2006

Synthesis Of Self-resetting Stage Logic Pipelines

Rashad Oreifej
University of Central Florida

SYNTHESIS OF SELF-RESETTING STAGE LOGIC PIPELINES

by

RASHAD OREIFEJ
B.S. University of Jordan, 2000

A thesis submitted in partial fulfillment of the requirements
for the degree of Masters of Science
in the School of Electrical Engineering and Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Summer Term
2006

ABSTRACT

As designers began to pack multi-million transistors onto a single chip, their reliance on a global clocking signal to orchestrate the operations of the chip has started to face almost insurmountable difficulties. As a result, designers started to explore clockless circuits to avoid the global clocking problem. Recently, self-resetting circuits implemented in dynamic logic families have been proposed as viable clockless alternatives. While these circuits can produce excellent performances, they display serious limitations in terms of area cost and power consumption. A middle-of-the-road alternative, which can provide a good performance and avoid the limitations seen in dynamic self-resetting circuits, would be to implement self-resetting behavior in static circuits. This alternative has been introduced recently as *Self-Resetting Stage Logic* and used to propose three types of clockless pipelines. Experimental studies show that these pipelines have the potential to produce high throughputs with a minimum area overhead if a suitable synthesis methodology is available.

This thesis proposes a novel synthesis methodology to design and verify clockless pipelines implemented in SRSL by taking advantage of the maturity of current CAD tools. This methodology formulates the synthesis problem as a combinatorial analytical problem for which a run-time efficient exact solution is difficult to derive. Consequently, a two-phase algorithm is proposed to synthesize these pipelines from gate netlists subject to user-specified constraints. The first phase is a heuristic based on the *as-soon-as-possible* scheduling strategy in which each gate of the netlist is assigned to a single pipeline stage without violating the period constraint of each pipeline stage. On the other hand, the second phase consists of a heuristic, based on the

Kernighan-Lin partitioning strategy, to minimize the number of nets crossing each pair of adjacent pipeline stages. The objective of this optimization is to reduce the number of latches separating pipeline stages since these latches tend to occupy large areas.

Experiments conducted on a prototype of the synthesis algorithm reveal that these self-resetting stage logic pipelines can easily reach throughputs higher than 1 GHz. Furthermore, these experiments reveal that the area overhead needed to implement the self-resetting circuitry of these pipelines can be easily amortized over the area of the logic embedded in the pipeline stages. In the overall, the synthesis methods developed for SRSL produce low area overhead pipelines for wide and deep gate netlists while it tends to produce high throughput pipelines for wide and shallow gate netlists. This shows that these pipelines are mostly suitable for coarse-grain datapaths.

ACKNOWLEDGMENTS

I would like to thank all who made this possible either directly by their contribution in this project, or indirectly by their support and love.

I wish to express my sincere gratitude to Dr. Abdel Ejnioui for his great support, continuous supervision, and valuable contribution in this thesis.

I also would like to thank my friend Abdelhalim Alsharqawi for the great team-spirit and professional atmosphere we experienced in working together through the many challenges in this project.

I'd like to thank my family in Jordan and my family in Orlando for their copious support, love, and care for which I am so grateful, and without which, it would have not been possible.

Finally, I'd like to thank all my friends here in Orlando who by their earnest friendship encouraged me to carry out this achievement.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER ONE: INTRODUCTION

This chapter describes the problems caused by the reliance on global clocking to synchronize the operations of digital circuits. Faced with these problems, designers are exploring other classes of circuits which do not rely on clocking. In this context, the chapter discusses a class of clockless circuits known as self-resetting circuits. Because these circuits suffer from serious limitations in spite of their high performances, the chapter introduces briefly self-resetting stage logic and its pipelining schemes. Based on this new self-resetting technique, it argues for a synthesis methodology that is suitable to support clockless pipelines.

1.1 Limitations of Clocked Circuits

Clocked circuits have been dominating digital design for some time. Because they are synchronized by a global signal, these circuits are easy to build and verify. By abstracting away the complex interactions between circuit signals in the time domain, the timing analysis of these circuits is greatly simplified [1]. This simplification is narrowed to the analysis of delays on the critical path of the underlying gate netlist of the circuit. In essence, the design process of digital circuits is reduced to embedding combinational logic between clocked registers. This approach simplifies further the timing analysis by ignoring completely the impact of unwanted signal transitions between clock events. As time went by, interest grew in designing larger clocked circuit to meet new emerging applications. At the same time, market forces began to compel designers to reduce the time-to-market of newly introduced technology products in order to maximize potential profits in the market. The combination of these factors contributed significantly to the development of automatic tools to design and verify these clocked circuits. This development effort culminated in the wide acceptance of a unified design methodology supported by widely available CAD tools. While these change were taking place, the quick pace of innovation in CMOS technology 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 of which the reliance on a clock signal to orchestrate logic operations across an entire chip seems to be the most important [2]. This problem is considered the primary cause of three consequential obstacles in current VLSI design [3]:

(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 no guarantee that the power budget will remain within initial estimates. Getting the power budget right is critical since excessive power consumption may disrupt the correct operations of the circuit.

(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. This additional area can increase significantly the final cost of the circuit.

1.2 Self-Resetting Circuits

As global clocking is causing these problems, designers are exploring the alternative of asynchronous circuits [4]. Recently, a special family of dynamic circuit, known as self-resetting logic, has been exploited successfully in memory design [5, 6]. Only a few attempts have been made to study the effectiveness of this logic family in implementing asynchronous datapath circuits [7, 8]. Self-resetting behavior can be described as the ability of a logic block to reset its output pulse a short time after it has been asserted. The reset signal is often generated within the block based on the output pulse. Depending on the implementation of the self-resetting behavior, the granularity of the block can range from a single gate to a large macro. Most self-resetting dynamic circuits are fine-grain implementations targeted to high performance arithmetic circuits. Since the majority of these circuits are pulse-mode circuits, they are usually organized into pulsed latch-free pipelines. These pipelines can produce high throughputs that are made possible by the fast cycle time of self-resetting dynamic circuits. Although dynamic circuits exhibit smaller area overhead than static circuits, the implementation of self-resetting dynamic circuits tend to occupy larger areas as shown in Figure 1.1 [9]. This area overhead is primarily caused by the self-resetting circuitry and additional buffering to equalize signal delay on various logic paths.

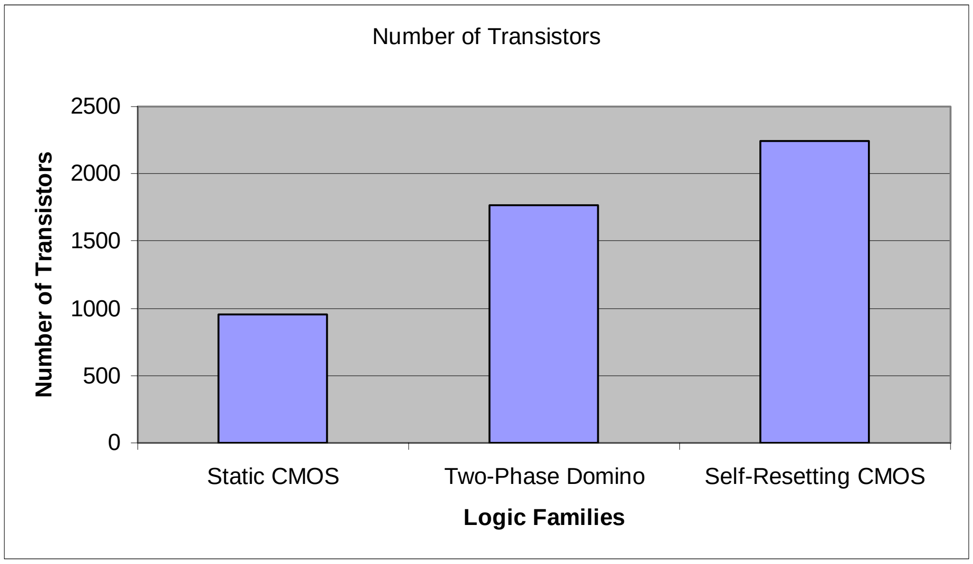

Figure 1.1: Area cost of an add-compare-select unit of a Viterbi Decoder implemented in three logic families.

While it is known that dynamic circuits can be power hungrier than static circuits, self-resetting dynamic circuits tend to consume substantially more power than even their clocked dynamic counterparts as shown in Figure 1.2 [10].

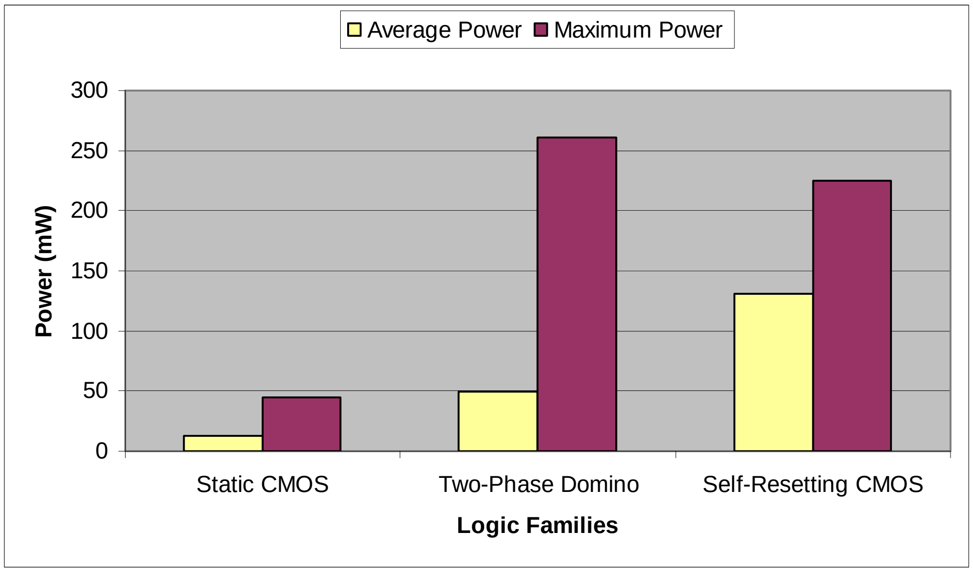

Figure 1.2: Power consumption of an add-compare-select unit of a Viterbi Decoder implemented in three logic families.

As for timing requirements, with the exception of a few self-resetting approaches [9, 11], most self-resetting circuits rely heavily on equalization of path delays [8, 12]. In fact, because some self-resetting circuits are intended for wave pipelining [7], rough padding is extensively applied on all paths in order to minimize the difference between fast and slow paths [13] as shown in Figure 1.3.

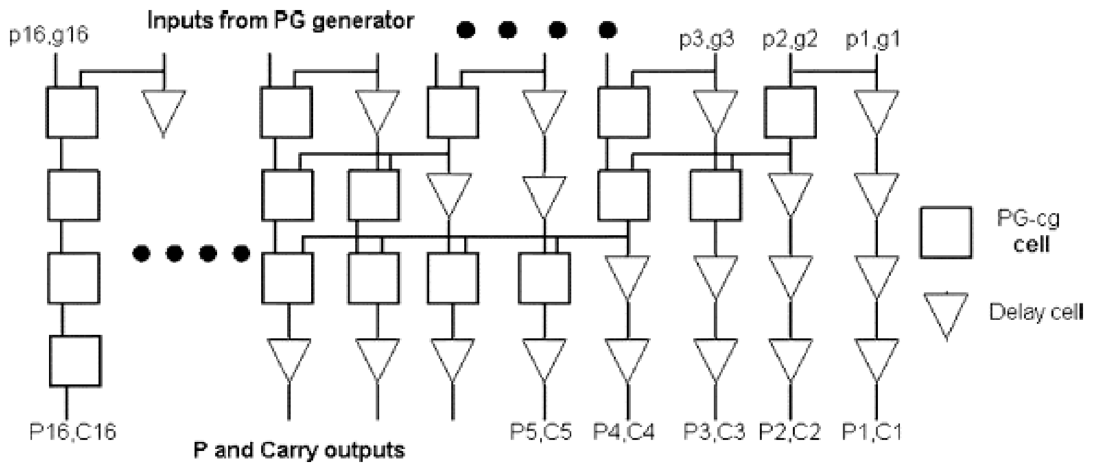

Figure 1.3: Rough padding in a carry generator block of a self-resetting carry lookahead adder [7].

Buffers can occupy up to 40% of the circuit area in some cases [10]. As a result, additional effort must be invested in meeting timing constraints that are specific to these circuits. This significant demand on maintaining signal integrity is exacerbated further by the pulse-driven nature of self-resetting circuits.

Since self-resetting behavior can be realized using any circuit family, one can opt to use static CMOS instead. Doing so presents several advantages. In static circuits, signals do not have to be pulses. Instead, voltage levels are sufficient to support self-resetting behavior. If voltage levels are used, the stringent timing constraints encountered in pulse mode circuits can be relaxed without affecting circuit robustness. In addition, significant power savings can be realized by using static circuits. While static circuits are not as fast as dynamic circuits, one can overcome more or less effectively this difficulty by adopting performance-enhancing techniques such as aggressive pipelining and the exclusion of the reset circuitry from the critical path of the datapath. Moreover, the use of static self-resetting latch-based pipelines is particularly beneficial since their timing verification is reduced to the verification process encountered in synchronous logic. Furthermore, these self-resetting circuits can be synthesized and verified using current synthesis and verification tools. It is worth noting that there are no mature synthesis and verification tools available for dynamic circuits [14]. In fact, the design community ought to exploit the maturity of current CAD tools to build large asynchronous architectures which go beyond proof-of-concepts designs. To do so, this community can pursue a design methodology which adopts as much as possible the existing CAD design flow and deviates from it as little as possible [15].

1.3 Self-Resetting Stage Logic and Its Synthesis Methodology

Following the objective of maximum adoption of the current design methodology, a novel coarse-grain self-resetting technique, called *self-resetting stage logic* implemented in static CMOS, has been recently proposed [16]. Based on this self-resetting stage logic (SRSL) technique, three pipelining schemes have been proposed where the first and second pipelines require that stages have equal delays while the third pipeline can tolerate any arbitrary stage delay [17-21]. This thesis proposes a synthesis methodology to build these SRSL pipelines [22-24]. This methodology operates on flat gate netlists synthesized by current CAD tools and implemented in standard library cells used in ASIC design. The synthesis methodology is assessed through experimentation on benchmark circuits with various depths and breadths. As an experimental requirement, these SRSL pipelines have been synthesized and verified using current CAD tools, then implemented using a standard static CMOS cell library [25, 26].

1.4 Thesis Contributions

This thesis presents a new synthesis methodology specifically designed for pipelining SRSL logic. As mentioned before, this methodology is highly suitable for existing CAD tools. Specifically, the contributions of this thesis are as follows:

(i) A novel design methodology based on synthesizing SRSL pipelines using current CAD tools and standard cell libraries. Designing clockless circuits using this methodology is highly similar to designing digital synchronous circuits.

(ii) 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.

(iii) Since the size of the integer programming problem formulation is significantly large, 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 is accomplished 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.

(iv) Pipelining experiments conducted on the SRSL pipeline show that they can reach throughputs above the GHz range without incurring an excessive area overhead.

(v) The same pipelining experiments reveal that the area overhead remain beneficial as long as it represents a small fraction of the logic area embedded in a pipeline stage. This requirement makes SRSL pipelines highly suitable for pipelining coarse-grain datapaths.

1.5 Thesis Overview

This thesis consists of six chapters in which the current chapter presents the motivation behind the synthesis methodology of SRSL pipelines by drawing attention to the global clocking problem. Chapter 2 reviews the self-resetting circuit techniques previously described. These techniques are all implemented in dynamic CMOS. Chapter 3 introduces SRSL and describes the operations of the three pipelines based on SRSL.

Chapter 4 describes the synthesis methodology that is proposed to support the design and verification 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. Chapter 5 presents the experiments conducted on benchmark circuits in order to evaluate the performance profiles of each SRSL pipeline. Finally, Chapter 6 concludes the thesis and suggests avenues for future work.

CHAPTER TWO: RELATED WORK

In this chapter, a review of the different design techniques based on self-resetting logic is presented. Section 2.1 presents delayed reset logic while section 2.2 describes a self-resetting technique controlled by local and global reset signals. Section 2.3 describes a self-resetting technique driven by local reset signals while section 2.4 describes a dual rail self-resetting technique with input disable. Section 2.5 concludes the chapter by presenting a summary of the reviewed techniques and contrasting them to SRSL.

2.1 Delayed Reset Logic

In [11] , the authors propose a pipelined technique based on delayed self-reset logic (DSRL). The refinement of this technique is inspired from the self-reset technique proposed in [27]. DSRL is a single rail logic optimized for pipelining memory access in multimedia processors. This reset technique is driven by pulses and can be modified to accommodate voltage levels. Figure 2.1 shows the structure of a DSRL pipeline while Figure 2.2 shows timing charts of control signals within the pipeline. In DSRL, a stage can transition through three states: evaluate, reset, and recover. Before computation begins, a stage is in a quiescent state. When the inputs (in_a and in_b) are absorbed, the stage enters the evaluate state as shown in Figure 2.2. The evaluation time depends on the delay within the NMOS and PMOS networks. At the end of this state, the output (out_n or out_p) becomes stable at which point the stage enters the reset state. The stage remains in this state as long as the reset signal (rst_in) has not arrived from the previous stage. Note that this signal is also labeled rst_out on the output side of the same stage.

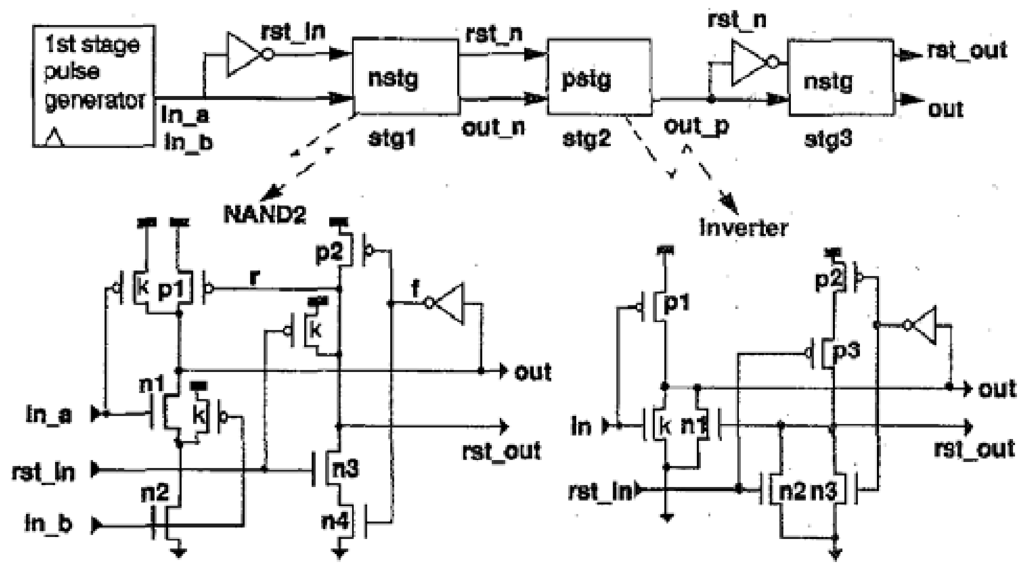

Figure 2.1: DSRL pipeline and its self-resetting circuitry.

