Mississippi State University
Scholars Junction

Theses and Dissertations Theses and Dissertations

5-5-2007

Automated Mapping of Clocked Logic to Quasi-Delay Insensitive Circuits

Lokesh Shivakumaraiah

Follow this and additional works at: https://scholarsjunction.msstate.edu/td

Recommended Citation

Shivakumaraiah, Lokesh, "Automated Mapping of Clocked Logic to Quasi-Delay Insensitive Circuits" (2007). *Theses and Dissertations*. 824.
https://scholarsjunction.msstate.edu/td/824

This Dissertation - Open Access is brought to you for free and open access by the Theses and Dissertations at Scholars Junction. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Scholars Junction. For more information, please contact scholcomm@msstate.libanswers.com.

AUTOMATED MAPPING OF CLOCKED LOGIC TO QUASI-DELAY INSENSITIVE CIRCUITS

By

Lokesh Shivakumaraiah

A Dissertation
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy
in Computer Engineering
in the Department of Electrical and Computer Engineering

Mississippi State, Mississippi

May 2007

Copyright by

Lokesh Shivakumaraiah

2007

AUTOMATED MAPPING OF CLOCKED LOGIC TO

QUASI-DELAY INSENSITIVE CIRCUITS

By

Lokesh Shivakumaraiah

Approved:

____
Robert B. Reese
Associate Professor of Electrical and
Computer Engineering
(Director of Dissertation)

____
Jerry W. Bruce
Associate Professor of Electrical and
Computer Engineering
(Committee Member)

____
James C. Harden
Retired Head of the Department and,
Professor of Electrical and Computer
Engineering
(Committee Member)

____
Thomas Philip
Professor of Computer Science and
Engineering (Committee Member)

____
Nicholas H. Younan
Graduate Program Director and Professor
of Electrical and Computer
Engineering

____
Kirk Schulz
Dean and Professor
Dean of Engineering

Name: Lokesh Shivakumaraiah

Date of Degree: May 4, 2007

Institution: Mississippi State University

Major Field: Computer Engineering

Major Professor: Robert (Bob) Reese

Title of Study: AUTOMATED MAPPING OF CLOCKED LOGIC TO QUASI-DELAY INSENSITIVE CIRCUITS

Pages in Study: 121

Candidate for Degree of Doctor of Philosophy

The use of computer aided design (CAD) tools has catalyzed the growth of IC design techniques. The rapid growth in transistor count for synchronous digital circuits has increased circuit complexity. This growing complexity of synchronous circuits has exposed design issues such as clock skew, increased power consumption, increased electromagnetic interference and worst case performance.

The increasing number of challenges posed by synchronous designs has encouraged researchers to explore asynchronous design techniques as an alternative methodology. Asynchronous circuits do not use a global clock signal that is the primary cause of many design challenges faced by synchronous designers. It has also been shown in some designs that asynchronous circuits consumes less power, and exhibits better average case performance than synchronous circuits.

Asynchronous design techniques, even with their various advantages over synchronous systems, are not widely accepted by logic designers. This is due to the

shortcomings of asynchronous design methodologies, primarily, the limited availability of CAD tool support and the use of proprietary specification languages.

To overcome the shortcomings of current asynchronous design techniques, this research uses a methodology for designing asynchronous circuits starting from clocked RTL design. This research extends the concepts of Phased Logic (PL) and marked graphs to quasi-delay insensitive gates (QDI) gates to create an asynchronous PL-QDI methodology. The PL methodology is easy to use as it maps conventional RTL designs into delay insensitive PL circuits using commercial CAD tools. Caltech's QDI gates exhibit fast forward latency, but the use of Caltech's methodology requires a user skilled in the pecurialities of the Caltech design methodology. This research uses best of Caltech's QDI circuit methodology and the PL methodology to come up with a new asynchronous PL-QDI methodology. It also presents a synthesis algorithm that uses commercially available synchronous CAD tools to map clocked designs to PL-QDI systems.

Results of this research show that third-party clocked RTL codes including intellectual property (IP) cores can be converted to asynchronous PL-QDI systems using the PL-QDI CAD tools presented in this research. This work shows how mature synchronous CAD tools can be used to design clockless circuits.

DEDICATION

I would like to dedicate this research to my parents (D. Shivakumaraiah and Sakamma) and my brother (Vasanthkumar Shivakumaraiah) who have greatly supported me throughout my life. They have always encouraged me to pursue my PhD. Without their encouragement it would not have been possible to reach this milestone in my career.

ACKNOWLEDGMENTS

I would like to extend my special thank to Dr. Robert (Bob) Reese for taking his precious time for guiding me through out the research. His words "*hard work never go unnoticed*" always keeps me motivated while working through the tough problem.

I also want to express gratitude to Dr. Jim C. Harden, Dr. Thomas Philip and Dr. Jerry W. Bruce who had their doors open for me to discuss either my academic/research topics or student life.

TABLE OF CONTENTS

Page

DEDICATION ... ii

ACKNOWLEDGMENTS ... iii

LIST OF TABLES ... vii

LIST OF FIGURES ... viii

CHAPTER

I. INTRODUCTION ... 1

1.1 Disadvantages of Synchronous Circuits ... 2
1.1.1 Clock Skew ... 2
1.1.2 Worst-case Performance ... 2
1.1.3 Increased Power Dissipation ... 3
1.1.4 High Electromagnetic Interference ... 3
1.1.5 Metastability ... 4
1.1.6 No Change in Performance with External Environment ... 4
1.2 Advantages of Asynchronous Circuits ... 4
1.2.1 Absence of Clock Skew ... 4
1.2.2 Average-case Performance ... 5
1.2.3 Low Power Consumption ... 5
1.2.4 Less Electromagnetic Interference ... 6
1.2.5 Tolerence to External Environment ... 6
1.2.6 Challenges Faced by Asynchronous Design ... 6
1.2.7 Research Overview ... 9
1.3 Thesis Organization ... 9

II. ASYNCHRONOUS DESIGN CONCEPTS ... 12

2.1 Data Communication in Synchronous and Asynchronous Circuits ... 12
2.2 Asynchronous Circuit Classification Based on Delay Models ... 15
2.2.1 Bounded Delay Model ... 16
2.2.2 Unbounded Delay Models ... 16
2.2.3 Delay insensitive Circuits ... 16
2.2.4 Quasi-delay Insensitive Circuits ... 17
2.2.5 Speed Independent Circuits ... 18
2.3 Data-encoding for Asynchronous Data ... 18
2.3.1 Single-rail Encoding ... 18

2.3.2 1-of-N Encoding 20
2.4 Handshaking Protocols 23
2.4.1 Two-phase Handshaking 25
2.4.2 Four-phase Handshaking 25

III. PHASED LOGIC SYSTEMS & QUASI-DELAY INSENSITIVE SYSTEMS 28

3.1 Phased Logic Systems 28
3.1.1 Level-encoded Dual-rail Encoding 29
3.1.2 Phased Logic Gates 30
3.1.3 Petri-nets 32
3.1.4 Marked Graph 33
3.1.5 Mapping Clocked Netlists to PL Netlists 34
3.2 Quasi-Delay Insensitive Systems 39
3.2.1 Weak-conditioned Half Buffer 39
3.2.2 Precharged Half Buffer and Precharge Full Buffer 40
3.2.3 Four-phase Handshaking in PCHB Gates 43
3.2.4 PCHB Gates Internal Operation 44
3.2.5 Caltech's Asynchronous Design Methodology 47
3.3 Summary 51

IV. PHASED LOGIC FOR QUASI-DELAY INSENSITIVE CIRCUITS 52

4.1 A Cell Design for PL-QDI systems 52
4.1.1 Performance of Caltech's QDI Design Cells 52
4.1.2 PL-QDI Template Gate 54
4.2 Token Abstraction for Quasi-Delay Insensitive Gates 54
4.3 Mapping of a Clocked Netlist to a PL-QDI Netlist 58
4.4 PL-QDI Gate Interaction 59
4.5 Modifications to the PL-QDI Gate Template for Barrier Gates 65
4.5.1 Forced Token at the Barrier Gate Output 65
4.5.2 Initial Token Removal on the Barrier Gate Output Feedback 67

V. CAD SUPPORT FOR PL-QDI CIRCUIT DESIGN 68

5.1 Introduction to Synchronous CAD Tools 68
5.2 An RTL Flow for Synchronous Circuits 70
5.3 CAD Tools and Asynchronous Integrated Circuit Design 73
5.4 PL-QDI Synthesis 74
5.4.1 PL-QDI Synthesis Algorithm 75
5.5 CAD Tool Flow of PL-QDI Methodology 84
5.6 PL-QDI Gate Library 90
5.7 Summary 93

VI. PL-QDI DESIGN EXAMPLES 95

6.1 PL-QDI System Features 95
6.1.1 PL-QDI Systems Maintain The Synchronous Property 95

6.1.2 PL-QDI Gates Exhibit Fast Forward Latency ..96
6.2 PL-QDI Testbench ..98
6.3 Design Examples 101
6.3.1 Counter Designs 101
6.3.2 64-bit Floating Point Clipper Circuit 104
6.3.3 picoJava-II Floating Point Unit 108

VII. CONCLUSION AND FUTURE WORK 114

7.1 Summary of Results 114
7.2 Future Work 115
7.2.1 Modified PL-QDI Gate Library 115
7.2.2 Physical Design 115

REFERENCES 116

LIST OF TABLES

TABLE Page

3.1 CHP notation 47

3.2 Operators and the production rules.. 50

6.1 Comparison of number of gates in clocked and PL-QDI designs……........ 102

6.2 Gate count for 64-bit clipper examples 108

6.3 Floating point unit arithmetic operations 111

6.4 Floating point unit data type conversion operations.................................... 111

6.5 Gate count of picoJava-II FPU designs 111

LIST OF FIGURES

FIGURE Page

2.1 Data communication in synchronous and asynchronous circuits. 13

2.2 Classifications of delay models 15

2.3 Wire delay and gate delay model 17

2.4 Asynchronous circuit using single-rail encoding. 20

2.5 Dual-rail four-phase and two-phase encoding. 23

2.6 Synchronous circuit with sequential and combinational gates 24

2.7 Handshaking protocols 27

3.1 LEDR encoding 29

3.2 PL Gate phase and corresponding token representation...............................31

3.3 PL translation steps...36

3.4 PL feedback insertion rules and corresponding token markings. 38

3.5 Weak conditioned half buffer gate 40

3.6 PCHB and PCFB gates 42

3.7 2-input PCHB AND gate operation...43

3.8 Data communication in PCHB gates 46

4.1 Pl-QDI gate template 53

4.2 Four-phase handshaking in PL-QDI gate 54

4.3 Token flow in PL-QDI gate, steps 1-3 56

4.4 Token flow in PL-QDI gate, steps 4-5 57

4.5 An example PL-QDI circuit.. 60

4.6 Token marking in PL-QDI system during reset 62

4.7 Initial token marking after the release of reset showing safety violation. 63

4.8 Live and safe initial token marking of PL-QDI circuit 64

4.9 PL-QDI barrier gate and waveforms explaining its working 66

5.1 RTL flow 72

5.2 PL-QDI synthesis pseudo code 76

5.3 One-to-one mapping of AND2 gate to PL-QDI AND2 gate 77

5.4 Clocked circuit and equivalent PL-QDI network 78

5.5 Splitter gate insertion to break direct barrier gate to barrier gate path 80

5.6 Buffer function insertion 81

5.7 Feedback in a PL-QDI circuit 82

5.8 PL-QDI 2-bit counter with count enable 83

5.9 PL-QDI CAD flow 85

5.10 RTL code of 2-bit counter with count enable 86

5.11 Synthesized gate netlist of 2-bit counter with count enable 87

5.12 PL-QDI system produced from the PL-QDI CAD flow 89

5.13 PL-QDI through gate 90

5.14 PL-QDI barrier gate 92

5.15 PL-QDI logic high constant generator ..93

5.16 PL-QDI logic low constant generator ..93

6.1 PL-QDI gate compute block ..97

6.2 Block diagram of PL-QDI testbench 100

6.3 Marked graph representation of PL-QDI testbench 101

6.4 PL-QDI counter testbench 103

6.5 Clipper circuit high level abstraction 104

6.6 Datapath and control of clipper circuit 106

6.7 PL-QDI 64-bit clipper testbench 107

6.8 PL-QDI wrapper used around microcode ROM 109

6.9 PL-QDI 64-bit FPU block diagram 112

6.10 PL-QDI 64-bit FPU marked graph representation 113

CHAPTER I

INTRODUCTION

Integrated chip design techniques have witnessed rapid growth since the invention of the first commercial integrated circuits in the 1960s [29]. Transistor sizes have shrunk to nanometer levels [43] and the number of transistors on a single chip has risen to millions [30]. The use of computer aided design (CAD) tools [35] has catalyzed the growth of IC design techniques. The rapid growth in transistor count for synchronous digital circuits has increased circuit complexity. This growing complexity of synchronous circuits has exposed design issues such as clock skew, increased power consumption, increased electromagnetic interference (EMI), metastability and worst case performance [28, 31, 32, 34, 36].

The increasing number of challenges posed by synchronous designs has encouraged researchers to explore asynchronous design techniques [2, 4, 7, 9, 11, 26, 32, 37] as an alternative methodology. Asynchronous circuits have advantages such as no clock skew, lower power consumption for average case performance, decreased electromagnetic interference and average case performance [31, 32, 24].

1.1 Disadvantages of Synchronous Circuits

1.1.1 Clock Skew

Global clocks are used to synchronize output computations in synchronous circuits. Shrinking transistor sizes has increased the number of gates on a chip, which has increased the capacitive load on the global clock signal. Furthermore, the increasing differences between gate delays (have decreased) and global wire delay (has remained relatively constant) have exacerbated clock skew problems. Clock edges arrive earlier at gates that are near, and late at gates that are far from the clock origin. Accounting for clock skew means an increase in clock period and a decrease in the circuit speed [31]. To tackle the problem of clock skew, designers must resort to using techniques such as hierarchical clocks [53], clock distribution [52] and clock deskewing [50]. Thus, clock skew is becoming increasingly challenging in synchronous IC design.

1.1.2 Worst-case Performance

Synchronous designers use clock periods that are greater than the longest path in the design. This allows enough time for the completion of the output computation in the longest path. At the same time, circuit paths whose delays are shorter than that of the longest path must wait until the end of the clock period for the start of the next computation. This results in idle time within the clock period and a decrease in performance [32].

1.1.3 Increased Power Dissipation

Power is dissipated in synchronous designs due to dynamic power dissipation and static power dissipation [47]. Static power dissipation is due to leakage current and sub-threshold currents. Dynamic power is dissipated during the charging and discharging of the gate's load capacitance, and accounts for most of the power dissipation in synchronous circuits [38]. The dynamic power consumption of synchronous circuits is directly proportional to clock frequency, gate load capacitance, and the square of the power supply. Advances in synchronous design methodologies have increased clock frequency and the number of gates per chip resulting in increased power dissipation. Sections of the synchronous circuit not involved in the current computation also undergo switching due to the availability of the clock signal. This unnecessary switching adds to the power dissipation in synchronous circuits. Furthermore, combinational gates undergo temporary transitions before settling down to a stable output value. This unwanted temporary switching also increases power dissipation in synchronous circuits. Increased power dissipation has become a cause of concern for synchronous designers.

1.1.4 High Electromagnetic Interference

Rapidly changing current in synchronous circuits causes EMI [39]. Increased EMI in synchronous circuits make it vulnerable to security attacks [28]. Thus, applications requiring high security that use synchronous circuits require EMI shielding.

1.1.5 Metastability

Metastability is a condition where the circuits that stores states (ex: cross coupled inverters) become biased at the midpoint of the two stable states representing logic high and low [38, 24]. A circuit can remain in the metastable state for an unknown amount of time before returning to a stable state. This can be harmful to synchronous designs as clocked circuits cannot wait for the circuit in a metastable state to return to a stable state. Synchronous systems should evaluate within a known period of time, so a metastable output value may either be interpreted as a logic low or high resulting in unknown circuit operation. After this kind of erroneous operation, the proper functionality of the synchronous system cannot be restored.

1.1.6 No Change in Performance with External Environment

Designers of synchronous systems must account for variations of gate delays with respect to temperature, power-supply voltage and fabrication parameters to determine the clock period of the design under construction. This clock period must account for worst case operating conditions and the longest delay path resulting in a longer clock period, reducing the performance of synchronous circuits.

1.2 Advantages of Asynchronous Circuits

1.2.1 Absence of Clock Skew

Asynchronous gates communicate with each other by means of a handshaking protocol, thus eliminating the need for a global clock. Absence of a clock means there is no clock skew in asynchronous circuits [28, 32, 31].

1.2.2 Average-case Performance

Asynchronous gates use handshaking signals such as input request and output acknowledgement to establish data communication between gates. These handshaking signals ensure that asynchronous gates begin a new computation at the arrival of new inputs and do not have to wait for other gates to complete their computation. The variable computation time of asynchronous gates results in an average case performance [24, 32, 31] of asynchronous systems that can be better than the worst-case performance of synchronous systems.

1.2.3 Low Power Consumption

The global clock network that is a major source of power dissipation in synchronous circuits is absent in asynchronous circuits. Asynchronous gates compute outputs only after the arrival of all of the input signals, thus eliminating temporary transitions seen in combinational gates. There are no transitions in unused parts of asynchronous systems as they are waiting for input arrivals. Thus, many asynchronous circuits show decreased power consumption over synchronous circuits that implement similar functionality [28, 32, 31]. For example, Theseus Logic [46] implemented Motorola's STAR08 8-bit microcontroller using asynchronous Null Conventional Logic (NCL). This asynchronous microcontroller (NCL08) exhibited approximately 38% less power consumption than the synchronous STAR08 [45].

1.2.4 *Less Electromagnetic Interference*

Asynchronous circuits exhibit less EMI due to reduced switching activity [28]. The asynchronous NCL08 had 11db lower EMI than the synchronous STAR08 [45].

1.2.5 *Tolerence to External Environment*

Asynchronous circuits dynamically adjust their performance to best/typical/worst case operating temperatures, power supply voltage, and fabrication variation [32]. This means that the performance of asynchronous system when subjected to best case external environment can be relatively faster than in the worst case. This is not the case with synchronous circuits as their fastest performance depends on their worst case clock period which is based upon the worst case environmental conditions irrespective of the actual environmental conditions.

1.2.6 *Challenges Faced by Asynchronous Design*

Asynchronous design techniques even with their various advantages over synchronous systems, are not widely accepted by logic designers. This is due to the shortcomings of asynchronous design methodologies, primarily the limited availability of CAD tool support [11] and the use of proprietory specification languages [36, 42]. This section briefly discusses some common asynchronous methodologies and highlights their drawbacks. Later, it explains how each of these disadvantages acts as barriers to adoption of these methodologies by the IC design industry.

Martin's asynchronous design methodology [7, 27] translates communicating sequential processes (CSP) into quasi-delay insensitive (QDI) asynchronous systems. In

this methodology, asynchronous circuit behavior is defined using sequential communication hardware processes (CHP). Process decomposition is used to convert asynchronous circuits described using CHP processes into a set of interactive concurrent CHP processes. Hand shaking expansions (HSE) are used to implement communication channels between the CHP processes using signal wires. HSEs obtained from the previous transformation process are converted into a set of production rules (PRs) that eliminates explicit sequencing. The operator reduction stage is then used to map PRs into standard hardware components and state variables. Additional information about this methodology is given in Chapter 3. The disadvantages of Caltech's asynchronous QDI methodology are the use of the CHP language and the systematic semantics-preserving transformations that requires a skilled designer in order to produce optimum results.

The asynchronous null convention logic (NCL) [41] methodology uses synchronous CAD tools to generate asynchronous circuits. This design technique requires the RTL to be coded using a specific coding style that separates registers and combinational logic. Designers must also explicitly specify each register's request and acknowledgement signals in the RTL code. The RTL is synthesized using an NCL-specific target library. After synthesis, the gate netlist is mapped to delay-insensitive minterm synthesis (DIMS) [48] type dual-rail assignments. DIMS is a method of boolean algebra simplification similar to sum of products simplification, but in DIMS, minterms are formed by using C-gates [20]. This methodology has some major drawbacks that make it user unfriendly. Restrictions on RTL coding style make this methodology unsuitable for using third party RTL or intellectual property (IP) RTL cores. Re-writing

RTL code in a specific coding style demands extensive work by the designer, and the resulting code must be reverified as having the same functionality as the original.

Beerel's asynchronous methodology [24] requires circuit specifications using state graphs with special properties. It uses cube lists to represent circuits. The synthesis algorithm performs a series of transformations on the cube list to create asynchronous designs. The disadvantage of this methodology is the use of a custom specification language.

The Phased Logic (PL) methodology maps clocked RTL to asynchronous circuits. Linder and Harden in [9] describes PL as a delay insensitive methodology used to describe asynchronous circuit operations in terms of token flow within marked graphs while maintaining the synchronous paradigm. It uses level encoded dual-rail (LEDR) signaling for data encoding to reduce power consumption due to signal transitions. This methodology is easy to use in that it starts from a clocked netlist. However static CMOS PL gates have an output latch that increases the critical path of PL systems. To help overcome this performance penalty, PL systems can sometimes use early evaluation (EE) gates in the critical paths. EE gates fire upon the arrival of an input subset (trigger function) that can guarantee the correct output value. EE gates contain extra logic that is used to detect the early trigger. The use of EE gates increases the transistor count of the system.

This section has highlighted some of the drawbacks in present asynchronous design techniques. The communicating processes compilation technique uses a full-custom methodology and is not supported by commercial CAD tools. NCL logic has

restrictions on the RTL coding style. Beerel's asynchronous design technique uses a custom tool for synthesis. The PL methodology use gates with output latches that add delay in the critical path.

The limited availability of commercially available mature CAD tools and inadequate skilled manpower are huge barriers to the IC industry for accepting an asynchronous methodology for logic design. To overcome these shortcomings, designers are calling for the development of new CAD tools [31] or the use of commercial CAD tools in asynchronous circuit design.

1.2.7 Research Overview

This research extends the concept of Phased Logic (PL) [9] and marked graphs to quasi-delay insensitive gates (QDI) [15] gates to create an asynchronous PL-QDI methodology. It also presents a synthesis algorithm that can make use of commercial CAD tools to map clocked designs to PL-QDI systems.

The marked graph token abstraction first introduced in the PL methodology is extended to QDI gates. A straight forward extension of PL concepts to QDI gates violates PL initial token marking rules as well as QDI handshaking protocols resulting in a dead system. This work overcomes these problems to construct a live PL-QDI system.

1.3 Thesis Organization

This section explains the organization of the remaining chapters in this thesis. It gives a brief review of the concepts explained in each of the chapters.

Chapter II: Asynchronous Design Concepts

Chapter II describes different types of data encoding schemes and handshaking protocols that are used in asynchronous design. Classification of asynchronous circuits based on delay models is also briefly discussed.

Chapter III: Phased Logic Systems & Quasi-Delay Insensitive Systems

This chapter describes phased logic (PL) and quasi-delay insensitive (QDI) circuits. The first part of this chapter gives an in-depth explanation of Petri-nets [10], marked graphs [10], token abstraction, PL gate operation, PL gate firing rules, and the PL-synthesis algorithm. The second half of the chapter gives an overview of QDI gates and Caltech asynchronous design methodologies.

Chapter IV: Phased Logic for Quasi-Delay Insensitive circuits

Chapter IV covers the asynchronous design methodology developed in this research. This chapter gives a detailed explanation of how token abstraction and marked graph concepts are extended to QDI systems and explains how Caltech's PCHB gate design are modified to suit the PL-QDI methodology. A simple example of a PL-QDI gate and its implementation is used to explain the operation of PL-QDI systems.

Chapter V: CAD Support for the PL-QDI methodology

This chapter explains the use of CAD tools in IC design and shows an example RTL CAD flow used in RTL synthesis. The current state of CAD tools in the asynchronous community is also discussed. A detailed explanation of the synthesis algorithm used to map clocked RTL designs to asynchronous PL-QDI designs is covered

in this chapter. Finally, it describes the PL-QDI CAD flow developed by using commercially-available mature CAD tools.

Chapter VI: Design examples

This chapter describes PL-QDI circuit features and explains how a PL-QDI system interacts with its external environment. Results from clocked systems mapped to PL-QDI systems are presented along with a discussion of test bench construction for PL-QDI systems.

Chapter VII: Conclusion and Future Work

This chapter summarizes the results and explores areas of future work.

CHAPTER II

ASYNCHRONOUS DESIGN CONCEPTS

Unlike synchronous designs that use a clock signal to control data movement, asynchronous designs use handshaking signals to exchange data between gates. Handshaking signals are sequenced in a particular order defined by a handshaking protocol to accomplish data communication. This chapter introduces the concepts of asynchronous design. Section 1 compares data communication in synchronous systems and asynchronous systems. Section 2 provides different classifications for asynchronous methodologies, while section 3 describes data encoding schemes used within asynchronous circuits. Finally, section 4 discusses common handshaking protocols for data transmission within asynchronous circuits.

2.1 Data Communication in Synchronous and Asynchronous Circuits

In synchronous circuits, a global clock signal is used to synchronize data flow within the circuit. Combinational gate outputs are assumed valid and latched into sequential gates on either a rising/falling edge signal (edge-triggered) or by a high/low level signal (level-sensitive). Asynchronous circuits do not have a global clock. Data communication between asynchronous gates is performed by using *handshaking signals* [22]. Input data arrival is detected by an input validity circuit, while output completion is detected by an output completion circuit. Handshaking signals used for data transfer are

generated by using additional circuitry and data encoding styles. Data encoding styles used in asynchronous gates are discussed later in this section.

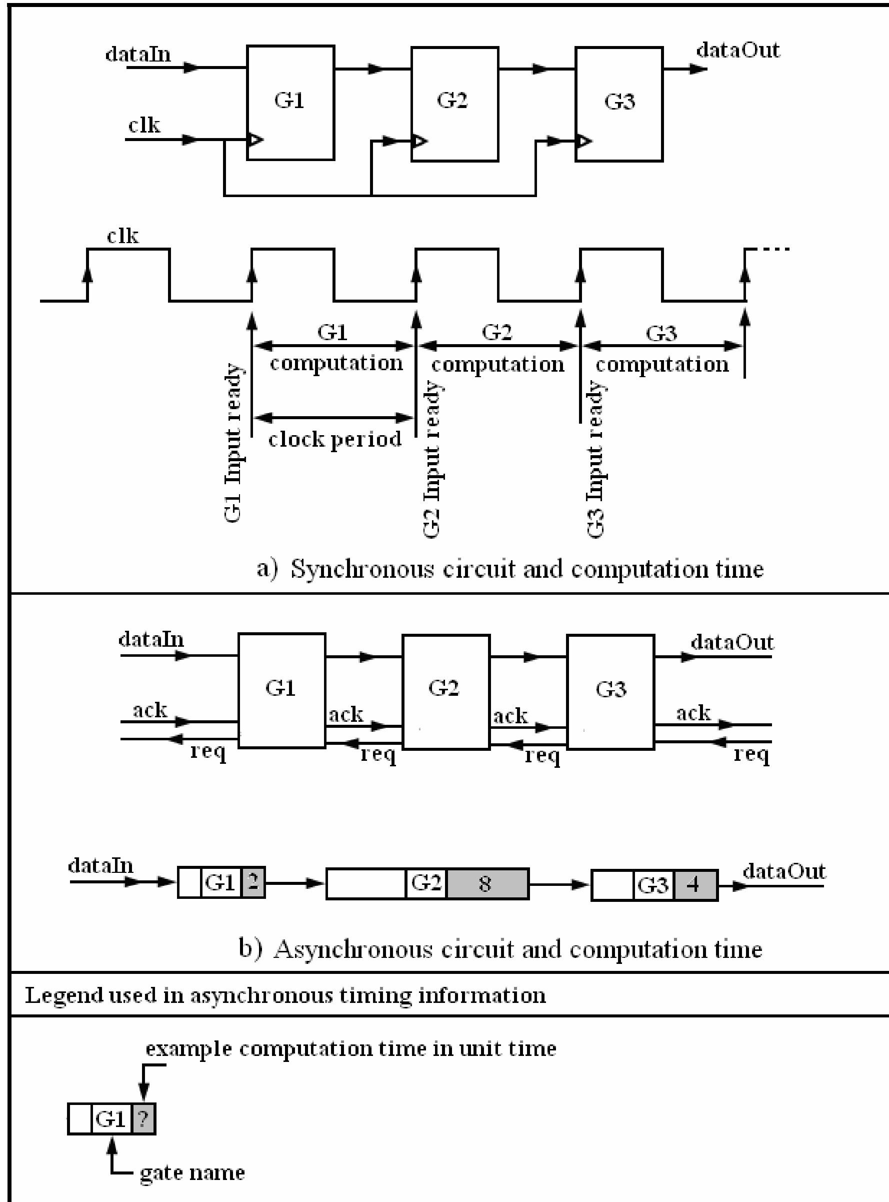


Figure 2.1 Data communication in synchronous and asynchronous circuits

Figures 2.1a and 2.1b compare data communication within synchronous and asynchronous circuits. Assume that the synchronous circuit shown in Figure 2.1a processes its inputs on the rising clock edge. In Figure 2.1a, gate G1 produces its output before the second rising clock edge for use by gate G2. Similarly, gates G2 and G3 must produce their outputs before the next corresponding rising clock edge. This means that the longest register-to-register path delay between synchronous gates G1, G2, G2 must all be less than or equal to the clock period. The computation time in synchronous circuits are fixed and is equal to the clock period. Figure 2.1b shows one method of asynchronous communication using separate channels for data and handshaking signals. Asynchronous gates G1 and G2 exchange data using acknowledgement *ack* and request *req* signals. Gate G2 sends a request signal to Gate G1 indicates that it has finished computation and is ready for new input data. Gate G2 responds by asserting the *ack* signal when the G2 output is ready. A similar exchange of handshaking signals is used for communication between gates G2 and G3. The order in which handshaking signals are exchanged to establish successful communication between asynchronous gates is defined as a handshaking protocol. The handshaking protocol used for data communication in asynchronous circuits allows the computation time of the asynchronous gates to be variable. The variable computation time of asynchronous gates can be explained by using the asynchronous circuit shown in Figure 2.1b. Gates G1, G2, G3 have computation times of 2 ns, 8 ns, and 4 ns respectively. In a straight pipelined system this is not an advantage, but in a system with feedback in the pipeline this can be used to produce a

lower average-case execution time during some interactive calculations in which the delay of each loop iteration are variable.

2.2 Asynchronous Circuit Classification Based on Delay Models

Asynchronous circuits can be classified on the basis of the delay models used in their designs. Delay models [11, 16] give information regarding the timing constraints used in the circuit design stage. Delay models used in the design of asynchronous circuits are classified as *bounded* and *unbounded*, depending on the timing assumptions of the gate and wire delays in the system. Figure 2.2 shows the delay model classifications.

