


2005

## Design And Synthesis Of Clockless Pipelines Based On Self-resetting Stage Logic

Abdelhalim Alsharqawi  
*University of Central Florida*

 Part of the [Computer Engineering Commons](#)  
Find similar works at: <https://stars.library.ucf.edu/etd>  
University of Central Florida Libraries <http://library.ucf.edu>

This Doctoral Dissertation (Open Access) is brought to you for free and open access by STARS. It has been accepted for inclusion in Electronic Theses and Dissertations, 2004-2019 by an authorized administrator of STARS. For more information, please contact [STARS@ucf.edu](mailto:STARS@ucf.edu).

---

### STARS Citation

Alsharqawi, Abdelhalim, "Design And Synthesis Of Clockless Pipelines Based On Self-resetting Stage Logic" (2005). *Electronic Theses and Dissertations, 2004-2019*. 524.  
<https://stars.library.ucf.edu/etd/524>

**DESIGN AND SYNTHESIS OF CLOCKLESS PIPELINES  
BASED ON  
SELF-RESETTING STAGE LOGIC**

by

ABDELHALIM M. ALSHARQAWI  
B.S. Princess Sumaya University, 2000  
M.S. University of Central Florida, 2002

A dissertation submitted in partial fulfillment of the requirements  
for the degree of Doctor of Philosophy  
in the Department of Electrical and Computer Engineering  
in the College of Engineering and Computer Science  
at the University of Central Florida  
Orlando, Florida

Fall Term  
2005

Major Professor: Abdel Ejnioui

© 2005 Abdelhalim Alsharqawi

## ABSTRACT

For decades, digital design has been primarily dominated by clocked circuits. With larger scales of integration made possible by improved semiconductor manufacturing techniques, relying on a clock signal to orchestrate logic operations across an entire chip became increasingly difficult. Motivated by this problem, designers are currently considering circuits which can operate without a clock. However, the wide acceptance of these circuits by the digital design community requires two ingredients: (i) a unified design methodology supported by widely available CAD tools, and (ii) a granularity of design techniques suitable for synthesizing large designs. Currently, there is no unified established design methodology to support the design and verification of these circuits. Moreover, the majority of clockless design techniques is conceived at circuit level, and is subsequently so fine-grain, that their application to large designs can have unacceptable area costs.

Given these considerations, this dissertation presents a new clockless technique, called *self-resetting stage logic* (SRSL), in which the computation of a block is reset periodically from within the block itself. SRSL is used as a building block for three coarse-grain pipelining techniques:

- (i) *Stage-controlled self-resetting stage logic* (S-SRSL) Pipelines: In these pipelines, the control of the communication between stages is performed locally between each pair of stages. This communication is performed in a uni-directional manner in order to simplify its implementation.

- (ii) *Pipeline-controlled self-resetting stage logic (P-SRSL) Pipelines*: In these pipelines, the communication between each pair of stages in the pipeline is driven by the oscillation of the last pipeline stage. Their communication scheme is identical to the one used in S-SRSL pipelines.
- (iii) *Delay-tolerant self-resetting stage logic (D-SRSL) Pipelines*: While communication in these pipelines is local in nature in a manner similar to the one used in S-SRL pipelines, this communication is nevertheless extended in both directions. The result of this bi-directional approach is an increase in the capability of the pipeline to handle stages with random delay.

Based on these pipelining techniques, a new design methodology is proposed to synthesize clockless designs. The synthesis problem consists of synthesizing an SRSL pipeline from a gate netlist with a minimum area overhead given a specified data rate. A two-phase heuristic algorithm is proposed to solve this problem. The goal of the algorithm is to pipeline a given datapath by minimizing the area occupied by inter-stage latches without violating any timing constraints. Experiments with this synthesis algorithm show that while P-SRSL pipelines can reach high throughputs in shallow pipelines, D-SRSL pipelines can achieve comparable throughputs in deeper pipelines.

## **ACKNOWLEDGMENTS**

I would like to express my gratitude to my advisor, Dr. Abdel Ejnoui, who invested time and patience toward the completion of this dissertation. Without his encouragement, support, and guidance, this dissertation would not have been possible. I would like to thank my committee members, Drs. Issa Batarseh, Ronald F. Demara, Hassan Foroosh, and Alain Kassab, for their support and willingness to serve on my defense examination. In particular, I would like to show my special gratitude to Dr. Issa Batarseh for supporting my research work in times when support resources were scarce. Beside Dr. Batarseh, I would like to show the same gratitude to Dr. Harold Klee for providing me with department support whenever possible. Also, I would like to thank Donald Harper for his patience and generosity in supporting the EDA design tools in the VLSI Lab. Furthermore, I am grateful to my friend and lab mate, Rashad Oreifej, for his insightful comments and helpful feedback on my work.

I would like to express my sincere thanks to my parents for their unconditional support, guidance, and sacrifice. They have always believed in me and gratified me with their unending love. Finally, I would like to thank my loving wife Amani for her generosity, support, and understanding throughout my graduate studies. Her presence has been a pillar of steadfastness on which to lean after long hours of toiling in the VLSI Lab.

## TABLE OF CONTENTS

|   |      |
|---|------|
| LIST OF FIGURES .....                               | xii  |
| LIST OF TABLES .....                                | xvii |
| CHAPTER ONE: INTRODUCTION.....                      | 1    |
| 1.1 Motivation.....                                 | 1    |
| 1.1.1 The Clocking Problem .....                    | 1    |
| 1.1.2 Growing Importance of Clockless Circuits..... | 2    |
| 1.1.3 Coarse-Grain Clockless Pipelining.....        | 4    |
| 1.2 Design Methodology in Clocked Circuits.....     | 5    |
| 1.2.1 Specification and Modeling .....              | 5    |
| 1.2.2 Verification .....                            | 6    |
| 1.2.3 Synthesis .....                               | 6    |
| 1.2.4 Mapping .....                                 | 8    |
| 1.3 Limitations of Clocked Circuits.....            | 8    |
| 1.3.1 Clock Frequency .....                         | 8    |
| 1.3.2 Timing Closure .....                          | 9    |
| 1.3.3 Power Implications .....                      | 9    |
| 1.3.4 Area Implications.....                        | 9    |
| 1.3.5 Noise Margins.....                            | 10   |
| 1.3.6 Multiple Clock Domains.....                   | 10   |
| 1.4 Clockless Circuits .....                        | 11   |
| 1.4.1 Self-Clocked Circuits.....                    | 11   |

|  |    |
|--|----|
| 1.4.2 Speed-Independent Circuits .....                     | 13 |
| 1.4.3 Delay-Insensitive Circuits.....                      | 13 |
| 1.4.4 Self-Timed Circuits.....                             | 14 |
| 1.4.5 Self-Resetting Circuits.....                         | 15 |
| 1.5 Design Methodology in Clockless Circuits .....         | 15 |
| 1.6 Contributions of the Dissertation .....                | 17 |
| 1.7 Overview of the Dissertation .....                     | 19 |
| CHAPTER TWO: RELATED CLOCKLESS DESIGN METHODOLOGIES .....  | 21 |
| 2.1 Petri Nets.....  | 21 |
| 2.2 Signal Transition Graphs .....                         | 25 |
| 2.3 Micropipelines .....                                   | 27 |
| 2.4 Null Convention Logic .....                            | 30 |
| 2.5 Burst Mode Machine.....                                | 34 |
| 2.6 Handshake Circuits .....                               | 36 |
| 2.7 Extended Delay Insensitive Model .....                 | 39 |
| 2.8 Summary .....  | 39 |
| CHAPTER THREE: STAGE-CONTROLLED SELF-RESETTING STAGE LOGIC |    |
| PIPELINES.....   | 42 |
| 3.1 SRSL .....   | 42 |
| 3.2 S-SRSL Linear Pipelines .....                          | 44 |
| 3.3 S-SRSL Non-linear Pipelines .....                      | 50 |
| 3.3.1 S-SRSL Join Operation.....                           | 50 |
| 3.3.2 S-SRSL Fork Operation.....                           | 52 |



|  |    |
|--|----|
| 3.4 Performance of the Pipeline.....                                   | 54 |
| 3.4.1 Parameter Definitions .....                                      | 54 |
| 3.4.2 Analysis of the Reset and Evaluate Phase .....                   | 55 |
| 3.4.3 Effect of $\delta$ on the Pipeline Stages .....                  | 58 |
| 3.4.4 $\delta$ and Pipeline Depth.....                                 | 59 |
| 3.4.5 Area Cost .....  | 61 |
| 3.4.6 Fault Handling .....   | 61 |
| 3.5 Prototype Implementation of the S-SRSL Pipelines.....              | 65 |
| 3.5.1 The S-SRSL Linear Pipeline.....                                  | 66 |
| 3.5.2 The S-SRSL Non-Linear Pipeline .....                             | 70 |
| 3.5.2.1 The S-SRSL Join Pipeline .....                                 | 70 |
| 3.5.2.2 The S-SRSL Fork Pipeline.....                                  | 72 |
| 3.6 Summary .....  | 74 |
| CHAPTER FOUR: PIPELINE-CONTROLLED SELF-RESETTING STAGE LOGIC PIPELINES |    |
| .....  | 75 |
| 4. 1 P-SRSL Linear Pipeline.....                                       | 75 |
| 4.2 P-SRSL Non-Linear Pipelines .....                                  | 82 |
| 4.2.1 P-SRSL Join Pipeline.....  | 82 |
| 4.2.2 P-SRSL Fork Pipeline.....  | 85 |
| 4.3 Performance of the Pipeline.....                                   | 87 |
| 4.3.1 Analysis of the Reset and Evaluate Phase .....                   | 87 |
| 4.3.2 Effect of $\delta$ on the Pipeline Stages .....                  | 90 |
| 4.3.3 Effect of the Period on the Latch Enable .....                   | 91 |

|   |     |
|---|-----|
| 4.3.4 Area Cost .....   | 92  |
| 4.3.5 Fault Handling .....  | 92  |
| 4.4 Prototype Implementation of the P-SRSL Pipeline .....           | 94  |
| 4.4.1 Implementation of the Linear Pipeline .....                   | 94  |
| 4.4.2 Implementation of the Non-Linear Pipelines.....               | 97  |
| 4.4.2.1 The P-SRSL Join Pipeline .....                              | 97  |
| 4.4.2.2 The P-SRSL Fork Pipeline.....                               | 100 |
| 4.5 Comparison of P-PRSL to S-SRSL Pipelines.....                   | 102 |
| 4.6 Summary .....   | 103 |
| CHAPTER FIVE: DELAY TOLERANT SELF-RESETTING STAGE LOGIC PIPELINES.. | 104 |
| 5.1. D-SRSL Linear Pipeline .....                                   | 104 |
| 5.1.1 Pipeline Structure.....                                       | 104 |
| 5.1.2 Phase Control Block .....                                     | 106 |
| 5.1.3 Latch Control Block.....                                      | 108 |
| 5.2. D-SRSL Non-Linear Pipelines .....                              | 109 |
| 5.2.1 D-SRSL Join Pipeline .....                                    | 109 |
| 5.2.2 D-SRSL Fork Pipeline .....                                    | 114 |
| 5.3. Performance of the Pipeline.....                               | 118 |
| 5.3.1 The Reset and Evaluate Phase .....                            | 118 |
| 5.3.2 Duration of Latch Enable.....                                 | 122 |
| 5.3.3 Stage Delay and Period.....                                   | 123 |
| 5.3.4 Area Cost .....   | 128 |
| 5.3.5 Fault Handling .....  | 128 |

|   |     |
|---|-----|
| 5.4 Prototype Implementation of the D-SRSL Pipeline.....  | 131 |
| 5.4.1 Implementation of the PC Block.....                 | 131 |
| 5.4.2 Implementation of the LC Block .....                | 132 |
| 5.4.3 Implementation of the Join Block.....               | 134 |
| 5.4.4 Implementation of the Fork Block.....               | 136 |
| 5.4.5 Implementation of D-SRSL Pipeline .....             | 137 |
| 5.5. Conclusion .....                                     | 141 |
| CHAPTER SIX: SYNTHESIS OF SRSL PIPELINES .....            | 143 |
| 6.1 SRSL Pipeline Design Methodology .....                | 143 |
| 6.2 Synthesis of SRSL Pipelines.....                      | 145 |
| 6.3 Preliminaries .....                                   | 146 |
| 6.4 Modeling of the Synthesis Problem.....                | 150 |
| 6.5 Proposed Solution of the SRSL Pipeline Synthesis..... | 156 |
| 6.5.1 Phase I: Stage Assignment.....                      | 157 |
| 6.5.1.1 Phase I Approach .....                            | 158 |
| 6.5.1.2 Phase I Algorithm .....                           | 159 |
| 6.5.2 Phase II: Vertex Shuffling .....                    | 161 |
| 6.5.2.1 Phase II Approach.....                            | 161 |
| 6.5.2.2 Phase II Algorithm.....                           | 168 |
| 6.6 Experimental Results .....                            | 170 |
| 6.6.1 P-SRSL Pipelining Experiments.....                  | 170 |
| 6.6.2 D-SRSL Pipelining Experiments .....                 | 177 |
| 6.6.3 Summary of the Experiment Results .....             | 182 |

|                                     |     |
|-------------------------------------|-----|
| 6.7 Summary .....                   | 184 |
| CHAPTER SEVEN: CONCLUSION.....      | 186 |
| 7.1 Summary of Completed Work ..... | 186 |
| 7.2 Future Work .....               | 190 |
| LIST OF REFERENCES .....            | 193 |

## LIST OF FIGURES

|   |    |
|---|----|
| Figure 1.1: Design flow of clocked circuits [14].                         | 7  |
| Figure 1.2: General architecture of SC circuits [25].                     | 12 |
| Figure 2.1: C-element and its surrounding dummy environment [27].         | 22 |
| Figure 2.2: The PN of the C-element shown in Figure 2.1 [27].             | 22 |
| Figure 2.3: Petrify framework.  | 24 |
| Figure 2.4 :Timing diagram of the C-element shown in Figure 2.1.          | 26 |
| Figure 2.5: STG of the C-element shown in Figure 2.1.                     | 26 |
| Figure 2.6: Synthesis flow of clockless circuits from STG specifications. | 27 |
| Figure 2.7: Micropipeline handshake protocols.                            | 28 |
| Figure 2.8: Basic structure of a micropipeline.                           | 28 |
| Figure 2.9: Pipefitter framework.   | 29 |
| Figure 2.10: NCL 2-of-3 threshold gate.                                   | 31 |
| Figure 2.11: A half adder circuit in conventional Boolean logic.          | 31 |
| Figure 2.12: NCL half adder circuit.                                      | 32 |
| Figure 2.13: RTL flow for NCL design [51].                                | 33 |
| Figure 2.14: Burst mode specification of a C-element.                     | 35 |
| Figure 2.15: Handshake channel.   | 36 |
| Figure 2.16: The Tangram Toolset.   | 38 |
| Figure 3.1: Reset and evaluate network of an SRSL stage.                  | 43 |
| Figure 3.2: STG of the reset network shown in Figure 3.1.                 | 43 |
| Figure 3.3: A four-stage S-SRSL pipeline.                                 | 44 |

|  |    |
|--|----|
| Figure 3.4: STG of the S-SRSL pipeline shown in Figure 3.3. ....   | 45 |
| Figure 3.5: Two execution cycles of a four-stage S-SRSL Pipeline.....  | 49 |
| Figure 3.6: Structure of the join S-SRSL pipeline.....   | 51 |
| Figure 3.7: STG of the S-SRSL join pipeline shown in Figure 3.6.....   | 52 |
| Figure 3.8: Structure of the fork S-SRSL pipeline. ....  | 53 |
| Figure 3.9: STG of the S-SRSL fork pipeline shown in Figure 3.8.....   | 54 |
| Figure 3.10: Simulation snapshot of stage 15 and 16 in a 16-stage prototype S-SRSL pipeline. ....                      | 56 |
| Figure 3.11: Simulation snapshot of stage 1 and 2 in a 16-stage prototype S-SRSL pipeline. ....                        | 56 |
| Figure 3.12: Chip layout of the four-bit 16-stage S-SRSL pipeline. ....  | 66 |
| Figure 3.13: Simulation results of $d(L^+)$ , $d(R)$ , $d(E)$ , $\delta$ , and $P$ in a 16-stage S-SRSL pipeline. .... | 68 |
| Figure 3.14: The empirical and analytical values of $d(R)$ and $d(L^+)$ in a 16-stage S-SRSL<br>pipeline.....          | 69 |
| Figure 3.15: Four-bit six-stage S-SRSL join pipeline. ....   | 70 |
| Figure 3.16: Simulation snapshot of the prototype S-SRSL join pipeline.....  | 71 |
| Figure 3.17: Simulation results of $d(L^+)$ , $d(R)$ , $d(E)$ , $\delta$ , and $P$ in the S-SRSL join pipeline.....    | 72 |
| Figure 3.18: Four-bit six-stage S-SRSL fork pipeline.....  | 72 |
| Figure 3.19: Simulation snapshot of the prototype S-SRSL fork pipeline .....   | 73 |
| Figure 3.20: Simulation results of $d(L^+)$ , $d(R)$ , $d(E)$ , $\delta$ , and $P$ in the S-SRSL fork pipeline. ....   | 74 |
| Figure 4.1: A four-stage P-SRSL pipeline.....  | 76 |
| Figure 4.2: STG of the P-SRSL pipeline shown in Figure 4.1. ....   | 78 |
| Figure 4.3: Two execution cycles of a four-stage P-SRSL Pipeline.....  | 81 |
| Figure 4.4: Structure of a join P-SRSL pipeline. ....  | 83 |
| Figure 4.5: STG of the P-SRSL join pipeline shown in Figure 4.3.....   | 84 |

|  |     |
|--|-----|
| Figure 4.6: Structure of a fork P-SRSL pipeline.....   | 85  |
| Figure 4.7: STG of the P-SRSL fork pipeline shown in Figure 4.6.....   | 86  |
| Figure 4.8: Simulation snapshot of stages 13, 14, 15 and 16 in a 16-stage prototype P-SRSL<br>pipeline.....                      | 89  |
| Figure 4.9: Chip layout of the four-bit 16-stage P-SRSL pipeline. ....   | 95  |
| Figure 4.10: Simulation results of $d(L^+)$ , $d(R)$ , $d(E)$ , $\delta$ , and $P$ in a P-SRSL pipeline. ....                    | 96  |
| Figure 4.11: Four-bit six-stage P-SRSL join pipeline. ....   | 98  |
| Figure 4.12: Simulation snapshot of the prototype P-SRSL join pipeline.....  | 98  |
| Figure 4.13: Simulation results of $d(L^+)$ , $d(R)$ , $d(E)$ , $\delta$ , and $P$ in the P-SRSL prototype join<br>pipeline..... | 99  |
| Figure 4.14: Four-bit six-stage P-SRSL fork pipeline. ....   | 100 |
| Figure 4.15: Simulation snapshot of the prototype P-SRSL fork pipeline. ....   | 100 |
| Figure 4.16: Simulation results of $d(L^+)$ , $d(R)$ , $d(E)$ , $\delta$ , and $P$ in the P-SRSL prototype fork<br>pipeline..... | 102 |
| Figure 5.1: A four-stage D-SRSL pipeline. ....   | 105 |
| Figure 5.2: STG of the D-SRSL pipeline shown in Figure 5.1. ....   | 106 |
| Figure 5.3: Phase control block. ....  | 107 |
| Figure 5.4: State graph of the PC block.....   | 107 |
| Figure 5.5: Latch control block.....   | 109 |
| Figure 5.6: State graph of the latch control block.....  | 109 |
| Figure 5.7: D-SRSL join pipeline. ....   | 110 |
| Figure 5.8: STG of the D-SRSL join pipeline shown in Figure 5.7. ....  | 111 |
| Figure 5.9: The Join block. ....   | 111 |

|  |     |
|--|-----|
| Figure 5.10: State graph of the Join block. ....   | 112 |
| Figure 5.11: Prototype D-SRSL join pipeline.....   | 112 |
| Figure 5.12: Simulation snapshot of the prototype D-SRSL join pipeline. ....                               | 113 |
| Figure 5.13: D-SRSL fork pipeline.....   | 115 |
| Figure 5.14: STG of the D-SRSL fork pipeline shown in Figure 5.13. ....                                    | 115 |
| Figure 5.15: Fork block.....   | 116 |
| Figure 5.16: State graph of the Fork block. ....   | 116 |
| Figure 5.17: Prototype D-SRSL fork pipeline. ....  | 116 |
| Figure 5.18: Simulation snapshot of the prototype D-SRSL fork pipeline.....                                | 117 |
| Figure 5.19: Simulation snapshot of stage 14, 15 and 16 in a 16-stage prototype D-SRSL<br>pipeline.....    | 119 |
| Figure 5.20: Simulation snapshot of stages 7 through 11 in a 17-stage prototype D-SRSL<br>pipeline.....    | 124 |
| Figure 5.21: Simulation snapshot of stages 8 and 9 in the 17-stage D-SRSL prototype pipeline.<br>.....     | 125 |
| Figure 5.22: Simulation snapshot of stages 9 and 10 in the 17-stage D-SRSL prototype pipeline.<br>.....    | 126 |
| Figure 5.23: Synthesized netlist of the PC block.....  | 132 |
| Figure 5.24: Synthesized netlist of the LC block.....  | 133 |
| Figure 5.25: Synthesized netlist of the Join block. ....   | 134 |
| Figure 5.26: Synthesized netlist of the Fork block. ....   | 136 |
| Figure 5.27: Simulation results of $P$ , $d(E)$ , $d(R)$ , and $d(L^+)$ in D-SRSL prototype pipeline 1.... | 139 |
| Figure 5.28: Simulation results of $P$ , $d(E)$ , $d(R)$ , and $d(L^+)$ in D-SRSL prototype pipeline 2.... | 140 |



|  |     |
|--|-----|
| Figure 5.29: Simulation results of $P$ , $d(E)$ , $d(R)$ , and $d(L^+)$ in D-SRSL prototype pipeline3.....                     | 141 |
| Figure 6.1: SRSL design flow.....  | 144 |
| Figure 6.2: Example of a Boolean network. ....   | 147 |
| Figure 6.3: Boolean graph of the Boolean network shown in Figure 6.2.....  | 149 |
| Figure 6.4: Latch insertion between two neighboring pipeline stages.....   | 162 |
| Figure 6.5: P-SRSL area as a percentage of the pipeline area across different pipelines of the<br>C6822 benchmark circuit..... | 171 |
| Figure 6.6: Pipeline throughputs for various P-SRSL pipeline depths.....   | 172 |
| Figure 6.7: P-SRSL area as a percentage of the pipeline area across various depth pipelines...                                 | 174 |
| Figure 6.8: Period over area ratios for different depths P-SRSL pipelines. ....  | 176 |
| Figure 6.9: D-SRSL area as a percentage of the pipeline area across different pipelines of the<br>C5135 benchmark circuit..... | 178 |
| Figure 6.10: Pipeline throughputs for various D-SRSL pipeline depths. ....   | 179 |
| Figure 6.11: D-SRSL area as a percentage of the pipeline area across various depth pipelines.                                  | 181 |

## LIST OF TABLES

|  |     |
|--|-----|
| Table 2.1: Summary of clockless design methodologies.....            | 41  |
| Table 3.1: S-SRSL linear pipeline implementation .....               | 67  |
| Table 4.1: P-SRSL pipeline implementation. ....                      | 95  |
| Table 4.2: Comparison summary of the P-SRSL to S-SRSL pipeline. .... | 103 |
| Table 5.1: D-SRSL pipeline implementation.....                       | 138 |
| Table 5.2: Gate area of a single D-SRSL stage. ....                  | 138 |
| Table 6.1: Experimental circuits. ....                               | 170 |
| Table 7.1: SRSL pipelining parameters. ....                          | 188 |
| Table 7.2: Fault handling in the three pipelines.....                | 189 |

# CHAPTER ONE: INTRODUCTION

## **1.1 Motivation**

In this section, the rationale and the motivation behind the work undertaken in this dissertation is presented.

### **1.1.1 The Clocking Problem**

For three decades, digital design has been primarily dominated by clocked circuits since these circuits can be extremely robust and fairly easy to build. The use of a clock signal in clocked circuits introduces a level of abstraction in the time domain that hides many details about the temporal relations among circuit signals. This greatly simplifies the timing analysis of such circuits by reducing it to a mere analysis of the critical paths contained within the circuit. In a clocked circuit, a designer can simply define the combinational logic necessary to compute the given functions and surround it with latches. By setting the clock rate to a sufficiently long period, concerns about undesired signal transitions and the dynamic state of the circuit are eliminated [1]. The ease of design in clocked circuits has made them inevitably highly attractive to members of the commercial and research communities. In return, this interest has lead to a significant investment in the automation of designing these circuits thus culminating in the wide acceptance of a unified design methodology supported by widely available CAD tools. Along this evolution, the semiconductor industry has kept improving fabrication processes by shrinking silicon features to attain larger scale of integration. New

manufacturing techniques have made the integration of multi-million transistors onto the same die possible. As designers kept packing more devices into chips to take advantage of these large scales of integration, significant challenges have emerged the most important of which is the reliance on a clock signal to orchestrate logic operations across an entire chip. This challenge is known as the *clocking problem*. Today, this problem is considered at the root of three consequential fundamental obstacles in current VLSI design:

- (i) *Design cycle time*: Design time can be extended significantly by unexpected clocking problems. These extensions can disturb product schedules and shrink potential market profits.
- (ii) *Power budget*: The power budget allocated for a design initially may be completely underestimated if clocking problems are not addressed early in the design cycle. Even if they are, there is still no guarantee that the power budget will remain within initial estimates.
- (iii) *Chip area*: To overcome the technical difficulties imposed by the distribution of the clock to different parts of a chip, substantial silicon area has to be sacrificed to support this distribution. As known in the economics of the semiconductor industry, area cost can add up to the fixed cost of producing each chip unit.

### **1.1.2 Growing Importance of Clockless Circuits**

Motivated by the gravity of the clocking problem and its severe consequences, designers are currently considering other circuits which can operate without a clock.

These circuits are known as *clockless circuits* [1]. Although considered esoteric by most digital designers, these circuits have been subject to intensive investigation for some time. While clockless circuits have some disadvantages, there is wide agreement among researchers that their well known advantages make them suitable to overcome the clocking problem. Unfortunately, at this time, there is no unified established design methodology to support the design and verification of clockless circuits although a plethora of ad-hoc design methodologies have been proposed in the past for various classes of clockless circuits [2-10]. In fact, this variety of design methodologies triggered strong reluctance from digital designers to consider clockless circuits as viable alternatives. As a result, since existing CAD tools have been intended for clocked designs, it would be reasonable to adapt them for clockless designs considering the massive investments that have been spent on the developments of these tools. An ideal solution to leverage these investments would be a design methodology that would exploit existing CAD tools as much as possible and deviate from them as little as possible. Even by adopting such a methodology, one quickly realizes that only a handful of clockless circuits can be designed and verified using this adopted methodology. For instance, most of the pipeline-based clockless techniques, such as micropipelines, are not adequate to synthesize large data paths. These clockless pipelines, often implemented at circuit level, are so fine-grain that their application to pipeline data paths can have an unacceptable area overhead.

### 1.1.3 Coarse-Grain Clockless Pipelining

Faced with this difficulty, it would make sense to (i) either select a few coarse-grain pipelining techniques among previously proposed clockless techniques, or (ii) propose new coarse-grain clockless pipelining techniques that seem supportable by existing CAD tools. A few attempts have been already undertaken in pursuing the former alternative [2, 11-12]. However, if the latter alternative is pursued, the best place to transform a clocked design into a clockless one is at the gate level where minimum disruption of the design flow supported by existing CAD tools is achievable. By doing so, the synthesis step of the clocked gate netlist from the initial register transfer level (RTL) model in the design flow is completely preserved. The obtained clockless gate netlist can be mapped using technology mappers packaged in existing CAD tools, and standard cell libraries which do not contain any specially designed handshaking components. In addition, the same gate netlist can be simulated using any existing simulators. Furthermore, the proposed clockless design technique is of sufficient granularity as to not impose a high area overhead.

Based on the rationale of the second alternative, this dissertation presents a novel clockless design technique highly adaptable for existing CAD tools. This technique can be incorporated within existing CAD tools without altering their design flow. This design technique is used to develop three coarse-grain pipelining techniques with distinct control mechanisms, which can be used to transform a clocked gate netlist into a highly pipelined clockless gate netlist based on data rate and area cost specifications. The remainder of this chapter reviews the design methodology of clocked circuits in section 1.2. In section

1.3, a review of the limitations of clocked circuits is presented while section 1.4 introduces several classes of clockless circuits. Section 1.5 gives an overview of the design methodologies used in synthesizing clockless circuits while section 1.6 presents the contributions of this dissertation. Finally, section 1.7 shows an overview of the dissertation.

## **1.2 Design Methodology in Clocked Circuits**

Today, the design methodology of clocked circuits is widely accepted and supported by existing CAD tools. As shown in Figure 1.1, this methodology consists of (i) specification and modeling, (ii) verification, (iii) synthesis, (iv) technology mapping, and (v) physical layout.

### **1.2.1 Specification and Modeling**

A clocked circuit is specified in both general and specific terms that provide design targets such as functionality, speed, and size. These specifications are used to create an abstract, high level model using a high level hardware description language (HDL). The abstract model contains information on behavior of each block and the interaction among the blocks in the circuit [13]. VHDL and Verilog HDL are the most widely used HDLs in the digital design community to model these circuits.

### **1.2.2 Verification**

The HDL model is subjected to extensive verification wherein the design is checked to ensure correct functionality. Simulation is the most widely used form of verification. During simulation, test benches are created and applied on the design to validate its functionality against initial requirements.

### **1.2.3 Synthesis**

The synthesis step consists of creating a gate implementation of the specified model. This step can be performed as follows:

- (i) Translate the abstract Register Transfer Level (RTL) design description to register elements and combinational logic.
- (ii) Optimize the combinational logic by minimizing, flattening and factoring the resulting Boolean equations.
- (iii) Translate the optimized logic level description to a generic gate netlist using cells from a generic library.



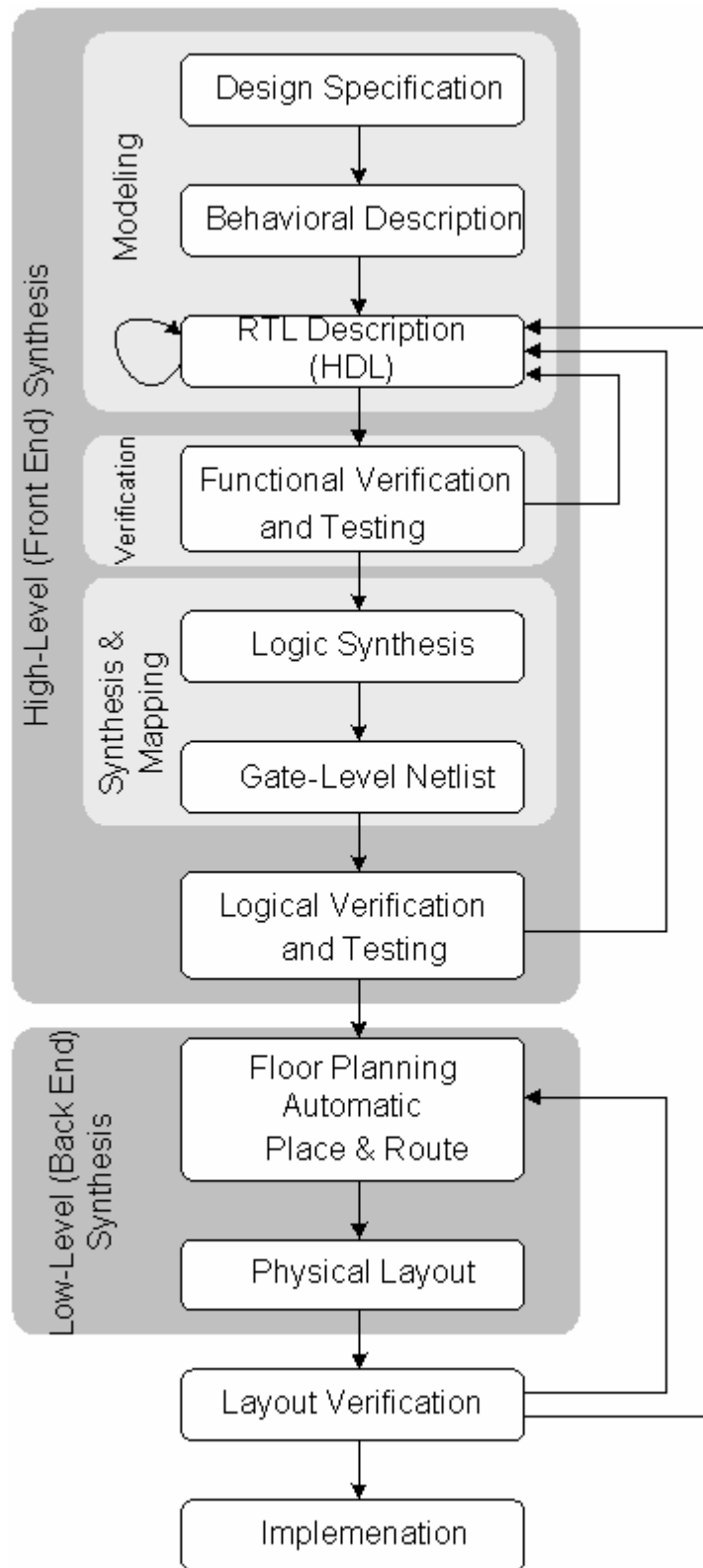


Figure 1.1: Design flow of clocked circuits [14].

### **1.2.4 Mapping**

In this step, the optimized generic gate netlist is mapped to a specific standard cell library in a given technology. The mapping must satisfy area and timing constraints specified earlier in the design flow. After mapping, simulation can be performed on the mapped gate netlist in order to compare its results with the results obtained from the simulation of the model specified in the modeling step of the design flow.

## **1.3 Limitations of Clocked Circuits**

In nanometer technology processes, circuit designers can build super fast transistors capable of processing data in several steps during the time it takes a wire to carry a signal from one side of a chip to the other [15]. Keeping operation frequency identical across the chip area requires substantial effort in distributing the clock to the various areas of the chip. To do so, the clock distribution at chip level can generate numerous costly difficulties that exacerbate the three fundamental obstacles encountered in current VLSI design.

### **1.3.1 Clock Frequency**

To insure correct synchronization of the latches across the chip, designers assume a clock frequency based on the worst case propagation delay through the slowest path in the design [16-18]. This pessimistic estimation usually accounts for maximum clock skew, and process variations due to process, voltage, and temperature. The margin allotted for these variations tend to increase as nanometer processes are adopted.

### **1.3.2 Timing Closure**

For high performance designs, timing closure can become a major bottleneck before tape out time. In principle, the delay through the slowest paths, augmented with the safety margins accounting for process variation and skew, should be less than the target clock period. Often, design teams realize that the target is not met after layout in spite of the extensive simulations at different levels of the design flow. Designers are forced then to iterate numerous times through the design flow cycle in order to meet the target. These iterations can cause costly delays in production schedules.

### **1.3.3 Power Implications**

It has been established that the clock network can consume a sizable portion of the chip power budget. This phenomenon is highly acute in high-capacity Field Programmable Gate Arrays (FPGA) chips [19]. In fact, the clock is continually switching unless clock gating techniques are used. This means that latches are dissipating heat whether they are processing data or not [17]. Clock gating techniques can alleviate the problem to some degree at the expense of added design complexity and a drop in performance [20, 21].

### **1.3.4 Area Implications**

Several researchers have proposed advanced approaches to alleviate clock distribution and de-skewing problems. However these approaches can, in some cases, impose a substantial cost in added area. There were reported instances in which a

complex clock-driver network on a commercial microprocessor was designed to keep the clock skew within 300 picoseconds. However, this resulted in a circuit that occupied 10% of the chip area and consumed more than 40% of the total power budget [22]. In semi-custom designs, the clock network can occupy an area that is even larger reaching 30 to 40% of the total chip area.

### **1.3.5 Noise Margins**

Beside increases in area and power cost, clock networks are highly noisy. By keeping all signal transitions in lock step, the clock network creates the worst environment to suppress noise. Similar to data signals, current transitions on clock lines become synchronized to some degree thus maximizing the AC component of any generated noise in relation to the harmonics of the clock frequency.

### **1.3.6 Multiple Clock Domains**

With increasing numbers of clock domains in current chips, concerns are growing about synchronization of cross-domain signal paths. A complex communication system-on-chip can contain up to 300 such domains [23]. The synchronization of these domains requires proper placement of synchronizers at precise points in the design. Even with proper placement, there is no guarantee that all the bits of a domain-crossing bit vector in a cross-domain path signal will cross domain boundaries at the same time through these synchronizers. This is further complicated by the fact that clock frequencies across domains differ widely, which necessitates the insertion of FIFO buffers at various points in the design. Insertion of FIFO buffers raises sizing and correct

implementation issues. While synchronizer placement and buffer insertion can be performed manually, it is not advised in most cases since it is labor-intensive and highly prone to errors. Designers can overcome these difficulties if automatic tools for placement and verification were available. Unfortunately, there are no specialized tools to support these tasks on the market at this time.

### **1.4 Clockless Circuits**

Clockless circuits are circuits that operate without the synchronization of a clock signal. Although numerous clockless circuits have been proposed before, they can nevertheless be classified based on a limited number of characteristics. The most important characteristic in distinguishing the underlying principle behind the operation of a clockless circuit is how signal delay is handled in order to insure the proper synchronization of the different components of the circuit. This assumption is known as the *delay model*. Based on this delay model, clockless circuits can be divided in five distinct classes of circuits.

#### **1.4.1 Self-Clocked Circuits**

In general, *self-clocked (SC) circuits* consist of three components as shown in Figure 1.2 [24] [25]:

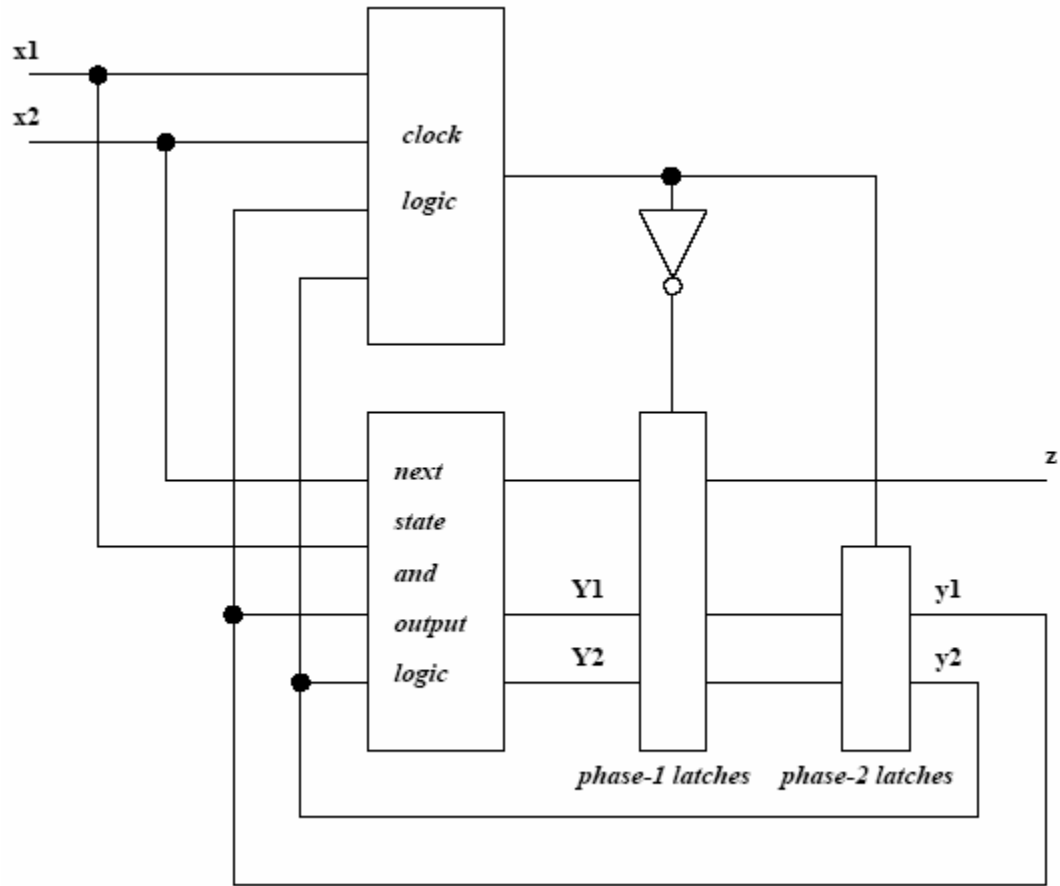


Figure 1.2: General architecture of SC circuits [25].

- (i) Clock logic: This component generates a clock pulse only whenever the state or output signals change. It is used to eliminate hazards and control state changes of the machine.
- (ii) Storage elements: These elements capture data by responding to the clock signal.
- (iii) Combinational logic: This component does not require special care to protect it from hazards. The clock component is chosen to be slow enough to allow outputs to settle before it is fed back to the combinational logic.

SC circuits are simple and attractive since they reduce the potential overhead due to the clock. In addition, they allow the realization of hazard-free logic based on the specification of finite state machines.

#### **1.4.2 Speed-Independent Circuits**

*Speed-independent (SI) circuits* were introduced by David Muller in the 1950s [26]. These circuits operate correctly regardless of gate delays. In these circuits, wires are assumed to have zero or negligible delay. As a result, every fork in the circuit is assumed to be an isochronic fork causing only a negligible skew. An isochronic fork is a wire fork in which the delays on the branches of the fork are equal. If this delay model is assumed, an SI circuit works properly for all possible ordering of events associated with all possible and varying relative delays of the components of the circuit. SI circuits can be synthesized from Petri nets and signal transition graphs used in synthesizing clockless circuits.

#### **1.4.3 Delay-Insensitive Circuits**

*Delay-Insensitive (DI) circuits* are circuits which operate correctly with positive and unbounded delay in wires and gates [27]. In a *bounded-delay model*, it is assumed that a circuit will settle in a stable state as a response to an input if given enough time. Immediately after, a new input can be safely fed to the circuit. Micropipelines and burst mode circuits are examples of circuits whose operations are based on the bounded-delay model. However, in an *unbounded delay model*, no matter how long a circuit waits, there is no guarantee that the input will be properly absorbed. This required some kind of

handshaking protocol between sender and receiver components of the circuit. The sender sends data and waits on an acknowledge signal from the receiver. The latter receives the data and sends the acknowledge signal back to the sender. By managing these signal exchanges, handshaking protocols can make circuits highly immune to hazards. Unfortunately, the number of DI circuits, built out of simple gates and operators, is quite small. In fact, it has been proven that almost no useful DI circuits can be built if one is restricted to a class of simple gates and operators [28]. However, many practical DI circuits can be built if one allows more complex components [29].

Because the unbounded delay model is too restrictive, it can be slightly relaxed by allowing bounded delays on wire forks or using isochronic forks. In adopting this modified unbounded delay model, DI circuits can be refined further into a subclass of *quasi-delay-insensitive (QDI) circuits* [27]. In contrast to QDI circuits, delays on the different fork branches of DI circuits are completely independent and may vary considerably. DI circuits can be built from Null Convention Logic, handshake-based circuits, and extended delay insensitive clockless models.

#### **1.4.4 Self-Timed Circuits**

In [30], *self-timed (ST) circuits* are described as interconnections of parts called “elements”. Each element is contained in an “equipotential region” in which wires have negligible or well-bounded delay. An element itself may be an SI circuit, or a circuit whose correct operation relies on the use of local timing assumptions. However, no timing assumptions are made on the communication between regions; that is,



communication between regions is delay-insensitive. Null Convention Logic (NCL) is considered as ST circuits.

#### **1.4.5 Self-Resetting Circuits**

Earlier implementations of self-resetting circuits rely on circuit techniques to realize self-resetting behavior. For instance, *self-resetting CMOS* (SRCMOS) operates on signals represented as short-duration pulses rather than as voltage levels [31]. After a logic gate processes a set of input pulses, a reset signal is activated to restore the logic gate to a state in which it is ready to receive another set of input pulses. The input pulses must arrive at the same time and must overlap with one another for a minimum duration. Several reset schemes have been proposed before. Jung has proposed two techniques to increase the robustness and efficiency of SRCMOS circuits [32]. The first technique uses a logical structure to properly sequence the reset and evaluates modes of an SRCMOS logic stage without having to rely on a timing chain. The second technique uses a pulse stretcher so that input pulses of widely different arrival times can be properly combined at a given stage logic. Beside Jung schemes, Dooply has proposed locally self resetting CMOS where the reset signal for each stage is generated locally [33]. This technique is based on single-rail domino logic stages in which the reset signal is obtained by sending the stage own output through a short delay chain.

### **1.5 Design Methodology in Clockless Circuits**

Various design methodologies have been proposed in the past to synthesize clockless circuits. In general, there is a close relationship between the theoretical model

used to represent the behavior of clockless circuits and the tools used to model this behavior. Given this relationship, design methodologies for clockless circuits can be classified as follows:

- (i) *Graph-based methodologies*: These methodologies require the modeling of the circuit as *Petri nets* (PNs) or *signal transition graphs* (STGs). Circuits are synthesized from these graphs and mapped onto general C-elements and complex gates [3-8].
- (ii) *HDL-based methodologies*: These methodologies require the modeling of a circuit using an existing HDL [2, 9-10], [34-37]. The model is translated to a netlist that can be mapped onto a standard cell library.
- (iii) *Script-based methodologies*: In these methodologies, circuit behavior is described using algebraic expressions and saved as scripts [38-41]. The scripts are expanded into graphs from which circuits are synthesized and mapped.
- (iv) *Compilation-based methodologies*: These methodologies require the use of high level programming languages, some of which are proprietary, designed to express concurrency, handshaking, and sequencing [42-45]. The source code of the program describing the circuit behavior is parsed and compiled into a circuit containing pre-designed components which support the programming language constructs for concurrency, handshaking, and sequencing.

Given this diversity of design methodologies, it is understandable why most designers are reluctant to delve in clockless logic. This reluctance can be justified by the

fact that adopting any methodology requires some amount of retraining and retooling on the part of the designers. This reluctance is reinforced further by a visible lack of simulation and verification tools at all levels of the design flow that is suggested by these methodologies. In addition, proprietary cell libraries are necessary to map circuits using some of these methodologies. What most designers are seeking instead is a single uniform design methodology that is (i) familiar, (ii) widely accepted, (iii) tested and proven by a long usage experience, (iv) and may use proprietary resources as little as possible. Such a methodology has been already in use for some time to produce clocked circuits in the form of successful commercial CAD or EDA tools. In this case, the design methodology of these CAD tools can be used to support clockless design techniques that can be specified and modeled using current HDLs. The obtained HDL models can be verified through simulation. Next, the HDL models can be synthesized into clockless gate netlist which can be mapped using standard cell libraries found in the realization of clocked circuits. Note that, in general, these libraries do not contain any special cells designed to handle events specific to clockless logic such as concurrency, rendez-vous, and handshaking. By implementing these clockless techniques using existing CAD tools with a minimum modification to the design flow of these tools, the need for relearning and retooling can be eliminated.

## **1.6 Contributions of the Dissertation**

This dissertation presents a new clockless design technique suitable for existing CAD tools. Specifically, its contributions are as follows:

- (i) A new self-resetting logic technique, called *self-resetting stage logic* (SRSL), in which the computation of a block is reset periodically from inside the block. This automatic self-resetting behavior manifests itself in the form of a periodic oscillation of the block driven by a reset loop similar to an internal clock. This simplifies the synchronization scheme by using a uni-directional communication channel between senders and receivers.
- (ii) A pipelining technique based on SRSL controlled at stage level, called *stage-controlled self-resetting stage logic* (S-SRSL). In S-SRSL, the control of the communication between stages is performed between each pair of stages.
- (iii) A pipelining technique based on SRSL controlled at pipeline level, called *pipeline-controlled self-resetting stage logic* (P-SRSL). In P-SRSL, the control of the communication between stages is performed by the last stage in the pipeline whereby the oscillation of the last stage drives the oscillatory behavior of the other stages in the pipeline.
- (iv) A coarse-grain pipelining technique called *delay-tolerant self-resetting stage logic* (D-SRSL) that is similar to S-SRSL pipelining where data flow across stages is orchestrated by each pair of neighboring stages. Whereas S-SRSL and P-SRSL pipelines require that intra-stage delay and communication scheme be identical and uni-directional respectively, D-SRSL can tolerate stages with arbitrarily different delays by using a bi-directional communication scheme.
- (v) Graph-theoretic and analytical formulations of a combinatorial problem encountered in the synthesis of SRSL pipelines. Specifically, this problem

consists of synthesizing an SRSL pipeline from a gate netlist with a minimum area overhead based on a specified data rate. The analytical formulation consists primarily of an integer programming problem.

- (vi) Since the size of the integer programming problem formulation is significantly large, and subsequently solving it using analytical approaches is impractical, a new heuristic algorithm is proposed to solve it. Because latches tend to occupy a large silicon area, the main goal of the algorithm is to minimize the area occupied by inter-stage latches without violating any timing constraints. This algorithm accomplishes this by executing two successive phases where phase I assigns each gate in the gate netlist to a specific pipeline stage whereas phase II minimizes the number of inter-stage latches between every pair of neighboring pipeline stages.

## **1.7 Overview of the Dissertation**

This dissertation consists of six chapters beside the current chapter. Chapter 2 reviews the main clockless design methodologies and evaluates their suitability for existing CAD tools. Chapter 3 explains the underlying concepts behind SRSL and introduces S-SRSL pipelines followed by an analysis of the experimental results conducted on these pipelines. Chapter 4 presents P-SRSL pipelines and the experiments conducted on these pipelines followed by a comparison of S-SRSL and P-SRSL pipelines. Chapter 5 presents D-SRSL pipelines and analyzes the results obtained from the prototyping experiments conducted on these pipelines. Chapter 6 introduces the synthesis problem of SRSL pipelines, presents the formulation of the combinatorial

problem stemming from the synthesis of SRSL pipelines, and describes the synthesis algorithm implemented for this purpose. Finally, Chapter 7 concludes the dissertation and suggests avenues for future work.

## CHAPTER TWO: RELATED CLOCKLESS DESIGN METHODOLOGIES

This chapter reviews the main clockless design methodologies and the available tools that support each design methodology as reported in the literature. Section 2.1 presents methodologies based on Petri nets while section 2.2 presents methodologies based on signal transition graphs. Section 2.3 presents micropipelines while section 2.4 presents Null Convention Logic. Burst mode machines are described in section 2.5. Section 2.6 describes handshake circuits while section 2.7 describes the extended delay insensitive model. Finally, section 2.8 gives a summary of the chapter and compares the listed design methodologies with the proposed SRS� technique.

### 2.1 Petri Nets

*Petri Nets* (PNs) is a formal syntax and semantic representation suitable to specify causality, concurrency and choice between events. PNs can be a powerful tool to model clockless circuits [3, 46, 47]. Formally, a PN is a triple  $N = (P, T, F)$  where:

- (i)  $P$  is a finite set of places.
- (ii)  $T$  is a finite set of transitions:  $T \cap P = \emptyset$ .
- (iii)  $F : F \subseteq (P \times T) \cup (T \times P)$  is the flow function.

Transitions in PNs represent events in the system such as a request to access a memory bank in a multi processor system. On the other hand, places in PNs represent placeholders for needed resources and conditions necessary for events to occur. Figure 2.1 shows a C-element and its surrounding environment while figure 2.2 shows its PN specification.

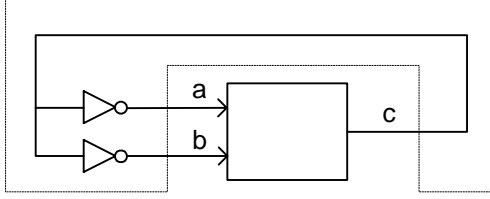


Figure 2.1: C-element and its surrounding dummy environment [27].

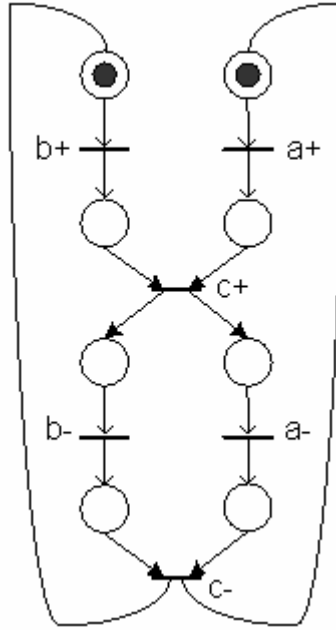


Figure 2.2: The PN of the C-element shown in Figure 2.1 [27].

The PN is marked with tokens on the input places to the  $a+$  and  $b+$  transition. The  $a+$  and  $b+$  transitions may fire in any order. The  $c+$  transition becomes enabled to fire when both  $a+$  and  $b+$  transitions fire [27]. Previously developed methods for PN-based synthesis of clockless circuits can fall in one of the following two approaches [48]:

- (i) A direct, syntax oriented, translation of the PN into logic.
- (ii) A translation of the PN into a *signal transition graph* (STG) followed by the synthesis of a circuit from the obtained STG.



