INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

**ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600**

UMI[®]

VLSI Implementation of A 2/3 Viterbi Decoder and Noise Effects in Digital ICs

By

Youxiong Zhao

A thesis submitted to the
Faculty of Graduate Studies and Research
In partial fulfillment of the requirements
For the degree of
Master of Applied Science

Ottawa-Carleton Institute of Electrical Engineering,
Department of Electronics,
Carleton University,
Ottawa, Ontario, Canada

January, 2002

© Youxiong Zhao, 2002

**National Library
of Canada**

**Acquisitions and
Bibliographic Services**

**385 Wellington Street
Ottawa ON K1A 0N4
Canada**

**Bibliothèque nationale
du Canada**

**Acquisitions et
services bibliographiques**

**385, rue Wellington
Ottawa ON K1A 0N4
Canada**

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

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

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

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

0-612-68318-4

Canada

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

**VLSI Implementation of A 2/3 Viterbi
Decoder and Noise Effects in Digital ICs**

Submitted by Youxiong Zhao in partial fulfillment of the requirements
For the degree of Master of Applied Science

Thesis Supervisor

Chairman, Department of Electronics

Ottawa-Carleton Institute of Electrical Engineering,
Department of Electronics,
Carleton University,
Ottawa, Ontario, Canada

January, 2002

Abstract

This thesis consists of two major objectives in the digital circuit area. The first objective is the VLSI implementation of a high speed $2/3$ Viterbi Decoder (ISO/IEC 16500) for a MMDS receiver using an automatic synthesis design flow. The design was implemented using TSMC CMOSP35 technologies using Verilog HDL, Cadence, and Synopsys. The results from different operation implementations are compared in order to improve speed. Union Bounds and computer simulations are also used to investigate the performance of the $2/3$ convolutional coding. As well, an effective hardware implementation is developed to prevent overflow. The decoder, using a pipelined structure, achieves a maximum frequency of 120 MHz which corresponds to a baud rate of 240 MHz.

The second objective of this thesis is to model and analyze noise effects in self-timed and synchronous digital circuits, with an emphasis on low voltage operation and high speed issues. Simulations show that noise performance of a self-timed system can be superior to that of a functionally-equivalent synchronous one, especially at higher operating speeds. Test results of a digital circuit in CMOSP35 also confirm these results. For I/O drivers, simulations show that the I/O pads of a self-timed system can have better noise performance than that of a synchronous system.

Acknowledgements

I greatly appreciate all the help given by Dr. John Knight, Jacques A. Lemieux, Brent Veitch, Richard Wieler, and Warren Faire. Their time and expertise have been invaluable to these projects.

The projects were sponsored by Cogency, Micronet, and CMC. I would like to express my sincere thanks to these organizations.

Thanks most of all to my supervisor, Ralph Mason, for his patience, motivation, and refreshing ideas.

Table of Contents

Abstract	iii
Table of Contents	v
List of Figures	vii
List of Tables	x
List of Abbreviations	xi
Chapter 1 Introduction	1
1.1 Introduction	1
1.1.1 Convolutional Coding	1
1.1.2 Noise in Digital Circuits	2
1.2 Thesis Objectives	4
1.3 Thesis Contributions	5
1.4 Thesis Organization	6
Chapter 2 Background	7
2.1 Overview of Digital Communications Systems	7
2.2 Convolutional Coding	9
2.3 Viterbi Algorithm	11
2.4 Self-Timed Logic	12
2.5 Noise Sources in VLSI Systems	15
2.5.1 An Overview	15
2.5.2 Substrate Noise and Packaging	18
2.5.3 Crosstalk and Radio Frequency Radiation	22
Chapter 3 Custom Decoder Chip Design	24
3.1 A 2/3 Convolutional Encoder	24
3.2 Design Implementation	27
3.2.1 Overview	27
3.2.2 Block Diagram	28
3.2.3 ACS Module and Overflow Prevention	30
3.2.4 Data_Carry_Forward module	38

3.2.5 Latency	41
3.2.6 Comparison	42
3.3 Performance of the 2/3 Viterbi Decoder	43
3.3.1 Union Bounds	43
3.3.2 Simulation Results	47
3.4 Design Process	48
3.4.1 RTL Model and Simulation	48
3.4.2 Logic Synthesis	50
3.4.3 Physical Design	52
3.5 Test and Conclusions	53
Chapter 4 Noise Effects in Digital Circuit	55
4.1 Technical Features of Self-Timed Design	55
4.2 An Novel Empirical Model and Simulations	56
4.2.1 Noise Effects on An Inverter	56
4.2.2 Noise Effects on a D Flip-Flop	63
4.3 VLSI Implementation	68
4.3.1 Test Results	69
4.4 Noise Between I/O Drivers	73
Chapter 5 Conclusions and Future Work	80
5.1 Conclusions	80
5.2 Future Work	81
References	82
Appendix A: Verilog code	88
Appendix B: Block Diagram and Schematic	123
Appendix C: PGA68 Package	125
Appendix D: Bonding diagram	126
Appendix E: MATLAB code and C code	127
Appendix F: Synopsys Timing Report	131

List of Figures

Figure 2.1 Block diagram of elements at the MMDS downstream head-end	8
Figure 2.2 Block diagram of elements at the MMDS downstream receiving site	9
Figure 2.3 Reference model at the transmitter and receiver site	11
Figure 2.4 Self-Timed bounded data interface.....	15
Figure 2.5 Noise model	15
Figure 2.6 Mechanisms for noise coupling	17
Figure 2.7 Digital Circuit EMI.....	22
Figure 3.1 Reference model of the TCM encoder.....	24
Figure 3.2 Detailed reference model of the TCM encoder	25
Figure 3.3 DAVIC 1.3.1 TCM Trellis diagram.....	26
Figure 3.4 The block diagram of encoder/decoder	28
Figure 3.5 Decoder flowchart.....	29
Figure 3.6 The block diagram to find the minimum accumulated distance in the real chip	32
Figure 3.7 The block diagram to find the surviving path for state 0000 in the real chip..	32
Figure 3.8 The block diagram to find the surviving path for state 0000 in Modulo Normalization	34
Figure 3.9 The pipelined structure to find the minimum accumulated distance for 16 states	35
Figure 3.10 The pipelined structure to find surviving path among 4 paths that merge to one state	35
Figure 3.11 The pipelined structure to find a surviving path using Modulo Normalization	36
Figure 3.12 State diagram for came_from0	37
Figure 3.13 A simple example of a convolutional encoder.....	38
Figure 3.14 Survivor paths	38
Figure 3.15 The Carry Path Forward	39
Figure 3.16 The path tracing in hardware	40

Figure 3.17 Shift register structure.....	41
Figure 3.18 Trellis diagram for deducing the second group of state expressions in equation 3-9	47
Figure 3.19 Bit error probability for the Viterbi decoding and BPSK modulation.....	47
Figure 3.20 The block diagram of computer simulation for evaluating bit error probability	48
Figure 3.21 RTL level simulation results for error correction	49
Figure 3.22 Test results of the non-pipelined decoder circuit.....	54
Figure 3.23 Data flowing diagram for the real decoder chip test.....	54
Figure 4.1 Circuit structure for noise effects on an inverter	57
Figure 4.2 Simulation results for the synchronous system	59
Figure 4.3 Simulation results at 2.9V operating voltage for the self-timed system.....	61
Figure 4.4 Simulation results for the self-timed system	62
Figure 4.5 The circuit structure for noise effects on a D flip-flop	63
Figure 4.6 Simulation of the falling edge D flip-flop works at $V_{dd}=1.2V$ in the synchronous system.....	65
Figure 4.7 Simulation of the falling edge D flip-flop works at $V_{dd}=1.4V$ in the self-timed system	66
Figure 4.8 Maximum on-chip clock frequency for the D flip-flop to work properly versus operating voltage	67
Figure 4.9 Maximum off-chip frequency versus operating voltage.....	68
Figure 4.10 VLSI implementation to verify simulation results	69
Figure 4.11 Test Setup	69
Figure 4.12 Input signal waveform of the D flip-flop.....	70
Figure 4.13 The D flip-flop output waveform at 415MHz frequency and 1.7V operating voltage for self-timed case.....	71
Figure 4.14 The D flip-flop output waveform at 415MHz frequency and 1.7V operating voltage for synchronous case.....	72
Figure 4.15 Maximum frequency at which the D flip-flop works properly versus operating voltage	73
Figure 4.16 Simulation results for the synchronous system at $C_{load}=1pF$	75
Figure 4.17 Simulation results for the synchronous system at $C_{load}=10pF$	76

Figure 4.18 Simulation results for the self-timed system at $C_{load}=1\text{pF}$	78
Figure 4.19 Simulation results for the self-timed system at $C_{load}=10\text{pF}$	79

List of Tables

Table 2-1 Substrate parameters	21
Table 3-1 The comparison for the different overflow prevention of the pipelined decoder	43

List of Abbreviations

CMC	Canadian Microelectronic Corporation
DAVIC	Digital Audio Visual Council
DEF	Design Exchange format
DFT	Design For Test
DRC	Design Rule Check
DUT	Device Under Test
EMI	Electromagnetic Interference
FEC	Forward Error Correction
FF	Flip-flop
LSB	Least-Significant-Bit
LVS	Layout Versus Schematic
MSB	Most-Significant-Bit
MMDS	Microwave Multipoint Distribution Services
PDP	Physical Design Planner
QAM	Quadrature Amplitude Modulation
RF	Radio Frequency
RTL	Register Transfer Level
TCM	Trellis-Coded Modulation
TSMC	Taiwan Semiconductor Microelectronic corporation
VA	Viterbi Algorithm

Chapter 1 Introduction

1.1 Introduction

1.1.1 Convolutional Coding

With the use of new integrated circuit and radio frequency (RF) technologies, the rate at which coded data may be transmitted and received has increased. This increased rate of data transmission, without an increase in transmit power, causes a reduction in the bit energy (E_b). This increases the overall Bit Error Rate (BER). To compensate for the decrease in bit energy, coding may be incorporated into the system which will create an increase in effective system gain due to the coding gain achieved [1]. To meet increasing data rate requirements, an encoder/decoder pair capable of operating at high rates is essential. Unfortunately, the gap between the rates at which individual decoders operate efficiently and the rates at which data is transmitted has also increased [2]. This gap is being reduced, mainly with the use of new decoder implementation technologies.

Convolutional coding is a bit-level encoding technique, as compared to block coding techniques such as Reed-Solomon coding. The advantages of convolutional codes over block codes for telecom/datacom applications are [1,2]:

- 1) Convolutionally encoded system gain degrades gracefully as the error rate increases with soft-decision data. Block codes correct errors up to a point, after which the gain drops off rapidly.
- 2) Convolutional codes do not require block synchronization.

- 3) For convolutional codes with non-pipelined implementation, latency which is 4 or 5 times the code constraint length is sufficient for a near-optimum decoder performance. Block codes introduce more latency by requiring the reception of an entire data block before decoding begins.

Convolutional encoder error-correction capabilities result from outputs that depend on past data values. Convolutional codes are commonly decoded by using a trellis diagram to find the most likely sequence of codes given the possible set of state transitions created from the dependence of the current symbol on past data. The allowable state transitions are represented in the trellis diagram. The VA (Viterbi Algorithm) simplifies the decoding task by limiting the number of sequences examined. The most likely path to each state is retained for each new symbol [3, 6, 8]. There is also a tradeoff between better performance and hardware implementation complexity for convolutional coding.

1.1.2 Noise in Digital Circuits

As the amount of digital logic operating at high frequencies and low voltage has increased, researchers have begun to focus on noise in digital circuits [10, 11, 12, 13, 16, 17, 20, 21, 22, 23]. Circuits at high frequency generate noise, and circuits at low operating voltage reduce noise margins. Digital noise alters the performance of circuits and systems and restricts design performance. For example:

- Mixed signal design, such as analog to digital converters, are sensitive to digital noise transmitted via the substrate and can benefit from the reduction of such noise [12].

- Synchronization error rates of digital systems increase in the presence of supply noise [13].
- Noise from simultaneous switching logic can cause false clocking [15].
- Electromagnetic radiation causes difficulty in meeting regulatory requirements such as those imposed by the Federal Communications Commission [14] [16].
- Advances in process technology require scaling down of the power supply voltage that reduces the noise margin of the digital circuit [15].

Recent academic research has resulted in an understanding of: 1) how the current from a switching logic gate generates noise; 2) the mechanisms of noise transmission to other logic; 3) solutions for reduction of the noise coupling [11].

A common premise behind the reduction of coupled noise is to reduce noise susceptibility through the separation of switching logic and sensitive circuits in space and time. Separation in space requires noise generators to be physically located as far as possible from sensitive circuits during layout. Separation in time requires the number of logic gates that switch at the same time to be kept to a minimum. Unfortunately, synchronous systems offer little latitude in reducing noise via these techniques, as the very nature of clocked circuits combined with aggressive minimum clock-skew requirements guarantee simultaneously switching logic throughout a VLSI circuit [11]. The noise transport mechanism for the low resistivity substrates common in CMOS processes is primarily through indirect paths such as the low impedance power routing, and not by direct coupling through the substrate [17]. There, physical separation of noise sources from noise sensitive circuits, or barrier methods such as guard rings, are less

effective options for isolating the switching noise of clocked logic in CMOS processes [47].

Self-timed logic offers a viable alternative for the reduction of noise [19]. Unlike synchronous design, self-timed systems propagate data using handshaking signalling between a sender and a receiver, eliminating the need for a global clock. Switching of logic is then governed by the processing delay between communicating components, rather than a central timekeeper [11]. Intuitively, one would expect a reduced number of simultaneously switching outputs in self-timed systems, and therefore reduced peak switched current compared to an equivalent synchronous implementation. Furthermore, the fact that self-timed circuits only consume energy when doing useful work would imply lower power consumption and less noise [19].

Since self-timed circuits have these potential benefits in noise performance, this thesis investigates the noise performance of self-timed circuits and synchronous circuits and focuses on comparing two approaches.

1.2 Thesis Objectives

The original plan for this work was to implement convolutional decoders in both synchronous and self-timed circuits and make a comparison. A synchronous 2/3 convolutional decoder (ISO/IEC 16500) is implemented and techniques for the VLSI implementation are developed to improve its speed. Due to implementation and time

constraints, a smaller digital circuit is used to investigate noise effects instead of a complete self-timed decoder.

Noise effects in self-timed and synchronous digital circuits are modeled, simulated, and compared. VLSI implementation of a small digital circuit with CMOSP35 technology to verify the simulation results is also reported.

1.3 Thesis Contributions

A synchronous 2/3 Viterbi decoder is implemented in CMOSP35 technology. The major contributions for the decoder are as follows:

- The state diagram for the convolutional code is extracted from the DAVIC 1.3.1 encoder.
- A novel approach to prevent overflow is developed, and comparison is made with Modulo Normalization.
- VLSI implementation of the synchronous decoder. The design process includes RTL coding, logic synthesis, scan chain insertion, floorplan, placement, clock tree generation, routing, timing analysis, DRC, LVS, antenna violation etc.
- A theoretical upper bound for the probability of bit error is calculated using a 16×16 state transition matrix and Matlab Symbolic Math toolbox. A computer simulation is performed in order to estimate the bit error probability of the real Verilog implementation and to compare with the theoretical upper bound.

- A PCB board is designed to test the chip. The decoder is tested using a programmable Pattern Generator, and a Logic Analyzer.

The major contributions for noise effects are as follows:

- A novel empirical model is developed to simulate and analyze noise effects in synchronous and self-timed digital systems.
- A novel circuit structure is developed to verify simulation results. The circuit is implemented in CMOS35. D flip-flop clock signals in the circuit are gated with delay buffers and Area Utilization, Channel Offset, and Average Row Utilization are adjusted to solve congestion problems in the physical design.
- The circuit is tested using the above PCB board.

1.4 Thesis Organization

The remainder of the thesis is organized as follows: Chapter 2 provides a brief introduction to basic digital communication systems and noise sources in VLSI systems. Chapter 3 details the implementation of a custom TCM decoder chip using CMOS35 technology provided through CMC. Chapter 4 investigates noise effects in digital circuits. Chapter 5 draws the conclusions of the overall research project.

Chapter 2 Background

2.1 Overview of Digital Communications Systems

In the past, data communication systems operated at very low speeds and with very light loads compared to present standards. Because of the lower speed of transmission, the bit and packet error rates were not as significant a problem as today. As the world of data communications expands in terms of services and applications, a growing need for more efficient communication services is required.

End users and service providers are no longer prepared to pay for the lower rates of data transmission. A great deal of thought and experimentation has gone into the improvement of data links and error recovery techniques. Forward Error Correction (FEC) schemes, using encoders in the transmitters and decoders in the receivers, increase the probability of error recovery at the receiver [1, 2, 5, 6]. The coding gain of the transmission link can be increased.

“Error-control coding involves the addition of redundancy to transmitted data so as to provide the means for detecting and correcting errors that inevitably occur in any real communication process “ [5]. Adding redundancy to the transmitted data and making use of soft-decision decoding, the bit-error rate can be reduced considerably without an increase in transmission power. These coding techniques have proved very useful in the

past decade, and many of them have been standardized for modems and other communication devices. The CCITT recommendation V.32 is a common standard that uses trellis-coded modulation and Viterbi decoding to achieve forward error correction [8].

The DAVIC (Digital Audio Visual Council) TCM coding is another standard that uses trellis-coded modulation and Viterbi decoding to achieve forward error correction for MMDS (Microwave Multipoint Distribution Services) receivers. Figure 2.1 and Figure 2.2 contain a conceptual block diagram of elements at the MMDS downstream head-end and receiving site [4].

Figure 2.1 Block diagram of elements at the MMDS downstream head-end

Figure 2.2 Block diagram of elements at the MMDS downstream receiving site

2.2 Convolutional Coding

In the past, the majority of voice waveforms were sent as analog signals in electrical form over a twisted pair of wires. These telephone voice signals had a bandwidth of 4KHz. If the channel polluted the signal with noise, the only thing that happened was that the conversation got a bit noisier. If noise corrupted a few bits, the corresponding sample value(s) could be slightly wrong or very wrong depending on whether the bad bits were

near the most-significant-bit (MSB) or least-significant-bit (LSB). For the most part, the conversation sounded noisier, but was still discernible. When people started to send data files rather than voice, corrupted bits became more important. Even one wrong bit could prevent a program from running properly. The reliability of the channel had to be improved.

The probability of error can be reduced by transmitting more bits than needed to represent the information being sent, and convolving each bit with neighbouring bits so that if one transmitted bit gets corrupted, enough information is carried by the neighbouring bits to allow a good estimate of what the corrupted bit was. This approach is called channel coding, and the particular approach of convolving the bits to distribute the information is referred to as convolution coding [1]. The need for coding became even more important in the use of cellular phones, as the cellular environment is extremely unreliable.

There are two modes: one uses TCM (Trellis Coded Modulation) coding; another uses byte-to-symbol conversion, differential encoding and QAM constellation mapping. The TCM's purpose is to convolutionally encode the bits into the modulation and perform the differential encoding. It is a coding scheme that consists of the combination of a differential encoder, a convolutional encoder, and a mapper, all in a single functional block. This takes place after the RS encoder and interleaver, as shown in Figure 2.3 [4].

Figure 2.3 Reference model at the transmitter and receiver site

This coding scheme does not modify the shape of the constellation, or the spectrum. It does provide additional coding gain that increases the coverage area of the MMDS transmitter [4].

2.3 Viterbi Algorithm

The Viterbi Algorithm discovered and analyzed by Viterbi [6] in 1967 has long been recognized as an effective solution to convolutional decoding. Generally defined, the Viterbi Algorithm operates on sets of received data sequences or symbols to determine the most probable path through a trellis diagram. The received symbols are correlated against certain predefined sets of binary signals representing transitions from one state to another. Such correlation is performed over a certain quantity of continuous branches.

The Viterbi algorithm essentially performs maximum likelihood decoding. It reduces the computational load by taking advantage of the special structure in the trellis code. The algorithm involves calculating a distance between the received signal, at time t , and all the trellis paths entering each state at time t . It removes from consideration those trellis paths that could not possibly be candidates for the maximum likelihood choice. When two paths enter the same state, the path having the minimum distance metric is chosen; this path is called a surviving path. This selection of surviving paths is performed for all the states. The decoder continues in this way to advance deeper into the trellis, making decisions by eliminating the non-candidate paths. These eliminated paths cannot be the most likely paths. The early rejection of the non-candidate paths reduces the decoding complexity [7, 8, 9].

The Viterbi Algorithm performs the optimum decoding of the symbols on several types of channels. The Viterbi Decoder does not perform well on a bursty channel, but when used with an interleaved convolutional code, good performance can be achieved [1]. The interleaver function serves to disperse bursts of errors over several noncontiguous single bit errors.

2.4 Self-Timed Logic

Virtually all digital design today is based on a synchronous approach. The total system is designed as the combination of one or more subsystems where each subsystem is a clocked finite state machine; the subsystem changes from one state to the next on the

edges of a regular clock. The state is held in a set of flip-flops (registers), and combinatorial logic is used to derive the new state and outputs from the old state and inputs. The new state is copied through the flip-flops on every rising edge of the clock signal. Special techniques are required whenever a signal crosses into the domain of a particular clock (either from outside the system or from the domain of a different clock within the same system), but otherwise, provided a few rules are followed, the system behaves in a discrete and deterministic way. These rules include managing the delays of the combinatorial logic so that the flip-flop set-up and hold times are met under all conditions.