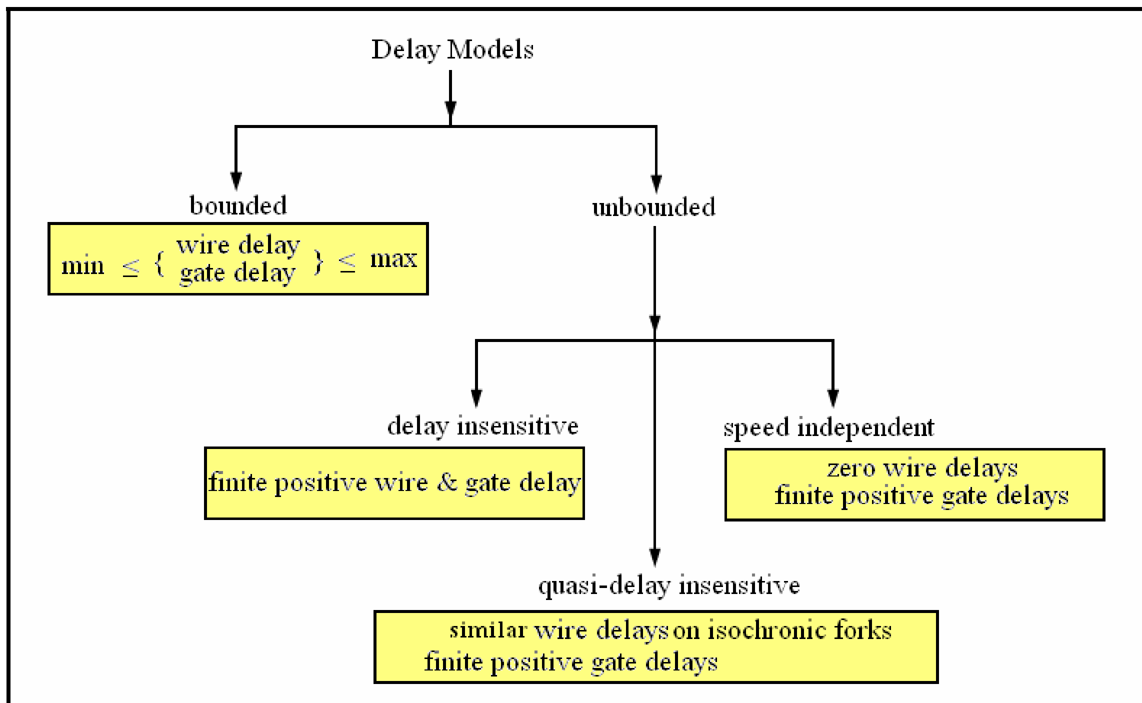


Figure 2.2 Classifications of delay models

2.2.1 *Bounded Delay Model*

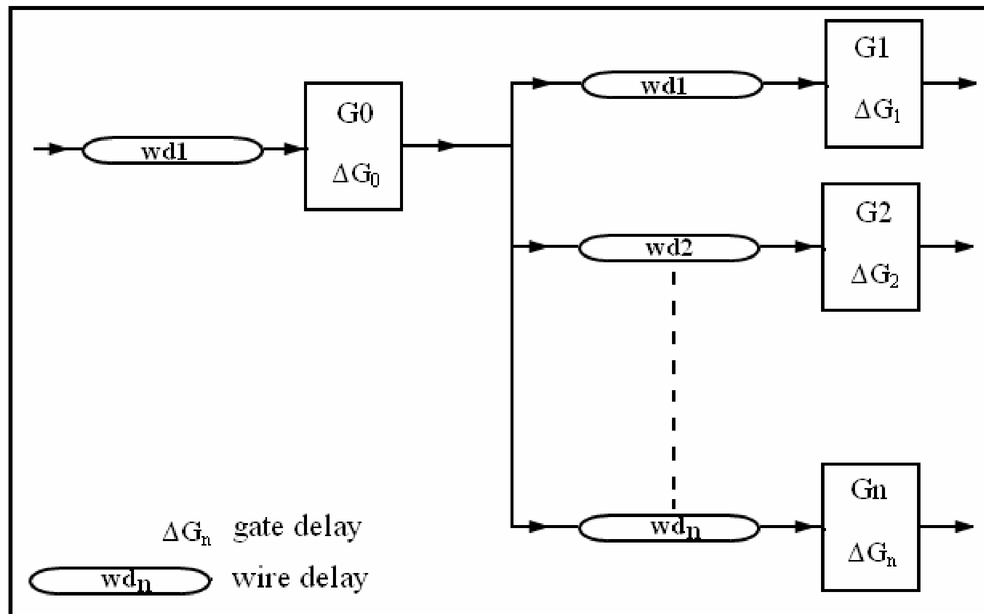
A bounded delay model assumes known or bounded gate and wire delays that fall within predefined minimum and maximum values. The bounded delay model concept is related to the synchronous design's delay model where gate and wire delays cannot exceed the clock period of the circuit. This similarity made the bounded delay models popular during the early stages of asynchronous design research [2, 16]. Circuits based on bounded delay models involve stringent timing constraints during physical implementation.

2.2.2 *Unbounded Delay Models*

An unbounded delay model assumes unbounded gate and wire delays, and is further divided into delay insensitive, quasi-delay insensitive, and speed-independent delay models based on the timing assumptions on the gate and wire delays.

2.2.3 *Delay insensitive Circuits*

The delay insensitive (DI) model is independent of gate and wire delays. Asynchronous circuits implemented using the DI model should exhibit proper operation for any arbitrary finite positive values for the gate and wire delays [11]. In Figure 2.3, this is represented by finite random positive delay for all the gate delays (ΔG_n) and wire delays (wd_n). In [14], Martin explains the limitations of DI circuits and shows that only circuits composed of C-elements and buffers can be classified as delay insensitive. Martin developed a new genre of delay insensitive circuits called *quasi-delay insensitive* (QDI) circuits, which is a practical approximation of delay insensitive circuits.



Note Adopted from [16]

Figure 2.3 Wire delay and gate delay model

2.2.4 *Quasi-delay Insensitive Circuits*

The delay model used in quasi-delay insensitive (QDI) circuits assumes equal delays on the branches of isochronic forks [13, 14, 7] and is independent of the other gate and wire delays. Isochronic forks are splits in the circuit interconnect where the wire delays of the splits are similar and small when compared to gate delays, such as to not affect the correct operation of the circuit. In Figure 2.3, this is represented by nearly equal wire delays ($wd_1 \approx wd_2 \approx wd_i$ where i = 3 to n) and random positive value for gate delays (ΔG_n).

2.2.5 *Speed Independent Circuits*

This delay model assumes random finite positive values for gate delays and negligible wire delays [24]. In Figure 2.3, this is represented by all wire delays set to zero ($wd_1= wd_2= wd_i = 0$) where i = 3 to n and a random positive value for the gate delays (ΔG_n). In modern CMOS circuits, this is an impractical delay model since wire delays are not negligible when compared to gate delays in submicron and nanometer designs.

2.3 Data-encoding for Asynchronous Data

Data is encoded in asynchronous designs to facilitate the generation of handshaking signals by the arrival of input data. Asynchronous gates have extra logic to compute handshaking signals such as input completion and output completion signals. The logic used for output completion detection and input data validity depends upon the data encoding [1, 2, 4] scheme that is used in asynchronous circuits. This section describes some of the different data-encoding schemes used in asynchronous design. Two common forms of asynchronous data encoding are single-rail encoding and 1-of-N encoding.

2.3.1 *Single-rail Encoding*

This is also called bundled-data or delay-matching encoding. In this type of encoding, data and data valid signals are in separate channels. The *data set* can be composed of many data signals and has an associated control signal called the *data valid* signal. Thus, the data set is said to be *bundled* with its control signal [1]. Each data signal in the data set is single-rail encoded, i.e. one physical signal is used for each bit of

information. The data valid signal is asserted only when the output is ready. Assertion of the data valid signal represents valid data being present on the data line(s) and completion of the functional computation. This is achieved by matching the delay of the data valid signal with the worst case delay of the *compute block* that produces the data. This technique is called *delay matching*. Circuits that use delay matching may encounter a race between the data valid signal and the data [3]. Figure 2.4 shows a block diagram of an asynchronous circuit that uses a bundled data approach. Each gate has a compute block and a delay block. The compute block is used to perform the logical operation of the gate, while the delay block is used to match the delay of the data valid signal to the worst case delay path of the compute block. Unequal wiring delays between compute and delay blocks can cause race conditions between the arrival of the data set and the data valid signal.

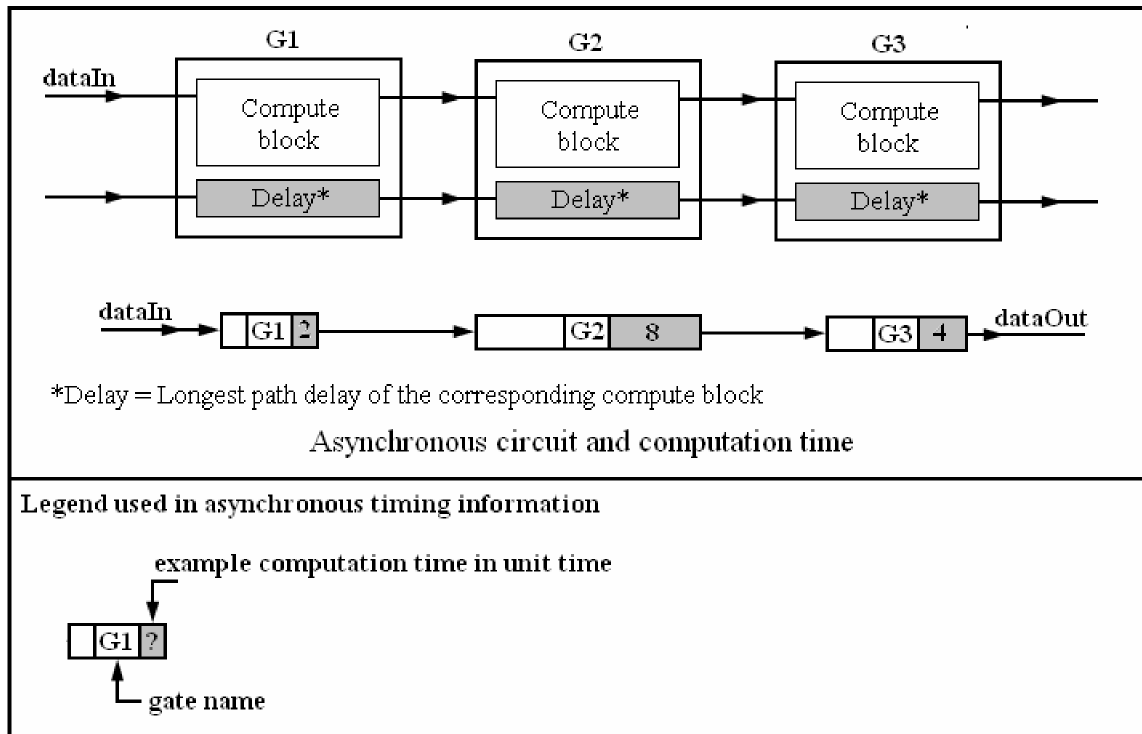


Figure 2.4 Asynchronous circuit using single-rail encoding

2.3.2 1-of-N Encoding

1-of-N encoding derives the data valid signal from the input signals. This is done by encoding N data values using N wires (i.e, encoding two values 0 and 1 requires two wires, with one wire asserted for each value). In this type of encoding there is no race between the arrival of the input data and the data valid signal, since the data valid signals are generated from the 1-of-N encoded input data. The most common 1-of-N encoding scheme is for N=2 and is called *dual-rail encoding* [23]. In a dual-rail encoding scheme, each bit is represented by a pair of wires.

As an example, consider a signal S represented in dual-rail format using the signals s_t and s_f, where the subscripts t, f stands for logic high and low values

respectively. Dual-rail signals can be encoded either by using 2-phase encoding or 4-phase encoding as shown in Figure 2.5. Figure 2.5a illustrates 4-phase encoding in which a logical high value is represented as '10' and a logical low value is encoded as '01' on the signal pair s_t, s_f. Data arrival at the input of a gate is detected by s_t and s_f being unequal to each other. After each transmission of a logical high/low value, a null code of '00' is placed on s_t, s_f, to prepare the signals for next data value [3]. The null code is also referred to as a *spacer code* [9] or a *reset code*. The code word '11' is considered as an invalid word in the dual-rail 4-phase encoding scheme. The logical values transmitted using 4-phase encoding scheme can be interpreted by using only the current code word irrespective of the previous codeword. This is called *context-free* encoding. A disadvantage of 4-phase encoding is that the time spent transmitting the spacer code can be thought of as wasted time, as this time period does not contain logic data. This return-to-null of the s_f, s_t signals also consumes power as it increases signal transitions. The advantage of 4-phase dual-rail encoding is the simpler logic used for input detection and output completion. Asynchronous circuits that use two phase encoding schemes has more complex logic for input detection and output completion as they need to remember the previous input data to interpret the current input data. Null code transmission time is used by dynamic gates for CMOS precharge operation before transmission of the next data value. Data arrival in two-phase and four-phase dual-rail signaling is detected by a transition on either s_t or s_f signal. Transitions of both the s_t and s_f signals during a code word transmission are not allowed.

Figure 2.5b shows a two-phase dual-rail encoding scheme. In this method, every transition on the dual-rail signals s_t and s_f represents a transmission. A '0' value is transmitted by toggling the s_f signal and a '1' value is transmitted by toggling the s_t signal. Both rising and falling transitions on s_t and s_f indicate transmissions of logic values. All the code words of 00, 01, 10 and 11 are used for transmitting logical data values. A logic gate must have an internal state that remembers the previously transmitted data value to determine if a code word represents a '1' or '0'. This is a disadvantage of 2-phase encoding as it requires more complex gate logic. The advantage of 2-phase encoding is that it is not necessary to transmit the null code after logical data transmission, resulting in less signal transitions and less power consumption.

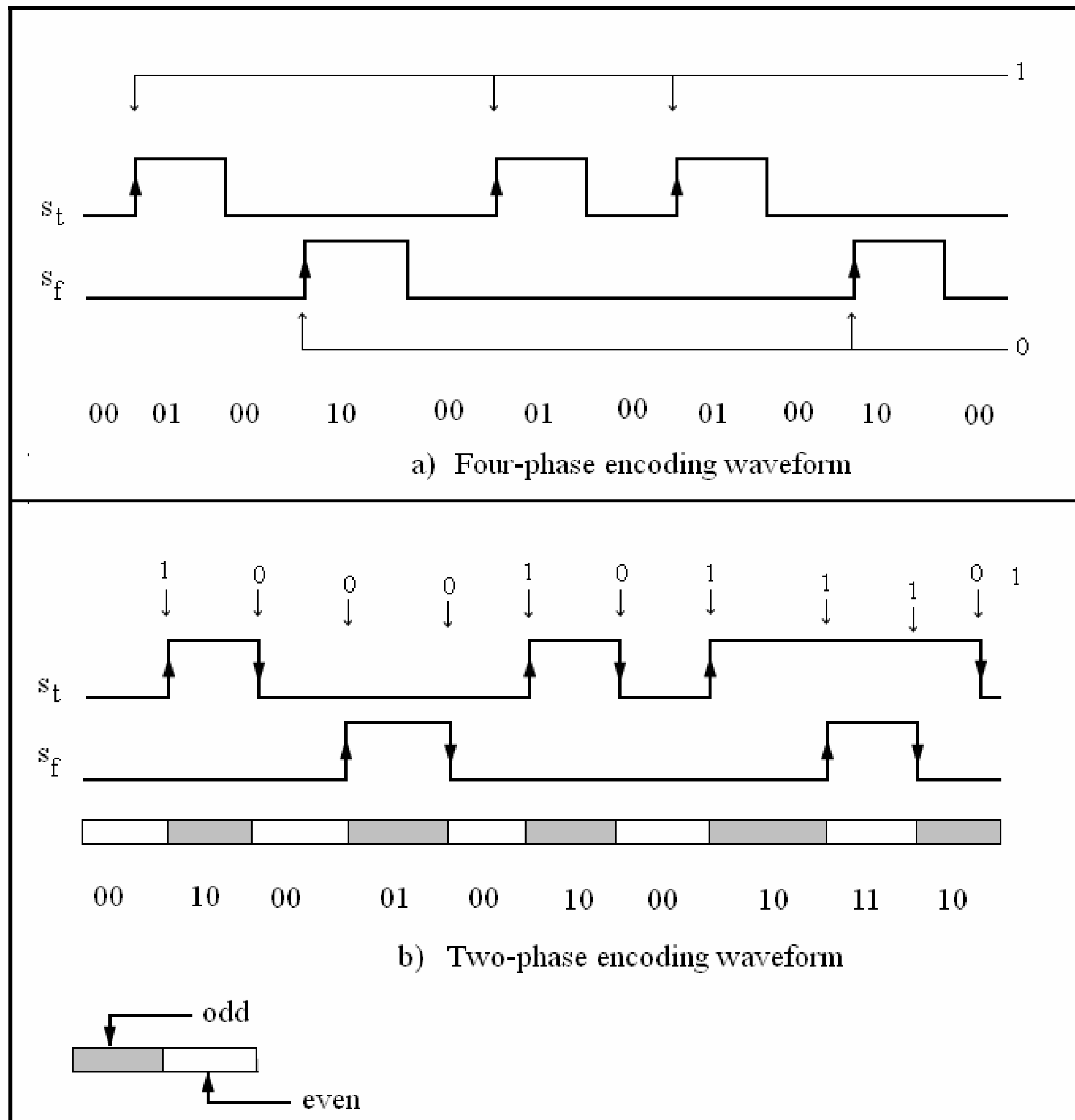


Figure 2.5 Dual-rail four-phase and two-phase encoding

2.4 Handshaking Protocols

In synchronous circuits, data flow between gates is synchronized by a global clock signal. An example synchronous gate is shown in Figure 2.6. It has combinational gates embedded between stages of sequential gates. Signals in the combinational block

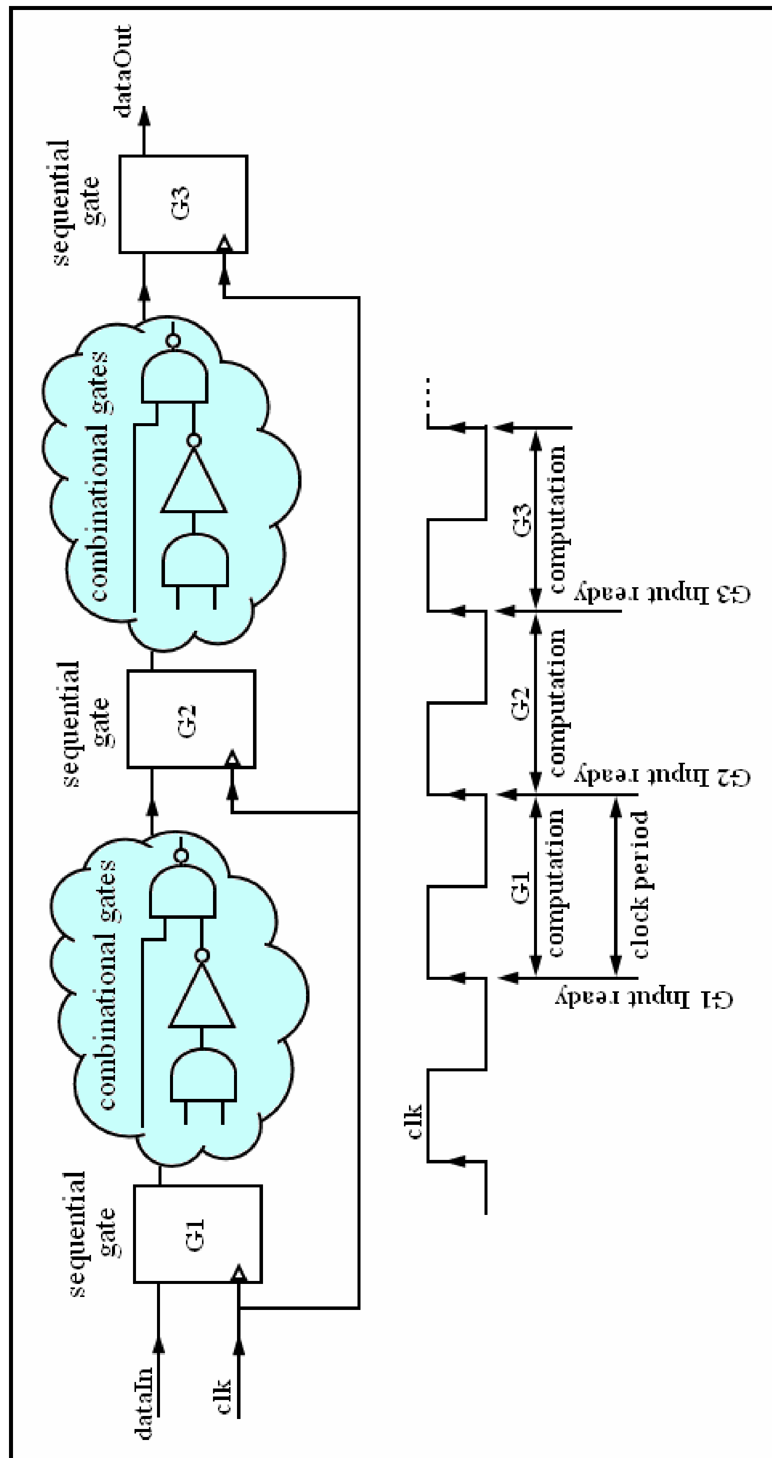


Figure 2.6 Synchronous circuit with sequential and combinational gates

may undergo several transitions in a clock period before they are captured by the sequential gates at the end of the clock cycle.

In asynchronous circuits, the data flow between gates is governed by handshaking protocols. There are two types of handshaking protocols, two-phase and four-phase.

2.4.1 Two-phase Handshaking

Two-phase handshaking protocol consists of two events for a complete handshaking cycle. In two-phase handshaking, each transition (rising/falling) on the handshaking signals represent either a request or an acknowledgement. It may be implemented either by using two wires, with each wire representing request and acknowledge signals, or by using a single wire representing both request and acknowledge signals [25]. In the two wire implementation, a transition on the request line represents a new request and a transition on the acknowledge line represents an acknowledgement of the request. In the single wire implementation, a rising edge represents a request and a falling edge represents an acknowledgement or vice-versa. Two-phase handshaking protocols using two wires and one wire are shown in Figure 2.7a and Figure 2.7b respectively.

2.4.2 Four-phase Handshaking

A complete four-phase handshaking cycle has four events and is level based. The four phases represents start of handshake (request), process completion (acknowledgement), reset of request to null, and reset of acknowledgement to null for completing the handshaking cycle. Implementations often use null states to precharge dynamic logic

gates. Four-phase handshaking operation is shown in Figure 2.7c. The request line goes high, indicating that there is a request for new data from the destination gate. The source gate responds by raising the acknowledgement signal when its output data is ready. The destination gate negates the request line after consuming the input data. The source gate then negates its acknowledgment signal to complete the four-phase handshaking cycle.

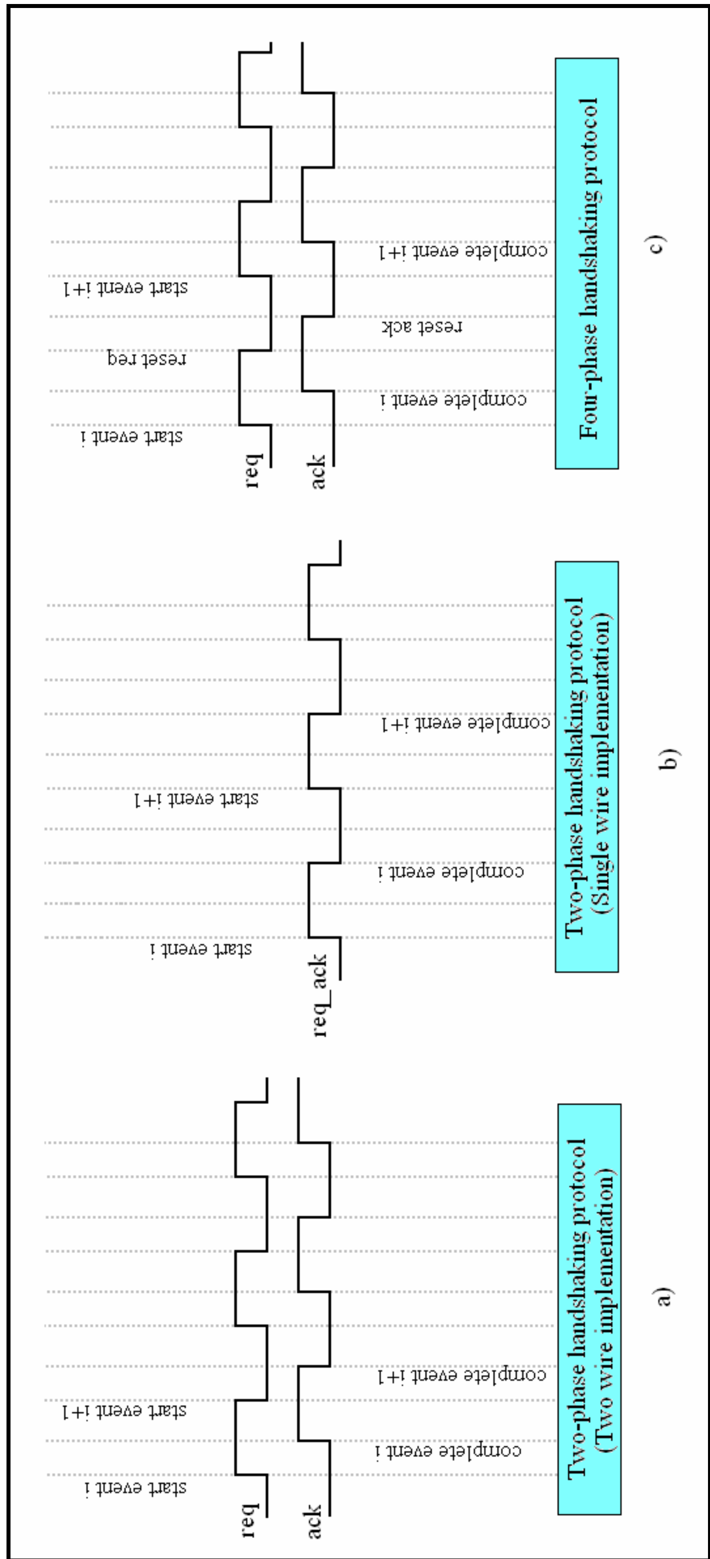


Figure 2.7 Handshaking protocols

CHAPTER III

PHASED LOGIC SYSTEMS & QUASI-DELAY INSENSITIVE SYSTEMS

Previous chapters gave an overview of asynchronous design. This chapter describes asynchronous phased logic (PL) and quasi-delay insensitive systems (QDI). PL[9] is a methodology used to describe asynchronous circuit operations in terms of data flow in a marked graph. QDI circuits use four-phase dual-rail signaling and assume isochronic forks [7]. Isochronic forks are the splits in the circuits where the difference in the wire delays of the splits is negligible when compared to gate delays [13, 14, 7]. Section 1 gives an overview of PL systems and section 2 explains QDI circuits.

3.1 Phased Logic Systems

The Phased logic (*PL*) [9] methodology permits translation of synchronous systems to asynchronous systems. *Marked graphs* (*MG*) are used to represent gate operation and data flow in phased logic systems. *MG*s are a subclass of directed graphs called *petri-nets* (*PN*) that are often used in representing asynchronous, concurrent systems. This section discusses the concepts of PL circuit and explains the properties of *PNs, MG*s as used in PL systems.

3.1.1 Level-encoded Dual-rail Encoding

PL circuits use a form of two-phase dual-rail encoding known as level-encoded dual rail (LEDR) [21] encoding. This is different from the previously discussed traditional two-phase dual-rail encoding in Chapter 2. Figure 3.1 shows the LEDR encoding scheme. Observe that the two signal lines used in two-phase encoding are named *value* 's_v' and *phase* 's_t' lines respectively, as per the naming convention adopted by Linder [9]. The *value* signal contains the logical value ('0' or '1') of the transmitted data, while the *phase* signal is designated as either *even* or *odd*. Thus, transmitted logical data is named even 0, odd 0, even 1, or odd 1. The phase always changes between successive transmissions, while the value may or may not change. In LEDR encoding only the *value* wire or the *phase* wire changes state between each transmission.

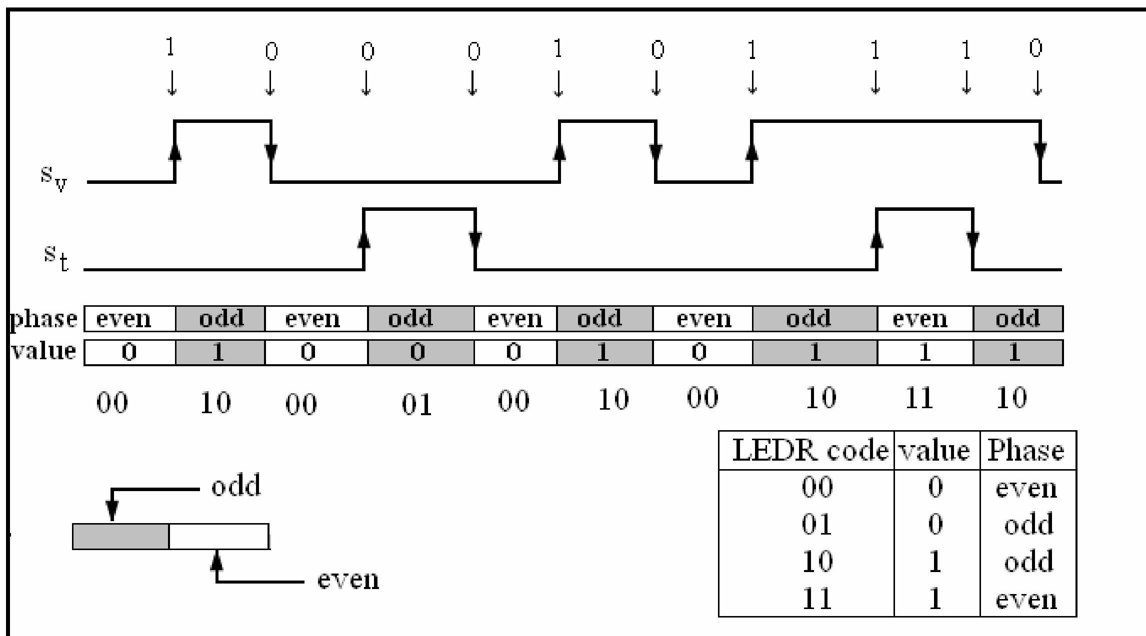


Figure 3.1 LEDR Encoding

3.1.2 Phased Logic Gates

Phased logic systems use phase logic gates for logic computation. The inputs and outputs of PL gates are LEDR encoded, with data containing both phase and value components as previously mentioned. Just as a *PL* signal has an even/odd phase, a *PL* gate also has a phase associated with it. The matching of all the input signal phases and the gate phase implies valid input data and the gate is ready for the computation. Logic computation by a PL gate at the arrival of valid input data is called *firing*. Matching of all input phases with gate phase causes the gate to *fire*. Matching of input phase and the gate phase is graphically represented by placing a dot at the corresponding input. This dot is called a *token*. The correspondence between the gate phase and the tokens is shown in Figure 3.2. The gate fires if there are tokens on all its inputs, causing the gate phase and output phase to toggle. PL gate firing causes consumption of all input tokens because the gate phase is toggled; each gate firing also places a token on all output (i.e. the phase of all outputs are toggled).