The approach in (i) can be used to obtain implemented circuit in three steps. In the first step, a net model is extracted from the circuit model described in the PN while in the second step the net model is transformed into an equivalent net where each signal event is associated with a unique transition. Finally, the net is translated into the circuit that can be constructed from a standard set of event-based modules. So far, most previous research focused on the synthesis of the clockless circuits from STGs. *Petrify* belongs to category (i) of synthesis tools. It is mainly a research tool used in the synthesis of clockless controllers from PN specifications [4]. *Petrify* reads a specification PN and generates a reduced version of the initial PN where the latter is used to produce an optimized netlist of a clockless controller based on a target gate library. Recent improvements to *Petrify* consist of generating circuits from STGs instead of solely PN specifications. These improvements help *Petrify* fall in category (ii) of synthesis tools. As shown in Figure 2.3, *Petrify* can be used as a standalone synthesis tool. The design flow shown in the figure starts from a specification of the system behavior described by a PN, state graph, or finite state machine (FSM) in a textual format. *Petrify* performs logic synthesis on the construction state graph in which each reachable state is assigned a binary code representing the value of each signal. This allows the generation of a circuit using logic minimization techniques. The circuit can be constructed from C-elements and generic complex gates. If these generic complex gates are not available in the gate library, *Petrify* performs combinational and sequential decomposition of the logic into primitive gates that are available in the target gate library. The PNs accepted by *Petrify* can also be interpreted as behavior-specifying STGs of clockless controllers.

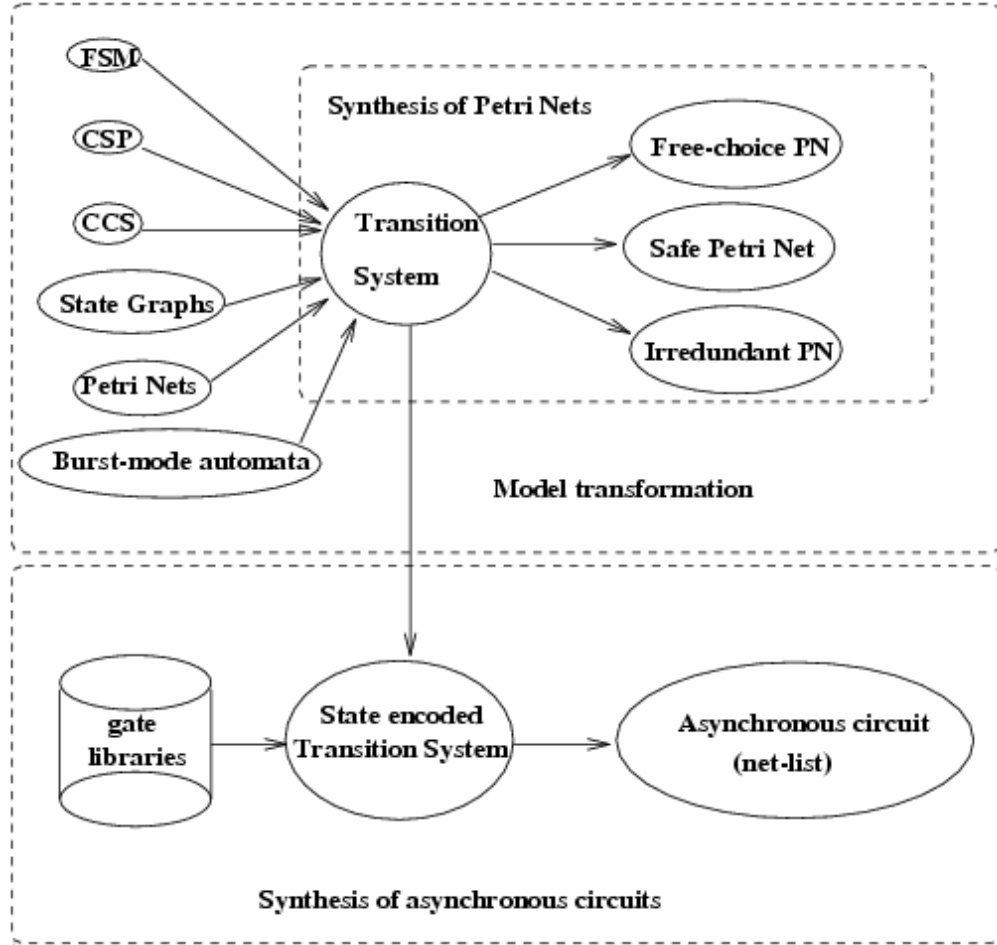


Figure 2.3: Petrify framework.

Since Petrify uses symbolic techniques to represent the state space, it can synthesize large controllers whose specifications consist of more than 20 signals if well-structured behavior is specified. However, previous experiments showed that Petrify is not appropriate for data-path synthesis since it cannot always guarantee a correct synthesized netlist [5]. Although Petrify starts its synthesis process from a PN or STG specification, the latter two representations are not widely used among digital designers. Specifying system behavior in these representations can be challenging if the designer does not have proper knowledge on how to use them. In addition, it is difficult to integrate Petrify with existing simulation and synthesis tools since it is intended to

operate as a stand-alone tool. Furthermore, Petrify does not offer any capability to support verification or simulation of the PNs or STGs before they are synthesized into circuits. Petrify support mapping the synthesized netlist to C-element and complex gates assuming that the target cell library contains such elements and gates.

## **2.2 Signal Transition Graphs**

Signal transition graphs (STGs) are a subset of PNs. When PNs are used to model clockless circuits, it is sometimes necessary to relate transitions to events on signal wires [6-8]. Several PN variants have been proposed to relate these transition events including M-nets, I-nets, change diagrams, and STGs. An STG is a labeled safe PN which is modeled as a 7-tuple  $(P, T, F, M_0, N, s_0, \lambda_T)$ , where:

- (i)  $P, T, F$  are defined in the PN section.
- (ii)  $M_0$  is the initial marking representing the function that maps the places to natural numbers.
- (iii)  $N = I \cup O$  is the set of signals where  $I$  is the set of the input signals and  $O$  is the set of the output signals
- (iv)  $s_0$  is the initial value for each signal in the initial state.
- (v)  $\lambda_T : T \rightarrow N \times \{+, -\}$  is the transition labeling function.

In an STG, each transition is labeled with a rising transition,  $s+$ , or a falling transition,  $s-$ . An  $s+$  label indicates that the transition corresponds to a  $0 \rightarrow 1$  transition on the signal wire  $s$ . On the other hand, an  $s-$  label indicates that the transition corresponds to a  $1 \rightarrow 0$  transition on  $s$ . Figure 2.5 shows the STG specification for the C-

element shown in Figure 2.1. This specification can be directly derived by following the causality arrows defined in the timing diagram shown in Figure 2.4 [27].

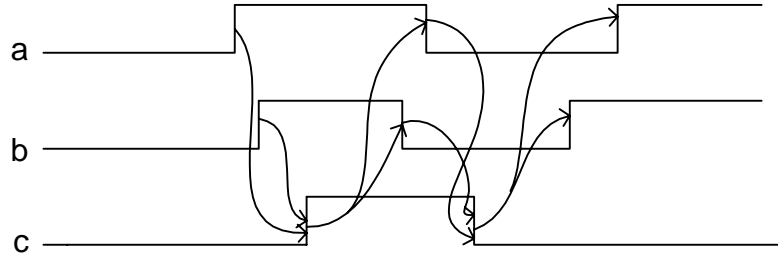


Figure 2.4 :Timing diagram of the C-element shown in Figure 2.1.

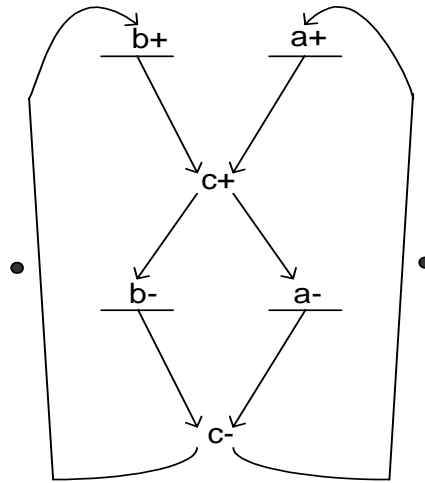


Figure 2.5: STG of the C-element shown in Figure 2.1.

The first step in STG-based synthesis is the generation of a *stage graph* (SG). After obtaining an SG, there are two approaches to implement a circuit. In the first approach, if the SG is free of *complete state coding* (CSC) violations, a Boolean equation is derived and directly implemented with an SI circuit using generalized C-elements. A CSC violation represents the situation in which different states of a state machine are encoded with the same binary code although they imply contradictory next values for at least one of the output signals. However, in the second approach, specific state encoding

methods are applied to get a realizable STG. The Boolean equation of the newly obtained realizable STG can be used to realize a circuit directly using generalized C-elements. In general, the derived Boolean equation may not be implementable as a single complex gate. In that case, logic decomposition is applied to transform the equation into smaller equations, which can be implemented using simple gates. Figure 2.6 shows the design flow to synthesize clockless circuit from STGs [47]. Petrify can be used to synthesize circuits from STG specifications and support mapping the synthesized netlist to C-elements and complex gates.

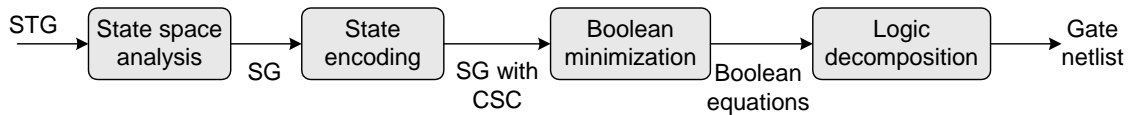
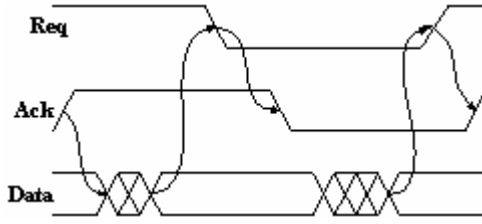


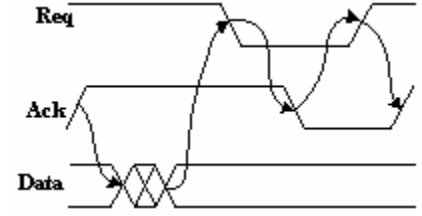
Figure 2.6: Synthesis flow of clockless circuits from STG specifications.

### **2.3 Micropipelines**

Micropipelines consist of event-driven elastic pipelines [49]. These pipelines can realize fast and efficient implementations of arithmetic circuits by using a two-phase handshake protocol instead of a four-phase handshake protocol. Both protocols are shown in Figure 2.7. The implementation structure for a micropipeline is the controlled first-in first-out (FIFO) queue, shown in Figure 2.8, in which the gates labeled C are Muller C-elements. In addition, the registers in the Figure 2.8 are level-sensitive latches that respond to transitions on two inputs instead of responding to a single clock wire as is done in clocked latches.



(a) Two-phase handshake protocol.



(b) Four-phase handshake protocol.

Figure 2.7: Micropipeline handshake protocols.

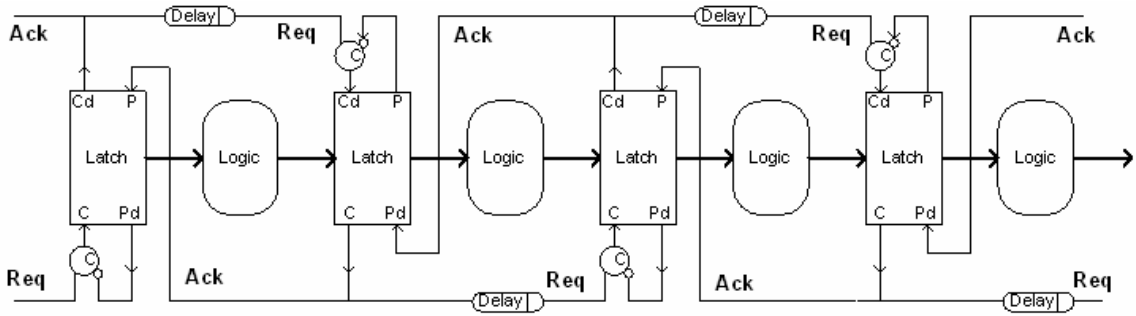


Figure 2.8: Basic structure of a micropipeline.

These latches are initially active by passing data directly from data inputs to data outputs. When a transition occurs on the *capture* wire of the latch, labeled C, data is no longer allowed to pass, and the current value of the outputs is statically maintained. Once a transition occurs on the *pass* input, labeled P, data is again allowed to pass from input to output, and the cycle repeats. The Cd and Pd ports on the latch simply keep copies of the control signals that are delayed so that the register completes its response to the control signal transitions before they are sent back out.

*Pipefitter* has been proposed as a tool for automated synthesis of micropipelined clockless circuits consisting of a 4-phase control unit and a clockless data path with matched delays [9, 10]. The synthesized control unit supports concurrency, sequencing

and choice. As shown in Figure 2.9, Pipefitter’s framework uses Verilog HDL as the output format for intermediate representations of both control unit and data path.

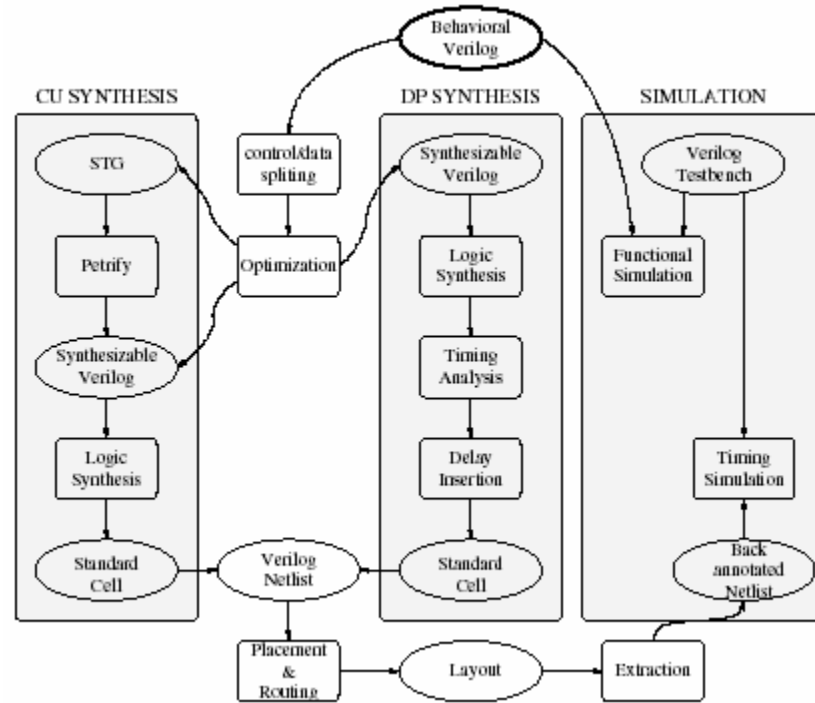


Figure 2.9: Pipefitter framework.

Based on this representation, designers can use existing EDA tools for most design phases, including synthesis, simulation and layout. The Verilog source code is optimized and split into two separated databases: one for the control unit and the other for the data path. After the Verilog netlist is generated, a standard logic synthesis tool can be used for technology mapping. Pipefitter can automatically generate a netlist of matched delays for each block in the data path. In addition, it can generate the netlist of the control unit by calling Petrify. By merging the netlist of both control unit and data path, a complete netlist is constructed. At this point, the netlist can be placed and routed in order to produce a final layout. While Pipefitter can be integrated with existing EDA tools better than Petrify can, its shortcomings stems from the fact that it supports only a

restricted subset of Verilog statements. Pipefitter uses existing EDA commercial tools for simulation, and the final standard netlist can be mapped to a standard cell library.

## **2.4 Null Convention Logic**

The *NULL Convention Logic* (NCL) synthesis flow is a framework that integrates data transformation and control into a single expression thus yielding delay-insensitive circuits [50]. NCL uses threshold gates with hysteresis to provide the basic building block of NCL designs. Threshold gate inputs and outputs can be in one of two states, DATA or NULL. DATA corresponds to a logic-1 voltage level while NULL corresponds to a logic-0 voltage level in the normal logic mapping [34, 36]. The operation of NCL gates is based on two primary properties of  $M$ -of- $N$  gate, namely *threshold behavior* and *hysteresis behavior*. Threshold behavior requires that the output becomes DATA if at least  $M$  of the  $N$  inputs are DATA. On the other hand, hysteresis behavior requires that the output changes only after a sufficiently complete set of input values have been established. In the case of a transition to DATA, the output remains at NULL until at least  $M$  of the  $N$  inputs become DATA. On the other hand, in the case of a transition to NULL, the output remains at DATA until all  $N$  inputs become NULL. Since these gates use two-value logic, as traditional Boolean logic does, they can be constructed with traditional CMOS, Bipolar, or even more exotic processes [36]. Figure 2.10 shows a 2-of-3 threshold gate that fires when two of its inputs are active and return to null when all of the inputs are null.



Figure 2.11 shows a half adder circuit in Boolean logic with its clock while Figure 2.12 shows its NCL counterpart [37].

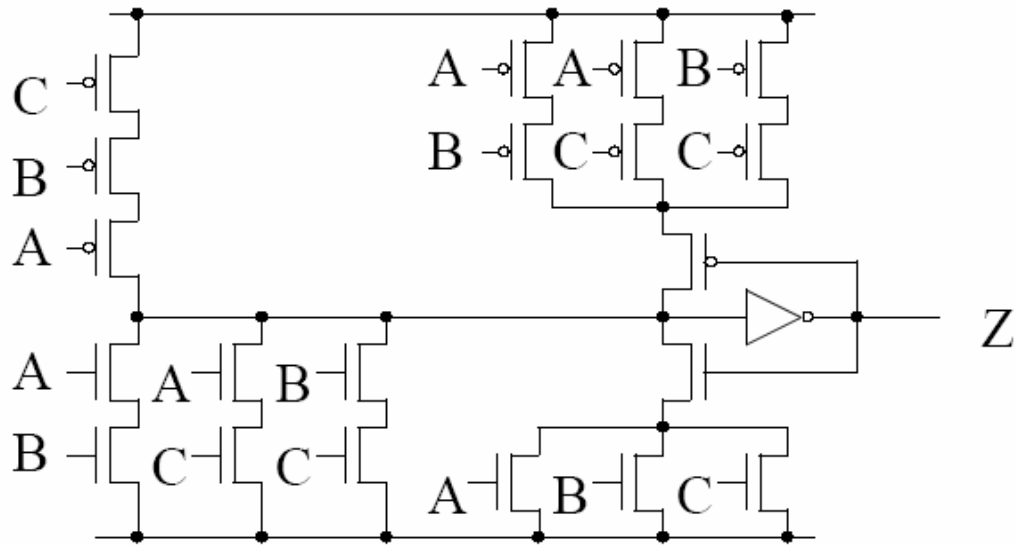


Figure 2.10: NCL 2-of-3 threshold gate.

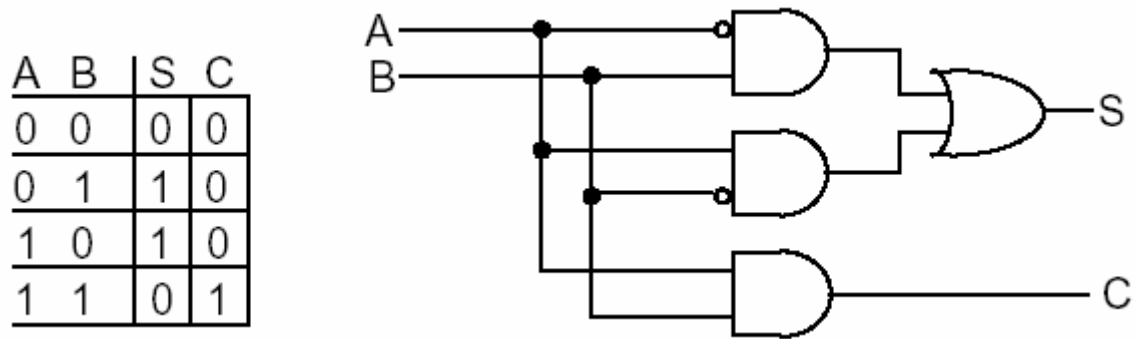


Figure 2.11: A half adder circuit in conventional Boolean logic.

Although NCL can use any delay insensitive encoding, it uses mostly a dual rail one-hot encoding in which the presence of DATA on one of two wires indicates a TRUE state while the presence of DATA on both wires indicates a FALSE state. Unlike previously described clockless approaches, the algebraic theory behind NCL makes it extremely applicable to high-level design methodologies such as RTL simulation, RTL

synthesis, and gate optimization. Motivated by these advantages, Theseus Logic has developed a synthesis and simulation flow based on existing, off-the-shelf, EDA tools from industry leaders such as Synopsys and Mentor Graphics. Based on this flow, NCL designers can specify their designs in VHDL or Verilog and simulate them using existing EDA tools. As shown in Figure 2.13, the NCL flow is centered around two primary synthesis steps [51, 52]:

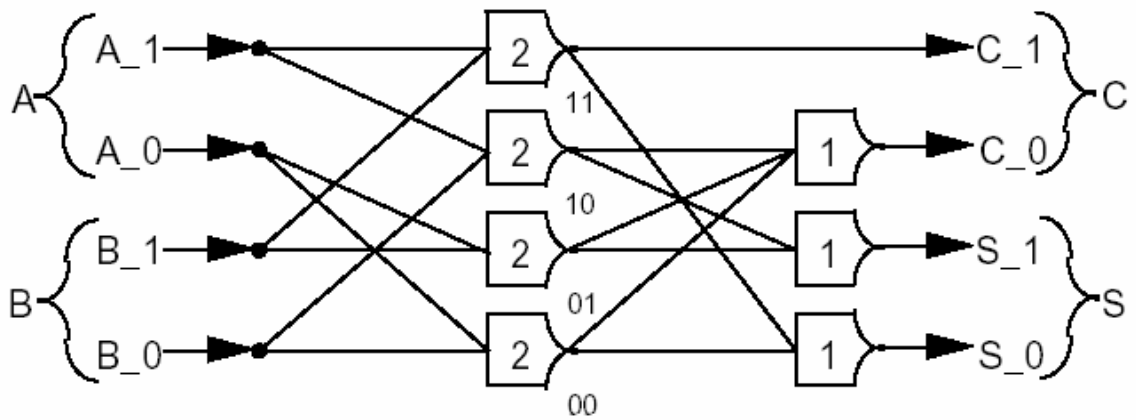


Figure 2.12: NCL half adder circuit.

- (i) Translate the HDL code into a 3NCL netlist: This stage starts with an HDL source code written with 3NCL, a single-rail multi-valued representation of the initial NCL. The synthesis tool performs HDL optimizations and outputs an unmapped VHDL dataflow description expressed by AND and INV assignments. This dataflow description is referred to as a 3NCL netlist.
- (ii) Optimize the 3NCL netlist into a 2NCL netlist: the second stage expands the intermediate 3NCL netlist into a fully dual-rail 2NCL netlist by overloading all AND and INV assignment as Delay-Insensitive Minterm Synthesis (DIMS) dual rail type assignments. This expansion is described in a VHDL

package. At this point, multilevel minimization of Boolean networks, available in existing CAD tools, can be performed if an NCL target library is available.

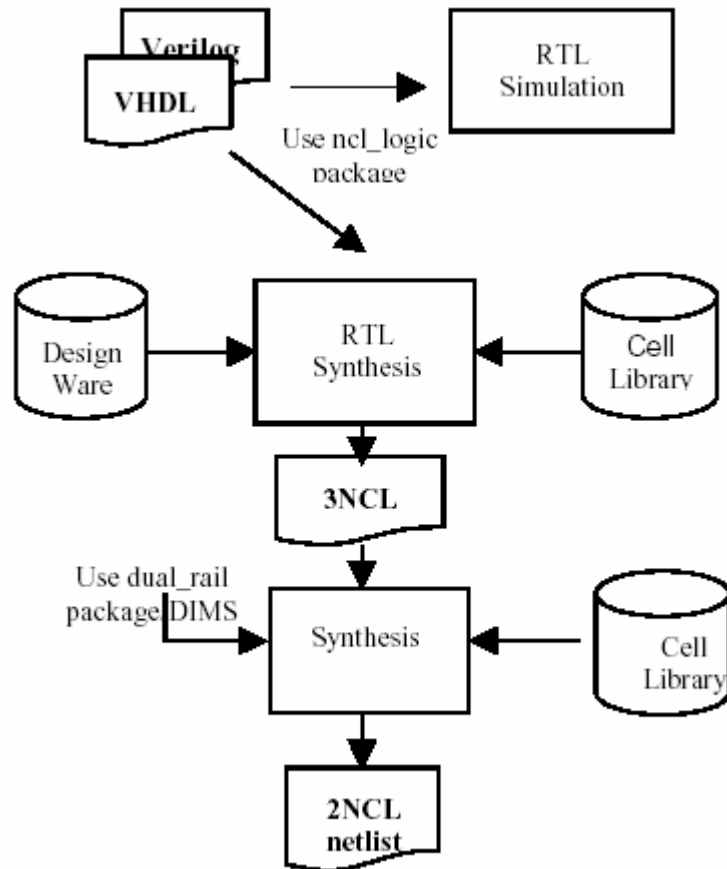


Figure 2.13: RTL flow for NCL design [51].

While threshold and hysteresis properties provide NCL with advantages that are not available in other clockless methodologies, they remain responsible for some of its disadvantages [5]:

- (i) By using existing synthesis tools, the area of some NCL designs can be sometimes two to three times larger than the area of the same designs synthesized in clocked logic.

- (ii) The throughput of NCL designs may suffer unless heavy pipelining is used which may result in an area increase.
- (iii) Experimentation shows that straightforward translations of clocked logic to NCL designs results in a substantial increase in power consumption.
- (iv) Although existing synthesis tools can be used to implement NCL designs, proprietary libraries owned by Theseus are necessary to map the synthesized designs onto library cells if a high quality implementation is desired.

To remedy the problem described in (i), synthesis tools tailored to NCL logic may be necessary. However, this would defeat the advantage of leveraging the investment spent on existing synthesis tools.

## **2.5 Burst Mode Machine**

When in a stable state, a burst-mode circuit waits for a set of input signals to change in arbitrary order. After this input burst has completed, the machine computes a burst of output signals and new values of internal variables. The surrounding environment is not allowed to change a new input burst until the circuit has completely reacted to the previous burst [44, 45, 53]. Figure 2.14 shows an example of burst mode circuit [27].

Burst-mode circuits are specified using state graphs similar to those used in the design of clocked circuits. Several tools for synthesizing burst-mode controllers have been previously developed primarily in academia. *MINIMALIST*, developed at Columbia University, is a CAD package for synthesis, optimization and verification of burst-mode

controllers [54]. The focus of the package is on technology-independent synthesis. *MINIMALIST* includes a number of highly-optimized algorithms for state minimization, optimal state assignment, two-level hazard-free logic minimization, synthesis of generalized C-element implementations, and verification. The latter is achieved by using a simulation environment to verify the modeled burst mode machines. The synthesized implementations are hazard-free gate-level circuits consisting of two-level AND-OR networks and generalized-C elements. These circuits can then be technology-mapped using existing technology mapping tools. To support this functionality, *MINIMALIST* provides a graphical display to show specifications and implementations, an interactive shell, design scripts, help menus, and a tutorial.

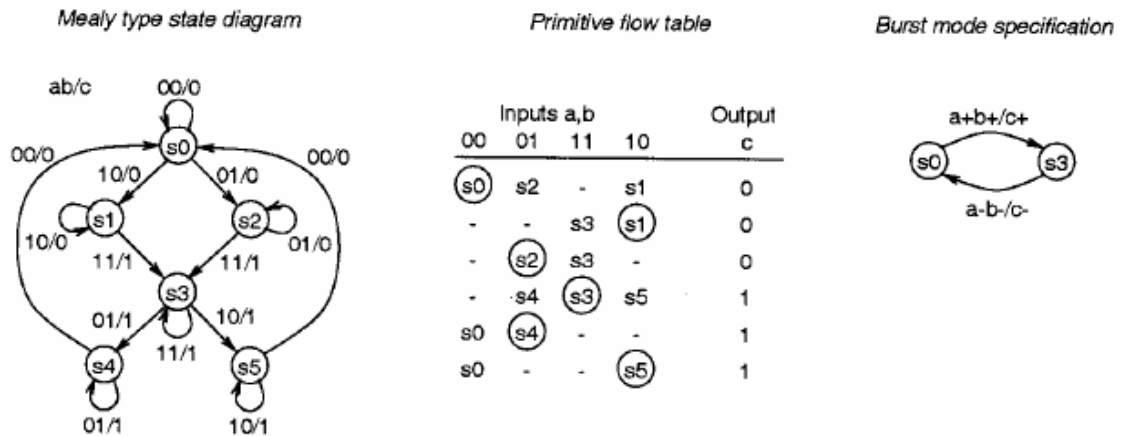


Figure 2.14: Burst mode specification of a C-element.

Beside *MINIMALIST*, *3D*, developed at University of California, is a synthesis package which uses the *extended burst-mode* (XBM) model [55]. The XBM design style covers a wide spectrum of sequential circuits ranging from DI to clocked circuits. *3D* can synthesize multiple-input change clockless finite state machines in addition to numerous

circuits that fall in the area between clocked and clockless logic. These circuits are difficult and sometimes impossible to synthesize automatically using existing methods. 3D synthesizes XBM controllers in two-level AND-OR networks, and maps these networks to a generic CMOS standard cell library or generalized C-elements. Both tools do not offer any HDL front-end interface. As a result, a designer can interact with these tools only in two modes: using prepared design scripts or typing individual commands.

## **2.6 Handshake Circuits**

An alternative to clockless finite-state machines that communicate using fundamental mode or burst-mode has been proposed as handshaking circuits. Figure 2.15 shows a handshake channel, which is a point-to-point connection between an active and a passive block.

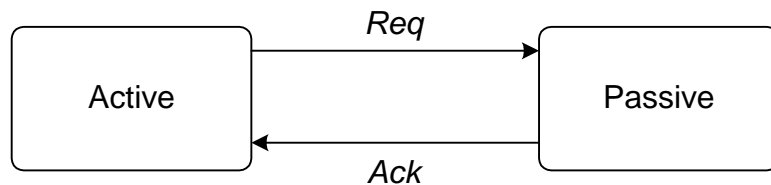


Figure 2.15: Handshake channel.

This approach requires that both blocks be connected by two wires: a request (*Req*) and an acknowledge (*Ack*) wire. A handshake is initiated by the active block, which starts by sending a signal via *Req*, and waits until a signal arrives via *Ack*. After a request arrives to the passive block, this block sends an acknowledge [27, 42, 56-58]. Most clockless circuits use a four-phase handshake protocol. This protocol consists of a channel which starts in a state where both *Req* and *Ack* are low. The active block starts a

handshake by making *Req* high. When the passive block receives *Req*, it sets *Ack* to high. A return-to-zero cycle follows, during which *Req* and *Ack* go low thus returning to the initial state.

To support this handshake methodology, the *Tangram* toolset has been proposed [43]. As shown in Figure 2.16, a design can be specified in Tangram, which is a programming language, similar to the C language, extended to include constructs that support concurrency and communication.

In fact, Tangram has language constructs which support blocks sharing and waiting for clock-like edges. A compiler translates Tangram programs into handshake circuits, which are netlists composed from a library of some 40 handshake components. Each handshake component implements a language construct, such sequencing, communication, and sharing. Packaged with the compiler, the handshake circuit simulator and performance analyzer give the designer feedback about the design function, area, timing, and power of the synthesized circuit. The process of mapping the handshake circuit using a conventional standard cell library can be done in two steps. In the first step, the component expander uses the component library to generate an abstract netlist of combinational logic, registers, and clockless cells, such as Muller C-elements. This step also determines the encoding of data and handshake protocol. In the second step, a commercial synthesis tool and technology mapper can be used to generate the cell netlist. Today, Tangram is considered one of the most complete toolset used to design medium size clockless integrated circuits. Besides being a proprietary toolset, designers will have

to endure the learning curve of a new programming language if they are interested in using the Tangram toolset.

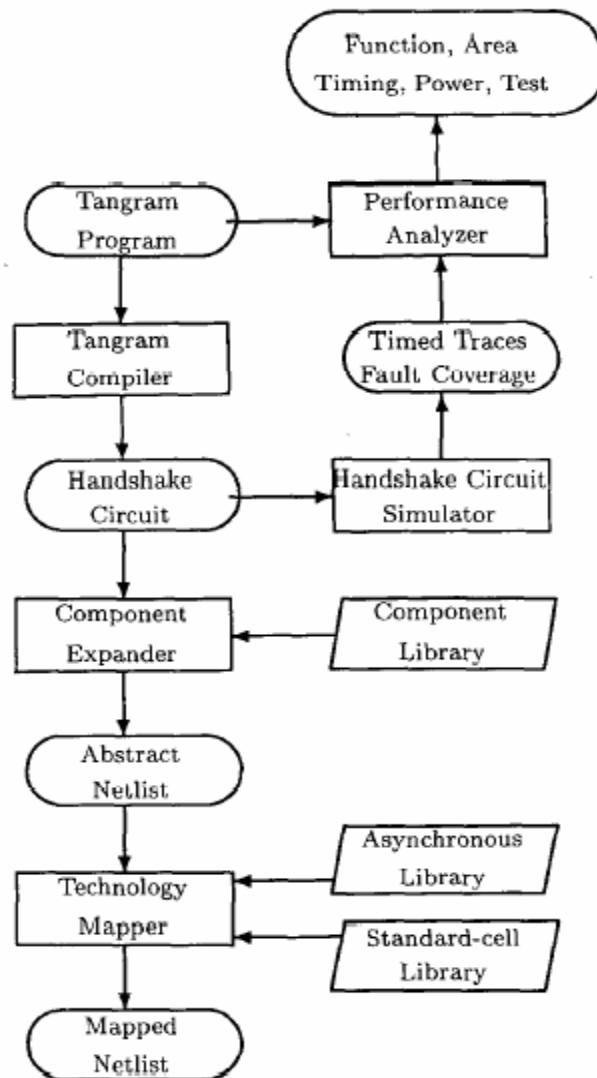


Figure 2.16: The Tangram Toolset.



## **2.7 Extended Delay Insensitive Model**

The *eXtended model for Delay-Insensitive systems* (XDI) is a theoretical framework used to define the external structure and observable behavior of DI systems. Besides being state-based, the framework includes refinement or satisfaction relations and composition operators. The XDI model specifies the conditions and the rules to implement a DI circuit from initial specifications by taking in consideration the expression of progress requirements for the circuit and its environment. XDI transforms these specification to DI-algebra first, and then to a state graph that is expressed in AND/IF-notation [38-40]. A handful of tools such *Digg* and *Ludwig* have been proposed to automate the refinement process [40]. *Digg* transforms a DI-algebra specification into XDI automata. DI-algebra specification can be expressed as recursive DI-algebraic expression while XDI automata can be represented as AND/IF graphs. After *Digg*'s transformation, *Ludwig* can analyze and synthesize the obtained state graph into a circuit. A major shortcoming of this design methodology is the absence of simulation and mapping tools based on existing cell libraries.

## **2.8 Summary**

This chapter presents a review of previously proposed clockless design methodologies. Although there are different design methodologies, none can be easily integrated in a complete design flow using existing CAD tools without significant modifications to the design flow. While designs in some design methodologies cannot be modeled using existing HDLs, others cannot be simulated using existing simulators. In addition, some design methodologies requires special synthesizers and mappers which

target special-purpose cell libraries. In contrast to these methodologies, the SRSL design technique is sufficiently flexible to be supported by existing CAD tools. SRSL can be modeled using existing HDLs, simulated using existing simulators, synthesized using existing synthesis compilers, and mapped using existing technology mappers. At the end, a pipelined SRSL netlist is produced which can be placed and routed using existing physical layout tools granted that design constraints are propagated from synthesis to layout tools. Table 2.1 shows a summary of the design methodologies.

Table 2.1: Summary of clockless design methodologies.

| Design Methodology         | Modeling                                     | Verification             | Synthesis   | Mapping   | Features   |
|----------------------------|--|--------------------------|---|---|--|
| <b>PN or STG</b>           | PN or STG                                    | Not supported            | Petrify as a stand alone tool   | Generalized C-elements and complex gates using a standard cell library  | Synthesis of large controllers<br>Not suitable for data path synthesis   |
| <b>Micropipeline</b>       | Verilog                                      | Off-the-shelf simulators | Petrify to synthesize the control unit<br>Pipefitter to synthesize data paths | Based on a standard cell library  | Separates the control unit from data path<br>Supports only a subset of Verilog statements  |
| <b>NCL</b>                 | VHDL   | Off-the-shelf simulators | Off-the-shelf synthesis tools   | In principle, a standard cell library can be used.<br>NCL design flow supports NCL proprietary cell libraries for high quality implementations. | Translates HDL to 3NCL and optimize the 3NCL into a 2NCL netlist<br>Increases area and power, and may degrade throughput.                  |
| <b>Burst Mode Circuits</b> | Design scripts<br>Command-driven interaction | MINIMALIST verification  | MINIMALIST<br>3D  | Generalized C-elements and complex gates using a standard cell library  | No HDL front-end interface   |
| <b>Handshake Circuits</b>  | Tangram language                             | Handshake simulator      | Off-the-shelf synthesis tools   | Handshake component library in addition to a standard cell library  | Compiler translates the program into handshake circuits.<br>Component expander uses the component library to generate an abstract netlist. |
| <b>XDI Model</b>           | DI algebra                                   | Not supported            | Ludwig synthesis tool   | Not supported   | DI algebra is translated to a state graph which can be synthesized with Ludwig.  |
| <b>SRSL</b>                | VHDL or Verilog                              | Off-the-shelf simulators | Off-the-shelf synthesis tools   | Standard cell library   | Transforms a gate netlist into a pipelined SRSL netlist<br>Suitable for data path and control  |

## CHAPTER THREE: STAGE-CONTROLLED SELF-RESETTING STAGE LOGIC PIPELINES

This chapter presents the concept of *self-resetting stage logic* (SRSL) and shows how it can be used as a building block in linear and non-linear pipelines. Section 3.1 introduces SRSL while section 3.2 describes how SRSL can be used in a linear pipeline controlled at stage level. Section 3.3 explains how SRSL can be used in a non-linear pipeline while section 3.4 presents a detailed timing analysis of a linear pipeline. Section 3.5 describes the implementation of a prototype pipeline while section 3.6 summarizes the chapter.

### 3.1 SRSL

In SRSL, a stage consists of two networks: a *reset network* and a *combinational network*. In Figure 3.1, the reset network consists of a NOR gate whose output  $O$  feeds one of its inputs. The other input is tied to a reset line. As long as the reset input is asserted,  $O$  remains 0. When the reset is de-asserted,  $O$  oscillates from 0 to 1 and vice versa. The oscillation frequency is controlled by the delay  $\Delta$  embedded in the loop between the NOR output and its input. When  $O$  is 0, the reset network is in the *reset* phase. Later, when  $O$  switches to 1, the reset network is in the *evaluate* phase. As such, a reset network can oscillate between phases in an autonomous fashion. The period of the reset network consists of the two phases: reset and evaluate. Based on this oscillation, a reset network can be embedded in a pipeline stage forcing the stage to oscillate between two phases. This oscillation can be used to synchronize data transfer between

neighboring stages in a pipeline. In fact, a stage is ready to accept inputs from the preceding stage when it is in the reset phase, and ready to produce outputs to the following stage when it is in the evaluate phase.

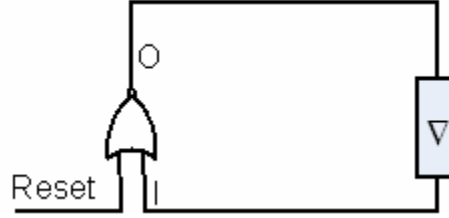


Figure 3.1: Reset and evaluate network of an SRSI stage.

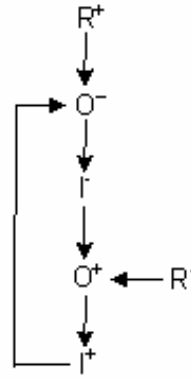


Figure 3.2: STG of the reset network shown in Figure 3.1.

Figure 3.2 shows the signal transition graph (STG) of the reset network shown in Figure 3.1 where the signals in the STG are labeled identically to the signals in Figure 3.1. In an STG, a node  $v$ , labeled  $v^+$ , represents a rising transition on signal  $v$  while the same node, labeled  $v^-$ , represents a falling transition on signal  $v$ . On the other hand, an edge going from node  $u$  to node  $v$  means that transition on signal  $u$  precedes in time the transition on signal  $v$ .

### 3.2 S-SRSL Linear Pipelines

Figure 3.3 shows the interconnection structure of a four-stage S-SRSL pipeline where each stage consists of a combinational and a reset network while Figure 3.4 shows the STG of the S-SRSL pipeline shown in Figure 3.3. Data flows from one stage to another through a latch in the linear pipeline. To insure proper data flow across stages, data is transferred from the current stage to the next one if the current stage is in the evaluate phase while the next stage is in the reset phase. Hence, the latch separating both stages is enabled when the left stage is in the evaluate while the right stage is in the reset phase [59-61].

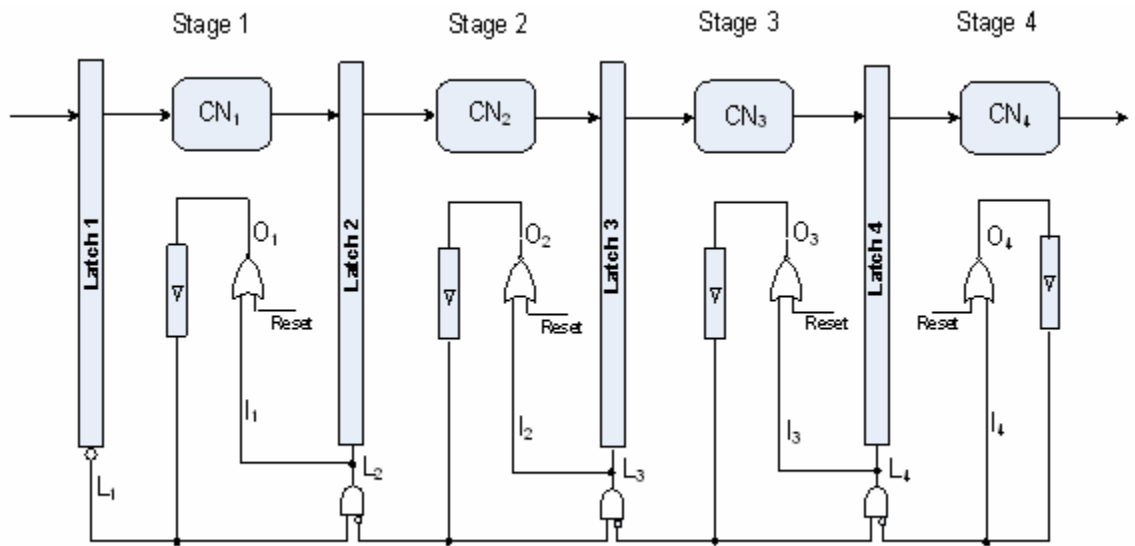


Figure 3.3: A four-stage S-SRSL pipeline.

The enable signal ( $L_i$ ) is the output of the AND gate that triggers the latch. For each latch, the inputs of the AND gate consists of the outputs of the NOR gates of the reset network in the current stage and the following stage. As a result, the synchronization of the entire pipeline depends on the communication between each pair

of neighboring stages. This locally controlled pipeline is called *stage-controlled self-resetting stage logic pipeline (S-SRSL)*. While a stage is accepting input, its reset network enters the reset phase ( $O = 0$ ), which disables the latch on its right side.

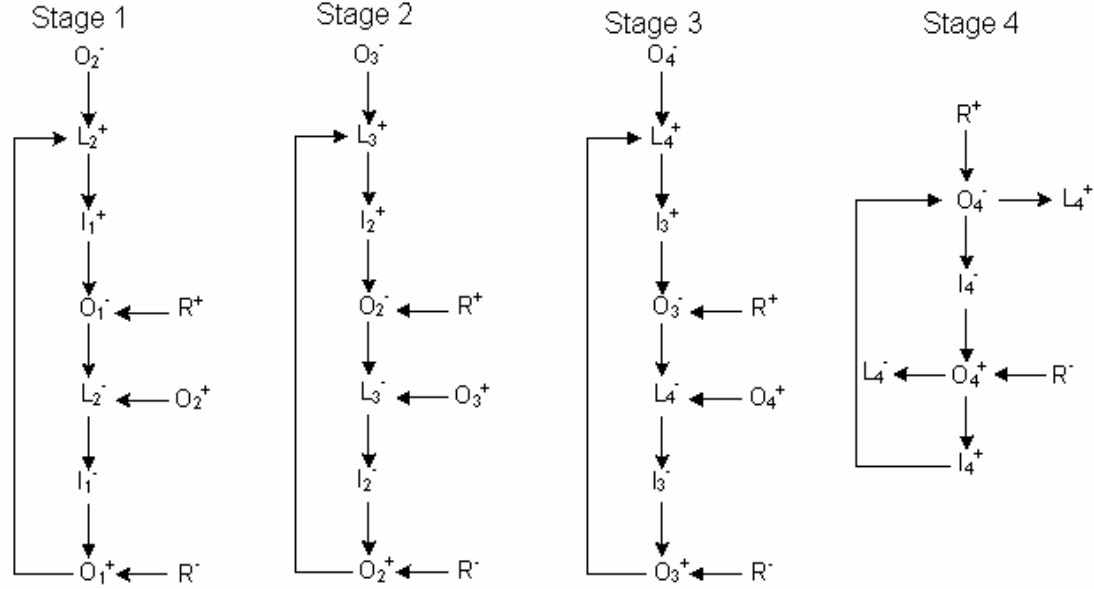
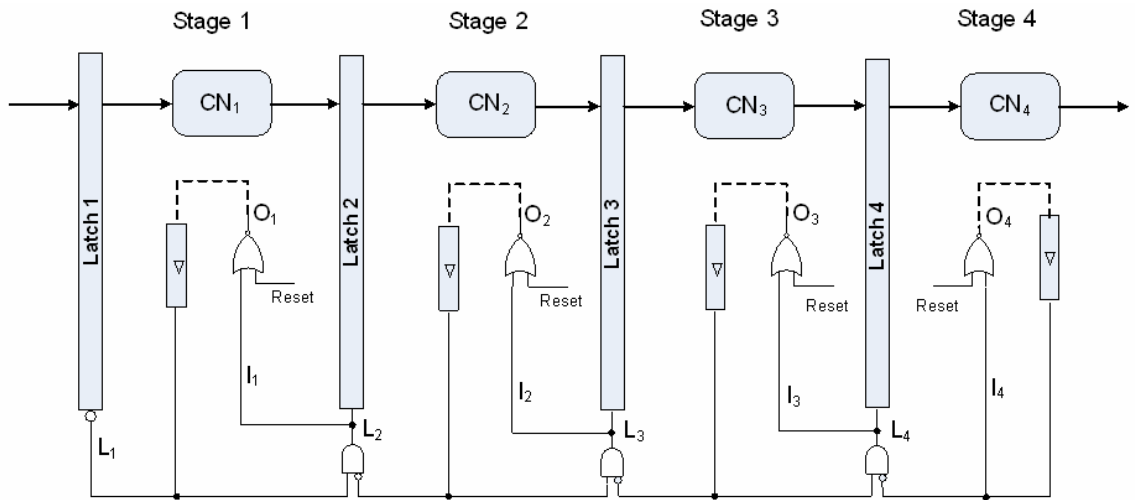


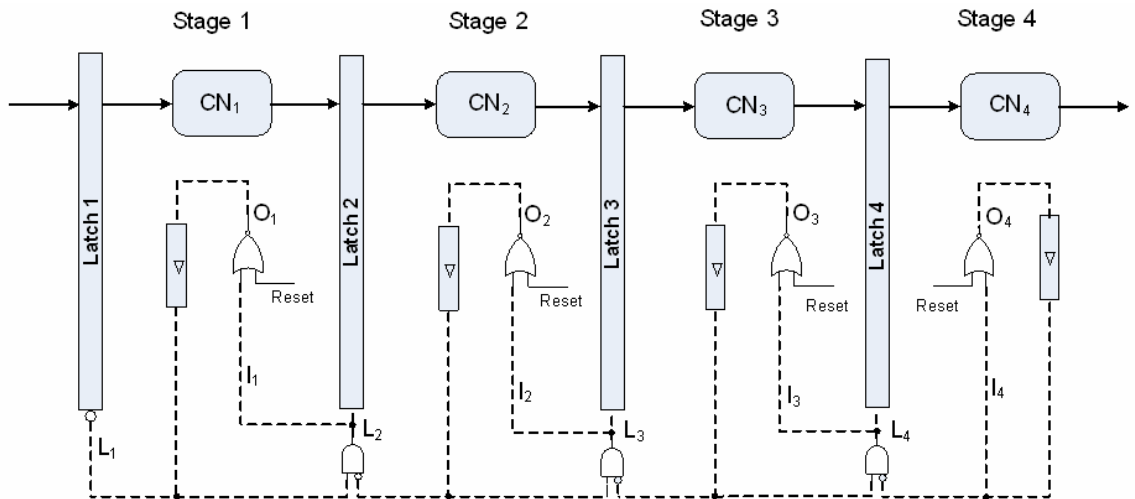
Figure 3.4: STG of the S-SRSL pipeline shown in Figure 3.3.

At any cycle, the latch on the left side of a stage in the reset phase will be enabled while the latch on its right side will be disabled. The latter will be enabled only when the stage enters its evaluate phase. As a result, during every cycle, every other stage will be in the reset phase while the remaining stages will be in the evaluate phase. A cycle later, the stages that were in the reset phase start their evaluate phases while the stages that were in the evaluate phase start their reset phases. In Figure 3.4, the STG shows that the rising transition of  $L_3$  occurs after  $O_2$  and  $O_3$  experience a rising and falling transition respectively. This means that latch 3 is enabled only when stage 2 is in the evaluate phase while stage 3 is in the reset phase. If  $O_3$  experiences a falling transition, this forces another falling transition on  $L_4$ . This shows that while latch 3 is enabled, latch 4 is

disabled. Figure 3.5 shows how the stages alternate between phases as data flows across the pipeline by representing asserted and de-asserted signals as solid and dashed lines respectively.

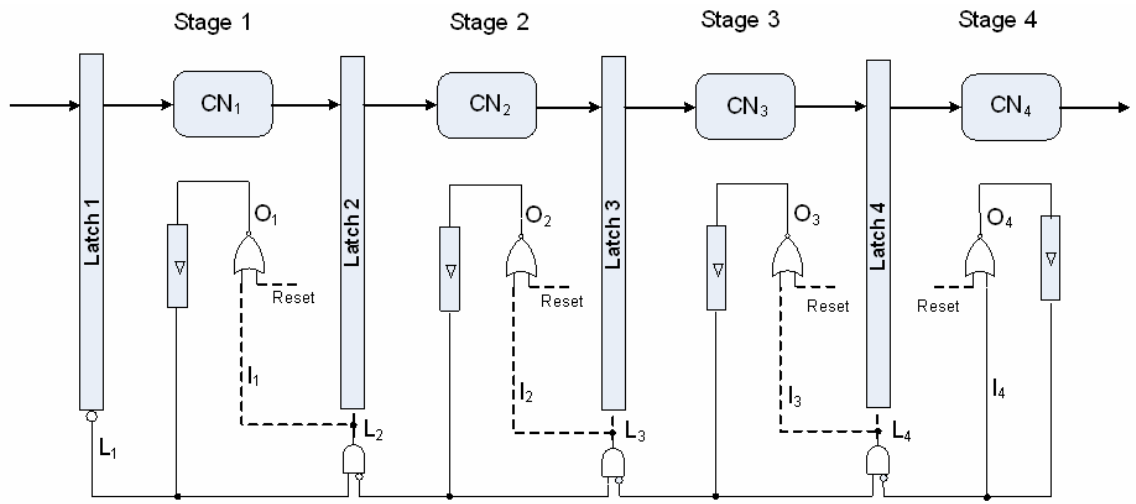


3.5(a): Assertion of the stage reset signals.

