shown [4][6] that the VA is still effective for channels which have memory (ie: exhibiting intersymbol interference).

2.3 Overview of Convolutional Codes

This section provides a brief overview of convolutional coding theory. It provides the necessary background for evaluating a decoder on the basis of system requirements and performance.

Forward error correction codes (FECC) [15][16][17] perform an *a priori* encoding of source data into transmittable symbols in such a way as to allow for correction of channel errors at the receiver/decoder. Convolutional codes, specifically, accomplish this by encoding the source data so as to affect several sequential transmitted symbols (the number of affected symbols is equal to the product of the constraint-length and the code rate). This technique has been shown [15] to be effective in correcting receiver errors caused by intersymbol interference on Markov channels existing in the presence of Gaussian noise (assuming a reasonably low BER).

2.4 Encoding by Use of Convolutional Codes

Figure 2 shows a block diagram of an encoder for generating convolutional codes⁷. This particular encoder implementation is for a constraint-length⁸ $K=7$, rate $r=1/2$ ($k=1$ bit in and $n=2$ symbols out) convolutional code. Figure 3 demonstrates another encoder,

⁷ Since there is already an abundance of investigative work (and published results) in this field, this investigation will make use of constraint parameters consistent with the majority of convolutionally encoded systems (ie: constraint-length $K=7$, rate $r=1/2$ and eight-level quantization of the received symbols).

⁸ The constraint-length of a convolutional code is an integer which is equal to one more than the number of bits of memory in the encoder (which is also the order of the code-generating polynomial). Alternatively, the constraint-length is equal to the total number of possible taps in the encoder shift register data path.

using the same structure as Figure 2, which creates a constraint-length $K=3$ convolutional code.

Figure 2. A Convolutional Encoder for Constraint-Length $K=7$ Codes

Figure 3. A Convolutional Encoder for Constraint-Length $K=3$ Codes

Data is applied to the shift register at the baseband data rate and two single-bit "symbols" are created according to a set of generating polynomial equations (equivalent to the binary tap coefficients of the encoder's shift-register). The two symbols are then multiplexed onto the transmitter input at the new data rate (the channel rate is equal to twice the baseband data rate, for rate $r=1/2$ codes). Henceforth, one pair of associated

symbol bits will be referred to as a "frame" of symbols. The "frame" may itself be transmitted as a symbol (dependent upon the modulation technique used). However, to avoid ambiguity, the symbol-pair will be referred to as a frame. Note that this implies that the receiver must have some method of frame synchronization.

The structure of the encoder is rudimentary. It consists of a (K-1)-stage linear feed-forward shift register and two sets (corresponding to $n=2$) of optimal taps^a terminating in modulo-2 adders. Equations (1) and (2) represent the two generating polynomials for the K=7 encoder shown in Figure 2. These polynomials will henceforth be referred to by the octal representations of their tap coefficientsⁱ (171₈ and 133₈ respectively). Similarly, Equations (3) and (4) represent the generator polynomials for the K=3 encoder shown in Figure 3 (coefficients 5₈ and 7₈ respectively).

$$y_i = h^6 + h^5 + h^4 + h^3 + 1 \quad (1)$$

$$y_{i+1} = h^6 + h^4 + h^3 + h + 1 \quad (2)$$

$$y_i = h^2 + 1 \quad (3)$$

$$y_{i+1} = h^2 + h + 1 \quad (4)$$

Since the structure of the encoder is based on a shift-register, it can be conceptualized as a finite state machine (FSM) with 2^{K-1} states. Each state has two possible departures (based on the incoming source bit) to make the transition from the current state to the next state. Similarly, there are two possible previous-states which will

^a The taps are chosen such that they form a polynomial which, if used in a linear feed-back shift register configuration, would produce the longest possible pseudo-random sequence. In this way, the encoder can ensure optimum source-bit spread without introducing redundancy or catastrophic behaviour.

ⁱ Tap coefficients are arranged such that the tap for h^0 corresponds to the oldest bit in the shift register (right-most bit in Figure 2 and Figure 3).

produce the current state. Associated with each transition are two symbols (one frame) of encoded data. Table I lists all possible states, their allowed transitions and the corresponding encoded data for the 64-state, $K=7$ encoder shown in Figure 2. Table II illustrates the same behaviour, but uses the four-state, $K=3$ encoder shown in Figure 3. Although Table I is verbose, it clearly illustrates the increased complexity for higher constraint-length codes. Further discussion on the encoder state diagrams and the nature of "good" convolutional codes can be found in [3].

If Table II were presented graphically, and twice replicated in time, then it would produce the two-stage, four-state, "trellis" diagram¹ shown in Figure 4. Note that there are two paths (based on a binary data source) arriving at and departing from each state/node in the trellis. Associated with each departure is an encoder input bit. The upper branch departing from each state/node represents an encoder input of '0'. The lower branch departing from each state/node represents an encoder input of '1'. The two branches arriving at any state/node always represent the same encoder input bit. Odd numbered states/nodes can only be arrived at through an encoder input of '1'. Even numbered states/nodes can only be arrived at through an encoder input of '0'. Associated with each transition within the trellis is a frame of transmitted symbol bits as indicated in Figure 4, and in Table II.

Prior knowledge of the encoder structure is essential when designing processors to decode convolutional codes.

¹ Due to space limitations, and for the purposes of clarity, it is not possible to show two stages of a 64-state $K=7$ trellis.

Table Ia. The State Transitions for a K=7 Encoder

CURRENT STATE		SOURCE DATA	ENCODED DATA	NEXT STATE	
DECIMAL	BINARY			BINARY	DECIMAL
0	000000	0	00	000000	0
		1	11	000001	1
1	000001	0	10	000010	2
		1	01	000011	3
2	000010	0	11	000100	4
		1	00	000101	5
3	000011	0	01	000110	6
		1	10	000111	7
4	000100	0	11	001000	8
		1	00	001001	9
5	000101	0	01	001010	10
		1	10	001011	11
6	000110	0	00	001100	12
		1	11	001101	13
7	000111	0	10	001110	14
		1	01	001111	15
8	001000	0	00	010000	16
		1	11	010001	17
9	001001	0	10	010010	18
		1	01	010011	19
10	001010	0	11	010100	20
		1	00	010101	21
11	001011	0	01	010110	22
		1	10	010111	23
12	001100	0	11	011000	24
		1	00	011001	25
13	001101	0	01	011010	26
		1	10	011011	27
14	001110	0	00	011100	28
		1	11	011101	29
15	001111	0	10	011110	30
		1	01	011111	31

Table Ib. The State Transitions for a K=7 Encoder (continued)

CURRENT STATE		SOURCE DATA	ENCODED DATA	NEXT STATE	
DECIMAL	BINARY			BINARY	DECIMAL
16	010000	0	01	100000	32
		1	10	100001	33
17	010001	0	11	100010	34
		1	00	100011	35
18	010010	0	10	100100	36
		1	01	100101	37
19	010011	0	00	100110	38
		1	11	100111	39
20	010100	0	10	101000	40
		1	01	101001	41
21	010101	0	00	101010	42
		1	11	101011	43
22	010110	0	01	101100	44
		1	10	101101	45
23	010111	0	11	101110	46
		1	00	101111	47
24	011000	0	01	110000	48
		1	10	110001	49
25	011001	0	11	110010	50
		1	00	110011	51
26	011010	0	10	110100	52
		1	01	110101	53
27	011011	0	00	110110	54
		1	11	110111	55
28	011100	0	10	111000	56
		1	01	111001	57
29	011101	0	00	111010	58
		1	11	111011	59
30	011110	0	01	111100	60
		1	10	111101	61
31	011111	0	11	111110	62
		1	00	111111	63

Table 1c. The State Transitions for a K=7 Encoder (continued)

CURRENT STATE		SOURCE DATA	ENCODED DATA	NEXT STATE	
DECIMAL	BINARY			BINARY	DECIMAL
32	100000	0	11	000000	0
		1	00	000001	1
33	100001	0	01	000010	2
		1	10	000011	3
34	100010	0	00	000100	4
		1	11	000101	5
35	100011	0	10	000110	6
		1	01	000111	7
36	100100	0	00	001000	8
		1	11	001001	9
37	100101	0	10	001010	10
		1	01	001011	11
38	100110	0	11	001100	12
		1	00	001101	13
39	100111	0	01	001110	14
		1	10	001111	15
40	101000	0	11	010000	16
		1	00	010001	17
41	101001	0	01	010010	18
		1	10	010011	19
42	101010	0	00	010100	20
		1	11	010101	21
43	101011	0	10	010110	22
		1	01	010111	23
44	101100	0	00	011000	24
		1	11	011001	25
45	101101	0	10	011010	26
		1	01	011011	27
46	101110	0	11	011100	28
		1	00	011101	29
47	101111	0	01	011110	30
		1	10	011111	31

Table Id. The State Transitions for a K=7 Encoder (continued)

CURRENT STATE		SOURCE DATA	ENCODED DATA	NEXT STATE	
DECIMAL	BINARY			BINARY	DECIMAL
48	110000	0	10	100000	32
		1	01	100001	33
49	110001	0	00	100010	34
		1	11	100011	35
50	110010	0	01	100100	36
		1	10	100101	37
51	110011	0	11	100110	38
		1	00	100111	39
52	110100	0	01	101000	40
		1	10	101001	41
53	110101	0	11	101010	42
		1	00	101011	43
54	110110	0	10	101100	44
		1	01	101101	45
55	110111	0	00	101110	46
		1	11	101111	47
56	111000	0	10	110000	48
		1	01	110001	49
57	111001	0	00	110010	50
		1	11	110011	51
58	111010	0	01	110100	52
		1	10	110101	53
59	111011	0	11	110110	54
		1	00	110111	55
60	111100	0	01	111000	56
		1	10	111001	57
61	111101	0	11	111010	58
		1	00	111011	59
62	111110	0	10	111100	60
		1	01	111101	61
63	111111	0	00	111110	62
		1	11	111111	63

Table II. The State Transitions for a K=3 Encoder

CURRENT STATE		SOURCE DATA	ENCODED DATA	NEXT STATE	
DECIMAL	BINARY			BINARY	DECIMAL
0	00	0	00	00	0
		1	11	01	1
1	01	0	01	10	2
		1	10	11	3
2	10	0	11	00	0
		1	00	01	1
3	11	0	10	10	2
		1	01	11	3

Figure 4. Two Stages of a Four-State Trellis Diagram

2.5 Decoding Convolutional Codes

This section provides a brief overview of the decoding of convolutional codes, with specific reference to the Viterbi Algorithm.

Convolutional encoding has the effect of spreading the energy of a single bit over several transmitted symbols. If a distortion occurs in one of the symbols, then the original bit can be recovered by using the other related symbols (assuming that the noise on multiple symbols is uncorrelated). This is the basis of convolutional code decoding.

2.5.1 The Viterbi Algorithm

The Viterbi algorithm has traditionally been the most effective method for decoding convolutional codes. As stated by Forney [5],

".. the VA may be viewed as a solution to the problem of maximum *a posteriori* probability (MAP) estimation of the state sequence of a finite-state discrete-time Markov process observed in memoryless noise."

For all practical purposes, the task of the VA is to find the convolutional-code trellis path, traced by the state sequence x , whose "length" is the shortest, given a specific received sequence z . This section describes^k the algorithmic flow and theory behind the Viterbi Algorithm.

^k For this analysis, only a finite number of transitions will be considered. The transitions represent an arbitrary code block starting at state 00, ending at state 00, and extending from time 0 to time T . The application of the VA, however, is easily extended to include infinite length sequences.

Definitions

First, a summary of the symbols and terminology of the VA [5] (Figure 1 through Figure 4 will aid in understanding the physical meaning of the various symbols).

- 1) The state, x_t , at time t , is one of a finite number (M) of states, where $M = m^r$ ($= 2^{K-1}$ for a binary system).
- 2) x is the sequence of states, represented by the vector $(x_0, \dots, x_T)$.
- 3) The transition from state x_t to state x_{t+1} is denoted by ξ_{t+1} .
- 4) The probability of the transition ξ_{t+1} is denoted by $P(x_{t+1}/x_t)$.
- 5) Ξ is the set of transitions ξ_{t+1} for which $P(x_{t+1}/x_t) \neq 0$ (ie : the allowable trellis transitions). If the data source can be considered random, then, within Ξ , all transitions are equally probable.
- 6) $|\Xi|$ is the number of transitions in Ξ (equal to m^{r+1}).
- 7) The vector u is the sequence of encoder input values (ie: the data source).
- 8) The vector y is the sequence of encoded symbols input to the channel (for every element in y , there are $r=k/n$ elements in u). This is equivalent to the encoder output if the interleaver/deinterleaver, modulator/demodulator, transmitter/receiver and the channel (see Figure 1) are conceptualized as one lumped channel.
- 9) The vector z is the sequence of symbols observed at the output of the memoryless channel (ie: the decoder input).
- 10) The term "metric" is used to describe a quality figure associated with the "length" of a path through a convolutional-code trellis. This "metric" will be used, through comparison, to determine the shortest path(s) through the trellis. The decoder will make

decisions based on this metric, which represents the linear¹ distance between the received symbols and the symbols which lie along the path in question.

Probabilities

The convolutionally coded communications system represents a v th-order m -ary Markov process (corresponding to a shift-register of length v). Because this is a Markov process observed in memoryless noise, the probability of receiving the sequence z , given the encoder state sequence x , is given by Equation (5).

$$P(z|x) = \prod_{i=0}^{T-1} P(z_i | \xi_i) \quad (5)$$

The probability of state sequence x and received sequence z is given by Equation (6).

$$P(x,z) = P(x)P(z|x) = \prod_{i=0}^{T-1} P(\xi_i) \prod_{i=0}^{T-1} P(z_i | \xi_i) \quad (6)$$

Path Metrics

Each transition (branch) in the convolutional code trellis can then be assigned a "length" according to (7).

$$\lambda(\xi_i) = -\ln[P(x_{i+1}|x_i)] - \ln[P(z_i|\xi_i)] \quad (7)$$

Then, complete path lengths can be calculated according to (8). Due to the nature of the log function, it can be seen that the highest probability path will have the shortest "length".

¹ The recommended method of metric calculation would involve a quadratic calculation (ie: $[\]^2$). However, to ease digital implementation, a linear approach is used. The consequences of this action will be discussed later.

$$-\ln[P(x,t)] = \sum_{i=0}^{T-1} \lambda(\xi_i) \quad (8)$$

At any time t , there will be several paths leading up to state x_t . These path lengths are calculated according to (9).

$$\lambda(x_t) = \sum_{i=0}^{t-1} \lambda(\xi_i) \quad (9)$$

The shortest of these paths is called the survivor for node x_t and is denoted as $X(x_t)$. At any time t , there will be M survivors. The path lengths of these survivors are given in (10).

$$\Gamma(x_t) = \lambda[X(x_t)] \quad (10)$$

Dynamic Programming

During the next time interval $(t+1)$, the survivors from time t are extended, and new branch-metrics, path-metrics and survivor-metrics are calculated. This incremental updating behaviour demonstrates that the VA is an example of dynamic programming. Hence, it is possible to implement the VA in a computer program by performing subsequent iterations on a set of consistent recursive equations. In algorithmic form, the VA can be "programmed" as follows :

Required storage locations :

t	time index
$X(x_t), 1 \leq x_t \leq M$	survivor terminating in x_t
$\Gamma(x_t), 1 \leq x_t \leq M$	survivor length

Initialization :

$$i = 0$$

$$X(x_0) = x_0$$

$$X(m) = \text{arbitrary}, m \neq x_0$$

$$\Gamma(x_0) = 0$$

$$\Gamma(m) = \infty, m \neq x_0$$

Recursive calculations :

$$\Gamma(x_{i+1}, x_i) = \Gamma(x_i) + \lambda[\xi_i = (x_{i+1}, x_i)],$$

$$\Gamma(x_{i+1}) = \min \Gamma(x_{i+1}, x_i),$$

Store $\Gamma(x_{i+1})$ with corresponding shortest path $X(x_{i+1})$, and

increment i .

2.5.2 Simplifying the Viterbi Algorithm

For practical purposes, the concept of the VA can be greatly simplified. The following steps illustrate, albeit simplistically, the flow of the Viterbi Algorithm (refer to the left half of Figure 4 to aid in the visualization of this process).

- 1) For each branch (transition) in the trellis, the two encoder data symbols (which appear on the branch) are compared with the received "frame" of symbols during that time interval. The deviations, from the ideal path, are used as the "branch-metrics" for the trellis transitions. These "branch metrics" are used as a measure of error/quality associated with the individual transitions.
- 2) Each node/state in the trellis maintains an error/quality figure, called a "path metric", which reflects the total sum of all the

branch metrics, along a single "survivor" path, which lead to that state. At each node/state, the two branch metrics entering the node/state are added to the path metrics of their source nodes/states (from the previous stage in the trellis). The path with the worst metric (ie: possessing the largest deviation from the ideal path) will be discarded, and that state/node will be updated with the new metric. During this decision process, all possible path lengths must be evaluated (this implies a large storage requirement, and a high processing overhead).

- 3) When a decision is made to keep one path/branch over another, its corresponding direction bit (the encoder input bit which would have caused that transition) is stored in a first-in-first-out (FIFO) register (referred to as a "path memory").
- 4) At each time interval, all the metrics stored in the states/nodes will be compared. The FIFO output associated with the state/node possessing the best metric will be used as the decoder output data.

2.5.3 Implementation of the Viterbi Algorithm

The preceding two sections illustrated an ideal algorithmic decoding scenario. Certain realistic expectations, however, must be placed on the operation of the decoder if it is to manifest itself physically in a digital format. These restrictions are commonplace in most implementations of the VA. This section will discuss some of these considerations/modifications.

2.5.3.1 Path Memory Truncation

The VA requires that decisions on the survivor path be deferred until the end of each code block^m. In typical continuous-source applications, the code block could be semi-infinite, which would create a need for a semi-infinite path memory (and produce a semi-infinite decoder latency). Hence, the depth of the path memory must be truncated to some predetermined lengthⁿ. This truncation recognizes the following :

- a) paths which have not merged with the correct path will accumulate distance from it as the length of the unmerged span continues; and
- b) the best path(s) to each state have diverged from the correct path by only few constraint lengths.

Thus, decoder complexity can be reduced, and code efficiency can be increased, by eliminating the need for $(K-1)$ "0" insertions in the encoder, and also by truncating the depth of the path memory.

2.5.3.2 Metric Quantization

Due to a digital implementation of the VA, the various metrics cannot have infinite precision. Hence, branch metrics (representing the distance between the trellis transition data and the received symbol) must be mapped to quantized results for use in

^m Recall that, for this analysis, only a finite number of transitions will be considered. The transitions represent an arbitrary code block starting at state 00 and ending at state 00. The final state is artificially forced by inserting $K-1$ "0" values into the encoder following the end of the code block.

ⁿ Research by Heller and Jacobs [2] has demonstrated theoretically, and through simulation, that a value of 4 or 5 times the constraint-length is sufficient.

determining the optimum surviving path. Also, as a result of accumulating branch metrics into state metrics, restrictions must be placed on the maximum range of possible values^o.

2.5.4 Performance of the Viterbi Algorithm

Figure 5 and Figure 6 contain semi-log plots of the binary probability of error versus the ratio of energy per bit to noise spectral density (BER versus E_b/N_0). Both these figures show plots for uncoded and coded communication links using Viterbi decoding with constraint-lengths $K=3$, $K=5$ and $K=7$. The data for these plots have been extracted from the simulation findings of Heller and Jacobs [2] for the Viterbi Algorithm using both hard and soft-decision quantization and a 32-bit path memory.

Figure 5 displays plots for hard-decision decoding (one-bit quantization of the received symbols). Figure 6 displays plots for soft-decision decoding (eight-level, three-bit symbol quantization). Note that as the constraint-length increases (along with decoder complexity), the effective BER decreases.

Figure 7 uses an expanded scale to display information found in Figure 5 and Figure 6 for constraint-length $K=7$. Figure 8 uses an expanded scale to display the information found in Figure 5 and Figure 6 for constraint-length $K=3$. These figures clearly illustrate the enormous advantage of using multi-level quantization (soft-decision) within the decoder to take advantage of all possible information available at the receiver.

The performance of a convolutional-code decoder is usually expressed in terms of the coding gain. The coding gain, expressed in dB, is effectively a reduction in the required E_b/N_0 to achieve the same effective BER. Based on the presentation of the data,

^o As a result of the finite limitations imposed on the metric values, there must also be a method of normalizing these values to prevent under/overflow during the various calculations.

one can extract a coding gain for each of the theoretical implementations. For example, the hard-decision decoder represented by Figure 5 shows a coding gain of approximately 1.5, 2.4 and 3.3 dB for constraint-lengths $K=3$, $K=5$ and $K=7$ respectively using a 32-bit path-memory VA at a BER of 10^{-5} . Using the eight-level soft-decision decoding shown in Figure 6, the coding gain increases to approximately 3.6, 4.4 and 5.2 dB for constraint-lengths $K=3$, $K=5$ and $K=7$ respectively. These values are 0.1, 0.3, and 0.2 dB, respectively, short of the theoretical limits for the VA [3] (using an infinite number of quantization levels). For all practical purposes, these differences shall be considered inconsequential when compared to the required increase in decoder complexity necessary for implementing quantization levels greater than eight (eight-level quantization requires three binary bits per symbol, six binary bits per frame).

2.6 Summary

This chapter has provided fundamental information on convolutional coding and decoding. It has also introduced the concept of the Viterbi Algorithm and provided theoretical and published simulation results for coded and uncoded channels. This chapter will serve as the basis for evaluating the coding-gain performance of any Viterbi decoder architecture.

Figure 5. Viterbi Algorithm Performance for Hard-Decision Decoding

Figure 6. Viterbi Algorithm Performance for Soft-Decision Decoding

Figure 7. Viterbi Algorithm Performance for Constraint-Length $K=7$

Figure 8. Viterbi Algorithm Performance for Constraint-Length K=3

CHAPTER 3 VLSI AND SYSTOLIC ARRAY APPLICATIONS

3.1 Introduction

This chapter introduces the concept of the systolic array architecture. It will be shown that systolic arrays have direct applications to VLSI design. Some of the current technology obstacles facing VLSI designs will be reviewed and systolic arrays will be analyzed as a potential solution to these problems.

This chapter also contains a history of previous work in the field of systolic-array applications to the Viterbi Algorithm. In addition, some existing non-systolic single-chip implementations of the Viterbi Algorithm will be discussed as a benchmark on which to compare performance.

All information presented in this chapter is the result of the work of others.

3.2 Existing VLSI Designs

This section contains examples of existing VLSI designs. The first subsection reviews existing designs for Viterbi decoders for the purpose of illustrating the current view of the technology. The second subsection reviews existing systolic array designs for the purpose of demonstrating VLSI technology potential (to be used as a datum in Chapter 7).

3.2.1 Viterbi Decoder Designs

This section contains a list of some of the currently available, non-systolic, single-chip, VLSI implementations^P of the Viterbi Algorithm. These designs are presented as an example of traditional approaches to IC design. The designs examined in this section are not directly related to the work presented in this document. They have been listed here only to provide a demonstration of the existing architectures and performance parameters.

All examples will have, as a minimum, the following capabilities :

- full duplex operation,
- rate 1/2 codes,
- constraint-length $K=7$ (codes 171₈ and 133₈),
- hard and soft (8-level) decision decoding,
- error monitoring, and
- automatic QPSK and BPSK modem synchronization.

Design Example A

One of the few currently available commercial products is manufactured by Qualcomm Incorporated [10]. This device has the following attributes:

- 5.2 db of coding gain at a BER of 10^{-5} ,
- 12 Mbps data rate (military temperature, 17 Mbps commercial),
- 5.0 Volt operation (+/- 10%),
- 1 Watt power consumption (at rated 12 MHz),
- 155-pin ceramic pin-grid array,

^P Note that for some of the examples listed in this section, the details of the internal architecture are not known.

- fabricated in (1987) 1.5 μm CMOS technology,
- CCITT V.35 scrambler and descrambler,
- erasure for punctured coding, and
- a cost of \$5000 per military part, \$3000 per commercial part,
- \$1333/\$975 in large quantities (US dollars, 1988 prices).

Design Example B

Although work by Cain and Kriete [9] (Harris Corporation's Government Systems Sector) is not a commercially available product, it does serve as a useful reference point for potential IC designs. This design has the following attributes :

- 5.0 db of coding gain at a BER of 10^{-5} ,
- 10 Mbps data rate (military temperature range),
- 3.3 Volt operation (+/- 10%),
- 200 milliwatt power consumption (at rated 10 MHz),
- 85-pin ceramic pin-grid array (45 pins for signals), and
- 330 gates per square millimetre (1983) technology⁹,
 - 20,120 gate equivalents,
 - 80,482 transistors,
 - die size 7.8 mm by 7.9 mm (307 mils by 311 mils).

Analysis of Designs A and B

The architecture of the Harris design [9] is well documented. It uses 12 internal processors to calculate the branch metrics. It also makes use of the following optimizations :

⁹ Based on the information given on the design by Orndorff *et al* [8] (362 gates per square millimetre), the Harris design must also be manufactured in a 2 μm process.

- lower power consumption through lower operating voltage, and
- four-bit path metrics (sacrificing 0.1 dB coding gain).

Despite the fact that it uses an older technology, its throughput is essentially equivalent to the Qualcomm design when the technologies and operational voltages are taken into consideration.

The price and performance of the Qualcomm offering would seem to indicate that it is a layout-intensive design which does not make use of regular structures. It has increased functionality over the Harris design; however, this can be expected when using a smaller geometry process to fabricate the IC. Speculation would indicate that it uses essentially the same architecture as the Harris device.

These designs demonstrate that, without architectural improvements, the advances made in VLSI technology do not necessarily translate into designs with a corresponding increase in performance.

3.2.2 Systolic Array Designs

A recent systolic-array design by Gharachorloo *et al* [18] is used as a reference process on which a VLSI technology analysis can be based (to be used later, in Chapter 7). This design has the following attributes :

- 1.2 μm technology,
- one million transistors,
- 256 processing elements,
- 25 MHz operation,
- less than 4 Watts power dissipation,
- 11.4 mm (449 mil) square die size, and

- better than 90% utilization.

Each processing element within the authors' systolic-array graphics engine (SAGE) design performs address, depth and colour calculations for an 8-bit/pixel three-dimensional polygon graphic rendering. For the purposes of demonstrating complexity, the following pseudo-code describes the calculations which are required by this design :

```

PROCEDURE SPANToPIXEL (xleft,xright,Z,deltaZ,RGB,deltaRGB)
  static int ZBuf [0..1023], RGBBuf [0..1023]
  FOR x := 0 to 1023 DO BEGIN
    IF xleft <= x <= xright THEN BEGIN
      IF Z <= ZBuf[x] THEN BEGIN
        ZBuf[x] := Z;
        RGBBuf[x] := RGB;
      END;
      Z := Z + deltaZ;
      RGB := RGB + deltaRGB;
    END;
  END;

```

This design is capable of a peak processing rate of 25,000 million pixels/sec. Four of these chips are required for a 1024x1024 graphics display. This design will serve as a datum for estimating the technology requirements of a VLSI implementation of the Viterbi Algorithm.

3.3 Systolic Array Architectures

This section defines and describes the nature of systolic array architectures and their benefits in some computation-intensive applications.

3.3.1 Systolic Array Definition

The word "systolic" comes from the word systole, which refers to a contraction. For this analogy, the contraction is that of the heart as blood is pumped through the system, which is likened to data being clocked through an array. The term "Systolic Array" encompasses many possible definitions. To standardize, the definitions accumulated by S.Y. Kung [19] will be used;

A systolic array is a computing network possessing the following attributes :

Synchrony : The data is rhythmically passed through the network,

Modularity and Regularity : modular processing elements with homogeneous interconnections which may be extended indefinitely,

Spatial and Temporal Locality : all communication is local, and there is at least one time-delay unit associated with each processing element, and

Pipelineability : calculations exhibit a linear speed-up with a linear increase in the number of processing elements.

Systolic arrays are well suited to VLSI due to their ease of design and reduced interconnection lengths. In most cases, only a single processing element need be designed, and then replicated with connections to only its nearest neighbours. Systolic arrays also have an advantage over conventional von Neumann processors in that they have the ability to perform pipelined calculations.

Provided that an algorithm may be reduced to a set of operations which are performed repetitively upon a single data entry, there is an advantage to using systolic array architectures. This is particularly useful in algorithms which are compute-bound (that is to say, throughput is restricted by the number of computations which must be performed upon each I/O), such as the VA.

3.3.2 Systolic Array Example

The following is a list of some of the numerous examples for systolic array applications :

- digital filters,
- matrix multiplication,
- matrix LU-decomposition,
- regular expression recognition,
- sorting queues, and
- priority queues.

As a typical representative of systolic array applications, this section will illustrate the example of a simple, (dual) one-dimensional, pattern-matching array as specified by Foster and Kung [20]. The application is to determine if there are any patterns in the input string (S) which match the reference pattern (P, which may contain "wild-card" letters).

The traditional method of performing this operation would be to slide the string past the reference pattern and compare every letter to determine if there is a match. This method, however, requires a global interconnect of elements to collate the comparison information for each letter.

A better approach is to slide^{*} both the string and the reference pattern past each other and accumulate comparison information locally. Figure 9 shows the architecture of such a pattern matching array.

Figure 9. Systolic Array Example of String Pattern Matching

The operation of the design is as follows :

- 1) The pattern to be matched (P) traverses the comparison-array (D) from left to right. Wild card letters (Ω) and an end-of-pattern signal (g) are provided to the decision-array (E) to facilitate fast decisions.
- 2) The string (S) traverses the comparison-array from right to left.
- 3) The comparison-result signal (ω) is passed from the comparison-array (D) to the corresponding element in the decision-array.
- 4) Decisions are accumulated locally in a temporary variable (TEMP), and results are calculated based on local decisions, incoming comparisons, end-of-pattern signals and incoming results.

^{*} Note that this requires a "re-timing" of the data such that subsequent letters in the input string do not skip any letters in the reference pattern.

5) The result (m) is a serial stream indicating pattern matches.

The comparison-array produces either a yes or no result (ω) for the comparison of P to S (this is its only function). Equation (11) through Equation (14) show the calculations which are performed by the decision-array.

$$g_{out} \sim g_{in} \quad (11)$$

$$\Omega_{out} \sim \Omega_{in} \quad (12)$$

$$m_{out} \sim m_{in} \overline{g_{in}} + TEMP \cdot g_{in} \quad (13)$$

$$TEMP \sim g_{in} + TEMP \cdot (\Omega_{in} + \omega_{in}) \quad (14)$$

This example makes use of two different types of processing elements to form two interconnected one-dimensional systolic arrays. It has an advantage over uniprocessor implementations in that it has parallelism and pipelining. In this way, pattern matching can be achieved at much faster rates.

3.4 CMOS VLSI Technology Considerations

This section provides a brief overview of VLSI technology hindrances. It contains the necessary information for understanding the limitations on the various decoder implementations.

3.4.1 CMOS VLSI Speed Considerations

One of the main problems associated with the CMOS technology is the presence of capacitive loads. Each transistor, in the output stage of a logic function, is essentially a current source. This current source drives a metal wire interconnect and the input transistors of the next logic stage. Both the interconnections and the transistors will have a capacitive load associated with them. The larger the capacitive load, the longer it will take to charge through the current source and, hence, the longer the delay in the logic path. An example of capacitive load¹, in a typical 4 μm process [21] using the Mead and Conway λ -design rules² [22] ($\lambda = 2 \mu\text{m}$), is as follows :

$$\begin{aligned} \text{poly} : C_p &= [(4\lambda * 4\lambda) + [\lambda + 2\lambda] * 2\lambda] * 0.6 \times 10^{-4} \text{ pF}/\mu\text{m}^2 \\ &= 0.0053 \text{ pF} \\ \text{gate} : C_g &= [2\lambda * 2\lambda] * 0.5 \times 10^{-4} \text{ pF}/\mu\text{m}^2 \\ &= 0.0080 \text{ pF} \\ \text{metal} : C_{\text{int}} &= [3\lambda * 100\lambda] * 0.3 \times 10^{-4} \text{ pF}/\mu\text{m}^2 \\ &= 0.0360 \text{ pF} \end{aligned}$$

Note how the metal interconnection capacitance dominates (73%) the total capacitance of the metal, polysilicon and gate capacitances. Since it is not possible to change the polysilicon or gate physical attributes without going to a completely new technology, then the only variable of note is the length of the metal interconnect. Clearly, if a way can be found to reduce the length of interconnect, then the capacitive loading could be

¹ Capacitance values in VLSI are heavily dependent on the process used, and are not necessarily linked to feature size. Oxide thickness is the predominant contributor and could vary significantly from one manufacturer to the next. The capacitive values used here are for a hypothetical process only.

