Soft-error Tolerant Quasi Delay-insensitive Circuits

Thesis by

Wonjin Jang

In Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy

California Institute of Technology
Pasadena, California

2008

(Defended September 11, 2007)

Acknowledgments

The journey at Caltech was an incredible experience to me, for its people, its traditions, and its enormous accumulated wealth of scientific knowledge. Lots of people have helped me to finish the journey. Obviously things would have been different without them.

First of all, I would like to thank my advisor Alain J. Martin for being my mentor. He has taught me the importance of choosing right research problems, which are not trivial but not intractable; he helped me to choose a right problem and to solve the problem, which is all about this thesis. Mika Nyström, a former post-doc of the Asynchronous VLSI Group at Caltech, has urged me to write everything succinctly and correctly not only in logic but also in English. He is the best critic for me. Most of chapters owe much of their clarity to Mika's demands. Remaining errors are mine. In addition to that, I wish to thank the members of the Friday morning meeting for many insightful discussions: Sean Keller, Chris Moore, Niki Mehta, Karl Papadantonakis, Jonathan Dama, and Piyush Prakash.

Byung-Jun Yoon at Caltech, has always been supportive of me in every aspect. I cannot imagine my life at Caltech without him; we have shared happiness and bitterness of graduate-student life together. I am lucky to meet such a friend during my life time. Juneseuk Shin, my high-school buddy, has kept me from taking a compromising answer for life. Even when I was frustrated by gap between reality and expectation, he encouraged me not to give up my faith on life and people. Moreover, he always shares his acute observation on life while he introduce insightful books and beautiful works of music. As he is my life-time resourceful friend, I hope to inspire him intellectually and religiously. Certainly I remember many delightful conversations with Jina Choi, Wonhee Lee, Ji Hun Kim, Hyunjoo Lee, Chihoon Ahn, Seung-yub Lee, and Nayoung Ko, who have helped me to survive here in different ways.

I owe a lot of my life-time learning to Rev. Byung-Joo Song, Rev. Dong Hwan Kim and Jung-Suk Kim, whom I have met in my church and in the Monday bible-study meeting.

They have shared with me their understanding, passion and love for God and people. I appreciate that I could meet these spiritual mentors, who have enriched my life, and have made me pursue what God tells us to do throughout life. Additionally, I would like to thank the undergraduate members of the bible-study meeting for their devotion to the meeting and Him: Hwan-seung Yeo, Jinwoo Lee, and Eui Woong Lee.

My uncle, Ki-young Kim, and my cousins, Moon-suk Kim, and Moon-sun Kim give me a warm welcome whenever I visit Korea. I am very glad to have a chance to say my appreciation to them.

Last, this thesis is dedicated to my parents, who started all this; I would like to give all my thanks to my parents for their supports and their love, hoping that my achievement comforts my mother, who have had hard time to take care of my lovely little sister, Moon-jung Jang. Although all I have come from theirs, especially I have learned sincerity from my mother and curiosity from my father, which are precious assets for me. I am sad to say that my father did not live to see me finish this thesis, but I am sure that he sees this in the Heaven.

I know that I have failed to follow His way many times, but I still admit that I cannot finish this without the belief that Jesus Christ is the way, the truth, and the life. Now I embark upon a new journey, through which I wish to be one step closer to Him.

Abstract

A hard error is an error that damages a circuit irrevocably; a soft error flips the logic states without causing any physical damage to the circuit, resulting in transient corruption of data. They result in transient, inconsistent corruption of data.

The soft-error tolerance of logic circuits is recently getting more attention, since the soft-error rate of advanced CMOS devices is higher than before. As a response to the concern on soft errors, we propose a new method for making asynchronous circuits tolerant to soft errors. Since it relies on a property unique to asynchronous circuits, the method is different from what is done in synchronous circuits with triple modular redundancy. Asynchronous circuits have been attractive to the designers of reliable systems, because of their clock-less design, which makes them more robust to variations on computation time of modules. The quasi delay-insensitive (QDI) design style is one of the most robust asynchronous design styles for general computation; it makes one minimal assumption on delays in gates and wires. QDI circuits are easy to verify, simple, and modular, because the correct operation of a QDI circuit is independent of delays in gates and wires.

Here, we shall overview how to design a QDI circuit, and what will happen if a soft error occurs on a QDI circuit. Then the crucial components of the method are shown: (1) a special kind of duplication for random logic (when each bit has to be corrected individually), (2) special protection circuitry for arbiter and synchronizer (as needed for example for external interrupts), (3) reconfigurable circuits using a special configuration unit, and (4) error correcting for memory arrays and other structures in which the data bits can be self-corrected. The solution of protecting random logic is compared with alternatives, which use other types of error correcting codes (e.g., parity code) in a QDI circuit. It turns out that the duplication generates efficient circuits more commonly than other possible constructions. Finally, the design of a soft-error tolerant asynchronous microprocessor is detailed and testing results of the soft-error tolerance of the microprocessor are shown.

Contents

Acknowledgments **iii**

Abstract **v**

List of Figures **ix**

List of Tables **xv**

1 Introduction **1**

2 Designing Quasi Delay-insensitive (QDI) Circuits **4**

2.1 Communicating Hardware Processes 4

2.2 Handshaking Expansion 5

2.3 Production Rule Set and PRS Computation 6

2.4 Example of QDI Circuits: Buffer 8

3 Soft Errors in QDI Circuits **12**

3.1 Errors on Circuits 12

3.2 Digital Model of Soft Errors 14

3.3 Analog Model of Soft Errors 17

4 Related Work **20**

5 Protecting QDI Circuits from Soft Errors by Design **23**

5.1 Duplicating QDI Circuits with Cross-coupled C-elements 24

5.1.1 Duplicated Double-checking QDI Circuit 24

5.1.2 Comparing Reliability of QDI Circuits and DD QDI Circuits 31

5.1.3 Simulation Results of Duplicated Double-checking Buffer 35

5.1.4 Summary 40

5.2 Arbiter with Mutual Excluders 40

5.2.1 Overview 40

5.2.2 Designing Error Tolerant Arbiter for DD Circuits 43

5.2.3 Mutual Exclusion of Outputs of Duplicated Error Tolerant Arbiter . 49

5.2.4 Simulation Results of Error Tolerant Arbiter 52

5.2.5 Summary 52

5.3 Duplicated FPGA Using Self-correcting Programmable Bits 54

5.3.1 Overview 54

5.3.2 Designing Error Tolerant FPGA Cells 54

5.3.3 Summary 57

5.4 Self-correcting Memory and Hamming-coded Memory 57

5.4.1 Overview 57

5.4.2 Error Tolerant Memory Using Self-correcting Memory Cells 60

5.4.3 Error Tolerant Memory Based on Hamming Code 62

5.4.4 Simulation Results of Error Tolerant Memory 72

5.4.5 Summary 74

6 QDI Circuits using Error Detecting Delay-insensitive Codes 76

6.1 Error Detecting Delay-insensitive (EDDI) Code 76

6.1.1 Definition of Delay-insensitive Codes 76

6.1.2 Definition of Error Detecting Delay-insensitive Codes 77

6.1.3 Error Detecting Delay-insensitive Code Based on Linear Code 79

6.2 Basic Component of QDI Circuit: Function Block and Completion Checker 82

6.3 Error Tolerant Function Block for Linear DI Code 84

6.3.1 Designing Error Tolerant Precharging-Evaluating Logic 85

6.3.2 Simplifying Based on Symmetry of Repetition Codes 89

6.3.3 Comparing Size of Function Blocks for Different Linear DI Codes . . 91

6.3.4 Function Blocks Using r-error Linear DI Codes 94

6.4 Error Tolerant Completion Checker for Linear DI Code 97

6.5 Cost of Error Tolerant QDI Circuits for Different Linear DI Codes 101

6.5.1 Size 102

6.5.2 Et^2 103

6.6 Simulation Results of QDI Circuits for Linear DI Codes 108

6.7 Summary 110

7 Design Example: Soft-error Tolerant Asynchronous Microprocessor 115

7.1 Overview 115

7.2 Designing Soft-Error Tolerant PCUNIT from CHP Description to Layout . 118

7.2.1 Implementing Arbitrated Branch 120

7.2.2 Implementing Body of PCUNIT 122

7.3 Evaluating Soft-error Tolerant Asynchronous Microprocessor in Simulations 126

7.3.1 Verifying Soft-error Tolerance 126

7.3.2 Performance 128

7.3.3 Comparing Overhead of Error Tolerance 130

7.4 Summary 133

8 Conclusion 134

8.1 Lessons Learned 134

8.2 Future Work 135

A Acronym Glossary 136

B Controller in Memory Unit 138

C Upper Bound of the Size of Code 141

D Minimizing Decomposed Function Block 144

E STAM Architecture 148

F Arbitrated Branch 152

G Designing CHP Process Using Synthesis Tool Flow 154

H Checking Error Tolerance of PRS 159

Bibliography 162

List of Figures

1.1 *Pipeline of Buffers.* 1

2.1 *Handshake Expansion of a Synchronization Channel.* 6
2.2 *Circuit and Environment.* 8
2.3 *Transition Diagram of Precharged Half Buffer. Each state represents the values of (L.i,L.e,R.i,R.e). The double-circled state is the initial state.* . . . 10
2.4 *Circuit Diagram of Precharged Half Buffer.* 10
2.5 *Circuit Diagram of C-element and its Gate Diagram.* 11

3.1 *Interaction of a Particle and Silicon Substrate.* 13
3.2 *Examples of Erroneous Computations Caused by Soft Error.* 15
3.3 *Transition Diagram of PCHB with Soft Error on L.e. The dotted circles indicate invalid states, and the dotted edges indicate erroneous transitions caused by an error on L.e.* 16
3.4 *Example of Normal Execution Path of Firings in PCHB.* 16
3.5 *Transition Diagram of Buffer with Soft Error on L.e. The dotted circles indicate invalid states, and the dotted edges indicate erroneous transitions caused by an error on L.e.* 17
3.6 *Pipeline of Buffers.* 19
3.7 *Waveforms of Input-channel Variable L.0 and Output-channel Variable R.0, and Input-channel Acknowledgment Variable L.e. A soft error on L.e at 7 ns causes premature acknowledgment of an input communication on L before an output communication on R is generated.* 19

4.1 *TMR QDI Circuit. It cannot tolerate a soft error, because a soft error can abort the communication of data, which is hard to detect in an asynchronous system.* 21

5.1 *Gate in Original Circuit and its Corresponding DD Gate in DD Circuit.* . . . 25

5.2 *Circuit and its Corresponding DD Circuit.* 25

5.3 *Soft Error on CO Variable. The propagation of an error is blocked by the next DD gates.* 29

5.4 *DD Gate with weak C-elements. It can correct a corrupted CI variable in a state-holding state.* 30

5.5 *Tree of C-element and its corresponding DD Tree. The DD tree of C-elements requires cross-coupled C-elements only for the primary outputs.* 31

5.6 *Precharged Half Buffer.* 36

5.7 *DD PCHB.* 38

5.8 *Waveforms of Inputs-channel Variable $L0_a$ and Output-channel Variable $R0_a$, and Input-channel Acknowledgment Variable Le_a of DD One-bit Buffer. A soft error on Le_a at 10 ns merely causes delay on an input communication on the input channel, which does not affect the correction of computation in a QDI circuit.* 39

5.9 *Basic Arbiter. When all inputs (x,y) are assigned to be 1 simultaneously, it provides the outputs of either (1,0) or (0,1) non-deterministically.* 42

5.10 *Duplicated Soft-error Tolerant Arbiter. When all inputs are assigned to be 1 simultaneously, it provides the outputs of either (1,1,0,0) or (0,0,1,1) non-deterministically.* 43

5.11 *DD Arbiter (does not work).* 45

5.12 *Arbiter with Double-checking C-elements (does not work).* 46

5.13 *Block Diagram of Soft-error Tolerant Arbiter.* 47

5.14 *Pass-gate Soft-error Tolerant Arbiter.* 49

5.15 *Waveforms for Nodes in DSET Arbiter. After the duplicated inputs arrive simultaneously, an error in the basic arbiter at 2.2 ns causes v $\downarrow$ and u $\uparrow$ (i.e., the result from a basic arbiter is flipped). Although the transition of one of the duplicated output $u_b \uparrow$ occurs, the transition of the other output $u_a \uparrow$ does not occur until the previous arbitration (i.e., $v_a \uparrow$, $v_b \uparrow$) is acknowledged.* . . . 53

5.16 *FPGA Architecture.* . . . 55

5.17 *(a) Double-checked Programmable Bit (b) Programmable Bit based on Dual Interlocked Cell.* . . . 56

5.18 *Block Diagram of Soft-error Tolerant FPGA Logic Cell.* . . . 56

5.19 *Decomposition of Memory Unit.* . . . 59

5.20 *Array of Storage Elements.* . . . 60

5.21 *Circuit of Mermoy Core.* . . . 61

5.22 *Calin's Dual Interlocked Cell (DICE).* . . . 62

5.23 *Decomposition of [7,4] Hamming-code Decoder. Each solid line represents a one-of-two code channel.* . . . 65

5.24 *Block Diagram of Memory Core Using Hamming Code.* . . . 65

5.25 *Converting a Duplicated Codeword into a Hamming Codeword.* . . . 66

5.26 *Interface Between DD Circuit and Input Side of Memory Unit.* . . . 67

5.27 *Copying the Bit Lines of the Array (does not work).* . . . 68

5.28 *Copying the Bit Lines with Filter (does not work).* . . . 68

5.29 *Problematic Case of Invalid Bit Lines.* . . . 69

5.30 *Interface Between Output Side of Memory Unit and DD Circuit.* . . . 70

5.31 *Mutual Excluder Based on Pass-gate Transformation.* . . . 70

5.32 *Waveforms for Bit Lines and Duplicated Bit Lines from Mutual Excluder. Although two bit lines are flipped simultaneously, only one of the duplicated bit lines is flipped, owing to a mutual excluder.* . . . 71

5.33 *Waveforms for Duplicated Bit Lines from Pass-gate Mutual Excluder. An error, which flips bit lines, occurs at 5.5 ns.* . . . 72

5.34 *Soft Errors on Register File Using DICE. An error in the regfile is corrected quickly.* . . . 73

5.35 *Soft Errors on a Hamming-coded Memory. An error in a DD circuit is corrected quickly, but an error on an SRAM in a Hamming-coded array remains uncorrected.* 74

6.1 *One Error Detecting Delay-insensitive Codes and their Upper Bounds.* . . . 80
6.2 *Block Diagram of PCHB.* 82
6.3 *Block Diagram of WCHB.* 82
6.4 *Example of Delay-insensitive Minterm Logic (Adder).* 83
6.5 *Example of Direct Logic (Adder).* 83
6.6 *Example of Precharging-evaluating Logic (Adder).* 84
6.7 *Part of Function Block of Buffer for [3,2] Parity Code.* 89
6.8 *Part of Function Block of Buffer for [4,2] Repetition Code.* 90
6.9 *Function Block for Repetition Code and its Simplified Function Block. Redundant gates of an original function block are removed.* 91
6.10 *Simplified Function Block for Repetition Code. The DD scheme, which duplicates a gate and adds cross-coupled C-elements, leads to the same design.* 92
6.11 *One-error Detecting Linear Codes for Four-Bit Message.* 94
6.12 *Three-bit Function Blocks based on Repetition Codes vs Non-repetition Codes. The x-coordinate and the y-coordinate of each point correspond to the size of a function block using the repetition code, and the size of a function block using a non-repetition code (e.g., the [4,3] code and the [5,3] code).* 94
6.13 *Four-bit Function Blocks based on the Repetition Codes vs Non-repetition Codes.* 95
6.14 *Five-bit Function Blocks based on the Repetition Codes vs Non-repetition Codes.* 95
6.15 *CC for Concatenation of Two One-of-two Codes.* 97
6.16 *Two-input Parity Checker.* 99
6.17 *Parity Decomposition.* 99
6.18 *CC for the [3,2] code.* 100
6.19 *Simplified CC for the [3,2] code.* 101
6.20 *CC for the [4,2] Code.* 101
6.21 *Size Comparison of Three-bit WCHB Processes.* 102

6.22 *Size Comparison of Three-bit PCHB Processes. The PCHB using a repetition code is always smaller than the PCHB using a non-repetition code, so that the boundary is located outside the range of the figure.* 103
6.23 *Energy Distribution of Three-bit Function Blocks.* 106
6.24 *Energy Distribution of Three-bit PCHB Processes.* 107
6.25 *Energy Distribution of Four-bit PCHB Processes.* 107
6.26 *Distribution of Cycle Time of Three-bit PCHB Processes.* 108
6.27 *Distribution of Cycle Time of Four-bit PCHB Processes.* 109
6.28 *Distribution of* Et^2 *of Three-bit PCHB Processes.* 109
6.29 *Distribution of* Et^2 *of Four-bit PCHB Processes.* 110
6.30 *PCHB based on the [4,2] Repetition Code. It is equivalent to the DD circuit of a 2-bit normal PCHB.* 111
6.31 *PCHB based on the [3,2] Parity Code.* 112
6.32 *Waveforms of Input and Output Channel Nodes of PCHB based on the [3,2] Parity Code. Until an error on the first bit of the input channel is corrected, all transitions of output-channel nodes are delayed.* 113

7.1 *Decomposition of the STAM. The dotted arrows denote control flows, and block arrows denote data flows. Most of the units are protected by the DD scheme.* 117
7.2 *Circuit for Probing Branch. If there is a pending communication on the channel DoBranch, then the value of the channel IsBranch is assigned to be true. If not, the value is assigned to be false.* 122
7.3 *Process Graph of PCUNIT_NOARB.* 124
7.4 *Synthesis Flow from CHP Description to Layout. A box represents a tool, and a cloud shape represents an input description for a program.* 125
7.5 *Floor Plan of the STAM and Locations of Flipped Nodes during Digital Simulation of the STAM. Each dot represents the location of a flipped node, and a box represents the bounding box of a circuit of a decomposed small process, which includes a few hundred transistors.* 128
7.6 *Waveform of One Corrupted Node in PCUNIT. An error occurs at 12 ns, and the corrupted node is restored quickly.* 128

7.7 *Waveform of One Corrupted Node in PCUNIT. An error occurs at 44 ns, which causes a premature transition, but it does not affect the correctness of computation, because cross-coupled C-elements of the DD scheme prevent the propagation of the premature firing.* 129

7.8 *Compared with a Non-DD Circuit, a DD Circuit Requires More Transitions in a Computation Cycle. A circuit for output-channel validity takes its inputs from function blocks, but a corresponding DD circuit takes its inputs from cross-coupled C-elements after function blocks.* 131

G.1 *Byte-skewed Distribution of Control Signals.* 155

List of Tables

2.1 C-element Truth Table 11

5.1 State Table of DD Gate 26

5.2 Performance Figures of PCHB and DD PCHB 38

6.1 Size of Two-bit Input/Ouput Precharging-evaluating Function Block 93

6.2 Performance Figures of PCHB for [4,2] Code and PCHB for [3,2] Code . . . 110

7.1 Performance Comparison between Simple Asynchronous Microprocessor and Its Corresponding Soft-error Tolerant Simple Asynchronous Microprocessor 132

7.2 Performance Comparison with Synchronous Competitors 132

C.1 One-error Detecting Delay-insensitive Code 143

Chapter 1

Introduction

There are two circuit-design paradigms: synchronous circuits and asynchronous circuits. In synchronous circuits, there is a single global clock signal to which all actions are synchronized; a clock signal is fed into each computation module. Each module must finish its computation in a clock period. If something delays the computation, the correctness of the computation cannot be guaranteed.

On the other hand, asynchronous circuits operate without a clock signal. The correctness of asynchronous computation is independent of the computation time of modules, because the computation is based on local communications by handshaking protocol. Figure 1.1 shows a synchronous and an asynchronous pipeline.

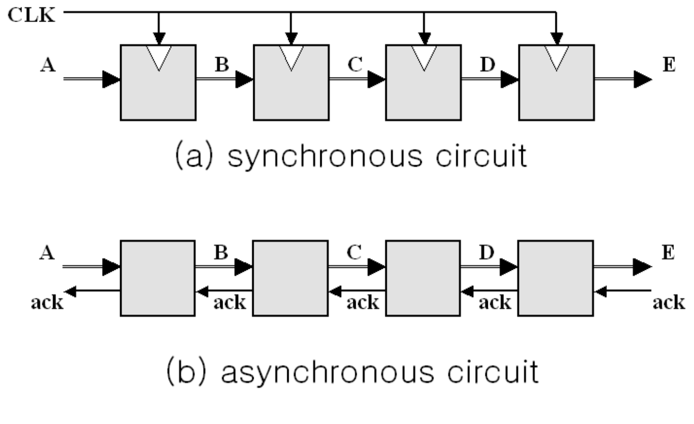


Figure 1.1: *Pipeline of Buffers.*

The quasi delay-insensitive (QDI) design style proposed in the 80s is one of the most robust asynchronous design styles for general computation [1]. The QDI style has one minimal assumption on delays in gates and wires: an electric signal occasionally must be copied to multiple destinations in such a way that the maximum difference in arrival time of the signal at the destinations is small compared to the gate delays. In practice, QDI-circuit designers can easily ensure that the assumption is not violated. Then the correct operation of a QDI circuit is independent of delays in gates and wires, so that QDI circuits are simple, modular, and easy to verify. The recent design of a sub-nanojoule microprocessor has demonstrated that QDI circuits can be very energy-efficient, because only active parts of the system draw power, and the energy spent by the clock is saved [2].

A system of QDI circuits can adapt itself to variations of physical parameters such as supply voltage, fabrication, temperature, doping, and so on. The effect of the variations on synchronous circuits can be severe, but QDI circuits are more robust to the variations, which cause changes in timing of the components. These features make QDI circuits attractive to the designers of reliable systems.

Besides the issues of variation, the tolerance of soft errors is getting even more attention as technology scaling advances. A *soft error* is defined as the erroneous switching of a node when the electrical charges that encode the boolean value of the node are erroneously changed by radiation or other noise sources. Unlike manufacturing defects, a soft error can be corrected by applying the proper charges to the node. The International Technology Roadmap of Semiconductors (ITRS) warns that the continuation of the established semiconductor roadmap is seriously threatened by the increasing occurrence of failures in chip operation [3]. As the feature size of circuits gets smaller, circuits experience more failures of devices and interconnects. To compensate for the inevitable increase of failures, the ITRS urges circuit designers to include error tolerance in their designs, especially for soft (transient) errors caused by radiation or other noise effects.

As a response to the concern, a new method, which is applied entirely at the logic level, is proposed for making QDI circuits tolerant to soft errors. Several soft errors can be simultaneously corrected provided they do not happen too close to each other in space and in time. The method has been demonstrated by designing a simple asynchronous microprocessor.

In the following two chapters, first we shall overview how to design a QDI circuit, and what will happen if a soft error occurs in a QDI circuit. In Chapter 4, we shall briefly review some soft-error tolerant methods for synchronous circuits and asynchronous circuits. In Chapter 5, the crucial components of the method are shown: (1) a special kind of duplication for random logic (when each bit has to be corrected individually), (2) special protection circuitry for arbiter and synchronizer (as needed for example for external interrupts), (3) reconfigurable circuits using a special configuration unit, and (4) error correcting for memory arrays and other structures in which the data bits can be self-corrected. In Chapter 6, the solution of protecting random logic is compared with alternatives which use other types of error correcting codes (e.g., parity code) in a QDI circuit. It turns out that the duplication generates efficient circuits more commonly than other possible constructions. Finally, the design of a soft-error tolerant asynchronous microprocessor (STAM) is detailed, and testing results of the soft-error tolerance of the STAM are shown in Chapter 7.

It is worth mentioning that a QDI circuit by itself is able to handle more malicious analog effects than a synchronous circuit. For example, radiation hitting a chip can change the charges on a node, not enough to make the bit flip, but enough to make a transition on the node much slower. On the other hand, radiation-dose effects accumulate in the chip substrate and slowly change the threshold voltages, affecting the timing. Both effects change the physical parameters of the system but do not switch a node. The changes in timing may be catastrophic for a synchronous system but can be completely transparent to a QDI system. In this way, soft-error tolerant QDI circuits can be suitable for applications where soft-error tolerance, combined with the advantages of QDI circuits (e.g., robustness to variations and low power), is needed.

Chapter 2

Designing Quasi Delay-insensitive (QDI) Circuits

One way of designing QDI circuits involves first writing a high-level description of a system, which is a sequential program in the communicating hardware processes (CHP) language. The sequential program is decomposed into concurrent small CHP processes that are small enough to be easily compiled into the intermediate handshaking expansion (HSE) language. The HSE description is subsequently transformed into a production rule set (PRS) that is the canonical representation of a QDI circuit and is the lowest-level description in the synthesis method. These compilation steps allow for the design of a transistor-level circuit which correctly implements a given high-level specification of a system.

2.1 Communicating Hardware Processes

A CHP program consists of one or more concurrent processes, each of which is a sequential program. There are no shared variables between concurrent processes; they communicate via *channels* that connect two processes.

For example, a CHP process in a CHP program is as follows:

$$P \equiv \;*[A?a, B?b;\; ...;\; F!f(a,\; b,\; ...),\; G!g(a,\; b,\; ...);\; ...;\; Z;\; ....]$$

$*[S]$ means "repeat S forever." $S1; S2$ means a sequential execution of $S1$ and $S2$. The process P receives messages from channel A and channel B, and sends out computation results to channel F and channel G. Then it waits for the action of channel Z, which is a synchronization channel between processes. And if the sending-side process sends messages n times as well to channel A, the receiving-side process will receive messages n times from channel A and vice versa. If the process on one end of a channel is not yet ready to send or receive, the process on the other end will stay waiting.

In the CHP program, there are other processes that send out messages to channel A and channel B, receive messages from channel F and channel G, and synchronize with channel Z, as follows:

$$\begin{aligned} Q &\equiv \; *[\ldots; A!0; \ldots; F?f; \ldots] \\ R &\equiv \; *[\ldots; B!1; \ldots; G?g; \ldots] \\ S &\equiv \; *[\ldots; Z; \ldots] \end{aligned}$$

2.2 Handshaking Expansion

We can transform a CHP description into a HSE description, where everything is described in boolean notation. Before compiling a CHP description into a HSE description, we decompose a big CHP process into a set of small concurrent processes for simple circuit-level implementation later.

Let us consider a synchronization channel X without data communication between two processes:

$$\begin{aligned} P_0 &\equiv \; *[\ldots; X; \ldots] \\ P_1 &\equiv \; *[\ldots; X; \ldots] \end{aligned}$$

If P_0 encounters the channel action X, then it will wait until P_1 reaches to the channel action X and vice versa. The communication channels with neighboring processes are replaced with elementary actions such as waiting and assignment of boolean variables. The elementary actions are specified by a protocol such as a four-phase handshaking protocol. For example, P_0 sets one boolean variable x_o to let P_1 know that P_0 initiates the communication; P_1 sets the other variable x_i to let P_0 know that P_1 acknowledges the communication initiated by P_o. Then the two variables are reset. This is the four-phase handshaking protocol. In the process P_0, channel X is implemented as follows:

$$x_o\uparrow; [x_i]; x_o\downarrow; [\neg x_i]$$

which is called *active*. And channel X in the process P_1 is implemented as follows:

$$[x_o]; x_i\uparrow; [\neg x_o]; x_i\downarrow$$

which is called *passive*. $x_o\uparrow$ is equivalent to x :=`true` and `[`x_i`]` means that the process waits until x_i becomes `true`. Thus the sequence of events during a channel communication is as follows:

$x_o\uparrow$; `[`x_o`]`; $x_i\uparrow$; `[`x_i`]`; $x_o\downarrow$; `[`$\neg x_o$`]`; $x_i\downarrow$; `[`$\neg x_i$`]`

We introduce an alternative form of active implementation, called *lazy active*, which is more amenable to implementation:

`[`$\neg x_i$`]`; $x_o\uparrow$; `[`x_i`]`; $x_o\downarrow$;

The four-phase handshake protocol uses one variable for initialization and the other variable for acknowledgment. For data communication, a variable is replaced with a set of variables, which implements a delay-insensitive code such as one-of-n code. (The details of a delay-insensitive code are explained in Chapter 6.) For brevity, the variables used in the implementation of a channel are called *channel variables*, which consist of *channel code variables* and a *channel acknowledgment variable*.

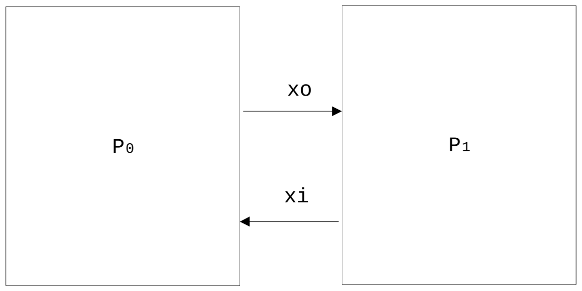


Figure 2.1: *Handshake Expansion of a Synchronization Channel.*

2.3 Production Rule Set and PRS Computation

A HSE description is transformed into a *production rule set* (PRS), which has no explicit sequencing. A PRS is a concurrent composition of production rules, and the execution of a PRS is a concurrent execution of the production rules in the set. Each production rule (PR) has the form $G \to S$, where G is a boolean expression of boolean variables called the *guard* of the PR, and S is a boolean *assignment*. The assignment is written as $z\uparrow$ or $z\downarrow$, which corresponds to z :=`true` or z :=`false`. An execution of a PR $G \to S$ is an unbounded sequence of *firings*. A firing of $G \to S$ when G is `true` amounts to the execution of S, and a firing with G `false` amounts to a skip. If the firing of a PR does change the value of any variable, the firing is called *effective*. For brevity, we shall refer to all effective firings simply as firings.

A PR $G \rightarrow S$ in a PRS is said to be *stable* if whenever G becomes `true` it remains `true` until the assignment S is completed. $G1 \rightarrow z\uparrow$ and $G2 \rightarrow z\downarrow$ are *non-interfering* if and only if $\neg G1 \vee \neg G2$ holds in every execution. Stability and non-interference guarantee that the execution result of PRS is deterministic. In physical implementation, an unstable PR can generate a glitch, which may cause a QDI circuit to malfunction. The interference manifests itself as a short circuit, which consumes power excessively, may damage the circuit physically, and also leads to indeterminate logic values.

The two complementary PRs that set and reset the same variable, such as

$$G1 \rightarrow z\uparrow$$
$$G2 \rightarrow z\downarrow,$$

comprise a *gate*. The variables in the guards are the *inputs* of the gate and the variable in the assignment is the *output* of the gate. If $G1 \neq \neg G2$ holds, then z is a *state-holding variable*, and the gate is a *state-holding gate*; if $G1 = \neg G2$ holds, then z is a *non-state-holding variable*, and the gate is a *combinational gate*. (These variables correspond to electrical nodes in the physical implementation.)

A QDI circuit is an interconnection of gates, interacting with its environment. Each input of a gate is either connected to the output of another gate, or to an environment. The output of a gate may be connected to any number of inputs of other gates, as well as to the environment. An input of a gate that is connected to the environment is a *primary input*; an output of a gate that is connected to the environment is a *primary output*.

The environment of a QDI circuit sets values of primary inputs by reacting to values of primary outputs of the circuit. A pair of a circuit and an environment, like Figure 2.2, is called a *system*.

If a PRS uses boolean variables $x_1, x_2, ..., x_n$ for guards and assignments, then the *state* of the PRS can be represented as an n-tuple of boolean values. Interchangeably we shall use '0' for `false` and '1' for `true`. And $s[x_k]$ is the value of x_k in the state s. A PR P in the state s is called *enabled* if and only if the guard of P is `true` in the state s. A PR P is called *effective* in the state s if and only if the firing of P in state s changes the value of a

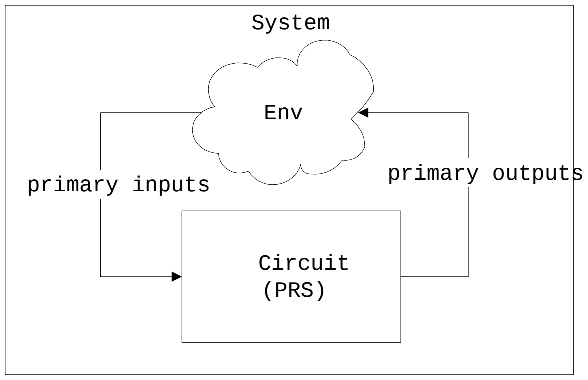


Figure 2.2: *Circuit and Environment.*

variable. An *execution path* of a PRS is a trace of firings of PRs from an initial state, which is described as $< P_1, ..., P_{m-1}, P_m >$, where P_i is a PR. An *execution-path set* of a PRS is the set of every possible execution path from an initial state of the PRS.

A *PRS computation (PRSC)* is defined as follows:

- Two disjoint sets: Σ_{Env}, called the environment, and $\Sigma_{Circuit}$, called the circuit, whose elements are PRs. ($\Sigma \overset{\text{let}}{=} \Sigma_{Env} \cup \Sigma_{Circuit}$.)
- An initial state $s_0 \in \{0,1\}^n$. (n is the number of distinct variables in Σ.)
- An execution-path set EP.

An *environment path* of an execution path is a projection of the execution path onto Σ_{Env}. A finite set S_v is called a *valid-state set* if its elements are states reachable from s_0 by firing of PRs in Σ. A PRS computation can be represented as a transition diagram. The vertices of the diagram correspond to the valid states in S_v. If a production rule P is effective in a state s, and the state s is turned into a state s' due to the firing of the production rule, then there is an edge labeled P from s to s' in the transition diagram.

2.4 Example of QDI Circuits: Buffer

The CHP description of a buffer, which is an essential component of an asynchronous system, is $*[L; R]$. Two boolean variables $L.i$ and $L.e$ implement the input synchronization channel L of the buffer, and likewise two variables $R.i$ and $R.e$ implement the output synchronization channel R. The corresponding HSE is

$$*[[L.i]; L.e\downarrow; [\neg L.i]; \quad L.e\uparrow; [R.e]; \quad R.i\uparrow; \quad [\neg R.e]; \quad R.i\downarrow;],$$

where L is passive, and R is lazy-active. A direct implementation of the HSE description produces an inefficient circuit, which requires extra variables and, ultimately, extra transistors. Instead, it is better to permute parts of the HSE to reduce the amount of sequencing and the number of extra variables. The transformation is called *reshuffling*, which is the source of significant optimization opportunities. Although there are several possible reshufflings, three reshufflings are commonly used: *precharged full buffer* (PCFB), *precharged half buffer* (PCHB) and *weak-conditioned half buffer* (WCHB) [4].

For example, the HSE based on the PCHB template is

$$*[[R.e \wedge L.i]; \quad R.i\uparrow; \quad L.e\downarrow; \quad [\neg R.e]; \quad R.i\downarrow; \quad [\neg L.i]; \quad L.e\uparrow].$$

A corresponding PRS is

$$\begin{aligned} L.e \wedge R.e \wedge L.i &\rightarrow R.i\uparrow \\ \neg L.e \wedge \neg R.e &\rightarrow R.i\downarrow \\ L.i \wedge R.i &\rightarrow L.e\downarrow \\ \neg L.i \wedge \neg R.i &\rightarrow L.e\uparrow. \end{aligned}$$

If the PRS of the environment is

$$\begin{aligned} L.e &\rightarrow L.i\uparrow \\ \neg L.e &\rightarrow L.i\downarrow \\ R.i &\rightarrow R.e\downarrow \\ \neg R.i &\rightarrow R.e\uparrow, \end{aligned}$$

and the initial state is $(L.i, L.e, R.i, R.e) = (0, 1, 0, 1)$, then the PRSC for the PCHB system is

- $\Sigma_{Env} = \{L.e \rightarrow L.i \uparrow, \neg L.e \rightarrow L.i \downarrow, R.i \rightarrow R.e\downarrow, \neg R.i \rightarrow R.e\uparrow\}$,
  $\Sigma_{Circuit} = \{L.e \wedge R.e \wedge L \rightarrow R.i \uparrow, \neg L.e \wedge \neg R.e \rightarrow R.i \downarrow, L.i \wedge R.i \rightarrow L.e\downarrow, \neg L.i \wedge \neg R.i \rightarrow L.e\uparrow\}$

- Initial state $s_0 = (0, 1, 0, 1)$

- The execution-path set EP is as follows: $\{< L.i\uparrow >, < L.i\uparrow, R\uparrow >, < L.i\uparrow, R\uparrow, R.e\downarrow>, < L.i\uparrow, R\uparrow, L.i\downarrow>, < L.i\uparrow, R\uparrow, L.i\downarrow, R.e\downarrow>, ...\}$.

Accordingly the environment-path set of the PRSC is as follows:

$$EnvP = \{< L.i{\uparrow} >, < L.i{\uparrow}, R.e{\downarrow}>, < L.i{\uparrow}, L.i{\downarrow}>, \ < L.i{\uparrow}, L.i{\downarrow}, R.e{\downarrow}>, ...\}.$$

The transition diagram of the PCHB is shown in Figure 2.3, and a circuit diagram of the PCHB in CMOS technology is shown in Figure 2.4.

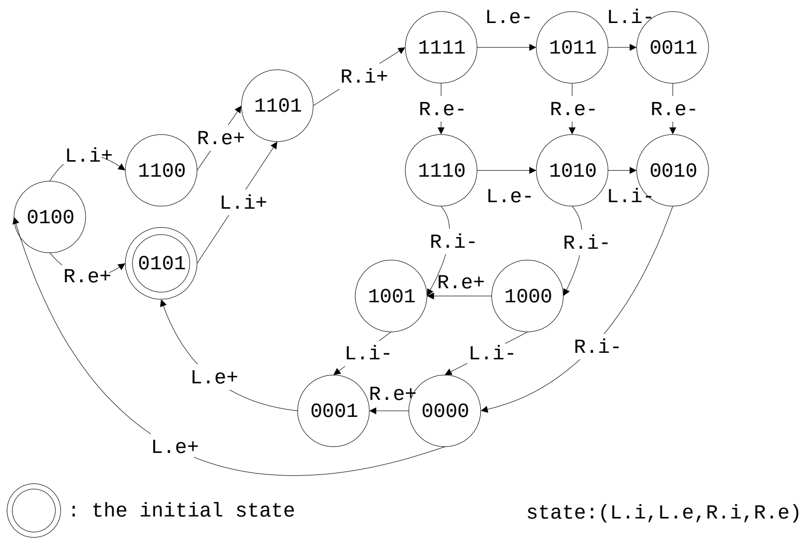


Figure 2.3: *Transition Diagram of Precharged Half Buffer. Each state represents the values of (L.i,L.e,R.i,R.e). The double-circled state is the initial state.*

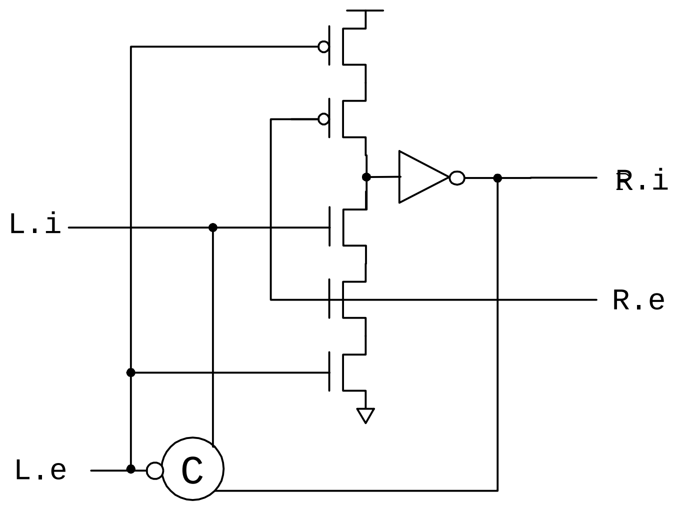


Figure 2.4: *Circuit Diagram of Precharged Half Buffer.*

Note that in the circuit diagram, a C-element, which was introduced by David E. Muller, is represented as a circle with the letter ‘C’, which is a commonly used asynchronous logic

Table 2.1: C-element Truth Table

x	y	z
0	0	0
1	1	1
0	1	previous value retained
1	0	previous value retained

component. The output of a C-element reflects the inputs when the values of all inputs match. One possible CMOS implementation of a C-element is shown in Figure 2.5, and its truth table is shown in Table 2.1.

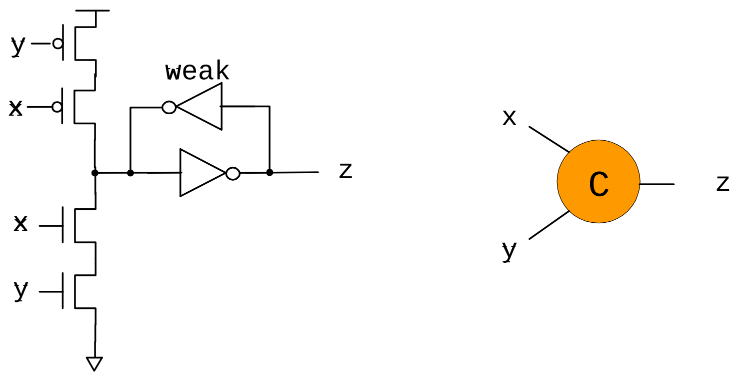


Figure 2.5: *Circuit Diagram of C-element and its Gate Diagram.*

Chapter 3

Soft Errors in QDI Circuits

In this chapter, different types of errors are overviewed first, and a digital model and an analog model of soft errors are shown. Then the effects of soft errors in QDI circuits are examined.

3.1 Errors on Circuits

Failures of a system at the circuit level can be broadly categorized as either manufacturing defects or operational errors. Manufacturing defects such as single stuck-at faults arise from a range of processing problems during fabrication. For example, improper doping in the channel of a transistor may cause a change in the threshold voltage and timing of the transistor, and may make the transistor unusable. Therefore testing methods for detecting manufacturing defects have been extensively studied. Commonly used testing methods involve adding extra test circuits to the system. While the test circuits are transparent in normal operations, the circuits are activated to detect defects in a special mode [5]. The method of testing QDI circuits has been also studied, which shows that defects in QDI circuits are testable [6].

While defects occur in manufacturing processes, operational errors happen unpredictably during the lifetime of a circuit. Operational errors can be subdivided into two types: hard (permanent) errors and soft (transient) errors. A hard error is an error that damages a circuit irrevocably. For example, when a high-energy particle activates a parasitic transistor to trigger a positive feedback, the current caused by the positive feedback can exceed the device's maximum specification and can destroy the device [7]. In addition to incident high-energy particles, there are other types of destructive causes such as electromigration [8].

On the other hand, soft errors deposit sufficient charges on the nodes of a circuit to flip the logic states without causing any physical damage to the circuit. They result in transient, inconsistent corruption of data. One of the main causes of soft errors is alpha particles from IC packages [9, 10]. A positively charged alpha particle from radioactive decay travels through a circuit and disturbs the distribution of electrons, as shown in Figure 3.1. If the disturbance is large enough, the value of a node can change from `false` to `true` or vice versa. In addition to alpha particles from the package, other types of radiation such as cosmic rays from space can also cause soft errors [11]. Many different particles are present in cosmic rays, but a major cause of soft errors is known to be neutrons. Since neutrons are uncharged, they cannot disturb electron distribution on their own, but they can undergo neutron capture by the nucleus of an atom in the circuit, generating an unstable isotope that produces an alpha particle when it decays. Computers operated at high altitudes or in aircraft experience an order of magnitude higher rate of soft errors compared to sea level, because the flux of cosmic ray depends on altitude. This is in contrast to alpha-particle decay from package, which does not change with location. Other than the radiation effects, soft errors can also be caused by random noise such as inductive or capacitive crosstalk, power supply noise, charge sharing, leakage noise, and so on [12, 13, 14].

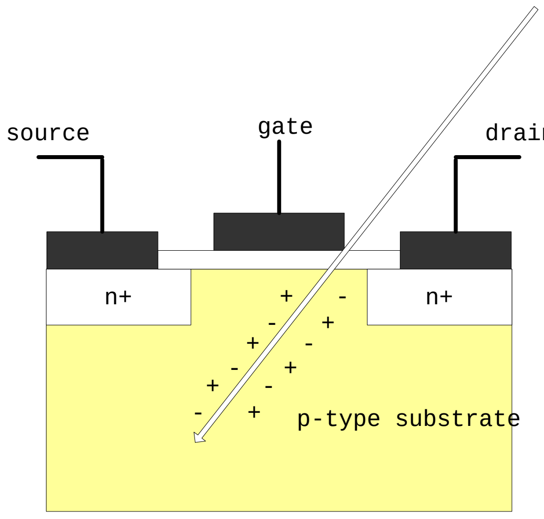


Figure 3.1: *Interaction of a Particle and Silicon Substrate.*

Soft-error tolerance of logic circuits has recently been receiving more attention, since the soft-error rate of advanced CMOS devices is higher than before [15]. For the past three decades, dense memories, including DRAM and SRAM, have been known to be more susceptible to soft errors than logic circuits. For example, the typical soft-error rate for SRAM circuits for 90 nm is reported to be about 1000 kFIT/Mbit, and a soft-error rate for logic circuits for 90 nm is reported to be about 100 kFIT/Mbit [16]. (Failure in time (FIT) is defined as the number of errors per one billion hours.) But it is anticipated that the soft-error rate for logic circuits will increase by nine orders of magnitude between 1992 to 2011, at which point it will be comparable to the soft-error rate for unprotected memory elements [17].

3.2 Digital Model of Soft Errors

A soft error is modeled as flipping the value of a single variable in a PRS; a PRS Computation (PRSC) with a soft error is defined as follows:

- Two disjoint finite sets Σ_{Env}, called the environment, and $\Sigma_{Circuit}$, called the circuit, whose elements are PRs. ($\Sigma \overset{\text{let}}{=} \Sigma_{Env} \cup \Sigma_{Circuit}$.)
- An initial state $s_0 \in \{0,1\}^n$. (n is the number of distinct variables in Σ.)
- A soft-error execution-path set EP_{error}.

A soft-error execution path can include a symbol $error_{x_i}$ representing an error on x_i. For example, a soft-error execution path is $< P_1, ..., P_{k-1}, error_{x_i}, P_{k+1} ..., P_{m-1}, P_m >$, which infers that a soft error occurs on x_i after the firing of the PR P_{k-1}, and the value of x_i is flipped. Elements of the valid-state set S are states reachable from the initial state s_0 only by firing of PRs in Σ; elements of the invalid-state set Q are states reachable only with a soft error on a variable. The vertices of the transition diagram correspond to states in S or Q. If $s[x_i] \neq s'[x_i]$ and $s[x_j] = s'[x_j]$ for all $j \neq i$, then there is a two-way edge labeled $error$ between s and s' in the transition diagram, because an error on x_i flips s to s' and vice versa.

There are two types of erroneous computations that may be caused by an error. (For brevity, sometimes we will refer to soft errors as simply errors.)

Deadlock An error may put a system into an invalid state such as the state q, as shown in Figure 3.2 (a). The invalid state cannot be reached in a normal execution. If no PR is effective in the invalid state, deadlock occurs.

Abnormal Computation An error may cause a transition from a valid state to another valid state. Or an error may put a system into an invalid state where some PRs are effective, and a system is put into a valid state after firings of PRs. Meanwhile some states are possibly skipped or revisited; the corresponding firings of PRs are skipped or repeated due to the abnormal transition, as shown in Figure 3.2 (b). In this case, data can be missed or can be generated accidentally.

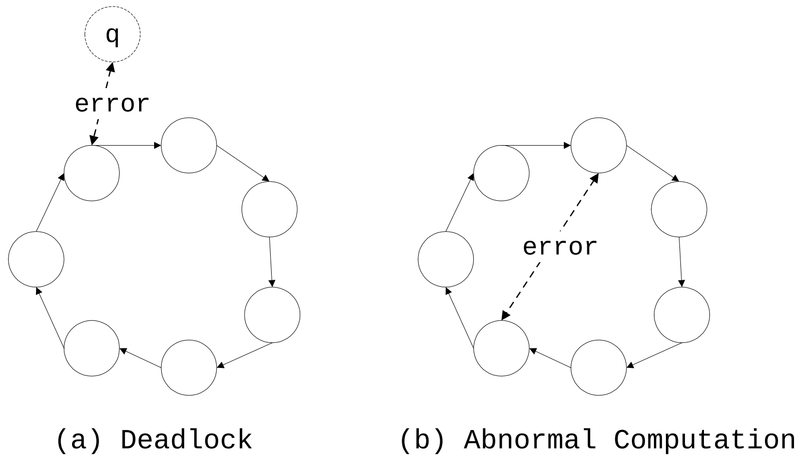


Figure 3.2: *Examples of Erroneous Computations Caused by Soft Error.*

As an example of a PRSC with a soft error, let us consider a soft error on $L.e$ in the buffer from the previous chapter. The PRSC with a soft error on $L.e$ is as follows:

- Σ_{Env} and $\Sigma_{Circuit}$ are the same as before.
- An initial state $s_0 = (0, 1, 0, 1)$.
- $EP_{error} = \{< error_{L.e} >, < L.i \uparrow >, ..., < L.i \uparrow, error_{L.e}, L.i \downarrow, L.e\uparrow, L.i \uparrow >, ..., \}$.

The transition diagram, including transitions caused by the error, is shown in Figure 3.3. In a normal computation, there are the same number of transitions on output variable $R.i$ as on input variable $L.i$ from the specification of a buffer. (Strictly speaking, the number of output transitions of a buffer can be fewer or the same. But in the shown implementation, the next input set assignment (i.e., $L.i \uparrow$) cannot start until the output reset assignment (i.e., $R.i \downarrow$) is done.) But an error on $L.e$ can cause premature acknowledgment of the communication on the channel L before an output communication on the channel R is generated. That is, compared to the normal execution cycle as shown in Figure 3.4, the erroneous cycle (i.e., $(L.i\uparrow, error, L.i\downarrow, L.e\uparrow)$) skips transitions such as $R.i\uparrow$, as shown in Figure 3.5.