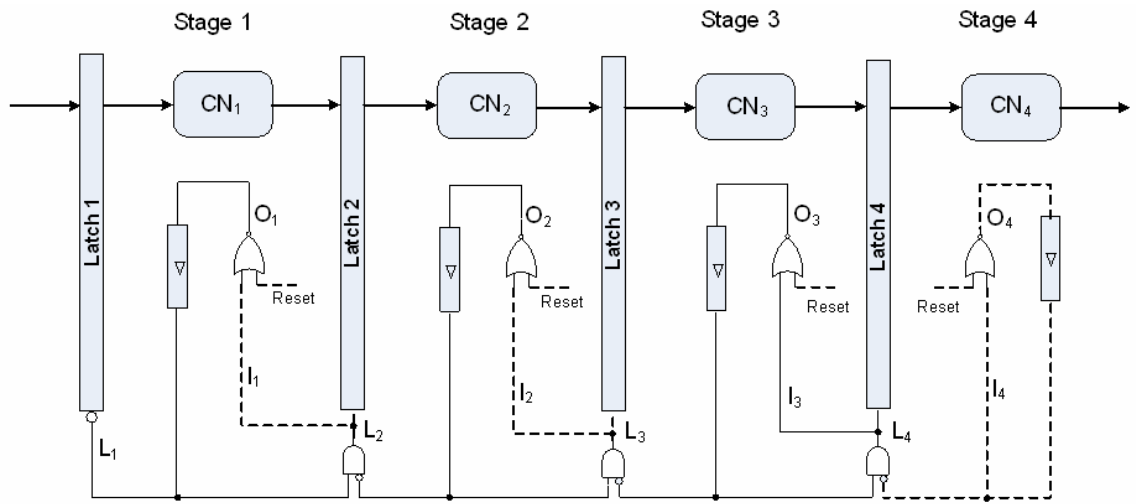


3.5(b): Reset phase of all stages.

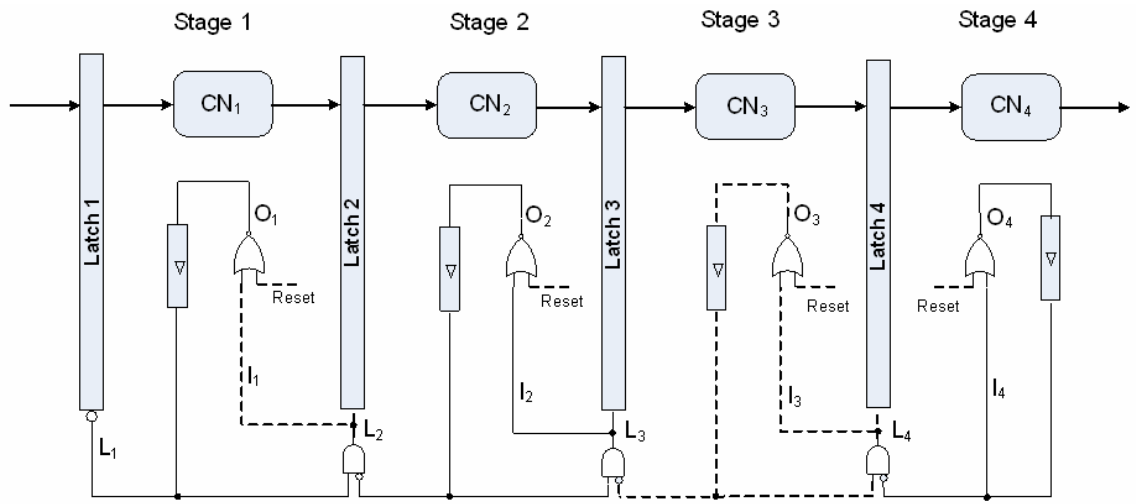




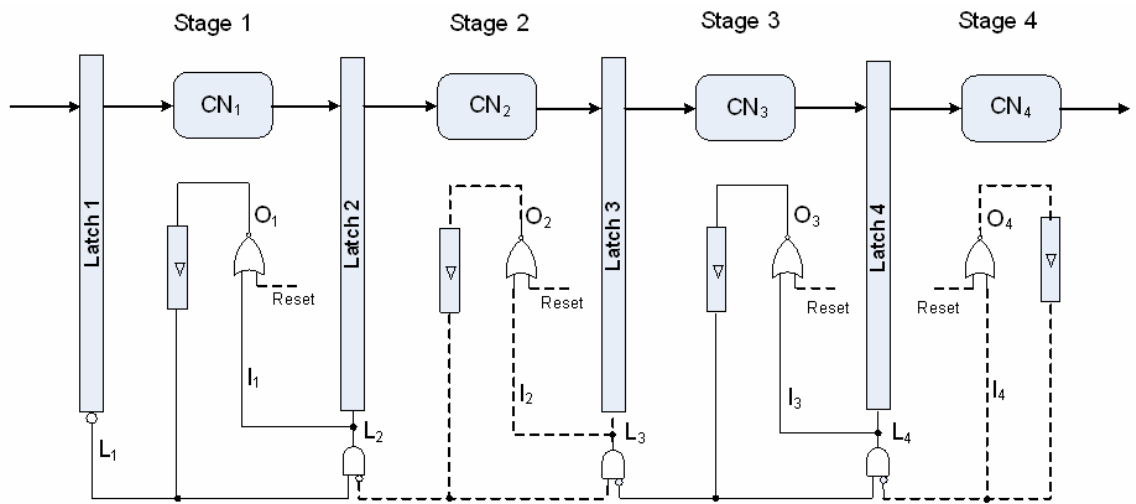
3.5(c): Evaluate phase of stage 4.



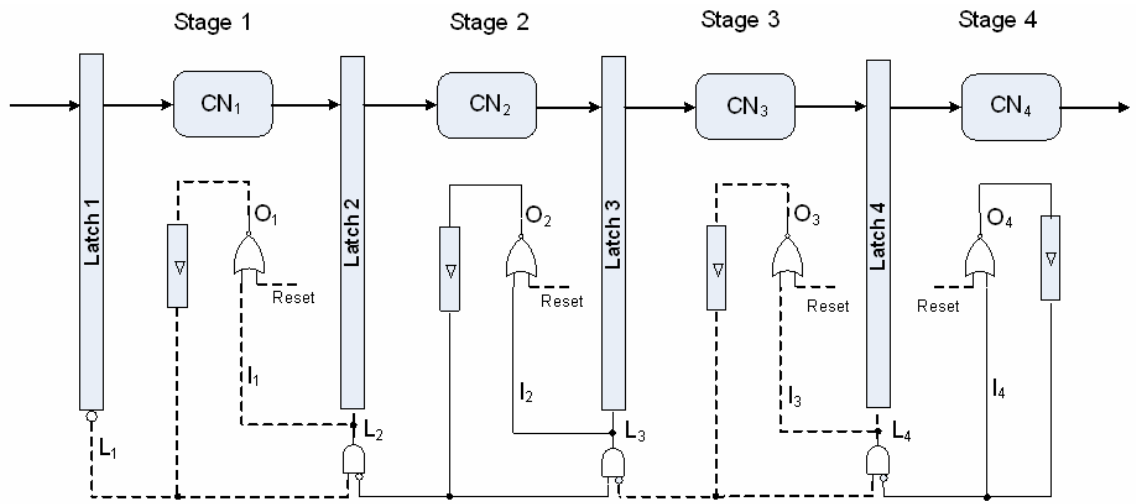
3.5(d): Evaluate phase of stage 3.



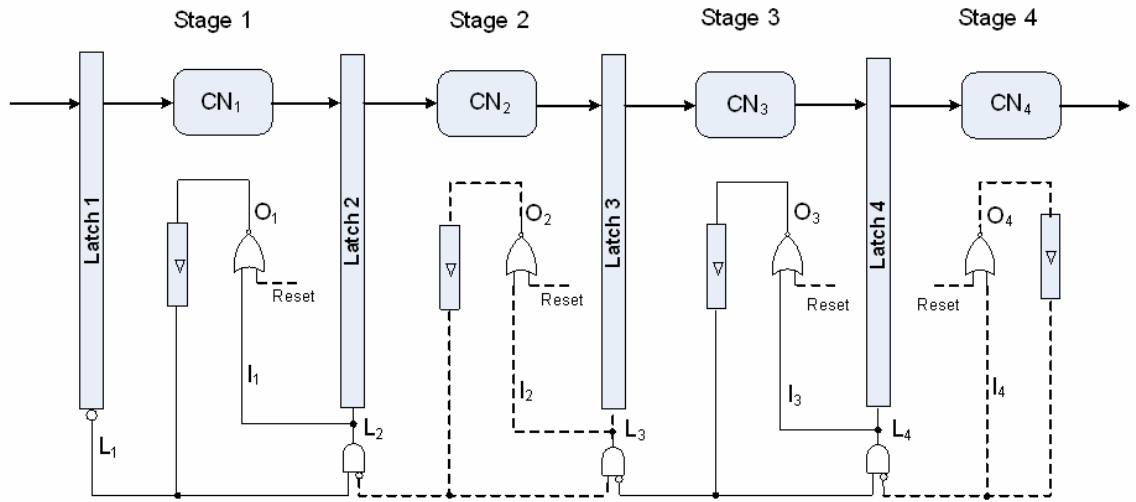
3.5(e): Evaluate phase of stage 2 and 4.



3.5(f): Evaluate phase of stage 1 and 3.



3.5(g): Evaluate phase of stage 2 and 4.



3.5(h): Evaluate phase of stage 1 and 3.

Figure 3.5: Two execution cycles of a four-stage S-SRSL Pipeline.

### **3.3 S-SRSL Non-linear Pipelines**

While linear pipelines can be used in many applications, complex systems require data to flow in divergent and convergent directions. Such systems can be realized as non linear pipelines [62-64]. To support divergence and convergence of data flow, primitives such as the *fork* and *join* operations have to be incorporated in the pipeline.

#### **3.3.1 S-SRSL Join Operation**

Figure 3.6 shows an S-SRSL join pipeline. Inter-stage data flow is similar to the data flow in a linear pipeline. Data is transferred from stage A to stage C when the former is in the evaluate phase while the latter is in the reset phase. Similarly, data flows from stage B to stage C when the former is in the evaluate phase while the latter is in the reset phase. When these conditions are true, latches 3 and 4 are activated to capture the outputs of stage A and B, and feed it to the inputs of stage C. Note that completion of the evaluate phase of stage A and B depends only on the arrival of the reset phase of stage C. By limiting the interaction only between these neighboring stages, a localized communication control between stages in the join is guaranteed.

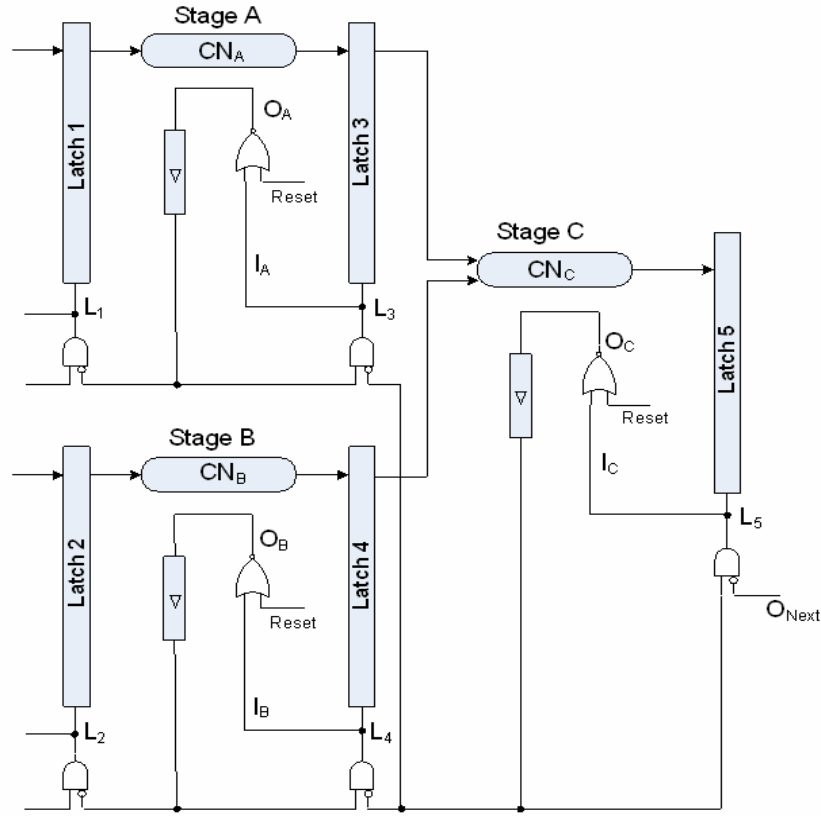


Figure 3.6: Structure of the join S-SRSL pipeline.

Figure 3.7 shows the STG of the join structure shown in Figure 3.6. In this STG, both  $L_3$  and  $L_4$  experience rising transitions when both  $O_A$  and  $O_B$  experience falling transitions while  $O_C$  experience a rising transition. This shows that latches 3 and 4 are enabled when both stages A and B are both in the evaluate phase while stage C is in the reset phase. While latches 3 and 4 are enabled latch 5 is disabled. The latter will be enabled when stage C is in the evaluate while the succeeding stage to stage C is in the reset phase.

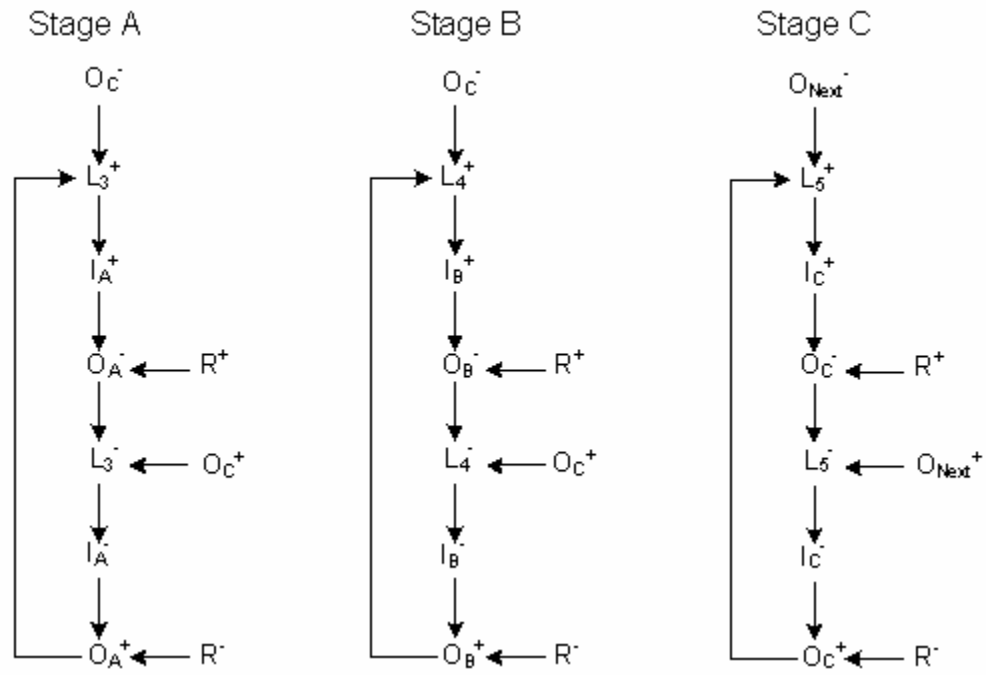


Figure 3.7: STG of the S-SRSL join pipeline shown in Figure 3.6.

### 3.3.2 S-SRSL Fork Operation

Figure 3.8 shows an S-SRSL fork pipeline. Data is transferred from stage A to stage B and C when the former is in the evaluate phase while the two latter stages are in the reset phase. When these conditions are true, latches 2 and 3 are enabled to capture the output of stage A and feed it to stages B and C. After L2 and L3 become asserted, they propagate through the gate G forcing the signal  $I_A$  to become asserted. This in turn forces signal  $O_A$  to switch to 0 at which time stage A enters its reset phase. Figure 3.9 shows the STG of the fork pipeline shown in Figure 3.8. In this STG,  $L_2$  experiences a rising transition when  $O_A$  and  $O_B$  experience a rising and a falling transition respectively. Similar observation can be made with regard to the rising transition of  $L_3$  as it relates to

the rising and falling transitions of  $O_A$  and  $O_C$  respectively. On the other hand,  $L_2$  experiences a falling transition when  $O_A$  and  $O_B$  experience a falling and rising transition respectively. While  $L_2$  and  $L_3$  experience a rising transition,  $L_4$  and  $L_5$  experience falling transitions. These two signals will experience rising transitions only when both  $O_B$  and  $O_C$  experience rising transitions while both  $O_{UpperNext}$  and  $O_{LowerNext}$  experience falling transitions.

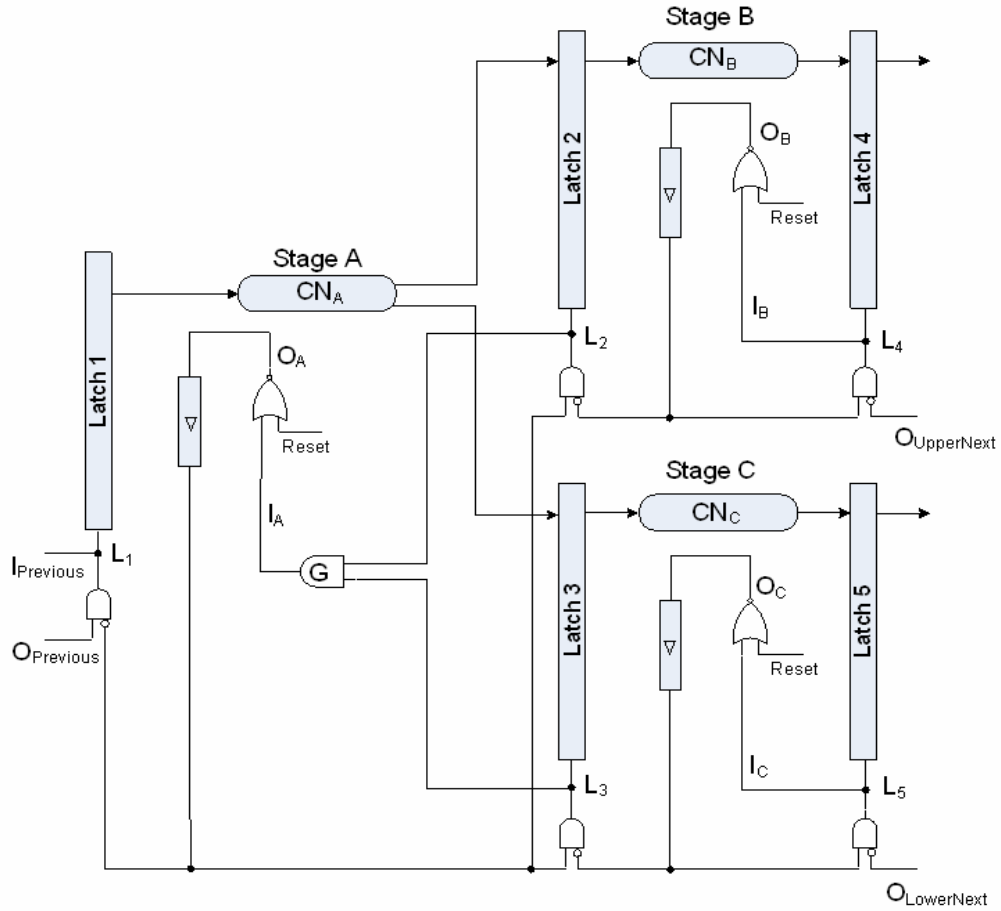


Figure 3.8: Structure of the fork S-SRSL pipeline.

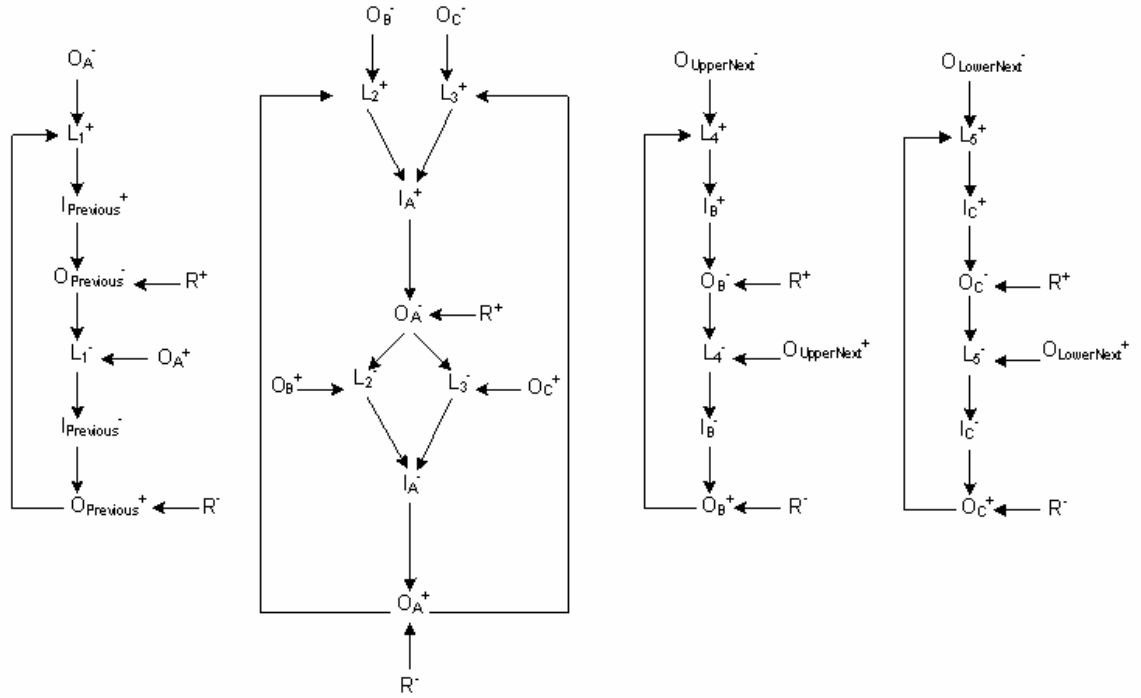


Figure 3.9: STG of the S-SRSL fork pipeline shown in Figure 3.8.

### **3.4 Performance of the Pipeline**

To explain the performance of the S-SRSL pipeline, several timing parameters are defined first. Next, these parameters are used in a signal timing analysis to characterize the performance of the pipeline.

#### **3.4.1 Parameter Definitions**

Let  $d(E_i)$  and  $d(R_i)$  be the time duration of the evaluate and reset phase in stage  $i$  respectively.

*Definition 3.1:*  $P_i = d(E_i) + d(R_i)$  is the *period* of stage  $i$ , which is the delay between the arrival of an input at the current stage  $i$  to the arrival of the next input at the current stage



*i*. The period  $P$  represents a single cycle of execution in a stage consisting of a reset and an evaluate phase.

### 3.4.2 Analysis of the Reset and Evaluate Phase

As shown in Figure 3.3, the internal phase of a stage  $i$  can be determined by observing signal  $O_i$ . When  $O_i = 0$ , stage  $i$  is in the reset phase. Otherwise, it is in the evaluate phase. Assume there are  $n$  stages in the pipeline. Since the evaluate phase of stage  $n$ , which is the last pipeline stage, does not depend on the reset phase of another stage, its reset and evaluate phase tend to have the same duration:

$$d(E_n) = d(R_n) = \frac{P_n}{2} \quad (3.1)$$

Figure 3.10 shows the waveforms of the stage outputs and the phase of stage 15 and 16 in a 16-stage prototype S-SRSL pipeline. It is clear that the reset and evaluate phase of stage 16 have the same duration (i.e.,  $d(E_{16}) = d(R_{16})$ ). However, this is not true for other stages. Figure 3.11 shows the waveforms of the stage outputs and the phases of stage 1 and 2 in a 16-stage S-SRSL prototype pipeline. Figure 3.11 shows how the duration of the evaluate phase of stage 2 is much greater than the duration of its reset phase.

The equal duration of the reset and evaluate phase on the right side of the pipeline can be explained by considering stage 4 in Figure 3.3 in which the reset loop oscillates without waiting on any incoming signal since stage 4 is the last stage in the pipeline.

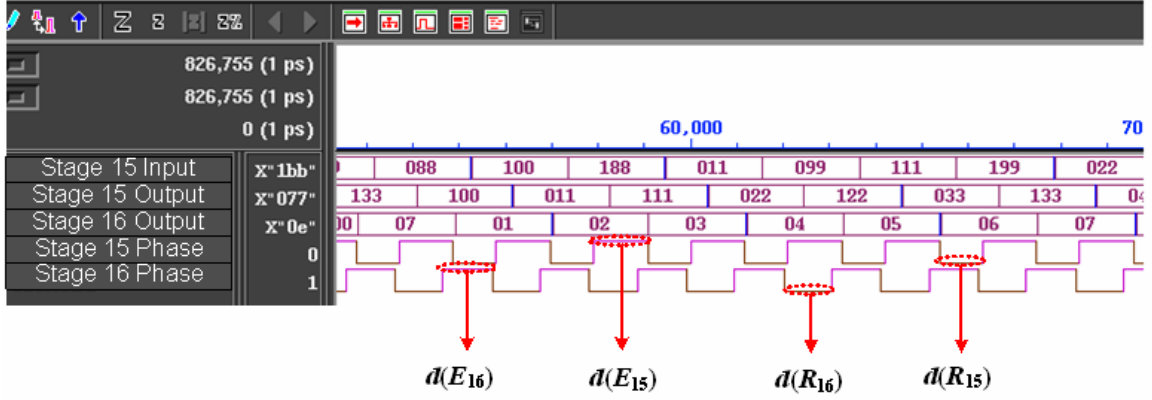


Figure 3.10: Simulation snapshot of stage 15 and 16 in a 16-stage prototype S-SRSL pipeline.

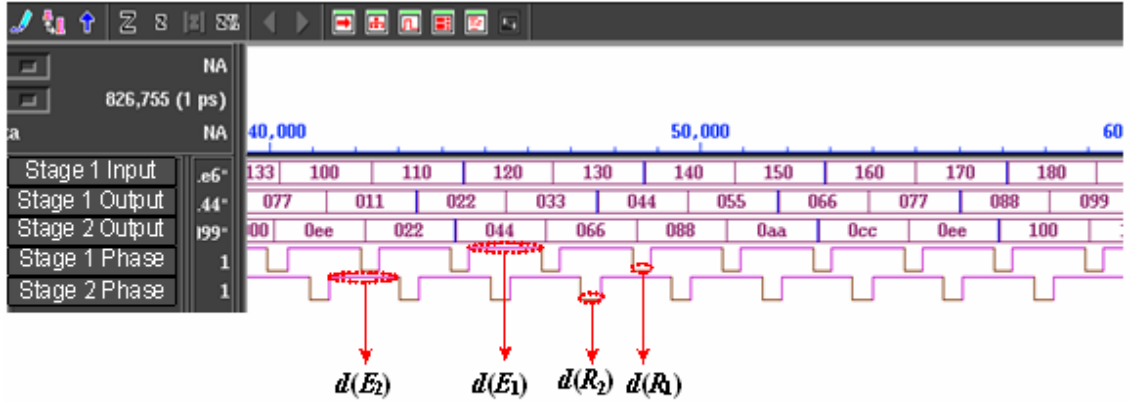


Figure 3.11: Simulation snapshot of stage 1 and 2 in a 16-stage prototype S-SRSL pipeline.

However, the evaluate phase of stage  $n-1$  has to wait on the arrival of the reset phase from stage  $n$  to the latch-enabling AND gate in order for data to flow from the former to the latter. This has the effect of stretching the duration of the evaluate phase of stage  $n-1$ :

$$d(E_{n-1}) > d(E_n) \quad (3.2)$$

In Figure 3.10, it is clear that  $d(E_{15})$  is slightly greater than  $d(R_{16})$ . Since  $d(E_{16}) = d(R_{16})$  by equation (3.1), then  $d(E_{15}) > d(E_{16})$  as stated in equation (3.2). This relationship between the duration of the evaluate phases in one stage and the next stage becomes more pronounced between stages located on the left side of the pipeline as shown in Figure 3.11. For example in Figure 3.3,  $O_3 = 1$  has to travel through the reset loop delay to reach the AND gate that enables latch 4. Next, it waits for the arrival of  $O_4 = 0$  to the same gate, which has the effect of increasing the duration of  $O_3 = 1$ . After a short time,  $L_4$  switches to 1 when  $O_4 = 0$  arrives to the AND gate that enables latch 4.  $L_4 = 1$  travels through the NOR gate before reaching  $O_3$  when the latter switches to 0. This cycle is much longer in stage 3 than in stage 4, which makes the evaluate phase of stage 3 longer than that of stage 4.

Since stage  $n$  starts its reset phase somewhat earlier, it tends to complete this phase also earlier, thus causing the reset phase of stage  $n-1$  to be somewhat shorter:

$$d(R_{n-1}) < d(R_n) \quad (3.3)$$

In Figure 3.10,  $d(R_{15}) < d(E_{16})$ . Also,  $d(R_1) < d(E_2)$  in Figure 3.11. Since  $d(E_{16}) = d(R_{16})$  by equation (3.1), then  $d(R_{15}) < d(R_{16})$  as stated in equation (3.3). The increase in the evaluate phase and the decrease in the reset phase of stage  $n-1$  with regard to the phases of stage  $n$  is exactly the same:

$$d(E_{n-1}) - d(E_n) = d(R_n) - d(R_{n-1}) \triangleq \delta \quad (3.4)$$

This equal increase and decrease is due to the fact that the period is equal for both stages  $n$  and  $n-1$ :

$$P_{n-1} = P_n = P \quad (3.5)$$

This can be seen in Figure 3.10 and 3.11, where the duration of the stage output when it is 0 plus the duration of the same stage output when it is not 0 is identical for all stages.

### 3.4.3 Effect of $\delta$ on the Pipeline Stages

The delay difference, denoted by  $\delta$ , is caused by the unequal lengths of the reset loop on which the phase signals travels in stage  $n-1$  and  $n$ . While the phase signal in stage  $n$  starts from the left NOR gate, passes through the buffer delay, and back to the same NOR gate, the phase signal in stage  $n-1$  crosses the same path in addition to an inverter and an AND gate. The AND gate with one inverted input is the latch enabling gate between stage  $n-1$  and  $n$ . Since the phase signal travels along this augmented path in stage  $n-1$  twice, once when  $O_{n-1} = 1$  and once when  $O_{n-1} = 0$ , the  $\delta$  delay difference between the two paths in both stages is at most equal to twice the delay of the inverter and latch enabling AND gate. Let  $d(\text{INV})$  and  $d(\text{AND})$  be the average delay through an inverter and an AND gate respectively, then:

$$\delta = 2(d(\text{INV}) + d(\text{AND})) \quad (3.6)$$

$\delta$  propagates leftward from stage  $n$  to stage 1 causing the duration of the evaluate and reset phase of each stage  $i$  to increase and decrease by  $\delta$  respectively with regard to its successor stage  $i+1$ :

$$d(E_i) = d(E_n) + (n-i)\delta \quad (3.7)$$

$$d(R_i) = d(R_n) - (n-i)\delta \quad (3.8)$$

In brief, this delay propagates toward the left side of the pipeline thus causing the duration of the evaluate and reset phases to gradually increase and decrease respectively with each stage to the left of the pipeline without changing the duration of a single period. The propagation of this delay is highly visible in Figure 3.11 where the phases of stage 1 and 2 are highlighted.

#### 3.4.4 $\delta$ and Pipeline Depth

Based on equation (3.4) shown in the previous section with regard to stage  $n-1$  and  $n$ ,

$$\begin{aligned} d(E_{n-1}) - d(E_n) &= \delta \\ d(E_{n-1}) &= d(E_n) + \delta \\ d(E_{n-1}) &= \frac{P}{2} + \delta \\ d(E_{n-1}) &= d(R_n) + \delta \end{aligned} \quad (3.9)$$

This implies

$$d(E_{n-1}) > d(R_n) \quad (3.10)$$

Let  $d(L_i^+)$  be the minimum duration of the enable of the latch at logic level 1 between stage  $i-1$  and  $i$ . Since the latch between stage  $n-1$  and  $n$  is enabled when the former is in the evaluate phase and the latter is in the reset phase, the duration of the latch enable depends primarily on that of the reset phase of stage  $n$  since this reset phase is shorter than the evaluate phase of stage  $n-1$  as shown in equation (3.10):

$$\begin{aligned} d(L_n^+) &= d(R_n) \\ d(L_n^+) &= \frac{P}{2} \end{aligned} \quad (3.11)$$

This can be seen in Figure 3.10 where  $d(L_{16}^+) = d(R_{16})$ . Given the  $\delta$  delay domino effect, this dependence of the duration of the latch enable on the duration of the reset phase of the stage to the right of the latch applies to every stage going leftward in the pipeline:

$$d(L_i^+) = d(R_i) \quad (3.12)$$

As a result, as the duration of the reset phase of each stage decreases by moving to stages on the left side of the pipeline, so does the duration of the latch enable:

$$d(L_i^+) = d(R_i) \Leftrightarrow d(L_i^+) = d(R_n) - (n-i)\delta \Leftrightarrow d(L_i^+) = \frac{P}{2} - (n-i)\delta \quad (3.13)$$

Based on the above equation, one can predict the maximum number of stages that the pipeline can accommodate by solving the above equation for the variable  $n$  starting from stage 1:

$$d(L_1^+) = \frac{P}{2} - (n-1)\delta \Leftrightarrow n = 1 + \frac{1}{\delta} \left( \frac{P}{2} - d(L_1^+) \right) \quad (3.14)$$

Based on equation (3.14), a deep pipeline can be realized by (i) decreasing  $\delta$ , (ii) increasing  $P$ , or (iii) decreasing  $d(L_1^+)$ . (i) can be achieved by using high speed AND gates, (ii) can be achieved by increasing the delay in the self-reset network of each stage through the insertion of buffers or inverter chains, while (iii) can be achieved by using high speed latches.

### 3.4.5 Area Cost

In order to shed light on the area cost of S-SRSL pipelines, they are briefly contrasted with the area cost of clocked pipelines. Whereas the latter require only flip-flop registers between the pipeline stages, S-SRSL pipelines require inter-stage latches in addition to intra-stage reset networks and delay buffers. Although the area of a flip-flop tends to be slightly greater than the area of a latch (by the equivalent of two gates in most library implementations), this difference is not sufficiently large to overcome the area overhead caused by the insertion of delay buffers. In general, the area of these buffers tends to grow proportionally with the delay on the critical path of the intra-stage logic.

### 3.4.6 Fault Handling

In analyzing how the S-SRSL pipeline handle faults, only stuck-at faults are considered. Focus is directed to the outcomes caused by the output of the reset network

of a given stage getting (i) stuck at 1, thus causing the stage to be locked in the evaluate phase, or (ii) stuck at 0 causing the stage to be locked in the reset phase.

- **Stage locked in the evaluate phase:** If the phase line, which is the output line of the reset network, of a stage  $j$  gets stuck at 1 for a time longer than  $P$ , the stage is locked into the evaluate phase. Two distinct behaviors can be observed throughout the pipeline depending on where the stage, displaying one behavior or another, is located in the pipeline:

- (i) **Left Side Stages:** When stage  $j$  is stuck in the evaluate phase, the right input of the AND gate which controls latch  $j$  is stuck on 1. This in turn causes the output of the AND gate to be stuck on 0. As a result latch  $j$  is closed and data does not flow between stage  $j$  and  $j-1$ . When the output of the AND gate gets stuck on 0, the output of the NOR gate of the reset network in stage  $j-1$  gets stuck on 1. As a result, stage  $j-1$  is locked into the evaluate phase. This phenomenon occurs in every pair of stages located on the left of stage  $j$ . In the overall, this automatically causes all stages  $i$ , where  $i < j$ , to complete their reset phases before getting stuck in their evaluate phases. Note that, in an S-SRSL pipeline, each stage completes its reset phase on its own. However, a stage cannot complete its evaluate phase unless its right neighbor enters its own reset phase. As each pair of neighboring pipeline stages gets stuck in the evaluate phase, starting from stage  $j$  and going leftward to stage 1, their inter-stages latches are disabled and subsequently the flow of data is interrupted in



all stages to the left of stage  $j$ . This forced locking of the stages in the evaluate phase will propagate as a wave to the left side of the pipeline starting from stage  $j$  until it reaches stage 1.

- (i) **Right Side Stages:** Even though stage  $j$  is stuck at 1, stage  $j+1$  can nevertheless complete its own reset phase based on how an S-SRSL pipeline operates. Note that the input of the reset network in stage  $j+1$  is driven by the output of the AND gate controlling the latch between stage  $j+1$  and  $j+2$ . As a result, the oscillations of the reset network in stage  $j+1$  depends primarily on those of the reset network in stage  $j+2$ . Since neither of the reset networks in these two stages is stuck, they can operate in lock-step fashion. So, when stage  $j+1$  enters its reset phase, its latch is transparent and data is subsequently passed from stage  $j$  to stage  $j+1$ . Just as stage  $j+1$  is able to complete its own reset phase, stage  $j+2$  can complete its own in a similar manner. As soon as stage  $j+2$  enters its reset phase, data is transferred from stage  $j+1$  to stage  $j+2$ . Sequence of events, similar to the ones described for stage  $j+1$  and  $j+2$ , occur in every pair of stages located to the right of stage  $j$ , thus allowing data to flow through the pipeline from stage  $j$  to stage  $n$  where  $n$  is the last stage in the pipeline. Since the flow of data is interrupted on the left side stages, the same data items keeps flowing repeatedly from stage  $j$  to stage  $n$  as long as stage  $j$  remains stuck in the evaluate phase.

- **Stage locked in the reset phase:** If the output the reset network of a stage  $j$  gets stuck at 0 for a time longer than  $P$ , the stage is locked into the reset phase. Two distinct behaviors can be observed throughout the pipeline depending on where the stage, displaying one behavior or another, is located in the pipeline:

- (i) **Left Side Stages:** When stage  $j$  is stuck in the reset phase, the right input of the AND gate, which controls latch  $j$ , is stuck at 0. This in turn causes the output of the AND gate to depend on the output of the reset network of stage  $j-1$ . If this output becomes 0, which indicates that stage  $j-1$  is in the reset phase, it forces the output of the AND gate to become 0 thus disabling latch  $j$ . The 0-output of the AND gate drives the input of the reset network in stage  $j-1$  to force its output to switch to 1. This indicates that stage  $j-1$  has started its evaluate phase. This 1-output of the reset network of stage  $j-1$  forces the output of the AND gate controlling latch  $j$  to switch to 1, thus enabling latch  $j$  and allowing data to flow from stage  $j-1$  to  $j$ . The 1-output of the AND gate drives the input of the reset network of stage  $j-1$  forcing the output of the latter to switch to 0 and allowing stage  $j-1$  to start a reset phase. In the overall, stage  $j-1$  continues to oscillate between the reset and evaluate phases even though stage  $j$  is stuck in the reset phase. Because stage  $j-1$  continues its normal oscillation, this allows all the stages to the left of stage  $j-1$  to oscillate normally in lock step fashion with each other. As a result, data flows uninterrupted from stage 1 to stage  $j$ .

- (ii) **Right Side Stages:** When stage  $j$  is stuck in the reset phase, the left input of the AND gate, which controls latch  $j+1$ , is stuck at 0. This will disable latch  $j+1$  as long as stage  $j$  is stuck in the reset phase. As a result, data is prohibited from flowing from stage  $j$  to  $j+1$ . However, this does not stop stage  $j+1$  from oscillating between its reset and evaluate phases. As stated before, the input of the reset network in stage  $j+1$  is driven by the output of the AND gate controlling the latch between stage  $j+1$  and  $j+2$ . As a result, the oscillations of the reset network in stage  $j+1$  depends primarily on those of the reset network in stage  $j+2$ . Since neither of the reset networks in these two stages is stuck, they can operate in lock-step fashion. In fact, every pair of stages located to the right of stage  $j$  allows data to flow through their latches thus establishing an uninterrupted data flow from stage  $j+1$  to  $n$ . Because the latch between stage  $j$  and  $j+1$  remains disabled, the flow of incoming data stops at latch  $j$ . As a result, data is overwritten at every period in stage  $j$  while the same data item keep flowing from stage  $j+1$  to  $n$ .

### **3.5 Prototype Implementation of the S-SRSL Pipelines**

To test and validate SRSL and its use in S-SRSL pipelines, several prototypes of linear and non-linear pipelines have been implemented.

### 3.5.1 The S-SRSL Linear Pipeline

A 16-stage four-bit S-SRSL pipeline was modeled in VHDL where each stage contains a four-bit ripple-carry adder. For validation purposes, it was decided to insert an adder in each stage in order to amplify delay effects and subsequently constrain the performance of the pipeline. The netlist of the pipeline was generated using Synopsys Design Compiler based on a 0.25  $\mu\text{m}$  CMOS library [65, 66]. Cadence's Silicon Ensemble was used to place and route the pipeline. The pipeline fits into a frame of 90,057.74  $\mu\text{m}^2$  as shown in Figure 3.12 yielding a total latency of 15.76 nanoseconds and a throughput of 453.95 Megaoutputs/second based on the 2.18 ns period of the last stage.

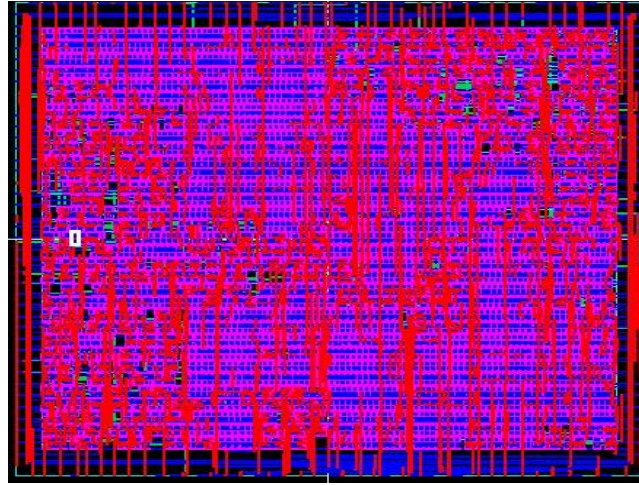


Figure 3.12: Chip layout of the four-bit 16-stage S-SRSL pipeline.

Table 3.1 shows the summary of the linear pipeline implementation. The layout of this pipeline contains 1,344 standard cells connected by 1,416 nets and 1,66 IO pins. four parameters were measured in layout simulations of the pipeline, the period of each stage  $P$ , the duration of the evaluate phase  $d(E)$ , the reset phase of each stage  $d(R)$ , and the enable of each latch  $d(L^+)$ .

Table 3.1: S-SRSL linear pipeline implementation.

|                            |  |
|----------------------------|--|
| Stages                     | 16   |
| Bit width                  | 4  |
| Combinational network      | 4-bit adder  |
| Synthesis                  | Synopsys Design Compiler   |
| Layout                     | Cadence Silicon Ensemble   |
| Simulation                 | Synopsys VCS Simulator   |
| Library                    | 0.25 $\mu\text{m}$ CMOS library                                  |
| Cells                      | 1,344  |
| Nets                       | 1,416  |
| IO pins                    | 166  |
| Area                       | 90,057.74 $\mu\text{m}^2$  |
| Latency                    | 15.76 ns   |
| Throughput                 | 453.95 Megaoutputs/second  |
| Stage period               | 2.18 ns  |
| Latch enable duration      | 1.01 ns (stage 16) down to 0.64 ns (stage 1)                     |
| $\delta$ delay             | Between any stage and the last stage                             |
| Theoretical pipeline depth | $n = 1 + \frac{1}{\delta} \left( \frac{P}{2} - d(L_1^+) \right)$ |

Figure 3.13 shows the duration of the latch enable, reset phase, evaluate phase,  $\delta$ , and the period of each stage. In this figure,  $\delta$ , labeled as Delta Delay, is almost constant from stage to stage. However, the reset phase gradually decreases from the right to the left of the pipeline while the evaluate phase gradually increases from the right to the left of the pipeline as predicted by equation (3.7) and (3.8). This gradual increase in the evaluate phase, from the right to the left of the pipeline, is attributed to the propagation of  $\delta$  based on the explanation proposed in the timing analysis section of the pipeline. Similarly, the observed gradual decrease in the reset phase, from the right to the left of the pipeline, is also attributed to the propagation of  $\delta$  based on the same explanation. Furthermore, the duration of the latch enable is almost equal to that of the reset phase in each stage. As a result, the duration of the latch enable decreases gradually at the same rate as the duration of the reset phase from the right to the left of the pipeline. This shows

how the duration of the latch enable is closely tied to the duration of the reset phase as derived in equation (3.12).

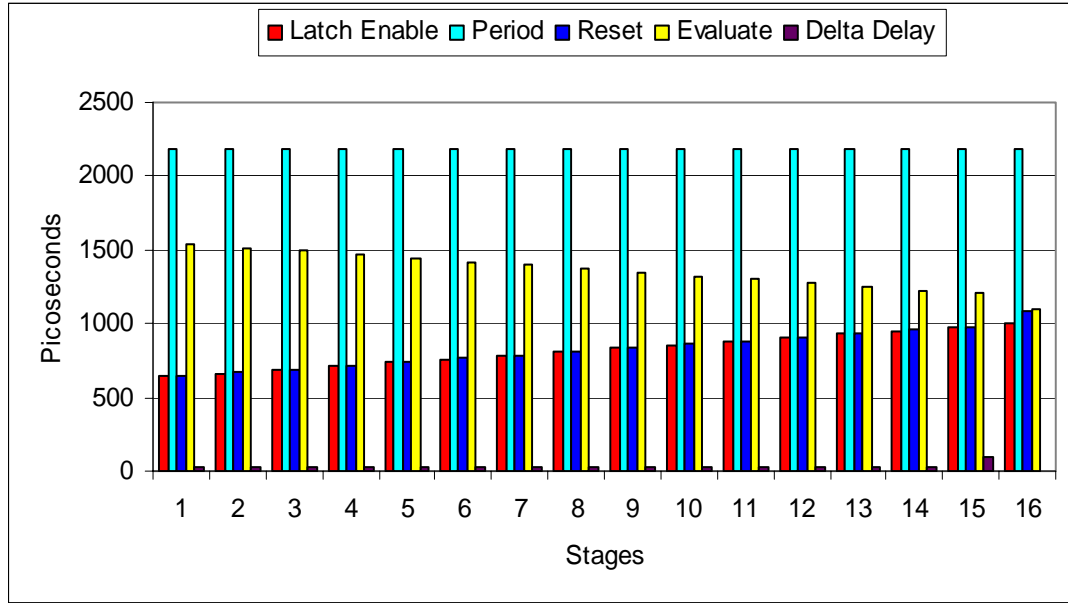


Figure 3.13: Simulation results of  $d(L^+)$ ,  $d(R)$ ,  $d(E)$ ,  $\delta$ , and  $P$  in a 16-stage S-SRSL pipeline.

Figure 3.14 shows the values obtained for the duration of the reset phase and the latch enables using simulation and the derived equations (3.8) and (3.13). The values obtained through simulation are labeled as empirical values while the values obtained analytically are labeled as analytical values. As shown in the figure, the values predicted by the equations and those obtained through simulation are highly correlated. On the overall, the empirical duration of the reset phase is higher than its analytical duration by 47 picoseconds on the average while the empirical duration of the latch enable is higher than its analytical duration by 35 picoseconds across all stages of the pipeline.

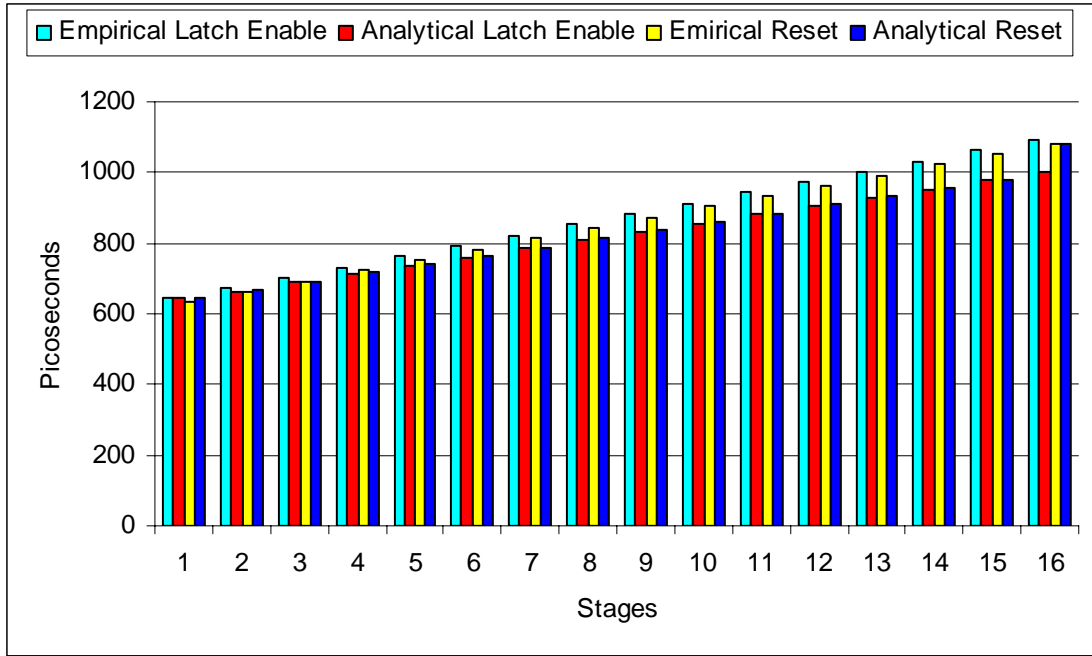


Figure 3.14: The empirical and analytical values of  $d(R)$  and  $d(L^+)$  in a 16-stage S-SRSL pipeline.

This difference can be viewed as under-estimation since the analytical values are slightly smaller than the simulation values. The 47 picoseconds underestimation represents 6.21% of the duration of the reset phase on the average across all stages of the pipeline. On the other hand, the 35 picoseconds underestimation represents 5.52% of the duration of the latch enable on the average in all stages of the pipeline. However, this underestimation is not constant across all stages. In fact, the underestimation increases slightly above the average in the stages located on the right side of the pipeline while it decreases slightly below the average in the stages located on the left side of the pipeline. This indicates that the prediction accuracy of equation (3.8) and (3.13) tends to be higher for stages on the left side of the pipeline. These non-constant underestimations can be accounted for by the fact that  $\delta$  does not remain exactly constant since it decreases at a

negligible rate while propagating from the left to the right stages across the pipeline. For simplicity,  $\delta$  was considered constant throughout the timing analysis of the pipeline.

### 3.5.2 The S-SRSL Non-Linear Pipeline

To evaluate the performance of the S-SRSL non-linear pipeline, two prototype pipelines were implemented in order to study the impact of the join and fork operation on the overall performance of the pipeline.

#### 3.5.2.1 The S-SRSL Join Pipeline

A four-bit six-stage pipeline was modeled in VHDL where each stage contains a four-bit adder as shown in Figure 3.15. The pipeline netlist was generated using Synopsys Design Compiler based on a  $0.25\mu\text{m}$  CMOS library [65, 66]. Four parameters were measured in layout simulations of the pipeline, namely the period of each stage ( $P$ ), the duration of the evaluate phase ( $d(E)$ ), the reset phase of each stage ( $d(R)$ ), and the enable of each latch ( $d(L^+)$ ).

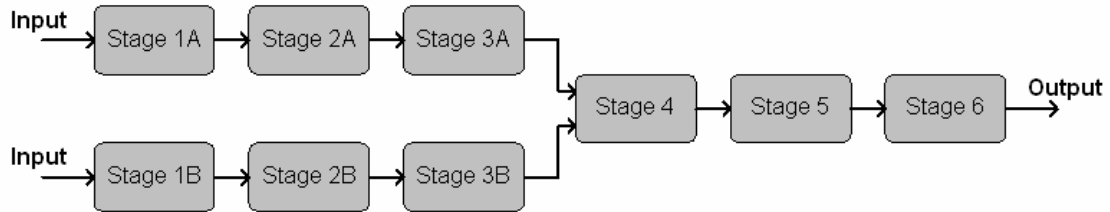


Figure 3.15: Four-bit six-stage S-SRSL join pipeline.

Figure 3.16 shows a simulation snapshot of stages 3A, 3B and 4 from the prototype pipeline shown in Figure 3.15. In Figure 3.16, the phase of stage 4 is always de-asserted when the phase of stage 3A and 3B are asserted and vice-versa. This shows



that both stages 3A and 3B oscillate in the same phase while stage 4 oscillates in the opposite phase. This ensures that data flow from stages 3A and 3B to stage 4 when both the former are in the evaluate phase while the latter is in the reset phase.

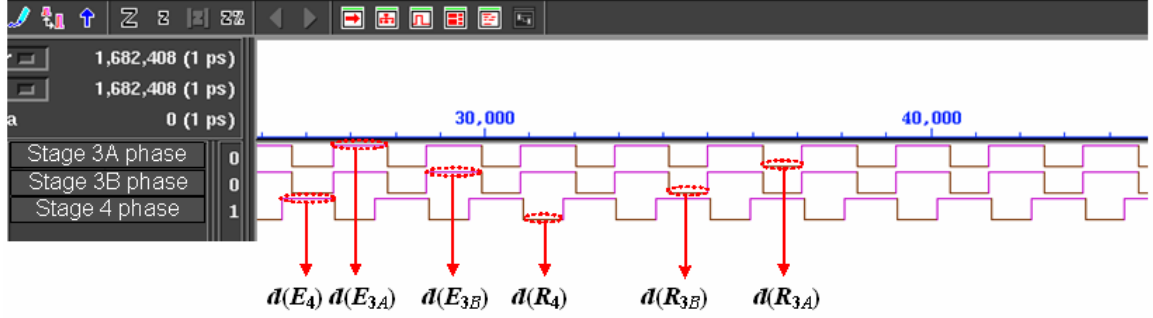


Figure 3.16: Simulation snapshot of the prototype S-SRSL join pipeline.