² Note : The symbol λ , in this example, does not represent the same λ used for the VA path lengths.

reduced as well. For example, if the interconnect were reduced to one-tenth of the value in the preceding example, then the total load capacitance would be reduced to approximately one-third of its former value (corresponding to a three-fold increase in speed in the same technology).

3.4.1.1 Systolic Array Solution

One of the potential solutions to the interconnection length problem is the use of a systolic array architecture. The advantage of a systolic array is that all inter-processor communications are local in nature. This provides a substantial reduction in the length of metal interconnect, and hence, an increase in the potential speed of the circuit (provided that the processing element is properly laid out to accommodate its nearest neighbours).

3.4.2 CMOS VLSI Layout Considerations

Typically, VLSI designs are composed of functional blocks. Each functional block must have its components physically placed and then interconnected. The blocks themselves will then be placed on a silicon die and globally interconnected according to a preconceived "floor plan". Depending on the design, this could involve a great deal of effort directed towards the traditional "place-and-route" activity (at several levels of hierarchy), which may or may not produce an optimum layout solution.

In an attempt to avoid the manually intensive place-and-route task, a standard-cell methodology was adopted. This approach used an automatic place-and-route tool to

achieve automated IC layouts. The automation, however, produced a less than optimum layout⁴. The alternative to the standard-cell methodology was a full-custom design, in which all of the transistors were drawn by hand and manually interconnected to form the logic blocks. The full-custom approach, however, is usually time consuming, and comparatively expensive (used only in high volume applications).

3.4.2.1 Systolic Array Solution

An approach which would alleviate most of the burden of hand-design and manual routing is the use of a systolic array architecture. With a systolic array, only one functional block need be manually laid out. This block can then be replicated in depth to form abutments with their nearest neighbours, thus eliminating the majority of the need for global routing. This combines the convenience of standard-cell design with the efficiency of full-custom design. The systolic array approach is also efficient in terms of I/O due to the fact that external connections are located only on the periphery. This makes it easier to arrange for package pins or interconnections to other blocks/arrays.

With the reduction of interconnect wiring also comes a reduction in wasted (inactive) silicon area and, hence, an increase in the effective utilization of the die. The increased utilization also has the potential for a reduction in the silicon die size, which results in better yields, more die per wafer, and lower silicon cost.

⁴ Typical ratios of active silicon area to total core die size, for gate-array and standard-cell designs, range from 30 to 45% (for DLM processes) or 60 to 75% (for TLM processes). This approach uses a significant proportion of the silicon for interconnection. The wasted area could otherwise be used for additional functionality, or, for a reduction in the silicon die size.

3.5 Viterbi Decoding by Systolic Array

This section introduces the idea of decoding convolutional codes by means of systolic array architectures. Methods of mapping the VA into processing elements as well as recent research in the field will also be discussed.

3.5.1 Potential Mapping of the VA into Systolic Arrays

The VA is an example of a compute-bound algorithm which can be reduced to a set of consistent recursive equations. This makes it a good candidate for a systolic array implementation.

For each symbol-frame I/O, the VA must perform $2^{(K-1)}$ iterations [5] of the recursive equation shown in (15).

$$\Gamma(x_{t+1}, x_t) = \Gamma(x_t) + \lambda[\xi_t = (x_{t+1}, x_t)] \quad (15)$$

(new state-metric = old state-metric + branch metric)

Various processing element designs and architecture arrangements can be created to satisfy (15). Theoretically, given enough processors with sufficiently small delay cycle-times, it would be possible to build a VA which is I/O-bound, rather than the currently accepted axiom of compute-bound.

Example

Given a $r=1/2$, $K=7$ decoder operating at a 16 Mhz symbol-frame rate.

Required In order to not be considered compute-bound, 64 evaluations of equation (15) must be computed in

less than 62.5 ns. This could be accomplished by 64 processors operating at a 16 MHz rate. Alternatively, the same throughput could be achieved by 128 processors operating at 8 MHz (or 256 processors operating at 4 MHz, etc).

The detailed mapping of the VA, as it pertains to this thesis, will be addressed in Chapter 4.

3.5.2 Recent Work on Viterbi Decoding Using Systolic Arrays

This section provides the relevant background on recent research into Viterbi decoding using systolic arrays.

3.5.2.1 Research by Chang and Yao (1985)

Work by Chang and Yao [23] made use of two one-dimensional systolic arrays. One array calculated the branch-metrics, while the other array updated the path-metrics. The theory behind the design is a densely connected (compacted) trellis structure which allows a transition from each state to any of the $2^{(K-1)}$ states by compacting (K-1) stages of the convolutional code trellis into a single stage.

Figure 10 illustrates the concept of trellis compaction, using K=3 as an example. The top half of the diagram illustrates two stages (corresponding to K-1) of the K=3 trellis diagram. Any terminal state can be reached from any start state by traversing K-1 stages of a trellis. The bottom half of the diagram illustrates, conceptually, that the K-1

Figure 10. Demonstration of Trellis Compaction for $K=3$

stages may be compacted into one densely-connected single stage. This method eliminates the need for trellis-style interconnections between stages and, hence, more readily lends itself to a systolic-array implementation. Each processing element would then perform $K-1$ steps of the VA.

The advantage of this approach is its simplicity and a high utilization of processing elements. The disadvantage of this approach is that it is relatively slow, due to the complexity of the processing elements. Their approach also relies on global interconnection to block memories (as in a standard implementation of the VA), which would cause a reduction in speed due to lengthy interconnect and limited memory access times.

3.5.2.2 Research by Gulak and Shwedyk (1986)

Research by Gulak and Shwedyk [24] is not truly a systolic-array architecture, but it is relevant in its application to VLSI layout. Apart from a uniprocessor implementation, the authors analyze two types of architectures : shuffle-exchange networks and cube-connected cycles. Both are area-intensive due to the required length of interconnections. However, these are both array processors, and can be more closely considered as Single-Instruction, Multiple-Data (SIMD) machines. From this analysis, they propose a lower bound on an area-per-baud metric to be applied towards the design of VLSI Viterbi decoders. The work serves as a partial basis for the area/speed trade-offs examined in section 3.5.2.5.

3.5.2.3 Research by Provence and Gupta (1987-88)

Work by Provence and Gupta [25][26] uses a two-dimensional systolic array structure as the main engine to calculate branch-metrics. This approach is shown in Figure 11.

The systolic-array configuration, shown as the first columns in Figure 11, performs the function of branch-metric calculation. The final column updates the new state-metrics by adding the branch-metric to the old state-metric. All possible symbol-pairs ($\sigma_{1,4}$) are stored in each column and then sequentially rotated in a vertical direction to generate the branch-metrics by way of successive multiply-and-add operations.

The author's design has the advantage that it is applicable to dynamic communications channels because the branch-metrics are recalculated every signalling interval. However, the design suffers from a relatively low utilization of the processing elements,

Figure 11. Two-Dimensional Design of Provence and Gupta

which can only be overcome by reducing the number of elements at the expense of throughput rate. There is also a requirement that the output of each column be globally routed to the top of each column creating lengthy and congested interconnections. The system design also requires a large cross-bar switch and associated tag memory address generator which further increases the number and length of global interconnections.

3.5.2.4 Research by Wen, Wang, Lee and Lin (1988)

Work by Wen, Wang, Lee and Lin [27] borrows from DSP theory and implements butterfly structures to process multiple states in one processing element. Their radix-2 butterflies will process states x_i and $x_{i+N/2}$ and output metrics for states x_{2i}

and x_{2k+1} . The DSP analogy is then extrapolated to produce butterflies of radix-4p (where $p > 1$). The butterfly approach is efficient in implementing Viterbi decoders with constraint-lengths greater than or equal to 8; however, the processing elements proposed by the authors are very complex and would limit the throughput of the design. Also, the design relies on a discrete random-access memory (RAM) to implement the path memory, requiring lengthy and complicated interconnections.

3.5.2.5 Research by Gulak and Kailath (1988)

Recent research by Gulak and Kailath [28] provides speed versus area comparison for various array implementations of the Viterbi Algorithm. The results vary from very fast, area-intensive, shuffle-exchange networks [24] to slower, smaller, uniprocessor implementations. Linear array and ring architectures are noted to be the most inefficient in terms of area/speed trade offs, whereas a multi-pass cascade approach was shown to be the most efficient solution.

Gulak and Kailath have introduced a VHSIC processing throughput per unit area metric (TP) defined in (16).

$$TP = (\text{gate-density}) * (\text{clock-rate}) = \frac{\text{gate-Hz}}{\text{mm}^2} \quad (16)$$

This metric is intended to gauge the complexity of a process in relation to existing and potential technologies. The example they give is a value of approximately 10^{10} for the Intel 80386 microprocessor (corresponding to approximately 250,000 gates, 4 million instructions per second and 100 mm² of area).

Applying the example value, of the TP metric (in the context of this investigation), to a Viterbi decoder which achieves a conservative baseband throughput of 6 MHz, would require a designer to use only 170,000 gates to complete the decoder. Alternatively, a 10 MHz Viterbi decoder, which contains a maximum of 550,000 gates (within the same silicon area), would produce a TP metric value of 5.5×10^{10} (5.5 times the complexity of an Intel 80386 microprocessor).

3.5.2.6 Research by Fettweis and Meyr (1990)

The most recent work, at the time of writing, in the field of systolic-array Viterbi decoders, is that of Fettweis and Meyr [29]. The authors propose the use of semiring^v algebra to transform the VA into a set of matrix operations which are better suited to VLSI architecture. This allows them to combine several (arbitrary N) stages of a trellis into one step which executes at a clock-speed which is 1/N that of the throughput. In addition to this mapping technique, the authors also make use of carry-save arithmetic and bit-level normalization in an attempt to minimize the amount of logic required to implement the design.

The sample design shown in [29] illustrates a four-chip set (25,000 gates and 15 RAMs per chip), implemented in 2 μ m standard-cell CMOS, which achieves a throughput rate of 50 Mbps. The authors also claim the potential to use 1 μ m full-custom dynamic CMOS logic to achieve a 1 Gbps decoder. Such a decoder would include 2.4 million

^v Semiring algebra is a method of applying commutative and associative laws, and the existence of neutral and inverse elements, to the operations of addition and the taking of a maximum (as applicable within the Viterbi Algorithm).

transistors in a 14.1 mm (one side) square die (resulting in a TP metric of 3×10^{12} ; 300 times the complexity of an Intel 80386 microprocessor).

3.6 Summary

This chapter has described the nature of the problems associated with VLSI design. As a result of this, the systolic array architecture has been introduced as a potential solution to the VLSI Viterbi Algorithm problem. Recent research into such an approach has also been covered, as have examples of single-chip VLSI implementations of the VA. This chapter will serve as a reference for the work performed in the next chapter (and the motivation behind this work).

CHAPTER 4 THE PROPOSED DECODER ARCHITECTURE

4.1 Introduction

The previous chapter has shown that the Viterbi Algorithm is a good candidate for a systolic-array application. The use of many parallel processors (each implementing the same function) can be used to advantage, by the VA. By arranging the processors in a systolic-array format, the effect of many VLSI technological hindrances can be reduced.

This chapter contains an account of the initial idea underlying a proposed decoder architecture. Its intent is to draw on the background material presented in the previous chapters and provide a high-level understanding of the rationale behind the decoder structure.

This chapter will only discuss the nature of the design at an architectural and algorithmic level. The design specification details (both physical and behavioural) for the various systolic arrays will be discussed in Chapter 5.

All material presented in this chapter, with the exception of comparisons to the work of others, is unique to the author.

4.2 The Systolic-Array Decoder Overview

This section provides an overview of the design of the systolic-array decoder. Its intent is to provide a frame of reference in which the operation of the various components may be understood. Subsequent sections will describe, in detail, the theoretical operation of the individual components.

Figure 12 displays the integration of one stage (of a multi-stage) convolutional code decoder, and is used to illustrate the interaction between the various components.

Each decoder stage consists of four key components :

- 1) The Correlation-Array,
- 2) The Comparison-Array,
- 3) The Metric/Latch-Array, and
- 4) The Trellis Direction Counter.

Partial path-metrics are produced by the output of each row of the correlation-array. These metrics represent path-lengths (over $K-1$ intervals of a convolutional code trellis) for paths which start from a predetermined starting state, but end in sequentially increasing terminal states. This sequentially incrementing behaviour is due to the use of a binary up-counter for generating trellis-direction "instructions".

Results, output from subsequent rows of the correlation-array, must be compared to determine which path, of the $2^{(K-1)}$ possible paths with the same terminal state, possesses the best path-metric (ie: the shortest path-length). This comparison is performed by a second systolic array, named the "comparison-array". Comparisons, within the comparison-array, are pipelined downwards along the outputs of the correlation-array rows, maintaining the best path-metric through comparisons on subsequent row outputs. The comparison-array will also maintain the starting state associated with each (best) path.

The comparison operation, however, must be preempted by the addition of the previous "state-metric" to the path-metric. The state-metrics are provided by the "best metric" output of the previous decoder stage for each terminal state and are maintained in the metric/latch-array to coincide with the row possessing the same starting state. In this way, the comparison-array performs the standard add-compare-select (ACS) function and provides the best metrics and starting states to the next decoder stage.

Figure 12. The Systolic-Array Decoder Architecture (Single Stage)

Because the first decoder stage has no previous stage from which to obtain state-metrics, it uses its own best-metric out as the state-metric input. Hence, the adders in the ACS circuit of the comparison-array would quickly overflow if some method of normalization were not used. For this purpose, the metric/latch-array is equipped with a subtracter to normalize the state-metrics before the comparison-array performs the addition. At the output of the comparison-array, there exists a normalization detector which analyzes the most significant bits of the best-metrics to determine if overflow is imminent. Such a condition raises a flag to instruct the subtractors to deduct a fixed number, in turn, from all the state-metrics.

Detailed design specifications for the various systolic arrays will be discussed in Chapter 5. Development and theoretical operation, of the correlation-array only, is described in the next section.

4.3 The Correlation-Array Overview

This section describes the structure and operational theory behind the main systolic-array compute-engine which will drive a convolutional-code decoder design.

4.3.1 Correlation-Array Background

The design of the main engine is based, to some degree, on work by Chang and Yao [23] and work by Provence and Gupta [25][26].

Work by Chang and Yao proposes that multiple stages of the convolutional-code trellis can be compacted into one densely-connected trellis (see Figure 10). In this way,

it is possible to reach any state in the trellis from any start-state by traversing the compacted trellis only once. This densely-connected trellis approach eliminates the need for trellis-style interconnections between processing elements. However, their approach requires very complex processing elements, and involves global routing from the last processing element in the array to all the other processing elements, as well as a path memory.

Work by Provence and Gupta proposes a two-dimensional array (see Figure 11) which shifts data vertically, and calculated results horizontally, to provide dynamically updatable branch-metrics to a modified (and separate) Viterbi decoder architecture. This two-dimensional approach allows for maximum parallelism with no wasted clock cycles. However, the utilization of their processing elements is poor, and their design requires global routing from the last processing element in each column to the top processing element in the same column. In addition, their design requires the use of multiplication, and requires a large number of clock cycles to perform the calculations for each signalling interval.

Both the designs, of Chang and Yao and that of Provence and Gupta, are not fully systolic in that they require global routing and a path-memory block. A combined approach, using a two-dimensional representation of a densely-connected trellis, has been incorporated in this thesis. This approach is illustrated in Figure 13. The top half of the diagram illustrates all possible transitions for $K-1$ stages of a $K=3$ trellis. The information is presented in $K-1$ sets so that it may provide a visual aid to the mapping. The bottom half of the diagram shows several $(K-1)$ by $2^{(K-1)}$ arrays. Within each row of processing elements are the $K-1$ encoder input bits which correspond to the transitions shown in the upper half of the diagram. Upon examining the pattern of bits within successive arrays, it becomes obvious that it is directly mappable to a single systolic-array (all the arrays shown are actually the same systolic-array, but they are presented at $K-1$

successive time intervals). Trellis direction "instructions" (counter bits) are injected into the array from the top. Trellis transitions are then readily produced, sequentially, from left to right.

This approach results in a design which has, as its main compute-engine, a two-dimensional systolic-array with a width equal to the number of memory elements in the encoder (6 elements for a $K=7$ code), and a height equal to the number of states in the encoder (64 states for a $K=7$ code). In this configuration, all calculations will be pipelined across its width and all data will be pipelined down its length. The development and operation of such an array will be discussed in the following subsections. Detailed design of the correlation-array will be discussed in Chapter 5.

Figure 13. Two-Dimensional Compacted Trellis Mapping of the VA

4.3.2 Correlation-Array Idea Development

This section provides an account of the development^w of the systolic-array decoder (SYSDEC) structure. The complete design specification for all the systolic-arrays, and their processing elements, will be discussed in Chapter 5.

4.3.2.1 Algorithmic Calculation Requirements

The calculations performed in most implementations of the VA are as follows :

- 1) Calculation of the branch-metrics,
- 2) Addition of the branch-metrics to the state-metrics, and
- 3) Comparison between state-metrics to select the best path.

The performance of an add-compare-select (ACS) function, although rudimentary, has a relatively lengthy execution time. For this reason, and due to the fact that fewer of these operations are required, the ACS function was eliminated from the main engine. Instead, the main engine concentrates on calculating the branch-metrics and combining them into partial path-metrics (composite branch-metrics).

In a densely-connected trellis configuration, each partial path-metric will be the sum of the $(K-1)$ previous branch-metrics. The branch-metrics themselves have been simplified^x to facilitate easy implementation in digital design. For this reason, the branch metrics are calculated as the numerical difference between the received symbol frame and

^w It should be noted that all development listed in this section is unique to the author alone.

^x Work by Heller and Jacobs [2] has demonstrated that decoder performance is relatively insensitive to digital metric quantization. Loss of coding gain is on the order of 0.1 dB.

the expected^y symbol frame. The partial path-metric represents an inverse measure of quality associated with traversing (K-1) stages of a standard trellis diagram starting at a known state and terminating at a known state (the larger the metric, the poorer the quality).

As previously stated, the main engine of the SYSDEC design is a systolic two-dimensional "correlation-array", whose dimensions are (K-1) columns by $2^{(K-1)}$ rows. A single row of processing elements from a K=7 array is shown in Figure 14.

Each row in the correlation array is responsible for providing one partial path-metric every clock cycle. Hence, each row in the correlation-array is responsible for calculating (17)^z.

$$\text{path-metric}(c \text{ to } d) = PM_c^d = \sum_{j=c_c}^{c_c+(K-1)} |z_j - s[\text{funct}(c, d, j)]| \quad (17)$$

where : c, d starting and terminating states, respectively,
 j column number,
 $\text{funct}()$ is a function, based on the starting and terminating states and the relative position within a densely-connected trellis transition, which produces a three-bit symbol value,
 z quantized (three bits) value of the received symbol with additive white Gaussian noise, and

^y A sequence of expected symbol frames is calculated based upon the known starting and terminal states for the selected path.

^z Note that the calculated metric is calculated as a linear "distance" between the received symbols and the expected symbols. Ideally, a nonlinear relationship (ie: $[\]^2$) should be used, but this would complicate a digital implementation. The consequences of this approach will be discussed later.

s expanded (three bits) expected data.

To be able to predict the expected symbol frame, each processing element requires two items of information :

- 1) The state from which a trellis transition begins, and
- 2) The choice (upper or lower) of departure branch.

Figure 14. Single Row of a K=7 Correlation-Array

Since only the first element in each row is provided with its starting state, then each element must provide the successive element with a new starting state. Hence, each processing element must now provide four functions :

- 1) Next-state calculation,
- 2) Expected data calculation,
- 3) Branch-metric calculation, and
- 4) Partial path-metric incremental updating, as indicated in (18).

By arranging the processing elements in tandem (as shown in Figure 14), successive branch-metrics will be accumulated into partial path-metrics as the calculation steps from left to right.

$$\text{partial-metric}(a \text{ to } b) = PM_a^b = PM_a + |z - s[\text{funct}(a,b)]| \quad (18)$$

where :

- a incoming starting state,
- b trellis direction instruction,
- funct() is a function, based on the starting and terminating states of a densely-connected trellis transition, which produces a three-bit symbol value,
- z quantized (three bits) value of the received symbol with additive white Gaussian noise, and
- s expanded (three bits) expected data.

Note : c, d and j in (17) will predetermine a and b.

Each processing element must also be capable of storing the received data and trellis direction instructions so that they may be passed on to the next row of processing elements. In this way, it becomes possible to maintain synchronization between data and instructions throughout the array.

4.3.2.2 Inter-Processor Coordination

The previous section described only one of the $2^{(K-1)}$ rows of the correlation-array. This section describes the structure of the correlation-array as a whole.

Each row in the correlation-array will be producing one partial path-metric on every clock cycle. This path-metric will represent the sum of (K-1) branch-metrics for

a path starting at a known^{aa} state and terminating in a predetermined state. These partial path-metrics can then be compared to determine the best possible path to reach a particular state.

Since it would be impractical to perform a comparison on the output of all rows at once, each row is "instructed" to have a unique terminating state. By making the terminating states sequential, then, on sequential clock cycles, sequential rows will produce identical terminating states^{bb}. This sequential behaviour can be produced by using a binary up-counter to provide the trellis direction instructions to the correlation-array.

Since the calculations are pipelined in a horizontal fashion, retiming/delay elements must be placed between the trellis-direction instructions and the correlation-array column inputs to allow for proper synchronization. Each processing element will, on subsequent clock cycles, transmit the trellis-direction instruction to the adjacent processing element, in the next row down. The same data flow and retiming is true for the received data as well as the trellis direction instructions.

4.3.2.3 Operational Data Flow

Since each calculation is pipelined, each element can contribute to a different path metric on each cycle. For example : once the first element in the first row starts the calculation of the path metric from starting state 00 to terminal state 00, it may then start

^{aa} To simplify the architecture, the starting state for each row will be set to the row number (in the range 0 to $2^{(K-1)}-1$).

^{bb} Sequential behaviour on the correlation-array outputs readily lends credence to the use of another systolic-array to compare the outputs of the correlation-array.

the calculation of the path metric from starting state 00 to terminal state 01 while the second processor in the same row continues the first calculation, and the first processor in the second row starts the calculation of the path metric from starting state 01 to terminal state 00.

Each row is essentially performing a correlation of the received data against the expected data for each transition across a densely connected trellis. Hence, the main array is named the "correlation-array". Figure 15 illustrates this behaviour in three "snapshots" at three successive time intervals (on-chip clock cycles). Only two rows (0 and 1) of the 64-state correlation-array are shown; however, this is sufficient to illustrate the data-flow behaviour of the entire array. The following is a list of conventions which will aid in the interpretation of Figure 15.

- 1) Each box represents one processing element in the correlation-array. The element is designated E_{ij} , where i is the row number and j is the column number.
- 2) Trellis-direction bits and received symbol-frames are pipelined in a downwards direction, and partial path-metric calculations are pipelined from left to right.
- 3) The number in the upper right corner of each box represents all of the following :
 - a) the column number,
 - b) the bit position from the trellis-direction counter, and
 - c) the time-index of the received symbol-frame from the symbol shifting unit associated with this decoder stage (zero being the oldest received symbol-pair).
- 4) The first mnemonic within each box indicates which set of $(K-1)$ received symbol-frames are being used to calculate the current path-metric. The

mnemonic "m0" indicates that the path-metric calculation is using the zeroth set of (K-1) symbol-frames. "m1" indicates the use of frame-set number one.

- 5) The second mnemonic within each box indicates the state of the trellis direction counter from which the trellis-direction bit was extracted. "d0" means that the direction counter was in state 00 when the bit was extracted. Similarly, "d63" means that the counter was in state 63 when the direction bit was extracted.
- 6) The third mnemonic within each box indicates the path for which the path-metric is being calculated. The first number indicates the starting state (consistent with the row number). The second number indicates the terminal state (consistent with the state of the trellis direction counter; for example, "1>59" indicates that the path-metric being calculated is that for a transition from state 01 to state 59 (available in row 01 only).

Figure 15a shows the state of the correlation-array at time $t=0$. The processing commences with E_{00} , using an entirely new set of six received symbol frames ("m1"), by performing the first calculation of the path-metric for traversing a densely-connected trellis from state 00 to state 00. E_{01} is currently on the second cycle of calculations for the transition from state 00 to state 63, using the previous set of symbol frames ("m0"). E_{02} is currently on the third cycle of calculations for the transition from state 00 to state 62 (using "m0"), etc. . E_{10} through E_{15} are performing calculations for transitions from state 01 to states 63 down to 58, respectively, using frame set "m0".

Figure 15a. Correlation-Array Data-Flow Behaviour : Time $t = 0$

Figure 15b. Correlation-Array Data-Flow Behaviour : Time $t = 1$

Figure 15c. Correlation-Array Data-Flow Behaviour : Time $t = 2$

At time $t=1$ (see Figure 15b), E_{00} starts the calculation for the path-metric from state 00 to state 01. E_{01} continues calculating the path-metric for the transition from state 00 to state 00 (as begun by E_{00} on the previous cycle). The trellis-direction bit and the symbol-frame, from E_{00} on the previous cycle, have now been passed downwards to E_{10} which proceeds to start the calculation for the path-metric from state 01 to state 00. Note that the terminal state has been passed downwards from one row to the next, and, due to the nature of the trellis-direction counter retiming elements, it has also been passed one column to the right (as indicated by the highlighted processing elements in all three parts of Figure 15). E_{11} through E_{15} continue the path-metric calculations which were started by E_{10} on the preceding five cycles.

Figure 15c shows that, at time $t=2$, the calculation for the transition from state 00 to state 00 has now been transferred to E_{02} (as it accepts a new frame, from frame-set "m1").

Through examination of the contents of the processing element at the end of any row (E_{n5} in Figure 15a, Figure 15b and Figure 15c), it can be seen that the output of said row is a series of path-metrics corresponding to paths which start at a state equal to the row number, and end at sequentially increasing state numbers. For example, the last element in row 00 will produce path-metrics for paths starting at state 00, but ending at states 59, 60 then 61 for Figure 15a, Figure 15b and Figure 15c respectively. This sequential state behaviour will facilitate integration with the comparison-array (as described in section 4.2).

4.3.3 Correlation-Array Processor Utilization Analysis

Each processing element is active 100% of the time. However, since there are only two possible instructions (ie: take the upper, or the lower, departure from each state) which can be performed in each element, and the path-metric and current-state inputs to the first element are fixed, then the effective utilization of processors in the first column is only that given by (19). Subsequent columns will have utilizations given by (20). The average processor utilization will be that of equation (21).

$$\frac{2}{2^{(K-1)}} = \frac{2}{64} = 0.03125 \quad (19)$$

$$0.03125 \leq \frac{2^{\text{column_number}}}{2^{(K-1)}} \leq 1.00000 \quad (20)$$

$$\frac{\sum_{i=1}^{(K-1)} \frac{2^i}{2^{(K-1)}}}{K-1} = \frac{\frac{2^K - 2}{2^{(K-1)}}}{K-1} = \frac{\frac{126}{64}}{6} = 0.328125 \quad (21)$$

Note that, for a K=7 implementation, the utilization rates vary from approximately 3% (for the first column) to 100% (for the last column), with the average utilization being approximately 33%.

4.4 Required Deviations from the Viterbi Algorithm

The proposed decoder design cannot be regarded as a "true" implementation of the VA. It deviates from the prescribed algorithm [5] in a number of significant areas.

The required deviations are as follows :

- 1) The use of normalization is not part of the VA. It is necessary, in this case^{cc}, to prevent the fixed-width digital registers and adders from overflowing (causing large bursts of decoder errors).
- 2) The true VA will discard undesirable paths at each stage of the convolutional code trellis. The proposed design only discards paths at (K-1)-stage intervals.
- 3) The mapping of received symbols to quantized (three bits) values is not part of the true VA. The mapping is required to facilitate digital implementation. Work by Heller and Jacobs [2] has shown that decoder performance is relatively insensitive to digital metric quantization.

Note that the first exception to the VA is commonplace in many designs. It is the second exception to the VA which will potentially contribute a distinguished behaviour for the proposed design.

4.5 Comparison to Alternative Methods

This section compares the expected throughput of the proposed architecture to that of alternative approaches. The alternative approaches considered are uniprocessor implementations and those proposed by other researchers in the field of systolic-array Viterbi decoders.

^{cc} The requirement for normalizing the state-metrics is a requirement for any digital implementation of the VA, not just systolic arrays.

4.5.1 Comparison to Uniprocessor Implementation

Work by Crozier *et al* [7] has implemented the VA using a Motorola MC6800 microprocessor. Their results for a Viterbi detector, with an algorithm memory of two, indicate the need for 275 clock cycles per received symbol frame. Given that the complexity of the VA is of order $O((K-1)^2)$, then, projecting their implementation to a $K=7$ detector, there would be a requirement for approximately 2475 clock cycles per received symbol frame. Assuming that the results of Crozier *et al* are typical^{dd} of microprocessors in general, then, using a commercially available microprocessor (a 33 MHz Intel 80386, for example), the typical decoder throughput which could be expected is approximately 0.013 Mbps. This data rate is two to three orders of magnitude slower than the expected data rate of the proposed architecture^{ee}.

4.5.2 Comparison to Work by Chang and Yao (1985)

The system overview, as presented by Chang and Yao [23], contains a path memory which is connected to each of the $2^{(K-1)}$ processors in their path-metric updating array. Given that each processing element requires $2^{(K-1)}$ iterations to update the path metric for its assigned state, and assuming that the relative speed of their processing elements is comparable to that of the proposed decoder design, then the proposed design would have a throughput which is $(K-1)$ times greater than the design of Chang and Yao.

^{dd} The number of clock cycles required to implement simple microprocessor functions is usually consistent across all generic microprocessor manufacturers.

^{ee} An estimate of the throughput rate of the proposed decoder appears in Chapter 5. Theoretical estimates of the throughput rate(s), based on a technology analysis, are performed in Chapter 7.

This can be attributed to the fact that the proposed decoder design operates on sets of $(K-1)$ frames at any given time. This can be viewed as a reduction in the symbol interval by a factor of $(K-1)$.

4.5.3 Comparison to Work by Provence and Gupta (1988)

The number of computation cycles required by the Provence and Gupta approach [25][26] is of $O(a^L)$, where "a" is the size of the signalling alphabet, and $L=K-1$. In the case of hard-decision $r=1/2$ decoding, the value of "a" is set to 4 (each received symbol-pair contains two one-bit symbols. hence, the alphabet size is 4). For a $K=7$ decoder, this produces a requirement for $4^6 = 4096$ cycles per signalling interval. Given that their processing elements require one multiplication (of unspecified accuracy/bit-width), and one accumulation, on each clock cycle, then the throughput of the proposed architecture would be N^{K-1} that of the Provence and Gupta approach (where N is the number of bits in each transmitted symbol-frame).

4.5.4 Comparison to Work by Fettweis and Meyr (1990)

Both proposals stated in [29] are for a $K=3$ configuration only. Given that their complexity is of the order $O(N^2)$, then a $K=7$ design would be 16 times larger than the stated configuration, with a throughput that would be disproportionately slower (due to their use of carry-save logic which is only valid for limited bit-widths). Extrapolating the data, the required size of the Fettweis and Meyr design would be 7.7 million transistors (making it impossible to implement their 16-stage example). Since the complexity of

their design is linearly scaled with M , then the best that they could hope for would be an M -step with five stages only (as opposed to six in the SYSDEC design). However, speculating on the probable reduced throughput of [29], it would only be able to execute at the same speed as the SYSDEC design (in its existing configuration).