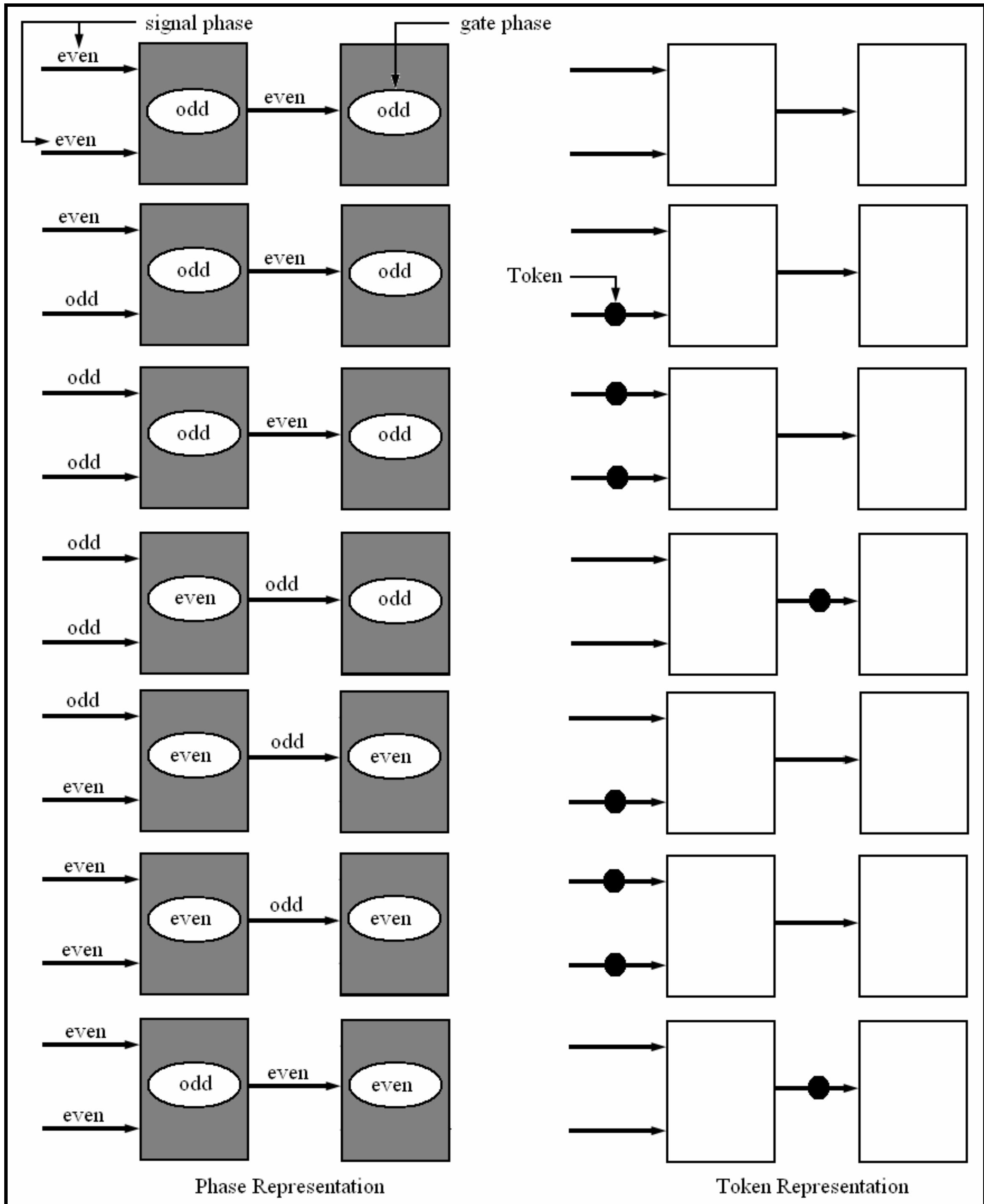


Figure 3.2 PL Gate phase and corresponding token representation

3.1.3 Petri-nets

The *Petri-nets* definitions in this section are adopted from [10]. Petri-nets are directed, bipartite graphs with two types of nodes namely, *places* and *transistions,* representing conditions and events in that order. Places contain tokens, and the number of tokens in a place *p*, is represented by *M(p)* where *M* is the marking. *M* is a function of the form M:F {0, 1, 2, … }. *PNs* have an initial state referred as the *initial marking* (M_0). Places connect to transitions, and transitions connect to places. The arcs going from places to transitions and from transitions to places represent the flow relationship of the system. A Petri-net whose places can hold an unlimited number of tokens is called an *infinite capacity* net whereas a Petri-net with a bounded number of tokens in its places is called a *finite capacity* net or *k-bounded* net. A transition is *enabled* if all of its input places contain at least one token. An enabled transition can *fire*, which places one token at its output places and removes one token from each of its input places. A source transition that does not have any input places are unconditionally enabled, while sink transitions with no output places consume their input tokens.

The firing sequence of a *PN* consists of the sequence of markings generated by the firings of enabled transitions. If the firing sequence is represented by a non-empty set $\{M_0, (t_1,M_1), (t_2,M_2), \dots (t_n,M_n)\}$ then the marking, M_n, can be reached from the initial marking M_0, by following the firing sequence.

A *petri-net* is said to be live if an enabled transition is possible from the current marking, M_n, by following the firing sequence, regardless of the current transition that has been reached starting from the initial marking, M_0. Thus, a live PN never encounters

a deadlock situation. A *petri-net* with a live initial marking, $M_{0,}$ will always result in a live network.

3.1.4 Marked Graph

A marked graph [9] is a subclass of petri-nets where each place has only one input transition and one output transition. The transitions of the *MG* are represented by the graph vertices, with arcs only shown between graph vertices and the intervening places assumed to be present. A directed circuit in an *MG* is defined as a directed path that starts and ends at the same transition. Like *PNs*, *MGs* also have an initial token marking. *MGs* are used to represent *PL* systems. Two properties [17] of *MGs* used in the *PL* methodology are safety and liveness properties and are defined as follows [9]:

Theorem 1: An *MG* is live if and only if the initial token marking places at least one token on each directed circuit of the *MG*.

Theorem 2: An *MG* is safe if and only if all of the edges are part of a directed circuit that contains at most one token.

A transition in a marked graph fires whenever all of its input arcs contain a token. Firing consumes one token from each input arc and places one token on each output arc. In a PL system, a MG transition is a PL gate, while arcs are signals between PL gates. PL circuits require a safe and live marked graph for effective functioning of the system. A live initial token marking ensures that there is always a gate ready to fire, i.e., that the circuit does not enter deadlock condition. A safe PL system means that there is only one token on an input or output arc at any point in time. Safety also implies that a PL gate cannot fire until its output tokens have been consumed by its destination gates.

3.1.5 Mapping Clocked Netlists to PL Netlists

The Phased logic methodology produces delay insensitive circuits starting from a clocked system. Linder in [9] introduced the concept of *barrier* gates and *through* gates to distinguish sequential and combinational gates in the clocked system to facilitate mapping the clocked netlist to a PL netlist. Sequential gates such as DFFs are mapped to *barrier* gates and any combinational gates are mapped to *through* gates. The distinction between combinational and sequential gates is necessary to satisfy the initial token marking rules of PL systems. In static CMOS PL systems, Barrier and through gates are connected by using LEDR signals. The translation process also involves inserting *feedback* signals to make the PL circuits live and safe. Feedback signals are single-rail inputs and represent the output phase of the destination gate.

This section gives a brief description of the translation algorithm used to map a clocked netlist to a PL netlist. For a detailed explanation of this algorithm, please refer to [9] and [17]. The algorithm is divided into four stages: *Initial token marking*, *splitter gate insertion*, *marking of safe and unsafe signals*, and *feedback insertion*.

Stage 1: Initial Token Marking

In this stage, a one-to-one mapping of the clocked netlist to a PL netlist is done. Single-rail data signals in the clocked netlist are replaced by LEDR data signals. An external global reset signal is used to reset all the PL gates to the same phase at the release of reset. During one-to-one mapping of the clocked netlist to PL netlist, sequential gates are mapped to barrier gates and combinational logic to through gates. Token markings on the barrier and through gates are defined by the initial token marking rules.

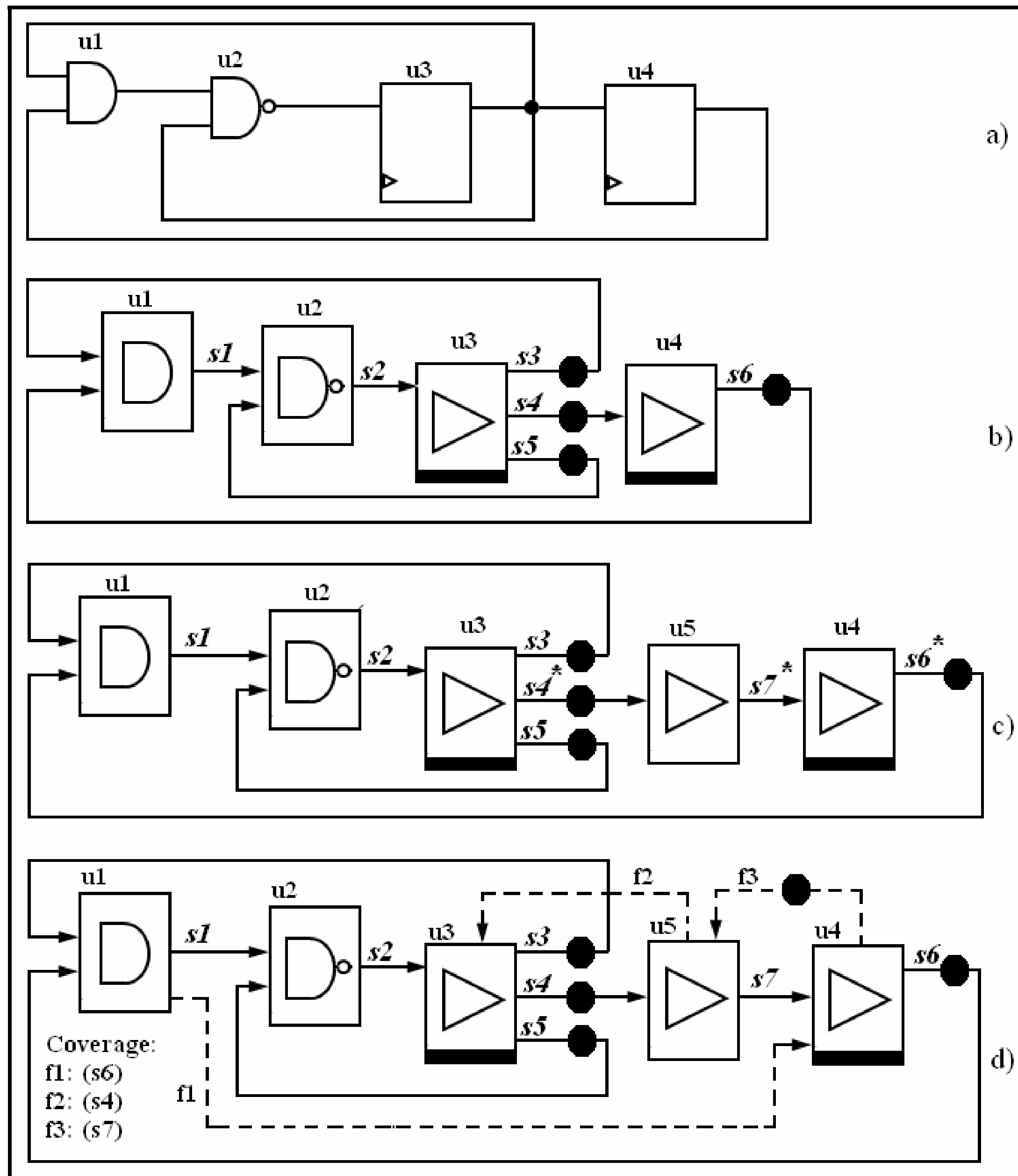
Initial token marking rules require tokens at the outputs of barrier gates. This means that the barrier gate outputs are connected to gates whose phase is equal to the barrier output phase. Initial token marking rules also require that the non-feedback outputs of a through gate cannot have tokens. This implies that a through gates output must be connected to the destination gate whose phase is opposite to that of the through gate output phase. Figure 3.3b shows the initial token markings at the release of reset.

Stage 2: Splitter Gate Insertion

Direct barrier to barrier gate connections can cause safety problems when feedback signals are inserted, so splitter gates are inserted to break barrier gate to barrier gate paths. Splitter gates are through gates and act as logical buffers when they are inserted in PL circuits. Figure 3.3c shows the PL system after the insertion of splitter gate. Splitter gate s1 is inserted between the two barrier gates u2 and u3.

Stage 3: Marking of safe signals

All signals that are a part of a directed circuit that have only one token after the initial token marking are marked as *safe* signals. Directed circuits are those circuits that has at least one path that originate and terminate at the same gate. If a gate has multiple fanouts, each fanout is treated as a separate arc in the marked graph and all of the arcs have to be safe in order to achieve a live and safe PL circuit. After the initial token markings, only the barrier gate outputs have an associated token, so any signal in a directed circuit that has only one barrier gate is considered safe. Figure 3.3c shows that the signals *s4*, *s6* and *s7* are unsafe after the initial token marking as they do not belong to a directed circuit with only one token. The unsafe signals are identified with a * designation.



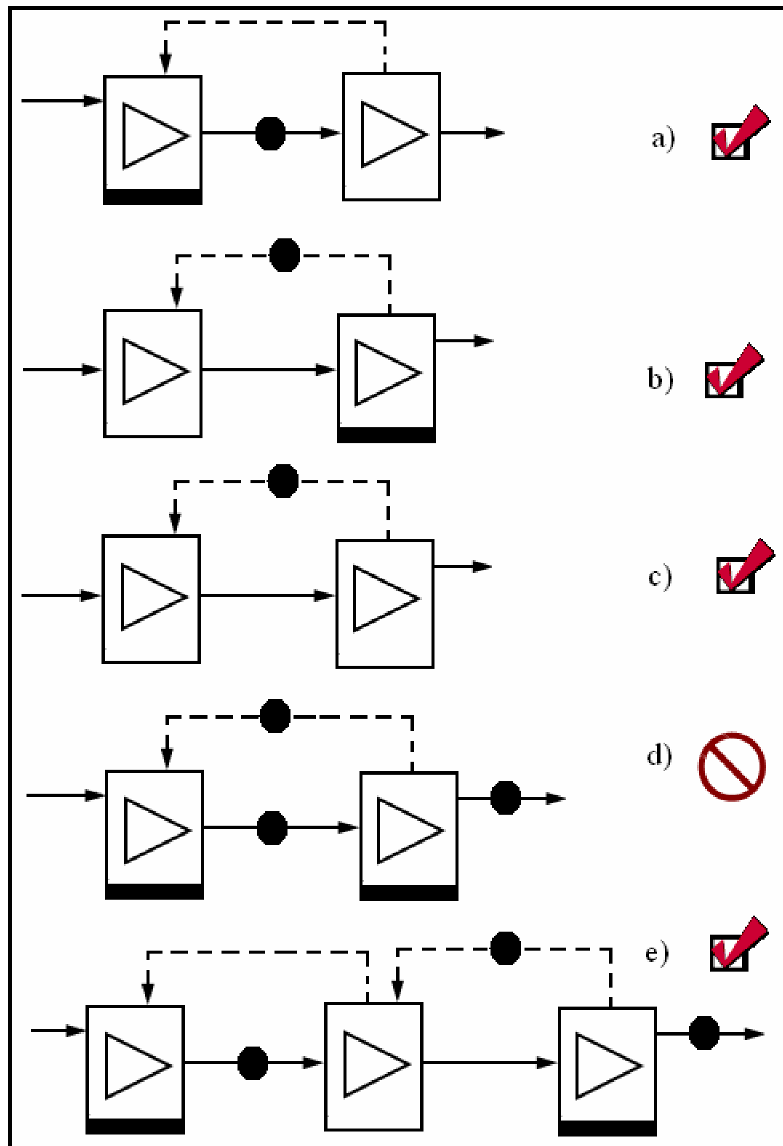
Note (a) Clocked circuit. (b) Initial token marking. (c) Splitter gate insertion. (d) Feedback insertion

Figure 3.3 PL translation steps

Stage 4: Feedback Insertion

Feedback signals are single-rail acknowledgement signals that contain the gate phase. Feedback signals are used when necessary to create new directed circuits with only one token in the initial token marking to convert *unsafe* signals to *safe* signals. If an initial *unsafe* signal is made *safe* by the addition of a feedback signal, then the signal is said to be *covered* by the feedback. Figure 3.3d represents the PL circuit after feedback signal insertion. The signals *s4*, *s6,* and *s7* that were initially unsafe are now covered by feedback signals *f2*, *f1,* and *f3* respectively. Feedback insertion rules and the corresponding initial token markings are shown in Figure 3.4. All of the allowable feedback insertion configurations are indicated with a check mark besides the Figure. Feedback signals that originate and terminate on the through gate should have an initial token as seen in Figure 3.4b. Feedback signals originating from a through gate and terminating on a barrier gate should not have an initial token (see Figure 3.4a). Any feedback signal originating from a barrier gate has an initial token because all outputs of a barrier gate have an initial token. Feedback signals that originate from and terminate on a through gate contain an initial token as shown in Figure 3.4c. Figure 3.4d shows that feedback signals cannot originate and terminate at a barrier gate as this creates a directed circuit with two token, which is a safety violation. To solve this problem, a splitter gate is inserted between the two barrier gates. The initial token marking of the two barrier gates separated by a splitter gate is indicated in Figure 3.4e.

There are multiple options for feedback insertion. A scoring function is used to aid in the process of feedback insertion [17]. The scoring function is given by



Note (a) Through gate to barrier gate feedback.
(b) Barrier gate to through gate feedback.
(c) Through gate to through gate feedback.
(d) Forbidden barrier gate to barrier gate feedback
(e) Splitter gates between Barrier gates

Figure 3.4 PL feedback insertion rules and corresponding token markings

score = S – F/k – p * L

The variables used by the scoring function are the number of unsafe signals covered by feedback insertion *S*, a user-defined constant *k* that restricts the number of feedbacks on a single node, feedback length *L*, and a user-specified constant *p* that favors shorter feedbacks over longer feedbacks.

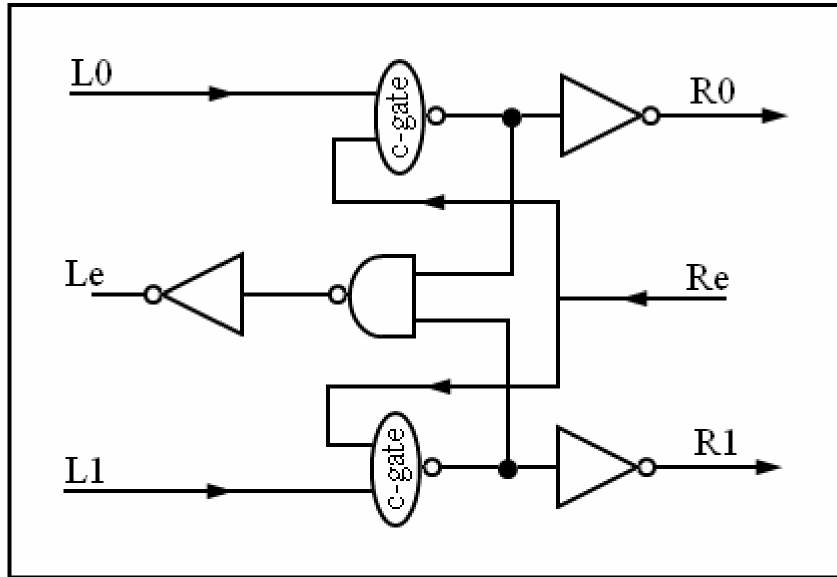
3.2 Quasi-Delay Insensitive Systems

Quasi-delay insensitive (QDI) circuits do not have any assumptions about delay of gates and wires except for isochronic forks [7, 13, 14, 32, 49]. The most popular QDI gates used in the design of asynchronous QDI circuits are Caltech's weak-conditioned half buffer (WCHB), precharged half buffer (PCHB), and precharged full buffer (PCFB) [5, 6, 7, 12]. These gates are viewed as communicating processes [26], and transfer data by using a four-phase handshaking protocol.

3.2.1 Weak-conditioned Half Buffer

Figure 3.5 shows the circuit diagram of a WCHB gate. Signals (*L0*, *L1*), (*R0*, *R1*) represent false and true inputs and outputs respectively. L_e and R_e are active low signals representing input and output acknowledgement signals. Initially, both L_e and R_e are high. Assertion of either *L0* or *L1* asserts L_e, thus acknowledging the input arrival. The output data is sent to destination gates that respond by asserting R_e. The gate then waits for input neutrality (either *L0* or *L1* is negated) before resetting the output. Its disadvantage is that the input neutrality is checked before resetting the output. This is known as *weak-conditioned logic* [11, 12]. Weak-conditioned logic increases the forward latency of

circuits built using WCHB gates as the gate waits for input neutrality before resetting the output.



Note Adopted from [12]

Figure 3.5 Weak conditioned half buffer gate

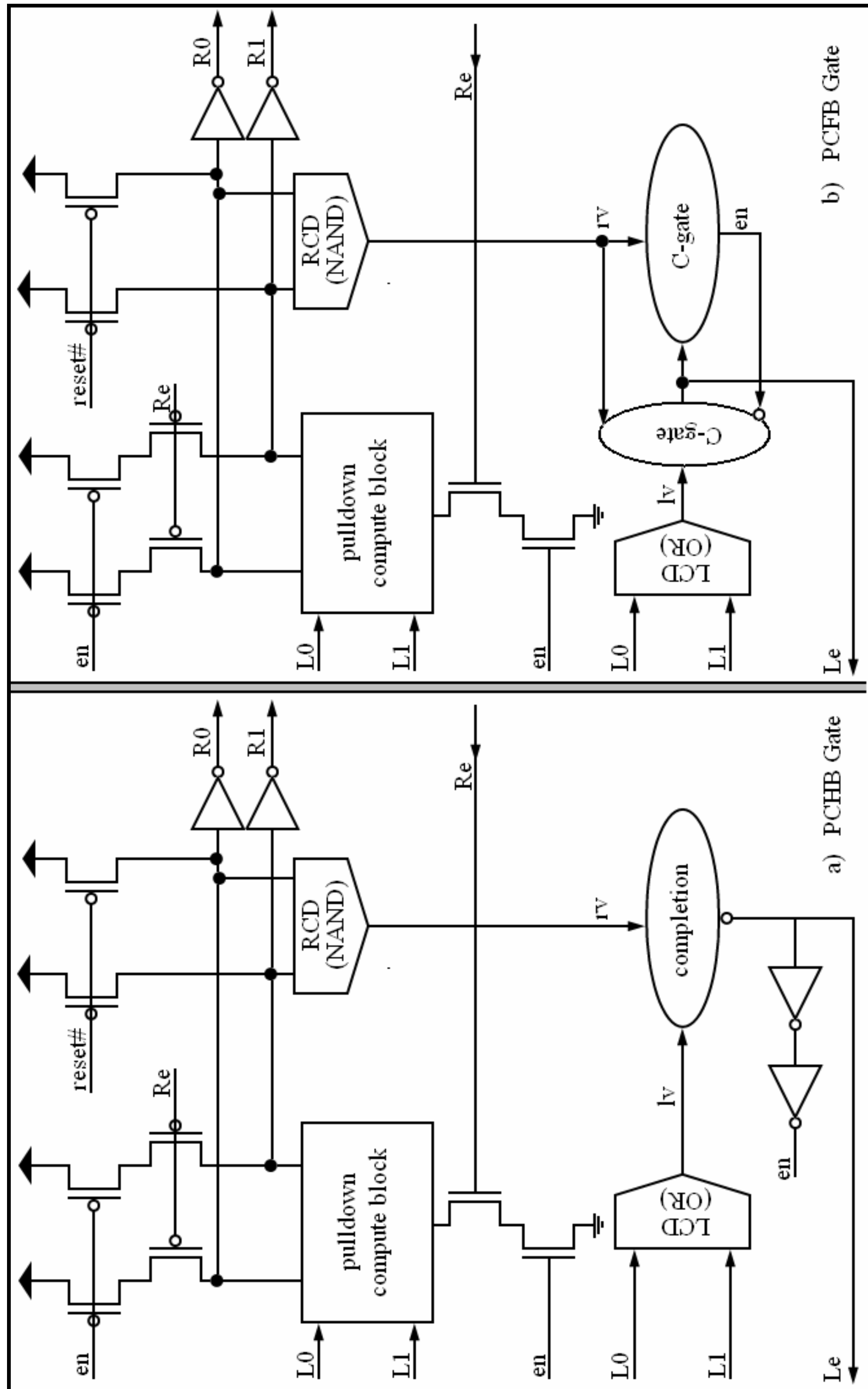
3.2.2 Precharged Half Buffer and Precharge Full Buffer

The PCHB and PCFB gates shown in Figure 3.6a and Figure 3.6b have separate input/output validity and neutrality checking circuits. This eliminates the dependency of resetting the output based on the input neutrality [12].

In Figure 3.6, the LCD block represents the input validity and neutrality detection circuit, while RCD represents the output completion and neutrality detection circuit. The compute block uses dual-rail domino logic for input evaluation. The difference between PCHB and PCFB circuits is that in PCFB gates input neutrality and output neutrality can

occur in parallel, whereas in PCHB gates the input will be neutralized after the output is neutralized.

The PCHB design has dual-rail input/output signals, a pull down compute block, a precharge circuit, completion detection circuits, handshaking signals, and a C-element [20]. The precharge circuit of a PCHB gate is composed of two *pmos* transistors in series. Dual-rail inputs are fed to the pulldown *nmos* compute block to generate the complimentary outputs. The dual-rail inputs are also fed to input completion (LCD) and output completion (RCD) detection circuits. There are two active low handshaking signals: *input acknowledgement* signal L_e and *output acknowledgement* signal R_e. The L_e and R_e signals are used to establish proper data communications between PCHB gates. Signal L_e is an acknowledgement sent to the source gate and R_e is an acknowledgement coming from the destination gate. Gates exchange data by using a four-phase handshaking protocol.



Note Adopted from [7]

Figure 3.6 PCHB and PCFB gates

3.2.3 Four-phase Handshaking in PCHB Gates

Figure 3.7 demonstrate the four-phase handshaking protocol of an PCHB AND gate. In this example, L_e and R_e represent active low input and output acknowledgement signals respectively, pairs (A_f,A_t) and (B_f,B_t) are dual-rail inputs and pair (Y_f,Y_t) is the dual-rail output.

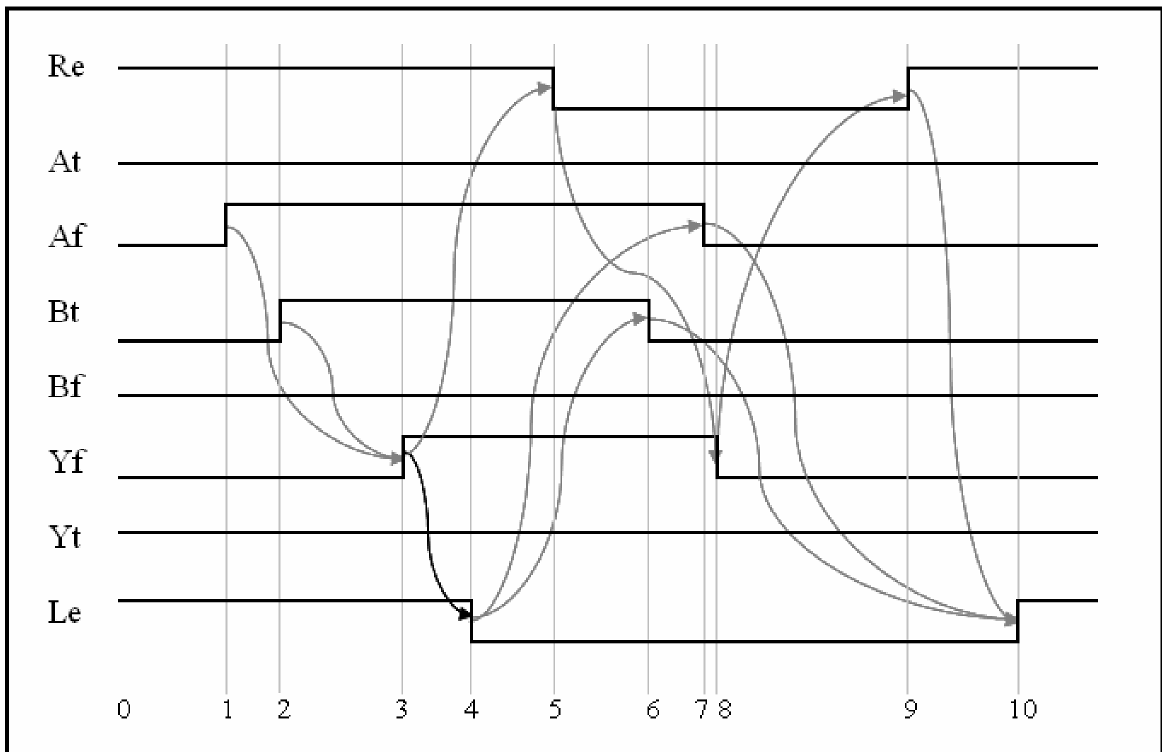


Figure 3.7 2-input PCHB AND gate operation

Initially after reset, R_e and L_e are both high. Arrival of all of the valid inputs at *time 2* causes the circuit to evaluate the new input data, producing a valid output at *time 3*. Signal L_e goes low at *time 4* acknowledging the input signals, stating that the input is processed and a valid output is ready. After some unknown time at *time 5* (depending

upon the destination gates), the output acknowledgement R_e goes low confirming the output consumption by the destination gate. Low values on the L_e and R_e signals precharge the dual-rail output Y_f, Y_t forcing them back low, as seen at *time 8*. R_e returns to '1' at *time 9* indicating that the destination gate is precharged and ready to accept new data. Signal L_e is negated at *time 10* after the negation of both input and output, completing the four-phase handshaking.

3.2.4 PCHB Gates Internal Operation

In this section, the relationship between the four-phase protocol of Figure 3.7 and the internal PCHB gate operation of Figure 3.6a is discussed. At reset, the L_e and R_e signal are both high. After the release of reset, the gate waits for the arrival of all of the valid inputs, where input arrival is defined as assertion of one of the dual-rail wires for a dual-rail input signal. Arrival of all inputs, regardless of the order of arrival, causes the input validity circuit (LCD) to go high and the compute block to produce complimentary outputs. The computation of the complimentary outputs after input arrival is denoted as the output "firing". The output firing causes the output completion detection circuit to be asserted high. Assertion of the outputs of both the LCD and RCD circuits forces the output L_e of the C-element to be asserted low. Note that the circuit produces an input acknowledgement L_e only after the arrival of all inputs and the evaluation of the outputs. The buffered L_e is used as an enable signal *en* in the PCHB gate for enabling the compute block. A low L_e precharges the preceding gate, causing the inputs to be negated. The assertion of L_e also states that the PCHB gate will not accept the next new input until it precharge its outputs to a zero. Output consumption by the destination gate (destination

gate firing) causes the output acknowledgement R_e to go low. At this stage both the input acknowledgement L_e and output acknowledgement R_e are low, enabling the PCHB gate to precharge.

Figure 3.8 demonstrates the data communication between the PCHB gates using a circuit composed of three PCHB gates (*G0, G1, G2*) represented as black boxes along with *data* and *handshaking* signals. Each gate alternates between the *evaluation* state and the *precharge* state.

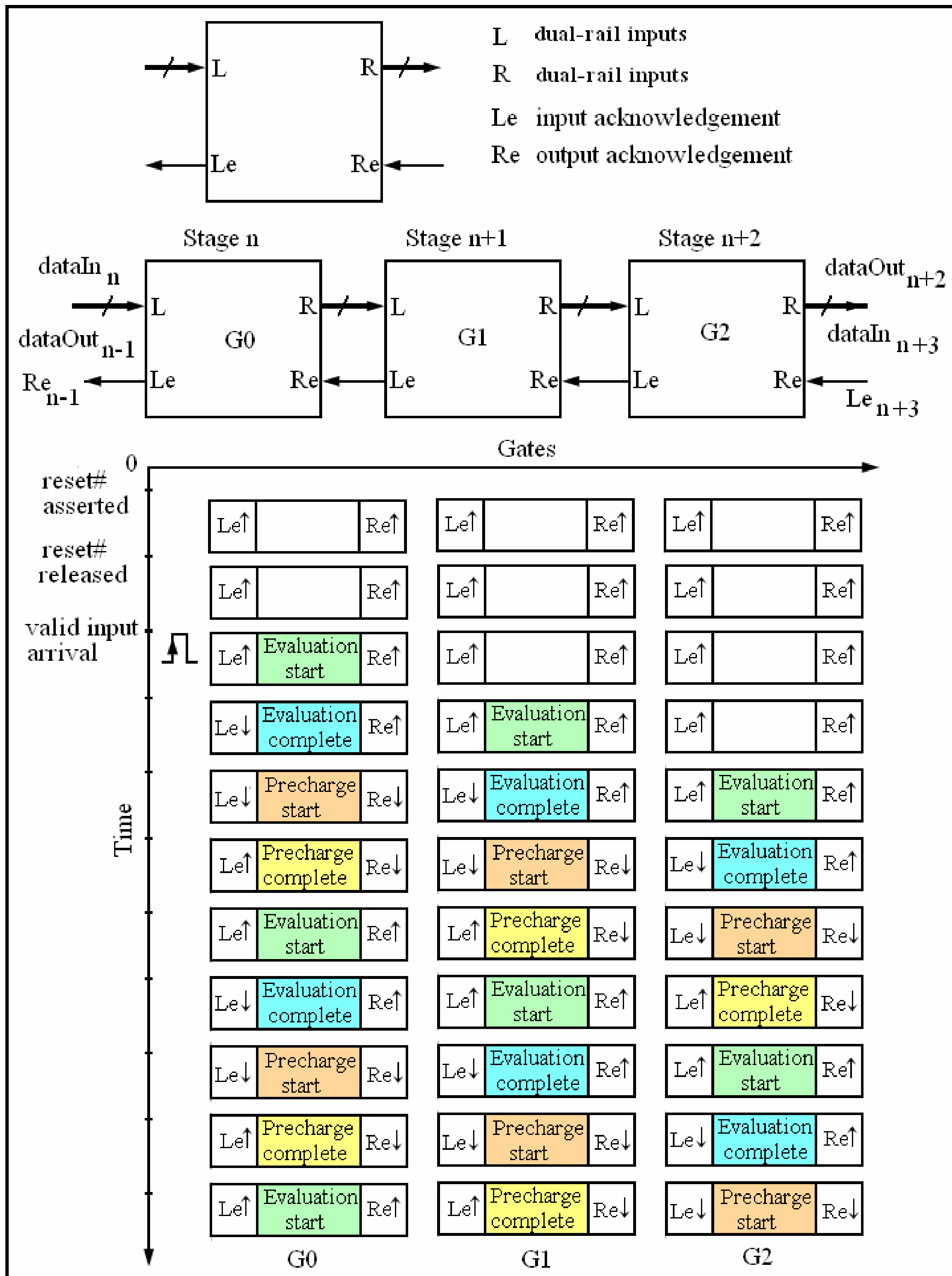


Figure 3.8 Data communication in PCHB gates

3.2.5 Caltech's Asynchronous Design Methodology

Both synchronous and asynchronous circuits can be modeled using concurrent processes that communicate with each other. The Caltech's asynchronous design methodology translates communicating sequential processes (CSP) into quasi-delay insensitive asynchronous systems by using a series of systematic semantics-preserving transformations. The transformation process is outlined in the following section. For a detailed explanation of this methodology, please refer to [27].