.

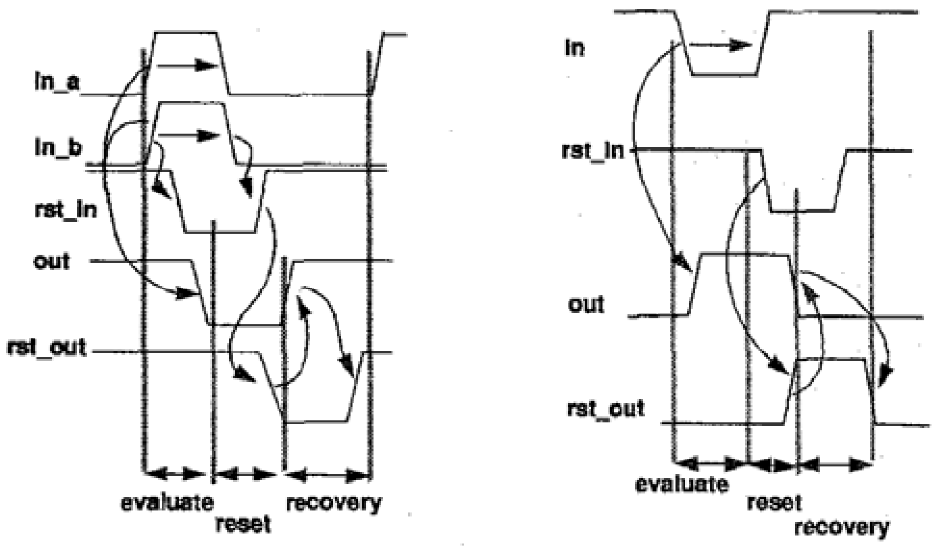

Figure 2.2: Timing chart of DSRL control signals of a pipeline stage.

As Figure 2.1 shows, the reset signal (rst_in) travels between every two adjacent stages. When the latter signal arrives, the stage enters the recovery state. This state is locally timed in each stage by insuring that transistor n3 is turned off before transistor n4 is turned on in the beginning of the evaluate state. This signal control allows the stages to have arbitrary different delays.

Implemented in domino logic, DSRL pipelines consist of alternating NMOS and PMOS stages without any latches between the stages. Although these pipelines are suitable for memory cell design, it is doubtful whether they are also suitable for datapaths or not given their fine-grain nature. In addition, their design requires that careful calibration be applied to the pulse generator located in the first pipeline stage as shown in Figure 2.1. This calibration is needed to compensate for environmental variations. Furthermore, since these pipelines are mainly custom circuits, their design and verification can be time-consuming and error-prone.

2.2 Global/Local Self-Resetting CMOS

In [8, 12], the authors propose a single-rail self-resetting technique in which the gates are reset locally (LSRCMOS). However, the gates within a large macro are reset through a global reset signal (GLSRCMOS). Figure 2.3 shows a basic GLSRCMOS gate. This gate has active-high pulsed inputs and outputs. Its non-inverting logic evaluation depends on the logic function of the NMOS tree. If the right input combination occurs at the right time, a conducting path from the TL node emerges, which leads to discharging the capacitance at the output side of the gate. This brings TL to logic 0 while the output goes up to logic 1, thus creating the leading edge of the output pulse. When the input pulse ends, the RL signal arrives by falling to logic 0, thus resetting the TL node to logic 1. After TL becomes asserted, the RH signal arrives, by rising to logic 1, to terminate the output pulse.

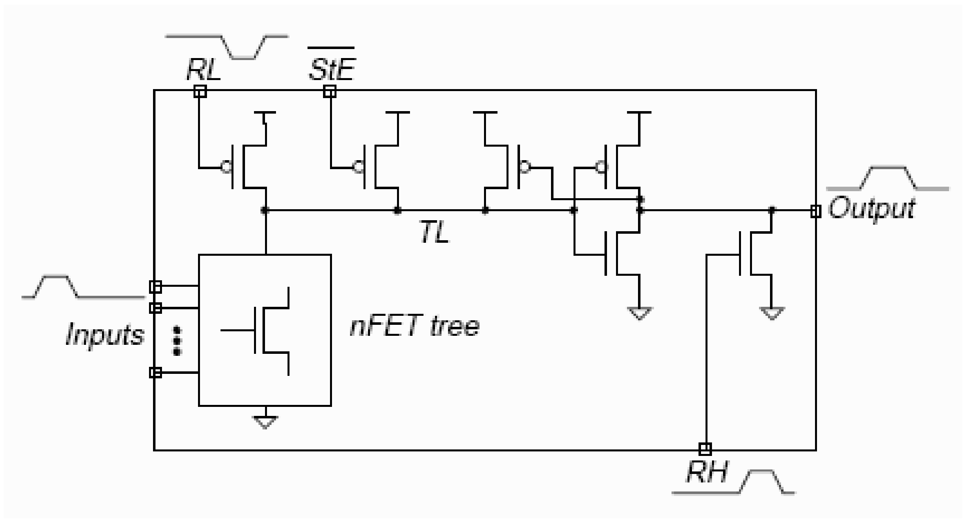

Figure 2.3: Basic SRCMOS gate.

These basic GLSRCMOS gates are incorporated within macros. An SRCMOS macro consists of a number of gates whose reset signals drive a reset generator circuit located inside the macro. Figure 2.4 shows a GLSRCMOS macro. The triggering of this generator can be realized through ORing of reset signals, majority circuits, or interlocking signals from multiple paths. As a rule, the reset generator must be triggered when the macro is active. A delay chain generates the required signals within the reset generator. Initially, a macro is in standby mode. Once it receives its triggering pulses, it enters the evaluate mode, then resets its outputs before returning to the standby mode.

In [8], the designs of a number of macros are assembled to implement a 64-bit carry lookahead adder in a 0.25 μm CMOS process. This adder can reach a throughput of 400 MHz. Since this adder uses a pipelined pulsed approach to increase its throughput, a number of buffers have been added to the adder in order to control delay on logic evaluation paths.

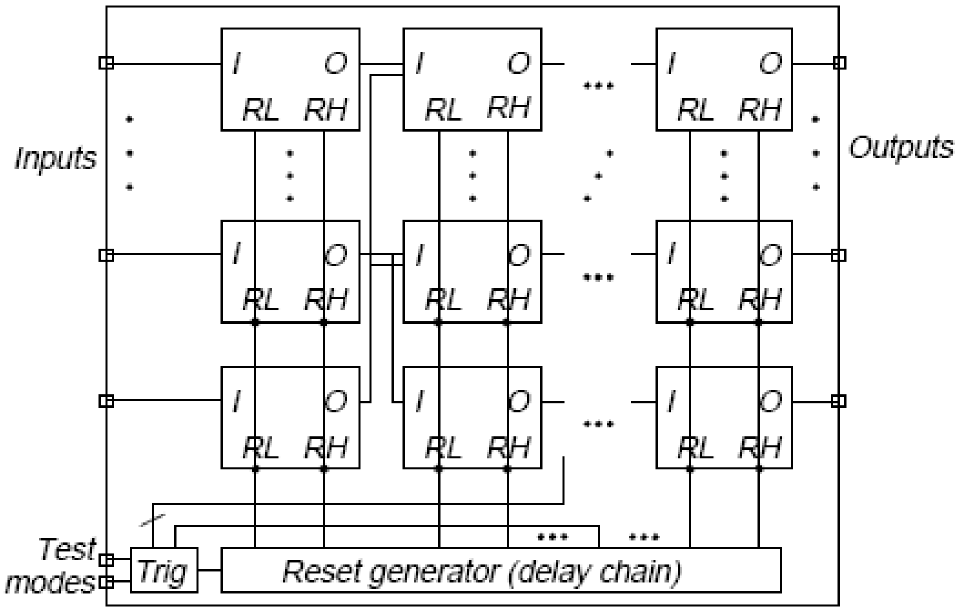

Figure 2.4: SRCMOS macro.

While the authors refer to pulse pipelining, they do not clearly describe how macros are pipelined within the adder. This omission does not clarify how two adjacent macros synchronize their state transition in order to exchange data safely. Whereas the insertion of buffers can help in overcoming timing issues, it can become quite unwieldy when dealing with large datapaths with large numbers of logic paths. At best, buffer insertion bloats the datapaths leading to a large area overhead. In addition, if macro size increases to accommodate deep logic, it may require increasing the length of the reset chains within a macro. This can be achieved by inserting inverters in this chain, which in turn complicates the timing verification of the macro. These ensuing timing difficulties explain the motivation of the authors in [12] to propose a special tool for performing accurate timing verification of GLSRCMOS circuits.

2.3 Local Self-Resetting CMOS

In [9], the authors propose a single-rail input dual-rail output self-resetting technique in dynamic CMOS. In this technique, the reset signal is generated within the stage by NORing the stage output and its complement. As a result, two NMOS networks are required to generate the output and its complement. Figure 2.5 shows the local self-resetting CMOS (LSRCMOS). In this circuit, node 1 will switch to low or high depending on the input. At the same time, node 2 will switch to the complement logic level of node 1 given the same input. At this time, both NMOS networks are in the evaluate state. Subsequently, signal f and f' go through the NOR gate whose output switches to low. The low output of the NOR gate turns on both precharge transistors connected to the reset node thus charging the capacitance at the outputs of both NMOS networks. As a result, nodes 1 and 2 switch to high to propel both NMOS networks in the reset state. At this moment, both NMOS networks are ready to accept new input pulses. Following this, signal f and f' switch both to low thus forcing the output of the NOR gate to switch to high. This in turn turns off both precharge transistors to allow both NMOS networks to evaluate the newly arrived input pulses.

Contrary to GLSRCMOS presented in the previous section, the reset signal in LSRCMOS does not go through any timing chain. As shown in Figure 2.5, the reset time remains constant regardless of the evaluation time of both NMOS networks. However, the delay through the loop consisting of an output node, the NOR gate, and the reset node should be longer than the duration of the input pulses to avoid in-fighting between the precharge transistors. This technique can be used to build latch-free pipelines in dynamic CMOS as shown in Figure 2.6.

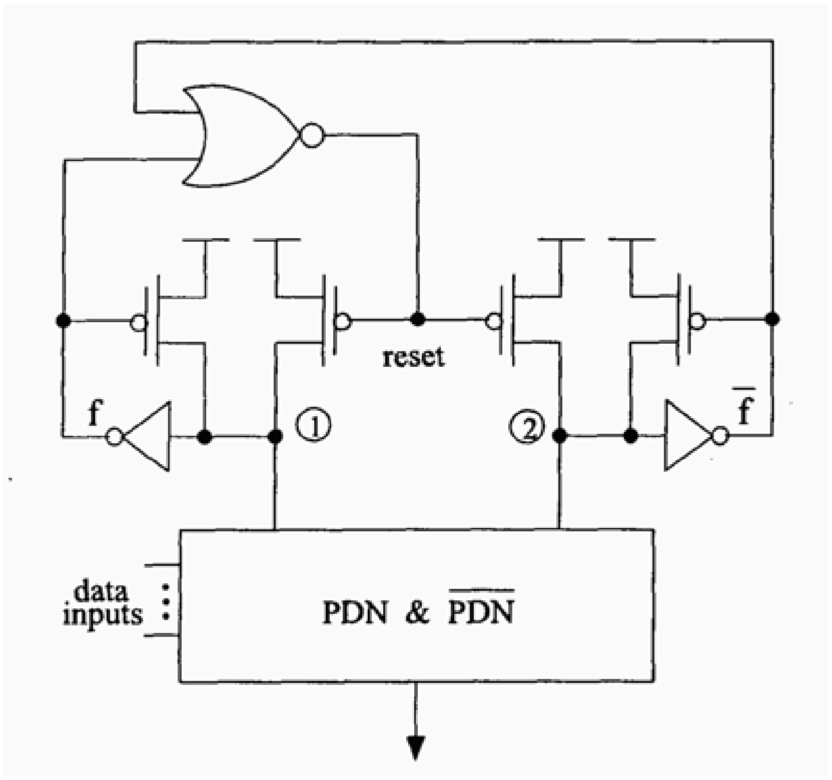

Figure 2.5: Local self-resetting CMOS.

In [10], the authors apply LSRCMOS to design an add-compare-select (ACS) unit of a Viterbi decoder. While the ACS unit is 1.71 faster than its counterpart in clocked static CMOS, it is 110 times power hungrier than its static counterpart. In addition, it occupies 2.35 times more area than its static counterpart in spite of the effort of the authors in using pulse stretchers to control path delay. While the authors claim that these stretchers reduce area overhead in contrast to buffers, they do not specify how many pulse stretchers they used within the ACS unit and how much area they occupy. In fact, a pulse stretcher consists of an SR latch whose R input is connected to a NOR gate as shown in Figure 2.7.

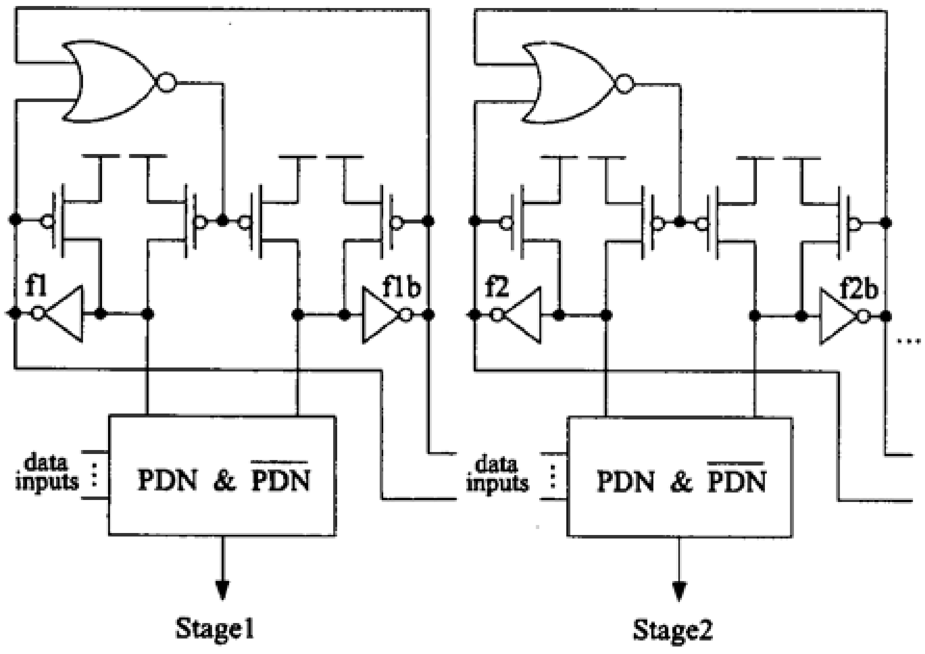

Figure 2.6: Latch-free pipeline based in LSRCMOS.

Based on popular latch layouts, an SR latch in static CMOS can easily occupy three to seven times more area than a two-input NAND gate. To scale to dynamic CMOS implementations, a rough estimate can be obtained by halving the area estimates in static CMOS. Based on this estimate, even pulse stretchers may add a substantial area overhead although it is doubtful it would be on the order of the area overhead caused by buffer insertion.

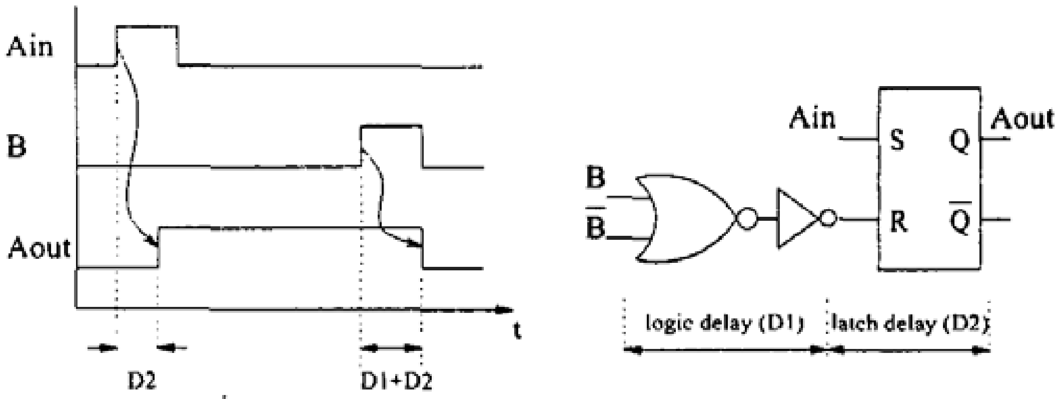

Figure 2.7: Pulse stretcher.

2.4 Dual-Rail Self-Reset Logic with Input Disable

In [7], the authors propose a dual-rail self-resetting technique, called DRSRL-ID, in which the reset signal is generated locally. Figure 2.8 shows a basic DRSRL-ID gate. The ID initials represent the input disable block shown in Figure 2.8. This block consists of an extra NMOS transistor NMe in series with the NMOS transistor NM1. When the gate is in standby mode, capacitance is pulled down to ground switching the node outn to high. In return, this makes the node rst1n switch to high to turn on the NMe transistor. At this point, the gate is ready to absorb the D input. If D becomes high, node outn switches to low, which turns on and off the PMa and NMa transistors respectively. After a short time, node rst1 switches to low to turn off the NMe transistor. When the latter device is off, the input is disabled. The discharging of the node outn causes output Y to rise, thus generating the leading edge of the output. Y is fed through inverter invFB to turn on the PM1 transistor in charge of pulling up node outn. This brings back the gate to its standby mode. As node outn starts going high, its voltage switches the transistors of the output stage forcing the Y output to go low. When Y becomes low, it deactivates the reset signal to enable input readout. As such, the gate re-enables the inputs only when the output pulse is completely formed. The layout of the basic DRSRL-ID gate forces the width of the output pulse to remain constant regardless of fanout. The width of the output pulse depends only on the output stage and the feedback loop that controls the reset signal. It is completely independent of the implementation of the NMOS network in charge of evaluating logic.