4.6 Summary

This chapter has described the rationale behind this investigation. It is a summary of the development of the main systolic engine to drive a convolutional-code decoder. This development is a result of many months of investigation and feasibility analysis, and thousands of hours of CPU simulation time.

This chapter has provided insight into the structure, efficiency and theoretical operation of the proposed decoder design (to be discussed in Chapter 5). Comparisons to other approaches have demonstrated the potential increase in throughput, and the relative ease of silicon layout of the design. The detailed implementation and performance of such a design will be analyzed in the following chapters.

CHAPTER 5 THE SYSDEC DESIGN

5.1 Introduction

This chapter describes, in detail, the design of the key components of the SYSDEC design proposed in Chapter 4, and used as a reference in Chapter 7. The specification uses a "top-down" approach to describe the basic building blocks of the system. For the purpose of illustration, a constraint-length $K=7$ decoder is used as the representative decoder for all explanations; however, this description is generic to any constraint-length decoder. Since the objective of this thesis is to explore the potential for systolic-array architectures only, the specification of the timing-control and data-capture/synchronization circuitry will be omitted. This specification describes the design of the systolic-array^{rr} based architectures only.

5.2 The Systolic-Array Decoder Detailed Design

This section describes, in detail, the top-level design of the systolic-array decoder architecture. It outlines the major blocks which will be described in subsequent sections.

^{rr} The design of the systolic-array elements is assumed to be the product of a custom layout process, however, the description contained in this appendix is at the standard-cell level for ease of explanation.

5.2.1 The Systolic-Array Decoder Interface

Figure 16 contains a single block which represents the external interface to the SYSDEC design. The positions of the signals, in relation to the block, are purely arbitrary and have no bearing on the physical implementation.

The decoder inputs are as follows :

SYMBOL FRAME A six-bit bus containing two three-bit symbols. Each symbol is an eight-level quantized value of the data from the receiver.

GLOBAL RESET A single reset signal to put the decoder into a known state. Note that this signal is not necessary if the user wishes instead to set *SYMBOL FRAME* to an all-zero state for several cycles.

FRAME CLOCK The clock used to register the data on *SYMBOL FRAME*. The frequency of this clock will be multiplied internal to the design to provide the faster clock for the processing elements.

The decoder outputs are as follows :

DECODED DATA A single serial stream of bits (synchronized to *FRAME CLOCK*) which represent the maximum-likelihood estimation of the source data.

Figure 16. The Systolic-Array Decoder Interface

5.2.2 The Systolic-Array Decoder Architecture

Figure 17 shows the internal architecture of the systolic-array decoder (SYSDEC). It contains a control-logic block, a data extraction unit, a symbol shifter, and various decoder stages.

The specific number of decoder stages is dependent on the constraint-length of the code. For example, to obtain a path length which is four or five times the constraint-length, there would be five or six stages for a $K=7$ decoder and at least seven stages for a $K=3$ decoder. The decoder stages themselves are the crucial element of the design. They contain three systolic arrays important to the investigation. The concept of these arrays has already been introduced in Chapter 4. The design details will be discussed in section 5.2.2.1.

Figure 17. The Systolic-Array Decoder Architecture

The control logic is used to synchronize the operation of the decoder, and to provide for the various clock rates and reset signals which will be needed.

The symbol shifter accepts data at the symbol frame rate, and block-shifts the data across the decoder stages from right to left. At the same time, the symbol shifter will re-time the symbols being sent to each correlation-array column of the individual decoder stages.

The right-most decoder stage provides a $(K-1)$ bit bus to the data extraction unit which then serially shifts this data out as the final decoded data.

5.2.2.1 The Decoder-Stage Architecture and Operation

Figure 18 shows a single stage of the multi-stage decoder architecture. Note, Figure 18 is identical to Figure 12. It has been repeated here for ease of reference.

With a few notable exceptions, the architecture of each decoder stage is identical. The only exception to this rule is the connections to the metric-latch array of the first stage. In general, each decoder stage consists of a $(K-1)$ -bit counter, $(K-2)$ delay/re-timing elements (ranging in delay from one to $(K-2)$ clock cycles), a $(K-1) \times 2^{(K-1)}$ -element correlation-array, a $2^{(K-1)}$ -element comparison-array and a $2^{(K-1)}$ -element metric-latch array. For the $K=7$ decoder depicted in Figure 18, this translates into one six-bit binary counter, five re-timing elements (linearly increasing in delay from one to five cycles), a $6 \times 64 = 384$ -element correlation-array, a 64-element comparison array and a 64-element metric-latch array.

Figure 18. The Systolic-Array Decoder Architecture (Single Stage)

The "best row" outputs of each stage are globally broadcast to the inputs of the next stage's Metric/Latch-array and latched under the control of the timing-control unit. The same is true for the resulting state metrics from each stage. All remaining data inputs are hard-wired to known values. All remaining control signals are provided by the timing-control unit.

The connections to the first stage, being the exception to the rule, are altered by connecting a normalization detector to the path metric outputs of the comparison array. The resulting normalization signal is used to instruct the metric-latch array to perform normalization on the state-metrics. This allows the "best-row" inputs to the first decoder stage to be hard-wired, and it uses its own path-metric outputs for its state-metric inputs.

Normalization of the state-metrics is accomplished by subtracting a fixed number from each of the state-metrics stored in the metric-latch array. The value of this number, as well as the maximum possible register value for a $K=7$ decoder, was determined by the following process :

- 1) The maximum (worst-case) branch-metric calculated within each correlation-array processing element is : $2 \times 7 = 14$. Hence the worst possible metric to leave each row is : $6 \times 14 = 84$. This means that the adders in the last stage will potentially be adding 84 to whatever state metric is passed to them.
- 2) The middle decoder stages will not select a path metric of 84 as being the best metric. Instead, the worst possible best-metric would be : $84 / 2 = 42$. Hence the worst metrics produced by the middle stages would be : $4 \times 42 = 168$.
- 3) Since there will be a delay of one block of frames before the need for normalization can be detected. There is a potential for the state metrics to accumulate an additional : $2 \times 42 = 84$. Hence the test for normalization

must have a threshold set greater than 84. To ease the binary implementation, this value should be increased to 127.

- 4) The total accumulation thus far is : $127 + 168 + 84 = 379$. Hence the maximum register value must be greater than 379. Again to ease the binary implementation, this value will be increased to 511.

For physical (binary) implementation, this establishes the following parameters :

- a) the comparison arrays will require 9-bit metric-capability,
- b) the normalization detection threshold and subtraction constant will be set to 128, and
- c) the partial path-metric registers in the correlation array will require 7 bit registers.

To summarize, the operation of the decoder is as follows. Each stage of the decoder performs a correlation of $(K-1)$ received data frames against each of the $2^{2(K-1)}$ legitimate state sequences over $(K-1)$ trellis intervals (corresponding to a total of $K-1$ input bits at the encoder). These correlations are then added to the path metrics, produced by the previous stage, and compared to create a subset of 2^{K-1} optimal path metrics. The outputs of each stage of the decoder will contain the starting point for the best possible path terminating at each of the 2^{K-1} states. At the end of the last stage it will be possible to extract the start-state for the best possible path⁸⁸. The $(K-1)$ binary bits of the start-state are, in order, the most likely representations of the $(K-1)$ encoder input bits.

At this point, another $(K-1)$ frames will be block-shifted across the stages starting from the last stage and progressing towards the first stage (oldest data). Note that each stage requires 2^{K-1} clock cycles to complete $(K-1)$ trellis depth's worth of correlation and

⁸⁸ According to the Viterbi Algorithm theory, all paths with acceptable path metrics will converge upon a common start point, given a sufficient depth through the trellis. This is generally true, provided that there is not an excessive number of localized errors in the received symbol sequence.

comparison. This reduces the baseband data throughput of the decoder to the value shown in (22), which, for an on-chip clock-rate of 64 MHz, becomes 6 MHz for a K=7 decoder (23), 16 MHz for a K=5 decoder or 32 MHz for a K=3 decoder.

$$DATA_{RATE}_{BASEBAND} = CLOCK_{RATE}_{ON-CHIP} * \frac{K-1}{2^{K-1}} \quad (22)$$

$$64 \text{ MHz} * \frac{6}{64} = 6 \text{ MHz} \quad (23)$$

5.3 The Correlation-Array Design

This section describes, in detail, the design and operation of the correlation-array. The correlation-array performs the main task of calculating path metrics for every possible path through a (K-1)-stage trellis.

5.3.1 The Correlation-Array Interface

Figure 19 shows a block diagram representing the external connections required by the correlation-array. The positions of the signals is indicative of the relative location or the correlation-array within a decoder stage.

2

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

Figure 19. The Correlation-Array Interface

The following is a list of the correlation-array inputs :

FRAME DATA SETS	A (K-1) x 6-bit wide bus containing (K-1) sets of frame symbols from the symbol shifter.
TRELLIS DIRECTION	A (K-1)-bit wide bus containing the (K-1) signals from the trellis direction counter (by way of the various delay elements).
LAST STATES	$2^{(K-1)}$ sets of (K-1)-bit lines which have been hardwired to the row number for each row in the correlation-array.
PATH METRICS	$2^{(K-1)}$ sets of 7-bit lines which have been hardwired to zero.

The following is a list of the correlation-array outputs :

PATH METRICS $2^{(K-1)}$ sets of 7-bit lines which represent the
best path metrics calculated by the array.

5.3.2 The Correlation-Array Architecture

Figure 20 shows the basic architecture of the correlation-array. It contains only two rows of a $2^{(K-1)} \times (K-1)$ -element matrix, but is sufficient to demonstrate the regularity of interconnection within the array.

Hardwired inputs enter the array from the left. Trellis direction bits and frame data enter the array from the top. All outputs are then sent to the comparison-array for set reduction. Each processing element (PE) within the array is connected to its nearest neighbours on all four sides. Calculations flow from left to right. Data and control signals flow from top to bottom.

5.3.3 The Correlation-Array Operation

The correlation-array uses $(K-1)$ processing elements in each of $2^{(K-1)}$ rows to produce one path metric, per row, every cycle. These path metrics represent the distance between the received multi-frame symbol data and the generated expected symbol frames which would occur when traversing a trellis starting from the state equal to the row number and ending in one of $2^{(K-1)}$ possible states (determined by the state of the trellis-direction counter).

Figure 20. The Correlation-Array Architecture

Each one of the trellis bits is skewed from its nearest neighbour, by means of an extra delay element per column placed between the trellis direction counter and the correlation-array. This synchronizes the bits such that as data is traversing across the array from left to right, each subsequent PE, in a serial calculation, takes the appropriate direction in the trellis for the partial path-metric it is calculating. The data at the end of each row will provide partial-path metrics for paths terminating in sequentially increasing state numbers.

Symbol data is passed downward through the array, unchanged. Similarly, trellis direction bits are propagated downwards through the array, unchanged. The only data that changes through the array are the path metrics and the sequence of next-states calculated by each PE (to be used only by that row's subsequent PE).

5.3.4 The Correlation-Array Processing Element Design

This section describes, in detail, the design of the processing element used within the correlation-array. This processing element is responsible for calculating next-states and branch-metrics and updating the partial-path metrics under calculation.

5.3.4.1 The Correlation-Array PE Interface

Figure 21 shows the interface to each processing element in the correlation-array. The location of each signal in relation to the processing element is indicative of the method used to connect each element to its nearest neighbours.

The following is a description of the inputs.

<i>LAST STATE</i>	This is the next state calculated by the previous processing element.
<i>LAST METRIC</i>	This is the accumulated path metric from the previous processing element.
<i>TRELLIS DIRECTION</i>	This bit is used to indicate which, of two possible departures, the processing element is to use when calculating <i>NEXT STATE</i> .
<i>RECEIVED SYMBOL "A"</i>	This is a three-bit quantized value which represents the first symbol in the received symbol-frame.
<i>RECEIVED SYMBOL "B"</i>	This is a three-bit quantized value which represents the second symbol in the received symbol-frame.

Figure 21. The Correlation-Array Processing Element Interface

CLOCK

This is the clocking signal used to latch the data on the outputs. It provides the synchronous behaviour for the processing element.

The following is a description of the outputs. Note that all outputs will be registered by the *CLOCK* signal.

NEXT STATE

This is the next-state calculated within the processing element using the *LAST STATE* and *TRELLIS DIRECTION* inputs.

PATH METRIC

This is the accumulated path metric calculated within the processing element using the *LAST METRIC*, *CURRENT STATE*, *RECEIVED SYMBOL "A"* and *RECEIVED SYMBOL "B"* inputs.

<i>TRELLIS DIRECTION</i>	This is a registered version of the <i>TRELLIS DIRECTION</i> input.
<i>RECEIVED SYMBOL "A"</i>	This is a registered version of the <i>RECEIVED SYMBOL "A"</i> input.
<i>RECEIVED SYMBOL "B"</i>	This is a registered version of the <i>RECEIVED SYMBOL "B"</i> input.

5.3.4.2 The Correlation-Array PE Architecture

Figure 22 shows the internal architecture of each correlation-array processing element. Each element consists of four major components. These are as follows :

Current-State Calculator	This block calculates the current state using <i>TRELLIS DIRECTION</i> and <i>LAST STATE</i> .
Symbol Generator	This block is equivalent to the convolutional-code encoder and is specific to the generating polynomials used. It uses <i>LAST STATE</i> and <i>TRELLIS DIRECTION</i> to calculate two symbols of expected data.
Branch Metric Generator	This block calculates a branch metric based on the difference between the symbols from the symbol generator and the received frame containing <i>RX SYMBOL "A"</i> and <i>RX SYMBOL "B"</i> . Note that the symbols from the symbol generator must be replicated (as in

	hard-decision receivers) for comparison to the two 3-bit values in the received frame.
Error Summation	This block adds the branch metric from the branch metric generator to <i>LAST METRIC</i> .
Output Registers	This block contains only register elements which are used to latch the outputs on the clock edge.

Figure 22. The Correlation-Array Processing Element Architecture

5.3.4.3 The Correlation-Array PE Operation

This section describes the operation of each processing element within the correlation-array.

Each of these processing elements contains a copy of the original convolutional-code encoder. This encoder is used to "predict" the symbols that should be present on the interface to the element, based on its current state and the choice of departure

branches. A branch-metric is then generated, based on the difference between the predicted symbols and the actual received symbols. This branch-metric is added to the incoming path-metric to produce a new path-metric.

Calculation of the current state is a simple matter of shifting the last state one position (multiply by 2) and adding the trellis direction (effectively inserting an encoder input bit).

5.4 The Comparison-Array Design

This section describes, in detail, the design and operation of the comparison-array. This array contains the add-compare-select (ACS) circuits which add state-metrics to path-metrics for comparison with best-metrics. This array also maintains the optimum starting state for each of the survivor paths' best-metrics stored in the array.

5.4.1 The Comparison-Array Interface

Figure 23 depicts the external connections required for the comparison-array. The positions of the signals is indicative of their connection within the decoder stage.

The inputs to the array are as follows :

BEST METRIC

This is a 9-bit hardwired bus which provides a default maximum path-metric to the first element in the comparison-array. For all purposes, this shall be set to 511.

BEST ROW

This is a six-bit hardwired bus which provides a default state to the first element in the comparison-array. For all purposes, this shall be set to 00.

PATH METRICS

$2^{(K-1)}$ sets of 7-bit buses containing the *PATH METRIC* outputs from each of the $2^{(K-1)}$ rows of the correlation-array.

PREVIOUS BEST ROWS

$2^{(K-1)}$ sets of (K-1)-bit buses containing the *BEST ROW* output from each of the $2^{(K-1)}$ elements in the metric-latch array.

Figure 23. The Comparison-Array Interface

PREVIOUS STATE METRICS $2^{(K-1)}$ sets of 9-bit buses containing the *STATE METRIC* outputs from the metric-latch array.

The outputs of the comparison-array are as follows :

BEST METRICS An eight-bit wide serial stream containing the best-metric for each of $2^{(K-1)}$ possible, sequential, termination states.

BEST ROWS A six-bit wide serial stream of best starting states (best-row) associated with *BEST METRIC*.

5.4.2 The Comparison-Array Architecture

Figure 24 shows the architecture of the comparison-array. The hardwired inputs to the array are inconsequential to its operation. They are there merely to provide the first PE with an adequate number of inputs to perform a comparison. The outputs of the correlation-array and the metric-latch array are the only inputs of consequence to the comparison-array. The outputs of the comparison-array are a sequence of best path-metrics with their associated start-states (best-rows).

Figure 24. The Comparison-Array Architecture

5.4.3 The Comparison-Array Operation

The purpose of the comparison-array is to reduce the outputs of the correlation-array into a subset of path-metric/best-row pairs which represent the $2^{(K-1)}$ survivor paths to get to the $2^{(K-1)}$ possible terminal states.

Each PE within the comparison-array adds a state-metric (from the metric-latch array) to a path-metric (from the correlation-array) and compares the result to that of the previous PE's comparison. The best-metric (and the associated best-row counterpart) is then passed to the next PE. In this way, in combination with the operation of the correlation-array, each subsequent PE in the comparison-array will be selecting the best possible path to get to one specific terminal state (as processing progresses down the array). Any one PE will be performing comparisons on sequentially increasing terminal-state numbers. Hence, the output of the array will be path-metrics (and best row counterparts) corresponding to sequentially increasing terminal-states.

5.4.4 The Comparison-Array Processing Element Design

This section describes, in detail, the design of the processing element used within the comparison-array. This element contains the add-compare-select (ACS) circuit and associated registers.

5.4.4.1 The Comparison-Array PE Interface

Figure 25 shows a block diagram representing the external connections required for each processing element within the comparison-array. The relative positions of the signals are indicative of the element's location within the array.

The inputs to the PE are as follows :

<i>BEST METRIC</i>	The best path-metric as provided by the previous processing element.
<i>BEST ROW</i>	The best-row associated with <i>BEST METRIC</i> .
<i>PATH METRIC</i>	The partial path-metric calculated by one row of the correlation-array.
<i>STATE METRIC</i>	The state-metric from the metric-latch array. The state, which is represented by this metric, must be the same as the row number.
<i>ROW</i>	The best start-state associated with <i>STATE METRIC</i> .
<i>CLOCK</i>	Since this is a sequential element, it requires a clock to latch all the inputs.

The outputs from the comparison-array processing element are as follows :

BEST METRIC	The best path-metric as decided by this processing element.
BEST ROW	The best-row starting-state associated with BEST METRIC .

Figure 25. The Comparison-Array Processing Element Interface

5.4.4.2 The Comparison-Array PE Architecture

Figure 26 illustrates the architecture of the processing element for each element in the comparison-array. It consists of an adder/comparator unit, two multiplexers and the output latches.

Figure 26. The Comparison-Array Processing Element Architecture

5.4.4.3 The Comparison-Array PE Operation

Partial path-metrics from the correlation-array are added to the state-metrics from the metric-latch array. The result is then compared to the output of the previous comparison-array processing element. The output of the comparator is used to control two multiplexers which allow a best-metric/best-row pair to be sent to the next processing element dependent on which comparator input has the best metric.

5.5 The Metric-Latch Array Design

This section describes, in detail, the design and operation of the metric-latch array. This array contains the state-metric registers and the normalization subtractors. It receives data input from the previous decoder stage, and provides start-state rows and normalized state-metrics to the comparison-array.

5.5.1 The Metric-Latch Array Interface

Figure 27 depicts the external connections required for the metric-latch array. The positions of the signals are indicative of the location within the decoder stage architecture.

The inputs to the array are as follows :

<i>PREVIOUS STAGE ROWS</i>	A (K-1)-bit bus containing serialized <i>BEST ROWS</i> from the previous decoder stage. In the case of the first decoder stage, this input will be hardwired to the row number for each row in the correlation-array.
<i>PREVIOUS STAGE METRICS</i>	A 9-bit bus containing serialized <i>BEST METRICS</i> outputs from the previous decoder stage.
<i>NORMALIZE</i>	A single-bit signal indicating that normalization is required on all the <i>STATE METRICS</i> .
<i>LATCH ENABLE</i>	A single-bit signal enabling the latching of the <i>STATE METRICS</i> .

The following is a list of the metric-latch array outputs :

<i>BEST ROWS</i>	$2^{(K-1)}$ sets of (K-1)-bit lines which represent the latched best-rows selected by the previous decoder stage.
<i>STATE METRICS</i>	$2^{(K-1)}$ sets of 9-bit buses containing the latched best-metrics selected by the previous decoder stage.

Figure 27. The Metric-Latch Array Interface

5.5.2 The Metric-Latch Array Architecture

Figure 28 shows the architecture of the one-dimensional metric-latch array. The best-row and best-metric busses from the previous decoder stage, and the normalization and latch-enable signals from the timing-controller are the only inputs to the array. The best-metric and best-row busses enter the broad-side of the array and are globally broadcast to every processing element within the array. The normalization and latch-enable signals propagate down the array as they are sequentially passed from one processing element to the next. The row and state-metric outputs of the array are passed to the individual rows of the comparison-array.

Figure 28. The Metric-Latch Array Architecture

5.5.3 The Metric-Latch Array Operation

The best-row and best-metric busses from the previous decoder stage are globally broadcast to each element in the metric-latch array. The latch-enable signal from the timing controller is used to capture the appropriate data from the incoming stream of best-row/best-metric pairs. The best-metric is then normalized (if required) under control of the normalization signal. The result is then passed to the comparison-array as the state-metric. The latched best-row of each processing element is transmitted directly to the corresponding row in the comparison-array as the start-state for the survivor path.

5.5.4 The Metric-Latch Array Processing Element Design

This section describes, in detail, the design of the processing element used within the metric-latch array. This element contains the registers and normalization circuitry used to provide the comparison-array with state-metrics

5.5.4.1 The Metric-Latch Array PE Interface

Figure 29 shows a block diagram representing the external connections required for each processing element within the metric-latch array.

Figure 29. The Metric-Latch Array Processing Element Interface

The positions of the signals are indicative of the location within the array. The inputs to the PE are as follows :

PREVIOUS STAGE ROWS

A (K-1)-bit bus containing serialized *BEST ROWS* from the previous decoder stage. In the case of the first decoder stage, this input will be hardwired to the element number (corresponding to the row number in the correlation-array).

<i>PREVIOUS STAGE METRICS</i>	A 9-bit bus containing serialized <i>BEST METRICS</i> from the previous decoder stage.
<i>NORMALIZE</i>	A single-bit signal indicating that normalization is required on the <i>PREVIOUS STAGE METRICS</i> input. This signal is the output from the previous processing element.
<i>LATCH</i>	A single-bit signal enabling the latching of the <i>PREVIOUS STAGE METRICS</i> input. This signal is the output from the previous processing element.
<i>CLOCK</i>	Since this is a synchronous element, it must be supplied with a clock. This signal latches the <i>NORMALIZE</i> and <i>LATCH</i> signals on the rising-edge of every cycle. This signal is also gated with <i>LATCH</i> to provide the clock for the data-capture registers.

The outputs from the metric-latch array processing element are as follows :

<i>BEST ROW</i>	A (K-1)-bit bus which represents the best-row selected by the previous decoder stage.
------------------------	---

<i>STATE METRIC</i>	A 9-bit bus containing the best-metric selected by the previous decoder stage.
<i>NORMALIZE</i>	This is a registered version of the input signal of the same name.
<i>LATCH ENABLE</i>	This is a registered version of the input signal of the same name.

5.5.4.2 The Metric-Latch Array PE Architecture

Figure 30 illustrates the architecture of the processing element for the metric-latch array. It contains a subtracter, two sets of latches and two AND-gates for enabling latching and normalization.

5.5.4.3 The Metric-Latch Array PE Operation

The latch-enable signal is used to enable latching, at the appropriate time, of the *PREVIOUS STAGE ROWS* and *PREVIOUS STAGE METRICS* inputs from the previous decoder stage. The normalization signal is used to enable the subtraction of a constant from the value of *PREVIOUS STAGE METRICS*, when normalization is requested. This value is then passed to the set of latches which are controlled by the latch-enable signal.

Figure 30. The Metric-Latch Array Processing Element Architecture

Both the normalization signal and the latch-enable signal are latched on every clock cycle, and subsequently transmitted to the next processing element.

5.6 Summary

This chapter has described the design of the systolic arrays which comprise the bulk of the SYSDEC design. This detailed specification will allow for behaviour models to be created, thus permitting VHDL and VLSI analysis (as discussed in the next two chapters).

CHAPTER 6 THE VERIFICATION ENVIRONMENT

6.1 Introduction

This chapter gives an introduction to the two simulation engines used in this investigation. The VHDL toolset was used to verify the integrity of a potential VLSI implementation of the decoder architecture. The communications link simulation package, LINKSIM, was used to provide coding-gain performance parameters for subsequent comparison to published results. An overview, of both of these tools, is presented in the following sections. Appendix A contains a history of the iterative design verification process which resulted in the final prototype of the SYSDEC design.

6.2 Digital Architecture Evaluation Environment

The tool used to evaluate the proposed architecture and its potential VLSI implementation was the VHSIC Hardware Design Language (VHDL).

The VHDL environment was selected because of its flexibility and standardization. Because it is flexible, it is ideally suited to systolic-array design where individual processing elements will have behavioural (or gate-level) models but must be instantiated as components to form arrays. Because it is an IEEE standard (1076), and since the US DoD requires it for its contractors, it is rapidly becoming a standard practice throughout the digital electronics industry to document (and potentially simulate) designs in VHDL.

Due to the nature and time constraints of this investigation, it will not be possible to perform a physical IC layout and fabrication. Hence, a method was needed to delve

into the design in sufficient detail such that there would be a high degree of confidence in the ability to transfer it to a silicon implementation. The use of the VHDL language makes it possible to perform a detailed identification of the timing/synchronization and all the required subcomponents for the SYSDEC design. This allows for a detailed parts list to be compiled for the purpose of evaluating the feasibility of a VLSI implementation.

6.2.1 Introduction to VHDL

VHDL is a high-level hardware design language which is intended for the modelling of digital circuits using levels of abstraction ranging from logic gates to complete systems. It was introduced by the United States Department of Defense (DoD) as a requirement for its Very High Speed Integrated Circuits (VHSIC) program. The VHDL language has the capability to make linkages between behavioural and component-level representations and, as such, can be technology-independent. This makes it ideal for simulating systems in which all the physical parameters are not yet known. In this way, VHDL can readily become a form of documentation, specification and/or synthesis for a system, design or component.

A full analysis of the VHDL environment is beyond the scope of this thesis. The interested reader can investigate further the many available sources of introduction [30], reference [31], instruction [32], and usage [33].

6.2.2 VHDL Tool Syntax

Access to an industrial VHDL tool set has been obtained for evaluation purposes only. The language syntax conforms to the IEEE VHDL 1076 standard [34]. Samples of commands and option files can be found in Appendix B. Samples of simulation output and generated reports can be found in Appendix C.

6.2.3 Mapping the Design into VHDL

The SYSDEC design was mapped into the VHDL environment through a "bottom-up" approach. The following steps illustrate the process :

- 1) Behavioural models were written to describe the operation of the individual processing elements.
- 2) Structural entities were written to assemble the processing elements into systolic arrays.
- 3) Structural entities were created to configure the arrays into a single decoder stage.
- 4) Behavioural models were written for the timing-controller, normalization-detector, symbol-shifter and data-extraction units.
- 5) A "test-bench" was created which integrated a number of decoder stages with the other discrete elements. This test-bench also contained the stimulus for performing the simulation.

In this way, changes could be made to the decoder performance by modifying only the behavioral models (at the lowest level of the hierarchy). Alternatively, the

interconnection and configuration of the architecture could be changed by altering the component instantiations (at the higher levels in the hierarchy).

Figure 31 illustrates the hierarchy for describing the SYSDEC design in VHDL. Note that all entities make use of the common package, SYSDEC_PACKAGE, which contains declarations of data types and constants as well as conversion and resolution functions. The source listings for the various VHDL entities and bodies of the SYSDEC design can be found in Appendix D.

6.3 Coding-Gain Evaluation Environment

The tool used to extract the coding-gain performance of the proposed architecture was a communications-link simulation package, dubbed LINKSIM. LINKSIM is a software package which was written, in FORTRAN, by Miller Communications Systems Ltd. under contract to the Communications Research Centre¹⁴. The LINKSIM package was selected because it already had in place a method of involving convolutional codes for obtaining bit error rates at user-specified signal-to-noise ratios for BPSK modulation.

This section describes the LINKSIM package and its use as applied to the SYSDEC design.

¹⁴ CRC contract number 36001-7-3579/01-ST, MCS file number 8832, SSC file number 04ST.36001-7-3579, dated April 28, 1988.

Figure 31. Hierarchy of the SYSDEC VHDL Implementation

6.3.1 Introduction to LINKSIM

LINKSIM is a communications-link simulation package intended for simulating mobile radio systems. It is capable of simulations using the following options :

- **various modulation techniques (2-CPSK used),**
- **trellis and convolutional coding using various constraint-lengths (rate 1/2, K=3 and K=7 convolutional codes used),**
- **Reed-Solomon and Shortened Reed-Solomon codes,**
- **concatenated and differential coding,**
- **interleaving/deinterleaving,**
- **user-specified pulse shaping,**
- **user-specified AM-AM/AM-PM nonlinearities,**
- **Rayleigh or Rician fading channels with user-specified spectral characteristics,**
- **arbitrary timing and frequency offsets,**
- **additive white Gaussian noise (used for E_b/N_0 of 3.5 to 8.0 dB),**
- **user-specified transmit and receive filtering,**
- **coherent or differential detection,**
- **Viterbi decoding of convolutional and trellis codes (used to verify the software installation, and benchmarking the package),**
- **error statistic gathering (used in 8-run groups per simulation), and**
- **optional spectra and eye diagram outputs.**

6.3.2 LINKSIM Usage

All simulation constraints are specified by the user in a parameter file (and optionally in pulse-shaping filter-files) prior to executing the LINKSIM program. Appendix E contains a sample listing of the LINKSIM parameter file used for simulating the SYSDEC design with constraint-length $K=3$, soft-decision decoding, using 2,000,000 bits per run (eight runs per simulation) at an E_b/N_0 of 6.5 dB. The parameters and the results of the LINKSIM execution are written to the simulation results file. Appendix F contains the results file for a simulation using the parameter file found in Appendix E. Note that each execution of the LINKSIM program causes eight separate simulation runs to be performed. A summary for each run is listed in a results file, along with the statistical average and standard deviation for the estimated BER of the system.

The LINKSIM source code consists of 85 FORTRAN modules, and has been adapted to run on a SUN workstation¹¹. Results of the LINKSIM package built-in convolutional coding/decoding are compared against the results of Heller and Jacobs [2] to provide a level of confidence in the software (shown in Figure 32 and Figure 33).

For the purposes of this investigation, the LINKSIM package was used to provide an additive white Gaussian noise BPSK channel with convolutional coding. Complete documentation of the LINKSIM package can be found in [35].

¹¹ The LINKSIM package was originally written to run on a VAX host, and has been modified by the author for adaptation to SUN workstations. The simulation results were verified against published results for Gaussian white-noise BPSK channels with, and without, convolutional coding and Viterbi decoding.

6.3.3 Integration with the LINKSIM Package

The LINKSIM package already had the capability to provide Viterbi decoding, hence, it was easily modified to accept alternate decoder algorithms. By modifying the "receiver.f" subroutine¹¹ to accept an alternate decoder/detector name (VITERBISYS in this case) from the user-specified parameter file, it was possible make calls to two new FORTRAN subroutines : "sysdecin.f" and "sysdec.f". The "sysdec.f" subroutine was written to emulate the functionality of the SYSDEC design, given a predetermined knowledge of the hardware implementation (from the VHDL analysis, and the detailed design in Chapter 5). The "sysdecin.f" routine was written to initialize the decoder internal states to known values before simulation commences. Listings of both "sysdecin.f" and "sysdec.f" can be found in Appendix G.

6.4 Summary

This chapter has introduced the two simulation tools used to verify the physical integrity and coding-gain performance of the architecture under investigation. Listings for the source-code (Appendix D and Appendix G) used with these tools , in concert with the design specification found in Chapter 5, will aid in the understanding of the decoder design and operation (as used in this investigation).

¹¹ The subroutine "receiver.f" is one of the 85 FORTRAN modules that constitute the LINKSIM package.