Asynchronous circuit behavior defined using a sequential Communication Hardware Processes (CHP) program is the starting point for this methodology. Table 3.1 adopted from [7] explains the CHP notation used to describe asynchronous circuits.

Table 3.1 CHP notation

Notation	Meaning
*[S]	*Repeat statement S for ever*
x↑	*x is high, where x is a boolean variable*
x↓	*x is low, where x is a boolean variable*
;	*sequential composition of two elementary actions*
,	*concurrent composition of two elementary actions*
$[G_1 \rightarrow S_1[] .. G_i \rightarrow S_i[]]$	*G_i represent Boolean expression (guards) and S_i represent program parts $G_i \rightarrow S_i$ is read as waiting until one of the guards is true and then executing, the corresponding S_i with true guard G_i*
[G]	*Waiting for condition G to become true*
[B]; x↑; [¬B]; x↓	*Represents four-phase handshaking by alternation of waits and boolean assignments*
R!x	*Send value x over the channel R*
L?x	*Receive value x over the channel L*

Transformation 1: Process Decomposition

In this step, asynchronous circuit behavior defined using CHP notation is converted into a set of interactive concurrent CHP processes. This iterative step is repeated until the transformation leads to simple processes that communicate with each other using input and output channels. The decomposition process leads to asynchronous

circuits that use one of the three possible Caltech QDI templates: WCHB, PCHB, or PCFB as discussed in the preceding section.

Transformation 2: Hand Shaking Expansion

This transformation process implements a communication channel between processes using signal wires. The channel (C, D) can be implemented using a pair of wires (c_o *w* d_i) and (d_o *w* c_i). After implementing the communication channel, a communication protocol must be established to exchange data using handshaking signals. The communication action between the processes is replaced by the four-phase handshaking protocol. A technique called HSE *reshuffling* is used to rearrange the non-data dependent portion of the four-phase communication. Reshuffling improves the speed and size by reducing the number of sequencing and the state variables required to implement the HSE. There are three types of reshufflings: *weak-conditioned half-buffer* (WCHB), *precharge-logic half buffer* (PCHB), and *precharge-logic full buffer* (PCFB). The WCHB, PCHB and PCFB gates described in the previous section are the result of different types of reshuffling.

A simple one-bit buffer circuit example adopted from [5] is used to outline the three types of reshuffling. The buffer circuit receives the data *x* on channel *L* and sends it on channel *R* without any computation. The CHP notation for this buffer is given by *[*L*?*x*; *R*!*x*]. The handshaking expansion of the communication action between the two channels L and R is represented as:

$$*[\ [L^0 \rightarrow x^0\uparrow\ []\ L^1 \rightarrow x^1\uparrow];\ L^e\downarrow;\ [\neg L^0 \wedge \neg L^1];\ L^e\uparrow;$$

$$[x^0 \rightarrow R^0\uparrow\ []\ x^1 \rightarrow R^1\uparrow];\ [\neg R^e];\ R^0\downarrow, R^1\downarrow; [\neg R^e];]$$

The three types of reshufflings for the buffer circuit are shown below.

$\text{WCHB} \equiv *[[R^e]; [L^0 \rightarrow R^0\uparrow\ []\ L^1 \rightarrow R^1\uparrow]; L^e\downarrow;$

$[\neg R^e]; [\neg L^0 \wedge \neg L^1]; R^0\downarrow, R^1\downarrow; L^e\uparrow;]$

As can be seen from the above WCHB reshuffling, the gate waits for the neutrality of the output before resetting the outputs. To eliminate this dependency of output reset on the input neutrality, precharge logic reshufflings are used, which postpone the neutrality of inputs $[\neg L^0 \wedge \neg L^1]$. The PCHB reshuffling of buffer circuit is:

$\text{PCHB} \equiv *[[R^e]; [L^0 \rightarrow R^0\uparrow\ []\ L^1 \rightarrow R^1\uparrow]; L^e\downarrow;$

$[\neg R^e]; R^0\downarrow, R^1\downarrow; [\neg L^0 \wedge \neg L^1]; L^e\uparrow;]$

In circuits where speed is critical, PCFB reshuffling can be used as it allows the reset phases of the input and output to execute concurrently. Although this requires an additional state variable *en* resulting in an increase in gate size, it also produces a faster CMOS implementation by removing a few transitions in the handshake cycle. So the PCFB reshufflings are primarily used in circuits that trade area for speed.

$\text{PCFB} \equiv *[[R^e]; [L^0 \rightarrow R^0\uparrow\ []\ L^1 \rightarrow R^1\uparrow]; L^e\downarrow; en\downarrow$

$([\neg R^e]; R^0\downarrow, R^1\downarrow), ([\neg L^0 \wedge \neg L^1]; L^e\uparrow); en\uparrow]$

Transformation 3: Production Rules Expansion

Hand shaking expansions obtained from the second transformation process is converted into a set of production rules (*PR)* that eliminates explicit sequencing. The production rule $G \rightarrow x\uparrow$ suggests that the node *x* goes high after the guard (G) becomes true. The guard function ensures that the PRs are fired as specified by the hand shaking expansion. The guard function G is said to be stable if it holds the value until the

production rule executes. In this step, the PRs that match operator semantics are identified to form a network of operators. Examples of primary operators are *and*, *or*, *C-element*, *wire*, and *fork*. Table 3.2 obtained from [27] shows the production rule sets for these primary operators.

Table 3.2 Operators and the production rules

Operator	Production Rules
C-element	$(x,y)\ \underline{C}\ z \equiv x \wedge y \rightarrow z\uparrow$ $\neg x \wedge \neg y \rightarrow z\downarrow$
AND	$(x,y)\ \Delta\ z \equiv x \wedge y \rightarrow z\uparrow$ $\neg x \vee \neg y \rightarrow z\downarrow$
OR	$(x,y)\ \underline{\vee}\ z \equiv x \vee y \rightarrow z\uparrow$ $\neg x \wedge \neg y \rightarrow z\downarrow$
Wire	$(x,y)\ \underline{w}\ z \equiv x \rightarrow y\uparrow$ $\neg x \rightarrow y\downarrow$
Fork	$x\ \underline{f}\ (y,z) \equiv x \rightarrow y\uparrow,\ z\uparrow$ $\neg x \rightarrow y\downarrow,\ z\downarrow$

Note Adopted from [27]

Transformation 4: Operator Reduction

In the final transformation process, production rule sets in the program and the production rule sets of operators are matched to represent the program as a network of operators. If the production rules cannot be mapped into the set of operators, the guards of the production rules are transformed to that of a guard of operators. A complex guard with a large number of variables is broken down into smaller production rules using new internal variables. This stage maps PRs to standard hardware components and state variables.

3.3 Summary

This chapter introduced two methods of implementing asynchronous circuits: the phased logic methodology and Caltech's asynchronous design methodology. The Phased logic methodology converts synchronous circuits into delay insensitive circuits. Since the PL methodology uses a clocked netlist as an input and automatically converts it into an asynchronous netlist, it does not require an expert to produce asynchronous circuits. Anyone with solid foundation in synchronous design can use the PL methodology to produce delay insensitive asynchronous circuits. This eliminates the arduous and lengthy learning period required in designing asynchronous circuits using other asynchronous methodologies.

The Caltech's asynchronous methodology described in this chapter produces QDI circuits starting from a behavioral description of the design in CHP notation. The advantages of Caltech's asynchronous QDI circuits are that they have fast forward latency when compared to synchronous domino logic, as they do not require output latch. They are also energy efficient and smaller in size when compared to non-pipelined QDI circuits [12]. The disadvantages of Caltech's asynchronous QDI methodology are the use of the CHP language and the systematic, semantics-preserving transformations that requires a skilled designer for creating an efficient asynchronous QDI implementation.

As will be seen later, this research combines the PL methodology and Caltech's QDI gates with their fast forward latency to produce QDI asynchronous circuits, thus using the best parts of each methodology.

CHAPTER IV

PHASED LOGIC FOR QUASI-DELAY INSENSITIVE CIRCUITS

This research extends the concept of phased logic and marked graphs to quasi-delay insensitive circuits. QDI systems with PL features are termed PL-QDI systems. The first step of extending PL concepts to QDI circuits is to extend the concept of token abstraction to QDI gates.

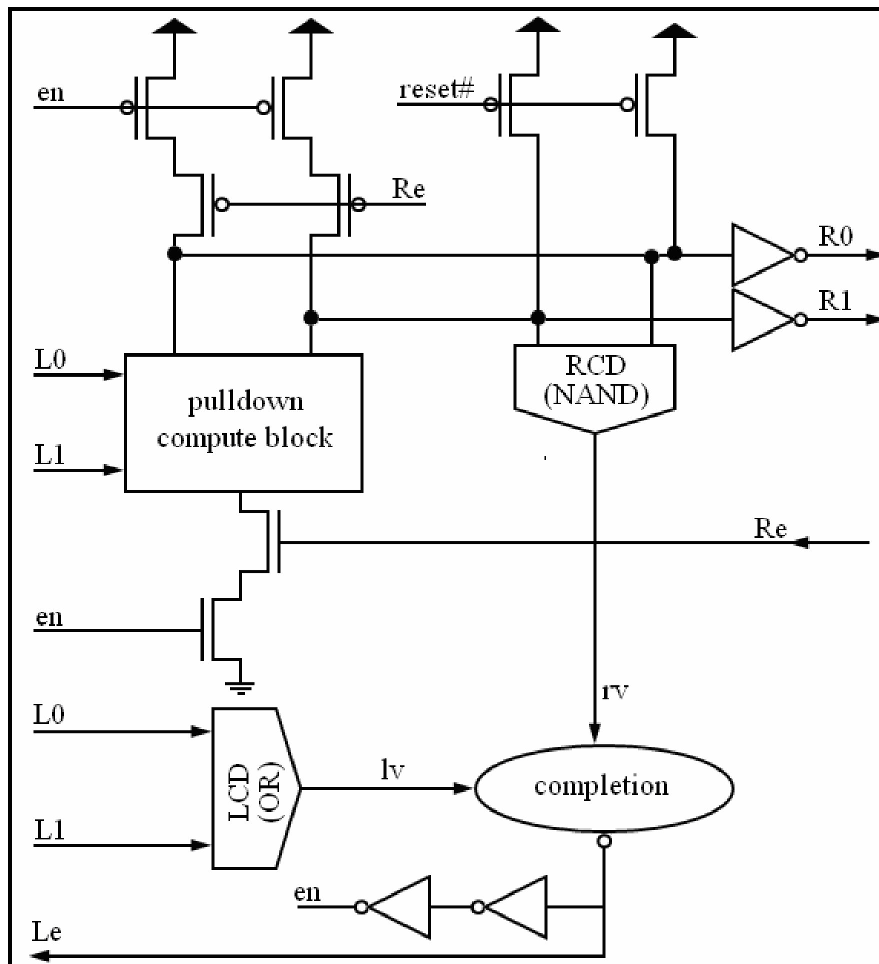
4.1 A Cell Design for PL-QDI systems

The first step of PL-QDI research is to decide which QDI cell design will be used for PL-QDI systems. A comparison of the performance of Caltech's QDI gates is helpful in selecting one of the three QDI design cells (WCHB, PCHB, PCFB).

4.1.1 Performance of Caltech's QDI Design Cells

The WCHB gate has the lowest cycle time of all the three Caltech's QDI design cells with 10 transition counts, but the use of WCHB gate increases the forward latency of the circuit. The PCFB gate has a cycle time of 12 transitions and a PCHB gate has a cycle time of 14 transitions, so obviously the PCFB gate is faster, but not by a large factor. This speed is achieved at the cost of larger gate size of PCFB gates because they require more logic for the generation of an extra state variable and extra completion

detection [5, 12]. Due to the larger size of PCFB gates, they are used only in design of circuits where speed is critical. The PCHB gates are considered the “work horses” for most applications since they have comparatively faster throughput, smaller in size and are easy to design [5, 12, 6].



Note Adopted from [7]

Figure 4.1 PL-QDI gate template

4.1.2 *PL-QDI Template Gate*

In this research, the Caltech's QDI PCHB gate is used as the PL-QDI gate due to its smaller size, substantial speed, and ease of design when compared to the WCHB and PCFB gates. The QDI gates used in the PL-QDI systems are called *PL-QDI* gates. The PCHB gate's compute blocks are restricted to having a maximum of two inputs in this work, but can be increased with additional logic at a speed cost due to the extra complexity of the pulldown network and input completion detection logic. The PL-QDI gate template is shown in Figure 4.1.

4.2 Token Abstraction for Quasi-Delay Insensitive Gates

This section gives a brief review of the four-phase handshaking protocol in QDI gates and explains how it is translated into a token abstraction.

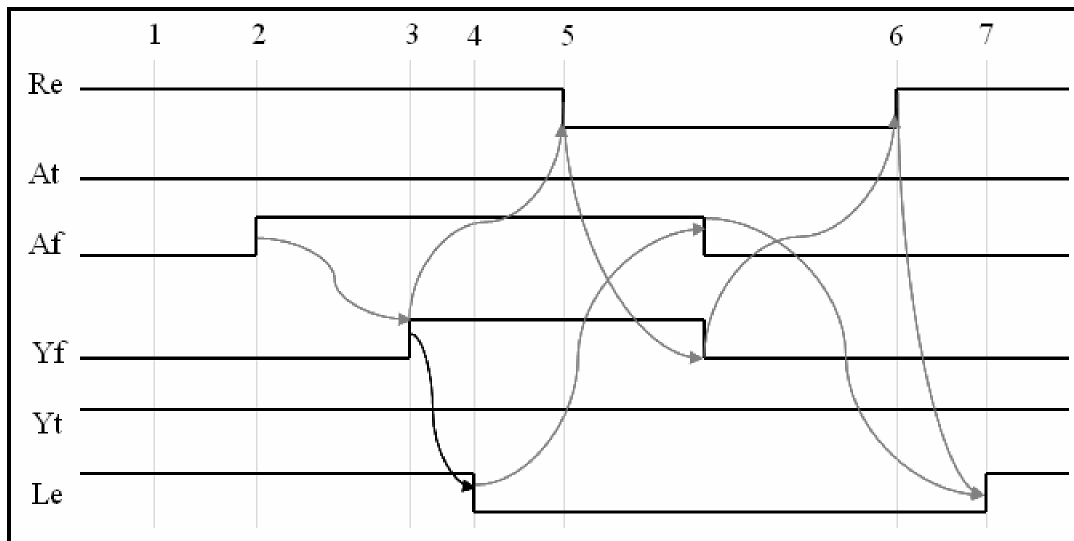


Figure 4.2 Four-phase handshaking in PL-QDI gate

Figure 4.2 illustrates the four-phase handshaking for a single input QDI gate. In Figure 4.2, L_e and R_e represents active low input feedback and output feedback signals, and the signal pairs (A_f, A_t) and (Y_f, Y_t) represents dual-rail input and output respectively. At time 1, the QDI gate is in its initial state where the gate is waiting for the arrival of new dual-rail input. Dual-rail input (A_f, A_t) arrives at time 2. Output computation takes place between time 2 and 3, and the dual-rail output (Y_f, Y_t) is ready at time 3. Active low input feedback signal L_e is asserted at time 4 indicating that the input is consumed and the output computation is done. Output feedback signal R_e goes low at time 5 indicating the output consumption by the destination gate. At time 6 R_e is deasserted to request for new output. L_e is deasserted at time 7 indicating that the gate has been precharged and ready for new input. This cycle repeats continuously to facilitate data communication among the QDI gates.

Let us now translate the four-phase handshaking protocol of QDI gates into a token abstraction. A QDI gate with token abstraction is referred to as a PL-QDI gate. To describe the token abstraction, a PL-QDI gate is represented as a marked graph as shown in Figure 4.3. In Figure 4.3, the *compute* node represents the compute block, *LCD* and *RCD* nodes represent input arrival and output completion detection circuits, and the *C* node represents the block used to generate the input feedback L_e and compute block enable *en*. The signals in Figure 4.3 are compute node input *in1*, LCD node input *in1_lcd*, output feedback R_e, input valid signal L_v, output valid signal R_v, compute block output ready *s3*, and PL-QDI gate outputs *s1* and *s2* going to a destination gate's *compute* node and *LCD* node.

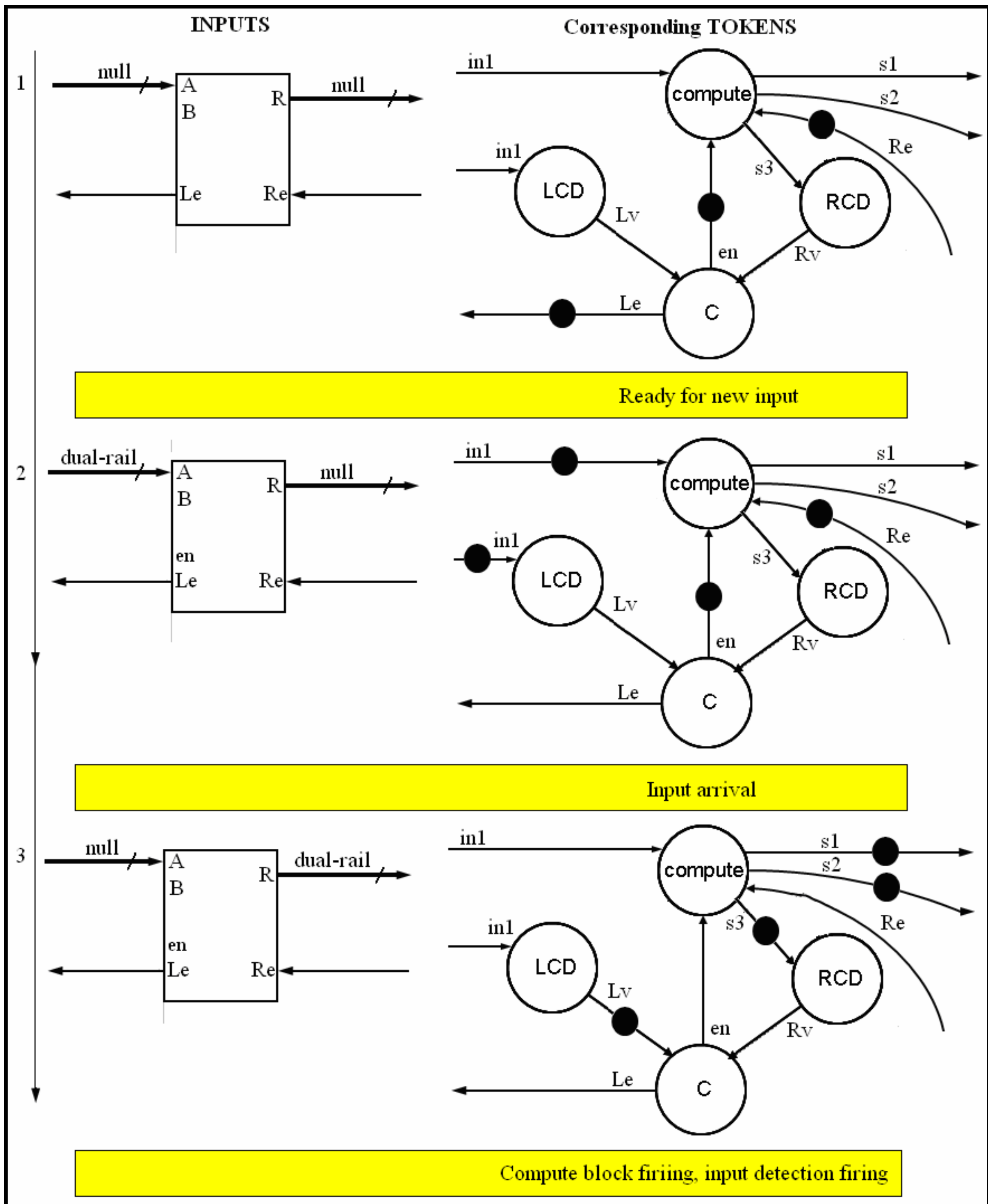


Figure 4.3 Token flow in PL-QDI gate, steps 1-3

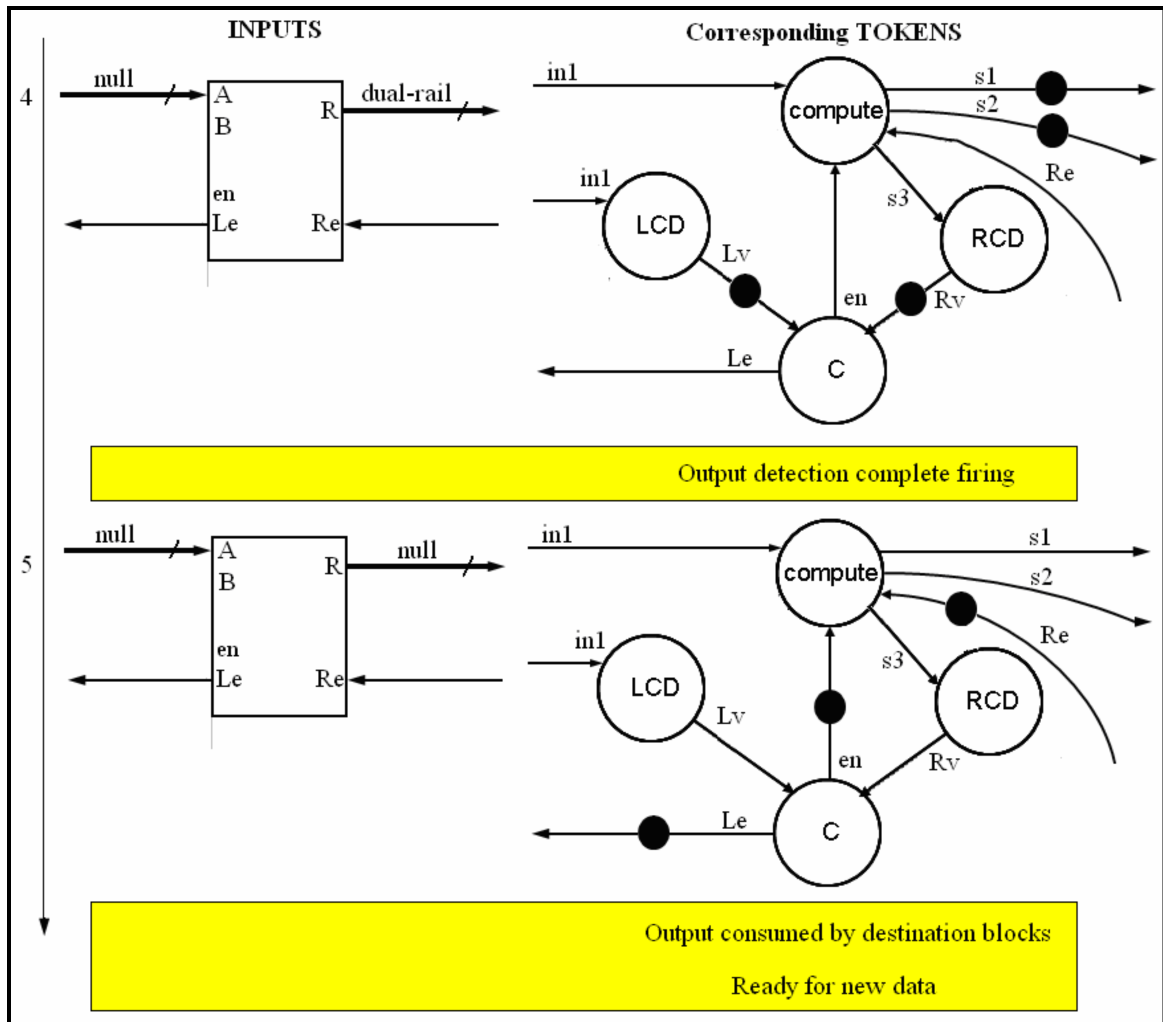


Figure 4.4 Token flow in PL-QDI gate, steps 4-5

Figure 4.3 and Figure 4.4 describes the token flow in the PL-QDI gate. At time 1, the PL-QDI gate is waiting for the arrival of dual-rail input. In this condition, there is no valid dual-rail data on the input, and L_e and R_e are deasserted. In terms of token abstraction, this is represented by no token on the input and tokens on L_e and R_e indicating that the gate will fire once it receives tokens on all of its inputs. At time 2, dual-rail inputs arrive. This places a token on the input *in1*, *in1_lcd* and consumes the

token on L_e because in order for the preceding gate to fire, it must consume the L_e token. Now the compute node and the *LCD* node have tokens on all their inputs and are ready to fire. At time 3, the compute and *LCD* nodes fire. Firing of the compute node consumes the token on its inputs (*in1*, *en*, R_e) and places the token on signal *s3* and PL-QDI gate outputs *s1* and *s2*. Firing of the input completion detection node *LCD* consumes the token on its input *in1_lcd* and places a token on L_v. The output completion detection node *RCD* fires at time 4 by consuming the token on *s3* and placing a token on *Rv*. At time 5, a token arrives on R_e indicating the output consumption by the destination blocks. Node *C* also fires by consuming the tokens on L_v and R_v, and placing new tokens on signals *en* and L_e. A token on the input feedback signal L_e indicates that the PL-QDI gate is ready for new data.

4.3 Mapping of a Clocked Netlist to a PL-QDI Netlist

The fine grain mapping [8, 19] methodology is used to convert a clocked netlist to an asynchronous PL-QDI netlist. In fine grain mapping, a one to one mapping of the gates in the clocked netlist to PL-QDI gates is done followed by feedback insertion to make the circuit live and safe.

PL-QDI circuits must satisfy the liveness property, safety property and initial token marking rules of a PL system and must also follow the four-phase handshaking protocol of QDI gates. The PL-QDI PCHB gates must be divided into through gates and barrier gates for the purpose of initial token marking [18]. Combinational gates such as AND, OR, NAND gates are mapped as through gates, and sequential gates such as DFFs, are mapped as barrier gates. As discussed in Chapter III, PL initial token marking places

a token on all barrier gate outputs. The initial token marking also requires that through gate to barrier gate feedback signals should not have an initial token to satisfy the safety property. But at the same time, the initial condition of the QDI gates at time 1 in Figure 4.3 suggests that L_e and R_e signals should contain a token at reset. A resolution for this problem is discussed in the next section.

If there is a directed circuit of PL-QDI gates, then there must be at least three PL-QDI gates to satisfy the four-way handshaking protocol. PL-QDI gates with a logical buffer function must be inserted into any directed circuit that does not fulfill this requirement. At least one of the gates in the directed circuit must be a barrier gate. There cannot be any directed circuit of only through gates in the netlist, as this would imply a combinational loop in the original clocked netlist, which is not allowed.

PL system rules also does not allow direct barrier gate to barrier gate connection. Splitter gates are inserted to break any barrier gate to barrier gate paths.

4.4 PL-QDI Gate Interaction

At this point the PL-QDI gate identified in section 4.1, the PL-QDI token abstraction from section 4.2 and the initial token marking from section 4.3 are used to examine PL-QDI gate interaction within a netlist. Consider an example PL-QDI directed circuit containing a barrier gate and two through gates as shown in Figure 4.5.

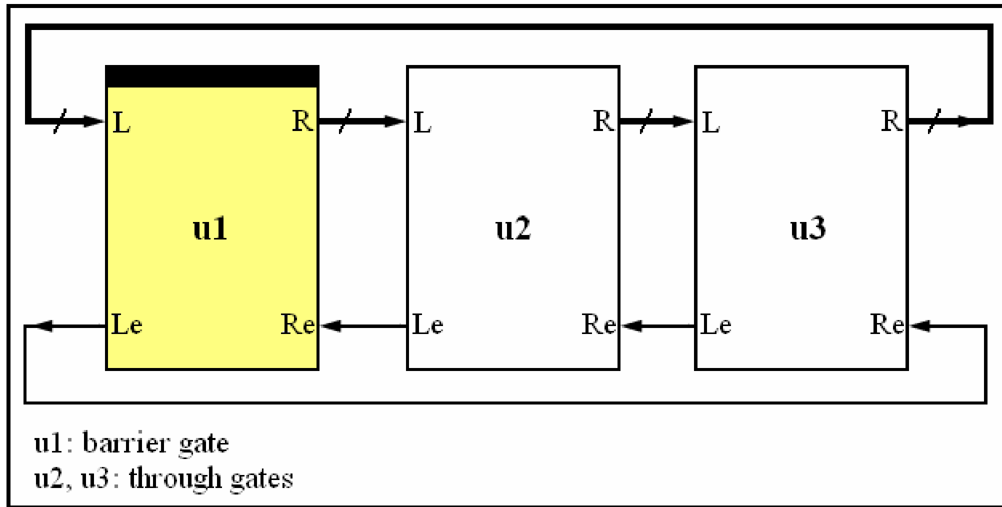


Figure 4.5 An example PL-QDI circuit

Figure 4.6 shows the token marking of the PL-QDI system during reset. As per the initial token requirement for QDI gates, input and output feedbacks (*Re1*, *Re2*, *Re3*), and *compute* node enables (*en1*, *en2*, *en3*) have initial tokens.

The release of reset must place tokens on the barrier gate outputs s1, s2 as shown in Figure 4.7. Placing tokens at the barrier gate outputs makes the circuit unsafe as the directed circuits (s2, Lv2, Re1) and (s1, s8, Rv2, Re1) contains two tokens, which is a violation of the safety rule. The unsafe signals are designated with a * besides them. Furthermore, the feedback *Re1* from the through gate *n2* to the barrier gate *n1* contains a token, which is a violation of the initial token marking rule.

Removal of the initial token on the output feedback signal *Re1* will fix the safety violation and satisfy the initial token marking rules stated earlier. Figure 4.8 shows the initial token marking of the circuit at the release of reset and without the token on signal Re1. This initial token marking is live, safe, and satisfies both PL and QDI system

properties. A live and safe initial token marking as in Figure 4.8 ensures that the system does not enter a deadlock condition.

The PL-QDI template gate shown in Figure 4.1 can be used as a through gate but not as barrier gate as it cannot generate the initial output token after reset. Also, it must somehow remove the initial token on its output feedback signal. So, in order to have a PL-QDI system that satisfies both PL and QDI gate properties, the PL-QDI template gate must be modified to be used as a PL-QDI barrier gate.