Figure 3.17 shows the duration of the latch enable, the reset phase, the evaluate phase,  $\delta$ , and the period of each stage in the S-SRSL join pipeline. As the figure shows, the duration of the latch enable and reset phase gradually decreases from the right to the left across the stages of the pipeline while the duration of the evaluate phase gradually increases from the right to the left across the stages of the pipeline due to the propagation of  $\delta$ . This propagation, characteristic of a linear pipeline, appears to occur also in the join pipeline.

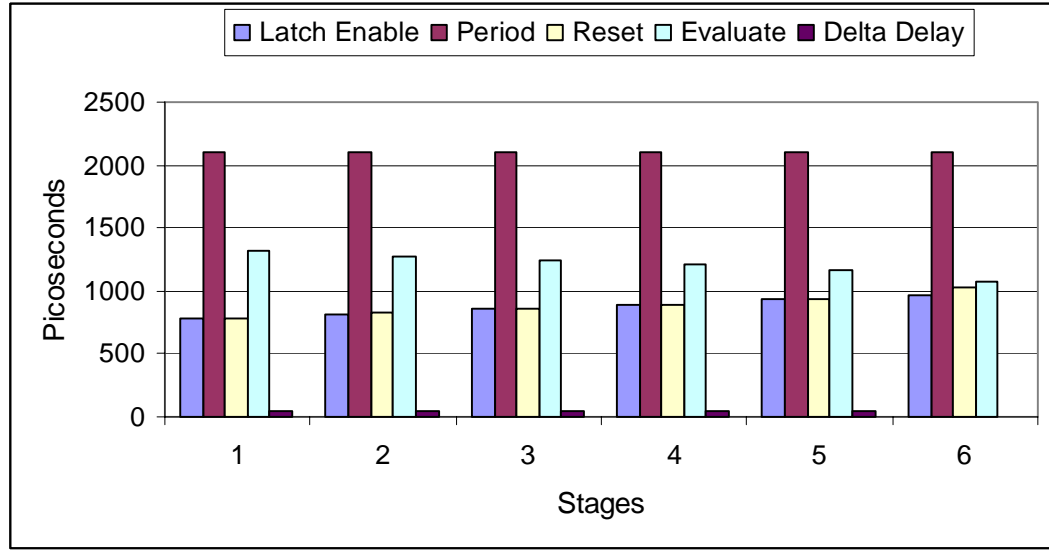


Figure 3.17: Simulation results of  $d(L^+)$ ,  $d(R)$ ,  $d(E)$ ,  $\delta$ , and  $P$  in the S-SRSL join pipeline.

### 3.5.2.2 The S-SRSL Fork Pipeline

A four-bit six-stage pipeline was modeled in VHDL where each stage contains a four-bit adder as shown in Figure 3.18. Its netlist was generated using Synopsys Design Compiler based on a 0.25 $\mu$ m CMOS library [65, 66]. Four parameters were measured in layout simulations of the pipeline, namely the period of each stage ( $P$ ), the duration of the evaluate phase ( $d(E)$ ), the reset phase of each stage ( $d(R)$ ), and the enable of each latch ( $d(L^+)$ ).

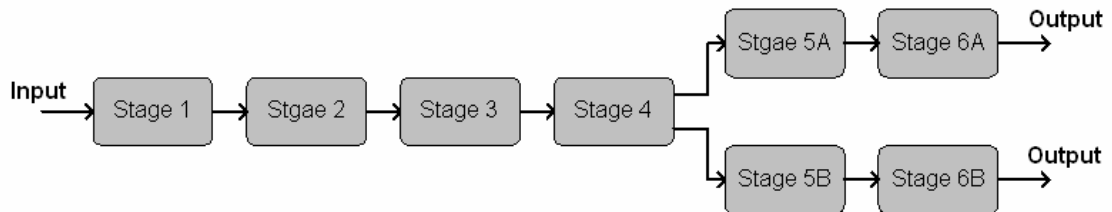


Figure 3.18: Four-bit six-stage S-SRSL fork pipeline.

Figure 3.19 shows a simulation snapshot of stages 4, 5A, and 5B from the prototype pipeline shown in Figure 3.18. In Figure 3.19, the phase of stages 5A and 5B are always de-asserted when the phase of stage 4 is asserted and vice-versa. This shows that stages 5A and 5B oscillate in the same phase while stage 4 oscillates in the opposite phase. This insures that data flows from stage 4 to stages 5A and 5B when the former is in the evaluate phase while the two latter are in the reset phase.

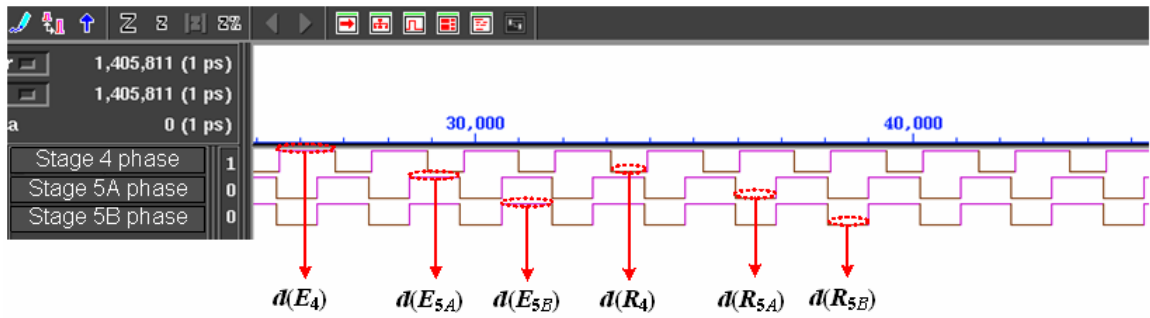


Figure 3.19: Simulation snapshot of the prototype S-SRSL fork pipeline

Figure 3.20 shows the duration of the latch enable, the reset phase, the evaluate phase,  $\delta$ , and the period of each stage in the S-SRSL fork pipeline. As the figure shows, the duration of the latch enable and reset phase gradually decreases from the right to the left across the stages of the pipeline while the duration of the evaluate phase gradually increases from the right to the left across the stages of the pipeline due to the propagation of  $\delta$ . This propagation, characteristic of an S-SRSL linear pipeline, appears to occur also in the S-SRSL fork pipeline.

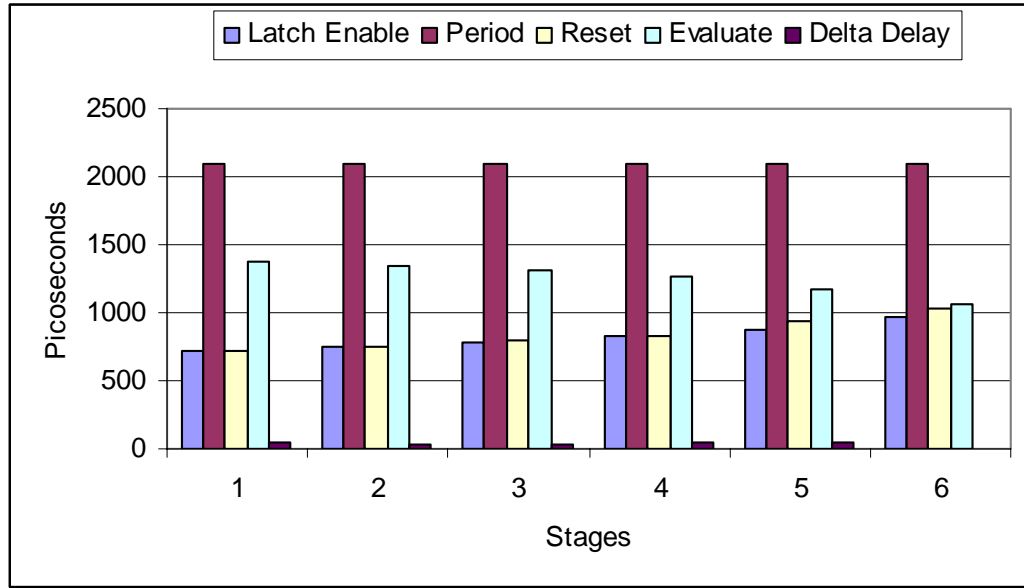


Figure 3.20: Simulation results of  $d(L^+)$ ,  $d(R)$ ,  $d(E)$ ,  $\delta$ , and  $P$  in the S-SRSL fork pipeline.

### 3.6 Summary

This chapter introduces SRSL, a clockless technique that can be used to pipeline computation and communication in order to circumvent problems associated with global clocking. In addition, the chapter describes how S-SRSL can be used to implement linear pipelines in addition to fork and join operations encountered in non-linear pipelines. Analysis of the pipeline performance shows that the depth of the pipeline is bound by its period,  $\delta$ , and the duration of the enable of the latch used in the pipeline implementation [59-64]. Prototyping experiments of the pipeline show that its actual performance is significantly closer to its analytical performance.

## **CHAPTER FOUR: PIPELINE-CONTROLLED SELF-RESETTING STAGE LOGIC PIPELINES**

This chapter presents the design and implementation of both linear and nonlinear P-SRSL pipelines. The communication protocol of these pipelines is quite different from that of the S-SRSL pipelines. Section 4.1 and 4.2 describe respectively how P-SRSL linear and non-linear pipelines operate while section 4.3 presents their timing analysis. Section 4.4 describes the implementation of P-SRSL prototype pipelines while section 4.5 presents a summarized comparison between the S-SRSL and P-SRSL pipelines. Section 4.6 concludes the chapter.

### **4.1 P-SRSL Linear Pipeline**

In P-SRSL pipelines, each stage consists of a combinational and a reset network similar to a stage in an S-SRSL pipeline as shown in Figure 4.1. Data flows from one stage to another through a latch in a linear pipeline. To insure proper data flow across stages, data is transferred from the current stage to the next one if the current stage is in the evaluate phase while the next stage is in the reset phase. Hence, the latch separating both stages is enabled when both stages are in the evaluate and reset phase respectively. This enable is the output of the AND gate that triggers the latch [67, 68].

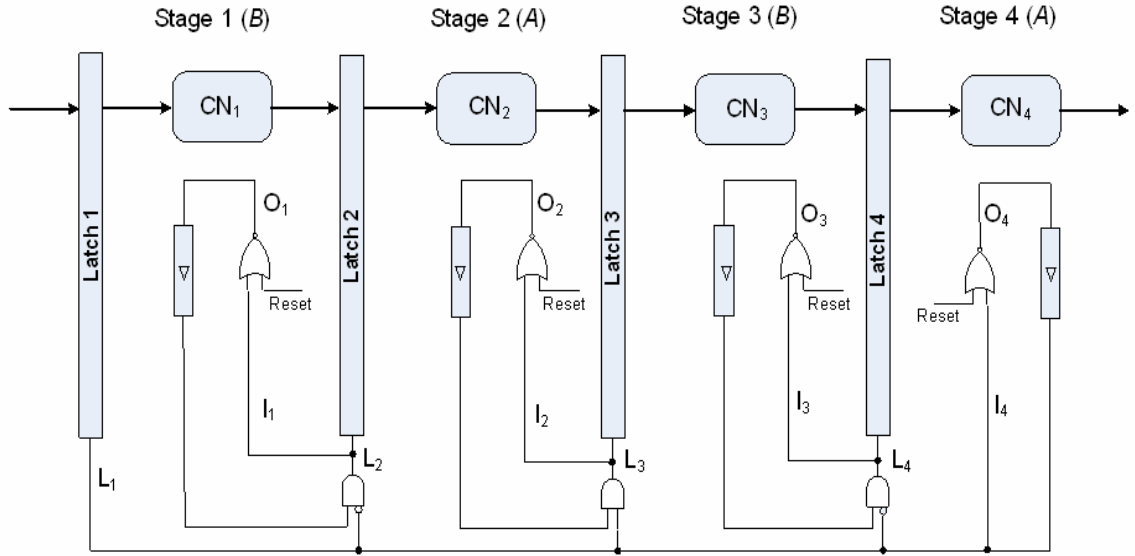


Figure 4.1: A four-stage P-SRSL pipeline.

Note that signal  $O_4$  drives the right input of each AND gate that enables each latch in the pipeline. This signal emanates from the last stage and travels along the pipeline to reach the AND gate of each inter-stage latch. Hence, the control of the phase sequences across the stages performed by this signal is exercised at the pipeline level. This approach is quite different from the S-SRSL pipeline where the output of the matching delay in a given stage drives one input of the AND gate that enables the latch separating it from the preceding stage. The control of the phase sequences in the latter approach is more local in nature since it propagates from stage to stage. In P-SRSL pipelines, stage synchronization is controlled in a semi-global manner whereby communication occurs primarily between the last stage and any other stage in the pipeline. To clarify the inner working of the P-SRSL pipeline, a stage is characterized based on the control signals of its proper latch.

*Definition 4.1:* A pipeline stage is said to be of *type A* if the phase signal of the last stage is inverted when it reaches the AND gate controlling the latch of the stage.

*Definition 4.2:* A pipeline stage is said to be of *type B* if the phase signal of the last stage is not inverted when it reaches the AND gate controlling the latch of the stage.

Note that the latch of a stage is the latch whose number is equal to the stage number in Figure 4.1. By default, stage 1 is of the complement type of that of stage 2, meaning that if stage 2 is of type *A* (*B*), stage 1 should be of type *B* (*A*). This stage characterization assigns opposite types to adjacent stages and identical types to every other stage. Stages of the same type oscillate in the same phase while stages of opposite types oscillate in opposite phases. When the last stage enters its reset phase, every stage of type *B* starts its own evaluate phase while every stage of type *A* starts its own reset phase. As soon as the last stage transitions to its evaluate phase, all the stages switch phase. During the reset phase of a stage of type *A*, the stage's left latch is enabled while the stage's right latch is disabled. Both latches are driven by the reset phase of the last stage in the pipeline. The latter latch will be enabled only when the stage switches phase, which occurs when the last stage enters its evaluate phase. At any cycle, every other stage will be in the reset phase while the remaining stages will be in the evaluate phase. A cycle later, the stages that were in the reset phase start their evaluate phases while the stages that were in the evaluate phase start their reset phases. Similarly to an S-SRSL pipeline, stages in a P-SRSL pipeline alternate between phases as computation progresses across the pipeline. Figure 4.2 shows the STG of the P-SRSL pipeline shown in Figure

4.1. This STG shows that the rising transition of  $L_3$  occurs after  $O_2$  and  $O_4$  experience both rising transitions. This means that latch 3 is enabled when both stages 2 and 4 are in the evaluate phase. However when  $O_4$  experiences a rising transition,  $L_2$  and  $L_4$  experience falling transitions. This shows that when latch 3 is enabled, latch 2 and 4 are disabled. Figure 4.3 shows how the stages alternate between phases as data flows across the pipeline by representing the asserted and de-asserted signals as solid and dashed lines respectively.

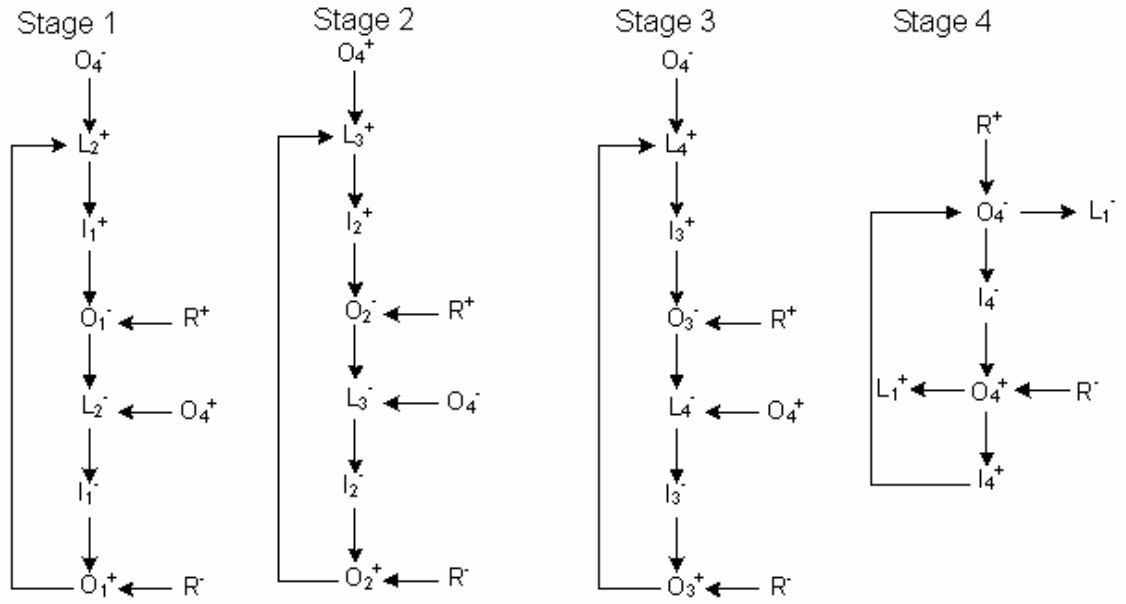
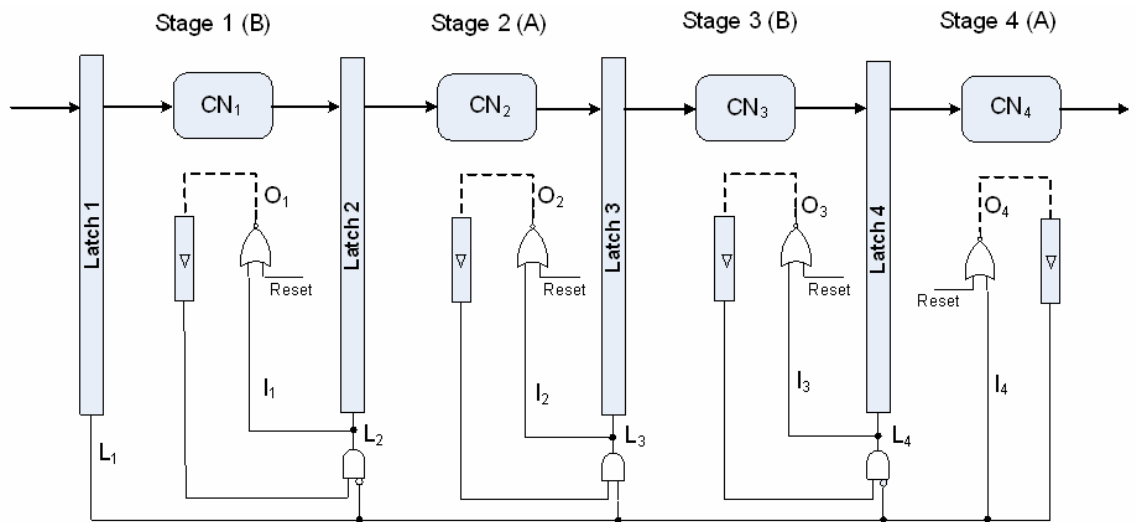
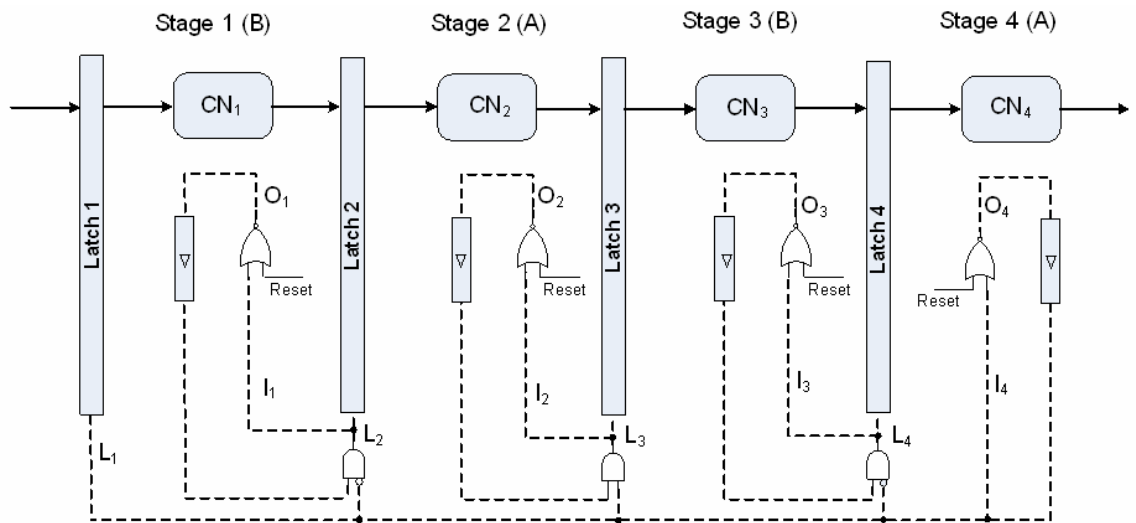


Figure 4.2: STG of the P-SRSL pipeline shown in Figure 4.1.

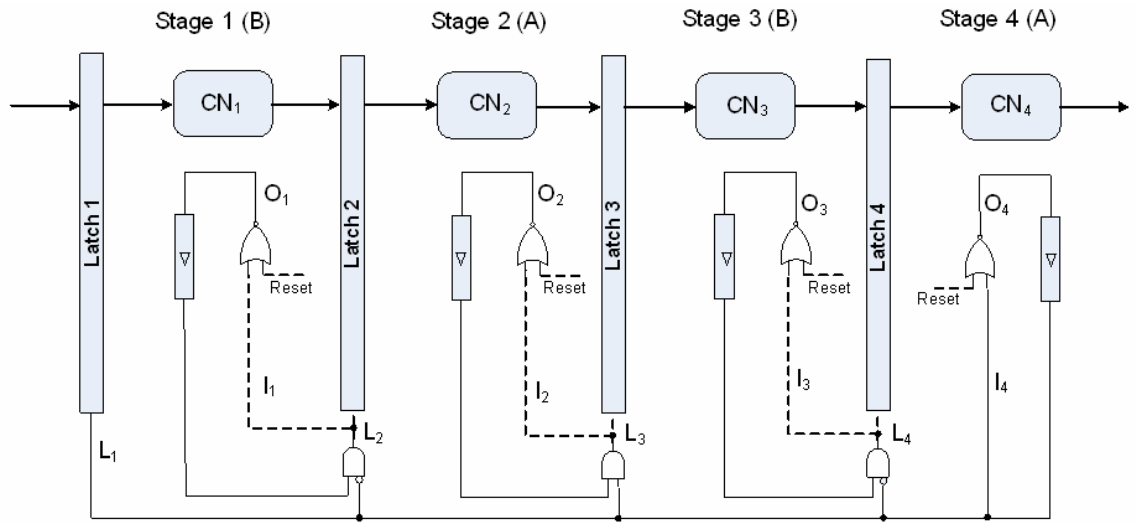




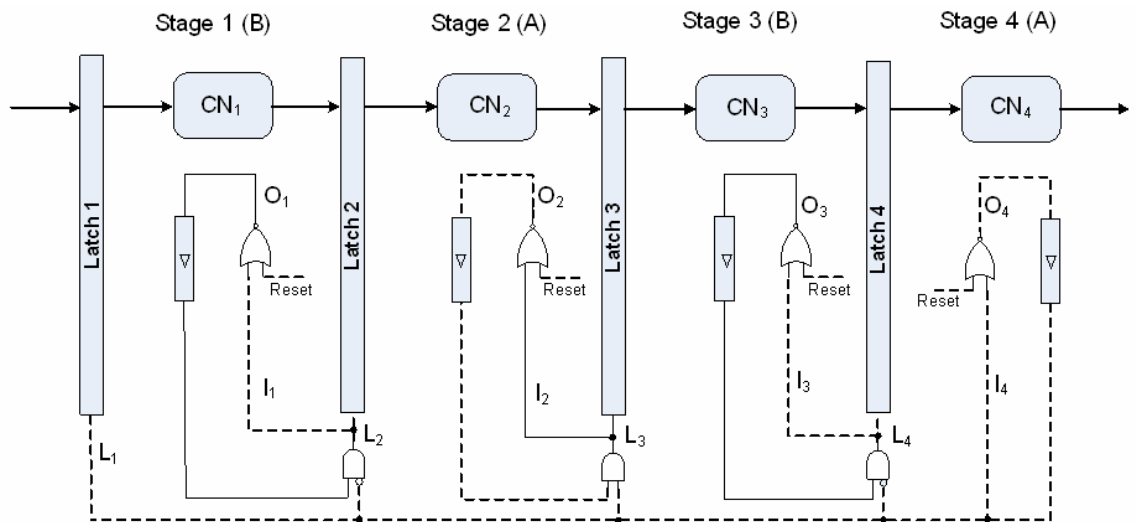
4.3(a): Assertion of the stage reset signals.



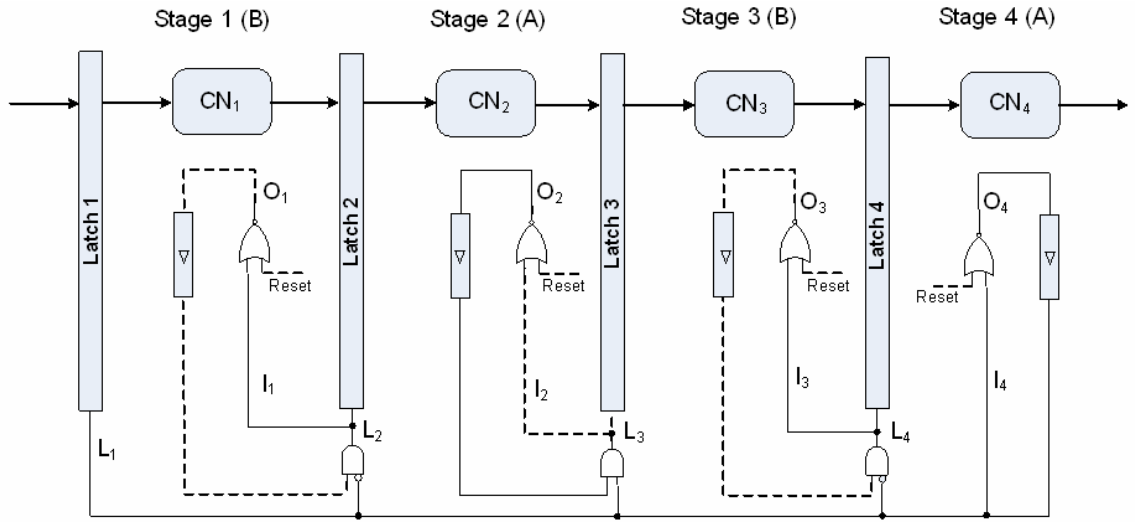
4.3(b): Reset phase of all stages.



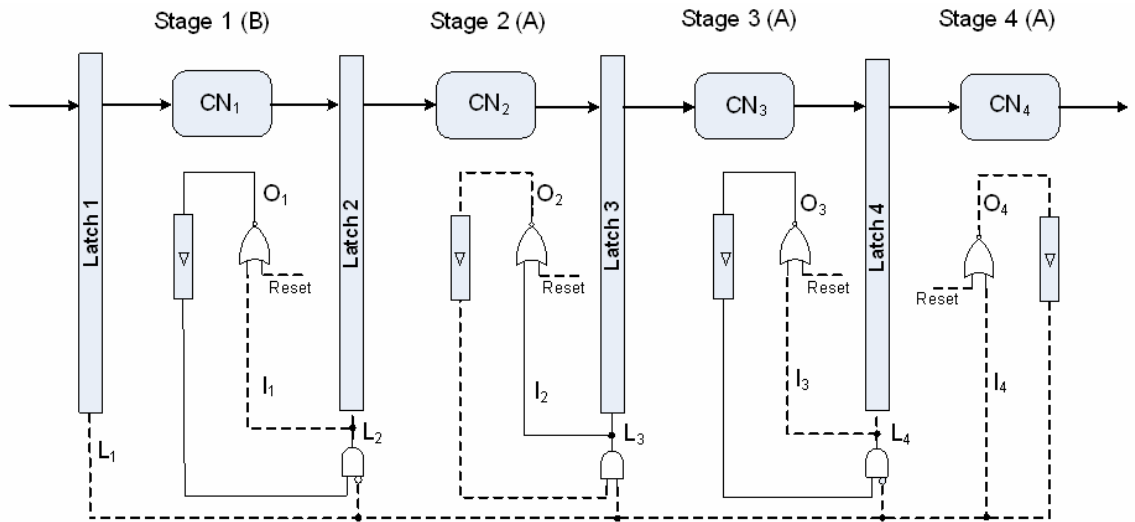
4.3(c): Evaluate phase of all stages.



4.3(d): Evaluate phase of stage 3 and 1.



4.3(e): Evaluate phase of stage 4 and 2.



4.3(f): Evaluate phase of stage 1 and 3.

Figure 4.3: Two execution cycles of a four-stage P-SRSL Pipeline.

## **4.2 P-SRSL Non-Linear Pipelines**

While linear pipelines can be used in many applications, complex systems require data to flow in divergent and convergent directions. Such systems can be realized as non-linear pipelines [63, 64]. To support divergence and convergence of data flow, primitives such as the *fork* and *join* operations have to be incorporated in the pipeline.

### **4.2.1 P-SRSL Join Pipeline**

Figure 4.4 shows a P-SRSL join pipeline. This pipeline operates similarly to the S-SRSL join pipeline. Data is transferred from stage A to stage C when the former is in the evaluate phase while the latter is in the reset phase. Similarly, data flows from stage B to stage C when the former is in the evaluate phase while the latter is in the reset phase. When data flows from stage A and B to C, the latches separating stage A and B from stage C are activated to capture the outputs of stage A and B thus feeding them to the inputs of stage C. Note that the phase signal of the last stage of the pipeline, namely  $O_{\text{Laststage}}$ , drives the three AND gates which enable the latches of stage A, B, and C as shown in Figure 4.4. Specifically, this phase signal drives the AND gate which enables the latch on the right side of stage C without being inverted. This means that data flows from stage C to its right neighbor when stage C and the last pipeline stage are both in the evaluate.

Moreover, the inverted value of the same phase signal drives the input of the AND gates that enable the latches on the right side of stage A and B. In this case, data



An alternative to this join structure will be a join operation in which stage A and B are of type A stages while stage C is of type B. In this case, both latches separating stage A and B from C will be enabled by AND gates whose outputs will be all non-inverted. In this pipeline, the control of latch 3 ( $L_3$ ) and 4 ( $L_4$ ) depends on the phase of stage A ( $O_A$ ), B ( $O_B$ ), and the last stage ( $O_{\text{Laststage}}$ ). In fact, signal  $O_{\text{Laststage}}$  reaches the left input of each AND gate enabling each inter-stage in the pipeline.

Figure 4.5 shows the STG of the pipeline shown in Figure 4.4. As shown in the figure,  $O_{\text{Laststage}}$  is involved in synchronizing the latch enables of each stage in the join structure.

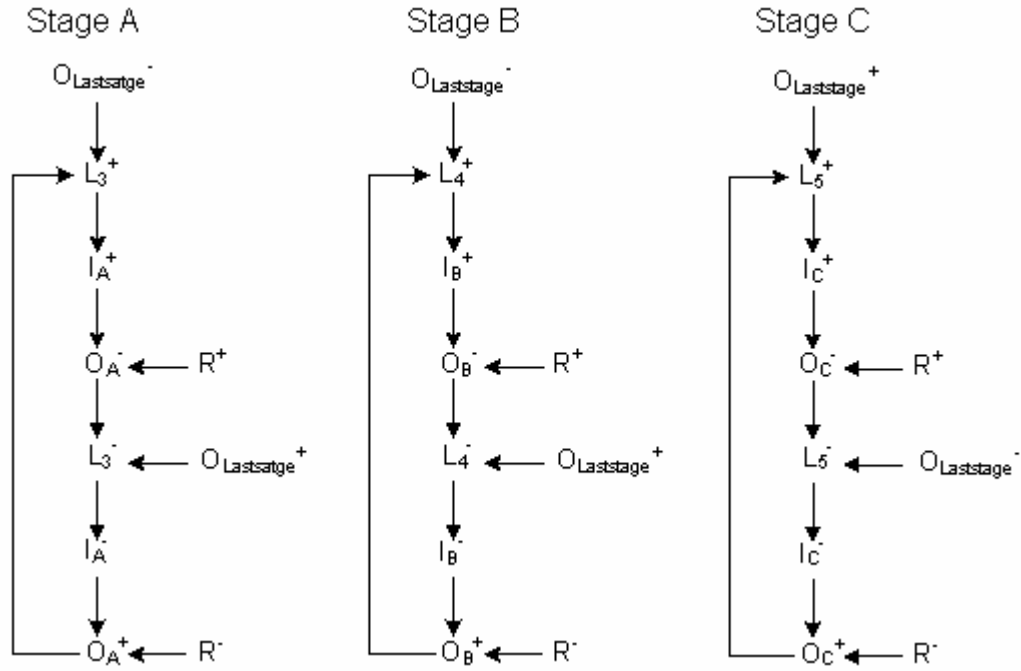


Figure 4.5: STG of the P-SRSL join pipeline shown in Figure 4.3.

### 4.2.2 P-SRSL Fork Pipeline

Figure 4.6 shows a P-SRSL fork pipeline. This pipeline operates similarly to the S-SRSL fork pipeline. However, in this pipeline, the enables of latches 2 ( $L_2$ ) and 4 ( $L_4$ ) depend on the phase of the last stage in the upper branch of the fork ( $O_{UpperLast}$ ), while the enables of latches 3 ( $L_3$ ) and 5 ( $L_5$ ) depend on the phase of the last stage in the lower branch of the fork ( $O_{LowerLast}$ ). In addition, the enable of latch 1 ( $L_1$ ) depends on the arrival of the phases of the last stages in both fork branches ( $O_{UpperLast}$  and  $O_{LowerLast}$ ). This arrival is captured by the H gate shown in Figure 4.6. The G gate plays the same role as the G gate of the S-SRSL fork pipeline.

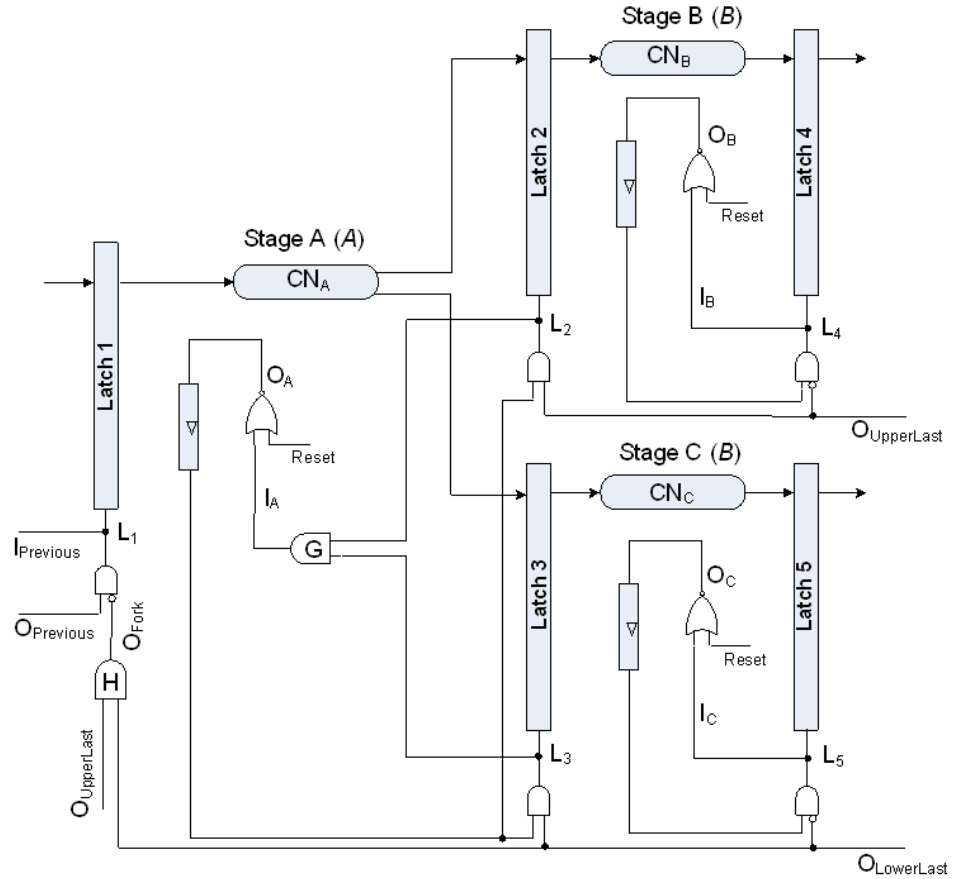


Figure 4.6: Structure of a fork P-SRSL pipeline.

Figure 4.7 shows the STG of the pipeline shown in Figure 4.6. In this STG,  $L_2$  experiences a rising transition after  $O_{UpperLast}$  experiences the same transition while  $L_4$  experiences a rising transition after  $O_{UpperLast}$  experiences a falling transition. A similar observation can be made for  $L_3$  and  $L_5$  with regard to  $O_{LowerLast}$ . On the other hand,  $L_1$  experiences a rising transition after  $O_{Fork}$  experiences a falling transition and vice-versa. The falling transition of  $O_{Fork}$  occurs after both  $O_{UpperLast}$  and  $O_{LowerLast}$  experience falling transitions.

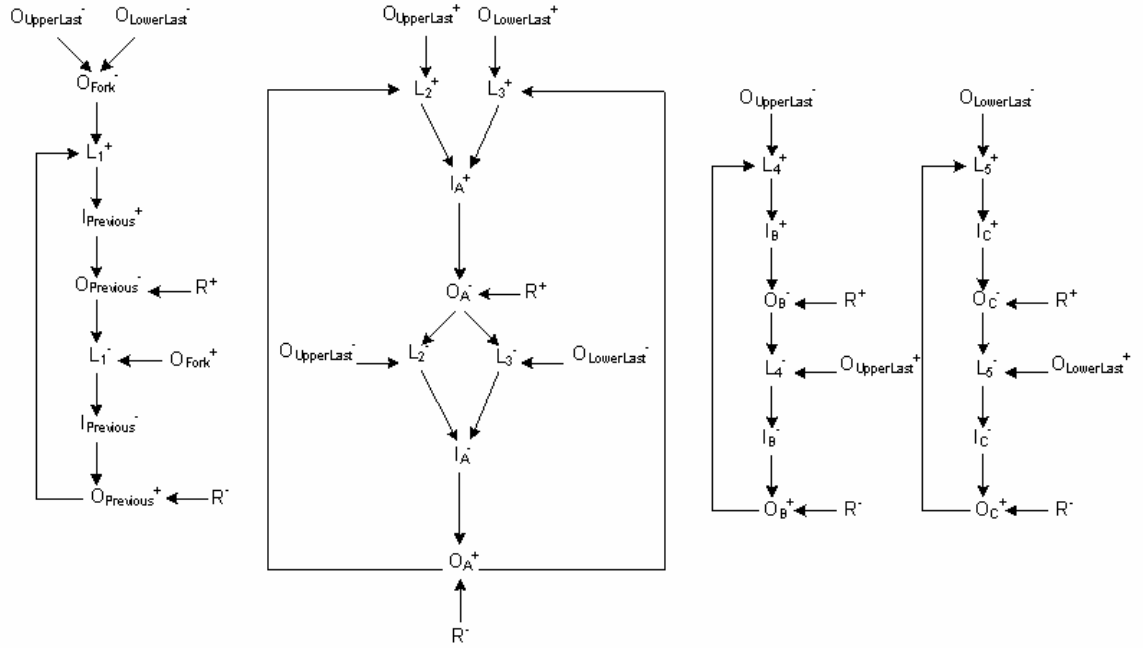


Figure 4.7: STG of the P-SRSL fork pipeline shown in Figure 4.6.

Contrary to the local control seen in the fork operation of the S-SRSL non-linear pipeline, the last stage in the upper and lower segments of the pipeline fork plays a primary role in synchronizing data transfer between neighboring stages in the fork operation of the P-SRSL non-linear pipeline. Note that Figure 4.6 shows a sample fork



structure in which stage B and C are of type *B* stages while stage A is of type *A* based on the stage characterization described in section 4.1. An alternative to this fork structure will be a fork operation in which stage B and C are of type *A* while stage A is of type *B*. In this case, latch 3 and 4 will be enabled by two-input AND gates where each gate has an inverted input.

### **4.3 Performance of the Pipeline**

To explain the performance of the P-SRSL pipeline, the same timing parameters defined in chapter 3, namely  $d(E_i)$ ,  $d(R_i)$ ,  $P_i$ , are used in this section. Next, these parameters are used in a signal timing analysis to characterize the performance of the pipeline.

#### **4.3.1 Analysis of the Reset and Evaluate Phase**

As shown in Figure 4.1, the internal phase of a stage  $i$  can be determined by observing signal  $O_i$ . When  $O_i = 0$ , stage  $i$  is in the reset phase. Otherwise, it is in the evaluate phase. Assume there are  $n$  stages in the pipeline. Since the evaluate phase of stage  $n$ , which is the last pipeline stage, does not depend on the reset phase of another stage, its reset and evaluate phase tend to have the same duration:

$$d(E_n) = d(R_n) = \frac{P_n}{2} \quad (4.1)$$

Figure 4.8 shows the waveforms of the stage outputs and the phase of stage 13, 14, 15 and 16 in a 16-stage prototype P-SRSL pipeline. It is clear that the reset and evaluate phase of stage 16 have the same duration (i.e.,  $d(E_{16}) = d(R_{16})$ ). However, this is not true for other stages. The equal duration of the reset and evaluate phase on the right side of the pipeline can be explained by considering stage 4 in Figure 4.1 in which the reset loop oscillates without waiting on any incoming signal since stage 4 is the last stage in the pipeline. However, the evaluate phase of stage  $n-1$  has to wait on the arrival of the reset phase from stage  $n$  to the latch-enabling AND gate in order for data to flow from the former to the latter. This has the effect of stretching the duration of the evaluate phase of stage  $n-1$ :

$$d(E_{n-1}) > d(E_n) \quad (4.2)$$

In Figure 4.1, it is clear that the evaluate phase of any stage  $i$  of type  $B$ ,  $0 < i < n$ , has to wait on the arrival of the reset phase from stage  $n$  while the evaluate phase of any stage  $i$  of type  $A$ ,  $0 < i < n$ , has to wait on the arrival of the evaluate phase from stage  $n$  to the latch-enabling AND gate in order for data to flow from stage  $i$  to stage  $i+1$ . This has the effect of stretching the duration of the evaluate phase of stage  $i$  compared to stage  $n$  as shown in Figure 4.8:

$$d(E_i) > d(E_n) \quad (4.3)$$

Since stage  $n$  starts its reset phase somewhat earlier, it tends to complete this phase also earlier, thus causing the reset phase of stage  $n-1$  to be somewhat shorter.

$$d(R_{n-1}) < d(R_n) \quad (4.4)$$

In fact, it is clear from Figure 4.8 that the reset phase of any stage  $i$ ,  $0 < i < n$ , is shorter than the reset phase of stage  $n$ :

$$d(R_i) < d(R_n) \quad (4.5)$$

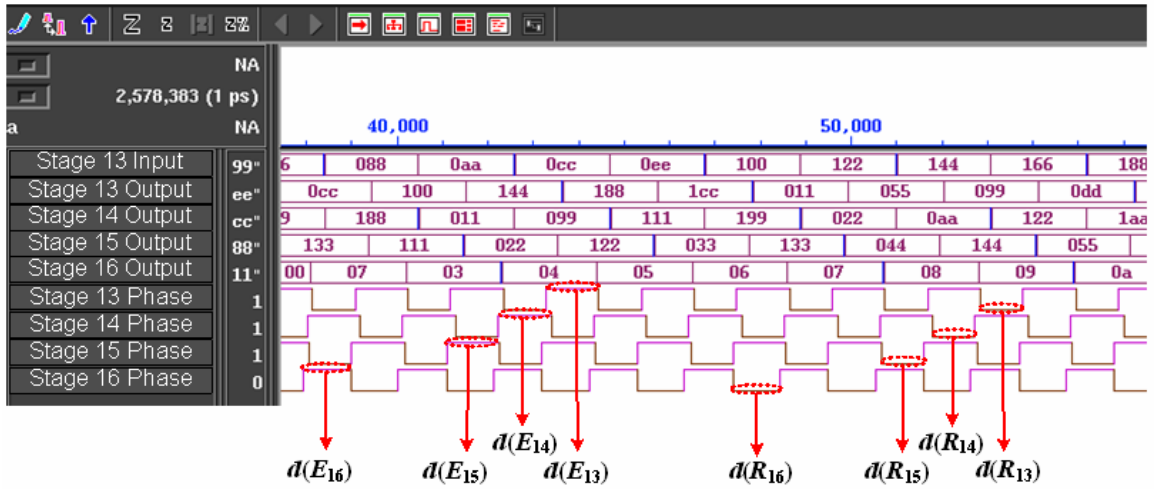


Figure 4.8: Simulation snapshot of stages 13, 14, 15 and 16 in a 16-stage prototype P-SRSL pipeline.

The increase in the evaluate phase and the decrease in the reset phase of stage  $i$ ,  $0 < i < n$ , with regard to the phases of stage  $n$  is exactly the same:

$$d(E_i) - d(E_n) = d(R_n) - d(R_i) \triangleq \delta \quad (4.6)$$

This equal increase and decrease is due to the fact that the period is equal for all stages in the pipeline:

$$P_i = P_n = P \quad (4.7)$$

### 4.3.2 Effect of $\delta$ on the Pipeline Stages

The  $\delta$  delay difference is caused by the unequal lengths of the reset loop on which the phase signals travels in stage  $n$  and  $i$ . While the phase signal in stage  $n$  starts from the left NOR gate, passes through the buffer delay, and back to the same NOR gate, the phase signal in stage  $n-1$  crosses the same path in addition to an inverter and an AND gate. The AND gate with one inverted input is the latch enabling gate between stage  $n-1$  and  $n$ . Since the phase signal travels along this augmented path in stage  $n-1$  twice, once when  $O_{n-1} = 1$  and once when  $O_{n-1} = 0$ , the  $\delta$  delay difference between the two paths in both stages is at most equal to twice the delay of the inverter and latch enabling AND gate. Let  $d(\text{INV})$  and  $d(\text{AND})$  be the average delay through an inverter and an AND gate respectively, then:

$$2(d(\text{AND})) \leq \delta \leq 2(d(\text{INV}) + d(\text{AND})) \quad (4.8)$$

$\delta$  propagates from stage  $n$  to any stage  $i$ ,  $0 < i < n$ , causing in the process the duration of the evaluate and reset phase of each stage  $i$  to increase and decrease by  $\delta$  respectively with regard to stage  $n$ :

$$d(E_i) = d(E_n) + \delta \quad (4.9)$$

$$d(R_i) = d(R_n) - \delta \quad (4.10)$$

Simulation experiments show that this delay difference is present in each stage before the last stage in the pipeline.

### 4.3.3 Effect of the Period on the Latch Enable

With regard to stage  $i$  and  $n$ , it follows from equation (4.6) that:

$$\begin{aligned}
 d(E_i) - d(E_n) &= \delta \\
 d(E_i) &= d(E_n) + \delta \\
 d(E_i) &= \frac{P}{2} + \delta \\
 d(E_i) &= d(R_n) + \delta
 \end{aligned} \tag{4.11}$$

This implies

$$d(E_i) > d(R_n) \tag{4.12}$$

Let  $d(L_i^+)$  be the minimum duration at logic level 1 of the enable of the latch between stage  $i-1$  and  $i$ . Since the latch between stage  $n-1$  and  $n$  is enabled when the former is in the evaluate phase and the latter is in the reset phase, the duration of the latch enable depends primarily on that of the reset phase of stage  $n$ . Because the duration of reset phase and evaluate phase are equal in stage  $n$ :

$$\begin{aligned}
 d(L_n^+) &= d(R_n) = d(E_n) \\
 d(L_n^+) &= \frac{P}{2}
 \end{aligned} \tag{4.13}$$

If stage  $i$  is of type  $A$ , the duration of the enable of latch  $i$ , namely  $d(L_i)$ , depends primarily on the duration of the reset phase of stage  $n$ . On the other hand, if stage  $i$  is of type  $B$ , the duration of the enable of latch  $i$  depends primarily on the duration of the evaluate phase of stage  $n$ :

$$d(L_i^+) = d(R_i) + \delta = d(R_n) = \frac{P}{2}, \quad \text{stage } i \text{ is of type } A \quad (4.14)$$

$$d(L_i^+) = d(E_i) - \delta = d(E_n) = \frac{P}{2}, \quad \text{stage } i \text{ is of type } B \quad (4.15)$$

A faster pipeline can be realized by reducing  $P$ , which requires (i) faster latches or (ii) a faster reset network within each stage. The latter can be realized by reducing the delay in the reset network of a stage.

#### 4.3.4 Area Cost

The same comparison used to contrast S-SRSL pipeline with clocked pipelines in section 3.4.5 can be applied to P-SRSL pipelines. Given the similarities between S-SRSL and P-SRSL pipelines, the outcome of this comparison applies in the case of P-SRSL pipelines. In the overall, the area of a P-SRSL pipeline will be higher than the area of its clocked counterpart.

#### 4.3.5 Fault Handling

In analyzing how the P-SRSL pipeline handle faults, only stuck-at faults are considered. Attention is paid to the outcomes caused by the output of the reset network of a given stage getting (i) stuck at 1, thus causing the stage to be locked in the evaluate phase, or (ii) stuck at 0 causing the stage to be locked in the reset phase.

- **Stage locked in the evaluate phase:** If the output of the reset network in a given stage  $j$  gets stuck at 1 for a time longer than  $P$ , stage  $j$  remains locked in the evaluate phase. In this case, the output of the AND gate controlling latch  $j+1$  depends on the output of the reset network of stage  $n$ . Note that the right input of the AND gates, which controls each inter-stage latch in the pipeline, is driven by the output of the reset network of the last stage (i.e., stage  $n$ ). On the other hand, the left inputs of the same AND gates are each driven by the outputs of the reset networks of each individual stage. If stage  $j+1$  is of type  $A$ , latch  $j+1$  becomes enabled when stage  $n$  enters its reset phase. However, if stage  $j+1$  is of type  $B$ , latch  $j+1$  becomes enabled when stage  $n$  enters its evaluate phase. As such, stage  $j+1$  oscillates in a normal fashion based on the oscillation of stage  $n$ . Consequently, data is transferred from stage  $j$  to stage  $j+1$  when latch  $j+1$  is enabled. Any stage after  $j$ , including stage  $j+1$ , oscillates in a normal fashion since its oscillation is exclusively based on the output of its reset network and the output of the reset network in stage  $n$ . As a result, data flows uninterrupted from stage  $j$  to  $n$ . With regard to stage  $j-1$ , it continues to oscillate in a normal fashion since its reset network is totally disconnected from the reset network of stage  $j$ . In fact, all the stages from 1 to  $j-1$  continue to behave similarly to stage  $j-1$  for the same reason. As a result, data flows uninterrupted from stage 1 to stage  $j$ . Taking into account the behavior of the stages before and after stage  $j$ , it is obvious that data can flow uninterrupted throughout the entire pipeline without missing a single data item.

- **Stage locked in the reset phase:** If the output the reset network of a stage  $j$  gets stuck at 0 for a time longer than  $P$ , stage  $j$  remains locked into the reset phase. In this case, the

output of the AND gate controlling latch  $j+1$  is forced to remain 0, thus disabling it as long as stage  $j$  is remains locked in the reset phase. As mentioned previously, stages, located on each side of stage  $j$ , continue to oscillate as expected since their individual reset networks are completely decoupled from each other and are individually driven by the oscillation of the reset network of the last stage. As a result, data flow uninterrupted from (i) stage 1 to stage  $j$ , and (ii) stage  $j+1$  to  $n$ . However, due to the disabled  $j+1^{\text{st}}$  latch, stage  $j$  acts as a barrier to the flow of data from stage 1 to stage  $j$  causing data to be overwritten in stage  $j$ . This results in the same data flowing repeatedly from stage  $j+1$  to stage  $n$ .

#### **4.4 Prototype Implementation of the P-SRSL Pipeline**

To test and validate SRSL and its use in P-SRSL pipelines, several pipeline prototypes have been implemented.

##### **4.4.1 Implementation of the Linear Pipeline**

A 16-stage four-bit pipeline was modeled in VHDL where each stage contains a four-bit ripple-carry adder. Similarly to S-SRSL pipelines, it was decided to insert an adder in each stage in order to amplify delay effects and subsequently constrain the performance of the pipeline. The corresponding netlist was generated using Synopsys Design Compiler based on a 0.25  $\mu\text{m}$  CMOS library [65, 66]. Cadence's Silicon Ensemble was used to place and route the pipeline. The pipeline fits into a frame of



89,731  $\mu\text{m}^2$  as shown in Figure 4.9 yielding a total latency of 14.40 nanoseconds and a throughput of 463.30 Megaoutputs/second based on the period of the last stage.