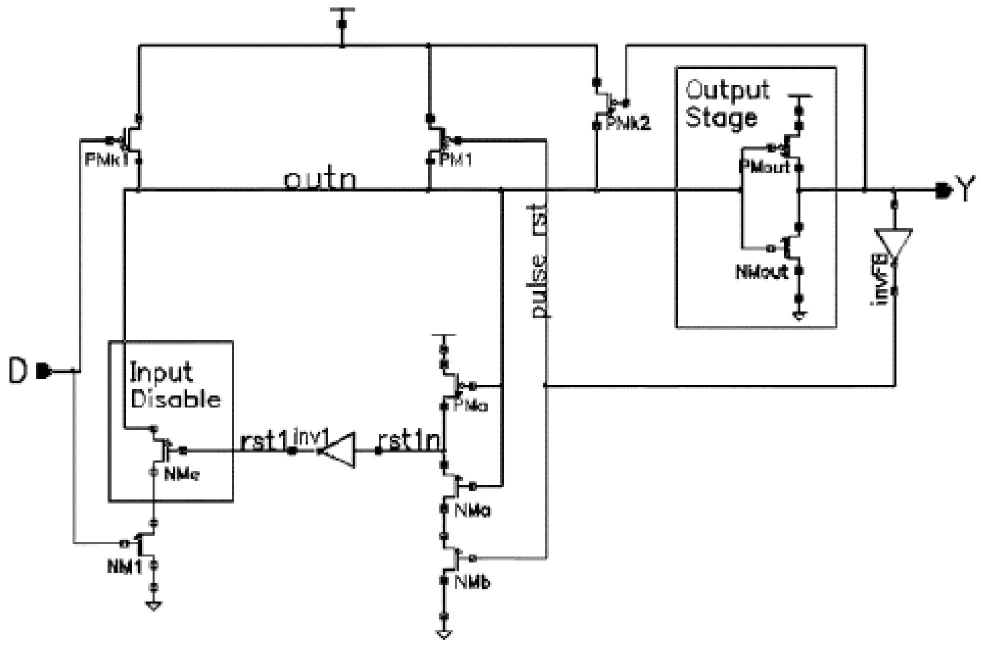

Figure 2.8: Basic DRSRL-ID gate.

The authors use this design technique to build a 16-bit wave-pipelined carry propagate adder in a 0.18 μm CMOS process which can reach 2.5 GHz throughput. Because wave pipelining aims at reducing the delay differences between long and short paths, the authors resort to extensive rough padding to reach this objective.

While timing calibration seems to be straightforward at gate level, it is not clear how it can be achieved at datapath level. In fact, if buffer insertion is used for path delay equalization, this indicates that substantial effort must be invested in timing calibration at datapath level. In addition, buffer insertion contributes to bloating datapath size. By considering the number of transistors needed to support input disabling and resetting behavior on a cell basis, it is clear that area overhead can be incurred also at cell level. In fact, some basic two-input gates can incur a penalty of 12 to 16 transistors to support their input-disabling and reset behavior.

2.5 Summary of Self-Resetting Techniques

Table 2.1 shows a summary of the self-resetting techniques reviewed in this chapter. The table shows that all the reviewed techniques are implemented in custom dynamic CMOS using pulse-driven circuits. Pulse driven circuits require precise sequencing of the input pulses. In contrast, voltage level circuits do not require such sequencing. In addition, custom circuits using dynamic CMOS require a substantial effort in verification. This effort is further hampered by a total lack of mature CAD tools destined for synthesis and verification of circuits implemented in dynamic CMOS. In the reviewed circuits, the reset signal is always tied to the output pulse making the reset signal tightly coupled to the path of evaluated logic. If glitches affect the output signals, these glitches will propagate to the reset signals resulting in a temporary or permanent disruption of the state transitions in these circuits. None of the reviewed papers speculate on the consequences of such glitches. To build datapaths, these circuits are assembled into fine-grain latch-free pipelines. While these pipelines tend to be small in area, their verification is not a trivial task. This situation is further exacerbated in the reviewed techniques that require equal stage delay across the pipeline.

In contrast to the reviewed self-resetting techniques, this thesis proposes a design technique to support SRSL which has been previously reported in [16]. Implemented in static CMOS, SRSL has adequate coarse granularity to make it suitable for implementing large datapaths.

Static circuits consume less power than dynamic circuits. Furthermore, since SRSL exploits self-resetting at datapath level, its area overhead is significantly smaller than the area overhead seen in the reviewed self-resetting techniques. The latter techniques implement self-resetting behavior at gate level instead. While dynamic circuits can display a superior performance, static circuits can provide similar performance levels if aggressive pipelining is applied in a disciplined manner. SRSL uses voltage levels instead of pulsed inputs and outputs. For circuit robustness, SRSL separates the path of self-resetting circuitry from that of logic circuitry. Since SRSL is implemented in static circuits, its pipelines use latches to separate logic stages. The insertion of latches facilitates the control of the cycle time and subsequently the timing validation of these pipelines.

Table 2.1: Summary of self-resetting techniques.

	DSRL	**LGSRCMOS**	**LSRCMOS**	**DRSRL-ID**	**SRSL**
Pulse vs. voltage level	Pulse	Pulse	Pulse	Pulse	Voltage levels
Source of reset signal	● Incoming reset from previous stage ● Outgoing reset in current stage	● Local reset in current gate ● Global reset in macro	Current stage	Current stage	● Current stage (S/P-SRSL) ● Previous stage (D-SRSL)
Destination of reset signal	Next stage	In macro	Current stage	Current stage	Previous stage
Tying of reset signal to output	Yes	Yes	Yes	Yes	No
Signal delay handling	None	Buffering	Pulse stretching	Buffering	Buffering
Path of reset signal	Logic path	Logic path	Logic path	Logic path	Control path
Stage delay	Arbitrary	Equal	Arbitrary	Equal	● Equal (S/P-SRSL) ● Arbitrary (D-SRSL)
Pipelining	No Latches	No Latches	No Latches	No Latches	Latches
CMOS	Dynamic	Dynamic	Dynamic	Dynamic	Static
Granularity	Fine	Fine	Fine	Fine	Coarse
Tools	None	Proprietary	None	None	Current CAD

This thesis proposes a design methodology which leverages the maturity of current CAD tools to synthesize and verify SRSL pipelines. The methodology does not deviate from the established design methodology used in synchronous logic. At the core of this methodology is a synthesis flow which transforms a synchronous gate netlist into an SRSL netlist before the latter is placed and routed.

CHAPTER THREE: SELF-RESETTING STAGE LOGIC

This chapter presents *self-resetting stage logic*, which is a digital design technique that is characterized by periodic oscillations. This technique can be used to establish handshaking exchanges between two computational stages in a dataflow pipeline. Its underlying concept is introduced in section 3.1. Section 3.2 describes the *stage-to-stage self-resetting stage logic* pipeline followed by the description of the *pipeline-controlled self-resetting stage logic* pipeline in section 3.3. Section 3.4 describes *delay-tolerant self-resetting stage logic* pipeline before section 3.5 concludes the chapter.

3.1 Self-Resetting Stage Logic

Self-resetting stage logic (SRSL) is a design approach which can be used to synchronize data flow between neighboring computing blocks without relying on a global clock signal. In the SRSL pipeline, each stage consists of two distinctive networks: a *combinational network* and a *reset network*. The combinational network represents the combinational logic embedded in a given stage while the reset network represents an oscillating loop used to control data transfer from one combinational network to another. The reset network consists of a two-input NOR gate whose output O feeds back one of its inputs I, while its other input is tied to the Reset global signal as shown in Figure 3.

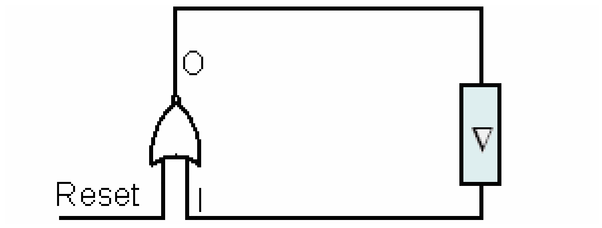

Figure 3.1: Reset network of an SRSL stage.

As long as the Reset signal is asserted, O remains 0. When the Reset signal is de-asserted, O oscillates between 0 and 1. The oscillation frequency is controlled by the delay block Δ embedded in the feedback 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 oscillates between two distinct phases in an autonomous fashion. The duration of these two phases constitutes a single oscillation cycle or *period*. This autonomous oscillation is illustrated with the *state transition graph* (STG) shown in Figure 3.2.

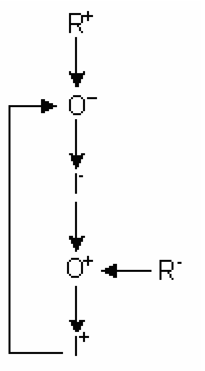

Figure 3.2: STG of the SRSL reset network.

In the above STG, O, I, and R are the three signals shown in Figure 3.1. O^+, I^+, and R^+ represent the rising transitions on those signals respectively while O^-, I^-, and R^- represent the falling transition on the same signals respectively. In addition, a directed edge connecting two transitions means that the transition at the tail of the edge precedes in time the transition at the head of the edge. The oscillations of a reset network can be used to synchronize data transfer between neighboring stages in a pipeline. To allow the combinational network of a stage sufficient time to absorb and process its inputs, SRSL prepares a stage to (i) receive inputs from the preceding stage when it is in the reset phase, and (ii) send its outputs to the following stage when it is in the evaluate phase.

3.2 Stage-to-Stage Self-Resetting Stage Logic

In stage-to-stage self-resetting stage logic (S-SRSL), the synchronization is realized between each pair of stages in the pipeline. In this pipeline, each stage is ready to absorb inputs when it enters the reset phase and produce an output when it enters its evaluate phase. As a result, data transfer occurs between two neighboring stages when the left stage is in the evaluate phase while the right stage is in the reset phase.

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. In addition to the reset network, SRSL relies on inter-stage latches to capture data moving from one stage to another. These latches are enabled when the left stage is ready to push data to the right stage in a pipeline. This occurs when the left and right stages are in the evaluate and reset phases respectively. As shown in Figure 3.3, the enable (L_i) of each inter-stage latch is tied to the output of an AND gate whose inputs are connected to the phase lines of the left and right stages. These phase lines represent the outputs of the NOR gates embedded in the reset networks of both stages. Note that the right input of the AND gate is inverted. Because the control of these inter-stage latches is exercised between each pair of pipeline stages, this synchronization technique is qualified as a stage-to-stage

SRSL. It is worth emphasizing the fact that all the stages in the pipeline should have equal cycles in order to insure correct data flow throughout the pipeline.

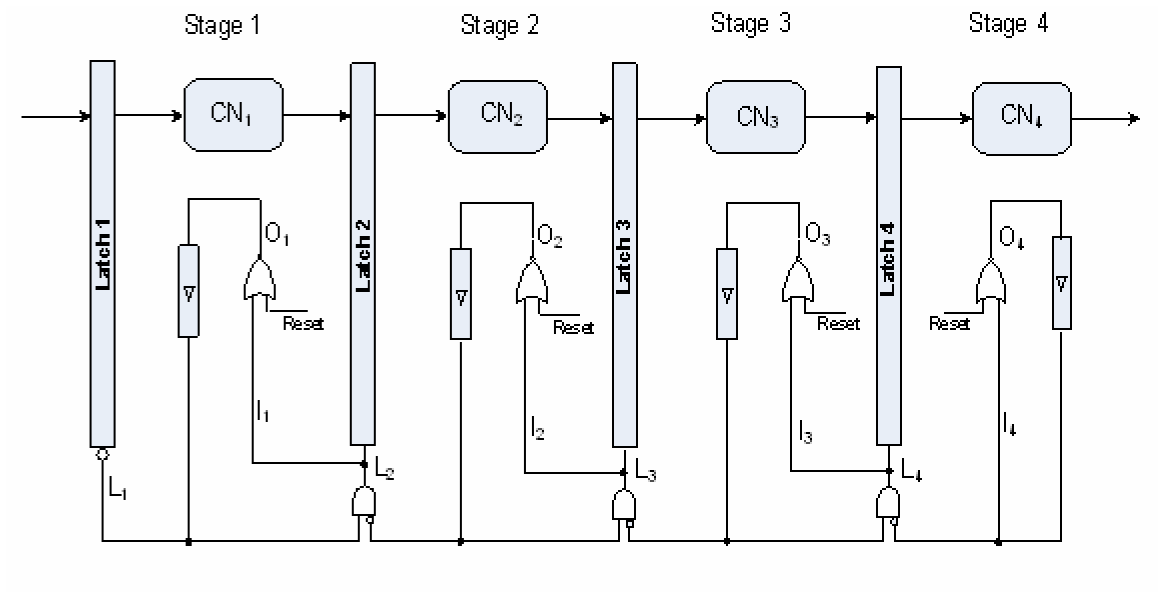

Figure 3.3: Four-stage S-SRSL pipeline.

At any oscillation 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. This periodic oscillation forces every other stage to enter the evaluate phase while the remaining stages enter their reset 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 phase.

The STG for the four-stage S-SRSL pipeline shown in Figure 3.3 is shown in Figure 3.4. 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.

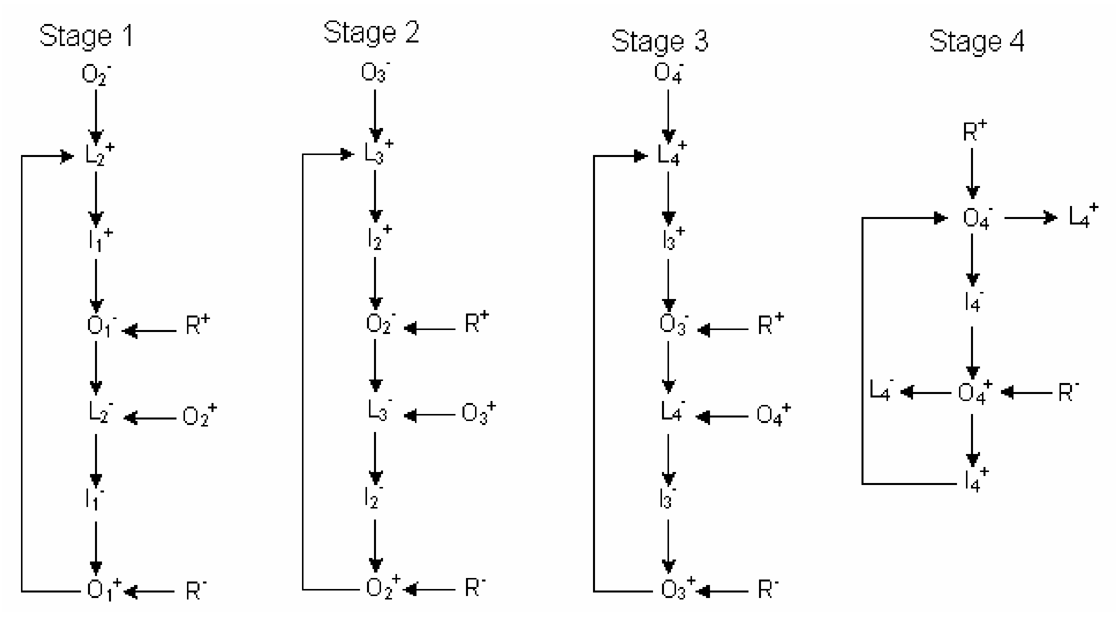

Figure 3.4: STG of the S-SRSL pipeline shown in Figure 3.3.

Figure 3.5 through Figure 3.12 show how the stages alternate between phases as data flows across the pipeline by depicting two execution cycles of a four-stage S-SRSL pipeline. The asserted and de-asserted signals are represented as solid and dashed lines respectively in these figures.

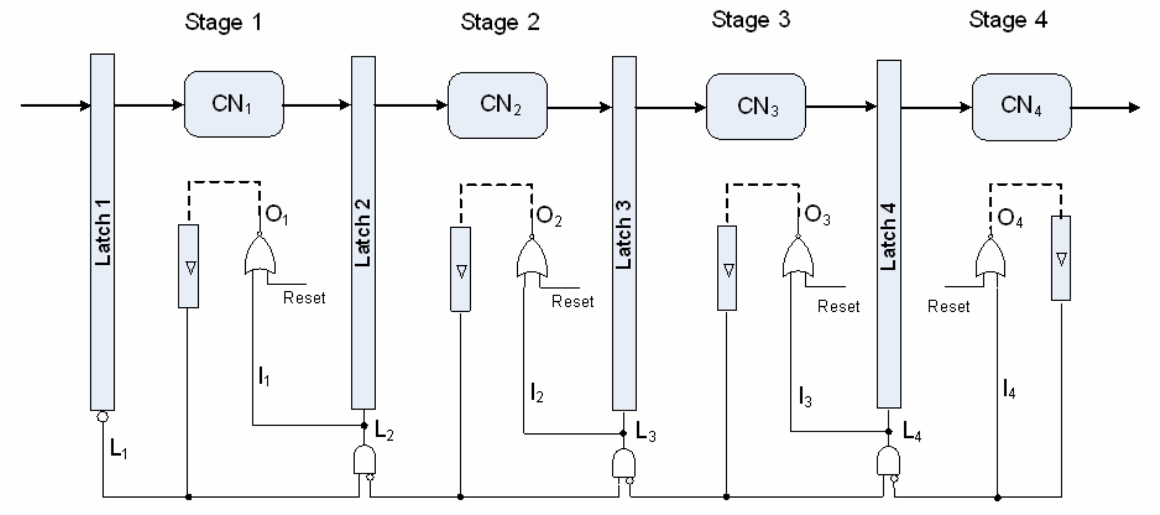

Figure 3.5: Assertion of the stage reset signals.

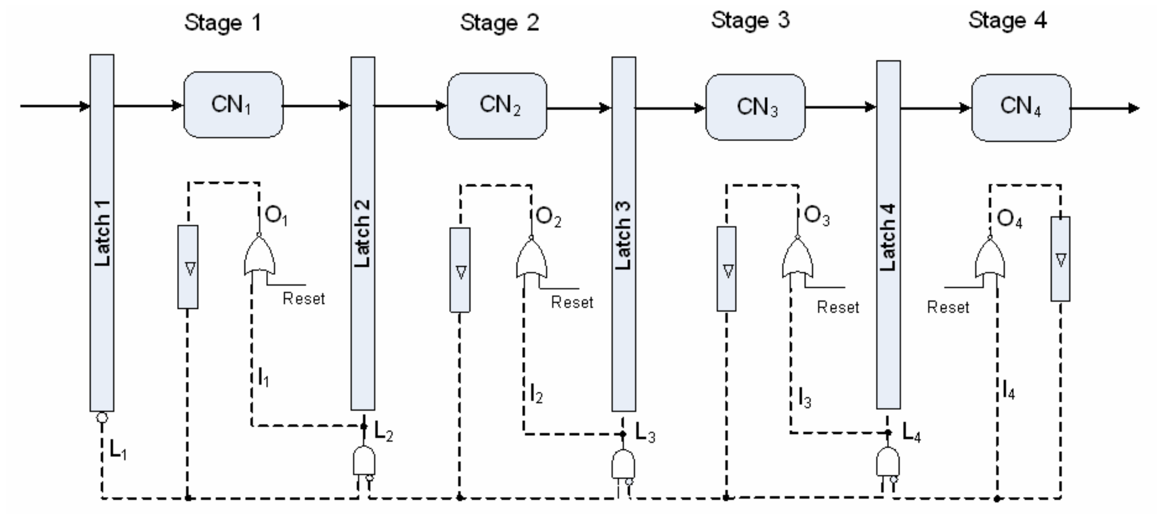

Figure 3.6: Reset phase of all stages.