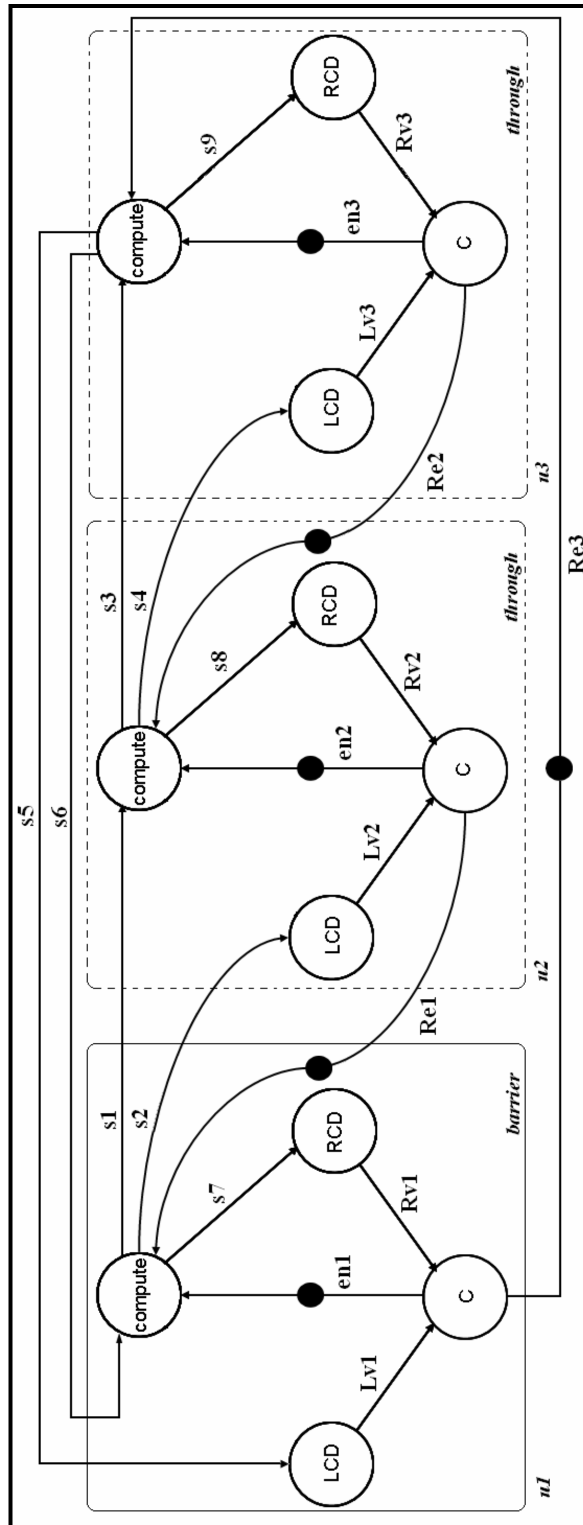


Figure 4.6 Token marking in PL-QDI system during reset

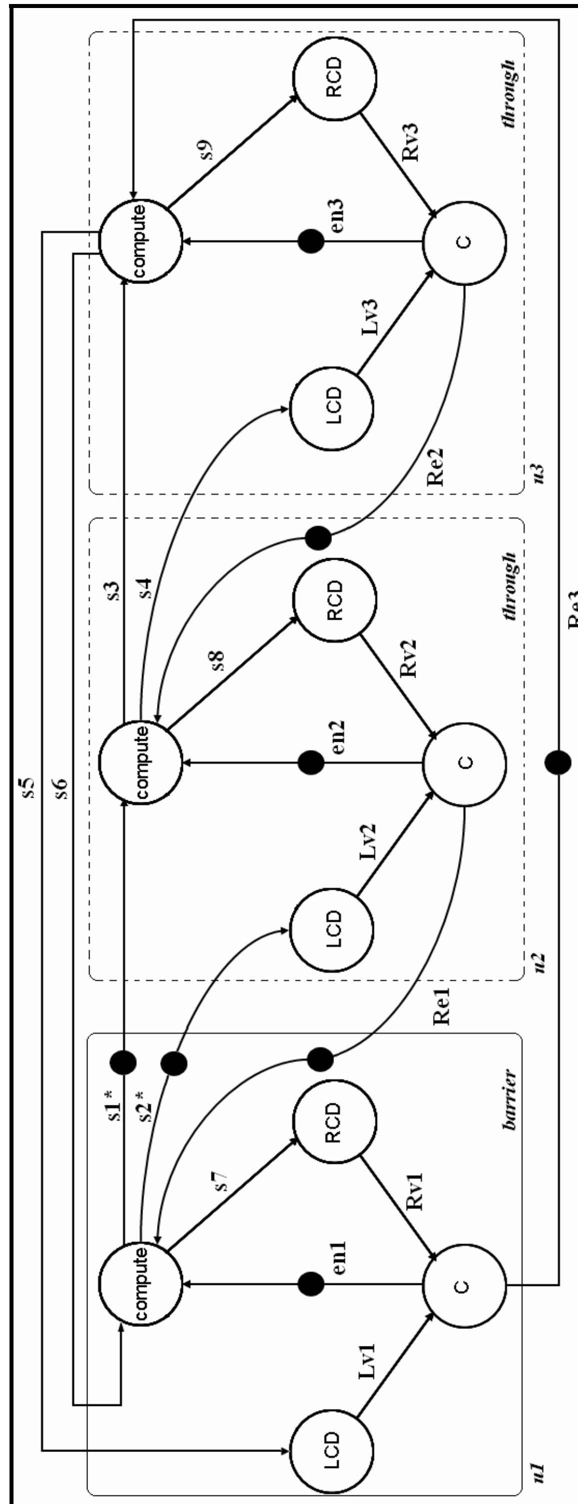


Figure 4.7 Initial token marking after the release of reset showing safety violation

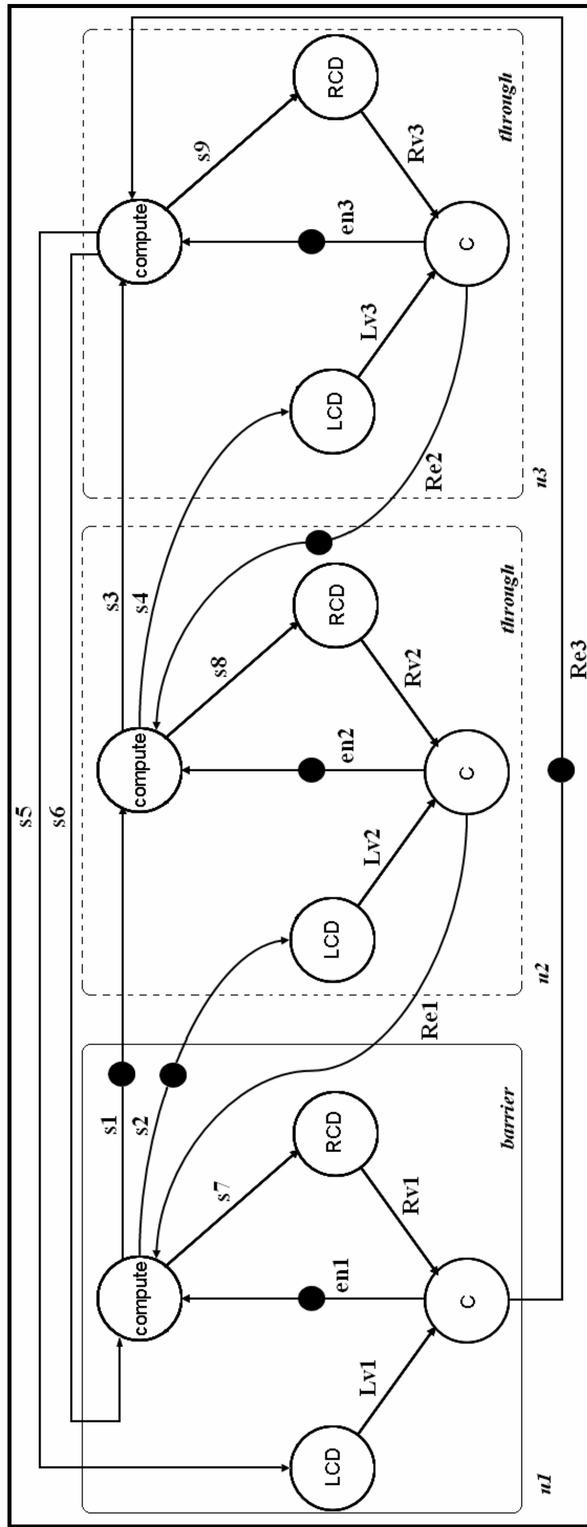


Figure 4.8 Live and safe initial token marking of PL-QDI circuit

4.5 Modifications to the PL-QDI Gate Template for Barrier Gates

The PL-QDI barrier gate must place an initial token at its outputs and must not have an initial token on its output feedback signal. Extra logic is added to the PL-QDI template gate to implement this functionality.

4.5.1 Forced Token at the Barrier Gate Output

The logic used to force a token at the output of the barrier gate after the release of reset is shown in the shaded region *#1* of Figure 4.9a. The signals *R0* and *R1* are the false and true outputs of the PCHB gate. The logic indicated in the shaded region *#1* pulls *R0* to high for a short period of time until the output feedback signal R_e, is asserted for the first time. The waveform shown in Figure 4.9b explains how a forced token is produced at the barrier gate output after the release of reset. At the release of *reset#* the SR latch output *x* goes high, pulling the NAND gate output *y* down to zero. A low *y* causes the PL-QDI output *R0* to go high while the output *R1* is still low. In terms of the token abstraction this places a token at the barrier gate output to trigger initial gate firing with in the system. When the destination gate consumes the output, it asserts the active low output feedback signal R_e. Once R_e is asserted, *x* goes low negating *y*. After this, the normal PL-QDI system operation continues until the next assertion of the *reset#* signal.

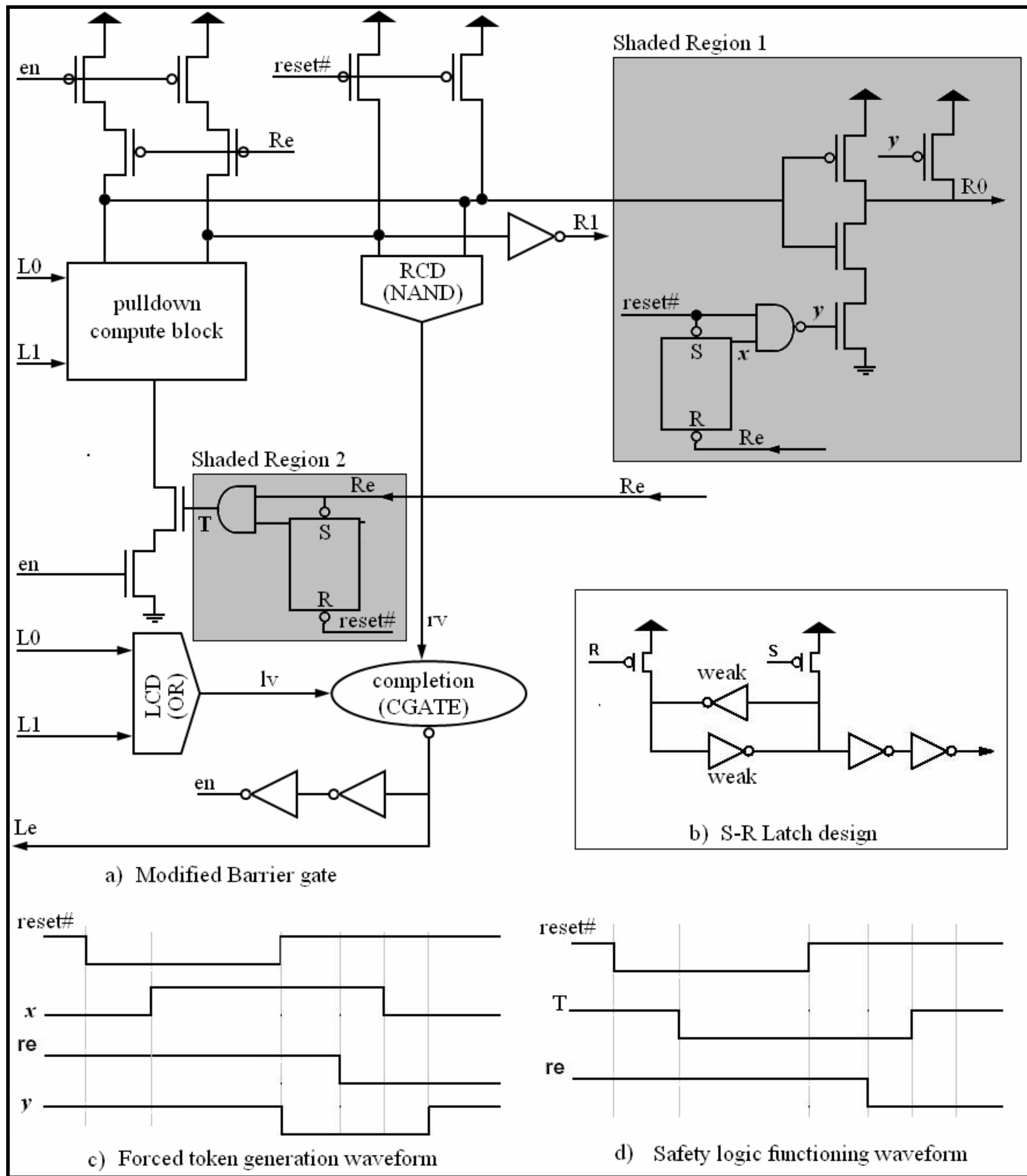


Figure 4.9 PL-QDI barrier gate and waveforms explaining its working

4.5.2 Initial Token Removal on the Barrier Gate Output Feedback

As discussed earlier, the presence of an initial token on the barrier gate output feedback signal at the release of reset causes an initial token marking rule violation. This problem can be solved by removing the initial token on the barrier gate output feedback signal R_e. The shaded region #2 in Figure 4.9 shows the logic used to remove the initial token on the feedback originating from a through gate. The waveform shown in Figure 4.9d explains the functionality of the extra logic. The AND gate is driven low during reset and until the output feedback signal R_e is asserted for the first time. The low value on the AND gate output disconnects the compute block from the pulldown tree and prevents the gate from evaluating until the first assertion of the output feedback signal. Thus, it removes the initial token on the output feedback signal until the initial tokens on the barrier gate outputs are consumed by the destination gates. When the output feedback signal is asserted, the SR latch output becomes '1', connecting the output feedback signal R_e to the pull down circuit of the compute block. This restores the normal operation of the PL-QDI gate.

The modified PL-QDI gate template shown in Figure 4.9 is used as the PL-QDI barrier gate. The initial token marking of the PL-QDI system (shown in Figure 4.5) using a PL-QDI barrier gate is the same as the initial token marking shown in Figure 4.8, resulting in a live and safe PL-QDI system.

CHAPTER V

CAD SUPPORT FOR PL-QDI CIRCUIT DESIGN

The advent of computer aided design (CAD) tools in the semiconductor industry has revolutionized the integrated circuits (IC) design process [29]. This chapter explains the CAD tool flow for designing PL-QDI circuits starting from clocked circuits. Section 1 discusses the evolution and advantages of synchronous CAD tools. Section 2 gives a brief explanation of the CAD tool flow used in a synchronous logic design. Section 3 provides an overview of the state of currently available asynchronous CAD tools. Section 4 discusses the synthesis algorithm used to map clocked system topology to asynchronous PL-QDI systems. Section 5 explains how synchronous CAD tools can be used for asynchronous PL-QDI circuit design and lists the commercial CAD tools used in each stage of the CAD flow. A brief overview of the PL-QDI gate library and the test bench approach used to simulate PL-QDI systems is also included in this section.

5.1 Introduction to Synchronous CAD Tools

The first commercial digital IC was developed by Texas Instruments in the early 1960s [30]. Since then, the IC industry has experienced a rapid growth from small scale integration (SSI) circuits with less than one hundred transistors to very large scale integration (VLSI) circuits or ultra large scale integration (ULSI) systems with millions of transistors. During the period of SSI circuits, the IC masks used in fabrication of ICs

were generated by hand drawn mask patterns [29]. As the semiconductor industry entered the age of VLSI/ULSI chips, it became increasingly difficult to design and fabricate integrated chips manually. To handle the increasing number of transistors and growing complexity of integrated chip design, IC designers started to make use of CAD tools. By the early 1980s, CAD tools were used in the functional design, physical implementation and verification of integrated circuits. Circuit designs are now described at the register transfer level (RTL) using hardware description languages (HDL) such as VHDL and Verilog, Logic synthesis from RTL to a gate level netlist and system level modeling are a few examples where CAD tools are used in digital circuit design. CAD tools are also used in the physical design of ICs and aid designers in layout generation, mask-pattern generation, and IC floor planning. In the verification of integrated circuits, CAD tools find applications in RTL, gate, and transistor level simulations, as well as in layout design rule checking. Furthermore, CAD tools have also found application in CAD tool management, which has helped CAD engineers to implement centralized maintenance and support for a corporate CAD flow. This has given the flexibility of adding new features and state- of-the-art off-the shelf CAD tools for continued product improvement. A corporate CAD tool flow refers to the general methodology used in the semiconductor industry for IC design [40]. The long-term use of CAD tools for IC design has enabled synchronous CAD tools to become reliable and efficient. The use of CAD tools means reduced design time to market, efficient use of resources, increased productivity, decreased cost due to mass production, reduced chances of human error, and has made it

easier to adopt to new technology or toolset. All these advantages in using CAD tools in IC designs have made it clear that CAD tools are essential to the modern designer.

5.2 An RTL Flow for Synchronous Circuits

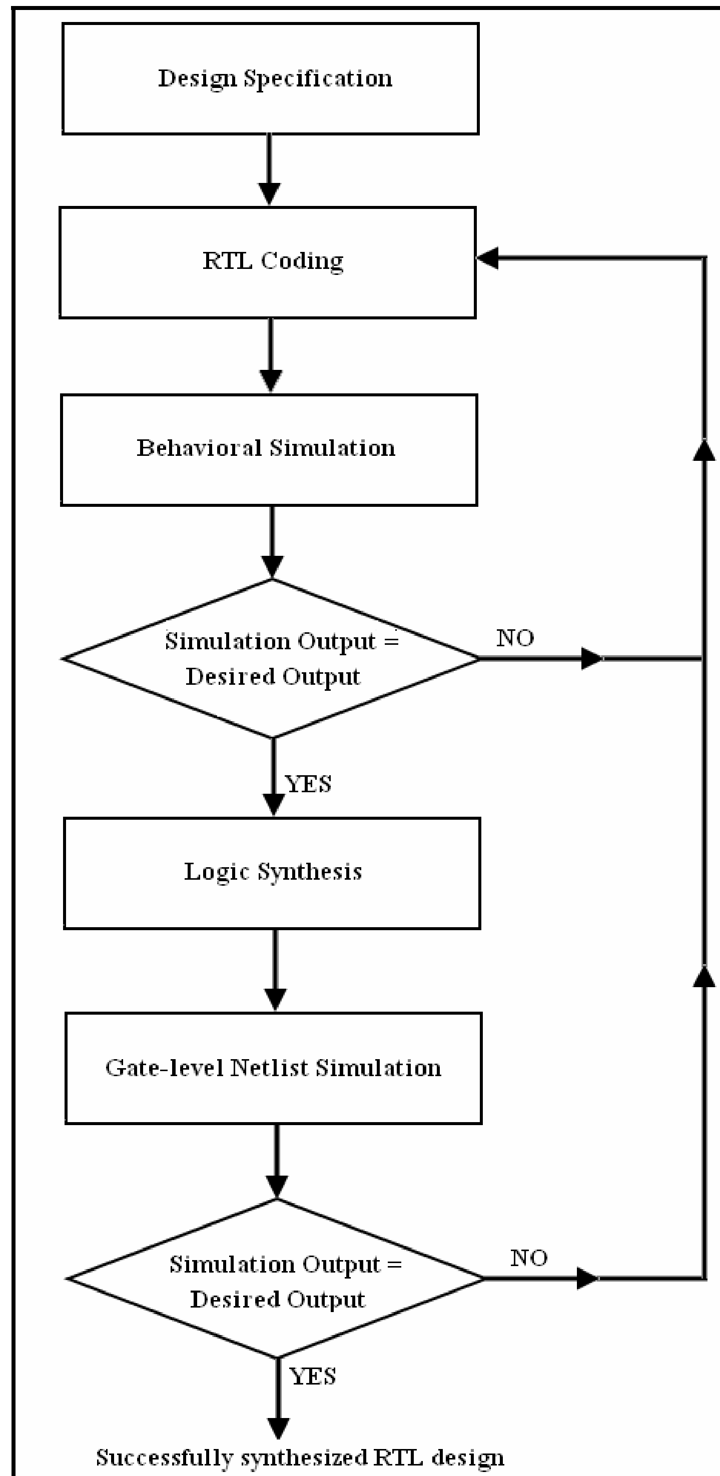
A synchronous circuit processes its inputs depending on either the clock edge (edge-triggered circuits) or the high/low level of the clock signal (level based circuits). For synchronous digital circuits, part of the CAD flow methodology is the transformation of an RTL code description of the circuit to a synthesized gate level netlist that passes functional verification. Functional design verification of synchronous circuits means that the circuit is tested for correct functionality by simulation with different input test cases. A synchronous circuit described using either RTL code or a gate netlist is functionally correct if the circuit yields the desired output during simulation of the RTL code or gate netlist.

Figure 5.1 adopted from [40] gives the RTL flow used in synchronous circuit design. A complete CAD flow for synchronous IC design is given in [40]. The first stage in the RTL flow is the design specification of the circuit under construction, which describes the functional behavior of the circuit, I/O interface, timing constraints, and the available resources. For example, assume the circuit to be designed is a rising edge triggered, synchronous 2-bit counter with count enable. "Synchronous" means that the counter changes state on the rising edge of the clock and only if the count enable is asserted. This design specification must be converted into the next level of abstraction called the RTL description of the design by using hardware description languages such as VHDL or Verilog.

The RTL code is simulated using circuit simulators to test the functionality of the design. If the functionality check fails, the RTL code has to be redefined until the design passes the functionality test. There are large numbers of circuit simulators available in the market for different operating systems. Mentor Graphics Modelsim [55] and MSIM [56] are examples of RTL simulators.

After successful completion of functional verification, the RTL code is synthesized to a gate level netlist using HDL synthesis tools. A synthesis tool maps RTL code to a gate netlist using a target gate library. The gate library contains gate definitions for all available gates that can be used in the design. Commercially available HDL synthesis tools include Synopsys Design compiler [59], Altera [57], and Xilinx [58].

The synthesized gate netlist is simulated to verify its functionality. If the gate level netlist produces the desired output, then the gate level circuit is functionally correct. If the gate netlist fails the functionality check, the RTL code must be modified, re-synthesized, and simulated until the synthesized netlist produces the correct output. Mentor Graphics Modelsim [55] is an example of a gate netlist simulator.



Note Adopted from [40]

Figure 5.1 RTL Flow

5.3 CAD Tools and Asynchronous Integrated Circuit Design

As mentioned previously, the number of transistors inside a chip and circuit complexity has increased dramatically over the last two decades. This has posed new design challenges like clock skew, increased power consumption, and increased EMI as described in Chapter I. These increasing challenges to synchronous design have encouraged researchers [27, 24, 1, 9, 41] to explore an alternative area of IC design – Asynchronous design.

Asynchronous design techniques are still in the research stage, and do not have the support of mature CAD tools that has aided the rapid growth of synchronous IC designs. Asynchronous designers make use of proprietary tools [41, 42] or custom design styles [12, 27] to build asynchronous circuits. The use of custom tools and design styles has several disadvantages when compared to the synchronous IC design process. The custom built tools demand considerable expertise in the corresponding asynchronous design methodology to produce efficient circuits. Furthermore, the majority of engineers are trained in synchronous techniques, not asynchronous techniques. As such, there are very few asynchronous design engineers when compared to the synchronous community. This deficiency of skilled manpower make an asynchronous design methodology cost ineffective for use by the IC industry. Training of manpower to design asynchronous circuits increases the production cost [41] of asynchronous systems. Asynchronous designs also suffer from the lack or limited availability [24] of CAD tools. The available asynchronous tools have very limited features relative to commercial synchronous CAD tools [41]. This increases time to market for asynchronous ICs. The production time of

asynchronous designs is further increased due to the additional time spent by asynchronous designers for developing custom tools to aid them in asynchronous circuit design. All the above drawbacks of the asynchronous methodology have caused the IC design industry to resist the use of asynchronous design techniques for IC design.

To counter these drawbacks, the PL methodology introduced by Linder and Harden [9] allows asynchronous designs to be produced from clocked networks. This research adopts the PL methodology for use with QDI circuits to produce a combined methodology termed PL-QDI.

5.4 PL-QDI Synthesis

This section formally defines a synchronous system and describes a synthesis algorithm adopted from the PL methodology for converting a synchronous system to an asynchronous PL-QDI system. The formal definitions of synchronous systems that follow are adopted from [9].

Definition 1: **Synchronous gate:** A synchronous gate G is a three-tuple (I, O, F) where,

$I = \{i_1, i_2, i_3, \ldots\}$ is a non-empty set representing input terminals of gate G,

$O = \{o_1, o_2, o_3, \ldots\}$ is a non-empty set representing output terminals of gate G

F is the logical behavior of the gate.

If gate G is a combinational gate, then the outputs is a function of current input values and are assigned as soon as there is a change in input values. If gate G is a sequential gate, it represents a finite state machine and its output is a function of the current inputs and the previous state output.

Definition 2: **Synchronous Signal:** A synchronous signal is a three-tuple (G_i, G_o, C) where G_i is the driving gate, G_o is destination gate and C represents the connection between driving gate and destination gate. C is a two-tuple (I, O); where I represent an output of G_i and O represents an input of G_o.

Definition 3: **Synchronous system:** A synchronous system is a two-tuple (G, S), where G represents synchronous gates and S represents synchronous signals.

There are a few restrictions on synchronous systems that are required for successful mapping of clocked systems to PL-QDI systems. A synchronous system should have only one clock signal driving all the sequential gates. The synchronous system cannot have directed graphs constituting of only combinational gates, i.e, each directed graph in the synchronous system should have at least one sequential gate.

5.4.1 PL-QDI Synthesis Algorithm

The synthesis algorithm takes a synchronous system topology and converts it into its asynchronous PL-QDI equivalent circuit. The resulting asynchronous PL-QDI system exhibits the same functionality as that of the original clocked system. The algorithm ensures that the PL-QDI system is live, safe, and satisfies the initial token marking rules as described in Chapter 2. The algorithm also guarantees that PL-QDI gates can communicate between each other using a four-phase handshaking protocol without any deadlock. The pseudo code of the algorithm used in the synthesis of a PL-QDI system is given in Figure 5.2. The synthesis algorithm involves three steps: one-to-one mapping of synchronous gates to PL-QDI gates, splitter gate insertion, and feedback insertion.

```
Step 1: One-to-one mapping of gates
Do a one-to-one mapping of gates in clocked netlist to PLQDI gates.
Remove clock signal convert single-rail I/O to dual-rail I/O to form
PLQDI network.

Step 2: Insert Splitter gates
// Check for direct barrier gate to barrier gate path
for (each barrier gate)
{
 for (each output)
   {
            if(output connected to any barrier gate input)
             {
               insert splitter between output and destination gate
             }
   }
}

for (each barrier gate)
{
 bg_node = save barrier gate node

 for (each output)
   {
            if(output connected to through gate input)
             {
               for (each output of destination gate)
                        {
                            if(output connected to input of bg_node)
                                   {
                                    insert  buffer  between output and
                                    destination gate
                                   }
                        }
             }
   }
}

Step 3: Feedback insertion
Connect output feedback originating from destination gate to
source gate feedback input
```

Figure 5.2 PL-QDI synthesis pseudo code

Step 1: One-to-One Mapping of Synchronous Gates to PL-QDI Gates

In the first stage of the synthesis process, a one-to-one mapping of clocked gates to PL-QDI gates is performed. Gates in the clocked system have single-rail inputs and outputs, while the PL-QDI gates have dual-rail inputs and outputs. During this stage, the global clock network is removed and all single rail signals are converted to dual-rail signals. All combinational gates in the clocked system are replaced by logically equivalent PL-QDI gates.

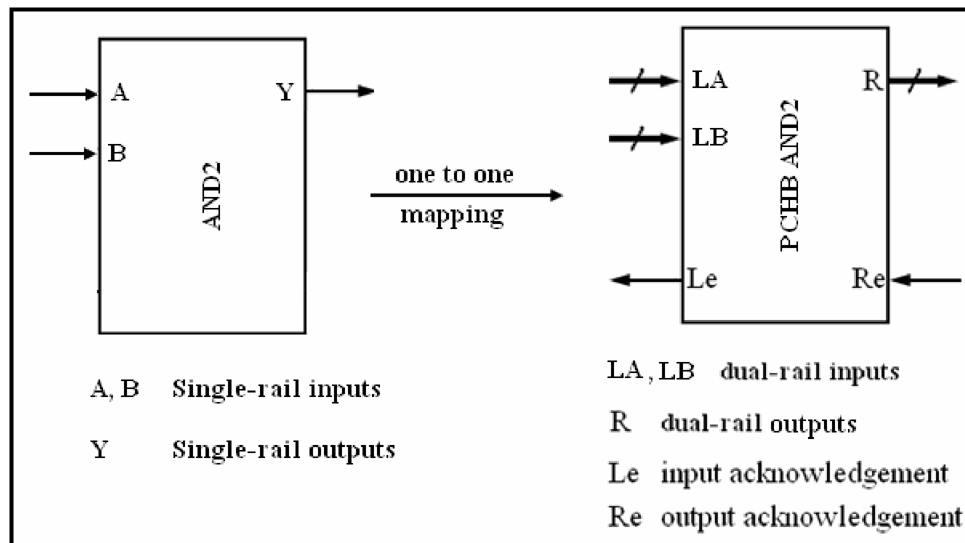


Figure 5.3 One-to-one mapping of AND2 gate to PL-QDI AND2 gate

Figure 5.3 shows the one-to-one mapping of a 2-input *AND2* gate to a PL-QDI *AND2* gate. Sequential gates such as a DFF are replaced by barrier gates in the PL-QDI netlist. Barrier gates do not perform any logical operation but only transport the input to its output. All barrier gate outputs have an initial token on them at the release of reset. The resulting gate topology at the end of step 1 is termed as a “PL-QDI network”. Figure

5.4 shows an example clocked system and its equivalent PL-QDI network at the end of step 1 of the synthesis algorithm.