Self-timed (asynchronous) design does not follow this methodology; in general, there is no clock to govern the timing of state changes. Subsystems exchange information at mutually negotiated times with no external timing regulation. For the last two decades self-timed design has all but disappeared from sight. If global synchronization is becoming ever more difficult to establish [19], and the use of the clock causes increasing power inefficiencies, it seems natural to reconsider issues of self-timed versus synchronous design. In academia, self-timed techniques have always retained a niche as they provide a good framework for some mathematical techniques for proving the correctness of circuits [30,31]. Now, however, industrial research organizations are becoming active in self-timed design. There is a strong feeling that interest is increasing and self-timed approaches are ready to make a come-back [11,19,30,31,32].

Synchronous design can be viewed as a special case representing a single point in a multi-dimensional self-timed design space. There are many different types of self-timed

logic, and they are different from each other as they are from synchronous design. Self-timed circuits keep the assumption that signals are binary and remove the assumption that time is discrete. Self-timed circuits have several possible benefits [11, 19]:

- No clock skew.
- Low power.
- Average-case instead of worst-case performance.
- Easing of global timing issues.
- Better technology migration potential.
- Automatic adaptation to physical properties.
- Robust mutual exclusion and external input handling.
- Reduced high frequency components of noise.
- Reduced ground bounce and substrate crosstalk.
- Lower EMI (Electromagnetic Interference).

Even though most of the advantages of self-timed circuits are towards higher performance, it is not clear that self-timed circuits are actually better in practice. More research in this area is necessary [19].

“Self-timed design involves the use of functional units with a simple handshake interface for the passing of data and control status. Complex functions are realized by applying the use of handshaking protocols throughout a design. Figure 2.4 shows two functional units; one unit is defined as a sender who wishes to pass data, the other is the receiver who wishes to receive data” [11].

Operation of the circuit of Figure 2.4 is straightforward. The sender prepares the data and then notifies the receiver that data is available by asserting the request signal (Req.). The receiver consumes the data and asserts the acknowledge signal (Ack) to complete the transfer [11].

Figure 2.4 Self-Timed bounded data interface

2.5 Noise Sources in VLSI Systems

An understanding of the sources and implications of noise in VLSI systems is necessary to appreciate the benefits that self-timed logic can offer.

2.5.1 An Overview

Figure 2.5 shows a model of noise transmission from a source through a coupling channel to a susceptible receptor. A remedy for a noise problem consists of any combination of removing the source, eliminating the coupling channel, or improving the noise tolerance of the receptor [11].

Figure 2.5 Noise model

The means of coupling a noise source to a receptor can be categorized into conductive coupling, common impedance coupling, and coupling via radiated electromagnetic fields as shown Figure 2.6(a), Figure 2.6(b), Figure 2.6(c), Figure 2.6(d), Figure 2.6(e) [11,16]. Integrated circuits couple noise via all three mechanisms.

Substrate noise is an example of conductive coupling and is a result of conductivity of the substrate leading to current flow between a switching transistor (noise source) and another device [11,22].

Current switching through the package pins and power routing of an IC generate common impedance $L di/dt$ noise (Figure 2.6(b), Figure 2.6(c)). The source of this noise is the current required to switch internal logic and input/output buffers. Noise of this form has the potential to change the state of logic [11, 21].

Field coupling in ICs occurs (Figure 2.6(d), Figure 2.6(e)) in the form of crosstalk between conductors in close proximity. For example, nets in the VLSI layout or adjacent package bonding wires. Crosstalk aware design tools are required to minimize coupling between adjacent lines [24]. For the most part, crosstalk presents a similar problem for both synchronous and self-timed systems, although differences in average fanout for the two design styles does afford an advantage to self-timed logic as the peak current will be less.

Figure 2.6 Mechanisms for noise coupling

A final form of noise in VLSI circuits is EMI (electromagnetic interference). Governing bodies such as the Federal Communications Commission (FCC) in the United States regulate the amount of radiation an electronic device can emit. Meeting the regulatory requirements is important as EMI from a VLSI component can disrupt the performance of the systems, such as radio communication [11].

Radiation from digital circuits occurs as a result of current flowing around loops formed by the conductors of the circuit. The loops act as an antenna, radiating an electromagnetic field. The strength of the field is proportional to the amplitude of the current. Of equal importance is the frequency spectrum of the radiated field, since organizations such as the FCC specify emission limits at different frequencies [11, 46]. Meeting requirements such as those of the FCC can be extremely costly if not addressed early in the design cycle. The cost of eliminating noise problems increases dramatically as a project moves from design to production [16].

2.5.2 Substrate Noise and Packaging

The coupling channel for substrate noise is different for low and high resistivity substrates. “High resistivity substrates are common in Bipolar as GaAs processes while low resistivity substrates are required in CMOS processes to avoid latchup [25]. This thesis will focus on CMOS process technology. However, a brief review of substrate noise in high resistivity substrates is important, because CMOS behaves in this manner at high frequencies or for circuits in very close proximity” [11,16,17,18].

Circuits in very close proximity or circuits manufactured in high resistivity substrates are coupled by a direct horizontal path through the substrate material. As a result, guard rings around the receptor circuit are effective in isolating substrate noise of this form. An equally effective means of isolation is to ensure noise sensitive circuitry is placed as far as possible from logic circuits, or to alter the process and use a buried oxide layer to completely DC isolate the noise source [11,21,22,23].

The conductive path for low resistivity substrates, by comparison, is vertical with current passing directly through the substrate to the backplane inductance. Therefore, noise in a low resistivity substrate can follow a path from the noise source directly to the power routing via the substrate inductance. Any spatial relationship between a source and receptor is eliminated once noise propagates through globally routed power rails. Use of guard rings or increasing the separation between source and receptor are ineffective [25]. Therefore, it becomes important to reduce noise at the source for standard CMOS processes.

“Before moving on with a discussion of noise in CMOS low resistivity substrates, the mechanism for injection of substrate noise needs to be understood” [11]. For low resistivity CMOS processes, this mechanism is primarily through the injection of current due to hot electron effects. Hot electrons are observed when the field in the depleted drain end of a transistor is large enough to cause impact ionization and generate electron-hole pairs. For CMOS devices, the hot electron current will be injected into the substrate for

both 0-1 and 1-0 transitions [25]. Additional current can be expected due to capacitive coupling.

Components of the substrate model of Figure 2.5 will be identical in self-timed and synchronous systems for the same process. Ignoring the large clock driver current present in synchronous systems, but not in self-timed logic, the amount of substrate noise generated is a function of the current injected into the substrate and passing through the substrate to the ground plane inductance. Note that in the simple model of Figure 2.6(a), L_{gnd} consists of both the inductance of on-chip power routing and the inductance of the package bonding wires [11,24,26].

The voltage at the receiver of Figure 2.6(a) is given by the following equation [11]:

$$\begin{aligned}
 V_{recv} = & s^3 C_{sub1} C_{sub2} L_{gnd} R_{load} V_{inj} / \\
 & (1 + s C_{sub2} (s L_{gnd} + R_{2g} + R_{load})) + \\
 & s C_{sub1} (R_{1g} (1 + s C_{sub2} (R_{2g} + R_{load})) + \\
 & s L_{gnd} (1 + s C_{sub2} (R_{1g} + R_{2g} + R_{load})))
 \end{aligned} \tag{2-1}$$

The combined inductance of the chip power routing and package power leads (L_{gnd}) plays a critical role when considering substrate noise. The amount of isolation is very sensitive to the backplane impedance [25]. The transmission medium from one contact to another is via the on-chip power routing, and not directly through the substrate for low resistivity CMOS processes. Therefore, reduction of noise can most easily be accomplished through reducing the inductance of the package. The cost of electrically enhanced packaging often makes this a prohibitive solution [11].

Consider the following example data (See Table 2.1) for a low resistivity substrate [15], and use the model in Figure 2.6(a) with a contact separation of 10 microns. For an enhanced BGA package, the backplane inductance is 0.1nH, and the self inductance of package leads is 3.8nH each. Therefore, using an inductance of 5nH is a reasonable approximation (assuming 2 parallel power and ground connections).

Table 2-1 Substrate parameters

Parameter	Value
$R1_g = R2_g$	1800 Ω
R12	5000 Ω
Rload	50 Ω
$C1_g = C2_g$	0.8 nF
L	5 nH

Specifically, the substrate and package inductance combine to form a high pass filter. This can be expected as the circuit topology of Figure 2.6 behaves as a resistive network at high frequencies, since the impedance of the inductor is large and the impedance of the capacitor is small. VLSI designers are keenly aware of the need to keep the package inductance to a minimum. Advanced packages with features such as ground and power planes have been designed to improve electrical characteristics by reducing the inductance of power pins and controlling the impedance of I/O. Although the model of Figure 1.6 is very simple, the resulting transfer function is complex. The step response will be, as a minimum, a second order differential equation that can result in underdamped oscillation across the package inductance. This has been reported in commercial designs [17]. Designs such as the latest Pentium modules can alleviate this problem through expensive packaging [11], but this is not a viable solution for mobile products destined for the consumer marketplace.

The package inductance figures prominently in the behavior of substrate noise as noted above. The $L di/dt$ voltage drop across the package also generates common impedance noise which, in the past, has been reduced by improvements in packaging technology (e.g. flip chip), or by increasing the number of power and ground connections. High frequency components of the instantaneous current must be minimized to avoid resonance [11,44].

2.5.3 Crosstalk and Radio Frequency Radiation

Crosstalk presents a similar problem to the designers of self-timed and synchronous chips. Fast edge rates propagating down long interconnects requires complex transmission line modeling to determine the amount of coupling between adjacent lines. These equations are a function of the physical medium and layout topology, such as dielectric constants and distance between lines. The worst case capacitive coupling occurs when all nodes switch simultaneously. Techniques such as crosstalk aware layout tools, and the use of ground traces between sensitive signals, can be used to minimize crosstalk [23, 45, 47].

Digital circuits generate radiation as a result of current loops formed between logic gates as shown in Figure 2.7.

Figure 2.7 Digital Circuit EMI

For a circuit operating over a ground plane, the strength of the field in microvolts per meters is defined by the following equation [11]:

$$E = 263 \times 10^{-16} \left(f^2 AI \right) \frac{1}{r} \quad (2-2)$$

The equation shows that the radiation is proportional to the current (I), the loop area (A), and the square of the frequency (f). Therefore, radiation can be controlled by reducing the frequency or harmonic content of the current, reducing the magnitude of the current, or reducing the loop area. The self-timed circuit, however, uses local communication, and, therefore, would be expected to have a smaller loop area. The dependence of EMI on the square of the frequency results in the synchronous design emitting RF energy an order of magnitude greater than the self-timed design at high frequencies [11, 33, 46].

Chapter 3 Custom Decoder Chip Design

3.1 A 2/3 Convolutional Encoder

Implementation of a hard decision 2/3 convolutional coding is described in this chapter. The corresponding encoder is a part of a DAVIC TCM encoder. The TCM coding block of DAVIC 1.3.1 consists of the differential encoding, the convolutional encoding, and the signal mapping functional blocks as shown in Figure 3.1 [4], where m equals to $2q + 3$. A differential encoder encodes two out of the three MSBs. Two bits (Z_1 and Z_2) are then convolutionally encoded into three encoded bits (z_0, z_1, z_2).

Figure 3.1 Reference model of the TCM encoder

A detailed reference model of the convolutional encoder as part of the TCM encoder is shown in Figure 3.2. The convolutional encoder is a 16 states rate 2/3 encoder that works

at half the baud rate. The state diagram is shown in Figure 3.2, where z_0 is obtained from Z_1 , Z_2 and B (bits delay) using equations (3-1). B is an operator.

$$\begin{aligned} z_0 &= \frac{Z_2 B^3 + Z_1 B + Z_1 B^2}{1 + B + B^4} \\ z_1 &= Z_1 \\ z_2 &= Z_2 \end{aligned} \quad (3-1)$$

Delay of two clock cycles $2T$ corresponds to B in equation 3-1

Figure 3.2 Detailed reference model of the TCM encoder

Figure 3.3 DAVIC 1.3.1 TCM Trellis diagram

3.2 Design Implementation

3.2.1 Overview

The most significant problem in implementing a Viterbi Decoder is that the complexity increases exponentially with the constraint length k to give 2^{k-1} states [42]. Once the constraint length has been fixed, the code may be optimized. Although the increase in complexity with constraint length is a disadvantage, there are advantages as shown in Chapter 1. To implement the high speed TCM decoder, a high speed convolutional decoder must be included on the silicon die.

The Viterbi decoding has to follow multiple paths through the trellis, and remember them for future decisions. For larger decoders, such as cell phone standards with constraint lengths (shift-register length +1) of 6 and 9 [3], the number of states can become large. This means a decoding ASIC may take considerable storage, and a designer may have to use slower, more compact, RAM type memory rather than faster flip-flops.

The Viterbi algorithm allows many paths to be discarded without tracking them to completion. If several paths merge to one state, only the path with the minimum cumulative distance needs to be remembered. The other paths can never be part of the most likely path. This means the 2/3 Viterbi decoder only has to remember 16 paths. In general, a constraint length K system will only have to remember 2^{k-1} paths [1]. In theory, the path length should extend for the length of the message in order to get the true

maximum likelihood path. However, it has been discovered that path lengths of 4 to 5 times the constraint length almost always give the best path [1]. If two branches entering a state have equal cumulated distances, then the decoder is unable to tell if one path is more likely than another path, and the decoder must pick one path at random.

3.2.2 Block Diagram

Figure 3.4 shows the levels of hierarchy for Verilog simulation. The encoder module creates transmitted data. An error generator module creates random error bits. The ACS (Add Compare and Survivor) module calculates accumulated distances and discards bad paths. The Data_Carry_Forward module, which basically consists of mutiplexers and shift registers, sends the selected data from flip-flop to flip-flop down the paths one clock cycle at a time to generate the decoded data. The top-level flowchart is shown in Figure 3.5. Normally the encoder and decoder are widely separated so they cannot run from a common clock. Some sort of clock recovery system will make the clock common.

Figure 3.4 The block diagram of encoder/decoder

Figure 3.5 Decoder flowchart

3.2.3 ACS Module and Overflow Prevention

The ACS module calculates accumulated distances for every possible path. When several paths merge to one state, the path with the minimum accumulated distance is remembered. The others are discarded.

The decoder contains a large number of operators, such as adders, subtractors, and comparators. Reducing the number of bits in each operator is important in order to reduce the clock period and silicon area. Developing approaches of overflow prevention are important for reducing the number of bits in adders, subtractors, and comparators. In this chapter, an approach is developed to prevent overflow.

A Novel Overflow Prevention Approach

Let ΔH be the distance for every path from the current state to the next state that is called a “branch metrics”. Let H be the accumulated distance for the present state at time t and H_n be accumulated distance for next state at time $t + 1$. Then $H_n = H + \Delta H$. The maximum ΔH is 3. The minimum ΔH is 0. H is unbounded. This means H register must be very large. But $\text{Max}(H_i - H_j)$ for any two states i and j is bounded. In fact in this case it is 6 or less based on simulation.

In this algorithm, we check the minimum H for all states. When it reaches 2, instead of adding ΔH on the next cycle one subtracts $\sim\Delta H$, ($\sim$ means bit inversion, ΔH will be bit inverted). This turns out to be equivalent to adding ΔH and subtracting 3.

From

$$H - 0 = H + 3 - 3 \text{ for } \Delta H = 3 \text{ and } \sim\Delta H = 0 \quad (3-2)$$

$$H - 1 = H + 2 - 3 \text{ for } \Delta H = 2 \text{ and } \sim\Delta H = 1 \quad (3-3)$$

$$H - 2 = H + 1 - 3 \text{ for } \Delta H = 1 \text{ and } \sim\Delta H = 2 \quad (3-4)$$

$$H - 3 = H + 0 - 3 \text{ for } \Delta H = 0 \text{ and } \sim\Delta H = 3 \quad (3-5)$$

We have

$$H - (\sim\Delta H) = H + \Delta H - 3 \quad (3-6)$$

If the minimum accumulated Distance among every possible path is greater than 3, $H_n = H + \Delta H$ is coded as $H_n = H - (\sim\Delta H)$ for each path to prevent overflow.

Based on simulations, this approach maintains a maximum accumulated distance of 6. Therefore, 3 bit operators should be sufficient. For safety, 4 bit adders, subtractors, and comparators are used in the Verilog code.

The block diagram for the circuit to find the minimum accumulated distance among 16 surviving distances for 16 states is shown in Figure 3.6. The block diagram to find the surviving path (minimum accumulated distance) among 4 paths that merge to one state, for example state 0000, is shown in Figure 3.7.

Figure 3.6 The block diagram to find the minimum accumulated distance in the real chip

Figure 3.7 The block diagram to find the surviving path for state 0000 in the real chip

Modulo Normalization

Another common method to prevent overflow is Modulo Normalization. This method renormalizes the accumulated distance by designing the registers to store the accumulated distance so that the length of the registers is greater than $2\delta H_{\max}$, where $\delta H_{\max}$ is the maximum possible difference between accumulated distance [38]. The maximum possible difference ($\delta H_{\max}$) equals $2nB_1$, where n is minimum number of stages to ensure complete connectivity among the trellis states, and B_1 is the upper bound on the branch metrics (ΔH) [39, 40]. For the 2/3 convolutional decoder in this thesis, $2\delta H_{\max} = 2*2nB_1 = 2*2*2*3 = 24$. From Figure 3.3, $n = 2$ and $B_1 = 3$. Therefore, the length of the registers to store accumulated distance is 5 bits and 2^5 is greater than 24. The way to determine which accumulated distance is larger or smaller without worrying about overflows in the D flip-flops is described as follows:

Let $H_1[m:0]$ and $H_2[m:0]$ be the two accumulated distances to be compared, and $\delta H[m:0] = H_1[m:0] - H_2[m:0]$ using two's complement arithmetic, then $\delta H[m] = 1$ if $H_1[m:0] < H_2[m:0]$, $\delta H[m] = 0$ if $H_1[m:0] > H_2[m:0]$.

Figure 8 shows the block diagram to find the surviving path using Modulo Normalization among 4 paths that merge to one state, for example state 0000.

Figure 3.8 The block diagram to find the surviving path for state 0000 in Modulo Normalization

Pipelined ACS Module

One of the keys to the high-speed chip design is a pipelined structure. The advantage of a traditional structure without pipelines, as shown in Figure 3.6, 3.7, and 3.8, is its simplicity and the reduced number of sequential elements. The drawback of this approach is its reduced speed performance. The new pipelined structure is designed like a shift register to eliminate a few cycles of waiting time for data to go through combination logic. The increased sequential logic can be easily accommodated using an advanced sub-micron CMOS technology. Figure 3.9 shows the pipelined structure to find the minimum accumulated distance among 16 surviving distances for 16 states. Figure 3.10 shows the pipelined structure to find the surviving path (minimum cumulative distance) among 4 paths that merge to one state using the overflow prevention method developed in this thesis. Figure 3.11 shows the pipelined structure to find the surviving path using Modulo Normalization among 4 paths that merge to one state.

Figure 3.11 The pipelined structure to find a surviving path using Modulo Normalization

MUX Control Signals for Data_Carry_Forward Module

This ACS module generates a 2-bit signal called `came_from` for each present state. This signal records which of the possible states in the last step give rise to the surviving branch entering the current state. The `came_from` signals control multiplexers to carry the data forward. From the state diagram as shown in Figure 3.3, it can be seen that every state at $t+1$ can be reached by four possible states at t . For example, state 0000 at $t+1$ can be reached from state 0000, 0001, 0110, 0111 as shown in Figure 3.12. Let $H1$, $H2$, $H3$, $H4$ be four cumulative distances. The `came_from0` signal is associated with state 0000 at time $t+1$. The `came_from0` signal is assigned as follows:

<code>came_from0 = 0</code>	If $H1$ terminates the surviving branch
<code>came_from0 = 1</code>	If $H2$ terminates the surviving branch
<code>came_from0 = 2</code>	If $H3$ terminates the surviving branch
<code>came_from0 = 3</code>	If $H4$ terminates the surviving branch

came_from1, came_from2, ... , camefrom15 for state 0001, 0010, ..., 1111 are assigned in the same way.

Figure 3.12 State diagram for came_from0

3.2.4 Data_Carry_Forward module

The Data carry forward algorithm is based on a previous implementation [29]. The figures in this thesis which explain how Carry Path Forward works are shown with the author's permission. Figure 3.13 shows a simple example of code rate $\frac{1}{2}$ decoder. The transmitted signal from the encoder is 11 01 01 00 10 11 from $t = n$ to $t = n+6$. The corresponding received data of the decoder is 11 11 01 01 10 11. Two errors occur. Figure 3.14 shows several survivor paths at the beginning.

Figure 3.13 A simple example of a convolutional encoder

Figure 3.14 Survivor paths

Figure 3.15 shows how 4 bits, inserted in the four paths at $t=n+3$, can be sent from flip-flop to flip-flop down the paths one clock cycle at a time. At each cycle the data in a path stays the same, but its position (the state it is associated with) changes. These data bits carry forward through the paths in the trellis diagram, storing the data in flip-flops during each clock cycle. They arrive at the output at $t = n+8$, and the flip-flop in state C indicates there was a 0 in this path between $n = n+2$ and $n+3$. The same result is obtained by tracing the path backward in the Figure. The path for the best output is illustrated with a thicker line.

Figure 3.15 The Carry Path Forward

Figure 3.16 shows how the path tracing can be done in hardware. The paths are made by multiplexers. The state and inputs at $t=n+k$ control the MUX, which selects what will be clocked into the flip-flop at $t=n+k+1$. At $t=n+6$, only the shaded MUXes and associated flip-flops are in use. This means that three shift registers can be compacted into one.

Figure 3.16 The path tracing in hardware

Figure 3.17 shows the Shift Register Structure. The flip-flops contain the data for $t=n+6$. The four path edges selected at $t=n+6$ are illustrated with thicker lines.