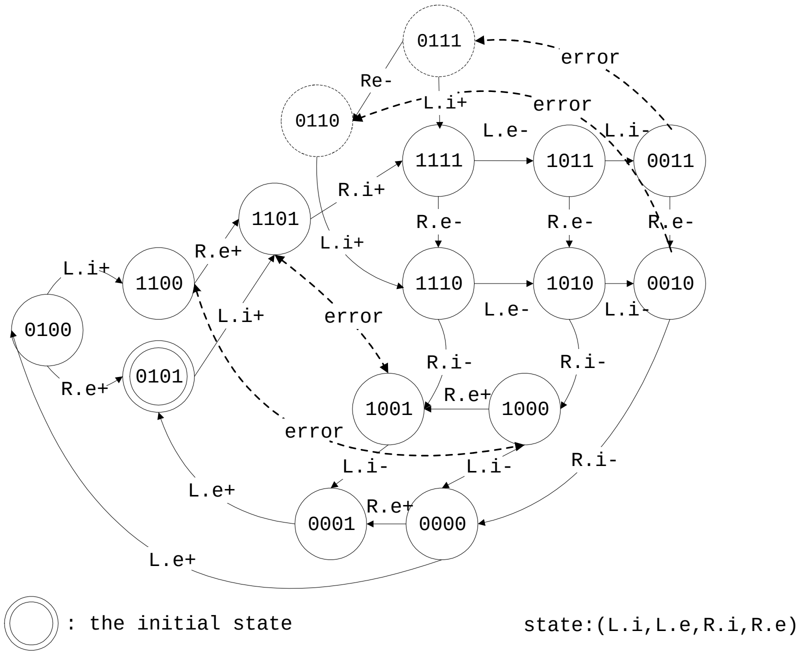


Figure 3.3: *Transition Diagram of PCHB with Soft Error on L.e. The dotted circles indicate invalid states, and the dotted edges indicate erroneous transitions caused by an error on L.e.*

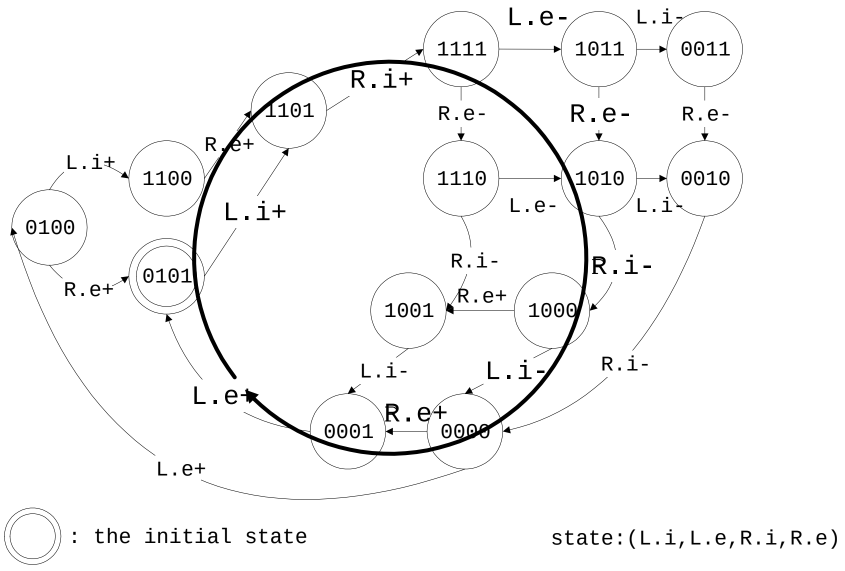


Figure 3.4: *Example of Normal Execution Path of Firings in PCHB.*

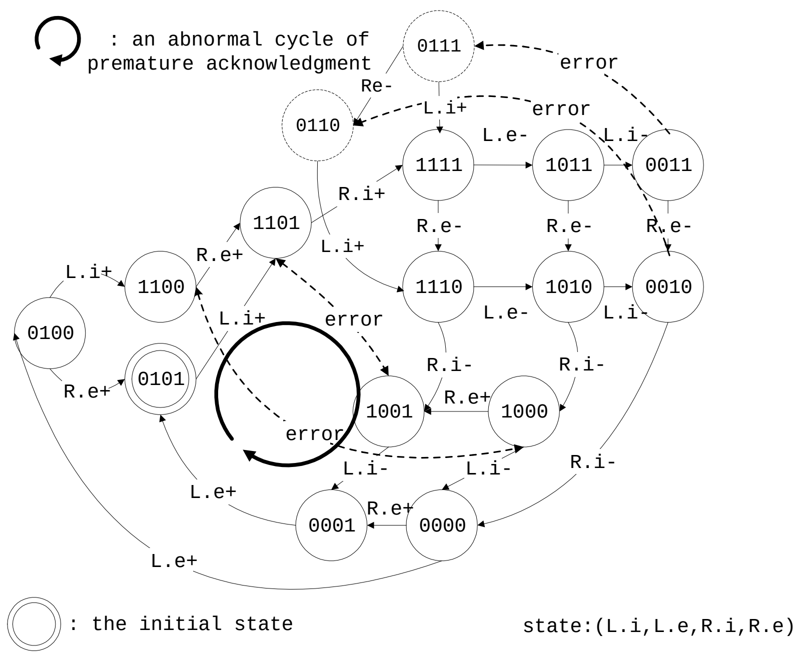


Figure 3.5: *Transition Diagram of Buffer with Soft Error on L.e. The dotted circles indicate invalid states, and the dotted edges indicate erroneous transitions caused by an error on L.e.*

3.3 Analog Model of Soft Errors

We can also identify the erroneous computation in SPICE simulation of a linear pipeline (an array of L-R buffers), as shown in Figure 3.6. A PRS of a one-bit PCHB is as follows:

$$\begin{array}{lcl}
en \wedge R.e \wedge L.0 & \to & \_r.0\downarrow \\
\neg en \wedge \neg R.e & \to & \_r.0\uparrow \\
en \wedge R.e \wedge L.1 & \to & \_r.1\downarrow \\
\neg en \wedge \neg R.e & \to & \_r.1\uparrow \\
\_r.0 & \to & R.0\downarrow \\
\neg\_r.0 & \to & R.0\uparrow \\
\_r.1 & \to & R.1\downarrow \\
\neg\_r.1 & \to & R.1\uparrow
\end{array}$$

$$\begin{array}{lcl}
L.0 \vee L.1 & \rightarrow & \_lv\downarrow \\
\neg L.0 \wedge \neg L.1 & \rightarrow & \_lv\uparrow \\
\_lv & \rightarrow & lv\downarrow \\
\neg\_lv & \rightarrow & lv\uparrow \\
\_r.0 \wedge \_r.1 & \rightarrow & rv\downarrow \\
\neg\_r.0 \vee \neg\_r.1 & \rightarrow & rv\uparrow
\end{array}$$

$$\begin{array}{lcl}
lv \wedge rv & \rightarrow & L.e\downarrow \\
\neg lv \wedge \neg rv & \rightarrow & L.e\uparrow \\
L.e & \rightarrow & \_en\downarrow \\
\neg L.e & \rightarrow & \_en\uparrow \\
\_en & \rightarrow & en\downarrow \\
\neg\_en & \rightarrow & en\uparrow
\end{array}$$

Variables $L.0$, $L1$ encoding a one-of-two code and an acknowledgment variable $L.e$ implement the input data channel L. Likewise variables $R.0$, $R.1$, and $R.e$ implement the output data channel R.

SPICE simulations of the pipeline have been done in TSMC 0.18-μm CMOS technology and at an operating voltage of 1.8 V. A soft error can be modeled in SPICE as a current pulse [18]. If the current pulse is short enough, compared to the response time of the gates, the specific shape of the pulse is not critical to the response of the circuit to the charge injection, so that we can model a soft error as a square current pulse for simplicity.

If the buffer works correctly, there should be the same number of transitions on output variable $R.0$ as on input variable $L.0$: the number of rising and falling signals on $R.0$ is the same as that of rising and falling signals on $L.0$. However when a charge is injected at the node $L.e$ at 7.0 ns, the assignment $R.0\uparrow$ is missed: there is no output signal between 7 ns and 8 ns, as shown in the upper panel of Figure 3.7. Since one variable is used to acknowledge communications between modules (buffers), an error on this variable can have catastrophic results.

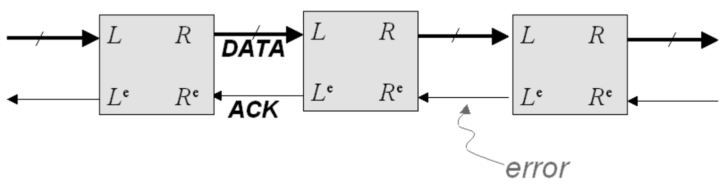


Figure 3.6: *Pipeline of Buffers.*

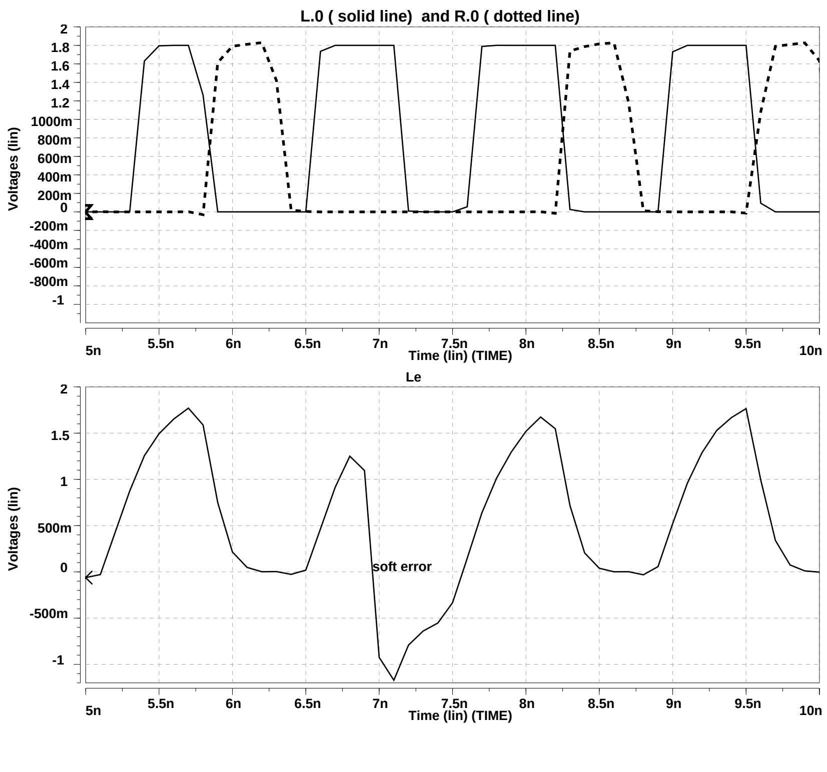


Figure 3.7: *Waveforms of Input-channel Variable L.0 and Output-channel Variable R.0, and Input-channel Acknowledgment Variable L.e. A soft error on L.e at 7 ns causes premature acknowledgment of an input communication on L before an output communication on R is generated.*

Chapter 4

Related Work

In order to cope with soft errors in a circuit, several approaches have been explored. At the device level, the silicon-on-insulator (SOI) process technology helps to increase the resistance of circuits to soft errors, especially those caused by radiation, because of reduced charge collection depth [19].

Traditional techniques for providing soft-error tolerance at the logic level rely on *triple modular redundancy* (TMR), in which a given circuit is triplicated, and a majority voting circuit is used to determine the final output [20]. Sometimes a voting circuit is also triplicated to tolerate an error on the voting circuit itself. Although TMR is a straightforward solution for synchronous circuits, it cannot readily be applied to QDI circuits. A TMR version of a QDI circuit is shown in Figure 4.1. The QDI voting circuit of the design waits for input data communications from three copies of a component. Since an error on one of the triple copies of a QDI circuit results in aborting the communication of the data, the voting circuit possibly waits indefinitely for the delivery of aborted data. Unlike clocked systems, a QDI component may deadlock in the presence of a soft error. Since there is no notion of time in QDI circuits, it is not trivial to adapt the TMR scheme to QDI circuits.

Error correcting codes (ECCs) are also widely used. Except for repetition codes (e.g., duplication and triplication), ECCs are rarely used for control logic because circuits based on ECCs tend to be complex. On the other hand, ECCs are mostly suitable for a large memory array, where we can exploit the density of a complex ECC, and can minimize the cost of decoding and encoding by having just one encoder and decoder for a whole memory [21]. An example of applying a Hamming code to the instruction memory of a 8051 microprocessor has been demonstrated [22].

When an ECC is applied to an array of SRAMs, the decoder corrects data later when the

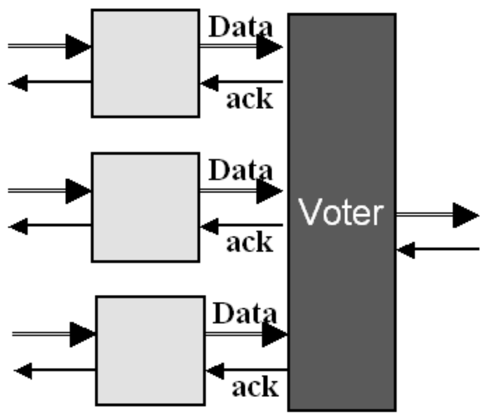


Figure 4.1: *TMR QDI Circuit. It cannot tolerate a soft error, because a soft error can abort the communication of data, which is hard to detect in an asynchronous system.*

data are accessed. On the other hand, self-correcting memory cells correct an error locally, even though the cells require more transistors than an SRAM. A self-correcting memory cell, which stores data in two different locations within the cell, restores corrupted data with an appropriate feedback. There exist several styles of self-correcting memory cells, and each style organize the feedback transistors differently [23, 24]. We shall adapt one of these self-correcting memory cells in order to design a soft-error tolerant asynchronous memory.

Besides the approaches based on TMR and ECCs (namely, space redundancy), approaches based on time redundancy have been also explored. For example, one circuit-level approach employs time-domain TMR, which samples the result of combinational gates multiple times [25]. Due to a larger sampling window, the approach incurs a performance overhead. The other approach for mitigating soft errors uses delay elements, which prevent malicious current pulse from spreading to latches, but it assumes short delays in restoring a soft error [26].

We can apply the time redundancy not to hardware but to software. There are several software-based schemes that execute duplicated instructions over duplicated data and compares the results to check their integrity, called rollback recovery [27, 28, 29]. Although the software-based approaches do not require much hardware modification, they incur speed penalty. Evaluation of the cost of these approaches for four different benchmark programs shows a speed penalty varying 110% to 354% [30]. Multithreading is another software-based transient fault detection and recovery approach that reduces memory and speed cost by exploiting the capabilities of modern processors to execute multiple threads of computation [31]. However these software-based approaches are limited to microprocessors.

A variety of soft-error tolerant schemes for synchronous circuits has been studied and compared [32, 33]. Meanwhile, a few error-detection methods and error-tolerance techniques for QDI circuits have been proposed. A duplication method has been employed to provide an indication of an error in asynchronous circuits, but the duplication method can fail to detect an error, since it assumes the maximum delay between an output from one circuit and an output from its counterpart circuit [34]. Although the approaches can improve the robustness of QDI circuits, these methods require significant timing assumptions to detect errors, and do not always guarantee the error tolerance of a QDI circuit [35, 36]. On the contrary, exploiting the stability of a QDI circuit, we can make a duplicated QDI circuit soft-error tolerant by adding cross-coupled C-elements, which will be demonstrated in the following chapter.

A reconfiguration method for designing a soft-error tolerant asynchronous circuit is presented, which is different from the above approaches [37]. In the reconfiguration method, by forcing an asynchronous circuit to stall in the case of an error, specific self-reconfiguration logic is activated by a detector of the stalling (e.g., watchdog timer). Then the array circuit is reconfigured around the faulty components and consequently the system recovers from errors. This approach incurs large overheads and is suitable for specific structures such as an adder.

Chapter 5

Protecting QDI Circuits from Soft Errors by Design

The key idea for protecting a system from an error is adding redundancy. We shall show how to make a QDI circuit soft-error tolerant by (1) detecting a soft error with duplicated gates and (2) preventing the propagation of an error with cross-coupled C-elements. This method is called the *duplicated double-checking* (DD) scheme. Although duplication by itself is not enough to correct an error, the duplicated QDI circuit with cross-coupled C-elements can correct an error by exploiting the stability property of QDI circuits. That is, if an error occurs on the output of one of duplicated gates, the cross-coupled C-elements prevent the corrupted output from propagating to the environment; the stable inputs eventually restore the proper outputs while the computation of QDI circuits is delayed. Since additional delay is transparent to QDI circuits, the result of the computation is not affected.

The DD scheme is general enough to be applied to most parts of a QDI system. However, arbiters and synchronizers, which are required for handling interrupts and interfacing a synchronous domain and an asynchronous domain, need special protection circuits, called mutual excluders, because of their non-deterministic feature. In addition, arrays of bit-storage units in a memory do not rely on the DD scheme for error tolerance. In bit-storage units such as SRAMs, the stability property cannot be used to restore the value of a corrupted variable. Once data is written in a storage unit, the inputs that triggered the writing do not persist. Hence, solutions based on error correcting codes are applied in memories instead of the DD scheme.

In the following sections, we first show that a QDI circuit based on the DD scheme is capable of tolerating an error. Then we show how to design a soft-error tolerant arbiter, field programmable gate arrays (FPGA), and memory.

5.1 Duplicating QDI Circuits with Cross-coupled C-elements

5.1.1 Duplicated Double-checking QDI Circuit

In the *duplicated double-checking* (DD) scheme, all gates in the original circuit are duplicated, and then state-holding variables are double-checked, which means that two cross-coupled C-elements are placed on state-holding variables. The reason cross-coupled C-elements for non-state-holding variables are inessential, is explained at the end of the proof of Theorem 1.

A DD gate consists of two plain gates and two cross-coupled C-elements, called *double-checking C-elements*, as shown in Figure 5.1. The C-elements share the inputs z'_a, z'_b, called *checked-in (CI) variables*, whose outputs are z_a and z_b, called *checked-out (CO) variables.* A set of CI variables and CO variables, such as z_a, z_b, z'_a, and z'_b, are called *cross-coupled variables.* A gate in PRS is

$$\begin{aligned}
G_p(\ldots, x, \ldots) &\rightarrow z\uparrow \\
G_n(\ldots, x, \ldots) &\rightarrow z\downarrow,
\end{aligned}$$

and a corresponding DD gate is

$$\begin{aligned}
G_p^a(\ldots, x_a, \ldots) &\rightarrow z'_a\uparrow \\
G_p^b(\ldots, x_b, \ldots) &\rightarrow z'_b\uparrow \\
G_n^a(\ldots, x_a, \ldots) &\rightarrow z'_a\downarrow \\
G_n^b(\ldots, x_b, \ldots) &\rightarrow z'_b\downarrow \\
z'_a \wedge z'_b &\rightarrow z_a\uparrow, z_b\uparrow \\
\neg z'_a \wedge \neg z'_b &\rightarrow z_a\downarrow, z_b\downarrow.
\end{aligned}$$

(For brevity, two PRs $G \rightarrow x\uparrow$ and $G \rightarrow y\uparrow$ are written as $G \rightarrow x\uparrow, y\uparrow$.) The variables x_a and x_b are copies of the variable x, and the variables z_a and z_b are copies of the variable z, and so forth.

A DD circuit consists of duplicated gates and DD gates. Figure 5.2 shows a circuit and its corresponding DD circuit. If the PRS of a circuit is stable and non-interfering, then the PRS of its corresponding DD circuit is also stable and non-interfering. As a matter of fact, the DD PRS has a *strong stability*: whenever a guard (e.g., G_p^a) becomes `true`, it remains

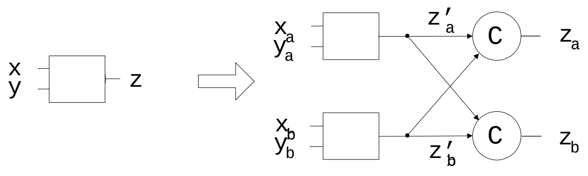


Figure 5.1: *Gate in Original Circuit and its Corresponding DD Gate in DD Circuit.*

`true` until both assignments of duplicated variables (e.g., $z_a\uparrow$, $z_b\uparrow$) are completed. The inputs of duplicated gates, which cause the transitions of the duplicated outputs, persist until the transitions propagate to the environment. The rationale behind the error tolerance of DD circuits that have the strong stability, is explained in the following theorem.

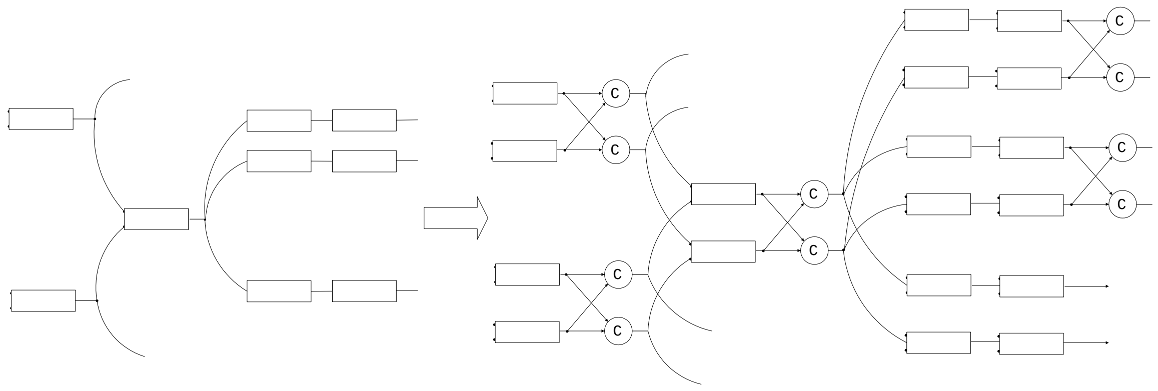


Figure 5.2: *Circuit and its Corresponding DD Circuit.*

Theorem 1 *If a QDI circuit is stable and deadlock-free, then its corresponding DD QDI circuit can tolerate a soft error: neither deadlock nor abnormal computations are caused by a soft error.*

Proof: In a DD circuit, duplicated variables are cross-coupled by C-elements whose outputs reflect the inputs when the values of all inputs match; an error corrupts only half of a DD circuit. As a result, the only erroneous computation, if incurred, is deadlock when an error causes inputs of cross-coupled C-elements (i.e., CI variables) to mismatch. Hence, in order to show the error tolerance of a DD circuit, it is enough to show that an error cannot cause deadlock: a mismatch between the inputs of cross-coupled C-elements, caused by an error, can be resolved eventually, or the mismatch does not stop expected assignments of CO variables by disabling effective PRs.

There are three kinds of variables in a DD circuit: a CI variable, a CO variable, and

Table 5.1: State Table of DD Gate

G_n^a	G_n^b	G_p^a	G_p^b	z'_a	z'_b	z_a	z_b	
D	D	D	D	0	0	0	0	(a)
		E	D	0	0	0	0	(b)
				1	0	0	0	(c)
		D	E	0	0	0	0	(d)
				0	1	0	0	(e)
		E	E	0	0	0	0	(f)
				1	0	0	0	(g)
				0	1	0	0	(h)
				1	1	0	0	(i)
				1	1	1	0	(j)
				1	1	0	1	(k)
				1	1	1	1	(l)
		D	E	1	1	1	1	(m)
		E	D	1	1	1	1	(n)

G_n^a	G_n^b	G_p^a	G_p^b	z'_a	z'_b	z_a	z_b	
D	D	D	D	1	1	1	1	(a’)
E	D			1	1	1	1	(b’)
				0	1	1	1	(c’)
D	E			1	1	1	1	(d’)
				1	0	1	1	(e’)
E	E			1	1	1	1	(f’)
				0	1	1	1	(g’)
				1	0	1	1	(h’)
				0	0	1	1	(i’)
				0	0	0	1	(j’)
				0	0	1	0	(k’)
				0	0	0	0	(l’)
D	E			0	0	0	0	(m’)
E	D			0	0	0	0	(n’)

* E stands for enabled, and D stands for disabled

an *ordinary* variable, which is an output of combinational gates without cross-coupled C-elements. In fact, corruption caused by an error on an ordinary variable is the same as a part of corruption caused by an error on a CO variable, which will be explained when we examine what an error on a CO variable causes. Therefore, in order to show the error tolerance, we shall demonstrate that an error on a cross-coupled variable does not cause deadlock. For this, first we enumerate all possible states of a DD gate in a stable and non-interfering DD circuit, as shown in Table 5.1, where 'E' stands for an enabled PR, and 'D' stands for a disabled PR of a DD gate. Because of the strong stability, only 28 possible states are allowed for a DD gate. Comparing the 14 states of the upper table to the 14 states of the lower table, we can see that the variables and the guards of the PRs are set/reset in an opposite sense. Therefore, it is enough to examine what happens if an error occurs in each of the 14 states from (a) to (n) in the upper table only.

Let us consider an error on a CI variable first. We can categorize the 14 states to three types, as follows.

1. Let us assume that an error occurs on z'_a in the state where G^a_p is `true` such as (b), (c), (f), (g), ..., (l), and (n). If z'_a is 0, then the error is equivalent to $z'_a\uparrow$, which is a firing of the effective PR $G^a_p \to z'_a\uparrow$. That is, the error is just a normal transition. On the other hand, if z'_a is 1, then the error is equivalent to $z'_a\downarrow$, which prevents $z'_a \land z'_b \to z_a\uparrow, z_b\uparrow$ from firing. Meanwhile the corrupted z'_a is restored, because G^a_p is persistent to be `true` to make $G^a_p \to z'_a\uparrow$ fire. Therefore, deadlock can be avoided.

2. Let us assume that an error occurs on z'_a in a state where G^a_p is `false` and (z'_a, z'_b) is $(0,0)$ or $(1,1)$, such as (a), (d), or (m). If (z'_a, z'_b) is $(0,0)$ or $(1,1)$, then (z_a, z_b) is $(0,0)$ or $(1,1)$ in the three states. If (z'_a, z'_b) is $(0,0)$, the error cannot enable $z'_a \land z'_b \to z_a\uparrow, z_b\uparrow$ to fire, and consequently (z_a, z_b) remains $(0,0)$. If (z'_a, z'_b) is $(1,1)$, the error cannot enable $\neg z'_a \land \neg z'_b \to z_a\downarrow, z_b\downarrow$ to fire, and consequently (z_a, z_b) remains $(1,1)$. In both cases, the error does not affect the values of z_a and z_b: the error is masked, and the system works as if no error occurs.

3. Let us consider an error on the state (e) where $(G^a_p, G^b_p : z'_a, z'_b, z_a, z_b) = (D, E : 0, 1, 0, 0)$. If an error occurs on z'_a, then (z'_a, z'_b) becomes $(1,1)$, and $z'_a \land z'_b \to z_a\uparrow, z_b\uparrow$ are enabled to fire. Although deadlock does not occur, the firings of $z_a\uparrow, z_b\uparrow$ caused by the error may be premature in the following sense. If the assignments of the inputs

such as $x_a\uparrow$ (or $x_a\downarrow$) has not been completed (i.e., G_p^a is still `false`), but the firings occur, then the environment assumes that the assignments are completed, and the state of a system is updated even before the assignments are completed. In other words, the assignments of some inputs can be missed owing to the error. But note that they do not affect the sequence of firings that the environment can observe, and do not stop the firings: the system works correctly.

As we see, an error on a CI variable is masked or corrected, so that the error does not cause deadlock in a DD circuit.

Let us consider an error occurring on a CO variable such as z_a. If it does not enable any PRs of subsequent gates such as g_p^a or g_n^a, an error on z_a cannot cause any erroneous behavior, and the corrupted z_a is restored in short order, because the values of CI variables become the same eventually, and cross-coupled C-elements of the corrupted CO variable corrects its corrupted output.

$$\begin{array}{lcl} z'_a \wedge z'_b & \rightarrow & z_a\uparrow, z_b\uparrow \\ \neg z'_a \wedge \neg z'_b & \rightarrow & z_a\downarrow, z_b\downarrow \end{array}$$

$$\begin{array}{lcl} g_p^a(\ldots, z_a, \ldots) & \rightarrow & w_a\uparrow \\ g_p^b(\ldots, z_b, \ldots) & \rightarrow & w_b\uparrow \\ g_n^a(\ldots, z_a, \ldots) & \rightarrow & w_a\downarrow \\ g_n^b(\ldots, z_b, \ldots) & \rightarrow & w_b\downarrow \end{array}$$

On the other hand, if an error on a CO variable enables PRs of subsequent gates to fire, then the outputs (i.e., ordinary variables) of the subsequent gates become corrupted, and the corrupted outputs may affect the outputs of next subsequent gates, and so on. However, since the propagation of the unexpected firings is blocked by DD gates, the corrupted region is confined between a DD gate and another DD gate, as shown in Figure 5.3. Meanwhile, the corrupted z_a will be restored, since CI variables are not corrupted; accordingly corrupted ordinary variables are also restored, since the outputs of combinational gates can be corrected if the inputs are corrected. Therefore, all corrupted variables in confined regions are corrected, and then blocked computations of the DD circuit can proceed. Likewise if an error on an ordinary variable occurs, corruption caused by the error is confined between DD gates, and the corruption is restored, because inputs of the corrupted combinational gate are still correct. Here, we can see why cross-coupled C-elements can be omitted for combinational gates: the output of a state-holding gate may not be restored even if the inputs of the state-holding gate are correct, but the output of a combinational gate is always restored if the inputs of the combinational gate are correct.

Note that we rely on an implicit assumption in order to show that a DD circuit tolerates an error. The assumption is that each loop of gates in a DD circuit, which is a set of gates whose inputs/outputs are outputs/inputs of gates in the set, has enough DD gates in order to prevent corruption of a CO variable (e.g., z_a) from propagating up to a corresponding CI variable (e.g., z'_a). If the assumption does not hold, then a CI variable can be corrupted because of an error on its corresponding CO variable: a set of cross-coupled variables are corrupted simultaneously, which prevents cross-coupled C-element from correcting the corrupted CO variable. As a result, deadlock occurs. In most practical circuits, our assumption is valid, and we can add more cross-coupled C-elements to a loop of gates to ensure the assumption, if necessary. ■

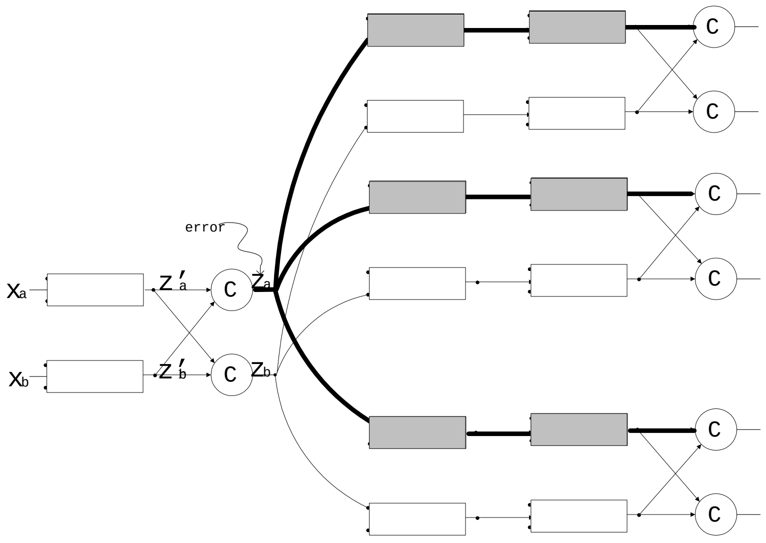


Figure 5.3: *Soft Error on CO Variable. The propagation of an error is blocked by the next DD gates.*

A DD circuit can tolerate not only a single error but also multiple errors, unless a set of cross-coupled variables is corrupted simultaneously. Even if multiple errors occur on a set of cross-coupled variables, and each error is corrected before the next error occurs, then the multiple errors on cross-coupled variables seem to be tolerated. However there is a problem that a CI variable (e.g., z'_a) in a state-holding state can remain corrupted for a long time after an error occurs on the variable; a DD circuit will fail if another error occurs on the other CI variable (e.g., z'_b) later.

In order to avoid the accumulation of errors, weak C-elements, as shown in Figure 5.4, can be used. They play the role of duplicated "keepers", which not only keep the voltage level of the state-holding nodes (z'_a, z'_b), but also can correct the nodes if one of them is corrupted, as follows. If a soft error occurs on z'_a in the state $s = (..., z'_a, z'_b, z_a, z_b, ...) = (..., 0, 0, 0, 0, ...)$ or $(..., 1, 1, 1, 1, ...)$, then the weak C-elements are enabled to restore z'_a. Hence the accumulated-error problem can be resolved by weak C-elements. In other possible states, the weak C-elements are functionally transparent in a DD circuit.

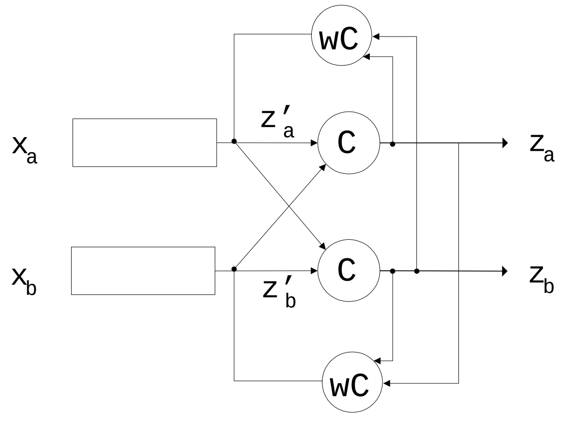


Figure 5.4: *DD Gate with weak C-elements. It can correct a corrupted CI variable in a state-holding state.*

Throughout the section, we have assumed that cross-coupled C-elements are added for every state-holding gate, but they are not always necessary. In fact, the necessity depends on how the environment sets/resets the inputs of a state-holding gate as a response to the change of an output of the gate. For example, let us assume that the environment of a tree of C-elements reacts to the output assignment $z\uparrow$ by setting all inputs to be `false`, and to the output assignment $z\downarrow$ by setting all inputs to be `true`. Then the tree of C-elements can be converted to a DD tree of C-elements using one pair of double-checking C-elements, as shown in Figure 5.5. In the design, an error occurs on the output of an internal C-element, say, y_a, which remains corrupted for the time being, but the corruption does not cause deadlock. (The effect of the error is similar to that of an error on a CI variable in the state (e), which we examined in the proof.) The environment, which only can see the intact primary outputs, causes all primary inputs to be either `false` or `true` eventually, and then the corrupted variable y_a is corrected.

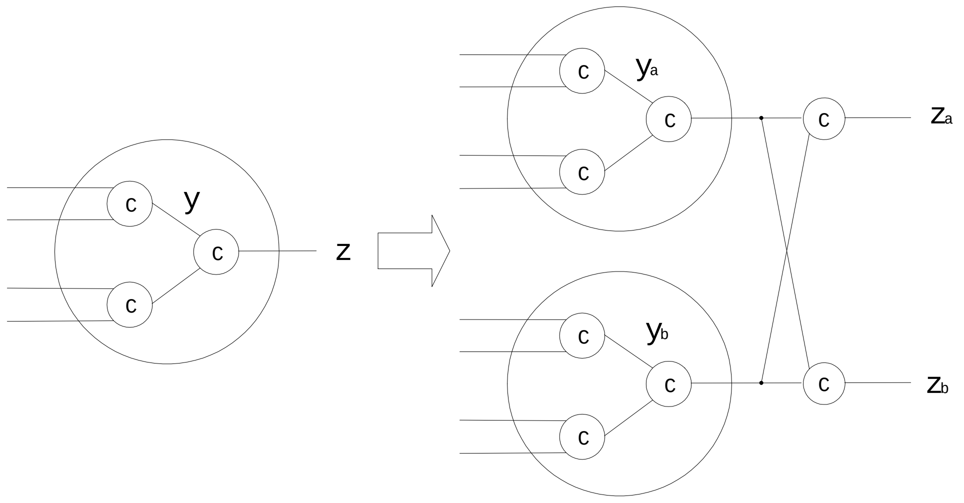


Figure 5.5: *Tree of C-element and its corresponding DD Tree. The DD tree of C-elements requires cross-coupled C-elements only for the primary outputs.*

5.1.2 Comparing Reliability of QDI Circuits and DD QDI Circuits

The *reliability* of a system is defined as the probability that a system will perform its intended function during a specified period of time under stated conditions. Here we analyze the reliability of a DD circuit, comparing it with that of a normal QDI circuit.

For simplicity, let us assume that an error that causes a circuit to fail occurs on each node with probability P_e, in independent and identically-distributed fashion, during a given period of time. Then the reliability of a system with N nodes is expressed as the following probability

$$P_{\text{Reliability}} = P[\text{N nodes are good}] = P_{good}^{N} = (1 - P_e)^N \approx (1 - NP_e) \ \ (\text{if } NP_e \ll 1).$$

Let us consider the reliability of a DD circuit. If a given QDI circuit has N nodes and cross-coupled C-elements are added to each and every gate, then the number of the nodes in a corresponding DD circuit will be $4N$, because of the duplication and cross-coupled C-elements, and there are N sets of cross-coupled variables. If multiple errors occur, at most one error should be located in one of N different sets of cross-coupled variables. If not, the DD circuit cannot tolerate the multiple errors. If n errors occur, only $4^n\binom{N}{n}$ different distributions of n errors can be tolerated: each error is assigned to one set of cross-coupled variables, which is $\binom{N}{n}$, and each set has four possible sites of an error, 4^n. Accordingly we have the following reliability for a DD circuit:

$$\begin{aligned}
P_{\text{Reliability}} &= P[\text{4N nodes are good}] \\
&\quad +4^1 \begin{pmatrix} N \\ 1 \end{pmatrix} P[4N-1 \text{ nodes are good}]P[\text{one error}] \\
&\quad +4^2 \begin{pmatrix} N \\ 2 \end{pmatrix} P[4N-2 \text{ nodes are good}]P[\text{two errors}] \\
&\quad +\ldots \\
&\quad +4^N \begin{pmatrix} N \\ N \end{pmatrix} P[4N-N \text{ nodes are good}]P[N \text{ errors}] \\
&= P_{good}^{4N} P_e^0 \\
&\quad +4^1 \begin{pmatrix} N \\ 1 \end{pmatrix} P_{good}^{4N-1} P_e^1 \\
&\quad +4^2 \begin{pmatrix} N \\ 2 \end{pmatrix} P_{good}^{4N-2} P_e^2 \\
&\quad +\ldots \\
&\quad +4^N \begin{pmatrix} N \\ N \end{pmatrix} P_{good}^{4N-N} P_e^N \\
&= P_{good}^{3N}(1+3P_e)^N \\
&= (1-P_e)^{3N}(1+3P_e)^N \\
&= (1-6P_e^2+8P_e^3-3P_e^4)^N \\
&\approx 1-6NP_e^2 \quad (\text{if } NP_e^2 \ll 1)
\end{aligned}$$

Compared with the reliability of a given QDI circuit, the first-order term of P_e does not appear in the reliability of a DD circuit. The difference between the reliability of a normal circuit and its DD circuit is obvious when $NP_e \approx 1$. For example if $N = 10^6$, and $P_e = 10^{-6}$, then $P_{\text{Reliability}}(\text{Normal Circuit}) = 0.37$, but $P_{\text{Reliability}}(\text{DD Circuit}) = 0.9999$.

Although we have considered the occurrence of errors to be independent, the occurrence of errors on adjacent nodes is likely to be more probable than that of an error on non-adjacent nodes. For example, given the occurrence of an error on z_a, the probability of an error on the counterpart node z_b may increase by a constant factor of c, compared with the probability of the occurrence of an error on z_b without that of an error on z_a. It can be written, as follows:

$$P(\text{error on } z_a | \text{error on } z_b) = c \times P(\text{error on } z_a | \text{no error on } z_b)$$

$$P(\text{error on } z_b | \text{error on } z_a) = c \times P(\text{error on } z_b | \text{no error on } z_a)$$

The notation $P(A|B)$ means the conditional probability of A given B. Then we have the following:

$$\frac{P(\text{error on both } z_a \text{ and } z_b)}{P(\text{error on } z_b)} = c \times \frac{P(\text{error on } z_a \text{ and no error on } z_b)}{P(\text{no error on } z_b)}$$

$$\frac{P(\text{error on both } z_a \text{ and } z_b)}{P(\text{error on } z_b)} = c \times \frac{P(\text{error on } z_b \text{ and no error on } z_a)}{P(\text{no error on } z_a)}$$

$$P(\text{error on } z_a) = P(\text{error on both } z_a \text{ and } z_b) + P(\text{error on } z_a \text{ and no error on } z_b) = P_e$$

$$P(\text{error on } z_b) = P(\text{error on both } z_a \text{ and } z_b) + P(\text{error on } z_b \text{ and no error on } z_a) = P_e$$

Solving the equations, we obtain the following:

$$\begin{aligned}
P(\text{error on neither } z_a \text{ nor } z_b) &= \frac{c(1-P_e)(1+(c-2)P_e)}{1+(c-1)P_e} \\
P(\text{error on } z_a \text{ and no error on } z_b) &= \frac{cP_e(1-P_e)}{1+(c-1)P_e} \\
P(\text{error on } z_b \text{ and no error on } z_a) &= \frac{cP_e(1-P_e)}{1+(c-1)P_e} \\
P(\text{error on both } z_a \text{ and } z_b) &= \frac{cP_e^2}{1+(c-1)P_e}
\end{aligned}$$

Here, we let $P(\text{error on a node}) = P_e$. Considering the effect of proximity on the error probability, we obtain the following reliability:

$$P_{Reliablilty}(DD) \approx 1 - c \times 6NP_e^2$$

Hence, if the correlation between errors on adjacent nodes is weak enough (i.e., $c \ll \frac{1}{P_e}$), then we still get better reliability by applying the DD scheme to a QDI circuit.

Although the DD scheme improves the reliability of a system, it has the area overhead of extra circuits, and a performance overhead, too. The size of transistors has decreased by a factor of 0.7 in each technology node so that the area of the same system becomes half, and the cycle time of the system can be increased by a factor of almost $\frac{1}{0.7} \approx 1.4$ [3]. (The technology node signifies the feature size of a circuit.) Meanwhile the soft-error rate is increasing by a factor of about seven in each technology node. If so, an old technology whose error rate is lower and whose performance is worse than that of a new technology may be considered an alternative to DD circuits for a reliable system. But it turns out that the DD circuit in a newer technology will be better than the normal circuit in an older technology.

For example, let us assume that the reliability of a normal circuit in an older technology and the reliability of its DD circuit in a newer technology are the same. Then,

$$(1 - p_e)^N \quad = \quad (1 - P_e)^{3N}(1 + P_e)^N$$

is satisfied, where p_e is the probability of the occurrence of an error in the older technology, P_e is the probability in the newer technology, and if s signifies the gap between the two technology nodes,

$$p_e \times 7^s \quad = \quad P_e$$

is satisfied, since the soft-error rate is increasing by a factor of about seven in each technology node. If we use the numbers $P_e = 10^{-6}$ again, then we obtain $s \approx 7$, which means that the feature size of the normal circuit is larger by a factor of $1.4^7 \approx 10$. In the newer technology, a DD circuit, which reduces the throughput by 50% and enlarges a protected circuit by a factor of three at worst, is faster and smaller than the normal circuit in the older technology. In fact, if $P_e \ll 1$, DD circuits can achieve superior reliability and performance figures than the corresponding normal circuit in an old technology.

5.1.3 Simulation Results of Duplicated Double-checking Buffer

Let us apply the DD scheme to a one-bit PCHB whose CMOS-implementable PRS is as follows:

$$\begin{array}{lcl}
en \wedge R.e \wedge L.0 & \rightarrow & \_r.0\downarrow \\
\neg en \wedge \neg R.e & \rightarrow & \_r.0\uparrow \\
en \wedge R.e \wedge L.1 & \rightarrow & \_r.1\downarrow \\
\neg en \wedge \neg R.e & \rightarrow & \_r.1\uparrow \\
\_r.0 & \rightarrow & R.0\downarrow \\
\neg \_r.0 & \rightarrow & R.0\uparrow \\
\_r.1 & \rightarrow & R.1\downarrow \\
\neg \_r.1 & \rightarrow & R.1\uparrow
\end{array}$$

$$\begin{array}{lcl}
L.0 \vee L.1 & \rightarrow & \_lv\downarrow \\
\neg L.0 \wedge \neg L.1 & \rightarrow & \_lv\uparrow \\
\_lv & \rightarrow & lv\downarrow \\
\neg \_lv & \rightarrow & lv\uparrow \\
\_r.0 \wedge \_r.1 & \rightarrow & rv\downarrow \\
\neg \_r.0 \vee \neg \_r.1 & \rightarrow & rv\uparrow
\end{array}$$

$$\begin{array}{lcl}
lv \wedge rv & \rightarrow & L.e\downarrow \\
\neg lv \wedge \neg rv & \rightarrow & L.e\uparrow \\
L.e & \rightarrow & \_en\downarrow \\
\neg L.e & \rightarrow & \_en\uparrow \\
\_en & \rightarrow & en\downarrow \\
\neg \_en & \rightarrow & en\uparrow
\end{array}$$

Its circuit diagram is shown in Figure 5.6. Variables $L.0$, $L.1$ encode a one-of-two code and the variables with an acknowledgment variable $L.e$ implement the one-bit input data channel L. Likewise variables $R.0$, $R.1$, and $R.e$ implement the one-bit output data channel R.

The PRS of the corresponding DD PCHB is as follows:

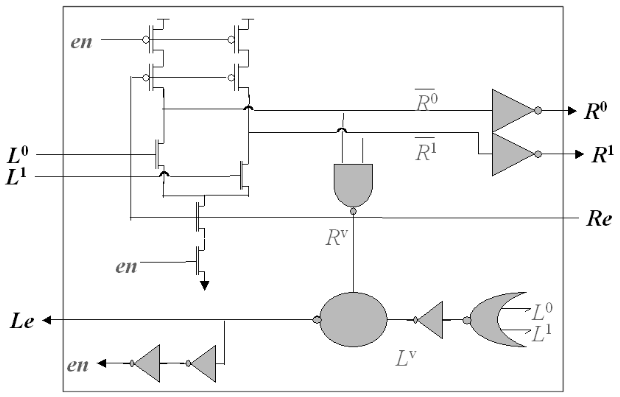


Figure 5.6: *Precharged Half Buffer.*

$$en_a \wedge R.e_a \wedge L.0_a \rightarrow \_r.0_a\!\downarrow$$
$$\neg en_a \wedge \neg R.e_a \rightarrow \_r.0_a\!\uparrow$$
$$en_a \wedge R.e_a \wedge L.1_a \rightarrow \_r.1_a\!\downarrow$$
$$\neg en_a \wedge \neg R.e_a \rightarrow \_r.1_a\!\uparrow$$

$$en_b \wedge R.e_b \wedge L.0_b \rightarrow \_r.0_b\!\downarrow$$
$$\neg en_b \wedge \neg R.e_b \rightarrow \_r.0_b\!\uparrow$$
$$en_b \wedge R.e_b \wedge L.1_b \rightarrow \_r.1_b\!\downarrow$$
$$\neg en_b \wedge \neg R.e_b \rightarrow \_r.1_b\!\uparrow$$

$$\_r.0_a \wedge \_r.0_b \rightarrow R.0_a\!\downarrow$$
$$\neg\_r.0_a \wedge \neg\_r.0_b \rightarrow R.0_a\!\uparrow$$
$$\_r.1_a \wedge \_r.1_b \rightarrow R.1_a\!\downarrow$$
$$\neg\_r.1_a \wedge \neg\_r.1_b \rightarrow R.1_a\!\uparrow$$

$$\_r.0_a \wedge \_r.0_b \rightarrow R.0_b\!\downarrow$$
$$\neg\_r.0_a \wedge \neg\_r.0_b \rightarrow R.0_b\!\uparrow$$
$$\_r.1_a \wedge \_r.1_b \rightarrow R.1_b\!\downarrow$$
$$\neg\_r.1_a \wedge \neg\_r.1_b \rightarrow R.1_b\!\uparrow$$

$$L.0_a \vee L.1_a \rightarrow \_lv_a\!\downarrow$$
$$\neg L.0_a \wedge \neg L.1_a \rightarrow \_lv_a\!\uparrow$$

$$L.0_b \vee L.1_b \rightarrow \_lv_b\!\downarrow$$
$$\neg L.0_b \wedge \neg L.1_b \rightarrow \_lv_b\!\uparrow$$

$$R.0_a \vee R.1_a \rightarrow \_rv_a\downarrow$$
$$\neg R.0_a \wedge \neg R.1_a \rightarrow \_rv_a\uparrow$$

$$R.0_b \vee R.1_b \rightarrow \_rv_b\downarrow$$
$$\neg R.0_b \wedge \neg R.1_b \rightarrow \_rv_b\uparrow$$

$$\_lv_a \wedge \_rv_a \rightarrow \_le_a\downarrow$$
$$\neg\_lv_a \wedge \neg\_rv_a \rightarrow \_le_a\uparrow$$

$$\_lv_b \wedge \_rv_b \rightarrow \_le_b\downarrow$$
$$\neg\_lv_b \wedge \neg\_rv_b \rightarrow \_le_b\uparrow$$

$$\_le_a \wedge \_le_b \rightarrow L.e_a\uparrow$$
$$\neg\_le_a \wedge \neg\_le_b \rightarrow L.e_a\downarrow$$
$$\_le_a \wedge \_le_b \rightarrow L.e_b\uparrow$$
$$\neg\_le_a \wedge \neg\_le_b \rightarrow L.e_b\downarrow$$

$$L.e_a \rightarrow \_en_a\downarrow$$
$$\neg L.e_a \rightarrow \_en_a\uparrow$$
$$\_en_a \rightarrow en_a\downarrow$$
$$\neg\_en_a \rightarrow en_a\uparrow$$

$$L.e_b \rightarrow \_en_b\downarrow$$
$$\neg L.e_b \rightarrow \_en_b\uparrow$$
$$\_en_b \rightarrow en_b\downarrow$$
$$\neg\_en_b \rightarrow en_b\uparrow$$

Now variables $L.0_a$, $L.0_b$, $L.1_a$, $L.1_b$ encode duplicated one-of-two codes, and the variables with two acknowledgment variables, $L.e_a$ and $L.e_b$, implement the input data channel L. ($L.0$ becomes $L.0_a$, $L.0_b$; $L.1$ becomes $L.1_a$, $L.1_b$; $L.e$ becomes $L.e_a$, $L.e_b$.) Likewise variables $R.0_a$, $R.0_b$, $R.1_a$, $R.1_b$, $R.e_a$, and $R.e_b$ implement the output data channel R. The circuit diagram of the one-bit DD PCHB is shown in Figure 5.7, where a circle with the letter 'C' stands for a cross-coupled C-element.

Table 5.2 summarizes the performance figures of the conventional PCHB, and the DD PCHB. The number of nodes in the DD PCHB is more than twice that of nodes in the

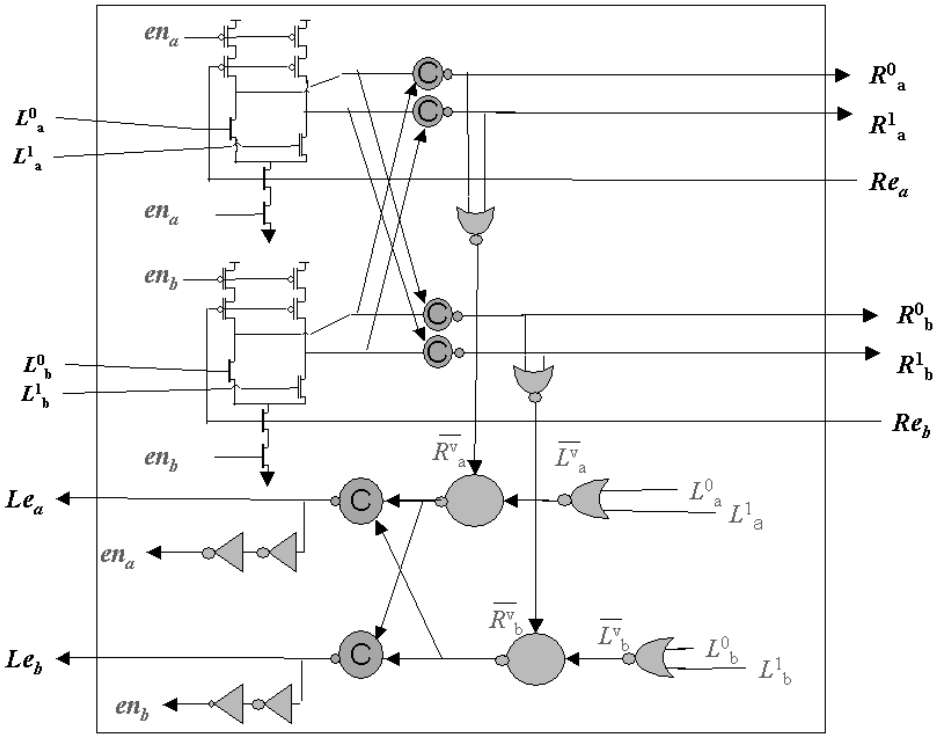


Figure 5.7: *DD PCHB.*

Table 5.2: Performance Figures of PCHB and DD PCHB

	e1of4 PCHB	e1of4 DD PCHB
# of nodes	25	52
Repetition Rate	610 MHz	360 MHz
Transitions in Cycle	14	18
Area(λ^2)	36736	79192
Energy per Cycle(pJ)	1.16	2.95

conventional PCHB; the DD PCHB is enlarged by a factor of 2.2 in area, because cross-coupled C-elements are introduced in addition to duplicating all of the existing gates: three pairs of cross-coupled C-elements are added in the given example. (In fact, at least three pairs of C-elements are necessary for making the conventional PCHB soft-error tolerant, and we cannot design an error tolerant PCHB with less than three pairs of cross-coupled C-elements.) The throughput is reduced by 40%, because the DD PCHB takes more transitions in a cycle due to the added cross-coupled C-elements. That is, in the conventional PCHB, the completion checker of the channel R (i.e., a NAND gate in Figure 5.6) takes its inputs ($\overline{R.0}$ and $\overline{R.1}$) from the pulldown stack; in a DD PCHB, a completion checker of the channel R (i.e., a NOR gate in Figure 5.7) takes its inputs ($R.0_a$ and $R.1_a$) from the cross-coupled

C-elements. Therefore it takes more transitions to generate acknowledgment signals: the PCHB takes 14 transitions per cycle, but the DD PCHB takes 18 transitions. In addition to that, some of the gates in the DD PCHB have more load, because the outputs are shared between cross-coupled C-elements, and the wiring tends to be longer, which also affects the latency.

One way of showing the error tolerance of the DD circuit is to simulate the circuit with soft errors. Doing this, we obtain the following results, which are depicted in Figure 5.8. Charges (i.e., a soft error) are injected to $L.e_a$ at 10 ns. Since the value of the node $L.e_a$ is flipped, the environment considers the buffer 'not ready' for a new input so that $L.0_a\uparrow$ and $L.0_b\uparrow$ do not fire until the corrupted value of $L.e_a$ is restored. That is, after the node $L.e_a$ has the same value as the node $L.e_b$ at 11 ns, the whole system works correctly, except that the computation was delayed momentarily.

