AREA, POWER AND PERFORMANCE OPTIMIZATION ALGORITHMS FOR ELASTIC CIRCUIT CONTROL NETWORKS

by

Eliyah Wadie Ragi Kilada

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Electrical and Computer Engineering

The University of Utah

May 2012

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Eliyah Wadie Ragi Kilada**

has been approved by the following supervisory committee members:

Kenneth S. Stevens	, Chair	**01/12/2012** Date Approved
Chris J. Myers	, Member	**01/04/2012** Date Approved
Erik Brunvand	, Member	**01/12/2012** Date Approved
Priyank Kalla	, Member	**01/12/2012** Date Approved
Michael Kishinevsky	, Member	**01/12/2012** Date Approved

and by **Gianluca Lazzi** , Chair of

the Department of **Electrical and Computer Engineering**

and by Charles A. Wight, Dean of The Graduate School.

ABSTRACT

Elasticity is a design paradigm in which circuits can tolerate arbitrary latency/delay variations in their computation units as well as communication channels. Creating elastic (both synchronous and asynchronous) designs from clocked designs has potential benefits of increased modularity and robustness to variations. Several transformations have been suggested in the literature and each of these require a handshake control network (examples include synchronous elasticization and desynchronization). Elastic control network area and power overheads may become prohibitive. This dissertation investigates different optimization avenues to reduce these overheads without sacrificing the control network performance. First, an algorithm and a tool, CNG, is introduced that generates a control network with minimal total number of join and fork control steering units.

Synchronous Elastic FLow (SELF) is a handshake protocol used over synchronous elastic designs. Comparing to its standard eager implementation (that uses eager forks - $EFork$s), lazy SELF can consume less power and area. However, it typically suffers from combinational cycles and can have inferior performance in some systems. Hence, lazy SELF has been rarely studied in the literature. This work formally and exhaustively investigates the specifications, different implementations, and verification of the lazy SELF protocol. Furthermore, several new and existing lazy designs are mapped to hybrid eager/lazy implementations that retain the performance advantage of the eager design but have power and area advantages of lazy implementations, and are combinational-cycle free.

This work also introduces a novel ultra simple fork ($USFork$) design. The $USFork$ has two advantages over lazy forks: it is composed of simpler logic (just wires) and does not form combinational cycles. The conditions under which an $EFork$ can be replaced by a $USFork$ without any performance loss are formally derived.

The last optimization avenue discussed in this dissertation is Elastic Buffer Controller (EBC) merging. In a typical synchronous elastic control network, some EBCs may activate their corresponding latches at similar schedules. This work provides a framework for finding and merging such controllers in any control network; including open networks (i.e., when the environment abstract is not available or required to be flexible) as well

as networks incorporating variable latency units. Replacing $EFork$s with $USFork$s under some equivalence conditions as well as EBC merging have been fully automated in a tool, HGEN.

The impact of this work will help achieve elasticity at a reduced cost. It will broaden the class of circuits that can be elasticized with acceptable overhead (circuits that designers would otherwise find it too expensive to elasticize). In a MiniMIPS processor case study, comparing to a basic control network implementation, the optimization techniques of this dissertation accumulatively achieve reductions in the control network area, dynamic, and leakage power of 73.2%, 68.6%, and 69.1%, respectively.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

DI Delay Insensitive *design*

LI Latency Insensitive *design*

SELF Synchronous ELastic Flow *protocol*

MIPS Microprocessor without Interlocked Pipeline Stages

EFork Eager Fork

LFork Lazy Fork

USFork Ultra Simple Fork

HFork Hybrid Fork

LJoin Lazy Join

EB Elastic Buffer

EBC Elastic Buffer Controller

CNG Control Network Generator *tool*

HGEN Hybrid GENerator *tool*

ACKNOWLEDGEMENTS

I am grateful to have the chance to do research in the University of Utah. Special thanks go to my advisor and mentor, Ken Stevens, for providing confidence, experience, directions, and funding throughout this past three years. Thanks to all my committee members for the fruitful discussions and technical inputs. Thanks to Chris Myers and Priyank Kalla for the classes they taught me and to Erik Brunvand for his continuous support for the 0.5 μ UoU library and the tool flow. Thanks to Michael Kishinevsky for sharing his expertise in synchronous elastic architectures, and for providing a verilog generic model for variable latency unit modules. Thanks to Alan Mishchenko for his support with the ABC tool, to Ganesh Gopalakrishnan for his help with the 6thSense tool license, and to Suresh Venkatasubramanian for his insights on the CNG complexity. Bennion Redd has been so helpful to me while working on the Verigy's V93000 SoC tester. I also like to thank Shomit Das for his help in the place, route, and layout of the 0.5 μm chips. Thanks to the stackoverflow.com community for the valuable help on LaTeX. Thanks to Lori Sather for her administrative help. This material is part of work supported by the National Science Foundation under Grant No. 0810408. Finally, I am thankful for my friends and relatives who made my time in Utah and Canada most fruitful and enjoyable, and for my parents who sowed the seeds of faith, discipline, and love of knowledge in my life.

CHAPTER 1

INTRODUCTION

The dissertation problem statement is to reduce the power and area overheads of elastic system control networks without compromising performance.

1.1 Background And Motivations

1.1.1 What Is Elasticity?

Elasticity is a design paradigm in which circuits can tolerate arbitrary latency/delay variations in their computation units as well as communication channels [2, 3]. Different levels of elasticity exist. Delay-Insensitive (DI) designs function correctly whatever the delay of their gates or wires [4]. Thus, DI designs provide the highest degree of elasticity. However, the number of circuits that can be implemented using DI methodology is limited [5].

This dissertation will focus on the synchronous implementation of elasticity (also known as latency insensitive (LI) design) [8, 9, 10, 11]. Some of the algorithms introduced in the work can also be extended to asynchronous elasticity with bundled data (and, for short, may be referred to later as just asynchronous elasticity or desynchronization) [4, 6, 7]. LI designs can tolerate discrete number (of clock cycles) of computation and communication latency variations, while asynchronous elasticity can tolerate finer delays.

1.1.2 Why Elasticity?

Elastic design provides advantages much needed in the nanometer era. Without loss of generality, and for the ease of explanation, most of the following advantages will be illustrated through synchronous elasticity. Since LI design provides discrete elasticity of the finer asynchronous elasticity [3], these advantages naturally extend to the asynchronous implementation as well.

1. *Provides tolerance for long interconnect latency variations and easier technology migration.* The International Technology Roadmap for Semiconductor (ITRS) reported in 2009 that chip-long communication cannot be done in a single clock cycle any more

[12]. Hence, interconnect pipelining is becoming a necessity. Interconnect delays are affected by many factors that may not be accurately estimated before the final layout (e.g., physical distance, metal layer used, crosstalk, etc.) [13, 14]. They also do not scale as well as logic gates [15, 16, 17]. Hence, due to technology migration or place and route extra delays, it is very likely to have interconnects that suffer different latencies than estimated at earlier stages of the design. Hence, unless the design implements some kind of latency insensitive technique, severe changes may be required in the system to accommodate the new latencies and, possibly, a number of iterations [9, 17, 12]. This increases the time-to-market of a product. On the other hand, LI designs tolerate the variations of interconnect latencies by inserting any required number of empty pipeline stages (called bubbles). This essentially cuts an interconnect into segments that meet the target timing constraints. By the definition of LI design, inserting empty pipeline stages does not affect the system functionality.

2. *Provides easier latency/throughput tradeoff exploration.* For either ordinary clocked designs or LI, architectural analysis is required to compute and optimize the impact of inserting pipeline stages on the overall system performance [18, 19, 20, 21]. Nonetheless, the LI methodology allows for an easier exploration of latency/throughput tradeoffs, since the computational blocks can be left untouched while inserting interconnect pipelines [22]. This also allows for easier exploration of new architectures [23, 24, 25].
3. *Provides more modular design and easier IP reuse.* IP reuse is a key consideration for increased productivity in the current technology [12]. LI methodology facilitates IP assembly and reuse in complex SoCs. It can tolerate variable interconnect latencies among IPs without need of changing them.
4. *Is a natural fit for variable latency designs/interfaces - increasing performance by targeting the more frequent faster cases rather than the worst case.* Some applications require flexible interfaces that can tolerate variable latencies. Examples include interfaces to variable latency ALUs, memories or network on chip [26, 27, 28, 29]. By its definition, LI methodology naturally fits in these applications. In fact, it has been reported that applying flexible latency design to the critical block of one of Intel® SOC (H.264 CABAC) can achieve 35% performance advantage [30]. Variable latency design aims at targeting an average performance rather than the worst case. In particular, instead of optimizing a circuit for all corner cases, variable latency design optimizes the fast paths in l_1 clock cycles, and the slow paths in l_2 cycles (with

$l_2 > l_1$). The average throughput increases as the probability of the input patterns that require longer latency decreases [31]. Though variable latency design comes at an area overhead, however, trying to achieve the same performance with static latency may lead to an even bigger design to meet the tight timing target.

5. *Enables pipelining cyclic systems - a goal that cannot be achieved by the standard bypass and retiming of regular clocked systems* [23]. To illustrate, consider the Read-Modify-Write (RMW) memory structure of Fig. 1.1. The memory structure supports three different operations (*op*s): read (*rd*), write (*wr*) and read-modify-write (*rmw*). An example of a *rmw* operation is updating a specific memory location through a modify function f_M (e.g., $f_M(mem[adr1]) = mem[adr1] + 1$). For simplicity, assume the *op*s arrive to the memory interface with a maximum rate of 1 operation per clock. Bypass logic is designed around the memory to guarantee that every read operation from a memory location gets the *most recent* data written to that location (also referred to as memory access coherency). With regular bypass design, and if back-to-back *rmw* operations (of the same memory address) are allowed, the modify function f_M cannot have a latency of more than 1 clock cycle (i.e., cannot be pipelined), otherwise the output of f_M may be required for a following operation while f_M is still being executed. Thus, the standard bypass and retiming of regular clocked designs cannot pipeline f_M in this cyclic system. This is a typical observation that I also noticed while designing and verifying memory bypass logic during my internship at Cisco Systems®, Canada (Jan - Jul, 2011). On the other hand, LI design is able to pipeline cyclic systems through its natural capability to tolerate variable latency and to stall. For example, in LI design f_M can be pipelined to take any number of clock

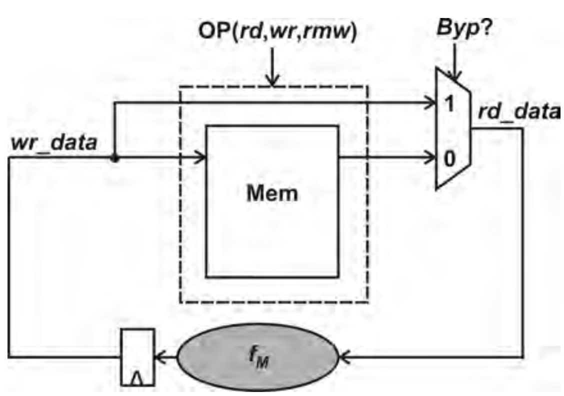


Figure 1.1: Sample read-modify-write memory structure.

cycle latencies (to decrease the clock period for example). Whenever the output of f_M is required while it is still executing, LI designs provide the natural ability to propagate back a stall signal through the system until f_M finishes execution. Moreover, whenever f_M is not required, LI design (with an early evaluation join [32], for example) provides the ability to ignore f_M output such that the system will operate unstalled (i.e., with its normal latencies). Solving this design problem with synchronous elasticity using an early evaluation join is illustrated in [23].