Figure 3.17 Shift register structure

3.2.5 Latency

In communications, latency is the term for the time difference between the time the input signal is received and the time the output signal is sent out. In this design, the latency of the Data_Carry_Forward module is 20 clock cycles (4×5). Constraint length of the convolutional encoder is 5 (shift-register length + 1). The real chip implemented by CMC is not a pipelined structure. So its total latency is 21 clock cycles. The latency increases for a pipelined structure. For the pipelined Viterbi decoder using the overflow prevention

developed in this thesis, the total latency is 27 clock cycles. For the pipelined Viterbi decoder using Modulo Normalization, the total latency is 23 clock cycles.

3.2.6 Comparison

Adder

Five different operator implementations for adders and subtractors, Ripple (rpl), Carry Look-Ahead (cla), Fast Carry Look-Ahead (clf), Brent-Kung architecture (bk), Conditional Sum (csm), are compared. The maximum periods of different operator implementations from the Synopsys timing reports for the non-pipelined decoder are listed as follows for TSMC CMOS35 technology. There is minimal difference between the approaches as only 4-bit operators are used.

- csm: 11.70 ns.
- cla: 11.80 ns.
- bk: 11.94 ns.
- rpl: 11.97 ns.
- clf: 12.00 ns.

Theoretically, the fastest adder is the conditional-sum adder [26]. This is consistent with the results above.

Overflow Prevention

Table 3.1 shows comparison between the overflow prevention and Modulo Normalization for the pipelined decoder implementation. There is a tradeoff here. The silicon area of a 4-bit operator is smaller than that of a 5-bit operator with same functionality. The decoder with the overflow prevention developed in this thesis has 64 more 4-bit subtractors and 15 more 4-bit comparators. It is slightly bigger than the decoder with Modulo Normalization. The maximum frequencies (Half the Baud Rate) from the Synopsys timing report (See Appendix F for maximum frequency) are exactly the same with pipelines.

Table 3-1 The comparison for the different overflow prevention of the pipelined decoder

	Operator bits #	Size	Maximum Frequency
Modulo Normalization overflow prevention	5	0.130 million gates	125MHz
Overflow prevention in this thesis	3+1	0.132 million gates	125MHz

3.3 Performance of the 2/3 Viterbi Decoder

3.3.1 Union Bounds

An important measure of performance for error-correction coding is the probability of a bit error. The probability of a bit error, P_B , for a binary convolution code using hard-decision has an upper bound given by [41]:

$$P_B \leq \frac{dT(D, N)}{dN} \Big|_{N=1, D=2\sqrt{p(1-p)}} \quad (3-7)$$

where $T(D,N)$ is the transfer function, p is the probability of channel symbol error. D is the indeterminate “place holder” associated with the weight of the output sequence. The exponent of D represents the cumulative tally of the numbers of ones of the path, hence the Hamming distance from the all-zeros path. N is the indeterminate associated with the weight of the input sequence. The factor N for branch transitions is caused by the input bit of one, only if that branch is due to an input bit of one. Thus, as each branch is traversed, the cumulative exponent on N increases by one. If the number of data bits that form an input to an encoder, k , is greater than 1, both this expression and equation 3-8 must be divided by k .

For BPSK modulation over an additive white Gaussian noise (AWGN) channel, it can be shown that the bit error probability is bounded by [41]

$$P_B \leq Q\left(\sqrt{2d_f \frac{E_c}{N_0}}\right) \exp\left(d_f \frac{E_c}{N_0}\right) \frac{dT(D,N)}{dN} \Big|_{N=1, D=\exp(-E_c/N_0)} \quad (3-8)$$

where

$Q(x)$ is the complementary error function

$$E_c/N_0 = rE_b/N_0$$

E_b/N_0 = information bit energy / noise power spectral density

E_s/N_0 = channel symbol energy / noise power spectral density

$r = k/n$ = rate of the code

k is the number of input bits to the encoder

n is the total number of bits in the associated codeword out of the encoder.

$r = 2/3$ for the 2/3 convolutional code

d_f = minimum free distance. For the 2/3 convolutional code, $d_f = 3$.

To calculate the upper bound, the transfer function $T(D,N)$ has to be calculated first. One technique for evaluating $T(D,N)$, which is useful in performance estimation, is described as follows [42]. Each branch in the state diagram (see Figure 3.3) has a gain (g) given by $N^{w_i}D^{w_o}$, where w_i is the weight of the input required to drive the coder between the two nodes connected by the branch, and w_o is the weight of the output associated with the branch. State 0 (i.e. state 0000 in Hex) is separated into an original state 0, and a terminating state represented as state 16 (2^4). x_i represents the value of the accumulated path gain from state 0 to state i as influenced by all other states. $x_0 = 1$ to simplify calculation. $T(D,N) = x_{16}/x_0 = x_{16}$. The state equations are written as follows:

$$\begin{aligned}
 x_{16} &= DNx_1 + DNx_6 + D^2N^2x_7 \\
 x_2 &= DNx_0 + x_1 + D^2N^2x_6 + DNx_7 \\
 x_{12} &= DNx_0 + D^2N^2x_1 + x_6 + DNx_7 \\
 x_{14} &= D^2N^2x_0 + DNx_1 + DNx_6 + x_7 \\
 \\
 x_4 &= x_2 + DNx_3 + DNx_4 + D^2N^2x_5 \\
 x_6 &= DNx_2 + x_3 + D^2N^2x_4 + DNx_5 \\
 x_8 &= DNx_2 + D^2N^2x_3 + x_4 + DNx_5 \\
 x_{10} &= D^2N^2x_2 + DNx_3 + DNx_4 + x_5 \\
 \\
 x_5 &= D^2Nx_8 + D^3N^2x_9 + Dx_{14} + D^2Nx_{15} \\
 x_7 &= D^3N^2x_8 + D^2Nx_9 + D^2Nx_{14} + Dx_{15}
 \end{aligned}$$

(3-9)

$$x_9 = Dx_8 + D^2Nx_9 + D^2Nx_{14} + D^3N^2x_{15}$$

$$x_{11} = D^2Nx_8 + Dx_9 + D^3N^2x_{14} + D^2Nx_{15}$$

$$x_1 = D^2Nx_{10} + D^3N^2x_{11} + Dx_{12} + D^2Nx_{13}$$

$$x_3 = D^3N^2x_{10} + D^2Nx_{11} + D^2Nx_{12} + Dx_{13}$$

$$x_{13} = Dx_{10} + D^2Nx_{11} + D^2Nx_{12} + D^3N^2x_{13}$$

$$x_{15} = D^2Nx_{10} + Dx_{11} + D^3N^2x_{12} + D^2Nx_{13}$$

To deduce these state equations, for example, see Figure 3.18 for the second group of expressions in equation 3-9. These equations can be written in matrix form as

$$\mathbf{x} = \mathbf{A}\mathbf{x} + \mathbf{x}^0 \quad (3-10)$$

where

$$\mathbf{x}^T = (x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}, x_{11}, x_{12}, x_{13}, x_{14}, x_{15}, x_{16})$$

$$\mathbf{x}^{0T} = (0, DN, 0, 0, 0, 0, 0, 0, 0, 0, 0, DN, 0, D^2N^2, 0, 0)$$

$\mathbf{A}$ is a state transition matrix which can be deduced from the equation 3.9. See Appendix E for the state transition matrix and the Matlab code.

The formal solution to these equations can be written as

$$\mathbf{x} = (\mathbf{I} - \mathbf{A})^{-1} \mathbf{x}_0 \quad (3-11)$$

where $\mathbf{I}$ is an identity matrix.

Since the paths in the terminating state are represented by x_{16} , we have

$$T(D,N) = x_{16} \quad (3-12)$$

The results of this calculation for error bit probability versus E_b/N_0 using Matlab Symbolic Math Tool box are shown in Figure 3.19.

Two input bits Z_1 and Z_2 to the encoder : $Z_1=z_1$ $Z_2=z_2$ $z_0 z_1 z_2 \rightarrow D^{z_0+z_1+z_2} N^{z_1+z_2}$

Figure 3.18 Trellis diagram for deducing the second group of state expressions in equation 3-9

Figure 3.19 Bit error probability for the Viterbi decoding and BPSK modulation

3.3.2 Simulation Results

Another useful technique for estimating the performance of convolutional coding is computer simulation. Both union bounds and computer simulation can be used to predict

performance for specific hardware configurations. The block diagram for simulation is shown in Figure 3.20. The encoder is in Verilog code. The Viterbi decoder is the real implementation in Verilog code. The other blocks are implemented in C code. See Appendix E for C code. The Box-Muller Method is used to create additive white Gaussian noise [43]. The following expressions are used:

$$E_b = E_s/r, r = k/n = 2/3, E_s \text{ is normalized to } 1.$$

$$\text{SNR} = 10\log_{10}(E_b/N_0), N_0 = 10^{-\text{SNR}/10}/r.$$

$$\sigma = (N_0/2)^{1/2} = (10^{-\text{SNR}/10}/2r)^{1/2}, \sigma \text{ is the variance of Gaussian distribution.}$$

Agreement of simulation with theory

The simulation results are shown in Figure 3.19. Union bound is for soft decision. Three-bit soft decisions of the BSC channel output result in approximately a 2-dB gain over the hard the hard decision BSC [48], as quoted in [1]. These results are consistent with results from Jerrold A. Heller [48].

Figure 3.20 The block diagram of computer simulation for evaluating bit error probability

3.4 Design Process

3.4.1 RTL Model and Simulation

Modeling the design is the most important aspect of implementing an efficient design.

For RTL designs, the operations are specified to occur at each clock edge. It is more time

efficient to model RTL designs rather than gate-level designs. In this way, the full power of synthesis and optimization tools can be used to generate a final gate-level design.

The hierarchical design of the decoder module was captured using Verilog, and debugged and validated at the register-transfer level. In the development phase, both behavioral and structural descriptions are applied in the Verilog code. In the validation phase of the high-level design, the circuit RTL model is simulated using the Cadence-XL simulator to verify its functionality and prove that the RTL circuit description captures the design intention. Figure 3.21 shows an example of RTL level simulation result. Signals x, y, and z are data inputs of the Viterbi decoder with transmission errors. Signals dataout_0, dataout_1, and data_out2 are the decoder outputs which are the same as the encoder outputs. Two hundred thousand test vectors are simulated. One bit error is generated every five vectors. All of the errors are corrected by the decoder from simulation. The RTL code can be seen in Appendix A.

Figure 3.21 RTL level simulation results for error correction

3.4.2 Logic Synthesis

Logic synthesis is the process of transforming a circuit defined at a high level of abstraction into a gate level definition that is based on constraints.

Design Compiler

Synopsys Design Compiler is employed to synthesize the Verilog description of the decoder chip into a technology dependent, gate level design. Area and timing goals for the design are defined before the design is optimized. The most important parameter is the clock constraints. The clock period is specified as 20 ns at the beginning and then increased until timing slack is reduced to zero. The skew type of the clock is propagation skew. A 0.5 ns uncertainty “window” surrounding each clock edge is defined. It was reported that the chip area was 0.139 million NAND gates after synthesis for the non-pipelined circuit.

Test Compiler

Today, with gate counts pushing into the millions, ASIC design requires management of complex and difficult test issues. Design-For-Test (DFT) methodologies can save time and money, particularly since it has been estimated that the cost of manufacturing tests can represent as much as 30% of the total manufacturing cost. Scan-based DFT is used to improve the capability of diagnosing non-functioning chips in the decoder module [35].

The Synopsys tool, **Test Compiler**, is part of the Design analyzer tool. The Test Compiler substitutes all sequential devices (flip-flops) with scan equivalents, and connects them together to form a scan chain. The Test Compiler is then used to create a set of test vectors that can detect “stuck at 1” and “stuck at 0” faults in the chip. The scan methodology is full scan. Full scan typically provides high fault coverage and improved diagnostic capabilities [36].

Multiplexed D flip-flops are selected to implement the full scan test. The multiplexed flip-flop implementation uses a multiplexed data input to provide serial shift capability. During the functional operation, the scan enable signal, which acts as the multiplexer select line, selects the system data input. During scan shift, the scan enable signal selects the scan data input, which comes from either the scan input port or the scan output pin of the previous cell in the scan chain. For the implemented design, 98.6% fault coverage is reported.

Gate-level Simulation

At the end of logic synthesis, the gate level simulation is performed using Cadence Verilog-XL to confirm that the functionality is still the same as that of the original RTL code. This gate level simulation does not take into account the parasitic or loading effects. A test fixture written in Verilog is used as the testbench. The simulation results match the result of the original simulations.

3.4.3 Physical Design

Floorplanning, Placement, and Routing

Floorplaning and placement are performed using Cadence's **Physical Design planner (PDP)** that serves as a bridge between the front-end logic design tools and routing tools. Routing of standard cells is done using Silicon Ensemble. The operation gives a layout after overcoming a number of Cadence place and route problems.

Verification

After the physical design, verification is required to ensure that the design does not violate the rules of CMOS35 technology. Two important checks are done in Cadence DFII environment: Design Rule Check (DRC) and Layout Versus Schematic (LVS). DRC checks for design rule errors that may have occurred during P&R. Diva LVS is used to compare:

1. the final layout in the form of a DEF (Design Exchange Format) file created from Silicon Ensemble to
2. the “golden” Verilog netlist which is created after clock tree synthesis, and subsequently functionally verified.

Since Diva cannot work directly with a Verilog netlist, the netlist is first converted to a schematic view. Also, the DEF file is translated into a layout view for the purpose of

layout extraction. LVS ensures that all nets and gates defined in the schematic are present and connected in the layout.

Once all the verification is completed, a stream file is generated from the layout file. The stream is then submitted for circuit fabrication. The final chip size is 2.778mm by 2.778mm. The chip is packaged in a 68-pin PGA.

3.5 Test and Conclusions

The circuit is tested using a high-speed custom test board. The board layout and schematic can be seen in appendix D and E. The test system consists of a HP16500C 200MHz Pattern Generator, a 100M/35M State/Timing Analyzer, a HP E3610A DC Power Supply and the Device Under Test (DUT). The Pattern Generator provides programmable digital outputs that are used to stimulate and control the DUT. The Timing Analyzer acquired data and stored it at equal time intervals as shown in Figure 3.22. The non-pipelined decoder implementation runs at 50MHz (Half the Baud Rate). These results are compared to the simulation result as shown in Figure 3.21. The data flow diagram for the chip test is shown in Figure 3.23. The Data Pattern Generator loads data from a floppy disk. The State/Timing Analyzer saves data to a floppy disk. Twenty thousands vectors are tested. One bit error is generated every five vectors. All of errors are corrected. The hardware test shows that the non-pipelined decoder works successfully at 50MHz. The measured power dissipation is 0.18 Watt at 50MHz. The logic analyzer does not support a 80MHz test option.

As shown in this chapter, the overflow prevention developed in this thesis works well for VLSI implementation of the 2/3 Viterbi decoder, although 64 more subtractors and 15 more comparators are used. The number of operator bits for the overflow prevention method developed in this thesis is less than the number of operators bits for the Modulo Normalization method.

Figure 3.22 Test results of the non-pipelined decoder circuit

Figure 3.23 Data flowing diagram for the real decoder chip test

Chapter 4 Noise Effects in Digital Circuit

4.1 Technical Features of Self-Timed Design

Recently, academic research [27, 28] has demonstrated the feasibility and advantages of self-timed designs. Since there is no global clock signal, self-timed design has specific advantages for IC designers in terms of power and noise: 1) Reduced power consumption; 2) Reduced substrate noise; 3) Reduced ground noise.

“Power consumption is a well known problem in VLSI design. The battery life of mobile products can be extended by reducing the power consumption. Self-timed design reduces power by only consuming energy when useful work is done. Additional power savings result from elimination of the clock buffers and routing” [37]. Other benefits include reduced packaging costs and improved component reliability.

Designers of complex mixed signal circuitry face considerable challenges in meeting aggressive specifications such as the resolution of A/D converters, integration of dynamic memory, or jitter in phase-locked loops. This is particularly true if the use of low cost CMOS process is desired. The self-timed digital circuit from commercial companies shows a dramatic reduction in substrate noise and ground noise [11, 33].

4.2 An Novel Empirical Model and Simulations

Reducing operating voltages provides power saving but generally degrades the noise margin of digital circuits. Substrate and ground noise may have a stronger effect on logic states as operating voltage is reduced and operating frequency is increased [11, 22, 23]. Self-timed design has the potential to reduce the possibility of noise changing the states of logic. In this chapter it will be shown how noise in synchronous and self-timed digital circuits changes the state of logic, with an emphasis on low-voltage operation and high-speed data link issues.

4.2.1 Noise Effects on An Inverter

In order to check the noise effects, a circuit that generates noise (noise generator) and a circuit that receives noise are needed. The noise generator is modeled as a number of large inverters ($W/L = 200\mu\text{m}/0.5\mu\text{m}$, multiplier = 2), whose inputs are V_{n1} , V_{n1} , V_{n3} , V_{n4} , V_{n5} , that inject noise into a normal inverter ($W/L = 80\mu\text{m}/0.5\mu\text{m}$), whose input is V_{in} , as shown in Figure 4.1. The large inverters are equivalent to several noise generators around the normal inverter, whose logic state may be changed by noise from the noise generators. The transmission model of noise is shown in Figure 2.6(a). The substrate capacitance of the noise generators, which is equivalent to a large digital circuit around the normal inverter, is 0.8nF [11]. Equivalent substrate capacitance, which is connected with each noise generator (output of each big inverter), is $C_{\text{sub}} = 0.8\text{nF}/5 = 160\text{ pF}$. Output capacitance C_{load} is 10fF . The other substrate parameters are shown in Table 2.1.

A Synchronous System

A synchronous system is modeled as inputs to the noise generators switching simultaneously (i.e., $V_{n1} = V_{n2} = V_{n3} = V_{n4} = V_{n5}$). They are equivalent to one global clock signal in circuits that emit noise. Simulation results for operating voltage V_{dd} of 3V, 2V, 1V are shown in Figure 4.2 (a, b, c), respectively.

1) At $V_{dd} = 3V$, for high logic output, minimum output $V_{omin3} = 1.339V$, $V_{omin3}/V_{dd} = 0.446$, which is less than 0.5. Therefore, the logic state may be changed.

Figure 4.1 Circuit structure for noise effects on an inverter

2) At $V_{dd} = 2V$, for high logic output, minimum output $V_{omin2} = 1.128V$, $V_{omin2}/V_{dd} = 0.564$, which is greater than 0.5, and therefore, the logic state maintained.

3) At $V_{dd} = 1V$, for high logic output, minimum output $V_{omin1} = 0.746V$, $V_{omin1}/V_{dd} = 0.746$, which is greater than both 0.564 and 0.5. The logic state is maintained with a larger noise margin than that of $V_{dd} = 2V$.

Based on the simulation results from the model, it can be seen that the lower the operating voltage, the higher the noise immunization capability. At a lower operating voltage, less energy is coupled through the capacitor, since $\Delta q = c\Delta V$, and less noise is produced.

(a) 3V operating voltage

(b) 2V operating voltage

(c) 1V operating voltage

Figure 4.2 Simulation results for the synchronous system

A Self-Timed System

A self-timed system is modeled when inputs, V_{n1} , V_{n2} , V_{n3} , V_{n4} , and V_{n5} , of the noise generators are as shown in Figure 4.3. In these simulations, $V_{n1}(t) = V_{n2}(t+\Delta t) = V_{n3}(t+2\Delta t) = V_{n3}(t+3\Delta t) = V_{n5}(t+4\Delta t)$ and $\Delta t = 0.3\text{ns}$. Simulation results for operating voltage V_{dd} of 2.9V, 2V and 1V are shown in Figure 4.3 and Figure 4.4(a, b), respectively.

- (1) At $V_{dd} = 2.9\text{V}$, for high logic output, minimum output $V_{omin3} = 2.273\text{V}$, $V_{omin3}/V_{dd} = 0.784$, which is greater than 0.5 and therefore the logic state is maintained.
- (2) At $V_{dd} = 2\text{V}$, for high logic output, minimum output $V_{omin2} = 1.651\text{V}$, $V_{omin2}/V_{dd} = 0.826$, which is greater than both 0.784 and 0.5. The logic state is maintained with increased noise margin compared with (1).
- (3) At $V_{dd} = 1\text{V}$, for high logic output, minimum output $V_{omin1} = 0.918\text{V}$, $V_{omin1}/V_{dd} = 0.918$, which is greater than both 0.826 and 0.5. The logic state is maintained with increased noise margin compared with (2).

Based on the simulation results from the model, the lower operating voltage means a higher noise immunization capability. From this model, the self-timed system has better noise immunity than the synchronous system as there is no global clock signal and the peak switched current decreases significantly.

Figure 4.3 Simulation results at 2.9V operating voltage for the self-timed system

(a) 2V operating voltage

(b) 1V operating voltage

Figure 4.4 Simulation results for the self-timed system

4.2.2 Noise Effects on a D Flip-Flop

On-Chip Simulation Results

In digital circuits, a D flip-flop is a basic storage component. As shown in Figure 4.5, a circuit is simulated to determine noise effects on a D flip-flop. The input noise waveforms are similar to the previous inverter case. The I/O pads are models as inverters, whose input is V_{pad} . The size of all transistors for I/O pads is extracted from CMC padsp35/wpadout by reverse engineering. $W/L=41.9\mu\text{m}/0.5\mu\text{m}$, multiplier=12*16 for P channel transistors, where 16 means 16 pads. $W/L=26\mu\text{m}/0.5\mu\text{m}$, multiplier=11*16 for N channel transistors.

Figure 4.5 The circuit structure for noise effects on a D flip-flop

For the synchronous system, Figure 4.6(a) shows that the falling edge flip-flop works successfully, since data is captured completely at an operating voltage of 1.2V and a clock frequency of 333 MHz. Figure 4.6(b) shows that the flip-flop does not work properly since partial data is lost at a frequency of 357 MHz for the same operating voltage.