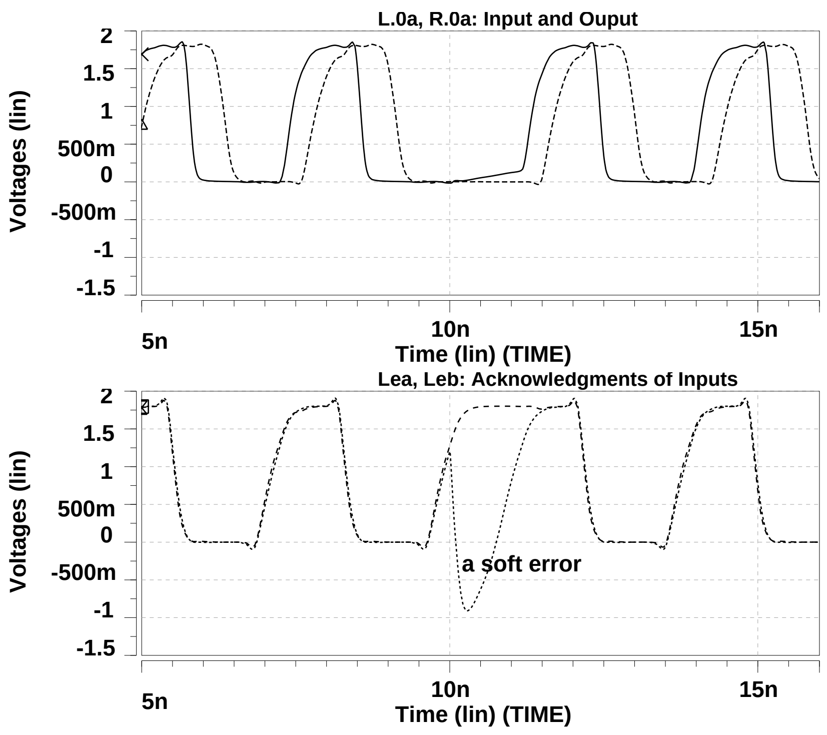


Figure 5.8: *Waveforms of Inputs-channel Variable $L0_a$ and Output-channel Variable $R0_a$, and Input-channel Acknowledgment Variable Le_a of DD One-bit Buffer. A soft error on Le_a at 10 ns merely causes delay on an input communication on the input channel, which does not affect the correction of computation in a QDI circuit.*

5.1.4 Summary

When a soft error occurs, a QDI system may compute incorrectly, or deadlock may occur. The DD scheme provides a simple way of making QDI circuits tolerant to soft errors: cross-coupled C-elements prevent a corrupted result caused by a soft error from propagating to subsequent gates, and the stability property ensures that the inputs of a DD gate persist until the duplicated outputs are properly generated, which provides the redundancy necessary to restore the corrupted node. Compared with a normal QDI circuit, the size of a DD circuit is enlarged by a factor of between two and three, and the throughput is reduced by between 40% and 50%, owing to the fact that the number of transitions in a cycle increases, and the loads on some gates increase.

5.2 Arbiter with Mutual Excluders

While most of computations in an asynchronous system are deterministic, sometimes the system requires non-determinism in order to make a choice among several possibilities. For example, if two or more components request a resource simultaneously in an asynchronous system, then an *arbiter*, which is a circuit to choose one of the requests non-deterministically, is necessary.

The DD scheme cannot be applied for a soft-error tolerant arbiter due to its non-deterministic property: two identical arbiters may produce different outputs even when the same inputs are provided to the arbiters. Therefore, we need another scheme to make an arbiter soft-error tolerant.

In the following subsections, we shall review a basic arbiter. Then a soft-error tolerant arbiter is proposed, and the correctness of the design is verified. Finally, the results of SPICE simulations of the soft-error tolerant arbiter with an error are shown.

5.2.1 Overview

Let us consider a PRS of a bare arbiter whose inputs are two variables x and y:

$$\begin{aligned} x \wedge \neg t &\rightarrow s\uparrow \\ y \wedge \neg s &\rightarrow t\uparrow \\ \neg x \vee t &\rightarrow s\downarrow \\ \neg y \vee s &\rightarrow t\downarrow \end{aligned}$$

The arbiter provides a non-deterministic choice when both $x\uparrow$ and $y\uparrow$ fire. While $s\uparrow$ or $t\uparrow$ fires to indicate which firing is chosen, the arbiter is intended to guarantee that $\neg s \vee \neg t$ always holds (i.e., the *mutual exclusion* of s and t). Without the mutual exclusion, the environment cannot discern which request has been chosen to be handled. However if the two events $x\uparrow$ and $y\uparrow$ happen simultaneously, the bare arbiter may fall into the *metastable state*, in which mutual exclusion may be violated. In the PRS model, the arbiter in the metastable state, $x \wedge y \wedge (s = t)$, may produce the unbounded sequence of firings: $s\uparrow, t\uparrow; s\downarrow, t\downarrow; s\uparrow, t\uparrow; \cdots$. Physically, the variables s and t take on an intermediate voltage for an unbounded time in the metastable state. Although the metastable state normally does not persist for long, the intermediate voltage of the nodes during the metastable state violates mutual exclusion.

To solve this problem, a *filter* is introduced, which blocks the appearance of the intermediate voltage on the output of the arbiter during the metastable state. The PRS of the filter is as follows:

$$s \wedge \neg t \rightarrow u\uparrow$$
$$\neg s \vee t \rightarrow u\downarrow$$

$$\neg s \wedge t \rightarrow v\uparrow$$
$$s \vee \neg t \rightarrow v\downarrow$$

The filter, which is NOR gates with an inverted input, is combined with the bare arbiter to yield a basic arbiter, as shown in Figure 5.9, where s and t are replaced with $\overline{s}$ and $\overline{t}$ for CMOS implementability, and pass gates are used. The outputs u and v of the basic arbiter do not change until the voltages on $\overline{s}$ and $\overline{t}$ are separated by at least a p-transistor threshold voltage. That is, u and v will be assigned after the internal nodes $\overline{s}$ and $\overline{t}$ leave the metastable state. At the HSE level, the basic arbiter can be described as follows:

```
*[[$x$  $\longrightarrow$  $u\uparrow$; [$\neg x$]; $u\downarrow$
  |$y$  $\longrightarrow$  $v\uparrow$; [$\neg y$]; $v\downarrow$
]]
```

In the HSE, '|' represents a non-deterministic selection between two enabled commands.

The environment of a basic arbiter can be

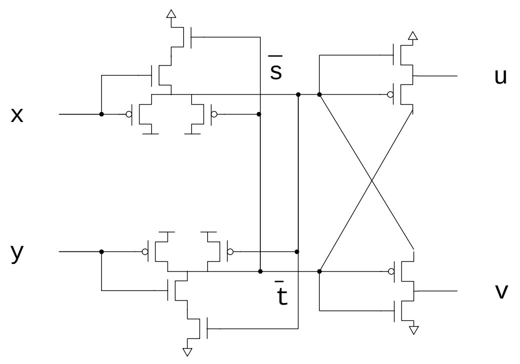


Figure 5.9: *Basic Arbiter. When all inputs (x,y) are assigned to be 1 simultaneously, it provides the outputs of either (1,0) or (0,1) non-deterministically.*

$$
\begin{aligned}
u &\rightarrow x.ack\uparrow\\
\neg u &\rightarrow x.ack\downarrow\\
\\
\neg x.ack &\rightarrow x\uparrow\\
x.ack &\rightarrow x\downarrow\\
\\
v &\rightarrow y.ack\uparrow\\
\neg v &\rightarrow y.ack\downarrow\\
\\
\neg y.ack &\rightarrow y\uparrow\\
y.ack &\rightarrow y\downarrow,
\end{aligned}
$$

where $x.ack$ and $y.ack$ are the acknowledgment variables of x and y. The event $x.ack\uparrow$ (or $y.ack\uparrow$) can be interpreted as a module's acquiring a resource, and $x.ack\downarrow$ (or $y.ack\downarrow$) as the module's releasing the resource. Accordingly, mutual exclusion of $x.ack$ and $y.ack$ should hold, but there is a transient moment when the mutual exclusion does not hold. While both acknowledgment variables are `false` initially, both $x\uparrow$ and $y\uparrow$ fire. The basic arbiter chooses $x\uparrow$, and then $u\uparrow$ fires. The environment acknowledges the choice by the firing of $x.ack\uparrow$. Consequently $x\downarrow$ fires. Since x becomes `false`, and y is `true` in the current state, $v\uparrow$ fires. If the firing of $u\downarrow$ is slow, and u has not yet been reset, then (u, v) becomes $(1, 1)$; $(x.ack, y.ack)$ also becomes $(1, 1)$. Hence a mutually exclusive resource is given to two modules simultaneously.

The problem can be avoided by the strengthened environment:

...

$u \wedge \neg y.ack \;\rightarrow\; x.ack\uparrow$

...

$v \wedge \neg x.ack \;\rightarrow\; y.ack\uparrow$

...

When a choice ($u\uparrow$) is being acknowledged (i.e., $x.ack$ is `true`) in the environment, the other choice ($v\uparrow$) cannot be acknowledged (i.e., $y.ack\uparrow$ cannot fire). Therefore, the mutual exclusion of the acknowledgment variables is guaranteed; the environment can discern which choice is made first by the basic arbiter. As a result, a basic arbiter processes resource requests in a mutually exclusive way.

5.2.2 Designing Error Tolerant Arbiter for DD Circuits

In this section, we shall specify a *duplicated soft-error tolerant (DSET)* arbiter, which is suitable for DD circuits.

A soft-error tolerant arbiter that is used with DD circuits behaves like a basic arbiter, but the inputs (x, y) and the outputs (u, v) are duplicated. That is, a *duplicated soft-error tolerant* (DSET) *arbiter* accepts duplicated inputs (x_a, x_b, y_a, y_b) and provides duplicated outputs (u_a, u_b, v_a, v_b), as shown in Figure 5.10. If $(x_a \wedge x_b)$ is `true`, then $\{u_a\uparrow, u_b\uparrow\}$ will fire, and if $(y_a \wedge y_b)$ is `true`, then $\{v_a\uparrow, v_b\uparrow\}$ will. If all of the inputs become `true` simultaneously, then a non-deterministic choice will be made between $\{u_a\uparrow, u_b\uparrow\}$ and $\{v_a\uparrow, v_b\uparrow\}$.

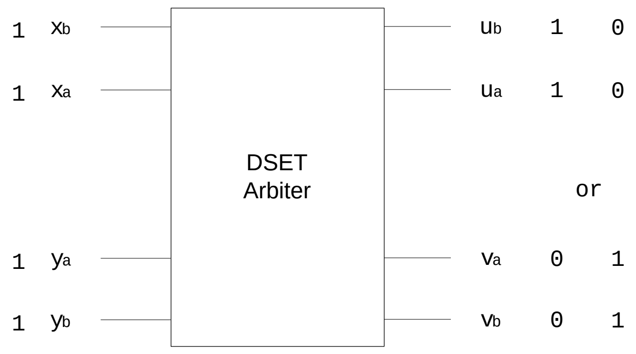


Figure 5.10: *Duplicated Soft-error Tolerant Arbiter. When all inputs are assigned to be 1 simultaneously, it provides the outputs of either (1,1,0,0) or (0,0,1,1) non-deterministically.*

The environment of a DSET arbiter, which is a DD version of the environment shown in the previous section, is as follows:

$$\begin{array}{lcl} u_a \wedge u_b \wedge \neg y.ack_a \wedge \neg y.ack_b & \rightarrow & x.ack_a\uparrow, x.ack_b\uparrow \\ \neg u_a \wedge \neg u_b & \rightarrow & x.ack_a\downarrow, x.ack_b\downarrow \end{array}$$

$$\begin{array}{lcl} \neg x.ack_a \wedge \neg x.ack_b & \rightarrow & x_a\uparrow, x_b\uparrow \\ x.ack_a \wedge x.ack_b & \rightarrow & x_a\downarrow, x_b\downarrow \end{array}$$

$$\begin{array}{lcl} v_a \wedge v_b \wedge \neg x.ack_a \wedge \neg x.ack_b & \rightarrow & y.ack_a\uparrow, y.ack_b\uparrow \\ \neg v_a \wedge \neg v_b & \rightarrow & y.ack_a\downarrow, y.ack_b\downarrow \end{array}$$

$$\begin{array}{lcl} \neg y.ack_a \wedge \neg y.ack_b & \rightarrow & y_a\uparrow, y_b\uparrow \\ y.ack_a \wedge y.ack_b & \rightarrow & y_a\downarrow, y_b\downarrow \end{array}$$

(For brevity, two PRs $G \rightarrow x\uparrow$ and $G \rightarrow y\uparrow$ are written as $G \rightarrow x\uparrow, y\uparrow$.) The PRs of the DD environment are not enabled until the values of a pair of duplicated variables are the same. Accordingly, if only one of the primary outputs of a DSET arbiter is corrupted (e.g., $(1,1,0,1)$, $(1,1,1,0)$, $(0,1,1,1)$, and $(1,0,1,1)$), it is still clear to the DD environment which module wins the arbitration.

As a basic arbiter guarantees the mutual exclusion between the outputs, a DSET arbiter guarantees the mutual exclusion of the duplicated outputs. Strictly speaking, a DSET arbiter with the DD environment guarantees the mutual exclusion of the duplicated acknowledgment variables, as explained in the previous section. Even if an error occurs on a DSET arbiter, it is critical to ensure that $(x.ack_a, x.ack_b, y.ack_a, y.ack_b)$ never becomes $(1,1,1,1)$: two modules never acquire one resource simultaneously. From the construction of the environment, we can see that the mutual exclusion can be violated only if (u_a, u_b, v_a, v_b) becomes $(1,1,1,1)$ while $(x.ack_a, x.ack_b, y.ack_a, y.ack_b)$ is $(0,0,0,0)$. Therefore, a DSET arbiter should ensure the mutual exclusion of the primary outputs when $(x.ack_a, x.ack_b, y.ack_a, y.ack_b)$ is $(0,0,0,0)$ even if an error occurs.

We show how to implement a DSET arbiter step by step from a first failed design to a successful design. Let us apply the DD scheme to a basic arbiter, as shown in Figure 5.11. If all inputs x_a, x_b, y_a, and y_b of the circuit become `true` simultaneously, one arbiter, whose inputs are x_a and y_a, may execute $u'_a\uparrow$, and the other arbiter, whose inputs are x_b and y_b, may execute $v'_b\uparrow$. The discrepancy prevents primary outputs from firing, so that the environment cannot observe transitions of any outputs: deadlock occurs.

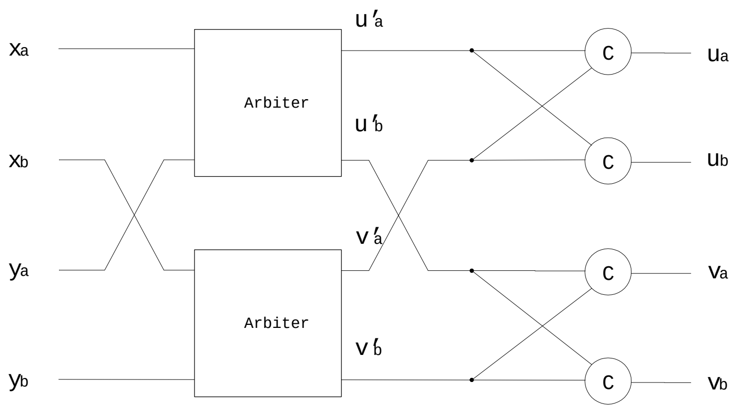


Figure 5.11: *DD Arbiter (does not work).*

Instead of using two arbiters, we shall use one basic arbiter to make a non-deterministic choice between inputs, and then duplicate the choice. The corresponding HSE is as follows:

```
*[[$x_a$ $\longrightarrow$ [$x_b$]; $u_a\uparrow, u_b\uparrow$; [$\neg x_a \wedge \neg x_b$]; $u_a\downarrow, u_b\downarrow$
  |$y_a$ $\longrightarrow$ [$y_b$]; $v_a\uparrow, v_b\uparrow$; [$\neg y_a \wedge \neg y_b$]; $v_a\downarrow, v_b\downarrow$
]]
```

The corresponding PRS is

$$\begin{array}{lcl} u \wedge x_b & \rightarrow & u_a\uparrow \\ u \wedge x_b & \rightarrow & u_b\uparrow \\ \neg u \wedge \neg x_b & \rightarrow & u_a\downarrow \\ \neg u \wedge \neg x_b & \rightarrow & u_b\downarrow \end{array}$$

$$\begin{array}{lcl} v \wedge y_b & \rightarrow & v_a\uparrow \\ v \wedge y_b & \rightarrow & v_b\uparrow \\ \neg v \wedge \neg y_b & \rightarrow & v_a\downarrow \\ \neg v \wedge \neg y_b & \rightarrow & v_b\downarrow \end{array}$$

That is, the outputs of an arbiter are copied with cross-coupled C-elements, as shown in Figure 5.12. Although it works as an arbiter for duplicated inputs and outputs, the mutual exclusion can be violated if an error occurs. For example, after the four inputs are assigned to be `true` simultaneously (i.e., $(x_a, x_b, y_a, y_b) = (1,1,1,1)$), the internal arbiter may choose $u\uparrow$ to fire, and, if so, $\{u_a\uparrow, u_b\uparrow\}$ will fire. However, if an error on $\overline{s}$ occurs at that moment, the outputs of the internal arbiter can be changed from $(u, v) = (1, 0)$ to $(0, 1)$. (An error in the internal basic arbiter can put the arbiter into metastable state, and then the metastable state can be resolved to a different stable state such as $(0,1)$ from the initial stable state such as $(1,0)$.) Now $\{v_a\uparrow, v_b\uparrow\}$ is firing: (u_a, u_b, v_a, v_b) becomes $(1,1,1,1)$. It can cause $(x.ack_a, x.ack_b, y.ack_a, y.ack_b)$ of the environment to be $(1,1,1,1)$.

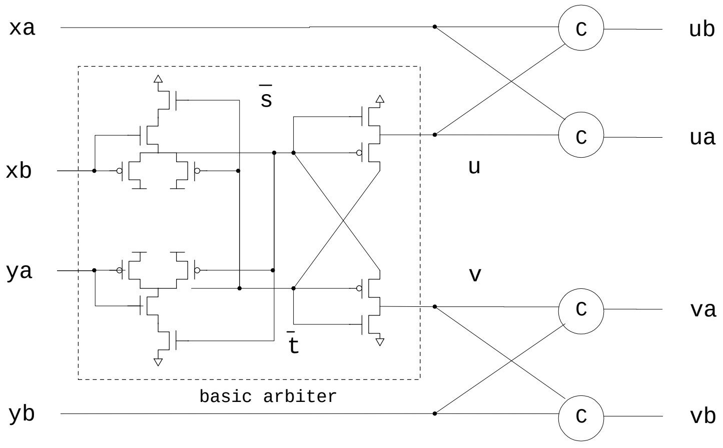


Figure 5.12: *Arbiter with Double-checking C-elements (does not work).*

The problem of the design is that the erroneous change of the outputs of the internal arbiter causes an unwanted resource grant even if the resource is still used. To address the problem, a DSET arbiter requires a feature that once a mutually exclusive resource is granted to one module, another resource grant should be prevented even if the change of the outputs of an internal arbiter occurs accidentally. That is, once $\{u_a\uparrow, u_b\uparrow\}$ fire, the firing of $\{v_a\uparrow, v_b\uparrow\}$ is not allowed until $\{x.ack_a\downarrow,\ x.ack_b\downarrow\}$, and then $\{u_a\downarrow, u_b\downarrow\}$ fire. Likewise once $\{v_a\uparrow, v_b\uparrow\}$ fire, the firing of $\{u_a\uparrow, u_b\uparrow\}$ is not allowed. For this, the cross-coupled C-elements are modified as follows:

$$\begin{array}{lcl}
\ldots & & \\
u \wedge x_b \wedge (\neg v_a \vee \neg v_b) & \rightarrow & u_a\uparrow \\
u \wedge x_b & \rightarrow & u_b\uparrow \\
\ldots & & \\
v \wedge y_b \wedge (\neg u_a \vee \neg u_b) & \rightarrow & v_a\uparrow \\
v \wedge y_b & \rightarrow & v_b\uparrow \\
\ldots & &
\end{array}$$

This modification seems to ensure the mutual exclusion, but still it is vulnerable to a specific error: if an error causes (u, v) to be $(1,1)$, and (u_a, u_b, v_a, v_b) is $(0,0,0,0)$ initially, then

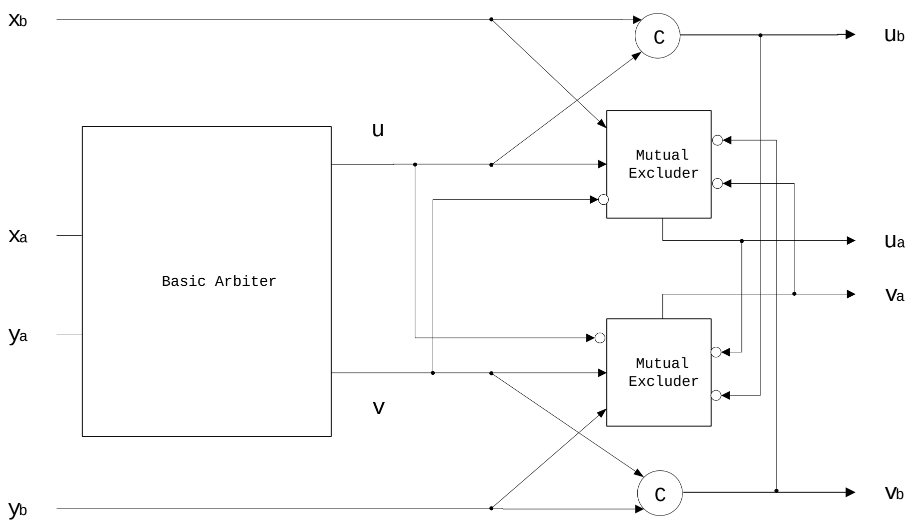


Figure 5.13: *Block Diagram of Soft-error Tolerant Arbiter.*

all PRs of the primary outputs are enabled to fire simultaneously; physically all primary outputs take intermediate voltages.

Here is the design which ensures the mutual exclusion in the presence of an error by checking whether the outputs (u, v) of the internal arbiter are valid (i.e., $(1, 0)$ or $(0, 1)$). Therefore, the terms of $(u \wedge \neg v)$ and $(v \wedge \neg u)$ are added to the guards of the modified C-elements:

$$\begin{array}{lll}
\ldots & & \\
u \wedge \neg v \wedge x_b \wedge (\neg v_a \vee \neg v_b) & \rightarrow u_a\uparrow & \ldots\ldots\ (1) \\
u \wedge x_b & \rightarrow u_b\uparrow & \ldots\ldots\ (2) \\
\ldots & & \\
v \wedge \neg u \wedge y_b \wedge (\neg u_a \vee \neg u_b) & \rightarrow v_a\uparrow & \ldots\ldots\ (3) \\
v \wedge y_b & \rightarrow v_b\uparrow & \ldots\ldots\ (4) \\
\ldots & &
\end{array}$$

The duplicated primary outputs can be assigned when the outputs of the internal arbiter are valid. The set of gates whose guards are strengthened to be mutually inhibited by checking counterparts, is called a *mutual excluder*. It will be shown in the following section that the mutual exclusion of the duplicated outputs holds even if an error occurs.

In order to implement the given PRS of a DSET arbiter in CMOS technology, the output variables are inverted, as follows:

$$
\begin{array}{lcl}
\ldots & & \\
u \wedge \neg v \wedge x_b \wedge (\overline{v_a} \vee \overline{v_b}) & \rightarrow & \overline{u_a}\downarrow \\
u \wedge x_b & \rightarrow & \overline{u_b}\downarrow \\
\neg u \wedge \neg x_b & \rightarrow & \overline{u_a}\uparrow \\
\neg u \wedge \neg x_b & \rightarrow & \overline{u_b}\uparrow \\
 & & \\
v \wedge \neg u \wedge y_b \wedge (\overline{u_a} \vee \overline{u_b}) & \rightarrow & \overline{v_a}\downarrow \\
v \wedge y_b & \rightarrow & \overline{v_b}\downarrow \\
\neg v \wedge \neg y_b & \rightarrow & \overline{v_a}\uparrow \\
\neg v \wedge \neg y_b & \rightarrow & \overline{v_b}\uparrow \\
\ldots & &
\end{array}
$$

PR (1) and PR (3), specifically the literals $\neg v$ and $\neg u$ of are implemented according to the pass-gate transformation, which replaces a transistor in a series with the inverse of the signal of the transistor. Then the remaining part of the PRS can be directly implemented, as shown in Figure 5.14. In the CMOS implementation, the outputs u_a and v_a do not change until the voltages on u and v are separated by at least a n-transistor threshold voltage. That is, the pass-gate part of a DSET arbiter in CMOS technology ensures that neither of $u_a\uparrow$ nor $v_a\uparrow$ is enabled, until an intermediate state of u and v caused by an error is resolved into $(0, 1)$ or $(1, 0)$. (In fact, the pass-gate part of the implementation is similar to the implementation of a filter in a basic arbiter.)

Overall a DSET arbiter grants a mutually exclusive resource to a module only if the internal arbiter makes a valid choice (i.e., (u, v) is $(1, 0)$ or $(0, 1)$), and the resource is not completely granted to another module (i.e., (u_a, u_b) and (v_a, v_b) are $(0, 0)$, $(1, 0)$, or $(0, 1)$).

The suggested DSET arbiter works correctly with the given DD environment, but the DSET arbiter with a different type of environments may incur deadlock, since a corrupted output of the DSET arbiter is not self-corrected. For example, let us consider a specific environment:

$$
\begin{array}{lcl}
u_a \wedge u_b \wedge \neg v_a \wedge \neg v_b \wedge \neg y.ack_a \wedge \neg y.ack_b & \rightarrow & x.ack_a\uparrow, x.ack_b\uparrow \\
v_a \wedge v_b \wedge \neg u_a \wedge \neg u_b \wedge \neg x.ack_a \wedge \neg x.ack_b & \rightarrow & y.ack_a\uparrow, y.ack_b\uparrow
\end{array}
$$

If the duplicated outputs (u_a, u_b, v_a, v_b) is $(1, 1, 1, 0)$ (i.e., v_a is corrupted) and are not

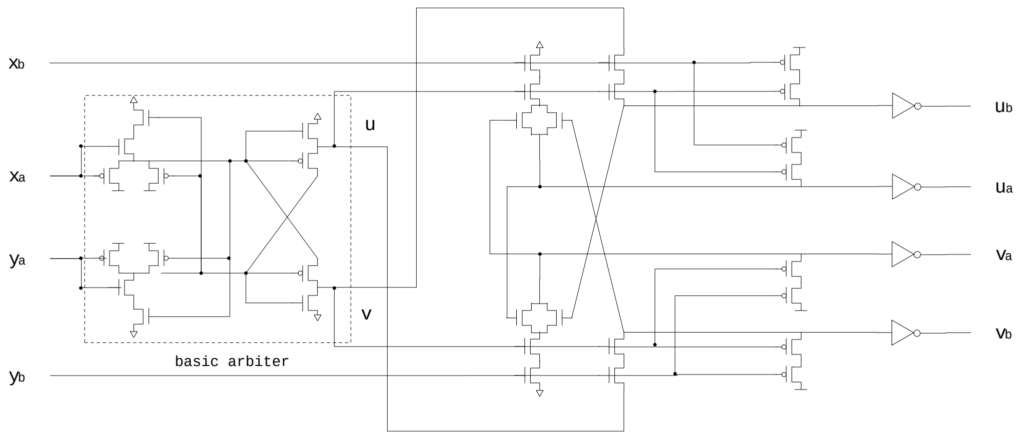


Figure 5.14: *Pass-gate Soft-error Tolerant Arbiter.*

corrected, then deadlock may occur. Fortunately we can easily avoid the deadlock by adding the following circuitry, which restore a corrupted output:

$$(\neg u \wedge v_a \wedge v_b) \;\rightarrow\; u_a\!\downarrow$$
$$(\neg u \wedge v_a \wedge v_b) \;\rightarrow\; u_b\!\downarrow$$
$$(\neg v \wedge u_a \wedge u_b) \;\rightarrow\; v_a\!\downarrow$$
$$(\neg v \wedge u_a \wedge u_b) \;\rightarrow\; v_b\!\downarrow$$

In other words, if a resource is granted to one module (e.g., (u_a, u_b) becomes $(1,1)$), then granting signals for another module (e.g., v_a and v_b) are expected to be reset. In fact, most of the practical environments for a DSET arbiter are similar to the previous DD environment, so that the extra restoring PRs are usually unnecessary.

5.2.3 Mutual Exclusion of Outputs of Duplicated Error Tolerant Arbiter

We have shown an implementation of a DSET arbiter, which is soft-error tolerant in a sense that mutual exclusion of the primary outputs is not violated even if an error occurs. A way of showing that the mutual exclusion, (i.e., $\neg u_a \vee \neg u_b \vee \neg v_a \vee \neg v_b$), always holds, is to check all normal and erroneous states of a DSET arbiter with the DD environment exhaustively. The number of states is a few hundred, so that we can easily check them automatically, but it is not convenient to present all of them here. Instead, we categorize an error into three cases: an error on the outputs, an error on the inputs that are not inputs for the internal arbiter (i.e., x_b and y_b), and an error in the internal arbiter. Then we show that each case of errors does not lead to the violation of the mutual exclusion of the primary outputs.

First, let us consider an error on the output. If there is no error, (u_a, u_b, v_a, v_b) is always $(1,1,0,0)$, $(0,0,1,1)$ or an intermediate state from the initial state $(0,0,0,0)$ to the two states (e.g., $(0,1,0,0)$). That is, there are at least two variables whose values are `false`. Accordingly, even if an error on one of the outputs occurs, there exists at least one variable whose value is `false`: the mutual exclusion holds. Note that the error on one of the outputs, which is an error on the inputs of the DD environment, can be tolerated by the DD environment.

Second, let us consider an error on either x_b (or y_b). The design of the mutual excluders, specifically, the terms $u \wedge x_b$ and $\neg u \wedge \neg x_b$, can mask out the error. (Note that this is similar to cross-coupled C-elements of DD circuits.) x_a, which is the twin of x_b, holds a correct value, and accordingly u holds a correct value, since $u\uparrow$ follows $x_a\uparrow$ and $u\downarrow$ follows $x_a\downarrow$ eventually: the assignments of u are considered to be delayed assignments of x_a. As a result, even if an error on x_b occurs, the value of u, which reflects the value of x_a, prevents the corrupted value of x_b from enabling the PRs of the primary outputs prematurely. This is the same to the case of an error on y_b. Hence, the mutual exclusion is ensured even though an error on either x_b or y_b occurs.

Third, let us consider an error in the internal basic arbiter of a DSET arbiter, which is an error on x_a, y_a, u, v, $\overline{s}$, and $\overline{t}$. Specifically out concern is the error that flips the outputs of the internal arbiter simultaneously: (u, v) is changed from $(1,0)$ to $(0,1)$ or from $(0,1)$ to $(1,0)$. If the flipping causes all PRs of $u_a\uparrow$, $u_b\uparrow$, $v_a\uparrow$, $v_b\uparrow$ to fire, the mutual exclusion can be violated.

Here we show that the violation of the mutual exclusion (i.e., all PRs of primary outputs fire) happens only if two or more abnormal changes of the outputs of the internal arbiter (e.g., (u, v): $(1.0) \to (0,1) \to (1,1)$). For showing that, we take the following two steps. (1) Constraints on concurrent firings between the PRs of $u_a\uparrow$, $u_b\uparrow$, $v_a\uparrow$, and $v_b\uparrow$ are shown. (2) All possible sequences of the firings, which satisfy the constraints, are enumerated; the sequences always require two abnormal changes of (u, v). Hence, we can conclude that a single error in the internal basic arbiter cannot break the mutual exclusion of a DSET arbiter.

There is a constraint that $u_a\uparrow$ and $v_a\uparrow$ cannot fire concurrently

$$u_a\uparrow \prec v_a\uparrow \text{ or } v_a\uparrow \prec u_a\uparrow.$$

A sequential order $\prec$ is a relation between firings: $p \prec q$ means that the event p happens before the event q. The constraint comes from the fact that the conjunction of the guards of (1) and (3) is invariably `false`:

$$(u \wedge \neg v \wedge x_b \wedge ...) \wedge (v \wedge \neg u \wedge y_b \wedge ...) \equiv \texttt{false}$$

Likewise, other constraints between the firings of $u_a\uparrow$, $u_b\uparrow$, $v_a\uparrow$, $v_b\uparrow$ also exist.

$$u_a\uparrow \prec v_b\uparrow \text{ or } v_b\uparrow \prec u_a\uparrow$$
$$u_b\uparrow \prec v_a\uparrow \text{ or } v_a\uparrow \prec u_b\uparrow.$$

The constraints come from the invariable relationships among the guards of (1), (2), (3), and (4):

$$(u \wedge \neg v \wedge x_b \wedge (\neg v_a \vee \neg v_b)) \wedge (v \wedge y_b) \equiv \texttt{false}$$
$$(u \wedge x_b) \wedge (v \wedge \neg u \wedge y_b \wedge (\neg u_a \vee \neg u_b)) \equiv \texttt{false}$$

In addition to this, $u_a\uparrow$ cannot fire after both $v_a\uparrow$ and $v_b\uparrow$ fire because of the term $(\neg v_a \vee \neg v_b)$ in the guard of (1), and similarly $v_a\uparrow$ cannot fire after both $u_a\uparrow$ and $u_b\uparrow$ fire. Accordingly we obtain the following constraints from the design of a mutual excluder:

$$u_a\uparrow \preceq v_a\uparrow \text{ or } u_a\uparrow \preceq v_b\uparrow$$
$$v_a\uparrow \preceq u_a\uparrow \text{ or } v_a\uparrow \preceq u_b\uparrow$$

The mutual exclusion (i.e., $\neg u_a \vee \neg u_b \vee \neg v_a \vee \neg v_b$) can be violated only if all PRs of $u_a\uparrow$, $u_b\uparrow$, $v_a\uparrow$, $v_b\uparrow$ fire. Given the ordering constraints on the concurrent firings, we can enumerate all possible sequences of firings of $\{u_a\uparrow,\ u_b\uparrow,\ v_a\uparrow,\ v_b\uparrow\}$. Each sequence of the firings, if it occurs, contains one of the following sub-sequences:

$$u_a\uparrow \prec v_a\uparrow \prec u_b\uparrow \qquad \ldots\ldots \quad (a)$$
$$v_a\uparrow \prec u_a\uparrow \prec v_b\uparrow \qquad \ldots\ldots \quad (b)$$

In order to show that an error cannot induce neither of the sub-sequences, let us assume without loss of generality that $u_a\uparrow \prec v_a\uparrow$. Including the assignments of u and v, the sub-sequence (a) can be expanded as follows:

$$\{u\uparrow, v\downarrow\} \prec u_a\uparrow \prec \{u\downarrow, v\uparrow\} \prec v_a\uparrow \prec \{u\uparrow\} \prec u_b\uparrow$$

The sequence implies that the internal arbiter changes its choice two times. That is, an internal arbiter makes a choice (i.e., $\{u\uparrow, v\downarrow\}$), and then changes its choice (i.e., $\{u\downarrow, v\uparrow\}$) after $u_a\uparrow$, and changes it again (i.e., $\{u\uparrow\}$). As we know, a single error in a basic arbiter can flip the choice (i.e., from $(u, v) = (1, 0)$ to $(0, 1)$ or from $(0, 1)$ to $(1, 0)$) at most one time. Hence, the sequence that satisfies the ordering constraints and leads to the violation of the mutual exclusion cannot occur unless two or more errors occur in the DSET arbiter. The mutual exclusion of a DSET arbiter holds even if an error occurs.

5.2.4 Simulation Results of Error Tolerant Arbiter

The layout of a DSET arbiter has been done in the TSMC 0.18$-\mu$m CMOS process, and has been tested in SPICE simulations.

A SPICE simulation result of a DSET arbiter is shown in Figure 5.15. In the simulation, all the inputs of a DSET arbiter become `true` simultaneously, and an error is injected into an internal basic arbiter, which results in a change of the choice of the internal basic arbiter. Specifically, the duplicated inputs arrive simultaneously at 1.5 ns, and an error on the basic arbiter inside the DSET arbiter occurs at 2.2 ns. Since the choice of the internal basic arbiter is changed from $v\uparrow$ to $u\uparrow$, $u_b\uparrow$ fires at 2.4 ns. But the firing of $u_a\uparrow$ is restrained by a mutual excluder because $\{v_a\uparrow, v_b\uparrow\}$ has already fired at 2.2 ns; after $\{v_a\uparrow, v_b\uparrow\}$ is acknowledged at 2.6 ns, $u_a\uparrow$ can fire. Accordingly the DD environment first sees the choice of $\{v_a\uparrow, v_b\uparrow\}$ at 2.2 ns, and then the choice of $\{u_a\uparrow, u_b\uparrow\}$ at 2.7 ns. Throughout the simulations, the mutual exclusion between the duplicated outputs of a DSET arbiter holds.

5.2.5 Summary

Arbiters are essential components in most asynchronous systems, but the DD scheme cannot be applied to an arbiter, because two copies of an arbiter may not provide the same result. For use with DD circuits, we have devised a DSET arbiter: one basic arbiter is used to make a non-deterministic choice, and the choice is duplicated through a mutual excluder, which prevents an error inside a DSET arbiter from being duplicated. In a DSET arbiter, mutual

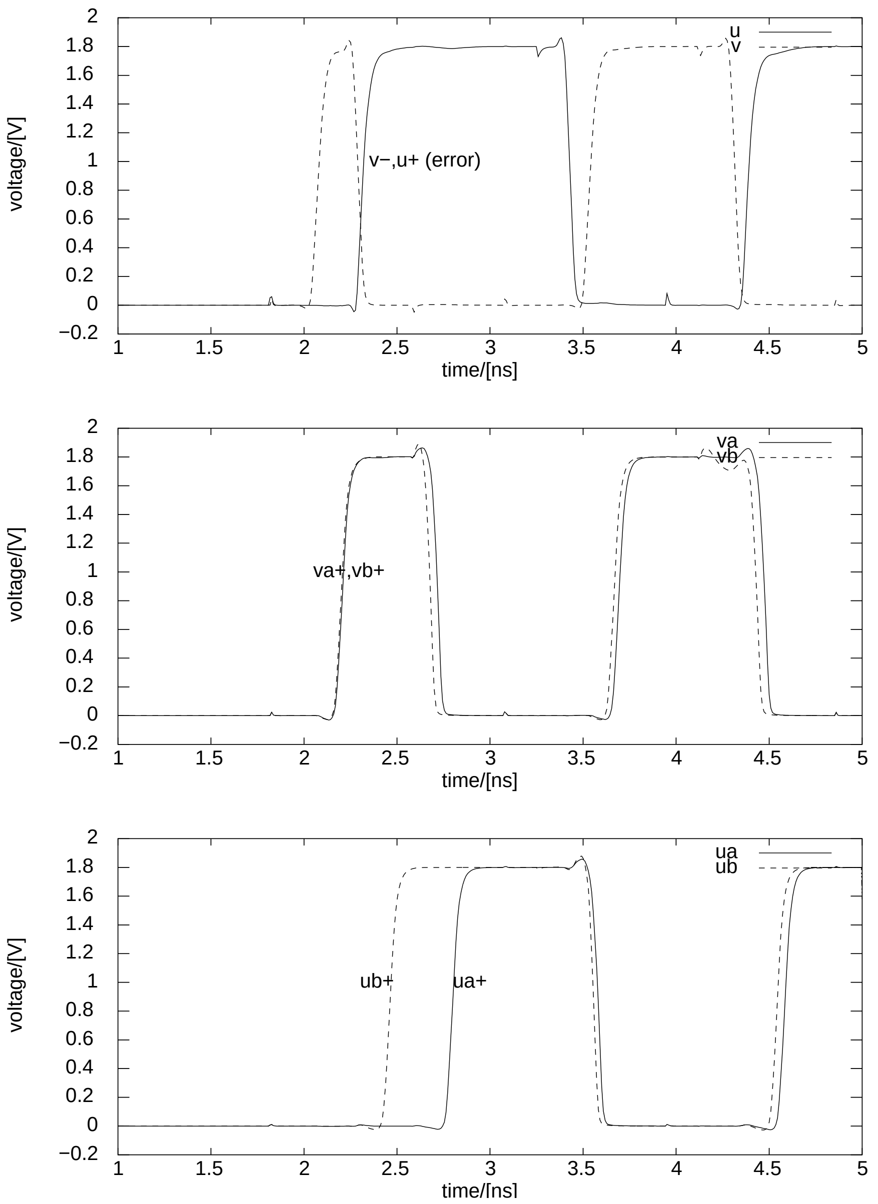


Figure 5.15: *Waveforms for Nodes in DSET Arbiter. After the duplicated inputs arrive simultaneously, an error in the basic arbiter at 2.2 ns causes* $v \downarrow$ *and* $u \uparrow$ *(i.e., the result from a basic arbiter is flipped). Although the transition of one of the duplicated output* $u_b \uparrow$ *occurs, the transition of the other output* $u_a \uparrow$ *does not occur until the previous arbitration (i.e.,* $v_a \uparrow$, $v_b \uparrow$*) is acknowledged.*

exclusion of the outputs is ensured, which allows granting of a shared resource mutually exclusively even if an error occurs.

5.3 Duplicated FPGA Using Self-correcting Programmable Bits

5.3.1 Overview

Field Programmable Gate Arrays (FPGA) are becoming steadily more attractive because of recent enhancements in their capacity and performance. Meanwhile, the concern for clock distribution and increased power consumption of synchronous FPGA is growing. As an alternative, it is noteworthy to consider QDI FPGA. Several general QDI FPGA architectures have been proposed [38, 39]. In the following sections, we shall demonstrate how to make the Wong's FPGA architecture soft-error tolerant.

A basic FPGA tile consists of a cluster, two connection boxes (C-boxes) and a switch box (S-box) [40]. A system described in a high-level language is decomposed into implementable modules, which correspond to clusters, and whose interconnection information is mapped into C-boxes and S-boxes. C-boxes are used for connecting a cluster to interconnect paths and S-boxes are used for switching interconnect paths. Logic cells in a cluster share inputs and outputs, and each logic cell is based on the PCHB template, which consists of a computation part and completion checkers of input and output channels. The cell contains programmable bits: some of the bits are for configuring computations and the others are for setting patterns of communication between cells.

There is the possibility of two types of soft errors in an FPGA: an error in computation parts (e.g., pulldown stacks) can cause temporarily incorrect computations, and an error on programmable bits may change the device's configuration semi-permanently, as well as leading to incorrect computations.

5.3.2 Designing Error Tolerant FPGA Cells

The computation parts of a FPGA cell are based on the PCHB template, which can be protected by the DD scheme. A programmable bit to control the functionality of a FPGA cell employs an SRAM cell, which consists of six transistors. To protect programmable bits, the duplicated keeper, which was explained in the Section 5.1.1, can be adapted, as shown in Figure 5.17 (a), which consists of 18 transistors.

Another protection scheme is based on a dual interlocked cell (DICE) [24]. A DICE

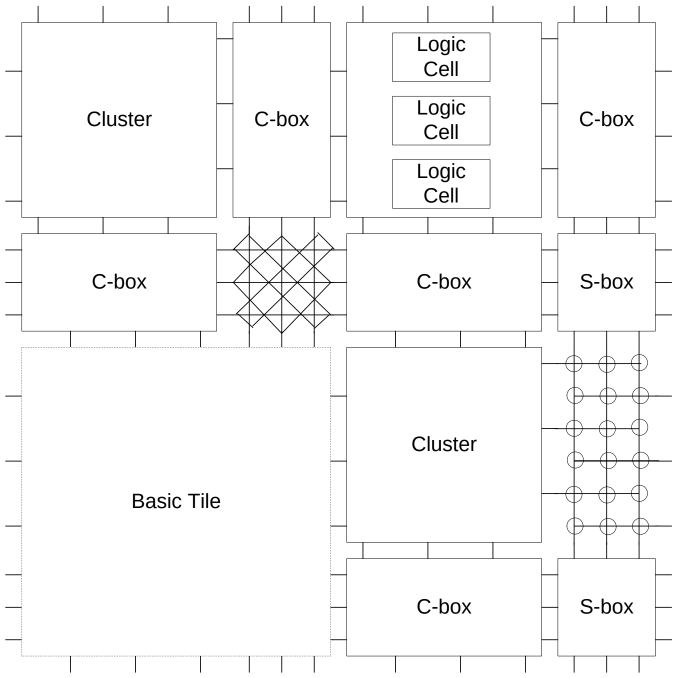


Figure 5.16: *FPGA Architecture.*

with the pass transistors, as shown in Figure 5.17 (b), consists of 10 transistors; this is more compact than the duplicated keeper, so we prefer to use this construction for soft-error tolerant FPGA. The way of tolerating an error in the DICE is the following. Four nodes in a DICE encode a bit with two pairs of complementary values: $(c_a, \overline{c}_a, c_b, \overline{c}_b)$ is $(0,1,0,1)$ or $(1,0,1,0)$. If one of four nodes in the cell is corrupted by an error, one of the adjacent nodes always keeps a correct value to restore corrupted data. For example, an error on c_a in the state $(0,1,0,1)$ causes $\overline{c}_b$ to be an unknown state since both pullup and pulldown transistors are turned on. But $\overline{c}_a$ still holds 1, which turns on the pulldown transistor of c_a to restore the corrupted c_a. Then $\overline{c}_b$ is also restored. Other erroneous cases can be similarly analyzed. Each DICE can tolerate any single error or even two errors in a pair of non-adjacent nodes (i.e., $\{c_a, c_b\}$, $\{\overline{c}_a, \overline{c}_b\}$).

We can construct a soft-error tolerant FPGA logic cell based on the DD scheme with DICEs, as shown in Figure 5.18. In the same manner, we can construct soft-tolerant S-boxes and C-boxes.

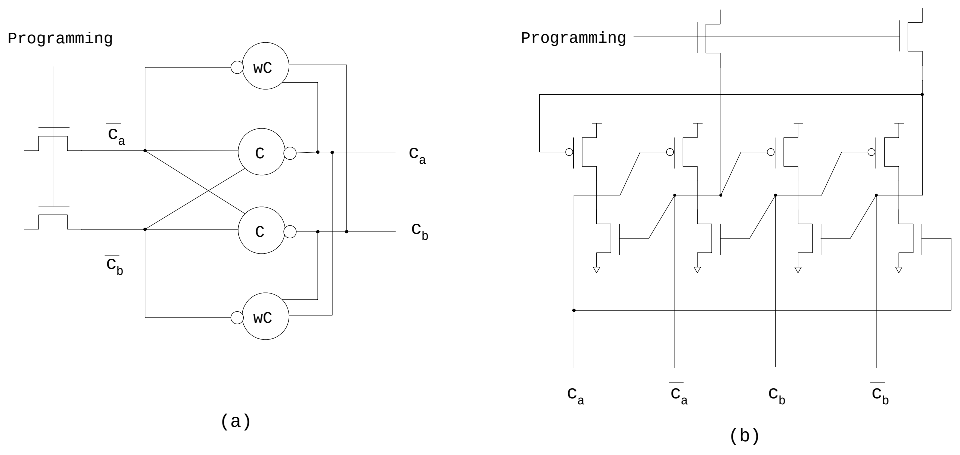


Figure 5.17: *(a) Double-checked Programmable Bit (b) Programmable Bit based on Dual Interlocked Cell.*

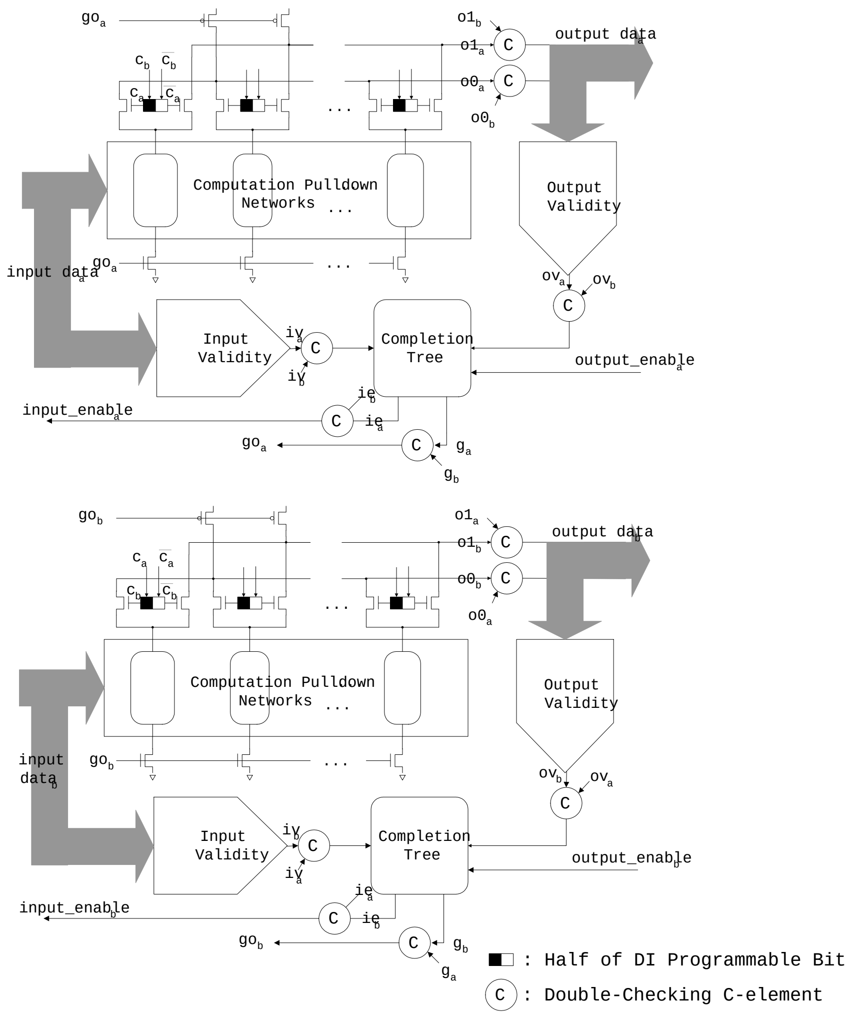


Figure 5.18: *Block Diagram of Soft-error Tolerant FPGA Logic Cell.*

5.3.3 Summary

Besides the computing logic, an FPGA has programmable bits, which determines the function the FPGA computes and its communication patterns. In order to protect an FPGA from soft errors, both the computation part and the configuration bits must be protected. While the DD scheme is applied to the computation part, the dual interlocked cells, which can self-correct an error, are used for programmable bits. Since the whole architecture of soft-error tolerant FPGAs is based on dual copies of a conventional FPGA, existing synthesis procedures for FPGAs can be easily adapted to the soft-error tolerant FPGAs.

5.4 Self-correcting Memory and Hamming-coded Memory

In this section, the overview of a memory unit is given first. Then two designs of error tolerant memory are detailed. Instead of applying the DD scheme to an entire memory unit, we employ conventional solutions to protect an array of bit storages in the memory unit. In one design, SRAMs are replaced with dual interlocked cells, which have a self-recovering structure; in another design, a Hamming code is applied to the array of SRAMs.

5.4.1 Overview

We shall study a memory unit whose description in CHP is

```
MEM ≡ *[RW?rw, ADDR?addr;
        [rw = read  ⟶ DATA_OUT!mem[addr]
        ▯rw = write ⟶ DATA_IN?mem[addr]
        ]]
```

The memory unit stores data from the input channel *DATA_IN* to an array of storage elements, and loads data from the array onto the output channel *DATA_OUT*, according to the control channel *RW* and to the address channel *ADDR*.