6. *Saves dynamic power by activating stages only when necessary.* LI design provides a fine-grained (per pipeline stage) clock gating based on dynamic data flow [8]. In LI designs, a stage is only activated when it is processing valid data and its downstream is not stalled. This can reduce the system dynamic power consumption. However, an offset to this power saving is the power overhead of the hand-shake control network.
7. *Avoids distribution of long stall signals that can be on critical paths.* LI design also provides an upstream stage-based stall propagation mechanism with no overhead on the clock frequency. This avoids distribution of long global stall signals that can be on critical paths and can limit scalability [8, 23].
8. *Asynchronous elastic designs provide low electro-magnetic interference (EMI)* [6].
9. *Asynchronous elastic designs provide finer and dynamic tracking of Process, Voltage, and Temperature (PVT) variations - allowing for better typical case performance rather than worst case.* Asynchronous elastic circuits synchronize through hand-shake signals (request/acknowledge) rather than a global clock. Hence, while the clock period of synchronous designs (and, in turn, their performance) is limited by the worst case conditions (of process, voltage, and temperature variations), asynchronous designs dynamically track the PVT variations providing better typical performance. Authors of [7] reported that a desynchronized DLX processor in 90 nm process has a performance degradation of 20% compared to a clocked one when both operate under worst case conditions. However, the desynchronized processor runs faster than the synchronous one in 90% of the time. They also reported 13.44% area overhead.

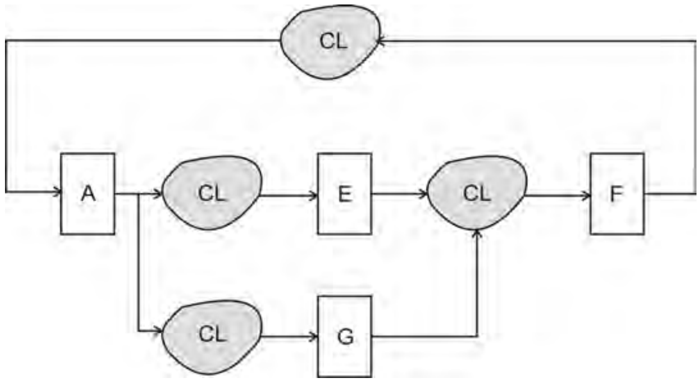
1.1.3 Elasticization: Converting a Normally Clocked System into Elastic

Because of the above advantages, converting an ordinary clocked system into elastic (also referred to as elasticization) has been frequently studied in literature. Carloni et al. [2] introduced the concept of *patient processes* as a theoretical model for latency insensitive design (aka synchronous elastic design). Informally, a module is a patient process if its behavior is defined based on signal events order rather than their exact latencies [2]. Since then, several approaches were proposed to convert a clocked circuit into elastic (in both its synchronous and asynchronous flavors). In all these approaches the resultant elastic and the ordinary clocked systems are *flow equivalent*. Two signals are flow equivalent if they exhibit the same sequence of informative events (i.e., after dropping all the *empty* events). Similarly, two systems are flow equivalent if, given flow equivalent input sequences, their outputs are flow equivalent [2, 33, 34].

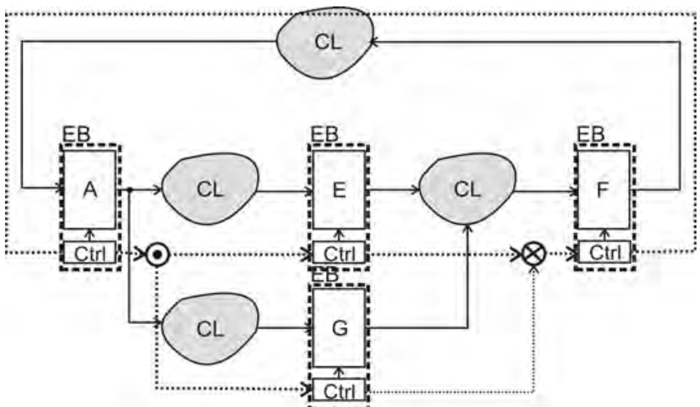
Before going further through the different elasticization schemes, it is useful to consider the elasticization example shown in Fig. 1.2. Fig. 1.2a shows a synchronous circuit composed of registers A, E, G, and F connected through combinational logic (CL). A typical first step in an elasticization scheme is to replace each flip-flop (or possibly a group of them) in the original clocked system with a synchronization element (possibly double latches) enabled through a corresponding controller[1]. Following this step, data communications among registers are analyzed. For each register-to-register data communication there must be a corresponding elastic control channel (shown in dotted lines in Fig. 1.2b) to control the data flow between these two registers. A control channel is usually composed of two signals, one in the forward direction indicating the data validity and the other in the backward direction carrying the stall information. These two signals are typically referred to as *Valid*/*Stall* and *Req*/*Ack* in synchronous and asynchronous elasticity, respectively. A network of control channels is formed where channels are connected through join and fork components. A join component (shown in Fig. 1.2b as $\otimes$) is used to join two or more input channels into one output channel. Similarly, a fork component (shown in Fig. 1.2b as $\odot$) is used to fork one input channel into two or more output channels. Implementations of the latch controllers, joins, forks, and channel protocol depend on the elasticization method.

On the asynchronous side, *desynchronization* was proposed to convert a normally clocked circuit into an asynchronous one [6, 7]. Desynchronized designs are synchronized through the regular asynchronous *Req* and *Ack* hand-shake signals rather than a universal clock. Bundled data protocols are normally used; examples include 4-phase, 2-phase, or single rail [4, 35]. For each register-to-register communication, delay elements are inserted in the control path to match the critical data path delay between these two registers. Thus, the

[1]LID-2ss and LID-1ss mentioned later in the chapter are slightly different. However, the main concepts of Fig. 1.2 still apply to them.



(a) Normally clocked.



(b) Elastic.

Figure 1.2: Converting a clocked system into elastic.

request signals are delayed long enough for the data signals to arrive. This guarantees each receiving latch is not activated before the data is ready at its input. Latch controller protocol design and implementation are crucial to achieve maximum concurrency among latch controllers, otherwise performance penalty can occur. Hence, different hand-shake protocols and latch controllers have been studied in the literature [36, 34, 37, 6, 35]. The matched delay elements keep track of their corresponding data path delays under different process, voltage and temperature variations. Thus, the desynchronized designs operate at a typical performance rather than the worst case (as in their clocked counterparts).

Algorithms have been developed for testing desynchronized circuits [38, 39, 40].

In the synchronous domain, an initial implementation for the latency insensitive design theorem was published in [22, 17, 41]. The initial implementation wraps normally clocked sequential modules inside latency insensitive wrappers (called pearls and shells, respectively). Channel latencies can be adjusted through what is called relay stations. The protocol requires a receiver to keep the *Stall* (also referred to as *Stop*) signal asserted for two consecutive clock cycles to stall the sender. Hence, the implementation was later referred to as *Latency Insensitive Design with two-stop-to-stall* (LID-2ss) [42]. To avoid data overflow, each shell contains (bypassable) input queues for each input of the corresponding pearl. The queues buffer the data tokens during stall conditions and are implemented by standard edge-triggered FIFOs [42].

Synchronous Interlocked Pipeline (SIP) technique was introduced with two major differences comparing to LID-2ss [8]. A stall condition is simpler and indicated by asserting the *Stall* signal for only one clock cycle. Second, instead of implementing external queues, SIP splits the same flip-flops used in the original clocked system into master-slave latches of opposite polarity and with separate enables. Under normal operation, the two latches will have one clock cycle forward latency (same as an edge triggered flip-flop). Under stall conditions, the two latches has the capacity (together) to carry two different data tokens while the stall signal is being propagated upstream if necessary. Thus, the SIP controllers consume less area than their LID-2ss counterparts [42].

The protocol used in SIP can, in principle, be used for arbitrary pipeline structures - including joins, forks, branches, and selects. However, the proposed implementation in [8] of the *aligned* (also referred to later as *lazy*) fork component can easily form combinational cycles when connected to join components in an arbitrary control network. The concept of state-machine based *nonaligned* (also referred to later as *eager*) fork was introduced in [8] but not implemented. Because of its *eagerness* eager forks can allow for shorter runtime comparing to lazy forks. Authors of [9, 10], based on a similar implementation to [8], proposed an automatic procedure to convert an arbitrary clocked circuit into LI, namely, *synchronous elasticization*. The protocol name was coined as *Synchronous ELastic Flow* (SELF). They also implemented the eager fork. Eager forks constitute no combinational cycles when connected to joins, allowing synchronous elasticization for arbitrary clocked designs. Also, support for synchronous variable latency controllers was included in [9, 10].

Other significant latency insensitive protocols include *Phased SELF* (or pSELF) and LID-1ss. pSELF is a modified version of SELF that maps easier to and from the asynchronous *Req/Ack* hand-shake protocol [26, 27]. LID-1ss was proposed as a modified version of LID-2ss with stall condition indicated by asserting the *Stall* signal for only one clock cycle [42]. A frame work for validating latency insensitive protocol families is given in [33].

Several enhancements to the original synchronous elasticity (with the SELF protocol) have then been reported. The regular join component waits for all its input channels to carry valid data before it passes the data token to the output. *Early evaluation* joins wait only for a required subset of inputs to be valid to start execution [32]. For correct operation, the early evaluation join must keep track of the inputs that were not required when they arrive later. This is done by sending *anti-token* on the opposite direction of their control channels. When an anti-token meets a token on a control channel they annihilate [32]. An example for that is a multiplexor where both the selection line and the selected input are valid while the nonselected input has not arrived yet. In such a case an early evaluation join will process the valid input, pass the data token to the output, and pass an anti-token to the nonrequired input. Early evaluation achieves performance advantage over lazy evaluation when join inputs have different arrival latencies [43].

Several transformations that are well-known in the synchronous design to improve performance have been carried over to synchronous elastic circuits in correct-by-construction fashion. These include retiming, recycling and speculation [44]. Nonetheless, other transformations that can also enhance performance are available only to elastic circuits. Examples include empty-FIFO (bubble) insertion, FIFO-capacity increase, anti-token insertion, and early evaluation [23].

1.2 Elasticity Overhead

Generating a control network is a necessary step in any of the elasticization approaches. The elastic control network area and power overheads may become prohibitive in some cases [3].

A desynchronized DLX processor in 90 nm process is reported to have a 13.44% area overhead (over the normally clocked one), and noticeable power overhead [7].

Authors of [42] show that elasticizing a 32×32 6-stage-pipelined multiplier with three different synchronous elasticization techniques results in an area overhead ranging from 10% to 19%.

Our measurements of a MiniMIPS processor fabricated in a 0.5 μm node show that synchronous elasticization with an eager SELF implementation results in area and dynamic power penalties of 29% and 13%, respectively [45].

Adding advanced features to synchronous elastic circuits (e.g., early evaluation and anti-token propagation) can pose an area versus controller performance tradeoff [32].

Elastic control networks reflect the register-to-register communications in the original clocked system. The network overhead may decrease with wider data paths. Nonetheless, the overhead is remarkable when a design has a communication complexity comparable to its computation complexity.

Furthermore, elasticity can be applied at different levels of granularity [3]. A design may be divided into very few register groups, with every group enabled by only one elastic controller. However, finer granularity typically results in more robustness to variations, better performance, and is sometimes required to enjoy some of the elasticity advantages mentioned in Sec. 1.1.2 [7]. On the other hand, finer granularity typically comes at a higher elasticity cost in terms of area and power consumption.