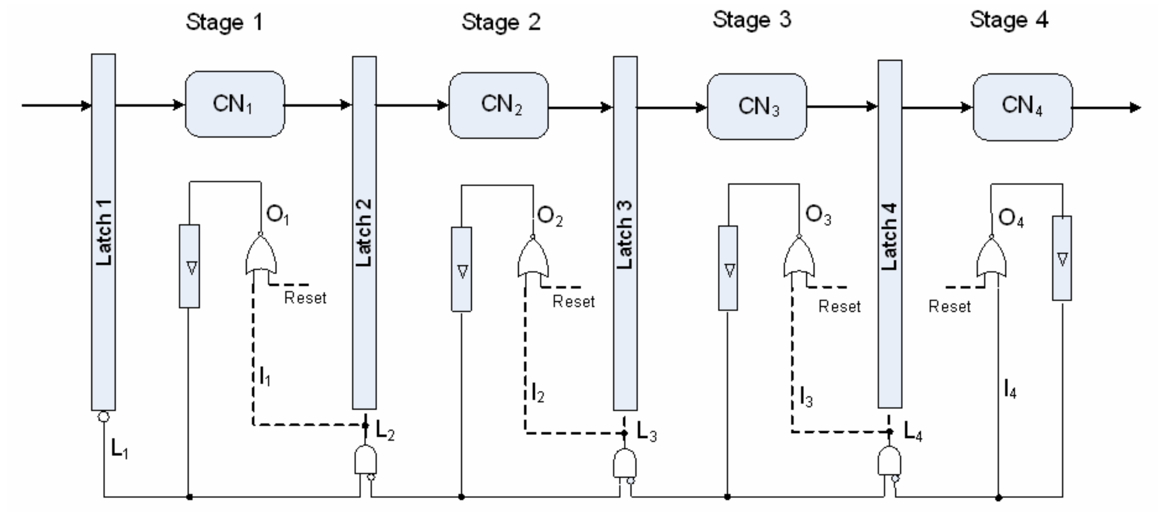

Figure 3.7: Evaluate phase of stage 4.

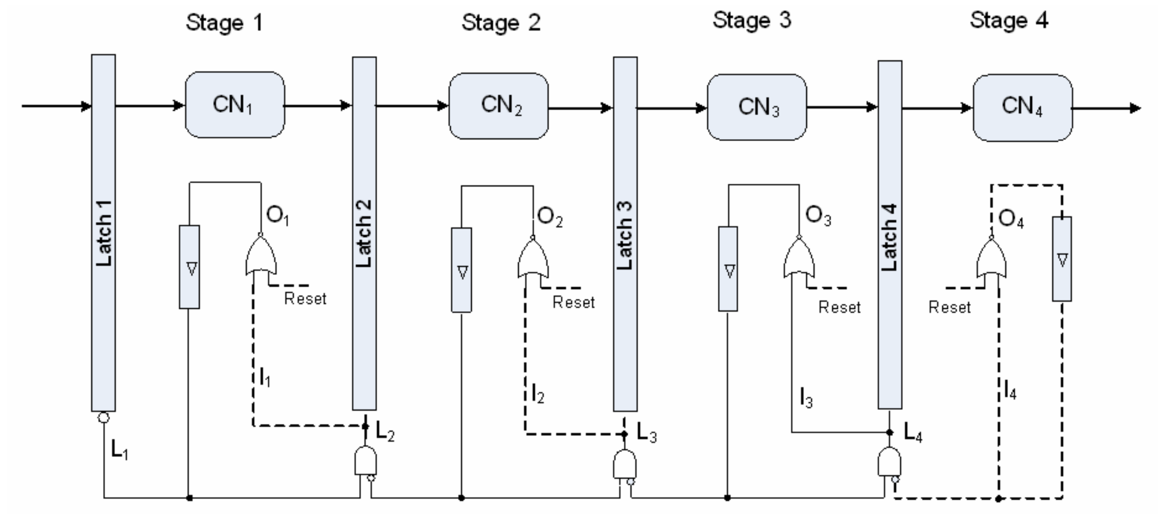

Figure 3.8: Evaluate phase of stage 3.

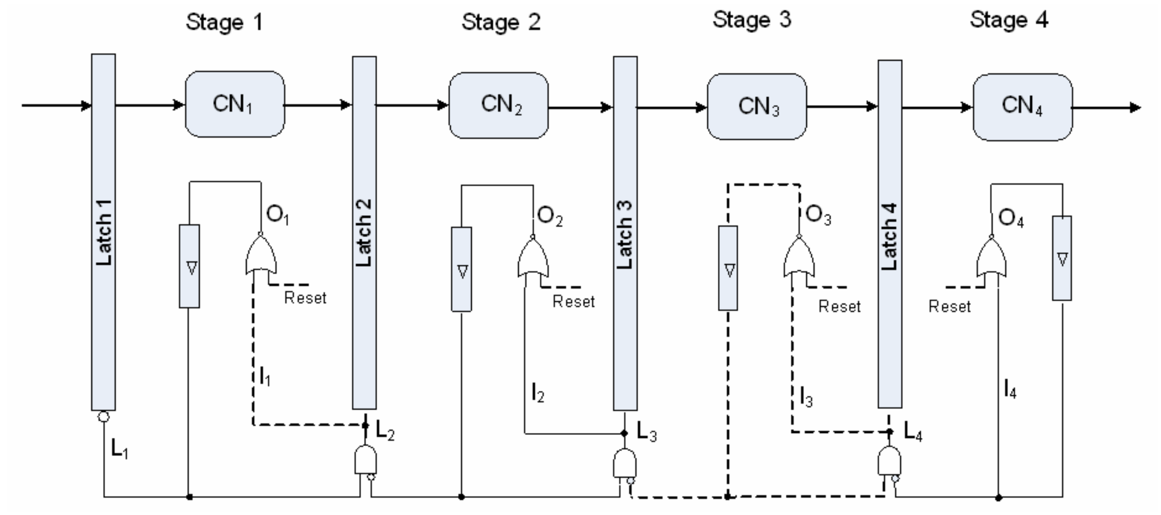

Figure 3.9: Evaluate phase of stage 2 and 4.

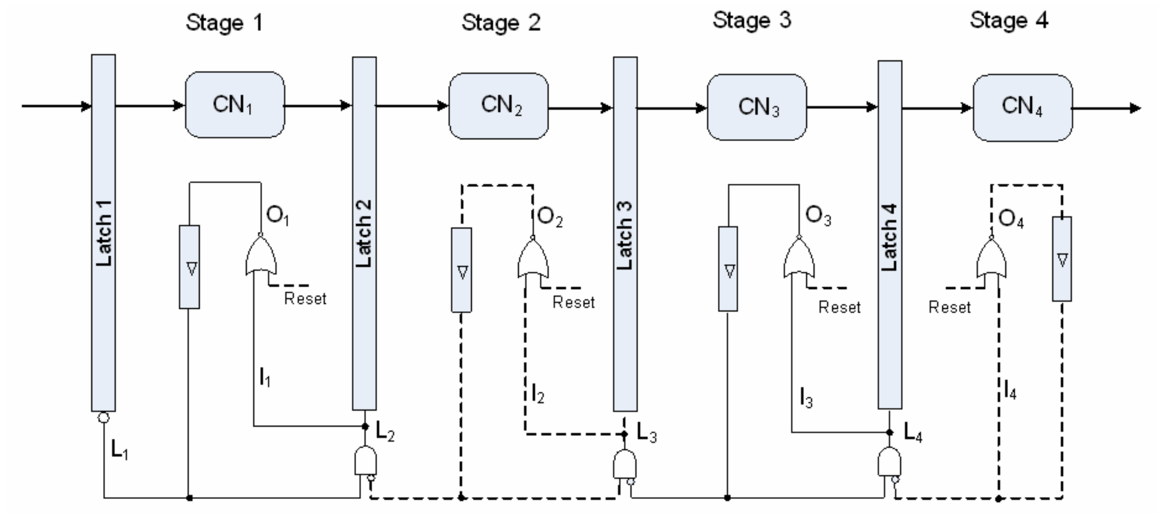

Figure 3.10: Evaluate phase of stage 1 and 3.

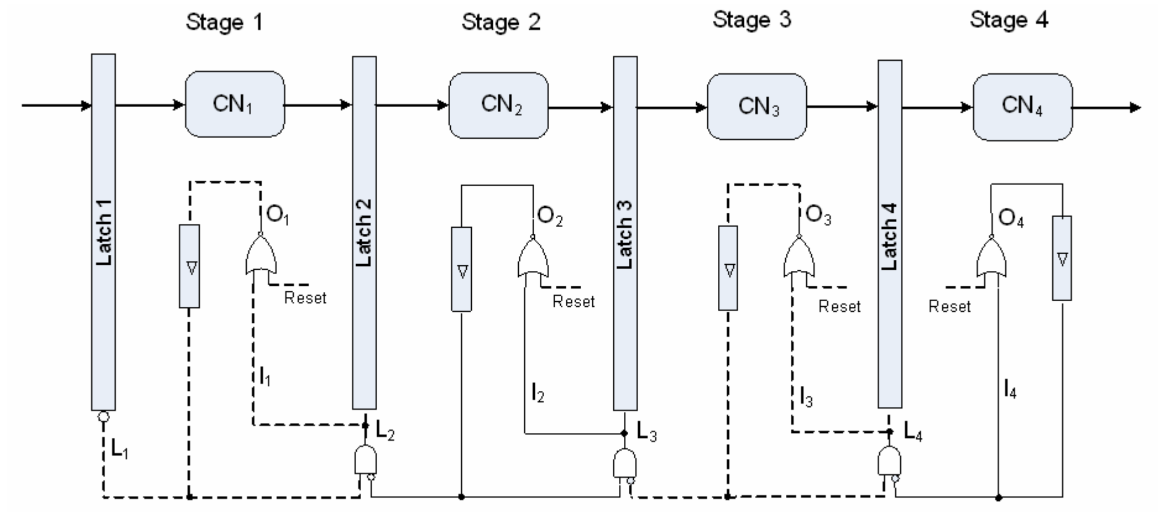

Figure 3.11: Evaluate phase of stage 2 and 4.

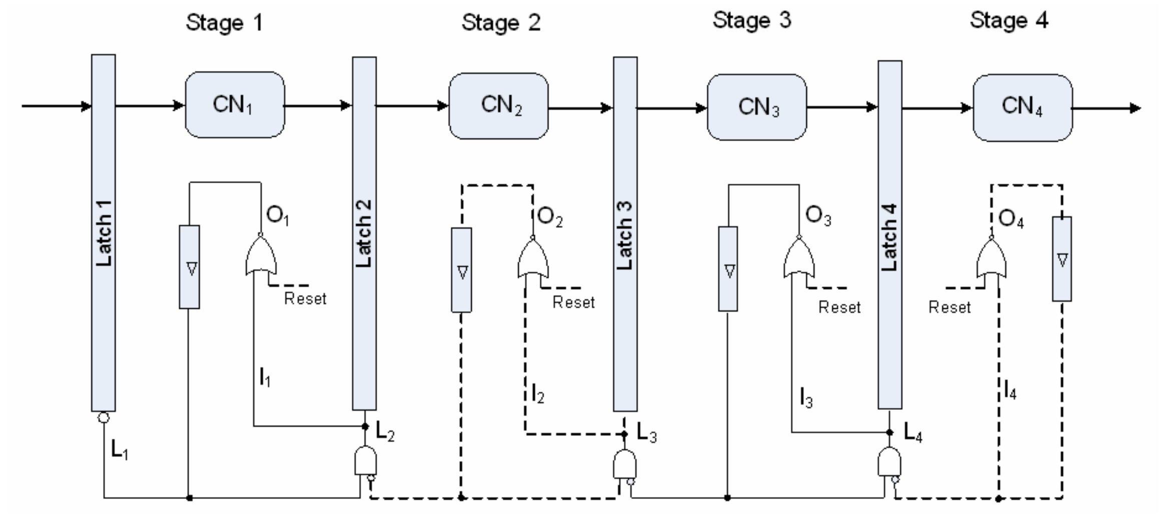

Figure 3.12: Evaluate phase of stage 1 and 3.

3.3 Pipeline-Controlled Self-Resetting Stage Logic

In pipeline-controlled self-resetting stage logic (P-SRSL), synchronization occurs between the last stage and any other stage in the pipeline. Similarly to S-SRSL, each stage is ready to absorb inputs when it enters the reset phase and produce an output when it enters its evaluate phase. As a result, data transfer occurs between two neighboring stages when the left stage is in the evaluate phase while the right stage is in the reset phase. Figure 3.13 shows the interconnection structure of a four-stage S-SRSL pipeline where each stage contains a combinational and a reset network. In addition to the two networks, each pair of stages is separated by a latch whose enable port is tied to the output of an AND gate. This AND gates has two inputs where the first is tied to the phase line of the last stage while the second is tied to the phase line of the stage located on the left side of the latch. Note that the right input of the AND gate is inverted in some while it is not in others.

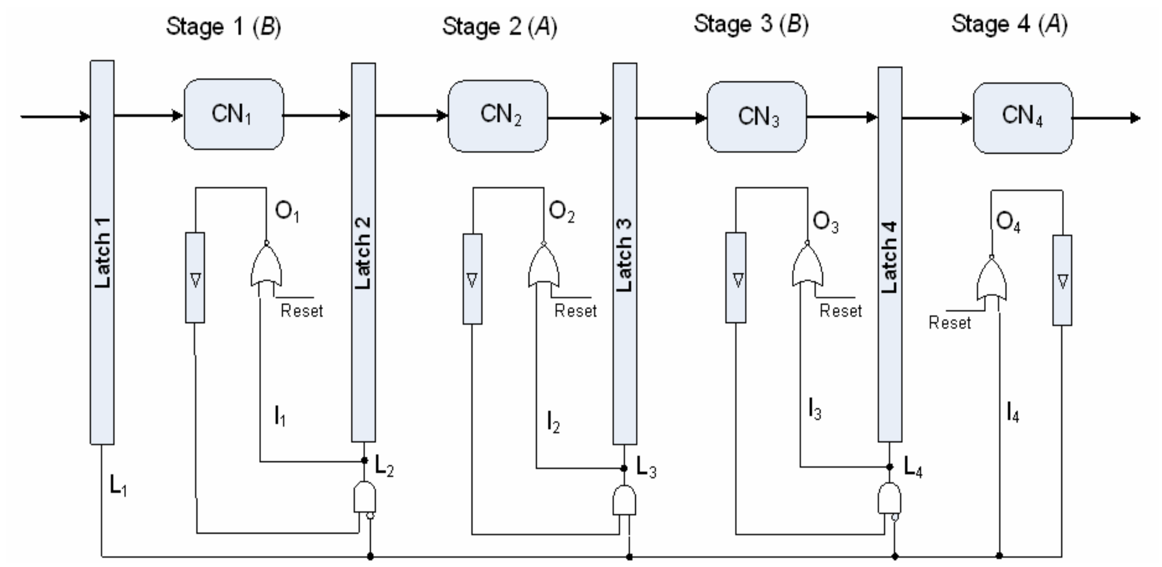

Figure 3.13: Four-stage P-SRSL pipeline.

Definition 3.1: 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 3.2: 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.

Based on these two definitions, stage 1 and 3 are of type *B* while stage 2 and 4 are of type *A* in Figure 3.13. 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 the S-SRSL pipeline, the P-SRSL pipeline requires that all stages in the pipeline have equal cycles to guarantee correct data flow. The STG for the four-stage P-SRSL pipeline shown in Figure 3.13 is shown in Figure 3.14. 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 asserted and de-asserted signals as solid and dashed lines respectively.

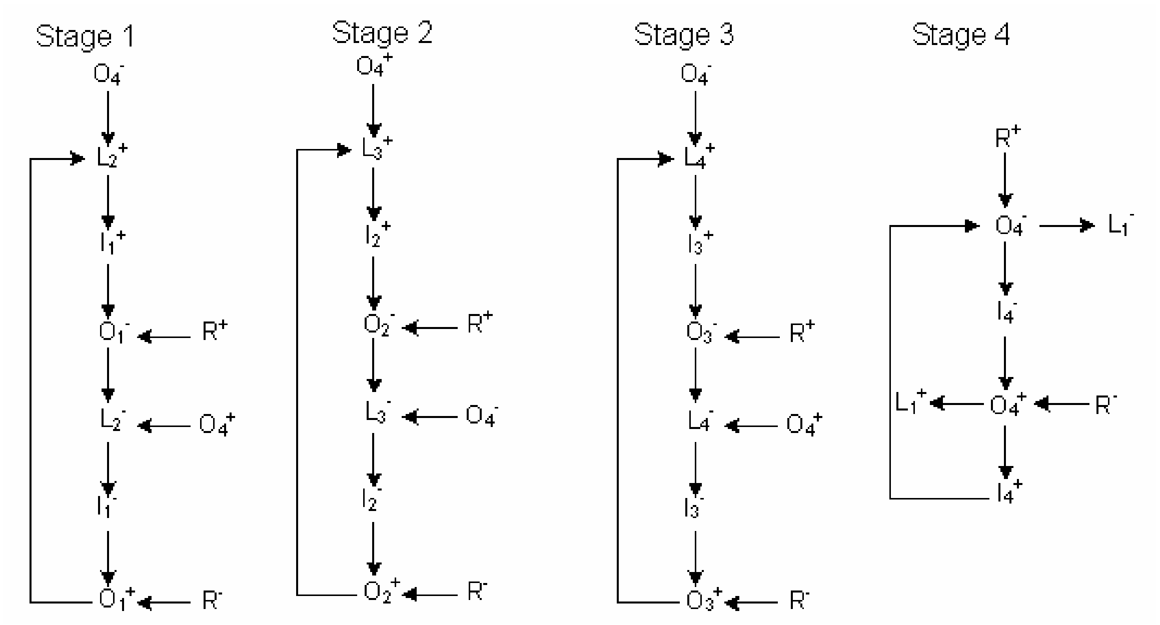

Figure 3.14: STG of the four-stage P-SRSL pipeline shown in Figure 3.13.

Figure 3.15 through Figure 3.20 show how the stages alternate between phases as data flows across the pipeline by depicting two execution cycles of a four-stage P-SRSL pipeline. The asserted and de-asserted signals are represented as solid and dashed lines respectively.