Figure 32. LINKSIM Benchmark Results for K=7

Figure 33. LINKSIM Benchmark Results for K=3

CHAPTER 7 VERIFICATION RESULTS

7.1 Introduction

This chapter contains the results of the SYSDEC feasibility analysis. It contains the following verification results :

- 1) Proof, through LINKSIM simulations, that the coding gain is adequate for effective system performance;
- 2) Proof, through VHDL simulations, of the completeness and integrity of the architecture; and
- 3) Proof, based on the VHDL analysis and a CMOS VLSI technology analysis, of the feasibility of VLSI implementation.

7.2 Results of LINKSIM Simulations

This section uses tables and graphs to present the results of the SYSDEC LINKSIM simulations for hard and soft-decision decoding of constraint-length $K=7$ and $K=3$ convolutional codes. In all cases, the data will be compared to uncoded channels, the theoretical limit of the Viterbi Algorithm [3], and the simulation results of Heller and Jacobs [2]. For this investigation, the BER for an uncoded, Gaussian noise channel was calculated using (24) [36]. Since this investigation is based on a BPSK ($m=2$) channel (at a specified E_b/N_0), then (24) can be reduced to (25).

$$P_e = \frac{1}{\text{LOG}_2 m} \cdot \text{Erfc} \sqrt{2 \frac{E}{\eta}} \quad (24)$$

$$P_e = \text{Erfc} \sqrt{2 * (10^{\frac{E/N_0}{10}})} \quad (25)$$

7.2.1 BER Results

The results for the simulations, performed with hard-decision decoding using a constraint-length of $K=7$, are shown in Table III. Table IV follows the same format as Table III, but it uses 8-level soft-decision decoding. Figure 34 shows a plot of the LINKSIM simulation results found in Table III and Table IV. For reference, Figure 34 also shows the BER for an uncoded Gaussian noise channel and the published computer simulation results of Heller and Jacobs [2] (as shown in Figure 7).

The results for the simulations, performed with hard-decision decoding using a constraint-length of $K=3$, are shown in Table V. Table VI follows the same format as Table V, but it uses 8-level soft-decision decoding. Figure 35 shows a plot of the LINKSIM simulation results found in Table V and Table VI. For reference, Figure 35 also shows the BER for an uncoded Gaussian noise channel and the published computer simulation results of Heller and Jacobs [2] (as shown in Figure 8).

The BER plots shown in Figure 34 and Figure 35 demonstrate a high degree of correlation between the results of the SYSDEC simulations¹² and those of Heller and Jacobs [2].

¹² Typical accuracies on the LINKSIM results include a standard deviation of up to 15% of the average value. Due to the lengthy simulation times, it was not possible to obtain a tighter tolerance at the lower bit error rates.

Table III. LINKSIM Simulation Results for Hard-Decision, K=7

Eb --- No	Viterbi Upper Bound	Uncoded BPSK Channel	Simulation Results of [2]	LINKSIM SYSDEC Results	
				BER	std. dev.
3.5 dB	1.3×10^{-4}	1.7×10^{-2}	--	1.4×10^{-3}	2.4×10^{-3}
4.0 dB	2.0×10^{-3}	1.3×10^{-2}	5.0×10^{-3}	4.4×10^{-3}	8.0×10^{-4}
4.5 dB	3.0×10^{-6}	8.8×10^{-3}	1.6×10^{-3}	1.8×10^{-3}	2.2×10^{-4}
5.0 dB	4.3×10^{-7}	6.0×10^{-3}	5.2×10^{-4}	5.3×10^{-4}	3.1×10^{-3}
5.5 dB	--	3.9×10^{-3}	1.3×10^{-4}	1.7×10^{-4}	2.1×10^{-3}
6.0 dB	--	2.4×10^{-3}	3.0×10^{-5}	3.8×10^{-5}	9.4×10^{-5}
6.5 dB	--	1.4×10^{-3}	--	9.6×10^{-6}	4.2×10^{-5}
7.0 dB	--	8.0×10^{-4}	--	--	--

Table IV. LINKSIM Simulation Results for Soft-Decision, K=7

Eb --- No	Viterbi Upper Bound	Uncoded BPSK Channel	Simulation Results of [2]	LINKSIM SYSDEC Results	
				BER	std. dev.
3.5 dB	1.3×10^{-4}	1.7×10^{-2}	2.0×10^{-4}	2.3×10^{-4}	2.3×10^{-3}
4.0 dB	2.0×10^{-3}	1.3×10^{-2}	3.0×10^{-3}	4.9×10^{-3}	1.2×10^{-3}
4.5 dB	3.0×10^{-6}	8.8×10^{-3}	5.5×10^{-6}	5.3×10^{-6}	3.6×10^{-6}
5.0 dB	4.3×10^{-7}	6.0×10^{-3}	--	--	--
5.5 dB	--	3.9×10^{-3}	--	--	--
6.0 dB	--	2.4×10^{-3}	--	--	--
6.5 dB	--	1.4×10^{-3}	--	--	--
7.0 dB	--	8.0×10^{-4}	--	--	--

Figure 34. LINKSIM Simulation Results for Constraint-Length $K=7$

Table V. LINKSIM Simulation Results for Hard-Decision, K=3

Eb --- No	Viterbi Upper Bound	Uncoded BPSK Channel	Simulation Results of [2]	LINKSIM SYSDEC Results	
				BER	std. dev.
4.5 dB	3.3×10^{-4}	8.8×10^{-3}	6.2×10^{-3}	5.7×10^{-3}	4.0×10^{-4}
5.0 dB	1.1×10^{-4}	6.0×10^{-3}	3.0×10^{-3}	2.9×10^{-3}	9.6×10^{-5}
5.5 dB	3.0×10^{-5}	3.9×10^{-3}	1.3×10^{-3}	1.4×10^{-3}	1.3×10^{-4}
6.0 dB	7.3×10^{-6}	2.4×10^{-3}	6.8×10^{-4}	5.7×10^{-4}	6.8×10^{-5}
6.5 dB	1.6×10^{-6}	1.4×10^{-3}	2.5×10^{-4}	2.6×10^{-4}	1.5×10^{-5}
7.0 dB	4.4×10^{-7}	8.0×10^{-4}	1.0×10^{-4}	9.0×10^{-5}	6.4×10^{-6}
7.5 dB	--	4.0×10^{-4}	3.3×10^{-5}	3.4×10^{-5}	2.1×10^{-6}
8.0 dB	--	1.9×10^{-4}	--	9.6×10^{-6}	9.0×10^{-7}

Table VI. LINKSIM Simulation Results for Soft-Decision, K=3

Eb --- No	Viterbi Upper Bound	Uncoded BPSK Channel	Simulation Results of [2]	LINKSIM SYSDEC Results	
				BER	std. dev.
4.5 dB	3.3×10^{-4}	8.8×10^{-3}	3.3×10^{-4}	3.5×10^{-4}	1.6×10^{-5}
5.0 dB	1.1×10^{-4}	6.0×10^{-3}	1.3×10^{-4}	1.3×10^{-4}	9.4×10^{-6}
5.5 dB	3.0×10^{-5}	3.9×10^{-3}	3.8×10^{-5}	3.5×10^{-5}	3.5×10^{-6}
6.0 dB	7.3×10^{-6}	2.4×10^{-3}	1.1×10^{-5}	1.1×10^{-5}	3.8×10^{-7}
6.5 dB	1.6×10^{-6}	1.4×10^{-3}	--	3.0×10^{-6}	1.3×10^{-7}
7.0 dB	4.4×10^{-7}	8.0×10^{-4}	--	--	--
7.5 dB	--	4.0×10^{-4}	--	--	--
8.0 dB	--	1.9×10^{-4}	--	--	--

Figure 35. LINKSIM Simulation Results for Constraint-Length $K=3$

7.2.2 Coding Gain Results

Table VII and Table VIII display the hard- and soft-decision, respectively, coding gain available from the SYSDEC design based on data extractions from Figure 34 (constraint-length $K=7$). It also contains the corresponding gains obtained from the published simulation results of Heller and Jacobs.

Table IX and Table X display the hard- and soft-decision, respectively, coding gain available from the SYSDEC design based on data extractions from Figure 35 (constraint-length $K=3$). It also contains the corresponding gains obtained from the published simulation results of Heller and Jacobs.

The coding gain results found in Table VII through Table X indicate that the SYSDEC design exhibits essentially the same performance as found by Heller and Jacobs (typically within 0.1 dB).

Table VII. Hard-Decision Coding Gains for Constraint Length K=7

BER	Uncoded Eb/No (dB)	Viterbi Algorithm Limit	Gain Results of [2]	Simulated Gain Results	Simulated vs [2]
10^{-2}	4.23	----	0.58	0.58	.00
10^{-3}	6.73	3.83	1.98	2.00	+.02
10^{-4}	8.37	4.77	2.77	2.70	-.07
10^{-5}	9.57	5.37	3.27	3.08	-.19
10^{-6}	10.53	5.73	----	----	----

Table VIII. Soft-Decision Coding Gains for Constraint Length K=7

BER	Uncoded Eb/No (dB)	Viterbi Algorithm Limit	Gain Results of [2]	Simulated Gain Results	Simulated vs [2]
10^{-2}	4.23	----	----	----	----
10^{-3}	6.73	3.83	3.73	----	----
10^{-4}	8.37	4.77	4.68	4.61	-.07
10^{-5}	9.57	5.37	5.24	5.22	-.02
10^{-6}	10.53	5.73	----	----	----

Table IX. Hard-Decision Coding Gains for Constraint Length K=3

BER	Uncoded Eb/No (dB)	Viterbi Algorithm Limit	Gain Results of [2]	Simulated Gain Results	Simulated vs [2]
10^{-2}	4.23	----	0.18	----	----
10^{-3}	6.73	3.83	1.00	1.03	+ .03
10^{-4}	8.37	4.77	1.37	1.42	+ .05
10^{-5}	9.57	5.37	2.47	2.60	+ .13
10^{-6}	10.53	5.73	----	----	----

Table X. Soft-Decision Coding Gains for Constraint Length K=3

BER	Uncoded Eb/No (dB)	Viterbi Algorithm Limit	Gain Results of [2]	Simulated Gain Results	Simulated vs [2]
10^{-2}	4.23	----	----	----	----
10^{-3}	6.73	2.68	2.88	----	----
10^{-4}	8.37	3.32	3.25	3.25	.00
10^{-5}	9.57	3.72	3.54	3.54	.00
10^{-6}	10.53	3.78	----	----	----

7.3 Results of VHDL Simulations

This section contains an evaluation of the VHDL simulation results. The evaluation is intended to prove the feasibility of mapping the SYSDEC architecture into a VLSI implementation. Sample listings of VHDL simulation output can be found in Appendix C.

As the detail of the design was being finalized (see Appendix A), the results of the VHDL simulations revealed the need for certain architectural changes in the SYSDEC design. These changes were implemented, and evaluated against LINKSIM simulations for potential performance changes. These changes were as follows :

- 1) The latching of the "best-rows" was moved to the new metric-latch array. This eliminated the need for best-row registers in each of the correlation-array processing elements (a saving of $6 \times 54 \times 6 = 2304$ registers per decoder stage).
- 2) The detection of the normalization requirement was moved external to the first decoder stage, and out of the comparison-array. This eliminated the need for an extra comparator in each processing element of the comparison-array (a saving of 64 comparators per decoder stage).
- 3) The operation of the decoder stages could now be simultaneous, without retiming elements between stages.

The VHDL simulations also revealed that the design of the timing-control unit was much simpler than previously estimated, but the design of the symbol-shifter was more complex in that it required the same retiming elements as those found in the trellis-direction counter.

Since the behavioural VHDL models of the individual elements were of such detail as to be easily mapped to a standard-cell representation¹¹, the technology analysis performed in section 7.4 could proceed by compiling a detailed parts list for the SYSDEC design.

7.4 Assessment of Technology Requirements and Constraints

This section provides an estimate of hardware complexity for the proposed decoder. This is done to justify the potential for mapping the design into a single VLSI chip. This chapter will also make estimates on the technology requirements, clock speed and power consumption.

Section 7.4.1 contains an accounting of the number of transistors required to implement the SYSDEC design. Section 7.4.2 applies the transistor requirements against projected utilization^{mm} figures to estimate the technology which would be required to fabricate the design. Section 7.4.3 uses parameters consistent with the estimated technology requirements to analyze the throughput speed of the design. Section 7.4.4 uses the estimated speed and transistor count to create an estimate of the power dissipated by a VLSI implementation of the SYSDEC design.

¹¹ This assumes that the analysis is targeted for a non-optimized standard-cell implementation. In fact, the actual implementation of this type would contain hand-optimized layouts of a synthesized version of the processing elements with optimal transistor widths and minimal internal interconnect lengths. The use of standard-cells is applied as an abstraction which can be closely linked with the number of transistors required to implement the individual logic blocks; hence, the technology requirements can then be analyzed based on a known reference.

^{mm} The utilization figure is a ratio of the active silicon area to the total core-area required to implement the design (including interconnect wiring).

7.4.1 SYSDEC Design Complexity

This section contains an account of the number of transistors required to implement the various components and, ultimately, the complete SYSDEC design. Transistor counts are extracted from established standard-cell implementations of merchant-ASIC logic functions. Logic functions are extracted from the VHDL behavioural models for individual design entities (Appendix D). The number of design entities is extracted from the design specification contained in Chapter 5 (and developed in Chapter 4), and from the VHDL analysis. Due to the flexibility of constraint-lengths and the number of stages required, the analysis will proceed based on one stage of a $K=7$, rate $1/2$ convolutional code decoder. Comparisons will be made to a constraint-length $K=3$ decoder to illustrate the universality of the design approach.

Table XI lists all the logic units required by the individual design blocks of the SYSDEC design. Table XII lists the total transistor count for the required logic units, and the number of transistors required to implement the design. For comparison, Table XIII and Table XIV show the hardware requirements for a constraint-length $K=3$ decoder, given the same processing elements²².

After allowing a 5% overhead for clock buffering, the total size of one stage of the SYSDEC design is approximately 370,000 transistors (more than 28 times larger than a $K=3$ implementation).

²² If the design were targeted for a $K=3$ decoder, then it is unlikely that the individual processing elements would contain the same number of components found in the processing elements of a $K=7$ design.

Table XI. SYSDEC Parts Count for a Single K=7 Decoder Stage

	REG	XOR	HA	FA	MUX	CMP	AND	F/F
Correlation Array	7296	4224	2304	1920	--	--	--	--
Comparison Array	960	--	192	384	960	64	--	--
MetricLatch Array	960	--	128	--	--	--	128	--
Symbol Shifter	162	--	--	--	--	--	--	--
Direction Counter	15	--	--	--	--	--	--	6
Data Extractor	--	--	--	--	--	--	--	6
Normalize Detector	--	--	--	--	--	1	5	6
TOTAL	9363	4224	2624	2304	960	65	133	18

Table XII. Transistor Requirements for a Single K=7 Decoder Stage

LOGIC UNIT	QUANTITY	TRANSISTORS	TOTAL TRANS.
Register	9393	16	150288
XOR	4224	12	50688
Half Adder	2624	16	41984
Full Adder	2304	32	73728
MUX	960	10	9600
Comparator	65	350	22750
AND	133	6	798
F/F w clear	18	22	396
TOTAL			350232

Table XIII. SYSDEC Parts Count for a Single K=3 Decoder Stage

	REG	XOR	HA	FA	MUX	CMP	AND	F/F
Correlation Array	152	152	48	40	--	--	--	--
Comparison Array	60	--	12	24	60	4	--	--
MetricLatch Array	60	--	8	--	--	--	8	--
Symbol Shifter	30	--	--	--	--	--	--	--
Direction Counter	1	--	--	--	--	--	--	2
Data Extractor	--	--	--	--	--	--	--	2
Normalize Detector	--	--	--	--	--	1	1	2
TOTAL	303	152	68	64	60	5	9	6

Table XIV. Transistor Requirements for a Single K=3 Decoder Stage

LOGIC UNIT	QUANTITY	TRANSISTORS	TOTAL TRANS.
Register	303	16	4848
XOR	152	12	1824
Half Adder	68	16	1088
Full Adder	64	32	2048
MUX	60	10	600
Comparator	5	350	1750
AND	9	6	60
F/F w clear	6	22	132
TOTAL			12330

7.4.2 Estimated Technology Requirements

This section will map various implementations of the SYSDEC into specified target technologies. For this procedure, parameters from an existing systolic-array VLSI chip [18] will be used as a reference point when demonstrating the feasibility of VLSI mapping. The reference CMOS design contains one million $1.2\text{ }\mu\text{m}$ transistors in a core-size of 11.4 mm per side with a utilization of almost 100%. The clock-cycle for each of the 256 processing elements was 40 ns (25 MHz), and the chip dissipated less than 4 Watts.

The following parameters are known about the reference process :

- 1) one million transistors occupy $(11.4\text{ mm})^2 = 130\text{ mm}^2$ of raw silicon area ; therefore,
- 2) the $1.2\text{ }\mu\text{m}$ reference process uses $130\text{ }\mu\text{m}^2$ per transistor.

The following parameters can be expected in the target process :

- 1) the effective area utilization is 90%^{oo}, and
- 2) there are 2.2 million transistors in a six-stage SYSDEC design.

7.4.2.1 Complete Decoder Implementation

This analysis will attempt to map the complete implementation of the proposed decoder architecture into a sub-micron CMOS processes.

If the target process for the proposed decoder was $0.6\text{ }\mu\text{m}$ CMOS, then the following can be calculated (based on the reference design [18]) :

^{oo} This value is actually lower than would be expected, to allow for some pessimism in the layout.

- a) the ratio of transistor lengths is : $0.6 \mu\text{m} / 1.2 \mu\text{m} = 0.5$,
- b) the ratio of transistor areas is : $(0.5)^2 = 0.25$,
- c) the required transistor area is : $0.25 \times 130 \mu\text{m}^2 \times 2,200,000 = 71.5 \text{ mm}^2$,
- d) the required raw silicon area is : $71.5 \text{ mm}^2 / 90\% \approx 79.4 \text{ mm}^2$, and
- e) the required core die size (one side) is : $\sqrt{79.4 \text{ mm}^2} \approx 8.9 \text{ mm}$.

Therefore, it should be possible to fabricate the SYSDEC design in a $0.6 \mu\text{m}$ CMOS process within a 8.9 mm square core die size.

7.4.2.2 Single-Stage Implementation

As an alternative, the decoder could be manufactured as a chip set containing one decoder-stage per chip. This section will analyze the technology requirements for fabricating only one stage of the proposed decoder design.

If the target process for the proposed decoder was $1.5 \mu\text{m}$ CMOS, then the following can be calculated (based on the reference design [18]) :

- a) the ratio of transistor lengths is : $1.5 \mu\text{m} / 1.2 \mu\text{m} = 1.25$,
- b) the ratio of transistor areas is : $(1.25)^2 \approx 1.56$,
- c) the required transistor area is : $1.56 \times 130 \mu\text{m}^2 \times 370,000 = 75.2 \text{ mm}^2$,
- d) the required raw silicon area is : $75.2 \text{ mm}^2 / 90\% \approx 83.5 \text{ mm}^2$, and
- e) the required core die size (one side) is : $\sqrt{83.5 \text{ mm}^2} \approx 9.1 \text{ mm}$.

Therefore, it should be possible to fabricate a single stage of the SYSDEC design in a $1.5 \mu\text{m}$ CMOS process within a 9.1 mm square core die size.

7.4.3 Estimated Circuit Speed

Since the complexity of the ACS circuit is comparable to that of the pixel processors used in the reference design [18], then it too should have a clock period of 40 ns if manufactured in a 1.2 μm process. If the design were to be manufactured in a 0.6 μm process, and delay is proportional to capacitance (or area^{PP}), then the design should run four times faster (at 10 ns). This translates to a baseband data rate of approximately 9.4 MHz, for the K=7 decoder.

7.4.4 Estimated Power Requirements

Regardless of the process, if the speed is proportional to area (as generalized in the previous section), and power is proportional to equivalent capacitance, then the power dissipation per transistor will remain constant. For example, if the reference process [18] dissipates 4 microwatts per transistor (on average) in a 1.2 μm process, and the power-dissipation capacitance is one fourth for a 0.6 μm (based on an approximate ratio of area), then each transistor should dissipate :

$$0.25 \times 4 \text{ microwatts} \times 100 \text{ MHz} / 25 \text{ MHz} = 4 \text{ microwatts.}$$

^{PP} Technology scaling, by feature size (α), is a standard theoretical procedure [21][22]. Circuit speed scaling is proportional to α , and circuit power is proportional to α^2 . However, the theory involves scaling the voltage as well (by a factor of α), to maintain constant electric field strengths. This is acceptable in theory, however, standard parts usually are made for a single voltage supply (historically, a nominal 5.0 Volts). In general, scaling the area of a technology by a given factor does not result in the same change in speed, and is highly dependent on the technology. However, for this analysis, this technique will be used to provide an estimate of the potential speed of the design as well as power dissipation.

Hence, at 370,000 transistors per stage, the proposed design should dissipate approximately 1.5 Watts per stage (8.8 Watts for six stages), regardless of the technology used.

Historically, 5.0 Volt operation has been chosen to provide adequate noise-margins in digital electronics. With the newer technologies (ie: 0.8 μm , and especially 0.6 μm), the voltage can to be reduced to 3.3 volts, to reduce power consumption, and still maintain an effective noise-margin (reducing the voltage also reduces the effect of the electric-field strength overcoming the dielectrics, and the diffusions). Reducing the voltage would have the effect of reducing the power by a factor equal to the square (based on $P=CV^2f$) of the voltage ratio (eg: reducing the voltage from 5.0 to 3.3 volts will result in approximately 0.44 of the original power). Hence, the power would be reduced to 0.65 Watts per stage (3.9 Watts for six stages). However, a reduction in voltage will also cause an almost linear reduction in operational speed. Therefore, a 3.3 volt 0.6 μm design would have a baseband throughput rate of approximately 6.2 Mbps.

7.4.5 Summary of Technology Requirements

Table XIV shows the effect of VLSI feature size on total die size, potential speed, and power dissipation (based on 5.0 Volt operation). The three older technologies (1.2, 1.5 and 2.0 μm) are used for implementing a single decoder stage only. The newer technologies (1.0, 0.8 and 0.6 μm) are used for complete implementation of the six decoder stages. The results indicate feasible die sizes with baseband speeds ranging from 0.8 to 9.4 MHz. Power is estimated at 8.8 Watts for six stages and 1.5 Watts for a single stage.

7.5 Summary

The results of the LINKSIM simulations have proven that the design has sufficient coding-gain performance to function as intended (within 0.1 dB of published simulation results in most instances).

Table XV. Summary of VLSI Parameters vs Technology

VLSI Parameter	Selected Technology					
	Six Stages (2.2M Transistors)			One Stage (370K Transistors)		
	0.6 μm	0.8 μm	1.0 μm	1.2 μm	1.5 μm	2.0 μm
Die Size*	8.9 mm	11.9 mm	14.9 mm	7.3 mm	9.1 mm	12.2 mm
Rate	9.4 MHz	5.3 MHz	3.4 MHz	2.3 MHz	1.5 MHz	0.8 MHz
Power	8.8 W	8.8 W	8.8 W	1.5 W	1.5 W	1.5 W

* - square die configuration

The results of the VHDL implementation and simulations revealed the need for minor architectural changes, but ultimately yielded the functional blocks required to implement the SYSDEC design.

The results of the technology evaluation indicate that the entire design may be fabricated on a single VLSI die using a 0.6 μm process, with a power dissipation of 8.8 Watts, with a baseband throughput of 9.4 MHz. One stage of the design may be fabricated in a 1.5 μm process, with a power dissipation of 1.5 Watts, with a baseband throughput of 1.5 MHz.

CHAPTER 8 RECOMMENDATIONS AND CONCLUSIONS

This chapter summarizes the results of this investigation. It will discuss the performance characteristics of the proposed decoder architecture in terms of its throughput, coding gain, and the potential for VLSI implementation. It will also discuss relevant work performed throughout the course of this investigation and propose topics for future work in this field.

8.1 Overview of Achievements and Contributions

This section summarizes the work achieved and the relative contributions made by this thesis, as they pertain to the objectives.

8.1.1 The Systolic-Array Decoder Simulation Summary

This section summarizes the simulation results of the decoder architecture (SYSDEC) which was used as a test case during this investigation.

8.1.1.1 Coding Gain Performance

The results of the LINKSIM simulations (see Chapter 7) demonstrated the functionality of the decoder. The coding gain performance results closely matched those

of published ideal computer simulation results of the VA [2], with only minor discrepancies (less than 0.1 dB in most cases).

The accuracy of the results at very low bit error rates (lower than 10^{-4}) is coarse due to the limited number of available points⁹⁹ on which to base an average (ie: K=7 simulation results listed in Table III and Table IV, and plotted in Figure 34). However, these plots of the simulation results indicate that the proposed design functions as intended, and is adequate for incorporation into communication systems designs.

8.1.1.2 VLSI Implementation Potential

The results of the VHDL simulations in section 7.3 demonstrated that all necessary functional elements have been included in the investigation. This provided a basis for the physical estimations performed in section 7.4.1.

As demonstrated in section 7.4.2, the proposed architecture could be realized in a variety of technologies, in single or multi-chip format. Specifically, a hypothetical 0.6 μm CMOS technology would yield a single 8.9 mm square die supporting an on-chip clock-rate of approximately 100 MHz, and dissipating approximately 8.8 Watts (this power drops to 3.9 Watts using 3.3 Volt operation). This analysis demonstrated the potential for a single-chip implementation of the proposed architecture.

⁹⁹ The number of available simulation results is limited due to the excessively long simulation times required to emulate $6 \times 64 \times 6 = 2304$ processors. For example : executing the LINKSIM program once, producing an average of eight individual runs, for K=7 soft decision, at 1,750,000 bits per run, consumes 980,000 seconds of raw CPU time (11.3 days) on a SUN Sparcstation 1+.

8.1.1.3 Throughput Rate

Based on the technology requirements, and the estimations provided in section 7.4.3, the proposed architecture would have the potential to operate at an on-chip clock rate as high as 100 MHz, producing a baseband throughput rate of 9.4 Mbps.

This demonstrates that the throughput rate of the proposed design is comparable to commercially available products. However, the current design methodology of commercially available decoders results in throughput delays of the order $O(2^{K-1})$, whereas the throughput delay of the proposed architecture is of the order $O(2^{K-1}/(K-1))$. Hence, for higher constraint-lengths, there is a definite advantage to using the systolic-array approach.

8.1.1.4 Alternative Methods

As discussed in section 4.5, the architecture used in this investigation is two orders of magnitude faster than a uniprocessor implementation. Comparing the SYSDEC design with previous work in the area of systolic-array Viterbi decoders; it has the same throughput as the Fettweis and Meyr [29] design but is $(K-1)$ times faster than the Chang and Yao [23] design, and N^{K-1} faster than the Provence and Gupta [25][26] design (where N is the number of bits in each transmitted symbol-frame).

8.1.2 Milestones vs Objectives

Specific milestones were established, during the course of this thesis, to monitor and guide the work so that the objectives could be achieved. The following sub-sections list some of the milestones used to reach specific objectives in three key areas.

New Algorithm Generation

- 1) Look for recursive patterns in the communications algorithm and alter its form into one which is more appropriate for VLSI considerations (ie: systolic arrays).

Design Methodology

- 1) Obtain access to a high-level communications-environment simulator (ie: LINKSIM).
- 2) Write and debug high-level language code to emulate the behaviour of the proposed architecture (see "New Algorithm Generation").
- 3) Integrate the emulation-code into the communications simulator.
- 4) Obtain access to a hierarchical VLSI evaluation/simulation tool (ie: VHDL).
- 5) Write and debug the VLSI entities and bodies to describe the proposed decoder design in a full-system configuration.
- 6) Simultaneously simulate the proposed design, in both simulation environments, and make appropriate performance and architectural analysis/enhancements as required (see "VLSI Implementation").

VLSI Implementation

- 1) After identifying all the required sub-components (through hierarchical simulation analysis), estimate the required logic complexity based on existing implementations of standard logic blocks.
- 2) Use the logic complexity estimate to determine the technology required.
- 3) After selecting the appropriate technology, estimate the maximum clock period, power requirements and maximum throughput.

Based on the new algorithm, and the vertically integrated design methodology, it was possible to demonstrate a complete VLSI maximum-likelihood sequence estimator based solely on systolic arrays. The proposed test-case produced adequate coding gain, throughput and VLSI potential, as demonstrated in the analysis (Chapter 7).

8.1.3 New Ideas

This thesis has proposed ideas (Chapter 4), and modifications to existing ideas [23][25][26], which are unique to the field. The following is a list of some of these ideas:

- the use of a two-dimensional array in the implementation of a compacted trellis approach,
- the elimination of path-memory from the implementation^{iv} of the Viterbi Algorithm,

^{iv} The elimination is that of the path memory block. The information, that is stored in the path memory of a traditional Viterbi decoder, is stored in an encoded form within the sequential elements of the SYSDEC arrays.

- the strict use of systolic arrays only, in the development of the decoder architecture, and
- the potential to be more robust than a standard application of the Viterbi Algorithm²².

8.2 Recommendations

This section provides insight into potential improvements in the decoder design, as well as topics for future consideration. Its purpose is to highlight areas which were not part of the initial investigation but bear particular relevance to any continuing investigations on this work.

8.2.1 Potential Improvements on the Decoder Design

The purpose of this section is to identify potential improvements to the SYSDEC design (as presented in Chapter 5).

²² This potential exists only in comparisons to digital fixed-hardware implementations of the VA (not when compared to the pure algorithm form). There is a potential in the standard VA to diverge from the ideal path more easily because it compares/discards after each symbol-frame interval, thus having the potential to venture into local minima. Since the SYSDEC design performs a comparison/discard after (K-1) symbol-frame intervals, it is less likely to diverge from the ideal path. This distinguishing behaviour could lead to a potential reduction in the length of the required path history (ie: fewer stages may be required).

8.2.1.1 Physical Improvements

There are a number of potential improvements to the SYSDEC design which could ultimately provide higher throughput, higher coding-gain, lower transistor count and/or reduced cycle time. Potential implementations of any of these improvements must follow the same design methodology used in the design of the prototype (ie: the vertical integration of all levels of the hierarchy to allow for simultaneous trade-off analysis of system performance versus VLSI implementation). Some of these potential improvements are listed below.

Architectural Improvements

- 1) It may be possible to reduce the number of stages from 6 to 1 by sending the best-row output of the first stage to the input of a FIFO. The data extraction unit could then be placed on the output of the FIFO, with decisions being made based on the outputs of the comparison array. This approach would require $2^{(K-1)}$ simple FIFOs, each with a width of $(K-1)$ bits, and a depth of 5 or 6 stages (approximately 2000 bits, for $K=7$).
- 2) The combination of one whole row of the correlation-array into a single processing "element" would reduce the number of storage elements required, and possibly reduce the number of cycles required to compute all $2^{2(K-1)}$ possible path-metrics. However, this would also increase the cycle time per element and lower the throughput rate.
- 3) By using an extra latching stage on the output of the metric-latch array, it would be possible to isolate the normalization function from the ACS function. This would effectively reduce the critical path by 50%, allowing the design to run much faster (limited only by the speed of the processing

elements in the correlation-array). However, this modification would also create a need for retiming elements between decoder stages.

Branch Metric Calculation

- 1) The fact that the branch-metrics are linearly scaled, as the quantized symbol digresses from the hard-decision value, causes a lack of resolution in the calculation of the path metrics. For example, given an expected all-zero transition scenario (000 for each symbol), if a minor perturbation persists (say 001 as the received symbol) for $K-1$ transitions, then it would be equivalent to the reception of 000 for the first $K-2$ symbols and a final branch metric of $K-1$ for the last symbol. Clearly, the case of a prolonged minor perturbation should be considered more likely than a perfect reception followed by a large "spike". This situation could have been avoided by introducing a non-linearity in the calculation of the branch-metrics.
- 2) Reduce the complexity of the expected symbol calculation, in each processing element, to that proposed by Chang and Yao [23]. This would replace the operation by use of another systolic-array (synchronized to the trellis direction counter) which would produce all the symbols by the intersection of a reduced symbol set.