For the self-timed system, Figure 4.7(a) shows that the falling edge flip-flop works successfully, since data is captured completely at an operating voltage of 1.4V and a clock frequency of 500 MHz. Figure 4.7(b) shows that the flip-flop does not work properly at a frequency of 526 MHz for the same operating voltage.

Figure 4.8 shows the maximum on-chip frequency versus operating voltage for both the synchronous system and the self-timed system.

Based on the first order model presented in Chapter 2, it can be seen that a self-timed system can work at a higher frequency than a synchronous system. From Figure 4.8, it can be seen that, at higher frequencies, a self-timed system performs significantly better. The magnitude and high frequency content of the current is significantly lower in self-timed designs than in equivalent synchronous designs [11, 33]. Since the substrate and package are effectively a high-pass filter, as shown in equation (2.1), eliminating the high frequency components of the current spectrum reduces the peak ground voltage due to substrate coupling [33].

(a) The falling edge D flip-flop works successfully at $f = 333$ MHz

(b) The falling edge D flip-flop does not work properly at $f = 357$ MHz

Figure 4.6 Simulation of the falling edge D flip-flop works at $V_{dd}=1.2V$ in the synchronous system

(a) The D flip-flop works successfully at $f = 500\text{MHz}$

(b) The D flip-flop does not work properly at $f = 526\text{MHz}$

Figure 4.7 Simulation of the falling edge D flip-flop works at $V_{dd}=1.4\text{V}$ in the self-timed system

The noise transmission medium from a source to susceptible receptor is the same for both the synchronous and self-timed systems. Self-time technology reduces the peak noise emitted by a source since communication in a self-timed design is local and there is no global signal.

Figure 4.8 Maximum on-chip clock frequency for the D flip-flop to work properly versus operating voltage

Off-Chip Simulation Results

Off-chip simulation signals after the on-chip D flip-flop output signal passes a typical driver with a load capacitance of 7.5 pF are determined in order to compare simulation results with hardware test results. The driver is extracted from a CMC pads35 output pad and consists of two stages. The first stage is a small inverter with $W/L=24.9\mu\text{m}/0.5\mu\text{m}$ and multiplier = 4 for a P channel transistor, $W/L=26\mu\text{m}/0.5\mu\text{m}$ and multiplier = 4 for a N channel transistor. The second stage is a large inverter with $W/L=41.9\mu\text{m}/0.5\mu\text{m}$ and multiplier = 12 for a P channel transistor, $W/L=26\mu\text{m}/0.5\mu\text{m}$ and multiplier = 11 for a N channel transistor.

Figure 4.9 shows the maximum off-chip frequency versus operating voltage for both the synchronous system and the self-timed system

Figure 4.9 Maximum off-chip frequency versus operating voltage

4.3 VLSI Implementation

A VLSI implementation of a novel circuit to verify the above simulation results is shown in Figure 4.10. The noise generator (a series of 96 D flip-flops, i.e. a 6*16 bit register) injects noise into a separate D flip-flop, which is a noise receptor. Logic state “1” on the select pin of the multiplexer corresponds to a synchronous system (All noise source flip-flops have the same clock). Logic state “0” corresponds to a self-timed system with delayed clocks to each registers.

Figure 4.10 VLSI implementation to verify simulation results

4.3.1 Test Results

The circuit is tested using a high-speed custom test board. The clock input is connected to ground through a 50 ohm resistor. The test system consists of a HP80000 Data generator System/HP E2903A 1GHz Data Module, a Tektronix TDS 684B 1GHz/s Digital Real-time Oscilloscope, a HP E3610A DC Power Supply and the Device Under Test (DUT). The Data Generator provides programmable digital outputs that are used to stimulate and control the DUT. Figure 4.11 shows the test setup. Figure 4.12 shows the input waveform to the D flip-flops.

Figure 4.11 Test Setup

Figure 4.12 Input signal waveform of the D flip-flop

Figure 4.13 shows the D flip-flop (noise receptor) output waveform at 415MHz frequency, and 1.7V operating voltage for the self-timed case. The output waveform is the same as the input waveform and the D flip-flop is working correctly.

Figure 4.13 The D flip-flop output waveform at 415MHz frequency and 1.7V operating voltage for self-timed case

Figure 4.14 shows the D flip-flop output waveform at 415MHz frequency and 1.7V operating voltage for the synchronous case. Part of the input data is not captured. The D flip-flop does not work correctly due to the noise.

Figure 4.14 The D flip-flop output waveform at 415MHz frequency and 1.7V operating voltage for synchronous case

Figure 4.15 shows the maximum frequency versus the operating voltage for which the D flip-flop worked properly for both self-timed and synchronous cases. It can be seen that the self-timed circuit works better than the synchronous circuit. The relative difference between the two lines is smaller than the relative difference from simulation results as shown in Figure 4.9. This is due to the fact that the convolutional decoder circuit and the noise circuit are on a single chip and use a common input clock signal. The synchronous

decoder clock tree buffers are on and inject noise to the noise circuit even when the noise circuit is in self-timed mode. Compared to Figure 4.9, there is lower frequency seen in Figure 4.15 for a certain operating voltage due to electronic noise from test equipment and noise from the test environment. This is the reason we have a difference between the simulation results as shown in Figure 4.9 and the test results as shown in Figure 4.15.

Figure 4.15 Maximum frequency at which the D flip-flop works properly versus operating voltage

4.4 Noise Between I/O Drivers

Current switching through the package pins and power routing of an IC generate common impedance Ldi/dt noise (Figure 2.6(a, b, c), Figure 4.1). The source of this noise is the current required to switch internal logic and input / output buffers [11, 19, 33,]. The noise effect between I/O drivers is investigated below.

The circuit structure for noise effects between I/O circuits is the same as the structure shown in Figure 4.1, except that ten I/O pad drivers (noise generator) are modeled as ten pairs of large inverters (large W/L ratio) which inject noise into one I/O pad. The size of all transistors is extracted from CMC padsp35/wpadout by reverse engineering. $W/L=41.9\mu\text{m}/0.5\mu\text{m}$, multiplier=12 for P channel transistors. $W/L=26\mu\text{m}/0.5\mu\text{m}$, multiplier=11 for N channel transistors.

For the synchronous system, simulation results for operating voltages of 1V, 2V, 3V and C_{load} of 1pF and 10pF are shown in Figure 4.16 - 4.17. At a 3V supply and 1pF load capacitance, the minimum logic "1" output voltage V_{min} is less than half of operating voltage. Therefore, noise may change the state of logic. Two main advantages are seen from the simulation results:

- 1) Lower operating voltage provides reduced noise effects.
- 2) Larger load capacitance provides reduced noise effects. Transient noise from other input / output buffers changes the threshold voltage of the receiver. The NMOS transistor may be temporarily on, due to the change in ground potential, which discharges C_{load} . The larger load capacitance and time constant provide better noise immunity.

(a) $V_{dd}=1V$ (b) $V_{dd}=2V$ (c) $V_{dd}=3V$ **Figure 4.16 Simulation results for the synchronous system at $C_{load}=1pF$**

Figure 4.17 Simulation results for the synchronous system at $C_{load}=10\text{pF}$

In a self-timed system, simulation results for operating voltages of 1V, 2V, 2.5V and C_{load} of 1pF and 10pF are shown in Figure 4.18 - 4.19. In all cases, the state of logic is maintained.

Overall, I/O pads of a self-timed system have a better noise performance than those of a synchronous system as can be seen by comparing Figures 4.16, 17 and Figures 4.18, 4.19. This can be explained as follows: the voltage across the package inductance of Figure 2.6(a) is the ground noise for the VLSI chip that occurs as a result of current conducting through the substrate. "The reduction of high frequency current components for the self-timed system results in the reduced of transients in the ground voltage due to substrate conduction".

(a) $V_{dd}=1V$ (b) $V_{dd}=2V$ (c) $V_{dd}=2.5V$ **Figure 4.18 Simulation results for the self-timed system at $C_{load}=1pF$**

(a) $V_{dd}=1V$ (b) $V_{dd}=2V$ (c) $V_{dd}=2.5V$ **Figure 4.19** Simulation results for the self-timed system at $C_{load}=10pF$

Chapter 5 Conclusions and Future Work

5.1 Conclusions

In Chapters 2 and 3 of the thesis, the basic theory of the Viterbi encoder/decoder is reviewed, and it is shown that the decoder can be efficiently implemented using a pipelined structure. The state diagram is extracted from the DAVIC 1.3.1 encoder circuit that works at half the baud rate. An efficient implementation called Carry Path Forward is employed. A novel approach to prevent overflow is used to reduce the number of bits for adders, subtractors, and comparators. The performance of the 2/3 convolutional coding is estimated using Union Bounds and computer simulation. The results from different operator implementations are also compared in order to improve the speed. A maximum frequency of 120 MHz of the pipelined 2/3 Viterbi decoder from the Synopsys timing report is reported using CMOS35 technology. This corresponds to a baud rate of 240 MHz.

In Chapter 2 of the thesis, the potential advantages of self-timed circuits are reviewed. One of the proposed advantages of self-timed designs is that they can reduce substrate noise and ground noise. In Chapter 4, noise effects in self-timed and synchronous digital circuits are modeled, analyzed, and compared, with an emphasis on low operating voltage and high speed issues. It is shown that a self-timed system can provide better noise immunity than a synchronous system, particularly in high speed ICs. For I/O drivers, I/O

pads in a self-timed system also have a better noise performance than those of a synchronous system.

5.2 Future Work

Future work will include a self-timed implementation of the Viterbi decoder that will show a direct comparison with the synchronous implementation. Another area of interest will be the lower power aspect of asynchronous circuits. Standard synchronous circuits have to toggle clock lines, and possibly precharge and discharge signals, in portions of a circuit unused in the current computation. Asynchronous circuits generally have transitions only in areas involved in the current computation.

References

1. Sklar, Digital Communication, pp. 345-365, chapter 6, 1988.
2. Edwards, Gwyn, "Forward Error Correction Encoding and Decoding", Stanford Telecom Application Note 108, 1990.
3. Henry Hendrix, "Viterbi Decoding Techniques in the TMS320C54x Family", SPRA071, June 1996, Texas Instruments.
4. DAVIC 1.4.1 Specification Part 8, "Lower Layer Protocols and Physical Interfaces", Digital Audio-Video Council, 1999, Geneva, Switzerland.
5. Michelson, A.M., and Levesque, A.H., Error-Control Techniques for Digital Communications, John Wiley & Sons, 1985.
6. A. J. Viterbi, "Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm", IEEE Trans. Inf. Theory, Vol. IT13, April 1967, pp. 260-269.
7. J. K. Omura, "On the Viterbi Decoding Algorithm", IEEE Trans. Inf. Theory, Vol. IT15, Jan. 1969, pp. 177-179.
8. Mansoor A. Chishtie, "Viterbi Implementation on the TMS320C5x for V.32 Modems", Digital Signal Processing Applications – Semiconductor Group, Texas Instruments Incorporated, 1996 (<http://www.ti.com>).
9. Roberto Alini, "A 200-Msamples/s Trellis-Coded PRML Read/Write Channel with Analog Adaptive Equalizer and Digital Servo", IEEE J. of Solid State Circuits, Vol. 32, No. 11, November 1997.

10. Kuntal Joardar, "A Simple Approach to Modeling Cross-Talk in Integrated Circuits", *IEEE J. of Solid State Circuits*, Vol.29, No.10, October 1994.
11. Warren Lien, Craig Farnsworth, Ron Glibbery, David Jackson, Nigel Paver, "Noise in a Self-timed and Synchronous Implementation of a DSP", Private communication, Cogency Technology Incorporated, 1999.
12. B. M. J. Kup, E. C. Dijkmans, P. J. A. Naus, and J. Snee, "A Bit Stream Digital-to-Analog Converter with 18-b Resolution", *IEEE J. Solid State Circuits*, Vol. 26, No. 12, pp. 1757-1763, Dec. 1991.
13. Clemenz L. Portmann, Teresa H. Y. Meng, "Supply Noise and CMOS Synchronization Errors", *IEEE J. Solid State Circuits*, Vol. 30, No. 9, pp. 1757-1763, Sept. 1995.
14. Tekla S. Perry, Linda Geppert, "Do Portable Electronics Endanger Flight ?", *IEEE Spectrum*, Vol.33, No. 9, pp26-33, Sep 1996.
15. H. B. Bakoglu, "Circuits, Interconnections, and Packaging for VLSI", Addison-Wesley, 1990.
16. Henry W. Ott, "Noise Reduction Techniques in Electronic Systems", p. 298, John Wiley & Sons, 1989.
17. Nishath K. Verghese, David J. Allstot, Mark A. Wilfe, "Verification Techniques for Substrate Coupling and Their Application to Mixed-Signal IC Design", *IEEE J. Solid State Circuits*, Vol. 31, No. 3, pp. 354-365, Mar. 1996.
18. David K. Su, Marc J. Loinaz, Shoichi Masui, Bruce A. Wooley, "Experimental Results and Modeling Techniques for Substrate Noise in Mixed-Signal Integrated Circuits", *IEEE J. of Solid State Circuits*, Vol. 28, No.4, April 1993.

19. Scott Hauck, "Asynchronous Design Methodologies: An Overview", Proceedings of the IEEE, Vol. 83, No. 1, pp. 69-93, January 1995.
20. P. Larson, "Power Supply Noise in Future IC's: A Crystal Ball Reading", IEEE 1999 Custom Integrated Circuits Conference 23.1.1, page 467.
21. Akio Koyama, Masatoshi Tsuge, Jun'ya Kudo, Tatsuhiro Aido, Makio Uchida, "Switching Well Noise Analysis and Minimization Strategy for Low Vth CMOS Integrated Circuits", IEEE 1999 Custom Integrated Circuits Conference 23.2.1, pp. 475.
22. John P.Z. Lee, Frank Wang, Abhijit Phanse and Linda C. Smith, "Substrate Cross Talk Noise Characterization and Prevention in 0.35um CMOS Technology", IEEE 1999 Custom Integrated Circuits Conference 23.3.1, pp. 479.
23. J. Briaire and K. S. Krisch, "Substrate Injection and Crosstalk in CMOS Circuits", IEEE 1999 Custom Integrated Circuits Conference 23.4.1, pp. 483.
24. Ashok Vittal, Malgorzata Marek-Sadowska, "Crosstalk Reduction for VLSI", IEEE J. Solid State Circuits, Vol. 29, No. 10, pp. 1212-1219, Oct., 1994.
25. Ranjit Gharpurey and Robert G. Meyer, "Modeling and Analysis of Substrate Coupling in Integrated Circuits", IEEE J. Solid State Circuits, Vol. 31, No. 3, pp. 344-353, Mar. 1996.
26. Bellaouar and Elmasary, Low-Power Digital VLSI Design, Kluwer, pp. 423-428.
27. S. B. Furber, P. Day, J. D. Garside, N. C. Paver, S. Temple, J. V. Woods, "The Design and Evaluation of an Asynchronous Microprocessor", Proc. of ICCD'94, pp. 217-220, Oct. 1994.

28. S. B. Furber, P. Day, J. D. Garside, J. Liu, N. C. Paver, S. Temple, "AMULET2e: An Asynchronous Embedded Controller", Proc. of Advanced Research in Asynchronous Circuits and Systems, Eindhoven, April 1997.
29. John Knight, Department of Electronics, Carleton Univ.. Personal communication.
30. S. M. Nowick, M. B. Joseph, and C. H. van Berkel, "Scanning the Special Issue on Asynchronous Circuits and Systems", Proceedings of the IEEE, February 1999, vol. 87, No. 2, pp. 219.
31. C. H. van Berkel, M. B. Joseph, S. M. Nowick, " Applications of Asynchronous Circuits", Proceedings of the IEEE, February 1999, vol. 87, No. 2 pp. 223.
32. M. B. Joseph, S. M. Nowick, C. H. van Berkel, "Modeling and Design of Asynchronous Circuits", Proceedings of the IEEE, February 1999, vol. 87, No. 2, pp. 234.
33. N. C. Paver, E. Cornu, P. Day, R. Glibbery, W. A. Lien, J. liu, "A Low-Noise, Self-timed DSP", 7th NASA Symposium Session 3, 1998, Asynchronous Topic, Cogency Technology Incorporated, 1999.
34. Olaf J. Joeressen, Heinrich Meyr, "A 40 Mb/s Soft-Output Viterbi Decoder", IEEE Journal of Solid-State Circuits, Vol. 30, No. 7, July 1995, pp. 812.
35. Tie Peng, "VLSI Implementation of High-Speed Cellular Automata Encryption Algorithm for Data Communications", M.ENG Thesis, Carleton University, pp. 62, 1999.
36. Synopsys Online Manual: Scan Synthesis User Guide v 1998.08.
37. Cogency Web Page www.cogency.com.
38. Hui-ling lou, "Implementing the Viterbi Algorithm", IEEE Signal Processing Magazine, pp. 42, September 1995.

39. C. Bernard Shung, Gottfried Ungerboeck, Hemant K. Thapar, "VLSI Architectures for Metric Normalization in the Viterbi algorithm", In: 1990 International Conference on Communications, Atlanta, Georgia, pp1732, April 1990 IEEE.
40. Andries P. Hekstra, "An Alternative to Metric Rescaling in Viterbi Decoders", IEEE Transactions on Communications, pp1220, Vol. 37, No.11, November 1989.
41. Andrew J. Viterbi, "Convolutional Codes and Their Performance in Communication Systems", IEEE Transactions on Communications Technology, Vol. COM19, No. 5, Oct. 1971, pp. 751-772.
42. George Clark, Error-correction Coding for Digital Communications. 1981, Plenum press, pp. 241.
43. Raj Jain, The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling, pp. 494.
44. H. B. Bakglu, Circuits, Interconnections, and Packaging for VLSI, 1990, Addison-Wesley.
45. Edoardo Charbon, Ranjit Gharpurey, Paolo Miliozzi, Robert G. Meyer, Alberto Sangiovanni-Vincentelli, Substrate Noise: Analysis and Optimization for IC Design, 2001, Kluwer Academia Publishers, pp99-133
46. Xavier Aragones, Jose Luis Gonzalez, Antonio Rubio, Analysis and Solutions for Switching Noise Coupling in Mixed-signal ICs, 1999, Kluwer Academia Publishers, pp. 160.
47. Willy Sansen, Soc Design from the Mixed-Signal Perspective, 27th European Solid-States Circuits Conference, Villach, Austria, September 2001.

48. Jerrold A. Heller, Irwin M. Jacobs, "Viterbi Decoding for Satellite and Space Communication", *IEEE Transactions on Communication Technology*, Vol. COM-19, Oct., 1971.