For all these reasons, this dissertation aims at achieving elasticity at a minimized cost. This will be done through minimizing the control network area and power overheads without sacrificing performance. The impact of this work will broaden the class of circuits that can be elasticized with acceptable overhead (circuits that designers would otherwise find it too expensive to elasticize). The impact will also enable designers to deepen the level of elastic granularity in their designs to enjoy the full benefit of elasticity at a reasonable cost. Furthermore, all the algorithms in this dissertation (except CNGT flow presented in Appendix B) have been automated and applied to various benchmarks ensuring their suitability for tight time-to-market constraints.

1.3 List of Contributions

1. *Elasticization and fabrication of a MiniMIPS processor case study in 0.5 μm technology.* The MiniMIPS processor is an 8-bit subset of the MIPS (Microprocessor without Interlocked Pipeline Stages) designed by Hennessy [1, 46]. It has been elasticized using an *all* eager implementation of the SELF protocol. No bubbles or variable latency units were used. The control network has been hand optimized. The 0.5 μm MiniMIPS represents a class of circuits in which the register-to-register communication complexity is comparable to the computation complexity. It, thus, provides a basic starting point to run the optimization algorithms introduced in this dissertation. The elasticization case study and results have been published in [45].
2. *The Control Network Generator (*CNG*) algorithm and tool.* The elastic control network can be constructed in many different ways. A direct approach is provided in [9, 3]. In that approach, for each register that is receiving data communications from multiple registers, one multi-input join is connected to this register controller input. Similarly, for each register that is sending data communications to multiple registers,

one multi-output fork is connected to this register controller output. This approach, however, could be inefficient in terms of the total number of joins and forks used. Hence, this dissertation introduces CNG. CNG is an algorithm (and a CAD tool) that generates a control network with minimum total number of 2-input joins and 2-output forks. This can substantially reduce the power and area of the control network. CNG automatically generates the optimal network for both synchronous elasticization or desynchronization. Comparing to the approach of [9], a MiniMIPS case study shows that synchronous elastic implementation of the network generated by CNG will save 27.9%, 31.4%, and 28.5% of the control network area, dynamic, and leakage power, respectively. CNG is published in [47] and an extended version in [48]. PreCNG tool is also introduced. PreCNG takes an ISCAS benchmark and automatically finds and expresses the register-to-register communications in *eqn* and *verilog* formats as well as another format that CNG accepts. The work also formalizes the problem of control network generation in a form that can be optimized by commercial synthesis tools. Results are compared.

3. *Formal investigation of the specifications, different implementations, and verification of the lazy SELF protocol.* The Synchronous Elastic Flow (SELF) protocol is a communication protocol in synchronous elastic designs [9]. Eager implementation of this protocol was reported in [9]. This implementation uses eager forks ($EFork$s) that try to optimize the control network runtime on the expense of more area and power consumption. A lazy SELF implementation (i.e., that uses normal or, so called, lazy forks ($LFork$s)) consumes less area and power. However, the latter suffers from combinational cycles and inferior runtime in some systems. Therefore, lazy SELF has been rarely studied in the literature. To exploit its area and power advantages, this work formally and exhaustively investigates the specifications, different implementations, and verification of the lazy SELF protocol.

4. *Hybrid ($EFork$-$LFork$) SELF implementation.* To make use of the eager SELF runtime advantage and the lazy logic simplicity, this work introduces a novel hybrid implementation of the SELF protocol, where both eager and lazy forks are incorporated. The hybrid SELF implementation proposed in this dissertation uses eager forks only when needed for runtime optimization and combinational cycle cutting, and lazy forks otherwise. Conditions for replacing eager with lazy forks without runtime loss are formally derived. A MiniMIPS case study shows that, comparing to an *all* eager

implementation, a hybrid SELF (*EFork-LFork*) will save 31.8%, 26.0%, and 30.8% in the control network area, dynamic, and leakage power, respectively, without any performance loss. This and the previous contribution have been published in [49].

5. *Introducing an Ultra Simple Fork (USFork) design and the hybrid (EFork-USFork) SELF implementation.* To further extend the concept of hybrid network, this work introduces a novel fork structure called the Ultra Simple Fork (*USFork*). The *USFork* has two advantages over the lazy fork: it has even simpler logic (just wires) and it forms no combinational cycles. This allows for even more area and power reduction in the control network. The conditions under which an *EFork* will be protocol equivalent to a *USFork* (and thus can be replaced) are formally derived. Comparing to an *all* eager implementation of the elastic MiniMIPS processor, hybrid (*EFork-USFork*) implementation shows 36.9%, 31.3%, and 32.0% savings in the control network area, dynamic, and leakage power, respectively.
6. *Merging Elastic Buffer Controllers (EBCs) under some equivalence conditions verifiable in any synchronous elastic control network.* In a typical synchronous elastic control network, some Elastic Buffer Controllers (*EBC*s) may activate their corresponding latches at similar schedules. This can allow for possible merging of these controllers into one controller that feeds them all (as much as the physical placement permits). Similar observation has been made by the authors of [50]. However, their algorithm requires both the control network and its environment to have static latencies. Hence, this dissertation introduces a framework for merging such controllers in any control network. That includes open networks (i.e., when the environment abstract is not available or required to be flexible) as well as networks incorporating variable latency units. Comparing to an *all* eager implementation of the elastic MiniMIPS processor, hybrid (*EFork-USFork*) implementation with merged *EBC*s shows 62.8%, 54.1%, and 56.9% savings in the control network area, dynamic, and leakage power, respectively.
7. *The Hybrid Network GENerator (*HGEN*) tool.* HGEN incorporates the above two contributions. It takes an input *verilog* description of a control network. It runs IBM® 6thSense [51] as an embedded verification engine. HGEN produces a *verilog* description of a minimized version of the control network (i.e., *EFork*s that are protocol equivalent to *USFork*s are replaced, and optionally, equivalent *EBC*s are merged). Though HGEN has been used in this dissertation to do the *EFork* to

$USFork$ conversion and EBC merging, its value is more than that. HGEN provides a framework where any type of synchronous elastic network can be formally verified. Any future verification-based research or optimization can be readily integrated in the tool. HGEN and the above two contributions have been published in [52].

8. *The* CNGT *transformation flow.* CNG does not guarantee providing the minimum possible critical path delay in a control network. Normally this is not a problem since the critical delay of the datapath is usually larger than that of the control network. Nonetheless, this work introduces a systematic flow (referred to as CNGT) of structural transformations of the synchronous elastic control network that reduces the network delay to meet tight timing constraints. CNGT is verified that the two versions of the control network (i.e., before and after the transformations) are functionally equivalent. The flow, in its current state, does not take into account wire delays.

1.4 This Dissertation Structure

Chapter 2 gives an overview of synchronous elasticity and the SELF protocol. It also introduces the MiniMIPS elasticization as a case study.

Chapter 3 formalizes the problem of minimizing the total number of 2-input joins and 2-output forks in an elastic control network. It introduces the CNG theory, algorithm, and tool. Chapter 3 also compares the results of CNG to other possible flows using Synopsys® Design Compiler® (DC) [53] or Berkeley ABC [54] over ISCAS benchmarks and other case studies.

Chapter 4 formally and exhaustively investigates the specifications and different implementations of the lazy SELF protocol. It also introduces a hybrid implementation of the SELF protocol where both eager and lazy forks are used.

Chapter 5 introduces two techniques for further reducing the area and power overheads of synchronous elastic control networks, namely, utilizing the Ultra Simple Fork ($USFork$) and EBC merging. The two techniques have been integrated in an automatic tool, HGEN, based on 6thSense as an embedded verification engine.

Chapter 6 concludes the dissertation.

Appendix A shows some preliminary heuristics for running CNG on big problems. Appendix B introduces CNGT flow and transformations. CNGT aims at transforming a given synchronous elastic control network such that it meets tight timing constraints.

CHAPTER 2

SYNCHRONOUS ELASTICIZATION AND THE MINIMIPS CASE STUDY

Synchronous elasticization converts an ordinary clocked circuit into Latency-Insensitive (LI) design [8, 9, 10]. The Synchronous Elastic Flow (SELF) is an LI protocol that can be used over synchronous elastic control network channels. This chapter gives an overview of the synchronous elastic architectures, SELF protocol and the process of synchronous elasticization. MiniMIPS elasticization is used as a case study. The chapter is concluded with investigation of the possible control network optimization avenues.

2.1 Synchronous Elastic Architectures[1]

A synchronous elastic system replaces the flip-flops used as pipeline latches in a clocked system with Elastic Buffers (EBs). EBs serve the purpose of pipelining a design as well as synchronization points that implement an LI protocol, also allowing the clocked pipeline to be stalled.

Fig. 2.1 [9] shows a block diagram implementation of an EB. An EB consists of a data-plane (double latches) and a controller. It can be in the Empty (bubble), Half or Full states depending on the number of data tokens its two latches are holding. A sample implementation of the EB controller can be found in [9]. EB controllers communicate through control channels. Each channel contains two control signals. *Valid* (V) travels in the same direction as the data and indicates the validity of the data coming from the transmitter. *Stall* (S) travels in the opposite direction and indicates that the receiver cannot store the current data.

The SELF channel protocol is shown in Fig. 2.2. It defines three channel states:

1. *Transfer* (T): $V\&!S$. The transmitter provides valid data and the receiver can accept it.

[1] Section 2.1 is a revised version of work originally published in [49].

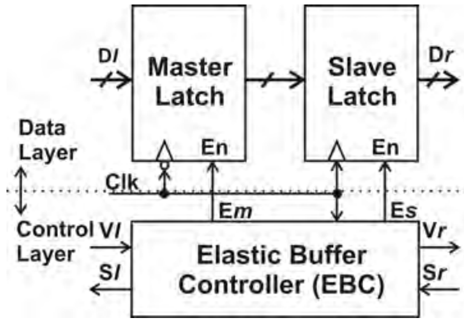


Figure 2.1: An EB implementation.

2. *Idle* (I): $!V$. The transmitter does not provide valid data. This dissertation identifies two Idle conditions: $I0$ ($!V\&!S$) where the receiver can accept data and $I1$ ($!V\&S$) where the receiver cannot accept data.
3. *Retry* (R): $V\&S$. The transmitter provides valid data, but the receiver cannot accept it. In the *Retry* state, the valid data must be maintained on the channel until it is stored by the receiver.

When the connection between EBs is not point-to-point, a control network is required to reflect the register-to-register communication in the original clocked circuit. The control network is composed of control channels connected through control steering units, namely, join and fork components. A join element combines two or more incoming control channels into one output control channel. A sample join design is shown in Fig. 2.3 [8, 9]. A fork element copies one incoming control channel into two or more output control channels. An n branch extension of the eager fork proposed in [9] is shown in Fig. 2.4. Fork and join components will be represented by $\odot$ and $\otimes$, respectively. Hereafter the term *control network* is used to aggregately refer to the joins, forks, and EB controllers in an elastic system.

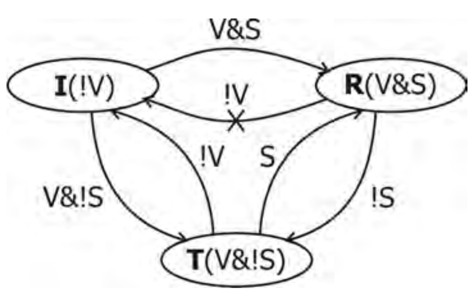


Figure 2.2: SELF channel protocol.