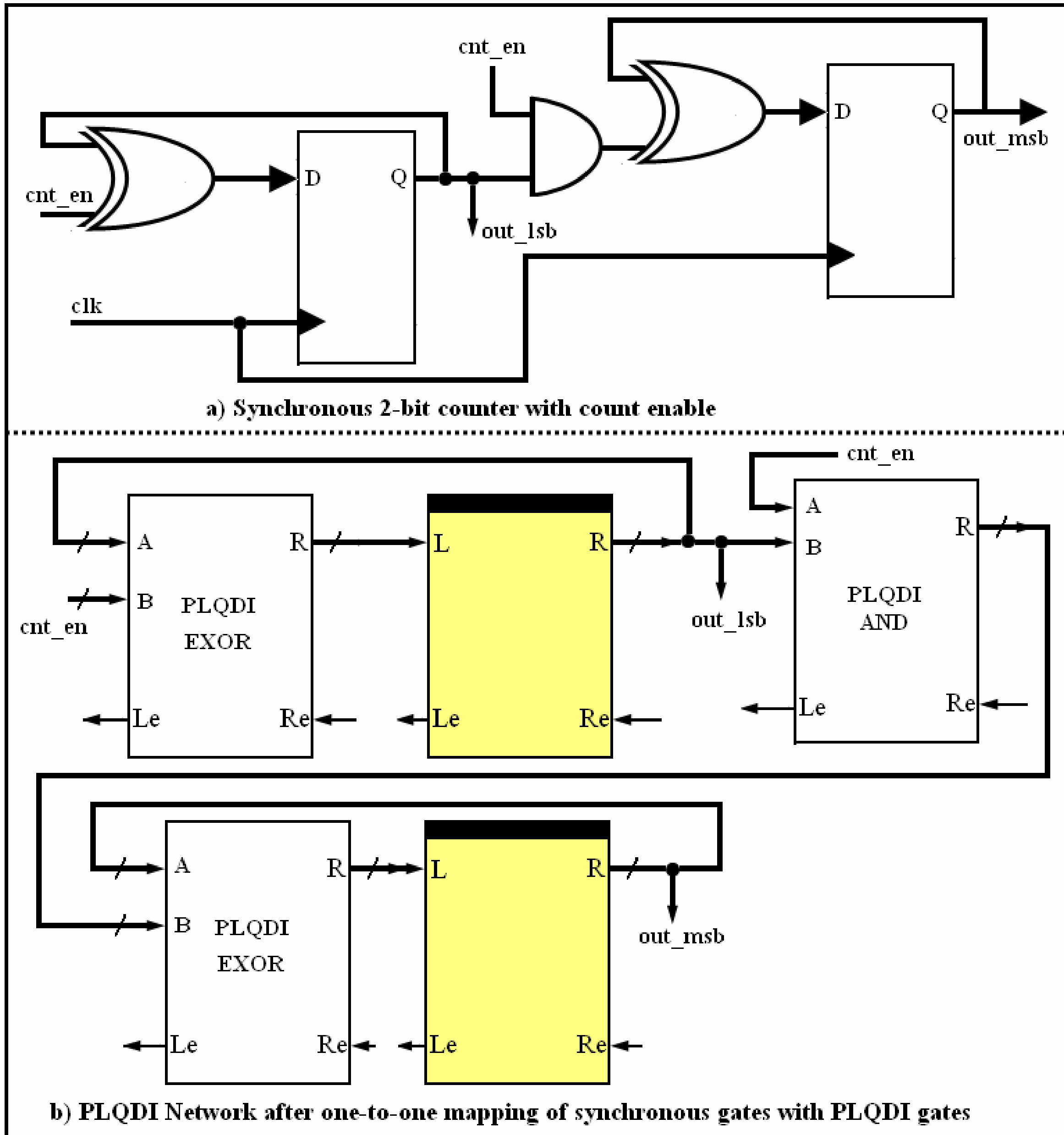
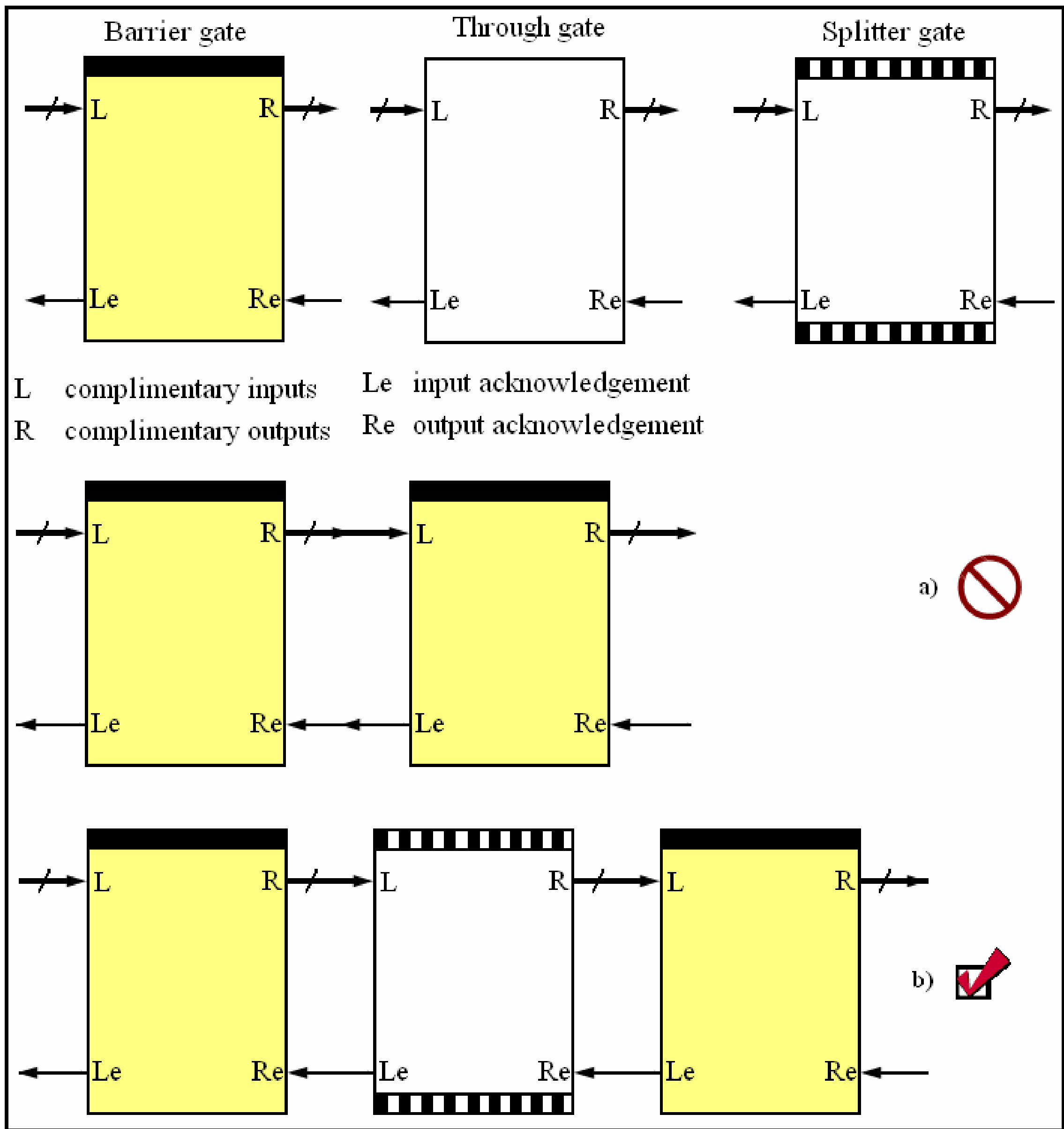


Figure 5.4 Clocked circuit and equivalent PL-QDI network

Step 2: Splitter Gate Insertion

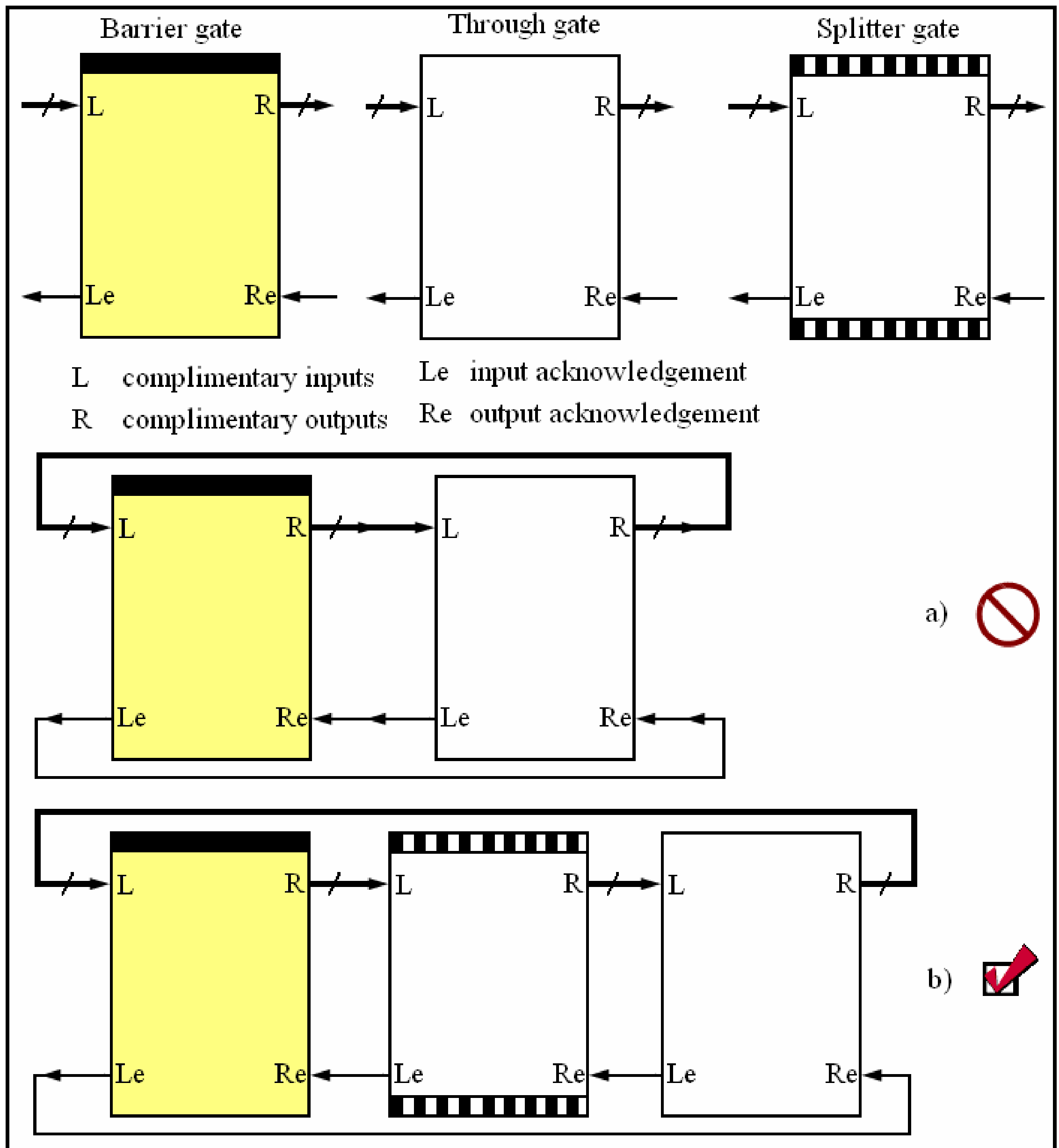
As described in Chapter 3, to ensure a live and safe PL-QDI circuit there should not be any direct barrier gate to barrier gate path. If there is any barrier gate to barrier gate connection, a splitter gate is inserted between them to break this path. Figure 5.5 shows splitter gate insertion to break a barrier gate to barrier gate path. Splitter gates are through gates that act as buffers when inserted in PL-QDI circuits.

Furthermore, to guarantee correct four-phase communication between PL-QDI gates, the PL-QDI network topology should not have directed graphs with less than three PL-QDI gates. If there are less than three PL-QDI gates in a directed graph, it will lead to a deadlock. Deadlock is a condition where PL-QDI gates stops communicating with each other. This can be prevented by adding a through (buffer logic function) gate in the loop that has less than three PL-QDI gates as shown in Figure 5.6. These loops can be detected in a network by doing a depth-first search until depth 2 of the network is reached. Consider a barrier gate *B0* as the root of a tree, and through gates *T1* and *T2* are its child nodes. Assume that through gate *T1* is connected to the input of B0. If we do a depth-first search with *B0* as the root it will first lead to *T1*, then by continuing the depth first tree at level 1 node *T1*, it will point to the root *B0* of the tree. This detects the existence of a loop with only two PL-QDI gates. A through gate has to be inserted between *B0* and *B1* to ensure a live PL-QDI system.



Note a) Forbidden continuous barrier gate to barrier gate path
b) Direct barrier gate to barrier gate path broken by splitter gate insertion

Figure 5.5 Splitter gate insertion to break continuous barrier gate to barrier gate path



Note a) Forbidden directed path with less than 3 PL-QDI gates
b) Through gate (buffer function) inserted to allow communication using 4-phase handshaking protocol

Figure 5.6 Buffer function insertion

Step 3: Feedback Concentrator Insertion

The QDI handshaking protocols require a feedback from each of a source gate's destinations. At the end of steps 1 and 2 there may still be some unsafe signals in the PL-QDI systems. There can be cases where a gate (e.g. G_1) drives multiple gates (G_2, G_3); in this case gate G_1 must receive feedback signals from gates G_2, G_3. The algorithm adds a feedback concentrator to receive feedbacks from all the destination gates as shown in Figure 5.7a. A Muller C-element is used as a feedback concentrator if a gate receives multiple feedbacks. If a gate (e.g. G_1) is driven by multiple gates (G_2 G_3), then the gate (G_1) should send feedback to its source gates (G_2, G_3) as shown in Figure 5.7b. Trees of C-gates are used to concentrate feedback signals if more than a 4-input C-gate is required. Figure 5.8 shows the PL-QDI circuit equivalent of the clocked 2-bit counter with count enable, first transformed in Figure 5.4.

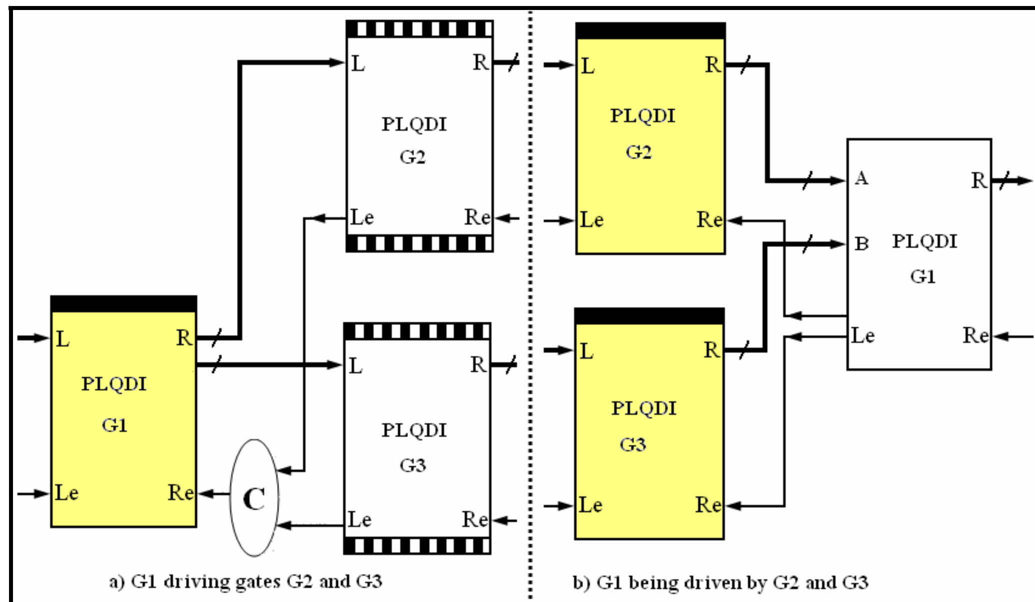


Figure 5.7 Feedback in a PL-QDI circuit

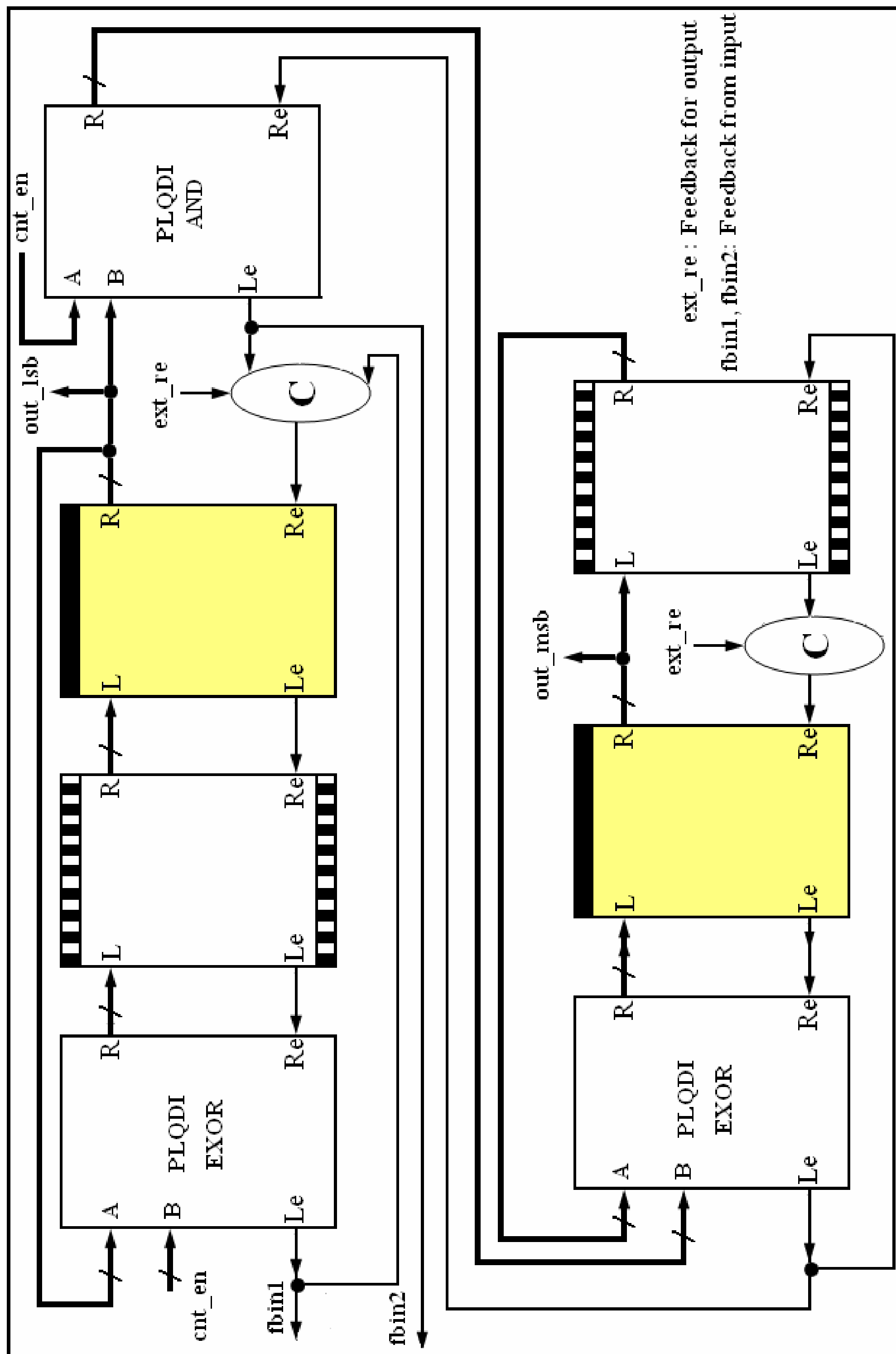


Figure 5.8 PL-QDI 2-bit counter with count enable

5.5 CAD Tool Flow of PL-QDI Methodology

This research implements the PL-QDI synthesis algorithm by using commercially available CAD tools. The PL-QDI CAD flow closely follows the synchronous RTL CAD flow explained earlier in this chapter. A PL-QDI mapping tool written in the *C* language is added to the synchronous RTL CAD flow to automatically convert a clocked netlist to an asynchronous PL-QDI netlist. The mapping tool outputs the PL-QDI netlist and a testbench for simulating the design. Figure 5.9 shows the complete PL-QDI CAD flow used to map clocked RTL to a PL-QDI gate netlist. The PL-QDI mapping tool is indicated in the shaded area. The steps of the synthesis algorithm described earlier are also shown in Figure 5.9.

The input to the PL-QDI CAD flow is an RTL description of a synchronous circuit written using VHDL or Verilog. For example, consider the RTL description of a 2-bit counter with count enable as shown in Figure 5.10 used as an input to the PL-QDI CAD flow. The clocked design is first simulated using a simulator to verify the circuit functionality.

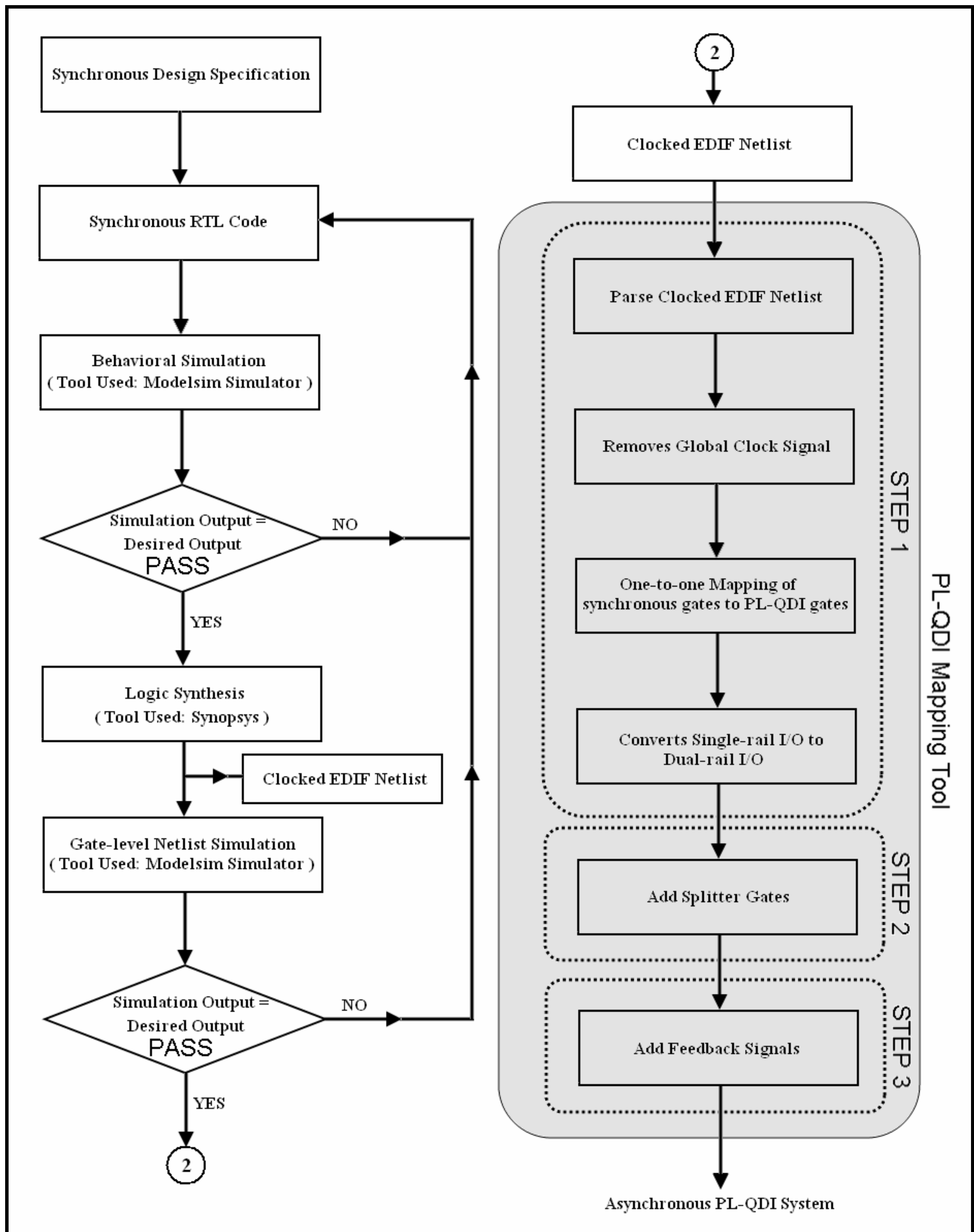


Figure 5.9 PL-QDI CAD flow

```
-- 2 Bit Counter
entity cnt2_cnten is
  port ( clk,reset:  in std_logic;
         cnt_en:  in std_logic;
         dout  : out std_logic_vector(1 downto 0)
  );
end cnt2_cnten;

architecture a of cnt2_cnten is

  signal  cnt_state: std_logic_vector(1 downto 0);

begin

  dout <= cnt_state;

  stateff:process(clk)
    begin
    if (reset = '0') then
      -- asynchronous reset
      cnt_state <= "00";
    elsif (clk'event and clk='1') then
     if (cnt_en = '1') then
      cnt_state <= cnt_state + 1;
     end if;
    end if;
   end process stateff;

end;
```

Figure 5.10 RTL Code of 2-bit counter with count enable

After the successful verification of the RTL design, logic synthesis is performed using a two-input static CMOS gate library to produce a gate-level netlist. The logic synthesis is done by using Synopsys Design Complier. The gate level netlist is then

simulated to check the functionality of the synthesized design. The clocked gate-level netlist of the 2-bit counter with count enable in VHDL format is shown in Figure 5.11.

```
entity cnt2_cnten is
  port( clk, reset, cnt_en : in std_logic;  dout : out std_logic_vector (1
      downto 0));
end cnt2_cnten;

architecture gate of cnt2_cnten is
  component and2
    port( A1, B1 : in std_logic;  O : out std_logic);
  end component;

  component xor2
    port( A1, B1 : in std_logic;  O : out std_logic);
  end component;

  component dfr
    port( DATA1, CLK2, RST3 : in std_logic;  Q, QN : out std_logic);
  end component;

  signal dout0_1_port, dout0_0_port, n113, n114, n115, n116, n117 : std_logic;
begin
  dout <= ( dout0_1_port, dout0_0_port );
  U31 : and2 port map( A1 => dout0_0_port, B1 => cnt_en, O => n113);
  U32 : xor2 port map( A1 => dout0_1_port, B1 => n113, O => n114);
  U33 : xor2 port map( A1 => dout0_0_port, B1 => cnt_en, O => n115);
  cnt_state_regx1x : dfr port map( DATA1 => n114, CLK2 => clk, RST3 => reset,
                    Q => dout0_1_port, QN => n116);
  cnt_state_regx0x : dfr port map( DATA1 => n115, CLK2 => clk, RST3 => reset,
                    Q => dout0_0_port, QN => n117);

end gate;
```

Figure 5.11 Synthesized gate netlist of 2-bit counter with count enable

The clocked netlist in EDIF format is used as the input for the PL-QDI mapping tool. The mapping tool reads the clocked netlist and does a fine grain mapping to a PL-

QDI netlist as previously described. The PL-QDI system generated by the mapping tool is a VHDL netlist of PL-QDI gates and C-gates. The VHDL netlist of the PL-QDI 2-bit counter with count enable is shown in Figure 5.12. The signal declarations and temporary signal instantiation are excluded from Figure 5.12 for length reasons. The mapping tool also outputs a template VHDL testbench for simulating the PL-QDI system. The PL-QDI testbench is explained in detail in Chapter 6.

The PL-QDI system is simulated using a library of PL-QDI gates defined in Verilog. The Mentor Graphics Modelsim simulator that was used for simulation of clocked RTL and gate netlist is also used for PL-QDI circuit simulation. Thus, the entire conversion of clocked RTL design to asynchronous PL-QDI is done using commercially available synchronous CAD tools and a custom PL-QDI mapping tool.

ENTITY

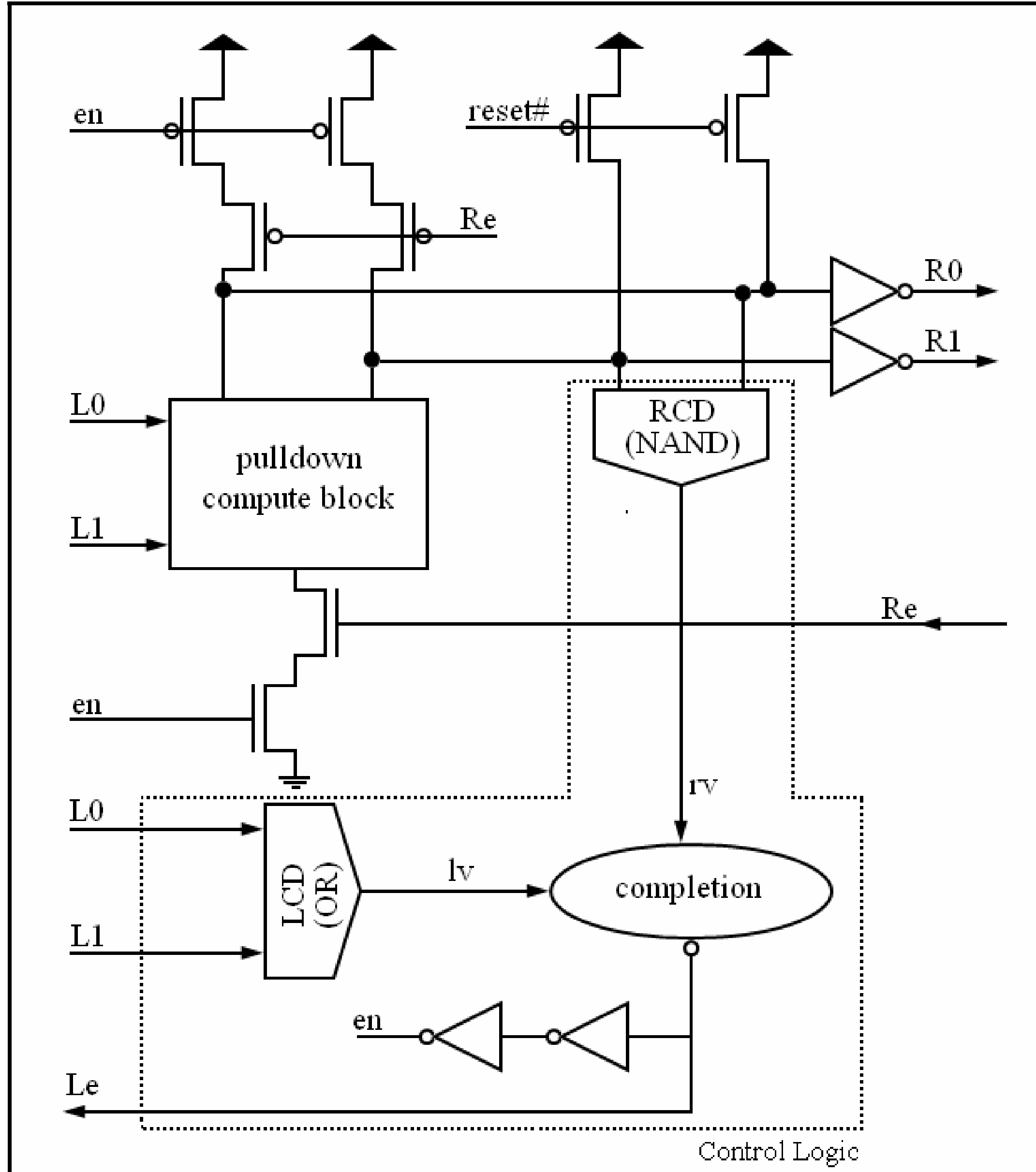
```
entity cnt2cnten is
 port (
   fbin0 : out std_logic;  -- feedback from input
   fbin1 : out std_logic;  -- feedback from input
   pchb_reset : in std_logic;
   ext_re : in std_logic;  -- feedback for output
   cnt_en : in std_logic;
   cnt_en_r0 : in std_logic;
   dout_1x : out std_logic;
   dout_1x_r0 : out std_logic;
   dout_0x : out std_logic;
   dout_0x_r0 : out std_logic
);
 end cnt2cnten;
```

ARCHITECTURE

```
architecture  a  of cnt2cnten is
n_u32_xbuff_n : pchb_buf -- u32_xbuff
  port map (
   reset => pchb_reset,
   re => n_cnt_state_regx1x_n_Le,
   a1 => n_u32_xnet_n,  -- u32_xnet
   a0 => n_u32_xnet_n_r0,  -- u32_xnet
   r1 => n_n114_n,  -- n114
   r0 => n_n114_n_r0,  -- n114
   le => n_u32_xbuff_n_Le );

 n_u33_xbuff_n : pchb_buf -- u33_xbuff
  port map (
   reset => pchb_reset,
   re => n_cnt_state_regx0x_n_Le,
   a1 => n_u33_xnet_n,  -- u33_xnet
   a0 => n_u33_xnet_n_r0,  -- u33_xnet
   r1 => n_n115_n,  -- n115
   r0 => n_n115_n_r0,  -- n115
   le => n_u33_xbuff_n_Le );

C0 : cgate_ext  -- CGATE for feedback concentration
  port map (
   reset => pchb_reset,
   a0 => ext_re,
   a1 => n_u33_n_Le,
   a2 => n_u31_n_Le,
   a3 => n_u31_n_Le,
    y => cgate_fb0);

n_cnt_state_regx0x_n : pchb_dfr -- cnt_state_regx0x
 port map (
  reset => pchb_reset,
  re => cgate_fb0,
  a1 => n_n115_n,  -- n115
  a0 => n_n115_n_r0,  -- n115
  r0 => n_0_NCnet_n,  -- 0_NCnet
  r1 => n_dout0_0_xn,  -- dout0_0_
  le => n_cnt_state_regx0x_n_Le );

C1 : cgate_ext  -- CGATE for feedback concentration
 port map (
  reset => pchb_reset,
  a0 => ext_re,
  a1 => n_u32_n_Le,
  a2 => n_u32_n_Le,
  a3 => n_u32_n_Le,
   y => cgate_fb1);

n_cnt_state_regx1x_n : pchb_dfr -- cnt_state_regx1x
 port map (
  reset => pchb_reset,
  re => cgate_fb1,
  a1 => n_n114_n,  -- n114
  a0 => n_n114_n_r0,  -- n114
  r0 => n_1_NCnet_n,  -- 1_NCnet
  r1 => n_dout0_1_xn,  -- dout0_1_
  le => n_cnt_state_regx1x_n_Le );

n_u33_n : pchb_xor2 -- u33
 port map (
  reset => pchb_reset,
  re => n_u33_xbuff_n_Le,
  a1 => n_cnt_en0_n,  -- cnt_en0
  a0 => n_cnt_en0_n_r0,  -- cnt_en0
  b1 => n_dout0_0_xn,  -- dout0_0_
  b0 => n_0_NCnet_n,  -- 0_NCnet
  r1 => n_u33_xnet_n,  -- u33_xnet
  r0 => n_u33_xnet_n_r0,  -- u33_xnet
  le => n_u33_n_Le );

n_u32_n : pchb_xor2 -- u32
 port map (
  reset => pchb_reset,
  re => n_u32_xbuff_n_Le,
  a1 => n_n113_n,  -- n113
  a0 => n_n113_n_r0,  -- n113
  b1 => n_dout0_1_xn,  -- dout0_1_
  b0 => n_1_NCnet_n,  -- 1_NCnet
  r1 => n_u32_xnet_n,  -- u32_xnet
  r0 => n_u32_xnet_n_r0,  -- u32_xnet
  le => n_u32_n_Le );

n_u31_n : pchb_and2 -- u31
 port map (
  reset => pchb_reset,
  re => n_u32_n_Le,
  a1 => n_cnt_en0_n,  -- cnt_en0
  a0 => n_cnt_en0_n_r0,  -- cnt_en0
  b1 => n_dout0_0_xn,  -- dout0_0_
  b0 => n_0_NCnet_n,  -- 0_NCnet
  r1 => n_n113_n,  -- n113
  r0 => n_n113_n_r0,  -- n113
  le => n_u31_n_Le );

end a;
```

Figure 5.12 PL-QDI system produced from the PL-QDI CAD flow

5.6 PL-QDI Gate Library



Note Adopted from [7]

Figure 5.13 PL-QDI through gate

The PL-QDI gate library consists of 2-input gates with dual rail inputs and outputs. Figure 5.13 gives a through gate block diagram. The through gate's pull down tree is used to realize basic logic functions such as AND, NAND, XOR, XNOR, OR and NOR. The circuit used to generate control signals required for the communication of data between PL-QDI gates is shown by dotted region in Figure 5.13. The detailed explanation of through gate operation is given in Chapter 4.