We can decompose a memory unit with a large array into several memory units with a small array for synthesizing circuits easily. For example, a decomposition of the memory unit, if it has 64 words, is as follows:

```
$ConvertAddr \equiv$ *[$ADDR?addr, RW?rw;$
       $RWC!rw,$
       $CONTROL!(addr$[4]$, addr$[5]$),$
       $ADDR\_LSB!(addr$[0]$, addr$[1]$, addr$[2]$, addr$[3]$);$
       [$rw =$ read $\longrightarrow$ $MergeC!(addr$[4]$, addr$[5]$)$
       $\Box rw =$ write $\longrightarrow$ $SplitC!(addr$[4]$, addr$[5]$)$
       ]
     ]

$SplitControl \equiv$ *[$CONTROL?c, RWC?rw, ADDR\_LSB?addr;$
       [$c = 0 \longrightarrow RW0!rw,\ ADDR0!addr$
       $\Box c = 1 \longrightarrow RW1!rw,\ ADDR1!addr$
       $\Box c = 2 \longrightarrow RW2!rw,\ ADDR2!addr$
       $\Box c = 3 \longrightarrow RW3!rw,\ ADDR3!addr$
     ]]

$SplitData \equiv$ *[$SplitC!c;$
       [$c = 0 \longrightarrow DATA\_IN?d;\ DATA\_IN0!d$
       $\Box c = 1 \longrightarrow DATA\_IN?d;\ DATA\_IN1!d$
       $\Box c = 2 \longrightarrow DATA\_IN?d;\ DATA\_IN2!d$
       $\Box c = 3 \longrightarrow DATA\_IN?d;\ DATA\_IN3!d$
     ]]

$MemCore0 \equiv$ *[$RW0?rw,\ ADDR0?addr;$
     [$rw =$ read $\longrightarrow DATA\_OUT0!mem$[$addr$]
     $\Box rw =$ write $\longrightarrow DATA\_IN0?mem$[$addr$]
     ]]
$MemCore1 \equiv$ *[$RW1?rw,\ ADDR1?addr;$
     ... ]
$MemCore2 \equiv$ *[$RW2?rw,\ ADDR2?addr;$
     ... ]
$MemCore3 \equiv$ *[$RW3?rw,\ ADDR3?addr;$
     ... ]
```

```
$MergeData \equiv$ *[$MergeC?c$
    [$c = 0 \longrightarrow DATA\_OUT0?data$
    ▯$c = 1 \longrightarrow DATA\_OUT1?data$
    ▯$c = 2 \longrightarrow DATA\_OUT2?data$
    ▯$c = 3 \longrightarrow DATA\_OUT3?data$
   ];
    $DATA\_OUT!data$
  ]
```

(The notation '*addr*[*n*]' represents the *n*-th bit of the address value.) The corresponding process diagram is shown in Figure 5.19. The 64-word memory unit is decomposed into four 16-word memory units, processes distributing controls, and processes splitting/merging data. When data is written on the memory unit, the process *SplitData* sends data from the channel *DATA_IN* into one of the four memory units, which is selected according to the address; when data is read, the process *MergeData* reads data from one of the memory units, and sends out the data to the channel *DATA_OUT*.

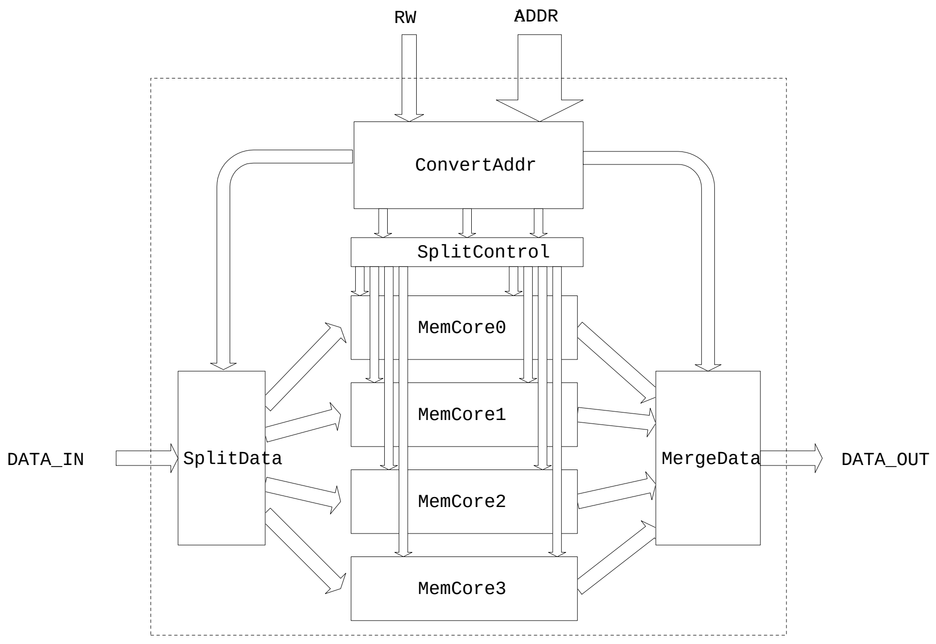


Figure 5.19: *Decomposition of Memory Unit.*

In the decomposition, each *MemCore* is identical to the process *Mem* except for the size of the array, so that we can apply this decomposition recursively, if necessary. Ultimately, the array of the memory unit can be decomposed into an array of bit-storage processes. But the fine-grained decomposition causes channel proliferation and excessive wirings at the circuit level. In order to avoid channel proliferation, we implement a memory unit in the following way. The control signals, input channels, and output channels are shared for the array, as shown in Figure 5.20. The bit lines are the shared input and output channels of the array, and the word lines are the shared address channel for the array. As a result, a circuit of the data array has a simple and regular structure. For the shared structure, auxiliary circuits around the array are added, as shown in Figure 5.21. For example, an address decoder converts the address of the channel $ADDR$ into word lines, which are based on a one-of-n code; a controller, whose specification is in Appendix B, generates signals for precharging bit lines and acknowledgment signals for the input channels $DATA\_IN$, $ADDR$, and RW.

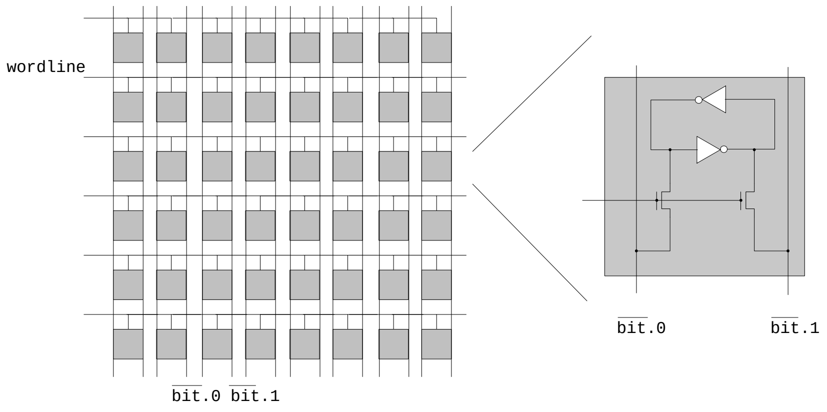

wordline
$\overline{\text{bit}}$.0 $\overline{\text{bit}}$.1
$\overline{\text{bit}}$.0
$\overline{\text{bit}}$.1

Figure 5.20: *Array of Storage Elements.*

5.4.2 Error Tolerant Memory Using Self-correcting Memory Cells

We protect auxiliary circuits around the array by employing the DD scheme: gates of original circuits are duplicated, and cross-coupled C-elements are added. On the other hand, we cannot apply the DD scheme to an array of SRAMs. As we have shown in the previous chapter, the DD scheme exploits the persistence of inputs of gates in QDI circuits, but the persistence does not hold for storage elements like SRAMs. (In fact, we can apply the DD scheme, but it requires adding cross-coupled C-elements to every SRAM, which is impractical.)

Instead, the dual interlocked cell (DICE), as shown in Figure 5.22, is adapted as an

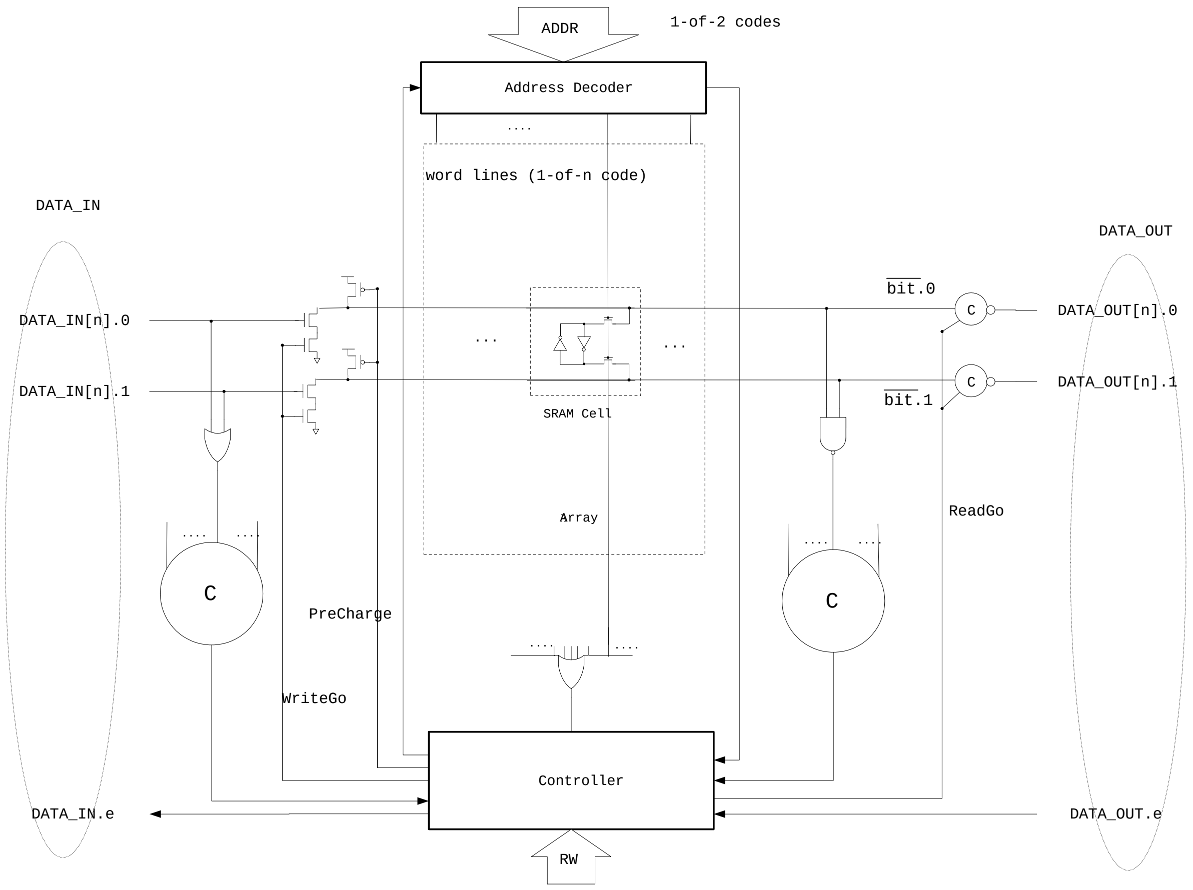


Figure 5.21: *Circuit of Mermoy Core.*

error tolerant storage element [24]. A DICE encodes a bit as two pairs of complementary values: $(\overline{x.0_a}, \overline{x.1_a}, \overline{x.0_b}, \overline{x.1_b}) = (0, 1, 0, 1)$ or $(1, 0, 1, 0)$. The value of each of four nodes of a DICE is controlled by the value of two complementary adjacent nodes. For example, $\overline{x.1_a}$ is assigned to be `true` if $\overline{x.0_a}$ is `false`, and $\overline{x.1_a}$ is assigned to be `false` if $\overline{x.0_b}$ is `true`.

A DICE corrects an error by itself. If an error, which occurs on $\overline{x.1_a}$ in the $(0, 1, 0, 1)$ state, turns the state into $(0, 0, 0, 1)$, then $\overline{x.0_b}$ will become an unknown state, because both n-transistor and p-transistor of the node $\overline{x.0_b}$ are turned on. However the p-transistor of $\overline{x.1_a}$ is kept from being turned on, because the node $\overline{x.0_a}$ holds *Vdd*. Consequently $\overline{x.1_a}$ is restored to be `true`, and the unknown state of $\overline{x.0_b}$ will be settled into `false`, so that the state of the four nodes becomes $(0, 1, 0, 1)$ again. (Similarly an error on the other nodes in different states can be tolerated.) Hence, a DICE tolerates an error.

A DICE has two pass gates in series, which are switched by duplicated word lines from a DD address decoder. The duplicated word lines are necessary to tolerate an error on a word line. If a DICE is accessed by a single word line and an error occurs on the single word line, then an incorrect row of an array can be read, or data can be written to an incorrect row by accident.

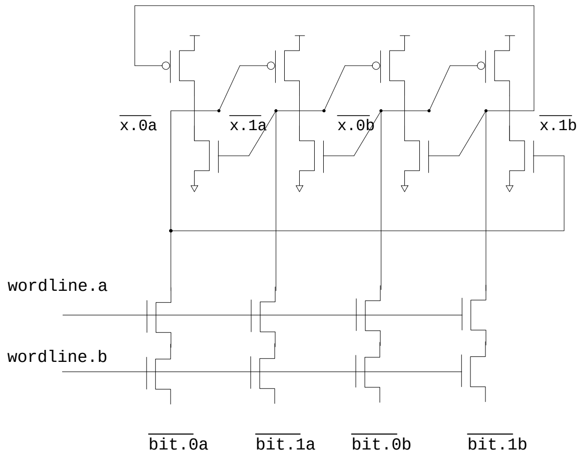


Figure 5.22: *Calin's Dual Interlocked Cell (DICE).*

Although a DICE is easy to use and is well matched with DD auxiliary circuits, a DICE at the circuit level is hard to design compactly due to its interwoven connections. A DICE is larger by approximately a factor of four than an SRAM, so that an error tolerant memory unit using DICEs becomes larger by a factor of four than the original memory unit.

5.4.3 Error Tolerant Memory Based on Hamming Code

An alternative to DICEs is Hamming code: redundant bits are added to each word. A Hamming code, which is a type of error correcting code, is widely used because it can be encoded/decoded using minimal circuitry compared with other error correcting codes. The Hamming-coded array uses less redundancy for restoration than the array of DICEs. For example, eight extra bits are needed for eight bits in the array of DICEs; only four extra parity bits are needed to tolerate an error in an eight-bit Hamming-coded array. Therefore a Hamming-coded memory unit with a large array is smaller than a corresponding DICE memory unit, even though a Hamming-coded array requires an encoder and a decoder whose extra cost does not depend on the size of the array. Note that a Hamming-coded array consists of SRAMs, so that it cannot correct an error by itself, and consequently multiple errors in one row of the Hamming-coded array cannot be tolerated. If necessary, this problem can be easily addressed by scrubbing the array continuously: reading out the data, checking the parity for data errors, then writing back any corrections to the array [41].

A Hamming-coded memory requires encoding incoming data and decoding outgoing data. For encoding, redundant bits are added to an original message. The redundant bits (i.e., parity bits) are the sums of selected bits in the message. For example, a four-bit message $u = (u_1, u_2, u_3, u_4)$ is mapped into a codeword of a Hamming code, which encodes four-bit data with three parity bits:

$$(u_1, u_2, u_3, u_4, u_2 \oplus u_3 \oplus u_4 \oplus, u_1 \oplus u_3 \oplus u_4 \oplus, u_1 \oplus u_2 \oplus u_4)$$

A CHP description of a corresponding encoder is as follows:

```
Encoder ≡
        *[Data?u;
        EncodedD!(u[0], u[1], u[2], u[3],
        u[1]⊕u[2]⊕u[3], u[0]⊕u[2]⊕u[3], u[0]⊕u[1]⊕u[3])
        ].
```

The CHP can be implemented according to the conventional synthesis method.

On the other hand, a decoder in the memory unit restores an original message even if a bit is corrupted in a codeword from a Hamming-coded array. In order to check whether a codeword is corrupted or not, a decoder of the four-bit message first computes the following parities:

$$\begin{aligned} p_1 &\equiv x_2 \oplus x_3 \oplus x_4 \oplus x_5 \\ p_2 &\equiv x_1 \oplus x_3 \oplus x_4 \oplus x_6 \\ p_3 &\equiv x_1 \oplus x_2 \oplus x_4 \oplus x_7 \end{aligned}$$

(x_i is an i-th bit of a codeword.) If a codeword is intact, then all the parities are `false`; if one bit in a codeword is corrupted, then the values of the parities indicate the location of the corrupted bit. For example, the first bit of a codeword is corrupted, then (p_1, p_2, p_3) becomes $(0, 1, 1)$, since x_1 is included only in the computation of p_2 and p_3; if the second bit of a codeword is corrupted, then (p_1, p_2, p_3) becomes $(1, 0, 1)$. Exploiting the conditions, a decoder reconstructs an original message, as follows:

$$\begin{aligned} v_1 &= x_1 \oplus (\overline{p_1} \cdot p_2 \cdot p_3) \\ v_2 &= x_2 \oplus (p_1 \cdot \overline{p_2} \cdot p_3) \\ v_3 &= x_3 \oplus (p_1 \cdot p_2 \cdot \overline{p_3}) \\ v_4 &= x_4 \oplus (p_1 \cdot p_2 \cdot p_3) \end{aligned}$$

The reconstructed message from a codeword $(x_1, x_2, ..., x_7)$ is (v_1, v_2, v_3, v_4). A decomposed CHP description of a corresponding decoder is as follows:

```
$Copier \equiv$

    $*[CodeWord?x;\quad CodeWord0!x,\quad CodeWord1!(x[0], x[1], x[2], x[3])]$

$Buffer \equiv$

    $*[CodeWord1?x;\quad CodeWord1Buf!x]$

$Parity\ \ Checker \equiv$

    $*[CodeWord0?x;$
    $Parity!(x[1] \oplus x[2] \oplus x[3] \oplus x[4], x[0] \oplus x[2] \oplus x[3] \oplus x[5], x[0] \oplus x[1] \oplus x[3] \oplus x[6])$
    $]$

$Corrector \equiv$

    $*[CodeWord1Buffered?x,\quad Parity?p;$
      $DecodedD!(x[0] \oplus (\overline{p[0]} \cdot p[1] \cdot p[2]), x[1] \oplus (p[0] \cdot \overline{p[1]} \cdot p[2]),$
                $x[2] \oplus (p[0] \cdot p[1] \cdot \overline{p[2]}), x[3] \oplus (p[0] \cdot p[1] \cdot p[2]))$
    $]$
$Decoder \equiv\ Copier \ \|\ Buffer \ \|\ Parity\ \ Checker \ \|\ Corrector$
```

The process graph of the decomposed CHP is shown in Figure 5.23. All of the processes are buffer-like processes, so that we can implement the processes easily according to the conventional synthesis method. Combined with an encoder and a decoder, the final block diagram of an error-tolerant memory core based on the Hamming code is shown in Figure 5.24.

Hamming codes are suitable for the storage array, but they cannot be applied readily to computing logic for error tolerance, because computations based on a Hamming code

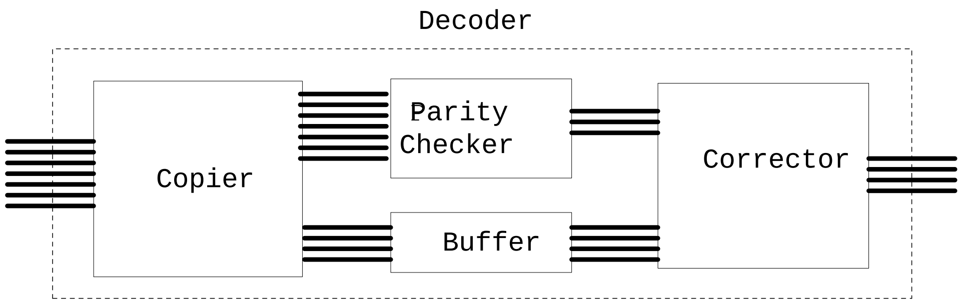


Figure 5.23: *Decomposition of [7,4] Hamming-code Decoder. Each solid line represents a one-of-two code channel.*

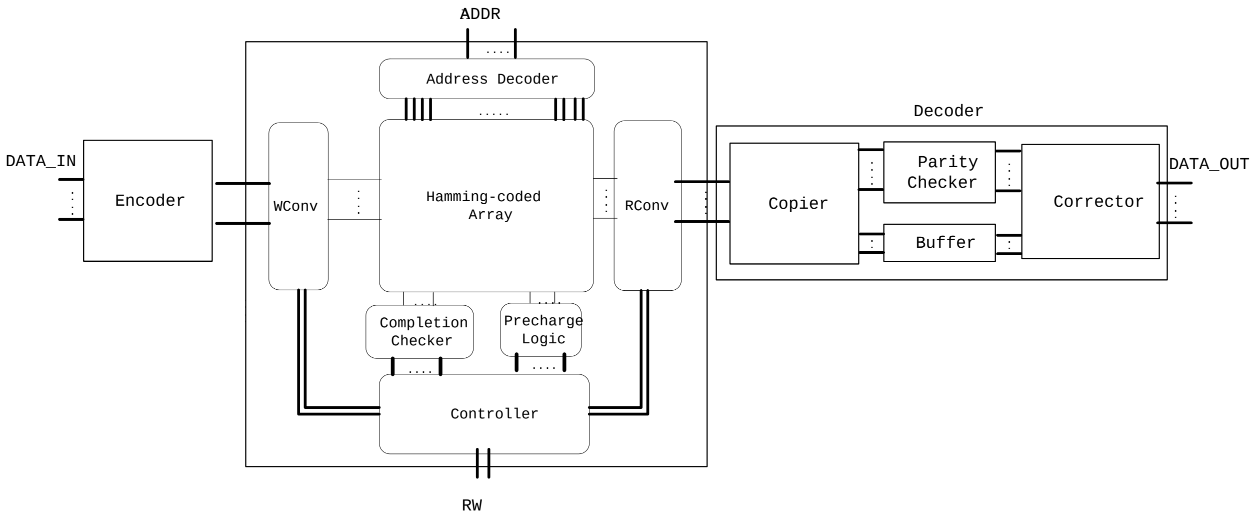


Figure 5.24: *Block Diagram of Memory Core Using Hamming Code.*

generally require complex circuits. Therefore, the DD scheme is applied to an encoder, a decoder, a controller and an address decoder of a memory unit in order to tolerate an error. Since the circuits around the array, including an encoder and a decoder, use duplication codes while the array adopts a Hamming code, it is necessary to change an encoding of a duplicated code into an encoding of a Hamming code and vice versa.

It is easy to convert a duplicated codeword into a Hamming codeword. For example, a DD encoder provides a duplicated Hamming codeword, and then half of the outputs are used as the inputs of the storage array, as shown in Figure 5.25, and half of the redundancy is discarded. The discarding does not affect the error tolerance of encoded codewords, since sufficient redundancy to tolerate one error is already encoded into a Hamming codeword. The circuit diagram of the interface between a DD circuit and an input side of a memory unit is shown in Figure 5.26. Note that all signals are duplicated in order to tolerate an error.

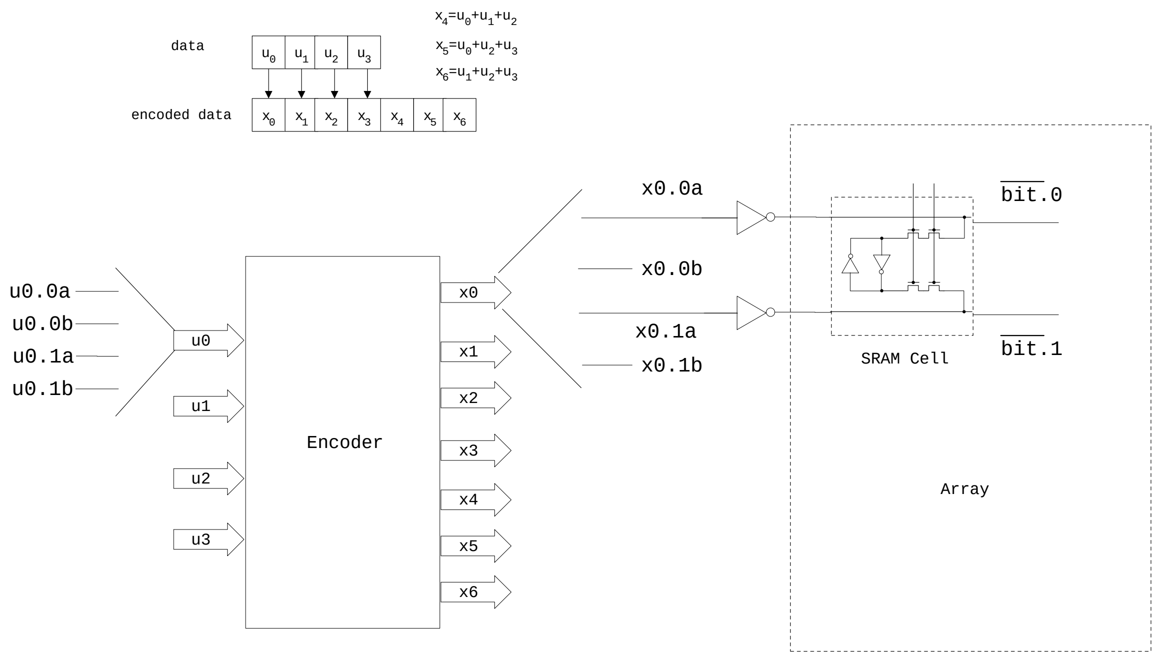


Figure 5.25: *Converting a Duplicated Codeword into a Hamming Codeword.*

However, it is not straightforward to convert a Hamming codeword to a duplicated codeword. The outputs of the array consist of one-of-two codes, each of which represents a bit of a Hamming codeword, while a DD circuit (e.g., a DD decoder) expects copies of a one-of-two code for each bit. A simple way of interfacing is to fork the outputs, as shown in Figure 5.27. But if an error occurs, it is copied to several places, and the copied errors can cause the DD decoder to fail as if multiple errors occurred in the DD decoder.

We can address the problem by inserting a circuit to check the validity of the bit lines, say, whether $bit.0 \wedge \neg bit.1 \vee \neg bit.0 \wedge bit.1$ holds or not, as follows:

$$bit.0 \wedge \neg bit.1 \;\rightarrow\; bit.0_a\uparrow$$
$$bit.0 \wedge \neg bit.1 \;\rightarrow\; bit.0_b\uparrow$$

$$bit.1 \wedge \neg bit.0 \;\rightarrow\; bit.1_a\uparrow$$
$$bit.1 \wedge \neg bit.0 \;\rightarrow\; bit.1_b\uparrow$$

The corresponding circuit diagram is shown in Figure 5.28. However there is another problem that we need to address: an error on one of the bit lines can cause two bit flips. Since $bit.0$ and $bit.1$ drive each other through inverters, an error causes $(bit.0, bit.1)$ to

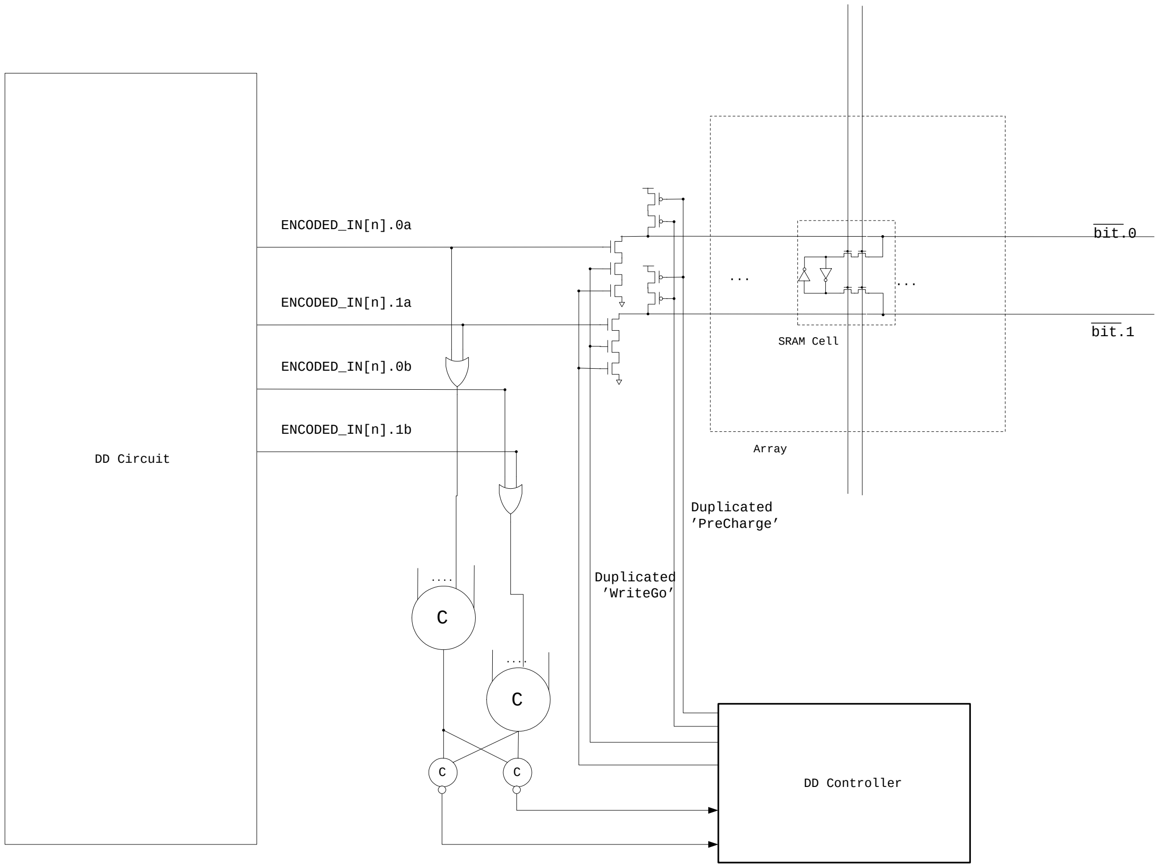


Figure 5.26: *Interface Between DD Circuit and Input Side of Memory Unit.*

change from $(1, 0)$ to $(0, 1)$ or vice versa. Consequently, the two bit flips cause the duplicated bit lines $(bit.0_a, bit.0_b, bit.1_a, bit.1_b)$ to be $(1, 1, 1, 1)$ even with the filter. If the duplicated bit lines are copied to two or more circuits, as shown in Figure 5.29, the invalid state is problematic: one circuit may read the duplicated bit lines of $(1, 1, 1, 1)$ as $(1, 1, 0, 0)$, but another circuit may read it as $(0, 0, 1, 1)$, since the ordering of assignments is not assumed in a QDI circuit. In other words, the invalid state provides different information to different parts (e.g., parity computation and message reconstruction) of a DD decoder, and ultimately it can cause erroneous decoding.

In order to avoid the problem, a protection interface is added to the bit lines, which disables the assignments of $\{bit.0_a\uparrow, bit.0_b\uparrow\}$ once the assignments of $\{bit.1_a\uparrow, bit.1_b\uparrow\}$ are completed, and vice versa:

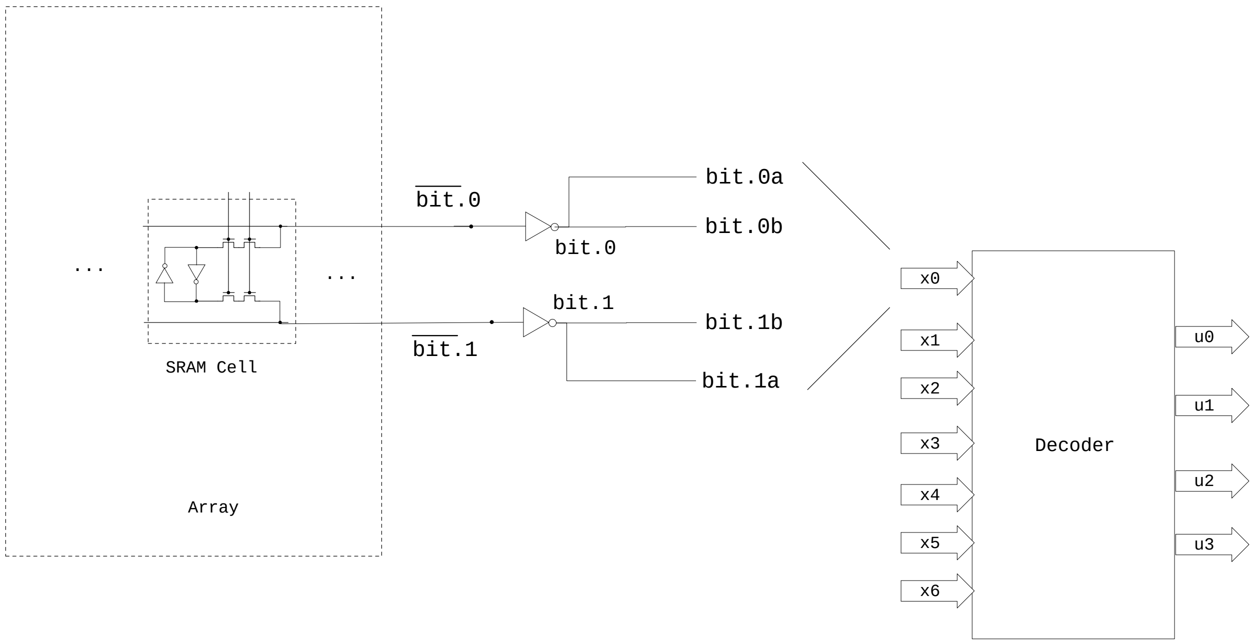


Figure 5.27: *Copying the Bit Lines of the Array (does not work).*

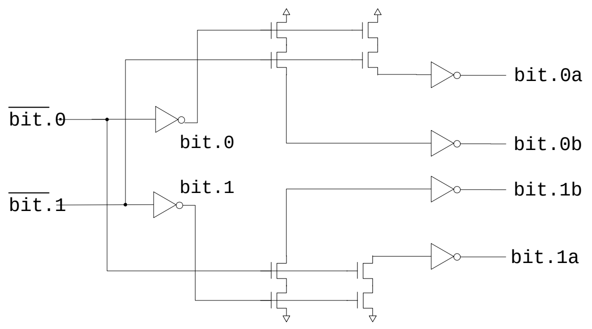


Figure 5.28: *Copying the Bit Lines with Filter (does not work).*

$$\begin{aligned} bit.0 \wedge \neg bit.1 \wedge (\neg bit.1_a \vee \neg bit.1_b) &\rightarrow bit.0_a\!\uparrow \\ bit.0 \wedge \neg bit.1 &\rightarrow bit.0_b\!\uparrow \end{aligned}$$

$$\begin{aligned} bit.1 \wedge \neg bit.0 \wedge (\neg bit.0_a \vee \neg bit.0_b) &\rightarrow bit.1_a\!\uparrow \\ bit.1 \wedge \neg bit.0 &\rightarrow bit.1_b\!\uparrow \end{aligned}$$

The firings of $bit.0_a\!\uparrow$ and $bit.1_a\!\uparrow$ are mutually inhibited by cross-checking, which is similar to the mutual excluder of a DSET arbiter. The mutual excluder prevents an error from causing the state of the duplicated bit lines to be the invalid state $(1, 1, 1, 1)$.

There are two possible CMOS implementations of the interface. For the first implemen-

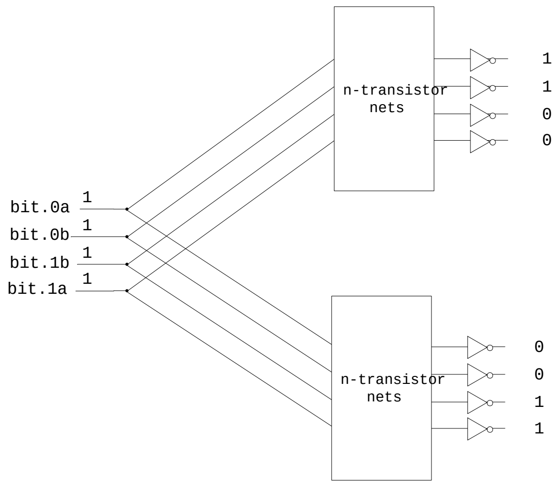


Figure 5.29: *Problematic Case of Invalid Bit Lines.*

tation, the inverted signals are introduced explicitly, as follows:

$$\neg\overline{bit.0} \rightarrow bit.0\uparrow$$
$$\overline{bit.0} \rightarrow bit.0\downarrow$$

$$\neg\overline{bit.1} \rightarrow bit.1\uparrow$$
$$\overline{bit.1} \rightarrow bit.1\downarrow$$

$$bit.0 \wedge \overline{bit.1} \wedge (\overline{bit.1_a} \vee \overline{bit.1_b}) \rightarrow \overline{bit.0_a}\downarrow$$
$$bit.0 \wedge \overline{bit.1} \rightarrow \overline{bit.0_b}\downarrow$$

$$bit.1 \wedge \overline{bit.0} \wedge (\overline{bit.0_a} \vee \overline{bit.0_b}) \rightarrow \overline{bit.1_a}\downarrow$$
$$bit.1 \wedge \overline{bit.0} \rightarrow \overline{bit.1_b}\downarrow$$

The inverted bit lines $\overline{bit.0}$ and $\overline{bit.1}$ come directly from the array. The circuit diagram of the interface with the mutual excluder between an output side of Hamming-coded memory and a DD circuit is shown in Figure 5.30. In CMOS implementation, an SRAM cannot charge the bit line up to *Vdd* because the signal comes out through n-type pass gates controlled by duplicated word lines. As a result, after an error flips a bit of the SRAM cell, a bit line holding *GND* may take on an intermediate voltage. In order to prevent this, a *keeper* is attached to the bit lines, which enables one bit line holding *GND* to drive another bit line up to *Vdd*.

Besides the implementation, there is a pass-gate implementation, which replaces a transistor in a series with the inverse of the signal of the transistor. That is, the literals $\neg\overline{bit.0}$

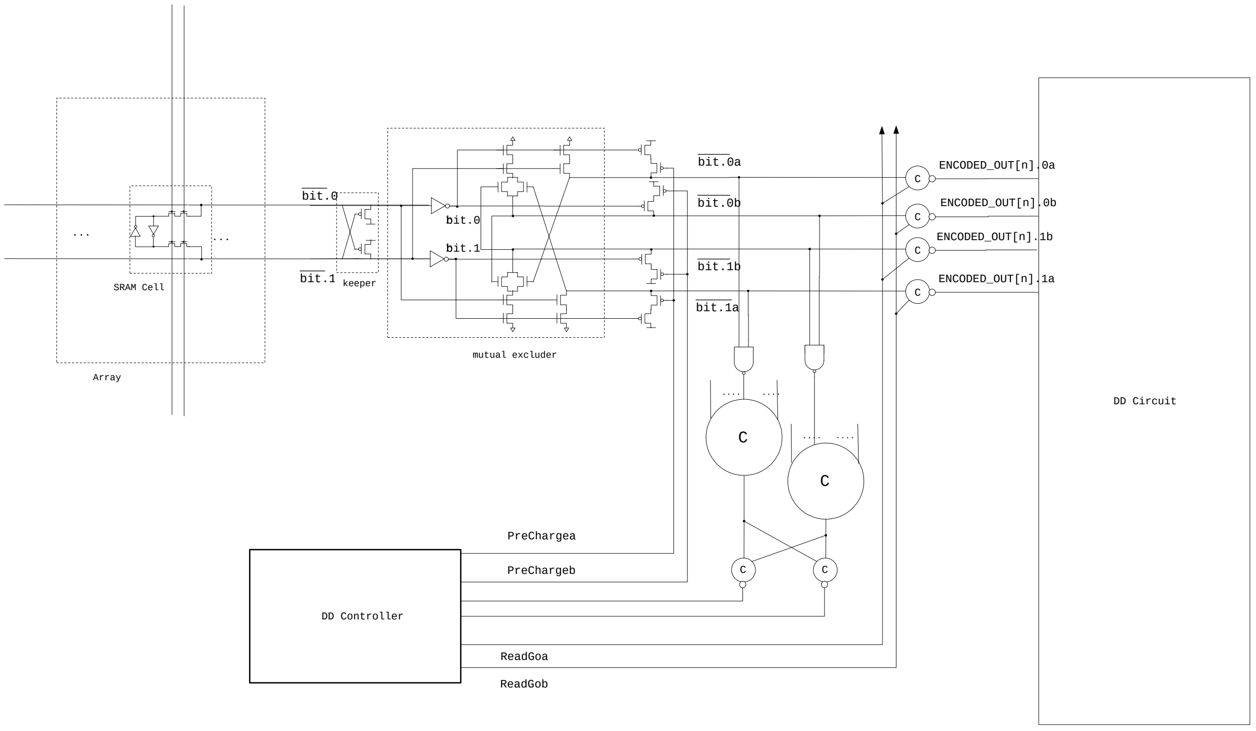


Figure 5.30: *Interface Between Output Side of Memory Unit and DD Circuit.*

and $\neg\overline{bit.1}$ are replaced with pass gates, as shown in Figure 5.31. Since the bit lines connected to the pass gates are also driven through pass gates in an SRAM cell, the inverters on the input sides of the mutual excluder are inserted in order to avoid pass gates driving other pass gates.

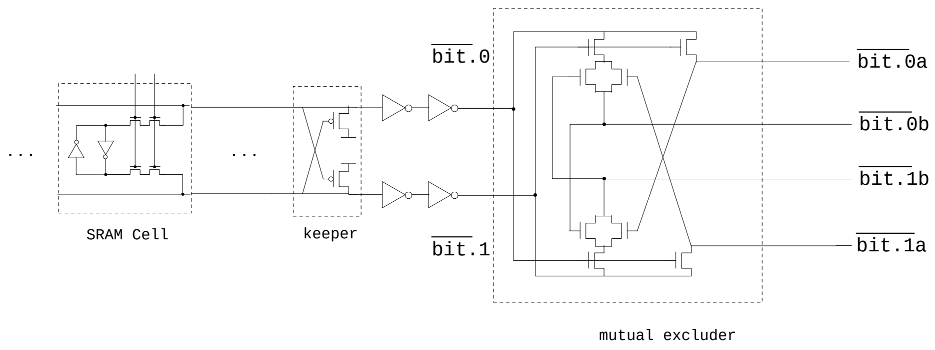
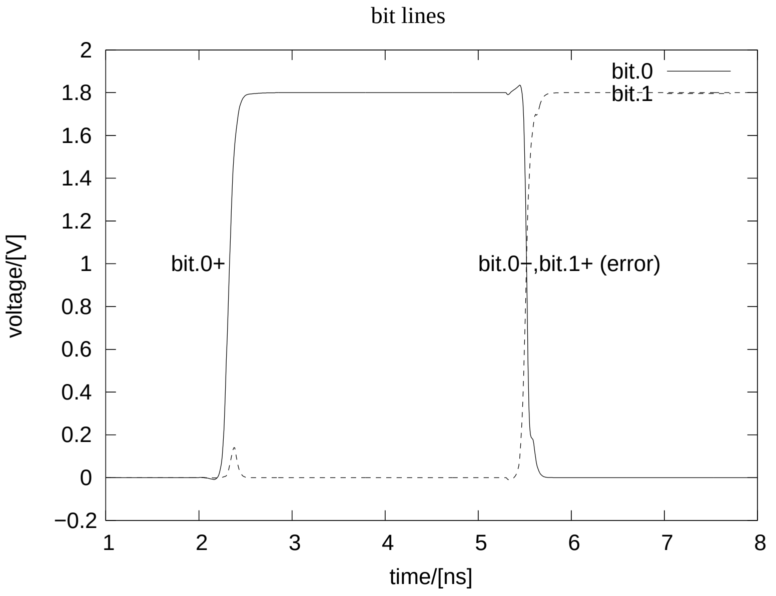


Figure 5.31: *Mutual Excluder Based on Pass-gate Transformation.*

We have simulated the two mutual excluders in TSMC 0.18-μm CMOS technology. Figure 5.32 and Figure 5.33 show the results of the SPICE simulations of the non-pass-gate mutual excluder and the pass-gate mutual excluder respectively. In these specific examples, charges have been injected into an SRAM cell around at about 5.3 ns after the bit lines are loaded at 2 ns, and the environment has intentionally kept the bit lines from being reset for illustration purpose. As we see, even after an error flips two bit lines $bit.0$ and $bit.1$, the duplicated bit lines $(bit.0_a, bit.0_b, bit.1_a, bit.1_b)$ do not become the invalid state $(1, 1, 1, 1)$, because of the mutual excluders.



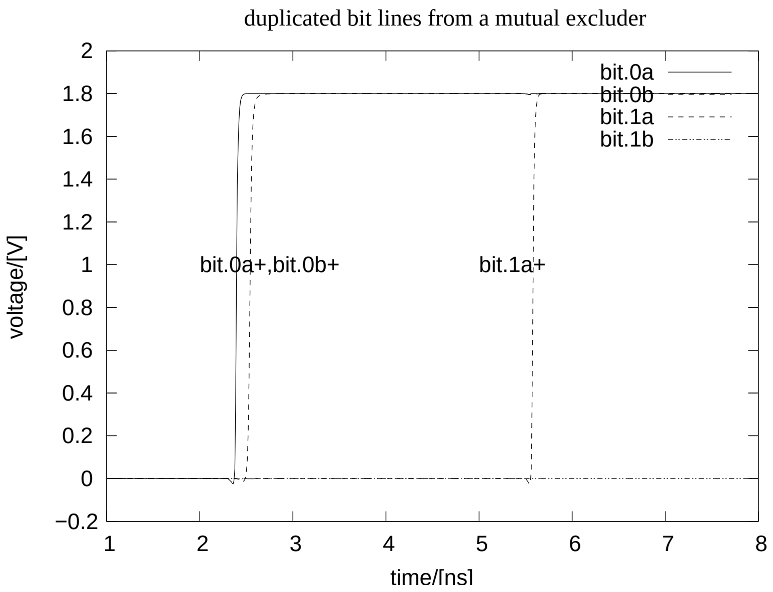


Figure 5.32: *Waveforms for Bit Lines and Duplicated Bit Lines from Mutual Excluder. Although two bit lines are flipped simultaneously, only one of the duplicated bit lines is flipped, owing to a mutual excluder.*

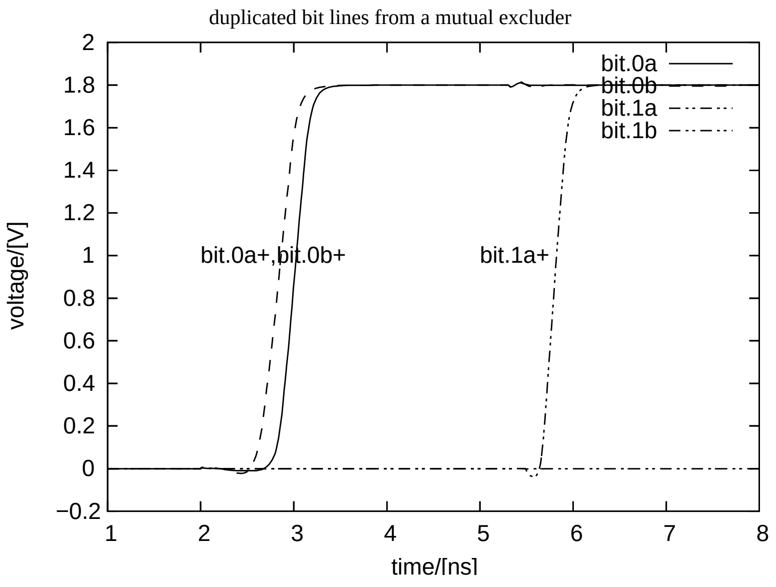


Figure 5.33: *Waveforms for Duplicated Bit Lines from Pass-gate Mutual Excluder. An error, which flips bit lines, occurs at 5.5 ns.*

5.4.4 Simulation Results of Error Tolerant Memory

A small memory unit (i.e., a register file), which has eight 32-bit registers, was designed with DICEs, and a big memory unit (i.e., a 512-byte instruction memory) was designed with the Hamming-coded array. In fact, as the number of rows in the array increases, the more advantageous a Hamming-coded memory is, since a Hamming-coded uses less redundancy, and the cost of an encoder and a decoder is fixed.

The soft-error tolerance has been tested in two different ways. One method was to inject errors randomly while the memory unit was running in a digital-level simulator. Then the result of the random-flipping simulation was compared with a normal simulation. Another method was to inject charges on randomly chosen nodes at every 4 ns in SPICE simulations.

For the first test, about five errors were injected in each cycle. Meanwhile, all enabled production rules were firing with random timing. Unless multiple errors occur together in one DICE of the register file, or on the same row of a Hamming-coded array, data from the memory units are read correctly.

For the second test, excess charges were injected to randomly chosen nodes every 4 ns in SPICE simulations. Initially, all data in the register file and the instructions memory were set to '0.' Figure 5.34 shows the waveforms of two corrupted nodes of the register file, first in a DICE and then in a DD control circuit. The corrupted nodes were restored quickly, and the errors do not affect the result of reading the register file: the environment could read the value of each register as '0' throughout the simulations.

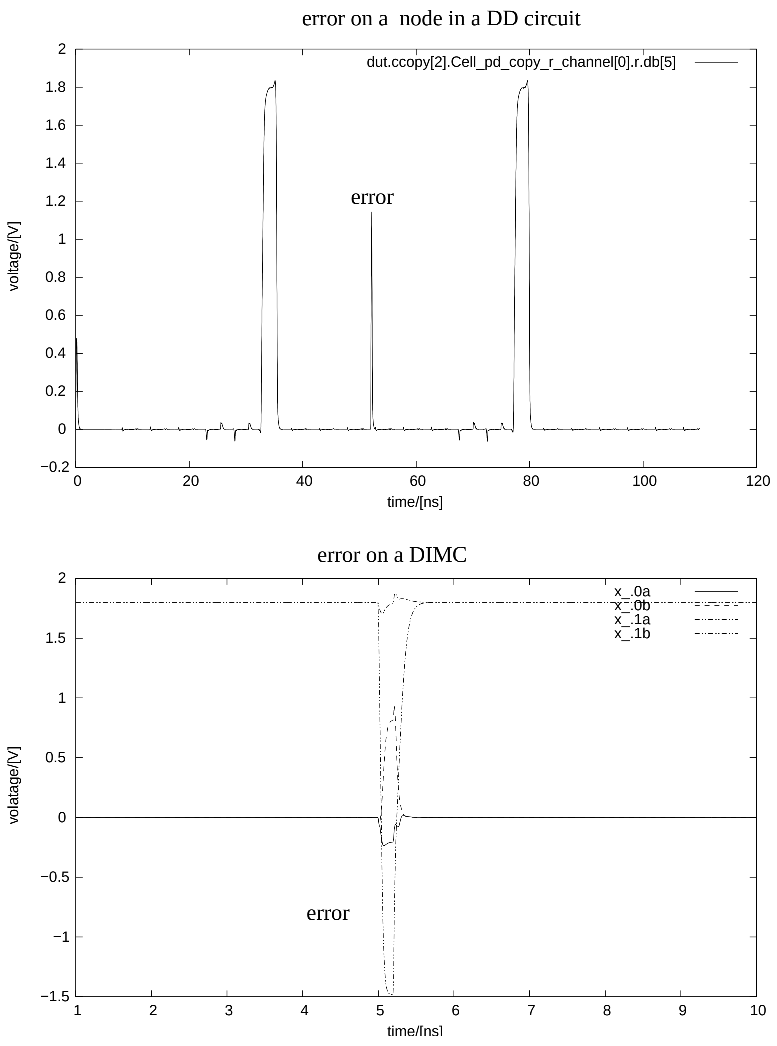
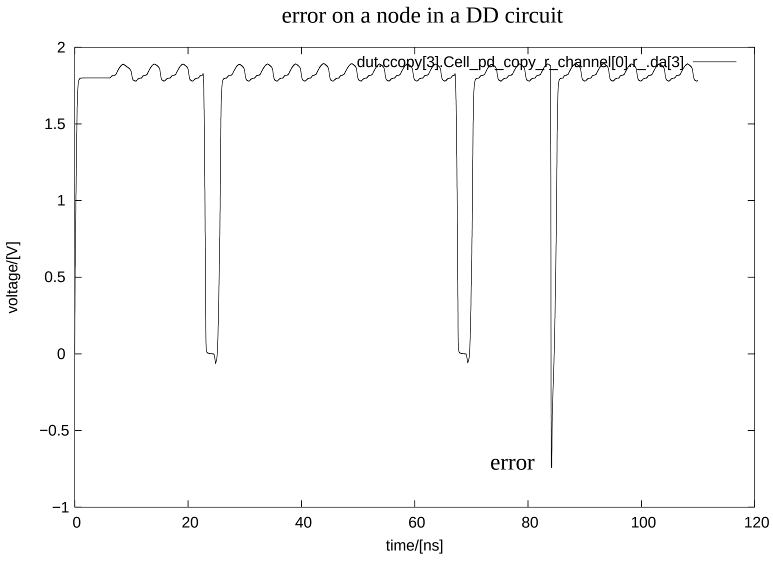


Figure 5.34: *Soft Errors on Register File Using DICE. An error in the regfile is corrected quickly.*

Figure 5.35 shows the waveforms of two corrupted nodes in the instruction memory. In the figure, an error on a DD control circuit occurs at 84 ns, and another error on an SRAM cell occurs at 88 ns. The node in a DD circuit is restored; the node in a SRAM cell remains corrupted, which is the weakness of a Hamming-coded memory unit, compared to a DICE memory unit. However, if at most one error occurs in each row of the array, the environment still can read instructions as '0.'