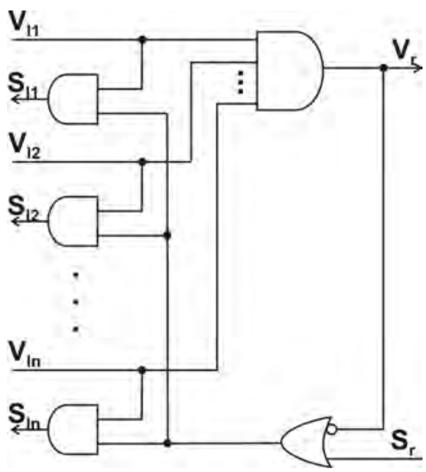


Figure 2.3: An n-to-1 lazy join.

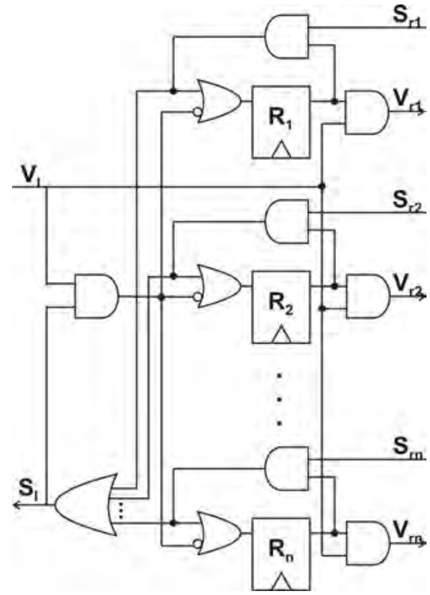


Figure 2.4: A 1-to-n $EFork$.

2.2 MiniMIPS Case Study and Results

MIPS (Microprocessor without Interlocked Pipeline Stages) is a 32-bit architecture with 32 registers, first designed by Hennessey [46]. The MiniMIPS is an 8-bit subset of MIPS, fully described in [1].

2.2.1 Elasticizing the MiniMIPS[2]

The MiniMIPS is used as a case study of elasticization. Fig. 2.5 shows a block diagram of the ordinary clocked MiniMIPS [55, 1]. The MiniMIPS has a total of 12 synchronization points (i.e., registers), shown as rectangles in Fig. 2.5: P (program counter), C (controller), $I1, I2, I3, I4$ (four instruction registers), A, B and L (ALU two input and one output registers, respectively), M (memory data register), R (register file) and Mem (memory).

To perform elasticization, each register is replaced by an elastic buffer (EB). Then, the register to register data communications in the MiniMIPS are analyzed. The following registers pass data to both A, B : R, to R : C, $I2$, $I3$, L, M, to C : C, $I1$, to $I1$, $I2$, $I3$, $I4$: C, Mem, to L : A, B, C, $I4$, P, to M : Mem, to Mem : B, C, L, P, and to P : A, B, C, $I4$, L, P. For each register to register data communication there must be a corresponding control channel to control the data flow of this communication. The resultant

[2]Section 2.2.1 is a revised version of work originally published in [45].

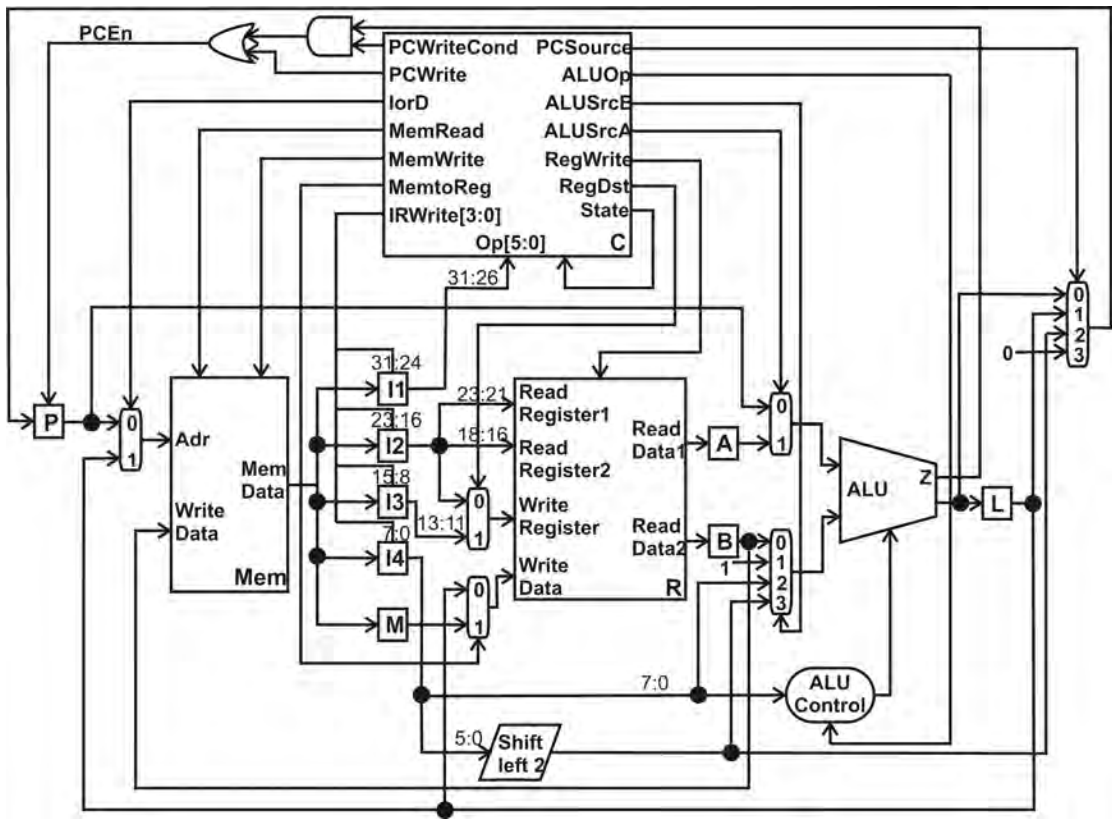


Figure 2.5: Block diagram of the ordinary clocked MiniMIPS.

control network can be implemented in different ways. Fig. 2.6 shows a control network that has been hand-optimized to minimize the number of joins and forks used in the control network (to reduce area and power consumption). From the control point of view, the register file (R) and memory (Mem) in a microprocessor can be treated as combinational units [9]. Hence, a separate *EB* for the register file (R) was not incorporated in Fig. 2.6. For the purpose of this case study, the memory (Mem) is off-chip.

From the elastic control point of view, the MiniMIPS control signals (e.g., *RegWrite*, *IRWrite*, etc. - see Fig. 2.5) are considered part of the data plane and they need their own corresponding control channels. Mapping between datapath signals in the clocked MiniMIPS (of Fig. 2.5) and the control channels in the elastic MiniMIPS (of Fig. 2.6) should be self explanatory for most signals. *RFWrite* in Fig. 2.6 is the *RegWrite* control channel. *RFWrite_valid* must be active if data is going to be written in the register file. Therefore, *RFWrite_valid* has been ANDed with *RegWrite* inside the register file.

Both the clocked and the elastic MiniMIPS have been synthesized, placed, routed and fabricated in a 0.5 μm technology. The functionality of the fabricated processors have been

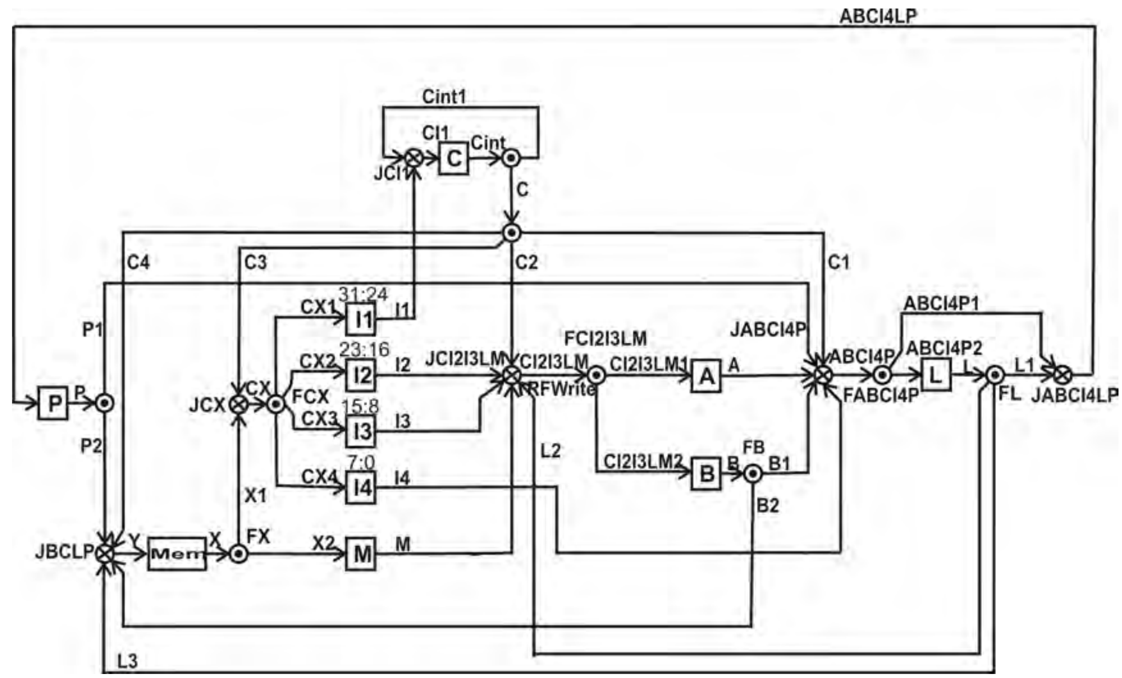


Figure 2.6: Hand-optimized control network of the elastic clocked MiniMIPS.

verified on Verigy's V93000 SoC tester using the testbench in [1]. An eager implementation of the SELF protocol has been used with the *EFork* and lazy join of Figures 2.4 and 2.3, respectively. Table 2.1 summarizes the chip measurements. It shows that elasticizing the MiniMIPS has area, dynamic and leakage power penalties of 29%, 13% and 58.3%, respectively. For accurate leakage power comparison, both designs have been set to the same state (through a test vector) before measuring the average leakage supply current.

Both MiniMIPS have been fabricated without the memory block. Memory values have been programmed inside the tester. An assumption about the memory access time was made. Since it affects the maximum operating frequency of both MiniMIPS designs in the same way, therefore, an arbitrary memory access time of zero was assumed. Schmoo plots

Table 2.1: Clocked and eager elastic MiniMIPS chip results. Measurements are done at 5V and 30°.

	Clocked MiniMIPS	Eager Elastic MiniMIPS	Penalty
Area (μm X μm)	1246.765 X 615.91	1284.1 X 771.54	29%
P_{dyn} @80 MHz (mW)	330	373	13%
P_{leak} (μW)	16.3	25.8	58.3%
f_{max} (MHz)	91.7	92.2	-0.5%

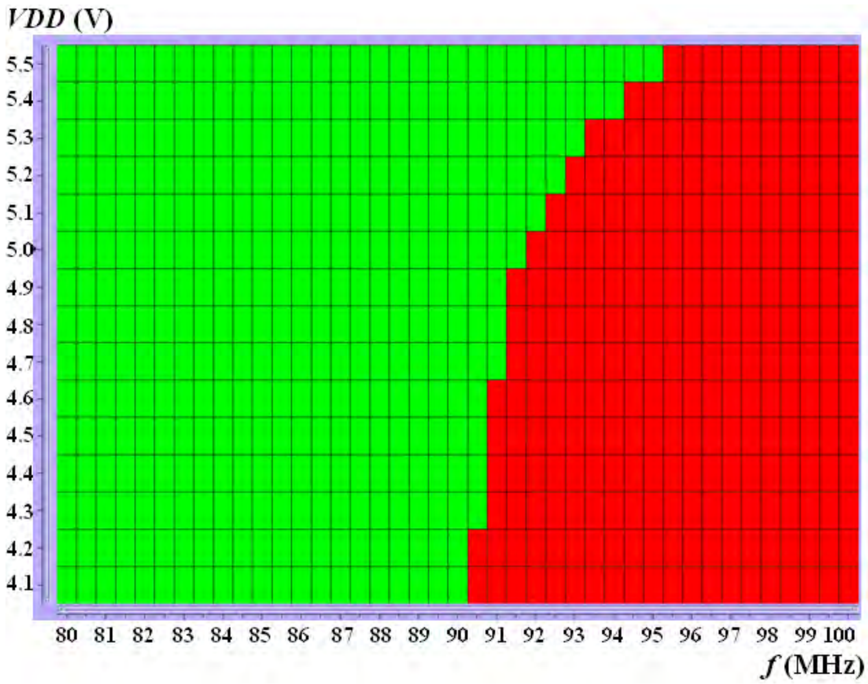
for both clocked and elastic MiniMIPS are shown in Fig. 2.7.