Appendix A: Verilog code

```

module thesischip(x, y, z, clk, reset, test_se, test_si, in_noise, sel,
in_inv, in_dff, en_decoder, en_noise, dataout_0, dataout_1, dataout_2,
out_inv, out_dff, d5);

input x, y, z, clk, reset, test_se, test_si;
wire x, y, z, clk, reset, test_se, test_si;

input in_noise, sel, in_inv, in_dff, en_decoder, en_noise;
wire in_noise, sel, in_inv, in_dff, en_decoder, en_noise;

output out_inv, out_dff;
wire out_inv, out_dff;
output [15:0] d5;
wire [15:0] d5;

output dataout_0, dataout_1, dataout_2;
wire dataout_0, dataout_1, dataout_2;

wire x_top, y_top, z_top, clk_top, reset_top, se_top, si_top;
wire dataout_0_top, dataout_1_top, dataout_2_top;

wire en_decoder_top, en_noise_top;

wire in_noise_top, sel_top, in_inv_top, in_dff_top;
wire out_inv_top, out_dff_top;
wire [15:0] d5_top;

thesis thesis(x_top, y_top, z_top, clk_top, reset_top, se_top, si_top,
in_noise_top, sel_top, in_inv_top, in_dff_top, en_decoder_top,
en_noise_top, dataout_0_top, dataout_1_top, dataout_2_top, out_inv_top,
out_dff_top, d5_top);

PDI px( .C(x_top), .PAD(x) );
PDI py( .C(y_top), .PAD(y) );
PDI pz( .C(z_top), .PAD(z) );
PDI pclk( .C(clk_top), .PAD(clk) );
PDI preset( .C(reset_top), .PAD(reset) );
PDI pse( .C(se_top), .PAD(test_se) );
PDI psi( .C(si_top), .PAD(test_si) );

PDI pen_decoder( .C(en_decoder_top), .PAD(en_decoder) );
PDI pen_noise( .C(en_noise_top), .PAD(en_noise) );

PDI pin_noise( .C(in_noise_top), .PAD(in_noise) );
PDI psel( .C(sel_top), .PAD(sel) );
PDI pin_inv( .C(in_inv_top), .PAD(in_inv) );
PDI pin_dff( .C(in_dff_top), .PAD(in_dff) );

```

```

PDT24C pdata_out0 ( .PAD(dataout_0), .I(dataout_0_top),
.OEN(en_decoder_top) );
PDT24C pdata_out1 ( .PAD(dataout_1), .I(dataout_1_top),
.OEN(en_decoder_top) );
PDT24C pdata_out2 ( .PAD(dataout_2), .I(dataout_2_top),
.OEN(en_decoder_top) );

PDT24C pout_inv ( .PAD(out_inv), .I(out_inv_top), .OEN(en_noise_top) );
PDT24C pout_dff ( .PAD(out_dff), .I(out_dff_top), .OEN(en_noise_top) );
PDT24C pd5_00 ( .PAD(d5[0]), .I(d5_top[0]), .OEN(en_noise_top) );
PDT24C pd5_01 ( .PAD(d5[1]), .I(d5_top[1]), .OEN(en_noise_top) );
PDT24C pd5_02 ( .PAD(d5[2]), .I(d5_top[2]), .OEN(en_noise_top) );
PDT24C pd5_03 ( .PAD(d5[3]), .I(d5_top[3]), .OEN(en_noise_top) );
PDT24C pd5_04 ( .PAD(d5[4]), .I(d5_top[4]), .OEN(en_noise_top) );
PDT24C pd5_05 ( .PAD(d5[5]), .I(d5_top[5]), .OEN(en_noise_top) );
PDT24C pd5_06 ( .PAD(d5[6]), .I(d5_top[6]), .OEN(en_noise_top) );
PDT24C pd5_07 ( .PAD(d5[7]), .I(d5_top[7]), .OEN(en_noise_top) );
PDT24C pd5_08 ( .PAD(d5[8]), .I(d5_top[8]), .OEN(en_noise_top) );
PDT24C pd5_09 ( .PAD(d5[9]), .I(d5_top[9]), .OEN(en_noise_top) );
PDT24C pd5_10 ( .PAD(d5[10]), .I(d5_top[10]), .OEN(en_noise_top) );
PDT24C pd5_11 ( .PAD(d5[11]), .I(d5_top[11]), .OEN(en_noise_top) );
PDT24C pd5_12 ( .PAD(d5[12]), .I(d5_top[12]), .OEN(en_noise_top) );
PDT24C pd5_13 ( .PAD(d5[13]), .I(d5_top[13]), .OEN(en_noise_top) );
PDT24C pd5_14 ( .PAD(d5[14]), .I(d5_top[14]), .OEN(en_noise_top) );
PDT24C pd5_15 ( .PAD(d5[15]), .I(d5_top[15]), .OEN(en_noise_top) );

endmodule

module thesis(x, y, z, clk, reset, test_se, test_si, in_noise, sel,
in_inv, in_dff, en_decoder, en_noise, dataout_0, dataout_1, dataout_2,
out_inv, out_dff, d5);

input x, y, z, clk, reset, test_se, test_si;
wire x, y, z, clk, reset, test_se, test_si;

input in_noise, sel, in_inv, in_dff;

input en_decoder, en_noise;

output out_inv, out_dff;
output [15:0] d5;
wire [15:0] d5;

output dataout_0, dataout_1, dataout_2;
wire dataout_0, dataout_1, dataout_2;

topdecoder topdecoder(x, y, z, clk, reset, test_se, test_si, dataout_0,
dataout_1, dataout_2);

noise noise(in_noise, sel, clk, reset, in_inv, in_dff, out_inv,
out_dff, d5);

endmodule

```

```

module topdecoder(x, y, z, clk, reset, test_se, test_si, dataout_0,
dataout_1, dataout_2);

input x, y, z, clk, reset, test_se, test_si;
wire x, y, z, clk, reset;

output dataout_0, dataout_1, dataout_2;
wire dataout_0, dataout_1, dataout_2;

wire [1:0] camefrom0, camefrom1, camefrom2, camefrom3;
wire [1:0] camefrom4, camefrom5, camefrom6, camefrom7;
wire [1:0] camefrom8, camefrom9, camefrom10, camefrom11;
wire [1:0] camefrom12, camefrom13, camefrom14, camefrom15;

Calc_Nxt_Sta_and_Ham_Dist Calc(x,y,z, clk, reset,
camefrom0, camefrom1, camefrom2, camefrom3,
camefrom4, camefrom5, camefrom6, camefrom7,
camefrom8, camefrom9, camefrom10, camefrom11,
camefrom12, camefrom13, camefrom14, camefrom15);

Data_Carry_Forward Data_Carry_forward1(
camefrom0, camefrom1, camefrom2, camefrom3,
camefrom4, camefrom5, camefrom6, camefrom7,
camefrom8, camefrom9, camefrom10, camefrom11,
camefrom12, camefrom13, camefrom14, camefrom15,
clk, reset, dataout_0, dataout_1, dataout_2);

endmodule

module Calc_Nxt_Sta_and_Ham_Dist(x,y,z, clk, reset,
camefrom0, camefrom1, camefrom2, camefrom3,
camefrom4, camefrom5, camefrom6, camefrom7,
camefrom8, camefrom9, camefrom10, camefrom11,
camefrom12, camefrom13, camefrom14, camefrom15);

input x,y,z,clk,reset;
wire x,y,z,clk,reset;
output [1:0] camefrom0, camefrom1, camefrom2, camefrom3;
output [1:0] camefrom4, camefrom5, camefrom6, camefrom7;
output [1:0] camefrom8, camefrom9, camefrom10, camefrom11;
output [1:0] camefrom12, camefrom13, camefrom14, camefrom15;

reg [1:0] camefrom0, camefrom1, camefrom2, camefrom3;
reg [1:0] camefrom4, camefrom5, camefrom6, camefrom7;
reg [1:0] camefrom8, camefrom9, camefrom10, camefrom11;
reg [1:0] camefrom12, camefrom13, camefrom14, camefrom15;

reg [1:0] camefrom0_ini0, camefrom1_ini0, camefrom2_ini0,
camefrom3_ini0;
reg [1:0] camefrom4_ini0, camefrom5_ini0, camefrom6_ini0,
camefrom7_ini0;

```



```

reg [1:0] camefrom8_ini0, camefrom9_ini0, camefrom10_ini0,
camefrom11_ini0;
reg [1:0] camefrom12_ini0, camefrom13_ini0, camefrom14_ini0,
camefrom15_ini0;

```

```

reg [1:0] camefrom0_ini1, camefrom1_ini1, camefrom2_ini1,
camefrom3_ini1;
reg [1:0] camefrom4_ini1, camefrom5_ini1, camefrom6_ini1,
camefrom7_ini1;
reg [1:0] camefrom8_ini1, camefrom9_ini1, camefrom10_ini1,
camefrom11_ini1;
reg [1:0] camefrom12_ini1, camefrom13_ini1, camefrom14_ini1,
camefrom15_ini1;

```

```

reg [1:0] camefrom0_ini0_Q, camefrom0_ini1_Q;
reg [1:0] camefrom1_ini0_Q, camefrom1_ini1_Q;
reg [1:0] camefrom2_ini0_Q, camefrom2_ini1_Q;
reg [1:0] camefrom3_ini0_Q, camefrom3_ini1_Q;
reg [1:0] camefrom4_ini0_Q, camefrom4_ini1_Q;
reg [1:0] camefrom5_ini0_Q, camefrom5_ini1_Q;
reg [1:0] camefrom6_ini0_Q, camefrom6_ini1_Q;
reg [1:0] camefrom7_ini0_Q, camefrom7_ini1_Q;
reg [1:0] camefrom8_ini0_Q, camefrom8_ini1_Q;
reg [1:0] camefrom9_ini0_Q, camefrom9_ini1_Q;
reg [1:0] camefrom10_ini0_Q, camefrom10_ini1_Q;
reg [1:0] camefrom11_ini0_Q, camefrom11_ini1_Q;
reg [1:0] camefrom12_ini0_Q, camefrom12_ini1_Q;
reg [1:0] camefrom13_ini0_Q, camefrom13_ini1_Q;
reg [1:0] camefrom14_ini0_Q, camefrom14_ini1_Q;
reg [1:0] camefrom15_ini0_Q, camefrom15_ini1_Q;

```

```

reg [3:0] H0, H1, H2, H3, H4, H5, H6, H7;
reg [3:0] H8, H9, H10, H11, H12, H13, H14, H15;
reg [3:0] H0next, H1next, H2next, H3next, H4next, H5next, H6next,
H7next;
reg [3:0] H8next, H9next, H10next, H11next, H12next, H13next, H14next,
H15next;

```

```

reg [1:0] h000, h001, h010, h011, h100, h101, h110, h111;
reg [1:0] hh000, hh001, hh010, hh011, hh100, hh101, hh110, hh111;

```

```

reg [3:0] H0h000, H0h001, H0h010, H0h011, H1h000, H1h001, H1h010, H1h011;
reg [3:0] H2h000, H2h001, H2h010, H2h011, H3h000, H3h001, H3h010, H3h011;
reg [3:0] H4h000, H4h001, H4h010, H4h011, H5h000, H5h001, H5h010, H5h011;
reg [3:0] H6h000, H6h001, H6h010, H6h011, H7h000, H7h001, H7h010, H7h011;
reg [3:0] H8h100, H8h101, H8h110, H8h111, H9h100, H9h101, H9h110, H9h111;
reg [3:0] H10h100, H10h101, H10h110, H10h111, H11h100, H11h101,
H11h110, H11h111;
reg [3:0] H12h100, H12h101, H12h110, H12h111, H13h100, H13h101,
H13h110, H13h111;
reg [3:0] H14h100, H14h101, H14h110, H14h111, H15h100, H15h101,
H15h110, H15h111;

```

```

reg [3:0] H0h000_Q, H0h001_Q, H0h010_Q, H0h011_Q, H1h000_Q, H1h001_Q,
H1h010_Q, H1h011_Q;
reg [3:0] H2h000_Q, H2h001_Q, H2h010_Q, H2h011_Q, H3h000_Q, H3h001_Q,
H3h010_Q, H3h011_Q;

```

```

reg [3:0] H4h000_Q, H4h001_Q, H4h010_Q, H4h011_Q, H5h000_Q, H5h001_Q,
H5h010_Q, H5h011_Q;
reg [3:0] H6h000_Q, H6h001_Q, H6h010_Q, H6h011_Q, H7h000_Q, H7h001_Q,
H7h010_Q, H7h011_Q;
reg [3:0] H8h100_Q, H8h101_Q, H8h110_Q, H8h111_Q, H9h100_Q, H9h101_Q,
H9h110_Q, H9h111_Q;
reg [3:0] H10h100_Q, H10h101_Q, H10h110_Q, H10h111_Q, H11h100_Q,
H11h101_Q, H11h110_Q, H11h111_Q;
reg [3:0] H12h100_Q, H12h101_Q, H12h110_Q, H12h111_Q, H13h100_Q,
H13h101_Q, H13h110_Q, H13h111_Q;
reg [3:0] H14h100_Q, H14h101_Q, H14h110_Q, H14h111_Q, H15h100_Q,
H15h101_Q, H15h110_Q, H15h111_Q;

reg [3:0] H0next_ini0, H1next_ini0, H2next_ini0, H3next_ini0;
reg [3:0] H4next_ini0, H5next_ini0, H6next_ini0, H7next_ini0;
reg [3:0] H8next_ini0, H9next_ini0, H10next_ini0, H11next_ini0;
reg [3:0] H12next_ini0, H13next_ini0, H14next_ini0, H15next_ini0;

reg [3:0] H0next_ini1, H1next_ini1, H2next_ini1, H3next_ini1;
reg [3:0] H4next_ini1, H5next_ini1, H6next_ini1, H7next_ini1;
reg [3:0] H8next_ini1, H9next_ini1, H10next_ini1, H11next_ini1;
reg [3:0] H12next_ini1, H13next_ini1, H14next_ini1, H15next_ini1;

reg [3:0] H0next_ini0_Q, H1next_ini0_Q, H2next_ini0_Q, H3next_ini0_Q;
reg [3:0] H4next_ini0_Q, H5next_ini0_Q, H6next_ini0_Q, H7next_ini0_Q;
reg [3:0] H8next_ini0_Q, H9next_ini0_Q, H10next_ini0_Q, H11next_ini0_Q;
reg [3:0] H12next_ini0_Q, H13next_ini0_Q, H14next_ini0_Q, H15next_ini0_Q;

reg [3:0] H0next_ini1_Q, H1next_ini1_Q, H2next_ini1_Q, H3next_ini1_Q;
reg [3:0] H4next_ini1_Q, H5next_ini1_Q, H6next_ini1_Q, H7next_ini1_Q;
reg [3:0] H8next_ini1_Q, H9next_ini1_Q, H10next_ini1_Q, H11next_ini1_Q;
reg [3:0] H12next_ini1_Q, H13next_ini1_Q, H14next_ini1_Q, H15next_ini1_Q;

reg [3:0] min0, min1, min2, min3, min4, min5, min6, min7;
reg [3:0] min8, min9, min10, min11, min12, min13;
reg [3:0] min;

reg [3:0] min_Q;
reg [3:0] min0_Q;
reg [3:0] min1_Q;
reg [3:0] min2_Q;
reg [3:0] min3_Q;
reg [3:0] min4_Q;
reg [3:0] min5_Q;
reg [3:0] min6_Q;
reg [3:0] min7_Q;
reg [3:0] min8_Q;
reg [3:0] min9_Q;
reg [3:0] min10_Q;
reg [3:0] min11_Q;
reg [3:0] min12_Q;
reg [3:0] min13_Q;

always @(H0 or H1 or H2 or H3 or H4 or H5 or H6 or H7 or
H8 or H9 or H10 or H11 or H12 or H13 or H14 or H15)
begin: b2

```

```

    if(H0 > H1)
    min0 = H1; else min0 = H0;
    if(H2 > H3)
    min1 = H3; else min1 = H2;
    if(H4 > H5)
    min2 = H5; else min2 = H4;
    if(H6 > H7)
    min3 = H7; else min3 = H6;
    if(H8 > H9)
    min4 = H9; else min4 = H8;
    if(H10 > H11)
    min5 = H11; else min5 = H10;
    if(H12 > H13)
    min6 = H13; else min6 = H12;
    if(H14 > H15)
    min7 = H15; else min7 = H14;
end

always @(min0_Q or min1_Q or min2_Q or min3_Q or min4_Q or min5_Q or
min6_Q or min7_Q)
begin: b3
    if(min0_Q > min1_Q)
    min8 = min1_Q; else min8 = min0_Q;
    if(min2_Q > min3_Q)
    min9 = min3_Q; else min9 = min2_Q;
    if(min4_Q > min5_Q)
    min10 = min5_Q; else min10 = min4_Q;
    if(min6_Q > min7_Q)
    min11 = min7_Q; else min11 = min6_Q;
end

always @(min8_Q or min9_Q or min10_Q or min11_Q)
begin: b4
    if(min8_Q > min9_Q)
    min12 = min9_Q; else min12 = min8_Q;
    if(min10_Q > min11_Q)
    min13 = min11_Q; else min13 = min10_Q;
end

always @(min12_Q or min13_Q)
begin: b5

    if(min12_Q > min13_Q)
    min = min13_Q; else min = min12_Q;

end

always @(x or y or z or H0 or H1 or H2 or H3 or H4 or H5 or H6 or H7 or
H8 or H9 or H10 or H11 or H12 or H13 or H14 or H15 or min_Q)
begin: b1

    case({x, y, z})

```

```

        3'b000: begin
h000=0;h001=1;h010=1;h011=2;h100=1;h101=2;h110=2;h111=3; end
        3'b001: begin
h000=1;h001=0;h010=2;h011=1;h100=2;h101=1;h110=3;h111=2; end
        3'b010: begin
h000=1;h001=2;h010=0;h011=1;h100=2;h101=3;h110=1;h111=2; end
        3'b011: begin
h000=2;h001=1;h010=1;h011=0;h100=3;h101=2;h110=2;h111=1; end
        3'b100: begin
h000=1;h001=2;h010=2;h011=3;h100=0;h101=1;h110=1;h111=2; end
        3'b101: begin
h000=2;h001=1;h010=3;h011=2;h100=1;h101=0;h110=2;h111=1; end
        3'b110: begin
h000=2;h001=3;h010=1;h011=2;h100=1;h101=2;h110=0;h111=1; end
        3'b111: begin
h000=3;h001=2;h010=2;h011=1;h100=2;h101=1;h110=1;h111=0; end
    endcase

    if( min_Q > 2 )
        begin
            hh000 = ~h000; hh001 = ~h001; hh010 = ~h010; hh011 = ~h011;
            hh100 = ~h100; hh101 = ~h101; hh110 = ~h110; hh111 = ~h111;
//state 0
            H0h000 = H0 - hh000;
            H1h001 = H1 - hh001;
            H6h010 = H6 - hh010;
            H7h011 = H7 - hh011;

//state 1
            H10h110 = H10 - hh110;
            H11h111 = H11 - hh111;
            H12h100 = H12 - hh100;
            H13h101 = H13 - hh101;

//state 2
            H0h001 = H0 - hh001;
            H1h000 = H1 - hh000;
            H6h011 = H6 - hh011;
            H7h010 = H7 - hh010;

//state 3
            H10h111 = H10 - hh111;
            H11h110 = H11 - hh110;
            H12h101 = H12 - hh101;
            H13h100 = H13 - hh100;

//state 4
            H2h000 = H2 - hh000;
            H3h001 = H3 - hh001;
            H4h010 = H4 - hh010;
            H5h011 = H5 - hh011;

//state 5
            H8h110 = H8 - hh110;
            H9h111 = H9 - hh111;
            H14h100 = H14 - hh100;

```

```

    H15h101 = H15 - hh101;

//state 6
    H2h001 = H2 - hh001;
    H3h000 = H3 - hh000;
    H4h011 = H4 - hh011;
    H5h010 = H5 - hh010;

//state 7
    H8h111 = H8 - hh111;
    H9h110 = H9 - hh110;
    H14h101 = H14 - hh101;
    H15h100 = H15 - hh100;

//state 8
    H2h010 = H2 - hh010;
    H3h011 = H3 - hh011;
    H4h000 = H4 - hh000;
    H5h001 = H5 - hh001;

// state 9
    H8h100 = H8 - hh100;
    H9h101 = H9 - hh101;
    H14h110 = H14 - hh110;
    H15h111 = H15 - hh111;

//state 10
    H2h011 = H2 - hh011;
    H3h010 = H3 - hh010;
    H4h001 = H4 - hh001;
    H5h000 = H5 - hh000;

//state 11
    H8h101 = H8 - hh101;
    H9h100 = H9 - hh100;
    H14h111 = H14 - hh111;
    H15h110 = H15 - hh110;

//state 12
    H0h010 = H0 - hh010;
    H1h011 = H1 - hh011;
    H6h000 = H6 - hh000;
    H7h001 = H7 - hh001;

//state 13
    H10h100 = H10 - hh100;
    H11h101 = H11 - hh101;
    H12h110 = H12 - hh110;
    H13h111 = H13 - hh111;

// state 14
    H0h011 = H0 - hh011;
    H1h010 = H1 - hh010;
    H6h001 = H6 - hh001;
    H7h000 = H7 - hh000;

//state 15

```

```

    H10h101 = H10 - hh101;
    H11h100 = H11 - hh100;
    H12h111 = H12 - hh111;
    H13h110 = H13 - hh110;

    end

    else begin

//state 0
    H0h000 = H0 + h000;
    H1h001 = H1 + h001;
    H6h010 = H6 + h010;
    H7h011 = H7 + h011;

//state 1
    H10h110 = H10 + h110;
    H11h111 = H11 + h111;
    H12h100 = H12 + h100;
    H13h101 = H13 + h101;

//state 2
    H0h001 = H0 + h001;
    H1h000 = H1 + h000;
    H6h011 = H6 + h011;
    H7h010 = H7 + h010;

//state 3
    H10h111 = H10 + h111;
    H11h110 = H11 + h110;
    H12h101 = H12 + h101;
    H13h100 = H13 + h100;

//state 4
    H2h000 = H2 + h000;
    H3h001 = H3 + h001;
    H4h010 = H4 + h010;
    H5h011 = H5 + h011;

//state 5
    H8h110 = H8 + h110;
    H9h111 = H9 + h111;
    H14h100 = H14 + h100;
    H15h101 = H15 + h101;

//state 6
    H2h001 = H2 + h001;
    H3h000 = H3 + h000;
    H4h011 = H4 + h011;
    H5h010 = H5 + h010;

//state 7
    H8h111 = H8 + h111;
    H9h110 = H9 + h110;
    H14h101 = H14 + h101;
    H15h100 = H15 + h100;

```

```

//state 8
    H2h010 = H2 + h010;
    H3h011 = H3 + h011;
    H4h000 = H4 + h000;
    H5h001 = H5 + h001;

//state 9
    H8h100 = H8 + h100;
    H9h101 = H9 + h101;
    H14h110 = H14 + h110;
    H15h111 = H15 + h111;

//state 10
    H2h011 = H2 + h011;
    H3h010 = H3 + h010;
    H4h001 = H4 + h001;
    H5h000 = H5 + h000;

//state 11
    H8h101 = H8 + h101;
    H9h100 = H9 + h100;
    H14h111 = H14 + h111;
    H15h110 = H15 + h110;

// state 12
    H0h010 = H0 + h010;
    H1h011 = H1 + h011;
    H6h000 = H6 + h000;
    H7h001 = H7 + h001;

//state 13
    H10h100 = H10 + h100;
    H11h101 = H11 + h101;
    H12h110 = H12 + h110;
    H13h111 = H13 + h111;

//state 14
    H0h011 = H0 + h011;
    H1h010 = H1 + h010;
    H6h001 = H6 + h001;
    H7h000 = H7 + h000;

//state 15
    H10h101 = H10 + h101;
    H11h100 = H11 + h100;
    H12h111 = H12 + h111;
    H13h110 = H13 + h110;

    end

end

always @(
H0h000_Q or H0h001_Q or H0h010_Q or H0h011_Q or H1h000_Q or H1h001_Q or
H1h010_Q or H1h011_Q or
H2h000_Q or H2h001_Q or H2h010_Q or H2h011_Q or H3h000_Q or H3h001_Q or
H3h010_Q or H3h011_Q or