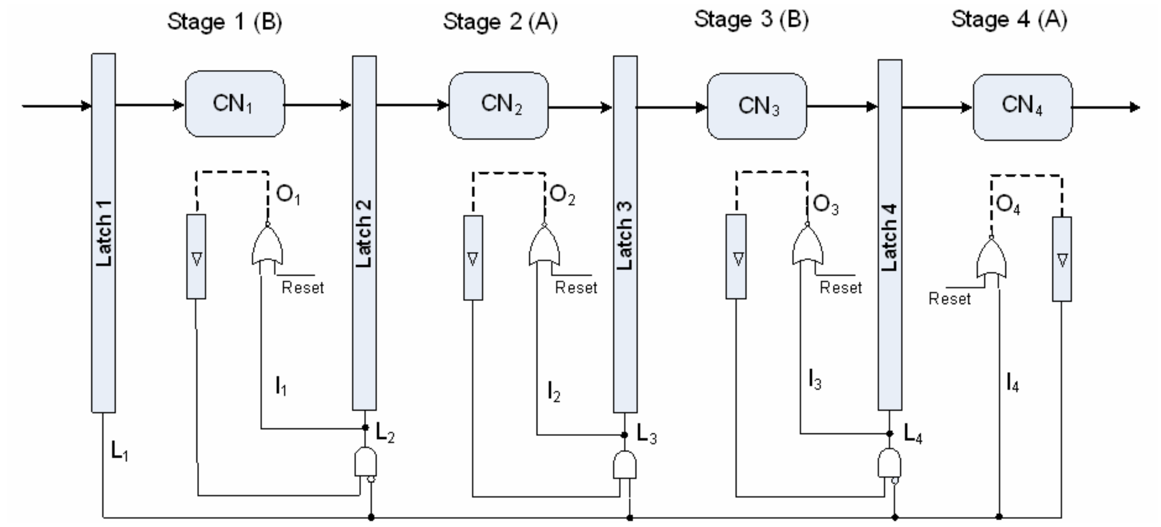

Figure 3.15: Assertion of the stage reset signals.

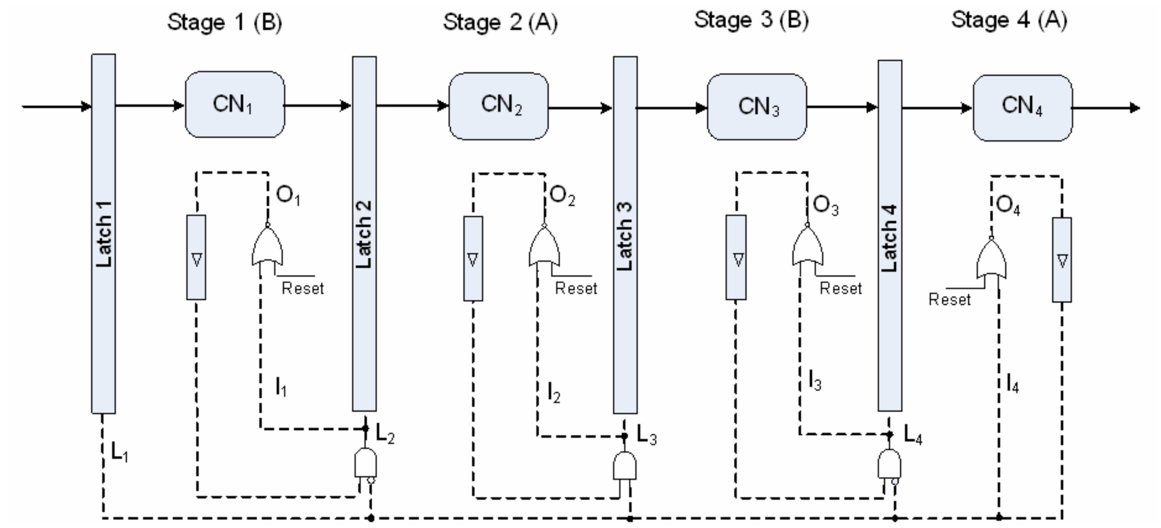

Figure 3.16: Reset phase of all stages.

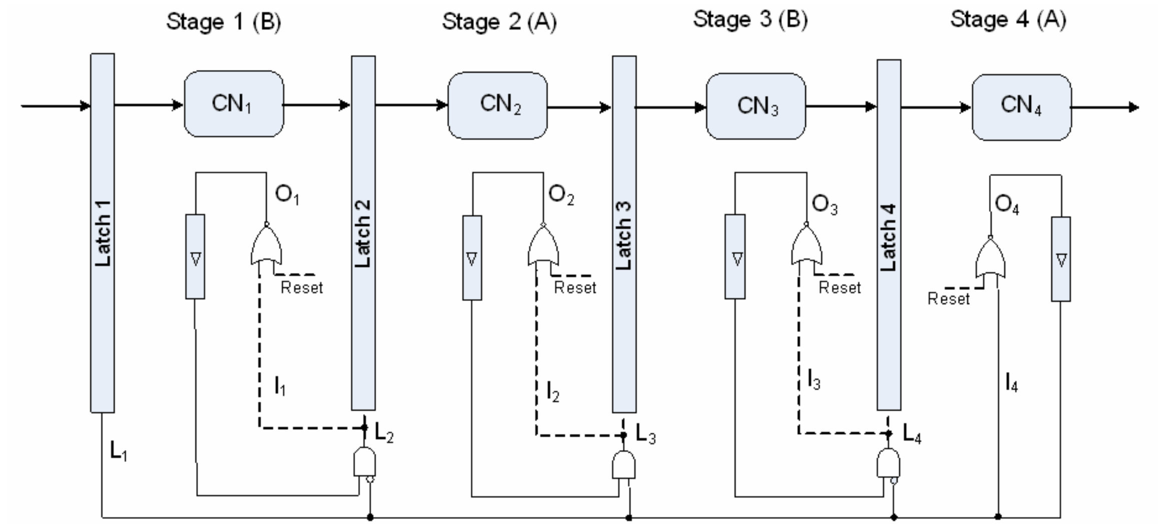

Figure 3.17: Evaluate phase of all stages.

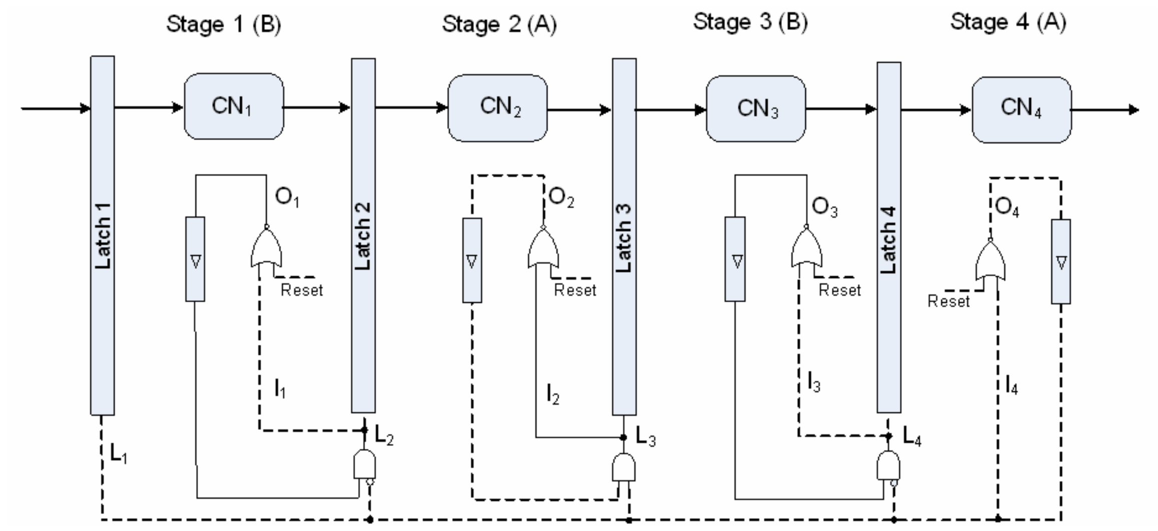

Figure 3.18: Evaluate phase of stage 3 and 1.

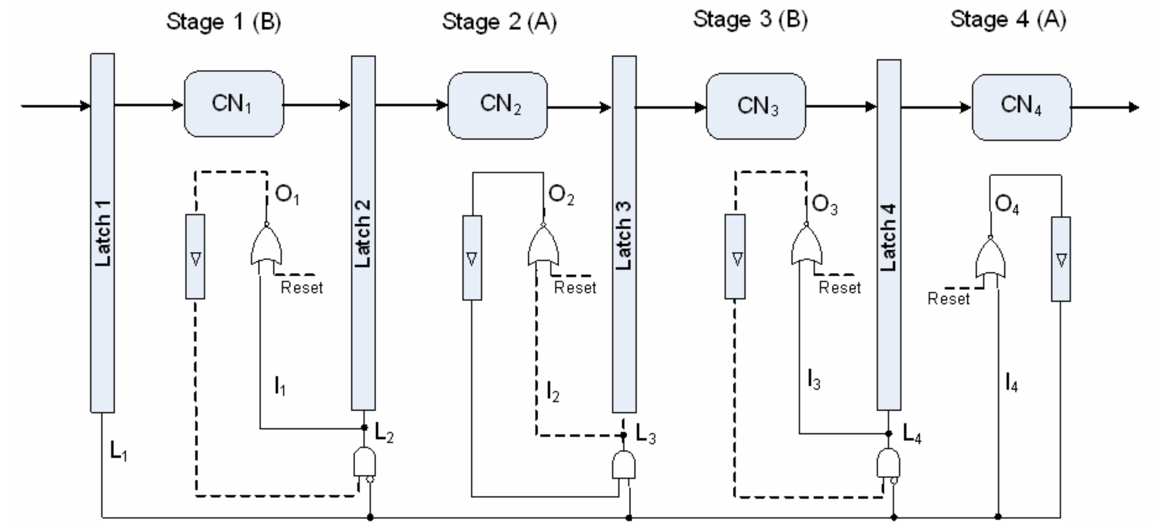

Figure 3.19: Evaluate phase of stage 4 and 2.

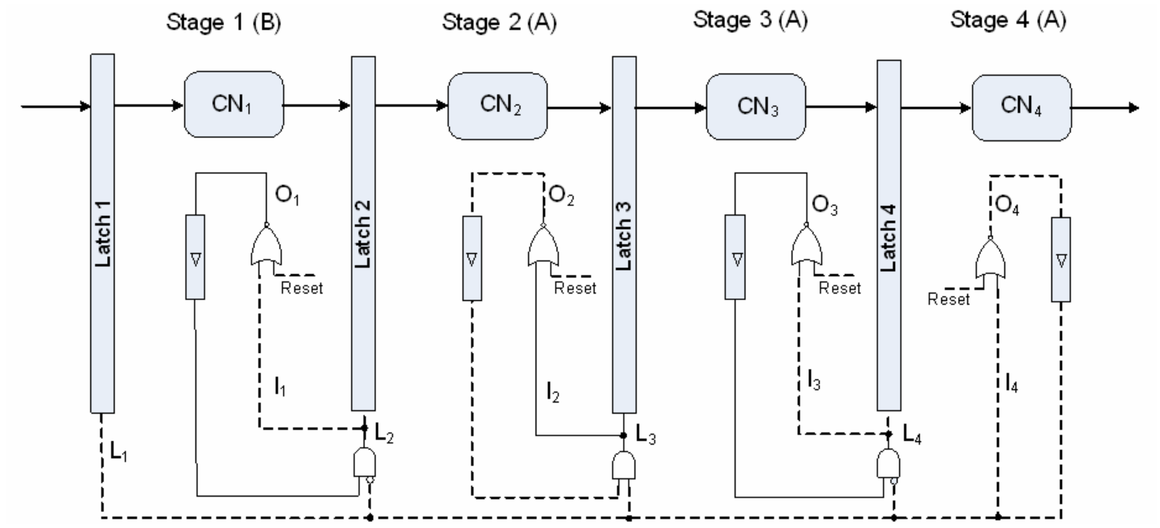

Figure 3.20: Evaluate phase of Stage 1 and 3.

3.4 Delay-Tolerant Self-Resetting Stage Logic

Whereas correct operation rests on stage having equal cycles in the S-SRSL and P-SRSL pipelines, this requirement is completely unnecessary in delay-tolerant self-resetting stage logic (D-SRSL) pipelines. In fact, stages can have arbitrary delays without affecting the correct data flow across the pipeline. Hence, the delay-tolerant property of these pipelines as implied by their name. In this approach, stages communicate with each other through their respective phases. Figure 3.21 shows a four-stage D-SRSL pipeline.

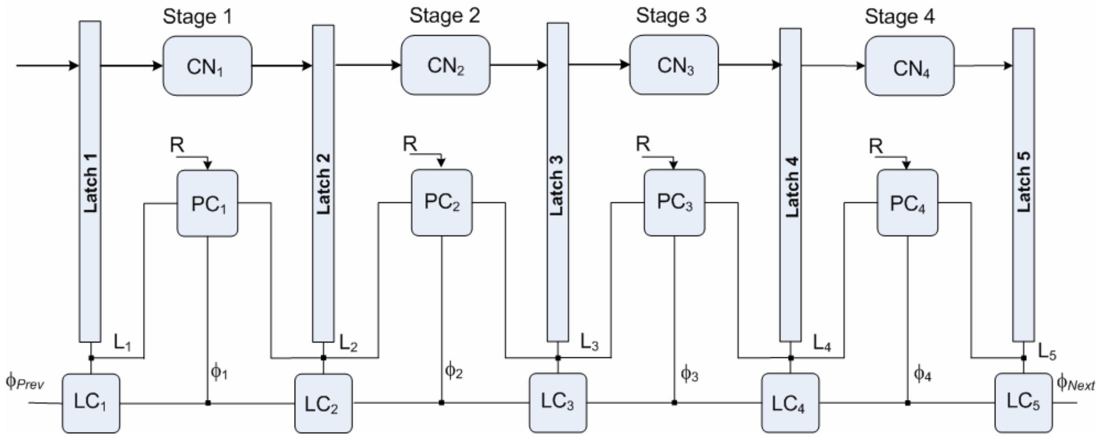

Figure 3.21: Four-stage D-SRSL pipeline.

3.4.1 Pipeline Structure

In D-SRSL pipelines, the latches are controlled by a *latch control* (LC) block. The phase oscillation of each stage is indicated by the signal ϕ as shown in Figure 3.21. A stage is ready to take in new inputs when it is in the reset phase while it produces outputs when it is the evaluate phase. The evaluation of the incoming inputs is performed by 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. Similarly to S-SRSL and P-SRSL pipelines, data flows from one stage to another when the first stage is in the evaluate phase while the second stage is in the reset phase.

Figure 3.22 shows the STG of the D-SRSL linear pipeline shown in Figure 3.21. Although the (*Clr)* signal in Figure 3.22 is not shown in Figure 3.21, its function within the LC block is described in the coming few paragraphs. The STG shows that the rising transition of L_3 occurs after both ϕ_2 and ϕ_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 ϕ_3 experiences a rising transition.

$$\begin{tikzcd}
 & & L_2^- \arrow[d] & \mathrm{Clr}^+ \arrow[d] & L_1^+ \arrow[l] & & & & & & \\
\phi_{Prev}^+ \arrow[r] & L_1^+ \arrow[r] & \phi_1^+ \arrow[r] \arrow[d] & L_1^- & L_3^- \arrow[d] & \mathrm{Clr}^+ \arrow[d] & L_2^+ \arrow[l] & & & & \\
 & \phi_1^- \arrow[u] & L_2^+ \arrow[l] \arrow[rr] & & \phi_2^+ \arrow[r] \arrow[d] & L_2^- & L_4^- \arrow[d] & \mathrm{Clr}^+ \arrow[d] & L_3^+ \arrow[l] & & \\
 & L_1^- \arrow[u] & \phi_2^- \arrow[u] & & L_3^+ \arrow[ll] \arrow[rr] & & \phi_3^+ \arrow[r] \arrow[d] & L_3^- & L_5^- \arrow[d] & \mathrm{Clr}^+ \arrow[d] & L_4^+ \arrow[l] \\
 & & L_2^- \arrow[u] & & \phi_3^- \arrow[u] & & L_4^+ \arrow[ll] \arrow[rr] & & \phi_4^+ \arrow[r] \arrow[d] & L_4^- & \\
 & & & & L_3^- \arrow[u] & & \phi_4^- \arrow[u] & & L_5^+ \arrow[ll] & & \\
 & & & & & & L_4^- \arrow[u] & & \phi_{Next}^- \arrow[u] & &
\end{tikzcd}$$

Figure 3.22: STG of the D-SRSL pipeline.

3.4.2 Phase Control Block

Figure 3.23 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 turn, it produces an output, ϕ_i, which is the phase signal of stage *i*.

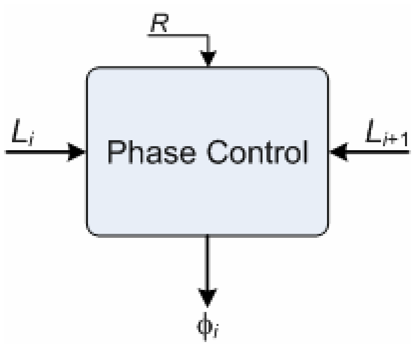

Figure 3.23: PC block.

To illustrate the behavior of the PC block, Figure 3.24 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 3.24, the PC block enters the reset state after the reset signal is de-asserted. In this state, ϕ_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 ϕ_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} become both asserted, in which case the PC block returns to the reset state. As ϕ_i switches back and forth, a stage can oscillate between a reset and evaluate phase in a single execution cycle. Given this oscillation, a stage is ready to absorb inputs when it is in the reset phase.

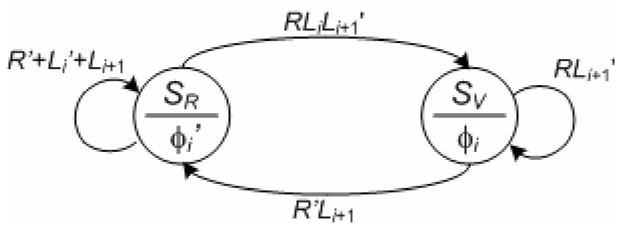

Figure 3.24: State graph of the PC block.

While the inputs are traveling along the critical path of the CN, ϕ_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 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.

3.4.3 Latch Control Block

Figure 3.25 shows the block diagram of the LC block. This block has three inputs, ϕ_i and ϕ_{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.

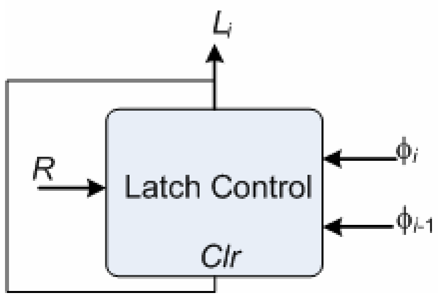

Figure 3.25: LC block.

To show the behavior of the LC block, Figure 3.26 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 ϕ_{i-1} is de-asserted while ϕ_i is asserted, the block remains in the disabled state. The LC block waits until ϕ_{i-1} gets asserted while ϕ_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 LC block returns automatically to the disabled state, thus disabling the latch.

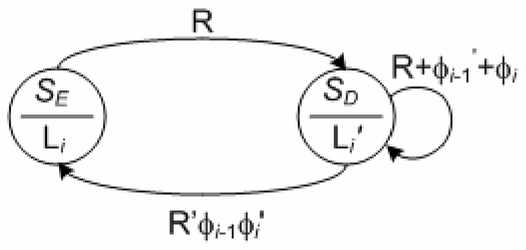

Figure 3.26: State graph of the LC block.