2.2.2 Case Study Evaluation

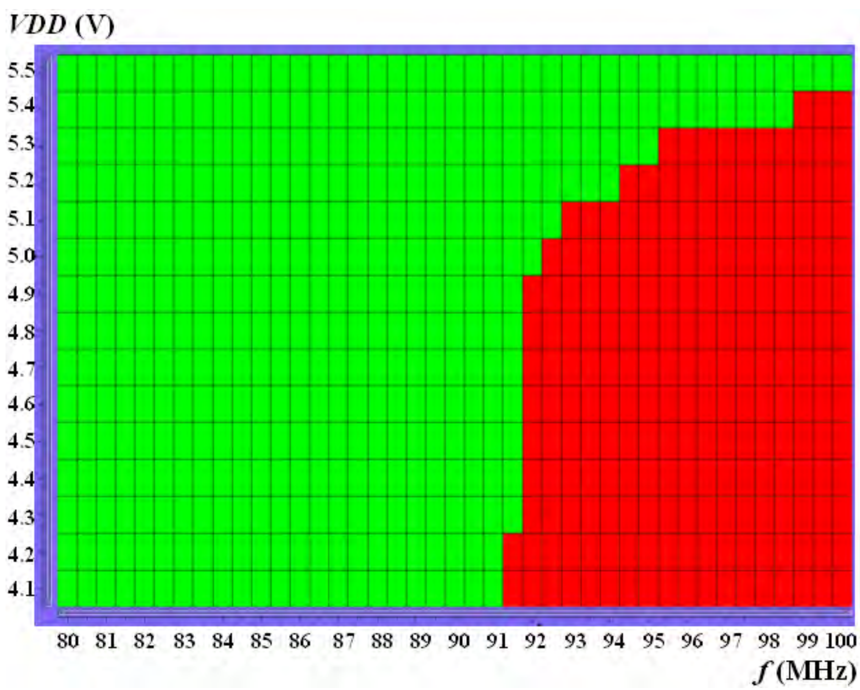
It should be noted that the elastic MiniMIPS has functional features that the clocked design does not have. The clocked design cannot support flexible interface latencies nor the addition of extra pipeline stages between registers. The fabricated MiniMIPS case study did not take advantage of these functional features. For example:

- The fabricated MiniMIPS (clocked and elastic) used an off-chip memory with static latency. If the memory latency is not static, the clocked design will have to implement some kind of latency insensitivity in the data path to accommodate for latency variations (e.g., cache miss). A sample approach could be a finite state machine waiting for the memory data *valid* signal to assert, while stalling the processor or running no-operation (NOP) tasks. This, on the other hand, is handled naturally in the elastic MiniMIPS by the means of the *Valid* and *Stall* control signals, without need for additional logic in the datapath. The overhead of adding some sort of latency insensitivity to the data path of the normally clocked MiniMIPS should be taken into account in the comparison. The power saving due to stalling the processor (in the elastic version) rather than running NOPs tasks (in the ordinary clocked one) should also be considered.
- The fabricated MiniMIPS (clocked and elastic) used fixed latency ALU. Similar argument applies as the above.
- The fabricated MiniMIPS (clocked and elastic) did not have long interconnects that had to be pipelined (i.e., no bubble insertion was needed). The synchronous elastic design naturally handles long interconnect latencies by inserting any number of empty pipeline stages (i.e., bubbles) to meet the target timing constraints. On the other hand, to handle the problem in the ordinary clocked version, severe changes in the design may be required and/or the system frequency may need to slow down.

Would elasticity be required (e.g., to accommodate variable latency interfaces, long interconnects, etc.), the presented MiniMIPS case study shows the cost of achieving this elasticity using the SELF protocol. The MiniMIPS is a relatively small design (8-bit datapath). The overhead of elasticization may decrease with increasing the word width. Nonetheless, the MiniMIPS represents a class of circuits in which the register-to-register communication complexity is comparable to the computation complexity. Thus, the control



(a) Schmoo plot for clocked MiniMIPS.



(b) Schmoo plot for elastic MiniMIPS.

Figure 2.7: Fabricated chips schmoo plots. Red boxes are for failed tests, while green are for passed ones.

network area and power overheads are remarkable. Other examples from the literature include:

- A desynchronized DLX processor in 90 nm process is reported to have a 13.44% area overhead (over the normally clocked one), and noticeable power overhead [7].
- Elasticizing a 32 $\times$ 32 pipelined multiplier for a pipeline depth ranging from 2 to 6 with three different synchronous elasticization techniques is reported to result in an area overhead ranging from as low as 5% to as much as 23% [42].

2.2.3 Optimization Avenues

1. Can the required register-to-register communication be achieved by using fewer number of joins and forks? What is the minimum? - Chapter 3.
2. Eager forks incorporate one flip-flop for each branch that is clocked every clock cycle. Thus, they are area and power expensive. Can the eager forks be replaced by lazy without sacrificing performance? - Chapter 4.
3. Are there any other fork structures that are cheaper in area and power than even lazy forks, do not form combinational cycles, and can substitute $EFork$s without any performance loss? What are the replacement conditions? - Chapter 5.
4. Elastic buffer controllers are area and power expensive. Is it possible to merge some of the EBCs without any performance loss? - Chapter 5.

CHAPTER 3

CONTROL NETWORK GENERATOR FOR ELASTIC CIRCUITS[1]

Creating latency insensitive or asynchronous designs from clocked designs has potential benefits of increased modularity and robustness to variations. Several transformations have been suggested in the literature and each of these require a handshake control network (examples include synchronous elasticization and desynchronization). Numerous implementations of the control network are possible. This chapter reports on an algorithm that generates an optimum control network consisting of the minimum total number of 2-input join and 2-output fork control components. This can substantially reduce the area and power consumption of the control network. The algorithm has been implemented in a CAD tool, CNG. It has been applied to the MiniMIPS processor showing a 14% reduction in the number of control steering units over the hand optimized version of Fig. 2.6, and a 42.9% reduction over a network that would be implemented using a basic approach introduced in [9]. CNG is also compared with control network synthesis approaches using industrial strength synthesis tools, e.g., Design Compiler® (DC) [53] from Synopsys® and ABC [54] from Berkeley. The tools were compared over many ISCAS-89 benchmarks as well as locally developed examples. In all complete benchmark runs in this chapter, DC and ABC produce a network with the same or more number of join (and fork) components than CNG. In s614, for example, ABC produces a network with 11.3% more joins than CNG (69 vs. 62). In s1238, DC produces a network with 10.9% more joins than CNG (51 vs. 46). Locally developed examples (in part based on observations seen in ISCAS benchmarks) show even more favor toward CNG. In one of the developed examples, DC produces a network with up to 50% more join components than CNG, and ABC with 57% more joins than CNG.

[1]This work has been submitted to the IEEE for possible publication [48]. Copyright may be transferred without notice.

3.1 Problem Definition

Example 3.1. Let I_1, I_2, X_1, X_2 be four registers in the original ordinary clocked design. Both registers I_1 and I_2 pass data to both registers X_1 and X_2. Find a control network implementation for the elastic version of this design.

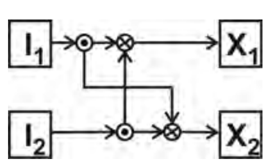
Figures 3.1a and 3.1b are two example implementations for such a control network. The control network in Fig. 3.1b has one fewer join and one fewer fork components than the network of Fig. 3.1a. Things get more complicated when the number of registers and their corresponding communications increase. Hence, the purpose of the proposed algorithm is, given a set of required register-to-register communications, the algorithm should automatically generate a control network with minimum total number of 2-input join and 2-output fork components.

This section lists a number of definitions required to formalize the problem. Example 3.2 will be used as a running example throughout the chapter.

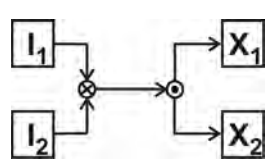
Example 3.2. Let $A, B, C, D, E, F, G, X_1, X_2, X_3, X_4, X_5$ be twelve registers in the original ordinary clocked design. The following registers pass data to $X_1 : B, C, G$, and to $X_2 : A, B, C, G$, and to $X_3 : A, B, C, D, E$, and to $X_4 : A, B, D, E, F$, and to $X_5 : A, B, E, F$. Find a control network implementation for the elastic version of this design, that incorporates minimum number of join and fork components.

A data transmitting register as well as a primary input will be referred to as an *input node* (or $INode$). Similarly, a data receiving register as well as a primary output will be referred to as an *output node* (or $ONode$).

The set of all $INode$s and the set of all $ONode$s in the network are designated as $INodeS$ and $ONodeS$, respectively. In Example 3.2, $INodeS = \{A, B, C, D, E, F, G\}$, and $ONodeS = \{X_1, X_2, X_3, X_4, X_5\}$. Note that, in a typical system, a register is both receiving



(a)



(b)

Figure 3.1: Two possible implementations of Example 3.1.

and transmitting data. Hence, from the data communication perspective, its data-input interface and data-output interface are $ONode$ and $INode$, respectively.

Definition 3.3. Term A set of one or more $INode$s.

Constructing a $Term$ typically means joining the control channels coming from its constituent $INode$s into one control channel. Each $Term$ has a unique identifier, $TermID$. As an example, a $Term$ that joins the control channels coming from: B, D, E, is $\{B, D, E\}$ and, for simplicity, will be referred to as BDE. $|Term_1|$ designates the cardinality of $Term_1$. A $Term$ that is associated with an *input node* (i.e., composed of only one $INode$) is called a $Source$. The set of all $Source$ $Term$s is designated as $SourceS$. Note that $|SourceS| = |INodeS|$.

Definition 3.4. Target A $Term$ that is associated with an *output node*. A $Target$ of a certain $ONode$ is a $Term$ composed of all $INode$s that send data to that $ONode$.

In Example 3.2, BCG is the $Target$ $Term$ associated with $ONode$ X_1. The set of all $Target$ $Term$s is designated as $TargetS$. Note that $|TargetS| = |ONodeS|$. The set of all $Term$s relevant to the problem is designated as $TermS$. Formally,

$$TermS = \{Term_i | Term_i \subseteq Target_j \quad \forall Target_j \in TargetS\} \tag{3.1}$$

$Term$s in $TermS$ or in any other $Term$ set introduced later are identified by their unique $TermID$ rather than their $INode$ set contents (see $Term$ definition in Def. 3.3). In general, every $INode$ set will map to at most one $TermID$. However, an exception for this rule, and without loss of generality, are the $INode$ sets of $Target$ $Term$s. This work assumes that $Target$ $Term$s are terminal in the sense that they cannot be used inside the control network to construct other $Term$s. If needed to be shared by other $Term$s, internal images that have the same $INode$ set are used inside the network instead. Hence, $TermS$ set of Eq. 3.1 can contain both a $Target$ as well as its internal image. An example in the $Term$s listed in Table 3.1 is the $Target$ whose $INode$ set is $\{B, C, G\}$ and $TermID = 1$. It has an internal image (i.e., with the same $INode$ set) which is the $Term$ whose $TermID = 8$.