```

```

H4h000_Q or H4h001_Q or H4h010_Q or H4h011_Q or H5h000_Q or H5h001_Q or
H5h010_Q or H5h011_Q or
H6h000_Q or H6h001_Q or H6h010_Q or H6h011_Q or H7h000_Q or H7h001_Q or
H7h010_Q or H7h011_Q or
H8h100_Q or H8h101_Q or H8h110_Q or H8h111_Q or H9h100_Q or H9h101_Q or
H9h110_Q or H9h111_Q or
H10h100_Q or H10h101_Q or H10h110_Q or H10h111_Q or H11h100_Q or
H11h101_Q or H11h110_Q or H11h111_Q or
H12h100_Q or H12h101_Q or H12h110_Q or H12h111_Q or H13h100_Q or
H13h101_Q or H13h110_Q or H13h111_Q or
H14h100_Q or H14h101_Q or H14h110_Q or H14h111_Q or H15h100_Q or
H15h101_Q or H15h110_Q or H15h111_Q )
    begin: b6
//state 0
    if(H0h000_Q < H1h001_Q)
        begin H0next_ini0 = H0h000_Q; camefrom0_ini0 = 0; end
    else
        begin H0next_ini0 = H1h001_Q; camefrom0_ini0 = 1; end

    if(H6h010_Q < H7h011_Q)
        begin H0next_ini1 = H6h010_Q; camefrom0_ini1 = 2; end
    else
        begin H0next_ini1 = H7h011_Q; camefrom0_ini1 = 3; end

//state1
    if(H10h110_Q < H11h111_Q)
        begin H1next_ini0 = H10h110_Q; camefrom1_ini0 = 0; end
    else
        begin H1next_ini0 = H11h111_Q; camefrom1_ini0 = 1; end

    if(H12h100_Q < H13h101_Q)
        begin H1next_ini1 = H12h100_Q; camefrom1_ini1 = 2; end
    else
        begin H1next_ini1 = H13h101_Q; camefrom1_ini1 = 3; end

//state 2
    if(H0h001_Q < H1h000_Q)
        begin H2next_ini0 = H0h001_Q; camefrom2_ini0 = 0; end
    else
        begin H2next_ini0 = H1h000_Q; camefrom2_ini0 = 1; end

    if(H6h011_Q < H7h010_Q)
        begin H2next_ini1 = H6h011_Q; camefrom2_ini1 = 2; end
    else
        begin H2next_ini1 = H7h010_Q; camefrom2_ini1 = 3; end

//state 3
    if(H10h111_Q < H11h110_Q)
        begin H3next_ini0 = H10h111_Q; camefrom3_ini0 = 0; end
    else
        begin H3next_ini0 = H11h110_Q; camefrom3_ini0 = 1; end

    if(H12h101_Q < H13h100_Q)
        begin H3next_ini1 = H12h101_Q; camefrom3_ini1 = 2; end
    else

```



```

        begin H3next_ini1 = H13h100_Q; camefrom3_ini1 = 3; end

//state 4
    if(H2h000_Q < H3h001_Q)
        begin H4next_ini0 = H2h000_Q; camefrom4_ini0 = 0; end
    else
        begin H4next_ini0 = H3h001_Q; camefrom4_ini0 = 1; end

    if(H4h010_Q < H5h011_Q)
        begin H4next_ini1 = H4h010_Q; camefrom4_ini1 = 2; end
    else
        begin H4next_ini1 = H5h011_Q; camefrom4_ini1 = 3; end

//state 5
    if(H8h110_Q < H9h111_Q)
        begin H5next_ini0 = H8h110_Q; camefrom5_ini0 = 0; end
    else
        begin H5next_ini0 = H9h111_Q; camefrom5_ini0 = 1; end

    if(H14h100_Q < H15h101_Q)
        begin H5next_ini1 = H14h100_Q; camefrom5_ini1 = 2; end
    else
        begin H5next_ini1 = H15h101_Q; camefrom5_ini1 = 3; end

//state 6
    if(H2h001_Q < H3h000_Q)
        begin H6next_ini0 = H2h001_Q; camefrom6_ini0 = 0; end
    else
        begin H6next_ini0 = H3h000_Q; camefrom6_ini0 = 1; end

    if(H4h011_Q < H5h010_Q)
        begin H6next_ini1 = H4h011_Q; camefrom6_ini1 = 2; end
    else
        begin H6next_ini1 = H5h010_Q; camefrom6_ini1 = 3; end

//state 7
    if(H8h111_Q < H9h110_Q)
        begin H7next_ini0 = H8h111_Q; camefrom7_ini0 = 0; end
    else
        begin H7next_ini0 = H9h110_Q; camefrom7_ini0 = 1; end

    if(H14h101_Q < H15h100_Q)
        begin H7next_ini1 = H14h101_Q; camefrom7_ini1 = 2; end
    else
        begin H7next_ini1 = H15h100_Q; camefrom7_ini1 = 3; end

//state 8
    if(H2h010_Q < H3h011_Q)
        begin H8next_ini0 = H2h010_Q; camefrom8_ini0 = 0; end
    else
        begin H8next_ini0 = H3h011_Q; camefrom8_ini0 = 1; end

```

```

    if(H4h000_Q < H5h001_Q)
        begin H8next_ini1 = H4h000_Q; camefrom8_ini1 = 2; end
    else
        begin H8next_ini1 = H5h001_Q; camefrom8_ini1 = 3; end

//state 9
    if(H8h100_Q < H9h101_Q)
        begin H9next_ini0 = H8h100_Q; camefrom9_ini0 = 0; end
    else
        begin H9next_ini0 = H9h101_Q; camefrom9_ini0 = 1; end

        if(H14h110_Q < H15h111_Q)
            begin H9next_ini1 = H14h110_Q; camefrom9_ini1 = 2; end
        else
            begin H9next_ini1 = H15h111_Q; camefrom9_ini1 = 3; end

//state 10
    if(H2h011_Q < H3h010_Q)
        begin H10next_ini0 = H2h011_Q; camefrom10_ini0 = 0; end
    else
        begin H10next_ini0 = H3h010_Q; camefrom10_ini0 = 1; end

        if(H4h001_Q < H5h000_Q)
            begin H10next_ini1 = H4h001_Q; camefrom10_ini1 = 2; end
        else
            begin H10next_ini1 = H5h000_Q; camefrom10_ini1 = 3; end

//state 11
    if(H8h101_Q < H9h100_Q)
        begin H11next_ini0 = H8h101_Q; camefrom11_ini0 = 0; end
    else
        begin H11next_ini0 = H9h100_Q; camefrom11_ini0 = 1; end

        if(H14h111_Q < H15h110_Q)
            begin H11next_ini1 = H14h111_Q; camefrom11_ini1 = 2; end
        else
            begin H11next_ini1 = H15h110_Q; camefrom11_ini1 = 3; end

//state 12
    if(H0h010_Q < H1h011_Q)
        begin H12next_ini0 = H0h010_Q; camefrom12_ini0 = 0; end
    else
        begin H12next_ini0 = H1h011_Q; camefrom12_ini0 = 1; end

        if(H6h000_Q < H7h001_Q)
            begin H12next_ini1 = H6h000_Q; camefrom12_ini1 = 2; end
        else
            begin H12next_ini1 = H7h001_Q; camefrom12_ini1 = 3; end

//state 13
    if(H10h100_Q < H11h101_Q)
        begin H13next_ini0 = H10h100_Q; camefrom13_ini0 = 0; end

```

```

else
    begin H13next_ini0 = H11h101_Q; camefrom13_ini0 = 1; end

    if(H12h110_Q < H13h111_Q)
        begin H13next_ini1 = H12h110_Q; camefrom13_ini1 = 2; end
    else
        begin H13next_ini1 = H13h111_Q; camefrom13_ini1 = 3; end

//state 14
    if(H0h011_Q < H1h010_Q)
        begin H14next_ini0 = H0h011_Q; camefrom14_ini0 = 0; end
    else
        begin H14next_ini0 = H1h010_Q; camefrom14_ini0 = 1; end

    if(H6h001_Q < H7h000_Q)
        begin H14next_ini1 = H6h001_Q; camefrom14_ini1 = 2; end
    else
        begin H14next_ini1 = H7h000_Q; camefrom14_ini1 = 3; end

//state 15
    if(H10h101_Q < H11h100_Q)
        begin H15next_ini0 = H10h101_Q; camefrom15_ini0 = 0; end
    else
        begin H15next_ini0 = H11h100_Q; camefrom15_ini0 = 1; end

    if(H12h111_Q < H13h110_Q)
        begin H15next_ini1 = H12h111_Q; camefrom15_ini1 = 2; end
    else
        begin H15next_ini1 = H13h110_Q; camefrom15_ini1 = 3; end

end

always @(
H0next_ini0_Q or H1next_ini0_Q or H2next_ini0_Q or H3next_ini0_Q or
H4next_ini0_Q or H5next_ini0_Q or H6next_ini0_Q or H7next_ini0_Q or
H8next_ini0_Q or H9next_ini0_Q or H10next_ini0_Q or H11next_ini0_Q or
H12next_ini0_Q or H13next_ini0_Q or H14next_ini0_Q or H15next_ini0_Q or

H0next_ini1_Q or H1next_ini1_Q or H2next_ini1_Q or H3next_ini1_Q or
H4next_ini1_Q or H5next_ini1_Q or H6next_ini1_Q or H7next_ini1_Q or
H8next_ini1_Q or H9next_ini1_Q or H10next_ini1_Q or H11next_ini1_Q or
H12next_ini1_Q or H13next_ini1_Q or H14next_ini1_Q or H15next_ini1_Q or

camefrom0_ini0_Q or camefrom0_ini1_Q or
camefrom1_ini0_Q or camefrom1_ini1_Q or
camefrom2_ini0_Q or camefrom2_ini1_Q or
camefrom3_ini0_Q or camefrom3_ini1_Q or
camefrom4_ini0_Q or camefrom4_ini1_Q or
camefrom5_ini0_Q or camefrom5_ini1_Q or
camefrom6_ini0_Q or camefrom6_ini1_Q or
camefrom7_ini0_Q or camefrom7_ini1_Q or
camefrom8_ini0_Q or camefrom8_ini1_Q or
camefrom9_ini0_Q or camefrom9_ini1_Q or
camefrom10_ini0_Q or camefrom10_ini1_Q or

```

```

camefrom11_ini0_Q or camefrom11_ini1_Q or
camefrom12_ini0_Q or camefrom12_ini1_Q or
camefrom13_ini0_Q or camefrom13_ini1_Q or
camefrom14_ini0_Q or camefrom14_ini1_Q or
camefrom15_ini0_Q or camefrom15_ini1_Q )

    begin: b7
//state 0
    if(H0next_ini0_Q < H0next_ini1_Q)
        begin H0next = H0next_ini0_Q; camefrom0 = camefrom0_ini0_Q; end
    else
        begin H0next = H0next_ini1_Q; camefrom0 = camefrom0_ini1_Q; end

//state 1
    if(H1next_ini0_Q < H1next_ini1_Q)
        begin H1next = H1next_ini0_Q; camefrom1 = camefrom1_ini0_Q; end
    else
        begin H1next = H1next_ini1_Q; camefrom1 = camefrom1_ini1_Q; end

//state 2
    if(H2next_ini0_Q < H2next_ini1_Q)
        begin H2next = H2next_ini0_Q; camefrom2 = camefrom2_ini0_Q; end
    else
        begin H2next = H2next_ini1_Q; camefrom2 = camefrom2_ini1_Q; end

//state 3
    if(H3next_ini0_Q < H3next_ini1_Q)
        begin H3next = H3next_ini0_Q; camefrom3 = camefrom3_ini0_Q; end
    else
        begin H3next = H3next_ini1_Q; camefrom3 = camefrom3_ini1_Q; end

//state 4
    if(H4next_ini0_Q < H4next_ini1_Q)
        begin H4next = H4next_ini0_Q; camefrom4 = camefrom4_ini0_Q; end
    else
        begin H4next = H4next_ini1_Q; camefrom4 = camefrom4_ini1_Q; end

//state 5
    if(H5next_ini0_Q < H5next_ini1_Q)
        begin H5next = H5next_ini0_Q; camefrom5 = camefrom5_ini0_Q; end
    else
        begin H5next = H5next_ini1_Q; camefrom5 = camefrom5_ini1_Q; end

//state 6
    if(H6next_ini0_Q < H6next_ini1_Q)
        begin H6next = H6next_ini0_Q; camefrom6 = camefrom6_ini0_Q; end
    else
        begin H6next = H6next_ini1_Q; camefrom6 = camefrom6_ini1_Q; end

//state 7
    if(H7next_ini0_Q < H7next_ini1_Q)
        begin H7next = H7next_ini0_Q; camefrom7 = camefrom7_ini0_Q; end
    else
        begin H7next = H7next_ini1_Q; camefrom7 = camefrom7_ini1_Q; end

//state 8
    if(H8next_ini0_Q < H8next_ini1_Q)

```

```

        begin H8next = H8next_ini0_Q; camefrom8 = camefrom8_ini0_Q; end
    else
        begin H8next = H8next_ini1_Q; camefrom8 = camefrom8_ini1_Q; end

//state 9
    if(H9next_ini0_Q < H9next_ini1_Q)
        begin H9next = H9next_ini0_Q; camefrom9 = camefrom9_ini0_Q; end
    else
        begin H9next = H9next_ini1_Q; camefrom9 = camefrom9_ini1_Q; end

//state 10
    if(H10next_ini0_Q < H10next_ini1_Q)
        begin H10next = H10next_ini0_Q; camefrom10 = camefrom10_ini0_Q;
    end
    else
        begin H10next = H10next_ini1_Q; camefrom10 = camefrom10_ini1_Q;
    end

//state 11
    if(H11next_ini0_Q < H11next_ini1_Q)
        begin H11next = H11next_ini0_Q; camefrom11 = camefrom11_ini0_Q;
    end
    else
        begin H11next = H11next_ini1_Q; camefrom11 = camefrom11_ini1_Q;
    end

//state 12
    if(H12next_ini0_Q < H12next_ini1_Q)
        begin H12next = H12next_ini0_Q; camefrom12 = camefrom12_ini0_Q;
    end
    else
        begin H12next = H12next_ini1_Q; camefrom12 = camefrom12_ini1_Q;
    end

//state 13
    if(H13next_ini0_Q < H13next_ini1_Q)
        begin H13next = H13next_ini0_Q ;camefrom13 = camefrom13_ini0_Q;
    end
    else
        begin H13next = H13next_ini1_Q; camefrom13 = camefrom13_ini1_Q;
    end

//state 14
    if(H14next_ini0_Q < H14next_ini1_Q)
        begin H14next = H14next_ini0_Q ;camefrom14 = camefrom14_ini0_Q;
    end
    else
        begin H14next = H14next_ini1_Q; camefrom14 = camefrom14_ini1_Q;
    end

//state 15
    if(H15next_ini0_Q < H15next_ini1_Q)
        begin H15next = H15next_ini0_Q ;camefrom15 = camefrom15_ini0_Q;
    end
    else
        begin H15next = H15next_ini1_Q; camefrom15 = camefrom15_ini1_Q;
    end
end

```

```

end

always @(posedge clk or posedge reset)
begin
    if(reset) begin
        H0 <= 0;
        H1 <= 0;
        H2 <= 0;
        H3 <= 0;
        H4 <= 0;
        H5 <= 0;
        H6 <= 0;
        H7 <= 0;
        H8 <= 0;
        H9 <= 0;
        H10 <= 0;
        H11 <= 0;
        H12 <= 0;
        H13 <= 0;
        H14 <= 0;
        H15 <= 0;

        min_Q <= 0;
        min0_Q <= 0;
        min1_Q <= 0;
        min2_Q <= 0;
        min3_Q <= 0;
        min4_Q <= 0;
        min5_Q <= 0;
        min6_Q <= 0;
        min7_Q <= 0;
        min8_Q <= 0;
        min9_Q <= 0;
        min10_Q <= 0;
        min11_Q <= 0;
        min12_Q <= 0;
        min13_Q <= 0;

//State 0
        H0h000_Q <= 0;
        H1h001_Q <= 0;
        H6h010_Q <= 0;
        H7h011_Q <= 0;

//state 1
        H10h110_Q <= 0;
        H11h111_Q <= 0;
        H12h100_Q <= 0;
        H13h101_Q <= 0;

//State 2
        H0h001_Q <= 0;
        H1h000_Q <= 0;
        H6h011_Q <= 0;
        H7h010_Q <= 0;

//State 3
        H10h111_Q <= 0;

```

```

        H11h110_Q <= 0;
        H12h101_Q <= 0;
        H13h100_Q <= 0;
//state 4
        H2h000_Q <= 0;
        H3h001_Q <= 0;
        H4h010_Q <= 0;
        H5h011_Q <= 0;
//State 5
        H8h110_Q <= 0;
        H9h111_Q <= 0;
        H14h100_Q <= 0;
        H15h101_Q <= 0;
//State 6
        H2h001_Q <= 0;
        H3h000_Q <= 0;
        H4h011_Q <= 0;
        H5h010_Q <= 0;
//State 7
        H8h111_Q <= 0;
        H9h110_Q <= 0;
        H14h101_Q <= 0;
        H15h100_Q <= 0;
//State 8
        H2h010_Q <= 0;
        H3h011_Q <= 0;
        H4h000_Q <= 0;
        H5h001_Q <= 0;
//state 9
        H8h100_Q <= 0;
        H9h101_Q <= 0;
        H14h110_Q <= 0;
        H15h111_Q <= 0;
//State 10
        H2h011_Q <= 0;
        H3h010_Q <= 0;
        H4h001_Q <= 0;
        H5h000_Q <= 0;
//State 11
        H8h101_Q <= 0;
        H9h100_Q <= 0;
        H14h111_Q <= 0;
        H15h110_Q <= 0;
//State 12
        H0h010_Q <= 0;
        H1h011_Q <= 0;
        H6h000_Q <= 0;
        H7h001_Q <= 0;
//State 13
        H10h100_Q <= 0;
        H11h101_Q <= 0;
        H12h110_Q <= 0;
        H13h111_Q <= 0;
//State 14
        H0h011_Q <= 0;
        H1h010_Q <= 0;
        H6h001_Q <= 0;

```

```

//State 15      H7h000_Q <= 0;

                H10h101_Q <= 0;
                H11h100_Q <= 0;
                H12h111_Q <= 0;
                H13h110_Q <= 0;

                H0next_ini0_Q <= 0;
                H1next_ini0_Q <= 0;
                H2next_ini0_Q <= 0;
                H3next_ini0_Q <= 0;
                H4next_ini0_Q <= 0;
                H5next_ini0_Q <= 0;
                H6next_ini0_Q <= 0;
                H7next_ini0_Q <= 0;
                H8next_ini0_Q <= 0;
                H9next_ini0_Q <= 0;
                H10next_ini0_Q <= 0;
                H11next_ini0_Q <= 0;
                H12next_ini0_Q <= 0;
                H13next_ini0_Q <= 0;
                H14next_ini0_Q <= 0;
                H15next_ini0_Q <= 0;

                H0next_inil_Q <= 0;
                H1next_inil_Q <= 0;
                H2next_inil_Q <= 0;
                H3next_inil_Q <= 0;
                H4next_inil_Q <= 0;
                H5next_inil_Q <= 0;
                H6next_inil_Q <= 0;
                H7next_inil_Q <= 0;
                H8next_inil_Q <= 0;
                H9next_inil_Q <= 0;
                H10next_inil_Q <= 0;
                H11next_inil_Q <= 0;
                H12next_inil_Q <= 0;
                H13next_inil_Q <= 0;
                H14next_inil_Q <= 0;
                H15next_inil_Q <= 0;

                camefrom0_ini0_Q <= 0;
                camefrom1_ini0_Q <= 0;
                camefrom2_ini0_Q <= 0;
                camefrom3_ini0_Q <= 0;
                camefrom4_ini0_Q <= 0;
                camefrom5_ini0_Q <= 0;
                camefrom6_ini0_Q <= 0;
                camefrom7_ini0_Q <= 0;
                camefrom8_ini0_Q <= 0;
                camefrom9_ini0_Q <= 0;
                camefrom10_ini0_Q <= 0;
                camefrom11_ini0_Q <= 0;
                camefrom12_ini0_Q <= 0;
                camefrom13_ini0_Q <= 0;
                camefrom14_ini0_Q <= 0;
                camefrom15_ini0_Q <= 0;

```



```

        camefrom0_ini1_Q <= 0;
        camefrom1_ini1_Q <= 0;
        camefrom2_ini1_Q <= 0;
        camefrom3_ini1_Q <= 0;
        camefrom4_ini1_Q <= 0;
        camefrom5_ini1_Q <= 0;
        camefrom6_ini1_Q <= 0;
        camefrom7_ini1_Q <= 0;
        camefrom8_ini1_Q <= 0;
        camefrom9_ini1_Q <= 0;
        camefrom10_ini1_Q <= 0;
        camefrom11_ini1_Q <= 0;
        camefrom12_ini1_Q <= 0;
        camefrom13_ini1_Q <= 0;
        camefrom14_ini1_Q <= 0;
        camefrom15_ini1_Q <= 0;

        end
    else begin
        H0 <= H0next;
        H1 <= H1next;
        H2 <= H2next;
        H3 <= H3next;
        H4 <= H4next;
        H5 <= H5next;
        H6 <= H6next;
        H7 <= H7next;
        H8 <= H8next;
        H9 <= H9next;
        H10 <= H10next;
        H11 <= H11next;
        H12 <= H12next;
        H13 <= H13next;
        H14 <= H14next;
        H15 <= H15next;

        min_Q <= min;
        min0_Q <= min0;
        min1_Q <= min1;
        min2_Q <= min2;
        min3_Q <= min3;
        min4_Q <= min4;
        min5_Q <= min5;
        min6_Q <= min6;
        min7_Q <= min7;

        min8_Q <= min8;
        min9_Q <= min9;
        min10_Q <= min10;
        min11_Q <= min11;
        min12_Q <= min12;
        min13_Q <= min13;

    //State 0
        H0h000_Q <= H0h000;
        H1h001_Q <= H1h001;