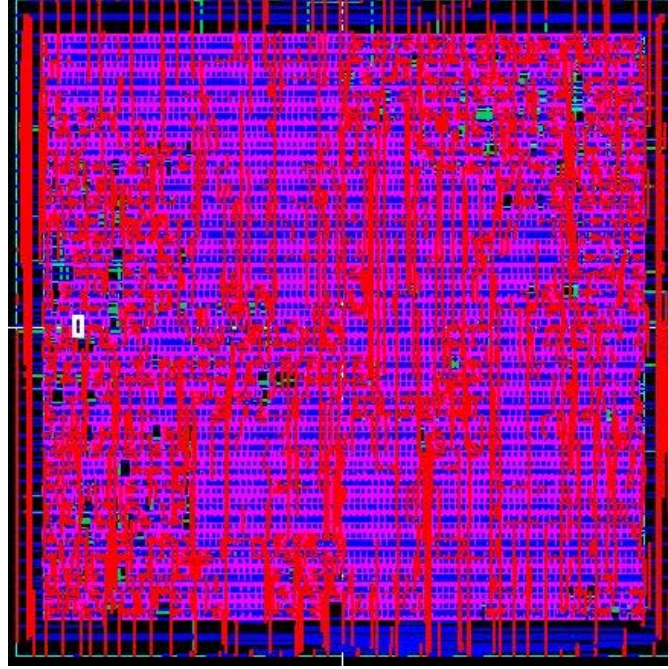


Figure 4.9: Chip layout of the four-bit 16-stage P-SRSL pipeline.

Table 4.1: P-SRSL pipeline implementation.

|                       |                                 |
|-----------------------|---------------------------------|
| Stages                | 16                              |
| Bit width             | 4                               |
| Combinational network | 4-bit adder                     |
| Synthesis             | Synopsys Design Compiler        |
| Layout                | Cadence Silicon Ensemble        |
| Simulation            | Synopsys VCS Simulator          |
| Library               | 0.25 $\mu\text{m}$ CMOS library |
| Cells                 | 1,144                           |
| Nets                  | 1,216                           |
| IO pins               | 166                             |
| Area                  | 89,731 $\mu\text{m}^2$          |
| Latency               | 14.40 ns                        |
| Throughput            | 463.30 Megaoutputs/second       |

Four parameters were measured in layout simulations of the pipeline, namely the period of each stage ( $P$ ), the duration of the evaluate phase ( $d(E_i)$ ), the reset phase of each stage ( $d(R_i)$ ), and the enable of each latch ( $d(L_i^+)$ ). Figure 4.10 shows the duration of the latch enable, the reset phase, the evaluate phase,  $\delta$  (labeled as Delta Delay), and the period across a 16-stage pipeline with a matching delay of 1.5 ns. In the figure,  $\delta$  remains constant across all stages. However, the duration of the evaluate phase of any stage located to the left of the last stage is larger than the duration phase of the last stage by  $\delta$ . In addition, the duration of the reset phase of any stage located to the left of the last stage is smaller than that of the last stage by  $\delta$ .

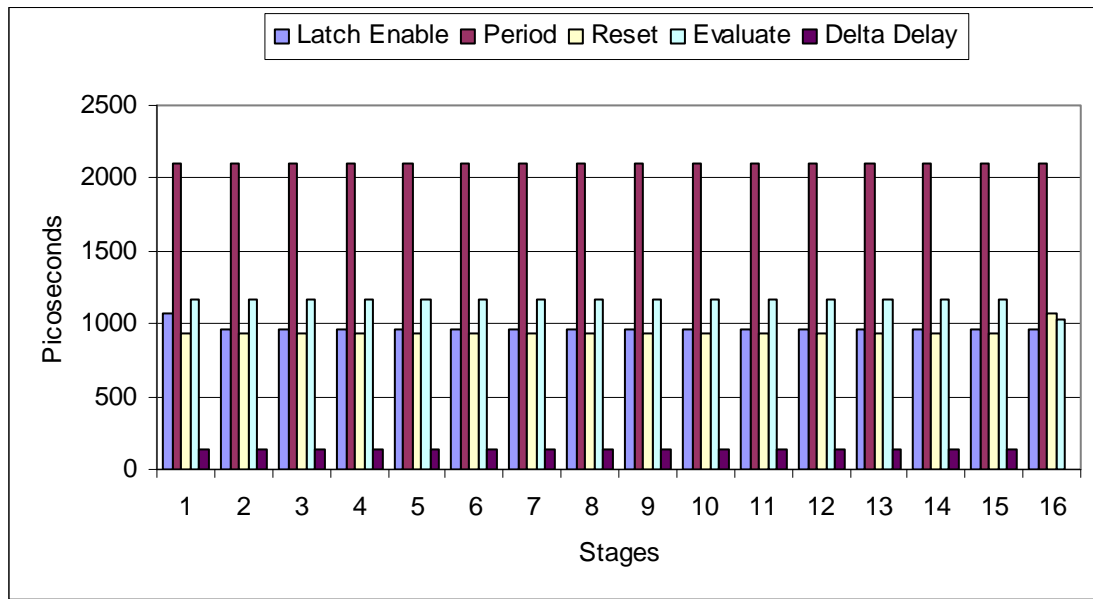


Figure 4.10: Simulation results of  $d(L^+)$ ,  $d(R)$ ,  $d(E)$ ,  $\delta$ , and  $P$  in a P-SRSL pipeline.

Both observations are expressed in equation (4.9) and (4.10). In addition, it is clear from the figure that the duration of the evaluate phase in the stages located to the left of the last stage are all equal. Similar observation can be made with regard to the duration of the reset phase in all the stages located to the left of the last stage. Both

observations are predicted by equation (4.9) and (4.10). However, the duration of the latch enable of each stage in the pipeline remains constant and is approximately equal to half of the period of each stage as predicted by equation (4.14) and (4.15). Although the matching delay inserted in the self-resetting loop of a single stage must be long enough to allow the outputs of the stage combinational network to settle, it can be reduced further by taking advantage of the overlapping of the opposite phases of two neighboring stages without disturbing the operation of the pipeline. After all, the reset phase of a stage will overlap for a brief moment with the evaluate phase of its neighbors.

#### **4.4.2 Implementation of the Non-Linear Pipelines**

To evaluate the performance of the P-SRSL non-linear pipeline, two prototype pipelines were implemented in order to study the impact of the join and fork operation on the overall performance of the pipeline.

##### **4.4.2.1 The P-SRSL Join Pipeline**

A four-bit six-stage pipeline was modeled in VHDL where each stage contains a four-bit adder as shown in Figure 4.11. The pipeline netlist was generated using Synopsys Design Compiler based on a 0.25 $\mu$ m CMOS library [65, 66]. Four parameters were measured in layout simulations of the pipeline, namely the period of each stage ( $P$ ), the duration of the evaluate phase ( $d(E)$ ), the reset phase of each stage ( $d(R)$ ), and the enable of each latch ( $d(L^+)$ ).

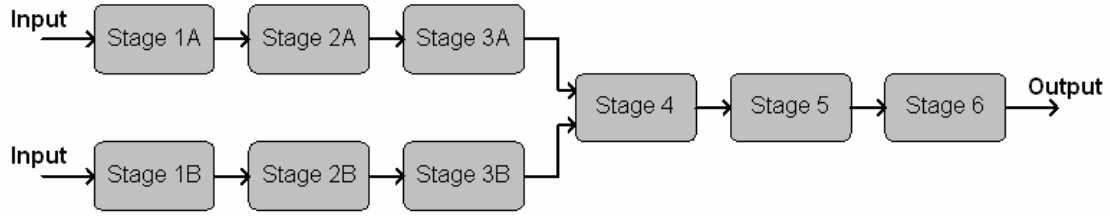


Figure 4.11: Four-bit six-stage P-SRSL join pipeline.

In order to verify the functional correctness of the P-SRSL join structure, simulation experiments were conducted on a prototype join pipeline similar to the pipeline shown in Figure 4.11. Figure 4.12 shows a simulation snapshot of only stages 3A, 3B, and 4 from the prototype pipeline of Figure 4.11.

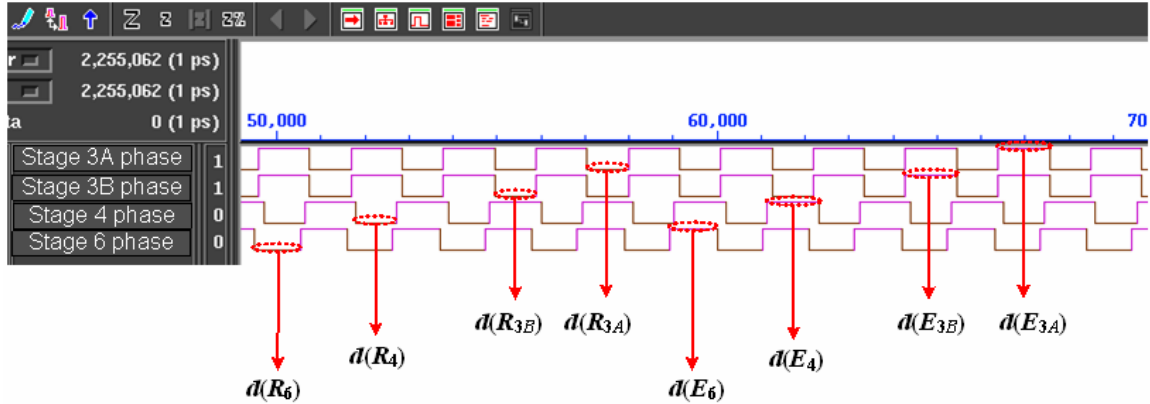


Figure 4.12: Simulation snapshot of the prototype P-SRSL join pipeline.

In Figure 4.12, the phase of stage 4 is always de-asserted when the phase of stage 3A and 3B are asserted and vice-versa. This shows that both stages 3A and 3B oscillate in the same phase while stage 4 oscillates in the opposite phase. This ensures that data flows from stages 3A and 3B to stage 4 when both the former are in the evaluate phase while the latter is in the reset phase. In addition, the phase of the last stage ( $O_6$ ) is identical to the phase of stage 4 ( $O_4$ ). If stage 6 is of type A, then stage 5 is of type B, and

consequently stage 4 is of type A. Stages of the same type will have identical phases as described in section 4.1.

Figure 4.13 shows the duration of the latch enable, the reset phase, the evaluate phase,  $\delta$ , and the period of each stage in the P-SRSL join pipeline. Note the stages numbered 1, 2, and 3 in Figure 4.13 represents the stages labeled 1A, 1B, 2A, 2B, 3A, and 3B in Figure 4.11. As the figure shows, the duration of the latch enable of each stage in the pipeline remains constant and is approximately equal to half of the period of each stage. The duration of the evaluate phase of any stage located to the left of the last stage is larger than the duration phase of the last stage by  $\delta$ , while the duration of the reset phase of any stage located to the left of the last stage is smaller than that of the last stage by  $\delta$ . These results are consistent with the findings of equations (4.9), (4.10), (4.14), and (4.15).

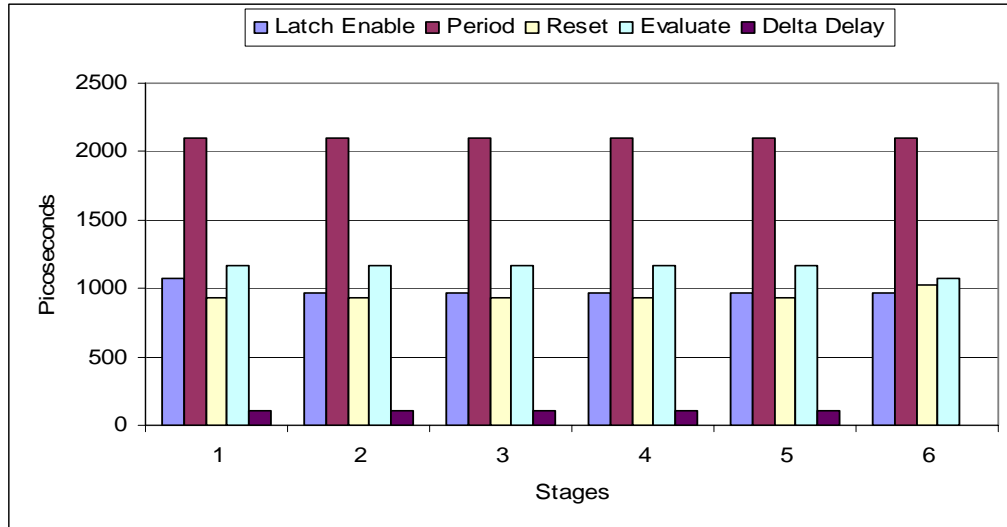


Figure 4.13: Simulation results of  $d(L^+)$ ,  $d(R)$ ,  $d(E)$ ,  $\delta$ , and  $P$  in the P-SRSL prototype join pipeline.

#### 4.4.2.2 The P-SRSL Fork Pipeline

A four-bit six-stage pipeline was modeled in VHDL where each stage contains a four-bit adder as shown in Figure 4.14. Its netlist was generated using Synopsys Design Compiler based on a 0.25 $\mu$ m CMOS library [65, 66]. Four parameters were measured in layout simulations of the pipeline, namely the period of each stage ( $P$ ), the duration of the evaluate phase ( $d(E)$ ), the reset phase of each stage ( $d(R)$ ), and the enable of each latch ( $d(L^+)$ ).

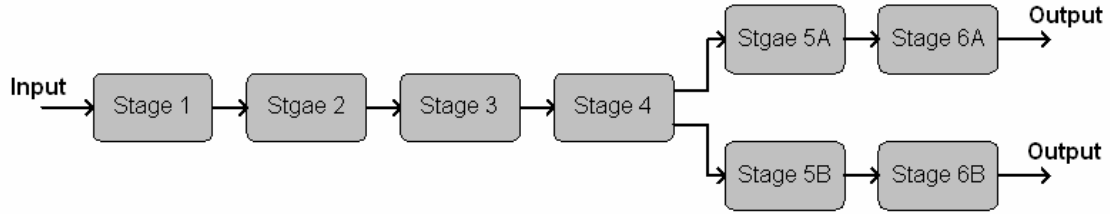


Figure 4.14: Four-bit six-stage P-SRSL fork pipeline.

In order to verify the functional correctness of the P-SRSL fork structure, simulation experiments were conducted on the prototype fork pipeline shown in Figure 4.14. Figure 4.15 shows a simulation snapshot of only stages 4, 5A, 5B, 6A, and 6B from the prototype pipeline.

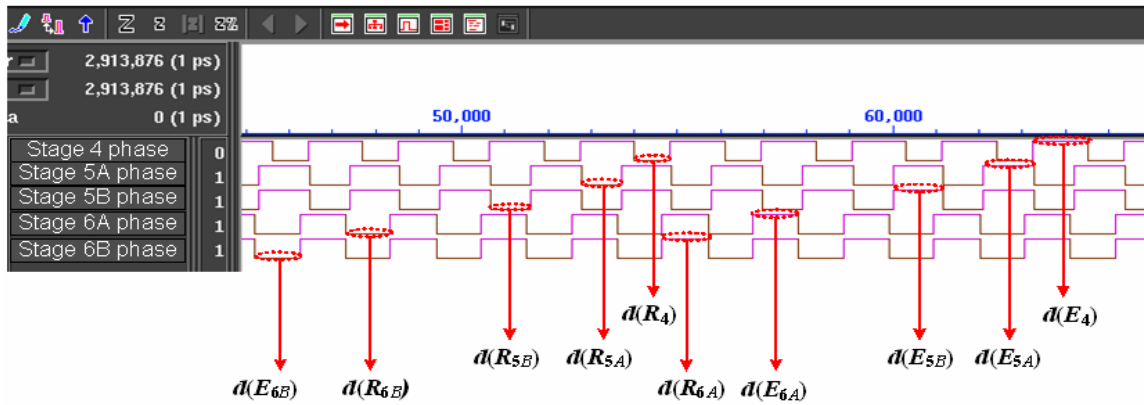


Figure 4.15: Simulation snapshot of the prototype P-SRSL fork pipeline.

In Figure 4.15, the phase of stages 5A and 5B are always de-asserted when the phase of stage 4 is asserted and vice-versa. This shows that stages 5A and 5B oscillate in the same phase while stage 4 oscillates in the opposite phase. This insures that data flows from stage 4 to stages 5A and 5B when the former is in the evaluate phase while the two latter are in the reset phase. Based on the stage characterization introduced in section 4.1, stage 6 and 4 are of the same type, and subsequently, their phases will be identical. This can be seen in Figure 4.16 by inspecting the phase signals of stage 4, 6A, and 6B.

Figure 4.16 shows the duration of the latch enable, the reset phase, the evaluate phase,  $\delta$ , and the period of each stage in the P-SRSL fork pipeline. Note that the stages numbered 5 and 6 in Figure 4.16 represent the stages labeled 5A, 5B, 6A, and 6B in Figure 4. 14. As the figure shows, the duration of the latch enable of each stage in the pipeline remains constant and is approximately equal to half of the period of each stage as expressed by equations (4.14) and (4.15). In addition, the duration of the evaluate phase of any stage located to the left of the last stage is larger than the duration of the evaluate phase of the last stage by  $\delta$  as found in equation (4.9), while the duration of the reset phase of any stage located to the left of the last stage is smaller than that of the last stage by  $\delta$  as found in equation (4.10). However, as can be seen in the figure, stage 4 has a slightly longer evaluate phase and shorter reset phase compared to other stages located on the left side of the last stage. When stage 4 transitions from the evaluate to the reset phase, the latch enables of stage 5A and 5B have to propagate through the G-labeled AND gate as shown in Figure 4.5. This has the effect of stretching the evaluate phase and shrinking the reset phase of stage 4 in particular.

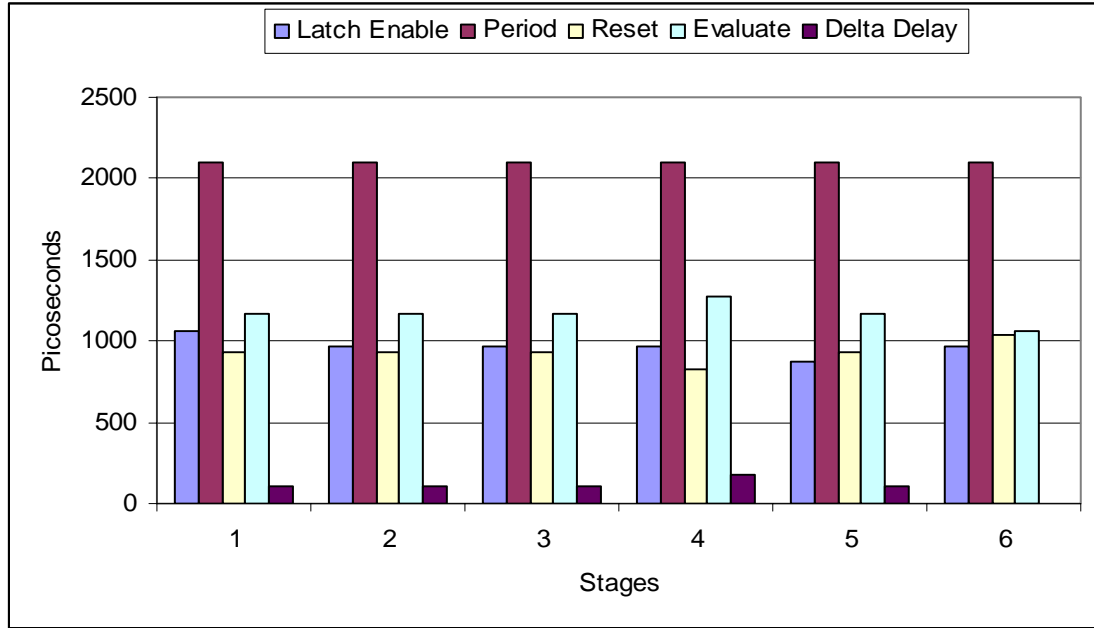


Figure 4.16: Simulation results of  $d(L^+)$ ,  $d(R)$ ,  $d(E)$ ,  $\delta$ , and  $P$  in the P-SRSL prototype fork pipeline.

#### **4.5 Comparison of P-PRSL to S-SRSL Pipelines**

Table 4.2 highlights the differences between P-SRSL and S-SRSL pipelines. It seems that the P-SRSL pipeline displays better latency and area performance than the S-SRSL pipeline. The table shows that the P-SRSL pipeline has 0.4% area reduction, 2.1% increase in the pipeline throughput. Whereas the P-SRSL has a constant duration of the latch enable, the S-SRSL pipeline has a variable duration of its latch enable. The variability in the latter depends on the pipeline location of the stage to which the latch is associated. This variability imposes a limit on the maximum number of stages in the S-SRSL pipeline. It can be conjectured that some of the mentioned performance improvements in the P-SRSL pipeline can be attributed to the fact that the  $\delta$  domino effect does not propagate across the pipeline stages. As a result, there is a general



uniformity in its timing behavior, which allows it to some degree to produce slightly faster responses.

Table 4.2: Comparison summary of the P-SRSL to S-SRSL pipeline.

| Parameter                  | P-SRSL Pipeline                      | S-SRSL Pipeline  |
|----------------------------|--------------------------------------|--|
| Period                     | 2.10 ns                              | 2.18 ns  |
| Total Latency              | 14.40 ns                             | 15.76 ns   |
| Pipeline area              | 89,731.14 $\mu\text{m}^2$            | 90,057.74 $\mu\text{m}^2$  |
| Latch Enable Duration      | 0.96 ns                              | 1.01 ns (stage 16) down to 0.64 ns (stage 1)                     |
| Throughput                 | 463.3 Megaoutputs/sec                | 453.95 Megaoutputs/sec   |
| Theoretical Pipeline Depth | No limit                             | $n = 1 + \frac{1}{\delta} \left( \frac{P}{2} - d(L_1^+) \right)$ |
| $\delta$ Delay Difference  | Between any stage and the last stage | Between any two neighboring stages                               |

## 4.6 Summary

This chapter shows how P-SRSL can be used to implement linear pipelines in addition to fork and join operations encountered in non-linear pipelines. The prototyping experiments show that the actual performance of the P-SRSL pipeline is significantly closer to its analytical performance. The timing analysis of the P-SRSL pipeline shows that the duration of the latch enable is constant for any stage in the pipeline. This is due to the fact that the  $\delta$  effect does not propagate across the pipeline stages, which in return keeps the duration of the evaluate and reset phases constant in the stages before the right stage of the pipeline. In contrast to the S-SRSL pipeline, the incremental delays caused by the propagation of  $\delta$  are completely absent in the P-SRSL pipeline. This can explain its better performance confirmed by the prototyping experiments conducted on the P-SRSL pipeline [67, 68].

## CHAPTER FIVE: DELAY TOLERANT SELF-RESETTING STAGE LOGIC PIPELINES

This chapter presents a clockless pipeline design technique, called *delay-tolerant self-resetting stage logic* (D-SRSL), which can be used to handle pipeline stages with significant delay differences. Section 5.1 introduces the two building blocks namely the phase control and latch control and then shows how they can be used as a building block in linear pipeline while section 5.2 shows the non-linear D-SRSL pipelines. Section 5.3 presents a detailed timing analysis of a linear pipeline and shows how the worst stage delay is impacting the period of the pipeline. Section 5.4 describes the implementation of three prototype pipelines while section 5.5 summarizes the chapter.

### **5.1. D-SRSL Linear Pipeline**

This section describes the various components of a D-SRSL linear pipeline and how they operate to support data flows across the pipeline.

#### **5.1.1 Pipeline Structure**

D-SRSL pipelines are supported by a clockless pipelining technique in which data flows across stages through latches as shown in Figure 5.1. These latches are controlled by a *latch control* (LC) block. Each stage oscillates between two phases: a reset and evaluate phase indicated by signal  $\phi$ . A stage is ready to absorb its inputs in the reset phase while it is ready to evaluate its inputs in the evaluate phase. The evaluation is performed by feeding the inputs to a combinational network (CN) embedded within the

stage. The control of this phase oscillation is performed by a *phase control* (PC) block, which can be reset at any moment by the reset signal  $R$ . In each stage, the CN is completely decoupled from the PC block, and can have an arbitrary delay.

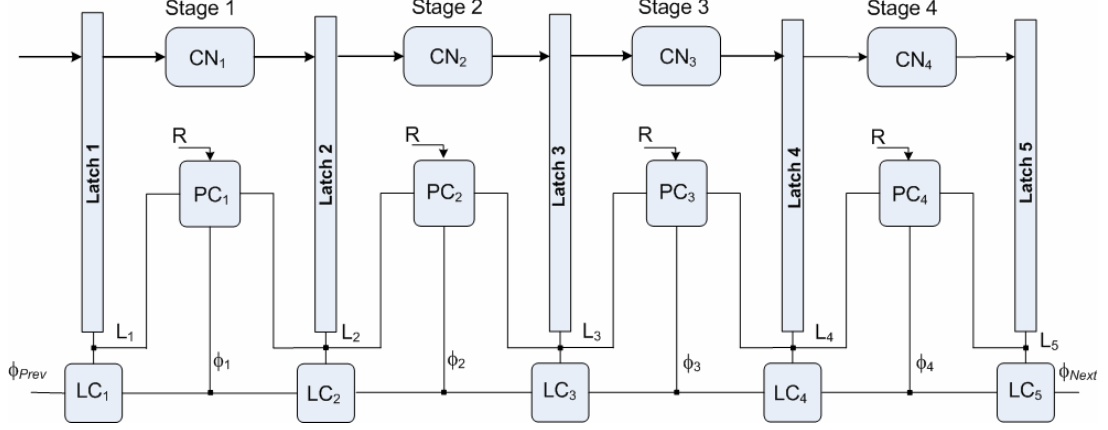


Figure 5.1: A four-stage D-SRSL pipeline.

Figure 5.1 shows the interconnection structure of a four-stage D-SRSL pipeline where each stage consists of a CN, PC and LC blocks. To insure proper data flow across stages, data is transferred from the current stage to the next one if the current stage is in the evaluate phase while the next stage is in the reset phase. Hence, the latch separating both stages is enabled when the left stage is in the evaluate while the right stage is in the reset phase. Beside the reset signal, the PC block takes as inputs the enable signal of the left and right latches and outputs the phase signal of the stage. On the other hand, the LC block takes as inputs the phases of the left and right PC blocks and outputs the signal enable of the latch it controls. Figure 5.2 shows the STG of the D-SRSL linear pipeline shown in Figure 5.1. Although the  $Clr$  signal in Figure 5.2 is not shown in Figure 1, its function within the LC block will be described in section 5.1.3. The STG shows that the rising transition of  $L_3$  occurs after  $\phi_2$  and  $\phi_3$  experience a rising and falling transition respectively. This means that latch 3 is enabled only when stage 2 is in the evaluate

phase while stage 3 is in the reset phase. Since  $L_3$  is asserted while stage 3 is in the reset phase, this guarantees that latch 4 will not be enabled until  $\phi_3$  experiences a rising transition.

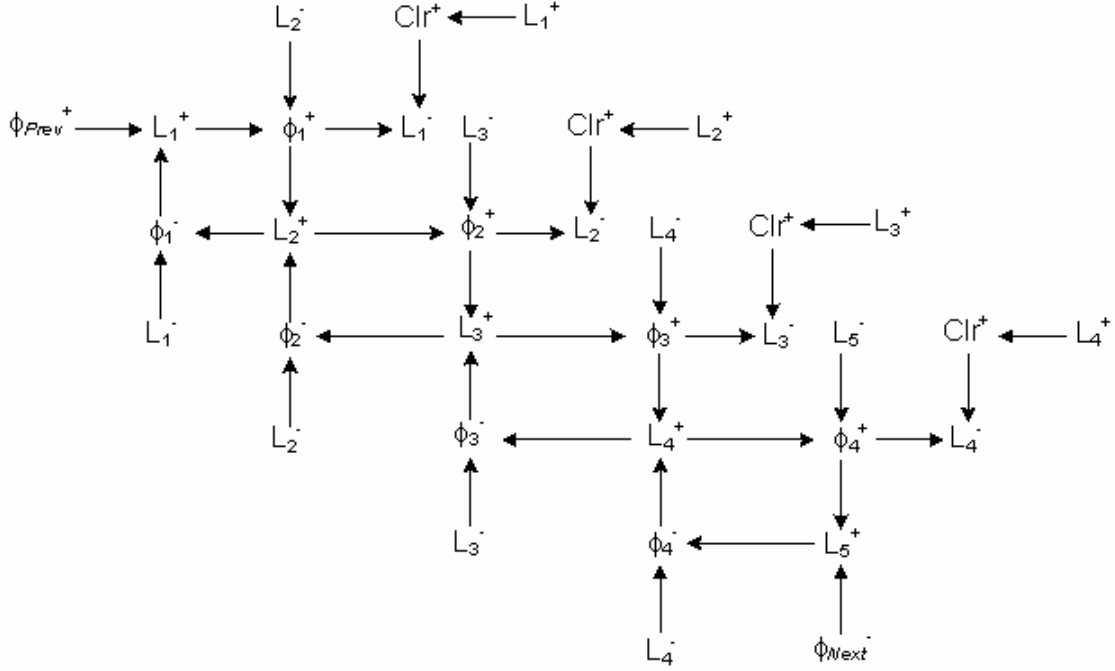


Figure 5.2: STG of the D-SRSL pipeline shown in Figure 5.1.

### 5.1.2 Phase Control Block

Figure 5.3 shows that the PC block receives three inputs: (i) the reset signal,  $R$ , which resets the PC block output to 0, (ii)  $L_i$  which is the latch enable of the left latch of stage  $i$ , and (iii)  $L_{i+1}$ , which is the latch enable of the right stage  $i+1$ . In addition, it produces an output,  $\phi_i$ , which is the phase signal of stage  $i$ . To illustrate the behavior of the PC block, Figure 5.4 shows its state graph which consists of two states: (i) the reset state,  $S_R$ , in which the phase signal becomes 0, and (ii) the evaluate state,  $S_V$ , in which the phase signal becomes 1. As shown in Figure 5.4, the PC block enters the reset state after

the reset signal is de-asserted. In this state,  $\phi_i$  is de-asserted, which indicates that the stage is in the reset phase. The PC block remains in this state as long as  $R$  and  $L_i$  are de-asserted while  $L_{i+1}$  is asserted. Once  $L_{i+1}$  is de-asserted while  $L_i$  becomes asserted, the PC block transitions to the evaluate state in which  $\phi_i$  is asserted. This means that the stage is in the evaluate phase. As long as  $L_{i+1}$  remains de-asserted, the PC block remains in the evaluate state until  $L_{i+1}$  becomes asserted, in which case the PC block returns to the reset state. As  $\phi_i$  switches back and forth, a stage can oscillate between a reset and evaluate phase in a single execution cycle or *period*. Given this oscillation, a stage is ready to absorb inputs when it is in the reset phase.

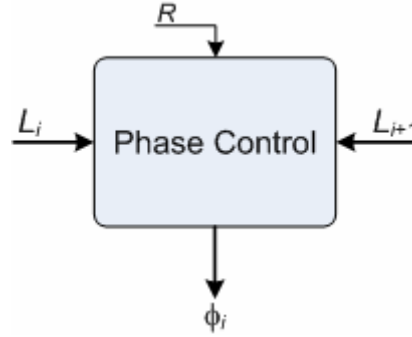


Figure 5.3.:Phase control block.

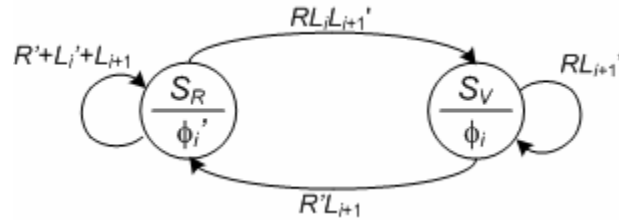


Figure 5.4: State graph of the PC block.

While the inputs are traveling along the critical path of the CN,  $\phi_i$  is similarly traveling along a path that is extended by a delay equal to the critical path delay in the CN. This extended delay is implemented by a delay buffer which delays the reset phase

long enough to allow CN outputs to stabilize. Based on this oscillation, a PC block can be embedded in a pipeline stage forcing the stage to oscillate between two phases. This oscillation can be used to synchronize data transfer between neighboring stages in a D-SRSL pipeline.

### 5.1.3 Latch Control Block

Figure 5.5 shows the block diagram of the LC block. This block has three inputs,  $\phi_i$  and  $\phi_{i-1}$ , which are the phases of the current and previous stages respectively, and the reset ( $R$ ) signal. In addition, it has one output  $L_i$ , as defined above, which feeds back into the clear port (Clr) of the LC block.  $L_i$  is the enable signal of the latch between stage  $i$  and its predecessor stage  $i-1$ . To show the behavior of the LC block, Figure 5.6 shows its state graph which consist of two states, namely the enabled state  $S_E$ , and the disabled state  $S_D$ . When the reset signal is asserted, the LC block enters the disabled state in which  $L_i$  gets de-asserted. As long as  $\phi_{i-1}$  is de-asserted while  $\phi_i$  is asserted, the block remains in the disabled state. The LC block waits until  $\phi_{i-1}$  gets asserted while  $\phi_i$  becomes de-asserted to transition to the enabled state. In this state,  $L_i$  gets asserted in order to allow the latch of stage  $i$  to capture the incoming data from stage  $i-1$ . After a delay, sufficiently long to allow the data to go through the latch, has elapsed, the latch block returns automatically to the disabled state, thus disabling the latch.

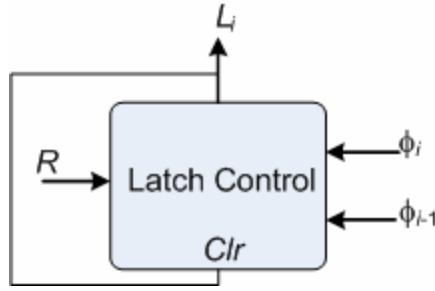


Figure 5.5: Latch control block.

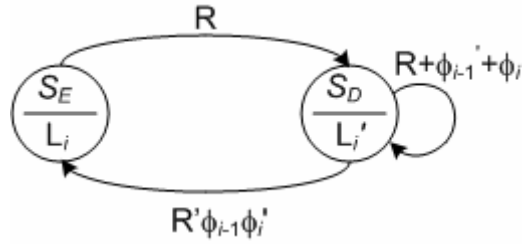


Figure 5.6: State graph of the latch control block.

## **5.2. D-SRSL Non-Linear Pipelines**

Most non-linear pipelines rely on primitives such as the fork and join operations.

In this section, the join and fork operations are described for the D-SRSL pipeline.

### **5.2.1 D-SRSL Join Pipeline**

Figure 5.7 shows a D-SRSL join pipeline. Inter-stage data flow is similar to the data flow in a linear pipeline. Data is transferred from stage A to stage C when the former is in the evaluate phase while the latter is in the reset phase. Similarly, data flows from stage B to stage C when the former is in the evaluate phase while the latter is in the reset phase. When these conditions are true, latches 3 and 4 are activated to capture the outputs of stage A and B, and feed it to the inputs of stage C. The coordination between

the stages A and B, and stage C is orchestrated by the *Join* block. Figure 5.8 shows the STG of the join structure shown in Figure 5.7. In this STG, the  $L_{Join}$  signal which drives the enable of both latches 3 and 4, experiences a rising transition when both  $\phi_A$  and  $\phi_B$  experience a rising transition while  $\phi_C$  experiences a falling transition. This shows that latches 3 and 4 are enabled when stages A and B are both in the evaluate phase while stage C is in the reset phase.

Figure 5.7: D-SRSL join pipeline.



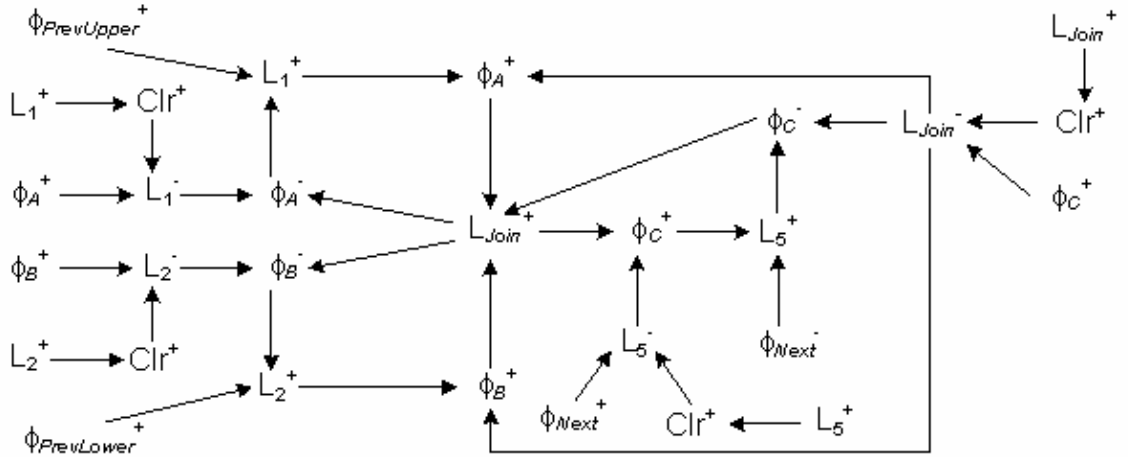


Figure 5.8: STG of the D-SRSL join pipeline shown in Figure 5.7.

As Figure 5.7 shows, the Join block takes four input signals, namely  $\phi_A$ ,  $\phi_B$ ,  $\phi_C$ , and  $R$ . In addition, it produces a single output, namely  $L_{Join}$ . Figure 5.9 shows the block diagram of the Join block while Figure 5.10 shows its state graph.

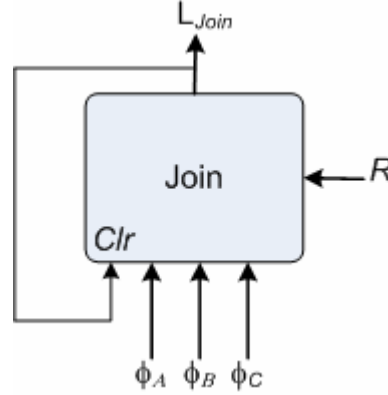


Figure 5.9: The Join block.

The Join block oscillates between two states: the disabled ( $S_D$ ) and enabled state ( $S_E$ ). The transition from the former to the latter state can occur if both  $\phi_A$  and  $\phi_B$  are asserted while  $\phi_C$  is de-asserted. In the enable state, the  $L_{Join}$  signal becomes asserted. After a delay, sufficiently long to allow the data to go through the latch, has elapsed, the

Join block returns automatically to the disabled state, thus disabling the latches 3 and 4 shown in Figure 5.7.

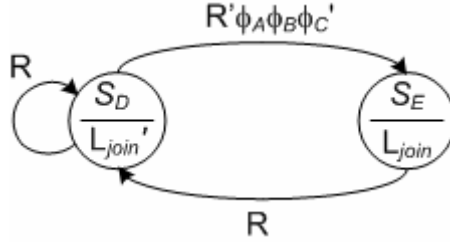


Figure 5.10: State graph of the Join block.

In order to verify the functional correctness of the D-SRSL join structure, simulation experiments were conducted on a prototype join pipeline shown in Figure 5.11. In this pipeline, each stage contains different CNs with different delays shown in parentheses in Figure 5.11. The total combinational delay through branch A is 3.9 ns while the total delay through branch B is 3 ns. The rationale behind using a different CN in each stage is to test the functional correctness of the join pipeline in the face of different delays. For purpose of clarity, Figure 5.12 shows a simulation snapshot of only stages 3A, 3B, and 4 from the prototype pipeline shown in Figure 5.11.

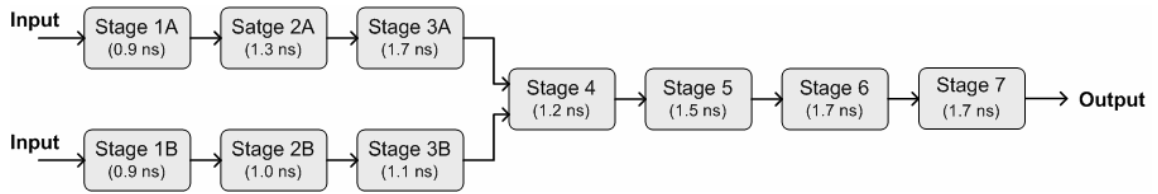


Figure 5.11: Prototype D-SRSL join pipeline.

Let  $d(E_i)$  and  $d(R_i)$  be the time duration of the evaluate and reset phase in stage  $i$  respectively. Also, let the period of stage  $i$ ,  $P_i$ , be the sum of duration of the evaluate and reset phase in stage  $i$ , namely  $P_i = d(E_i) + d(R_i)$ . Note that for each stage, the reset and

evaluate phase are indicated by logic 0 and 1 respectively. Since stage 3B has a smaller CN delay, its evaluate and reset phases should be in principle shorter than the evaluate and reset phase of stage 3A. As a result, its period should be shorter than the period of stage 3A. Although its period should be shorter, it is nevertheless extended in order to force stage 3B to wait for stages 3A and 4 to enter their evaluate and reset phases respectively. Only then, the  $L_{Join}$  signal becomes asserted as shown in Figure 5.12. When  $L_{Join}$  is asserted, it then forces stages 3A and 3B to enter their reset phase, and stage 4 to enter its evaluate phase. In general, if two branches of a join pipeline has different delays, the last stage before the join stage in the fastest branch will remain in the evaluate phase until the last stage in the slowest branch enter its evaluate phase. Thus, the Join block synchronizes both branches before computation proceeds past the join stage.

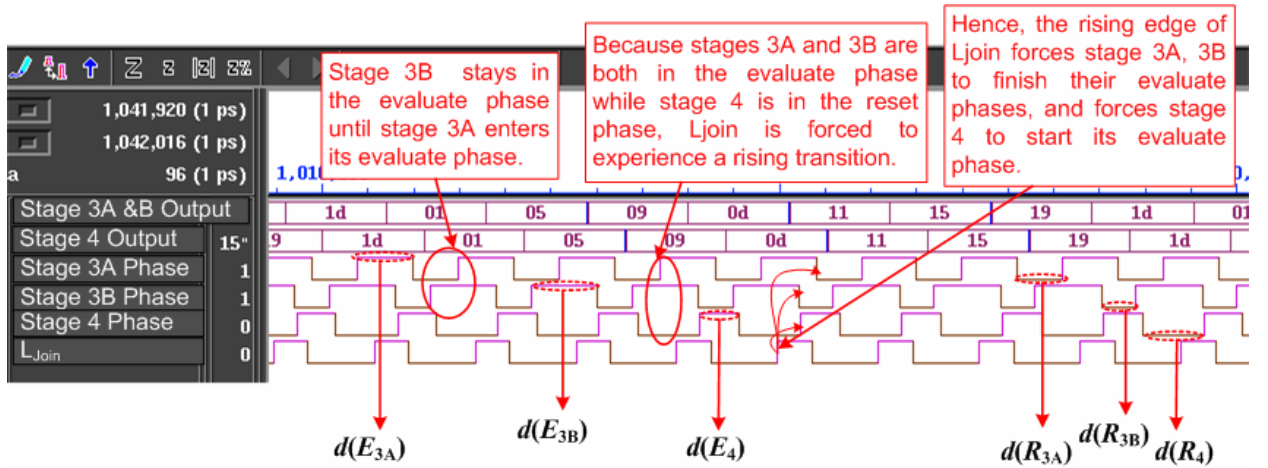


Figure 5.12: Simulation snapshot of the prototype D-SRSL join pipeline.

### 5.2.2 D-SRSL Fork Pipeline

Figure 5.13 shows a D-SRSL fork pipeline. Data is transferred from stage A to stage B and C when the former is in the evaluate phase while the two latter stages are in the reset phase. When these conditions are true, latches 2 and 3 are enabled to capture the output of stage A and feed it to stages B and C. The coordination between the three stages is orchestrated by the *Fork* block. Figure 5.14 shows the STG of the fork structure shown in Figure 5.13. In this STG,  $L_2$  experiences a rising transition when  $\phi_A$  and  $\phi_B$  experience a rising and a falling transition respectively. Similar observation can be made with regard to  $L_3$ ,  $\phi_A$  and  $\phi_C$ . Once both signals  $L_2$  and  $L_3$  experience rising transitions, so does  $L_{Fork}$ , thus forcing stage A to finish its evaluate phase while stages B and C are forced to start their evaluate phases. When  $L_{Fork}$  becomes asserted, the *Clr* signal gets asserted in return, which triggers the Fork block to transition to the disabled state.

As Figure 5.15 shows, the *Fork* block has three inputs  $L_2$ ,  $L_3$ , and  $R$ . In addition, it has one output  $L_{Fork}$ . Figure 5.16 shows the state graph of the Fork block which consist of two states, namely the enabled state  $S_E$ , and the disabled state  $S_D$ . As long as  $R$  is asserted, the Fork block remains in the disabled state. It wait until  $L_2$  and  $L_3$  become asserted to transition to the enabled state. After a delay, sufficiently long to allow the data to go through the latch, has elapsed, the Fork block returns automatically to the disabled state, thus disabling the latches 2 and 3 shown in Figure 5.13.

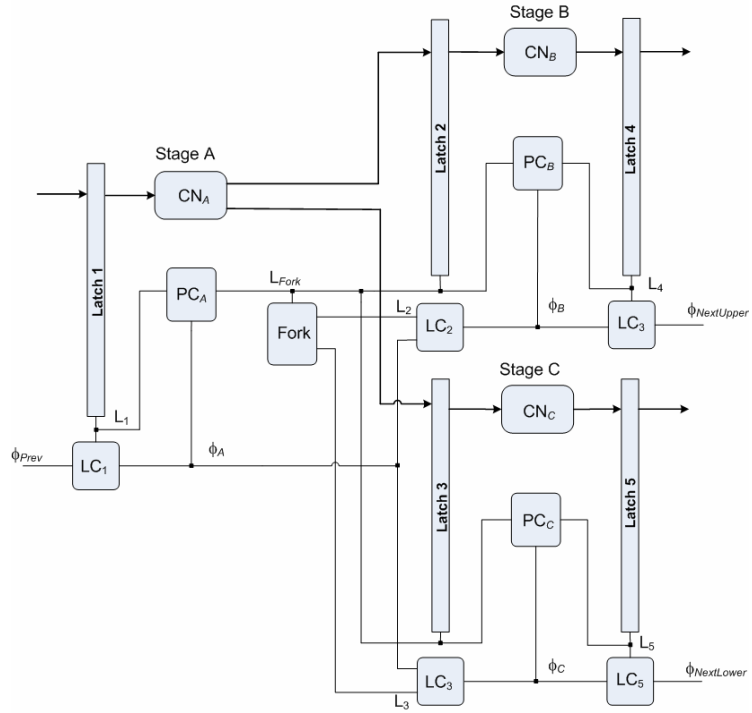


Figure 5.13: D-SRSL fork pipeline.

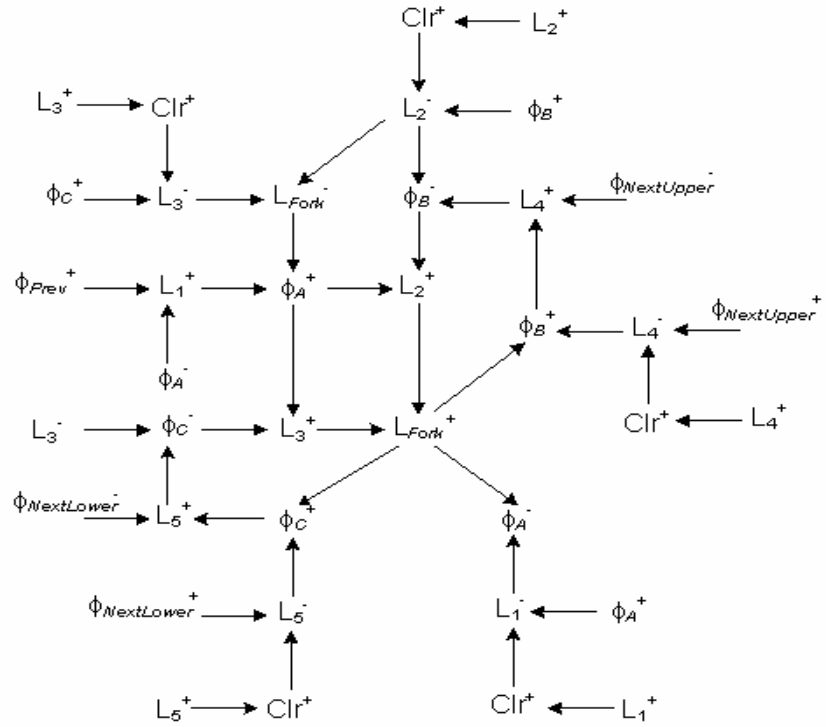


Figure 5.14: STG of the D-SRSL fork pipeline shown in Figure 5.13.

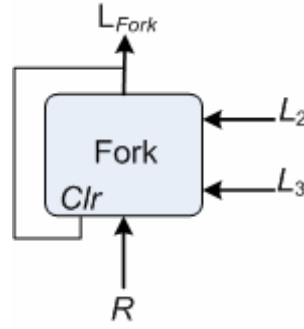


Figure 5.15: Fork block.

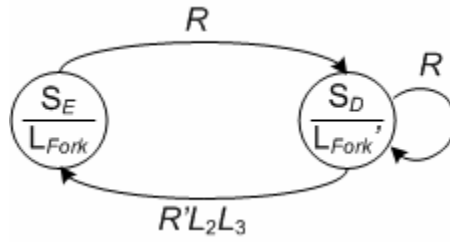


Figure 5.16: State graph of the Fork block.

In order to verify the functional correctness of the D-SRSL fork structure, simulation experiments were conducted on a prototype fork pipeline shown in Figure 5.17 in which each stage contains a CN with a different delay. The same rationale used in the simulation experiment of the join prototype pipeline is also adopted in simulating the fork structure on the fork prototype pipeline. In the prototype pipeline, the total delay of the CNs through branch A is 4.5 ns while it reaches 3.6 ns through branch B.

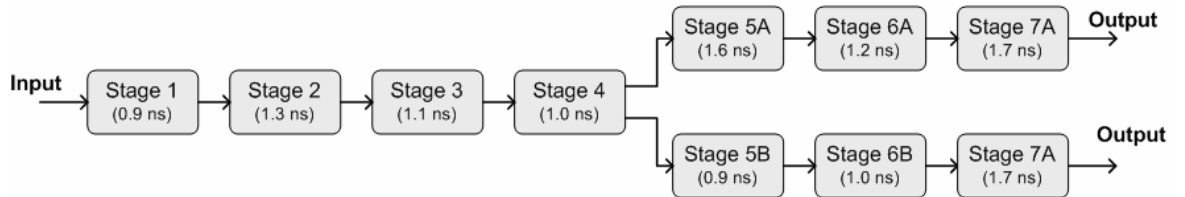


Figure 5.17: Prototype D-SRSL fork pipeline.

For purpose of illustration, Figure 5.18 shows a simulation snapshot of stages 4, 5A, and 5B for the prototype pipeline shown in Figure 5.17. Note that although the delay difference between stage 5A and 5B is quite significant, they seem to be synchronized in the way they start and complete their respective evaluate phases. As soon as the  $L_{Fork}$  experiences a rising transition, both stages 5A and 5B start their evaluate phases. As shown in Figure 5.18, stage 5A starts its evaluate phase slightly after stage 5B since its CN has a higher delay than the CN of stage 5B. After both stages 5A and 5B finish their evaluate phases, they start their reset phase. Although the CN in stage 5B has a smaller delay, its reset phase is nevertheless extended for the purpose of waiting for a rising transition on  $L_{Fork}$ , which occurs only when the latch enables of stages 5A and 5B experience rising transitions. These transitions take place only when stage 4 is in the evaluate phase while stages 5A and 5B are both in the reset phase. As a result, by delaying the rising transition of  $L_{Fork}$  until the rising transitions of the latch enables of stages 5A and 5B take place, both stages are forced to start their evaluate phases simultaneously.