Definition 3.5. Partial Solution or PS A set of $Term$s that could be used to implement another $Term$. Formally, PS_t (set) is a partial solution of $Term_t$, iff $\bigcup_{i=1}^{|PS_t|} Term_i = Term_t \wedge \forall Term_i \in PS_t : Term_iID \neq Term_tID$, where $Term_iID$ and $Term_tID$ are the $TermID$s of $Term_i$ and $Term_t$, respectively.

Table 3.1: $Terms$ and PSs of Example 3.2. $Term$ types are: $Target$ (T), $PTerm$ (P) and $Source(S)$.

$TermID$	$Term$	Type	$PSID$	PS	Initial $nUsed$	
					Max	Min
1	BCG	T	1	$\{BCG\}$	0	0
2	$ABCG$	T	1 2	$\{BCG, A\}$ $\{ABC, G\}$	0	0
3	$ABCDE$	T	1 2	$\{ABDE, C\}$ $\{ABC, D, E\}$	0	0
4	$ABDEF$	T	1 2	$\{ABDE, F\}$ $\{ABEF, D\}$	0	0
5	$ABEF$	T	1	$\{ABEF\}$	0	0
6	$ABDE$	P	1	$\{ABE, D\}$	2	0
7	$ABEF$	P	1	$\{ABE, F\}$	2	1
8	BCG	P	1	$\{BC, G\}$	2	1
9	ABC	P	1 2	$\{BC, A\}$ $\{AB, C\}$	2	0
10	ABE	P	1	$\{AB, E\}$	2	1
11	BC	P	1	$\{B, C\}$	2	1
12	AB	P	1	$\{A, B\}$	2	1
13-19	$A - G$	S, P	1			

PS_t represents one way of constructing $Term_t$. One $Term$ could be constructed in multiple ways, and thus has more than one PS. In Example 3.2, to construct $Term_t = ABCDE$, one possible PS is $\{ABC, D, E\}$. Another is $\{ABDE, C\}$. Note that, by definition, a $Term$ cannot be used to implement itself. Also, $Sources$ do not have PSs.

Definition 3.6. Solution or Soln A vector of PSs, where $TermIDs$ are used as indices (first index is 1). If $Soln_1$ is a $Solution$, and $Term_tID$ is the $TermID$ of $Term_t$, then $Soln_1[Term_tID]$ (or, for short, $Soln_1[Term_t]$) is the chosen PS to construct $Term_t$ in $Soln_1$. $Soln_1[Term_i] = \emptyset \Rightarrow Term_i \in SourceS$.[2]

In Example 3.2, the following is a possible $Solution$ ($Terms$ are sorted by their $TermIDs$ of Table 3.1, and $Source$ PSs are ignored):

$$\begin{aligned} Soln_1 = <& \{BCG\}, \{BCG, A\}, \{ABDE, C\}, \{ABDE, F\}, \\ & \{ABEF\}, \{ABE, D\}, \{ABE, F\}, \{BC, G\}, \\ & \{AB, C\}, \{AB, E\}, \{B, C\}, \{A, B\} > \end{aligned} \tag{3.2}$$

[2]Throughout this chapter, the $\Rightarrow$ symbol will be used to indicate implication, while $\rightarrow$ will indicate the domain and codomain of a function.

Hence, a $Solution$ can be seen as a vector of PS choices of different $Terms$. For example, $Soln_1[2] = \{BCG, A\}$. This means the $PS = \{BCG, A\}$ is used in $Soln_1$ to construct $Term\ ABCG$ (whose $TermID$ is 2). $Soln_1$ is depicted in Fig. 3.2. The set of all $Solutions$ is designated as $SolnS$.

Definition 3.7. nUsed $nUsed[Term_i]\Big|_{Soln_1}$ defines how many times $Term_i$ is used to construct other *useful* $Terms$ in $Solution$, $Soln_1$. Formally, $nUsed[Term_i]\Big|_{Soln_1}$ is defined recursively to be the number of $Terms$, $Term_t$, that satisfy the following two conditions:

1. $Term_i \in Soln_1[Term_t]$.
2. $nUsed[Term_t]\Big|_{Soln_1} > 0 \vee Term_t \in TargetS$.

By definition, $\forall Term_i \in TargetS : nUsed[Term_i] = 0$.

Definition 3.8. Useful Term $Term_i$ is said to be *useful* in $Soln_1$ (or $Soln_1$ uses $Term_i$), if any of the following two conditions hold:

- $Term_i \in TaregtS$.
- $nUsed[Term_i]\Big|_{Soln_1} > 0$.

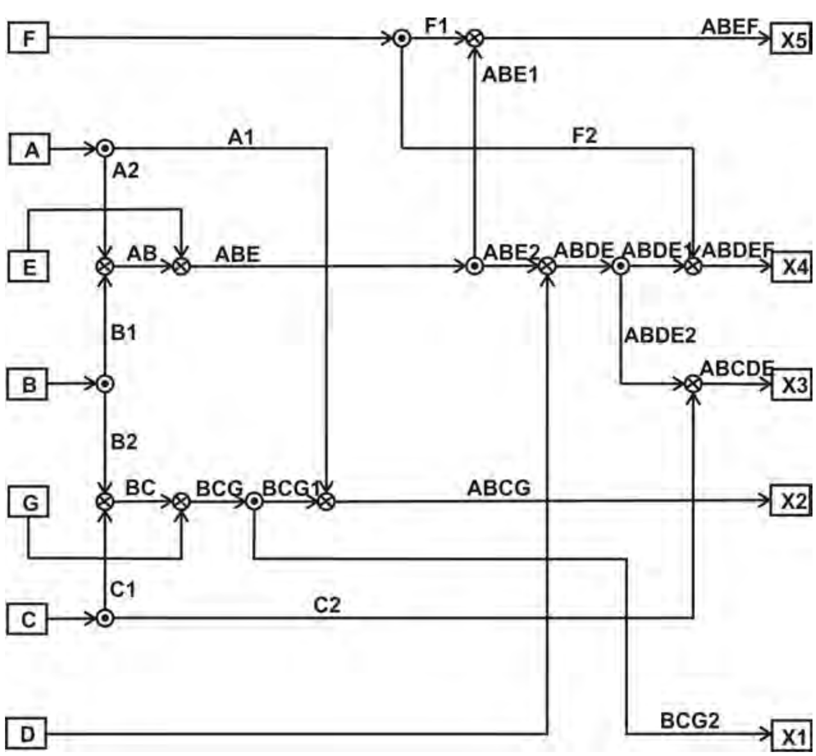


Figure 3.2: A sample control network of Example 3.2.

The function $UsefulTermS(Soln_1) : SolnS \rightarrow 2^{TermS}$ is defined to return the *useful* $Terms$ in a given $Solution$. Formally, $UsefulTermS(Soln_1) = UTermS$, where $UTermS = \{Term_i \in TermS \,|Term_i$ is *useful* in $Soln_1\}$.

The suffix $\Big|_{Soln_1}$ may be omitted from $nUsed$ and other data structures and functions when the context is clear. For Example 3.2 and $Soln_1$ of Eq. 3.2: $Term\ ABE$ (with $TermID$ of 10) is used to construct both $Terms$ $ABDE$ (with $TermID$ of 6) and $ABEF$ (with $TermID$ of 7). Hence, $nUsed[ABE]\Big|_{Soln_1} = 2$. Also, $Term\ ABC$ (with $TermID$ of 9) is not *useful* in $Soln_1$. $Term\ AB$ (with $TermID$ of 12) is used to construct both $Terms$ ABC (with $TermID$ of 9) and ABE (with $TermID$ of 10). However, since $Term\ ABC$ is not *useful* in $Soln_1$, therefore, $nUsed[AB]\Big|_{Soln_1}$ is only 1.

Definition 3.9. Solution Graph or SG SG is a Directed Acyclic Graph (DAG) composed of the ordered pair (V, A). V is the set of vertices and $A \subset V \times V$, the set of directed arcs. Any $Soln$, $Soln_1$, can be represented by an SG, SG_1, such that:

- $V = \{TargetS, SourceS, ITermS\}$. And, for short, $V = \{T, S, I\}$. $ITermS = \{Term_i \in TermS | Term_i \notin (SourceS \cup TargetS) \wedge Term_i$ is $useful$ in $Soln_1\}$.
- $A = \{(v_i, v_j)\,|v_i, v_j \in V \wedge v_i \in Soln_1[v_j]\}$.

For Example 3.2 and $Soln_1$ of Eq. 3.2, SG_1 is shown in Fig. 3.3.

Note that from the A definition above and PS and $Soln$ definitions (Definitions 3.5 and 3.6, respectively), SG_1 is acyclic (i.e., no possible sequence of arcs can start from and end at the same vertex). The following functions are defined for each vertex, $v_i \in V$:

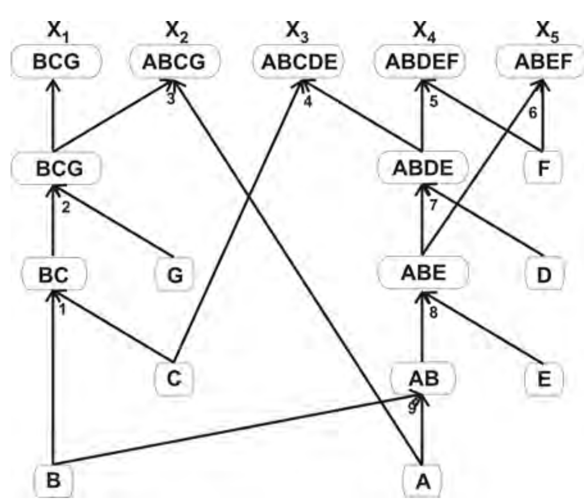


Figure 3.3: A *Solution* graph for Example 3.2 *Solution* of Eq. 3.2.

- $A_{in}(v_i) : V \rightarrow 2^A$. For each v_i, $A_{in}(v_i)$ returns the set of arcs that end at v_i. Formally: $A_{in}(v_i) = \{a_j = (v_j, v_i) | a_j \in A\}$.
- Similarly, $A_{out}(v_i) : V \rightarrow 2^A$. For each v_i, $A_{out}(v_i)$ returns the set of arcs that start at v_i. Formally: $A_{out}(v_i) = \{a_j = (v_i, v_j) | a_j \in A\}$.
- $nJ_2(v_i) : V \rightarrow \mathbb{N}$. A function that returns the number of 2-input joins constructing the $Term$ represented by vertex v_i in the $Solution$ represented by the graph. It is assumed in this work that an n-input join is implemented using $(n-1)J_2$s. Formally,

$$nJ_2(v_i) = \begin{cases} |A_{in}(v_i)| - 1 & |A_{in}(v_i)| \geq 1 \\ 0 & |A_{in}(v_i)| = 0 \end{cases} \tag{3.3}$$

- Similarly, $nF_2(v_i) : V \rightarrow \mathbb{N}$. A function that returns the number of 2-output forks immediately branching from the $Term$ represented by v_i. It is assumed in this work that an n-output fork is implemented using $(n-1)$ F_2s. Formally,

$$nF_2(v_i) = \begin{cases} |A_{out}(v_i)| - 1 & |A_{out}(v_i)| \geq 1 \\ 0 & |A_{out}(v_i)| = 0 \end{cases} \tag{3.4}$$

Definition 3.10. Cost A function that returns the number of 2-input joins (J_2s) required to implement a PS, a $Term$, or a $Soln$.