```

```

        H6h010_Q <= H6h010;
        H7h011_Q <= H7h011;
//State 1
        H10h110_Q <= H10h110;
        H11h111_Q <= H11h111;
        H12h100_Q <= H12h100;
        H13h101_Q <= H13h101;
//State 2
        H0h001_Q <= H0h001;
        H1h000_Q <= H1h000;
        H6h011_Q <= H6h011;
        H7h010_Q <= H7h010;
//state 3
        H10h111_Q <= H10h111;
        H11h110_Q <= H11h110;
        H12h101_Q <= H12h101;
        H13h100_Q <= H13h100;
//State 4
        H2h000_Q <= H2h000;
        H3h001_Q <= H3h001;
        H4h010_Q <= H4h010;
        H5h011_Q <= H5h011;
//state 5
        H8h110_Q <= H8h110;
        H9h111_Q <= H9h111;
        H14h100_Q <= H14h100;
        H15h101_Q <= H15h101;
//State 6
        H2h001_Q <= H2h001;
        H3h000_Q <= H3h000;
        H4h011_Q <= H4h011;
        H5h010_Q <= H5h010;
//State 7
        H8h111_Q <= H8h111;
        H9h110_Q <= H9h110;
        H14h101_Q <= H14h101;
        H15h100_Q <= H15h100;
//Sate 8
        H2h010_Q <= H2h010;
        H3h011_Q <= H3h011;
        H4h000_Q <= H4h000;
        H5h001_Q <= H5h001;
//state 9
        H8h100_Q <= H8h100;
        H9h101_Q <= H9h101;
        H14h110_Q <= H14h110;
        H15h111_Q <= H15h111;
//State 10
        H2h011_Q <= H2h011;
        H3h010_Q <= H3h010;
        H4h001_Q <= H4h001;
        H5h000_Q <= H5h000;
//State 11
        H8h101_Q <= H8h101;
        H9h100_Q <= H9h100;
        H14h111_Q <= H14h111;
        H15h110_Q <= H15h110;

```

```

//State 12
    H0h010_Q <= H0h010;
    H1h011_Q <= H1h011;
    H6h000_Q <= H6h000;
    H7h001_Q <= H7h001;
//State 13
    H10h100_Q <= H10h100;
    H11h101_Q <= H11h101;
    H12h110_Q <= H12h110;
    H13h111_Q <= H13h111;
//State 14
    H0h011_Q <= H0h011;
    H1h010_Q <= H1h010;
    H6h001_Q <= H6h001;
    H7h000_Q <= H7h000;
//State 15
    H10h101_Q <= H10h101;
    H11h100_Q <= H11h100;
    H12h111_Q <= H12h111;
    H13h110_Q <= H13h110;

    H0next_ini0_Q <= H0next_ini0;
    H1next_ini0_Q <= H1next_ini0;
    H2next_ini0_Q <= H2next_ini0;
    H3next_ini0_Q <= H3next_ini0;
    H4next_ini0_Q <= H4next_ini0;
    H5next_ini0_Q <= H5next_ini0;
    H6next_ini0_Q <= H6next_ini0;
    H7next_ini0_Q <= H7next_ini0;
    H8next_ini0_Q <= H8next_ini0;
    H9next_ini0_Q <= H9next_ini0;
    H10next_ini0_Q <= H10next_ini0;
    H11next_ini0_Q <= H11next_ini0;
    H12next_ini0_Q <= H12next_ini0;
    H13next_ini0_Q <= H13next_ini0;
    H14next_ini0_Q <= H14next_ini0;
    H15next_ini0_Q <= H15next_ini0;

    H0next_inil_Q <= H0next_inil;
    H1next_inil_Q <= H1next_inil;
    H2next_inil_Q <= H2next_inil;
    H3next_inil_Q <= H3next_inil;
    H4next_inil_Q <= H4next_inil;
    H5next_inil_Q <= H5next_inil;
    H6next_inil_Q <= H6next_inil;
    H7next_inil_Q <= H7next_inil;
    H8next_inil_Q <= H8next_inil;
    H9next_inil_Q <= H9next_inil;
    H10next_inil_Q <= H10next_inil;
    H11next_inil_Q <= H11next_inil;
    H12next_inil_Q <= H12next_inil;
    H13next_inil_Q <= H13next_inil;
    H14next_inil_Q <= H14next_inil;
    H15next_inil_Q <= H15next_inil;

    camefrom0_ini0_Q <= camefrom0_ini0;
    camefrom1_ini0_Q <= camefrom1_ini0;

```

```

    camefrom2_ini0_Q <= camefrom2_ini0;
    camefrom3_ini0_Q <= camefrom3_ini0;
    camefrom4_ini0_Q <= camefrom4_ini0;
    camefrom5_ini0_Q <= camefrom5_ini0;
    camefrom6_ini0_Q <= camefrom6_ini0;
    camefrom7_ini0_Q <= camefrom7_ini0;
    camefrom8_ini0_Q <= camefrom8_ini0;
    camefrom9_ini0_Q <= camefrom9_ini0;
    camefrom10_ini0_Q <= camefrom10_ini0;
    camefrom11_ini0_Q <= camefrom11_ini0;
    camefrom12_ini0_Q <= camefrom12_ini0;
    camefrom13_ini0_Q <= camefrom13_ini0;
    camefrom14_ini0_Q <= camefrom14_ini0;
    camefrom15_ini0_Q <= camefrom15_ini0;

    camefrom0_inil_Q <= camefrom0_inil;
    camefrom1_inil_Q <= camefrom1_inil;
    camefrom2_inil_Q <= camefrom2_inil;
    camefrom3_inil_Q <= camefrom3_inil;
    camefrom4_inil_Q <= camefrom4_inil;
    camefrom5_inil_Q <= camefrom5_inil;
    camefrom6_inil_Q <= camefrom6_inil;
    camefrom7_inil_Q <= camefrom7_inil;
    camefrom8_inil_Q <= camefrom8_inil;
    camefrom9_inil_Q <= camefrom9_inil;
    camefrom10_inil_Q <= camefrom10_inil;
    camefrom11_inil_Q <= camefrom11_inil;
    camefrom12_inil_Q <= camefrom12_inil;
    camefrom13_inil_Q <= camefrom13_inil;
    camefrom14_inil_Q <= camefrom14_inil;
    camefrom15_inil_Q <= camefrom15_inil;

    end
end

endmodule

module Data_Carry_Forward(camefrom0, camefrom1, camefrom2, camefrom3,
camefrom4, camefrom5, camefrom6, camefrom7,
camefrom8, camefrom9, camefrom10, camefrom11,
camefrom12, camefrom13, camefrom14, camefrom15, clk, reset,
dataout_0, dataout_1, dataout_2 );

    input  clk, reset;
    output dataout_0, dataout_1, dataout_2;

    input [1:0] camefrom0, camefrom1, camefrom2, camefrom3;
    input [1:0] camefrom4, camefrom5, camefrom6, camefrom7;
    input [1:0] camefrom8, camefrom9, camefrom10, camefrom11;
    input [1:0] camefrom12, camefrom13, camefrom14, camefrom15;

    wire [1:0] camefrom0, camefrom1, camefrom2, camefrom3;
    wire [1:0] camefrom4, camefrom5, camefrom6, camefrom7;
    wire [1:0] camefrom8, camefrom9, camefrom10, camefrom11;
    wire [1:0] camefrom12, camefrom13, camefrom14, camefrom15;

```

```

wire clk, reset;
reg  dataout_0, dataout_1, dataout_2;

parameter latency = 20;

reg [latency:1] memNxt0_0, memNxt0_1, memNxt0_2;
reg [latency:1] memNxt1_0, memNxt1_1, memNxt1_2;
reg [latency:1] memNxt2_0, memNxt2_1, memNxt2_2;
reg [latency:1] memNxt3_0, memNxt3_1, memNxt3_2;
reg [latency:1] memNxt4_0, memNxt4_1, memNxt4_2;
reg [latency:1] memNxt5_0, memNxt5_1, memNxt5_2;
reg [latency:1] memNxt6_0, memNxt6_1, memNxt6_2;
reg [latency:1] memNxt7_0, memNxt7_1, memNxt7_2;
reg [latency:1] memNxt8_0, memNxt8_1, memNxt8_2;
reg [latency:1] memNxt9_0, memNxt9_1, memNxt9_2;
reg [latency:1] memNxt10_0, memNxt10_1, memNxt10_2;
reg [latency:1] memNxt11_0, memNxt11_1, memNxt11_2;
reg [latency:1] memNxt12_0, memNxt12_1, memNxt12_2;
reg [latency:1] memNxt13_0, memNxt13_1, memNxt13_2;
reg [latency:1] memNxt14_0, memNxt14_1, memNxt14_2;
reg [latency:1] memNxt15_0, memNxt15_1, memNxt15_2;

reg [latency:1] mem0_0, mem0_1, mem0_2, mem1_0, mem1_1, mem1_2;
reg [latency:1] mem2_0, mem2_1, mem2_2, mem3_0, mem3_1, mem3_2;
reg [latency:1] mem4_0, mem4_1, mem4_2, mem5_0, mem5_1, mem5_2;
reg [latency:1] mem6_0, mem6_1, mem6_2, mem7_0, mem7_1, mem7_2;
reg [latency:1] mem8_0, mem8_1, mem8_2, mem9_0, mem9_1, mem9_2;
reg [latency:1] mem10_0, mem10_1, mem10_2, mem11_0, mem11_1, mem11_2;
reg [latency:1] mem12_0, mem12_1, mem12_2, mem13_0, mem13_1, mem13_2;
reg [latency:1] mem14_0, mem14_1, mem14_2, mem15_0, mem15_1, mem15_2;

always @(camefrom0 or camefrom1 or camefrom2 or camefrom3 or
        camefrom4 or camefrom5 or camefrom6 or camefrom7 or
        camefrom8 or camefrom9 or camefrom10 or camefrom11 or
        camefrom12 or camefrom13 or camefrom14 or camefrom15 or
        mem0_0 or mem0_1 or mem0_2 or mem1_0 or mem1_1 or mem1_2 or
        mem2_0 or mem2_1 or mem2_2 or mem3_0 or mem3_1 or mem3_2 or
        mem4_0 or mem4_1 or mem4_2 or mem5_0 or mem5_1 or mem5_2 or
        mem6_0 or mem6_1 or mem6_2 or mem7_0 or mem7_1 or mem7_2 or
        mem8_0 or mem8_1 or mem8_2 or mem9_0 or mem9_1 or mem9_2 or
        mem10_0 or mem10_1 or mem10_2 or mem11_0 or mem11_1 or mem11_2
or
        mem12_0 or mem12_1 or mem12_2 or mem13_0 or mem13_1 or mem13_2
or
        mem14_0 or mem14_1 or mem14_2 or mem15_0 or mem15_1 or mem15_2
)
begin
    case (camefrom0)
        2'b00: begin
            memNxt0_0 = { 1'b0, mem0_0 } >>1;
            memNxt0_1 = { 1'b0, mem0_1 } >>1;
            memNxt0_2 = { 1'b0, mem0_2 } >>1;
            end
        2'b01: begin
            memNxt0_0 = { 1'b0, mem1_0 } >>1;
            memNxt0_1 = { 1'b0, mem1_1 } >>1;
    
```

```

        memNxt0_2 = { 1'b1,mem1_2} >>1;
    end
2'b10: begin
    memNxt0_0 = { 1'b0,mem6_0} >>1;
    memNxt0_1 = { 1'b1,mem6_1} >>1;
    memNxt0_2 = { 1'b0,mem6_2} >>1;
    end
2'b11: begin
    memNxt0_0 = { 1'b0,mem7_0} >>1;
    memNxt0_1 = { 1'b1,mem7_1} >>1;
    memNxt0_2 = { 1'b1,mem7_2} >>1;
    end
endcase

dataout_0 = mem0_0[1];
dataout_1 = mem0_1[1];
dataout_2 = mem0_2[1];

```

```

case (camefrom1)
2'b00: begin
    memNxt1_0 = { 1'b1,mem10_0} >>1;
    memNxt1_1 = { 1'b1,mem10_1} >>1;
    memNxt1_2 = { 1'b0,mem10_2} >>1;
    end
2'b01: begin
    memNxt1_0 = { 1'b1,mem11_0} >>1;
    memNxt1_1 = { 1'b1,mem11_1} >>1;
    memNxt1_2 = { 1'b1,mem11_2} >>1;
    end
2'b10: begin
    memNxt1_0 = { 1'b1,mem12_0} >>1;
    memNxt1_1 = { 1'b0,mem12_1} >>1;
    memNxt1_2 = { 1'b0,mem12_2} >>1;
    end
2'b11: begin
    memNxt1_0 = { 1'b1,mem13_0} >>1;
    memNxt1_1 = { 1'b0,mem13_1} >>1;
    memNxt1_2 = { 1'b1,mem13_2} >>1;
    end
endcase

```

```

case (camefrom2)
2'b00: begin
    memNxt2_0 = { 1'b0,mem0_0} >>1;
    memNxt2_1 = { 1'b0,mem0_1} >>1;
    memNxt2_2 = { 1'b1,mem0_2} >>1;
    end
2'b01: begin
    memNxt2_0 = { 1'b0,mem1_0} >>1;
    memNxt2_1 = { 1'b0,mem1_1} >>1;
    memNxt2_2 = { 1'b0,mem1_2} >>1;
    end
2'b10: begin
    memNxt2_0 = { 1'b0,mem6_0} >>1;
    memNxt2_1 = { 1'b1,mem6_1} >>1;
    memNxt2_2 = { 1'b1,mem6_2} >>1;

```

```

        end
2'b11: begin
    memNxt2_0 = { 1'b0,mem7_0} >>1;
    memNxt2_1 = { 1'b1,mem7_1} >>1;
    memNxt2_2 = { 1'b0,mem7_2} >>1;
    end
endcase

case (camefrom3)
2'b00: begin
    memNxt3_0 = { 1'b1,mem10_0} >>1;
    memNxt3_1 = { 1'b1,mem10_1} >>1;
    memNxt3_2 = { 1'b1,mem10_2} >>1;
    end
2'b01: begin
    memNxt3_0 = { 1'b1,mem11_0} >>1;
    memNxt3_1 = { 1'b1,mem11_1} >>1;
    memNxt3_2 = { 1'b0,mem11_2} >>1;
    end
2'b10: begin
    memNxt3_0 = { 1'b1,mem12_0} >>1;
    memNxt3_1 = { 1'b0,mem12_1} >>1;
    memNxt3_2 = { 1'b1,mem12_2} >>1;
    end
2'b11: begin
    memNxt3_0 = { 1'b1,mem13_0} >>1;
    memNxt3_1 = { 1'b0,mem13_1} >>1;
    memNxt3_2 = { 1'b0,mem13_2} >>1;
    end
endcase

case (camefrom4)
2'b00: begin
    memNxt4_0 = { 1'b0,mem2_0} >>1;
    memNxt4_1 = { 1'b0,mem2_1} >>1;
    memNxt4_2 = { 1'b0,mem2_2} >>1;
    end
2'b01: begin
    memNxt4_0 = { 1'b0,mem3_0} >>1;
    memNxt4_1 = { 1'b0,mem3_1} >>1;
    memNxt4_2 = { 1'b1,mem3_2} >>1;
    end
2'b10: begin
    memNxt4_0 = { 1'b0,mem4_0} >>1;
    memNxt4_1 = { 1'b1,mem4_1} >>1;
    memNxt4_2 = { 1'b0,mem4_2} >>1;
    end
2'b11: begin
    memNxt4_0 = { 1'b0,mem5_0} >>1;
    memNxt4_1 = { 1'b1,mem5_1} >>1;
    memNxt4_2 = { 1'b1,mem5_2} >>1;
    end
endcase

case (camefrom5)
2'b00: begin
    memNxt5_0 = { 1'b1,mem8_0} >>1;

```

```

        memNxt5_1 = { 1'b1,mem8_1} >>1;
        memNxt5_2 = { 1'b0,mem8_2} >>1;
        end
    2'b01: begin
        memNxt5_0 = { 1'b1,mem9_0} >>1;
        memNxt5_1 = { 1'b1,mem9_1} >>1;
        memNxt5_2 = { 1'b1,mem9_2} >>1;
        end
    2'b10: begin
        memNxt5_0 = { 1'b1,mem14_0} >>1;
        memNxt5_1 = { 1'b0,mem14_1} >>1;
        memNxt5_2 = { 1'b0,mem14_2} >>1;
        end
    2'b11: begin
        memNxt5_0 = { 1'b1,mem15_0} >>1;
        memNxt5_1 = { 1'b0,mem15_1} >>1;
        memNxt5_2 = { 1'b1,mem15_2} >>1;
        end
    endcase

    case (camefrom6)
    2'b00: begin
        memNxt6_0 = { 1'b0,mem2_0} >>1;
        memNxt6_1 = { 1'b0,mem2_1} >>1;
        memNxt6_2 = { 1'b1,mem2_2} >>1;
        end
    2'b01: begin
        memNxt6_0 = { 1'b0,mem3_0} >>1;
        memNxt6_1 = { 1'b0,mem3_1} >>1;
        memNxt6_2 = { 1'b0,mem3_2} >>1;
        end
    2'b10: begin
        memNxt6_0 = { 1'b0,mem4_0} >>1;
        memNxt6_1 = { 1'b1,mem4_1} >>1;
        memNxt6_2 = { 1'b1,mem4_2} >>1;
        end
    2'b11: begin
        memNxt6_0 = { 1'b0,mem5_0} >>1;
        memNxt6_1 = { 1'b1,mem5_1} >>1;
        memNxt6_2 = { 1'b0,mem5_2} >>1;
        end
    endcase

    case (camefrom7)
    2'b00: begin
        memNxt7_0 = { 1'b1,mem8_0} >>1;
        memNxt7_1 = { 1'b1,mem8_1} >>1;
        memNxt7_2 = { 1'b1,mem8_2} >>1;
        end
    2'b01: begin
        memNxt7_0 = { 1'b1,mem9_0} >>1;
        memNxt7_1 = { 1'b1,mem9_1} >>1;
        memNxt7_2 = { 1'b0,mem9_2} >>1;
        end
    2'b10: begin
        memNxt7_0 = { 1'b1,mem14_0} >>1;
        memNxt7_1 = { 1'b0,mem14_1} >>1;

```



```

        memNxt7_2 = { 1'b1,mem14_2} >>1;
    end
2'b11: begin
    memNxt7_0 = { 1'b1,mem15_0} >>1;
    memNxt7_1 = { 1'b0,mem15_1} >>1;
    memNxt7_2 = { 1'b0,mem15_2} >>1;
    end
endcase

case (camefrom8)
2'b00: begin
    memNxt8_0 = { 1'b0,mem2_0} >>1;
    memNxt8_1 = { 1'b1,mem2_1} >>1;
    memNxt8_2 = { 1'b0,mem2_2} >>1;
    end
2'b01: begin
    memNxt8_0 = { 1'b0,mem3_0} >>1;
    memNxt8_1 = { 1'b1,mem3_1} >>1;
    memNxt8_2 = { 1'b1,mem3_2} >>1;
    end
2'b10: begin
    memNxt8_0 = { 1'b0,mem4_0} >>1;
    memNxt8_1 = { 1'b0,mem4_1} >>1;
    memNxt8_2 = { 1'b0,mem4_2} >>1;
    end
2'b11: begin
    memNxt8_0 = { 1'b0,mem5_0} >>1;
    memNxt8_1 = { 1'b0,mem5_1} >>1;
    memNxt8_2 = { 1'b1,mem5_2} >>1;
    end
endcase

case (camefrom9)
2'b00: begin
    memNxt9_0 = { 1'b1,mem8_0} >>1;
    memNxt9_1 = { 1'b0,mem8_1} >>1;
    memNxt9_2 = { 1'b0,mem8_2} >>1;
    end
2'b01: begin
    memNxt9_0 = { 1'b1,mem9_0} >>1;
    memNxt9_1 = { 1'b0,mem9_1} >>1;
    memNxt9_2 = { 1'b1,mem9_2} >>1;
    end
2'b10: begin
    memNxt9_0 = { 1'b1,mem14_0} >>1;
    memNxt9_1 = { 1'b1,mem14_1} >>1;
    memNxt9_2 = { 1'b0,mem14_2} >>1;
    end
2'b11: begin
    memNxt9_0 = { 1'b1,mem15_0} >>1;
    memNxt9_1 = { 1'b1,mem15_1} >>1;
    memNxt9_2 = { 1'b1,mem15_2} >>1;
    end
endcase

case (camefrom10)
2'b00: begin