The PL-QDI gate library has a second gate type for barrier gates as previously explained for replacing sequential gates. A PL-QDI barrier gate is shown in Figure 5.14. The PL-QDI barrier gate operation is explained in detail in Chapter 4. There are two types of barrier gates. The first type is a barrier gate whose initial output after reset represents dual-rail logic high ("10") and the other one is a barrier gate whose initial output after reset is a dual-rail logic low ("01"). Figure 5.14 shows a barrier gate whose initial output after reset is logic low.

Designers often assign some signal values to constant high or low values in clocked circuit design (ex: *output_ready* = 1). To implement this functionality in PL-QDI designs, a modified through gate is used to generate dual-rail constant values. PL-QDI constant generators should alternate between evaluate phase and precharge phase depending on the output feedback signal received from the destination gate. This is necessary for other PL-QDI gates to continue communicating with each other using the four-phase handshaking protocol. Figure 5.15 shows a modified PL-QDI gate used for generating the logical high equivalent in dual-rail encoding ("10"). If output feedback R_e of a destination PL-QDI gate is high, the constant generator output is a valid dual-rail logic high ("10")

value. When R_e goes low, the constant generator output is a *null code* or *spacer code* ("00"). Similarly, a PL-QDI logic low constant generator is shown in Figure 5.16 and is also implemented using a through gate.

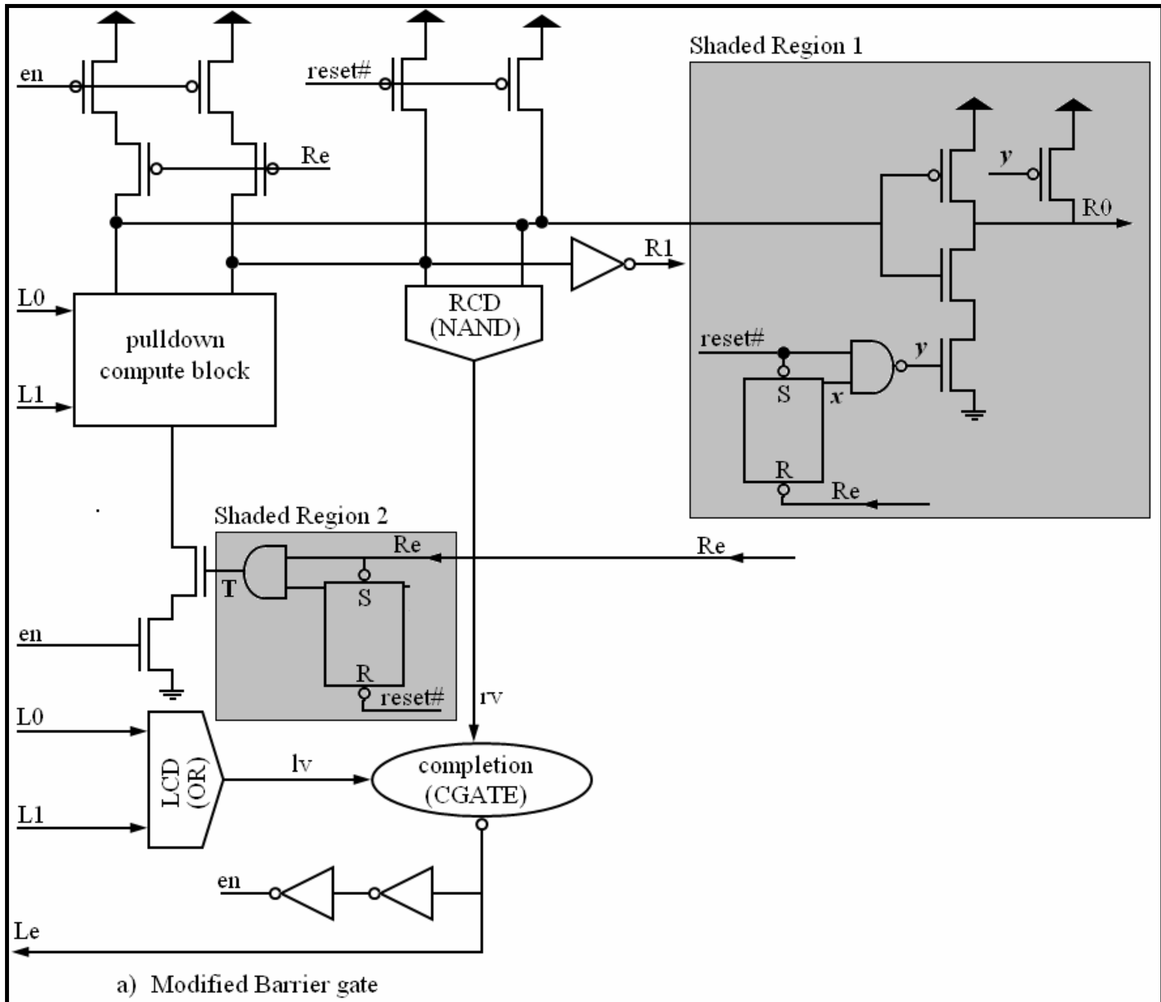


Figure 5.14 PL-QDI barrier gate

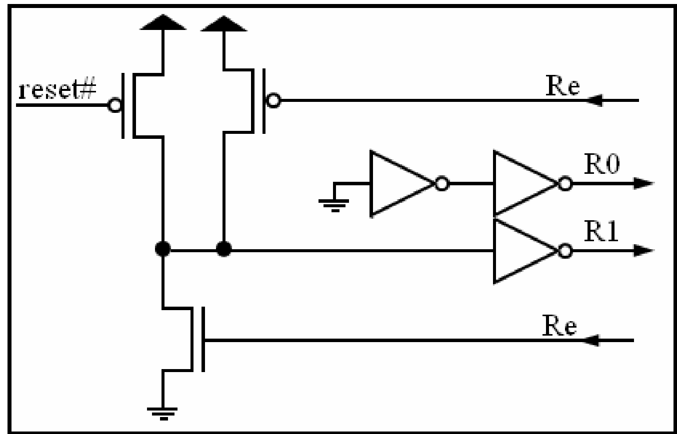


Figure 5.15 PL-QDI logic high constant generator

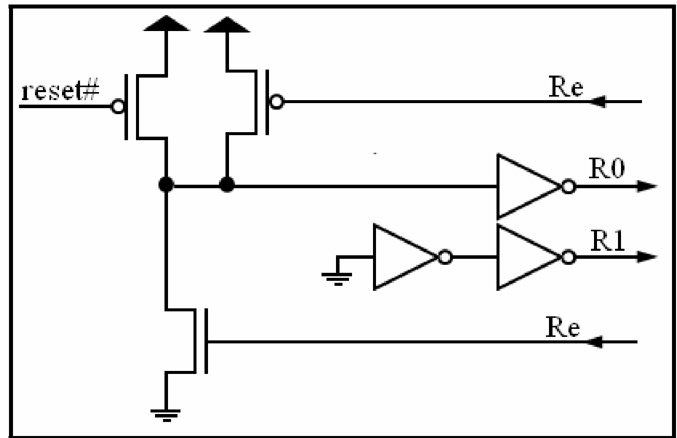


Figure 5.16 PL-QDI logic low constant generator

5.7 Summary

This chapter has described the synthesis algorithm used to design PL-QDI systems starting from a clocked netlist. It has also introduced a PL-QDI CAD flow that uses commercially available synchronous CAD tools for the design of PL-QDI systems. The use of commercial synchronous CAD tools in the design of PL-QDI systems reduces time to market and production cost. Designing a PL-QDI system starting from a clocked

netlist and using synchronous CAD tools makes it possible for anyone with design knowledge of clocked circuits to create asynchronous circuits.

CHAPTER VI

PL-QDI DESIGN EXAMPLES

The PL-QDI CAD flow was introduced in Chapter 5. This chapter describes circuits that were generated using the PL-QDI methodology. Section 1 explains PL-QDI circuit features. Section 2 describes the template testbench used to simulate a PL-QDI system. Section 3 gives PL-QDI design examples varying from a simple counter to a complex 64-bit floating point unit.

6.1 PL-QDI System Features

6.1.1 PL-QDI Systems Maintain The Synchronous Property

The word synchronous is associated with logic designs that have one or more clock signals. Clock signals are used to determine when data values are stable in clocked circuits. The clock period is the time taken for one complete clock cycle. Digital designers adjust the clock period depending on the largest delay path in the clocked system. Clocked systems are made up of two types of gates, namely, sequential and combinational gates. Combinational gates are placed between sequential gates. Sequential gates in the clocked system are evaluated once per clock cycle. This ensures that the sequential gate evaluating for the nth time uses the $(n\text{-}1)^{th}$ output of itself or other

sequential gates that has passed through the combinational gates. The level or edge of clock signal is used to determine whether the data values are stable or undergoing intermediate transitions. This concept is used to synchronize logic computations in clocked systems. The number of clock cycles used for an output computation is multiplied by the clock period to determine the output computation time.

Before discussing the PL-QDI system's synchronous property, it is important to briefly describe how asynchronous PL systems maintain the synchronous property [9]. In PL systems, the cycle number of a gate gives the number of times the gate has fired since the release of reset. A fully functional PL system is live, safe and satisfies initial token marking rules and feedback insertion rules as discussed previously. This means that the nth firing of a barrier gate uses the $(n\text{-}1)^{th}$ evaluation output of itself or other barrier gates. This is analogous to a clocked system's characteristic that the sequential gate evaluating for the nth time uses the $(n\text{-}1)^{th}$ output of itself or other sequential gates that have passed through the combinational gates. Thus, the PL system maintains the synchronous paradigm.

Because PL-QDI systems are formed by extending the concepts of PL systems to QDI gates, a fully functional PL-QDI system is live safe and satisfies initial token marking rules and feedback insertion rules. As such, a PL-QDI system also maintains the synchronous paradigm.

6.1.2 PL-QDI Gates Exhibit Fast Forward Latency

PL-QDI gates generate dual-rail outputs by using a pulldown compute block, so PL-QDI gates can fire early, that is produce an output before all inputs have arrived. The

parallel NFET tree in the PL-QDI gate compute block can compute the output at the arrival of an early input, but the input feedback signal is generated only after the arrival of all the input values. Thus, PL-QDI gates have fast forward latency and slow backward latency. This is the same functionality as produced by early evaluation PL gates in a static CMOS PL system, without the need for additional logic. An example pull-down block of PL-QDI compute block is shown in Figure 6.4.

In Figure 6.4, if input *A* is assumed to be early arrival input and input *B* is the late input, the PL-QDI gate computes the output at the arrival of *A*, but it only sends the input feedback after the arrival of input *B*.

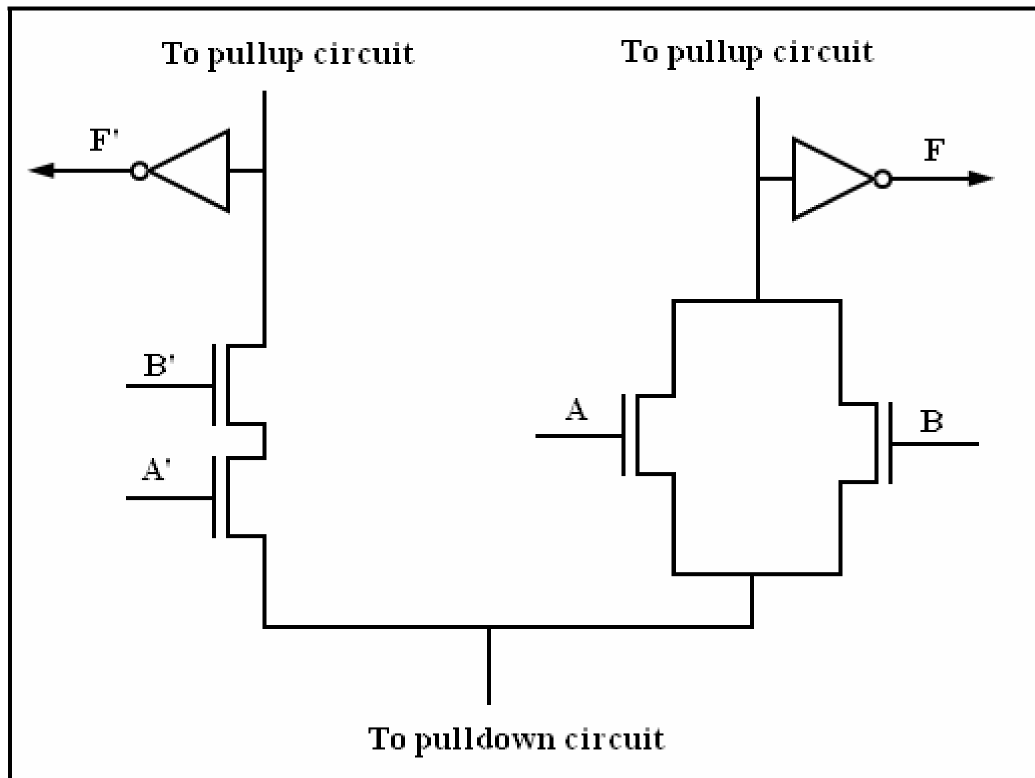


Figure 6.1 PL-QDI gate compute block

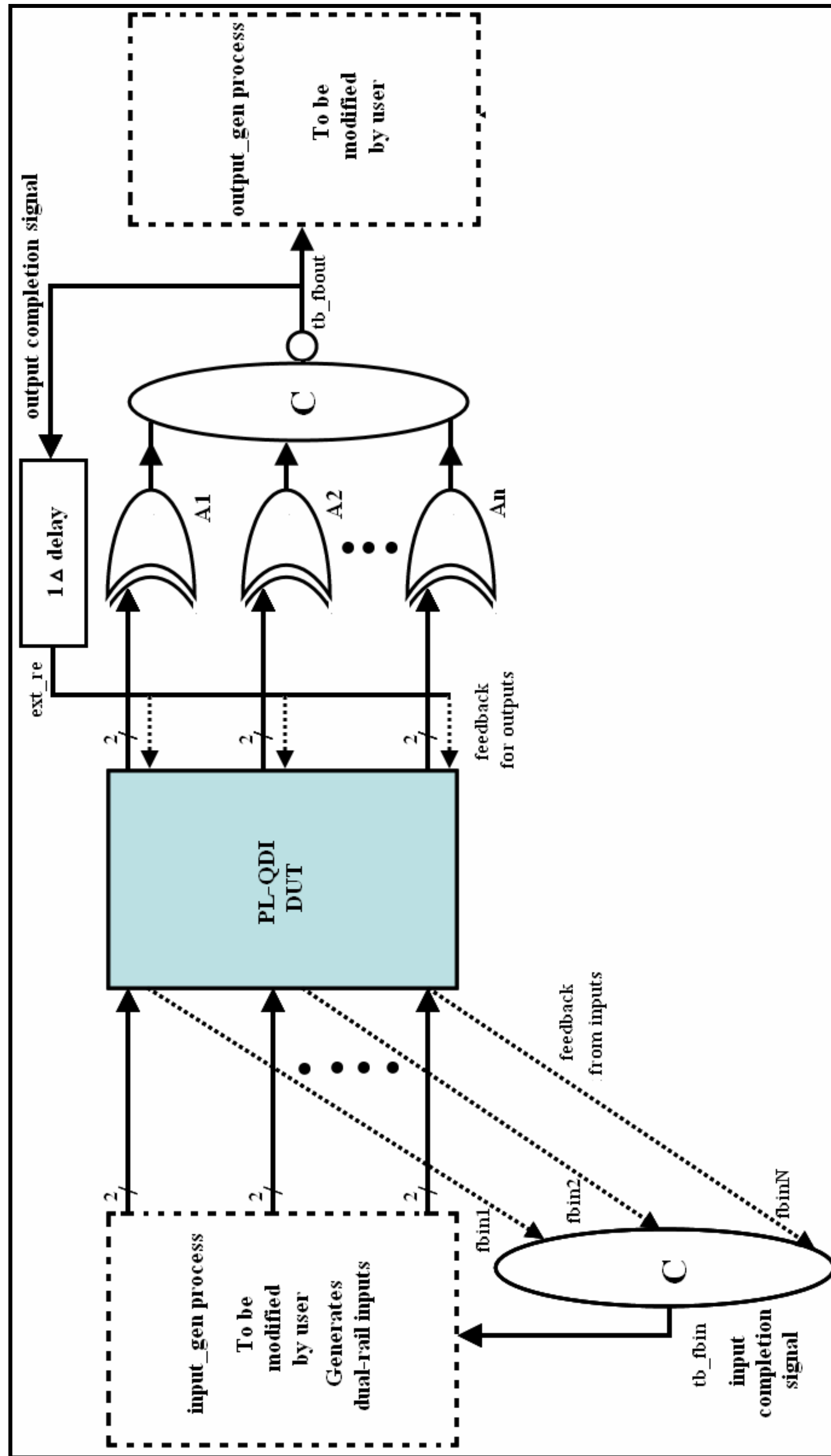
6.2 PL-QDI Testbench

Figure 6.2 gives a block diagram representation of the VHDL testbench generated by the PL-QDI mapping tool. The testbench instantiates the device under test (DUT), provides dual-rail input test vectors to the DUT and traces dual-rail output values of the PL-QDI system. The testbench receives feedbacks *fbin1, fbin2, …fbinN* (where N is the number of inputs) from the DUT inputs and generates input feedback *tb_fbin* by using the C-gate *cg_fbin.* The testbench also detects individual output completions by XORing the dual-rail outputs and computes an overall output completion signal *tb_fbout*. Signal *ext_re*, a delayed version of *tb_fbout* is used as feedback for the DUT outputs. VHDL processes *input_gen* and *ouput_gen* are used to generate dual-rail input values and trace output values.

The *Input_gen* process is triggered if there is an event on the *tb_fbin* signal. A low *tb_fbin* value indicates that the input values have been consumed and that the input PL-QDI gates are ready to precharge. At this point the *input_gen* process drives the dual-rail inputs with the null code to cause the input PL-QDI gates to precharge. If *tb_fbin* is high, this indicates that the input PL-QDI gates have been precharged and are ready to accept dual-rail inputs. This also means that *input_gen* process should provide *valid* dual-rail inputs but not necessarily *new* dual-rail inputs. In many digital systems handshaking output signals are used to differentiate valid outputs from temporary output values. They also indicate when the circuit is ready for new input values. Similarly, in PL-QDI circuits, handshaking output signals are used to indicate when the PL-QDI system is

ready to accept new dual-rail inputs. In cases where the PL-QDI DUT requires new dual-rail inputs each time *tb_fbin* is driven high, then new dual-rail inputs are provided by the *input_gen* process. This case is analogous to a clocked design that accepts new inputs each clock cycle.

The *Output_gen* process is used to trace PL-QDI DUT outputs. This process is instantiated by an event on *tb_fbout*. XOR gates A_1, A_2, ….. A_n are used to detect individual output completion signals. A high on the output on any of these XOR gates indicate a valid output for that corresponding dual-rail output. All of the individual output completion signals are fed to a C-gate *cg_fbout* to generate an overall output completion signal *tb_ fbout*. A low *tb_ fbout* indicates valid dual-rail outputs on all of the outputs and indicates that the output PL-QDI gates can be precharged after output consumption. Signal *ext_re* is a delayed version of *tb_fbout* and is used to indicate output consumption. Signal *ext_re* is used as a feedback input to the DUT outputs. A low *ext_re* enables precharge of the output PL-QDI gates. A high *tb_fbout* means the output PL-QDI gates have been precharged and are waiting for signal *ext_re* to request new outputs. Thus, the *tb_fbin*, *tb_fbout* signals are used to sequence the input and output PL-QDI gates between the evaluation and precharge states.



Note Adopted from [60]

Figure 6.2 Block diagram of PL-QDI testbench

The marked graph representation for the testbench generated by the PL-QDI mapping tool is shown in Figure 6.3. It shows the interface between the testbench and the DUT as a marked graph. Figure 6.3 shows that the initial token marking for the marked graph is live and safe, ensuring correct operation.

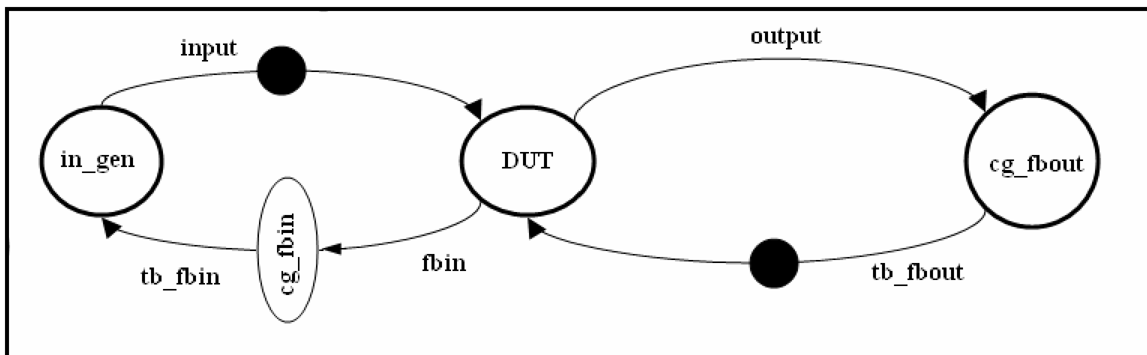


Figure 6.3 Marked graph representation of PL-QDI testbench

6.3 Design Examples

This section describes example designs developed using the PL-QDI CAD flow. Clocked RTL designs were used as the input to the PL-QDI CAD flow. These PL-QDI circuits produced from the PL-QDI CAD flow were simulated to verify that their functionality matched their clocked counter parts.

6.3.1 Counter Designs

For initial testing, four different clocked counters were used to exercise the PL-QDI methodology as shown below:

- 2-bit counter
- 2-bit counter with count enable

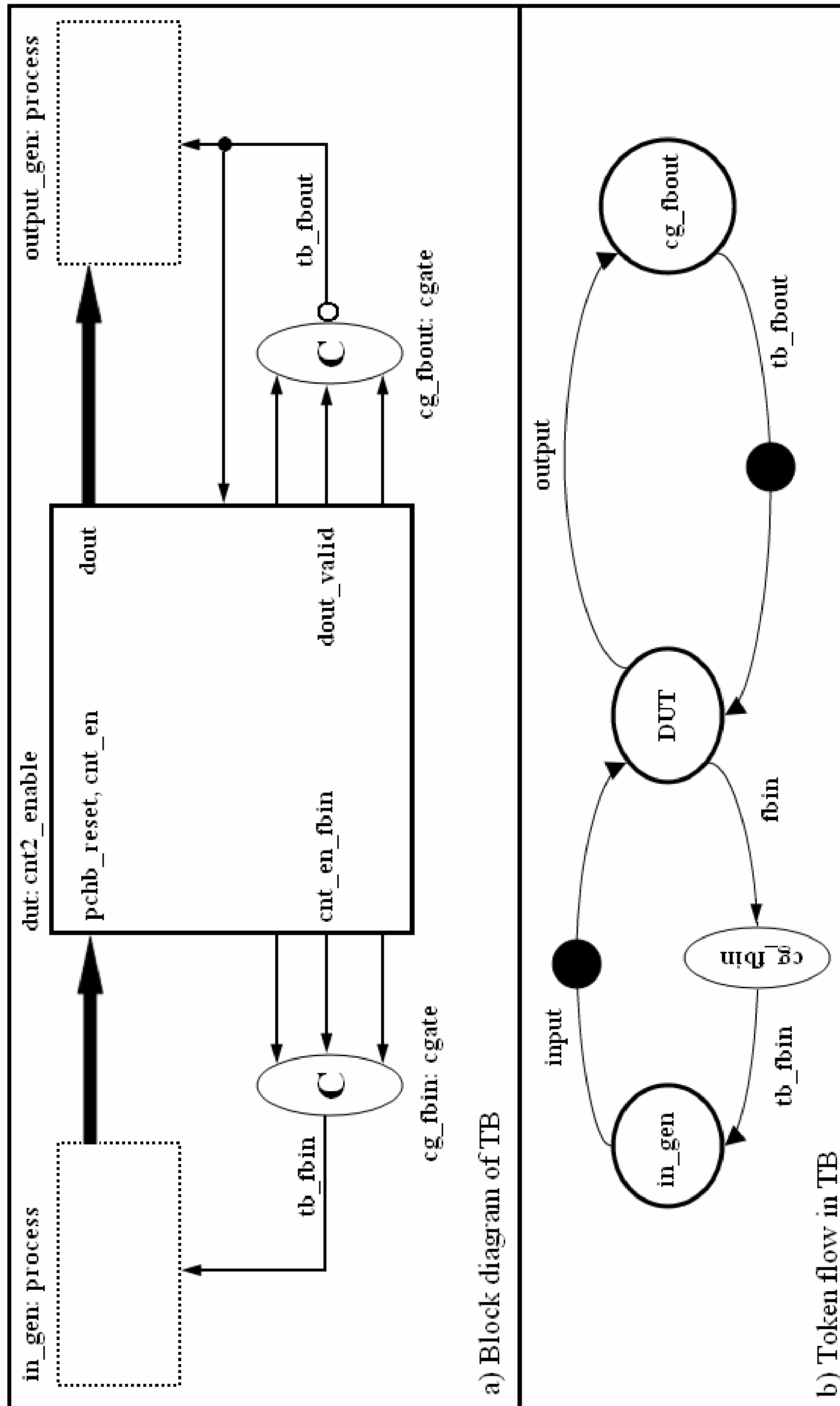
- 4-bit counter
- 4-bit counter with count enable

The RTL for the counters were synthesized to clocked gate level netlists in EDIF format. The PL-QDI mapper tool was used to map the clocked netlists to PL-QDI gate netlists. Each PL-QDI netlist was simulated using PL-QDI testbench. The block diagram representation and the marked graph representation of the testbench used for simulation of the 2-bit PL-QDI counter with count enable are shown in Figure 6.4a and Figure 6.4b. Similar testbenches were used for simulating the other PL-QDI counters.

Table 6.1 compares the number of gates in the synthesized clock design and the PL-QDI gate netlists. The increase in the number of gates in the PL-QDI gate netlists are due to the insertion of splitter gates and buffer gates.

Table 6.2 Comparison of number of gates in clocked and PL-QDI designs

Designs	**Clocked**	**PL-QDI**
	Total gates	**Total gates**
2-bit counter	3	6
2-bit counter with enable	5	7
4-bit counter	9	14
4-bit counter with enable	11	15



Note (Adopted from [60]) (a) Block diagram (b) Marked graph representation

Figure 6.4 PL-QDI counter testbench

6.3.2 64-bit Floating Point Clipper Circuit

The 64-bit floating point clipper designs used in this example are adopted from [60]. A clipper circuit passes an input stream to the output constraining the output to lie between user-specified lower and upper bound values. A high level abstraction of the logic used in the non-pipelined clipper circuit is shown in Figure 6.5. The VHDL RTL design implementation has a four-state finite state machine (FSM) and a datapath. The datapath and control used for the clipper circuit is shown in Figure 6.6. Two states were used to load the upper and lower bound values into the clipper circuit, and the remaining two states were used to compute clipped output values.

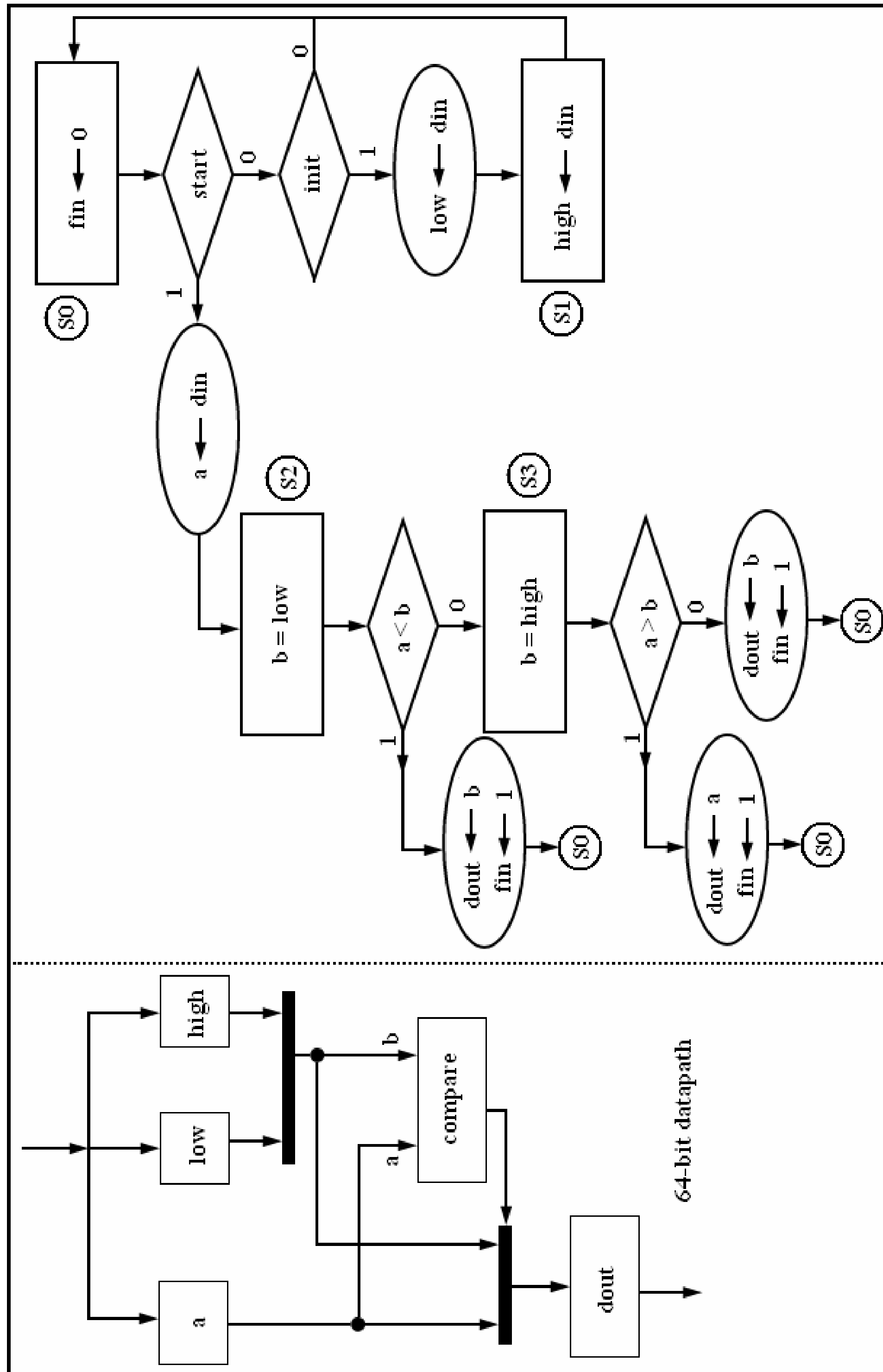
```
if(input < lowerbound) then
{
   output = lowerbound
}
else if (input > higherbound) then
{
   output = highbound
}
else
{
   output = input
}
```

Note Adopted from [60]

Figure 6.5 Clipper circuit high level abstraction

The 64-bit clocked netlist was converted to a 64-bit PL-QDI netlist using the PL-QDI CAD flow. The PL-QDI design was simulated using 1000 randomly generated input

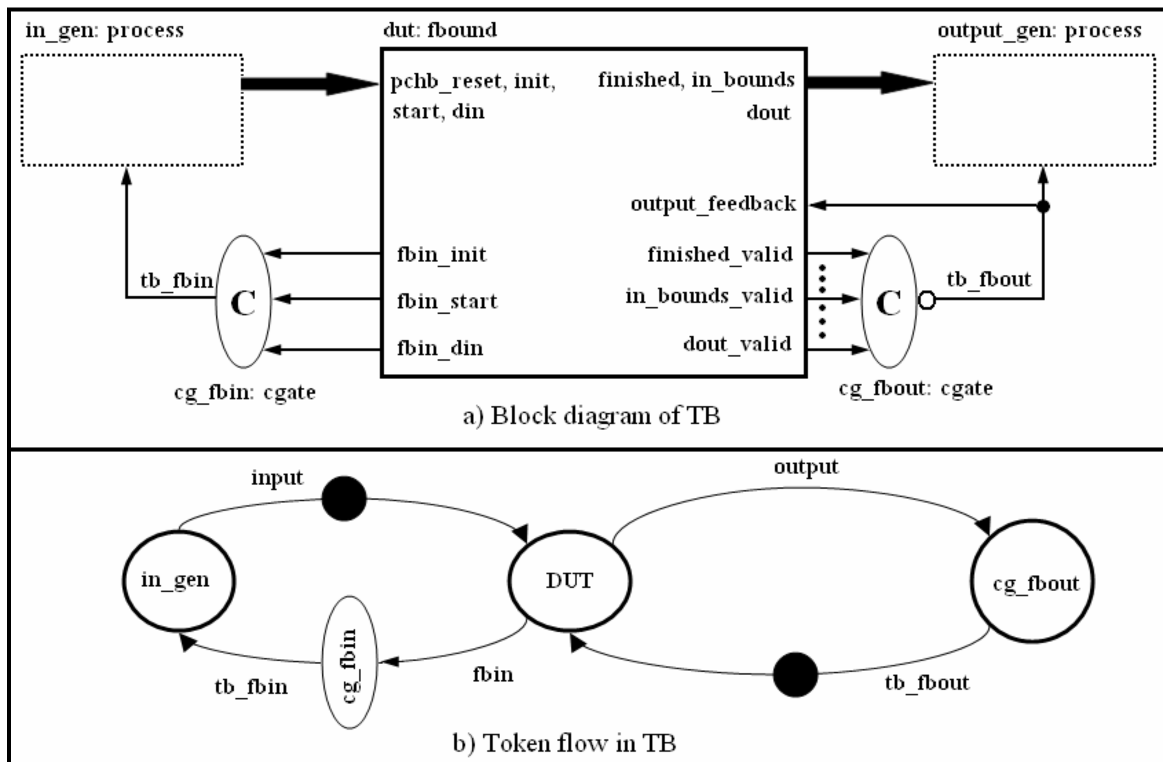
test vectors between +15 and -15, with -5 and +5 used as the upper and lower bound values for the clipper circuits. The block diagram and marked graph of the testbench used for PL-QDI simulation is shown in Figure 6.7. Table 2 gives the comparison of the number of gates in the clocked and PL-QDI designs. Simulation results verified that the PL-QDI output computations matched those of the clocked system.