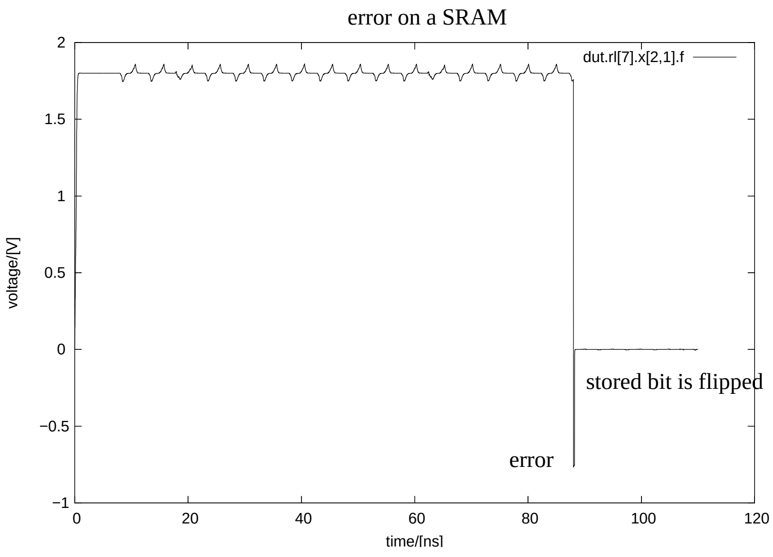


Figure 5.35: *Soft Errors on a Hamming-coded Memory. An error in a DD circuit is corrected quickly, but an error on an SRAM in a Hamming-coded array remains uncorrected.*

5.4.5 Summary

One error tolerant memory consists of DD auxiliary circuits and a DICE array. It has the feature of self-correcting, but it incurs large area overhead. A DICE array is larger by a factor of almost four than a corresponding SRAM array.

Another error tolerant memory employs Hamming codes. Although it requires an encoder and a decoder, the use of Hamming codes leads to an area-efficient array. Since the cost of an encoder and a decoder does not depend on the size of the array, a Hamming-coded array is more advantageous than a DICE array as the number of words in a data array increases. A Hamming code is applied to the array; the DD scheme is applied to auxiliary circuits around the array. Since the codings between auxiliary circuits and the array are different, it is necessary to convert a Hamming codeword into a duplicated codeword and vice versa. It is simple to convert a duplicated codeword to a Hamming codeword, but the conversion from a Hamming codeword to a duplicated codeword requires an interface circuit, which is similar to the mutual excluder of a DSET arbiter.

The soft-error tolerance of a memory unit with a DICE array and a memory unit with a Hamming-coded array have been verified in digital and analog simulations.

Chapter 6

QDI Circuits using Error Detecting Delay-insensitive Codes

We have devised the DD scheme to protect QDI circuits from soft errors. In the DD scheme, the data communications between CHP processes are encoded with repetition codes, which are a type of error correcting codes (ECCs). (For brevity, we use the term 'ECC' to refer to both error correcting codes and error detecting codes.) Instead of using repetition codes, we can use other ECCs, such as parity codes, in communication channels to protect a QDI circuit. Among various ECCs, which code is the best for designing a small and fast soft-error tolerant QDI circuit? To address the question, we shall design soft-error tolerant QDI circuits with different ECCs, and then compare them according to several metrics.

6.1 Error Detecting Delay-insensitive (EDDI) Code

In this section, a class of codes that can be used in asynchronous communications is defined. Then asynchronous versions of ECCs are defined, and applied to QDI circuits for error tolerance.

6.1.1 Definition of Delay-insensitive Codes

We start from the definition of a code. Let I be a finite set of indices of the boolean variables, and let C, called set of *codewords*, be a set of subsets of I. Each codeword represents a message (or data) for communications. A *code* is defined as a pair (I, C) where $|I|$ is the *length* of the code, and $|C|$ is the *size* of the code [42]. Elements of a codeword X in C correspond to the indices of the variables that are set to be `true`. For example, *one-of-two code* is defined as $(I, C) = (\{1, 2\}, \{C_1, C_2\})$, where $C_1 = \{1\}$ and $C_2 = \{2\}$; the two codewords are can be written as $c_1 = (0, 1)$ and $c_2 = (1, 0)$ in tuple representation. (A capital letter will be used for the set representation of a codeword, and a small letter will be used for the tuple representation of a codeword.)

A code (I, C) is called a *delay-insensitive* (DI) code when $|X - Y| > 0$ and $|Y - X| > 0$ are satisfied for all X, $Y \in C$. In other words, delay-insensitive codewords do not contain each other, which is called *no-containment property.* For example, a one-of-two code is a DI code, because a codeword $(1, 0)$ and another codeword $(0, 1)$ do not contain each other. What happens if a codeword contains another codeword? Let us assume that two codewords $(1, 1, 1)$ and $(1, 0, 1)$ are used for data communication in a QDI circuit. If a sender sets a codeword $(1, 0, 1)$ on a channel, then the receiver without timing information cannot tell whether the sender sent the codeword $(1, 0, 1)$, or it is in the middle of sending the codeword $(1, 1, 1)$. To avoid the confusion, the no-containment property is necessary, which guarantees a correct communication even without the notion of time. Widely-used codes in QDI circuits are the concatenation of one-of-two codes or one-of-four codes, which have four codewords: $(0, 0, 0, 1)$, $(0, 0, 1, 0)$, $(0, 1, 0, 0)$, $(1, 0, 0, 0)$. For brevity, $(0, 0..., 0)$, which corresponds to $\emptyset$ in the set-representation, is called the *spacer*, and the subsets of a valid codeword X are called intermediate codewords of X. The remaining subsets of I are called invalid codewords. For example, in one-of-two code, $(0, 0)$ is the spacer, $(1, 0)$ and $(0, 1)$ are valid codewords, and $(1, 1)$ is an invalid codeword.

A DI code does not use any extra information, such as a clock signal to communicate messages. Since the timing information should be encoded into the codewords themselves, a DI code is inefficient at utilizing code variables. While the n variables of a binary code encode 2^n messages, the n variables of $\frac{n}{2}$ one-of-two codes encode $2^{\frac{n}{2}}$ messages. It was shown that the maximum number of messages which a DI code can encode, given a length of code n, is $\binom{n}{\lfloor \frac{n}{2} \rfloor} < 2^n$ [43]. A DI code that encodes the maximum amount of messages within the lenth of the code given, is called an optimal DI code; it is hard to find in practice, because of the complexity of the corresponding decoding/encoding circuit.

6.1.2 Definition of Error Detecting Delay-insensitive Codes

In a QDI circuit, the steps of communication using a DI code are usually specified by the four-phase protocol. If two variables $r.0$, $r.1$ encode a one-of-two code, and the variables with an acknowledgment variable $r.e$ implement a channel R between a sender process and a receiver process, then the steps of a communication between two processes through the channel R can be described as follows:

1. The sender process sets a DI codeword on the channel variables $(r.0, r.1)$; the sender assigns $(1, 0)$ (or $(0, 1)$) to the variables.

2. The receiver process acknowledges the codeword by setting the acknowledgment variable $r.e$.

3. The sender resets the spacer on the channel variables; $(r.0, r.1)$ becomes $(0, 0)$.

4. The receiver resets the acknowledgment variable $r.e$.

If an error occurs on one of the code variables of a channel, an unintended message can be communicated. For example, an error on $r.0$ can turn the spacer into $(r.0, r.1) = (1, 0)$, which is a codeword, and the accidental codeword can be acknowledged by the receiver. Although the sender did not initiate a communication, the receiver can observe an unexpected data communication, because of the error. If an error occurs on $r.0$ while the codeword $(0, 1)$ is sent, then the codeword $(0, 1)$ is turned into an invalid codeword $(1, 1)$, and the receiver may interpret the invalid codeword $(1, 1)$ as the codeword $(1, 0)$. That is, an error can cause the receiver to get the wrong message. These erroneous communications can happen if the spacer and a codeword X are too close ($|X| \leq 1$), or if two codewords X and Y are too close to each other ($|X - Y| = |X \cap Y^c| \leq 1$ or $|Y - X| = |Y \cap X^c| \leq 1$).

In order to avoid the erroneous communications, an error detecting DI (EDDI) code is defined in such a way that, even if some of the code variables are corrupted, the no-containment property still holds. A code C is a *one-error EDDI code* if and only if $|X| > 1$, $|X - Y| > 1$ for all X, $Y \in C$. The definition implies the following:

- $|X| > 1$ ensures that an error cannot cause an unexpected communication.

- $|X - Y| > 1$ ensures that an error cannot turn a valid message X into another valid message Y.

In the following sections, we shall examine error tolerant QDI circuits based on different one-error EDDI codes. (The duplicated DI codes that are used in the DD scheme, are also one-error EDDI codes.) In the same manner, a code C is an *r-error* EDDI code if and only if $|X - Y| > r$ for all X, $Y \in C$; r-error tolerant QDI circuits use r-error EDDI codes for channel communications.

6.1.3 Error Detecting Delay-insensitive Code Based on Linear Code

Methods for constructing EDDI codes from non-linear ECCs have been studied [44, 45, 46]. But the decoding/encoding circuits for non-linear EDDI codes are complex, so they are not widely used in computing logic.

On the other hand, EDDI codes from linear codes are easily analyzed and implemented. In addition to that, the coding densities of EDDI codes from linear codes (e.g., repetition code and parity code) are comparable to optimal codes, especially in the range of a small number of bits, as shown in Figure 6.1, where the upper bound is decided by the following theorem. Hence, we shall focus on EDDI codes from linear codes from now on.

Theorem 2 *If C is an r-error detecting DI code, then*

$$|C| \leq \frac{\begin{pmatrix} n+r \\ \lfloor \frac{n+r}{2} \rfloor \end{pmatrix}}{\begin{pmatrix} n+r \\ r \end{pmatrix}} \tag{6.1}$$

holds.

Proof: See Appendix C. ■

A code is an $[n, k]$ *linear code* if and only if codewords $x = (x_1, x_2, \ldots, x_n)$ of length n, where x_i belongs to a field F, comprise a k-dimensional subspace of the n-dimensional vector over the field F [47]. (A field is an algebraic structure in which the operation of addition, subtraction, multiplication, and division are defined.) If the Hamming distance between every pair of codewords in an $[n, k]$ code is at least d_{min}, then the code can be also referred as a $[n, k, d_{min}]$ code.

An $[n, k]$ linear code can be completely described by any set of k linearly independent codewords. We can arrange the independent codewords of an $[n, k]$ linear code into $k \times n$ matrix G, called a *generator matrix*. A message $u = (u_1, \ldots, u_k)$ is mapped into a codeword

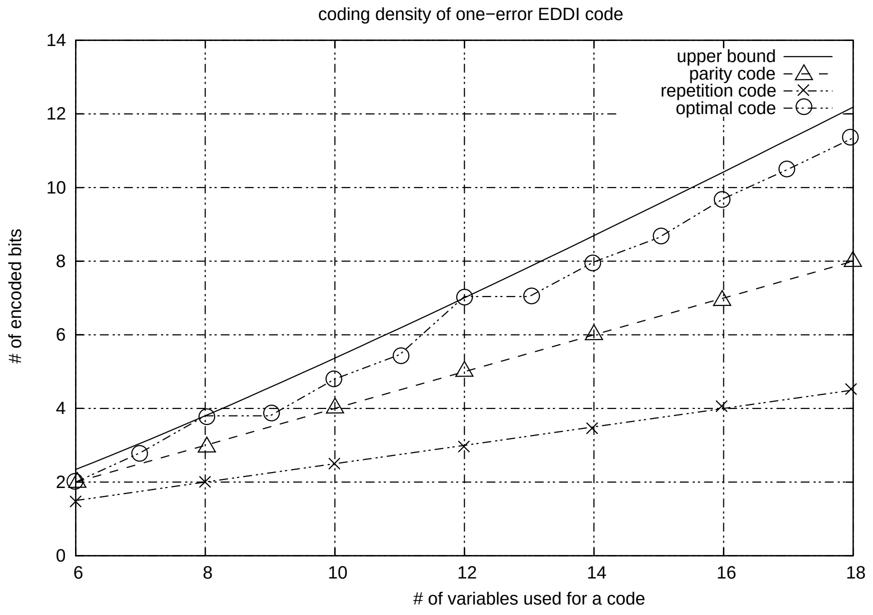


Figure 6.1: *One Error Detecting Delay-insensitive Codes and their Upper Bounds.*

$u \cdot G = (x_1, \ldots, x_n)$. For example, a 3×9 generator matrix for a three-repetition code is

$$G_1 = \begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}.$$

There is another useful matrix associated with a linear code, called a *parity-check matrix*. For every codeword x, a parity-check matrix H satisfies $H \cdot x^T = 0$. If a generator matrix G has the form $G = [I_k A]$, then a parity-check matrix H has the form $H = [-A^T I_{n-k}]$. For example, H_1 is a corresponding parity-check matrix of G_1:

$$H_1 = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

A parity-check matrix of a linear code takes an important role in constructing error tolerant QDI circuits, which will be explained in the following sections.

We can obtain an EDDI code by converting a linear code into a set of DI codes. For example, a binary linear code whose codewords consist of elements of $\{0, 1\}$ can be converted into a set of one-of-two codes; a linear code whose F is a non-binary field, is converted into a set of one-of-n codes, where n is the size of a field (i.e., $|F|$). The following theorem shows that we can obtain an EDDI code by the conversion.

Before proving the theorem, we first define the following notations. Let $[n]$ be $\{1, ..., n\}$. Let f be a mapping of the power set of $[n]$ onto the power set of $[2n]$: $2x \in f(X)$ iff $x \in X$, and $2x-1 \in f(X)$ iff $x \in [n]-X$. (The power set of S is the set of all subsets of S.) In other words, f maps a codeword of the ECC onto a codeword of the concatenation one-of-two codes.

Theorem 3 *A binary $[n, k, r+1]$ ECC, $([n], C)$ is given such that $|X - Y| \geq r+1$ or $|Y - X| \geq r+1$ for all $X \neq Y \in C$. The code $([2n], f(C))$ based on the mapping is an r-error EDDI.*

Proof: Let us assume that $([2n], f(C))$ is not an r-error EDDI code. Then,

$$\begin{aligned}
&\exists X, Y \in C \text{such that} X \neq Y,\ |f(X) - f(Y)| = k \leq r \\
&|f(X) \cap f(Y)| = n - k \text{ because } |f(X)| = |f(X) - f(Y)| + |f(X) \cap f(Y)| = n \\
&|f(X) \cap f(Y)| = |(X \cap Y) \cup (X^c \cap Y^c)| = n - k \text{ from the definition of } f, \\
&|(X \cap Y^c) \cup (X^c \cap Y)| = k \\
&|X \cap Y^c| = |X - Y| \leq k \leq r \text{ and } |X^c \cap Y| = |Y - X| \leq k \leq r
\end{aligned}$$

which contradicts the condition that $|X - Y| \geq r+1$ or $|Y - X| \geq r+1$ for all $X \neq Y \in C$. Thus, $|f(X) - f(Y)| > r$ and $|f(Y) - f(X)| > r$ are satisfied. According to the definition of an r-error EDDI code, the $([2n], f(C))$ code is also an r-error EDDI code. ■

Although we have proved the code consisting of one-of-two codes, it is easy to generalize the theorem for one-of-n codes; a DI code we obtain from a linear code by applying the mapping, is called a *linear DI code.*

6.2 Basic Component of QDI Circuit: Function Block and Completion Checker

Although several styles of buffers exist such as the PCHB shown in Figure 6.2, and the WCHB shown in Figure 6.3, most of the styles have two major components in common: a function block and a completion checker (CC) of variables of DI codes for delay-insensitive communications.

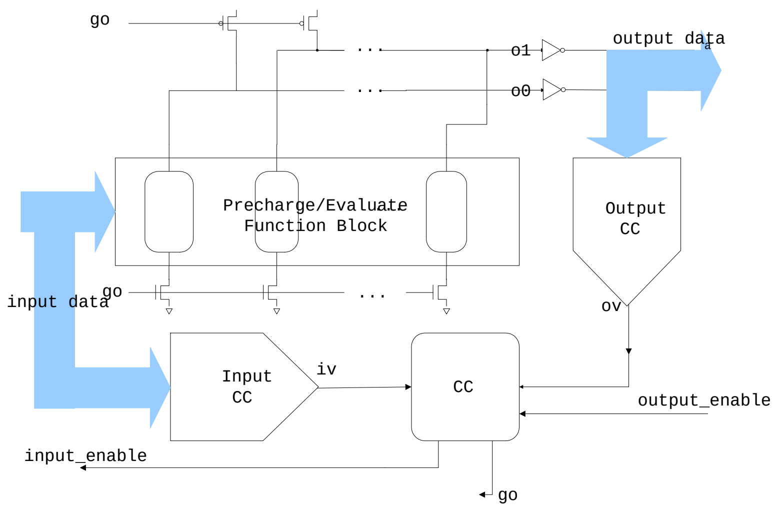


Figure 6.2: *Block Diagram of PCHB.*

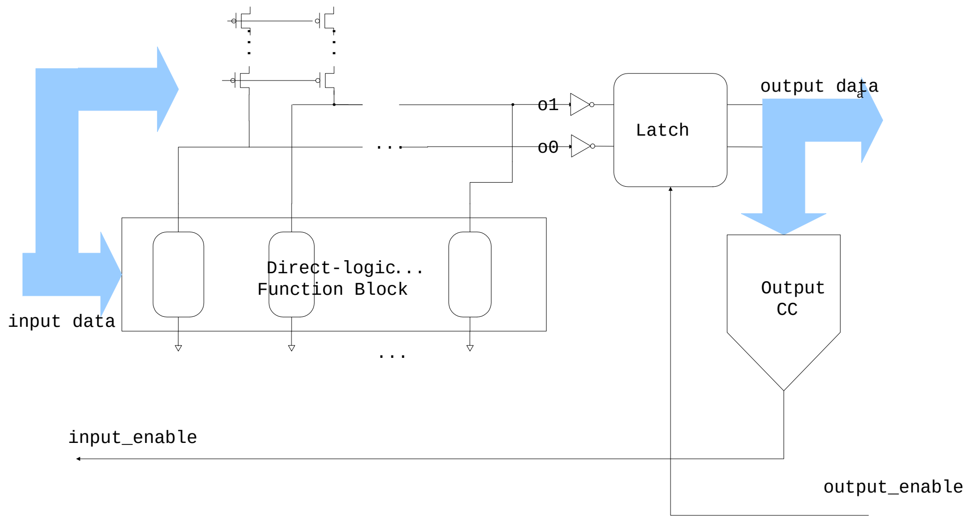


Figure 6.3: *Block Diagram of WCHB.*

A function block is a circuit to map a message of an input channel to a message of an output channel, which are encoded with DI codes. There are several styles of function blocks: delay-insensitive minterm logic (DIML), direct logics, precharging-evaluating logics, and so on [48, 49].

The DIML resembles the traditional sum-of-products (SOP) construction, but the minterms are formed using C-elements instead of AND gates, as shown in Figure 6.4 [50]. The C-elements guarantee that the outputs will not change until the transitions of all inputs are completed (i.e., the inputs read as either a codeword or the spacer). The direct logic can be obtained by merging inputs of the OR gate of the DIML function block, as shown in Figure 6.5 [51].

A problem of the direct logic is that it requires a long series chain of p-transistors, which is inefficient in CMOS technology. That is the reason why precharging-evaluating logics, as shown in Figure 6.6, are mostly used. This logic can be formed by replacing slow p-transistor nets with the control signals (e.g., *en*), which come from other circuits such as completion checkers of channels.

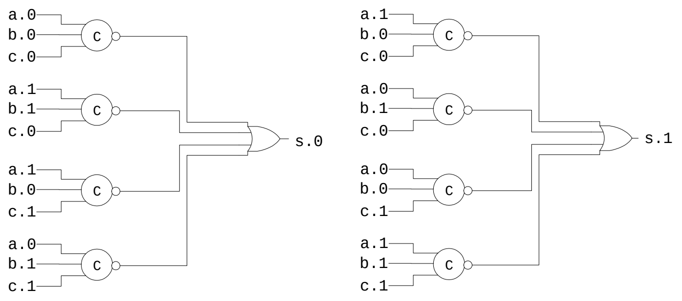


Figure 6.4: *Example of Delay-insensitive Minterm Logic (Adder).*

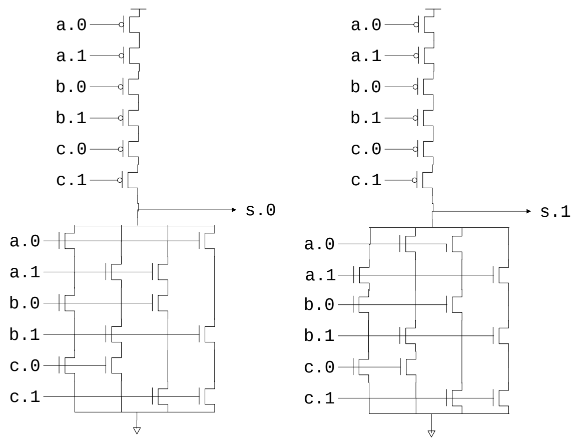


Figure 6.5: *Example of Direct Logic (Adder).*

The inputs of a CC are channel-code variables, and the output of the CC reflects whether

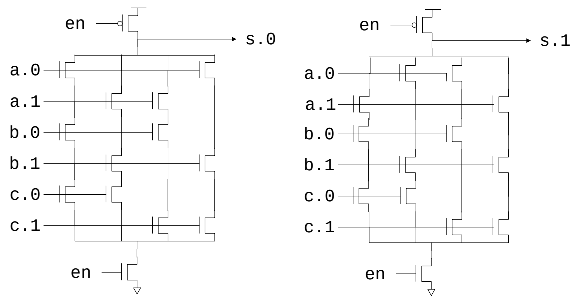


Figure 6.6: *Example of Precharging-evaluating Logic (Adder).*

the inputs read as a codeword or as the spacer. In other words, a CC checks whether a data communication on a channel is completed or not. There are several styles of CCs. (1) A design of CCs for various DI codes has been suggested in a form of a two-level circuit [52]. The design is similar to the DIML. (2) Enumeration-based CCs and comparison-based CCs have been proposed [53]. (3) A CC for important classes of DI codes, including m-of-n codes and the Berger codes, can be built in a systematic way by using a multi-output threshold circuit [54]. These constructions for general DI codes are not commonly used in practice, because of their complexity. Instead, we shall employ a simple CC for the concatenation of one-of-n codes, which will be shown in the following section.

A buffer is a good example to study, which is one of basic units of a QDI system. The buffer, $*[L?l; R!f(l)]$ in CHP, accepts an input from the channel L, computes the function f, and sends the result on the channel R. In the WCHB implementation of the buffer, a direct-logic function block and one CC are used. In the PCHB implementation, precharging-evaluating logics and two CCs are used: one CC for an input channel, and another CC for an output channel. The results of the CCs are combined through a C-element to provide the acknowledgment/control signals. (The way of combining the results depends on how to use channels such as conditional inputs, conditional outputs, and so on; the details are explained elsewhere [4]).

6.3 Error Tolerant Function Block for Linear DI Code

We show how to design an error tolerant function block based on a linear DI code. Then we shall compare the size of function blocks using various linear DI codes in order to find an efficient code for designing error tolerant function blocks. It will turn out that the family of repetition codes is generally advantageous.

6.3.1 Designing Error Tolerant Precharging-Evaluating Logic

In order to build an error tolerant function block, linear DI codes are used to encode input and output messages; an error tolerant function block must satisfy the following conditions:

Cond. E1 An error on the input of an error tolerant function block is *masked out*: while an invalid codeword is assigned to the inputs owing to an error, an error tolerant function block does not set an invalid codeword on the outputs.

Cond. E2 Once a codeword has been established on the primary output of an error tolerant function block, the codeword should be retained until it is acknowledged by the environment. Even if an error turns the codeword into an intermediate codeword, the intermediate codeword should be restored to the original codeword.

Cond. E1 ensures that an error is not propagated to the environment; **Cond. E2** is necessary to avoid deadlock. It is clear what happens in case that **Cond. E1** does not hold, but it is better to give an example in case that **Cond. E2** does not hold. Let us assume that currently the outputs of a function block hold a DI codeword of two one-of-two codes such as $(l_0.0, l_0.1, l_1.0, l_1.1) = (1, 0, 1, 0)$. If an error turns the codeword into the intermediate codeword $(0, 0, 1, 0)$, then the environment cannot acknowledge the intermediate codeword, and will wait indefinitely until the intermediate codeword is restored to the codeword. Therefore, **Cond. E2** is needed to avoid deadlock. An intermediate codeword caused by an error is called a *underflowed* codeword, in order to distinguish it from a normal intermediate codeword, which appears during transitions from the space to a codeword or vice versa.

We shall start to design an error tolerant function block from a normal function block step by step. A normal PRS of the precharging-evaluating function block of $*[L?l; R!l]$ is as follows:

$$\begin{array}{lcl} en \wedge l_1.0 & \rightarrow & r_1.0\uparrow \\ \neg en & \rightarrow & r_1.0\downarrow \\ en \wedge l_1.1 & \rightarrow & r_1.1\uparrow \\ \neg en & \rightarrow & r_1.1\downarrow \\ \ldots & & \end{array}$$

$$\begin{array}{lcl} en \wedge l_n.0 & \rightarrow & r_n.0\uparrow \\ \neg en & \rightarrow & r_n.0\downarrow \\ en \wedge l_n.1 & \rightarrow & r_n.1\uparrow \\ \neg en & \rightarrow & r_n.1\downarrow \end{array}$$

Here, $l_i.0$, $l_i.1$ implement a one-of-two code, which represents the i-th bit of a linear DI code of the channel L. Likewise $r_i.0$, $r_i.1$ represent the i-th bit of the channel R. The variable 'en' is a control signal, which indicates whether both input channel and output channel are ready to be used. **Cond E1** of error tolerance does not hold in this function block. For example, if an error causes $(l_i.0, l_i.1)$ to be $(1, 1)$, then the PRs of $r_i.0\uparrow$ and $r_i.1\uparrow$ are effective, and accordingly $(r_i.0, r_i.1)$ becomes an invalid codeword $(1, 1)$.

To solve the problem, we strengthen the function block to check whether an input codeword is corrupted or not. (To strengthen a PR is to multiply conjunctive factors to the guard of the PR.) For the strengthening, we use conditions of valid codewords, which are specified by a parity-check matrix in the function block: $(l_1, ..., l_n)$ is a codeword if and only if $H \cdot (l_1, ..., l_n)^T = 0$. For example, given a parity-check matrix $H = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}$ of the $[3, 2]$ parity code, a codeword (l_1, l_2, l_3) satisfies $H \cdot (l_1, l_2, l_3)^T = l_1 \oplus l_2 \oplus l_3 = 0$. which is written in the corresponding linear DI code, as follows:

$$(l_1.0 \wedge l_2.0 \wedge l_3.0) \vee (l_1.0 \wedge l_2.1 \wedge l_3.1) \vee (l_1.1 \wedge l_2.0 \wedge l_3.1) \vee (l_1.1 \wedge l_2.1 \wedge l_3.0)$$

If $l_1.0$ is `true` for a codeword, then $(l_2.0 \wedge l_3.0 \vee l_2.1 \wedge l_3.1)$ should evaluate to `true`; if $l_1.1$ is `true` for a codeword, then $(l_2.0 \wedge l_3.1 \vee l_2.1 \wedge l_3.0)$ should evaluate to `true`. Based on the conditions, we can strengthen the guards of $r_1.0\uparrow$ and $r_1.1\uparrow$, and then we obtain the following PRS:

$$\begin{array}{lcl} en_1 \wedge en_2 \wedge l_1.0 \wedge (l_2.0 \wedge l_3.0 \vee l_2.1 \wedge l_3.1) & \rightarrow & r_1.0\uparrow \\ en_1 \wedge en_2 \wedge l_1.1 \wedge (l_2.0 \wedge l_3.1 \vee l_2.1 \wedge l_3.0) & \rightarrow & r_1.1\uparrow \\ \neg en_1 \wedge \neg en_2 & \rightarrow & r_1.0\downarrow \\ \neg en_1 \wedge \neg en_2 & \rightarrow & r_1.1\downarrow \\ \ldots & & \end{array}$$

The strengthening prevents an invalid input from enabling PRs of the function block. Note that there are also two copies of a control signal, en_1 and en_2, in order to avoid a control signal becoming a single point of failure.

Let us consider a function block for the $[4,2]$ repetition code, whose parity-check matrix is

$$H = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}.$$

A valid codeword satisfies that $l_1 \oplus l_3 = 0$ and $l_2 \oplus l_4 = 0$, which can be written in a corresponding linear DI code, as follows:

$$\begin{gathered} (l_1.0 \wedge l_3.0) \vee (l_1.1 \wedge l_3.1) \\ (l_2.0 \wedge l_4.0) \vee (l_2.1 \wedge l_4.1) \end{gathered}$$

Compared with the function block for the parity $[3,2]$ code, the strengthened function block for the repetition $[4,2]$ code is as follows:

$$\begin{aligned} en \wedge l_1.0 \to r_1.0\uparrow \quad & \overset{repetition\ [4,2]}{\to} \quad en_1 \wedge en_2 \wedge l_1.0 \wedge l_3.0 \to r_1.0\uparrow \\ & \overset{parity\ [3,2]}{\to} \quad en_1 \wedge en_2 \wedge l_1.0 \wedge (l_2.0 \wedge l_3.0 \vee l_2.1 \wedge l_3.1) \to r_1.0\uparrow. \end{aligned}$$

The function block for the repetition code has fewer literals than that of the parity code. To check whether a variable such as $l_1.0$ of the repetition code is corrupted or not, it is enough to check just one counterpart variable $l_3.0$. On the other hand, in the case of a parity code, it is necessary to read more variables such as $l_2.0$, $l_3.0$, $l_2.1$, and $l_3.1$ together. Consequently, checking the parity code requires more literals (i.e., transistors).

In the same manner, we can strengthen the PRs of function blocks for general computations. Let us assume that the process $*[L?l; R!f(l)]$ uses a linear DI code encoding two-bit messages, and $f(l) = f(l_1, l_2) = l_1 \oplus l_2$ where l_1 and l_2 are the first and the second bit of the value l. A normal PRS of the precharging-evaluating function block is as follows:

$$\begin{aligned} en \wedge (l_1.0 \wedge l_2.0 \vee l_1.1 \wedge l_2.1) \ &\to\ r_1.0\uparrow \\ \neg en \ &\to\ r_1.0\downarrow \\ en \wedge (l_1.0 \wedge l_2.1 \vee l_1.1 \wedge l_2.0) \ &\to\ r_1.1\uparrow \\ \neg en \ &\to\ r_1.1\downarrow \\ \ldots \end{aligned}$$

By applying the conditions of the $[4,2]$ code and the $[3,2]$ code to the function block, we can obtain strengthened function blocks, as follows:

$$\begin{aligned}
&en \wedge (l_1.0 \wedge l_2.0 \vee l_1.1 \wedge l_2.1) \rightarrow r_1.0\uparrow \\
&\quad \overset{repetition}{\rightarrow} \quad en_1 \wedge en_2 \wedge (l_1.0 \wedge l_3.0 \wedge l_2.0 \wedge l_4.0 \vee l_1.1 \wedge l_3.1 \wedge l_2.1 \wedge l_4.1) \rightarrow r_1.0\uparrow \\
&\quad \overset{parity}{\rightarrow} \quad en_1 \wedge en_2 \wedge (l_1.0 \wedge l_2.0 \vee l_1.1 \wedge l_2.1) \wedge l_3.0 \rightarrow r_1.0\uparrow
\end{aligned}$$

As we see, the $[3,2]$ code leads to a smaller function block for this computation. The reason for this difference is that the function f computes the parity of a codeword so that the function block for the $[3,2]$ code can take advantage of the inherent structure of the parity code itself. However, there are not many functions that can exploit the structure of the parity code, which will be shown later.

Although the strengthened function blocks satisfy **Cond. E1**, the strengthened function blocks do not satisfy **Cond. E2**, since an error on the output is not corrected after the inputs are reset. One way of getting a strengthened function block to satisfy **Cond E2**, is to decompose a gate of a strengthened function block into two gates, and to add cross-coupled C-elements, as follows:

$$\begin{aligned}
&en_1 \wedge l_1.0 && \rightarrow w_1.0\uparrow \\
&\neg en_1 && \rightarrow w_1.0\downarrow \\
&en_2 \wedge (l_2.0 \wedge l_3.0 \vee l_2.1 \wedge l_3.1) && \rightarrow z_1.0\uparrow \\
&\neg en_2 && \rightarrow z_1.0\downarrow
\end{aligned}$$

$$\begin{aligned}
&w_1.0 \wedge z_1.0 && \rightarrow r_1.0\uparrow \\
&\neg w_1.0 \wedge \neg z_1.0 && \rightarrow r_1.0\downarrow \\
&\ldots
\end{aligned}$$

This is similar to what we did in the DD scheme. The shown PRS is an error tolerant function block of a buffer for the $[3,2]$ code. The conjunction of $l_1.0$ and $(l_2.0 \wedge l_3.0 \vee l_2.1 \wedge l_3.1)$ of a gate $r_1.0$ is separated, which can be viewed as the separation of computation (i.e., $l_1.0$) and parity checking (i.e., $(l_2.0 \wedge l_3.0 \vee l_2.1 \wedge l_3.1)$). (Some alternative ways of decomposing a function block exist. They are discussed in Appendix D.) The assignments of the primary outputs such as $r_i.0$ and $r_i.1$ are done only if both the output of the computation (i.e., $w_i.0$) and the output of the parity-checking (i.e., $z_i.0$) are completed.

Once a codeword is established on the primary outputs such as $r_i.0$, $r_i.1$, the values of the intermediate outputs such as $w_i.0$, $z_i.0$, $w_i.1$, and $z_i.1$ persist until the primary outputs are acknowledged by the receiver (i.e., en_1 and en_2 are reset). If an error occurs on the primary outputs, the corrupted primary output can be restored, because of the persistence of the intermediate outputs; if an error occurs on the intermediate outputs, then cross-coupled C-elements prevent the error from affecting the primary outputs. As a result, the function block can resolve the problem of a underflowed codeword. Error tolerant function blocks of $f(l) = l_1 \oplus l_2$ of the $[3, 2]$ code and the $[4, 2]$ code are shown in Figures 6.7 and 6.8. (Compared with the DD scheme, $r_i.0$, $r_i.1$ correspond to checked-out variables, and $w_i.0$, $w_i.1$, $z_i.0$, $z_i.1$ correspond to checked-in variables. See Chapter 5.)

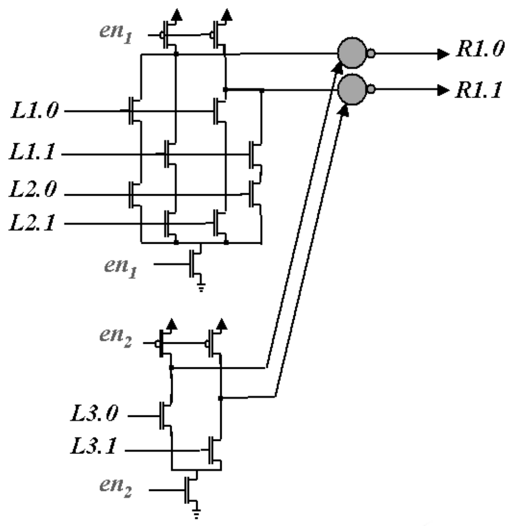


Figure 6.7: *Part of Function Block of Buffer for [3,2] Parity Code.*

6.3.2 Simplifying Based on Symmetry of Repetition Codes

We can simplify function blocks of repetition codes by exploiting the symmetry of repetition codes, which leads to the PRs of the computation and the PRs of the parity-checking being identical. Some of the identical gates are removed, and the outputs are shared, as shown in Figure 6.9.

Let us simplify an error tolerant function block of $f(l) = l_1 \oplus l_2$ for the $[4, 2]$ code:

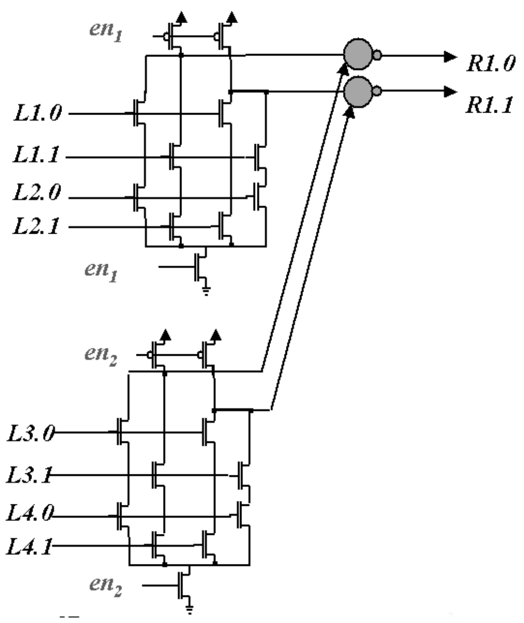


Figure 6.8: *Part of Function Block of Buffer for [4,2] Repetition Code.*

$$\begin{aligned}
en_1 \wedge (l_1.0 \wedge l_2.0 \vee l_1.1 \wedge l_2.1) &\rightarrow w_1.0\uparrow \\
\neg en_1 &\rightarrow w_1.0\downarrow \\
en_2 \wedge (l_3.0 \wedge l_4.0 \vee l_3.1 \wedge l_4.1) &\rightarrow z_1.0\uparrow \\
\neg en_2 &\rightarrow z_1.0\downarrow
\end{aligned}$$

$$\begin{aligned}
w_1.0 \wedge z_1.0 &\rightarrow r_1.0\uparrow \\
\neg w_1.0 \wedge \neg z_1.0 &\rightarrow r_1.0\downarrow
\end{aligned}$$

...

$$\begin{aligned}
en_1 \wedge (l_1.0 \wedge l_2.0 \vee l_1.1 \wedge l_2.1) &\rightarrow w_3.0\uparrow \\
\neg en_1 &\rightarrow w_3.0\downarrow \\
en_2 \wedge (l_3.0 \wedge l_4.0 \vee l_3.1 \wedge l_4.1) &\rightarrow z_3.0\uparrow \\
\neg en_2 &\rightarrow z_3.0\downarrow
\end{aligned}$$

$$\begin{aligned}
w_3.0 \wedge z_3.0 &\rightarrow r_3.0\uparrow \\
\neg w_3.0 \wedge \neg z_3.0 &\rightarrow r_3.0\downarrow
\end{aligned}$$

...

The gates of $w_1.0$ and $w_3.0$ are identical except for the outputs; the gates of $z_1.0$ and $z_3.0$ are identical except for the outputs, which allows sharing of the gates, as follows:

$$
\begin{array}{lcl}
en_1 \wedge (l_1.0 \wedge l_3.0 \vee l_1.1 \wedge l_3.1) & \rightarrow & w_1.0\uparrow \\
\neg en_1 & \rightarrow & w_1.0\downarrow \\
\\
w_1.0 \wedge z_1.0 & \rightarrow & r_1.0\uparrow \\
\neg w_1.0 \wedge \neg z_1.0 & \rightarrow & r_1.0\downarrow \\
\ldots \\
en_2 \wedge (l_2.0 \wedge l_4.0 \vee l_2.1 \wedge l_4.1) & \rightarrow & z_1.0\uparrow \\
\neg en_2 & \rightarrow & z_1.0\downarrow
\end{array}
$$

$$
\begin{array}{lcl}
w_1.0 \wedge z_1.0 & \rightarrow & r_3.0\uparrow \\
\neg w_1.0 \wedge \neg z_1.0 & \rightarrow & r_3.0\downarrow \\
\ldots
\end{array}
$$

The corresponding circuit diagram is shown in Figure 6.10. This function block is exactly the same as a function block based on the DD scheme. The symmetry of repetition codes allows us to save almost half the transistors of the function block, which makes repetition codes advantageous in designing error tolerant circuits, compared with other types of linear DI codes.

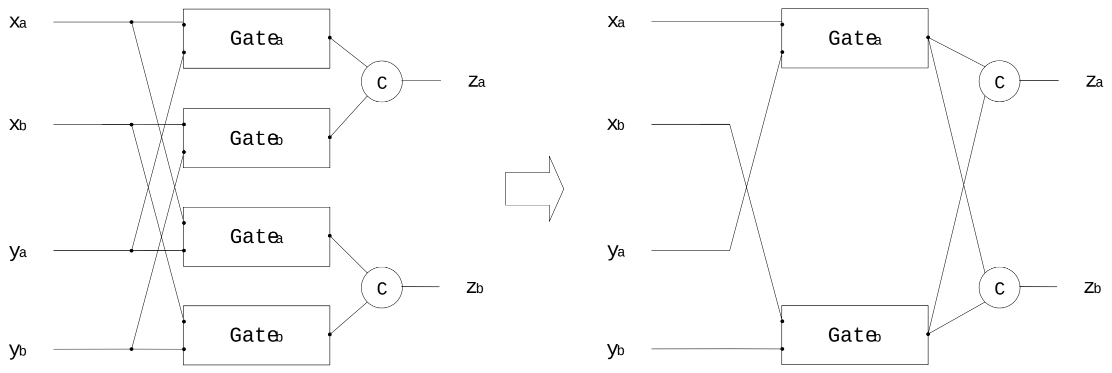


Figure 6.9: *Function Block for Repetition Code and its Simplified Function Block. Redundant gates of an original function block are removed.*

6.3.3 Comparing Size of Function Blocks for Different Linear DI Codes

The size of a function block, specifically the number of transistors, depends on design style. For example, the size of a DIML function block is proportional to the size of the code,

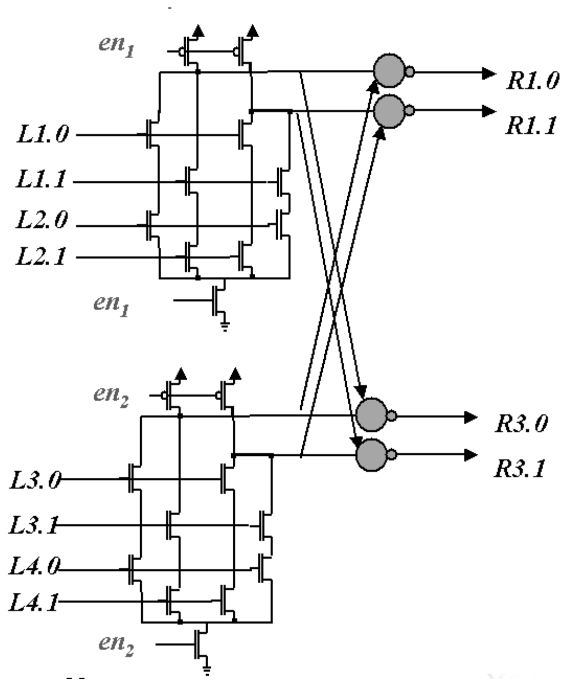


Figure 6.10: *Simplified Function Block for Repetition Code. The DD scheme, which duplicates a gate and adds cross-coupled C-elements, leads to the same design.*

regardless of the function it computes. On the other hand, the size of a direct-logic function block and the size of a precharging-evaluating function block depend on the minimization of the boolean formula of the function f. However, it is hard to get the smallest representation of a given function because minimizing a boolean formula is a coNP-hard problem [55]. In order to compare the cost of function blocks employing different linear DI codes, we resort to existing heuristic logic-minimization techniques for minimizing the size of a precharging-evaluating function block [56] [57].

A repetition code is advantageous in implementing a function (e.g., an identity function), but a parity code is advantageous in implementing another function (e.g., a parity function). If so, which code is efficient for designing a function block? Let us first answer the question for the two-bit process $*[L?l; R!f(l)]$. There are only two types of linear DI codes encoding two-bit messages, which are the $[3,2]$ parity code and the $[4,2]$ repetition code; there are $(2^2)^{2^2} = 256$ different functions that the process can implement, since there exist 2^2 input messages and 2^2 output messages. Table 6.1 shows the distribution of the number of transistors in two-bit function blocks. Out of 256 functions, only eight function blocks of the repetition code are larger than the function blocks of the parity code. (The

Table 6.1: Size of Two-bit Input/Ouput Precharging-evaluating Function Block

# of function blocks	the size of a function block using the [4,2] code	the size of a function block using the [3,2] code
4	32	24
16	36	48
40	40	48
8	40	60
64	44	60
24	48	48
48	48	60
16	52	60
32	56	60
4	64	48

eight cases are relevant to computing the parity of the inputs.)

There are more possibilities of encoding as the number of bits in a message increases. For example, a four-bit message can be encoded by five different types of linear DI codes, as shown in Figure 6.11, where the first one is one-bit parity code and the last one is a repetition code. Moreover the number of possible computable functions grows so rapidly that it is practically impossible to enumerate all possibilities for comparison. There are $(2^4)^{2^4} \approx 2 \times 10^{19}$ possible functions for a four-bit message. Instead of enumerating all functions for comparison, functions can be generated randomly, and the sizes of minimized function blocks using different linear DI codes are compared. The size distributions of three-bit, four-bit, and five-bit function blocks for different linear DI codes are shown in Figures 6.12, 6.13, 6.14. In the figures, the x-coordinate corresponds to the size of a function block using the repetition code and the y-coordinate corresponds to the size of a function block using a non-repetition code. The dotted line indicates the boundary where the size of a function block using the repetition code is the same as that of a function block using a non-repetition code. While we generated 1,000 functions randomly, we can see that corresponding function blocks for repetition codes are generally smaller than those for non-repetition codes. As the number of bits increases in a message, the advantage of repetition codes is more apparent, since the number of literals in the guards for parity-checking increases exponentially.

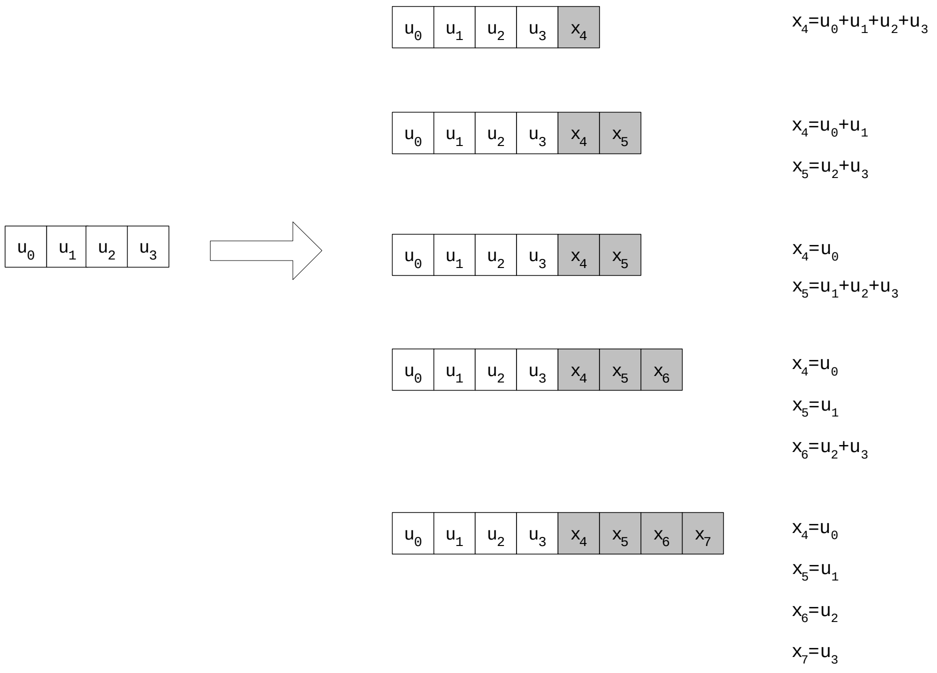

u_0 u_1 u_2 u_3
u_0 u_1 u_2 u_3 x_4
$x_4=u_0+u_1+u_2+u_3$
u_0 u_1 u_2 u_3 x_4 x_5
$x_4=u_0+u_1$
$x_5=u_2+u_3$
u_0 u_1 u_2 u_3 x_4 x_5
$x_4=u_0$
$x_5=u_1+u_2+u_3$
u_0 u_1 u_2 u_3 x_4 x_5 x_6
$x_4=u_0$
$x_5=u_1$
$x_6=u_2+u_3$
u_0 u_1 u_2 u_3 x_4 x_5 x_6 x_7
$x_4=u_0$
$x_5=u_1$
$x_6=u_2$
$x_7=u_3$

Figure 6.11: *One-error Detecting Linear Codes for Four-Bit Message.*

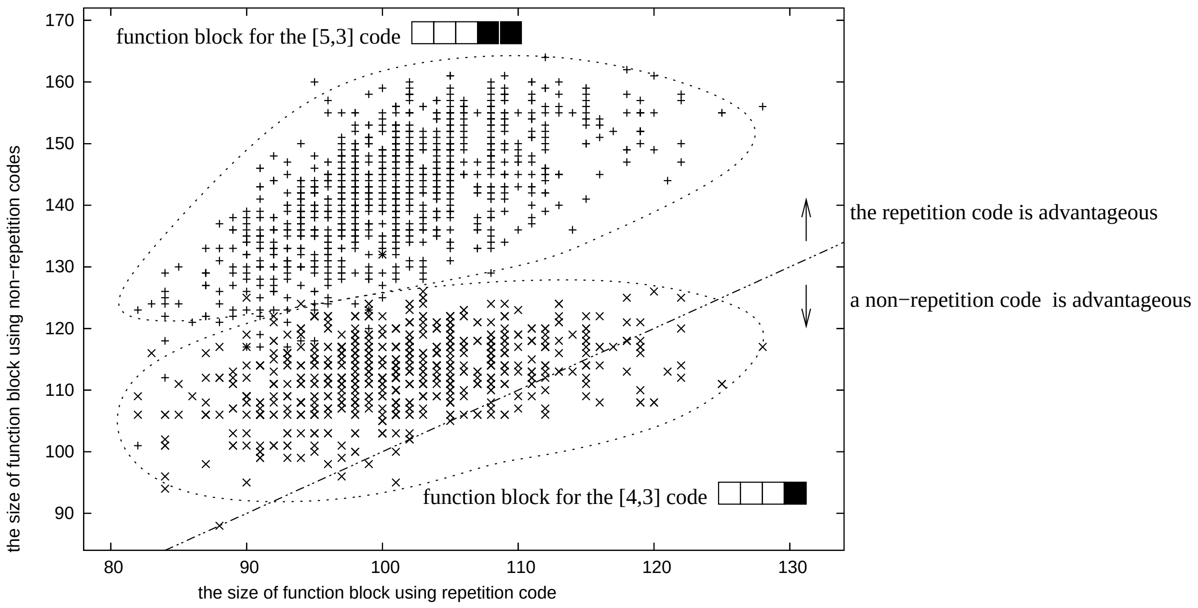


Figure 6.12: *Three-bit Function Blocks based on Repetition Codes vs Non-repetition Codes. The x-coordinate and the y-coordinate of each point correspond to the size of a function block using the repetition code, and the size of a function block using a non-repetition code (e.g., the [4,3] code and the [5,3] code).*

6.3.4 Function Blocks Using r-error Linear DI Codes

In order to build an r-error tolerant function block, we can use the same approach as we used to build one-error tolerant function blocks. That is, we strengthen a function block

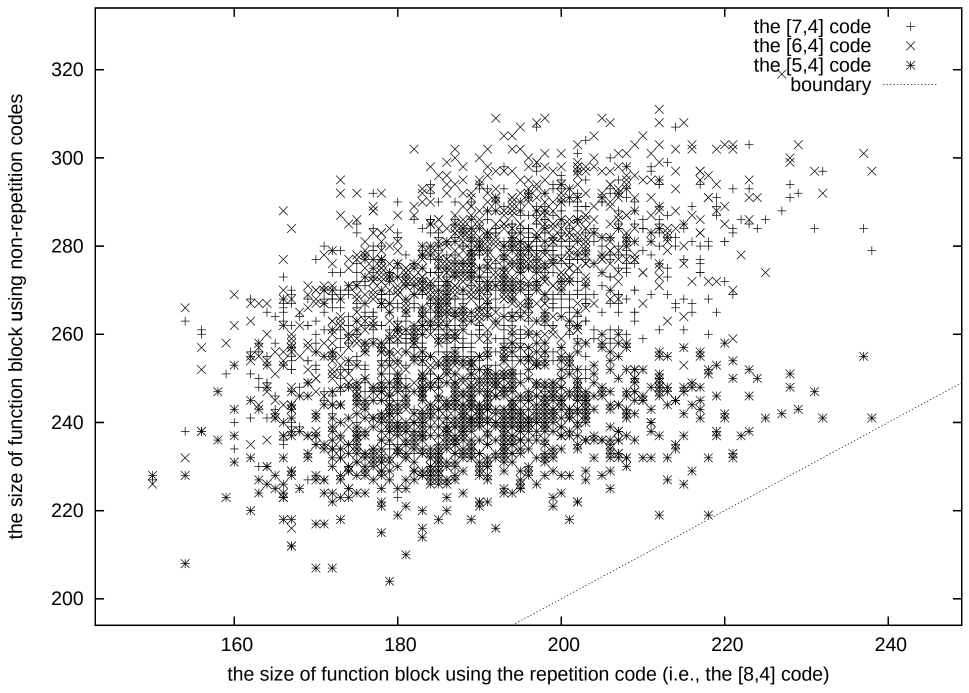


Figure 6.13: *Four-bit Function Blocks based on the Repetition Codes vs Non-repetition Codes.*

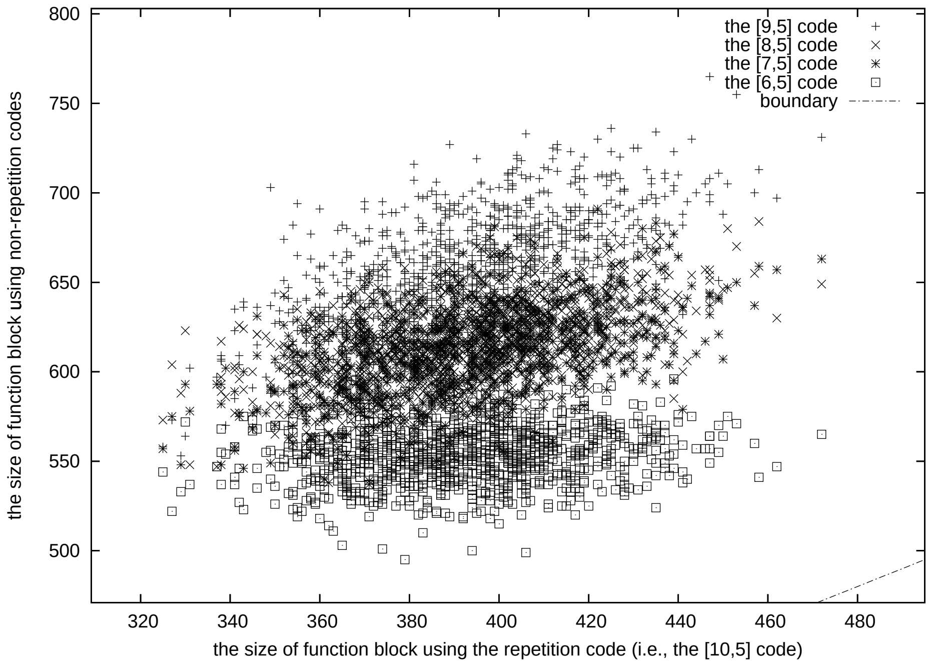


Figure 6.14: *Five-bit Function Blocks based on the Repetition Codes vs Non-repetition Codes.*

with conditions specified by a parity-check matrix, and add cross-coupled C-elements. For example, a parity-check matrix of the $[7, 4]$ Hamming DI code, which can detect two errors, is

$$H = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}.$$

The following always holds for a codeword $(l_1, l_2, ..., l_7)$:

$$\begin{aligned} l_2 \oplus l_3 \oplus l_4 \oplus l_5 = 0 \\ l_1 \oplus l_3 \oplus l_4 \oplus l_6 = 0 \\ l_1 \oplus l_2 \oplus l_4 \oplus l_7 = 0 \end{aligned}$$

Accordingly a two-error tolerant function block in $*[L?l; R!l]$, is as follows:

$$\begin{array}{lcl} en_1 \wedge l_1.0 \wedge p_f.0(l_3, l_4, l_6) & \rightarrow & x_1.0\uparrow \\ en_2 \wedge p_f.0(l_2, l_4, l_7) & \rightarrow & y_1.0\uparrow \\ en_3 \wedge p_f.0(l_2, l_3, l_4, l_5) & \rightarrow & z_1.0\uparrow \\ x_1.0 \wedge y_1.0 \wedge z_1.0 & \rightarrow & r_1.0\uparrow \\ \ldots & & \\ & & \\ en_1 \wedge l_1.1 \wedge p_f.1(l_3, l_4, l_6) & \rightarrow & x_1.1\uparrow \\ en_2 \wedge p_f.1(l_2, l_4, l_7) & \rightarrow & y_1.1\uparrow \\ en_3 \wedge p_f.1(l_2, l_3, l_4, l_5) & \rightarrow & z_1.1\uparrow \\ x_1.1 \wedge y_1.1 \wedge z_1.1 & \rightarrow & r_1.1\uparrow \\ \ldots & & \end{array}$$

Note that a gate is decomposed into three gates, and three-input C-elements are used instead of two-input C-elements. In the shown PRS, $p_f.0$ (or $p_f.1$) is a short-hand notation of a boolean expression of even (or odd) parity of one-of-two codes. For example,

$$p_f.0(l_3, l_4, l_6) = (l_3.0 \wedge l_4.0 \wedge l_6.0) \vee \cdots \vee (l_3.1 \wedge l_4.1 \wedge l_6.0),$$

$$p_f.1(l_3, l_4, l_6) = (l_3.0 \wedge l_4.0 \wedge l_6.1) \vee \cdots \vee (l_3.1 \wedge l_4.1 \wedge l_6.1).$$

In this manner, we can design multiple-error tolerant function blocks of r-error EDDI codes.