Formally, let PS_t be the PS of $Term$, $Term_t$, in $Soln$, $Soln_1$ (i.e., $Soln_1[Term_t] = PS_t$), then $Cost(Term_t)$ in $Soln_1$, $Cost(Term_t)\Big|_{Soln_1} : TermS \times SolnS \rightarrow \mathbb{N}$, is defined as follows:

$$Cost(Term_t)\Big|_{Soln_1} = |PS_t| - 1 + \sum_{i=1}^{|PS_t|} \frac{Cost(Term_i)\Big|_{Soln_1}}{nUsed[Term_i]\Big|_{Soln_1}} \tag{3.5}$$

where $Term_i \in PS_t \quad \forall i = 1, 2, \ldots |PS_t|$. $Cost(Term_t)\Big|_{Soln_1}$ and $Cost(PS_t)\Big|_{Soln_1}$ will be used interchangeably (since $Soln_1[Term_t] = PS_t$). Two factors contribute to $Cost(Term_t)$ in a $Solution$. First is the number of J_2s used to join the PS_t constituent terms. It is assumed in Eq. 3.5 that to implement an n-input join, $(n-1)J_2$s are required. The other factor is the $Cost$ of the constituent $Term$s themselves, taking into account how much these $Term$s are shared among other $Term$s in that $Solution$. The $Term$ sharing information is provided by the $nUsed$ vector. By definition, $\forall Term_i \in SourceS : Cost(Term_i) = 0$.

For Example 3.2 and SG_1 of Fig. 3.3, the chosen PS to construct $Term$ ABE is $\{AB, E\}$. $nUsed[AB] = 1$. Hence, $Cost(ABE) = 1 + Cost(AB)$. The chosen PS to construct $Term$ AB is $\{A, B\}$, and hence, $Cost(AB) = 1$. Therefore, $Cost(ABE)$ in $Soln_1$ is 2. Similarly, $Cost(ABDE) = 2$.

Similarly, the function $Cost(Soln_1) : SolnS \to \mathbb{N}$ is defined to return the total number of J_2s used to construct all $TargetS$ in $Soln_1$. Formally,

$$Cost(Soln_1) = \sum_{i=1}^{|TargetS|} Cost(Target_i) \tag{3.6}$$

where $Target_i \in TargetS \quad \forall i = 1, 2, \ldots |TargetS|$. For Example 3.2 and $Soln_1$ of Eq. 3.2 (or SG_1 of Fig. 3.3), five $Target$s exist, namely, BCG, $ABCG$, $ABCDE$, $ABDEF$, $ABEF$. The summation of the $Cost$s of these $Target$s in $Soln_1$ (i.e., $Cost(Soln_1)$) is 9.

Definition 3.11. OptCost The minimum $Cost$ among all $Solution\ Cost$s. Formally, $OptCost = \min_{i=1}^{|SolnS|} Cost(Soln_i)$.

The $Optimum\ Solution\ or\ OptSoln$ is defined to be a $Solution$ such that $Cost(OptSoln) = OptCost$. An $OptSoln$ may not be unique for a given problem, since multiple $Solution$s can have the same minimum $Cost$ among all $Solution$s. Hence, $OptSolnS$ is defined to be the set of all *optimum* $Solution$s.

Definition 3.12. Search Space or Space A $Space$ (designated as S_k) is a set of $Solution$s.

The (*whole*) search $Space$ (designated as S_o) is initialized with $SolnS$, and then refined throughout the algorithm until an $OptSoln$ is found.

Definition 3.13. Cone(Term) $Cone(Term_t)\Big|_{Soln_1} : TermS \times SolnS \to 2^{TermS}$, a function that returns the set of all $Term$s (down to $SourceS$) used in implementing $Term_t$ in $Soln_1$. Formally, let $Soln_1[Term_t] = PS_t$, then:

$$Cone(Term_t)\Big|_{Soln_1} = PS_t \bigcup_{i=1}^{|PS_t|} Cone(Term_i)\Big|_{Soln_1} \tag{3.7}$$

where $Term_i \in PS_t \quad \forall i = 1, 2, \ldots |PS_t|$.

By definition, $\forall Term_i \in SourceS : Cone(Term_i) = \emptyset$. For Example 3.2 and SG_1 of Fig. 3.3: $Cone(BCG) = \{BC, G, B, C\}$. Similarly, let PS' be a set of $Term$s (not necessarily a PS of any $Term$), then define $Cone(PS')\Big|_{Soln_1} : 2^{TermS} \times SolnS \to 2^{TermS}$ as follows:

$$Cone(PS')\Big|_{Soln_1} = PS' \bigcup_{i=1}^{|PS'|} Cone(Term_i)\Big|_{Soln_1} \tag{3.8}$$

where $Term_i \in PS' \quad \forall i = 1, 2, \ldots |PS'|$. Hence, if $Soln_1[Term_t] = PS_t$, then $Cone(Term_t)\Big|_{Soln_1}$ and $Cone(PS_t)\Big|_{Soln_1}$ will be used interchangeably.

Definition 3.14. Del operator - $\mathbf{Soln_1/D}$ The *Del* operator (/) accompanied by a *Del* set $D \subseteq TermS$ are applied to a *Solution*. Applied to $Soln_1$, it effectively removes all the $Terms$ in D from $Soln_1$. Formally,

$$Soln_1/D[Term_i] = \begin{cases} Soln_1[Term_i] & Term_i \notin D \\ \emptyset & Term_i \in D \end{cases} \tag{3.9}$$

Applying $/D$ on $Soln_1$ vector will also affect its associated data structures and functions (e.g., $nUsed$, $Cost$ and $Cone$). This will be denoted as, for example, $nUsed[Term_i]\Big|_{Soln_1/D}$. Some of the *useful* $Terms$ in $Soln_1$ can become unused (i.e., their $nUsed\Big|_{Soln_1/D} = 0$) as so some of the $Terms$ in their respective $Cones$. For Example 3.2 and SG_1 of Fig. 3.3, deleting $Term\ BCG$, will decrease $nUsed$ of the following $Terms$ by 1: BC (will become unused), G (will become unused), B, and C.

Definition 3.15. nAddedJoins or nAJ(Term) $nAJ(Term_i)\Big|_{Soln_1} : TermS \times SolnS \rightarrow \mathbb{N}$, a function that returns the number of J_2s that exist in $Soln_1$ just to construct $Term_i$ (i.e., the J_2s that, otherwise, would not be used if $Term_i$ was deleted from $Soln_1$). Formally, let $Soln_1[Term_t] = PS_t$, then:

$$nAJ(Term_t)\Big|_{Soln_1} = u_t\Big|_{Soln_1} \times \left(|PS_t| - 1 + \sum_{i=1}^{|Cone(Term_t)|} s_i\Big|_{Soln_1/\{Term_t\}} \times nAJ_o(Term_i) \right) \tag{3.10}$$

where $\forall i = 1, 2, \ldots |Cone(Term_t)| : Term_i \in Cone(Term_t)$, and:

$$nAJ_o(Term_i)\Big|_{Soln_1} = \begin{cases} |Soln_1[Term_i]| - 1 & Term_i \notin SourceS \\ 0 & Term_i \in SourceS \end{cases} \tag{3.11}$$

$$u_t\Big|_{Soln_1} \left(\text{or } u[Term_t]\Big|_{Soln_1} \right) = \begin{cases} 1 & Term_t \text{ is } \textit{useful} \text{ in } Soln_1 \\ 0 & Term_t \text{ is not } \textit{useful} \text{ in } Soln_1 \end{cases} \tag{3.12}$$

$$s_i\Big|_{Soln_1/\{Term_t\}} \left(\text{or } s[Term_i]\Big|_{Soln_1/\{Term_t\}} \right) = \begin{cases} 1 & nUsed[Term_i]\Big|_{Soln_1/\{Term_t\}} = 0 \\ 0 & nUsed[Term_i]\Big|_{Soln_1/\{Term_t\}} > 0 \end{cases} \tag{3.13}$$

Unless otherwise specified, nAJ will be calculated for *useful* $Terms$ only. Hence, $u[Term_t]$ (or interchangeably u_t) in Eq. 3.10 will be frequently omitted. Note the analogy between nAJ_o of Eq. 3.11 and $nJ_2(v_i)$ of Eq. 3.3. If $Term_i \in Cone(Term_t)\Big|_{Soln_1}$, then $nAJ_o(Term_i)\Big|_{Soln_1}$ contributes to $nAJ(Term_t)\Big|_{Soln_1}$ only if $Term_i$ is constructed in $Soln_1$

for the sole purpose of constructing $Term_t$ in $Soln_1$ (in other words, only if $Term_i$ would not be *useful* in $Soln_1$ if $Term_t$ was deleted from $Soln_1$). This information is provided through $s[Term_i]$ (or interchangeably s_i) defined in Eq. 3.13. $nAJ(Term_t)\Big|_{Soln_1}$ and $nAJ(PS_t)\Big|_{Soln_1}$ will be used interchangeably (since $Soln_1[Term_t] = PS_t$). As an example, let all the $Terms$ used by PS_t be already shared by other $Terms$ in $Soln_1$. In this case, all that is added to the network to construct PS_t are the J_2s required to join its constituent $Terms$ (i.e., $|PS_t| - 1$).

For Example 3.2 and SG_1 of Fig. 3.3, $nAJ_o(AB)\Big|_{Soln_1} = 1$ and $nUsed[AB]\Big|_{Soln_1/\{ABE\}} = 0$, therefore, $nAJ(ABE)\Big|_{Soln_1} = 2$. Although the $Cost$ of $ABDE$ is two, its nAJ is only one. The reason is, $Term\ ABE$ which is used to construct $ABDE$ in $Soln_1$ is also used in the $Solution$ to construct another $Term$ (i.e., $Term\ ABEF$). Hence, to construct $Term\ ABDE$, the only added J_2 to $Soln_1$ is the join required to join ABE with D.