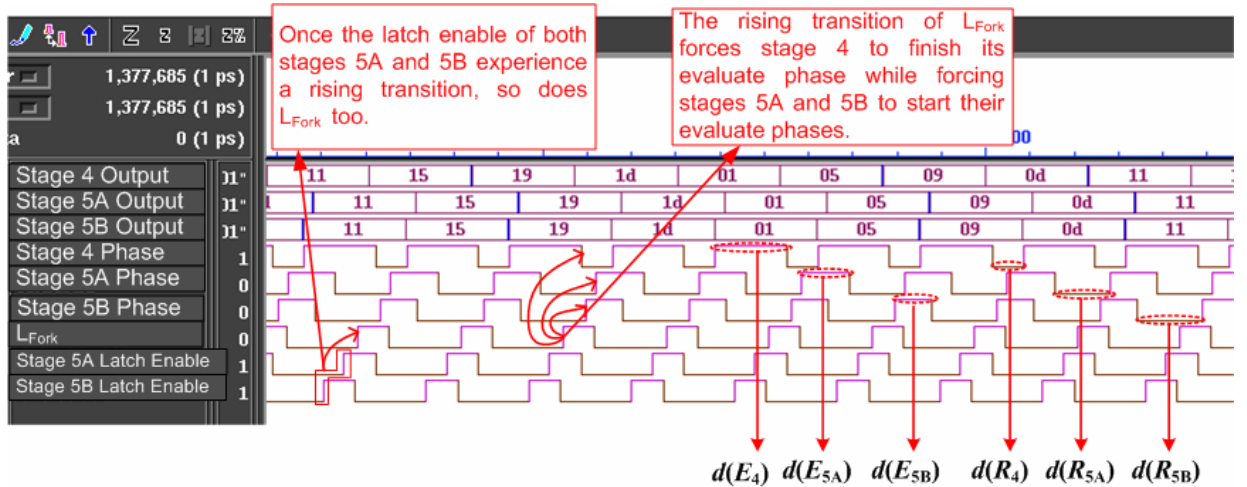


Figure 5.18: Simulation snapshot of the prototype D-SRSL fork pipeline.

### **5.3. Performance of the Pipeline**

This section starts by examining the relationships between the duration of the reset and evaluate phase in two neighboring stages of a D-SRSL pipeline and how these two parameters depend on the delays through the PC and LC blocks. This explanation is followed by a brief description of how the relationships between the duration of the reset and evaluate phases affect the duration of the latch enable in a given stage. Finally, an elaboration on how the delay of a CN embedded in a stage affects the duration of the reset and evaluate phases of a stage, based on the stage which contains the CN with the longest delay in a D-SRSL pipeline, is presented.

#### **5.3.1 The Reset and Evaluate Phase**

The phase of a stage  $i$  can be determined by observing  $\phi_i$ . When  $\phi_i = 0$ , stage  $i$  is in the reset phase. Otherwise, it is in the evaluate phase. Since the start and end of the evaluate phase of stage  $i$  depends on the rising transition of the  $L_i$  and  $L_{i+1}$  signals, the duration of the evaluate phase of any stage  $i$  is:

$$d(E_i) = t(L_{i+1}^+) - t_1(L_i^+) \quad (5.1)$$

where  $t(L_{i+1}^+)$  represents the time at which the latch enable of stage  $i+1$  experiences a rising transition while  $t_1(L_i^+)$  represents the time at which the latch enable of stage  $i$  experiences a rising transition. Note that the subscript 1 of  $t$  indicates that  $t_1(L_i^+)$  precedes  $t(L_{i+1}^+)$  in time. On the other hand, since the start and end of the reset phase in



stage  $i$  depend on the rising transitions of  $L_i$  and  $L_{i+1}$  signals, the duration of the reset phase of any stage  $i$  is:

$$d(R_i) = t_2(L_i^+) - t(L_{i+1}^+) \quad (5.2)$$

where  $t(L_{i+1}^+)$  is defined as above and  $t_2(L_i^+)$  represents the time at which the latch enable of stage  $i$  experiences a rising transition. Note that the subscript 2 of  $t$  indicates that  $t_2(L_i^+)$  succeeds  $t(L_{i+1}^+)$ . Since  $t_2(L_i^+)$  succeeds  $t(L_{i+1}^+)$ , it succeeds by transitivity  $t_1(L_i^+)$ . Figure 5.19 shows the simulation waveforms the latch enables and phases of stages 14, 15, and 16 in a 16-stage D-SRSL pipeline.

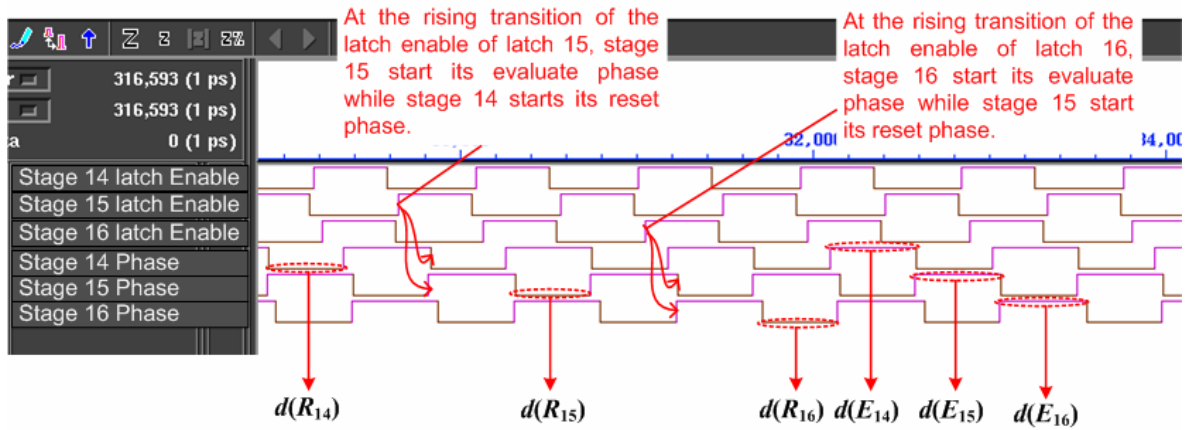


Figure 5.19: Simulation snapshot of stage 14, 15 and 16 in a 16-stage prototype D-SRSL pipeline.

To illustrate the proper operation of the pipeline based on the waveforms shown in Figure 5.19, focus is placed on how stage 15 reacts to the phases of the neighboring stages, namely stages 14 and 16. As the left callout in the figure shows, when the latch enable of stage 15 is asserted, stage 14 is in the evaluate phase while stage 15 is in the reset phase. After a short time, stage 15 enters its evaluate phase while stage 14 ends its own evaluate phase. Later, stage 15 ends its evaluate phase a short time after the latch

enable of stage 16 becomes asserted. Since both short times are almost equal, they cancel each other thus making the duration of the evaluate phase in stage 15 start when its latch enable becomes asserted, and ends when the latch enable of stage 16 becomes asserted. In essence, this validates equation (5.1). Similarly, as the right callout in the figure shows, when the latch enable of stage 16 is asserted, stage 15 is still in the evaluate phase while stage 16 is in the reset phase. A short time later, stage 15 enters its reset phase while stage 16 starts its evaluate phase. Stage 15 will remain in its reset phase until a short time after its own latch has been enabled. Since both short times are almost equal, they cancel each other thus making the duration of the reset phase of stage 15 start when the latch enable of stage 16 is asserted, and ends when the latch enable of stage 15 becomes asserted. In essence, this validates equation (5.2). Let  $D(PC_i)$  be the delay from an input port to the output port of PC block  $i$ . As Figure 5.5 shows, the LC block has a left and right input port in addition to an output port. Let  $D_{left}(LC_i)$  be the delay from the left input port to the output port of LC block  $i$ . Similarly, let  $D_{right}(LC_i)$  be the delay from the right input port to the output port of LC block  $i$ . These newly defined delays can be expressed as follows:

$$D(PC_i) = t(\phi_i^+) - t(L_i^+) \cong t(\phi_i^-) - t(L_{i+1}^+) \quad (5.3)$$

$$D_{right}(LC_i) = t(L_i^+) - t(\phi_i^-) \quad (5.4)$$

$$D_{left}(LC_i) = t(L_i^+) - t(\phi_{i-1}^+) \quad (5.5)$$

Note that  $t(\phi_i^+)$  and  $t(L_i^+)$  represent the time at which  $\phi_i$  and  $L_i$  experience rising transitions. By replacing the  $+$  with a  $-$ , the same notation can be used to indicate falling transitions. By adding the delay through the phase control block of stage  $i$  and the delay

from the left port to the output of the latch control block of latch  $i+1$ , one can determine  $d(E_i)$  as follows:

$$\begin{aligned}
D(PC_i) + D_{left}(LC_{i+1}) &= t(\phi_i^+) - t(L_i^+) + t(L_{i+1}^+) - t(\phi_i^+) \\
&= t(L_{i+1}^+) - t(L_i^+) \\
&= d(E_i)
\end{aligned} \tag{5.6}$$

Similarly, by adding the delay through the phase control block of stage  $i$  and the delay from the right port to the output of the latch control block of latch  $i$ , one can determine  $d(R_i)$  as follows:

$$\begin{aligned}
D(PC_i) + D_{right}(LC_i) &= t(\phi_i^-) - t(L_{i+1}^+) + t(L_i^+) - t(\phi_i^-) \\
&= t(L_i^+) - t(L_{i+1}^+) \\
&= d(R_i)
\end{aligned} \tag{5.7}$$

In the overall, to insure correct operation of the D-SRSL pipeline, the propagation delay through the latch of any stage  $i$ ,  $D(L_i)$ , plus the delay through the combinational network,  $D(CN_i)$ , should be less than the period of the stage  $P_i$ . As a result, a delay block  $\Delta_i$  with delay  $D(\Delta_i)$ , has to be inserted in the PC block to satisfy the following constraint:

$$\begin{aligned}
d(E_i) + d(R_i) &\geq D(L_i) + D(CN_i) \\
P_i &\geq D(L_i) + D(CN_i)
\end{aligned} \tag{5.8}$$

### 5.3.2 Duration of Latch Enable

As Figure 5.5 shows, the LC block can be reset by asserting the  $R$  signal, which can be done manually or when  $L_i$  is fed back to the  $Clr$  port of the LC block after its assertion. Let  $D_{clr}(LC_i)$  be the time elapsed between the instant in which  $Clr$  is asserted and the instant in which the latch enable  $L_i$  is de-asserted. This time lapse can be expressed as:

$$D_{clr}(LC_i) = t(L_i^-) - t(Clr^+) \quad (5.9)$$

The duration of the latch enable,  $d(L_i)$ , can be characterized based on two distinct scenarios:

(i) If  $D(CN_i) < D_{clr}(LC_i)$ ,  $L_i$  becomes de-asserted when  $\phi_i$  is asserted. In this case,

$$d(L_i) = d(R_i) \quad (5.10)$$

(ii) If  $D(CN_i) > D_{clr}(LC_i)$ ,  $L_i$  becomes de-asserted when  $Clr$  is asserted. In this case,

$$d(L_i) = D_{clr}(LC_i) \quad (5.11)$$

In brief, the duration of the latch enabled can be quantified as:

$$d(L_i) = \min\{d(R_i), D_{clr}(LC_i)\} \quad (5.12)$$

Scenario (i) represents the case in which the CN is so small that its delay is less than the delay of latch control block. In this case, the duration of the latch enable depends on the duration of the reset phase. On the other hand, scenario (ii) represents the case in which the delay through the CN is larger than the delay of the latch control block.

In this case, the duration of the latch enable depends on the delay through the latch control block.

### 5.3.3 Stage Delay and Period

To study the impact of CN delay on stage periods across the pipeline, a prototype 17-stage pipeline has been implemented in which the CNs of the stages have different delays. In this pipeline, stage 9 has the CN with the longest delay of 2.4 ns while stages 1 through 8 and 10 through 17 have randomly distributed CN delays of 0.9 ns to 2.3 ns and 0.9 ns to 2.1 ns respectively. Figure 5.20 shows a simulation snapshot of stage 7 through 11 of the 17-stage prototype pipeline in order to illustrate how the evaluate and reset phases of the stages on each side of stage 9 behave. It is clear from the figure that  $d(E_7) > d(E_9)$  and  $d(E_8) > d(E_9)$  while  $d(R_{10}) > d(R_9)$  and  $d(R_{11}) > d(R_9)$ . In fact, the duration of the evaluate phase of any stage before the worst-delay stage will be greater than the duration of the evaluate phase of the worst-delay stage. On the other hand, the duration of the reset phase of any stage after the worst-delay stage will be greater than the duration of the reset phase of the worst-delay stage. If stage  $k$  is the stage which contains the longest-delay CN, then

$$d(E_i) > d(E_k), \quad i < k \quad (5.13) \quad \text{and} \quad d(R_j) > d(R_k), \quad j > k \quad (5.14)$$

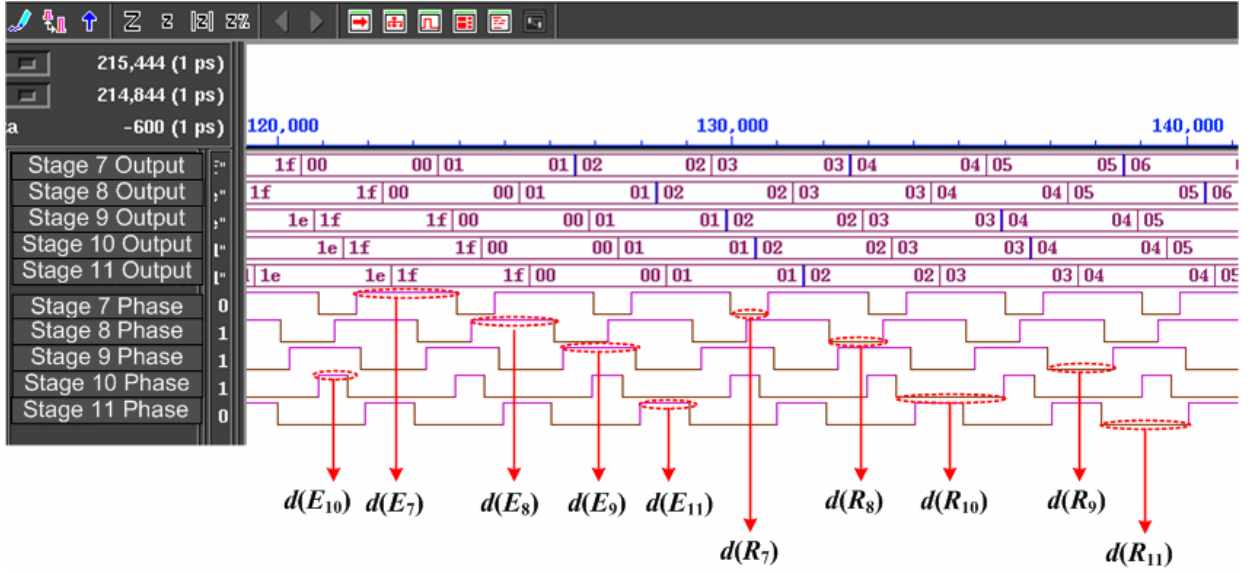


Figure 5.20: Simulation snapshot of stages 7 through 11 in a 17-stage prototype D-SRSL pipeline.

In addition, the figure shows that the period of every stage is identical in an  $n$ -stage pipeline:

$$P = P_i, \quad 1 \leq i \leq n \quad (5.15)$$

To explain how  $d(E_8) > d(E_9)$ , Figure 5.21 shows a simulation snapshot of stages 8 and 9 in the same 17-stage prototype pipeline described above. In the figure, the latch enable of stage 9 experiences a rising transition when stage 8 is in the evaluate phase while stage 9 is in the reset phase. This transition allows stages 8 and 9 to finish and start their own evaluate phases respectively. Since stage 9 contains the longest-delay CN, it has a relatively longer evaluate and reset phases. The long evaluate phase of stage 9 delays the onset of its own reset phase, which in return delays the rising transition of its own latch enable. As a result, the evaluate phase of stage 8 is stretched further as it waits for the rising transition on the latch enable of stage 9, even though the CN delay in stage

8 is smaller than the CN delay in stage 9. This explains equation (5.13). Initially, when the pipeline starts operating, the stretching of the evaluate phase of stage 8 is somewhat smaller as shown in the leftmost callout in Figure 5.21. After the first pipeline throughput, the pipeline reaches a steady state in which the stretching of the evaluate phase of stage 8 is at its maximum as shown in the rightmost callout in Figure 5.21. In general, any stage before stage 9 will not be able to finish its evaluate phase until its own successor stage finishes its own evaluate phase.

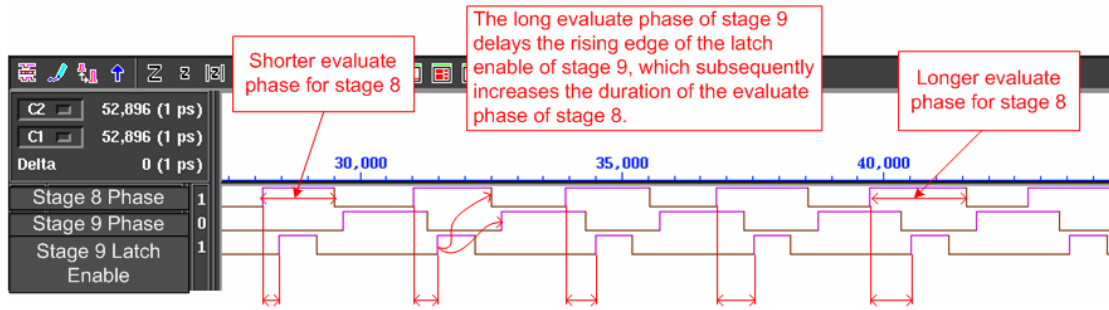


Figure 5.21: Simulation snapshot of stages 8 and 9 in the 17-stage D-SRSL prototype pipeline.

To explain how  $d(R_{10}) > d(R_9)$ , Figure 5.22 shows a simulation snapshot of stages 9, and 10 in the same 17-stage prototype pipeline described above. In the figure, the rising edge of the latch enable of stage 10 allows stages 9 and 10 to finish and start their own evaluate phases respectively. The evaluate phase of stage 10 will last for a slightly shorter time since its CN has a smaller delay than the CN delay of stage 9. This results in stage 10 finishing its evaluate phase and starting its reset phase before stage 9 completes its own evaluate phase. Hence, stage 10 remains in the reset phase thereby waiting for stage 9 to complete its evaluate phase, then start and complete its own reset phase. This long wait time causes a long reset phase in stage 10 which in turn delays the onset of the reset phase of stage 11. The domino effect of these delays is that every stage after stage 9

ends up with a reset phase that is longer than the reset phase of stage 9 as expressed in equation (5.14).

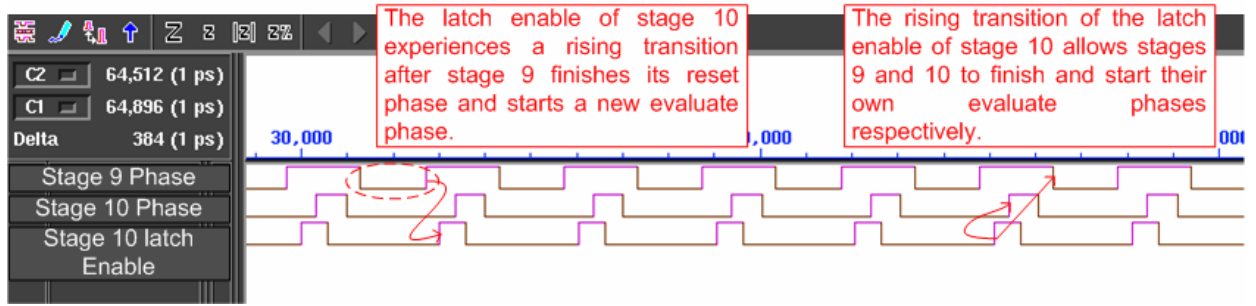


Figure 5.22: Simulation snapshot of stages 9 and 10 in the 17-stage D-SRSL prototype pipeline.

If equation (5.13) is true, it becomes possible to determine how  $d(R_i)$  relates to  $d(R_k)$ . If both sides of equation (5.13) are replaced with equation (5.8), equation (5.13) can be rewritten as follows:

$$D(L_i) + D(CN_i) - d(R_i) > D(L_k) + D(CN_k) - d(R_k) \quad (5.16)$$

Because the latches and the PC blocks are identical in all stages of the pipeline, then  $D(L_i) = D(L_k)$ . Based on this equality, the two quantities can be dropped from equation (5.16) to rewrite it as:

$$D(CN_i) - d(R_i) > D(CN_k) - d(R_k) \quad (5.17)$$

Since stage  $k$  has the worst CN delay, it follows that  $D(CN_k) > D(CN_i)$ . Given this remark, equation (5.17) remains valid only if:

$$d(R_i) < d(R_k), \quad i < k \quad (5.18)$$



The differences between the evaluate and reset phases of any stage before stage  $k$  can be quantified as follows:

$$d(E_i) - d(E_k) = d(R_k) - d(R_i) > D(CN_k) - D(CN_i), i < k \quad (5.19)$$

Similar reasoning can be followed to characterize the evaluate phases of the stages after stage  $k$ . In this case, both sides of equation (5.14) can be replaced with equation (5.8) as follows:

$$D(L_j) + D(CN_j) - d(E_j) > D(L_k) + D(CN_k) - d(E_k) \quad (5.20)$$

Because the latches and the PC blocks are identical in all stages of the pipeline, then  $D(L_j) = D(L_k)$ . Based on this equality, the two quantities can be dropped from equation (5.20) to rewrite it as:

$$D(CN_j) - d(E_j) > D(CN_k) - d(E_k) \quad (5.21)$$

Since stage  $k$  has the worst CN delay, it follows that  $D(CN_k) > D(CN_j)$ . Given this remark, equation (5.20) remains valid only if:

$$d(E_j) < d(E_k), j > k \quad (5.22)$$

The differences between the evaluate and reset phases of any stage after stage  $k$  can be quantified as follows:

$$d(E_k) - d(E_j) = d(R_j) - d(R_k) > D(CN_k) - D(CN_j), j > k \quad (5.23)$$

### 5.3.4 Area Cost

To assess the area cost of D-SRSL pipelines, they are briefly compared to clocked pipelines. While the latter require only flip-flops between pipeline stages, D-SRSL pipelines require inter-stage latches in addition to intra-stage PC blocks, which contain delay buffers, and LC blocks. Since both blocks are in essence small state machines, their area is more than marginal. In fact, the implementation of the PC blocks require three NAND gates, one AND gate, and one inverter while the implementation of the LC block requires one AND gate, one OR gate, one inverter, and one D flip-flop. Within a single stage, both blocks can consume the equivalent of eight gates and one flip-flop in addition to the delay block whose area can be proportional to the critical path delay of the intra-stage logic. Given this area overhead, it is obvious that D-SRSL pipelining is suitable for coarse-grain logic in general, and shallow and wide logic in particular. In any case, the area cost of D-SRSL pipelines is clearly greater than the area cost of clocked pipelines.

### 5.3.5 Fault Handling

Similarly to the analysis elaborated on S-SRSL and P-SRSL pipelines, only stuck-at faults are considered based on whether a given stage gets stuck in the evaluate or reset phase.

- **Stage locked in the evaluate phase:** If the output of the PC block of a given stage  $j$  gets stuck at 1 (i.e.,  $\phi_j = 1$ ), stage  $j$  remains locked in the evaluate phase. As long as  $\phi_j$  is equal to 1,  $LC_j$  block remains in the disabled state. This in turn forces  $L_j$  to switch to 0, thus disabling latch  $j$ . As a result, data is prohibited from passing from stage  $j-1$  to  $j$ . After  $L_j$  switches to 0, this forces  $PC_{j-1}$  block to transition to the enabled state, thus forcing stage  $j-1$  into the evaluate phase. Since  $\phi_{j-1}$  remains equal to 1, it triggers the same sequence of responses in  $LC_{j-1}$  block,  $L_{j-1}$ , and latch  $j-1$  thus locking stage  $j-2$  into the evaluate phase. This phenomenon propagates leftward from stage  $j$  to stage 1 of the pipeline locking every stage from 1 to  $j$  into the evaluate phase. As a result, data flow is completely stopped in this segment of the pipeline. When stage  $j$  remains locked in the evaluate phase, this allows  $LC_{j+1}$  block to transition to the enabled state, which in turn enables latch  $j+1$ . As a result, data flows between stage  $j$  and  $j+1$ . After  $L_{j+1}$  becomes equal to 1, it allows  $PC_{j+1}$  block to transition to the enabled state thus forcing stage  $j+1$  into the evaluate phase. This in turn allows  $LC_{j+2}$  block to transition to the enabled state after which stage  $j+1$  and  $j+2$  enter the reset and evaluate phase respectively. The former remains in the reset phase as long as  $L_{j+1}$  is equal to 0 due to the fact that stage  $j$  is stuck in the evaluate phase. The same sequence of events occurs between stage  $j+2$  and  $j+3$  resulting in stage  $j+2$  being stuck in the reset phase. This phenomenon propagates rightward locking every stage from  $j$  to  $n$  into the reset phase. Whereas data flows uninterrupted from stage 1 to  $j$  for one period before each stage before stage  $j$  get locked in the reset phase, its flow is completely blocked from stage  $j+1$  to  $n$ .

- **Stage locked in the reset phase:** If the output of the PC block of stage  $j$  gets stuck at 0 (i.e.,  $\phi_j = 0$ ), stage  $j$  remains locked in the reset phase. Since  $\phi_j$  is equal to 0,  $LC_j$  block transitions to the enabled state. This in turn forces  $L_j$  to switch to 1, thus enabling latch  $j$ . As a result, data is allowed to flow from stage  $j-1$  to  $j$ . After  $L_j$  switches to 1,  $PC_{j-1}$  block transitions to the disabled state forcing stage  $j-1$  into the reset phase. After stage  $j-1$  enters the reset phase,  $LC_{j-2}$  block transitions to the enabled stage, which in turn forces  $L_{j-1}$  to switch to 1, thus enabling latch  $j-1$ . As a result, data flows from stage  $j-2$  to  $j-1$ . After  $L_{j-1}$  switches to 1,  $PC_{j-2}$  block transitions to the disabled state forcing stage  $j-2$  into the reset phase. This sequence of events occurs in every stage from  $j$  to 1 at the end of which each one of these stage remains locked in the reset phase. On the other hand, when stage  $j$  remains locked in the reset phase,  $LC_{j+1}$  block transitions to the disabled state, which in turn forces  $L_{j+1}$  to switch to 0. In this case, latch  $j+1$  is disabled which prohibits data from passing from stage  $j$  to  $j+1$ . After  $L_{j+1}$  switches to 0,  $PC_{j+1}$  block transitions to the disabled state, thus forcing stage  $j+1$  in the reset phase. After stage  $j+1$  enters the reset phase,  $LC_{j+2}$  block transitions to the disabled state, which in turn forces  $L_{j+2}$  to switch to 0. In this case, latch  $j+2$  is disabled, which prohibits data from passing from stage  $j+1$  to  $j+2$ . This phenomenon propagates rightward from stage  $j$  to  $n$  locking in its propagation all these stages in the reset phase. In the overall, data flows from stage 1 to  $j$  for a single period after which each stage before  $j$  gets locked in the reset phase. At the same time, data is completely blocked in the stages after stage  $j$ .

## **5.4 Prototype Implementation of the D-SRSL Pipeline**

This section presents the implementation details of the blocks used in the D-SRSL pipeline where the delay path of each implementation is used to illustrate how it impacts the overall delay of the pipeline. These blocks consist of the PC block, LC block, the Join and Fork blocks. Next, simulation results of three prototype pipelines and their interpretations are presented.

### **5.4.1 Implementation of the PC Block**

The PC block was modeled in VHDL, synthesized, and optimized using Synopsys Design Compiler based on a 0.25  $\mu\text{m}$  CMOS library [65, 66]. Figure 5.23 shows the synthesized netlist as a sequential circuit which implements the state machine shown in Figure 5.4. Note that this sequential circuit can be reset by an active low *Reset* signal. Since this circuit is located within the self-resetting loop embedded within a stage of a D-SRSL pipeline, its critical path becomes part of the self-resetting loop path.

In Figure 5.23, this critical path starts at the inverter I, crosses the gates N1, N2, and A, before reaching the delay block  $\Delta$ . Based on this critical path, equation (5.3) can be rewritten as:

$$D(PC_i) = t(\phi_i^+) - t(L_i^+) = D(INV) + 2D(NAND) + D(AND) + D(\Delta_i) \quad (5.24)$$

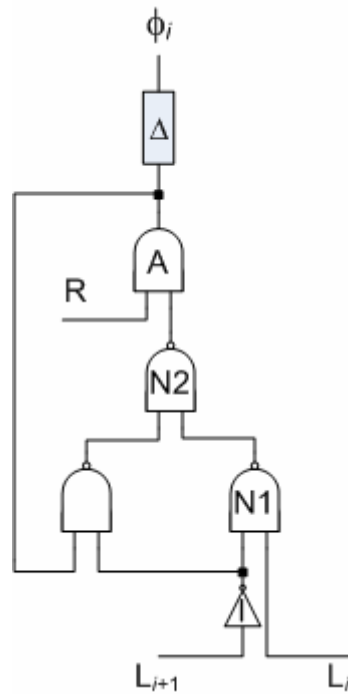


Figure 5.23: Synthesized netlist of the PC block.

### 5.4.2 Implementation of the LC Block

The LC block was modeled in VHDL, synthesized, and optimized using Synopsys Design Compiler based on a 0.25  $\mu\text{m}$  CMOS library [65, 66]. Figure 5.24 shows the synthesized netlist as a sequential machine consisting of one flip-flop and two gates. This netlist implements the sequential machine shown in Figure 5.6.

Because the LC block is part of the self-resetting loop embedded within a stage in a D-SRSL pipeline, this circuit becomes part of the self-resetting path. As a result, the path delays of this netlist add up the overall delay of the self-resetting loop. However, there are two possible paths of interest in the netlist shown in Figure 5.24. The right path

starts at the inverter I, and traverses the gates A and O. Based on this path, equation (5.4) can be rewritten as follows:

$$D_{right}(LC_i) = t(L_i^+) - t(\phi_i^-) = D(INV) + D(AND) + D(OR) \quad (5.25)$$

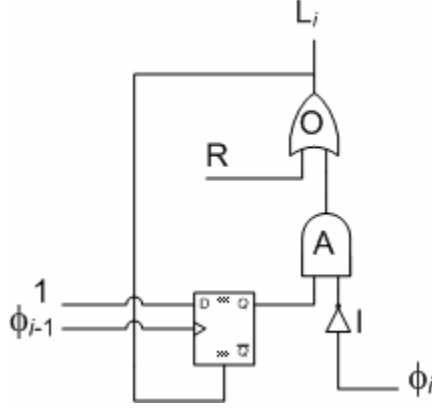


Figure 5.24: Synthesized netlist of the LC block.

On the other hand, the left path starts at the clock port of the D flip-flop, goes out the output port of the flip-flop, and traverses the gates A and O. Based on this path, equation (5) can be rewritten as follows:

$$D_{left}(LC_i) = t(L_i^+) - t(\phi_{i-1}^+) = D(\text{clk\_to\_Q}) + D(AND) + D(OR) \quad (5.26)$$

Note that in the cell library used in this implementation,  $D(INV) < D(\text{clk\_to\_Q})$ . As a result,  $D_{right}(LC_i) < D_{left}(LC_i)$  based on equations (5.25) and (5.26). From this inequality, it follows that equation (5.6) relates to equation (5.7) as follows:

$$D(PC_i) + D_{right}(LC_i) < D(PC_i) + D_{left}(LC_i) \quad (5.27)$$

By substituting the left and right sides of equation (5.27) for equations (5.6) and (5.7) respectively, equation (5.27) can be rewritten as:

$$d(R_i) < d(E_i) \quad (5.28)$$

As mentioned in section 5.2, the LC block can be reset by asserting the  $R$  signal, which can be done manually or when  $L_i$  is fed back to the  $Clr$  port of the LC block after its assertion. The resetting of the LC block follows a path which starts at the  $Clr$  port of the flip-flop, goes out the output port of the flip-flop, and traverses the gates A and O. Since  $D_{clr}(LC_i)$  denotes the delay on this path, equation (5.9) can be rewritten as follows:

$$D_{clr}(LC_i) = t(L_i^-) - t(Clr^+) = D(clr\_to\_Q) + D(AND) + D(OR) \quad (5.29)$$

### 5.4.3 Implementation of the Join Block

The Join block was modeled in VHDL, synthesized, and optimized using Synopsys Design Compiler based on a 0.25  $\mu\text{m}$  CMOS library [65, 66]. Figure 5.25 shows the synthesized netlist as a sequential machine consisting of two flip-flops and two gates. This netlist implements the sequential machine shown in Figure 5.10.

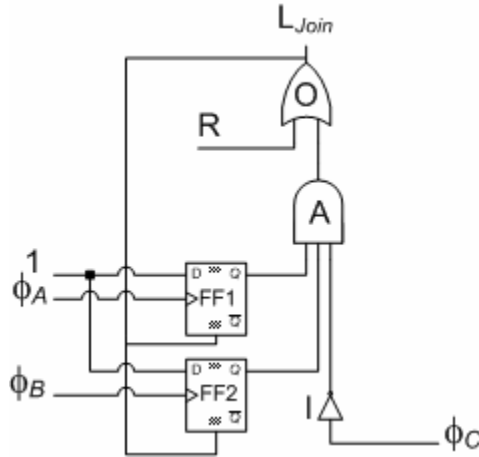


Figure 5.25: Synthesized netlist of the Join block.



In the case of a join pipeline, the Join block becomes part of the self-resetting loop embedded in each stage around the join block, namely stages A, B, and C as shown in Figure 5.7. Note that for either stage A or B in Figure 5.7, the Join block replaces both the  $LC_A$  and  $LC_B$  blocks. As a result, the delay contributed by the LC block in each stage can be replaced by the delay of the Join block. Using the same nomenclature adopted in the implementation of the LC block, the right path through the Join block, shown in Figure 5.25, starts at the inverter I, and traverses the gates A and O. Based on this path,

$$D_{right}(Join) = D(INV) + D(AND) + D(OR) \quad (5.30)$$

Note that  $D_{right}(Join) = D_{right}(LC)$ . On the other hand, the left path starts at the clock port of either D flip-flop, goes out the output port of the flip-flops, and traverses the gates A and O. Based on this path,

$$D_{left}(Join) = D(clk\_to\_Q) + D(AND) + D(OR) \quad (5.31)$$

Note that  $D_{left}(Join) = D_{left}(LC)$ . As mentioned in section 5.2.1, the Join block can be reset by asserting the  $R$  signal, which can be done manually or when  $L_{Join}$  is fed back to the  $Clr$  port of the Join block after its assertion. The resetting of the Join block follows a path which starts at the  $Clr$  port of either flip-flop, goes out the output port of the flip-flop, and traverses the gates A and O. Using the same nomenclature, the delay on this path can be expressed as follows:

$$D_{clr}(Join) = D(clr\_to\_Q) + D(AND) + D(OR) \quad (5.32)$$

Note that  $D_{clr}(Join) = D_{clr}(LC)$ .

#### 5.4.4 Implementation of the Fork Block

The Fork block was modeled in VHDL, synthesized, and optimized using Synopsys Design Compiler based on a 0.25  $\mu\text{m}$  CMOS library [65, 66]. Figure 5.26 shows the synthesized netlist as a sequential machine consisting of two flip-flops and two gates. This netlist implements the sequential machine shown in Figure 5.16.

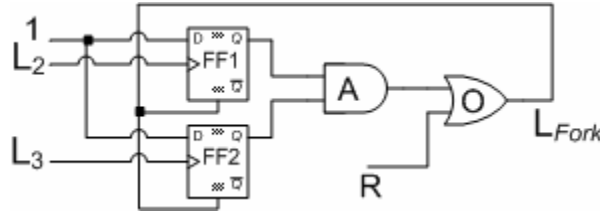


Figure 5.26: Synthesized netlist of the Fork block.

In the case of a fork pipeline, the Fork block becomes part of the self-resetting loop embedded in the stage containing the Fork block, namely stage A in Figure 5.13. Contrary to the case of the Join block in a join pipeline, the Fork block augments the path of the self resetting loop embedded in the stage containing the Fork block in a fork pipeline. As a result, the delay contributed by the Fork block can be added to the overall delay of the self-resetting loop. This delay through the Fork block starts at the clock port of either D flip-flop, goes out the output port of the flip-flop, and traverses the gates A and O. This delay can be expressed as:

$$D(Fork) = D(\text{clk\_to\_Q}) + D(AND) + D(OR) \quad (5.33)$$

Note that equation (5.6) expresses  $d(E)$  as a function of the delay of the self-resetting loop. Based on the delay path of the Fork block, equation (5.6) can be rewritten as follows:

$$\begin{aligned}
D(PC_i) + D_{left}(LC_{i+1}) + D(Fork) &= D(INV) + 2D(NAND) + D(AND) + D(\Delta_i) \\
&\quad + 2(D(\text{clk\_to\_Q}) + D(AND) + D(OR)) \\
&= 3D(AND) + 2D(NAND) + 2D(\text{clk\_to\_Q}) + 2D(OR) + D(INV) + D(\Delta_i) \\
&= d(E_i)
\end{aligned} \tag{5.34}$$

As mentioned in section 5.2.2, the Fork block can be reset by asserting the  $R$  signal, which can be done manually or when  $L_{Fork}$  is fed back to the  $Clr$  port of the Fork block after its assertion. The resetting of the Fork block follows a path which starts at the  $Clr$  port of either flip-flop, goes out the output port of the flip-flop, and traverses the gates A and O. Using the same nomenclature, the delay on this path can be expressed as follows:

$$D_{clr}(Fork) = D(\text{clr\_to\_Q}) + D(AND) + D(OR) \tag{5.35}$$

#### 5.4.5 Implementation of D-SRSL Pipeline

A 16-stage pipeline was modeled in VHDL and the corresponding netlist was generated using Synopsys Design Compiler based on a 0.25  $\mu\text{m}$  CMOS library [65, 66]. Cadence's Silicon Ensemble was used to place and route the pipeline. This pipeline displays a total latency of 15.3 ns and a throughput of 1088.14 Megaoutputs/sec based on the period of the last stage as shown in Table 5.1. Table 5.2 shows the gate area of the various blocks in a single D-SRSL stage.

Table 5.1: D-SRSL pipeline implementation.

|                            |                                 |
|----------------------------|---------------------------------|
| Stages                     | 16                              |
| Bit width                  | 5                               |
| Combinational network      | None                            |
| Synthesis                  | Synopsys Design Compiler        |
| Layout                     | Cadence Silicon Ensemble        |
| Simulation                 | Synopsys Scirocco Simulator     |
| Library                    | 0.25 $\mu\text{m}$ CMOS library |
| Latency                    | 15.3 ns                         |
| Throughput                 | 1088.14 Megaoutputs/second      |
| Stage period               | 0.916 ns                        |
| Latch enable duration      | 0.42 ns                         |
| Theoretical pipeline depth | No limit                        |

Table 5.2: Gate area of a single D-SRSL stage.

| Parameter   | Gate Cost                            |
|-------------|--------------------------------------|
| PC block    | 3 NAND gate, 1 AND gate, 1 INV       |
| Delay block | Area of the CN critical path         |
| LC block    | 1 AND gate, 1 OR gate, 1 INV, 1 D-FF |

Simulations of the pipeline were conducted in order to measure  $P$ ,  $d(E)$ ,  $d(R)$ , and  $d(L)$ . Figure 5.27 shows these four parameters in D-SRSL prototype pipeline 1. This pipeline is a 16-stage D-SRSL pipeline in which the stages are empty (i.e., they do not contain CNs). In the figure,  $d(E)$  is identical in all the stages of the pipeline. Similarly,  $d(R)$  is identical in every stage of the pipeline. However,  $d(E) > d(R)$  in any stage as expressed by equation (5.28). In addition,  $d(L)$  is almost equal to  $d(R)$  as predicted by equation (5.10) since  $D(CN_i) = 0$  for any stage in this pipeline.

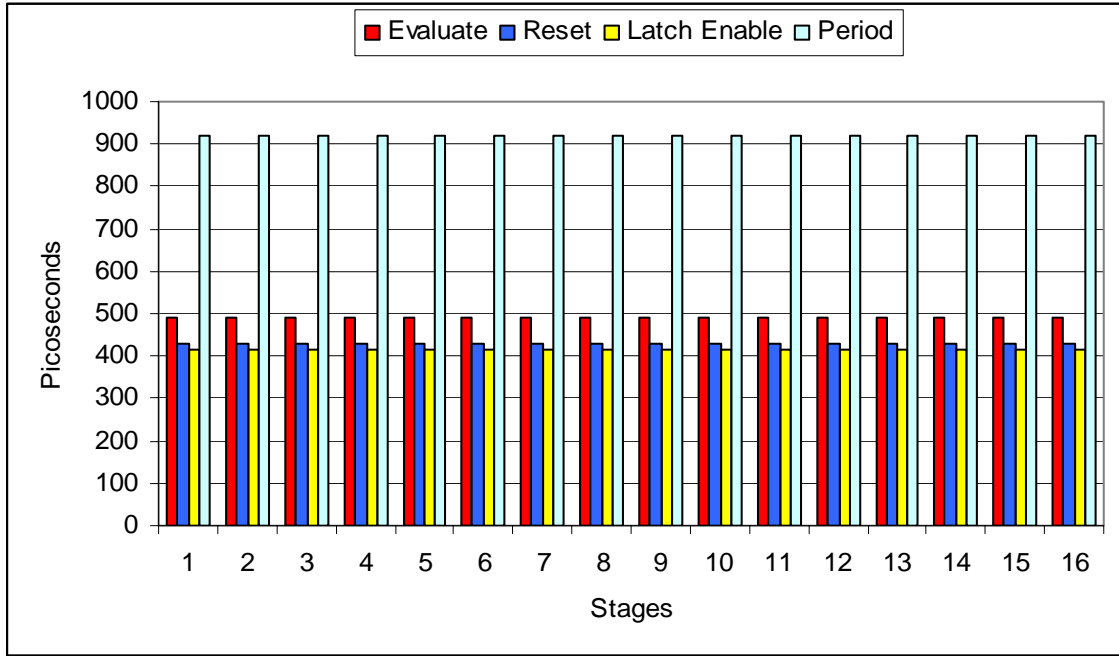


Figure 5.27: Simulation results of  $P$ ,  $d(E)$ ,  $d(R)$ , and  $d(L^+)$  in D-SRSL prototype pipeline 1.

Figure 5.28 shows these four parameters in D-SRSL prototype pipeline 2. This pipeline is a 16-stage D-SRSL pipeline where  $D(CN_i) > D_{clr}(LC_i)$  in each stage. To this end, a 0.6 ns delay CN was embedded in each stage of the pipeline. As the figure shows,  $d(L) < d(R)$ . In fact,  $d(L) = D_{clr}(LC)$  in every stage based on the value of  $D_{clr}(LC_i)$  extracted from the implementation of the LC block as shown in Figure 5.24. This validates equation (5.11).

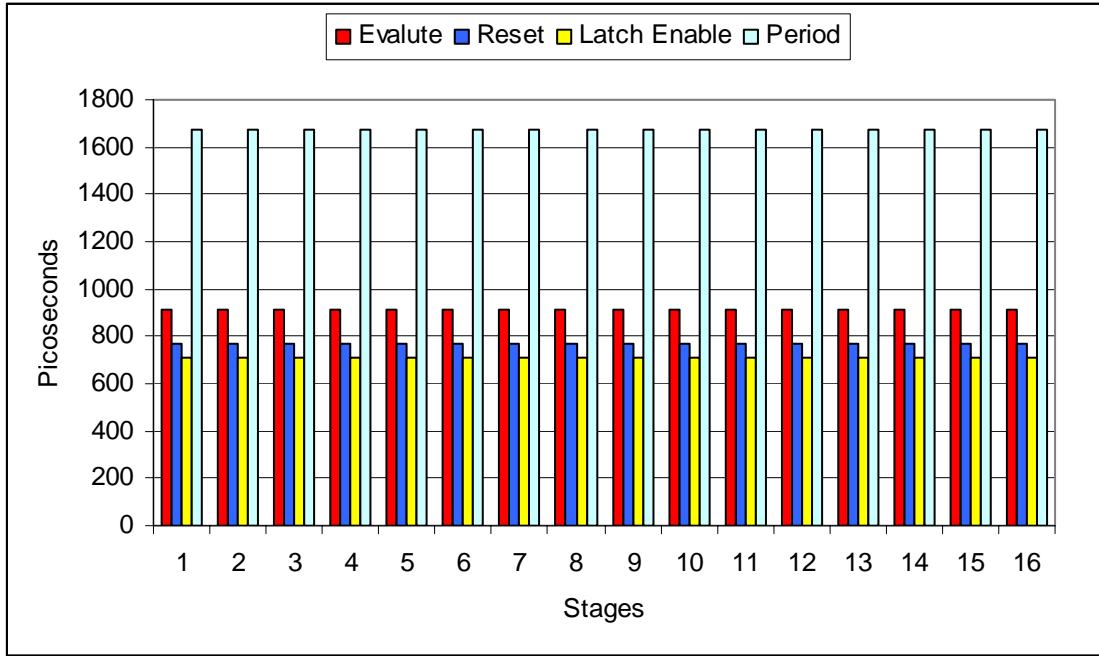


Figure 5.28: Simulation results of  $P$ ,  $d(E)$ ,  $d(R)$ , and  $d(L^+)$  in D-SRSL prototype pipeline 2.

Figure 5.29 show the implementation of a D-SRSL prototype pipeline 3. This pipeline is a 17-stage D-SRSL pipeline in which stage 9 has the longest CN delay while the remaining stages have CNs with randomly distributed delays that are smaller than the delay of the CN embedded in stage 9. It is clear from the figure that, in stage 9,  $d(E)$  is closer to  $d(R)$  than in any other stage. In the stages before stage 9,  $d(E_i) > d(E_9)$ ,  $i < 9$ , as stated in equation (5.13). This results in  $d(R_i) < d(R_9)$ ,  $i < 9$ , as predicted by equation (19). On the other hand,  $d(R_j) > d(R_9)$ ,  $j > 9$ , in the stages after stage 9 as stated in equation (5.14, which results in  $d(E_i) < d(E_9)$ ,  $j > 9$ , as predicted by equation (5.22). Note that equation (5.19) regarding the stages before stage 9, and equation (5.23) regarding the stags after stage 9, are both valid based on the simulation results of Figure 5.29. Regardless of the delay in stage 9,  $P$  is identical in all stages as is the case in pipeline 1,

2, and 3. This shows that stage 9 determines  $P$  for the remaining stages in the pipeline although these stages have smaller delays than stage 9.

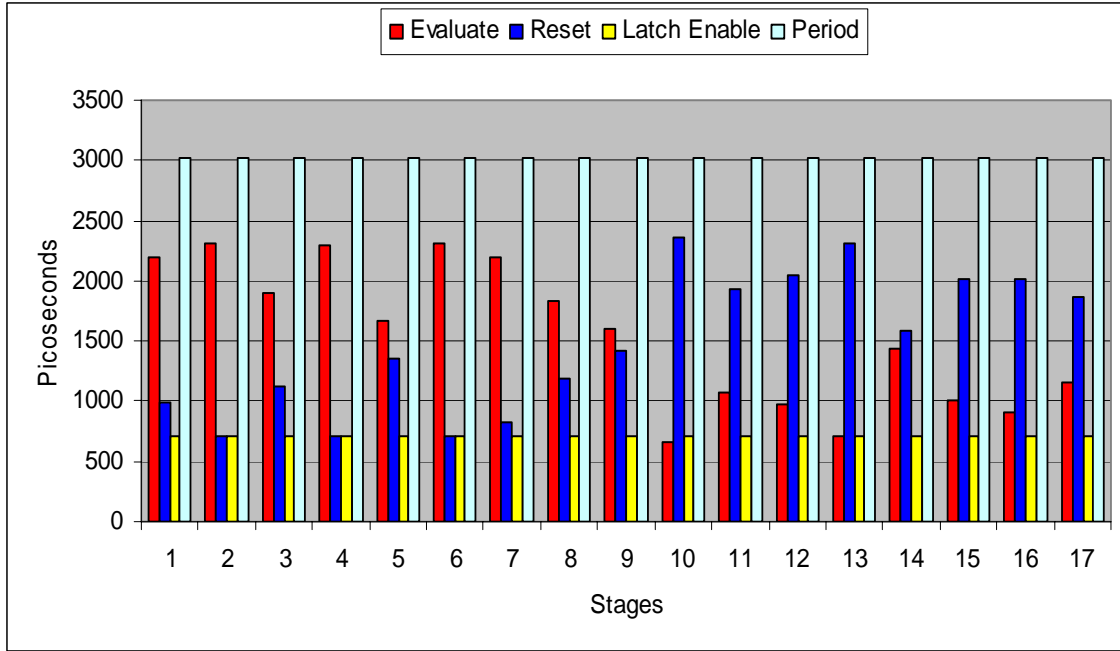


Figure 5.29: Simulation results of  $P$ ,  $d(E)$ ,  $d(R)$ , and  $d(L^+)$  in D-SRSL prototype pipeline3.

## 5.5. Conclusion

This chapter presents D-SRSL as a new clockless pipeline design technique that can handle significant delays difference between the stages of the pipeline. This capability provides a high degree of flexibility in pipelining coarse grain datapaths. In the D-SRSL approach, stages communicate with each other through their respective phases. The timing analysis showed that in the D-SRSL pipeline, it was observed that the duration of the evaluate phase and the duration of the reset phase are equal for all the stage in equal delay pipelines. However, in random delay pipelines, it was observed that

the duration of the evaluate phase increases on the left side of the worst stage delay while the reset phase duration increases on the right side of the worst stage delay. This makes the worst delay stage the stage which controls the period of the pipeline. This timing analysis is validated through experiments with three pipelines with different stage delay assumptions.



## CHAPTER SIX: SYNTHESIS OF SRSL PIPELINES

This chapter presents the proposed SRSL design methodology in section 6.1 while section 6.2 presents the synthesis of SRSL pipelines. Section 6.3 reviews the preliminary concepts used to formulate the synthesis of the SRSL pipeline synthesis problem. The modeling and the formulation of this problem is presented in section 6.4 while section 6.5 explains the proposed heuristic solution. Section 6.6 discusses the obtained experimental results for SRSL pipelines. Finally, section 6.7 gives a summary of the chapter.

### **6.1 SRSL Pipeline Design Methodology**

In order to leverage the investment spent on current commercial design tools used in clocked logic, it would make sense to adopt the same design methodology and flow supported by these tools to synthesize SRSL pipelines as argued in chapter 1. Ideally, minimum disturbance to this design methodology is highly desirable. Figure 6.1 proposes the adopted design flow for SRSL logic. In the figure, a parser extracts the clocked gate netlist in order to build a Boolean graph. Next, an SRSL pipeline synthesizer partitions the graph into stages and inserts the latches and the reset network of each stage in appropriate places inside the graph without violating performance constraints. At the end, the synthesizer produces an SRSL pipeline represented as a gate netlist. The SRSL gate netlist can be simulated with any commercial simulator. It can also be mapped onto a standard cell library using any commercial technology mapper in order to produce a cell netlist. The latter can be placed and routed using conventional physical synthesis tools by

propagating the same performance constraints used in high level synthesis to the physical synthesis tools.

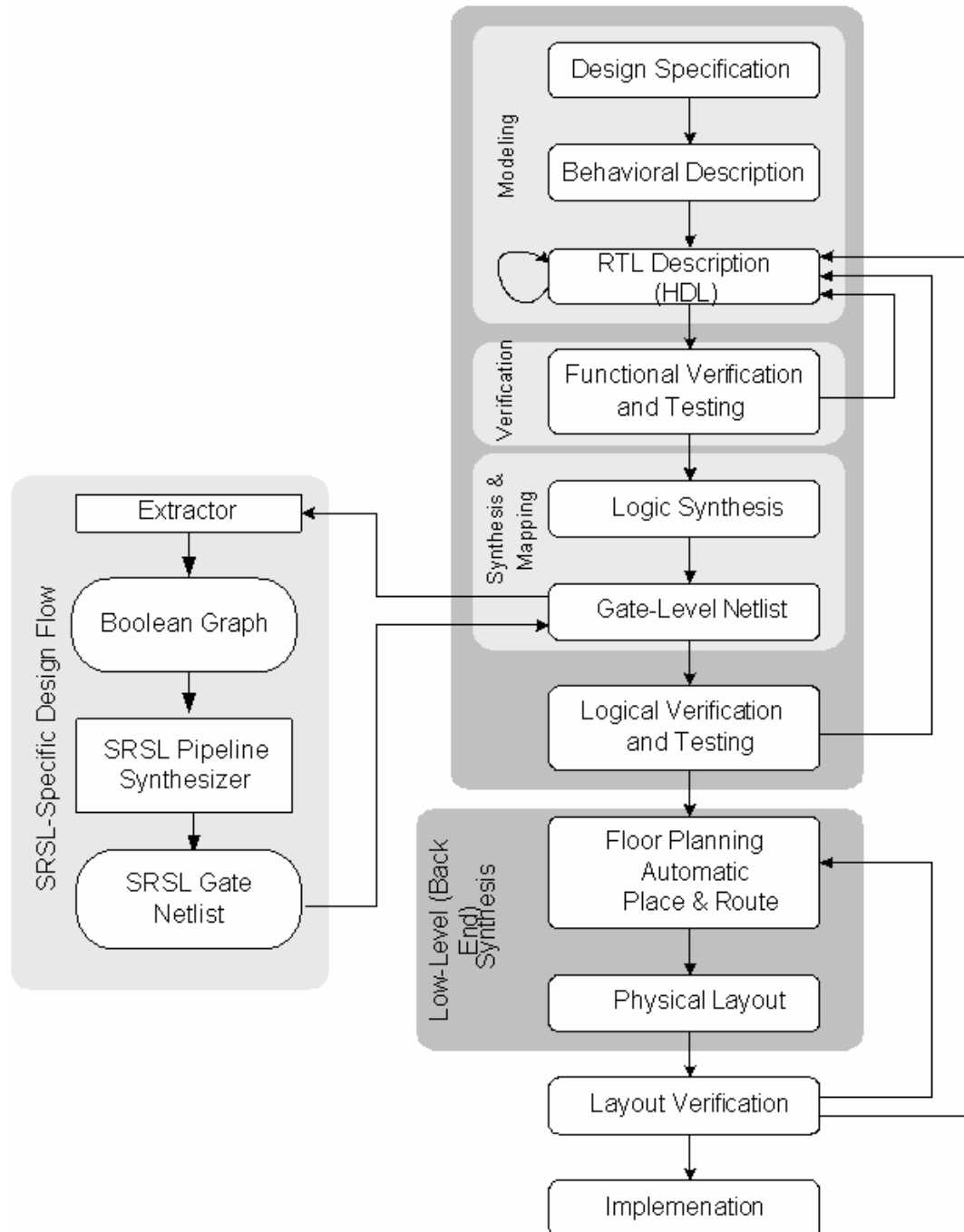


Figure 6.1: SRSL design flow.