Circuit Physical Implementation

- 1) An optimization of the processing elements would result in a decreased transistor count and possibly an increase in speed. Synthesis would play an important role in this optimization.

- 2) The use of dynamic logic in the design of the processing elements could potentially reduce the required transistor count, particularly for the correlation-array.
- 3) To increase the utilization of the processing elements in the correlation-array, it may be possible to implement each row as a pyramidal structure, and mate every second row at 180 degree inversions. Each column of the array would contain 2^j elements (where $0 \leq j \leq K-1$). This would increase the utilization of each processing element to 100%, and reduce the number of elements to 33% of its original designed number. The effect of this "wave-front" approach, on silicon layout, has yet to be determined.

Normalization Improvements

- 1) The use of a better normalization routine could result in an increase in the coding-gain, and a possible reduction in the number of stages required to implement the design.

8.2.1.2 Additional Operational Features

There are numerous features which are incorporated in commercial applications of the VA which would be beneficial to a physical realization of the SYSDEC design. Some of these features are listed below.

- 1) A loss-of-sync indicator to raise an alarm if the decoder should fail to synchronize to the incoming symbol-pairs. In the terminology of this document, this would indicate frame synchronization.

- 2) A channel-quality monitor which would give an indication of the number of potential errors in the received sequence. This could be connected directly to the final state-metric comparator register in the data extraction unit.
- 3) A built-in interleaver/deinterleaver to provide compensation for channels whose errors are bursty in nature.
- 4) The inclusion of the convolutional encoder which is specific to the processing elements of the correlation-array.

8.2.2 Topics for Future Consideration

In terms of the overall realization of the underlying idea, there are a number of topics which might be considered in the continuation of this work. Some of these thoughts are listed below.

- 1) The entire SYSDEC design need not be implemented in a single package. Instead, it is possible to subdivide the design into a single device. This device would contain a single decoder stage of the SYSDEC design. Sequential instantiations of this device would provide the same functionality as that of a single IC implementation. Such a system would require that the chips be linked in tandem with a 64 MHz state-metric/best-row bus travelling in one direction, and a 6 MHz received-symbol bus travelling in the opposite direction (based on 6 Mbps baseband throughput).
- 2) To reduce the number of clock cycles required for full correlation, it may be possible to implement only a subset of the $2^{2(K-1)}$

possible paths through the densely connected trellis. This would have the effect of increasing (linearly) the throughput of the design, and decreasing (linearly) the size of the correlation-array. The effect of this methodology on the coding-gain has yet to be determined.

- 3) Moving the metric-latch array before the correlation array would reduce the time required for the ACS critical path in the comparison array. However, this would increase the number of gates required to implement the correlation-array due to the increase in bus width for path metrics.

8.3 Conclusions

This work has demonstrated the suitability of using multiple systolic array architectures in the design of single-chip, CMOS VLSI, convolutional code decoders. The level of detail has been sufficient to provide a high degree of confidence for the feasibility analysis and the mapping of the proposed design into physical implementation. During the course of this investigation, new ideas have been presented in the areas of design methodology, communications algorithms and VLSI implementation.

Various simulation engines were used in the analysis of the proposed design. The VHDL toolset was used to provide an accurate part count and subsequent summary of the design size. VHDL simulations revealed the need for approximately 2.2 million transistors to be implemented in a sub-micron CMOS technology. The LINKSIM package was used to prove that the coding-gain performance of the proposed design was virtually the same as existing non-systolic designs.

REFERENCES

1. A.J. Viterbi, "*Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm*", IEEE Transactions on Information Theory, vol. IT-13, April 1967, pp 260-269.
2. J.A. Heller and I.M. Jacobs, "*Viterbi Decoding for Satellite and Space Communication*", IEEE Transactions on Communication Technology, vol. COM-19 No. 5, October 1971, pp 835-848.
3. A.J. Viterbi, "*Convolutional Codes and Their Performance in Communication Systems*", IEEE Transactions on Communications Technology, vol. COM-19 No. 5, October 1971.
4. G.D. Forney, "*Maximum-Likelihood Sequence Estimation of Digital Sequences in the Presence of Intersymbol Interference*", IEEE Transactions on Information Theory, vol. IT-18, No. 3, May 1972, pp 363-378.
5. G.D. Forney, "*The Viterbi Algorithm*", Proceedings of the IEEE, vol. 61 No. 3, March 1973, pp 268-278.
6. G. Ungerboeck, "*Adaptive Maximum-Likelihood Receiver for Carrier-Modulated Data-Transmission Systems*", IEEE Transactions on Communications, vol. COM-22 No. 5, May 1974, pp 624-636.
7. S. Crozier, M. Wilson, K.W. Moreland, J. Camelon and P. McLane, "*Microprocessor Based Implementation and Testing of a Simple Viterbi Detector*", Canadian Electrical Engineering Journal, Vol 6 No 3 1981, pp 3-8.
8. R.M. Orndorff, P.C. Chou, J.D. Krcmarik, T.W. Doak and R. Koralek, "*Viterbi Decoder VLSI Integrated Circuit for Bit Error Correction*", IEEE International Conference on Communications, 1983, pp 149-152.
9. J.B. Cain and R.A. Kriete, "*A VLSI $R=1/2$, $K=7$ Viterbi Decoder*", National Aeronautics Conference, 1984, pp 20-27.
10. Qualcomm Incorporated, "*Q1401 $K=7$ Rate $1/2$ Single Chip Viterbi Decoder Technical Data Sheet*", September 1987.
11. H.T. Kung and C.R. Leiserson, "*Algorithms for VLSI Processor Arrays*", in C. Mead and L. Conway, "*Introduction to VLSI Systems*", Addison-Wesley, 1980.
12. H.T. Kung, "*Why Systolic Architectures*", IEEE Computer Magazine, January 1982, pp 37-46.

13. J.D. Ullman, *"Computational Aspects of VLSI"*, Computer Science Press, 1984, Chapter 5, pp 175-208.
14. S.Y. Kung, *"VLSI Array Processors"*, IEEE ASSP Magazine, July 1985, pp 4-22.
15. A.J. Viterbi and J.K. Omura, *"Principles of Digital Communication and Coding"*, McGraw-Hill Inc., 1979.
16. S. Lin and D.J. Costello, *"Error Control Coding: Fundamentals and Applications"*, Prentice-Hall Inc., 1983.
17. R.E. Blahut, *"The Theory and Practice of Error Control Codes"*, Addison-Wesley, 1983, pp 347-382, pp 457-478.
18. N. Gharachorloo, S. Gupta, E. Hokenek, P. Balasubramanian, W. Bogholtz, C. Mathieu and C. Zoulas, *"A Million Transistor Systolic Array Graphics Engine"*, Proceedings of the IEEE Conference on Systolic Arrays, 1988, pp 193-202.
19. S.Y. Kung, *"VLSI Array Processors"*, Prentice Hall, 1988.
20. M.J. Foster and H.T. Kung, *"The Design of Special Purpose VLSI Chips"*, IEEE Computer, 1980, pp 26-40.
21. N. Weste and K. Eshraghian, *"Principles of CMOS VLSI Design: A Systems Perspective"*, Addison-Wesley, 1985.
22. C. Mead and L. Conway, *"Introduction to VLSI Systems"*, Addison-Wesley, 1980.
23. C.Y. Chang and K. Yao, *"Viterbi Decoding by Systolic Array"*, 23rd Allerton Conference on Communications, Control and Computers, 1985, pp 430-439.
24. 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, January 1986, pp 142-154.
25. J.D. Provenca, *"Systolic Array Implementation for an Adaptive Viterbi Receiver"*, IEEE International Communications Conference, vol. 2 section 28.6, 1987, pp 1021-1027.
26. J.D. Provenca and Someshwar C. Gupta, *"Systolic Arrays for Viterbi Processing in Communication Systems with a Time-Dispersive Channel"*, IEEE Transactions on Communications, vol. 36 No. 10, October 1988, pp 1148-1156.
27. K. Wen, J. Wang, J. Lee and M. Lin, *"Implementation of Array Structured Maximum Likelihood Decoders"*, Proceedings of the IEEE Conference on Systolic Arrays, 1988, pp 227-236.

28. 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, pp 527-537.
29. G. Fettweis and H. Meyr, "*High-Rate Viterbi Processor: A Systolic Array Solution*", IEEE Journal on Selected Areas in Communications, vol. 8 No. 8, October 1990, pp 1520-1534.
30. R. Lipsett, E. Marschner and M. Shahdad, "*VHDL-The Language*", IEEE Design and Test, April 1986, pp 28-41.
31. IEEE, "*IEEE Standard VHDL Language Reference Manual - Std 1076-1987*", New York, IEEE, 1988.
32. R. Lipsett, C. Schaefer and C. Ussery, "*VHDL: Hardware Description and Design*", Kluwer Academic Publishers, 1989.
33. Steve Carlson, "*Introduction to HDL-Based Design Using VHDL*", Synopsys Inc., February 1, 1990.
34. Intermetrics Inc., "*User's Manual for the Standard VHDL 1076 Support Environment (Draft)*", 5 August 1988.
35. P. Idez and M. Moher, "*LINKSIM Simulation Program Software Users' Guide*", Miller Communications Systems Ltd., April 28, 1988.
36. Ferrel G. Stremler, "*Introduction to Communication Systems*", 2nd Edition, Addison-Wesley, 1982.

APPENDIX A : SYSDEC Design Verification History

This appendix describes the history of the iterative design and verification process which resulted in the final SYSDEC prototype.

A.1 Introduction

While the SYSDEC design was maturing, and simultaneous LINKSIM and VHDL simulations were being performed, much was learned about the original architecture. Through a process of refinement, there developed a convergence upon the final prototype design.

A.2 Design Maturation

The following points summarize revelations made during the design process :

- 1) The original correlation-array processing elements were designed to transfer the "best-row" start-state along with the partial path metrics (as they were being calculated) so that they could be used immediately upon exiting the correlation-array (and entering the comparison-array). After writing the VHDL interface and behavioural model, it became obvious that the registers inserted into the processing element served no useful function. The desired function was to delay and hold the "best-row" values for subsequent input into the comparison-array. Additionally, since the previous state-metrics accompanied the calculation, the bit-width of the processing-element adders had to be increased (adding to the delay time). A simple architectural change saw the delay-and-hold function transferred to a newly created entity; the metric-latch array. This change would not have affected the LINKSIM results.
- 2) The original design of the comparison-array called for normalization (and

normalization detection) to be performed in each element. VHDL simulations revealed that this produced an unacceptable delay in the calculations. Again, a minor alteration was made to the SYSDEC architecture. The normalization detector was made a separate entity (which was only required on the first stage), and the normalization itself was made part of the function of the metric-latch array. Subsequent LINKSIM simulations indicated no change in the coding gain.

- 3) The original design called for re-timing elements to be placed between subsequent decoder stages. But, due to the modifications made in 1) and 2), and after VHDL timing simulations, it was discovered that this was no longer necessary.
- 4) Original design-size estimates had a sizable percentage of the design devoted to the timing-control unit. This was found not to be the case. After writing the VHDL behavioural model, it was discovered that a very simple, 64-state, FSM would accomplish the task of coordinating the various design blocks.
- 5) Original design-size estimates had assumed that the design of the symbol-shifters was trivial. This was found not to be the case. After writing the VHDL behavioural model, it was discovered that in order to frequency shift the incoming data from the channel rate to the on-chip clock rate divided by $(K-1)$, it required a large number of storage and re-timing registers.

A.3 Design Potential

The following points summarize ideas, made during the design process, which would benefit the architecture if implemented :

- 1) Through VHDL simulations, it was discovered that the largest delay path was not the correlation-array processing elements (as was originally assumed), but rather the concatenation of latching, normalizing, adding and comparing. This problem could be avoided by inserting an extra latching stage on the output of the metric-latch array (thus isolating the normalization function). However, it was realized that this would create a need for extra re-timing latches between decoder stages and, as such, was not implemented.
- 2) Original LINKSIM simulations indicated that the SYSDEC design suffered when a linear-distance branch-metric calculation was performed rather than using a non-linear function. However, a non-linear function, placed within the correlation-array processing elements, would significantly increase the area of the design. Hence, this coding-gain hit was allowed.
- 3) The structural nature of the VHDL behavioural partitioning allowed for ease of implementation. However, when the VLSI analysis was performed, it became obvious that the logical boundaries within processing elements should be more flexible to allow synthesis/optimization to play an important role in design reduction.

APPENDIX B : Sample VHDL Control Files

This appendix contains samples of files and commands which are required for execution of the VHDL tool set. The following examples are given :

- 1) All the commands required to analyze, module-generate and build the executable code to simulate the SYSDEC design.**
- 2) Syntax of the commands used to a) invoke the simulator, b) generate a complete signal map and c) generate a signal-trace report.**
- 3) An example of the file used by the simulator to invoke tracing of specific signals within the design. Each signal has been specified by a full path name within the SYSDEC hierarchy.**
- 4) An example of the control file used in generating the signal-trace report. It contains signal lists and format commands which specify the appearance of the generated report.**

B.1 Analyze, Generate and Build Commands

```

--
-- Commands used to analyze, module-generate and build the simulation code
--
vhdl -lis sysdec_package.vhd
mq sysdec_package
mq -body sysdec_package
--
vhdl -lis pe_latch.vhd
mq pe_latch\{pe_latch_behavioural\}
--
vhdl -lis pe_comp.vhd
mq pe_comp\{pe_comp_behavioural\}
--
vhdl -lis pe_corr.vhd
mq pe_corr\{pe_corr_behavioural\}
--
vhdl -lis corr_row.vhd
mq corr_row\{structure\}
--
vhdl -lis corr_array.vhd
mq corr_array\{structure\}
--
vhdl -lis comp_array.vhd
mq comp_array\{structure\}
--
vhdl -lis latch_array.vhd
mq latch_array\{structure\}
--
vhdl -lis direction_counter.vhd
mq direction_counter\{counter_behavioural\}
--
vhdl -lis decoder_stage.vhd
mq decoder_stage\{structure\}
--
vhdl -lis symbol_shifter.vhd
mq symbol_shifter\{shift_behavioural\}
--
vhdl -lis normalize_detector.vhd
mq normalize_detector\{detect_behavioural\}
--
vhdl -lis data_extractor.vhd
mq data_extractor\{extract_behavioural\}
--
vhdl -lis timing_controller.vhd
mq timing_controller\{timing_behavioural\}
--
vhdl -lis test_bench.vhd
mq -top test_bench\{test\}
--
build -replace test_bench\{test\}

```

B.2 Simulation and Report Generation Commands

```
--
--
-- Command to simulate and trace specific signals (listed in a separate file)
--
sim -full -trace=test.trace -param=40 test
--
--
-- Command to produce a simulation map of all possible signals
--
sim -map=test.rpm test
--
--
-- Command to generate a simulation report with signal transition times
--
rg -report=test.rpt test\(\1\) test
```

B.3 Simulation Trace File

```
--
-- Control file used to specify the signals to be traced during the simulation
--
--
: FRAME_CLOCK ;
: CHIP_CLOCK ;
: FRAME_DATA_A(0) ;
: FRAME_DATA_B(0) ;
: LATCH_ENABLE ;
: NORMALIZE ;
/EXTRACT_DATA : BEST_METRICS ;
/EXTRACT_DATA : TEMP_METRIC ;
/EXTRACT_DATA : TEMP_ROW ;
/EXTRACT_DATA : EXTRACT_ENABLE ;
/NORMALIZATION_TESTER : METRIC_IN ;
/NORMALIZATION_TESTER : START_DETECT ;
/NORMALIZATION_TESTER : NORM_COUNT ;
/DC0 : PATH_METRICS(0) ;
/DC0 : STATE_METRICS(0) ;
/DC0/CORRELATION/ROW0/PE0 : STATE_OUT ;
/DC0/CORRELATION/ROW0/PE0 : METRIC_OUT ;
/DC0/CORRELATION.ROW2(TRUE)/ROWf : SYMBOL_A_IN ;
/DC0/CORRELATION.ROW2(TRUE)/ROWf : SYMBOL_B_IN ;
/DC0/CORRELATION.ROW2(TRUE)/ROWf.PE2(TRUE)/PEf : STATE_OUT ;
/DC0/CORRELATION.ROW2(TRUE)/ROWf.PE2(TRUE)/PEf : METRIC_OUT ;
/DC0/COMPARISON.PE2(TRUE)/PEf : BEST_ROW_OUT ;
/DC0/COMPARISON.PE2(TRUE)/PEf : BEST_METRIC_OUT ;
/DC0/COMPARISON.PE2(TRUE)/PEf : BEST_ROW_IN ;
/DC0/COMPARISON.PE2(TRUE)/PEf : BEST_METRIC_IN ;
/DC0/COMPARISON.PE2(TRUE)/PEf : PATH_METRIC ;
/DC0/COMPARISON.PE2(TRUE)/PEf : BEST_ROW ;
/DC0/COMPARISON.PE2(TRUE)/PEf : STATE_METRIC ;
/DC0/LATCH/PE0 : BEST_METRIC_IN ;
/DC0/LATCH/PE0 : BEST_METRIC_OUT ;
/DC0/LATCH/PE0 : LATCH_ENABLE_IN ;
/DC0/LATCH.PE2(TRUE)/PEf : LATCH_ENABLE_IN ;
/DC0/LATCH.PE2(TRUE)/PEf : BEST_METRIC_IN ;
/DC0/LATCH.PE2(TRUE)/PEf : BEST_METRIC_OUT ;
/DC0/COMPARISON/PE0 : BEST_ROW_OUT ;
/DC0/COMPARISON/PE0 : BEST_METRIC_OUT ;
/DC0/COMPARISON/PE0 : PATH_METRIC ;
/DC0/COMPARISON/PE0 : BEST_ROW ;
/DC0/COMPARISON/PE0 : STATE_METRIC ;
```

B.4 Report Generation Control File

```

--
-- Control file used to specify the format of the report output
--
--
simulation_report TEST is
begin
  sample_signals by_event in ps;
  select_start_time 0ns;
  -- select_stop_time 1500ns;
  page_width is 80;
  -- PAGE_WIDTH IS 116;
  -- page_length is 72;
  -- PAGE_LENGTH IS 1000;
  -- remove_page_id;
  -- remove_all_id;
  signal_format is horizontal;
  -- select_signal : FRAME_CLOCK;
  -- select_signal : CHIP_CLOCK;
  -- select_signal : FRAME_DATA_A(0);
  -- select_signal : FRAME_DATA_B(0);
  select_signal /DC0/CORRELATION.ROW2(TRUE)/ROWf.PE2(TRUE)/PEf : STATE_OUT;
  select_signal /DC0/COMPARISON.PE2(TRUE)/PEf : STATE_METRIC;
  select_signal /DC0/COMPARISON.PE2(TRUE)/PEf : BEST_METRIC_OUT;
  -- select_signal /DC0 : PATH_METRICS(0);
  -- select_signal /DC0 : STATE_METRICS(0);
  -- select_signal /DC0/TRELLIS_DIRECTION_COUNTER : COUNTER_STATE;
  -- select_signal /EXTRACT_DATA : BEST_METRICS;
  -- select_signal /EXTRACT_DATA : TEMP_METRIC;
  -- select_signal /EXTRACT_DATA : TEMP_ROW;
  -- select_signal /EXTRACT_DATA : EXTRACT_ENABLE;
  -- select_signal /NORMALIZATION_TESTER : METRIC_IN;
  -- select_signal /NORMALIZATION_TESTER : START_DETECT;
  -- select_signal /NORMALIZATION_TESTER : NORM_COUNT;
  -- select_signal /DC0/LATCH/PE0 : BEST_METRIC_IN;
  -- select_signal /DC0/LATCH/PE0 : BEST_METRIC_OUT;
  -- select_signal /DC0/LATCH/PE0 : LATCH_ENABLE_IN;
  -- select_signal /DC0/LATCH/PE2/PEf : LATCH_ENABLE_IN;
  -- select_signal /DC0/LATCH/PE2/PEf : BEST_METRIC_IN;
  -- select_signal /DC0/LATCH/PE2/PEf : BEST_METRIC_OUT;
  -- select_signal /DC0/COMPARISON/PE0 : BEST_ROW_OUT;
  -- select_signal /DC0/COMPARISON/PE0 : STATE_METRIC;
  -- select_signal /DC0/COMPARISON/PE0 : BEST_METRIC_OUT;
  -- select_signal /DC0/COMPARISON.PE1(1)/PEn : BEST_ROW_OUT;
  -- select_signal /DC0/COMPARISON.PE1(1)/PEn : BEST_METRIC_OUT;
  -- select_signal : LATCH_ENABLE,NORMALIZE;
end TEST;

```

APPENDIX C : Sample VHDL Simulation Output

This appendix contains samples of the VHDL simulation outputs produced by simulations of the SYSDEC design. It consists of two file listings :

- 1) The standard output produced after invoking the simulator. The assertion violations are at the "Note" level only, and are used to monitor the progress of the simulation during execution.
- 2) The results of the report generation program. This listing corresponds to the report control file listed in Appendix A.

C.1 Sample Simulation Execution Output

Standard VHDL 1076 Support Environment - Intermetrics, Inc.
Version 2.0 - 1 January 1990

Simulation started on Friday, August 24, 1990 at 10:03:18 AM

Kernel is <<LWRIGHT>>TEST

Traced signals taken from test.trace
System messages displayed at or above the NOTE level
Assertion Violations displayed at or above the NOTE level

%VHDSIM-N-SIGTRAN Signal Tracing turned on

%VHDSIM-N-ASSERTV Assertion Violation after 999999996 fs
 at line 55 in design unit <<LWRIGHT>>SYMBOL_SHIFTER(SHIFT_BEHAVIOURAL)
End of one densely connected trellis stage.

%VHDSIM-N-ASSERTV Assertion Violation after 1999999992 fs
 at line 55 in design unit <<LWRIGHT>>SYMBOL_SHIFTER(SHIFT_BEHAVIOURAL)
End of one densely connected trellis stage.

%VHDSIM-N-ASSERTV Assertion Violation after 2999999988 fs
 at line 55 in design unit <<LWRIGHT>>SYMBOL_SHIFTER(SHIFT_BEHAVIOURAL)
End of one densely connected trellis stage.

%VHDSIM-N-ASSERTV Assertion Violation after 3999999984 fs
 at line 55 in design unit <<LWRIGHT>>SYMBOL_SHIFTER(SHIFT_BEHAVIOURAL)
End of one densely connected trellis stage.

%VHDSIM-N-QUIESCE Quiescent state reached with no response after 5158854150 fs

System Message Summary: 2 notes, 0 warnings, 0 errors
Assertion Summary: 4 notes, 0 warnings, 0 errors
Simulation contained 15270 signals and 3990 behaviors
There were 3614452 transactions in this simulation
Simulation completed on Friday, August 24, 1990 at 12:42:54 AM
Elapsed simulation time of 2:39:36 (H:MM:SS)
Initialization CPU time was 0:12.52 (MM:SS.ss)
Execution CPU time was and 10:51.22 (MM:SS.ss)

Elapsed CPU: 2280.360

C.2 Sample Signal-Trace Report

AUG-21-1990 07:38:54

VHDL Report Generator

PAGE 1

Vhdl Simulation Report

Kernel Library Name: <<LWRIGHT>>TEST
 Kernel Creation Date: AUG-19-1990
 Kernel Creation Time: 11:08:59
 Run Identifier: 1
 Run Date: AUG-19-1990
 Run Time: 11:08:59

Report Control Language File: test.rcl
 Report Output File : test.rpt

Max Time: 9223372036854775807
 Max Delta: 2147483646

Report Control Language :

```
simulation_report TEST is
begin
  sample_signals by_event in ps;
  select_start_time 0ns;
  -- select_stop_time 1500ns;
  page_width is 80;
  -- PAGE_WIDTH IS 116;
  -- page_length is 72;
  PAGE_LENGTH IS 1000;
  -- remove_page_id;
  -- remove_all_id;
  signal_format is horizontal;
  -- select_signal : FRAME_CLOCK;
  -- select_signal : CHIP_CLOCK;
  -- select_signal : FRAME_DATA_A(0);
  -- select_signal : FRAME_DATA_B(0);
  select_signal /DCO/CORRELATION.ROW2(TRUE)/ROW1.PE2(TRUE)/PE1 : STATE_OUT;
  select_signal /DCO/COMPARISON.PE2(TRUE)/PE1 : STATE_METRIC;
  select_signal /DCO/COMPARISON.PE2(TRUE)/PE1 : BEST_METRIC_OUT;
  -- select_signal /DCO : PATH_METRICS(0);
  -- select_signal /DCO : STATE_METRICS(0);
  -- select_signal /EXTRACT_DATA : BEST_METRICS;
  -- select_signal /EXTRACT_DATA : TEMP_METRIC;
  -- select_signal /EXTRACT_DATA : TEMP_ROW;
  -- select_signal /EXTRACT_DATA : EXTRACT_ENABLE;
  -- select_signal /NORMALIZATION_TESTER : METRIC_IN;
  -- select_signal /NORMALIZATION_TESTER : START_DETECT;
  -- select_signal /NORMALIZATION_TESTER : NORM_COUNT;
  -- select_signal /DCO/LATCH/PE0 : BEST_METRIC_IN;
  -- select_signal /DCO/LATCH/PE0 : BEST_METRIC_OUT;
  -- select_signal /DCO/LATCH/PE0 : LATCH_ENABLE_IN;
  -- select_signal /DCO/LATCH/PE2/PE1 : LATCH_ENABLE_IN;
  -- select_signal /DCO/LATCH/PE2/PE1 : BEST_METRIC_IN;
  -- select_signal /DCO/LATCH/PE2/PE1 : BEST_METRIC_OUT;
  -- select_signal /DCO/COMPARISON/PE0 : BEST_ROW_OUT;
  -- select_signal /DCO/COMPARISON/PE0 : STATE_METRIC;
  -- select_signal /DCO/COMPARISON/PE0 : BEST_METRIC_OUT;
  -- select_signal /DCO/COMPARISON.PE1(1)/PE1 : BEST_ROW_OUT;
  -- select_signal /DCO/COMPARISON.PE1(1)/PE1 : BEST_METRIC_OUT;
  -- select_signal : LATCH_ENABLE,NORMALIZE;
end TEST;
```

Report Format Information :

Time is in PS relative to the start of simulation
 Time period for report is from 0 PS to End of Simulation
 Signal values are reported by event (' ' indicates no event)

AUG-21-1990 07:38:54

VHDL Report Generator

PAGE 2

TIME	-----SIGNAL NAMES-----		
(PS)	STATE_OUT(5 DOWNT0 0)	STATE_METRIC	BEST_METRIC_OUT
0	"000000"	0	0
1094649*	"000001"		
1095449*			14
1110274*	"000010"		
1111074*			21
1125899*	"000011"		
1141524*	"000100"		
1142324*			14
1157149*	"000101"		
1172774*	"000110"		
1188399*	"000111"		
1204024*	"001000"		
1219649*	"001001"		
1235274*	"001010"		
1236074*			21
1250899*	"001011"		
1266524*	"001100"		
1267324*			14
1282149*	"001101"		
1282949*			7
1297774*	"001110"		
1298574*			14
1313399*	"001111"		
1329024*	"010000"		
1344649*	"010001"		
1360274*	"010010"		
1375899*	"010011"		
1391524*	"010100"		
1407149*	"010101"		
1407949*			21
1422774*	"010110"		
1423574*			14
1438399*	"010111"		
1454024*	"011000"		
1469649*	"011001"		
1485274*	"011010"		
1500899*	"011011"		
1516524*	"011100"		
1517324*			7
1532149*	"011101"		
1532949*			14
1545048*			7
1547774*	"011110"		
1556674*			28
1560673*			14
1563399*	"011111"		
1579024*	"100000"		
1579824*			21
1594649*	"100001"		
1610274*	"100010"		
1625899*	"100011"		
1626699*			7
1641524*	"100100"		
1642324*			21
1657149*	"100101"		
1672774*	"100110"		
1673574*			14
1688399*	"100111"		
1689199*			7
1704024*	"101000"		
1704824*			14
1719649*	"101001"		

1735274*	"101010"	
1750899*	"101011"	
1751699*		7
1766524*	"101100"	
1767324*		21
1782149*	"101101"	
1797774*	"101110"	
1798574*		7
1813399*	"101111"	
1814199*		14
1829024*	"110000"	
1829824*		21
1844649*	"110001"	
1860274*	"110010"	
1861074*		14
1875899*	"110011"	
1876699*		7
1891524*	"110100"	
1907149*	"110101"	
1922774*	"110110"	
1923574*		14
1938399*	"110111"	
1954024*	"111000"	
1969649*	"111001"	
1985274*	"111010"	
1986074*		7
2000899*	"111011"	
2001699*		14
2016524*	"111100"	
2017324*		21
2032149*	"111101"	
2032949*		28
2033749*		
2047774*	"111110"	
2063399*	"111111"	
2064199*		21
2079024*	"000000"	
2079824*		0
2094649*	"000001"	
2095449*		14
2110274*	"000010"	
2111074*		21
2125899*	"000011"	
2141524*	"000100"	
2142324*		28
2157149*	"000101"	
2157949*		21
2172774*	"000110"	
2173574*		28
2188399*	"000111"	
2204024*	"001000"	
2204824*		21
2219649*	"001001"	
2235274*	"001010"	
2236074*		28
2250899*	"001011"	
2266524*	"001100"	
2267324*		21
2282149*	"001101"	
2282949*		14
2297774*	"001110"	
2313399*	"001111"	
2329024*	"010000"	
2329824*		28
2344649*	"010001"	
2345449*		21
2360274*	"010010"	
2361074*		28
2375899*	"010011"	

2391524*	"010100"	
2392324*		14
2407149*	"010101"	
2407949*		28
2422774*	"010110"	
2423574*		21
2438399*	"010111"	
2454024*	"011000"	
2454824*		28
2469649*	"011001"	
2485274*	"011010"	
2500899*	"011011"	
2516524*	"011100"	
2532149*	"011101"	
2532949*		21
2547774*	"011110"	
2548574*		28
2563399*	"011111"	
2579024*	"100000"	
2579824*		35
2594649*	"100001"	
2610274*	"100010"	
2611074*		28
2625899*	"100011"	
2626699*		21
2641524*	"100100"	
2642324*		28
2657149*	"100101"	
2672774*	"100110"	
2673574*		21
2688399*	"100111"	
2689199*		14
2704024*	"101000"	
2719649*	"101001"	
2735274*	"101010"	
2736074*		21
2750899*	"101011"	
2766524*	"101100"	
2767324*		28
2782149*	"101101"	
2797774*	"101110"	
2798574*		14
2813399*	"101111"	
2814199*		21
2829024*	"110000"	
2829824*		28
2844649*	"110001"	
2860274*	"110010"	
2875899*	"110011"	
2876699*		21
2891524*	"110100"	
2892324*		14
2907149*	"110101"	
2922774*	"110110"	
2923574*		28
2938399*	"110111"	
2939199*		21
2954024*	"111000"	
2954824*		28
2969649*	"111001"	
2985274*	"111010"	
2986074*		21
3000899*	"111011"	
3001699*		28
3016524*	"111100"	
3032149*	"111101"	
3033749*		
3047774*	"111110"	
3048574*		35

3063399*	"111111"	
3079024*	"000000"	
3079824*		36
3094649*	"000001"	
3095449*		32
3110274*	"000010"	
3111074*		39
3125899*	"000011"	
3126699*		41
3141524*	"000100"	
3142324*		39
3157149*	"000101"	
3157949*		42
3172774*	"000110"	
3173574*		37
3188399*	"000111"	
3189199*		39
3204024*	"001000"	
3204824*		41
3219649*	"001001"	
3220449*		37
3235274*	"001010"	
3236074*		38
3250899*	"001011"	
3251699*		36
3266524*	"001100"	
3267324*		44
3282149*	"001101"	
3282949*		40
3297774*	"001110"	
3298574*		41
3313399*	"001111"	
3314199*		42
3329024*	"010000"	
3329824*		43
3344649*	"010001"	
3345449*		39
3360274*	"010010"	
3361074*		40
3375899*	"010011"	
3376699*		42
3391524*	"010100"	
3392324*		39
3407149*	"010101"	
3407949*		41
3422774*	"010110"	
3423574*		42
3438399*	"010111"	
3439199*		40
3454024*	"011000"	
3454824*		34
3469649*	"011001"	
3470449*		38
3485274*	"011010"	
3486074*		39
3500899*	"011011"	
3501699*		37
3516524*	"011100"	
3517324*		39
3532149*	"011101"	
3532949*		35
3547774*	"011110"	
3548574*		36
3563399*	"011111"	
3564199*		38
3579024*	"100000"	
3594649*	"100001"	
3595449*		36

APPENDIX D : SYSDEC VHDL Source Listings

This appendix contains all the VHDL listings used to simulate the hardware implementation of the SYSDEC design. This appendix contains the following VHDL listings :

COMP_ARRAY.VHD	The structural representation of the comparison-array.
CORR_ARRAY.VHD	The structural representation of the correlation-array.
DATA_EXTRACTOR.VHD	The behavioral model for the data-extraction unit.
DECODER_STAGE.VHD	The structural representation of a single decoder-stage.
DIRECTION_COUNTER.VHD	The behavioral model for the trellis direction-counter.
LATCH_ARRAY.VHD	The structural representation of the metric-latch array.
NORMALIZE_DETECTOR.VHD	The behavioral model of the normalization detector used after the first decoder-stage.
PE_COMP.VHD	The behavioral model for a processing element in the comparison-array.
PE_CORR.VHD	The behavioral model for a processing element in the correlation-array.