6.4 Error Tolerant Completion Checker for Linear DI Code

A completion checker (CC) is a circuit whose inputs are the channel-code variables and whose output indicates whether the input variables read as a codeword or the spacer. For example, a CC for a one-of-n code is an n-input OR gate. A CC for the concatenation of one-of-n codes uses an n-input OR gate for each one-of-n code, and the outputs of the OR gates are combined by a C-element (or a tree of C-elements), as shown in Figure 6.15. In the PCHB implementation of $*[L?l; R!f(l)]$, there is one CC for the channel L, and one for the channel R. If the primary outputs of both CCs are set (reset), then the acknowledgment of the channel L is set (reset).

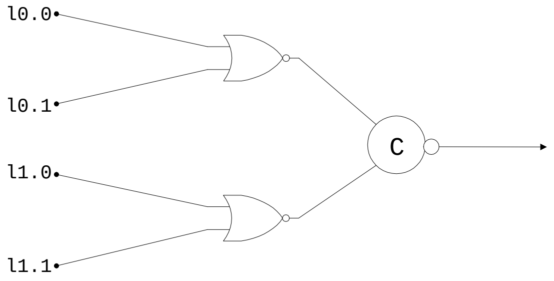


Figure 6.15: *CC for Concatenation of Two One-of-two Codes.*

A soft-error tolerant buffer using linear DI codes includes a CC for a linear DI code, which should tolerate an error. For error tolerance, the CC requires at least two primary outputs in order to avoid a single point of failure while the CC checks the validity of its primary inputs. Accordingly one possible design of an error tolerant CC is to duplicate a CC for a linear DI code and to add cross-coupled C-elements. For example, a PRS of a CC for the output channel R of the $[3, 2]$ parity code, whose outputs are V_a and V_b, is as follows:

$$\begin{array}{lcl}
r_1.0 \wedge r_2.0 \wedge r_3.0 & \rightarrow & v_a\uparrow \\
r_1.0 \wedge r_2.1 \wedge r_3.1 & \rightarrow & v_a\uparrow \\
r_1.1 \wedge r_2.0 \wedge r_3.1 & \rightarrow & v_a\uparrow \\
r_1.1 \wedge r_2.1 \wedge r_3.0 & \rightarrow & v_a\uparrow \\
\neg r_1.0 \wedge \neg r_2.0 \wedge \neg r_3.0 \wedge \neg r_1.1 \wedge \neg r_2.1 \wedge \neg r_3.1 & \rightarrow & v_a\downarrow
\end{array}$$

$$\begin{array}{lcl}
r_1.0 \wedge r_2.0 \wedge r_3.0 & \rightarrow & v_b\uparrow \\
r_1.0 \wedge r_2.1 \wedge r_3.1 & \rightarrow & v_b\uparrow \\
r_1.1 \wedge r_2.0 \wedge r_3.1 & \rightarrow & v_b\uparrow \\
r_1.1 \wedge r_2.1 \wedge r_3.0 & \rightarrow & v_b\uparrow \\
\neg r_1.0 \wedge \neg r_2.0 \wedge \neg r_3.0 \wedge \neg r_1.1 \wedge \neg r_2.1 \wedge \neg r_3.1 & \rightarrow & v_b\downarrow
\end{array}$$

$$\begin{array}{lcl}
v_a \wedge v_b & \rightarrow & V_a\uparrow \\
\neg v_a \wedge \neg v_b & \rightarrow & V_a\downarrow
\end{array}$$

$$\begin{array}{lcl}
v_a \wedge v_b & \rightarrow & V_b\uparrow \\
\neg v_a \wedge \neg v_b & \rightarrow & V_b\downarrow
\end{array}$$

It consists of two sub-CCs and two C-elements. The four PRs of $v_a\uparrow$ correspond to the four codewords of the $[3, 2]$ code, and the last PR of $v_a\downarrow$ corresponds to the spacer. The CC is soft-error tolerant as follows. If an error on a primary input turns a codeword into an invalid codeword, then the transitions of the primary outputs are disabled; if an error occurs on one of the sub-CCs (e.g., an error on v_a), then only one of input of the cross-coupled C-elements is affected, and the duplicated primary outputs are not affected. In both cases, the error is masked out and corrected by the CC.

The suggested CC, however, incurs a long series chain of transistors, which is inefficient in CMOS technology. In order to avoid the problem, we decompose the CC into a set of two-input parity checkers, which compute one-of-two XOR of two one-of-two inputs, as follows:

$$\begin{array}{lcl}
x1.0 \wedge x2.0 & \rightarrow & y0\uparrow \\
\neg x1.0 \wedge \neg x2.0 & \rightarrow & y0\downarrow \\
x1.1 \wedge x2.1 & \rightarrow & z0\uparrow \\
\neg x1.1 \wedge \neg x2.1 & \rightarrow & z0\downarrow
\end{array}$$

$$\begin{array}{lcl}
x1.0 \wedge x2.1 & \rightarrow & y1\uparrow \\
\neg x1.0 \wedge \neg x2.1 & \rightarrow & y1\downarrow \\
x1.1 \wedge x2.0 & \rightarrow & z1\uparrow \\
\neg x1.1 \wedge \neg x2.0 & \rightarrow & z1\downarrow
\end{array}$$

$$\begin{aligned}
y0 \vee z0 &\quad \rightarrow \quad parity.0\uparrow \\
\neg y0 \wedge \neg z0 &\quad \rightarrow \quad parity.0\downarrow
\end{aligned}$$

$$\begin{aligned}
y1 \vee z1 &\quad \rightarrow \quad parity.1\uparrow \\
\neg y1 \wedge \neg z1 &\quad \rightarrow \quad parity.1\downarrow
\end{aligned}$$

Its corresponding circuit is shown in Figure 6.16, and recursively we can compute XOR of n one-of-two inputs, as shown in Figure 6.17. If we decompose a sub-CC for the [3,2] code into parity checkers, we obtain the circuit, which can easily be synthesized in CMOS technology, as shown in Figure 6.18.

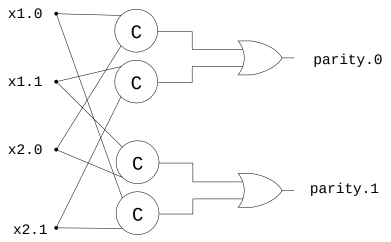


Figure 6.16: *Two-input Parity Checker.*

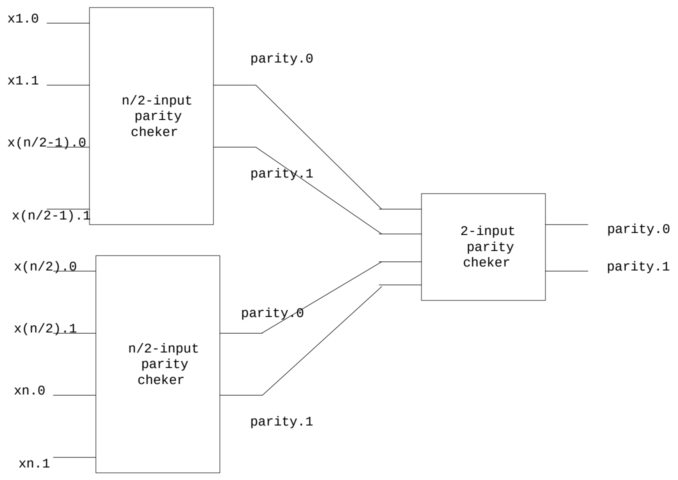


Figure 6.17: *Parity Decomposition.*

Two copies of a parity-check CC are used to construct a CC for a linear DI code in the previous design, but one of them can be replaced with a CC for the concatenation of one-of-n codes, as shown in Figure 6.19. The simplification seems to weaken the error tolerance, but the simplified design still can tolerate an error in the following manner. If an error occurs

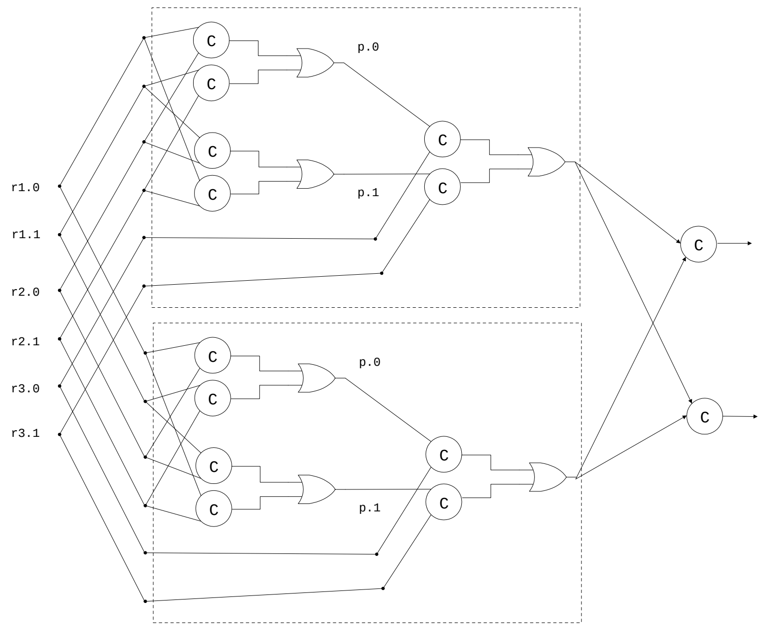


Figure 6.18: *CC for the [3,2] code.*

on one of the sub-CCs, then it affects only one input of the cross-coupled C-elements, so that the primary outputs are not affected. On the other hand, if an error occurs on the primary inputs, the normal sub-CC may set its output, but the parity-checker sub-CC does not. That is, an error on the primary inputs affects only one input of the cross-coupled C-elements, so that the primary outputs are not affected.

The CC for a repetition code can be simplified further by exploiting the symmetrical structure of the CC. The primary inputs are divided into two sub-CCs for the concatenation of two one-of-two codes; one sub-CC checks half of one-of-two codes, and the other sub-CC checks the remaining half. For example, a CC for the $[4, 2]$ code consists of two CCs for two one-of-two codes and cross-coupled C-elements, as shown in Figure 6.20. The size of the simplified CC for a repetition code is roughly as small as half the size of the CCs of non-repetition codes. However non-repetition codes do not have the symmetrical structure.

An r-error tolerant CC for an r-error tolerant linear DI code, consists of $r + 1$ sub-CCs: r copies of parity checkers given by a parity-check matrix, and one normal CC; the outputs of the sub-CCs are used as the inputs of $(r + 1)$-input C-elements.

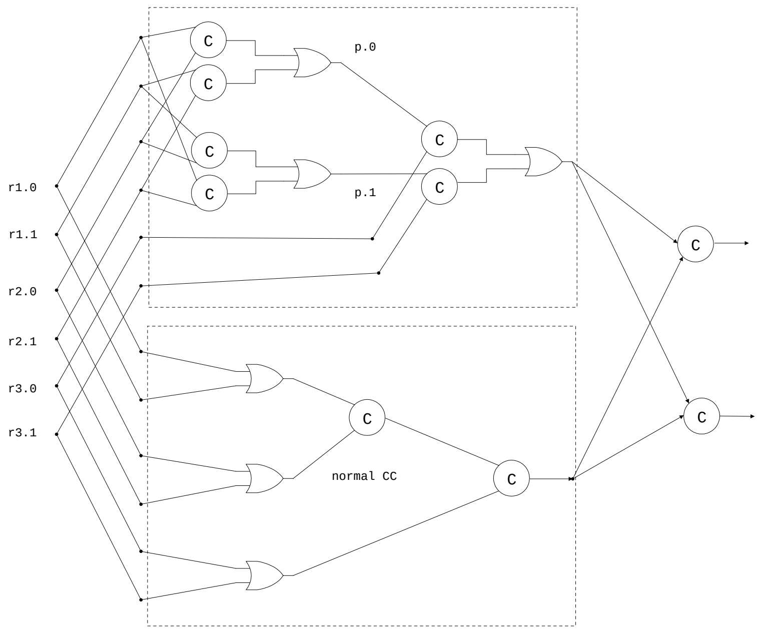


Figure 6.19: *Simplified CC for the [3,2] code.*

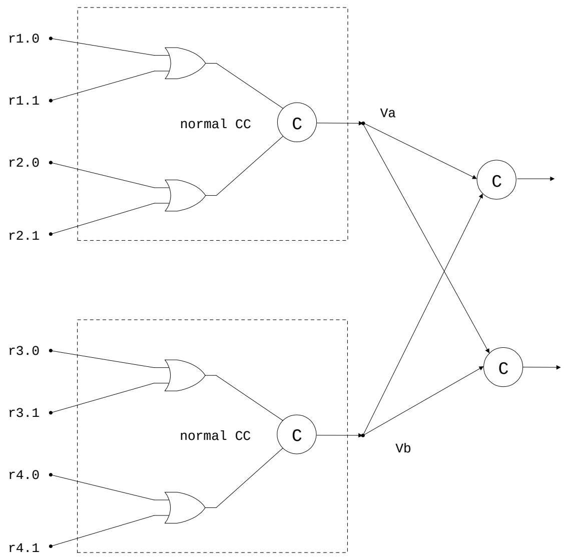


Figure 6.20: *CC for the [4,2] Code.*

6.5 Cost of Error Tolerant QDI Circuits for Different Linear DI Codes

There are several metrics by which to evaluate a circuit, such as the size (i.e., the number of transistors), the speed, the power, the latency, and so on. Here we use the size and the Et^2 metric to evaluate the circuits of the process $*[L?l; R!f(l)]$ of different linear DI codes.

6.5.1 Size

As shown in the previous section, the size of a CC for a repetition code, which can exploit the symmetrical structure of repetition codes, is as small as half the size of a CC for corresponding non-repetition codes; function blocks of repetition codes are generally smaller than those of non-repetition codes, even though there are a few exceptions. Figure 6.21 and Figure 6.22 show the distribution of the sizes of three-bit WCHB processes and the distribution of three-bit PCHB processes. A WCHB template consists of one direct-logic function block and one CC, and a PCHB template consists of one precharging-evaluating function block and two CCs. Since repetition codes are always favorable for CCs, the size difference between circuits for a repetition code and circuits for non-repetition codes is more apparent for the PCHB template, because the PCHB template requires two CCs. As the number of bits in a message increases, the size difference becomes more apparent, because a function block for non-repetition codes (e.g., a parity code) needs to examine more variables in order to ensure the validity of input variables than a function block for a repetition code.

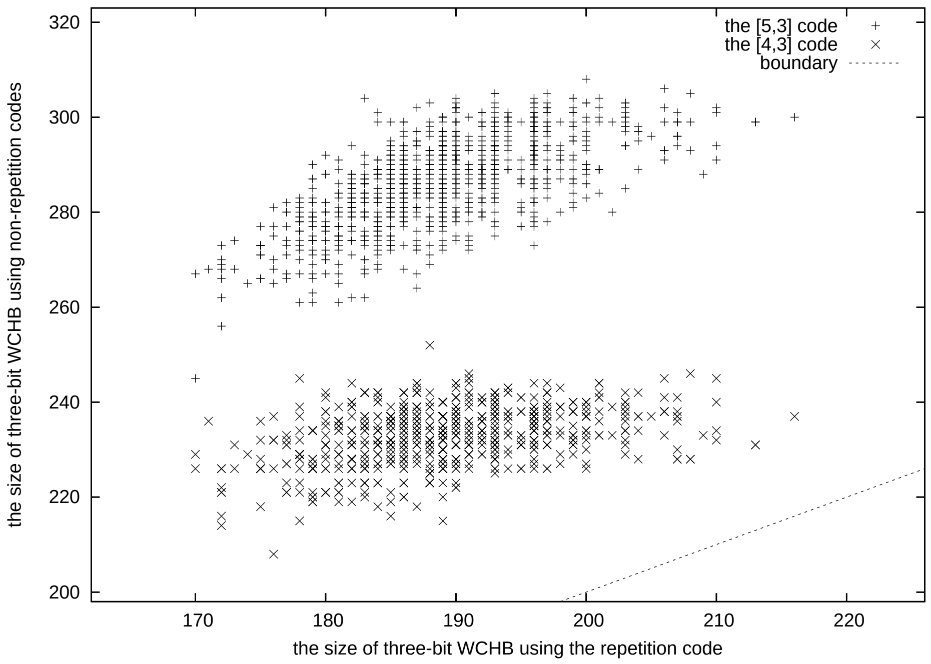


Figure 6.21: *Size Comparison of Three-bit WCHB Processes.*

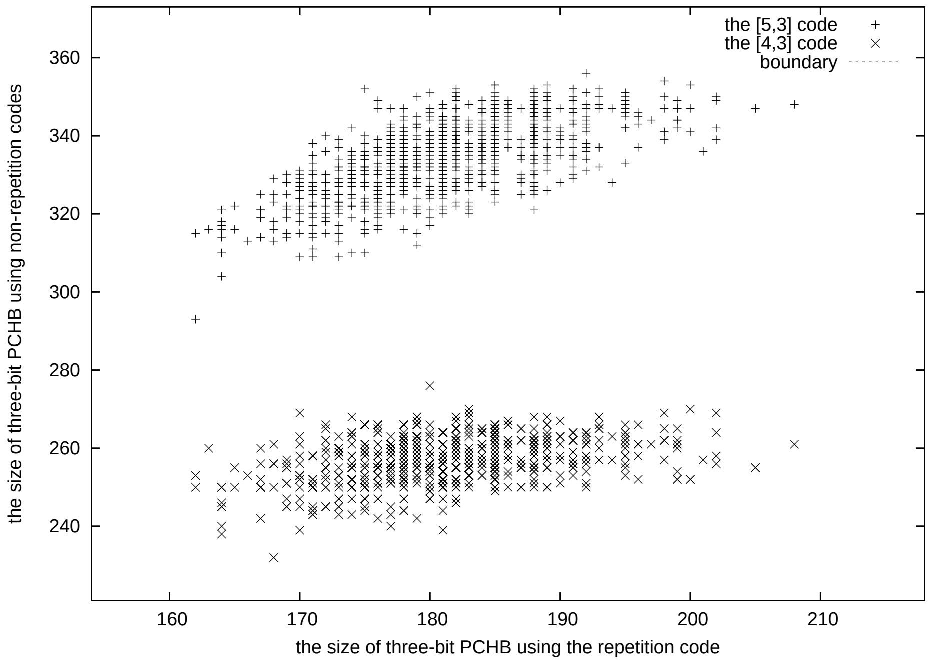


Figure 6.22: *Size Comparison of Three-bit PCHB Processes. The PCHB using a repetition code is always smaller than the PCHB using a non-repetition code, so that the boundary is located outside the range of the figure.*

6.5.2 Et^2

We shall evaluate circuits with other metrics, namely, energy and time. In CMOS technology, each transition of a boolean variable corresponds to charging or discharging the capacitor of the physical node corresponding to that variable by bringing its voltage either to the supply voltage *Vdd* or to the ground voltage *GND*. The energy consumed through the charging and the discharging of the capacitor determines the system energy, and the time spent to charge and discharge capacitors determines the system speed.

The energy, $E_{z\uparrow}$, spent during the execution of a PR $G \to z\uparrow$ is the energy dissipated as heat in the pull-up network during charging the capacitor C_z associated with the node of z, which is expressed as $E_{z\uparrow} = \frac{C_z\, Vdd^2}{2}$. Likewise we can have the same $E_{z\downarrow}$ for $G' \to z\downarrow$. The energy considers only dynamic energy during transitions, and for simplicity ignores energy consumption caused by leakage and short-circuit.

The delay $t_{z\uparrow}$ for charging the node z is the time it takes for the current $i_{z\uparrow}$ to carry the amount of charge, $Q_z = C_z V$, in the capacitor. We can approximate delay to be the ratio of the final charge Q_z on C_z to the current $i_{z\uparrow} \approx K_{z\uparrow} Vdd^2$ so that $t_{z\uparrow} = \frac{C_z}{K_{z\uparrow} V}$; similarly for the delay $t_{z\downarrow}$ and current $i_{z\downarrow}$. Combining the expressions for delay and for energy, it has been found that $E_z t_z^2$ of each transistor is independent of Vdd. Therefore Et^2 of a system has been proposed as a metric for tradeoff between energy and throughput of a computation in the normal range of operation [58].

A physical node corresponding to a variable in PRS is related to several capacitive components: the gate capacitance of transistors that the physical node drives, and the wiring capacitance source/drain diffusion capacitance of the transistors that drive the node. Additionally, there is the capacitance of internal nodes of transistor networks. Although all capacitive components should be included in calculating energy dissipation and delay, it was shown that the discrepancy between the simulation results of the simple charge/discharge model and of the SPICE model, is less than 8% in small circuits [59]. Even if the computation considers only the gate capacitance, which is called a fanout-weighted transition-counting model, the discrepancy is about 20%. Although some accuracy is lost, for simplicity we use the fanout-weighted transition-counting model to estimate the energy consumption. That is, it is assumed that C_z is proportional only to the load of z, (i.e., the fanout of z) since we are looking into a circuit of a small process like $*[L?l, R!l]$, where the wiring load is negligible.

Let us first estimate the energy consumption of the function blocks of a two-bit buffer $*[L?l; R!l]$. The CMOS-implementable PRS of the function block using the $[3, 2]$ code is as follows:

$$\begin{array}{ll}
en_1 \wedge r.e_1 \wedge l_1.0 & \to \_w_1.0\downarrow \\
en_2 \wedge r.e_2 \wedge (l_2.0 \wedge l_3.0 \vee l_2.1 \wedge l_3.1) & \to \_z_1.0\downarrow \\
\neg en_1 \wedge \neg r.e_1 & \to \_w_1.0\uparrow \\
\neg en_2 \wedge \neg r.e_2 & \to \_z_1.0\uparrow
\end{array}$$

$$\begin{array}{ll}
\neg\_w_1.0 \wedge \neg\_z_1.0 & \to r_1.0\uparrow \\
\_w_1.0 \wedge \_z_1.0 & \to r_1.0\downarrow
\end{array}$$

$$\begin{array}{ll}
en_1 \wedge r.e_1 \wedge l_1.1 & \to \_w_1.1\downarrow \\
en_2 \wedge r.e_2 \wedge (l_2.1 \wedge l_3.0 \vee l_2.0 \wedge l_3.1) & \to \_z_1.1\downarrow \\
\neg en_1 \wedge \neg r.e_1 & \to \_w_1.1\uparrow \\
\neg en_2 \wedge \neg r.e_2 & \to \_z_1.1\uparrow
\end{array}$$

$$\begin{aligned}
&\neg\_w_1.1 \wedge \neg\_z_1.1 &&\rightarrow\ r_1.1\uparrow\\
&\_w_1.1 \wedge \_z_1.1 &&\rightarrow\ r_1.1\downarrow\\
&\ldots
\end{aligned}$$

The acknowledgment signal $r.e_1$ and $r.e_2$ of the channel R are separated from the control signals en_1 and en_2. To measure the energy consumption of the function block, we count how many literals are charged/discharged in one cycle. (Each literal corresponds to one transistor, and one transistor corresponds to one unit of capacitance. Here we do not consider the sizing of transistors.) In the buffer of the $[3, 2]$ code, 75 units of capacitance are charged/discharged during one cycle.

A function block of the $[4, 2]$ repetition code is as follows:

$$\begin{aligned}
&en_1 \wedge r.e_1 \wedge l_1.0 &&\rightarrow\ \_w_1.0\downarrow\\
&en_2 \wedge r.e_2 \wedge l_3.0 &&\rightarrow\ \_z_3.0\downarrow\\
&\neg en_1 \wedge \neg r.e_1 &&\rightarrow\ \_w_1.0\uparrow\\
&\neg en_2 \wedge \neg r.e_2 &&\rightarrow\ \_z_3.0\uparrow
\end{aligned}$$

$$\begin{aligned}
&\neg\_w_1.0 \wedge \neg\_z_3.0 &&\rightarrow\ r_1.0\uparrow\\
&\_w_1.0 \wedge \_z_3.0 &&\rightarrow\ r_1.0\downarrow\\
&\neg\_w_1.0 \wedge \neg\_z_3.0 &&\rightarrow\ r_3.0\uparrow\\
&\_w_1.0 \wedge \_z_3.0 &&\rightarrow\ r_3.0\downarrow
\end{aligned}$$

$$\begin{aligned}
&en_1 \wedge r.e_1 \wedge l_1.1 &&\rightarrow\ \_w_1.1\downarrow\\
&en_2 \wedge r.e_2 \wedge l_3.1 &&\rightarrow\ \_z_3.1\downarrow\\
&\neg en_1 \wedge \neg r.e_1 &&\rightarrow\ \_w_1.1\uparrow\\
&\neg en_2 \wedge \neg r.e_2 &&\rightarrow\ \_w_3.1_2\uparrow
\end{aligned}$$

$$\begin{aligned}
&\neg\_w_1.1 \wedge \neg\_z_3.1 &&\rightarrow\ r_1.1\uparrow\\
&\_w_1.1 \wedge \_z_3.1 &&\rightarrow\ r_1.1\downarrow\\
&\neg\_w_1.1 \wedge \neg\_z_3.1 &&\rightarrow\ r_3.1\uparrow\\
&\_w_1.1 \wedge \_z_3.1 &&\rightarrow\ r_3.1\downarrow\\
&\ldots
\end{aligned}$$

In this function block, 52 units of capacitance in total are charged/discharged in a cycle. Therefore, the function block of the $[4, 2]$ code is expected to consume less energy than that of the $[3, 2]$ code if the sizing of transistors is not considered.

The same approach is used to estimate the energy consumption of function blocks computing a function f of a linear DI code encoding three-bit messages. The energy consumption of function blocks using the repetition code is compared with that of function blocks using non-repetition codes, as shown Figure 6.23. The x-position and the y-position of each dot correspond to the energy consumption of a function block using the repetition code and the energy consumption of a function block using a non-repetition code. Note that the unit of energy consumption is a hypothetical unit for comparison.

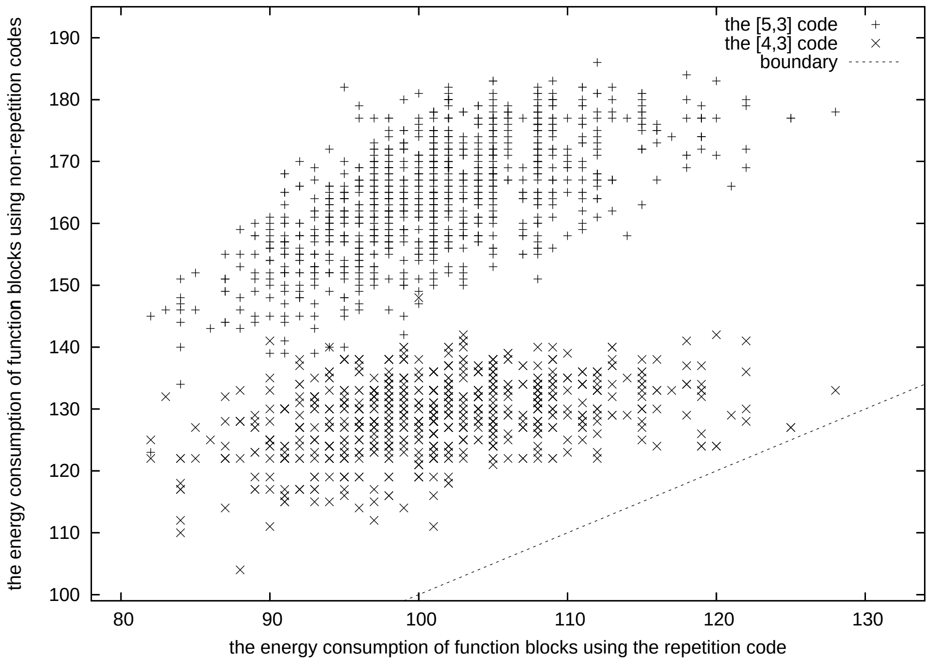


Figure 6.23: *Energy Distribution of Three-bit Function Blocks.*

Likewise the number of units of capacitors of a CC that are charged and discharged during one cycle can be counted. Since the number does not depend on the function f, but on a type of linear DI codes, constant units of energy are added to the distribution of the energy consumption of a function block of a linear DI code. The final distributions of the energy consumption of the PCHB implementation of $*[L?l; R!f(l)]$ are shown in Figure 6.24 and Figure 6.25. The energy consumption of a repetition code is always smaller than that of non-repetition codes when function blocks based on randomly generated 1,000 functions are compared.

The cycle time of a computation is determined by the delay of each gate (i.e., the time it takes to charge/discharge capacitors), and the number of gates in a loop of gates (i.e., the

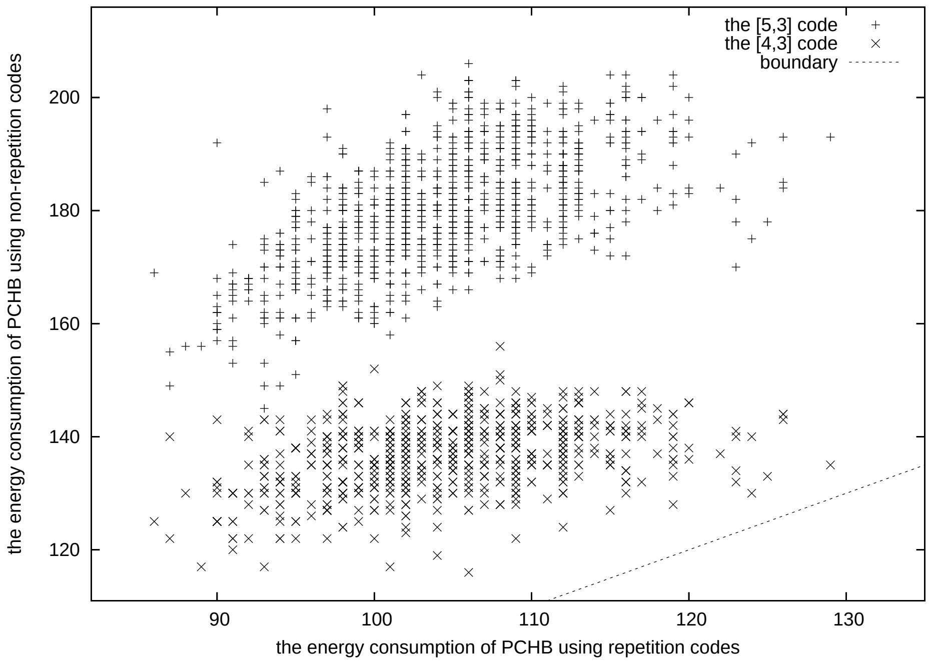


Figure 6.24: *Energy Distribution of Three-bit PCHB Processes.*

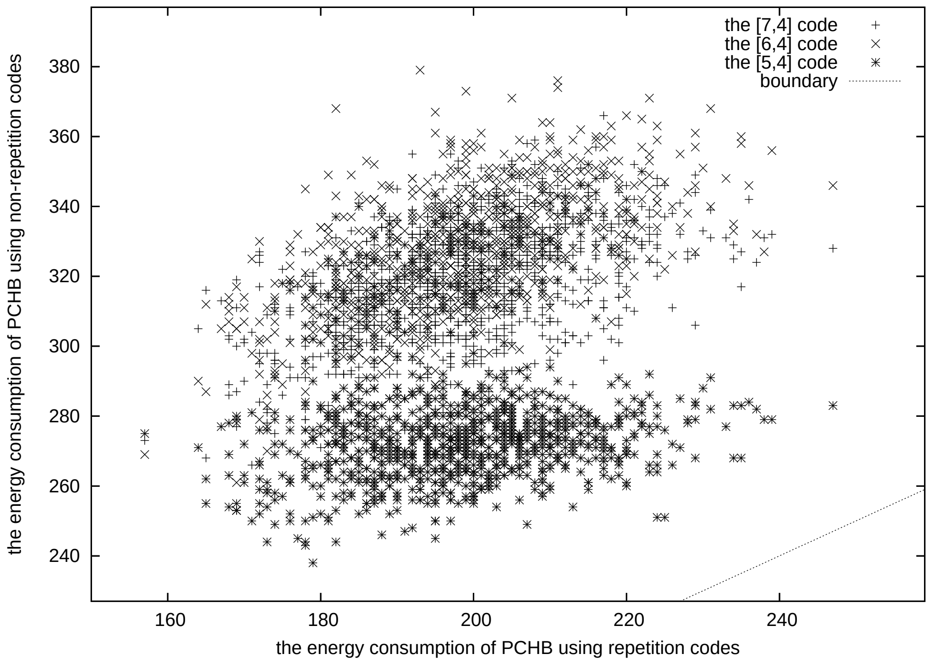


Figure 6.25: *Energy Distribution of Four-bit PCHB Processes.*

number of transitions in a cycle). The delay is determined by the fanout and the load of gates; the number of gates is determined by the depth of the function block, which is two, and the depth of a CC, which depends on a type of linear codes. (The depth is the maximum number of gates between a primary input and a primary output.) For example, the depth of the CC for the $[3,2]$ code is five, but that for $[4,2]$ code is three. The distributions of cycle times of the processes $*[L?l; R!f(l)]$ are shown in Figure 6.26 and Figure 6.27. (The function f are randomly generated.) After combining the delay and the energy numbers, we obtain the distributions of Et^2 of three-bit and four-bit processes, as shown in Figure 6.28 and Figure 6.29. According to the distributions, repetition codes are generally advantageous in the Et^2 metric, too.

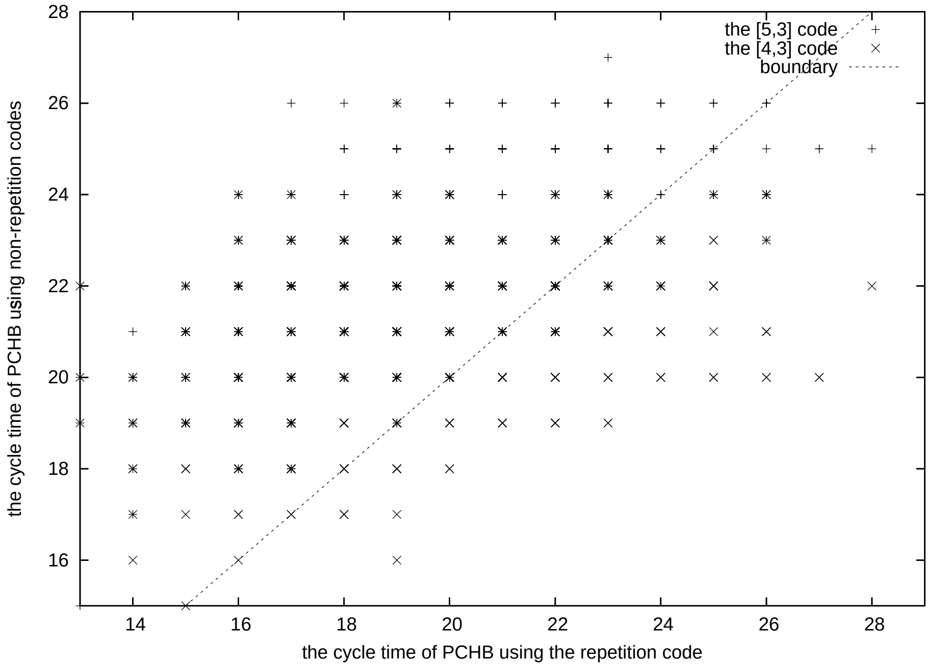


Figure 6.26: *Distribution of Cycle Time of Three-bit PCHB Processes.*

6.6 Simulation Results of QDI Circuits for Linear DI Codes

The circuit diagrams of a PCHB based on the $[4,2]$ repetition code and the $[3,2]$ parity code are shown in Figure 6.30 and Figure 6.31. As we expect, the circuit of the $[4,2]$ code is simpler than that of the $[3,2]$ code.

SPICE simulations were done in TSMC 0.18-μm CMOS technology. A result of simulating the buffer for the $[3,2]$ code with an error is shown in Figure 6.32. Charges (i.e., a soft error) are injected to $l_1.0$ at 8 ns in the simulation. We can see that no communication on the output channel R is initiated, while the input channel L is corrupted. Eventually

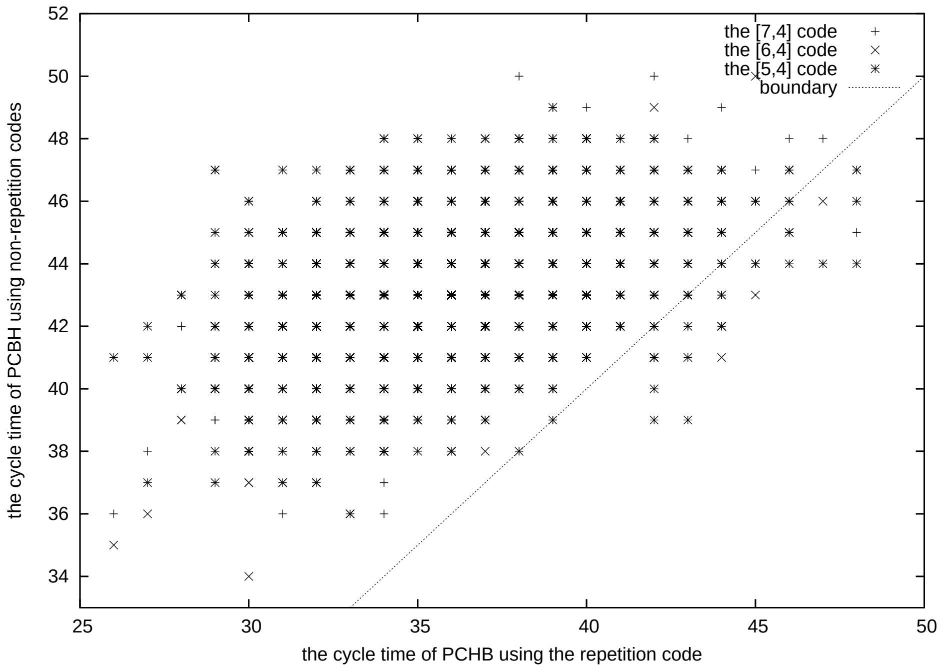


Figure 6.27: *Distribution of Cycle Time of Four-bit PCHB Processes.*

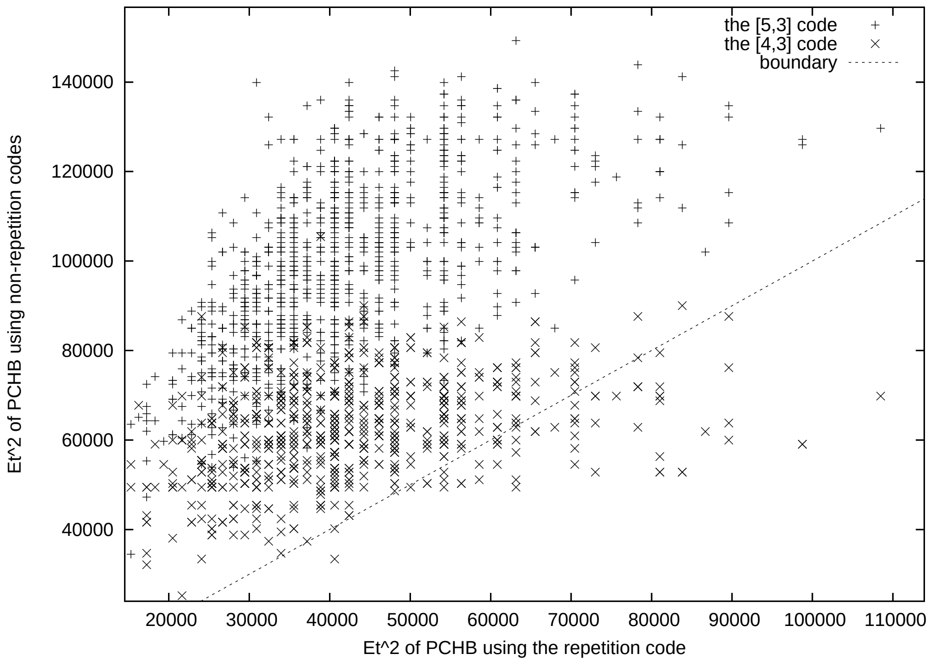


Figure 6.28: *Distribution of Et^2 of Three-bit PCHB Processes.*

the corrupted node $l_1.0$ is restored, and then the primary outputs are computed. That is, the whole system continues to work normally after a short delay.

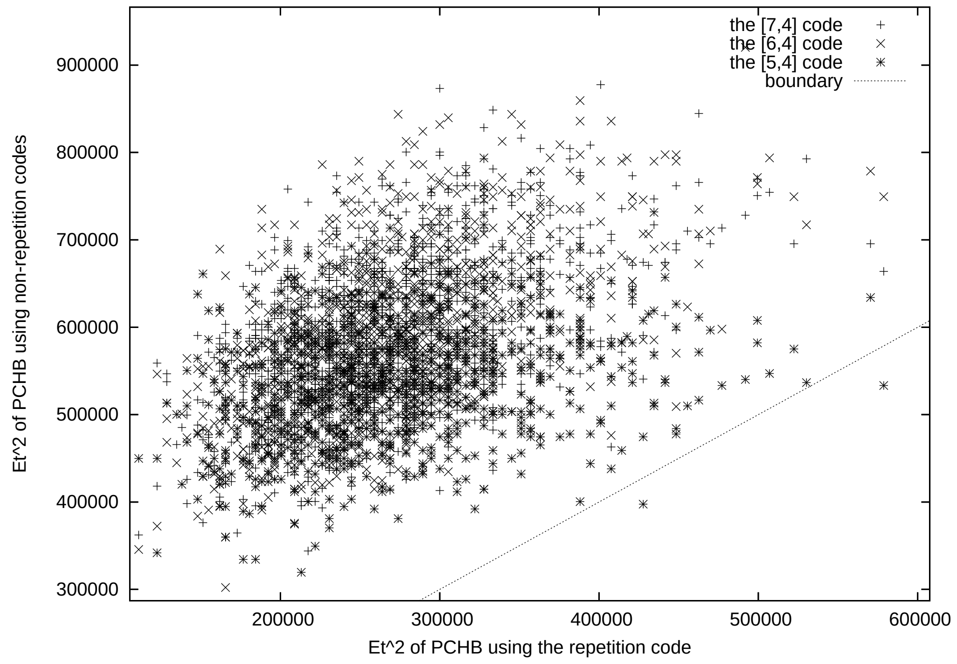


Figure 6.29: *Distribution of Et^2 of Four-bit PCHB Processes.*

Table 6.2: Performance Figures of PCHB for [4,2] Code and PCHB for [3,2] Code

	2-bit normal PCHB	[4,2] PCHB	[3,2] PCHB
# of transistors	68	156	264
Repetition Rate	660 MHz	446 MHz	343
Transitions in Cycle	14	18	22
Area(λ^2)	37536	84240	147888
Energy per Cycle(pJ)	1.6	3.5	5.7

6.7 Summary

In QDI circuits, which assume no ordering of assignments of variables, timing information should be encoded to data communications, so that DI codes are used for QDI circuits instead of general binary codes. Conventional DI codes such as one-of-n codes, however, are vulnerable to errors. If a QDI circuit employs conventional DI codes and an error occurs in the circuit, deadlock or incorrect computations can occur. For this reason, error detecting delay-insensitive (EDDI) codes are introduced, which provide both delay-insensitivity for asynchronous communications and redundancy for error tolerance. Although there are many types of EDDI codes, we have focused on linear DI codes, which are delay-insensitive versions of linear ECCs, since the function blocks and the CCs of linear DI codes are easy

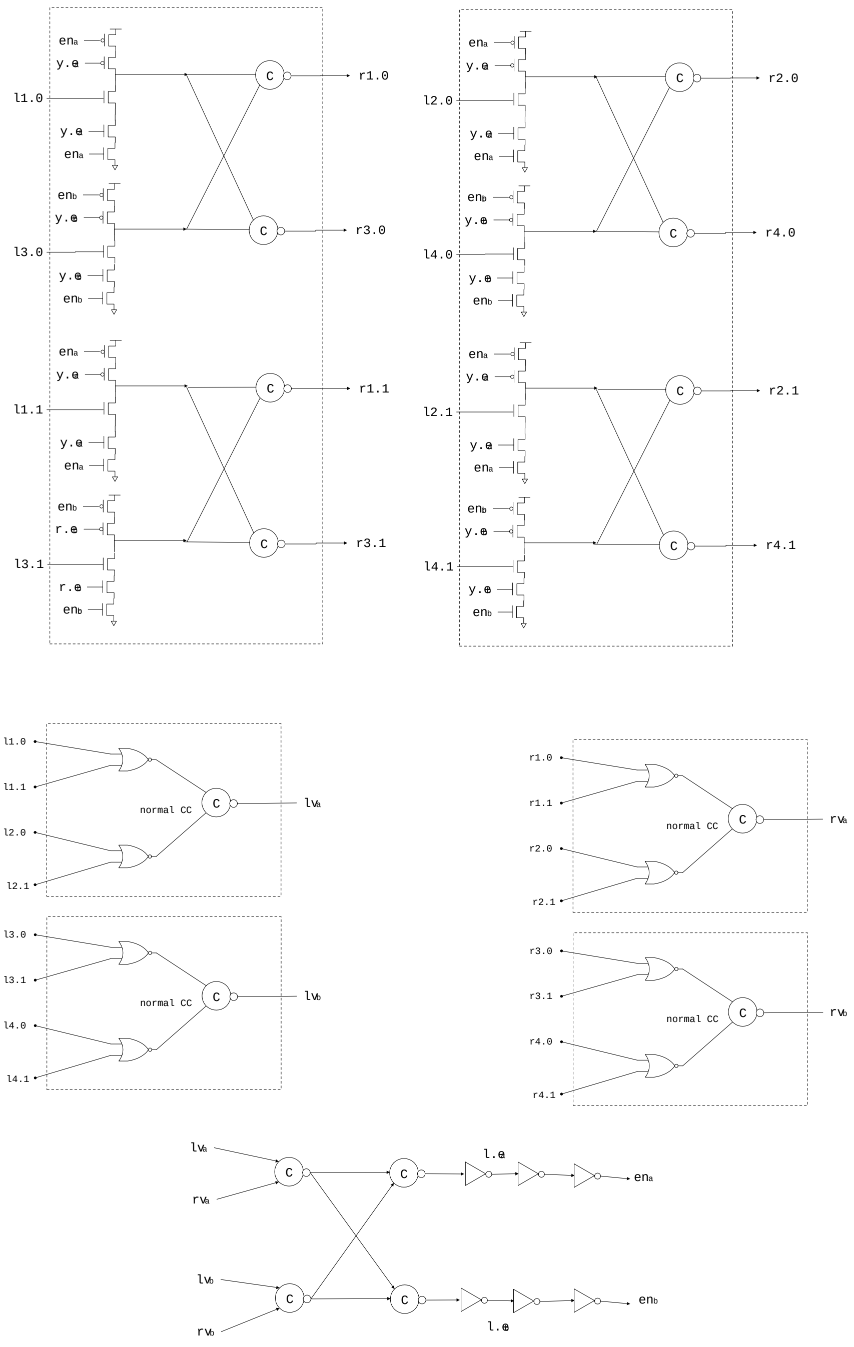


Figure 6.30: *PCHB based on the [4,2] Repetition Code. It is equivalent to the DD circuit of a 2-bit normal PCHB.*

to build and to analyze.