```

```

        memNxt10_0 = { 1'b0,mem2_0} >>1;
        memNxt10_1 = { 1'b1,mem2_1} >>1;
        memNxt10_2 = { 1'b1,mem2_2} >>1;
    end
    2'b01: begin
        memNxt10_0 = { 1'b0,mem3_0} >>1;
        memNxt10_1 = { 1'b1,mem3_1} >>1;
        memNxt10_2 = { 1'b0,mem3_2} >>1;
    end
    2'b10: begin
        memNxt10_0 = { 1'b0,mem4_0} >>1;
        memNxt10_1 = { 1'b0,mem4_1} >>1;
        memNxt10_2 = { 1'b1,mem4_2} >>1;
    end
    2'b11: begin
        memNxt10_0 = { 1'b0,mem5_0} >>1;
        memNxt10_1 = { 1'b0,mem5_1} >>1;
        memNxt10_2 = { 1'b0,mem5_2} >>1;
    end
endcase

case (camefrom11)
    2'b00: begin
        memNxt11_0 = { 1'b1,mem8_0} >>1;
        memNxt11_1 = { 1'b0,mem8_1} >>1;
        memNxt11_2 = { 1'b1,mem8_2} >>1;
    end
    2'b01: begin
        memNxt11_0 = { 1'b1,mem9_0} >>1;
        memNxt11_1 = { 1'b0,mem9_1} >>1;
        memNxt11_2 = { 1'b0,mem9_2} >>1;
    end
    2'b10: begin
        memNxt11_0 = { 1'b1,mem14_0} >>1;
        memNxt11_1 = { 1'b1,mem14_1} >>1;
        memNxt11_2 = { 1'b1,mem14_2} >>1;
    end
    2'b11: begin
        memNxt11_0 = { 1'b1,mem15_0} >>1;
        memNxt11_1 = { 1'b1,mem15_1} >>1;
        memNxt11_2 = { 1'b0,mem15_2} >>1;
    end
endcase

case (camefrom12)
    2'b00: begin
        memNxt12_0 = { 1'b0,mem0_0} >>1;
        memNxt12_1 = { 1'b1,mem0_1} >>1;
        memNxt12_2 = { 1'b0,mem0_2} >>1;
    end
    2'b01: begin
        memNxt12_0 = { 1'b0,mem1_0} >>1;
        memNxt12_1 = { 1'b1,mem1_1} >>1;
        memNxt12_2 = { 1'b1,mem1_2} >>1;
    end
    2'b10: begin
        memNxt12_0 = { 1'b0,mem6_0} >>1;

```

```

        memNxt12_1 = { 1'b0,mem6_1} >>1;
        memNxt12_2 = { 1'b0,mem6_2} >>1;
    end
    2'b11: begin
        memNxt12_0 = { 1'b0,mem7_0} >>1;
        memNxt12_1 = { 1'b0,mem7_1} >>1;
        memNxt12_2 = { 1'b1,mem7_2} >>1;
    end
endcase

case (camefrom13)
    2'b00: begin
        memNxt13_0 = { 1'b1,mem10_0} >>1;
        memNxt13_1 = { 1'b0,mem10_1} >>1;
        memNxt13_2 = { 1'b0,mem10_2} >>1;
    end
    2'b01: begin
        memNxt13_0 = { 1'b1,mem11_0} >>1;
        memNxt13_1 = { 1'b0,mem11_1} >>1;
        memNxt13_2 = { 1'b1,mem11_2} >>1;
    end
    2'b10: begin
        memNxt13_0 = { 1'b1,mem12_0} >>1;
        memNxt13_1 = { 1'b1,mem12_1} >>1;
        memNxt13_2 = { 1'b0,mem12_2} >>1;
    end
    2'b11: begin
        memNxt13_0 = { 1'b1,mem13_0} >>1;
        memNxt13_1 = { 1'b1,mem13_1} >>1;
        memNxt13_2 = { 1'b1,mem13_2} >>1;
    end
endcase

case (camefrom14)
    2'b00: begin
        memNxt14_0 = { 1'b0,mem0_0} >>1;
        memNxt14_1 = { 1'b1,mem0_1} >>1;
        memNxt14_2 = { 1'b1,mem0_2} >>1;
    end
    2'b01: begin
        memNxt14_0 = { 1'b0,mem1_0} >>1;
        memNxt14_1 = { 1'b1,mem1_1} >>1;
        memNxt14_2 = { 1'b0,mem1_2} >>1;
    end
    2'b10: begin
        memNxt14_0 = { 1'b0,mem6_0} >>1;
        memNxt14_1 = { 1'b0,mem6_1} >>1;
        memNxt14_2 = { 1'b1,mem6_2} >>1;
    end
    2'b11: begin
        memNxt14_0 = { 1'b0,mem7_0} >>1;
        memNxt14_1 = { 1'b0,mem7_1} >>1;
        memNxt14_2 = { 1'b0,mem7_2} >>1;
    end
endcase

case (camefrom15)

```

```

    2'b00: begin
        memNxt15_0 = { 1'b1,mem10_0} >>1;
        memNxt15_1 = { 1'b0,mem10_1} >>1;
        memNxt15_2 = { 1'b1,mem10_2} >>1;
    end
    2'b01: begin
        memNxt15_0 = { 1'b1,mem11_0} >>1;
        memNxt15_1 = { 1'b0,mem11_1} >>1;
        memNxt15_2 = { 1'b0,mem11_2} >>1;
    end
    2'b10: begin
        memNxt15_0 = { 1'b1,mem12_0} >>1;
        memNxt15_1 = { 1'b1,mem12_1} >>1;
        memNxt15_2 = { 1'b1,mem12_2} >>1;
    end
    2'b11: begin
        memNxt15_0 = { 1'b1,mem13_0} >>1;
        memNxt15_1 = { 1'b1,mem13_1} >>1;
        memNxt15_2 = { 1'b0,mem13_2} >>1;
    end
endcase

end

always @(posedge clk or posedge reset)
begin
    if(reset) begin
        mem0_0 <= 0;
        mem0_1 <= 0;
        mem0_2 <= 0;
        mem1_0 <= 0;
        mem1_1 <= 0;
        mem1_2 <= 0;
        mem2_0 <= 0;
        mem2_1 <= 0;
        mem2_2 <= 0;
        mem3_0 <= 0;
        mem3_1 <= 0;
        mem3_2 <= 0;
        mem4_0 <= 0;
        mem4_1 <= 0;
        mem4_2 <= 0;
        mem5_0 <= 0;
        mem5_1 <= 0;
        mem5_2 <= 0;
        mem6_0 <= 0;
        mem6_1 <= 0;
        mem6_2 <= 0;
        mem7_0 <= 0;
        mem7_1 <= 0;
        mem7_2 <= 0;
        mem8_0 <= 0;
        mem8_1 <= 0;
        mem8_2 <= 0;
        mem9_0 <= 0;
        mem9_1 <= 0;
    end
end

```

```

        mem9_2 <= 0;
        mem10_0 <= 0;
        mem10_1 <= 0;
        mem10_2 <= 0;
        mem11_0 <= 0;
        mem11_1 <= 0;
        mem11_2 <= 0;
        mem12_0 <= 0;
        mem12_1 <= 0;
        mem12_2 <= 0;
        mem13_0 <= 0;
        mem13_1 <= 0;
        mem13_2 <= 0;
        mem14_0 <= 0;
        mem14_1 <= 0;
        mem14_2 <= 0;
        mem15_0 <= 0;
        mem15_1 <= 0;
        mem15_2 <= 0;

        end
    else begin
        mem0_0 <= memNxt0_0;
        mem0_1 <= memNxt0_1;
        mem0_2 <= memNxt0_2;
        mem1_0 <= memNxt1_0;
        mem1_1 <= memNxt1_1;
        mem1_2 <= memNxt1_2;
        mem2_0 <= memNxt2_0;
        mem2_1 <= memNxt2_1;
        mem2_2 <= memNxt2_2;
        mem3_0 <= memNxt3_0;
        mem3_1 <= memNxt3_1;
        mem3_2 <= memNxt3_2;
        mem4_0 <= memNxt4_0;
        mem4_1 <= memNxt4_1;
        mem4_2 <= memNxt4_2;
        mem5_0 <= memNxt5_0;
        mem5_1 <= memNxt5_1;
        mem5_2 <= memNxt5_2;
        mem6_0 <= memNxt6_0;
        mem6_1 <= memNxt6_1;
        mem6_2 <= memNxt6_2;
        mem7_0 <= memNxt7_0;
        mem7_1 <= memNxt7_1;
        mem7_2 <= memNxt7_2;
        mem8_0 <= memNxt8_0;
        mem8_1 <= memNxt8_1;
        mem8_2 <= memNxt8_2;
        mem9_0 <= memNxt9_0;
        mem9_1 <= memNxt9_1;
        mem9_2 <= memNxt9_2;
        mem10_0 <= memNxt10_0;
        mem10_1 <= memNxt10_1;
        mem10_2 <= memNxt10_2;
        mem11_0 <= memNxt11_0;
        mem11_1 <= memNxt11_1;

```

```

        mem11_2 <= memNxt11_2;
        mem12_0 <= memNxt12_0;
        mem12_1 <= memNxt12_1;
        mem12_2 <= memNxt12_2;
        mem13_0 <= memNxt13_0;
        mem13_1 <= memNxt13_1;
        mem13_2 <= memNxt13_2;
        mem14_0 <= memNxt14_0;
        mem14_1 <= memNxt14_1;
        mem14_2 <= memNxt14_2;
        mem15_0 <= memNxt15_0;
        mem15_1 <= memNxt15_1;
        mem15_2 <= memNxt15_2;

    end

end

endmodule

module noise(in_noise, sel, clk, reset, in_inv, in_dff, out_inv,
out_dff, d5);

input in_noise, sel, clk, reset, in_inv, in_dff;
output out_inv, out_dff;
output [15:0] d5;

wire clk, clk1, clk2, clk3, clk4, clk5;
wire clk_0, clk_1, clk_2, clk_3, clk_4, clk_5;
wire clk1_1, clk1_2;
wire clk2_1, clk2_2;
wire clk3_1, clk3_2;
wire clk4_1, clk4_2;
wire clk5_1, clk5_2;
wire [15:0] din, d0, d1, d2, d3, d4, d5;

assign out_inv = ! in_inv;
dff_1b dff_1b(in_dff, clk_0, reset, out_dff);

BUF5 BUF1_1( .I(clk),      .Z(clk1_0) );

BUF5 BUF1_2( .I(clk1_0), .Z(clk1_1) );
BUF5 BUF1_3( .I(clk1_1), .Z(clk1_2) );
BUF5 BUF1_4( .I(clk1_2), .Z(clk1) );

BUF5 BUF2_1( .I(clk1),    .Z(clk2_1) );
BUF5 BUF2_2( .I(clk2_1), .Z(clk2_2) );
BUF5 BUF2_3( .I(clk2_2), .Z(clk2) );

BUF5 BUF3_1( .I(clk2),    .Z(clk3_1) );
BUF5 BUF3_2( .I(clk3_1), .Z(clk3_2) );
BUF5 BUF3_3( .I(clk3_2), .Z(clk3) );

BUF5 BUF4_1( .I(clk3),    .Z(clk4_1) );
BUF5 BUF4_2( .I(clk4_1), .Z(clk4_2) );
BUF5 BUF4_3( .I(clk4_2), .Z(clk4) );

BUF5 BUF5_1( .I(clk4),    .Z(clk5_1) );

```

```

BUF5 BUF5_2( .I(clk5_1), .Z(clk5_2) );
BUF5 BUF5_3( .I(clk5_2), .Z(clk5) );

```

```

mux mux0(sel, clk, clk1_0, clk_0);

```

```

mux mux1(sel, clk, clk1, clk_1);
mux mux2(sel, clk, clk2, clk_2);
mux mux3(sel, clk, clk3, clk_3);
mux mux4(sel, clk, clk4, clk_4);
mux mux5(sel, clk, clk5, clk_5);

```

```

assign din[0] = in_noise;
assign din[1] = in_noise;
assign din[2] = in_noise;
assign din[3] = in_noise;
assign din[4] = in_noise;
assign din[5] = in_noise;
assign din[6] = in_noise;
assign din[7] = in_noise;
assign din[8] = in_noise;
assign din[9] = in_noise;
assign din[10] = in_noise;
assign din[11] = in_noise;
assign din[12] = in_noise;
assign din[13] = in_noise;
assign din[14] = in_noise;
assign din[15] = in_noise;

```

```

dff dff0(din, clk_0, reset, d0);
dff dff1(d0, clk_1, reset, d1);
dff dff2(d1, clk_2, reset, d2);
dff dff3(d2, clk_3, reset, d3);
dff dff4(d3, clk_4, reset, d4);
dff dff5(d4, clk_5, reset, d5);

```

```

endmodule

```

```

module mux(sel, clk, clk_in, clk_out);

```

```

input sel, clk, clk_in;
output clk_out;
reg clk_out;

```

```

always @(sel or clk or clk_in)
begin
if(sel) clk_out = clk;
else clk_out = clk_in;
end

```

```

endmodule

```

```

module dff(D, clk, reset, Q);
input clk, reset;
input [15:0] D;

```

```
wire [15:0] D;

output [15:0] Q;
reg [15:0] Q;

always @( posedge clk or posedge reset)
begin
    if(reset) Q <= 0;
    else      Q <= D;
end

endmodule


module dff_1b(D, clk, reset, Q);
input  clk, reset, D;
output Q;
reg Q;

always @( posedge clk or posedge reset)
begin
    if(reset) Q <= 0;
    else      Q <= D;
end
endmodule
```


Appendix B: Block Diagram and Schematic

Appendix C: PGA68 Package

PGA68 pin bonding and layout

Bonding Diagram

Pin Out Diagram (bottom view)

Appendix D: Bonding diagram

Appendix E: MATLAB code and C code

```

diary('DAVIC_MATRIX_inv_end16.log')
clear D N TDN DTDN
syms D N TDN DTDN
%A is a state transition matrix
%
%      X1      X2      X3      X4      X5      X6      X7      X8 -odd row
%      X9      X10     X11     X12     X13     X14     X15     X16-Even row
A = [ ...
      0      0      0      0      0      0      0      0
      0      D^2*N  D^3*N^2  D      D^2*N  0      0      0;
      1      0      0      0      0      D^2*N^2  D*N  0
      0      0      0      0      0      0      0      0;
      0      0      0      0      0      0      0      0
      0      D^3*N^2  D^2*N  D^2*N  D      0      0      0;
      0      1      D*N      D*N  D^2*N^2  0      0      0
      0      0      0      0      0      0      0      0;
      0      0      0      0      0      0      0      D^2*N
      D^3*N^2  0      0      0      0      D      D^2*N  0;
      0      D*N      1      D^2*N^2  D*N      0      0      0
      0      0      0      0      0      0      0      0;
      0      0      0      0      0      0      0      D^3*N^2
      D^2*N  0      0      0      0      D^2*N  D      0;
      0      D*N  D^2*N^2  1      D*N      0      0      0
      0      0      0      0      0      0      0      0;
      0      0      0      0      0      0      0      D
      D^2*N  0      0      0      0      D^2*N  D^3*N^2  0;
      0      D^2*N^2  D*N      D*N      1      0      0      0
      0      0      0      0      0      0      0      0;
      0      0      0      0      0      0      0      D^2*N
      D      0      0      0      0      D^3*N^2  D^2*N  0;
      D^2*N^2  0      0      0      0      1      D*N      0
      0      0      0      0      0      0      0      0;
      0      0      0      0      0      0      0      0
      0      D      D^2*N  D^2*N  D^3*N^2  0      0      0;
      D*N      0      0      0      0      D*N      1      0
      0      0      0      0      0      0      0      0;
      0      0      0      0      0      0      0      0
      0      D^2*N  D      D^3*N^2  D^2*N  0      0      0;
      D*N      0      0      0      0      D*N  D^2*N^2  0
      0      0      0      0      0      0      0      0]
b = [0; D*N; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; D*N; 0; D^2*N^2; 0; 0]
rref(A)
I=eye(16,16)
IA=I-A
INV_IA=inv(IA);
pretty(INV_IA);
X=INV_IA*b;
TDN = X(16);
pretty(TDN) ;
%TDN is the transfer function
DTDN = diff(TDN,N);
%DNDT=dT(D,N)/dN
pretty(DTDN);
df = 3.0;
Rate = 2./3.;

```

```

SNR = 4.0                                %unit is dB
EbN0 = 10.0^(SNR/10.0);
EcN0 = Rate * EbN0;
Dvalue = exp(-1.0*EcN0);
DTDN1 = subs(DDTN, {D, N}, {Dvalue, 1.0});
Pb = 0.5*erfc(sqrt(df*EcN0))*exp(df*EcN0)*DDTN1/2.
%Q(x)=0.5*erfc(x/sqrt(2)).divided by 2 is for b=2, i.e. two input bits.

```

AWGN.c

```

#include <stdio.h>
#include <math.h>
#include <stdlib.h>

const double pi = 3.141592654;

int main()
{
    FILE *ifp, *ofp;

    double SNR, sigma, awgn1, awgn2, awgn3, x1, x2, x3, x4, x5, x6,
x_ini, y_ini, z_ini;
    int i, n;
    int z0, z1, z2;
    int x, y, z;

    ifp = fopen("transmitted.out", "r");
    ofp = fopen("received_SNR7.in", "w");

    printf("Hello, Input SNR(dB) !\n");
    scanf("%lf", &SNR);          /* must be "%lf", not "%f"  !!!!!!!!!!!!!!! */
    printf("Hello, Input the number of events !\n");
    scanf("%d", &n);

    sigma = sqrt(0.75 * pow(10.0, -1.0*SNR/10.0));
    printf("sigma= %lf \n", sigma);

    for(i=1; i <= n; ++i)
    {
        fscanf(ifp, "%d %d %d\n", &z0, &z1, &z2);

        /* BPSK 0 -> -1, 1 -> + 1 */
        if(z0 == 0) z0 = -1;
        if(z1 == 0) z1 = -1;
        if(z2 == 0) z2 = -1;

        x1 = drand48();
        x2 = drand48();
        x3 = drand48();
        x4 = drand48();
        x5 = drand48();
        x6 = drand48();

        awgn1 = sigma * sqrt(-2.*log(x1)) * cos(2.0*pi*x2);
        awgn2 = sigma * sqrt(-2.*log(x3)) * cos(2.0*pi*x4);
    }
}

```

```

    awgn3 = sigma * sqrt(-2.*log(x5)) * cos(2.0*pi*x6);

    x_ini = z0 + awgn1;
    y_ini = z1 + awgn2;
    z_ini = z2 + awgn3;

    if(x_ini > 0) x = 1;
    else      x = 0;

    if(y_ini > 0) y = 1;
    else      y = 0;

    if(z_ini > 0) z = 1;
    else      z = 0;

    fprintf(ofp, "%d%d%d\n", x, y, z);

}

fclose(ofp);
}

```

Compare.c

```

#include <stdio.h>

int main()
{
    FILE *ifp1, *ifp2, *ofp;
    int i;
    int z1, z2;
    int z11, z22;
    int total_number = 2000000;
    int cnt = 0;

    ifp1 = fopen("precoded.out", "r");
    ifp2 = fopen("decoded_SNR7.out", "r");
    ofp = fopen("Pr_SNR7.out", "w");

    printf("Hello, word!\n");

    for(i=1; i < total_number-21; ++i)
    {
        fscanf(ifp1, "%d %d\n", &z1, &z2);
        fscanf(ifp2, "%d %d\n", &z11, &z22);

        if(z11 != z1)
        {
            cnt = cnt + 1;
            printf("Mismatch at line %d    precoded bit z1 = %d\n", i,
z1);
        }
    }
}

```

```

        if(z22 != z2)
        {
            cnt = cnt + 1;
            printf("Mismatch at line %d    precoded bit z2 =  %d\n", i,
z2);
        }

    }

        printf("cnt = %d\n", cnt);
        printf("i = %d\n", i);
        printf("Error Pr =  %12.101f \n",
(float)cnt/(2.0*total_number));
        fprintf(ofp, "Error Pr =  %12.101f \n",
(float)cnt/(2.0*total_number));

        fclose(ifp1);
        fclose(ifp2);
        fclose(ofp);
}

```


Appendix F: Synopsys Timing Report

This is an example of Synopsys Timing Report for the pipelined DVAIC 1.3.1 Viterbi decoder using the overflow prevention method developed in this thesis.

```
*****
Report : timing
        -path full
        -delay max
        -max_paths 1
Design : thesischip
Version: 2000.05
Date   : Wed Aug 29 00:52:54 2001
*****
```

Operating Conditions: WCCOM Library: tcb773pwc
Wire Load Model Mode: segmented

```
Startpoint: thesis/topdecoder/Calc/H0next_ini1_Q_reg_1_
            (rising edge-triggered flip-flop clocked by clk)
Endpoint:   thesis/topdecoder/Data_Carry_forward1/mem0_0_reg_4_
            (rising edge-triggered flip-flop clocked by clk)
Path Group: clk
Path Type:  max
```

Des/Clust/Port	Wire Load Model	Library
thesischip	TSMC64K_Conservative	tcb773pwc
thesis	TSMC32K_Conservative	tcb773pwc

Point	Incr	Path
clock clk (rise edge)	0.00	0.00
clock network delay (propagated)	6.92	6.92
thesis/topdecoder/Calc/H0next_ini1_Q_reg_1_/CP (DFXCNS1)	0.00	6.92 r
thesis/topdecoder/Calc/H0next_ini1_Q_reg_1_/QN (DFXCNS1)	3.31	10.23 r
thesis/U5315/ZN (NR2D1H)	0.20	10.43 f
thesis/U5313/ZN (NR2D1H)	0.31	10.74 r
thesis/U5310/ZN (OAI22D1H)	0.24	10.98 f
thesis/U1809/ZN (ND2D1)	0.33	11.31 r
thesis/U5316/ZN (ND2D1H)	0.29	11.60 f
thesis/U2239/Z (MUX2D1)	0.76	12.36 r
thesis/U2276/ZN (INV0)	0.50	12.86 f
thesis/U3594/ZN (IND2D2)	0.98	13.84 f
thesis/U3595/Z (BUF2)	1.08	14.93 f
thesis/U5512/ZN (IOA221D1B)	1.07	15.99 r
thesis/topdecoder/Data_Carry_forward1/mem0_0_reg_4_/DA (DFXCNS1N)	0.00	15.99 r
data arrival time		15.99
clock clk (rise edge)	8.00	8.00
clock network delay (propagated)	6.92	14.92
clock uncertainty	-0.50	14.42
thesis/topdecoder/Data_Carry_forward1/mem0_0_reg_4_/CP (DFXCNS1N)	0.00	14.42 r
library setup time	1.57	15.99
data required time		15.99
data required time		15.99
data arrival time		-15.99
slack (MET)		0.00