## **6.2 Synthesis of SRS� Pipelines**

The synthesis of SRS� pipelines consist of transforming a clocked gate netlist into an SRS� pipeline characterized by a data rate and an area cost. Note that by area cost, it is meant the gate area needed to support an SRS� pipeline structure. This gate area consists primarily of (i) latches located between pipeline stages, and (ii) delay elements needed for the reset network of each stage. As such, this synthesis requires (i) the availability of specific gate resources, and (ii) the specification of performance constraints. The gate resources consist of primitive combinational gates, latches, and delay elements. Each resource is characterized by a function, area, and delay attributes. On the other hand, performance constraints can be area or timing constraints. The former refers to a specified upper limit on gate area needed to convert a gate netlist into an SRS� pipeline while the latter refers to a specified lower limit on data rates that can be achieved by converting a gate netlist into an SRS� pipeline.

To transform a gate netlist into an SRS� pipeline, a designer is faced with three problems:

**Problem 1 (P1):** Given a gate netlist and a data rate, transform the gate netlist into an SRS� pipeline by incurring the smallest area cost. P1 can be called *the data rate constrained minimum area SRS� pipelining problem*.

**Problem 2 (P2):** Given a gate netlist and an area cost, transform the gate netlist into an SRSL pipeline by achieving the highest data rate. P2 can be called *the area constrained maximum data rate SRSL pipelining problem*.

**Problem 3 (P3):** Given a gate netlist, transform the netlist into an SRSL pipeline with the smallest area cost and the highest data rate. P3 can be called *the unconstrained maximum data rate minimum area SRSL pipelining problem*.

Based on their formulations, both P1 and P2 are dual problems. From a practical perspective, P1 is more relevant to designers than P2 and P3.

### **6.3 Preliminaries**

In order to transform a gate netlist into an SRSL pipeline, a gate netlist is abstracted into an algebraic representation suitable for computation.

*Definition 6.1:* An *incidence structure* consists of a set of modules, a set of nets, and an incidence relation among modules and nets [69, 70].

For instance, an incidence structure can be specified by representing each module with its terminals, also called pins or ports, and to describe the incidence among nets and pins. The incidence relationship can be represented by a matrix.

*Definition 6.2:* A *Boolean network* is an incidence structure where:

- Each module performs a Boolean function.
- Each module has multiple inputs and a single output.
- Pins are partitioned into input and output pins.
- Pins that do not belong to modules are primary inputs and primary outputs.
- Each net has a terminal, called source, and an orientation from the source to the other terminals, called sinks.
- The source of a net can be either a primary input or the output of a module.
- The sink of a net can be either a module input or a primary output.
- The relation induced by the nets on the module is a partial order [70].

Figure 6.2 shows a Boolean network with 10 primary inputs, 10 modules, and four primary outputs [70].

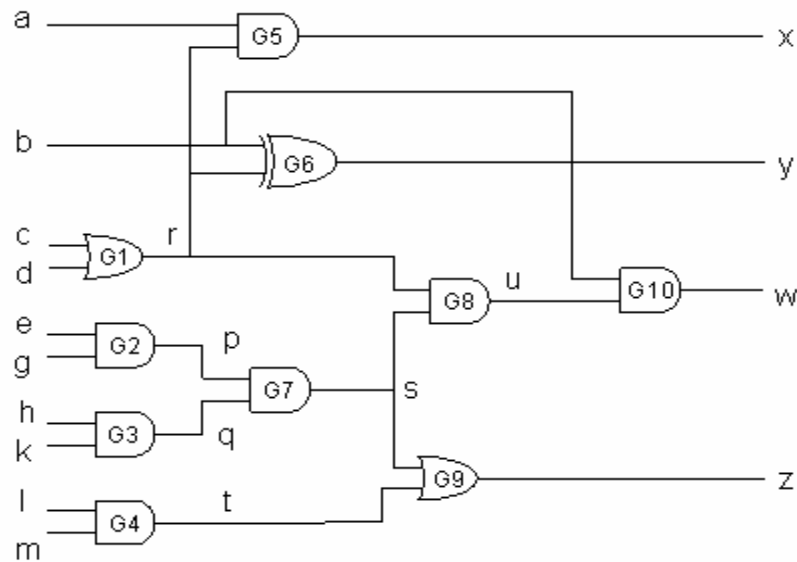


Figure 6.2: Example of a Boolean network.

Boolean networks can be represented in abstract algebraic structures such as graphs.

*Definition 6.3:* A graph  $G(V, E)$  is a pair  $(V, E)$  where  $V$  is a set and  $E$  is a binary relation on  $V$ .

Two vertices in  $V$  are *neighbors* or *adjacent* if they are connected by an edge in  $E$ . In this dissertation, only finite graphs are considered, meaning graphs with finite sets  $V$ . The elements of  $V$  are *vertices* while the elements of  $E$  are *edges*.

*Definition 6.4:* A *directed graph* is graph  $G(V, E)$  where  $E$  is a set of ordered pairs of vertices.

In a directed graph, if two vertices,  $v_i$  and  $v_j$ , are adjacent, meaning  $(v_i, v_j) \in E$ , the *predecessor* is the vertex located at the tail of the edge, namely  $v_i$ , while the *successor* is the vertex located at the head of the same edge, namely  $v_j$ . In contrast, the edges are unordered pairs in an *undirected graph*.

*Definition 6.5:* A *path* from vertex  $v$  to  $w$  in a graph  $G(V, E)$  is a sequence of edges  $v_0v_1, v_1v_2, \dots, v_{k-1}v_k$ , such that  $v = v_0$  and  $v_k = w$ . The length of the path is  $k$ .

Such a path can also be represented as an ordered  $(k+1)$ -tuple:  $\pi = (v_0, v_1, v_2, \dots, v_k)$ . In directed graphs, paths follow the direction of the edges.

*Definition 6.6:* A *cycle* in a directed graph is a nonempty path such that the first vertex and the last vertex are identical.

*Definition 6.7:* A graph is *acyclic* if it has no cycles.

*Definition 6.8:* A *Boolean graph*  $G(V, E)$  is a directed graph where:

- The vertex set  $V$  is a one-to-one correspondence with the primary inputs, modules, and primary outputs of a Boolean network.
- The directed edge set  $E$  represents the decomposition of the multi-terminal nets of the Boolean network into two-terminal nets.

Figure 6.3 shows a Boolean graph based on the Boolean network of Figure 6.2. Note that the Boolean graph is acyclic since the nets induce a partial order on the modules.

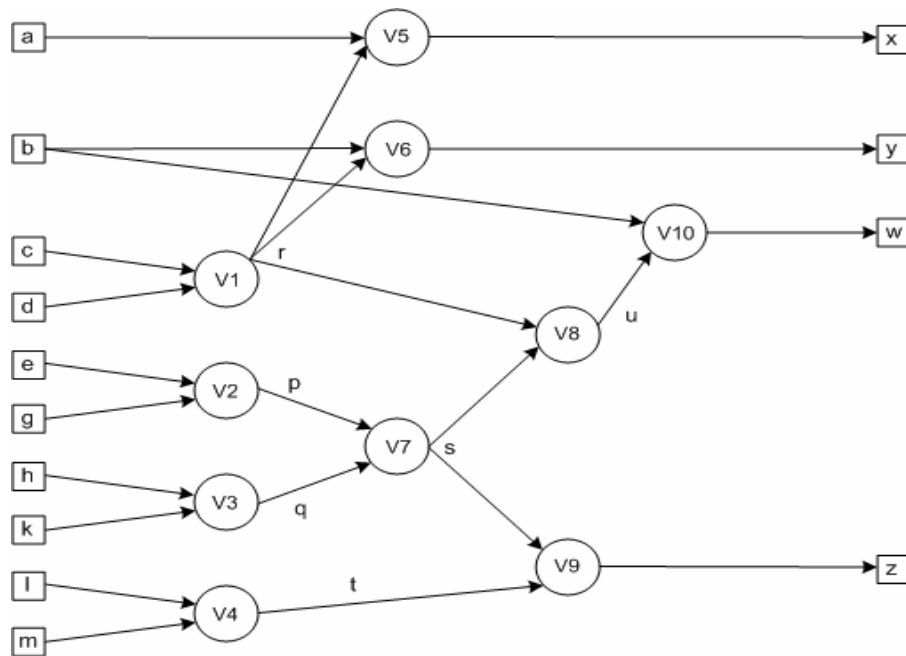


Figure 6.3: Boolean graph of the Boolean network shown in Figure 6.2.

The modules of a Boolean network can be mapped to Boolean gates. In this case, its resulting Boolean graph is *a mapped or bound* Boolean graph. The gate netlist produced by the compiler in Figure 6.1 is a mapped Boolean network. Before it is transformed into an SRTL pipeline, it is translated into a Boolean graph.

#### **6.4 Modeling of the Synthesis Problem**

It is assumed that a clocked gate netlist is specified by a mapped Boolean graph which is subject to a set of constraints. In addition, it is assumed that the function, area, and delay of each gate representing each vertex in the Boolean graph  $G(V, E)$  are known. The constraints can be either data rates or area costs. Transforming a gate netlist into an SRTL pipeline is equivalent to partitioning the Boolean graph of the gate netlist into partitions and assigning each partition to a distinct pipeline stage. Let  $S = \{s_1, s_2, \dots, s_{|S|}\}$  be the set of pipeline stages where the size of this set,  $|S|$ , is some positive integer. Let  $V = \{v_i ; i = 1, 2, \dots, |V|\}$  and  $E = \{(v_i, v_j) ; i, j = 1, 2, \dots, |E|\}$ .

*Definition 6.9:* A *pipelining* of a Boolean graph is a function  $\varphi : V \rightarrow Z^+$  where  $\varphi(v_i) = s_k$  denotes the gate assignment to a stage  $s_k \in S$  such that  $\varphi(v_i) \leq \varphi(v_j), \forall (v_i, v_j) \in E$ .

Since each vertex in  $V$  has a delay,  $D = \{d_i ; i = 1, 2, \dots, |V|\}$ . It is assumed that there are no delays on edges in  $E$  beside the delays on the vertices in  $V$ . Adding delays to the edges will not disturb the modeling of the synthesis problem although it will improve



the quality of its solution. Obviously, such a graph, in which a delay is attributed to each vertex, will have a critical path.

*Definition 6.10:* The delay of a path  $p$  in a graph  $G$ , denoted by  $d_p$ , is the sum of the delays of the vertices in  $p$ , i.e.,  $d_p = \sum_{i: v_i \in p} d_i$ .

*Definition 6.11:* Let  $\Pi$  be the set of all paths in a Boolean graph  $G(V, E)$ . A *critical path* in  $G$  is a path  $\pi$  whose delay is the largest path delay in  $\Pi$ , i.e.,  $d_\pi = \max \{d_p : p \in \Pi\}$ .

In P1, a data rate  $f$  is given and the objective is to minimize the area cost incurred by partitioning the Boolean graph into stage partitions. The period  $P$  of a single stage can be obtained from  $f$  as  $P = \frac{1}{f}$ . Surely, there is a critical path  $\pi$  in the Boolean graph  $G$

whose delay is  $d_\pi$ . An upper bound on the number of stages in the pipeline, called *maximum pipeline depth*, can be obtained from  $P$  and  $d_\pi$ . If  $|S|$  is the cardinality of  $S$ , the

maximum pipeline depth is  $|S| = \left\lceil \frac{d_\pi}{P} \right\rceil = \lceil d_\pi f \rceil$ . Moreover,  $|S|$  can be refined further by

using equation 3.14 from chapter 3 if an S-SRSL pipeline is being synthesized. In this

$$\text{case, } |S| = \min \left\{ \left\lceil \frac{d_\pi}{P} \right\rceil, \left\lfloor 1 + \frac{1}{\delta} \left( \frac{P}{2} - d(L_1^+) \right) \right\rfloor \right\}.$$

*Definition 6.12:* A binary variable  $x_{i,s}$  is associated with each vertex  $v_i$  in  $V$  of  $G(V, E)$  where:

- (i)  $x_{i,s} = 1$  iff the gate  $i$ , represented by  $v_i$ , is assigned to stage  $s$
- (ii)  $x_{i,s} = 0$  otherwise.

In order to realize a correct partitioning, it is imperative that each vertex in the Boolean graph be assigned to a single stage. This requirement is the foundation for a set of constraints called *assignment constraints*:

$$\sum_{s=1}^{|S|} x_{i,s} = 1, \quad i = 1, 2, \dots, |V| \quad (6.1)$$

There are  $|V|$  such constraints in the problem. It also imperative to observe the condition stated in Definition 6.9, namely that the successor of a vertex should be assigned to (i) the same stage as its predecessor, or (ii) a stage located after the stage of its predecessor. This requirement is the foundation for a set of constraints called *precedence constraints*:

$$\sum_{s=1}^{|S|} s x_{i,s} \leq \sum_{s=1}^{|S|} s x_{j,s}, \quad \forall (v_i, v_j) \in E \quad (6.2)$$

These constraints can be rewritten as:

$$\sum_{s=1}^{|S|} s x_{j,s} - \sum_{s=1}^{|S|} s x_{i,s} \geq 0, \quad \forall (v_i, v_j) \in E \quad (6.3)$$

There are  $|E|$  such constraints in the problem. Since  $P$  can be obtained from the given data rate, it is important that the delay through each stage does not exceed  $P$ :

$$\sum_{i: v_i \in \pi} d_i x_{i,s} \leq p, \quad s = 1, 2, \dots, |S| \quad (6.4)$$

There are  $|S|$  such constraints in the problem. By partitioning the Boolean graph into stages, segments of the critical path, or subpaths, are assigned to different stages. The delay on these subpaths determines primarily the period of the stage in which they are included. Constraint (6.4) can be rewritten as an equality if a *balanced pipeline* is desired. A balanced pipelined is a pipeline in which all the stages have the same period, i.e.,  $P = P_i$ ,  $i = 1, 2, \dots, |S|$ . The partitioning of the gate netlist into stages requires the insertion of (i) latches to separate neighboring stages, and (ii) delay elements to realize the reset network of each pipeline stage. In general, the number of latches inserted between two adjacent vertices,  $(v_i, v_j) \in E$ , depend on the stages,  $s_k$  and  $s_l \in S$ , to which both vertices are assigned respectively. Two cases are possible based on the precedence constraints (6.2):

- (i)  $s_k = s_l$ : This means that both stages represent the same stage. In this case,  $v_i$  and  $v_j$  are assigned to the same stage.
- (ii)  $s_k \neq s_l$ : This means that both stages are different. In this case,  $v_i$  and  $v_j$  are assigned to distinct stages. However, there is no indication that both stages,  $s_k$  and  $s_l$  are neighbors.

In fact, it is possible that two adjacent vertices may be assigned to two non-neighboring stages. For example, if  $v_i$  is assigned to stage 3 and  $v_j$  is assigned to stage 7, the edge between the two vertices has to cross the latches of stage 3, 4, 6, and 7, which may require the insertion of four latches to accommodate this case.

*Definition 6.13:* If two adjacent vertices,  $(v_i, v_j) \in E$ , are assigned to stages  $s_k$  and  $s_l \in S$  respectively, the *pipeline distance* between  $v_i$  and  $v_j$ , denoted by  $\delta_{i,j}$ , is  $\delta_{i,j} = l - k$ .

Depending on the bit width of the combinational network in a given stage, latches of different bit widths can be used to separate a stage from its neighbor. It would make sense to quantify the area of the inter-stage latches by multiplying the area of a single-bit latch by the number of output bit lines crossing from stage to stage. These lines correspond to edges in the Boolean graph. Assume that  $a_l$  is the area of a single-bit latch. If  $n$  bit lines are crossing from a stage to another,  $n$  latches are needed adding up to an area of  $na_l$ . Using the definition of pipeline distance, the number of 1-bit latches between two adjacent vertices can be determined as:

$$\delta_{i,j} = \sum_{s=1}^{|S|} sx_{j,s} - \sum_{s=1}^{|S|} sx_{i,s}, \quad (v_i, v_j) \in E \quad (6.5)$$

If applied to a single edge, (6.5) is similar to the left-hand side of (6.3). The latch area needed to support the stages between  $v_i$  and  $v_j$  is  $\delta_{i,j}a_l$ . By considering all the edges in the Boolean graph, the total latch area needed in an entire pipeline can be determined as follows:

$$\sum_{(v_i, v_j) \in E} a_l \left( \sum_{s=1}^{|S|} sx_{j,s} - \sum_{s=1}^{|S|} sx_{i,s} \right) \quad (6.6)$$

Beside the insertion of latches, the insertion of delay elements is also needed to realize the reset network of a stage. These delay elements can be inverters, buffers, or

gates. Since the role of the matching delay of a reset network in SRS� is to provide a delay equal to the delay of the critical path of the combinational network, it would make sense to use gates as delay elements to realize the matching delay of the reset network. In fact, the critical path of the combinational network can be merely duplicated and the obtained copy can be used as a matching delay in the reset network. In this case, the area of the matching delay to be inserted in the reset network of a stage can be determined by obtaining the area of the critical path of the combinational network in the stage. Since each vertex in  $V$  has an area,  $A = \{a_i ; i = 1, 2, \dots, |V|\}$ . If the area of the matching delay of a stage  $s$  is  $a_s$ , then:

$$a_s = \sum_{i: v_i \in \pi} a_i x_{i,s}, \quad s = 1, 2, \dots, |S| \quad (6.7)$$

By considering all the stages in the pipeline, the total area of matching delays can be determined as:

$$\sum_{s=1}^{|S|} \left( \sum_{i: v_i \in \pi} a_i x_{i,s} \right) \quad (6.8)$$

By summing the total area needed for latches shown in (6.6), and matching delays shown in (6.7), the minimization of the area cost can be expressed as the following objective function:

$$\min a_l \sum_{\forall (v_i, v_j) \in E} \left( \sum_{s=1}^{|S|} s x_{j,s} - \sum_{s=1}^{|S|} s x_{i,s} \right) + \sum_{s=1}^{|S|} \left( \sum_{i: v_i \in \pi} a_i x_{i,s} \right) \quad (6.9)$$

In summary, P1 can be formulated as the following *integer programming* (IP) problem:

$$\min a_l \sum_{\forall (v_i, v_j) \in E} \left( \sum_{s=1}^{|S|} s x_{j,s} - \sum_{s=1}^{|S|} s x_{i,s} \right) + \sum_{s=1}^{|S|} \left( \sum_{i: v_i \in \pi} a_i x_{i,s} \right) \quad (6.9)$$

$$\sum_{s=1}^{|S|} x_{i,s} = 1, \quad i = 1, 2, \dots, |V| \quad (6.1)$$

$$\sum_{s=1}^{|S|} s x_{j,s} - \sum_{s=1}^{|S|} s x_{i,s} \geq 0, \quad \forall (v_i, v_j) \in E \quad (6.3)$$

$$\sum_{i: v_i \in \pi} d_i x_{i,s} \leq p, \quad s = 1, 2, \dots, |S| \quad (6.4)$$

$$x_{i,s} \in \{0, 1\}, \quad i = 1, 2, \dots, |V|, \quad s = 1, 2, \dots, |S| \quad (6.10)$$

### **6.5 Proposed Solution of the SRSL Pipeline Synthesis**

The SRSL pipeline synthesis problem can be solved in different ways to obtain two types of solutions:

- (i) *Exact solutions* which can be obtained by solving the IP problems formulated for P1. Several mathematical programming software packages can be used to obtain such solutions. However, obtaining these solutions can take an unreasonable time depending on the size of the IP formulation represented by the number of variables and constraints in the formulation. In fact, the formulation of the IP problem based on the C6822 circuit, one of the benchmark circuits used in the synthesis experiments of D-SRSL pipelines, can generate 6656 assignments constraints (6.1), 9082

precedence constraints (6.3), and 245 period constraints (6.4). In total, the IP formulation consists of a matrix that has 15983 rows and 245 columns.

- (ii) *Approximate solutions* which can be obtained in a short time although they do not guarantee optimality. Such solutions can be reached by applying heuristic algorithms on P1.

The approximate solution has been implemented as a heuristic algorithm consisting of two phases: a stage assignment phase and a vertex shuffling phase. The first phase assigns each gate to a pipeline stage by partitioning the Boolean graph of the gate netlist into subgraphs that meet specific timing constraints. On the other hand, the second phase minimizes the area occupied by inter-stage latches by shuffling nearby vertices from the Boolean graph between adjacent stages without violating the specified timing constraints.

### **6.5.1 Phase I: Stage Assignment**

This section explains the graph-theoretic approach behind the stage assignment performed in phase I. This explanation is followed by a presentation of the algorithm used in phase I.

### **6.5.1.1 Phase I Approach**

In order to pipeline the gate netlist, the Boolean graph of the netlist has to be partitioned into subgraphs whose critical path delays do not exceed a pre-defined value. Each subgraph represents a subnetlist that is assigned to a distinct pipeline stage. Assume that the Boolean graph  $G(V, E)$  can be partitioned into  $n$  partitions or subgraphs where  $G = \bigcup_{i=1}^n G_i$  such that  $V = \bigcup_{i=1}^n V_i$  and  $E = \bigcup_{i=1}^n E_i$ . In order to construct an operationally correct pipeline, the pipeline stages have to be connected through proper insertion of latches between the stages and duplication of the critical path in each stage. This is equivalent to inserting vertices to represent inserted pipeline latches and duplicated critical paths. In fact, the pipeline distance  $\delta$  between two adjacent vertices in  $G(V, E)$  determines the number of latches that needs to be inserted. The edge connecting these two adjacent vertices in  $E$  has to be broken in  $\delta$  edges to accommodate the insertion of  $\delta$  vertices whereby each vertex represents a latch. The resulting graph is an augmented graph  $G'(V', E')$  where  $G \subseteq G'$  such that  $V \subseteq V'$  and  $E \subseteq E'$ . The objective is to add as few vertices as possible in order to realize the smallest area cost possible. For each partition, its critical path delay is determined and a delay block matching the partition's critical path delay is inserted at the appropriate places in the partition. In addition, for each edge crossing one or more partition in the partitioned graph, the pipeline distance  $\delta$  is computed and  $\delta$  vertices representing latches are inserted in the appropriate places in the partitioned graph. The final graph  $G'(V', E')$  represents the Boolean graph of the pipeline gate netlist with inserted latches and matching delays. The following heuristic procedure



can be used to an initial assignment of every gate in the gate netlist to a given pipeline stage:

### **6.5.1.2 Phase I Algorithm**

The pseudocode of the graph partitioning algorithm is as follows:

```

Input:   $G(V, E)$ 
         $D = \{d_i ; i = 1, 2, \dots, |V|\}$ 
         $A = \{a_i ; i = 1, 2, \dots, |V|\}$ 
         $f$ 
Output: Partitioned graph  $G'(V', E')$ 

1.  Let  $P = \frac{1}{f}$ ;
2.  While there are unassigned vertices in  $V$ 
3.      Select a vertex  $v$  in  $V$  whose predecessors are all assigned to
        the current partition;
4.      Get the critical path of the vertices within the current
        partition including  $v$ ;
5.      If the delay of the critical path is less than or equal to  $P$ 
6.          Assign  $v$  to the current partition;
7.      Else
8.          Add another partition;
          Assign  $v$  to the newly added partition;
9.      Endif
10. Endwhile
11. For each edge in  $E$ 
12.     Compute the pipeline distance  $\delta$ ;
13.     Add  $\delta$  vertices to  $V'$ ;
14.     Add  $\delta$  edges to  $E'$ ;
15. Endfor
16. For each partition in  $V'$ ;
17.     Get the critical path in the current partition;
18.     Duplicate the path and insert it into the current partition;
19. Endfor
20. The final obtained partitioned graph is  $G'(V', E')$ ;

```

In line 1, the stage delay is obtained. The algorithm starts with partition 1 which does not contain any vertices at this point. Line 2 shows a loop which looks for vertices in  $V$  which have not been assigned to any partition. Line 3 shows that the first step in

assigning a vertex from  $V$  to the vertex set of the current partition is to select a vertex whose predecessors have been already assigned to the vertex set of the current partition. Next, the critical path of the Boolean graph including vertex  $v$  is obtained in line 4. In line 5 through 9, the algorithm checks if the critical path of the Boolean graph obtained in line 4 is less than or equal to the period of the partition. If the check result is true the selected vertex is added to the vertex set of the current partition. Otherwise, a new graph partition is created to which the selected vertex is subsequently added. The algorithm repeats the line 3 through 9 until there no unassigned vertices in  $V$ . At the end, each vertex in  $V$  is assigned to a distinct vertex set  $V_i$  which belongs to a subgraph  $G_i (V_i, E_i)$  as defined above. After the initial graph  $G(V, E)$  is partitioned, the next step consists of adding vertices between the partitions to represent latches between pipeline stages as shown in line 11 through 15. For each edge in  $E$  crossing two neighboring partitions, a vertex is added followed by the addition of an edge to connect the newly added vertex to its predecessor. This step is followed by a second step in which the portion of the critical path contained in a partition is duplicated and added to that partition as shown in line 16 through 19. This duplicated path represents the matching delay of the reset network which will be attached to the combinational network of the stage represented by the partition. At the end, the augmented graph  $G'(V', E')$  is obtained.

### 6.5.2 Phase II: Vertex Shuffling

This section explains the graph-theoretic approach behind the vertex shuffling performed in phase II. This explanation is followed by a presentation of the algorithm used in phase II.

#### **6.5.2.1 Phase II Approach**

The input to phase II is the augmented partitioned graph  $G'(V', E')$  where each partition represents the portion of the gate netlist embedded in a single pipeline stage. Thus, the number of partitions in the graph represents the number of stages in the pipeline. Every edge that crosses from a partition to another represents a single 1-bit latch in the pipeline. Because latches tend to occupy a significant portion of the overall area of the pipeline, it makes sense to invest additional effort in minimizing the number of latches used in the pipeline. As a result, the objective of phase II is to minimize the number of edges crossing each inter-partition boundary in  $G'(V', E')$ . Note that each inter-partition boundary in  $G'(V', E')$  represents the set of latches separating two adjacent stages in the pipeline corresponding to the two adjacent partitions in  $G'(V', E')$ . Figure 6.4 shows two adjacent partitions where the left partition contains vertices labeled 1 through 10 while the right partition contains vertices labeled 11 through 17.

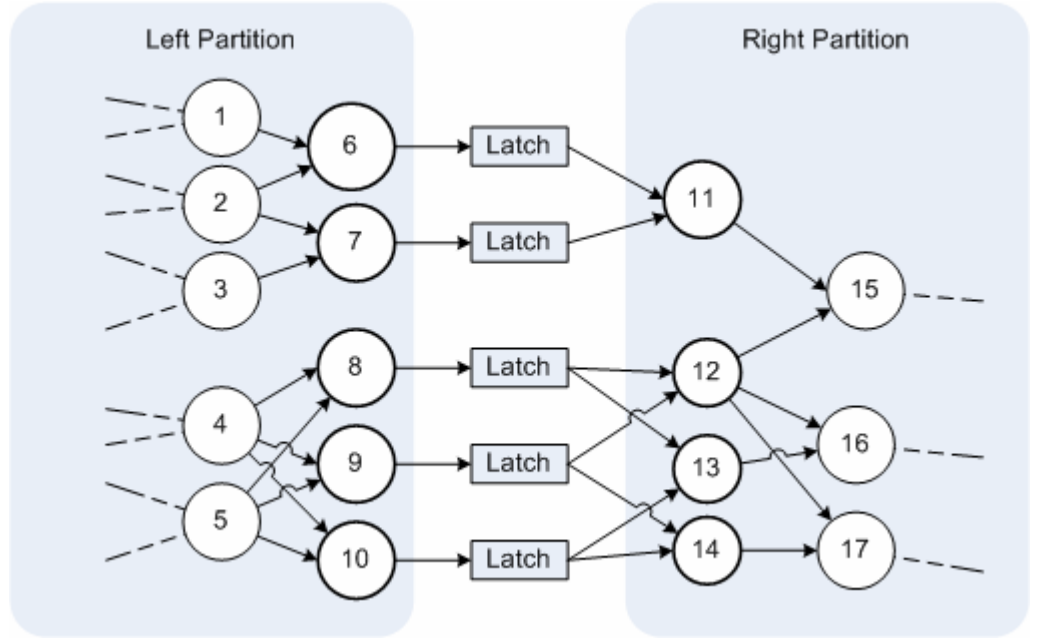


Figure 6.4: Latch insertion between two neighboring pipeline stages.

*Definition 6.14:* Let  $u$  be a vertex in the left partition  $G_L(V_L, E_L)$ , i.e.  $u \in V_L$ .  $u$  is called a *left cut vertex* if it does not have any successors in the left partition, i.e.,  $\nexists v : v \in V_L \text{ and } (u, v) \in E_L$ .

For example, vertices 6, 7, 8, 9, and 10 in Figure 6.4 are all left cut vertices.

*Definition 6.15:* Let  $G_L(V_L, E_L)$  be the left partition. A subset  $C_L$  of  $V_L$ , i.e.,  $C_L \subseteq V_L$ , is called a *left cut vertex set* if every vertex in  $C_L$  is a left cut vertex, i.e.,  $\forall u \in C_L, \nexists v : v \in V_L \text{ and } (u, v) \in E_L$ .

Since vertices 6, 7, 8, 9, and 10 in Figure 6.4 are all left cut vertices, they make up a left cut vertex set.

*Definition 6.16:* Let  $w$  be a vertex in the right partition  $G_R(V_R, E_R)$ , i.e.  $w \in V_R$ .  $w$  is called a *right cut vertex* if it does not have any predecessors in the right partition, i.e.,  $\nexists v : v \in V_R \text{ and } (v, w) \in E_R$ .

For example, vertices 11, 12, 13, and 14 in Figure 6.4 are all right cut vertices.

*Definition 6.17:* Let  $G_R(V_R, E_R)$  be the right partition. A subset  $C_R$  of  $V_R$ , i.e.,  $C_R \subseteq V_R$ , is called a *right cut vertex set* if every vertex in  $C_R$  is a right cut vertex, i.e.,  $\forall v \in C_R, \nexists w : w \in V_R \text{ and } (v, w) \in E_R$ .

Since vertices 11, 12, 13, and 14 in Figure 6.4 are all right cut vertices, they make up a right cut vertex set.

*Definition 6.18:* Let  $C_L$  and  $C_R$  be the left and right cut vertex sets respectively. The set  $C_v$ , called the *cut vertex set*, is the union of the left and right cut vertex sets, i.e.,  $C_v = C_L \cup C_R$ .

While the set of vertices 6, 7, 8, 9, and 10 in Figure 6.4 make up the left cut vertex set, the set of vertices 11, 12, 13, and 14 make up the right cut vertex set. The union of these two sets, namely vertices 6, 7, 8, 9, 10, 11, 12, 13, and 14 makes up a cut vertex set.

*Definition 6.19:* Let edge  $e = (u, v) \in E'$  in the initial partitioned graph  $G'(V', E')$ .  $e$  is called a *cut edge* if  $u$  is a vertex in  $C_L$  and  $v$  is a vertex in  $C_R$ , i.e.,  $(u, v) \in E'$  and  $u \in C_L$  and  $v \in C_R$ .

For example, the edge between vertex 6 and 11 in Figure 6.4 is a cut edge.

*Definition 6.20:* Let  $C_L$  and  $C_R$  be the left and right cut vertex sets respectively. A set  $C_e$  is called a *cut edge set* if every edge in  $C_e$  is a cut edge, i.e.,  $\forall (u, v) \in C_e, (u, v) \in E'$  and  $u \in C_L$  and  $v \in C_R$ .

In Figure 6.4, the set of edges between vertices 6 and 11, 7 and 11, 8 and 12, 8 and 13, 9 and 12, 9 and 13, 10 and 13, and 10 and 14 make up the cut edge set.

*Definition 6.21:* Let edge  $e = (u, v) \in E'$  in the initial partitioned graph  $G'(V', E')$ .  $e$  is called an *internal edge* if  $e$  is not a cut edge, i.e.,  $(u, v) \in E'$  and  $(u, v) \notin C_L$  and  $(u, v) \notin C_R$ .

For example, the edges between vertices 1 and 6, 2 and 6, 11 and 15, and 12 and 15 are all internal edges in Figure 6.4. Consider a vertex  $v$  in the initial partitioned graph  $G'(V', E')$ . It is possible that a number of internal edges may be incident to  $v$ . In this case, let  $I(v)$  denote the set of these internal edges. It is also possible that a number of cut edges may be incident to  $v$ . Let  $C(v)$  denote the set of these cut edges. Note that, depending on where  $v$  is located in  $G'(V', E')$ , it is possible that  $I(v) = \emptyset$  or  $C(v) = \emptyset$ . The proposed vertex shuffling algorithm uses a gain function to guide how it shuffles cut vertices from one partition to another.

*Definition 6.22:* Let  $v$  be a cut vertex in a partition  $H(V_H, E_H)$  where  $H$  can be a left or right partition, i.e.,  $v \in V_H$ . The gain function of  $v$ , denoted as  $g(v)$ , is the difference between the sizes of the set of cut edges and the set of internal edges of all the edges incident to  $v$ , i.e.,  $g(v) = |C(v)| - |I(v)|$ .

In Figure 6.4, vertex 6 has two internal edges and one cut edge. Its gain is  $g(v_2) = |C(v_2)| - |I(v_2)| = 1 - 2 = -1$ . On the other hand, since vertex 11 has one internal and two cut edges, its gain is  $g(v_8) = |C(v_8)| - |I(v_8)| = 2 - 1 = 1$ .

The ultimate objective of the vertex shuffling algorithm is to minimize the number of cut edges. After shuffling a number of cut vertices, the algorithm evaluates the overall cost of these shuffling moves by using a move cost function. This move function is based on the size of the cut edge set. Note that after a cut vertex is moved from one partition to

another, its predecessors and successors in  $G'(V', E')$  will have to be added or removed from a given cut vertex set depending on which cut vertex set contains the moved vertex.

*Definition 6.23:* Let  $v$  be a left cut vertex (i.e.,  $v \in C_L$ ). If  $v$  is moved to the right cut vertex set (i.e.,  $C_L = C_L - \{v\}$  and  $C_R = C_R \cup \{v\}$ ), (i) each predecessor of  $v$  in  $G'(V', E')$  must be added to the left cut vertex set (i.e.,  $\{u \mid u \in V' \text{ and } (u, v) \in E'\} \cup C_L$ ), and (ii) each successor of  $v$  in  $G'(V', E')$  must be removed from the right cut vertex set (i.e.,  $\{w \mid w \in V' \text{ and } (v, w) \in E'\} - C_R$ ). The set of these moves is called the *set of induced moves by  $v$* .

In Figure 6.4, if vertex 6 is moved to the right cut vertex set, (i) all its predecessors, namely vertices 1 and 2, must be added to the left cut vertex set, and (ii) its sole successor, namely vertex 11, must be removed from the right cut vertex set. These three moves make up the set of induced moves by vertex 6. The effect of these moves leaves the left cut vertex set consisting of vertices 1, 2, 7, 8, 9, and 10, while the right cut vertex set consisting of vertices 6, 12, 13, and 14.

*Definition 6.24:* Let  $v$  be a right cut vertex (i.e.,  $v \in C_R$ ). If  $v$  is moved to the left cut vertex set, (i) each successor of  $v$  in  $G'(V', E')$  must be added to the right cut vertex set (i.e.,  $\{w \mid w \in V' \text{ and } (v, w) \in E'\} \cup C_R$ ), and (ii) each predecessor of  $v$  in  $G'(V', E')$  must be removed from the left cut vertex set (i.e.,  $\{u \mid u \in V' \text{ and } (u, v) \in E'\} - C_L$ ). The set of these moves is called the *set of induced moves by  $v$* .



In Figure 6.4, if vertex 11 is moved to the left cut vertex set, (i) its sole successor, namely vertex 15, must be added to the right cut vertex set, and (ii) all its predecessors, namely vertices 6 and 7, must be removed from the left cut vertex set. These three moves make up the set of induced moves by vertex 7. The effect of these moves leaves the left cut vertex set consisting of vertices 8, 9, 10, and 11, while the right cut vertex set consisting of vertices 12, 13, 14, and 15.

*Definition 6.25:* Assume that the shuffling algorithm is on the point of moving a cut vertex  $v$  from one partition to another. The cost function of this move, denoted by  $m(v)$ , is the size of the left cut vertex set if this move and the set of induced moves by  $v$  are completed, i.e.,  $m(v) = |C_L|$ .

Since moving vertex 6 in Figure 6.4 leaves the left cut vertex set consisting of vertices 1, 2, 7, 8, 9, and 10 after the set of its induced moves is completed,  $m(v_6) = |C_L| = |\{1, 2, 7, 8, 9, 10\}| = 6$ . Note that the number of latches between the two pipeline stages represented by the two partitions shown in Figure 6.4 is equal to the size of the left vertex cut set.

### 6.5.2.2 Phase II Algorithm

The pseudocode of the vertex shuffling algorithm is as follows:

Input:  $G'(V', E')$  SRSI pipelined graph that meets  $p$

$D = \{d_i ; i = 1, 2, \dots, |V|\}$

$A = \{a_i ; i = 1, 2, \dots, |V|\}$

Output: Partitioned graph  $G''(V'', E'')$  with minimum cost function between each pair of partitions.

```
1.  For every pair of adjacent partitions in  $G'(V', E')$ 
2.      While the minimum move cost function in the current pass is less
          than the minimum move cost function in the previous pass
3.      While there are unmarked vertices in the left and right cut
          vertex sets
4.          For every unmarked vertex in this cut vertex set
5.              Compute its gain function;
6.          Endfor
7.          Get the vertex with the next highest gain function and
          whose delay does not violate the period constraint in
          its opposite partition;
8.          Compute the move cost function of this vertex;
9.          Mark this vertex and insert it into a queue;
10.     Endwhile
11.     For every cut vertex in the queue starting from the first
          vertex to the vertex with the minimum move cost function
12.         If this vertex is a left cut vertex
13.             Move it to the right cut vertex set;
14.             Perform the set of its induced moves;
15.         Else
16.             Move it to the left cut vertex set;
17.             Perform the set of its induced moves;
18.         Endif
19.     Endfor
20.     For every cut vertex in the queue starting from the vertex
          following the minimum move cost function vertex to the
          last vertex
21.         Unmark this vertex;
22.     Endfor
23. Endwhile
24. Endfor
```

Line 1 shows that phase II algorithm executes for every pair of adjacent partitions in  $G'(V', E')$ . A minimum cost function from a given cut vertex, that is selected to be moved from one partition to another, will be computed in every pass of the procedure, whereby a pass consists of the pseudocode shown in lines 2 through 23. As long as this

cost functions is less than the cost function computed in the previous pass as shown in line 2, another pass is executed. In line 3, all the unmarked vertices in the left and right cut vertex sets will be processed. This processing starts first by computing the gain function for each vertex in these two sets as shown in lines 4 through 6. Next, the move cost function of the vertex with the highest gain function is computed as shown in lines 7 and 8, after which the vertex is marked and inserted in a queue as shown in line 9. This procedure is repeated for every unmarked vertex with the next highest gain function until there are no more unmarked vertices in the left and cut vertex sets as shown in line 3 through 10. Note that from the current iteration to the next, computing the gain function of the remaining unmarked vertices assumes that the induced moves by the marked vertex in the current iteration have been completed. After all unmarked vertices in the vertex cut set are processed, the queue is searched to find the vertex with the minimum move cost function. As shown in lines 11 through 19, every vertex in the queue, starting from the vertex in the first entry of the queue until the vertex with the minimum move cost function in the queue, is moved to the opposite partition followed by the completion of the set of its induced moves. The remaining vertices in the queue are unmarked as shown in lines 20 through 22 to be possibly processed in another pass starting from line 2. To give the unmarked vertices an opportunity to reduce the minimum cost function further, the pseudocode between lines 3 and 22 is re-executed with a different ordering in picking the vertices to compute their move cost functions. To this end, the vertices are processed in non-decreasing order of gain function instead of non-increasing order of gain function as shown in line 7. For simplicity, this pseudocode is omitted from the pseudocode shown above.

## **6.6 Experimental Results**

This section shows the experimental results of both P-SRSL and D-SRSL pipelines. Both phases of the algorithm have been implemented and applied on a set of six circuits shown in Table 6.1.

Table 6.1: Experimental circuits.

| <b>Circuit</b>    | <b>Functionality</b>  | <b>Gates</b> | <b>Critical Path Delay (ps)</b> |
|-------------------|---|--------------|---------------------------------|
| C6288             | 16x16 Multiplier (Largest and deepest)  | 6656         | 25355                           |
| C7552             | 34-bit adder and magnitude comparator with input parity checking (Large and shallowest) | 3569         | 4957                            |
| C5135             | 9-bit ALU (Medium size and shallow)   | 2332         | 6026                            |
| 16_Bit_Multiplier | 16x16 Multiplier (Medium size and medium depth)   | 1456         | 12658                           |
| 32_Bit_Adder      | 32 Bit Adder (Small and deep)   | 160          | 18850                           |
| 16_Bit_Adder      | 16 Bit Adder (Smallest and medium depth)  | 80           | 9380                            |

In this table, column 1 shows the six circuits where the top three are borrowed from the ISCAS-85 benchmark suite while column 2 shows the functionality of each circuit. Column 3 shows the number of gates in the netlist of each circuit while column 4 shows the delay on the critical path. Since S-SRSL and P-SRSL pipelines resemble each other in terms of components, it was decided to apply pipelining experiments on the P-SRSL and D-SRSL pipelines.

### **6.6.1 P-SRSL Pipelining Experiments**

To study the cost of the P-SRSL area, the largest benchmark circuit, namely C6288, was chosen for experimentation since it can accommodate deeper pipelines. It is meant by the P-SRSL area the area that includes the area of the inter-stage latches, intra-stage delay

buffers, and NOR and AND gates used for synchronization. Figure 6.5 shows the P-SRSL area as a percentage of the overall pipeline circuit area including the P-SRSL area. In the figure, as the number of the stages increases the percentage of the P-SRSL area increases too. For example, the P-SRSL area represents only 26% of the pipeline area in the four-stage pipeline. However, this percentage reaches 81% in the 35-stage pipeline. In addition, the figure shows that most P-SRSL area is occupied by the latches. For example, the area of the latches alone consumes 23% of the pipeline area of a four-stage pipeline, and can grow up to 79% of the pipeline area of the 35-stage pipeline. On the other hand, the area of the NOR, AND gates and delay buffers barely consume 5% of the pipeline area across all the pipelines.

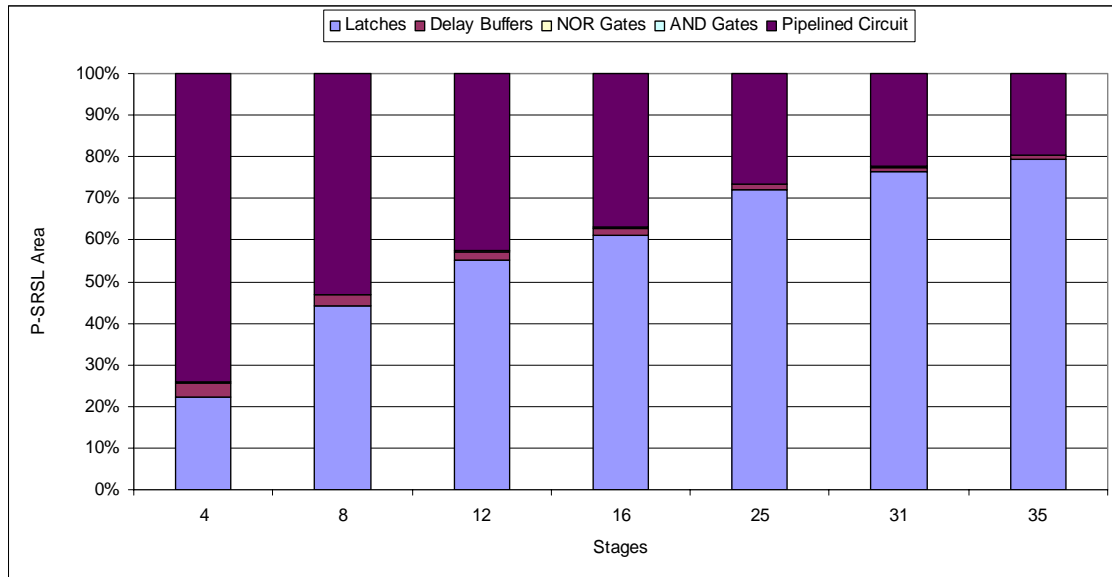


Figure 6.5: P-SRSL area as a percentage of the pipeline area across different pipelines of the C6822 benchmark circuit.

In order to study how P-SRSL pipelining affects the throughput of a circuit, the pipelining algorithm is applied on the six circuits for different pipeline depths as shown in Figure 6.6. For each circuit, the pipeline depth is increased until the circuit ceases to

operate correctly. This situation occurs when the delay in a given stage is so small that the duration of its reset phase is just as small. Note that the inter-stage latches are enabled as long as the stage reset phase lasts. If this duration is smaller than the required enable of the latches used in the actual implementation of the pipeline, these latches will not have sufficient time to capture incoming data, and subsequently the pipeline ceases to operate correctly.

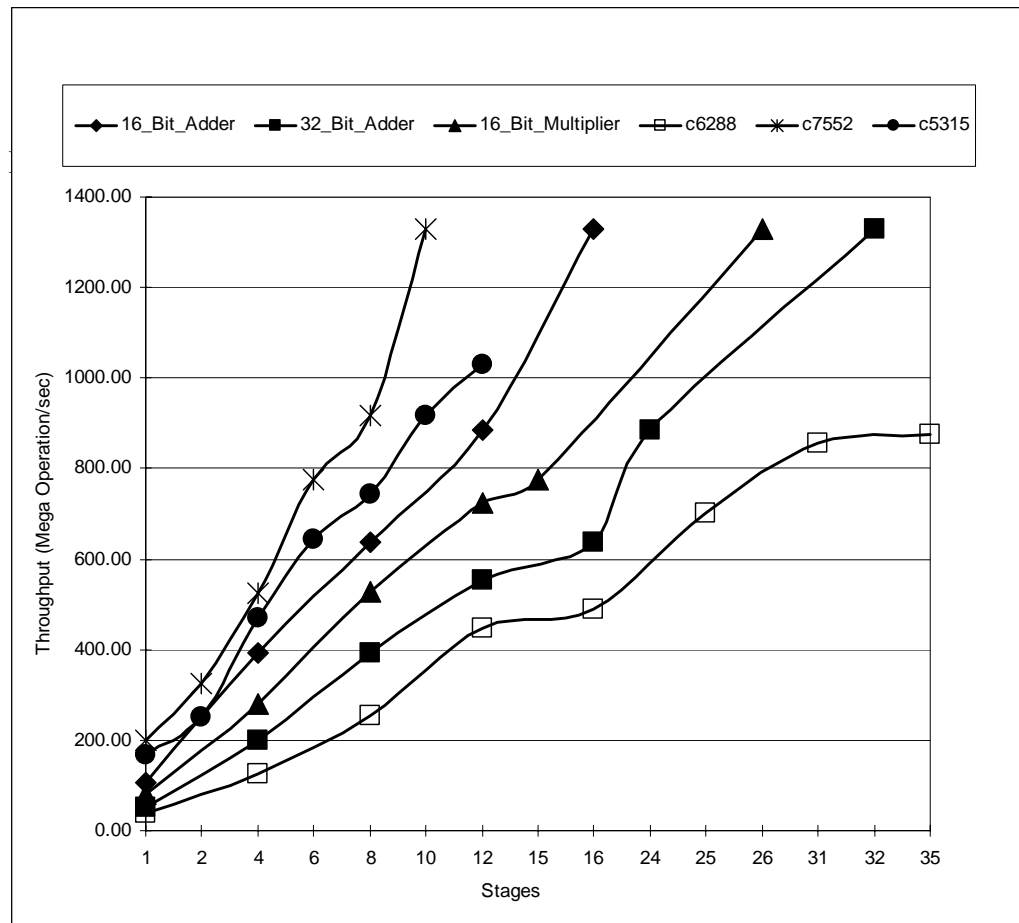


Figure 6.6: Pipeline throughputs for various P-SRSL pipeline depths.

In Figure 6.6, one stage represents the circuit in its non-pipelined version. This figure shows that the throughput of a circuit can increase significantly depending on the

pipeline depth. Indeed, for a shallow circuit, such as C7552, the throughput goes from 201 Megaoperations/sec in its non-pipelined version to 1327.79 Megaoperations/sec in its 10-stage SRSL pipeline. This increase is equivalent to a 6.6 times improvement in throughput. This improvement is even more pronounced in deep circuits. For example, the throughput of C6288 goes from 39.44 Megaoperations/sec in its non-pipelined version to 875.66 Megaoperations/sec in its 35-stage SRSL pipeline. This increase represents 22.2 fold in throughput improvement. While the throughput increases as more stages are added to the pipeline, it is obvious that the rate of throughput increase is not the same for all circuits. It seems that shallow circuits, such as C7552 and C5315, display the fastest throughput increase as opposed to deep circuits such as C6288 and 32\_Bit\_Adder. In fact, shallow circuits have lower latency before they are pipelined. This can be seen by examining stage delays in equal depth pipelines where the delay of a single stage is usually higher in deep circuits than the delay of a single stage in shallow circuits. As a result, the throughput will be higher in shallow circuits as opposed to deep circuits for the same pipeline depth. Furthermore, it is obvious that the maximum possible pipeline depth will be higher in deep circuits than in shallow circuits. Deep circuits can be partitioned into large numbers of stages before the partitioning renders the pipeline inoperable as opposed to shallow circuits.

Figure 6.7 shows the P-SRSL area as a percentage of the total area of a pipeline for each circuit across different pipeline depths. It is clear that the area of each pipeline increases as the circuit is partitioned into a deeper pipeline. However, the largest increases in areas tend to occur in larger circuits partitioned into deeper pipelines.

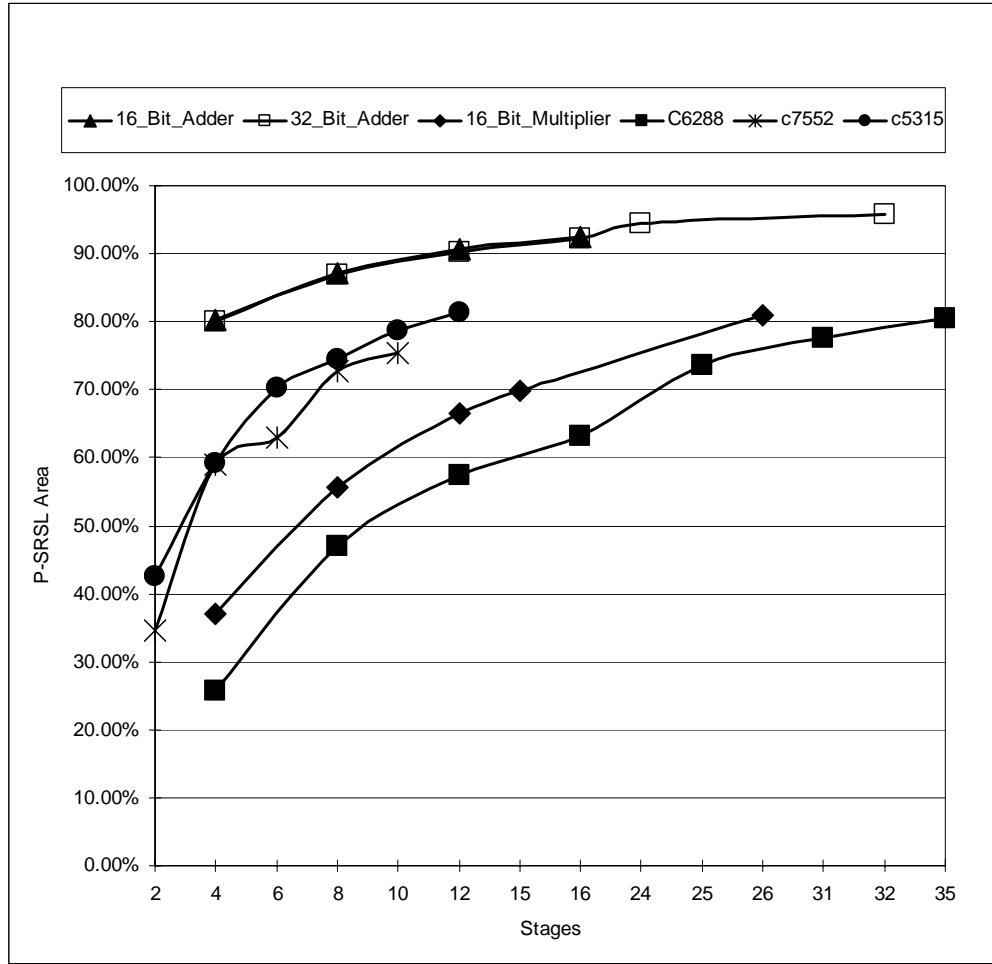


Figure 6.7: P-SRSL area as a percentage of the pipeline area across various depth pipelines.