An error tolerant QDI circuit using linear DI codes prevents an invalid codeword from propagating, which ensures the persistence of correct inputs. Since the persistence can restore the corrupted outputs, the error tolerant QDI circuit can compute correctly even

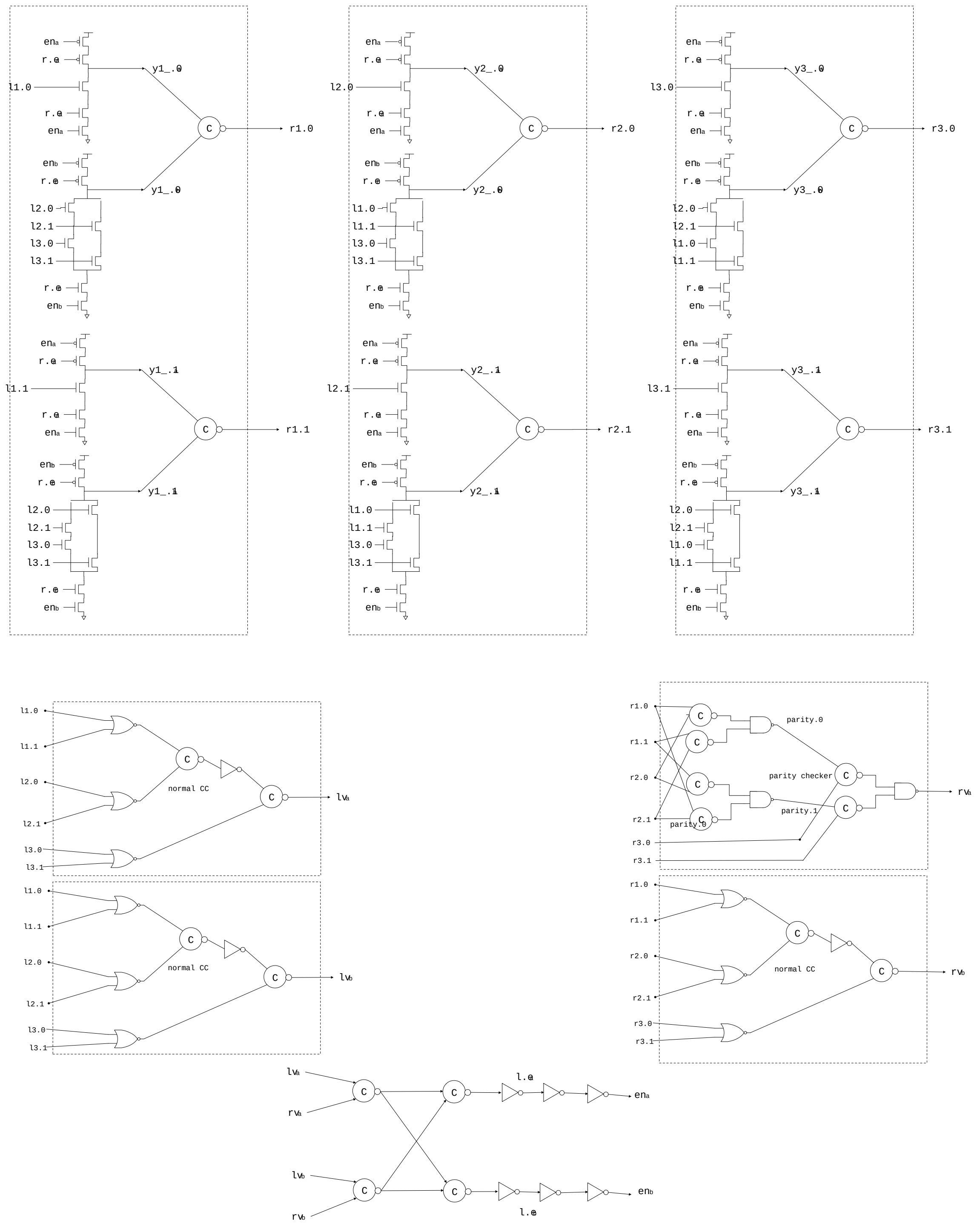
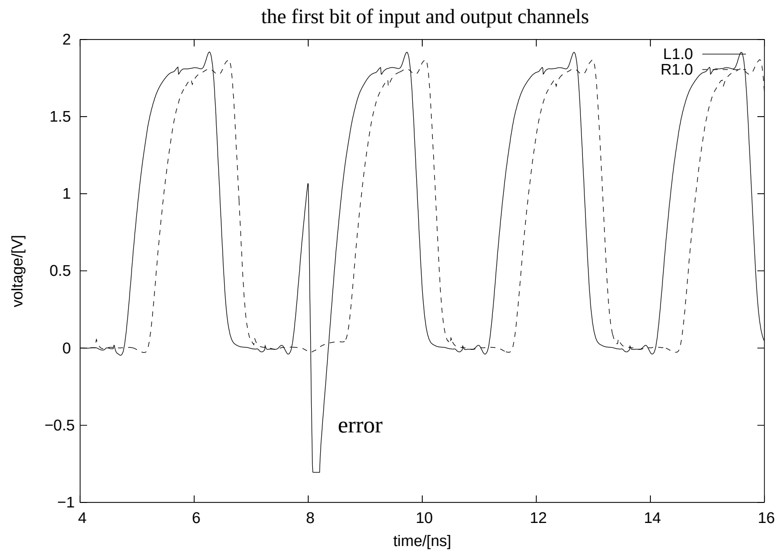


Figure 6.31: *PCHB based on the [3,2] Parity Code.*

though an error may delay computation momentarily. We have shown how to design an error tolerant QDI circuit for implementing the process $*[L?l; R!f(l)]$ with linear DI codes. The error tolerant circuit consists of an error tolerant function block and error tolerant



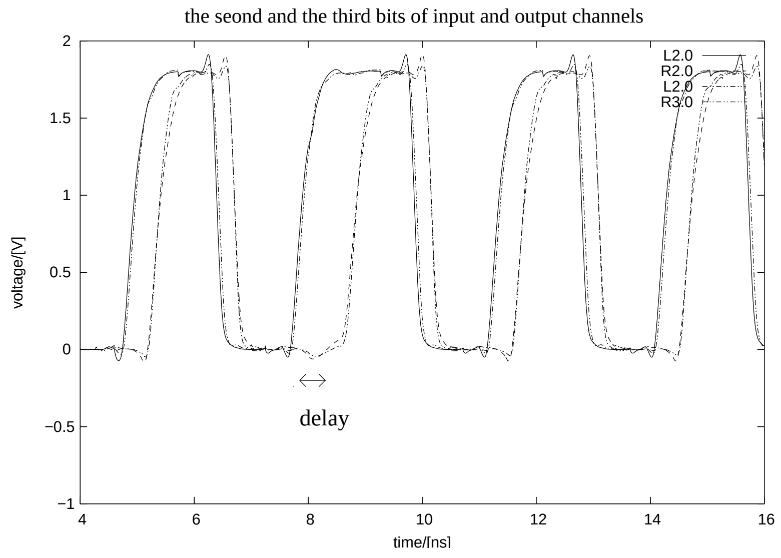


Figure 6.32: *Waveforms of Input and Output Channel Nodes of PCHB based on the [3,2] Parity Code. Until an error on the first bit of the input channel is corrected, all transitions of output-channel nodes are delayed.*

completion checkers. Although we have examined buffer-like processes only, function blocks and completion checkers are basic components of any QDI circuits, so that the results can be expanded to build general error tolerant processes.

Among several metrics under which to evaluate a circuit, the number of transistors is our first consideration. It is observed that most of the function blocks of repetition codes are smaller than those of non-repetition linear DI codes; the size of a CC for repetition codes is roughly half the size of a CC for non-repetition linear DI codes. Even under the

Et^2 metric, circuits based on repetition codes are better because they tend to be simpler than circuits based on non-repetition codes. In another aspect, it is advisable to employ the repetition codes (i.e., the DD scheme), because it is easier to turn a QDI circuit into a repetition-code circuit (i.e., a DD circuit) than to apply a non-repetition code. Hence, we conclude that the DD scheme should be generally used in the design of error tolerant QDI circuits.

Chapter 7

Design Example: Soft-error Tolerant Asynchronous Microprocessor

We have proposed a method for making all components of QDI systems soft-error tolerant, in the previous chapters. Here we shall demonstrate the method by detailing the design of a soft-error tolerant asynchronous microprocessor (STAM) down to transistor level.

The overview of the STAM is given first, including its CHP description and its top-level decomposition. Then, as an example of synthesizing a circuit from a CHP description, the design of its program counter is detailed. In the end, the results of simulating the STAM are shown.

7.1 Overview

The STAM architecture implements a simple 32-bit RISC instruction set. The STAM has eight general-purpose registers, and it has four types of instructions: arithmetic, branch, shift, and memory operations. The detail of the architecture of the STAM is defined in Appendix E, which is identical to the architecture previously used in designing a simple pulsed asynchronous microprocessor (SPAM) [60].

The sequential description of the STAM in CHP is as follows:

```
$SEQ\_STAM \equiv$
    $*[instr := imem[pc];$

        $opx := gpr[instr.rx],$
        $[\ \ instr.ymode = REG \longrightarrow opy := gpr[instr.ry]$
         $[]\ instr.ymode = IMM \longrightarrow opy := instr.imm$
         $[]\ instr.ymode = IMMSHIFT \longrightarrow opy := instr.imm << 16$
         $[]\ instr.ymode = REGIMM \longrightarrow opy := gpr[instr.ry] + instr.imm$
        $];$

        $[\ \ instr.unit = ALU \longrightarrow opz := OP\_ALU(instr.op)(opx, opy)$
        $[]\ instr.unit = BRANCH \longrightarrow brch := OP\_BRANCH(instr.op)(opx, opy)$
        $[]\ instr.unit = DMEM \longrightarrow opz := OP\_DMEM(instr.op)(opx, opy)$
        $[]\ instr.unit = SHIFT \longrightarrow opz := OP\_SHIFT(instr.op)(opy)$
        $];$

        $[\ instr.unit \neq BRANCH \longrightarrow gpr[instr.rz] = opz,\ pc := pc + 4$
        $[]\ instr.unit = BRANCH \longrightarrow pc := brch$
        $]$
    $]$
```

For synthesizing the STAM easily, the sequential description of the STAM is decomposed into small concurrent processes. Although there are several possible ways of decomposing the CHP process, we shall re-employ the existing decomposition of the SPAM for convenience. At the top level, the STAM is decomposed into seven processes: *IMEM*, *DECODE*, *REGFILE*, *OPERANDS*, *EXEC*, *PCUNIT*, and *WB*, as shown in Figure 7.1. Certainly these units will be decomposed further into smaller components so that they can be fit into circuit templates such as the PCHB template.

The description of each unit is as follows:

- *IMEM* is an internal instruction memory holding up to 128 instructions. If necessary, we can easily expand the size of the instruction memory. Its CHP description is

  $$*[\ Addr?addr;\ \ Instr!imem[addr]],$$

  which corresponds to $instr := imem[pc]$ in the sequential description. This description is similar to the memory unit in Chapter 5, even though the writing part is

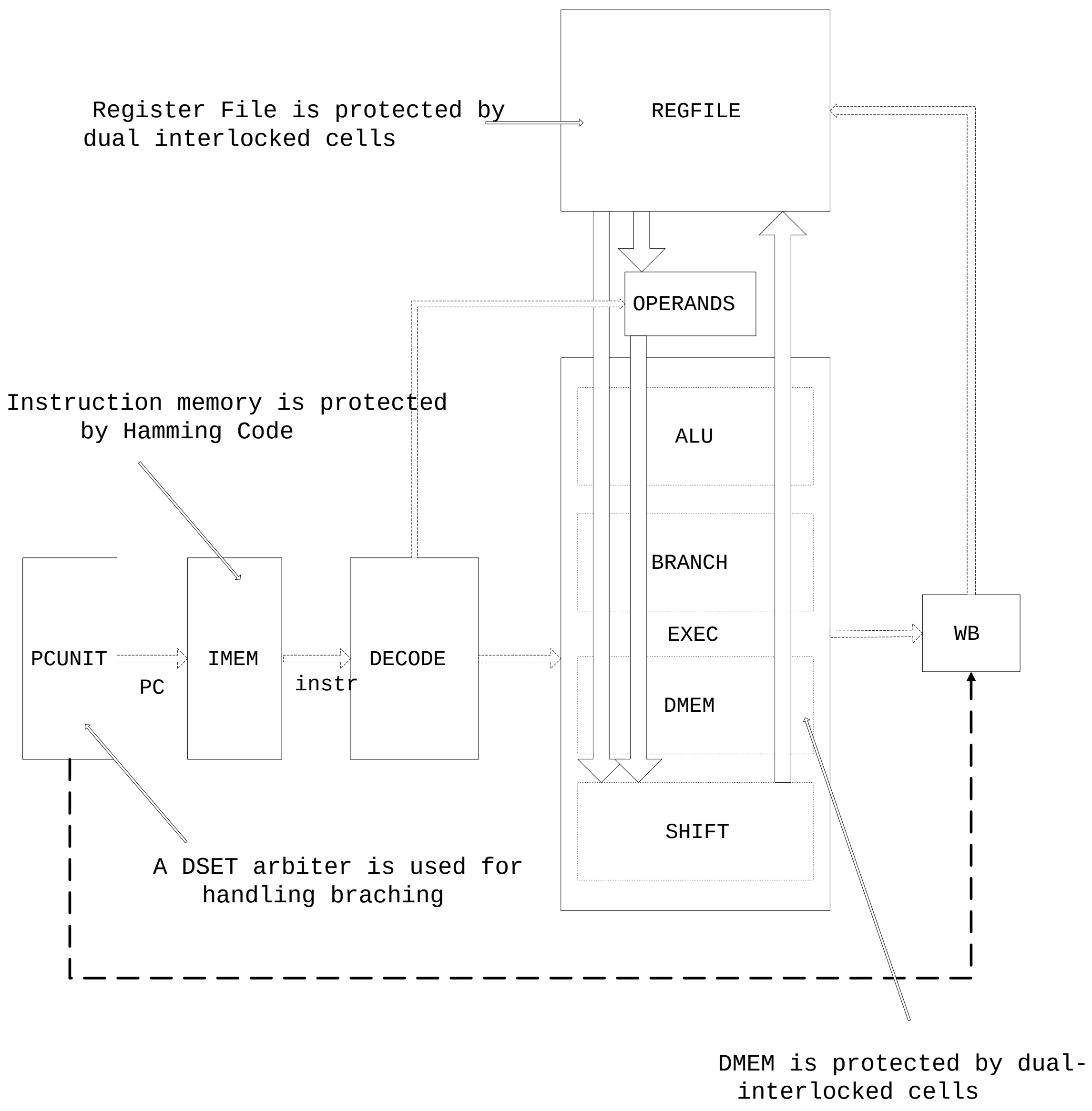

Register File is protected by dual interlocked cells
REGFILE
OPERANDS
Instruction memory is protected by Hamming Code
ALU
BRANCH
EXEC
DMEM
SHIFT
PCUNIT
IMEM
DECODE
WB
PC
instr
A DSET arbiter is used for handling braching
DMEM is protected by dual-interlocked cells

Figure 7.1: *Decomposition of the STAM. The dotted arrows denote control flows, and block arrows denote data flows. Most of the units are protected by the DD scheme.*

removed; the instruction memory can be easily designed with either the array of dual interlocked cells or a Hamming-coded array. Both designs have been implemented for the STAM.

- *DECODE* is simple to build, since the format of all instructions is the same. It merely copies the bit fields of a fetched instruction to several processes as control signals. For example, the most significant two bits of an instruction represent the type of an instruction, and the bit fields determine where operands are distributed in the *EXEC* process.

- *REGFILE*, which has eight 32-bit registers, implements $opx := gpr$[$instr.rx$], $opy := gpr$[$instr.ry$], and gpr[$instr.rz$] $:= opz$. The execution unit can avoid waiting to write a result into a register when it reads and writes the same register, using the bypass mechanism of the MiniMIPS [61].

- *OPERANDS* computes *opy* according to *instr.ymode*, which implements [$instr.ymode = REG \rightarrow opy := gpr$[$instr.ry$] [] ...].

- *EXEC* does calculation with given operands. There are four sub-processes such as *ALU*, *BRANCH*, *DMEM*, and *SHIFTER*, which handle the four types of instructions.

- *PCUNIT* updates the program counter (PC). Although there are several methods of handling branching, the arbitrated-branch mechanism of the SPAM is employed.

- *WB* is a write-back unit, which decides whether the result of the *EXEC* is written back to the *REGFILE*. That is, the *WB* guarantees the sequencing of instructions after a branching instruction. (This process is not necessary in the sequential description.)

The arbitrated branch and the role of *WB* are explained in Appendix F.

7.2 Designing Soft-Error Tolerant PCUNIT from CHP Description to Layout

In this section, the design of the *PCUNIT* is detailed. The role of the *PCUNIT* is to increment the PC every cycle and, if necessary, to replace the PC with a branch target address. There are a few approaches to manage branching. For example, when an instruction is fetched from the *IMEM*, a pre-decoding stage determines whether the fetched instruction is a branch instruction or not, and the pre-decoding notifies the *PCUNIT* to wait for a branch-target address from the *BRANCH*. This approach is in the design of a sub-nanojoule microprocessor, the Lutonium [2]. For designing the *PCUINT* of the STAM, we employ another approach, as follows. The *PCUNIT* probes a channel from the *BRANCH* to decide whether the PC should be changed or not; the communication between the *PCUNIT* and the *BRANCH* is established only if a branch instruction is encountered. The probed branching was proposed in the implementation of the SPAM, which was inspired by the exception handling mechanism in MiniMIPS and one of the AMULET designs [61, 62, 63].

The sequential CHP of the *PCUNIT* is as follows:

```
$PCUNIT \equiv$

    $pc$ := $init\_pc$, $Branched$!false;
    *[$Addr$!$pc$;
      $pc$ := $pc$ + 4;
     [$\neg\overline{DoBranch}$ $\longrightarrow$ $Branched$!false
     | $\overline{DoBranch}$ $\longrightarrow$ $BranchTo$?$pc$ , $Branched$!true
     ]
    ]
```

($\overline{X}$ in the CHP is the probe of the channel X, which is a boolean function that indicates whether or not there is a pending communication on the channel X.) By probing the channel *DoBranch* from the *BRANCH*, the *PCUNIT* learns whether it needs to update the current PC or not. Finally, the information about branching is sent to the *WB* through the channel *Branched*.

The sequential CHP of the *PCUNIT* can be decomposed into two units by factoring out the arbitrated mechanism, as follows:

```
$EVAL\_BRANCH \equiv$

    *[$\neg\overline{DoBranch}$ $\longrightarrow$ $IsBranch$!false
    | $\overline{DoBranch}$ $\longrightarrow$ $IsBranch$!true, $DoBranch$
    ]
```

$PCUNIT\_NOARB \equiv$

```
    $pc\ :=\ init\_pc,\ Branched!\mathtt{false};$
    $*[Addr!pc;$
       $pc\ :=\ pc\ +\ 4;$
       $IsBranch?is\_branch;$
       $[\ is\_branch = \mathtt{false}\ \longrightarrow\ Branched!\mathtt{false}$
       $[]\ is\_branch = \mathtt{true}\ \longrightarrow\ BranchTo?pc,\ Branched!\mathtt{true}$
       $]$
    $]$
```

In the following subsections, the details of the *EVAL_BRANCH* and of the *PCUNIT_NOARB* are shown.

7.2.1 Implementing Arbitrated Branch

If we assume that the channels *DoBranch* and *IsBranch* are both passive, we can compile the CHP process *EVAL_BRANCH* into the following HSE (The 'passive' channel is explained in Chapter 2.):

```
    $*[[\ \neg DoBranch.i\ \longrightarrow\ [IsBranch.e];\ IsBranch.false\uparrow;$
                                $[\neg IsBranch.e];\ IsBranch.false\downarrow$
     $|\ DoBranch.i\ \longrightarrow\ [IsBranch.e];\ IsBranch.true\uparrow;$
                                $[\neg IsBranch.e];\ DoBranch.e\downarrow;\ [\neg DoBranch.i];$
                                $IsBranch.true\downarrow;\ DoBranch.e\uparrow$
     $]]$
```

Since *DoBranch* is a synchronization channel, which does not carry data, a variable *DoBranch.i* and an acknowledgment variable *DoBranch.e* implement the channel *DoBranch*; variables *IsBranch.true* and *IsBranch.false* encode a one-of-two code, and the variables with an acknowledgment variable *IsBranch.e* implement the channel *IsBranch*. In order to compile this particular HSE into a PRS easily, the check of $\neg DoBranch.i$ in the HSE for the first guarded command is eliminated, as follows:

```
*[[ $IsBranch.e \longrightarrow IsBranch.false\uparrow$; [$\neg IsBranch.e$]; $IsBranch.false\downarrow$
 | $DoBranch.i \longrightarrow$ [$IsBranch.e$]; $IsBranch.true\uparrow$;
                                   [$\neg IsBranch.e$]; $DoBranch.e\downarrow$; [$\neg DoBranch.i$];
                                   $IsBranch.true\downarrow$; $DoBranch.e\uparrow$
 ]]
```

The CMOS implementation of a two-way arbitrated selection statement corresponds to a basic arbiter in Chapter 5, which alternatively executes every selection whose check is evaluated to be `true`. In other words, if $DoBranch.i$ is `true`, then a basic arbiter ensures that the second selection (i.e., branching) executes eventually. Because of the fairness, we can replace the check $\neg DoBranch.i$ with the check $IsBranch.e$.

After factoring out a basic arbiter from the HSE, we obtain the following:

```
*[[ $IsBranch.e \longrightarrow u\uparrow$; [$\neg IsBranch.e$]; $u\downarrow$
 | $DoBranch.i \longrightarrow v\uparrow$; [$\neg DoBranch.i$]; $v\downarrow$
 ]]
$\|$
*[[ $u \longrightarrow IsBranch.false\uparrow$; [$\neg u$]; $IsBranch.false\downarrow$
 [] $v \longrightarrow$ [$IsBranch.e$]; $IsBranch.true\uparrow$; [$\neg IsBranch.e$]; $DoBranch.e\downarrow$;
        [$\neg v$]; $IsBranch.true\downarrow$; $DoBranch.e\uparrow$
 ]]
```

The first process shown in the decomposition above is an arbiter between $IsBranch.e$ and $DoBranch.i$; the compilation of the second process results in the following PRS:

$u \rightarrow \_IsBranch.false\uparrow$
$\neg u \rightarrow \_IsBranch.false\downarrow$

$v \wedge IsBranch.e \rightarrow \_IsBranch.true\downarrow$
$\neg v \wedge \neg IsBranch.e \rightarrow \_IsBranch.true\uparrow$

$\_IsBranch.true \rightarrow IsBranch.true\downarrow$
$\neg\_IsBranch.true \rightarrow IsBranch.true\uparrow$

$\_IsBranch.true \rightarrow \_DoBranch.e\downarrow$
$\neg\_IsBranch.true \wedge \neg IsBranch.e \rightarrow \_DoBranch.e\uparrow$

$$\begin{aligned} \_DoBranch.e &\to DoBranch.e\downarrow \\ \neg\_DoBranch.e &\to DoBranch.e\uparrow \end{aligned}$$

In fact, a PRS of a similar HSE has been already demonstrated in the design of the MiniMIPS, but the MiniMIPS's PRS has more latency than the PRS shown above [62]. The corresponding circuit of the PRS with a basic arbiter is shown in Figure 7.2. It is easy to make the circuit soft-error tolerant: an arbiter is replaced with the DSET arbiter, and the DD scheme is applied to the rest of the circuit.

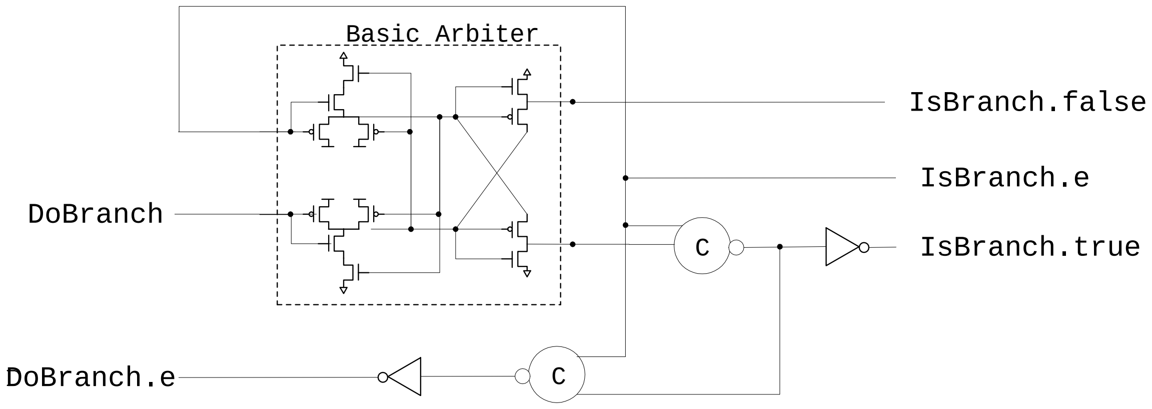


Figure 7.2: *Circuit for Probing Branch. If there is a pending communication on the channel DoBranch, then the value of the channel IsBranch is assigned to be true. If not, the value is assigned to be false.*

7.2.2 Implementing Body of PCUNIT

A decomposition of the *PCUNIT_NOARB* is as follows:

```
$INCREMENTER \equiv$
    $PCIncOut!init\_pc;$
    $*[PCIncIn?pc; PCIncOut!pc + 4]$
$SELECTOR \equiv$
    $*[BC?bc,\ PCIncOut?pc,\ BranchTo?branch\_to$
      [ $bc$ = false $\longrightarrow$ $NewPC!pc$
      [] $bc$ = true $\longrightarrow$ $NewPC!branch\_to$
      ]
    ]
```

$COPY \equiv$

```
    *[NewPC?pc;  Addr!pc,  PCIncIn!pc]
```

$CONTROL \equiv$

```
    Branched!false;
    *[IsBranch?is_branch
      [ is_branch  =  false  ⟶  BC!false, Branched!false
      [] is_branch  =  true  ⟶  BC!true, Branched!true
      ]
    ]
```

$PCUNIT\_NOARB \equiv INCREMENTER \parallel SELECTOR \parallel COPY \parallel CONTROL$

The corresponding process graph is shown in Figure 7.3. Extra buffers, which implement initial send/receive commands (e.g., *PCIncOut*!*init_pc* in the *INCREMENTER*), are not shown in the decomposition but explicitly depicted in the process graph. If the boolean channel *IsBranch* from the *EVAL_BRANCH* carries `false`, then the current PC is copied to the *EXEC* and to the *IMEM*. If the boolean channel *IsBranch* carries `true`, then the target address on the channel *BranchTo* is read, and the PC will eventually be updated according to the target address.

Although the process is decomposed functionally enough, there is still an extra decomposition step: splitting up wide channels such as the 32-bit channel *NewPC* into the small one or two-bit wide channels that can be implemented by one-of-two codes or one-of-four codes. This step, called *vertical decomposition*, requires the distribution of signals of the control channels to the vertically decomposed processes. A byte-skewing scheme of the SPAM is used again to distribute control signals in the STAM, since it provides a good trade-off between throughput and latency. In byte skewing, a control signal is simultaneously distributed within a byte, but the process of the next byte gets a copy of the control signal in the next time step, which is illustrated in Appendix G.

After finishing vertical decomposition, we now have small CHP processes, which can easily be synthesized. For obtaining a circuit from a CHP description, we can use a semi-automatic synthesis flow. (Some existing tools such as `pl2xprs` and `cflat` have been modified to generate layouts of DD circuits.) The steps of the synthesis flow are illustrated

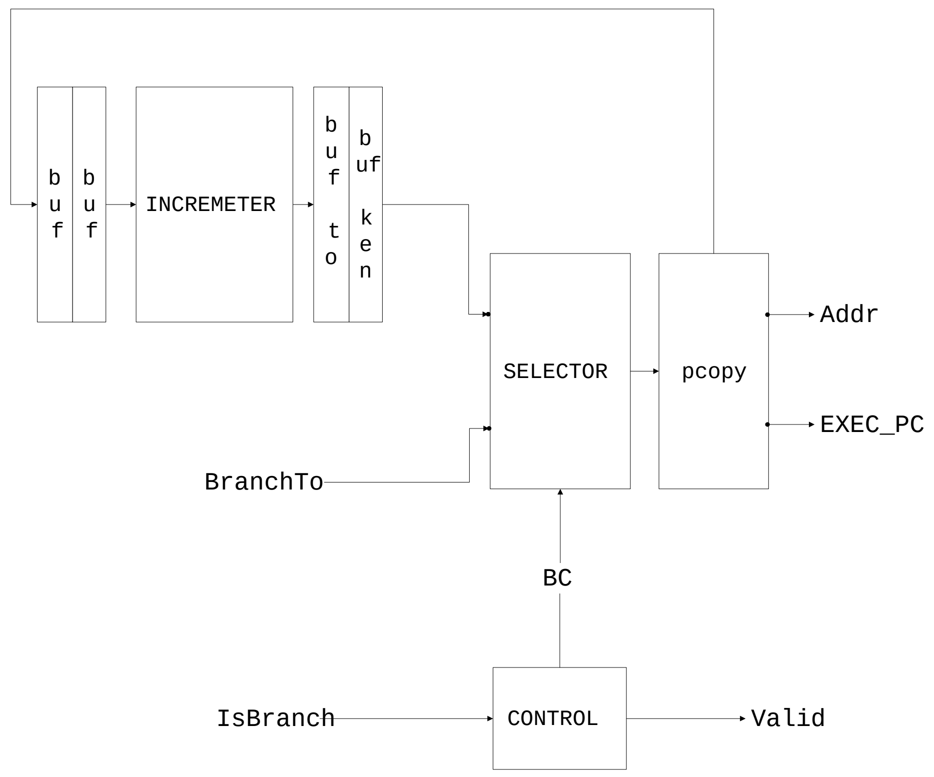


Figure 7.3: *Process Graph of PCUNIT_NOARB.*

in Figure 7.4. As an example, the details of synthesizing the *SELECTOR* according to the tool flow are given in Appendix G. Besides CHP, PRS, and HSE, a language PL2 is introduced to describe a system in the synthesis flow. The language can only describe CHP processes whose input channels and output channels are used at most once in each iteration, but PL2-describable CHP processes can be readily fit into circuit templates [60]. After decomposing a complex CHP process into PL2-compatible processes, we can compile a PL2 process into a CAST description of a DD circuit with the complier `pl2ddxprs`, which is a DD version of `pl2xprs`. (CAST is a hierarchical, lexically scoped circuit description language [64].) By extending the template-based PCHB design, `pl2ddxprs` first does boolean minimization with one-of-n encoded multi-valued variable and does synthesis of multiple-input and multiple-output logic networks. Then the program duplicates a generated circuit, and adds cross-coupled C-elements. `pl2ddxprs` provides an automatic pathway from a high-level description to a low-level description.

`cflat_layout` converts the CAST description into a PRS and generates auxiliary information for the layout tools. Using the extracted information, a layout tool `xprs2stack`

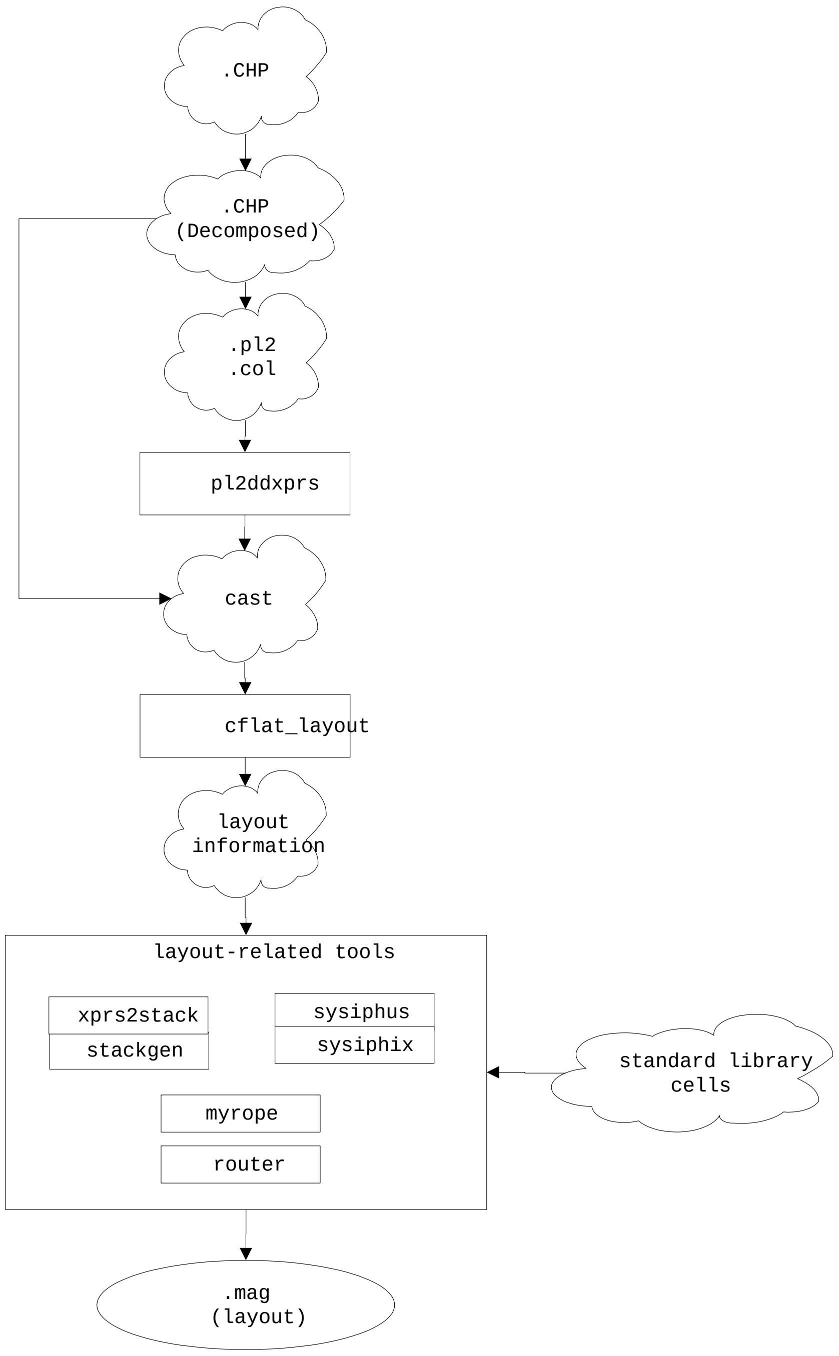


Figure 7.4: *Synthesis Flow from CHP Description to Layout. A box represents a tool, and a cloud shape represents an input description for a program.*

together with `stackgen` generates a stack of transistors for a specific computation, which can be a part of computation blocks in a circuit template. Given a list of instances of cells, a placement tool `sysiphus` uses a simulated annealing heuristic to place the instances. That is, `sysiphus` places generated stacks of transistors and standard library cells (e.g., inverter, NAND, NOR) to generate a large cell. While the cost function of `sysiphus` is set up to minimize the expected Manhattan length of connection wires in the placement, it is hard to impose an internal structure on the placement, so `sysiphus` is mostly used to produce a layout of small processes such as bit-wide processes. (Manhattan length is the sum of the length of the projections of the line segment between two points onto the coordinate axes.) On the other hand, the placement tool `myrope` is used to place sysiphused layouts into a larger layout according to a floor plan determined by a designer. For example, we can compose layout of *EXEC* by putting layouts of *ALU*, *BRANCH*, *DMEM*, and *SHIFTER* side by side with `myrope`. After the placement is done, a suite of routers completes the layout with wires. The final layout of a CHP process is in the format of `magic`, a VLSI layout system [65].

The wired layout of the soft-error tolerant *PCUNIT* based on the DD scheme has been completed in TSMC 0.18-μm CMOS technology. The *PCUNIT_NOARB* of the *PCUNIT* has been synthesized by the tools, and the *EVAL_BRANCH* has been done by hand.

7.3 Evaluating Soft-error Tolerant Asynchronous Microprocessor in Simulations

The partially wired layout of the whole STAM was completed in 0.18-μm CMOS technology. The STAM has approximately one-and-a-half million transistors. We have designed most of the STAM, following the synthesis flow shown in the previous section and employing the DD scheme for soft-error tolerance. Certainly designing memory units, especially array, controller, and arbiters, has required human intervention.

7.3.1 Verifying Soft-error Tolerance

The soft-error tolerance of the STAM has been verified in three different ways. (1) The tolerance of small processes are verified exhaustively in PRS computation. From each valid state, every erroneous execution path is checked in order to ensure that an error cannot cause any erroneous behavior. (2) While the STAM runs a program in the production-rule simulator `csim`, variables of the STAM are flipped randomly. After the simulation is finished, the values of the registers are verified. (3) Charges (i.e., an error) are injected into the STAM every few nanoseconds in SPICE simulations. As before, the values of the registers are verified in the end of a simulation.

First, `seucheck` has been written for the exhaustive testing. It facilitates verification of error tolerance of a given PRS by exploring all possible states of the PRS with an error on each variable of the PRS. (An example of using `seucheck` is in Appendix H.) That is, a valid-state set of the PRS of a process is generated, whose elements are states reachable from the initial state of the process by firing of PRs. Then the program checks whether or not a soft error on any node (a bit flipping) can cause a transition to an invalid state (deadlock state) where no PRs are effective; it checks whether the firings of PRs are excluded or repeated, owing to an error (i.e., data can be missed or can be generated unexpectedly). Tested processes in the STAM have been verified to tolerate an error in all possible states, as we expect. In fact, this testing is suited for small processes whose PRS has around 50 variables, since the number of states increases exponentially as the number of variables increases. If `seucheck` is used for a PRS, which has 100 or more variables, the program does not finish within more than a week in a machine with a 2.0 GHz Intel Pentium 4.

Secondly, an extended `csim`, which has been written for simulating bit-flipping (i.e., soft errors), can randomly choose a variable in a given PRS and can flip the value of the variable during the execution of the PRS. If the `csim` is set to flip 15 nodes per cycle on average while the STAM runs the RC4 stream cipher, approximately 25% of electrical nets of the STAM experience bit-flipping until the end of simulation. (The RC4 stream cipher is used in popular protocols such as secure sockets layer to protect Internet traffic, and wired encryption protocol to secure wireless networks [66].) The coverage of bit-flipping is shown in Figure 7.5. The results of the bit-flipping tests in the `csim` have shown that corrupted nodes are restored quickly, so that the STAM is demonstrated to tolerate multiple errors. In fact, if we allow more than 20 errors per cycle on average, we have observed that the STAM fails to compute once in a while, because multiple errors occurred too close in distance (e.g., errors on a set of cross-coupled variables).

Thirdly, one soft error, which is modeled as excess charges, is injected into randomly chosen nodes every cycle on average in SPICE simulations. For example, excess charges are injected into two nodes in the *PCUNIT*, as shown in Figures 7.6 and 7.7. In the first waveform, we can see that the corrupted node is restored quickly; in the second waveform, the erroneous transition is a premature firing of an expected transition, which is not propagated to the environment until the transition of a corresponding duplicated node happens.

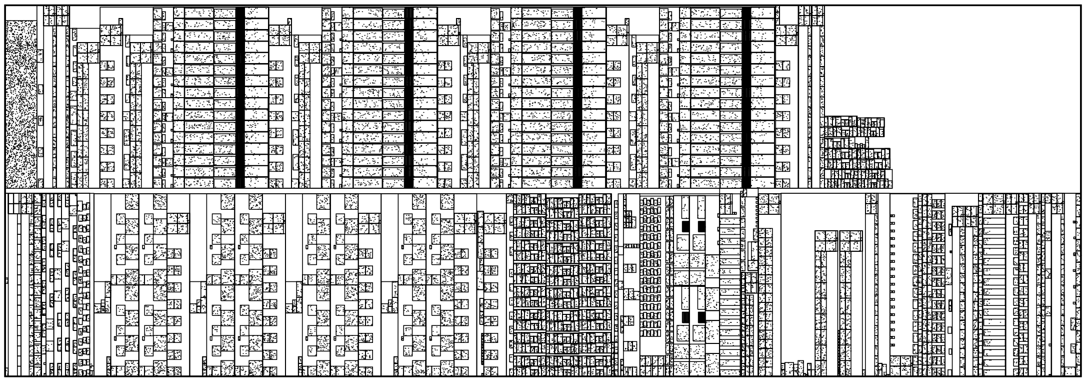


Figure 7.5: *Floor Plan of the STAM and Locations of Flipped Nodes during Digital Simulation of the STAM. Each dot represents the location of a flipped node, and a box represents the bounding box of a circuit of a decomposed small process, which includes a few hundred transistors.*

Throughout the several SPICE simulations, it has been verified that the STAM can tolerate each soft error, which occurs at every cycle on average.

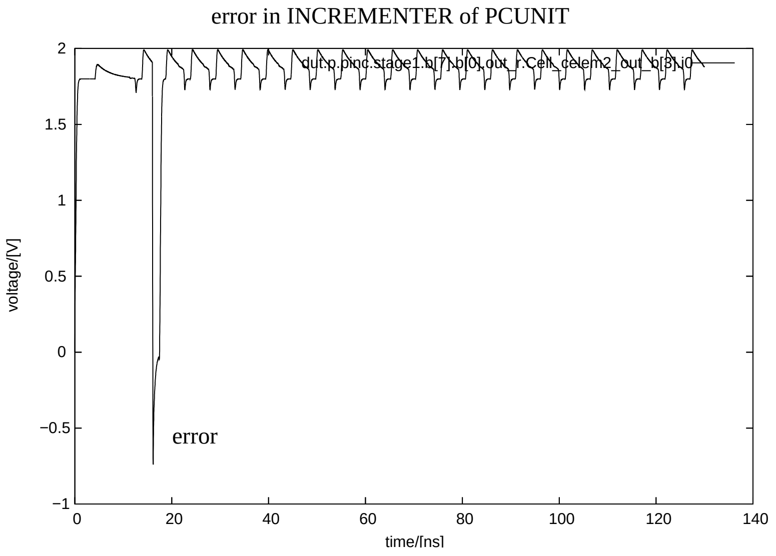


Figure 7.6: *Waveform of One Corrupted Node in PCUNIT. An error occurs at 12 ns, and the corrupted node is restored quickly.*

7.3.2 Performance

For obtaining the performance figures, the STAM has been tested in SPICE and digital simulations with small programs such as a no-operation program, and the RC4 stream cipher.

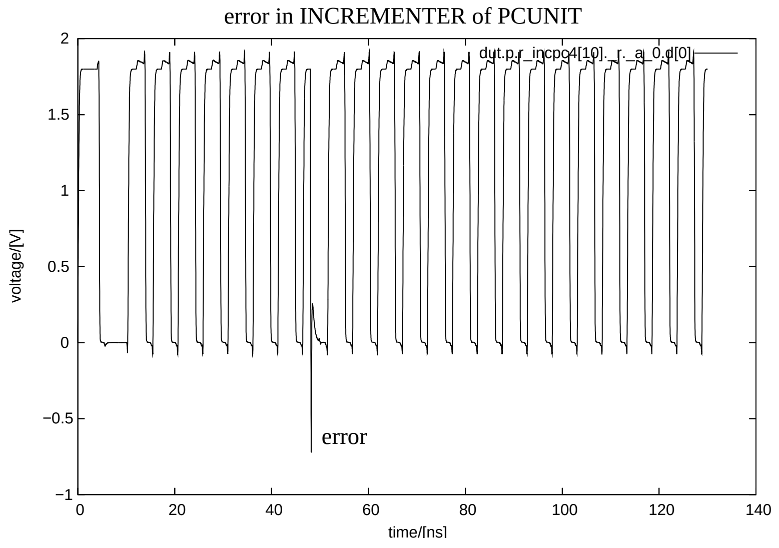


Figure 7.7: *Waveform of One Corrupted Node in PCUNIT. An error occurs at 44 ns, which causes a premature transition, but it does not affect the correctness of computation, because cross-coupled C-elements of the DD scheme prevent the propagation of the premature firing.*

The no-operation program consists of a series of an instructions '`and r0=r0,r0`.' It merely fills the pipeline of the STAM so that the maximum throughput of the STAM can be tested. The STAM runs at 22 transitions per cycle. SPICE simulations have shown that the STAM runs approximately 170 MHz with the no-operation program, which is limited by the performance of the *REGFILE*; it consumes approximately 5.1 nJ per fetched instruction.

On the other hand, the RC4 stream cipher uses most of the functionality of the STAM. According to the results of simulations in `csim`, the program runs at 35 transitions per cycle on average; `csim` has revealed that the critical path includes the carry chain of the ripple-carry adder in the *ALU*. If necessary, the ripple-carry adder can be replaced with a carry-lookahead or a carry-select adder in order to achieve the full throughput. `csim` can also estimate the energy consumption based on a fanout-weighted transition-counting model, which is off by less than 20% compared with the results of SPICE simulations. For the RC4 program, it has been estimated that 7.1 nJ is consumed per effective instructions or 6.4 nJ is consumed per fetched instruction. (The fetched instructions include effective instructions whose results are written to the *REGFILE*, and vacuous instructions which are fetched in the middle of branching.)

7.3.3 Comparing Overhead of Error Tolerance

Adding error tolerance reduces throughput up to by 40%. For example, the *PCUNIT* of the STAM runs at 197 MHz, which is approximately 60% of the throughput of the *PCUNIT* in a corresponding simple asynchronous microprocessor (SAM), which employs no error tolerant methods. The reasons for the reduction of the throughput are as follows. The first cause is that the STAM requires more transitions per cycle than the SAM. While completion checkers of an output channel in a normal circuit take their inputs from pulldown stacks of function blocks directly, DD completion checkers take their inputs from cross-coupled C-elements, as shown in Figure 7.8. Therefore DD circuits take more transitions in a cycle: the SAM takes 18 transitions, but the STAM takes 22 transitions per cycle. In addition to the increase of transitions, some of the gates of the SAM are replaced with slower gates. For example, the inverters in a normal STAM are replaced with cross-coupled C-elements. Additionally, the average loads of gates also increase, because the average length of wires of the STAM is 1.2 times longer than that of the SAM, and there are more forking of signals (e.g., duplicated gates in a DD circuit need to drive two cross-coupled C-elements).

The area penalty of major units is about a factor of three, not only because transistors are duplicated with cross-coupled C-elements, but also because the complexity of wiring increases. For example, the *PCUNIT* has about 56,500 transistors while a corresponding normal *PCUNIT*, which does not employ error tolerant methods, has about 19,500 transistors; the area of the layout increases by a factor of three or so. The penalty on the energy consumption is also about a factor of between two and three, since the average loads of gates also increase in addition to duplicated transistors. Because of the throughput penalty and the energy penalty, adding error tolerance needs the increase of Et^2 by a factor of 7. The penalties are summarized in Table 7.1 where the speed of a system is measured in a unit of million instructions per second (MIPS).

Comparing performance and cost of microprocessors designed in different architectures is difficult. Instead of finding synchronous competitors of the STAM, we compare existing radiation-hardened MIPS-like microprocessors with a soft-error tolerant MiniMIPS whose performance is estimated from the performance of the MiniMIPS [67]. In fact, radiation-hardened (also known as rad-hard) circuits are designed in order to resist malfunctions such as soft errors and dose effects caused by high-energy subatomic particles and electromag-

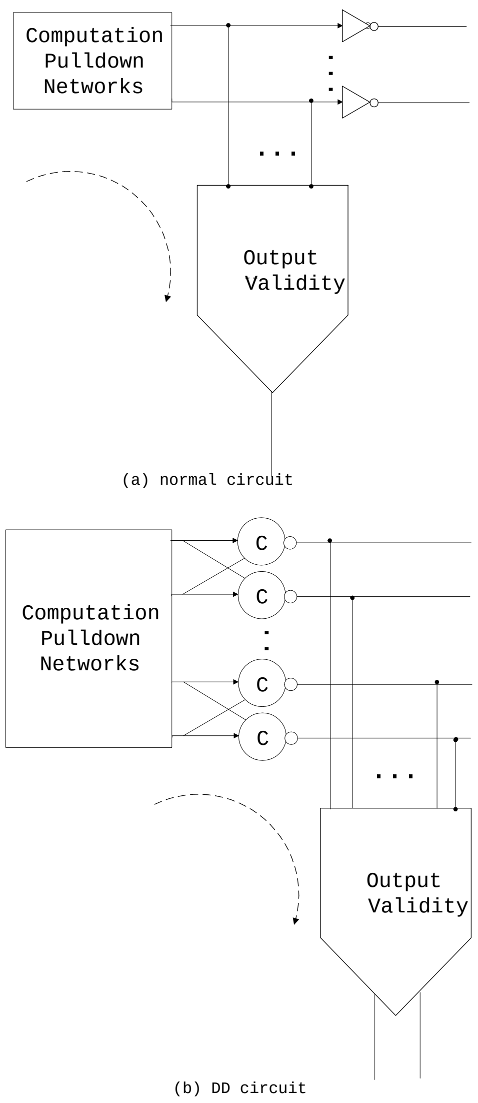


Figure 7.8: *Compared with a Non-DD Circuit, a DD Circuit Requires More Transitions in a Computation Cycle. A circuit for output-channel validity takes its inputs from function blocks, but a corresponding DD circuit takes its inputs from cross-coupled C-elements after function blocks.*

netic radiation. Since a soft-error tolerant asynchronous circuit also resists them, comparing a DD MiniMIPS with rad-hard circuits can give rough ideas on how good or bad our scheme is. There are a few known rad-hard microprocessors. AT697E, which uses triple modular redundancy and error correcting codes, is a rad-hard 32-bit RISC embedded processor based on the SPARC V8 architecture [68]. It is manufactured using the Atmel 0.18-μm CMOS process, which has been especially designed for space. Mongoose-V is a rad-hard MIPS R3000 32-bit microprocessor that is fabricated in 1.0-μm CMOS Silicon-on-Insulator [69].

Table 7.1: Performance Comparison between Simple Asynchronous Microprocessor and Its Corresponding Soft-error Tolerant Simple Asynchronous Microprocessor

	Energy (nJ/inst)	Speed (MIPS*)	Et^2 (10^{-24}Js^2)
SAM	2.0	280	0.025
STAM	5.1	170	0.176

* MIPS is million instructions per second

Table 7.2: Performance Comparison with Synchronous Competitors

Processor	Energy (nJ/inst)	Speed (MIPS*)	Et^2 (10^{-24}Js^2)
0.18-μm DD MiniMIPS (scaled)	2.1	246	0.034
0.18-μm Rad-hard AT697E	8	86	1.08
0.18-μm Rad-hard Mongoose V (scaled)	0.5	50	0.2
0.35-μm Rad-hard Sandia Pentium	35	200	0.875
0.25-μm Rad-hard RAD 750	-	260	-

* MIPS is million instructions per second

Since Mongoose-V is designed in the old technology, it is necessary to estimate the performance of the processor in 0.18-μm CMOS technology. If the constant E-field scaling is assumed, delay is reduced by the scale factor (i.e., 0.18) and the energy is reduced by the scale factor cubed, which gives us the rough estimation. Although most of the numbers are estimated, the soft-error tolerant MiniMIPS outperforms its synchronous competitors by a factor of 6 or more in the Et^2 metric. The comparison between the microprocessors is shown in Table 7.2. Besides these microprocessors, there exists a rad-hard Pentium processor of the Sandia National Labs, and RAD750, which implements the advanced features in the PowerPC 750 and adds radiation hardening [70]. Their performances are also shown in the table for reference purpose.

7.4 Summary

Here we demonstrated the method by designing a soft-error tolerant asynchronous microprocessor (STAM), which is a simple 32-bit RISC microprocessor. Using a synthesis flow, we have finished the partially wired layout of the STAM in TSMC 0.18-μm CMOS technology. The STAM can run at 170 MHz. The soft-error tolerance of the STAM has also been verified in digital testing and analog testing.

Chapter 8

Conclusion

8.1 Lessons Learned

QDI circuits are vulnerable to soft errors, which may cause incorrect computations or cause deadlock. The duplicated double-checking (DD) scheme provides a simple way to make a QDI circuit soft-error tolerant, and we have shown how DD QDI circuits can tolerate soft errors. However the DD design requires that two copies of a component provide the same outputs when the same inputs are given. Most of the components of QDI circuits satisfy the condition with the exception of arbiters. To address this, we introduced a protection circuit, called a mutual excluder to adapt an arbiter for use with DD circuits. Additionally a mutual excluder is also used to design an error tolerant memory using a Hamming code. While a Hamming-coded array is suitable for a large memory such as a data memory, an array of dual interlocked cells (DICEs) is suitable for a small memory such as a register file.

Although the DD scheme uses a repetition code for the channels of CHP processes, other error correcting codes (ECCs), such as parity codes, can be used for channel communications. In a few cases, non-repetition codes are advantageous, but it turned out that the DD scheme, which is based on repetition codes, generally generates more efficient circuits because the symmetry of repetition codes allows us to simplify the circuits.

We were able to demonstrate the feasibility of designing a complex system of soft-error tolerant QDI circuits through designing a soft-error tolerant simple asynchronous microprocessor (STAM), whose architecture defines a 32-bit RISC instruction set. Most parts of the STAM are based on the DD scheme. Using the automated tools, we have completed the partially-wired layout of the STAM. SPICE simulations show that the STAM in 0.18-μm CMOS technology can run at 170 MHz. The error tolerance of the STAM has been tested in a few different manners. The STAM can tolerate injected errors unless multiple errors occur on one set of cross-coupled variables. An error may cause a delay due to the time it takes to restore from the error, but QDI circuits can absorb the delay without affecting the correctness of computations. Meanwhile the cost in throughput is about 30% to 40%, and the area penalty is between a factor of two and a factor of three.