3.5 Summary

This chapter introduces SRSL and shows how this technique can be used to establish handshake signaling between two specific stages in a clockless pipeline. SRSL is used as a foundation to propose three different pipelining techniques: (i) S-SRSL pipelines in which synchronization takes place between two neighboring stages using a fine grain approach, (ii) P-SRSL pipelines in which the oscillation of the pipeline stages are driven by the oscillation of the last stage in the pipeline, and (iii) D-SRSL pipelines where synchronization occurs between each pair of neighboring stages using a coarse-grain approach. While S-SRSL and P-SRSL pipelines require that the stages display equal cycles in the pipeline, D-SRSL can handle individual pipeline stages with arbitrary delays. Although this chapter presents only the linear variants of these three pipelines, the examination of their non-linear variants and the presentation of a detailed timing analysis for each pipeline is presented in [16].

CHAPTER FOUR: SYNTHESIS OF SRSL PIPELINES

This chapter presents a specific SRSL design methodology in section 4.1 while section 4.2 presents the synthesis of SRSL pipelines. This design methodology has been formulated in [16]. Section 4.3 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 4.4 while section 4.5 explains the proposed heuristic solution.

4.1 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 [16]. Figure 4.1 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.

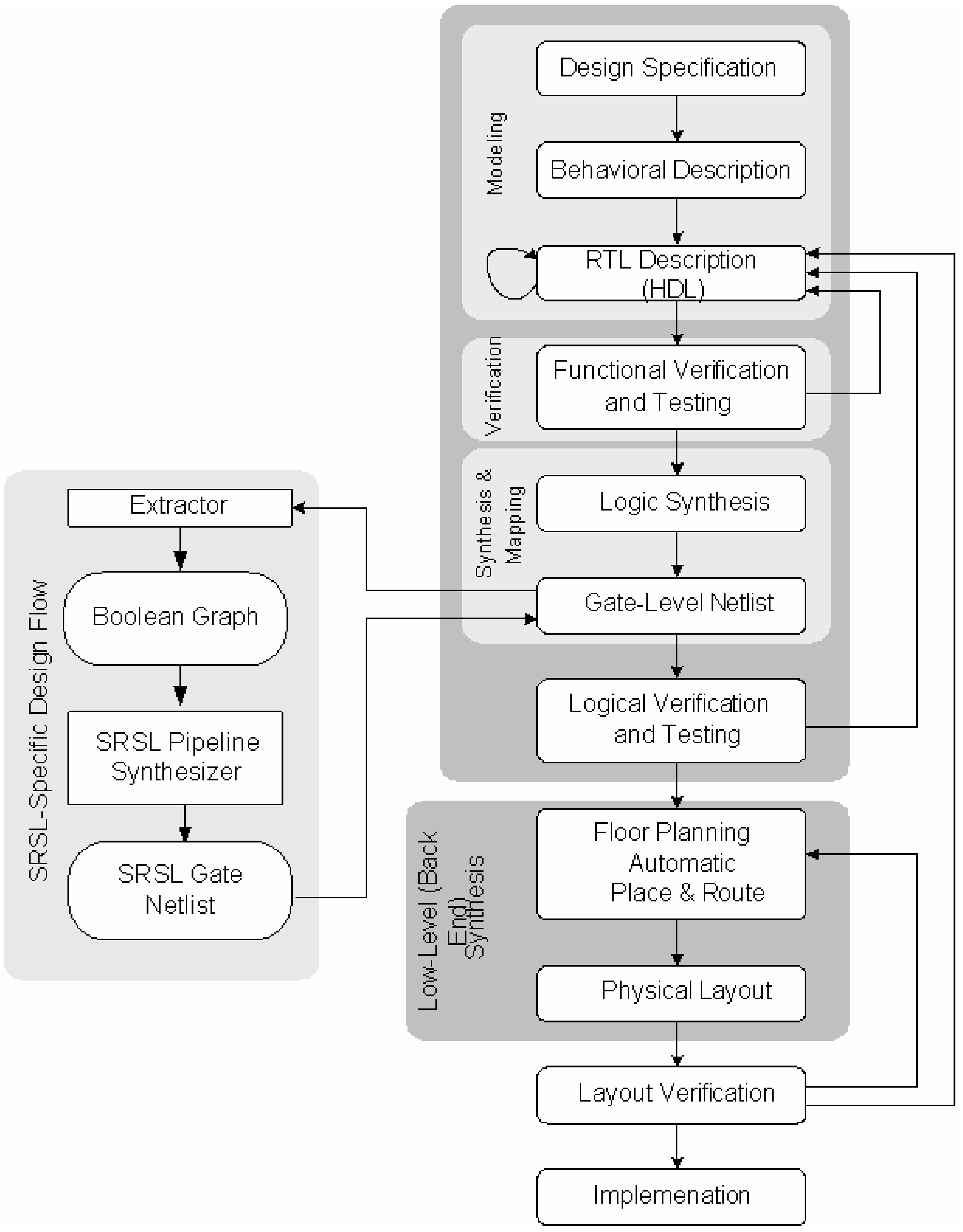

Figure 4.1: SRSL design flow.

4.2 Synthesis of SRSL Pipelines

The synthesis of SRSL pipelines consist of transforming a clocked gate netlist into an SRSL 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 SRSL 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 SRSL pipeline while the latter refers to a specified lower limit on data rates that can be achieved by converting a gate netlist into an SRSL pipeline.

To transform a gate netlist into an SRSL pipeline, three problems emerge:

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

4.3 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 4.1: An *incidence structure* consists of a set of modules, a set of nets, and an incidence relation among modules and nets [28].

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 4.2: 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 [28].

Figure 4.2 shows a Boolean network with 10 primary inputs, 10 modules, and four primary outputs [28].

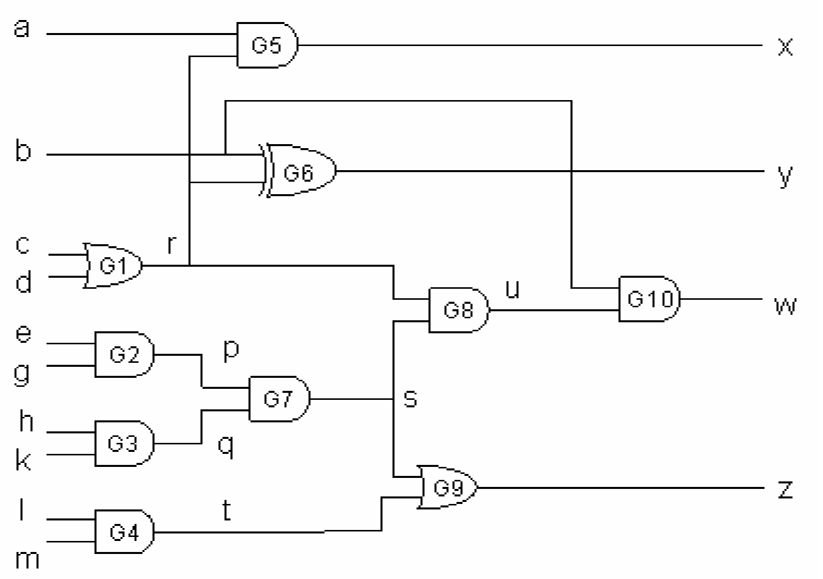

Figure 4.2: Example of a Boolean network.

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

Definition 4.3: 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 thesis, 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 4.4: 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 4.5: A *path* from vertex v to w in a graph $G(V, E)$ is a sequence of edges v_0v_1, v_1v_2, …, $v_{k-1}v_k$, 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: $\pi = (v_0, v_1, v_2, \ldots, v_k)$. In directed graphs, paths follow the direction of the edges.

Definition 4.6: A *cycle* in a directed graph is a nonempty path such that the first vertex and the last vertex are identical.

Definition 4.7: A graph is *acyclic* if it has no cycles.

Definition 4.8: 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 4.3 shows a Boolean graph based on the Boolean network of Figure 4.2. Note that the Boolean graph is acyclic since the nets induce a partial order on the modules.

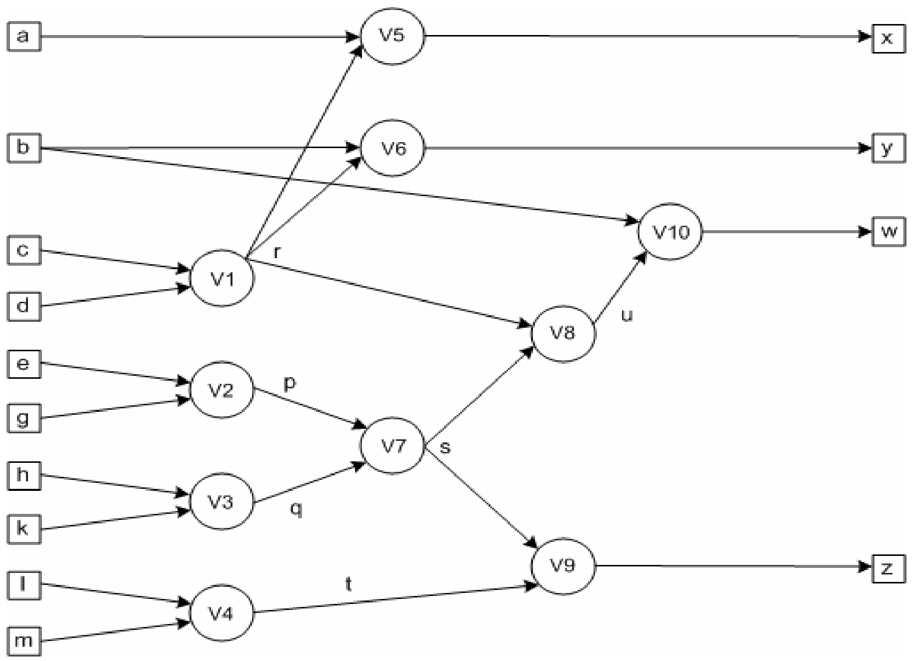

Figure 4.3: Boolean graph of the Boolean network shown in Figure 4.2.

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 is a mapped Boolean network. Before it is transformed into an SRSL pipeline, it is translated into a Boolean graph.

4.4 Analytical Formulation of the Pipelining 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 SRSL pipeline is equivalent to partitioning the Boolean graph of the gate netlist into partitions and assigning each partition to a distinct pipeline stage. Let $S = \{s_1, s_2, \ldots, s_{|S|}\}$ be the set of pipeline stages where the size of this set, $|S|$, is some positive integer. Let $V = \{v_i \,;\, i = 1, 2, \ldots, |V|\}$ and $E = \{(v_i, v_j) \,;\, i, j = 1, 2, \ldots, |E|\}$.

Definition 4.9: 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 $\varphi(v_i) \le \varphi(v_j), \forall (v_i, v_j) \in E$.

Since each vertex in V has a delay, $D = \{d_i;\, i = 1, 2, \ldots, |V|\}$. 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; in fact, 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 4.10: 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 4.11: Let Π be the set of all paths in a Boolean graph $G(V, E)$. A *critical path* in G is a path π whose delay is the largest path delay in Π, 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 π in the Boolean graph G whose delay is d_π. An upper bound on the number of stages in the pipeline, called *maximum pipeline depth*, can be obtained from P and d_π. 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 an alternative upper bound if the synthesized pipeline is an S-SRSL pipeline as discussed in [16]. In this case, $|S| = \min\left\{ \left\lceil \frac{d_\pi}{P} \right\rceil, \left\lfloor 1 + \frac{1}{\delta}\left(\frac{P}{2} - d\left(L_1^+\right)\right) \right\rfloor \right\}$ based on [16].

Definition 4.12: 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*:

$$\sum_{s=1}^{|S|} x_{i,s} = 1, \quad i = 1, 2, ..., |V| \qquad (4.1)$$

There are $|V|$ such constraints in the problem. It also imperative to observe the condition stated in Definition 4.9, 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*:

$$\sum_{s=1}^{|S|} sx_{i,s} \le \sum_{s=1}^{|S|} sx_{j,s}, \qquad \forall \left(v_i, v_j\right) \in E \qquad (4.2)$$

These constraints can be rewritten as:

$$\sum_{s=1}^{|S|} sx_{j,s} - \sum_{s=1}^{|S|} sx_{i,s} \ge 0, \qquad \forall \left(v_i, v_j\right) \in E \qquad (4.3)$$

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:*

$$\sum_{i:v_i \in \pi} d_i x_{i,s} \le p, \;\; s = 1, 2, ..., |S| \qquad (4.4)$$

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 (4.4) 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., $P = P_i, \; i = 1, 2, ..., \; |S|$. Balancing a pipeline is relevant only when synthesizing S-SRSL and P-SRSL pipelines. 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 (4.2):

(i) $k = l$: This means that both stages represent the same stage. In this case, v_i and v_j are assigned to the same stage.

(ii) $k \neq l$: 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 4.13: 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:

$$\delta_{i,j} = \sum_{s=1}^{|S|} sx_{j,s} - \sum_{s=1}^{|S|} sx_{i,s}, \qquad (v_i, v_j) \in E \qquad (4.5)$$

If applied to a single edge, (4.5) is similar to the left-hand side of (4.3). 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:

$$\sum_{\forall (v_i, v_j) \in E} a_l \left(\sum_{s=1}^{|S|} sx_{j,s} - \sum_{s=1}^{|S|} sx_{i,s} \right) \qquad (4.6)$$

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 SRSL 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, $A = \{a_i;\ i = 1, 2, \ldots, |V|\}$. If the area of the matching delay of a stage s is a_s, then:

$$a_s = \sum_{i:v_i \in \pi} a_i x_{i,s}, \quad s = 1, 2, ..., |S| \tag{4.7}$$

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

$$\sum_{s=1}^{|S|} \left(\sum_{i:v_i \in \pi} a_i x_{i,s} \right) \tag{4.8}$$

By summing the total area needed for latches shown in (4.6), and matching delays shown in (4.8), the minimization of the area cost can be expressed as the following objective function:

$$\min a_l \sum_{\forall (v_i, v_j) \in E} \left(\sum_{s=1}^{|S|} s x_{j,s} - \sum_{s=1}^{|S|} s x_{i,s} \right) + \sum_{s=1}^{|S|} \left(\sum_{i:v_i \in \pi} a_i x_{i,s} \right) \tag{4.9}$$

In summary, P1 can be formulated as the following *integer programming* (IP) problem [29]:

$$\min a_l \sum_{\forall (v_i, v_j) \in E} \left(\sum_{s=1}^{|S|} s x_{j,s} - \sum_{s=1}^{|S|} s x_{i,s} \right) + \sum_{s=1}^{|S|} \left(\sum_{i:v_i \in \pi} a_i x_{i,s} \right) \tag{4.9}$$

$$\sum_{s=1}^{|S|} x_{i,s} = 1, \quad i = 1, 2, ..., |V| \tag{4.1}$$

$$\sum_{s=1}^{|S|} s x_{j,s} - \sum_{s=1}^{|S|} s x_{i,s} \geq 0, \quad \forall (v_i, v_j) \in E \tag{4.3}$$

$$\sum_{i:v_i \in \pi} d_i x_{i,s} \leq p, \ s = 1, 2, ..., |S| \tag{4.4}$$

<u>**4.5 Heuristic SRSL Pipelining**</u>

Although it is possible to solve P1 using standard combinatorial approaches suitable for general IPs, using such methods may not be efficient due to the size of P1's IP formulation in some cases [29, 30]. For example the IP formulation of C6822 circuit, which is an ISCAS benchmark circuit consisting of 6,656 gates and a 245-gate long critical path, can generate 6,656 constraints (4.1), 9,082 constraints (4.2), and 245 constraints (4.3). In total, the matrix of the IP has 15,983 rows and 245 columns. As a result, a two-phase efficient heuristic solution is proposed to obtain the solution of P1 instead. The first phase is a stage-assignment algorithm which assigns each gate to a single stage by partitioning the Boolean graph of the gate netlist into subgraphs that meet specific timing constraints. On the other hand, the second phase is a vertex-shuffling algorithm which minimizes the area occupied by inter-stage latches through the shuffling of nearby vertices from the Boolean graph between adjacent stages without violating timing constraints.

4.5.1 Stage Assignment Phase

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.

4.5.1.1 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 δ 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 δ edges to accommodate the insertion of δ 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 δ is computed and δ 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:

4.5.1.2 Phase I Algorithm

The pseudocode of the graph partitioning algorithm is as follows:

```
Input:  $G(V, E)$
        $D = \{d_i ; i = 1, 2, …, |V|\}$
        $A = \{a_i ; i = 1, 2, …, |V|\}$
        $f$
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 or previous 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
```

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 or previous partition. This heuristic rule is based on the *as-soon-as-possible* scheduling strategy [31]. 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 is 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.

4.5.2 Vertex Shuffling Phase

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. Finally, a step-by-step of the trace of the proposed algorithm on a sample partitioned graph is presented for purpose of illustration.

4.5.2.1 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 4.4 shows two adjacent partitions where the left partition contains vertices labeled 1 through 10 while the right partition contains vertices labeled 11 through 17.

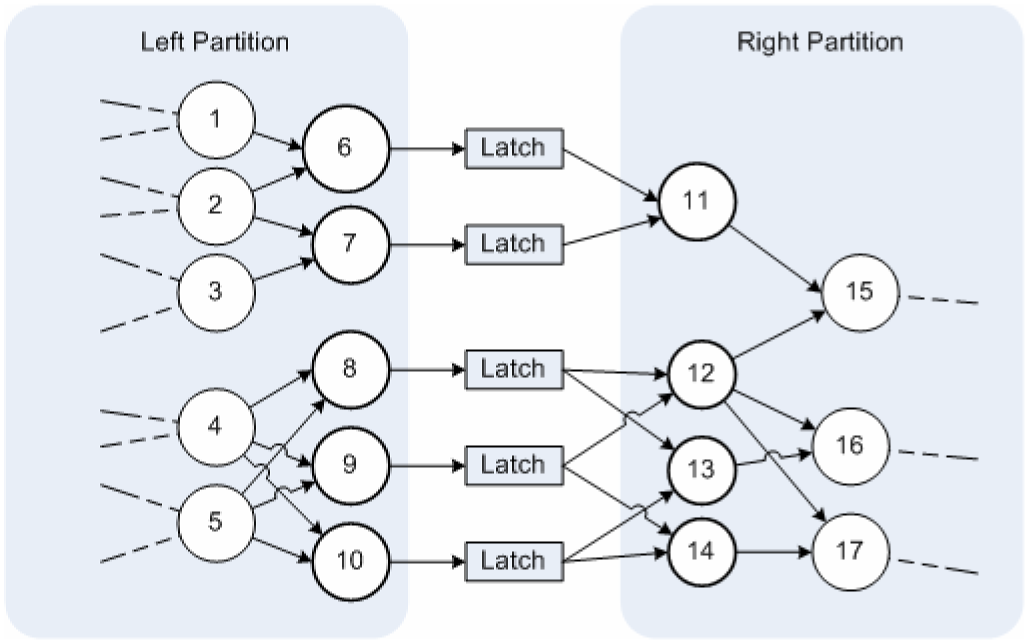

Figure 4.4: Latch placement two Boolean graph partitions.

Definition 4.14: 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 4.4 are all left cut vertices.