PE_LATCH.VHD	The behavioral model for a processing element in the metric-latch array.
SYMBOL_SHIFTER.VHD	The behavioral model for the symbol-shifter.
SYSDEC_PACKAGE.VHD	A package containing declarations for the data types and constant values. This package also contains the necessary conversion and resolution functions.
TEST_BENCH.VHD	This is the structural integration of the design sub-components. It also contains the necessary simulation stimulus and "watch-dog" functions.
TIMING_CONTROLLER.VHD	The behavioral model for the timing-control unit.

The listing of the design entities has been in alphabetical order, and does not reflect the structure of the design hierarchy.

D.1 The Comparison Array

```

-- ++
-- facility:
--     COMP_ARRAY
--
-- abstract:
--     The comparison array for one decoder stage.
--
-- author:
--     Lance G. Wright, P.Eng.
--
-- modification history:
--     03/08/90 : Created
--
use work.SYSDEC_PACKAGE.all;

entity COMP_ARRAY is
    generic (
        Comparator_Delay : Time := 1 ns;
        Multiplexor_Delay : Time := 1 ns;
        Big_Adder_Delay   : Time := 1 ns;
        FF_Delay           : Time := 1 ns;
    );
    port (
        CLOCK           : in Bit;
        STATE_METRIC     : in S_Metric_Array;
        BEST_ROW         : in State_Array;
        PATH_METRIC       : in P_Metric_Array;
        BEST_ROW_OUT      : out State;
        BEST_METRIC_OUT   : out S_Metric;
    );
end COMP_ARRAY;

architecture STRUCTURE of COMP_ARRAY is
    component COMP_PE
        generic (
            Comparator_Delay : Time := 10 ns;
            Multiplexor_Delay : Time := 10 ns;
            Big_Adder_Delay   : Time := 10 ns;
            FF_Delay           : Time := 10 ns;
        );
        port (
            CLOCK           : in Bit;
            STATE_METRIC     : in S_Metric;
            BEST_ROW         : in State;
            PATH_METRIC       : in P_Metric;
            BEST_ROW_IN       : in State;
            BEST_METRIC_IN    : in S_Metric;
            BEST_ROW_OUT      : out State;
            BEST_METRIC_OUT   : out S_Metric;
        );
    end component;

    for all: COMP_PE use entity work.PE_COMP (PE_COMP_BEHAVIOURAL);

    signal BEST_ROW_IN      : State;
    signal BEST_METRIC_IN   : S_Metric;
    signal TEMP_METRIC       : S_Metric_Array;

```

```

    signal TEMP_ROW      : State_Array;

begin
    HARDWIRED_INPUTS:
    process
    begin
        BEST_ROW_IN <= B"000000";
        BEST_METRIC_IN <= Max_Reg;
        wait; -- forever
    end process;

    PE0:
        COMP_PE
        generic map
        (
            Comparator_Delay => Comparator_Delay,
            Multiplexor_Delay => Multiplexor_Delay,
            Big_Adder_Delay => Big_Adder_Delay,
            FF_Delay => FF_Delay
        )
        port map
        (
            CLOCK => CLOCK,
            STATE_METRIC => STATE_METRIC(0),
            BEST_ROW => BEST_ROW(0),
            PATH_METRIC => PATH_METRIC(0),
            BEST_ROW_IN => BEST_ROW_IN,
            BEST_METRIC_IN => BEST_METRIC_IN,
            BEST_ROW_OUT => TEMP_ROW(0),
            BEST_METRIC_OUT => TEMP_METRIC(0)
        );

    PE1:
        for i in 1 to State_Count-2 generate
            PEn:
                COMP_PE
                generic map
                (
                    Comparator_Delay => Comparator_Delay,
                    Multiplexor_Delay => Multiplexor_Delay,
                    Big_Adder_Delay => Big_Adder_Delay,
                    FF_Delay => FF_Delay
                )
                port map
                (
                    CLOCK => CLOCK,
                    STATE_METRIC => STATE_METRIC(i),
                    BEST_ROW => BEST_ROW(i),
                    PATH_METRIC => PATH_METRIC(i),
                    BEST_ROW_IN => TEMP_ROW(i-1),
                    BEST_METRIC_IN => TEMP_METRIC(i-1),
                    BEST_ROW_OUT => TEMP_ROW(i),
                    BEST_METRIC_OUT => TEMP_METRIC(i)
                );
            end generate;

    PE2:
        if State_Count > 1 generate
            PEF:
                COMP_PE
                generic map
                (
                    Comparator_Delay => Comparator_Delay,
                    Multiplexor_Delay => Multiplexor_Delay,
                    Big_Adder_Delay => Big_Adder_Delay,
                    FF_Delay => FF_Delay
                )
                port map

```

```

(
  CLOCK => CLOCK,
  STATE_METRIC => STATE_METRIC(State_Count-1),
  BEST_ROW => BEST_ROW(State_Count-1),
  PATH_METRIC => PATH_METRIC(State_Count-1),
  BEST_ROW_IN => TEMP_ROW(State_Count-2),
  BEST_METRIC_IN => TEMP_METRIC(State_Count-2),
  BEST_ROW_OUT => BEST_ROW_OUT,
  BEST_METRIC_OUT => BEST_METRIC_OUT
);
end generate;
end STRUCTURE;

```

D.2 The Correlation Array

```

-- **
-- facility:
--
-- CORR ARRAY
--
-- abstract:
--
-- The correlation array for one decoder stage.
--
-- author:
--
-- Lance G. Wright, P.Eng.
--
-- modification history:
--
-- 03/08/90 : Created
--
--

use work.SYSDEC_PACKAGE.all;

entity CORR_ARRAY is
  generic (
    Symbol_Delay : Time := 1 ns;
    Small_Adder_Delay : Time := 1 ns;
    Adder_Delay : Time := 1 ns;
    FF_Delay : Time := 1 ns
  );
  port (
    CLOCK : in Bit;
    DIR_IN : in Bit_Vector (Constraint_length-2 downto 0);
    SYMBOL_A_IN : in Symbol_Array;
    SYMBOL_B_IN : in Symbol_Array;
    METRIC_OUT : out P_Metric_Array
  );
end CORR_ARRAY;

architecture STRUCTURE of CORR_ARRAY is
  component BEST_ROW_CORR
    generic (
      Symbol_Delay : Time;
      Small_Adder_Delay : Time;
      Adder_Delay : Time;
      FF_Delay : Time
    )
    port (
      CLOCK : in Bit;

```

```

        DIR_IN      : in Bit_Vector;
        STATE_IN    : in State;
        SYMBOL_A_IN  : in Symbol_Array;
        SYMBOL_B_IN  : in Symbol_Array;
        DIR_OUT      : out Bit_Vector;
        METRIC_OUT   : out P_Metric;
        SYMBOL_A_OUT  : out Symbol_Array;
        SYMBOL_B_OUT  : out Symbol_Array
    );
end component;

for all: BEST_ROW_CORR
    use entity work.CORR_ROW(STRUCTURE);

    type Direction_Array is array (State_Count-1 downto 0) of State;
    type Symbol_Array_Array is array (State_Count-1 downto 0) of Symbol_Array;

    signal STATE_IN      : State_Array;
    signal TEMP_DIR      : Direction_Array;
    signal TEMP_A         : Symbol_Array_Array;
    signal TEMP_B         : Symbol_Array_Array;

begin

    HARDWIRED_INPUTS:
    process
        variable BINARY_ONE : State := int_to_bit(1);
        variable BINARY_COUNT : State := int_to_bit(0);
    begin
        for i in 0 to State_Count-1 loop
            STATE_IN(i) <= BINARY_COUNT;
            BINARY_COUNT := BINARY_COUNT + BINARY_ONE;
        end loop;
        wait; -- forever
    end process;

    ROW0:
        BEST_ROW_CORR
        generic map
        (
            Symbol_Delay => Symbol_Delay,
            Small_Adder_Delay => Small_Adder_Delay,
            Adder_Delay => Adder_Delay,
            FF_Delay => FF_Delay
        )
        port map
        (
            CLOCK => CLOCK,
            DIR_IN => DIR_IN,
            STATE_IN => STATE_IN(0),
            SYMBOL_A_IN => SYMBOL_A_IN,
            SYMBOL_B_IN => SYMBOL_B_IN,
            DIR_OUT => TEMP_DIR(0),
            METRIC_OUT => METRIC_OUT(0),
            SYMBOL_A_OUT => TEMP_A(0),
            SYMBOL_B_OUT => TEMP_B(0)
        );

    ROW1:
        for i in 1 to State_Count-2 generate
            ROWn:
                BEST_ROW_CORR
                generic map
                (
                    Symbol_Delay => Symbol_Delay,
                    Small_Adder_Delay => Small_Adder_Delay,
                    Adder_Delay => Adder_Delay,
                    FF_Delay => FF_Delay
                )

```



```

        port map
        (
            CLOCK           => CLOCK,
            DIR_IN           => TEMP_DIR(i-1),
            STATE_IN         => STATE_IN(i),
            SYMBOL_A_IN      => TEMP_A(i-1),
            SYMBOL_B_IN      => TEMP_B(i-1),
            DIR_OUT           => TEMP_DIR(i),
            METRIC_OUT        => METRIC_OUT(i),
            SYMBOL_A_OUT      => TEMP_A(i),
            SYMBOL_B_OUT      => TEMP_B(i)
        );
    end generate;

ROW2:
    if State_Count > 1 generate
        ROWf:
            BEST_ROW_CORR
            generic map
            (
                Symbol_Delay      => Symbol_Delay,
                Small_Adder_Delay => Small_Adder_Delay,
                Adder_Delay        => Adder_Delay,
                FF_Delay           => FF_Delay
            )
            port map
            (
                CLOCK           => CLOCK,
                DIR_IN           => TEMP_DIR(State_Count-2),
                STATE_IN         => STATE_IN(State_Count-1),
                SYMBOL_A_IN      => TEMP_A(State_Count-2),
                SYMBOL_B_IN      => TEMP_B(State_Count-2),
                DIR_OUT           => TEMP_DIR(State_Count-1),
                METRIC_OUT        => METRIC_OUT(State_Count-1),
                SYMBOL_A_OUT      => TEMP_A(State_Count-1),
                SYMBOL_B_OUT      => TEMP_B(State_Count-1)
            );
        end generate;
    end STRUCTURE;

```

D.3 One Row of the Correlation Array

```

--  **
--  facility:
--
--      CORR_ROW
--
--  abstract:
--
--      One row of the correlation array.
--
--  author:
--
--      Lance G. Wright, P.Eng.
--
--  modification history:
--
--      02/08/90 : Created
--  --

```

```

use work.SYSDEC_PACKAGE.all;

```

```

entity CORR_ROW is

```

```

generic (
    Symbol_Delay      : Time      := 1 ns;
    Small_Adder_Delay : Time      := 1 ns;
    Adder_Delay       : Time      := 1 ns;
    FF_Delay          : Time      := 1 ns
);
port
(
    CLOCK      : in  Bit;
    DIR_IN     : in  State;
    STATE_IN   : in  State;
    SYMBOL_A_IN : in  Symbol_Array;
    SYMBOL_B_IN : in  Symbol_Array;
    DIR_OUT    : out State;
    METRIC_OUT  : out P_Metric;
    SYMBOL_A_OUT : out Symbol_Array;
    SYMBOL_B_OUT : out Symbol_Array
);
end CORR_ROW;

```

architecture STRUCTURE of CORR_ROW is

```

component CORR_PE
    generic (
        Symbol_Delay      : Time;
        Small_Adder_Delay : Time;
        Adder_Delay       : Time;
        FF_Delay          : Time
    );
    port
    (
        CLOCK      : in  Bit;
        DIRECTION  : in  Bit;
        LAST_STATE : in  State;
        LAST_METRIC : in  P_Metric;
        RX_SYMBOL_A : in  Symbol;
        RX_SYMBOL_B : in  Symbol;
        DIRECTION_OUT : out Bit;
        STATE_OUT    : out State;
        METRIC_OUT    : out P_Metric;
        SYMBOL_A_OUT  : out Symbol;
        SYMBOL_B_OUT  : out Symbol
    );
end component;

for all: CORR_PE
    use entity work.PE_CORR (PE_CORR_BEHAVIOURAL);

    type Temp_State_Array is array (Constraint_Length-2 downto 0) of State;
    type Temp_Metric_Array is array (Constraint_Length-2 downto 0) of P_Metric;

    signal METRIC_IN      : P_Metric;
    signal TEMP_STATE     : Temp_State_Array;
    signal TEMP_METRIC     : Temp_Metric_Array;

begin

    HARDWIRED_INPUTS:
    process
    begin
        METRIC_IN <= 0;
        wait; -- forever
    end process;

    PE0:
        CORR_PE
            generic map
            (
                Symbol_Delay      => Symbol_Delay,
                Small_Adder_Delay => Small_Adder_Delay,

```

```

        Adder_Delay      => Adder_Delay,
        FF_Delay         => FF_Delay
    )
    port map
    (
        CLOCK             => CLOCK,
        DIRECTION         => DIR_IN(0),
        LAST_STATE        => STATE_IN,
        LAST_METRIC       => METRIC_IN,
        RX_SYMBOL_A       => SYMBOL_A_IN(0),
        RX_SYMBOL_B       => SYMBOL_B_IN(0),
        DIRECTION_OUT     => DIR_OUT(0),
        STATE_OUT         => TEMP_STATE(0),
        METRIC_OUT        => TEMP_METRIC(0),
        SYMBOL_A_OUT      => SYMBOL_A_OUT(0),
        SYMBOL_B_OUT      => SYMBOL_B_OUT(0)
    );
PE1:
    for i in 1 to Constraint_Length-3 generate
        PEn:
            CORR_PE
            generic map
            (
                Symbol_Delay      => Symbol_Delay,
                Small_Adder_Delay => Small_Adder_Delay,
                Adder_Delay       => Adder_Delay,
                FF_Delay          => FF_Delay
            )
            port map
            (
                CLOCK             => CLOCK,
                DIRECTION         => DIR_IN(i),
                LAST_STATE        => TEMP_STATE(i-1),
                LAST_METRIC       => TEMP_METRIC(i-1),
                RX_SYMBOL_A       => SYMBOL_A_IN(i),
                RX_SYMBOL_B       => SYMBOL_B_IN(i),
                DIRECTION_OUT     => DIR_OUT(i),
                STATE_OUT         => TEMP_STATE(i),
                METRIC_OUT        => TEMP_METRIC(i),
                SYMBOL_A_OUT      => SYMBOL_A_OUT(i),
                SYMBOL_B_OUT      => SYMBOL_B_OUT(i)
            );
        end generate;
    end generate;
PE2:
    if Constraint_Length > 2 generate
        Pef:
            CORR_PE
            generic map
            (
                Symbol_Delay      => Symbol_Delay,
                Small_Adder_Delay => Small_Adder_Delay,
                Adder_Delay       => Adder_Delay,
                FF_Delay          => FF_Delay
            )
            port map
            (
                CLOCK             => CLOCK,
                DIRECTION         => DIR_IN(Constraint_Length-2),
                LAST_STATE        => TEMP_STATE(Constraint_Length-3),
                LAST_METRIC       => TEMP_METRIC(Constraint_Length-3),
                RX_SYMBOL_A       => SYMBOL_A_IN(Constraint_Length-2),
                RX_SYMBOL_B       => SYMBOL_B_IN(Constraint_Length-2),
                DIRECTION_OUT     => DIR_OUT(Constraint_Length-2),
                STATE_OUT         => TEMP_STATE(Constraint_Length-2),
                METRIC_OUT        => METRIC_OUT,
                SYMBOL_A_OUT      => SYMBOL_A_OUT(Constraint_Length-2),
                SYMBOL_B_OUT      => SYMBOL_B_OUT(Constraint_Length-2)
            );
    end generate;

```

```

        end generate;
    end STRUCTURE;

```

D.4 The Data-Extraction Unit

```

-- ++
-- facility:
--
--     DATA_EXTRACTER
--
-- abstract:
--
--     This unit monitors the output of the last decoder stage and
--     compares all the outputs to determine the best state metric
--     and hence the best start-state.
--
-- author:
--
--     Lance G. Wright, P.Eng.
--
-- modification history:
--
--     12/08/90 : Created
--
--

use work.SYSDEC_PACKAGE.all;

entity DATA_EXTRACTER is
    generic (
        Compare_Delay      : Time      := 1 ns;
        FF_Delay           : Time      := 1 ns;
    );
    port (
        FRAME_CLK           : in  Bit;
        CHIP_CLK            : in  Bit;
        EXTRACT_ENABLE      : in  Bit;
        BEST_METRICS        : in  S Metric;
        BEST_ROWS           : in  State;
        DATA_OUT           : out Bit;
    );
end DATA_EXTRACTER;

architecture EXTRACT_BEHAVIOURAL of DATA_EXTRACTER is

    signal FRAME_COUNT      : Integer := 0;
    signal FRAME_RESET      : Bit;
    signal SURVIVOR         : State;
    signal TEMP_METRIC      : S Metric;
    signal TEMP_ROW         : State;

begin

    RESET:    -- rising-edge triggered
    process
    begin
        wait until EXTRACT_ENABLE'event and EXTRACT_ENABLE '1';
        SURVIVOR <= BEST_ROWS;
        FRAME_RESET <= '1', '0' after Frame_Clock_Period;
    end process;

    CHIP_CLOCK:    -- positive-edge triggered
    process

```

```

begin
  wait until CHIP_CLK'event and CHIP_CLK='1';
  if EXTRACT_ENABLE='1' then
    TEMP_METRIC <= BEST_METRICS;
    TEMP_ROW    <= BEST_ROWS;
  else
    if BEST_METRICS < TEMP_METRIC then
      TEMP_METRIC <= BEST_METRICS;
      TEMP_ROW    <= BEST_ROWS;
    end if;
  end if;
end process;

FRAME_CLOCK:    -- positive-edge triggered
process
begin
  wait until FRAME_CLK'event and FRAME_CLK='1';
  if FRAME_RESET='1' or FRAME_COUNT > Constraint_Length-2 then
    DATA_OUT <= SURVIVOR(0);
    FRAME_COUNT <= 1;
  else
    DATA_OUT <= SURVIVOR(FRAME_COUNT);
    FRAME_COUNT <= FRAME_COUNT + 1;
  end if;
end process;
end EXTRACT_BEHAVIOURAL;

```

D.5 One Complete Decoder Stage

```

-- ++
-- facility:
--
--     DECODER_STAGE
--
-- abstract:
--
--     One full decoder stage of the SYSDEC design.
--
-- author:
--
--     Lance G. Wright, P.Eng.
--
-- modification history:
--
--     01/08/90 : Created
--
--

```

```

use work.SYSDEC_PACKAGE.all;

```

```

entity DECODER_STAGE is

```

```

  generic (
    FF_Delay           : Time      := 1 ns;
    Symbol_Delay       : Time      := 1 ns;
    Small_Adder_Delay  : Time      := 1 ns;
    Adder_Delay        : Time      := 1 ns;
    Big_Adder_Delay    : Time      := 1 ns;
    Multiplexor_Delay  : Time      := 1 ns;
    Subtractor_Delay   : Time      := 1 ns;
    Comparator_Delay   : Time      := 1 ns;
  );

```

```

  port
  (
    CLOCK              : in Bit;
    RESET              : in Bit;

```

```

    LATCH_ENABLE      : in Bit;
    NORMALIZE         : in Bit;
    BEST_ROWS_IN      : in State;
    BEST_METRICS_IN   : in S_Metric;
    SYMBOL_A_IN       : in Symbol_Array;
    SYMBOL_B_IN       : in Symbol_Array;
    BEST_ROWS_OUT     : out State;
    BEST_METRICS_OUT  : out S_Metric
  );
end DECODER_STAGE;

```

architecture STRUCTURE of DECODER_STAGE is

```

  component TRELLIS_DIRECTION
  generic (
    FF_Delay          : Time
  );
  port
  (
    CLOCK             : in Bit;
    RESET             : in Bit;
    DIRECTION_BIT     : inout State
  );
  end component;

  component CORRELATION_ARRAY
  generic (
    Symbol_Delay      : Time;
    Small_Adder_Delay : Time;
    Adder_Delay       : Time;
    FF_Delay          : Time
  );
  port
  (
    CLOCK             : in Bit;
    DIR_IN            : in State;
    SYMBOL_A_IN       : in Symbol_Array;
    SYMBOL_B_IN       : in Symbol_Array;
    METRIC_OUT        : out P_Metric_Array
  );
  end component;

  component COMPARISON_ARRAY
  generic (
    Comparator_Delay  : Time;
    Multiplexor_Delay : Time;
    Big_Adder_Delay   : Time;
    FF_Delay          : Time
  );
  port
  (
    CLOCK             : in Bit;
    STATE_METRIC      : in S_Metric_Array;
    BEST_ROW          : in State_Array;
    PATH_METRIC       : in P_Metric_Array;
    BEST_ROW_OUT      : out State;
    BEST_METRIC_OUT   : out S_Metric
  );
  end component;

  component METRIC_LATCH_NORM_ARRAY
  generic (
    Subtractor_Delay  : Time;
    FF_Delay          : Time
  );
  port
  (
    CLOCK             : in Bit;
    LATCH_ENABLE_IN   : in Bit;
    NORMALIZE_IN      : in Bit;
    BEST_ROW_IN       : in State;
    BEST_METRIC_IN    : in S_Metric;
  );

```

```

        BEST_ROW_OUT      : out State_Array;
        BEST_METRIC_OUT   : out S_Metric_Array;
    );
end component;

for all: TRELLIS_DIRECTION
    use entity work.DIRECTION_COUNTER(COUNTER_BEHAVIOURAL);

for all: CORRELATION_ARRAY
    use entity work.CORR_ARRAY(STRUCTURE);

for all: COMPARISON_ARRAY
    use entity work.COMP_ARRAY(STRUCTURE);

for all: METRIC_LATCH_NORM_ARRAY
    use entity work.LATCH_ARRAY(STRUCTURE);

signal DIR                : State;
signal PATH_METRICS       : P_Metric_Array;
signal STATE_METRICS      : S_Metric_Array;
signal BEST_ROWS          : State_Array;

begin

TRELLIS_DIRECTION_COUNTER:
    TRELLIS_DIRECTION
    generic map
    (
        FF_Delay      => FF_Delay
    )
    port map
    (
        CLOCK          => CLOCK,
        RESET          => RESET,
        DIRECTION_BIT  => DIR
    );

CORRELATION:
    CORRELATION_ARRAY
    generic map
    (
        Symbol_Delay      => Symbol_Delay,
        Small_Adder_Delay => Small_Adder_Delay,
        Adder_Delay       => Adder_Delay,
        FF_Delay          => FF_Delay
    )
    port map
    (
        CLOCK            => CLOCK,
        DIR_IN           => DIR,
        SYMBOL_A_IN      => SYMBOL_A_IN,
        SYMBOL_B_IN      => SYMBOL_B_IN,
        METRIC_OUT       => PATH_METRICS
    );

COMPARISON:
    COMPARISON_ARRAY
    generic map
    (
        Comparator_Delay  => Comparator_Delay,
        Multiplexor_Delay => Multiplexor_Delay,
        Big_Adder_Delay   => Big_Adder_Delay,
        FF_Delay          => FF_Delay
    )
    port map
    (
        CLOCK            => CLOCK,
        STATE_METRIC     => STATE_METRICS,
        BEST_ROW         => BEST_ROWS,
    );

```

```

        PATH_METRIC      => PATH_METRICS,
        BEST_ROW_OUT     => BEST_ROWS_OUT,
        BEST_METRIC_OUT  => BEST_METRICS_OUT
    );

LATCH:
    METRIC_LATCH_NORM_ARRAY
    generic map
    (
        Subtractor_Delay => Subtractor_Delay,
        FF_Delay         => FF_Delay
    )
    port map
    (
        CLOCK            => CLOCK,
        LATCH_ENABLE_IN  => LATCH_ENABLE,
        NORMALIZE_IN     => NORMALIZE,
        BEST_ROW_IN      => BEST_ROWS_IN,
        BEST_METRIC_IN   => BEST_METRICS_IN,
        BEST_ROW_OUT     => BEST_ROWS,
        BEST_METRIC_OUT  => STATE_METRICS
    );

end STRUCTURE;

```

D.6 The Trellis-Direction Counter

```

-- ++
-- facility:
--
--     DIRECTION_COUNTER
--
-- abstract:
--
--     The trellis direction counter for one decoder stage.
--
-- author:
--
--     Lance G. Wright, P.Eng.
--
-- modification history:
--
--     02/08/90 : Created
--
--

```

```
use work.SYSDEC_PACKAGE.all;
```

```

entity DIRECTION_COUNTER is
    generic (
        FF_Delay      : Time      := 1 ns
    );
    port
    (
        CLOCK          : in      Bit;
        RESET          : in      Bit;
        DIRECTION_BIT   : inout   State
    );
end DIRECTION_COUNTER;

```

```
architecture COUNTER_BEHAVIOURAL of DIRECTION_COUNTER is
```

```

    signal RETIME_BIT5 : Bit_Vector (4 downto 0);
    signal RETIME_BIT4 : Bit_Vector (3 downto 0);
    signal RETIME_BIT3 : Bit_Vector (2 downto 0);

```



```

signal RETIME_BIT2 : Bit_Vector (1 downto 0);
signal RETIME_BIT1 : Bit_Vector (0 downto 0);
signal COUNTER_STATE : State := int_to_bit(0);

begin

  COUNTERS: -- negative-edge triggered
  process
    variable BINARY_ONE : State := int_to_bit(1);
  begin
    wait until CLOCK'event and CLOCK='0';
    if RESET='1' then
      COUNTER_STATE <= int_to_bit(0);
    else
      COUNTER_STATE <= COUNTER_STATE + BINARY_ONE;
    end if;
  end process COUNTERS;

  COL45:
    if Constraint_Length >= 7 generate
      COLUMNS_4_AND_5: -- positive-edge triggered
      process
        begin
          wait until CLOCK'event and CLOCK='1';
-- column 5
          DIRECTION_BIT(5) <= RETIME_BIT5(4) after FF_Delay;
          RETIME_BIT5(4) <= RETIME_BIT5(3) after FF_Delay;
          RETIME_BIT5(3) <= RETIME_BIT5(2) after FF_Delay;
          RETIME_BIT5(2) <= RETIME_BIT5(1) after FF_Delay;
          RETIME_BIT5(1) <= RETIME_BIT5(0) after FF_Delay;
          RETIME_BIT5(0) <= COUNTER_STATE(Constraint_Length-7)
            after FF_Delay;
-- column 4
          DIRECTION_BIT(4) <= RETIME_BIT4(3) after FF_Delay;
          RETIME_BIT4(3) <= RETIME_BIT4(2) after FF_Delay;
          RETIME_BIT4(2) <= RETIME_BIT4(1) after FF_Delay;
          RETIME_BIT4(1) <= RETIME_BIT4(0) after FF_Delay;
          RETIME_BIT4(0) <= COUNTER_STATE(Constraint_Length-6)
            after FF_Delay;
        end process COLUMNS_4_AND_5;
      end generate;

  COL23:
    if Constraint_Length >= 5 generate
      COLUMNS_2_AND_3: -- positive-edge triggered
      process
        begin
          wait until CLOCK'event and CLOCK='1';
-- column 3
          DIRECTION_BIT(3) <= RETIME_BIT3(2) after FF_Delay;
          RETIME_BIT3(2) <= RETIME_BIT3(1) after FF_Delay;
          RETIME_BIT3(1) <= RETIME_BIT3(0) after FF_Delay;
          RETIME_BIT3(0) <= COUNTER_STATE(Constraint_Length-5)
            after FF_Delay;
-- column 2
          DIRECTION_BIT(2) <= RETIME_BIT2(1) after FF_Delay;
          RETIME_BIT2(1) <= RETIME_BIT2(0) after FF_Delay;
          RETIME_BIT2(0) <= COUNTER_STATE(Constraint_Length-4)
            after FF_Delay;
        end process COLUMNS_2_AND_3;
      end generate;

  COLUMNS_0_AND_1: -- positive-edge triggered
  process
    begin
      wait until CLOCK'event and CLOCK='1';
-- column 1
      DIRECTION_BIT(1) <= RETIME_BIT1(0) after FF_Delay;
      RETIME_BIT1(0) <= COUNTER_STATE(Constraint_Length-3)

```

```

                                after FF_Delay;
-- column 0
    DIRECTION_BIT(0) <= COUNTER_STATE(Constraint Length-2)
                                after FF_Delay;
    end process COLUMNS_0_AND_1;

end COUNTER_BEHAVIOURAL;

```

D.7 The Metric-Latch Array

```

-- ++
-- facility:
--
--     LATCH_ARRAY
--
-- abstract:
--
--     The metric-latch/normalization array for one decoder stage.
--
-- author:
--
--     Lance G. Wright, P.Eng.
--
-- modification history:
--
--     03/08/90 : Created
--
--

```

```

use work.SYSDEC_PACKAGE.all;

```

```

entity LATCH_ARRAY is
  generic (
    Subtractor_Delay : Time := 1 ns;
    FF_Delay         : Time := 1 ns;
  );
  port (
    CLOCK           : in Bit;
    LATCH_ENABLE_IN : in Bit;
    NORMALIZE_IN    : in Bit;
    BEST_ROW_IN     : in State;
    BEST_METRIC_IN  : in S_Metric;
    BEST_ROW_OUT    : out State_Array;
    BEST_METRIC_OUT : out S_Metric_Array;
  );
end LATCH_ARRAY;

```

architecture STRUCTURE of LATCH_ARRAY is

```

  component LATCH_PE
    generic (
      Subtractor_Delay : Time;
      FF_Delay         : Time;
    );
    port (
      CLOCK           : in Bit;
      LATCH_ENABLE_IN : in Bit;
      NORMALIZE_IN    : in Bit;
      BEST_ROW_IN     : in State;
      BEST_METRIC_IN  : in S_Metric;
      LATCH_ENABLE_OUT : out Bit;
      NORMALIZE_OUT   : inout Bit;
      BEST_ROW_OUT    : out State;
      BEST_METRIC_OUT : out S_Metric;
    );
  end component LATCH_PE;