Note Adopted from [60]

Figure 6.6 Datapath and control of clipper circuit

A pipelined variation of the 64-bit clipper circuit [60] was also tested. The pipeline contains three stages. The first pipeline stage is used to load upper/lower bound values and input test vector values for the clipper circuit. The second stage is used to compare the input value with the lower bound value and the third stage for comparing the input value with the higher bound value. Table 6.2 shows the number of gates for the clocked and PL-QDI design. The same test vectors used for the non-pipelined designs were also used for this design, and the PL-QDI simulation results matched the clocked simulation results.



Note Adopted from [60] (a) Block diagram (b) Marked graph representation

Figure 6.7 PL-QDI 64-bit clipper testbench

Table 6.3 Gate count for 64-bit clipper circuits

Designs	Clocked	PL-QDI
	Total gates	**Total gates**
Non-pipelined 64-bit clipper	1964	1964
3-stage pipelined 64-bit clipper	9180	9308

6.3.3 picoJava-II Floating Point Unit

This mapping was done to illustrate that the PL-QDI methodology can also work for complex IP cores implemented by others. A floating point unit from Sun Microsystem's [59] picoJavaII CPU that performs both single and double precision floating point operations in IEEE 754 format was used to test the PL-QDI methodology. Verilog RTL of the FPU is available from Sun Microsystems. The FPU is a microcoded design with a 32-bit datapath and that uses two 160 X 54 bit ROMs to store the microcode. This test case is different from previous cases, as the FPU RTL is specified in Verilog and the design uses a microcoded architecture. The FPU Verilog RTL was restructured to separate the microcode ROMs from the rest of the synthesizable datapath. This was needed so that a PL-QDI wrapper could be placed around the microcoded ROMs. The Verilog FPU RTL was synthesized to a clocked gate netlist, and then mapped to a PL-QDI gate netlist using the PL-QDI mapper tool.

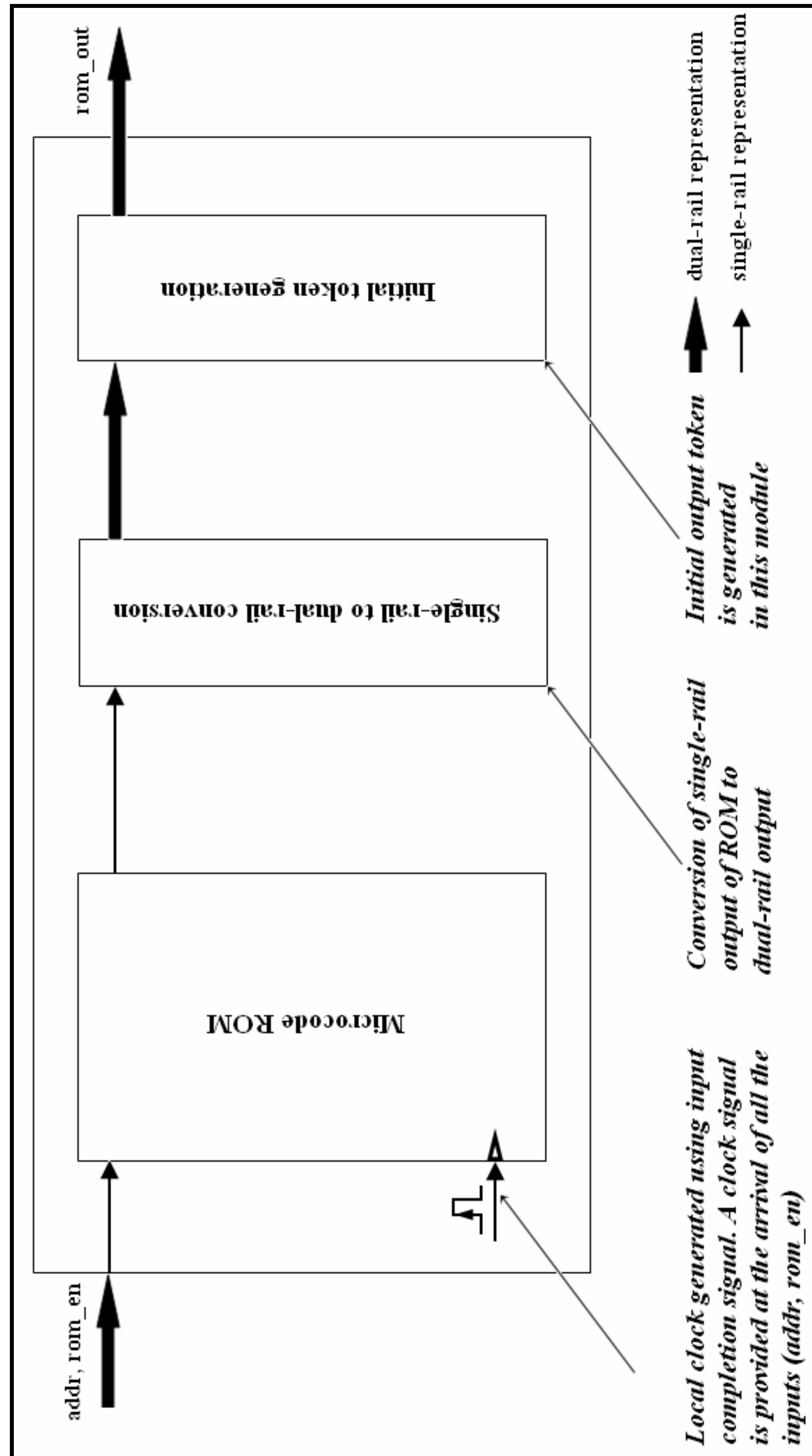


Figure 6.8 PL-QDI wrapper used around microcode ROM

A PL-QDI wrapper was created around the microcoded ROM to make it function as a PL-QDI ROM. The block diagram of the PL-QDI wrapper around the microded ROM is shown in Figure 6.8. The PL-QDI ROM was interfaced with the PL-QDI gate netlist obtained from the PL-QDI mapper tool. The wrapper functionality is twofold: it behaves as a PL-QDI gate when interacting with the PL-QDI design and as a regular clocked I/O interface for the internal clocked microcoded ROM. The wrapper reads dual-rail outputs and input feedback from the PL-QDI design, generates output feedback, converts single-rail outputs of the microcode ROM to dual-rail outputs, and places an initial token after the release of reset at its output. The wrapper supplies inputs from the PL-QDI design to the clocked ROM, converts single-rail ROM output to dual-rail output required for PL-QDI design, and generates a local clock signal required for the clocked ROM triggered by the arrival of all input signals.

The PL-QDI FPU was simulated by using 20 randomly generated test vectors to test each of the single, double precision arithmetic operations and conversion operations listed in Table 6.3 and Table 6.4. Simulation results of the PL-QDI system matched those of the clocked system. The block diagram and marked graph representation of the testbench used to simulate PL-QDI FPU is shown in Figure 6.9 and Figure 6.10. Table 6.5 shows the comparison of the number of gates used in clocked picoJava-II FPU and PL-QDI FPU unit.

Table 6.4 Floating point unit arithmetic operations

No	Floating Point Arithmetic Operations Description
1	Single precision addition
2	Single precision subtraction
3	Single precision multiplication
4	Single precision division
5	Single precision reminder
6	Single precision less than comparison
7	Single precision greater than comparison
8	Double precision addition
9	Double precision subtraction
10	Double precision multiplication
11	Double precision division
12	Double precision reminder
13	Double precision less than comparison
14	Double precision greater than comparison

Table 6.5 Floating point unit data type conversion operations

No	Floating Point Unit Conversion Operations Description
1	IEEE 754 single precision floating point number to double precision number
2	IEEE 754 single precision floating point number to integer number
3	IEEE 754 single precision floating point number to long number
4	IEEE 754 double precision floating point number to single precision number
5	IEEE 754 double precision floating point number to integer number
6	IEEE 754 double precision floating point number to long number
7	Integer number to IEEE 754 single precision floating point number
8	Integer number to IEEE 754 double precision floating point number
9	Long number to IEEE 754 single precision floating point number
10	Long number to IEEE 754 double precision floating point number

Table 6.6 Gate count of picoJava-II FPU designs

Designs	Clocked	PL-QDI
	Total gates	Total gates
picoJava-II FPU	17561	17571

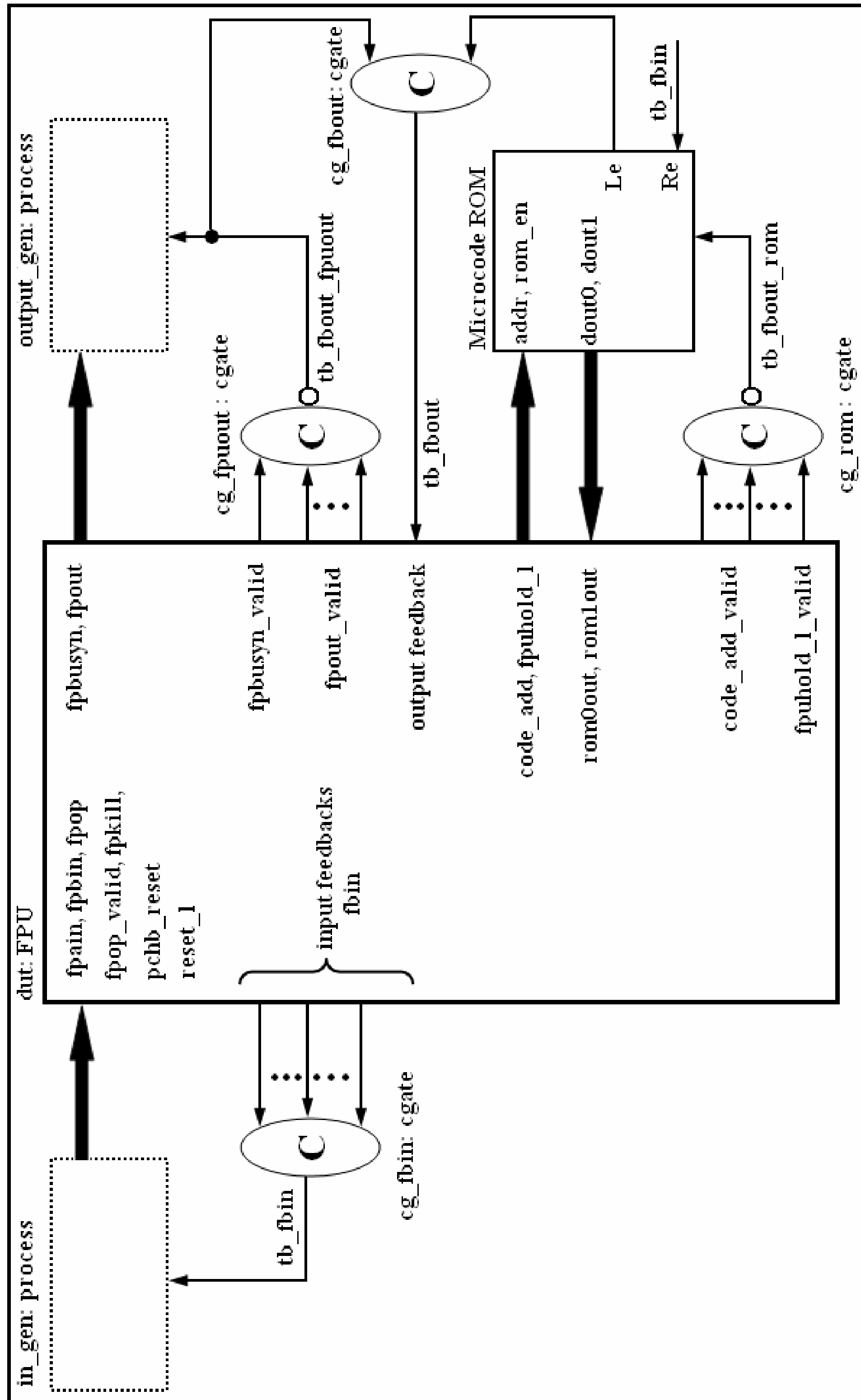


Figure 6.9 PL-QDI 64-bit FPU block diagram

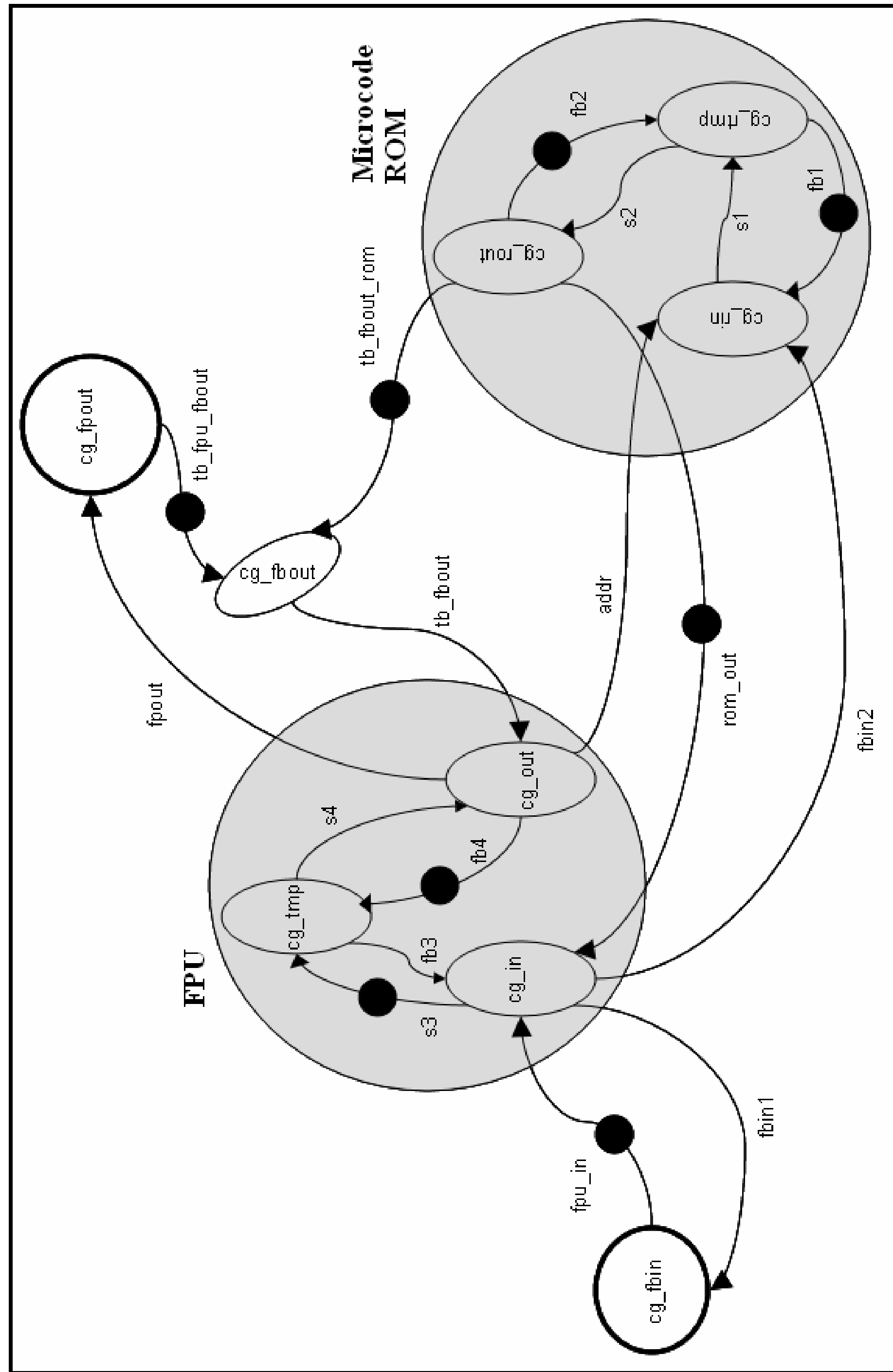


Figure 6.10 PL-QDI 64-bit FPU marked graph representation

CHAPTER VII

CONCLUSION AND FUTURE WORK

This research presented an asynchronous PL-QDI synthesis algorithm and CAD flow to counter the growing problems of synchronous circuits and to address the drawbacks of current asynchronous methodologies. This chapter summarizes the results and future work involving the PL-QDI methodology.

7.1 Summary of Results

- The concept of marked graphs and token abstraction were extended to QDI circuits. PL-QDI gates for PL-QDI design were built by modifying Caltech's PCHB gate design. The final PL-QDI systems obtained are live, safe, satisfy initial token marking rules, maintain the synchronous paradigm, and use four-phase handshaking protocol for communication between PL-QDI gates.
- A PL-QDI CAD tool flow was developed using commercial synchronous CAD tools. The use of PL-QDI CAD tool flow aims to decrease design time to market of asynchronous PL-QDI systems and reduces production cost by reducing the need for proprietary tools.
- Because the PL-QDI methodology begins with a clocked netlist, it does not require any special skills to use this methodology. Anyone with synchronous RTL design skills can use this tool for designing asynchronous PL-QDI systems. This

encourages people outside the asynchronous community to experiment with an asynchronous methodology and increases the manpower available to the IC industry for implementing asynchronous designs.

- PL-QDI systems take natural advantage of the early evaluation capability of the PL-QDI gate without the need for extra logic as required for the static CMOS EE gates used in [18].
- PL-QDI gates do not have output latches, thus reducing the forward latency compared to original static CMOS PL systems.
- The PL-QDI sample designs in Chapter VI showed that clocked RTL designs, including IP cores, can be used to generate asynchronous PL-QDI systems.

In short, the PL-QDI methodology and PL-QDI CAD flow encourage designers to explore PL-QDI asynchronous design as an alternative design technique for logic design.

7.2 Future Work

7.2.1 Modified PL-QDI Gate Library

The PL-QDI gate library should be expanded to include PL-QDI gates with three or four inputs to take advantage of the parallel structure in the pulldown compute block.

7.2.2 Physical Design

PL-QDI systems should be physically implemented and compared to other asynchronous and synchronous designs. This will help to evaluate area, power, and speed of PL-QDI systems, allowing comparisons to other asynchronous implementations.

REFERENCES

[1] M. B. Josephs, S. M. Nowick, C. H. V. Berkel, “Modeling and Design of Asynchronous circuits”, *Proceedings of IEEE on Asynchronous Circuits and Systems*, version 87, February 1999.

[2] R. O. Ozdag, P. A. Beerel, “High-Speed QDI Asynchronous Pipelines”, *Proceedings of the 8th International Symposium on Asynchronus Circuits and Systems* (ASYNC’02) 2002.

[3] C. Piguet, "Logic Synthesis of Race-Free Asynchronous CMOS Circuits" *IEEE JSolid State Circuits,* Vol 26, no 3, March 1991, pp. 271-380.

[4] T. Verhoeff, “Delay-insensitive codes-An overview,” *Distributed Computing*, vol. 3, no. 1, pp. 1-8, 1988.

[5] A. J. Martin, A. Lines, R. Manohar, M. Nystrom, P. Penzes, R. Southworth, U. Cummings, T. K. Lee, “The Design of an Asynchronous MIPS R3000 Microprocessor”, in *Proceedings of 17th Conference on Advanced Research in VLSI*, 164-181, IEEE Computer Society Press, 1997.

[6] A. J. Martin, M. Nystrom, K. Papadantonakis, P. I. Penzes, P. Prakash, C. G. Wong, J. Chang, K. S. Ko, B. Lee, E. Ou, J. Pugh, E. V. Talvala, “The Lutonium: A Sub-Nanojoule Asynchronous 8051 Microcontroller”, *Proceedings of 9th IEEE International Symposium on Asynchronous Systems & Circuits (ASYNC)*, May 2003.

[7] A. J. Martin, M. Nystrom, C. G. Wong, “Three Generatinos of Asynchronous Microprocessors”, *IEEE Design & Test of Computers, special issue on Clockless VLSI Design*, November/December 2003.

[8] K. Saleh, H. Pedram, M. Naderi, M. H. Shafiaabadi, H. Kalantari, A. Farhoodfar, “Synthesis Tools for Asynchronous Circuits Based on PCFB and PCHB”, in *Proceedings of the 9th Annual Computer Society of Iran Computer Conference (CSICC2004)*, Feb. 2004.

[9] D. Linder, “Phased Logic: A methodology for delay insensitive Synchronous Circuitry”, PhD Thesis, Mississippi State University, 1994.

[10] T. Murata, "Petri-Nets: Properties, Analysis and Applications", *IEEE Proceedings*, Vol 77, pp 541-580, April 1989.

[11] R. O. Ozdug, "Template Based Asynchronous Design", PhD Thesis, University of Southern California, November 2003.

[12] A. Lines, "Pipelined Asynchronous Circuits", Masters Thesis, California Institute of Technology, June 1995.

[13] K. V. Berkel, "Beware the Isochronic fork," Integration, the VLSI Journal., vol. 13, pp. 103-128, 1992.

[14] A. J. Martin "The limitations of delay-insensitivity in Asynchronous Circuits", in *Proceedings of the 1990 MIT Conference on Advanced Research in Asynchronous Circuits*, 1990, pp 263-278.

[15] C. J. Myers, *"Asynchronous circuit design"*, Wiley-Interscience Publication, 2001.

[16] J. Sparso, S. Furber, "Principles of Asynchronous circuit design", Kluwer Academic Publishers.

[17] D Linder, J. Harden, "Phased Logic: Supporting the Asynchronous design paradigm with delay-insensitive circuit", *IEEE transactions on Computers*, pp 1031-1044, September 1996 .

[18] R. B. Reese, M. A. Thornton, C. Traver, D. Hemmendinger, "Early Enhancement for Performance Enhancement in Phased Logic", Vol 24, pp 532-550, April 2005.

[19] R. B. Reese, M. A. Thornton, C. Traver, "A Fine-grain Phased Logic CPU", in *IEEE Computer Society Annual Symposium* on VLSI (ISVLSI), Feb 2003, pp 70-79.

[20] D. E. Muller, W. S. Bartky, "A Theory of Asynchronous Circuits", in Proceedings of International Sypmosium on Theory of Switching, vol 29, pp.204-243, 1959.

[21] M. E. Dean, T. E. Williams, D. L. Dill, "Efficient Self-Timing with Level-Encoded 2-Phase Dual-Rail (LEDR)", in *Advanced Research in VLSI*, 1991, pp. 55-70.

[22] A. Davis, S. M. Nowick, “An introduction to asynchronous circuit design”, in Encylopedia of Computer Science and Technology, vol. 38, supplement 23, 1998.

[23] C. L. Seitz, “System timing”, in “Introduction to VLSI Systems”, C. A. Mead and L. A. Conway, Eds. MA: Addison-Wesley, 1980, ch. 7.

[24] P A. Beerel, “CAD Tools for the Synthesis, Verification, and Testability of Robust Asynchronous Circuits”, PhD thesis, Stanford University, 1994.

[25] K. Berkel, A. Bink, “Single-track handshaking signaling with application to micropipelines and handshake circuits”, *Proceedings of International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 122-133, IEEE Computer Society press, March 1996.

[26] C. A. R. Hoare, “Communicating Sequential Processes”, *ACM*, vol. 21, no. 8, pp. 666-677, 1978.

[27] A. J. Martin, “Compiling Communicating Processes into Delay-Insensitive VLSI Circuits”, *Distributed Computing*, pp 226-234, 1986.

[28] A. Kondratyev, K. Lwin, "Design of Asynchronous Circuits by Synchronous CAD Tools”, *Proceedings of IEEE Design & Test*, vol 19, pp 107 – 177.

[29] A. R. Newton, A. L. Sangiovanni-Vincentelli, “Computer-Aided Design for VLSI Circuits”, *IEEE Communication Magazine*, August 2002.

[30] G. E. Moore, “Intel – Memories and the Microprocessors”, The National Academies Press http://darwin.nap.edu/books/0309054451/html/77.html

[31] K. D. Emerson, “Asynchronous Design – an Interesting Alternative” , *Proceedings of 10th International Conference on VLSI Design*, January, 1997.

[32] S. Hauck, “Asynchronous Design Methodologies: An Overview”, *Proceedings of the IEEE*, Vol. 83, No. 1, pp 69-93, January, 1995.

[33] M. Afghani, C. Svensson “Performance of Synchronous and Asynchronous Schemes for VLSI Systems”, *Proceedings of IEEE Transactiona on computers*, vol 41, no. 7, pp 858-872, July 1992.

[34] A. Smirnov, A. Taubin, M. Karpovsky, L. Rozenblyum, “Gate Transfer Level Synthesis as an Automated Approach to Fine-Grain Pipelining”, *Proceedings of Workshop on Token Based Computing*, June 2004.

[35] S. M. Kang, “Computer-Aided Design For VLSI”, *Proceedings of International Conference on Circuits and Systems*, vol 1, pp 1-5, June 1991.

[36] R. B. Reese, M. A. Thornton, C. Traver, “A Standard Cell Implementation of a Phased Logic CPU”, *Proceedings of Workshop on Token based computing (ToBaCo), Satellite Event of the 25-th International conference on application and theory of Petri nets,* June, 2004.

[37] R. B. Reese, M. A. Thornton, C. Traver, “A Fine-Grain Phased Logic CPU”, *Proceedings of IEEE Computer Society Annual Symposium on VLSI*, pp. 70-79, Feb 2003.

[38] J. M. Rabaey, A. Chandrakasan, B. Nikolic, *“Digital Integrated Circuits – A Design Perspective Second Edition”*, 2nd Edition, Prentice Hall Electronics and VLSI Series, 2003.

[39] H. W. Johnson, M. Graham, *“High-Speed Digital Design – A handbook of Black Magic”*, Prentice Hall PTR, 1993.

[40] P. Kurup, T. Abbasi, R. Bedi, *“It’s the Methodology, Stupid!”*, ByteK Designs, Inc, 1998.

[41] M. Ligthart, K. Fant, R. Smith, A. Taubin, A. Kondratyev, “Asynchronous Design Using Commercial HDL Synthesis Tool”, *Proceedings of Sixth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp 114-125, 2000.

[42] C. P. Sotiriou, “Implementing Asynchronous Circuits using a Conventional EDA Tool-Flow”, *Proceedings of Design Automation Conference*, pp 415-418, June 2002.

[43] J. W. Specks, “Computer-Aided Design and Scaling of Deep Submicron CMOS”, *IEEE Transactions on Computer Aided Design of Integrated Circuit and Systems,* vol 12, no 9, pp. 1357-1367, 1993.

[44] W. Maly, “Computer-Aided Design for VLSI Circuit Manufacturability”, *Proceedings of the IEEE*, Vol 78, No 2, pp 356-392, February 1990.

[45] J. McCardle, D. Chester, “Measuring an Asynchronous Processor’s Power and Noise”, Synopsys Solvenet, https://solvnet.synopsys.com/

[46] www.theseuslogic.com

[47] M. K. Gowan, L. L. Biro, D. B. Jackson, “Power Considerations in the Design of the Alpha 21264 Microprocessor”, *Design Automation Conference*, pp 726-731, 1998.

[48] J. Sparso, J. Staunstrup, “Design and Performance Analysis of Delay Insensitive Multi-Ring Structures”, *Proceedings of the twenty-sixth International Conference on Systems Sciences*, vol 1, pp. 349-358, Jan 1993.

[49] K. V. Berkel, F. Huberts, A. Peeters, “Stretching Quasi Delay Insensitivity by Means of Extended Isochronic Forks”, *Proceedings of Second Working Conference on Asynchronous Design Methodologies*, pp 99-106, 1995.

[50] S. Tam, S. Rusu, U. N. Desai, R. Kim, J. Zhang, I. Young, “Clock Generation and Distribution for the First IA-64 Microprocessor”, *Proceedings of IEEE Journal of Solid-State Circuits*, vol. 35, no. 11, pp 1545-1552, November 2000.

[51] C. H. V. Berkel, M. B. Josephs, S. M. Nowick, “Scanning the Technology – Applications of Asynchronous Circuits”, *Proceedings of the IEEE*, vol 87, no. 2, pp 234-242, February 1999.

[52] P. E. Gronowski, W. J. Bowhill, R. P. Preston, M. K. Gowan, R. L. Allmon, “High-Performamce Microprocessor Design”, *Proceedings of IEEE Journal of Solid-State Circuits*, vol. 33, no. 5, pp 676-686, May 1998.

[53] D. W. Bailey, B. J. Benschneider, "Clocking Design and Analysis for a 600-MHz Alpha Microprocessor", *Proceedings of IEEE Journal of Solid-State Circuits*, vol. 33, no. 11, pp. 1627-1633, November 1998.

[54] Synopsys, www.synopsys.com

[55] Mentor Graphics Modelsim, http://www.model.com/default.asp

[56] Legend Design Technology, http://www.legenddesign.com/products/msim.shtml

[57] Altera, www.altera.com

[58] Xilinx, www.xilinx.com

[59] SUN Microsystems, www.sun.com

[60] R. B. Reese, "Phased Logic Async 2004 Tutorial Document", http://www.hpc.msstate.edu/mpl/projects/phased_logic/publications/async2004_tut.pdf