Definition 4.15: 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 4.4 are all left cut vertices, they make up a left cut vertex set.

Definition 4.16: 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 4.4 are all right cut vertices.

Definition 4.17: 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 4.4 are all right cut vertices, they make up a right cut vertex set.

Definition 4.18: 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 4.4 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 4.19: 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 4.4 is a cut edge.

Definition 4.20: 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 4.4, 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 4.21: 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 4.4. 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 $C(v)$ denotes the set of these cut edges. Note that, depending on where v is located in $G'(V', E')$, it is possible that $I(v) = \varnothing$ or $C(v) = \varnothing$. The proposed vertex shuffling algorithm uses a gain function to guide how it shuffles cut vertices from one partition to another.

Definition 4.22: 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., $g(v) = |C(v)| - |I(v)|$.

Given that no matter how many edges are connected to the output of any vertex, only one latch is needed to latch its signal. Consequently, it is always the case that $C(v_i) = 1$ for the vertices in the left cut vertex set, while it is always the case that $I(v_i) = 1$, and for the vertices in the right cut vertex set. Based on this observation, vertex 6 in Figure 4.4 has two internal edges and one cut edge. Its gain is $g(v_6) = |C(v_6)| - |I(v_6)| = 1 - 2 = -1$. On the other hand, since vertex 12 has three internal that can be latched by only one latch and two cut edges, its gain is $g(v_{12}) = |C(v_{12})| - |I(v_{12})| = 2 - 1 = 1$.

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 and resembles closely the move function proposed in [32]. 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 4.23: 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., $\{u \mid u \in V'$ 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., $\{w \mid w \in V'$ and $(v, w) \in E'\} - C_R$). The set of these moves is called the *set of induced moves by v*.

In Figure 4.4, 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 4.24: 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 4.4, 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 4.25: 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 4.4 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 4.4 is equal to the size of the left vertex cut set.

4.5.2.2 Phase II Algorithm

The pseudocode of the vertex shuffling algorithm is as follows:

```
Input: $G'(V', E')$ SRSL pipelined graph that meets $p$
       $D = \{d_i ; i = 1, 2, …, |V|\}$
       $A = \{a_i ; i = 1, 2, …, |V|\}$

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
25. For each edge in E''
26.   Compute the pipeline distance $\delta$;
27.   Add $\delta$ vertices to V'';
28.   Add $\delta$ edges to E'';
29. Endfor
30. For each partition in V'';
31.    Get the critical path in the current partition;
32.    Duplicate the path and insert it into the current partition;
33. Endfor
34. The final obtained graph is G''(V'', E'');
```

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. After the partitioned graph $G''(V'', E'')$ is obtained, the next step consists of adding vertices between the partitions to represent latches between pipeline stages as shown in line 25 through 29. 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 30 through 33. 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 streamlined graph $G''(V'', E'')$ is obtained as shown in line 34.

CHAPTER FIVE: EXPERIMENTAL RESULTS

In this chapter, the experimental results of applying the two phase heuristic to synthesize P-SRSL and D-SRSL pipelines are presented. For the sake of brevity, S-SRSL pipelines were omitted for the experiments since they resemble closely P-SRSL pipelines. As a result, the results obtained from P-SRSL pipelines can be easily extended to S-SRSL pipelines. Section 5.1 describes the experimental setup to evaluate the synthesis algorithm while section 5.2 presents the results obtained from the conducted experiments and their interpretations. Section 5.3 concludes this chapter.

5.1. Experimental Setup

The computing resources used in the experimental setup consist of:

(i) Sun server: This server houses Synopsys software tools, which are used for synthesis and simulation of the synthesized SRSL pipelines.

(ii) Sun Workstation: A Sun-Blade 1000 is used to run the synthesis and simulation experiments by pulling netlist files from the server to the workstation. The simulation runs are performed to verify the functional correctness of the synthesized SRSL pipelines.

(iii) Dell Personal Computer (PC): This PC is used to transform a gate netlist into an SRSL pipeline by applying the two-phase heuristic algorithm. This algorithm has been implemented in a Java tool "*SRSL Synthesizer*" on this PC. The executable of this algorithm runs also on this PC.

The two-phase heuristic algorithm has been applied on a set of six circuits shown in Table 5.1. The benchmark circuits were selected to exhibit a variety of depth and breadth in order to see how these circuits characteristic affect pipeline performance.

Table 5.1: Experimental circuits.

Circuit	**Functionality**	**Gates**	**Critical Path Delay (ps)**
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 Table 5.1, column 1 shows the six circuits where the top three are borrowed from the ISCAS-85 benchmark suite [33]. Column 2 shows the functionality of each circuit while column 3 shows the number of gates in the netlist of each circuit. Column 4 shows the delay on the critical path of each circuit. The step-by-step detailed procedure of the SRSL Synthesizer's heuristic algorithm is shown in Figure 5.1 where the gate netlist is translated into a Boolean graph on which the two-phase heuristic is applied. The obtained graph is a partitioned graph in which each partition represents a pipeline stage. The partitions of the graphs are glued with SRSL components such as delay buffers and latches after which the graph is translated back into a gate netlist. This netlist serves to generate a synthesis report showing the throughput and area cost of the pipelined gate netlist.

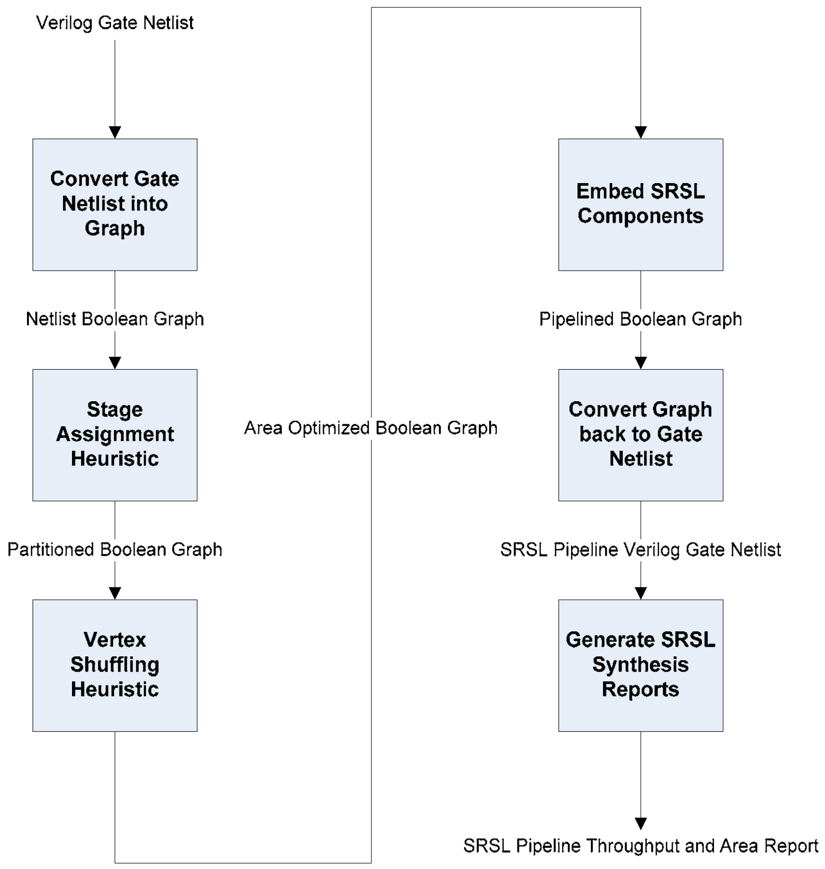
Verilog Gate Netlist
Convert Gate Netlist into Graph
Netlist Boolean Graph
Stage Assignment Heuristic
Partitioned Boolean Graph
Vertex Shuffling Heuristic
Area Optimized Boolean Graph
Embed SRSL Components
Pipelined Boolean Graph
Convert Graph back to Gate Netlist
SRSL Pipeline Verilog Gate Netlist
Generate SRSL Synthesis Reports
SRSL Pipeline Throughput and Area Report

Figure 5.1: Java SRSL Synthesizer's pipelining procedure of the benchmark circuits.

5.2. P-SRSL Experiments

In order to profile the performance of the P-SRSL pipelines, two sets of experiments were conducted to evaluate the impact of pipelining on area cost and throughput.

5.2.1. P-SRSL Area Cost

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 5.2 shows the P-SRSL area as a percentage of the overall pipeline circuit area including the P-SRSL area.

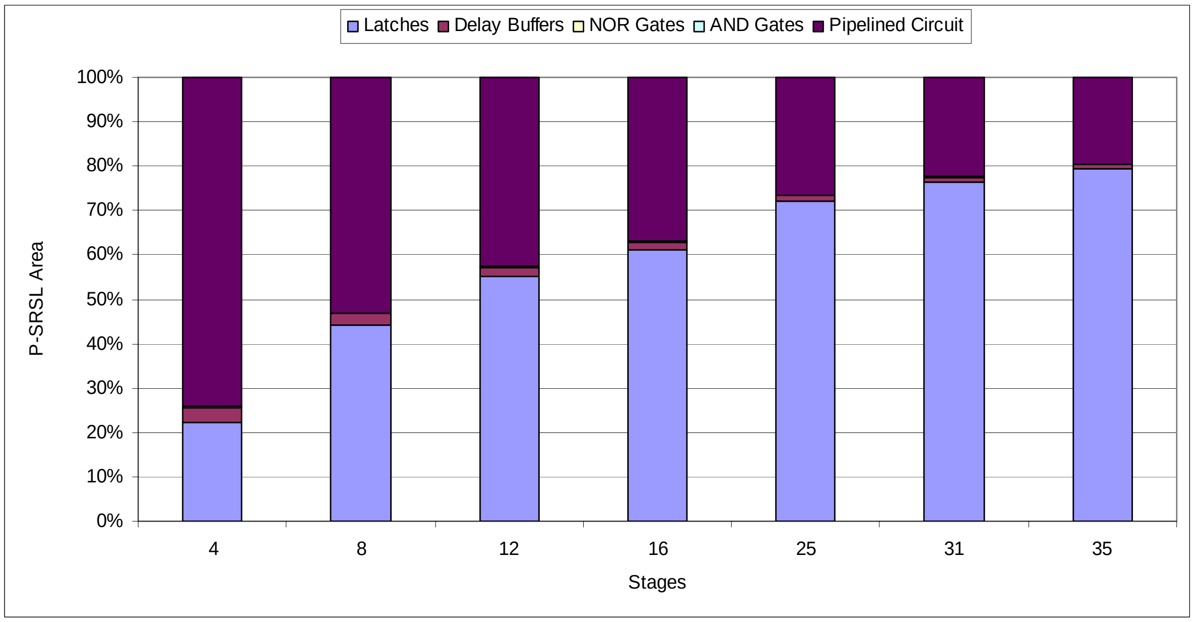

Figure 5.2: P-SRSL area as a percentage of the pipeline area across different pipelines of the C6822 benchmark circuit.

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. In fact, a basic latch in our standard cell library can take up to seven times the area of a two-input AND gate. Figure 5.3 shows the P-SRSL area as a percentage of the total area of a pipeline for each circuit across different pipeline depths.

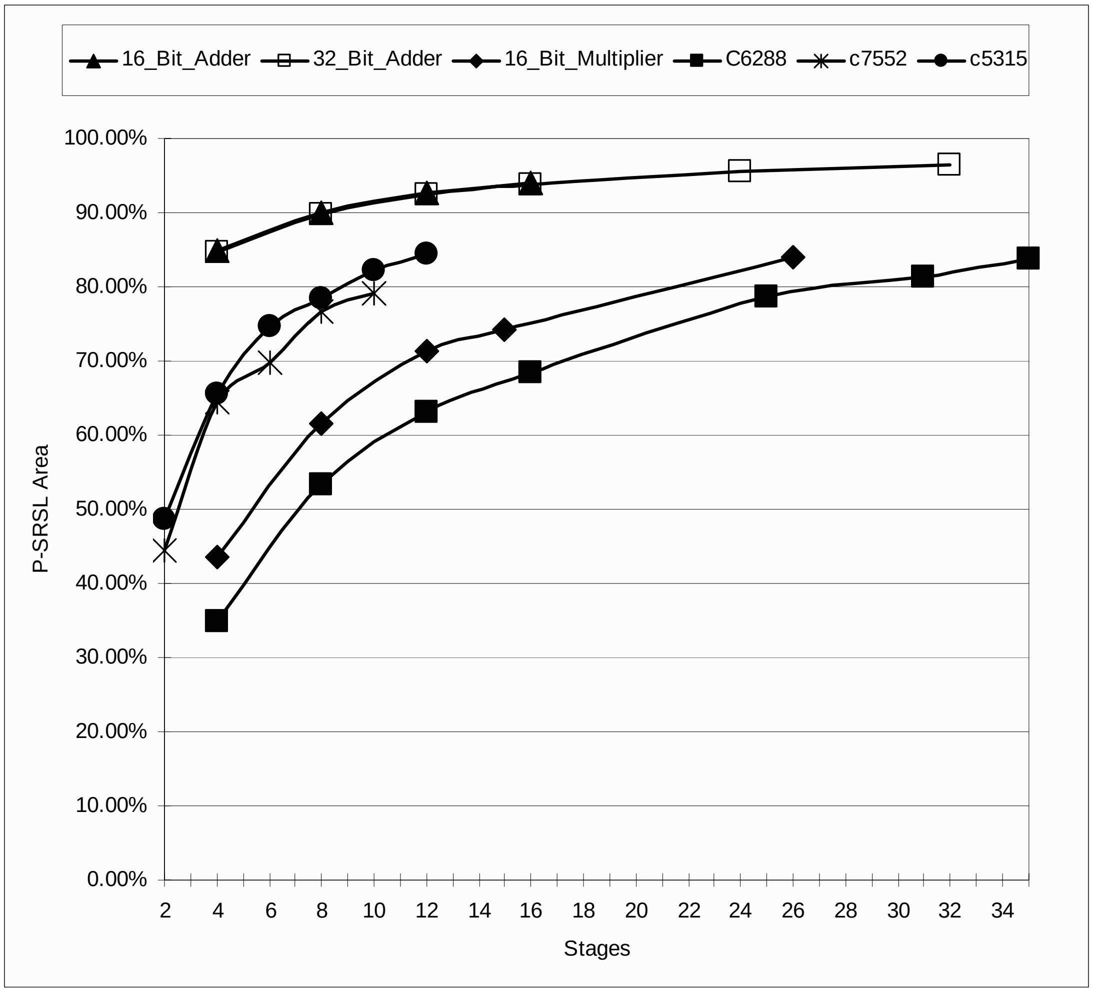

Figure 5.3: P-SRSL area as a percentage of the pipeline area across various depth pipelines.

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. 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.

5.2.2. P-SRSL Throughput

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 5.4. 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.

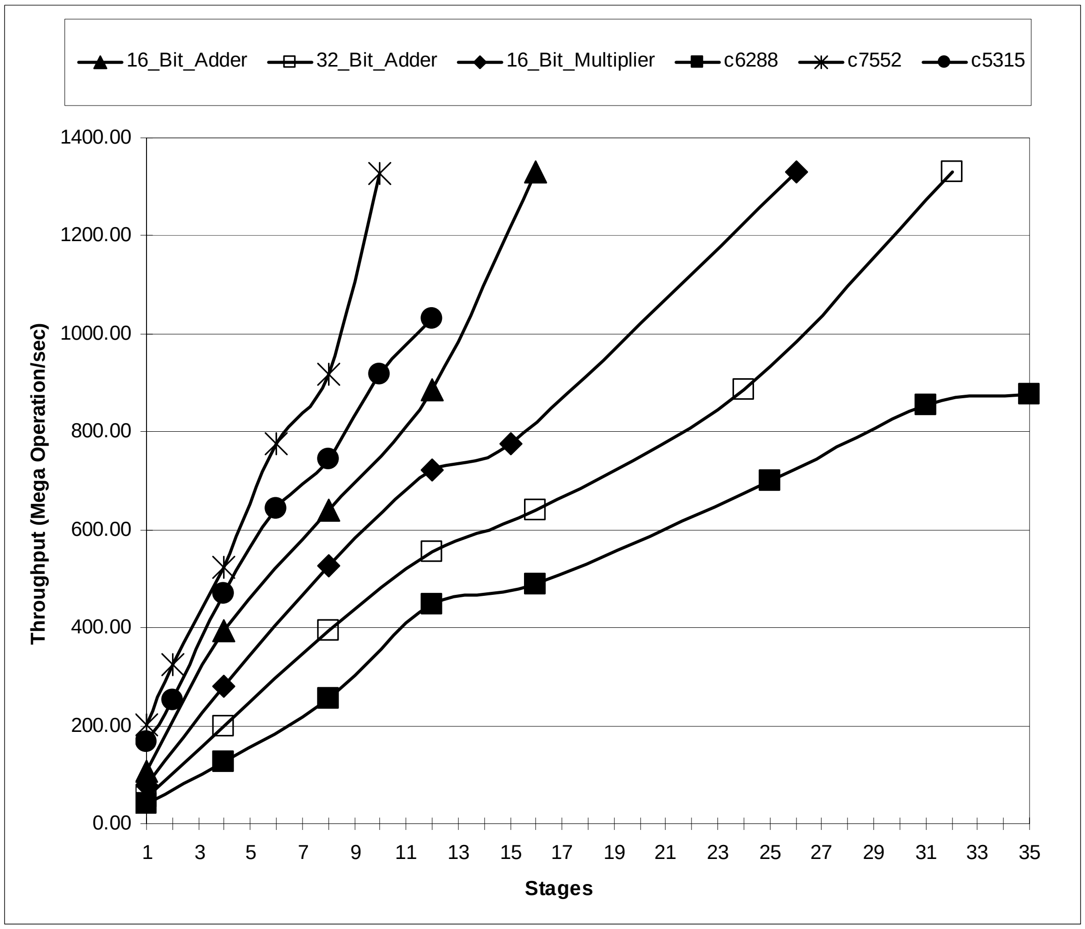

Figure 5.4: Pipeline throughputs for various P-SRSL pipeline depths.

In Figure 5.4, 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.

5.3. D-SRSL Experiments

The same two sets of experiments were conducted to evaluate the impact of pipelining on area cost and throughput in D-SRSL pipelines.

5.3.1. D-SRSL Area Cost

To study the cost of the additional area that is required to synchronize the D-SRSL pipeline, Circuit C5135 is chosen as an example. The D-SRSL area includes the area of the PC blocks, the LC blocks, inter-stage latches, and the intra-stage delay buffers.

Figure 5.5 shows the area percentage of each component that contributes to D-SRSL area.