```

```

    );
end component;

for all: LATCH_PE use entity work.PE_LATCH(PE_LATCH_BEHAVIOURAL);

signal LATCH_ENABLE_OUT : Bit;
signal NORMALIZE_OUT    : Bit;
signal TEMP_LATCH       : Bit_Vector (State_Count-2 downto 0);
signal TEMP_NORM        : Bit_Vector (State_Count-2 downto 0);

begin

PE0:
  LATCH_PE
    generic map
      (
        Subtractor_Delay => Subtractor_Delay,
        FF_Delay         => FF_Delay
      )
    port map
      (
        CLOCK           => CLOCK,
        LATCH_ENABLE_IN => LATCH_ENABLE_IN,
        NORMALIZE_IN    => NORMALIZE_IN,
        BEST_ROW_IN     => BEST_ROW_IN,
        BEST_METRIC_IN  => BEST_METRIC_IN,
        LATCH_ENABLE_OUT => TEMP_LATCH(0),
        NORMALIZE_OUT   => TEMP_NORM(0),
        BEST_ROW_OUT    => BEST_ROW_OUT(0),
        BEST_METRIC_OUT => BEST_METRIC_OUT(0)
      );

PE1:
  for i in 1 to State_Count-2 generate
    PEn:
      LATCH_PE
        generic map
          (
            Subtractor_Delay => Subtractor_Delay,
            FF_Delay         => FF_Delay
          )
        port map
          (
            CLOCK           => CLOCK,
            LATCH_ENABLE_IN => TEMP_LATCH(i-1),
            NORMALIZE_IN    => TEMP_NORM(i-1),
            BEST_ROW_IN     => BEST_ROW_IN,
            BEST_METRIC_IN  => BEST_METRIC_IN,
            LATCH_ENABLE_OUT => TEMP_LATCH(i),
            NORMALIZE_OUT   => TEMP_NORM(i),
            BEST_ROW_OUT    => BEST_ROW_OUT(i),
            BEST_METRIC_OUT => BEST_METRIC_OUT(i)
          );
      end generate;

PE2:
  if State_Count > 1 generate
    Pef:
      LATCH_PE
        generic map
          (
            Subtractor_Delay => Subtractor_Delay,
            FF_Delay         => FF_Delay
          )
        port map
          (
            CLOCK           => CLOCK,
            LATCH_ENABLE_IN => TEMP_LATCH(State_Count-2),
            NORMALIZE_IN    => TEMP_NORM(State_Count-2),

```

```

        BEST_ROW_IN      => BEST_ROW_IN,
        BEST_METRIC_IN   => BEST_METRIC_IN,
        LATCH_ENABLE_OUT => LATCH_ENABLE_OUT,
        NORMALIZE_OUT    => NORMALIZE_OUT,
        BEST_ROW_OUT     => BEST_ROW_OUT(State_Count-1),
        BEST_METRIC_OUT  => BEST_METRIC_OUT(State_Count-1)
    );
    end generate;
end STRUCTURE;

```

D.8 The Normalization Detector

```

-- ++
-- /facility:
--
--     NORMALIZE_DETECTOR
--
-- abstract:
--
--     This unit detects if state-metric normalization is necessary.
--
-- author:
--
--     Lance G. Wright, P.Eng.
--
-- modification history:
--
--     11/08/90 : Created
--
--

```

```

use work.SYSDEC_PACKAGE.all;

```

```

entity NORMALIZE_DETECTOR is
    port
        (
            CLOCK           : in  Bit;
            START_DETECT    : in  Bit;
            METRIC_IN       : in  S_Metric;
            NORMALIZE       : out Bit
        );
end NORMALIZE_DETECTOR;

```

```

architecture DETECT_BEHAVIOURAL of NORMALIZE_DETECTOR is

```

```

    signal    NORM_COUNT    : Integer    := 0;

```

```

begin

```

```

    RESET:
        process
        begin
            wait on START_DETECT;
            if NORM_COUNT = State_Count then
                NORMALIZE <= '1' after Chip_Clock_Period/2,
                           '0' after 3*Chip_Clock_Period/2;
            end if;
        end process;

```

```

    CHIP_CLOCK:
        process    -- positive-edge triggered
        begin
            wait until CLOCK'event and CLOCK='1';
            if METRIC_IN >= Normalize_Value then

```



```

        if START_DETECT='1' then
            NORM_COUNT <= 1;
        else
            NORM_COUNT <= NORM_COUNT + 1;
        end if;
    end if;
end process;

end DETECT_BEHAVIOURAL;

```

D.9 The Comparison Array Processing Element

```

-- ++
-- facility:
--
--     PE_COMP
--
-- abstract:
--
--     Represents one processing element of the comparison-array of the
--     SYSDEC design.
--
-- author:
--
--     Lance G. Wright, P.Eng.
--
-- modification history:
--
--     03/08/90 : Created
--
--

use work.SYSDEC_PACKAGE.all;

entity PE_COMP is
    generic (
        Comparator_Delay : Time := 1 ns;
        Multiplexor_Delay : Time := 1 ns;
        Big_Adder_Delay : Time := 1 ns;
        FF_Delay : Time := 1 ns
    );
    port (
        CLOCK : in Bit;
        STATE_METRIC : in S_Metric;
        BEST_ROW : in State;
        PATH_METRIC : in P_Metric;
        BEST_ROW_IN : in State;
        BEST_METRIC_IN : in S_Metric;
        BEST_ROW_OUT : out State;
        BEST_METRIC_OUT : out S_Metric
    );
end PE_COMP;

architecture PE_COMP_BEHAVIOURAL of PE_COMP is

    signal LATCHED_METRIC : S_Metric;
    signal LATCHED_ROW : State;
    signal NEW_METRIC : S_Metric;
    signal MUXSEL : Bit;

begin

    METRIC_ADDER:
    process

```

```

        variable TMP_METRIC : Integer := 0;
begin
    TMP_METRIC := PATH_METRIC;
    TMP_METRIC := TMP_METRIC + STATE_METRIC;
    if TMP_METRIC > Max_Reg
    then NEW_METRIC <= Max_Reg after Big_Adder_Delay;
    else NEW_METRIC <= TMP_METRIC after Big_Adder_Delay;
    end if;
    wait on PATH_METRIC, STATE_METRIC;
end process;

COMPARATOR:
process
begin
    if NEW_METRIC < LATCHED_METRIC
    then MUXSEL <= '1' after Comparator_Delay;
    else MUXSEL <= '0' after Comparator_Delay;
    end if;
    wait on NEW_METRIC, LATCHED_METRIC;
end process;

MULTIPLEXORS:
process
begin
    if MUXSEL='1' then
        BEST_METRIC_OUT <= NEW_METRIC      after Multiplexor_Delay;
        BEST_ROW_OUT   <= BEST_ROW        after Multiplexor_Delay;
    else
        BEST_METRIC_OUT <= LATCHED_METRIC after Multiplexor_Delay;
        BEST_ROW_OUT   <= LATCHED_ROW    after Multiplexor_Delay;
    end if;
    wait on MUXSEL,
           NEW_METRIC, LATCHED_METRIC,
           BEST_ROW, LATCHED_ROW;
end process;

INPUT_REGISTERS:  -- positive-edge triggered
process
begin
    wait until CLOCK'event and CLOCK='1';
    LATCHED_METRIC <= BEST_METRIC_IN after FF_Delay;
    LATCHED_ROW   <= BEST_ROW_IN   after FF_Delay;
end process;

end PE_COMP_BEHAVIOURAL;

```

D.10 The Correlation Array Processing Element

```

-- ++
-- facility:
--
--     PE_CORR
--
-- abstract:
--
--     Represents one processing element of the correlation-array of the
--     SYSDEC design.
--
-- author:
--
--     Lance G. Wright, P.Eng.
--
-- modification history:
--
--     30/07/90 : Created
--
--

```

```

use work.SYSDEC_PACKAGE.all;

entity PE_CORR is
  generic (
    Symbol_Delay      : Time := 1 ns;
    Small_Adder_Delay : Time := 1 ns;
    Adder_Delay       : Time := 1 ns;
    FF_Delay          : Time := 1 ns
  );
  port (
    CLOCK      : in Bit;
    DIRECTION  : in Bit;
    LAST_STATE : in State;
    LAST_METRIC : in P_Metric;
    RX_SYMBOL_A : in Symbol;
    RX_SYMBOL_B : in Symbol;
    DIRECTION_OUT : out Bit;
    STATE_OUT : out State;
    METRIC_OUT : out P_Metric;
    SYMBOL_A_OUT : out Symbol;
    SYMBOL_B_OUT : out Symbol
  );
end PE_CORR;

architecture PE_CORR_BEHAVIOURAL of PE_CORR is

  signal EXPECTED_A : Symbol;
  signal EXPECTED_B : Symbol;
  signal BRANCH_METRIC : P_Metric;
  signal NEW_METRIC : P_Metric;

begin
  SYMBOL_GENERATOR:
  process
    variable TEMP_BIT : Bit;
  begin
    TEMP_BIT := LAST_STATE(5)
              xor LAST_STATE(2)
              xor LAST_STATE(1)
              xor LAST_STATE(0)
              xor DIRECTION;
    if TEMP_BIT='1'
    then EXPECTED_A <= 7 after Symbol_Delay;
    else EXPECTED_A <= 0 after Symbol_Delay;
    end if;
    TEMP_BIT := LAST_STATE(5)
              xor LAST_STATE(4)
              xor LAST_STATE(2)
              xor LAST_STATE(1)
              xor DIRECTION;
    if TEMP_BIT='1'
    then EXPECTED_B <= 7 after Symbol_Delay;
    else EXPECTED_B <= 0 after Symbol_Delay;
    end if;
    wait on LAST_STATE(5),
             LAST_STATE(4),
             LAST_STATE(2),
             LAST_STATE(1),
             LAST_STATE(0),
             DIRECTION;
  end process;

  BRANCH_METRIC_GENERATOR:
  process
    variable TEMP_METRIC_A : Symbol;

```



```

        variable TEMP_METRIC_B : Symbol;
    begin
        if EXPECTED_A > RX_SYMBOL_A
            then TEMP_METRIC_A := EXPECTED_A - RX_SYMBOL_A;
            else TEMP_METRIC_A := RX_SYMBOL_A - EXPECTED_A;
        end if;
        if EXPECTED_B > RX_SYMBOL_B
            then TEMP_METRIC_B := EXPECTED_B - RX_SYMBOL_B;
            else TEMP_METRIC_B := RX_SYMBOL_B - EXPECTED_B;
        end if;
        BRANCH_METRIC <= TEMP_METRIC_A + TEMP_METRIC_B after Small_Adder_Delay;
        wait on EXPECTED_A, EXPECTED_B, RX_SYMBOL_A, RX_SYMBOL_B;
    end process;

    ERROR_SUMMATION:
    process
    begin
        NEW_METRIC <= LAST_METRIC + BRANCH_METRIC after Adder_Delay;
        wait on BRANCH_METRIC, LAST_METRIC;
    end process;

    OUTPUT_REGISTERS:    -- positive-edge triggered
    process
    begin
        wait until CLOCK'event and CLOCK='1';
        DIRECTION_OUT <= DIRECTION      after FF_Delay;
        SYMBOL_A_OUT  <= RX_SYMBOL_A    after FF_Delay;
        SYMBOL_B_OUT  <= RX_SYMBOL_B    after FF_Delay;
        STATE_OUT(5)  <= LAST_STATE(4)  after FF_Delay;
        STATE_OUT(4)  <= LAST_STATE(3)  after FF_Delay;
        STATE_OUT(3)  <= LAST_STATE(2)  after FF_Delay;
        STATE_OUT(2)  <= LAST_STATE(1)  after FF_Delay;
        STATE_OUT(1)  <= LAST_STATE(0)  after FF_Delay;
        STATE_OUT(0)  <= DIRECTION      after FF_Delay;
        METRIC_OUT    <= NEW_METRIC      after FF_Delay;
    end process;

end PE_CORR_BEHAVIOURAL;

```

D.11 The Metric-Latch Array Processing Element

```

-- ++
-- facility:
--
--     PE_LATCH
--
-- abstract:
--
--     Represents one processing element of the metric/latch-array of the
--     SYSDEC design.
--
-- author:
--
--     Lance G. Wright, P.Eng.
--
-- modification history:
--
--     03/08/90 : Created
--
--

```

use work.SYSDEC_PACKAGE.all;

entity PE_LATCH is
generic (

```

        Subtractor_Delay : Time := 1 ns;
        FF_Delay         : Time := 1 ns
    );
port
    (
        CLOCK           : in Bit;
        LATCH_ENABLE_IN : in Bit;
        NORMALIZE_IN    : in Bit;
        BEST_ROW_IN     : in State;
        BEST_METRIC_IN   : in S_Metric;
        LATCH_ENABLE_OUT : out Bit;
        NORMALIZE_OUT    : out Bit;
        BEST_ROW_OUT     : out State;
        BEST_METRIC_OUT  : out S_Metric
    );
end PE_LATCH;

architecture PE_LATCH_BEHAVIOURAL of PE_LATCH is
    signal LATCHED_METRIC : S_Metric;
begin
    METRIC_NORMALIZER:
    process
    begin
        if NORMALIZE_IN='1' then
            BEST_METRIC_OUT <= LATCHED_METRIC - Normalize_Value
                after Subtractor_Delay;
        else
            BEST_METRIC_OUT <= LATCHED_METRIC
                after Subtractor_Delay;
        end if;
        wait on NORMALIZE_IN, LATCHED_METRIC;
    end process;

    REGISTERS: -- positive-edge triggered
    process
    begin
        wait until CLOCK'event and CLOCK='1';
        if LATCH_ENABLE_IN='1' then
            LATCHED_METRIC <= BEST_METRIC_IN after FF_Delay;
            BEST_ROW_OUT <= BEST_ROW_IN after FF_Delay;
        end if;
        LATCH_ENABLE_OUT <= LATCH_ENABLE_IN;
        NORMALIZE_OUT <= NORMALIZE_IN;
    end process;
end PE_LATCH_BEHAVIOURAL;

```

D.12 The Symbol-Shifting Unit

```

-- ++
-- facility:
--
--     SYMBOL_SHIFTER
--
-- abstract:
--
--     This unit accepts symbols at the frame-data rate, stores them,
--     and block-shifts them across the decoder stages.
--
-- author:
--
--     Lance G. Wright, P.Eng.
--

```

```

-- modification history:
--
--      11/08/90 : Created
--      --

use work.SYSDEC_PACKAGE.all;

entity SYMBOL_SHIFTER is
  generic (
    FF_Delay          : Time      := 1 ns
  );
  port
  (
    FRAME_CLK         : in  Bit;
    RESET             : in  Bit;
    RX_DATA_A         : in  Symbol;
    RX_DATA_B         : in  Symbol;
    FRAME_DATA_A      : out Shift_Array;
    FRAME_DATA_B      : out Shift_Array
  );
end SYMBOL_SHIFTER;

architecture SHIFT_BEHAVIOURAL of SYMBOL_SHIFTER is
begin
  FRAME_CLOCK:    -- positive-edge triggered
  process
    constant Initial_A_Data_0 : Symbol_Array
      := Symbol_Array'(7,7,7,0,7,0);
    constant Initial_B_Data_0 : Symbol_Array
      := Symbol_Array'(7,0,0,7,0,0);
    constant Initial_A_Data_Nil : Symbol_Array
      := Symbol_Array'(0,0,0,0,0,0);
    constant Initial_B_Data_Nil : Symbol_Array
      := Symbol_Array'(0,0,0,0,0,0);
    variable FRAME_COUNT      : Integer := 0;
    variable TEMP_DATA_A      : Shift_Array;
    variable TEMP_DATA_B      : Shift_Array;
    variable STAGE_A          : Symbol_Array;
    variable STAGE_B          : Symbol_Array;
    variable FIRST_TIME       : Integer := 0;
  begin
    -- first time through ?
    -- initialize the data for a starting-state of 63, proceeding to 0
    if FIRST_TIME=0 then
      FRAME_DATA_A(0) <= Initial_A_Data_0;
      FRAME_DATA_B(0) <= Initial_B_Data_0;
      for i in 1 to Stages-1 loop
        FRAME_DATA_A(i) <= Initial_A_Data_Nil;
        FRAME_DATA_B(i) <= Initial_B_Data_Nil;
      end loop;
      FIRST_TIME := 1;
    end if;
    -- end of initialization
    wait until FRAME_CLK'event and FRAME_CLK='1';
    if RESET='1' then
      FRAME_COUNT := 0;
    end if;
    -- progress monitor
    assert not (FRAME_COUNT = Constraint_Length-1)
      report "End of One densely connected trellis stage."
        severity Note;
    if FRAME_COUNT=Constraint_Length-1 then
      FRAME_COUNT := 0;
    end if;
    -- symbol re-timing
    TEMP_DATA_A(Stages-1) := STAGE_A;
  end process;
end architecture;

```

```

TEMP_DATA_B(Stages-1) := STAGE_B;
for I in 0 to Stages-1 loop
  STAGE_A := TEMP_DATA_A(I);
  STAGE_B := TEMP_DATA_B(I);
  for j in 0 to Constraint_Length-2 loop
    FRAME_DATA_A(I)(j) <= STAGE_A(j) after
      FF_Delay +
      j * Chip_Clock_Period;
    FRAME_DATA_B(I)(j) <= STAGE_B(j) after
      FF_Delay +
      j * Chip_Clock_Period;
  end loop;
end loop;
-- block shifting
for i in 0 to Stages-2 loop
  TEMP_DATA_A(i) := TEMP_DATA_A(i+1);
  TEMP_DATA_B(i) := TEMP_DATA_B(i+1);
end loop;
end if;
-- data storage
STAGE_A(FRAME_COUNT) := RX_DATA_A;
STAGE_B(FRAME_COUNT) := RX_DATA_B;
FRAME_COUNT := FRAME_COUNT + 1;
end process;

end SHIFT_BEHAVIOURAL;

```

D.13 The SYSDEC Package

```

-- ++
-- facility:
--
--     SYSDEC_PACKAGE
--
-- abstract:
--
--     Package containing the data type declarations and constants.
--
-- author:
--
--     Lance G. Wright, P.Eng.
--
-- modification history:
--
--     30/07/90 : Created
--
--
package SYSDEC_PACKAGE is

  constant Constraint_Length : Integer := 7;
  constant Stages            : Integer := 5;
  constant Stages            : Integer := 2;
  constant State_Count       : Integer := 2**(Constraint_Length-1);
  constant Normalize_Value   : Integer := 128;
  constant Max_Bits          : Integer := 9;
  constant Max_Reg           : Integer := 2**(Max_Bits)-1;
  constant Chip_Clock_Period : Time    := 15625000 fs; -- 64 MHz
  constant Frame_Clock_Period : Time    := (State_Count*Chip_Clock_Period)
                                           / (Constraint_Length-1);

  subtype State is Bit_Vector (Constraint_Length-2 downto 0);
  subtype S_Metric is Integer range 0 to Max_Reg;
  subtype P_Metric is Integer range 0 to 84;
  subtype B_Metric is Integer range 0 to 14;
  subtype Symbol is Integer range 0 to 7;

```

```

type      State_Array      is array (State_Count-1 downto 0) of State;
type      P_Metric_Array   is array (State_Count-1 downto 0) of P_Metric;
type      S_Metric_Array   is array (State_Count-1 downto 0) of S_Metric;
type      Symbol_Array     is array (Constraint_Length-2 downto 0) of Symbol;
type      Shift_Array      is array (Stages-1 downto 0) of Symbol Array;

function "+" ( X , Y : Bit_Vector )
    return Bit_Vector;

function int_to_bit ( X : Integer )
    return Bit_Vector;

end SYSDEC_PACKAGE;

package body SYSDEC_PACKAGE is

function "+" ( X , Y : Bit_Vector )
    return Bit_Vector is
    variable SUM      : Bit_Vector (X'range);
    variable CARRY    : Bit      := '0';
begin
    for i in X'low to X'high loop
        SUM(i) := CARRY xor X(i) xor Y(i);
        CARRY := ( CARRY and X(i) )
            or ( CARRY and Y(i) )
            or ( X(i) and Y(i) );
    end loop;
    return SUM;
end "+";

function int_to_bit ( X : Integer )
    return Bit_Vector is
    variable BINARY_BITS : State;
    variable TEMP1       : Integer := 0;
    variable TEMP2       : Integer := 0;
begin
    TEMP2 := X;
    for i in 0 to Constraint_Length-2 loop
        TEMP1 := TEMP2 / 2;
        if TEMP2 /= (TEMP1 * 2)
            then BINARY_BITS(i) := '1';
            else BINARY_BITS(i) := '0';
        end if;
        TEMP2 := TEMP1;
    end loop;
    return BINARY_BITS;
end int_to_bit;

end SYSDEC_PACKAGE;

```

D.14 The Test Bench

```

-- ++
-- facility:
--     TEST_BENCH
--
-- abstract:
--     Test-bench for exercising the SYSDEC design.
--
-- author:

```

```

--
--      Lance G. Wright, P.Eng.
--
--      modification history:
--
--      01/08/90 : Created
--
--

use std.simulator_standard.all;
use work.SYSDEC_PACKAGE.all;

entity TEST_BENCH is
  generic (
    Max_Frame_Clocks : Integer := 10
  );
end TEST_BENCH;

architecture TEST of TEST_BENCH is

  component ONE_STAGE_DECODER
    generic (
      FF_Delay           : Time;
      Symbol_Delay       : Time;
      Small_Adder_Delay  : Time;
      Adder_Delay        : Time;
      Big_Adder_Delay    : Time;
      Multiplexor_Delay  : Time;
      Subtractor_Delay   : Time;
      Comparator_Delay   : Time;
    );
    port
      (
        CLOCK           : in Bit;
        RESET           : in Bit;
        LATCH_ENABLE    : in Bit;
        NORMALIZE        : in Bit;
        BEST_ROWS_IN    : in State;
        BEST_METRICS_IN  : in S_Metric;
        SYMBOL_A_IN      : in Symbol_Array;
        SYMBOL_B_IN      : in Symbol_Array;
        BEST_ROWS_OUT    : out State;
        BEST_METRICS_OUT : out S_Metric;
      );
  end component;

  component CONTROL_UNIT
    port
      (
        GLOBAL_RESET : in Bit;
        FRAME_CLK    : in Bit;
        CHIP_CLK     : inout Bit;
        LATCH_ENABLE : out Bit;
        COUNTER_RESET : out Bit;
        SHIFTER_RESET : out Bit;
        START_DETECT  : out Bit;
        EXTRACT_ENABLE : out Bit;
      );
  end component;

  component DETECTOR_NORMALIZE
    port
      (
        CLOCK           : in Bit;
        START_DETECT    : in Bit;
        METRIC_IN        : in S_Metric;
        NORMALIZE        : out Bit;
      );
  end component;

```

```

component SHIFTER_SYMBOL
  generic (
    FF_Delay      : Time
  );
  port (
    FRAME_CLK      : in Bit;
    RESET          : in Bit;
    RX_DATA_A      : in Symbol;
    RX_DATA_B      : in Symbol;
    FRAME_DATA_A    : out Shift_Array;
    FRAME_DATA_B    : out Shift_Array
  );
end component;

component EXTRACTER_DATA
  generic (
    Compare_Delay  : Time;
    FF_Delay       : Time
  );
  port (
    FRAME_CLK      : in Bit;
    CHIP_CLK       : in Bit;
    EXTRACT_ENABLE : in Bit;
    BEST_METRICS   : in S_Metric;
    BEST_ROWS      : in State;
    DATA_OUT      : out Bit
  );
end component;

for all: EXTRACTER_DATA
  use entity work.DATA_EXTRACTER(EXTRACT_BEHAVIOURAL);

for all: SHIFTER_SYMBOL
  use entity work.SYMBOL_SHIFTER(SHIFT_BEHAVIOURAL);

for all: DETECTOR_NORMALIZE
  use entity work.NORMALIZE_DETECTOR(DETECT_BEHAVIOURAL);

for all: CONTROL_UNIT
  use entity work.TIMING_CONTROLLER(TIMING_BEHAVIOURAL);

for all: ONE_STAGE_DECODER
  use entity work.DECODER_STAGE(STRUCTURE);

constant FF_Delay      : Time      := 900 ps;
constant Small_Adder_Delay : Time   := 3200 ps;
constant Adder_Delay    : Time      := 6500 ps;
constant Big_Adder_Delay : Time      := 8100 ps;
constant Symbol_Delay   : Time      := 1500 ps;
constant Multiplexor_Delay : Time    := 800 ps;
constant Subtractor_Delay : Time     := 1600 ps;
constant Comparator_Delay : Time     := 3999 ps;

type Best_Metric_Array is array (Stages-1 downto 0) of S_Metric;
type Best_Row_Array   is array (Stages-1 downto 0) of State;

signal FRAME_CLOCK      : Bit;
signal GLOBAL_RESET     : Bit;
signal RX_DATA_A        : Symbol;
signal RX_DATA_B        : Symbol;
signal DATA_OUT        : Bit;

signal CHIP_CLOCK       : Bit;
signal DIR_IN           : State;
signal BEST_METRICS_IN  : Best_Metric_Array;
signal BEST_METRICS_OUT : Best_Metric_Array;
signal BEST_ROWS_IN     : Best_Row_Array;

```

```

signal BEST_ROWS_OUT      : Best_Row_Array;
signal LATCH_ENABLE       : Bit;
signal COUNTER_RESET      : Bit;
signal SHIFTER_RESET      : Bit;
signal EXTRACT_ENABLE     : Bit;
signal START_DETECT       : Bit;
signal NORMALIZE          : Bit;
signal FRAME_DATA_A       : Shift_Array;
signal FRAME_DATA_B       : Shift_Array;

signal FRAME_CLOCK_TICKS : Integer := 0;
signal ZERO_BIT          : Bit := '0';

begin

  TIMING_UNIT:
    CONTROL_UNIT
      port map
      (
        GLOBAL_RESET => GLOBAL_RESET,
        FRAME_CLK    => FRAME_CLOCK,
        CHIP_CLK     => CHIP_CLOCK,
        LATCH_ENABLE => LATCH_ENABLE,
        COUNTER_RESET => COUNTER_RESET,
        SHIFTER_RESET => SHIFTER_RESET,
        START_DETECT  => START_DETECT,
        EXTRACT_ENABLE => EXTRACT_ENABLE
      );

  RECEIVED_SYMBOLS:
    SHIFTER_SYMBOL
      generic map
      (
        FF_Delay => FF_Delay
      )
      port map
      (
        FRAME_CLK    => FRAME_CLOCK,
        RESET        => SHIFTER_RESET,
        RX_DATA_A    => RX_DATA_A,
        RX_DATA_B    => RX_DATA_B,
        FRAME_DATA_A => FRAME_DATA_A,
        FRAME_DATA_B => FRAME_DATA_B
      );

  DCO:
    ONE_STAGE_DECODER
      generic map
      (
        FF_Delay           => FF_Delay,
        Symbol_Delay       => Symbol_Delay,
        Small_Adder_Delay  => Small_Adder_Delay,
        Adder_Delay        => Adder_Delay,
        Big_Adder_Delay    => Big_Adder_Delay,
        Multiplexor_Delay  => Multiplexor_Delay,
        Subtractor_Delay   => Subtractor_Delay,
        Comparator_Delay   => Comparator_Delay
      )
      port map
      (
        CLOCK            => CHIP_CLOCK,
        RESET            => COUNTER_RESET,
        LATCH_ENABLE     => LATCH_ENABLE,
        NORMALIZE        => NORMALIZE,
        BEST_ROWS_IN     => BEST_ROWS_OUT(0),
        BEST_METRICS_IN  => BEST_METRICS_OUT(0),
        SYMBOL_A_IN      => FRAME_DATA_A(0),
        SYMBOL_B_IN      => FRAME_DATA_B(0),
        BEST_ROWS_OUT    => BEST_ROWS_OUT(0),

```



```

BEST_METRICS_OUT  -> BEST_METRICS_OUT(0)
);

DC1:
  for i in 1 to Stages-2 generate
    DCn:
      ONE_STAGE_DECODER
        generic map
        (
          FF_Delay          => FF_Delay,
          Symbol_Delay      => Symbol_Delay,
          Small_Adder_Delay => Small_Adder_Delay,
          Adder_Delay       => Adder_Delay,
          Big_Adder_Delay   => Big_Adder_Delay,
          Multiplexor_Delay => Multiplexor_Delay,
          Subtractor_Delay  => Subtractor_Delay,
          Comparator_Delay  => Comparator_Delay
        )
        port map
        (
          CLOCK          => CHIP_CLOCK,
          RESET          => COUNTER_RESET,
          LATCH_ENABLE   => LATCH_ENABLE,
          NORMALIZE      => ZERO_BIT,
          BEST_ROWS_IN   => BEST_ROWS_OUT(i-1),
          BEST_METRICS_IN => BEST_METRICS_OUT(i-1),
          SYMBOL_A_IN    => FRAME_DATA_A(i),
          SYMBOL_B_IN    => FRAME_DATA_B(i),
          BEST_ROWS_OUT  => BEST_ROWS_OUT(i),
          BEST_METRICS_OUT => BEST_METRICS_OUT(i)
        );
      end generate;
    end generate;

DC2:
  if Stages > 1 generate
    DCf:
      ONE_STAGE_DECODER
        generic map
        (
          FF_Delay          => FF_Delay,
          Symbol_Delay      => Symbol_Delay,
          Small_Adder_Delay => Small_Adder_Delay,
          Adder_Delay       => Adder_Delay,
          Big_Adder_Delay   => Big_Adder_Delay,
          Multiplexor_Delay => Multiplexor_Delay,
          Subtractor_Delay  => Subtractor_Delay,
          Comparator_Delay  => Comparator_Delay
        )
        port map
        (
          CLOCK          => CHIP_CLOCK,
          RESET          => COUNTER_RESET,
          LATCH_ENABLE   => LATCH_ENABLE,
          NORMALIZE      => ZERO_BIT,
          BEST_ROWS_IN   => BEST_ROWS_OUT(Stages-2),
          BEST_METRICS_IN => BEST_METRICS_OUT(Stages-2),
          SYMBOL_A_IN    => FRAME_DATA_A(Stages-1),
          SYMBOL_B_IN    => FRAME_DATA_B(Stages-1),
          BEST_ROWS_OUT  => BEST_ROWS_OUT(Stages-1),
          BEST_METRICS_OUT => BEST_METRICS_OUT(Stages-1)
        );
      end generate;
    end generate;

NORMALIZATION_TESTER:
  DETECTOR_NORMALIZE
    port map
    (
      CLOCK          => CHIP_CLOCK,
      START_DETECT   => START_DETECT,

```

```

        METRIC IN      -> BEST_METRICS_OUT(0),
        NORMALIZE      -> NORMALIZE
    );

EXTRACT DATA:
EXTRACTER_DATA
generic map
(
    Compare_Delay      => Comparator_Delay,
    FF_Delay           => FF_Delay
)
port map
(
    FRAME_CLK          => FRAME_CLOCK,
    CHIP_CLK           => CHIP_CLOCK,
    EXTRACT_ENABLE     => EXTRACT_ENABLE,
    BEST_METRICS       => BEST_METRICS_OUT(Stages-1),
    BEST_ROWS          => BEST_ROWS_OUT(Stages-1),
    DATA_OUT          => DATA_OUT
);

STIMULUS:
process
begin
-- signal initialization
    GLOBAL_RESET      <= '0';
-- reception of data
-- start of one densely-connected trellis
-- cycle 1 (state 0 to state 1, branch metric = 0)
    RX_DATA_A      <= 7;
    RX_DATA_B      <= 7;
    wait for 0 fs;
    wait for Frame_Clock_Period/2;
    FRAME_CLOCK    <= '1';
    FRAME_CLOCK    <= '0';
-- cycle 2 (state 1 to state 3, branch metric = 1)
    RX_DATA_A      <= 0;
    RX_DATA_B      <= 6;
    wait for Frame_Clock_Period/2;
    wait for Frame_Clock_Period/2;
    FRAME_CLOCK    <= '1';
    FRAME_CLOCK    <= '0';
-- cycle 3 (state 3 to state 7, branch metric = 2)
    RX_DATA_A      <= 6;
    RX_DATA_B      <= 1;
    wait for Frame_Clock_Period/2;
    wait for Frame_Clock_Period/2;
    FRAME_CLOCK    <= '1';
    FRAME_CLOCK    <= '0';
-- cycle 4 (state 7 to state 15, branch metric = 3)
    RX_DATA_A      <= 2;
    RX_DATA_B      <= 6;
    wait for Frame_Clock_Period/2;
    wait for Frame_Clock_Period/2;
    FRAME_CLOCK    <= '1';
    FRAME_CLOCK    <= '0';
-- cycle 5 (state 15 to state 31, branch metric = 4)
    RX_DATA_A      <= 2;
    RX_DATA_B      <= 5;
    wait for Frame_Clock_Period/2;
    wait for Frame_Clock_Period/2;
    FRAME_CLOCK    <= '1';
    FRAME_CLOCK    <= '0';
-- cycle 6 (state 31 to state 63, branch metric = 5)
    RX_DATA_A      <= 2;
    RX_DATA_B      <= 3;
    wait for Frame_Clock_Period/2;
    wait for Frame_Clock_Period/2;
    FRAME_CLOCK    <= '1';
    FRAME_CLOCK    <= '0';
-- end of one densely-connected trellis
-- start of one densely-connected trellis
-- cycle 7 (state 63 to state 62, branch metric = 0)
    RX_DATA_A      <= 0;
    RX_DATA_B      <= 0;
    wait for Frame_Clock_Period/2;
    wait for Frame_Clock_Period/2;
    FRAME_CLOCK    <= '1';
    FRAME_CLOCK    <= '0';
-- cycle 8 (state 62 to state 60, branch metric = 0)
    RX_DATA_A      <= 7;
    RX_DATA_B      <= 0;
    wait for Frame_Clock_Period/2;
    wait for Frame_Clock_Period/2;
    FRAME_CLOCK    <= '1';
    FRAME_CLOCK    <= '0';
-- cycle 9 (state 60 to state 56, branch metric = 0)
    RX_DATA_A      <= 0;
    RX_DATA_B      <= 7;
    wait for Frame_Clock_Period/2;
    wait for Frame_Clock_Period/2;
    FRAME_CLOCK    <= '1';
    FRAME_CLOCK    <= '0';
-- cycle 10 (state 56 to state 48, branch metric = 0)
    RX_DATA_A      <= 7;
    RX_DATA_B      <= 0;
    wait for Frame_Clock_Period/2;
    FRAME_CLOCK    <= '1';