Hence this work has demonstrated a complete method for designing a system of soft-error tolerant QDI circuits.

8.2 Future Work

It is desirable to improve and expand the present method of error tolerant asynchronous circuit designs. Although a preliminary idea of designing fault tolerant reconfigurable circuits such as a FPGA was presented, it can be expanded in order to design defect tolerant DD circuits. That is, if a hard error occurs in a defect tolerant reconfigurable DD system, then the system, which tolerates soft errors, can be designed to reconfigure itself to bypass the hard error.

Furthermore it is desirable to find a way to apply the robustness of asynchronous circuits to nano-scale circuits. Designs at this scale will experience much higher fault and defect rates than conventional circuits.

Appendix A

Acronym Glossary

The acronyms that are used in the thesis are described.

Acronym	**Meaning**
CI	checked-in
CO	checked-out
CHP	communicating hardware processes
CC	completion checker
DI	delay-insensitive
DIML	delay-insensitive minterm logic
DICE	dual interlocked cell
DD	duplicated double-checking
DSET	duplicated soft-error tolerant
ECC	error correcting code
EDDI	error detecting delay-insensitive
FIT	failure in time
FPGA	field programmable gate arrays
HSE	handshaking expansion
PCHB	precharged half buffer
PCFB	precharged full buffer
PR	production rule
PRS	production-rule set
PC	program counter
SOI	silicon-on-insulator

SPAM	simple pulsed asynchronous microprocessor
SEL	single-event latch-up
SEU	single-event upset
STAM	soft-error tolerant simple asynchronous microprocessor
SOP	sum-of-products
QDI	quasi delay-insensitive
TMR	triple modular redundancy
WCHB	weak-conditioned half buffer

Appendix B

Controller in Memory Unit

A controller in a memory unit generates signals for precharging bit lines and acknowledgment signals for the input channels *DATA_IN*, *ADDR*, and *RW*. Its specfiction in HSE is as follows:

```
*[[$Addrv$];
 [$RW.Read \wedge DATA\_OUT.e \longrightarrow Decoder.en\uparrow$;
                        [$Bitlinev \wedge Wordlinev$];
                        $ADDR.e\downarrow,\ rgo\uparrow,\ Decoder.en\downarrow$;
                        [$\neg DATA\_OUT.e \wedge \neg Addrv \wedge \neg Wordlinev$]; $pchg\downarrow,\ rgo\downarrow$;
                        [$DATA\_OUT.e$];
 ▯$RW.Write \longrightarrow wgo\uparrow$;
             [$DATA\_INv \wedge Bitlinev$]; $Decoder.en\uparrow$
             [$Wordlinev$]; $wgo\downarrow, ADDR.e\downarrow, DATA\_IN.e\downarrow, Decoder.en\downarrow$;
             [$\neg DATA\_INv \wedge \neg Wordlinev \wedge \neg Addrv$]; $DATA\_IN.e\uparrow, pchg\downarrow$;
 ];
 [$\neg Bitlinev$]; $pchg\uparrow$; $ADDR.e\uparrow$
 ]
```

A corresponding CMOS-implementable PRS is as follows:

$Bitlinev \rightarrow \_Bitlinev\downarrow$

$\neg Bitlinev \rightarrow \_Bitlinev\uparrow$

$Wordlinev \rightarrow \_Wordlinev\downarrow$

$\neg Wordlinev \rightarrow \_Wordlinev\uparrow$

$$\begin{aligned}
\_Wordlinev \wedge Addrv &\to dgo{\uparrow} \\
\neg\_Wordlinev \wedge \neg Addrv &\to dgo{\downarrow}
\end{aligned}$$

$$\begin{aligned}
dgo \wedge pchg \wedge (RW.Read \wedge DATA\_OUT.e \vee Bitlinev \wedge RW.Write) &\to \_Decoder.en{\downarrow} \\
\neg dgo \wedge \neg\_Bitlinev &\to \_Decoder.en{\uparrow}
\end{aligned}$$

$$\begin{aligned}
\_Decoder.en &\to Decoder.en{\downarrow} \\
\neg\_Decoder.en &\to Decoder.en{\uparrow}
\end{aligned}$$

$$\begin{aligned}
(\neg\_rgo \vee \neg\_wgo) \wedge \neg\_Bitlinev &\to rwgo{\uparrow} \\
(\_rgo \wedge \_wgo) \wedge \_Bitlinev &\to rwgo{\downarrow}
\end{aligned}$$

$$\begin{aligned}
rwgo &\to \_rwgo{\downarrow} \\
\neg rwgo &\to \_rwgo{\uparrow}
\end{aligned}$$

$$\begin{aligned}
\neg\_Wordlinev \wedge \neg\_rwgo &\to \_ADDR.e{\uparrow} \\
\_rwgo &\to \_ADDR.e{\downarrow}
\end{aligned}$$

$$\begin{aligned}
\_ADDR.e &\to ADDR.e{\downarrow} \\
\neg\_ADDR.e &\to ADDR.e{\uparrow}
\end{aligned}$$

$$\begin{aligned}
RW.Read &\to \_RW.Read{\downarrow} \\
\neg RW.Read &\to \_RW.Read{\uparrow}
\end{aligned}$$

$$\begin{aligned}
DATA\_OUT.e &\to \_DATA\_OUT.e{\downarrow} \\
\neg DATA\_OUT.e &\to \_DATA\_OUT.e{\uparrow}
\end{aligned}$$

$$\begin{aligned}
\_DATA\_OUT.e &\to \_\_DATA\_OUT.e{\downarrow} \\
\neg\_DATA\_OUT.e &\to \_\_DATA\_OUT.e{\uparrow}
\end{aligned}$$

$$\begin{aligned}
\neg\_RW.Read \wedge \neg\_Bitlinev &\to \_rgo{\downarrow} \\
\_DATA\_OUT.e \wedge \_RW.Read \wedge \_Wordlinev &\to \_rgo{\uparrow}
\end{aligned}$$

$$(\neg\_\_DATA\_OUT.e \vee \neg DATA\_IN.e \wedge \neg DATA\_INv) \rightarrow done\uparrow$$
$$(\_\_DATA\_OUT.e \wedge DATA\_IN.e) \rightarrow done\downarrow$$

$$\neg\_Bitlinev \wedge \neg Addrv \rightarrow badrv\uparrow$$
$$\_Bitlinev \rightarrow badrv\downarrow$$

$$badrv \wedge done \wedge \_Wordlinev \rightarrow pchg\downarrow$$
$$\neg badrv \wedge \neg done \rightarrow pchg\uparrow$$

$$Addrv \wedge DATA\_IN.e \wedge pchg \wedge RW.Write \wedge DATA\_OUT.e \rightarrow \_wg\mathrm{o}\downarrow$$
$$\neg Addrv \wedge \neg DATA\_IN.e \rightarrow \_wg\mathrm{o}\uparrow$$

$$\_wg\mathrm{o} \rightarrow wg\mathrm{o}\downarrow$$
$$\neg\_wg\mathrm{o} \rightarrow wg\mathrm{o}\uparrow$$

$$DATA\_INv \wedge wg\mathrm{o} \rightarrow dinvwg\mathrm{o}v\uparrow$$
$$\neg DATA\_INv \wedge \neg wg\mathrm{o} \rightarrow dinvwg\mathrm{o}v\downarrow$$

$$dinvwg\mathrm{o}v \wedge Bitlinev \rightarrow DATA\_IN.e\downarrow$$
$$\neg dinvwg\mathrm{o}v \wedge \neg pchg \rightarrow DATA\_IN.e\uparrow$$

Appendix C

Upper Bound of the Size of Code

Given the length n, the size of an EDDI code is bounded, as follows. If C is an r-error detecting DI code, then

$$|C| \leq \frac{\begin{pmatrix} n+r \\ \lfloor \frac{n+r}{2} \rfloor \end{pmatrix}}{\begin{pmatrix} n+r \\ r \end{pmatrix}} \tag{C.1}$$

holds.

Proof: In case $|X - Y| > 0$ and $|Y - X| > 0$, C is a Sperner code, and $|C| \leq \begin{pmatrix} n+1 \\ \lfloor \frac{n}{2} \rfloor \end{pmatrix}$. Sperner's theorem can be easily proven by the Lubell-Yamamoto-Meshalkin inequality (more commonly known as the LYM inequality), widely used in combinatorial mathematics. Here we are going to prove our theorem using Lubell's approach [43].

Let us begin by proving the theorem in case of $r = 1$, and then generalize the proof. Each maximal chain ordered by inclusion in $power - set$ (i.e., set of all subsets of $\{1, ..., n\}$) intersects at most one codeword in C. (e.g., if $n = 3$, then a maximal chain is $\{1\} - \{1, 3\} - \{1, 2, 3\}$.) Let a_k denote the number of k-sets, sets with size k in C. We can generate k (k - 1)-sets out of a k-set by removing one element. For example, three 2-sets $\{1, 2\}, \{1, 3\}, \{2, 3\}$ come from a 3-set $\{1, 2, 3\}$. All $(k - 1)$-sets from k-sets in C are different from each other. If $Y_k \neq X_k$ in C, and a $(k - 1)$-set x_{k-1} out of X_k and y_{k-1} out of Y_k in C are the same, then $|X_k - Y_k| = |x_{k-1} \cup \{x\} - y_{k-1} \cup \{y\}| = |\{x\}| = 1$, which contradicts the condition that $|X - Y| > r = 1$ and $|Y - X| > r = 1$.

We can construct a DI C', which consists of codewords generated from C by removing one element in each codeword for every possible choice. That is, each codeword, c, in C can generate $|c|$ codewords with size $|c|-1$ in C'. Let $b_k = (k+1)a_{k+1}$ denote the number of k-sets in C'. Then every maximal chain meets C' in at most one element, say, k-set. (If not, we have $X'_{k-1} \subset Y'_{l-1}$ or $X'_{k-1} \supset Y'_{l-1}$ for X'_{k-1} and Y'_{l-1} in C' which is generated from X_k and Y_l. And it implies that $|X_k - Y_l| = 1$ or $|Y_l - X_k| = 1$, which contradicts the condition.)

For $i = k$ to 1, there are i choices of an $(i-1)$-set out of each i-set chosen previously. Thus, in total, this k-set is contained in $k!$ chains of subsets. Likewise, for $i = k$ to n, there are $n-i$ choices of an $(i+1)$-set on top of each i-set chosen previously. Thus, in total, it is contained in $(n-k)!$ chains of supersets. So each of b_k k-sets meets $k!(n-k)!$ maximal chains. Counting the number of maximal chains meeting k-sets of all possible sizes from 0 to n, we obtain at most $\sum_{k=0}^{n} b_k k!(n-k)! = \sum_{k=0}^{n} a_k k(k-1)!(n-(k-1))!$ maximal chains, which cannot be greater than $n!$, the maximum number of all possible maximal chains. Accordingly, we obtain the following:

$$\begin{aligned} n! &\geq \sum_{k=0}^{n} a_k k(k-1)!(n-(k-1))! \\ \frac{1}{n+1} &\geq \sum_{k=0}^{n} a_k \frac{k!(n+1-k)!}{(n+1)!} \\ &= \sum_{k=0}^{n} \frac{a_k}{\begin{pmatrix} n+1 \\ k \end{pmatrix}} \\ &\geq \sum_{k=0}^{n} \frac{a_k}{\begin{pmatrix} n+1 \\ \lfloor \frac{n+1}{2} \rfloor \end{pmatrix}} \end{aligned}$$

Finally we have an upper bound for the maximum size of a one-error-detection DI code, which is as follows:

$$|C| = \sum_{k=0}^{n} a_k \leq \frac{\begin{pmatrix} n+1 \\ \lfloor \frac{n+1}{2} \rfloor \end{pmatrix}}{n+1}$$

In the same manner, we can prove that r is set as an arbitrary value. All $(k-r)$-sets

Table C.1: One-error Detecting Delay-insensitive Code

	# of codewords	Upper bound	One-error Correction Binary Code
4	2	2	
5	2	3.33	
6	4	5	
7	7	8.75	16
8	14	14	
9	14	25.2	40
10	28	42	
11	44	77	144
12	132	132	
13	132	245.1	
14	252	429	

generated from k-sets in r-error-detection DI C are different from each other. That is, we can construct a DI C', which consists of codewords generated from C by removing r elements in each codeword of every possible choice. And then we have $b_k = \left(\begin{array}{c} k+r \\ r \end{array} \right) a_{k+r}$, the number of k-sets in C'. Then every maximal chain meets C' in at most one element, say, k-set. In the same manner, we can have r-error-detection DI code (I, C): $|X - Y| > r$ and $|Y - X| > r$ for all X,$Y \in C$. Note that a code whose size satisfies the equality does not always exist, as shown in Table C.1.

■

Appendix D

Minimizing Decomposed Function Block

In Chapter 6, we decompose a function block and add cross-coupled C-elements in order to satisfy **Cond E2**; we have shown one way of decomposing a function block. Here we show alternative ways of decomposing a function block.

A PRS of the function block in $*[L?l; R!f(l)]$ using a linear DI code is the following:

$$\begin{array}{lll}
\ldots & & \\
en_1 \wedge \kappa_i^0 & \rightarrow & w_i.0\uparrow \\
\neg en_1 & \rightarrow & w_i.0\downarrow \\
en_2 \wedge \mu_i^0 & \rightarrow & z_i.0\uparrow \\
\neg en_2 & \rightarrow & z_i.0\downarrow \\
 & & \\
w_i.0 \wedge z_i.0 & \rightarrow & r_i.0\uparrow \\
\neg w_i.0 \wedge \neg z_i.0 & \rightarrow & r_i.0\downarrow \\
 & & \\
en_1 \wedge \kappa_i^1 & \rightarrow & w_i.1\uparrow \\
\neg en_1 & \rightarrow & w_i.1\downarrow \\
en_2 \wedge \mu_i^1 & \rightarrow & z_i.1\uparrow \\
\neg en_2 & \rightarrow & z_i.1\downarrow \\
 & & \\
w_i.1 \wedge z_i.1 & \rightarrow & r_i.1\uparrow \\
\neg w_i.1 \wedge \neg z_i.1 & \rightarrow & r_i.1\downarrow \\
\ldots & &
\end{array}$$

The function f of the process is defined as $f(x_1, ...x_k) = (f_1(x_1, ..., x_k), ..., f_k(x_1, ..., x_k))$ where x_i is a boolean variable, and the function is defined only for codewords satisfying that $\mathbf{u}G = (x_1, ..., x_n)$ and $\mathbf{u} = (u_1, ...u_k)$ is a message. The remaining step is to represent the function in the linear DI code, that is, to find the boolean expressions of κ_i^j and μ_i^j satisfying the condition of error-tolerance.

Let us define boolean functions $k_i^j(x_1, ...x_k)$ and $m_i^j(x_1, ...x_k)$, which are associated with κ_i^j and μ_i^j, as the following:

$$\begin{aligned} k_i^0(x_1, ..., x_n) &= \neg f_i(u) \\ k_i^1(x_1, ..., x_n) &= f_i(u) \\ m_i^0(x_1, ..., x_n) &= \neg f_i(u) \\ m_i^1(x_1, ..., x_n) &= f_i(u) \end{aligned}$$

where $\mathbf{u}G = (x_1, ..., x_n)$ is satisfied. But the definition is incomplete because the functions are not defined for invalid codewords. As a simple choice, all invalid codewords can be mapped as `false`. For example, let us consider a process using a $[4, 2]$ DI code whose generator matrix is

$$G = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}.$$

If f is an identity function and all invalid codewords are mapped as `false`, then its k_1^1 and m_1^1 are the following:

$$\begin{aligned} k_1^1(x_1, x_2, x_3, x_4) &= x_1 \wedge x_2 \wedge (x_3 \wedge x_4 \vee \neg x_3 \wedge \neg x_4) \\ m_1^1(x_1, x_2, x_3, x_4) &= x_1 \wedge x_2 \wedge (x_3 \wedge x_4 \vee \neg x_3 \wedge \neg x_4). \end{aligned}$$

Then we have the corresponding one-of-two code representation of κ_i^j and μ_i^j such as

$$\begin{aligned} \kappa_1^1 &= x_1.1 \wedge x_3.1 \wedge (x_2.1 \wedge x_4.1 \vee x_2.0 \wedge x_4.0) \\ \mu_1^1 &= x_1.1 \wedge x_3.1 \wedge (x_2.1 \wedge x_4.1 \vee x_2.0 \wedge x_4.0). \end{aligned}$$

Considering an obvious minimization of the guards such as $\kappa_1^1 = x_1.1$ and $\mu_1^1 = x_3.1$, we shall note the inefficiency of assigning `false` to all invalid codewords. Instead, the value of one of a pair of functions k_i^j and m_i^j is set as `false` at an invalid codeword, and the value of the other is set as `dont-care`, which can be either `true` or `false` during minimization. That is, we shall guarantee that at least one of κ_i^j and μ_i^j is evaluated to be `false`, which would be sufficient enough to prevent double-checking C-elements from firing primary outputs unexpectedly. In addition, the function values at an invalid codeword, whose distance from other codewords is more than one, can be assigned to be `dont-care` since the invalid codeword cannot be observed when only one error occurs.

The suggested method of using `dont-care` produces better minimization results. For example, a function $f_1(u_1, u_2)$ is defined as $u_1 \oplus u_2$ before applying a linear DI code. In case of using the parity $[3, 2]$ code, k_1^1 and m_1^1 have the following truth tables:

k_1^1	
codeword	output
000	0
011	1
101	1
110	0
invalid codeword	output
001	0
010	-
100	-
111	0

m_1^1	
codeword	output
000	0
011	1
101	1
110	0
invalid codeword	output
001	-
010	0
100	0
111	-

From these, we have minimal representations of two functions: $k_1^1 = x_1 \wedge \neg x_2 \vee \neg x_1 \wedge x_2$ and $m_1^1 = x_3$, which correspond to

$$\begin{aligned} \kappa_1^1 &= x_1.1 \wedge x_2.0 \vee x_1.0 \wedge x_2.1 \\ \mu_1^1 &= x_3.1, \end{aligned}$$

as shown in Figure 6.7. On the other hand, k_1^1 and m_1^1 of the $[4, 2]$ code are the following:

codeword	k_1^1
0000	0
0011	1
1100	1
1111	0
invalid codeword (d=1)	
0100	0
0001	0
1011	0
1101	0
0111	-
0010	-
1000	-
1101	-
invalid codeword (d=2)	
1001	-
0101	-
1010	-
0110	-

codeword	m_1^1
0000	0
0011	1
1100	1
1111	0
invalid codeword (d=1)	
0100	-
0001	-
1011	-
1101	-
0111	0
0010	0
1000	0
1101	0
invalid codeword (d=2)	
1001	-
0101	-
1010	-
0110	-

After minimizing the representation, $k_1^1 = x_1 \wedge \neg x_2 \vee \neg x_1 \wedge x_2$ and $m_1^1 = x_3 \wedge \neg x_4 \vee \neg x_3 \wedge x_4$, which correspond to

$$\begin{aligned} \kappa_1^1 &= x_1.1 \wedge x_2.0 \vee x_1.0 \wedge x_2.1 \\ \mu_1^1 &= x_3.1 \wedge x_4.0 \vee x_3.0 \wedge x_4.1, \end{aligned}$$

as shown in Figure 6.8. Note that we can simplify the function block of the repetition codes, because of the symmetry of the construction.

Although there are many ways of assigning `false` and `dont-care` for invalid codewords, there is a heuristic method in case of a systematic code. We shall find a minimal representation of a given function f, and the minimized form will be k_i^j. If the value of k_i^j at an invalid codeword is `true`, then the value of m_i^j at the invalid codeword is assigned to be `false`; otherwise it is assigned to be `dont-care`.

Appendix E

STAM Architecture

The STAM architecture defines eight general-purpose registers, `gpr[0]` through `gpr[7]`, of which `gpr[0]` is always read as zero, although it may be written by any instruction. The remaining state of the processor is the program counter `pc`.

All STAM instructions have the same format. The instruction format is a four-operand RISC format with three register operands and a single immediate operand. The opcode format has two fields, which are also the same across all instructions. These fields are the operation unit and the operation function. The operation "Y-mode," which determines the addressing mode used to conjure operand `opy`, is further defined in a fixed position in the instruction.

STAM instructions are 32 bits wide. Considering that a STAM instruction i as a 32-bit array of bits, we identify the fields of the instruction:

1. The `opcode` $= i[31 \ldots 27]$, further divided into:
   (a) The unit number `unit` $= i[31 \ldots 30]$.
   (b) The function `fxn` $= i[29 \ldots 27]$.
2. The Y-mode `ymode` $= i[26 \ldots 25]$.
3. The result register number `rz` $= i[24 \ldots 22]$.
4. The X-operand register number `rx` $= i[21 \ldots 19]$.
5. The Y-operand register number `ry` $= i[18 \ldots 16]$.
6. The immediate field `imm` $= i[15 \ldots 0]$.

The first operand, `opx`, is always the value of `gpr[`i`.rx]`. The second operand, `opy`, is computed from the value of `gpr[`i`.ry]` and the immediate field, depending on i`.ymode`.

Allowable values for i`.ymode` are as follows, where *sext* signifies sign extension:

i`.ymode` Mnemonic	Decimal value	Operand generated
`YMODE_REG`	0	`opy := gpr[`i`.ry]`
`YMODE_IMM`	1	`opy :=` *sext*`(`i`.imm)`
`YMODE_IMMSHIFT`	2	`opy :=` i`.imm` $<<$ `16`
`YMODE_REGIMM`	3	`opy := gpr[`i`.ry] +` *sext*`(`i`.imm)`

Operations are defined on two-complement numbers. There are no flags or condition codes. We divide the operations by four units.

All ALU operations take two operands and produce one result. The *bitwise_NOR* is included in the instruction set for the express purpose of computing the bitwise inverse of `opx` using a zero operand for `opy`.

Mnemonic	Name	i`.fxn`	Operation
`add`	Add	0	`opz := (opx + opy)`$_{31 \ldots 0}$
`sub`	Subtract	1	`opz := (opx - opy)`$_{31 \ldots 0}$
`nor`	NOR	4	`opz :=` *bitwise_NOR*`(opx,opy)`
`and`	AND	5	`opz :=` *bitwise_AND*`(opx,opy)`
`or`	OR	6	`opz :=` *bitwise_OR*`(opx,opy)`
`xor`	Exclusive OR	7	`opz :=` *bitwise_XOR*`(opx,opy)`

All branch operations unconditionally produce the same result as `opz`, namely the value of `pc`, right-shifted by two. Likewise, a branch taken will branch to the address denoted by `opy` left-shifted by two and incremented by one. The shifting avoids alignment errors.

The `hlt` instruction halts the processor. An external action, not defined within the architecture, is required to restart the machine.

Conditional branches branch according to the value of `opx`.

Mnemonic	Name	i.`fxn`	Branch if	Target
`hlt`	Halt	0	**true**	$\perp$
`beq`	Branch on Equal	1	`opx` $= 0$	`opy`$_{29\ldots0}\vert 00$
`bne`	Branch on Not Equal	2	`opx` $\neq 0$	`opy`$_{29\ldots0}\vert 00$
`bgt`	Branch on Greater Than	3	`opx` > 0	`opy`$_{29\ldots0}\vert 00$
`blt`	Branch on Less Than	4	`opx` < 0	`opy`$_{29\ldots0}\vert 00$
`ble`	Branch on Less or Equal	5	`opx` ≤ 0	`opy`$_{29\ldots0}\vert 00$
`bge`	Branch on Greater or Equal	6	`opx` ≥ 0	`opy`$_{29\ldots0}\vert 00$
`jmp`	Jump	7	**true**	`opy`$_{29\ldots0}\vert 00$

Only two memory operations are defined, load word, `lw`, and store word, `sw`. The address of the memory access is determined by `opy`. On a memory load, `opx` is ignored, whereas on a store, it becomes the value stored. A store returns `opy` (the computed address) as `opz`.

Mnemonic	Name	i.`fxn`	Operation
`lw`	Load Word	0	`opz := dmem[opy]`
`sw`	Store Word	4	`dmem[opy] := opx, opz := opy`

The STAM architecture defines a restricted shifter that is capable only of logical shifts. Arithmetic shifts must be simulated using `blt`. The STAM shifter can shift by one or eight. Shifts-by-eight are provided so that byte memory operations can proceed at a reasonable speed.

Mnemonic	Name	i.`fxn`	Operation
`sr1`	Shift Right by One	0	`opz :=` $0 \vert$ `opy`$_{31\ldots1}$
`sr8`	Shift Right by Eight	1	`opz :=` $00000000 \vert$ `opy`$_{31\ldots8}$
`sl1`	Shift Left by One	2	`opz :=` `opy`$_{30\ldots0} \vert 0$
`sl8`	Shift Left by Eight	3	`opz :=` `opy`$_{23\ldots0} \vert 00000000$

Appendix F

Arbitrated Branch

A rough sketch of the STAM with the arbitrated branch mechanism is shown as follows:

```
$STAM \equiv$    $valid\uparrow$;
         *[  $PCUNIT$ :  [$\neg\overline{DoBranch} \longrightarrow branched\downarrow, pc := "next\ \ pc"$
                         | $\overline{DoBranch} \longrightarrow branched\uparrow, pc := "branch\ \ pc", DoBranch?$
                        ];
             $IMEM$ :  $i := imem[pc]$;
           $DECODE$ :  $id := decode(i)$;
             $EXEC$ :  $"read\ \ operands"$;
                       $"execute\ \ instruction"$;
                       $b := "branch\ \ instruction?"$;
               $WB$ :  [$valid \vee branched \longrightarrow$ [$\neg b \longrightarrow valid\uparrow, "write\ \ results"$
                                                   []$b \longrightarrow valid\downarrow, DoBranch!$
                                                  ]
                       []$\neg valid \wedge \neg branched \longrightarrow skip$
                      ]
         ]
```

We introduce a channel *DoBranch* that has a one-place buffer that is used to notify *PCUNIT* of the presence of a branch instruction. When a branch occurs, *WB* inserts a token into this buffer without blocking. The *PCUNIT* polls the state of the buffer and if it finds a token in it, the token is removed and the *PCUNIT* sets the program counter to the branch target address, which is given by the *EXEC* unit. In the CHP notation, *PCUNIT* uses the probe of channel *DoBranch* to check if there is a token in the buffer. Naturally there is no dependency between $PCUNIT$ and $EXEC$ when $b =$`false`.

Since we do not make assumptions about the speed of buffers, the $PCUNIT$ may not immediately notice the presence of a token in the buffer. All we assume is that $PCUNIT$ will eventually notice the token and detect the branch. Therefore, $EXEC$ might execute instructions that are invalid, since the sequence of PC values could have changed. $EXEC$ only executes these invalid instructions after the branch token has been inserted into the $DoBranch$ buffer by WB, and before the branch is detected by $PCUNIT$. We introduce variable *branched* that is set to `true` when the branch is detected by $PCUNIT$, and variable *valid* that is set to `false` when the branch token is inserted into the buffer by WB. The STAM is therefore executing invalid instructions when *valid* is set to `false` and *branched* is also `false`.

The probe $\overline{DoBranch}$ becomes `true` eventually, causing the $PCUNIT$ to detect the presence of the branch token in the buffer. WB inserts a token into the buffer by performing an $DoBranch!$ communication when a branch instruction is encountered. Then the communication $DoBranch?$ removes the token from the buffer once an branch is detected, and $PCUNIT$ reads the target address from $EXEC$.

Appendix G

Designing CHP Process Using Synthesis Tool Flow

Here, the details of synthesizing the *SELECTOR*, which is a sub-process of the *PCUNIT*, according to the tool flow described in Chapter 7. It is desirable to decompose the *SELECTOR* vertically, which requires the distribution of signals of the control channels to the vertically decomposed processes. In the MiniMIPS, control signals for vertical decompositions are distributed to the datapath by a pipelined tree of copying processes: a control signal is copied to several copying processes at each stage, and each bit of datapath gets a control signal simultaneously from the leaves of the copying tree. On the other hand, if the throughput of a system is a primary concern (e.g., DSP applications), each bit of datapath is made to copy the received control signal at the same time as it performs its data computation, which is called bit skewing [71]. While the two approaches stand on each extreme, a compromise between the two approaches, called byte skewing, was used for the SPAM. In byte skewing, a control signal is simultaneously distributed within a byte, but the process of the next byte gets a copy of the control signal in the next time step, as shown in Figure G.1. After decomposing the *SELECTOR* vertically with byte skewing, we obtain the following:

$*[BC?bc;\quad BC'[0]!bc, ..., BC'[3]!bc, BC\_S[0]!bc]$

$*[BC\_S[0]?bc;\quad BC'[4]!bc, ..., BC'[7]!bc, BC\_S[1]!bc]$

$*[BC\_S[1]?bc;\quad BC'[8]!bc, ..., BC'[11]!bc, BC\_S[2]!bc]$

$*[BC\_S[2]?bc;\quad BC'[12]!bc, ..., BC'[15]!bc]$

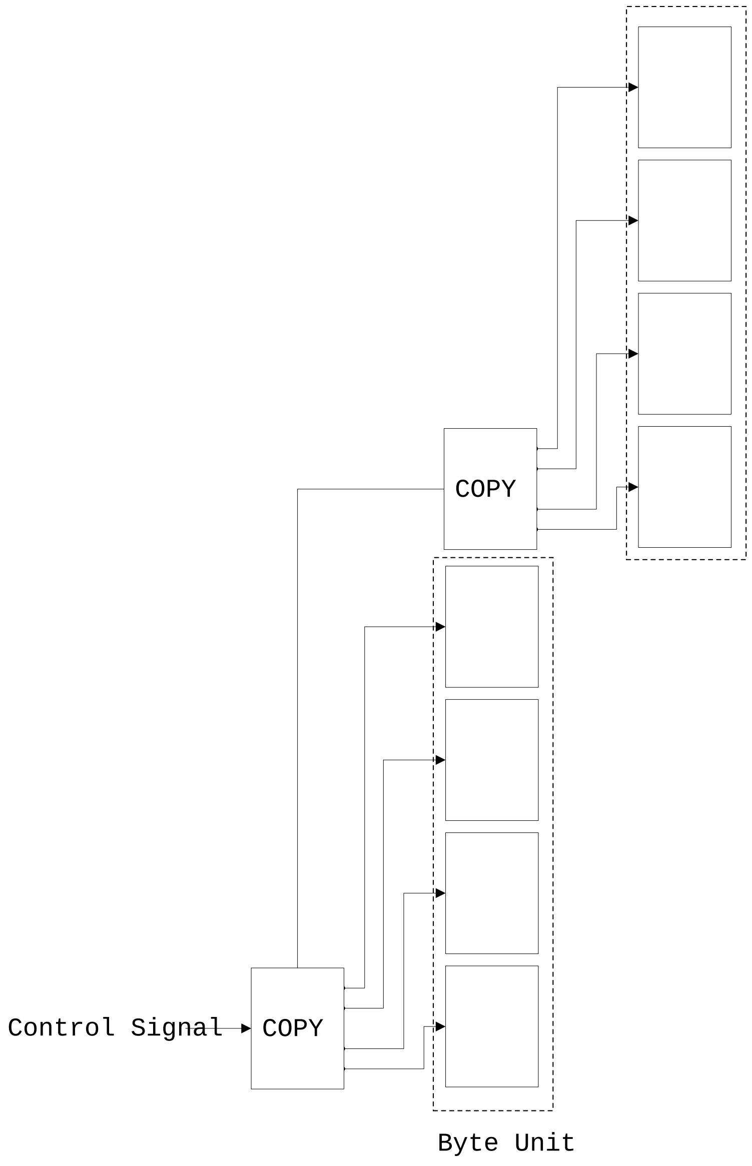


Figure G.1: *Byte-skewed Distribution of Control Signals.*

```
*[$BC'$[0]?$bc$,  $PCIncOut$[0]?$pc$[0],  $BranchTo$[0]?$branch\_to$[0]
  [ $bc$ = false $\longrightarrow$ $NewPC$[0]!$pc$[0]
  [] $bc$ = true $\longrightarrow$ $NewPC$[0]!$branch\_to$[0]
  ]
]
```

```
...
*[$BC'$[15]?$bc$,  $PCIncOut$[15]?$pc$[15],  $BranchTo$[15]?$branch\_to$[15]
  [ $bc$ = false $\longrightarrow$ $NewPC$[15]!$pc$[15]
  [] $bc$ = true $\longrightarrow$ $NewPC$[15]!$branch\_to$[15]
  ]
]
```

The channels $PCIncOut$[n], $BranchTo$[n], $NewPC$[n] are the vertically decomposed channels of 32-bit channels, and the channels BC[n] are copies of the control channel BC. A control signal of the BC is simultaneously distributed within a byte. In a 32-bit datapath, the most significant bit of the bit-skewing design experiences the latency of 32 time units; in contrast, the most significant bit of the byte-skewing design experiences a latency of only four time units. The byte skewing, which is applied to the STAM, produces a good trade-off between throughput and latency.

Although PL2 can describe only a restricted subset of CHP programs, the basic processes whose input channels and output channels are used at most once in each iteration, can be generally described in PL2 [60]. The syntax of PL2 is based on the CHP and C languages. PL2 programs do not specify the sequencing of a set of actions explicitly, but list the set in such a way that the actions are performed concurrently under certain conditions.

The PL2 level of description maintains the channel communications similar to CHP descriptions but introduces channel encodings such as e1of4, e1of2. For example, a PL2 description of a vertically decomposed process of $SELECTOR$ is the following:

```
  PROCESSFRAGMENT pc_sel(IN c:e1of2; IN incpc:e1of4;
                         IN genpc:e1of4; OUT newpc:e1of4) =
  BEGIN
TRUE -> c?, incpc?
|| c = 0 -> newpc!incpc
|| c = 1 -> newpc!genpc, genpc?
  END pc_sel
```

There is an auxiliary language which arranges PL2 processes in one collection:

```
  PROCESS pc_sel(  c, incpc, genpc, newpc : CHANNEL) =
  VAR
cpc_sel := pc_sel( c, incpc, genpc, newpc);
  END pc_sel
```

It is possible to combine several PL2 processes in one collection but having internal channels between the processes is not allowed. The following is a part of the CAST description:

```
 sypdefine pc_sel()(2xe1of(2) c; 2xe1of(4) incpc;
                  2xe1of(4) genpc; 2xe1of(4) newpc)
 {
node enG0a;
node enG0b;
2x_1of4 _newpc;
...
celem2 Cell_newpc_da[0](_newpc.da[0], _newpc.db[0], newpc.da[0]);
celem2 Cell_newpc_da[1](_newpc.da[1], _newpc.db[1], newpc.da[1]);
celem2 Cell_newpc_da[2](_newpc.da[2], _newpc.db[2], newpc.da[2]);
celem2 Cell_newpc_da[3](_newpc.da[3], _newpc.db[3], newpc.da[3]);
...
stackdefine newpc_pda()(node enG0a; 2xe1of(4) incpc; 2xe1of(4) genpc;
                       2xe1of(2) c; 2x_1of4 _newpc; 2xe1of(4) newpc)
  {sprs {
enG0a & newpc.ea & (incpc.da[0] & c.da[0] | genpc.da[0] & c.da[1])
                                                       -> _newpc.da[0]-
enG0a & newpc.ea & (genpc.da[1] & c.da[1] | incpc.da[1] & c.da[0])
                                                       -> _newpc.da[1]-
enG0a & newpc.ea & (incpc.da[2] & c.da[0] | genpc.da[2] & c.da[1])
                                                       -> _newpc.da[2]-
enG0a & newpc.ea & (genpc.da[3] & c.da[1] | incpc.da[3] & c.da[0])
                                                       -> _newpc.da[3]-
}}
newpc_pda() Cell__newpc_pda(enG0a, incpc, genpc, c, _newpc, newpc);
node _newpc_va0;
nor3 Cell__newpc_va0(newpc.da[0], newpc.da[1], newpc.da[2], _newpc_va0);
...
  }
```

The generated circuit is similar enough to a corresponding hand-designed circuit.

The final layout of the *PCUNIT* is a combination of generated transistor stacks, standard cells such as inverters, NANDs, C-elements, and so on.

For example, a part of the description of the floor plan of the *SELECTOR* is as follows:

```
HORIZ {
  VERT {
    CELL pcunit/pc_sel p(0) PAD_E;
    CELL pcunit/pc_sel p(1) PAD_E;
    CELL pcunit/pc_sel p(2) PAD_E;
    CELL pcunit/pc_sel p(3) PAD_E;
...
  }
...
  HGLUE 8 LAMBDA;
  CELL cells/cdist_2 ctrl PAD_N;
}
```

where `HORIZ` and `VERT` represent a horizontal box and a vertical box.

Appendix H

Checking Error Tolerance of PRS

`seucheck` is a 1500-line C program, which facilitates error-tolerance verification of a given PRS by exploring all possible states of the PRS with an error on each variable of the PRS. The input form of the program is as follows:

```
init Le 1
init L 0

init R 0
init Re 1

seu R
seu Le

Le & Re & L  -> R +
~Le & ~Re    -> R -

 L & R  -> Le -
~L &~R  -> Le +

input  Le  -> L +
input ~Le  -> L -

output  R  -> Re -
output ~R  -> Re +
```

In this input form, the values of variables are initialized first, and variables which experience an error are indicated. `seucheck` checks whether PRs associated with the keywords 'input' and 'output' fire the same number of times in one cycle of a computation. With the input, `seucheck` generates the following output:

```
# of init status : 14
Le flipping
 -> (0 , 7)
R flipping
 -> (0 , 7)
# of deadlock : 0
# of abnormality : 14
```

It shows that the given PRS can experience an abnormal computation (i.e., data can be missed or can be generated unexpectedly), but an error does not cause deadlock. Besides the output, the program generates more informative output files: a list of valid states of the given PRS of whose elements are states reachable from the initial state of the process by firing of PRs, and a list of execution paths, which are caused by an error on a variable, from each valid state. The first output is as follows:

```
(1001)  (0, 0):(0, 0) :
(1101)  (1, 0):(0, 0) : L+
(1111)  (1, 0):(0, 0) : L+ R+
(0111)  (1, 0):(0, 0) : L+ R+ Le-
(0011)  (1, 1):(0, 0) : L+ R+ Le- L-
(0010)  (1, 1):(1, 0) : L+ R+ Le- L- Re-
(0000)  (1, 1):(1, 0) : L+ R+ Le- L- Re- R-
(1000)  (1, 1):(1, 0) : L+ R+ Le- L- Re- R- Le+
(1100)  (2, 1):(1, 0) : L+ R+ Le- L- Re- R- Le+ L+
(0001)  (0, 0):(0, 0) : L+ R+ Le- L- Re- R- Re+
(0110)  (1, 0):(1, 0) : L+ R+ Le- Re-
(0100)  (1, 0):(1, 0) : L+ R+ Le- Re- R-
```

```
(0101)  (1, 0):(1, 1) : L+ R+ Le- Re- R- Re+
(1110)  (1, 0):(1, 0) : L+ R+ Re-
```

It shows 14 valid states. The first column has a list of the representation of valid states; the second column has a list of the number of firings of input and output PRs. The execution path from the initial state is shown after the two columns. The second output is as follows:

```
...

Le:<(1001)...(0010)>SEU<(1010)...(1001)> Invalid State
Abnormal : (1, 1):(0, 0) - L+  to the state : (1110)

Le:<(1001)...(0011)>SEU<(1011)...(1001)> Invalid State

Le:<(1001)...(0100)>SEU<(1100)...(1001)>

...
```

This output information shows that if an error on Le occurs in the state (0010), the error turns a valid state into an invalid state, and causes the PR of $L\uparrow$ to fire. As a result, a set of firings of the input PRs (i.e., $L\uparrow$ and $L\downarrow$) are acknowledged before a set of firings of the output PRs occurs: an output communication is missed, owing to the error. By reading the second output, we can figure out when and how an error causes deadlock or an abnormal computation.

Bibliography

[1] Alain J. Martin. Synthesis of asynchronous VLSI circuits. Technical Report CS-TR-92-03, Department of Computer Science, Caltech, 1990.

[2] Alain J. Martin and et al. The Lutonium: a sub-nanojoule asynchronous 8051 microcontroller. In *Proceedings of 9th International Symposium on Asynchronous Circuits and Systems*, Mar 2003.

[3] Semiconductor Industry Association. *International Technology Roadmap for Semiconductors (ITRS)*, 2005.

[4] Andrew M. Lines. Pipelined asynchronous circuits. Master's thesis, Department of Computer Science, Caltech, 1995.

[5] B.T. Murray and J.P. Hayes. Testing ICs: getting to the core of the problem. *Computer*, 29:32–38, Nov 1996.

[6] Pieter Johannes Hazewindus. *Testing Delay-Insensitive Circuits.* PhD thesis, Department of Computer Science, Caltech, 1992.

[7] Kenneth A. Label. Single event effect criticality analysis. Technical report, NASA, Jun 1996.

[8] D. Pierce and P.G. Brusius. Electromigration: A review. *Microelectronics Reliability*, 37(7), 1997.

[9] T.C. May and M.H. Woods. A new physical mechanism for soft errors in dynamic memories. In *Proceedings of 16th International Reliability Physics Symposium*, Apr 1979.

[10] Sherra E. Kerns. *Transient-ionization and Single-Event Phenomena*, chapter 9. John Wiley & Sons, 1989.

[11] J.F. Ziegler and W.A. Landford. Effect of cosmic rays on computer machines. *Science*, 206, 1979.

[12] K. Aingaran, F. Klass, C. M. Kim, C. Amir, J. Mitra, E. You, J. Mohd, and S. K. Dong. Coupling noise analysis for VLSI and ULSI circuits. In *Proceedings of IEEE ISQED 2000*, pages 485–489, Mar 2000.

[13] K. L. Shepard and V. Narayanan. Noise in deep submicron digital design. In *Proceedings of IEEE/ACM ICCAD-96*, pages 524–531, Nov 1996.

[14] P. Larsson. Power supply noise in future IC's: a crystal ball reading. In *Proceedings of the IEEE, Custom Integrated Circuits*, 1999.

[15] R. Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Transactions on Device and Materials Reliability*, 5(3):305–316, Sep 2005.

[16] R. Baumann. The impact of technology scaling on soft error rate performance and limits to the efficacy of error correction. In *Proceedings of IEEE International Electron Devices Meeting Tech. Dig.*, pages 329–332, Dec 2002.

[17] P. Shivakumar, M. Kistler, S.S. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proceedings of International Conference on Dependable Systems and Networks*, pages 389–398, 2002.

[18] G. L. Ries, G.S. Choi, and R. K. Iyer. Device-level transient fault modeling. In *Proceedings of International Symposium on Fault-Tolerant Computing*, 1994.

[19] F. Irom, F. H. Farmanesh, A. H. Johnston, G. M. Swift, and D. G. Millward. Single-event upset in commercial silicon-on-insulator PowerPC microprocessors. *IEEE Transactions on Nuclear Science*, 49(6):3148–3155, Dec 2002.

[20] J. von Neumann. Probabilistic logics and synthesis of reliable organisms from unreliable components. In C. Shannon and J. McCarthy, editors, *Automata Studies*, pages 43–98. Princeton University Press, 1956.

[21] C.L. Chen and M.Y. Hsiao. Error-correcting codes for semiconductor memory applications: A state-of-the-art review. *IBM Journal of Research and Development*, 28(2), 1984.

[22] Fernanda G. Lima, Érika Cota, Luigi Carro, Marcelo Lubaszewski, Ricardo Reis, Raoul Velazco, and Sana Rezgui. Designing a radiation hardened 8051-like micro-controller. In *Proceedings of SBCCI Conference*, 2000.

[23] D. Wiseman, J. Canaris, S. Whitaker, J. Vembrux, K. Cameron, K. Arave, L. Arave, N. Liu, and K. Liu. Design and testing of SEU/SEL immune memory and logic circuits in a commercial CMOS process. In *Radiation Effects Data Workshop*, 1993.

[24] T. Calin, M. Nicolaidis, and R. Velazco. Upset hardened memory design for submicron CMOS technology. *IEEE Transactions on Nuclear Science*, 43(6), Dec 1996.

[25] M. Nicolaidis. Time redundancy based soft-error tolerance to rescue nanometer technologies. In *Proceedings of 17th IEEE VLSI Test Symposium*, pages 86–94, Apr 1999.

[26] K. Hass, L. Gambles, B. Walker, and M. Zampaglione. Mitigating single event upsets from combinational logic. In *Proceedings of 7th NASA Symposium on VLSI Design*, 1998.

[27] D.K. Pradhan. *Fault-Tolerant Computing System Design*. Prentice-Hall, Upper Saddle River, NJ, 1996.

[28] P. Cheynet, B. Nicolescu, R. Velazco, M. Rebaudengo, M. Sonza Reorda, and M. Violante. Experimentally evaluating an automatic approach for generating safety-critical software with respect to transient errors. *IEEE Transactions on Nuclear Science*, 47(6):2231–2236, Dec 2000.

[29] Nahmsuk Oh, Subhasish Mitra, and Edward J. McCluskey. ED4I: Error detection by diverse data and duplicated instructions. *IEEE Transactions on Computers*, 51(2):180–199, 2002.

[30] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, and M. Violante. Soft-error detection using control flow assertions. In *DFT '03: Proceedings of the 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 581–582, Washington, DC, USA, 2003. IEEE Computer Society.

[31] Steven K. Reinhardt and Shubhendu S. Mukherjee. Transient fault detection via simultaneous multithreading. In *ISCA '00: Proceedings of the 27th Annual International*

Symposium on Computer Architecture, pages 25–36, New York, NY, USA, 2000. ACM Press.

[32] Michael Nicolaidis. Design for soft error mitigation. *IEEE Transactions on Device and Materials Reliability*, 5(3):405–418, Sep 2005.

[33] Subhasish Mitra and Edward J. McCluskey. Which concurrent error detection scheme to choose? In *ITC '00: Proceedings of the 2000 IEEE International Test Conference*, page 985, Washington, DC, USA, 2000. IEEE Computer Society.

[34] T. Verdel and Y. Makris. Duplication-based concurrent error detection in asynchronous circuits: shortcomings and remedies. In *Proceedings of 17th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 345–353, 2002.

[35] C. LaFrieda and R. Manohar. Robust fault detection and tolerance in quasi delay-insensitive circuits. In *Proceedings of International Conference on Dependable Systems and Networks*, 2004.

[36] Y. Monnet, M. Renaudin, and R. Leveugle. Designing resistant circuits against malicious faults injections using asynchronous logic. *IEEE Transactions on Computers*, 55(9), Sep 2006.

[37] Song Peng and Rajit Manohar. Self-healing asynchronous arrays. In *Proceedings of IEEE International Symposium on Asynchronous Circuits and Systems*, Mar 2006.

[38] Catherin G. Wong, Alain J. Martin, and Peter Thomas. An architecture for asynchronous FPGAs. In *Proceedings of IEEE International Conference on Field-Programmable Technology*, Dec 2003.

[39] J. Teifel and R. Manohar. Programmable asynchronous pipeline arrays. In *Proceedings of 13th International Conference on Field-Programmable Logic and Application*, Dec 2003.

[40] J. Rose, R.J. Francis, D. Lewis, and P. Chow. Architecture of field-programmable gate arrays: the effect of logic block functionality on area efficiency. *IEEE Journal of Solid-State Circuits*, 25(5):1217–1225, Oct 1990.

[41] A.M. Saleh, J.J. Serrano, and J.H. Patel. Reliability of scrubbing recovery–techniques for memory systems. *IEEE Transactions on Reliability*, 39:114–122, 1990.

[42] Tom Verhoeff. Delay-insensitive codes — an overview. *Distributed Computing*, 3(1):1–8, 1988.

[43] D. Lubell. A short proof of Sperner's theorem. *Journal of Combinatorial Theory*, 1, 1966.

[44] B. Bose and D. K. Pradhan. Optimal unidirectional error detecting/correcting codes. *IEEE Transactions on Computers*, C-31:564–568, Jun 1982.

[45] Sandip Kundu and Sudhakar M. Reddy. On symmetric error correcting and all unidirectional error detecting codes. *IEEE Transactions on Computers*, 39(6):752–761, Jun 1990.

[46] Mario Blaum and Jehoshua Bruck. Unordered error-correcting code and their applications. In *Proceedings of 22nd International Symposium on Fault-Tolerant Computing*, Jul 1992.

[47] Robert J. McEliece. *The Theory of Information and Coding.* Cambridge, 1985.

[48] Christian D. Nielsen. Evaluation of function blocks for asynchronous design. In *EURO-DAC '94: Proceedings of the conference on European design automation*, pages 454–459, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[49] Xiaolei Li. Efficient function-block implementation of self-timed circuits. In *Proceedings of the 47th IEEE Midwest Symposium on Circuits and Systems*, 2004.

[50] Jens Sparsø and Jørgen Staunstrup. Delay-insensitive multi-ring structures. *Integration, the VLSI Journal*, 15(3):313–340, 1993.

[51] Ted E. Williams. *Self-Timed Rings and their Application to Division.* PhD thesis, Department of EECS, Stanford, 1990.

[52] M. A. Kishinevskiμi and Alexandre V. Yakovlev. *Self-Timed Control of Concurrent Processes: The Design of Aperiodic Logical Circuits in Computers and Discrete Systems.* Kluwer Academic Publishers, Norwell, MA, USA, 1990.

[53] V. Akella, N.H. Vaidya, and G.R. Redinbo. Limitations of VLSI implementation of delay-insensitive codes. In *Proceedings of Annual Symposium on Fault Tolerant Computing*, Jun 1996.

[54] Stanislaw J. Piestrak. Membership test logic for delay-insensitive codes. In *Proceedings of 4th International Symposium on Advanced Research in Asynchronous Circuits and Systems*, Mar 1998.

[55] Edith Hemaspaandra and Gerd Wechsung. The minimization problem for boolean formulas. In *Proceedings of the 38th IEEE Symposium on Foundations of Computer Science*, 1997.

[56] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35:671–691, Aug 1986.

[57] R. Rudell and A. Sangiovanni-Vincentelli. Espresso-mv: Algorithms for multiple-valued logic minimization. In *Proceedings of Custom Integrated Circuit Conference*, May 1985.

[58] Alain J. Martin, Mika Nyström, and Paul Penzes. Et^2: A metric for time and energy efficiency of computation. In *Power-Aware Computing.* Kluwer Academic Publishers, 2001.

[59] Paul Penzes. *Energy-Delay Complexity of Asynchronous Circuits.* PhD thesis, Dept of Computer Science, Caltech, 2002.

[60] Mika Nyström and Alain J. Martin. *Asynchronous Pulse Logic.* Kluwer Academic Publishers, 2002.

[61] Alain J. Martin, Andrew Lines, Rajit Manohar, Mika Nyström, Paul Penzes, Robert Southworth, Uri Cummings, and Tak Kwan Lee. The design of an asynchronous MIPS R3000 microprocessor. In *Proceedings of 17th Conference on Advanced Research in VLSI*, 1997.

[62] Rajit Manohar, Mika Nyström, and Alain J. Martin. Precise exceptions in asynchronous processors. In *Proceedings of 21th Conference on Advanced Research in VLSI.* IEEE Computer Society Press, Mar 2001.

[63] J.V. Woods, P. Day, S.B. Furber, J.D. Garside, N.C. Paver, and S. Temple. Amulet1: An asynchronous arm microprocessor. *IEEE Transactions on Computers*, 46(4):385–398, 1997.

[64] Rajit Manohar. *Cast: Language description and libraries*, 1997.

[65] Robert N Mayo, Michael H. Arnold, Walter S. Sctoo, Don Start, and Gordon T. Hamachi. 1990 DECWRL/Livermore magic release. Technical Report WRL-90-7, HP Labs, 1990.

[66] Serge Mister and Stafford E. Tavares. Cryptanalysis of RC4-like ciphers. In *SAC '98: Proceedings of the Selected Areas in Cryptography*, pages 131–143, London, UK, 1999. Springer-Verlag.

[67] Alain J. Martin, Mika Nyström, Paul Penzes, and Catherine Wong. Speed and energy performance of an asynchronous MIPS R3000 microprocessor. Technical report, Dept of Computer Science, Caltech, Jun 2001.

[68] *Rad-Hard 32 bit SPARC V8 Processor: AT697E*, Sep 2006.

[69] John I. Gaona Jr. A radiation-hardened computer for satellite applications. In *Proceedings of the 10th Annual Conference on Small Satellites*, Sep 1996.

[70] J.R. Marshall and R.W. Berger. A processor solution for the second century of power space flight. In *Proceedings of the 19th Digital Avionics System Conferences*, Dec 2000.

[71] Uri Cumming, Andrew Lines, and Alain J. Martin. An asynchronous pipeline lattice-structure filter. In *Proceedings of International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 1994.