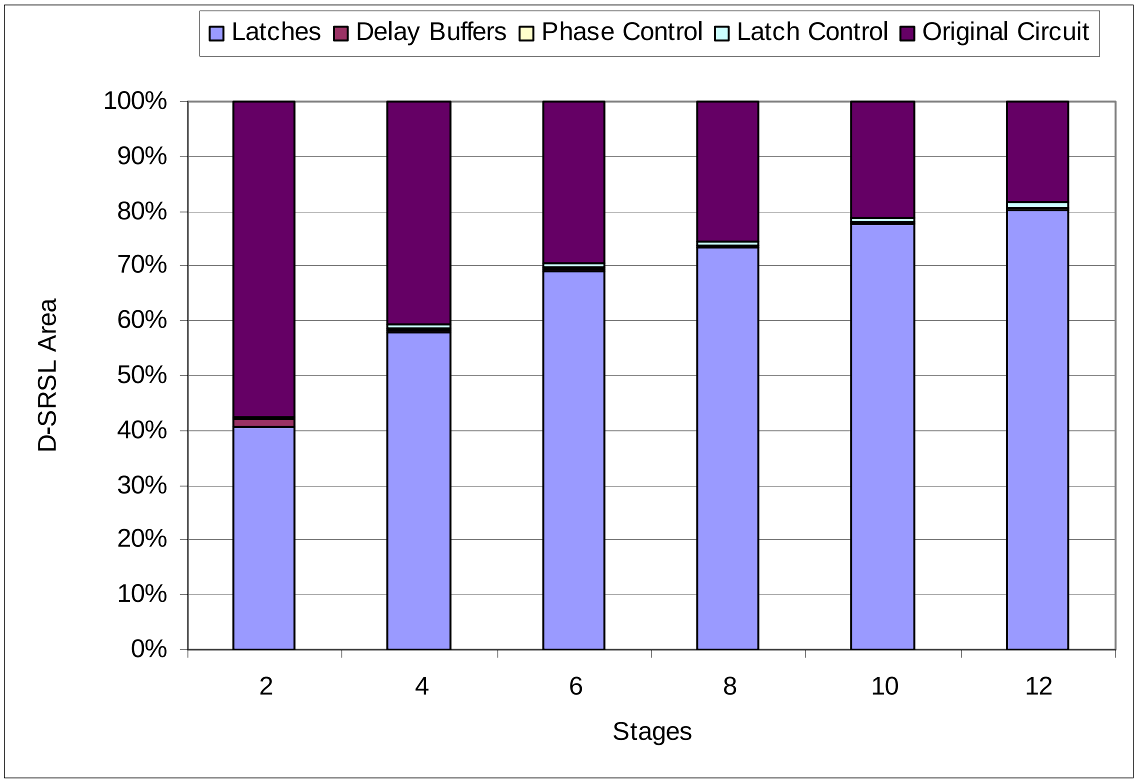

Figure 5.5: D-SRSL area as a percentage of the pipeline area across different pipelines of the C5135 benchmark circuit.

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. This shows that most of the area occupied is consumed by latches.

With regard to the area as a percentage of the total area of a pipeline for each circuit across different pipeline depths, the same observations made regarding the P-SRSL pipelines in Figure 5.3 are valid also for the D-SRSL pipelines.

5.3.2. D-SRSL Throughput

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 5.6. For each circuit, the pipeline depth is increased until the circuit throughput cannot be improved any more. In Figure 5.6, 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.

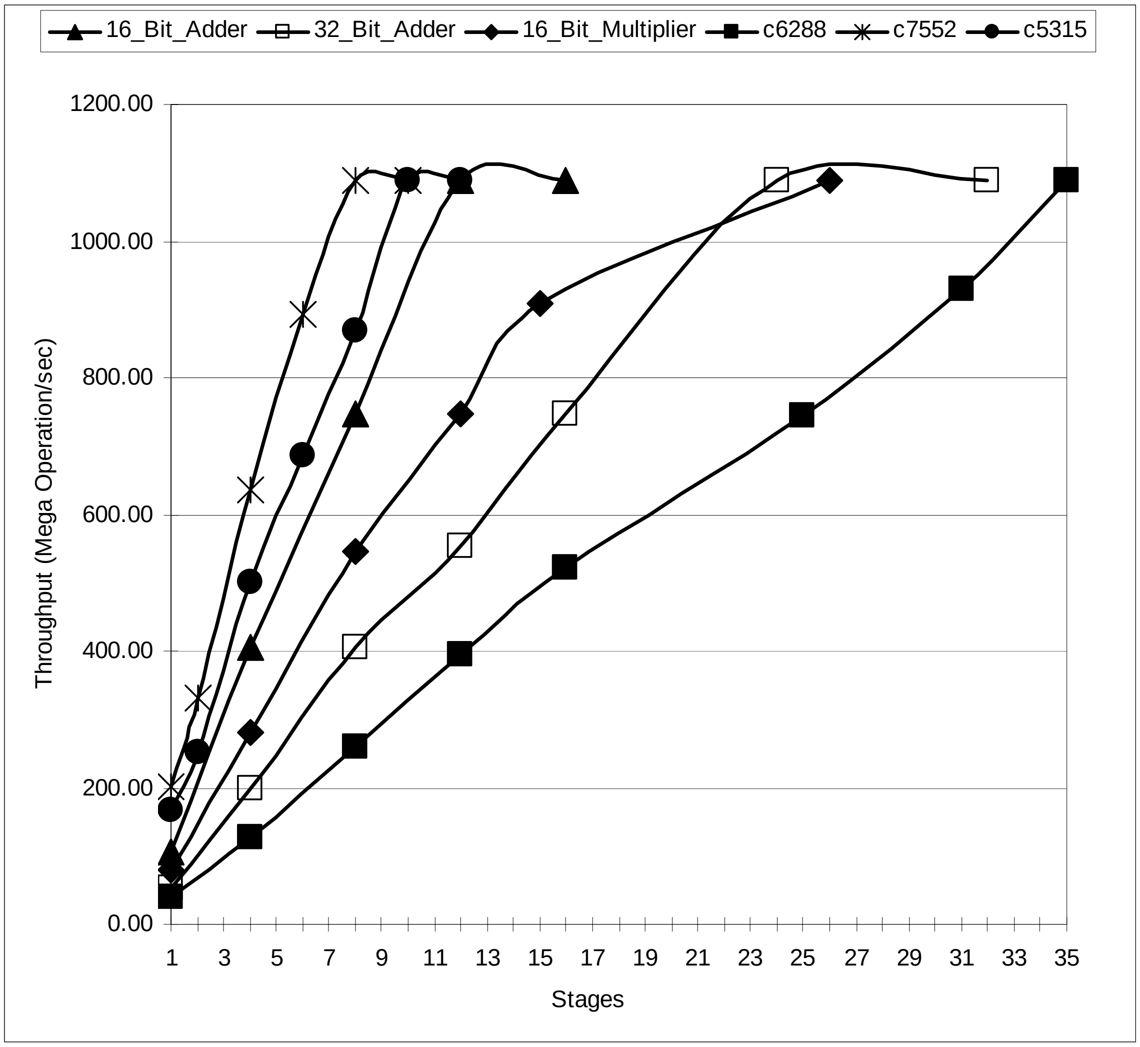

Figure 5.6: Pipeline throughputs for various D-SRSL pipeline depths.

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 throughputs 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. By partitioning these circuits further after this point, the delay of the LC block is not affected, and subsequently, the duration of the latch and reset phase remain constant. This has the effect of keeping the period 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.

5.4. Summary

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. As shown in Table 5.2, the experimental results reveal that P-SRSL and D-SRSL pipelines can reach higher throughput in wide and shallow pipelines for a relatively small number of stages in general. Furthermore, both pipelines can produce higher throughputs if wide and deep circuits are pipelined into deeper pipelines. With regard to area cost, it tends to be higher in narrow and shallow circuits for P-SRSL and D-SRSL pipelines. This shows that both pipelining techniques are suitable for coarse-grain datapaths.

Table 5.2: Area cost and throughput summary of the experimental circuits.

Circuit		**Area**		**Throughput**	
Breadth	**Depth**	**P-SRSL**	**D-SRSL**	**P-SRSL**	**D-SRSL**
Narrow	Shallow	High	High	Moderate	Moderate
	Deep	High	High	Moderate	Moderate
Wide	Shallow	Moderate	Moderate	High	High
	Deep	Low	Low	Low	Low

CHAPTER SIX: SUMMARY AND CONCLUSION

This thesis presents a synthesis methodology for SRSL pipelines based on SRSL logic. Since SRSL is totally implementable in static CMOS, the proposed synthesis methodology takes maximum advantage of the maturity of current CAD tools and the availability of standard cell CMOS libraries to synthesize and verify these SRSL pipelines. This methodology formulates the synthesis problems as an integer programming problem whose size is too large for a time-efficient solution. As a result, a two-phase algorithm is proposed instead to solve this synthesis problem. The first phase of this algorithm assigns each gate of the netlist to a specific pipeline stage without violating the pipeline period constraint while the second phase minimizes the number of inter-stag latches between every pair of adjacent stages in the pipeline. The second phase is necessary since latches in our standard cell library tend to occupy an area that is two to seven times larger than a two-input gate. The experiments conducted to validate SRSL pipelining show that SRSL pipelines can easily reach a throughput above 1 GHz although this throughput depends mainly on the pipeline depth and the standard cell library used in the implementation. Pipeline depth can be limited only by the duration of the reset phase of a single stage and the minimum time required for a latch to be transparent in order to capture data [16]. The pipelining experiments reveal also that the ratio of area overhead needed for SRSL and the area of the logic embedded within a single stage is relatively low. This shows that in the overall SRSL pipelining incurs a lower cost when applied to coarse-grain datpaths.

While the proposed synthesis methodology is a first step toward supporting the design and verification of SRSL pipelines, it still does not address in its current form the following issues:

(i) *How can the interconnect effects be incorporated in the delay model used by the synthesis methodology?* These effects are increasingly dominant in the nanometer range of CMOS and subsequently cannot be ignored [34].

(ii) *How can power effects be incorporated in the synthesis methodology?* By taking account of power consumption, the methodology can synthesize pipelines that are optimal in terms of area, throughput, and power. Similarly to interconnect effects, power is becoming a critical factor in the performance and reliability of circuits implemented in nanometer CMOS devices [35].

(iii) *How would SRSL fare in comparison to previous self-resetting circuits implemented in dynamic circuits?* The answer to this question can be addressed only after addressing issue (i) and (ii). In fact, by incorporating interconnect and power effects in the proposed synthesis methodology, additional experiments can be conducted in order to tally power and throughput numbers for comparison purposes.

(iv) *How can the synthesis algorithm be modified to handle different implementations with a high degree of flexibility?* It is worth noting that the primary goal of the second phase of the synthesis algorithm is to minimize the area occupied by the latches in the pipeline. As stated before, this optimization is necessary since the latches of the cell library used in the implementation tend to occupy a sizable area. What if a new cell library is used whose latches have comparable sizes to gates? What if the design is implemented on a FPGA chip which is already populated with copious amount of latches that can be used in the SRSL pipeline with no additional area cost? In that case, the optimization in the second phase may have to be completely bypassed. At first consideration, it seems that a multi-objective optimization approach would be highly suitable for the second phase of the synthesis algorithm.

(v) *How can the optimality of the second phase of the synthesis algorithm be evaluated?* Since this phase is totally heuristic, it would be interesting from a practical perspective to quantify how sub-optimal this phase is. Its sub-optimality can have a significant impact the area cost of the synthesized SRSL pipeline.

(vi) *How can this synthesis methodology be extended to sequential circuits?* Because the experimental benchmarks are all combinational circuits, it would be interesting to consider how to extend this methodology to handle sequential circuits. A straightforward approach would be to (1) replace the clocked flip-flops of a sequential circuit by latches and (2) pad the feedback loops, encountered in sequential machines, with delay buffers as suggested in [36].

LIST OF REFERENCES

[1] K. Emerson, "Asynchronous design: an interesting alternative," *Proc. Tenth International Conference on VLSI Design*, 4-7 January 1997, pp. 318-320.

[2] C. H. V. Berkel, M. B. Josephs, and S. M. Nowick, "Scanning the technology: applications of asynchronous circuits," *Proceedings of the IEEE*, vol. 87, no. 2, pp. 223-233, February 1999.

[3] Fulcrum Microsystems, "Asynchronous circuit technology: An alternative for high-speed semiconductors," available at http://www.fulcrummicro.com/Library/async_wp.htm, 2004.

[4] I. E. Sutherland and J. Ebergen, "Computers without clocks," *Scientific American*, pp. 62-69, August 2002.

[5] T. I. Chappell, B. A. Chappell, S. E. Schuster, A. J. W., S. P. Klepner, R. V. Joshi, and R. L. Franch, "A 2-ns cycle, 3.8 ns access 512-Kb CMOS ECL SRAM with a fully pipelined architecture," *IEEE Journal of Solid-State Circuits*, vol. 26, no. 11, pp. 1577-1585, November 1991.

[6] R. A. Heald and J. C. Hols, "A 6-ns Cycle 256-kb cache memory and memory management unit," *IEEE Journal of Solid-State Circuits*, vol. 28, no. 11, pp. 1078-1083, November 1993.

[7] M. E. Litvin and S. Mourad, "Self-reset logic for fast arithmetic applications," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 13, no. 4, pp. 462-475, April 2005.

[8] W. Hwang, G. D. Gristede, P. Sanda, S. Y. Wang, and D. 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, pp. 1108-1117, August 1999.

[9] G. Jung, V. Sundarajan, and G. E. Sobelman, "A robust self-resetting CMOS 32-bit parallel adder," *Proc. IEEE International Symposium on Circuits and Systems*, 26-29 May 2002, pp. 473-476.

[10] G. Jung, J. Kong, G. E. Sobelman, and K. K. Parhi, "High-speed add-compare-select units using locally self-resetting CMOS," *Proc. International Symposium on Circuits and Systems*, 26-29 May 2002, pp. 889-892.

[11] A. L. Lavi, A. Charnas, M. Tremblay, A. R. Dalal, B. A. Frederick, C. R. Srivasta, D. Greenhill, D. L. Wendell, D. D. Pham, E. Anderson, H. K. Hingarh, I. Razzack, J. M. Kaku, K. Shin, M. E. Levitt, M. Allen, P. A. Ferolito, R. L. Bartolotti, R. K. Yu, R. J. Melanson, S. I. Shah, S. Nguyen, S. S. Mitra, V. Reddy, V. Ganesan, and W. J. d. Lange, "A 64-b microprocessor with multimedia support," *IEEE Journal of Solid-State Circuits*, vol. 30, no. 11, pp. 1227-1238, November 1995.

[12] V. Narayanan, B. A. Chappell, and B. M. Chappell, "Static timing analysis for self-resetting circuits," *Proc. IEEE/ACM International Conference on Computer-Aided Design*, San Jose, CA, November 1996, pp. 119-126.

[13] F. Klass, M. J. Flynn, and A. J. v. d. Goor, "16x16-bit Static CMOS wave-pipelined multiplier," *Proc. IEEE International Symposium on Circuits and Systems*, June 1994, pp. 143-146.

[14] T. J. Thorp, G. S. Yee, and C. M. Sechen, "Design and synthesis of dynamic circuits," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 11, no. 1, pp. 141-149, February 2003.

[15] C. P. Soitriou, "Implementing asynchronous circuits using a conventional EDA tool-flow," *Proc. Design Automation Conference*, New Orleans, LA, 10-14 June 2002, pp. 415-418.

[16] A. Sharqawi, "DESIGN AND SYNTHESIS OF CLOCKLESS PIPELINES BASED ON SELF-RESETTING STAGE LOGIC," Ph.D. Dissertation, Department of Electrical and Computer Engineering, University of Central Florida, Orlando, Florida, 2005.

[17] A. Ejnioui and A. Alsharqawi, "A clockless reconfigurable array based on self-resetting logic," *Proc. Multi-conference on Systemics, Cybernetics, and Informatics*, Orlando, Florida, vol. 11, June 2004, pp. 61-66.

[18] A. Ejnioui and A. Alsharqawi, "Self-resetting stage logic pipelines," *Proc. ACM Great Lakes Symposium on VLSI*, Boston, Massachusetts, April 2004, pp. 174-177.

[19] A. Ejnioui and A. Alsharqawi, "Pipeline design based on self-resetting stage logic," *Proc. IEEE Computer Society Annual Symposium on VLSI*, Lafayette, Louisiana, February 2004, pp. 254-257.

[20] A. Ejnioui and A. Alsharqawi, "Pipeline-level control of self-resetting pipelines," *Proc. Euromicro Symposium on Digital System Design*, Rennes, France, September 2004, pp. 342-349.

[21] A. Ejnioui and A. Alsharqawi, "Pipeline level control of self-resetting stage logic pipelines," *Proc. IEEE Northeast Workshop on Circuits and Systems*, Montreal, Canada, June 2004, pp. 389-392.

[22] R. Oreifej, A. Alsharqawi, and A. Ejnioui, "Pipeline synthesis of SRSL circuits," *Proc. IEEE International Conference on Electronics, Circuits and Systems*, Gammarth, Tunisia, December 2005.

[23] A. Alsharqawi and A. Ejnioui, "Synthesis of self-resetting stage logic pipelines," *Proc. IEEE Computer Society Annual Symposium on VLSI, 2005*, St. Petersburg, Florida, May 2005, pp. 260-263.

[24] R. Oreifej, A. Alsharqawi, and A. Ejnioui, "Synthesis of Pipelined SRSL Circuits," *Proc. IEEE Computer Society Annual Symposium on VLSI*, Karlsruhe, Germany, March 2006, pp. 71 - 76.

[25] 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, pp. 667-678, 2002.

[26] 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.

[27] D. Wendell, "Reset logic circuit and method." U.S. Patent 5,430,283.

[28] G. DeMicheli, *Synthesis and optimization of digital circuits*: McGraw-Hill, 1994.

[29] G. L. Nemhauser and L. A. Wolsey, *Integer and Combinatorial Optimization*: John Wiley and Sons, 1988.

[30] M. Minoux, *Mathematical Programming: Theory and Algorithms*: John Wiley and Sons, 1986.

[31] D. Gajski, N. Dutt, A. Wu, and S. Lin, *High-Level Synthesis: Introduction to Chip and System Design*: Kluwer Academic Publishers, 1992.

[32] B. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell System Technical Journal*, vol. 49, no. 2, pp. 291-307, February 1970.

[33] M. Hansen, H. Yalcin, and J. P. Hayes, "Unveiling the ISCAS-85 benchmarks: A case study in reverse engineering," *IEEE Design & Test of Computers*, vol. 16, no. 3, pp. 72-80, July-September 1999.

[34] Y. Taur, D. A. Buchanan, W. Chen, D. J. Frank, K. E. Ismail, S.-H. Lo, G. A. Sai-Halasz, R. G. Viswanathan, H.-J. C. Wann, S. J. Wind, and H.-S. Wong, "CMOS scaling into the nanometer regime," *Proceedings of the IEEE*, vol. 85, no. 4, pp. 486-504, April 1997.

[35] G. Yeap, *Practical Low Power Digital VLSI Design*: Kluwer Academic Publishers, 1998.

[36] R. O. Ozdag and P. A. Beerel, "High-speed QDI asynchronous pipelines," *Proc. Eighth International Symposium on Asynchronous Circuits and Systems*, 8-11 April 2002, pp. 13-22.