```

```

        wait for Frame_Clock_Period/2;    FRAME_CLOCK <= '0';
-- cycle 11 (state 48 to state 32, branch metric = 0)
    RX_DATA_A <= 7;                      RX_DATA_B <= 0;
    wait for Frame_Clock_Period/2;        FRAME_CLOCK <= '1';
    wait for Frame_Clock_Period/2;        FRAME_CLOCK <= '0';
-- cycle 12 (state 32 to state 0, branch metric = 0)
    RX_DATA_A <= 7;                      RX_DATA_B <= 7;
    wait for Frame_Clock_Period/2;        FRAME_CLOCK <= '1';
    wait for Frame_Clock_Period/2;        FRAME_CLOCK <= '0';
-- end of one densely-connected trellis
-- subsequent cycles (stuck at state 0, branch metric = 1 for each transition)
    FRAME_CLOCK_TICKS <= 12;
    RX_DATA_A <= 0;                      RX_DATA_B <= 1;
    for i in 1 to (Constraint_Length-1)*(Stages+1) loop
        wait for Frame_Clock_Period/2;    FRAME_CLOCK <= '1';
        wait for Frame_Clock_Period/2;    FRAME_CLOCK <= '0';
        FRAME_CLOCK_TICKS <= FRAME_CLOCK_TICKS + 1;
    end loop;
    wait;
end process;

TIME_BOMB:
process
begin
    wait until FRAME_CLOCK'event and FRAME_CLOCK='1';
    if FRAME_CLOCK_TICKS=Max_Frame_Clocks then
        terminate;
    end if;
end process;

end TEST;
```

D.15 The Timing-Control Unit

```

-- ++
-- facility:
--
--     TIMING_CONTROLLER
--
-- abstract:
--
--     This unit generates the clocks and timing signals for
--     the SYSDEC design.
--
-- author:
--
--     Lance G. Wright, P.Eng.
--
-- modification history:
--
--     07/08/90 : Created
--
```

```
use work.SYSDEC_PACKAGE.all;
```

```
entity TIMING_CONTROLLER is
port
(
    GLOBAL_RESET : in Bit;
    FRAME_CLK    : in Bit;
    CHIP_CLK     : inout Bit;
    LATCH_ENABLE : out Bit;
    COUNTER_RESET : out Bit;
    SHIFTER_RESET : out Bit;
    START_DETECT : out Bit;
```

```

        EXTRACT_ENAPLE : out Bit
    );
end TIMING_CONTROLLER;

architecture TIMING_BEHAVIOURAL of TIMING_CONTROLLER is

    signal RESET          : Bit          := '0';

begin

    CLOCKS:    -- positive-edge triggered
    process
        variable CLOCK_TICKS    : Integer := 0;
    begin
        wait until FRAME_CLK'event and FRAME_CLK='1';
        CHIP_CLK <= '1';
        CLOCK_TICKS := 0;
        for i in 0 to 2*(State_Count-1) loop
            wait for Chip_Clock_Period/2;
            CHIP_CLK <= not CHIP_CLK;
            CLOCK_TICKS := CLOCK_TICKS + 1;
        -- data extraction enable signal
            if CLOCK_TICKS = 2*(Constraint_Length-2) then
                EXTRACT_ENABLE <= '1' after Chip_Clock_Period/2,
                                   '0' after 3*Chip_Clock_Period/2;
            end if;
        -- counter reset signal derivation
            if CLOCK_TICKS = 2*(State_Count-1) and RESET='1' then
                COUNTER_RESET <= '1' after Chip_Clock_Period/2,
                                   '0' after 3*Chip_Clock_Period/2;
            end if;
        -- normalization detection start signal
            if CLOCK_TICKS = 2*(Constraint_Length-2) then
                START_DETECT <= '1' after Chip_Clock_Period/2,
                                   '0' after 3*Chip_Clock_Period/2;
            end if;
        -- latch enable signal derivation
            if CLOCK_TICKS = 2*(Constraint_Length-2) then
                LATCH_ENABLE <= '1' after Chip_Clock_Period/2,
                                   '0' after 3*Chip_Clock_Period/2;
            end if;
        end loop;
    end process CLOCKS;

    RESETS:
    process
    begin
        wait on GLOBAL_RESET;
        if GLOBAL_RESET='1' then
            RESET <= '1' after 0 fs,
                    '0' after Frame_Clock_Period;
            SHIFTER_RESET <= '1' after 0 fs,
                             '0' after Frame_Clock_Period;
        end if;
    end process RESETS;

end TIMING_BEHAVIOURAL;

```

APPENDIX E : Sample LINKSIM Parameter File

This appendix contains a sample listing of a LINKSIM parameter file. This file is used to set the simulation constraints for simulating the SYSDEC design.

```

testd
!
! Parameters related to DATA
!
4          ! SAMPRT -- samples per bit period
1024       ! BLKSZ  -- block size
0          ! FIRLEN -- number of samples of overlap
2000000    ! NBITS  -- total number of bits per run
0          ! DSEED  -- seed for data generator
4800       ! DRATE  -- information rate in bps
!
! Parameters related to CODEC
!
          ! OUTCOD -- Outer-code name
CONV123    ! INCOD  -- Inner-code name
none       ! PUNRAT -- Puncture code
          ! PUNMSK -- Puncture mask
2-CPSK     ! MODNAM -- encoder (val:{2|4|A|O)-(DE){(C|D)}PSK
1          ! DATINT -- Interleave of codec
14         ! WINDOW -- window of Viterbi decoder
-6.00      ! PILOT  -- energy of pilot (TCMP or ACSSB) relative to data (dB)
1.         ! REFINT -- interleave of reference sequence (val:3, 5)
!
! Parameters related to MODULATOR
!
none       ! PULNAM -- pulse shaping filter
none       ! TXNAM  -- transmit filter
!
! Parameters related to NONLINEARITY
!
none       ! NLNAM  -- non-linearity name
1.0        ! OPPT  -- non-linearity operating point
0          ! CMPSAT -- number of bits used to estimate nonlinearity
!
! Parameters related to CHANNEL
!
6.5        ! EBNO   -- Eb/No
-100.      ! KRICIA -- Rician channel K factor ( dB )
L100       ! MPNAM  -- multipath filter profile
0          ! MSEED  -- seed for multipath
0          ! CSEED  -- seed for channel noise
0.         ! TIMOFF
0.         ! FRQOFF
0.         ! PHSOFF
!
! Parameters related to DEMODULATOR
!
none       ! RXNAM  -- receive filter
none       ! CEFNAM -- Channel estimation filter
!
! Miscellaneous Parameters.
!
Q3....     ! ODEBUG -- Optional outputs (used for debugging purpose)
VITERBISYS ! DETNAM
KEEP       ! STAINP -- Status when closing input file

```

APPENDIX F : Sample LINKSIM Results File

This appendix contains a sample listing of a LINKSIM results file. This file is the standard output when simulating the SYSDEC design.

Results of LINKSIM Simulation Program

System Characteristics

```

Pulse Shaping Filter:  none.dat
Transmit Filter:       none.dat
Nonlinear Device:      none.dat
Receive Filter:        none.dat
Outer code:
Inner code:            CONV123
Puncture rate:         none
Puncture mask:
Modulator:             2-CPSK
Detector Type:         VITERBISYS
Interleave Factor:     1
Interleave of Ref.:    1
Information Rate:      4800.000
Symbol Rate:           9600.000
Size of Window:        14
Debug parameter:       Q3....

```

Simulation Characteristics

```

Block Size:            1024
Samples per Bit Period: 4
Bit per Block:         128
Overlap Size:          0

```

Random Number Seeds

```

Data Seed:             4360825

```

System Performance

```

Estimated EB/NO (dB): 6.500
Estimated Channel K Factor (dB): -100.000
Multipath Filter:      L100
Fading Bandwidth:      100.00000
Channel estimation filter: none
Noise                  0.669137

```

Miscellaneous

```

Estimated Channel Delay: 0.000
End 1 Errors=           9
End 2 Errors=           5
End 3 Errors=           4
End 4 Errors=          13
End 5 Errors=           7
End 6 Errors=           9
End 7 Errors=           7
End 8 Errors=           5

```


Error Statistics

	Run number			
	1	2	3	4
Average bit error rate (BER):	4.500E-06	2.500E-06	2.000E-06	6.500E-06
Number of bits simulated:	2000114	2000000	2000000	2000000
Number of +1 errors:	5	2	1	6
Number of -1 errors:	4	3	3	7
Number of error gaps 0 <...< 10	2	1	1	3
10 <...< 100	0	0	0	0
100 <...< 1000	0	0	0	0
1000 <...< 10000	0	0	0	0
...> 10000	7	4	3	10

	Run number			
	5	6	7	8
Average bit error rate (BER):	3.500E-06	4.500E-06	3.500E-06	2.500E-06
Number of bits simulated:	2000000	2000000	2000000	2000000
Number of +1 errors:	4	6	4	4
Number of -1 errors:	3	3	3	1
Number of error gaps 0 <...< 10	1	3	4	2
10 <...< 100	0	0	0	0
100 <...< 1000	1	0	0	0
1000 <...< 10000	1	0	0	0
...> 10000	4	6	3	3

Net average bit error rate: 3.687E-06
 Est. Standard deviation: 5.170E-07

APPENDIX G : FORTRAN Code for SYSDEC LINKSIM Simulation

This appendix contains the FORTRAN listings for the SYSDEC design. It consists of two subroutines : `sysdec.f` and `sysdecin.f`. The subroutine "`sysdecin.f`" is used to initialize the state of the decoder before simulation commences. The subroutine "`sysdec.f`" emulates the functionality of the SYSDEC design.

G.1 FORTRAN Listing for SYSDECIN.F

```

*
*--DESCRIPTION-----
*
* ROUTINE:      SYSDECIN
*
* PURPOSE:      To initialize variables required by SYSDEC
*
* OPERATION:
*
* INPUTS:       INCODE
*
* OUTPUTS:      BRMET, NEXTST, RXSYM, STATES, STAGES, MEMO
*
* NOTES:
*
*--HISTORY-----
*
* CREATED ON: March 25, 1990
*
* ALTERATION:
*
*--DECLARATIONS-----
*
*      SUBROUTINE SYSDECIN(INCODE, NEXTST, RXSYM, BRMET, BINSTATE,
*      & ARSTAGES, MEMO, PATHMET, NOSTAT, STATEMET)
*
*      IMPLICIT COMPLEX (A-Z,A-Z)
*
*      code name
*      CHARACTER INCODE*20
*
*      max number of states in trellis code
*      INTEGER*2 MAXSTA
*      PARAMETER (MAXSTA = 64)
*
*      number of state
*      INTEGER*2 NOSTAT
*
*      loop counters
*      INTEGER*2 I, J, K, L
*
*      number of shift-register bits in encoder
*      INTEGER*2 MEMO
*
*      max number of bits in the encoder
*      INTEGER*2 NOCOLUMNS
*      PARAMETER (NOCOLUMNS = 6)
*
*      binary representation of the state
*      INTEGER*2 BINSTATE(0:(NOCOLUMNS-1),0:(MAXSTA-1))
*
*      temporary variable
*      INTEGER*2 ITEMP
*
*      calculated symbols (based on encoder)
*      LOGICAL SYMBOLS(0:(MAXSTA-1),0:1,0:1)
*
*      the calculated next-states
*      INTEGER*2 NEXTST(0:(MAXSTA-1),0:1)
*
*      maximum number of stages
*      INTEGER*2 NOSTAGES
*      PARAMETER (NOSTAGES=7)
*
*      number of stages used

```

```

      INTEGER*2 ARSTAGES
*
* maximum number of quantization levels for the received symbol
      INTEGER*2 QUANTLEVEL
      PARAMETER (QUANTLEVEL = 8)
*
* maximum number of bits in a received symbol
      INTEGER*2 NOSYMBITS
      PARAMETER (NOSYMBITS = 3)
*
* received symbols stored in matrix
      INTEGER*2 RXSYM(0:(NOCOLUMNS-1),NOSTAGES,0:1)
*
* state metrics stored in the array
      INTEGER*2 STATEMET(NOSTAGES,0:(MAXSTA-1))
*
* path metrics stored in the array
      INTEGER*2 PATHMET(NOSTAGES,0:(MAXSTA-1))
*
* branch metrics stored in the array
      INTEGER*2 BRMET(0:(MAXSTA-1),0:(QUANTLEVEL-1),0:1,0:1)
*
*--PROCESS-----
*
* Get the encoder memory length
      IF (INCODE(7:7) .EQ. '3') THEN
        MEMO = 2
        ARSTAGES = 7
      ELSE IF (INCODE(7:7) .EQ. '5') THEN
        MEMO = 4
        ARSTAGES = 6
      ELSE IF (INCODE(7:7) .EQ. '7') THEN
        MEMO = 6
        ARSTAGES = 5
      ELSE
        WRITE(*,*) 'Illegal constraint-length'
        STOP
      END IF
*
* Initialize variables
      NOSTAT = 2 ** MEMO
*
* Main loop
      DO 210 I = 0, (NOSTAT-1)
*
* convert the state counter (i) to binary
        ITEMP = I
        DO 100 J = 0, (MEMO-1)
          BINSTATE(J,I) = MOD(ITEMP,2)
          ITEMP = ITEMP / 2
        100 CONTINUE
*
* calculate the symbols (based on the encoder)
*
* based on an encoder input of "0"
      IF (INCODE(7:7) .EQ. '3') THEN
        SYMBOLS(I,0,0) = BINSTATE(1,I).EQ.1
        SYMBOLS(I,0,1) = BINSTATE(1,I).EQ.1 .NEQV. BINSTATE(0,I).EQ.1
      ELSE IF (INCODE(7:7) .EQ. '5') THEN
        SYMBOLS(I,0,0) = BINSTATE(3,I).EQ.1 .NEQV. BINSTATE(1,I).EQ.1
        & SYMBOLS(I,0,1) = BINSTATE(3,I).EQ.1 .NEQV. BINSTATE(0,I).EQ.1
        & SYMBOLS(I,0,1) = BINSTATE(3,I).EQ.1 .NEQV. BINSTATE(2,I).EQ.1
      ELSE IF (INCODE(7:7) .EQ. '7') THEN
        SYMBOLS(I,0,0) = BINSTATE(5,I).EQ.1 .NEQV. BINSTATE(4,I).EQ.1
        & SYMBOLS(I,0,1) = BINSTATE(5,I).EQ.1 .NEQV. BINSTATE(2,I).EQ.1
        & SYMBOLS(I,0,1) = BINSTATE(5,I).EQ.1 .NEQV. BINSTATE(1,I).EQ.1
        & SYMBOLS(I,0,1) = BINSTATE(5,I).EQ.1 .NEQV. BINSTATE(2,I).EQ.1
        & SYMBOLS(I,0,1) = BINSTATE(5,I).EQ.1 .NEQV. BINSTATE(1,I).EQ.1
        & SYMBOLS(I,0,1) = BINSTATE(5,I).EQ.1 .NEQV. BINSTATE(0,I).EQ.1

```

```

      END IF
*
*   based on an encoder input of "1"
      SYMBOLS(I,1,0) = .NOT. SYMBOLS(I,0,0)
      SYMBOLS(I,1,1) = .NOT. SYMBOLS(I,0,1)
*
*   Calculate the next state(s)
*
*   based on an encoder input of "0"
      NEXTST(I,0) = 2 * I
      IF ( NEXTST(I,0) .GE. NOSTAT )
        &   NEXTST(I,0) = NEXTST(I,0) - NOSTAT
*
*   based on an encoder input of "1"
      NEXTST(I,1) = NEXTST(I,0) + 1
*
*   Calculate the branch metrics
*
*   do for each possible received symbol
      DO 200 J = 0, (QUANTLEVEL-1)
*
*   based on an encoder input of "0"
      IF ( .NOT. SYMBOLS(I,0,0) ) THEN
        BRMET(I,J,0,0) = J
      ELSE
        BRMET(I,J,0,0) = 7 - J
      END IF
      IF ( .NOT. SYMBOLS(I,0,1) ) THEN
        BRMET(I,J,0,1) = J
      ELSE
        BRMET(I,J,0,1) = 7 - J
      END IF
*
*   based on an encoder input of "1"
      IF ( .NOT. SYMBOLS(I,1,0) ) THEN
        BRMET(I,J,1,0) = J
      ELSE
        BRMET(I,J,1,0) = 7 - J
      END IF
      IF ( .NOT. SYMBOLS(I,1,1) ) THEN
        BRMET(I,J,1,1) = J
      ELSE
        BRMET(I,J,1,1) = 7 - J
      END IF
200   CONTINUE
210   CONTINUE
*
*   Initialize the array
*
*   Do for all stages
      DO 400 J = 1, ARSTAGES
        DO 220 K = 0, (MEMO-1)
          RXSYM(K,J,0) = 0
          RXSYM(K,J,1) = 0
220     CONTINUE
*
*   do for all rows
      DO 300 L = 0, (NOSTAT-1)
*
*   initialize path and state metrics
        STATEMET(J,L) = 0
        PATHMET(J,L) = 0
300     CONTINUE
400     CONTINUE
*
*   Done
      RETURN
      END

```

G.2 FORTRAN Listing for SYSDEC.F

```

*
*-----
*      NAME:          SYSDEC
*
*      FUNCTION:      A systolic array viterbi decoder for rate 1/2
*                     convolutional codes.
*
*      INPUTS:        BITBLK - the number of received symbols
*                     in this block.
*                     DECISI - the quantized (8-level) received symbol
*                     values
*                     DATINT - the data interleave factor. This must
*                     must be equal to 1 (ie : no interleaving)
*                     WINDOW - the pipeline delay for the encoder in
*                     baseband data periods
*                     INCODE - character string describing the coding
*                     scheme (ie: CONV127 is convolutional
*                     coding (a must), rate 1/2, constraint-
*                     length 7
*                     BITDEL - the delay through the comm system. This
*                     value is in tx symbol periods.
*
*      OUTPUTS:       RXBITS - the decoder data output. These are
*                     logical values. There are half as many
*                     of these as received symbols (rate 1/2).
*
*      OPERATION:     Uses a modified Viterbi decoding algorithm which
*                     is implemented using a systolic-array architecture
*
*      SIDE EFFECTS:
*
*      NOTES:
*-----
*
*      PROGRAMMER(S)   Lance G. Wright, P.Eng
*
*      CREATION DATE:   March 25, 1990
*
*      COPYRIGHT (C)    Lance G. Wright, P.Eng
*
*      VERSION NUMBER:  2.1
*
*      EDIT NUMBER:     0
*
*      LAST EDIT DATE:  19 July 1990
*
*      ALTERATION HISTORY
*-----
*
* [Parameters]
*
*      SUBROUTINE SYSDEC(BITBLK, DECISI, DATINT, WINDOW, INCODE, BITDEL,
*      6          SYSINI, RXBITS)
*
*      IMPLICIT COMPLEX (A-Z, A-Z)
*
*      maximum block size
*      INTEGER*2 MAXBLK
*      PARAMETER (MAXBLK = 1024)
*
*      max number of states in trellis code
*      INTEGER*2 MAXSTA

```

```

      PARAMETER (MAXSTA = 64)
*
* max number of quantization levels
  INTEGER*2 QUANTLEVEL
  PARAMETER (QUANTLEVEL = 8)
*
* maximum number of columns
  INTEGER*2 NOCOLUMNS
  PARAMETER (NOCOLUMNS = 6)
*
* maximum number of stages
  INTEGER*2 NOSTAGES
  PARAMETER (NOSTAGES=7)
*
* maximum number of bits in a received symbol
  INTEGER*2 NOSYMBITS
  PARAMETER (NOSYMBITS = 3)
*
* constant value for subtraction/normalization
  INTEGER*2 NORMCONST
  PARAMETER (NORMCONST = 64)
*
* minimum value for the statemetric overflow condition
  INTEGER*2 MAXMETRIC
  PARAMETER (MAXMETRIC = 63)
*
* maximum value for the statemetric register
  INTEGER*2 MAXREG
  PARAMETER (MAXREG = 511)
*
* Number of bits per block
  INTEGER BITBLK
*
* interleave factor
  INTEGER DATINT
*
* size of window in Viterbi algorithm
  INTEGER WINDOW
*
* code name
  CHARACTER INCODE*20
*
* bit delay of channel
  INTEGER BITDEL
*
* Estimated received data
  REAL DECISI(*)
*
* the estimated bits
  LOGICAL RXBITS(MAXBLK)
*
* loop counters
  INTEGER*2 H, I, J, K, L, M
*
* syms being decoded
  INTEGER*2 SYMB(0:(NOCOLUMNS-1),0:1)
*
* effective number of states in trellis code
  INTEGER*2 NOSTAT
*
* rx syms with overlap bit due to channel delay
  INTEGER*2 SYMVLP(2*NOCOLUMNS-1)
*
* offset for odd bit delays
  INTEGER*2 ILOW
*
* Bit Delay prior Viterbi
  INTEGER*2 OLDBIT
*

```

```

* set when initialization must be performed
  LOGICAL SYSINI
*
* integer representation of the quantized decision values
  INTEGER*2 INTDEC(MAXBLK)
*
* Next state table for both possible paths
  INTEGER*2 NEXTST(0:(MAXSTA-1),0:1)
*
* syms stored in the array
  INTEGER*2 RXSYM(0:(NOCOLUMNS-1),NOSTAGES,0:1)
*
* stages in the array
  INTEGER*2 ARSTAGES
*
* memory in the encoder (bits)
  INTEGER*2 MEMO
*
* stored branch metrics
  INTEGER*2 BRMET(0:(MAXSTA-1),0:(QUANTLEVEL-1),0:1,0:1)
*
* stored state metrics
  INTEGER*2 STATEMET(NOSTAGES,0:(MAXSTA-1))
*
* stored path metrics
  INTEGER*2 PATHMET(NOSTAGES,0:(MAXSTA-1))
*
* binary representation of the state
  INTEGER*2 BINSTATE(0:(NOCOLUMNS-1),0:(MAXSTA-1))
*
* error totals for each row for each terminal state
  INTEGER*2 ERROR(0:(MAXSTA-1),0:(MAXSTA-1))
*
* survivor row at each stage for each possible terminal state
  INTEGER*2 BESTROW(NOSTAGES,0:(MAXSTA-1))
*
* flags to signal that normalization is necessary (counter overflow)
  LOGICAL NORMALIZE
  LOGICAL NORM2
*
* temporary variables
  INTEGER*2 ITEMP
  INTEGER*2 PREVST
  INTEGER*2 TMPERR
  INTEGER*2 INDEX
*
*
*--PROCESS-----
*
  SAVE BRMET,NEXTST,RXSYM,NOSTAT,MEMO,PATHMET,
    & ARSTAGES,OLDBIT,SYMVLP,STATEMET,BINSTATE,
    & NORM2
  DATA SYMVLP / 11*0 /
  DATA NORM2 / .FALSE. /
*
* Sanity checks
  IF (DATINT.GT.1) THEN
    WRITE(*,*) 'Systolic-array decoder cannot be interleaved'
    STOP
  END IF
*
* initialization of the look-up tables
  IF (SYSINI) THEN
    CALL SYSDECIN(INCODE, NEXTST, RXSYM, BRMET, BINSTATE,
    & ARSTAGES, MEMO, PATHMET, NOSTAT, STATEMET)
    &
    SYSINI = .FALSE.
    OLDBIT = BITDEL
    BITBLK = 2 * MEMO * ARSTAGES
    DO 90 I = 1, BITBLK

```



```

          DECISI(I) = -1.0
90      CONTINUE
      END IF
*
* convert (max 8-level) quantized decision values into integer values
      DO 100 I = 1, BITBLK
          IF ( DECISI(I) .GT. 0.72 ) THEN
              INTDEC(I) = 7
          ELSE IF ( DECISI(I) .LT. -0.72 ) THEN
              INTDEC(I) = 0
          ELSE IF ( DECISI(I) .GT. 0.43 ) THEN
              INTDEC(I) = 6
          ELSE IF ( DECISI(I) .LT. -0.43 ) THEN
              INTDEC(I) = 1
          ELSE IF ( DECISI(I) .GT. 0.15 ) THEN
              INTDEC(I) = 5
          ELSE IF ( DECISI(I) .LT. -0.15 ) THEN
              INTDEC(I) = 2
          ELSE IF ( DECISI(I) .GT. 0.0 ) THEN
              INTDEC(I) = 4
          ELSE IF ( DECISI(I) .LT. 0.0 ) THEN
              INTDEC(I) = 3
          ELSE
              WRITE(*,*) DECISI(I), '*** ERROR : Invalid quantized symbol'
              STOP
          END IF
100      CONTINUE
*
* [compute Information bit delay from channel bit delay]
      BITDEL = ((BITDEL + 1) / 2) + (WINDOW * DATINT)
*
* [Set first symb, Handle the overlap bit, in case of odd BITDEL]
      IF (MOD(OLDBIT,2) .EQ. 1) THEN
          ILOW = 0
*
* get one full stage's worth of symbols
          I = 1
          DO 103 J = 0, (MEMO-2)
              SYMB(J,0) = SYMVLP(I)
              SYMB(J,1) = SYMVLP(I+1)
              I = I + 2
103      CONTINUE
          SYMB((MEMO-1),0) = SYMVLP(2*MEMO-1)
          SYMB((MEMO-1),1) = INTDEC(1)
*
* [No overlapping]
          ELSE
              ILOW = 1
*
* get one full stage's worth of symbols
          DO 106 J = 0, (MEMO-1)
              SYMB(J,0) = INTDEC(ILOW+2*J)
              SYMB(J,1) = INTDEC(ILOW+2*J+1)
106      CONTINUE
          END IF
*
* [Start decoding]
          M = 1 - MEMO
          DO 200 I = ILOW, (BITELK - 1), 2 * MEMO
              M = M + MEMO
*
* shift syms into the array
*
* do for each stage
          DO 120 K = 1, (ARSTAGES-1)
*
* do for each column
          DO 110 L = 0, (MEMO-1)
              RXSYM(L,K,0) = RXSYM(L,(K+1),0)

```

```

        RXSYM(L,K,1) = RXSYM(L,(K+1),1)
110      CONTINUE
120      CONTINUE
*
* last stage gets the newest symbols
      DO 123 L = 0, (MEMO-1)
        RXSYM(L,ARSTAGES,0) = SYMB(L,0)
        RXSYM(L,ARSTAGES,1) = SYMB(L,1)
123      CONTINUE
*
* do for each stage
      DO 180 H = 1, ARSTAGES
*
* perform calculations for each possible terminal state
      DO 170 J = 0, (NOSTAT-1)
*
* do for each row in the array
        DO 160 K = 0, (NOSTAT-1)
          PREVST = K
          TMPERR = 0
*
* do for each column in the row
          DO 150 L = 0, (MEMO-1)
            ITEM = BINSTATE((MEMO-L-1),J)
            TMPERR = TMPERR
            &          + BRMET(PREVST,RXSYM(L,H,0),ITEM,0)
            &          + BRMET(PREVST,RXSYM(L,H,1),ITEM,1)
            PREVST = NEXTST(PREVST,ITEM)
150          CONTINUE
            ERROR(J,K) = TMPERR
160          CONTINUE
*
170          CONTINUE
*
* determine the best metric for each start-state
* based on the lowest error at the output of the rows
        DO 176 J = 0, (NOSTAT-1)
          ITEM = MAXREG
          DO 173 K = 0, (NOSTAT-1)
            IF ( H.EQ. 1 ) THEN
              TMPERR = ERROR(J,K) + STATEMET(H,K)
            ELSE
              TMPERR = ERROR(J,K)
              &          + STATEMET((H-1),K)
            END IF
            IF ( TMPERR.LT. ITEM ) THEN
              ITEM = TMPERR
              INDEX = K
            END IF
173          CONTINUE
*
* get the path metric
          PATHMET(H,J) = ITEM
          IF ( PATHMET(H,J) .GT. MAXREG ) PATHMET(H,J) = MAXREG
*
* extract the start-state
          IF ( H.EQ. 1 ) THEN
            BESTROW(H,J) = INDEX
          ELSE
            BESTROW(H,J) = BESTROW((H-1),INDEX)
          END IF
176          CONTINUE
*
* Normalize the path metrics
          IF ( H.EQ. 1 ) THEN
*
* Test for normalization on next pass
            J = NOSTAT-1
1771          CONTINUE

```

```

        IF ( PATHMET(H,J) .LE. MAXMETRIC ) GOTO 1772
        J = J - 1
        IF ( J .GE. 0 ) GOTO 1771
        NORMALIZE = .TRUE.
        GOTO 1773
1772    NORMALIZE = .FALSE.
1773    CONTINUE
*
* Perform the normalization if necessary
        IF (NORM2) THEN
            DO 178 J = 0, (NOSTAT-1)
                PATHMET(H,J) = PATHMET(H,J) - NORMCONST
178    CONTINUE
            NORMALIZE = .FALSE.
        END IF
    END IF
*
* Update the state metrics
        DO 179 J = 0, (NOSTAT-1)
            STATEMET(H,J) = PATHMET(H,J)
179    CONTINUE
*
180    CONTINUE
*
* determine the decoded bit by selecting the start-state row
        ITEMP = 9999
        DO 190 J = 0, (NOSTAT-1)
            IF ( PATHMET(ARSTAGES,J) .LT. ITEMP ) THEN
                ITEMP = PATHMET(ARSTAGES,J)
                INDEX = J
            END IF
190    CONTINUE
*
* The best state at this point represents the memo decoded bits
* Hence, convert them to binary
        ITEMP = BESTROW(ARSTAGES,INDEX)
        DO 193 K = 0, (MEMO-1)
            RXBITS(M+MEMO-K-1) = MOD(ITEMP,2) .EQ. 1
            ITEMP = ITEMP / 2
193    CONTINUE
*
* [Set symb for next pass]
        DO 196 J = 0, (MEMO-1)
            SYMB(J,0) = INTDEC(I + 2*J + 2*MEMO)
            SYMB(J,1) = INTDEC(I + 2*J + 2*MEMO + 1)
196    CONTINUE
*
* set the nrmalization flag for the next pass
        NORM2 = NORMALIZE
*
200    CONTINUE
*
* [Save last symb if bitdel is odd]
        IF (MOD(OLDBIT,2) .EQ. 1) THEN
            DO 210 J = BITBLK-(2*MEMO-1)+1, BITBLK
                SYMVLP(J) = INTDEC(J)
210    CONTINUE
        END IF
*
* [Reduce BITBLK to its original value]
        BITBLK = BITBLK / 2
*
* Done
        RETURN
    END

```

END

04.03.92

FIN