For example, C6288 shows an increase in P-SRSL area from 26% in a four-stage pipeline to 80% into its maximum depth 35-stage P-SRSL pipeline. On the other hand, slightly smaller area increases can occur in shallow circuits partitioned into shallower pipelines. For example, C5315 shows an increase in P-SRSL area from 42% in a two-stage pipeline to 81% in its maximum depth 12-stage pipeline. Furthermore, it is clear from the figure that the area occupied by P-SRSL circuitry tends to be smaller in general for large and deep circuits than for large and shallow circuits or small and deep circuits.



For example, the P-SRSL area of C6288 occupies around 62% of the total area of its 12-stage pipeline while it can occupy up to 92% of the total area of the 12-stage pipeline in 32\_Bit\_Adder. In any case, small circuits tend to experience high P-SRSL areas regardless of pipeline depth. Since Figure 6.6 and 6.7 show that increasing throughput leads in general to larger P-SRSL areas, it would make sense to evaluate this associated area cost with regard to gains or losses in throughput. A relatively accurate way to measure this relationship is to examine the ratio of the pipeline period over P-SRSL area for all circuits across different pipeline depths as shown in Figure 6.8.

This figure shows that for all circuits, the decrease rate of this ratio speeds up in shallow pipelines and slows down in deep pipelines. This can be explained by the fact that in partitioning the circuit graph into a few partitions, the number of vertices in the partitions is significantly large. As a result, there is a relatively large number of edges crossing the partitions. These edges will all be covered by latches to synchronize the data flow across partitions or pipeline stages thus leading to a large P-SRSL area. As the circuits get partitioned into deeper pipelines, the number of graph partitions increases, which yields to a decrease in the number of vertices in the partitions in general. This decrease is accompanied by a decrease in the number of inter-partition edges leading to a decrease in the number of latches needed to cover these edges.

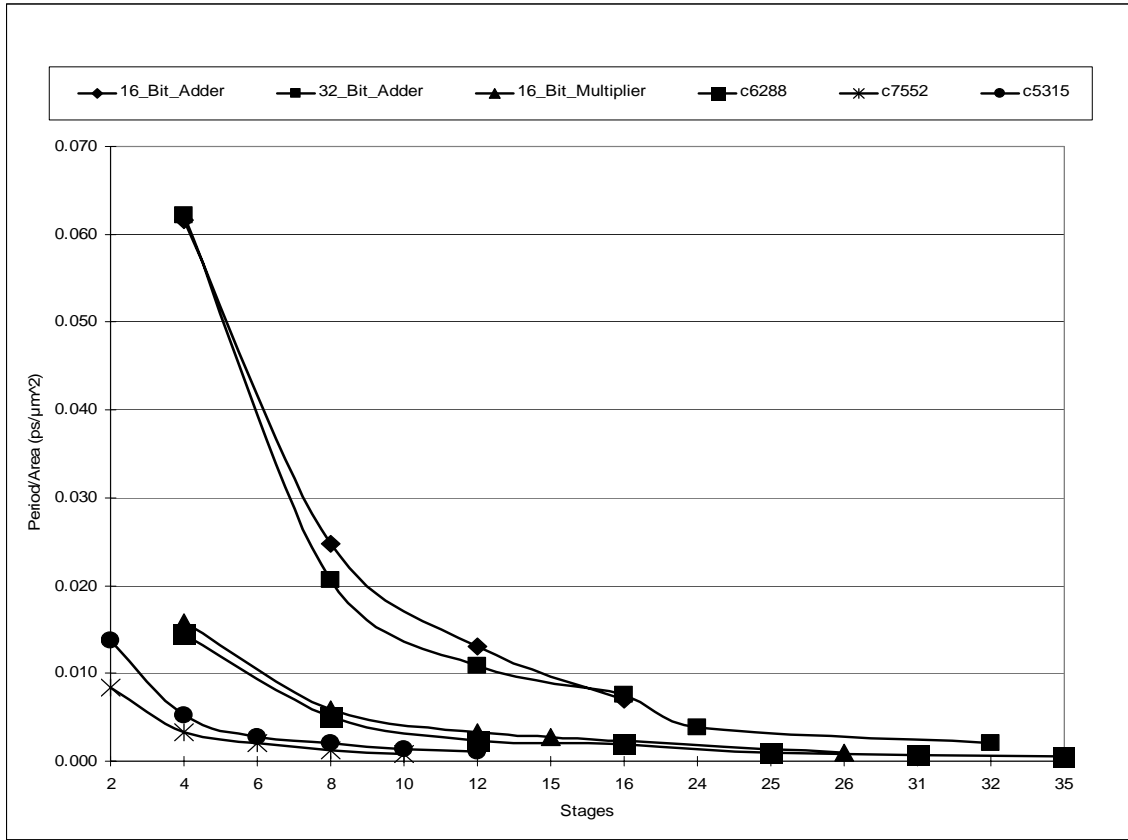


Figure 6.8: Period over area ratios for different depths P-SRSL pipelines.

Note that among the components used to support P-SRSL synchronization, such as latches, delay buffers, AND and NOR gates, the latches are the components with the largest areas. In any case, this shows that the tradeoff of throughput gain or loss vs. P-SRSL area is beneficial for deep S-SRSL pipelines and costly in shallower P-SRSL pipelines. In the ideal case, the period-area ratio should be decreasing or at least remain constant. However, Figure 6.8 shows that this ratio decreases for all circuits at different rates. If this is the case, a ratio with a slow decreasing rate is highly desirable since it would indicate that the P-SRSL area increases slowly as more stages are added to the pipeline of a given circuit. Figure 6.8 shows that whereas the slowest decrease in this ratio occurs for large and deep circuits such as C6288, this ratio decreases quite rapidly

for small and deep circuits, such as 16\_Bit\_Adder, particularly when partitioned into shallow pipelines. The decrease is even slower for large and shallow circuits such as C7552. This shows that partitioning small and deep circuits requires relatively larger P-SRSL areas to support their P-SRSL pipelines. The increase in area cost can be offset in throughput gains only when large and deep datapaths are converted into deep P-SRSL pipelines. Without a doubt, it can be concluded P-SRSL pipelining is highly suitable for coarse-grain datapaths.

### **6.6.2 D-SRSL Pipelining Experiments**

To study the cost of the additional area that is required to synchronize the D-SRSL pipeline, Circuit C5135 is chosen as an example. It is meant by the D-SRSL area the area that includes the area of the PC blocks, the LC blocks, inter-stage latches, and the intra-stage delay buffers. Figure 6.9 shows the area percentage of each component that contributes to D-SRSL area. This figure shows that as the number of stages increases, the percentage of the D-SRSL area increases too. For example, the D-SRSL area is around 43 % of the overall all area of a four-stage pipeline. This percentage can go up to 81 % in a 12-stage pipeline. Among the components used in D-SRSL pipelines, the area of inter-stage latches is significantly large since it occupies around 41% of the overall area of a four-stage pipeline. This percentage can go up to 80.3% in a 12-stage pipeline. However, the entire area of the PC blocks, LC blocks, and delay buffers occupies barely 2% of the overall area of a four-stage pipeline, and 0.7 % in a 12-stage pipeline.

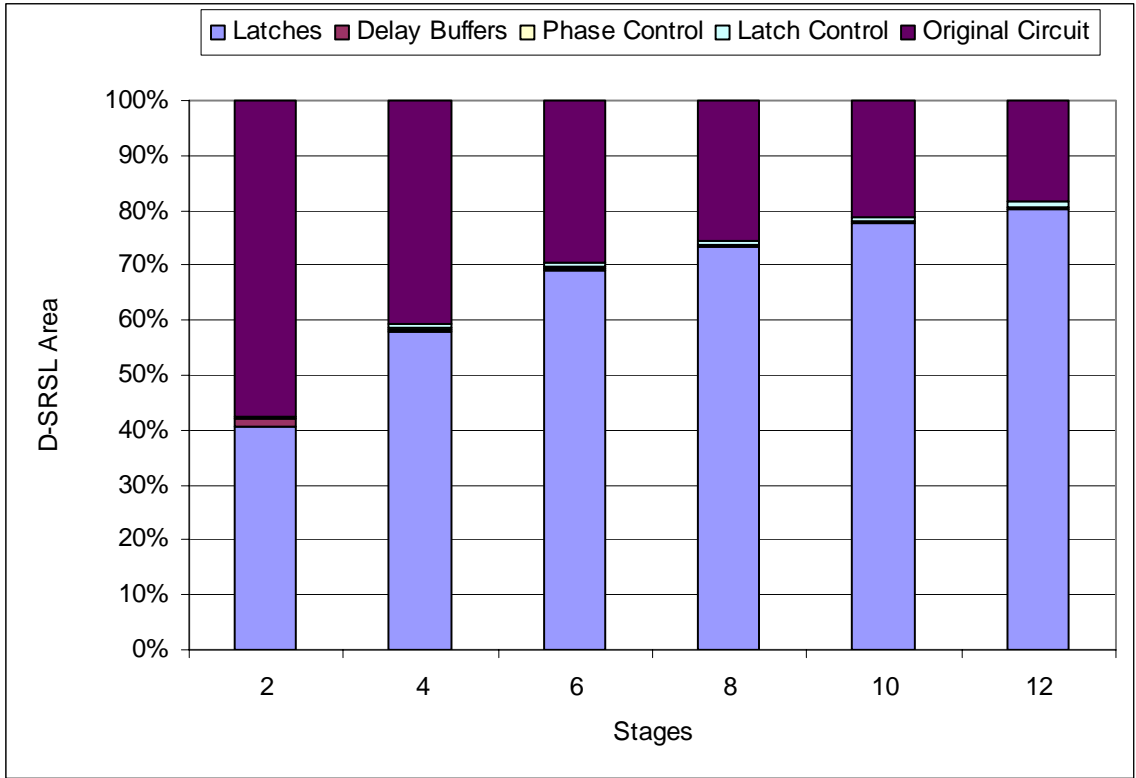


Figure 6.9: D-SRSL area as a percentage of the pipeline area across different pipelines of the C5135 benchmark circuit.

In order to study how D-SRSL pipelining affects the throughput of a circuit, the pipelining algorithm is applied to the six experimental circuits for different pipeline depths as shown in Figure 6.10. For each circuit, the pipeline depth is increased until the circuit throughput cannot be improved any more. This situation occurs when the CN is so small that its delay is less than the delay of the LC block (i.e.,  $D(CN_i) < D_{clr}(LC_i)$ ) as described in scenario (i) of equation (5.10).

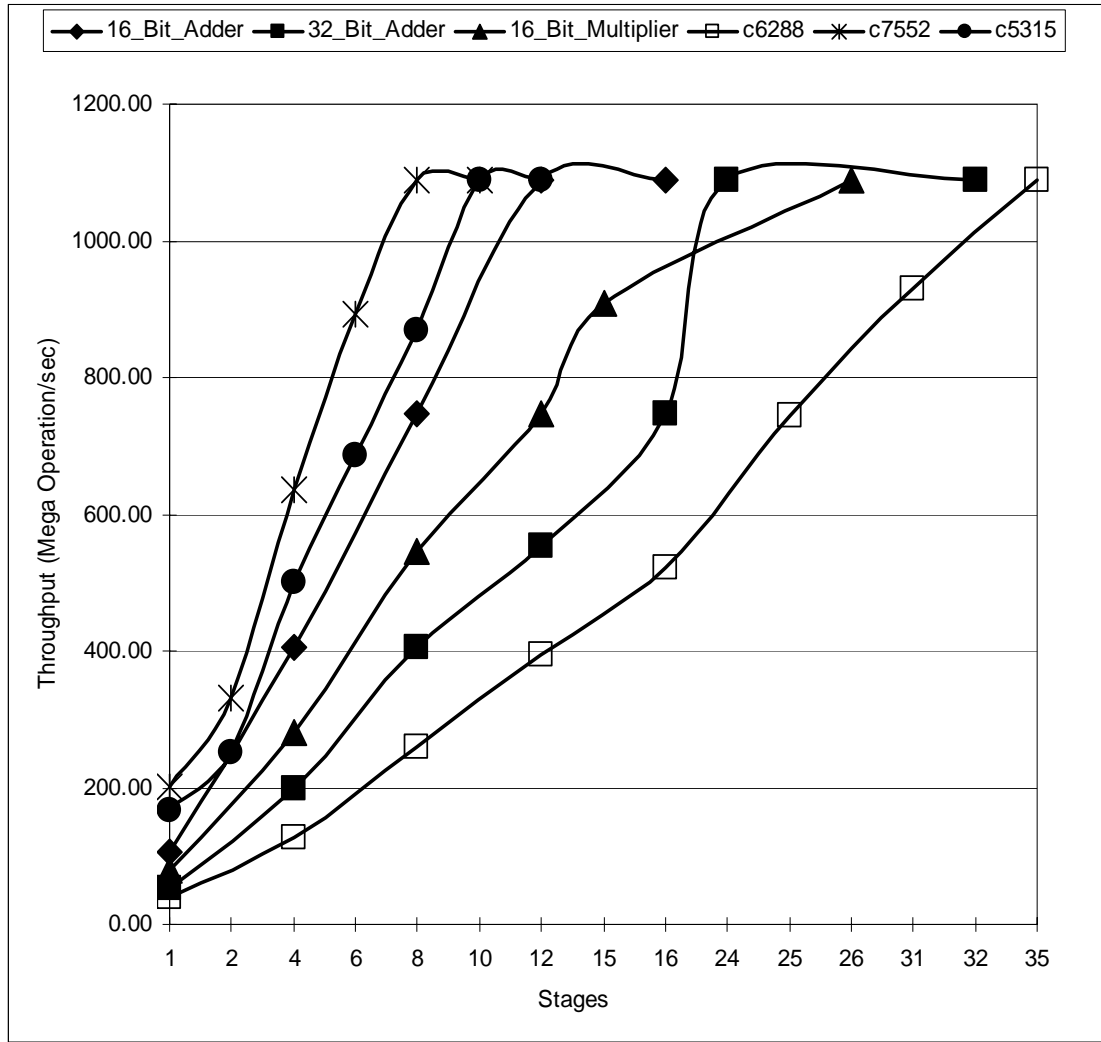


Figure 6.10: Pipeline throughputs for various D-SRSL pipeline depths.

In Figure 6.10, one stage represents a circuit in its non-pipelined version. This figure shows that the throughput of a pipeline can increase significantly depending on the pipeline depth. In the case of C7552, which is the shallowest circuit in the benchmark set, the throughput goes from 200 Megaoperations/sec in its non-pipelined version to 1088.14 Megaoperations/sec in its eight-stage D-SRSL pipeline. This increase is equivalent to a 5.44 times throughput improvement. This improvement is even more pronounced in deep circuits. For example, the throughput of C6288 goes from 39.44

Megaoperations/sec in its non-pipelined version to 1088.14 Megaoperations/sec in its 35-stage D-SRSL pipeline. This increase represents 27.58 fold in throughput improvement.

While some circuits, such as C7552, can reach their maximum throughput in a few stages, other circuits, such as C6288, do not seem to reach a maximum throughput even when partitioned into deeper pipelines of 35 stages. In fact, the throughput of shallow circuits, such as C7552, seems to level off after they have been partitioned into short pipelines. On the other hand, the throughput of deep circuits, such as C6288, do not display this leveled-off curve. In a smaller number of stages, shallow circuits can get partitioned so much that their intra-stage CNs are quite small. As a result, the delay of these CNs becomes smaller than the delay of the LC block (i.e.,  $D(CN_i) < D_{clr}(LC_i)$ ). By partitioning these circuits further after this point,  $D_{clr}(LC_i)$  does not change, and subsequently,  $d(L_i)$  and  $d(R_i)$  remain constant. This has the effect of keeping  $P$  constant, which results in a leveling off of the throughput. In deeper circuits, this throughput improvement limit does not appear so quickly, and consequently these circuits display a continuous increase in throughput improvement even when partitioned in deeper pipelines.

Note that, similarly to P-SRSL pipelines, shallow circuits tend to have a higher throughput than deep circuits for the same pipeline depth. This can be attributed to the fact that the delay of a single stage is usually higher in deep circuits than the delay of a single stage in shallow circuits. As a result, the throughput will be higher in shallow circuits as opposed to deep circuits for the same pipeline depth.

Figure 6.11 shows the D-SRSL area as a percentage of the overall pipeline area for each circuit across different pipeline depths. It is clear that the area of each circuit increases as the circuit is partitioned into a deeper pipeline. However, the largest increases in areas tend to occur in larger circuits partitioned into deeper pipelines in a similar fashion to the area increase in P-SRSL pipelines.

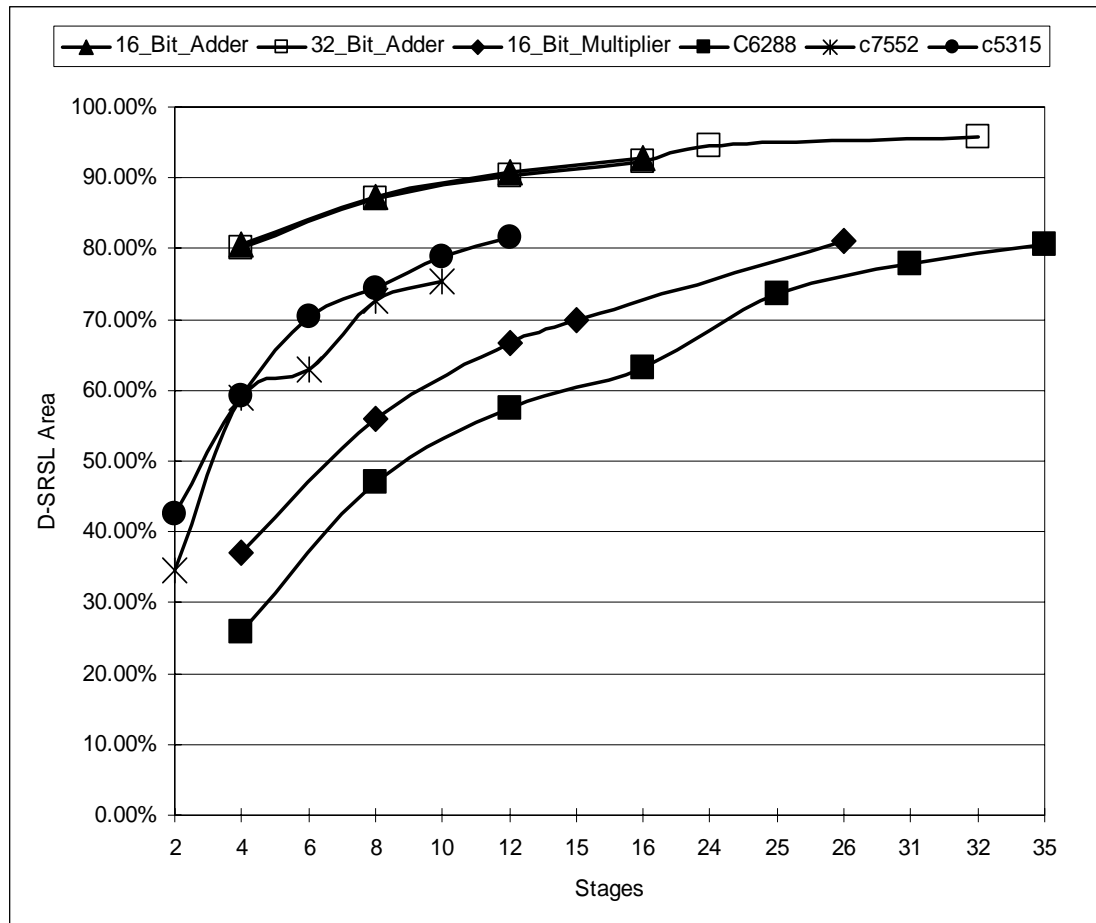


Figure 6.11: D-SRSL area as a percentage of the pipeline area across various depth pipelines.

For example, C6288 shows an increase in D-SRSL area from 26% in a four-stage pipeline to 80% into its maximum depth 35-stage D-SRSL pipeline. On the other hand, slightly smaller area increases can occur in shallow circuits partitioned into shallower

pipelines. For example, C7552 shows an increase in D-SRSL area from 35% in a two-stage pipeline to 75% in its maximum depth 10-stage pipeline. Furthermore, it is clear from the figure that the D-SRSL area tends to be smaller in general for large and deep circuits than for small circuits. For example, the D-SRSL area of C6288 occupies around 58% of the total area of its 12-stage pipeline while it can occupy up to 90% of the total area of the 12-stage pipeline in 32\_Bit\_Adder. In any case, small circuits tend to experience high D-SRSL areas regardless of pipeline depth.

### 6.6.3 Summary of the Experiment Results

The experimental results for both P-SRSL and D-SRSL pipeline shows that P-SRSL pipeline can reach a throughput of 1327 Megaoperations/sec while D-SRSL can reach only 1088 Megaoperations/sec throughput. This can be explained by considering the parameters which affect  $d(L)$ . In D-SRSL pipelines, extreme pipelining can lead to the situation where  $D(CN_i) < D_{ctr}(LC_i)$  thus making  $d(L_i) = d(R_i)$  as shown in equation (5.10). However, equation (5.7) states that  $d(R_i) = D(PC_i) + D_{right}(LC_i)$ . Note that extreme fine-grain pipelining will affect  $D(CN_i)$ , but not  $D(PC_i)$  and  $D_{right}(LC_i)$ . As a result,  $d(R_i)$  and subsequently  $P$  remain constant beyond this point. Once  $P$  ceases to decrease, the pipeline throughput ceases to increase. On the other hand,  $d(L)$  in P-SRSL pipelines is affected differently. In fact,  $d(L_i^+) = \frac{P}{2}$  as equation (4.14) and (4.15) state.

This means that, as the pipeline gets partitioned into fine-grain logic,  $P$  decreases without affecting the ability of the latch to capture data during pipeline operation. As a result, the



P-SRSL pipeline can reach throughputs that are higher than those reached by the D-SRSL pipeline.

Whereas the P-SRSL pipeline seems to display a higher throughput in deeper pipelines in general, D-SRSL pipelines reach a higher throughput in deeper pipelines of large and deep circuits. For instance, circuit C6288 can reach a throughput of 1088 Megaoperations/sec in its 35-stage D-SRSL pipeline while it can only reach a throughput of 875 Megaoperations/sec in 35-stage P-SRSL pipeline. In the case of D-SRSL pipelines, equation (5.8) states that:

$$d(E_i) + d(R_i) \geq D(L_i) + D(CN_i) \quad (5.8)$$

Using equations (5.6) and (5.7), equation (5.8) can be rewritten as:

$$D(PC_i) + D_{left}(LC_{i+1}) + D(PC_i) + D_{right}(LC_i) \geq D(CN_i) + D(LC_i) \quad (6.11)$$

This is equivalent to:

$$2D(PC_i) + D_{right}(LC_i) + D_{left}(LC_{i+1}) \geq D(CN_i) + D(LC_i) \quad (6.12)$$

If delay parameters relevant to the implementations of the PC and LC blocks are considered, equations (6.12) can be rewritten by using equations (5.24), (5.25), and (5.26) as follows:

$$\begin{aligned} 3D(INV) + 4D(NAND) + 4D(AND) + 2D(OR) + D(\text{clk\_to\_Q}) + D(\Delta) \\ \geq D(CN_i) + D(LC_i) \end{aligned} \quad (6.13)$$

From an implementation perspective, a straightforward optimization would be to reduce the slack of equation (6.13). This reduction can be achieved only by reducing  $D(\Delta)$ . Since all the other terms are all library-dependent, only  $D(\Delta)$  can be fine-tuned by the

designer. Experimentation shows that by reducing  $D(\Delta)$  further without violating equation (5.8),  $P$  can be reduced further leading to a higher throughput. Whereas this optimization of the implementation is possible in D-SRSL pipelines of deeper circuits, it is not suitable for the implementation P-SRSL pipelines. As a result,  $D(\Delta) \geq D(CN)$  in each stage in P-SRSL pipelines. This explains the higher throughput displayed by D-SRSL pipelines of deep circuits in deep pipelines.

With regard to area, both pipelines display the same overhead in SRSL circuitry area. As explained in the previous paragraph, the optimization of D-SRSL implementations rely on reducing the delay matching  $D(\Delta)$  of  $D(CN)$ . This reduction leads to a reduction in the number and size of the buffers used to calibrate this delay. Since buffers are second to latches in consuming large silicon areas, this reduction in the number and size of buffers yields a significant reduction in the area occupied by D-SRSL circuitry. Although D-SRSL circuitry requires more coarse-grain components such as the PC and LC blocks in a pipeline, the area reduction stemming from the elimination of buffers brings the D-SRSL circuitry to a level that is sufficiently low to be comparable to the area occupied by the P-SRSL circuitry.

## **6.7 Summary**

In this chapter, the conventional design flow is minimally modified in order to support the synthesis of SRSL pipelines. The synthesis of these pipelines is formulated as an optimization problem subject to a set of data rate and timing constraints. Analytical

formulation of this problem is presented as a standard IP problem [71]. Since the size of the IP problem is significantly large, and subsequently solving it using analytical approaches is impractical, a heuristic algorithm is proposed to solve it. The goal of the algorithm is to minimize the area occupied by inter-stage latches without violating any timing constraints [72]. This algorithm reaches this goal into two phases: (i) Phase I in which a partitioning procedure is applied on the Boolean graph of the gate netlist, and (ii) Phase II in which partition vertices are swapped between each pair of adjacent partitions in order to minimize the cut size between the pairs of partitions. The goal of Phase I is to assign each gate in the gate netlist to a specific pipeline stage. On the other hand, the goal of Phase II is to minimize the number of inter-stage latches between every pair of neighboring pipeline stages.

The heuristic algorithm has been implemented and applied to six different circuits for the purpose of producing P-SRSL and D-SRSL pipelines with different depths. The experimental results reveal that P-SRSL pipelines can reach higher throughput in deeper pipelines in general while D-SRSL pipelines produce the same performance if large and deep circuits are partitioned into deep pipelines. In addition, these results show that both pipelines exact the same cost in terms of the area occupied by SRSL circuitry.

## CHAPTER SEVEN: CONCLUSION

This chapter summarizes the work presented in this dissertation. This summary is followed by a discussion of future work.

### **7.1 Summary of Completed Work**

This dissertation presents SRSL as a clockless design technique highly suitable for existing CAD tools. This technique displays self-resetting characteristics in the form of a periodic oscillation of a logic block driven by a reset loop similar to an internal clock. Based on SRSL, three pipelining techniques are proposed: S-SRSL, P-SRSL and D-SRSL.

In S-SRSL, communication between stages is controlled at the stage level. The timing analysis of S-SRSL pipelines reveals insights on how the duration of the evaluate phase gradually increases while the duration of the reset phase and the latch enable gradually decreases toward the left stages of the pipeline. This gradual decrease in the duration of the enable of the latches between stages is used to derive a bound on the maximum possible depth of the pipeline.

In P-SRSL, pipeline stages are synchronized with the oscillations of the last pipeline stage. In this communication scheme, stages of type A oscillate in the same phase with the last stage while stages of type B oscillate in opposite phase with the last stage. Timing analysis of P-SRSL pipelines reveals that the duration of the evaluate and

reset phase remains constant in the stages located to the left of the last stage in the pipeline. Also, this analysis shows that the duration of the latch enable is constant regardless of the stages in the pipeline. This is due to the fact that the  $\delta$  effect does not propagate across the pipeline stages as seen in S-SRSL pipelines, which in return keeps the duration of the evaluate and reset phases constant in the stages before the last stage of the pipeline. In contrast to S-SRSL pipelines, the incremental delays caused by the propagation of  $\delta$  are completely absent in P-SRSL pipelines.

Contrary to S-SRSL and P-SRSL pipelines in which the stages must have equal delays, D-SRSL pipelines can tolerate stages with different delays. As a result, this pipelining style is highly suitable for coarse-grain datapaths. Similarly to S-SRSL and P-SRSL pipelines, the stages in D-SRSL pipelines oscillate between an evaluate and reset phase. Timing analysis of these pipelines shows that, although pipeline stages have equal period, the duration of their reset and evaluate phase depends on the location of the stage in relation to the location of the slowest stage in the pipeline. The ability of the pipeline to handle stages with different delays is made possible by stretching the evaluate phases and shrinking the reset phases of the stages before the slowest stage in the pipeline. The amount of stretching and shrinkage is roughly equal to the difference between logic delay in the slowest stage and in any stage before it. While this phenomenon appears on the stages before the slowest stage, its dual manifests itself in the stages after the slowest stage in the pipeline. Table 7.1 highlights the characteristics of the three SRSL pipelining techniques by contrasting their performance parameters while table 7.2 contrasts their capabilities in handling stuck-at faults.

Table 7.1: SRS� pipelining parameters.

| Parameter                  | P-SRS� Pipeline  | S-SRS� Pipeline  | D-SRS� Pipeline  |
|----------------------------|--|--|--|
| Data Encoding              | Bundled data   | Bundled data   | Bundled data   |
| Synchronization Scheme     | Pipeline level   | Stage level  | Stage level  |
| Synchronization Directions | Uni-directional  | Uni-directional  | Bi-directional   |
| Delay tolerance            | Comparable stage delays  | Comparable stage delays  | Unequal stage delays   |
| SRS� area                  | 1 NOR gate, 1 AND gate, Delay block                              | 1 NOR gate, 1 AND gate, Delay block                              | PC block, LC block, Delay block  |
| Matching Delay             | $D(\Delta) \geq D(CN)$   | $D(\Delta) \geq D(CN)$   | $D(\Delta) < D(CN)$  |
| Reset Phase                | $d(R_i) = d(R_j),$<br>$1 \leq i \leq (n-1), 1 \leq j \leq (n-1)$ | $d(R_i) = d(R_n) - (n-i)\delta,$<br>$1 \leq i \leq (n-1)$        | $d(R_i) < d(R_k), i < k$<br>$d(R_j) > d(R_k), j > k$<br>where $k$ is the slowest stage |
| Evaluate Phase             | $d(E_i) = d(E_j),$<br>$1 \leq i \leq (n-1), 1 \leq j \leq (n-1)$ | $d(E_i) = d(E_n) + (n-i)\delta,$<br>$1 \leq i \leq (n-1)$        | $d(E_i) > d(E_k), i < k$<br>$d(E_j) < d(E_k), j > k$<br>where $k$ is the slowest stage |
| Evaluate vs. Reset Phase   | $d(R_i) < d(E_i), 1 \leq i \leq (n-1)$                           | $d(R_i) < d(E_i), 1 \leq i \leq (n-1)$                           | $d(R_i) < d(E_i), i < k$<br>$d(R_j) > d(E_j), j > k$<br>where $k$ is the slowest stage |
| Period                     | $P \leq 2(D(NOR) + D(\Delta) + D(L))$                            | $P \leq 2(D(NOR) + D(\Delta) + D(L))$                            | $P \leq 2D(PC_i) + D_{right}(LC_i) + D_{left}(LC_{i+1})$                               |
| Latch Enable               | $d(L_i^+) = \frac{P}{2}$   | $d(L_i^+) = \frac{P}{2} - (n-i)\delta$                           | $d(L_i) = \min\{d(R_i), D_{Clr}(LC_i)\}$   |
| $\delta$ Delay Difference  | Between any stage and the last stage                             | Between any two neighboring stages                               | None   |
| Theoretical Pipeline Depth | No limit   | $n = 1 + \frac{1}{\delta} \left( \frac{P}{2} - d(L_1^+) \right)$ | No limit   |

Table 7.2: Fault handling in the three pipelines.

| <b>Fault</b>                             | <b>P-SRSL Pipeline</b>   | <b>S-SRSL Pipeline</b>   | <b>D-SRSL Pipeline</b>  |
|--|--|--|---|
| Stage $j$ is stuck in the evaluate phase | <ul style="list-style-type: none"> <li>• Data keeps flowing uninterrupted throughout the pipeline.</li> </ul>  | <ul style="list-style-type: none"> <li>• Data flow is blocked from stage 1 to stage <math>j</math>.</li> <li>• The same data item keeps moving from stage <math>j+1</math> to stage <math>n</math>.</li> </ul>   | <ul style="list-style-type: none"> <li>• Data flows from stage 1 to stage <math>j</math> for one period before its flow is blocked.</li> <li>• Data flow is blocked from stage <math>j+1</math> to <math>n</math>.</li> </ul> |
| Stage $j$ is stuck in the reset phase    | <ul style="list-style-type: none"> <li>• Data flows uninterrupted from stage 1 to stage <math>j</math> resulting in overwriting data in stage <math>j</math>.</li> <li>• The same data item keeps moving from stage <math>j+1</math> to stage <math>n</math>.</li> </ul> | <ul style="list-style-type: none"> <li>• Data flows uninterrupted from stage 1 to stage <math>j</math> resulting in overwriting data in stage <math>j</math>.</li> <li>• The same data item keeps moving from stage <math>j+1</math> to stage <math>n</math>.</li> </ul> | <ul style="list-style-type: none"> <li>• Data flows from stage 1 to stage <math>j</math> for one period before its flow is blocked.</li> <li>• Data flow is blocked from stage <math>j+1</math> to <math>n</math>.</li> </ul> |

Since SRSL is intended to be supported by existing CAD tools, the synthesis of these pipelines is formulated as an optimization problem, in the form of an IP, subject to a set of data rate and timing constraints. Because the size of the IP problem is significantly large, a two-phase heuristic algorithm is proposed to solve it. The goal of the algorithm is to minimize the area occupied by inter-stage latches without violating any timing constraints. This goal is reached by executing Phase I of the algorithm in which each gate in the gate netlist is assigned to a specific pipeline stage. Subsequent to Phase I, Phase II is executed in order to minimize the number of inter-stage latches between every pair of neighboring pipeline stages. Application of this pipelining to a set of experimental circuits reveals that high throughputs can be achieved by P-SRSL and D-SRSL in shallow and deep pipelines respectively. Whereas the pipeline throughput of the experimental circuits depends on the specific SRSL technique used for pipelining, their area cost tends to be comparable regardless of the SRSL technique used.

## **7.2 Future Work**

While the research in this dissertation explored the inner working of three SRSL pipelines, namely S-SRSL, P-SRSL, and D-SRSL pipelines, and proposed a synthesis framework for such pipelines, this research raised during its course an additional set of questions that can be addressed as an extension to this dissertation:

- (i) Incorporation of interconnect effects as a factor which can affect the performance of the pipeline [73-75]. Preliminary examination of the three pipelines suggests that this effect may be highly relevant in the P-SRSL pipeline. In this pipeline, the phase signal leaves the last stage to drive the AND gate of each inter-stage latch, thus acting as a long synchronizing signal that spans the entire length of the pipeline. It remains to be seen how far this signal can travel before its RC effects starts to affect the correct operation of the pipeline.
- (ii) Refinement of the delay models of the three pipelines by incorporating the same interconnect effects. These effects are considered important in delays exacerbated by high fanout gates in large gate netlists, which are prevalent in most datapaths.
- (iii) Incorporation of power effects on the performance of the three pipelines. Although pipelining has been used to alleviate power effects [76, 77], deep pipelining can, in some cases, have the reverse effect by increasing the power budget of a pipeline, which in return will degrade its overall performance. In the case of SRSL pipelines, it is not known at what point pipelining ceases to



alleviate power consumption and subsequently their heat dissipation. In addition, it is not well understood how much of the performance of the pipelines is caused by their power budgets.

- (iv) Refinement of the pipeline synthesis algorithm by taking into account the fanout delay of each net in the pipelined circuits. By incorporating the interconnect effects mentioned in (i), the synthesis algorithm can build an accurate model for each gate and each net in the circuit. This delay model can guide both phases of the synthesis algorithm to produce a delay-accurate gate netlist in each stage of the pipeline.
- (v) While phase II of the synthesis algorithm is completely heuristic, it is not known at this point how sub-optimal the solutions produced by phase II can be. From an optimization perspective, it would be interesting to quantify the sub-optimality of these heuristic solutions.
- (vi) From a practical perspective, if the approximative power of the heuristic used in phase II is not satisfactory, it can be used as a strong rationale for developing a better heuristic approach which has the potential to reduce the sub-optimality of the initial vertex shuffling heuristic proposed in phase II of the synthesis algorithm. The overall benefit of this improvement in the quality of the solutions produced by the heuristic in phase II is a maximal minimization of inter-stage latches of the pipelines since the latter occupy a significant portion of the overall pipeline area.
- (vii) While the circuits used to prototype the three SRS� pipelines were all combinational datapaths, it is imperative to extend pipelining techniques

based on SRSL to implement control-dominated circuits. The latter circuits are known to have feedback loops and clocked storage elements embedded within their gate netlists. A straightforward conversion of these netlists to SRSL pipelines would require the substitution of these clocked storage elements with latches and the padding of the feedback loops with matching delays as suggested in [78].

- (viii) If robust pipelining techniques based on SRSL are possible for control circuitry, suitable synthesis approaches need to be devised to synthesize SRSL pipelines for controlled datapaths without violating data rate constraints. It would be interesting to see whether it is possible to extend the current synthesis algorithm to the synthesis of controlled datapaths, or devise an entirely new algorithm for this task.

## LIST OF REFERENCES

- [1] K. Emerson, "Asynchronous design: an interesting alternative," *Tenth International Conference on VLSI Design*, 4-7 January 1997, pp. 318-320.
- [2] I. Blunno and L. Lavagno, "Automated synthesis of micro-pipelines from behavioral Verilog HDL," *International Symposium on Advanced Research in Asynchronous Circuits and Systems*, Eilat, Israel, 2000, pp. 84-92.
- [3] J. Cortadella, M. Kishnevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, *Synthesis of asynchronous controllers and interfaces*. New York: Springer-Verlag, 2002.
- [4] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, "Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers," *IEICE Transactions on Information and Systems*, vol. E80-D, no. 3, 1997, pp. 315-325.
- [5] D. A. Edwards and B. W. Toms, "Design automation and test for asynchronous circuits and systems," Information Society Technologies Programme Technical Report IST-1999-29119, 2004.
- [6] K.-J. Lin and C.-W. Kuo, "On synthesis of speed-independent circuits at STG level," *Asia and South Pacific Design Automation Conference*, 28-31 January 1997, pp. 619-624.
- [7] V. Khomenko, M. Koutny, and A. Yakovlev, "Logic synthesis for asynchronous circuits based on Petri net unfoldings and incremental SAT," *Fourth International Conference*, 16-18 June 2004, pp. 16-25.

- [8] N. Starodoubtsev, S. Bystrov, M. Goncharov, I. Klotchkov, and A. Smirnov, "Towards synthesis of monotonic asynchronous circuits from signal transition graphs," *Application of Concurrency to System Design*, 25-29 June 2001, pp. 179-188.
- [9] I. Blunno and L. Lavagno, "Designing an asynchronous microcontroller using Pipefitter," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, no. 7, July 2004, pp. 696-699.
- [10] I. Blunno and L. Lavagno, "Designing an asynchronous microcontroller using Pipefitter," *International Conference on Computer Design: VLSI in Computers & Processors*, September 2002, pp. 488-493.
- [11] C. P. Soitriou, "Implementing asynchronous circuits using a conventional EDA tool-flow," *Design Automation Conference*, New Orleans, LA, 2002, pp. 415-418.
- [12] C. P. Sotiriou and L. Lavagno, "Desynchronization: asynchronous circuits from synchronous specifications," *IEEE International SOC Conference*, Portland, OR, 2003, pp. 165-168.
- [13] J. P. Uyemura, *Introduction to VLSI circuits and systems*: John Wiley & Sons, Inc., 2002.
- [14] S. Palnitkar, *Verilog HDL: A guide to digital design and synthesis*: Sun Soft Press, 1996.
- [15] I. E. Sutherland and J. Ebergen, "Computers without clocks," *Scientific American*, August 2002, pp. 62-69.
- [16] S. Hauck, "Asynchronous design methodologies: an overview," *Proceedings of the IEEE*, vol. 83, no.1, January 1995, pp. 69-93.

- [17] C. H. Van Berkel, M. B. Josephs, and S. M. Nowick, "Applications of asynchronous circuits," *Proceedings of the IEEE*, vol. 87, no. 2, February 1999, pp. 223-233.
- [18] S. M. Nowick, "Automatic synthesis of burst-mode asynchronous controllers," *Department of Electrical Engineering and Computer Science: Stanford University*, March 1993.
- [19] V. George and J. M. Rabaey, *Low-energy FPGAs: architecture and design*, Kluwer Academic Publishers, 2001.
- [20] M. Donno, A. Ivaldi, L. Benini, and E. Macii, "Clock-tree power optimization based on RTL clock-gating," *Design Automation Conference*, 2003, pp. 622-627.
- [21] A. Farrahi, C. Chen, A. Srivastava, G. Tellez, and M. Sarrafzadeh, "Activity-driven clock design," *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, vol. 20, no. 6, June 2001, pp. 705-714.
- [22] Fulcrum Microsystems, "Asynchronous circuit technology: an alternative for high-speed semiconductors," available at [http://www.fulcrummicro.com/Library/async\\_wp.htm](http://www.fulcrummicro.com/Library/async_wp.htm), 2004.
- [23] J. Rennert and D. Hafeman, "Clock domain modeling is essential in high density SOC design," *EE Times*, vol. 36, no.1, June 2003.
- [24] S. M. Nowick and D. L. Dill, "Automatic synthesis of locally-clocked asynchronous state machines," *IEEE International Conference on Computer-Aided Design*, 11-14 November 1991, pp. 318-321.

- [25] S. M. Nowick and D. L. Dill, "Synthesis of asynchronous state machines using a local clock," *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 14-16 October 1991, pp. 192-197.
- [26] Miller, E. Raymond, *Switching theory*: New York, Wiley, 1965.
- [27] J. Sparso and S. Furber, *Principles of asynchronous circuit design*: Kluwer Academic Publishers, 2001.
- [28] A. J. Martin, "The limitation to delay-insensitivity in asynchronous circuits," *Advanced Research in VLSI: Proceedings of the Sixth MIT Conference*, 1990, pp. 263-278.
- [29] J. C. Ebergen, "A formal approach to designing delay-insensitive circuits," *Distributed Computing*, vol. 5, no. 3, 1991, pp. 107-119.
- [30] C. Mead and L. Conway, *Introduction to VLSI systems*: Addison-Wesley, 1980.
- [31] W. Hwang, G. D. Gristede, P. Sanda, S. Y. Wang, and F. Heidel, "Implementation of a self-resetting CMOS 64-bit parallel adder with enhanced testability," *IEEE Journal of Solid-State Circuits*, vol. 34, no. 8, August 1999, pp. 1108-1117.
- [32] G. Jung, V. Sudarajan, and G. E. Sobelman, "A robust self-resetting CMOS 32-bit parallel adder," *IEEE International Symposium on Circuits and Systems*, 2002, pp. I/473-I/476.
- [33] A. E. Dooply and K. Y. Yun, "Optimal clocking and enhanced testability for high-performance self-resetting domino pipelines," *20th Conference on Advanced Research in VLSI*, 1999, pp. 200-214.

- [34] S. C. Smith, R. F. DeMara, J. S. Yuan, M. Hagedorn, and D. Ferguston, "Speedup of delay-insensitive digital systems using NULL cycle reduction," *International Workshop on Logic and Synthesis*, 2001.
- [35] S. C. Smith, R. F. DeMara, J. S. Yuan, M. Hagedorn, and D. Ferguston, "NULL convention multiply and accumulate with conditional rounding, scaling and saturation," *Journal of Systems Architecture*, vol. 47, no. 12, June 2002, pp. 977-998.
- [36] G. E. Sobelman and K. M. Fant, "CMOS circuit design of threshold gates with hysteresis," *IEEE Transactions on Circuits and Systems*, vol. 2, no. 2, June 1998, pp. 61-64.
- [37] K. M. Fant and S. A. Brandt, "NULL convention logic: a complete and consistent logic for asynchronous digital circuit synthesis," *International Conference on Application Specific Systems, Architectures, and Processors*, 1996, pp. 261-273.
- [38] T. Verhoeff, "Analyzing specifications for delay-insensitive circuits," *Fourth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 1998, pp. 172-183.
- [39] T. Verhoeff, "Encyclopedia of Delay-Insensitive Systems (EDIS)," available at <http://edis.win.tue.nl/>.
- [40] W. C. Mallon, J. T. Udding, and T. Verhoeff, "Analysis and applications of the XDI model," *Fifth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 1999, pp. 231-242.

- [41] W. C. Mallon and J. T. Udding, "Building finite automata from DI specifications," *International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 30 March-2 April 1998, pp. 184-193.
- [42] K. van Berkel, J. Kessels, M. Roncken, R. Saeijs, and F. Schalij, "The VLSI-programming language Tangram and its translation into handshake circuits," *European Conference on Design Automation*, 1991, pp. 384-389.
- [43] J. Kessels and A. Peeters, "The Tangram framework: asynchronous circuits for low power," *Asia and South Pacific Design Automation Conference*, 2001, pp. 255-260.
- [44] K. Y. Yun and D. L. Dill, "Automatic synthesis of extended burst-mode circuits. II. (Automatic synthesis)," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 2, February 1999, pp. 118-132.
- [45] K. Y. Yun and D. L. Dill, "Automatic synthesis of extended burst-mode circuits. I. (Specification and hazard-free implementations)," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 2, February 1999, pp. 101-117.
- [46] T. Murata, "Petri Nets: Properties, analysis and application," *Proceedings of the IEEE*, vol. 77, no. 4, 1989, pp. 541-574.
- [47] C. J. Myers, *Asynchronous circuit design*: John Wiley & Sons, Inc., 2001.
- [48] A. Yakovlev, A. Semenov, A. M. Koelmans, and D. J. Kinniment, "Petri nets and asynchronous circuit design," *IEE Colloquium on Design and Test of Asynchronous Systems*, 1996, pp. 8/1-8/6.



- [49] I. E. Sutherland, "Micropipelines," *Communications of the ACM*, vol. 32, no. 6, 1989, pp. 270-733.
- [50] M. Ligthart, K. M. Fant, R. Smith, A. Taubin, and A. Kondratyev, "Asynchronous design using commercial HDL synthesis tools," *Advanced Research in Asynchronous Circuits and Systems*, 2-6 April 2000, pp. 114-125.
- [51] J. Fragoso, G. Sicard, and M. Renaudin, "Automatic generation of 1-of-M QDI asynchronous adders," *Integrated Circuits and Systems Design*, September 2003, pp. 149-154.
- [52] R. Smith and M. Ligthart, "High-level design for asynchronous logic," *Asia and South Pacific Design Automation Conference*, February 2001, pp. 431-436.
- [53] S. Chakraborty, K. Y. Yun, and D. L. Dill, "Timing analysis of asynchronous systems using time separation of events," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 8, August 1999, pp. 1061-1076.
- [54] M. F. Robert, M. N. Steven, T. Michael, K. J. Niraj, L. Bill, and P. Luis, "MINIMALIST: an environment for the synthesis, verification and testability of burst-mode asynchronous machines," Columbia University, Computer Science Department, July 1999.
- [55] K. Y. Yun, "Automatic synthesis of extended burst-mode circuits using generalized C-elements," *European Design Automation Conference*, Geneva, Switzerland, 1996, pp. 290-295.

- [56] I. Blunno, J. Cortadella, A. Kondratyev, L. Lavagno, K. Lwin, and C. Sotiriou, "Handshake protocols for de-synchronization," *International Symposium on Asynchronous Circuits and Systems*, 2004, pp. 149-158.
- [57] Z. Shengxian, L. Weidong, J. Carlsson, K. Palmkvist, and L. Wanhammar, "An asynchronous wrapper with novel handshake circuits for GALS systems," *International Conference on Communications, Circuits and Systems*, West Sino Expositions, 2002, pp. 1521-1525.
- [58] A. Peeters and K. van Berkel, "Single-rail handshake circuits," *Second Working Conference on Asynchronous Design Methodologies*, 1995, pp. 53–62.
- [59] A. Alsharqawi and A. Ejnioui, "Clockless pipelining for coarse grain datapaths," *accepted in 19th International Conference on VLSI Design and 5th International Conference on Embedded Systems*, 2006.
- [60] A. Ejnioui and A. Alsharqawi, "A clockless reconfigurable array based on self-resetting logic," *Multi-conference on Systemics, Cybernetics, and Informatics*, 2004, pp. 61-66.
- [61] A. Ejnioui and A. Alsharqawi, "Self-resetting stage logic pipelines," *ACM Great Lakes Symposium on VLSI*, 2004, pp. 174-177.
- [62] A. Ejnioui and A. Alsharqawi, "Pipeline design based on self-resetting stage logic," *IEEE Computer Society Annual Symposium on VLSI*, 2004, pp. 254-257.
- [63] A. Ejnioui and A. Alsharqawi, "Coarse-grain clockless pipelining based on self-resetting stage logic," *submitted to the Journal of Circuits, Systems, and Computer*.

- [64] A. Alsharqwi and A. Ejnoui, "A clockless pipelining technique based on self-resetting stage logic," *submitted to the IEEE Transactions on Circuits and Systems*.
- [65] J. B. Sulistyo and D. S. Ha, "A new characterization method for delay and power dissipation of standard library cells," *VLSI Design*, vol. 15, no. 3, 2002, pp. 667-678.
- [66] J. B. Sulistyo, J. Perry, and D. S. Ha, "Developing standard cells for TSMC 0.25um technology under MOSIS deep rules," Department of Electrical and Computer Engineering, Virginia Tech Technical Report VISC-2003-01, November 2003.
- [67] A. Ejnoui and A. Alsharqawi, "Pipeline-level control of self-resetting pipelines," *Euromicro Symposium on Digital System Design*, 2004, pp. 342-349.
- [68] A. Ejnoui and A. Alsharqawi, "Pipeline level control of self-resetting stage logic pipelines," *IEEE Northeast Workshop on Circuits and Systems*, 2004, pp. 389-392.
- [69] S. C. Smith, R. F. DeMara, J. S. Yuan, M. Hagedorn, and D. Ferguston, "Delay-insensitive gate-level pipelining," *Integration: the VLSI Journal*, vol. 30, no. 2, October 2001, pp. 103-131.
- [70] G. DeMicheli, *Synthesis and optimization of digital circuits*: McGraw-Hill, 1994.
- [71] R. Oreifej, A. Alsharqawi, and A. Ejnoui, "Pipeline synthesis of SRSL circuits," *accepted in IEEE International Conference on Electronics, Circuits and Systems*, 2005.

- [72] A. Alsharqawi and A. Ejnoui, "Synthesis of self-resetting stage logic pipelines," *IEEE Computer Society Annual Symposium on VLSI*, 2005, pp. 260-263.
- [73] S. Bothra, B. Rogers, and M. Kellam, "Analysis of the effects of scaling on interconnect delay in ULSI circuits," *IEEE Transactions on Electron Devices*, vol. 40, no. 3, March 1993, pp. 591-597.
- [74] K. K. Ryu, A. Talpasanu, V. J. Mooney, and J. A. Davis, "Interconnect delay aware RTL Verilog bus architecture generation for an SOC," *Advanced System Integrated Circuits Conference*, 2004, pp. 176-179.
- [75] Y. Zhang, Q. Zhou, X. Hong, and Y. Cai, "Path-based timing optimization by buffer insertion with accurate delay model," *International Conference on ASICs*, 21-24 October 2003, pp. 89-92.
- [76] A. Hartstein and T. R. Puzak, "Optimum power/performance pipeline depth," *IEEE/ACM International Symposium on Microarchitecture*, 2003, pp. 117-125.
- [77] V. Srinivasan, D. Brooks, M. Gschwind, P. Bose, V. Zyuban, P. N. Strenski, and P. G. Emma, "Optimizing pipelines for power and performance," *35th Annual IEEE/ACM International Symposium on Microarchitecture*, 2002, pp. 333-344.
- [78] R. O. Ozdag and P. A. Beerel, "High-speed QDI asynchronous pipelines," *Eighth International Symposium on Asynchronous Circuits and Systems*, 2002, pp. 13-22.