3.2 The Algorithm

Lemma 3.1. *Let nJ_2 and nF_2 be the total number of J_2s and F_2s in a network, respectively. Then, the following equality holds for any $Solution \in SolnS$ (i.e., whatever the PS choices of the different $Terms$):*

$$nJ_2 - nF_2 = |SourceS| - |TargetS| \tag{3.14}$$

Proof. Construct a *Solution graph*, SG_1, of a *Solution*, $Soln_1$ (see Fig. 3.3, for example). Following Def. 3.9 of the SG, each arc starts at a vertex (i.e., a $Term$) and ends at a vertex (i.e., another term), therefore, the following equation holds:

$$\sum_{i=1}^{|V|} |A_{in}(v_i)| = \sum_{i=1}^{|V|} |A_{out}(v_i)| \tag{3.15}$$

By definition, $\forall v_i \in SourceS : |A_{in}(v_i)| = 0$, and $\forall v_i \in TargetS : |A_{out}(v_i)| = 0$. Hence, Eq. 3.15 is reduced to:

$$\sum_{j=1}^{|I|+|T|} |A_{in}(v_j)| = \sum_{j=1}^{|I|+|S|} |A_{out}(v_j)| \tag{3.16}$$

Since all SG_1 vertices represent *useful* $Terms$ in $Soln_1$ (see Def. 3.8), and since by the definition of *Solution* (Def. 3.6) all *useful* $Terms$ must be implemented using other $Terms$ (except $SourceS$), therefore, the following holds:

$$\forall v_i \in (ITermS \cup TargetS) : |A_{in}(v_i)| \geq 1 \tag{3.17}$$

$$\forall v_i \in (ITermS \cup SourceS) : |A_{out}(v_i)| \geq 1 \tag{3.18}$$

Hence, from Equations 3.3 and 3.4, Eq. 3.16 can be rewritten in terms of $nJ_2(v_j)$ and $nF_2(v_j)$, as follows:

$$\sum_{j=1}^{|I|+|T|} (nJ_2(v_j) + 1) = \sum_{j=1}^{|I|+|S|} (nF_2(v_j) + 1) \tag{3.19}$$

The total number of 2-input joins and 2-output forks in $Soln_1$ (i.e., nJ_2 and nF_2, respectively) can be computed as follows:

$$nJ_2 = \sum_{j=1}^{|I|+|T|} nJ_2(v_j) \tag{3.20}$$

$$nF_2 = \sum_{j=1}^{|I|+|S|} nF_2(v_j) \tag{3.21}$$

Substituting Equations 3.20 and 3.21 in Eq. 3.19 concludes the proof. ■

Theorem 3.2. *An algorithm that minimizes nJ_2 will also minimize nF_2 and also nJ_2+nF_2.*

Proof. The theorem follows directly from Lemma 3.1. ■

In other words, for some required communications in a control network, since an $OptSoln$ (Def. 3.11) utilizes the minimum number of J_2s, therefore, it will also incorporate the minimum total number of J_2s and F_2s.

3.2.1 Algorithm Overview

Theorem 3.2 narrows down the problem to: Construct the $TargetS$ from the $SourceS$ using a minimum total number of J_2s (i.e., find an $OptSoln$). The proposed algorithm consists of four main steps, covered in the following four subsections. Step I finds the candidate $Terms$ that can be used in an $OptSoln$. Then, for each of the candidate $Terms$, Step II finds the candidate PSs that may be used by an $OptSoln$. Step II uses a set of proven rules to identify (and exclude) PSs that are not needed to find an $OptSoln$. At this point, the search $Space$ of the problem consists of all the remaining possible PS choices of all the candidate $Terms$. Step III collects statistics about the search $Space$. Metrics computed include the max/min possible usage (or sharing) of the remaining $Terms$ in the search $Space$, from which the max/min possible nAJ value of each remaining PS can be computed. Based on these metrics, Step III eliminates expensive PSs from the search $Space$. The latter $Space$ reduction does in turn affect the $Space$ metrics, which in turn can lead to removing further expensive PSs. Hence, Step III through a number of iterations prune out the search $Space$ until no further reduction is possible, at which point the algorithm moves to Step IV. Choosing a certain PS for a $Term$ (and omitting the other PSs from the search $Space$) does affect the max/min possible usage of the constituent $Term$s of these PSs. This in turn can affect the max/min possible nAJ value of other PSs which use these $Term$s, providing opportunity for removing expensive PSs. Hence, Step IV makes use of this fact in case there are more than one $Solution$ still left in the search $Space$ after Step III. Step IV splits the remaining search $Space$ into multiple $Space$s, each with mutually exclusive PS choices for some $Term$s (called $STermS$). It then updates each sub-$Space$ metrics based on the specific PS choices made for that sub-$Space$, allowing for further reduction. The splitting continues until there is only one $Solution$ left in each sub-$Space$. The $Cost$ of each $Solution$ of each sub-$Space$ is calculated and compared. An $OptSoln$ is returned.

3.2.2 Step I: Construct the Potential $Term$s

The first step in the algorithm is to determine which $Term$s could be used to construct the $TargetS$ $Term$s and eliminate the rest.

Definition 3.16. Potential Terms or PTermS A set of $Term$s from which an $OptSoln$ can be constructed. Formally,

$$PTermS \cap TargetS = \phi \wedge$$
$$\exists OptSoln_i \in OptSolnS : (PTermS \cup TargetS) \supseteq UsefulTermS(OptSoln_i) \tag{3.22}$$

where $UsefulTermS$ function is defined in Def. 3.8.

Definition 3.17. Common Terms or CTermS

$$CTermS = \{Term_c \in (TermS - TargetS) | Term_c = Target_i \cap Target_j$$
$$\forall Target_i, Target_j \in TargetS, Target_i \neq Target_j\} \tag{3.23}$$

Following are the different methods used to construct the *potential* $Term$s ($PTermS$):

3.2.2.1 Method I: All Subsets of All $CTermS$ $Term$s

Define

$$PTermS_o^1 = \{Term_p \, | Term_p \subseteq Term_{ci} \quad \forall Term_{ci} \in CTermS\} \tag{3.24}$$

$$PTermS^1 = PTermS_o^1 \cup SourceS \tag{3.25}$$

Theorem 3.3. Potential Terms of Method I *$PTermS^1$ satisfies Def. 3.16 of the potential Terms (i.e., $\exists OptSoln_i \in OptSolnS : \left(PTermS^1 \cup TargetS\right) \supseteq UsefulTermS(OptSoln_i)$). Hence, an optimum Solution can be constructed by using only $Terms$ from $PTermS^1$.*

Proof. The proof relies on other theorems to be stated later in the text. The reader is advised to read the proof after finishing Sec. 3.2.4.

Define the function $FTargetS$(read *Father-TargetS*): $(TermS - TargetS) \rightarrow 2^{TargetS}$, as follows:

$$FTargetS(Term_i) = \{Target_j \in TargetS \,|Term_i \subseteq Target_j\}$$

$FTargetS(Term_i)$ returns the set of $Target$s that $Term_i$ can be used in their construction. Also, define the following $Term$ set:

$$\begin{aligned} UnSharedTermS = \{&Term_i \in (TermS - (TargetS \cup SourceS))| \\ &|FTargetS(Term_i)| = 1\} \end{aligned} \tag{3.26}$$

From $TermS$ definition in Eq. 3.1, $PTermS^1$ can be redefined as follows: $PTermS^1 = TermS - TargetS - UnSharedTermS$, and Theorem 3.3 can be rewritten as follows: An *optimum Solution* can be found without using the $Terms$ in $UnSharedTermS$.

The proof will be done by iteratively using Theorem 3.15 Rule V. It is easy to show that each $Term$ in $UnSharedTermS$ can *maximally* be used by *only* one $Target$ and zero or more other terms from $UnSharedTermS$. Define $UnSharedTermS_1$ to be the $Terms$ in $UnSharedTermS$ which are maximally used once (i.e., by one $Target$ and zero other $Terms$ from $UnSharedTermS$). Formally,

$$\begin{aligned} UnSharedTermS_1 = \{&Term_i \in UnSharedTermS| \\ &Term_i \subseteq Term_t \in TermS \Rightarrow Term_t \in TargetS\} \end{aligned} \tag{3.27}$$

Obviously, $\forall Term_i \in UnSharedTermS_1 : nUsedMax[Term_i] = 1$. Hence, by Theorem 3.15 Rule V, all $Terms$ in $UnSharedTermS_1$ can be omitted from the search $Space$ (i.e., an $OptSoln$ can be found without using them).

Similarly, define $UnSharedTermS_2$ to be the $Terms$ in $UnSharedTermS$ which are maximally used by only one $Target$ and one or more $Terms$ from $UnSharedTermS_1$:

$$\begin{aligned} UnSharedTermS_2 = \{&Term_i \in UnSharedTermS| \\ &Term_i \subseteq Term_t \in TermS \Rightarrow Term_t \in (TargetS \cup UnSharedTermS_1)\} \end{aligned} \tag{3.28}$$

Since the $Terms$ in $UnSharedTermS_1$ are omitted from the search $Space$, therefore, $\forall Term_i \in UnSharedTermS_2 : nUsedMax[Term_i] = 1$. Hence, by Theorem 3.15 Rule V, all $Terms$ in $UnSharedTermS_2$ can also be omitted from the search $Space$. The above iterations can be repeated until all $Terms$ in $UnSharedTermS$ are omitted from the search $Space$. Hence, an *optimum Solution* can be found without using any $Term$ from $UnSharedTermS$. That concludes the proof. ■

Method I includes in $PTermS^1$ all $CTermS$ $Terms$ as well as all their subsets. The number of *potential* $Terms$ will thus quickly increase as the number and sizes of $CTerms$ increase. This adversely affects the algorithm runtime. Hence, following are some methods that try to minimize the number of $PTerms$.

3.2.2.2 Method II: All Intersections and Differences of $CTermS$ $Terms$

This method initially populates $PTermS$ (will be referred to, in this method, as $PTermS^2$) with $CTermS$. It then considers the intersection of and the difference between any two $PTerms$ to be another $PTerm$. Formally, define $PTermS^2_o$ to be the *smallest* set (in cardinality) that satisfies the following two conditions:

1. $PTermS^2_o \supseteq CTermS$.
2. $\forall Term_{pi}, Term_{pj} \in PTermS^2_o : Term_{pi} - Term_{pj} \in PTermS^2_o \wedge Term_{pi} \cap Term_{pj} \in PTermS^2_o$.

$$PTermS^2 = PTermS^2_o \cup SourceS \tag{3.29}$$

It is easy to show that $PTermS^2 \subseteq PTermS^1$. A proof (or counter proof) that $PTermS^2$ satisfies the definition of $PTermS$ (Def. 3.16) could not be found. Hence, using Method II to construct $PTermS$, while typically incorporates less number of $Terms$, is not proved (or disproved) to result in an *optimum Solution* for all problems. Nonetheless, for all the examples where Method I and Method II ran to completion, Method II provided *optimum Solutions*.

3.2.2.3 Method III: Target Division

This method gives a label to each $Term \in TermS$. The label reflects whether, for each $Target$, all the $INodes$ (or $Sources$) joined by this $Term$ belong to that $Target$, or only part of them, or none of them. It then groups $Terms$ with similar label together. The biggest $Term$ (in cardinality) in each group is then included in $PTermS^3$. Non-$Source$ $Terms$ that cannot be used for constructing more than one $Target$ are excluded from $PTermS^3$ (since an $OptSoln$ can be found without using them according to the proof of Theorem 3.3). Formally, the $Label$ function ($L : (TermS - TargetS) \rightarrow \{0, 1, -\}^{|TargetS|}$) is defined as follows:

$$L(Term_t) = V_t \text{ such that } V_t[i] = \begin{cases} 1 & Term_t \cap Target_i = Term_t \\ 0 & Term_t \cap Target_i = \emptyset \\ - & \emptyset \subset Term_t \cap Target_i \subset Term_t \end{cases} \tag{3.30}$$

$\forall i = 1, 2, \ldots |TargetS|$.

Also define $nL(Term_t) : (TermS - TargetS) \rightarrow \mathbb{N}$ to be the number of $V_t[i] = 1, \forall i = \{1, \ldots, |V_t|\}$ where $V_t = L(Term_t)$. Define:

$$\begin{aligned} PTermS_o^3 =&\{Term_p \in (TermS - TargetS) | nL(Term_p) > 1 \wedge \\ &\forall Term_i \in (TermS - TargetS), Term_i \neq Term_p : \\ &L(Term_i) = L(Term_p) \Rightarrow Term_i \subset Term_p\} &(3.31) \\ PTermS^3 =&PTermS_o^3 \cup SourceS &(3.32) \end{aligned}$$

It is easy to show that $PTermS^3 \subseteq PTermS^2$. However, similar to $PTermS^2$, a proof (or counter proof) that $PTermS^3$ satisfies the definition of $PTermS$ (Def. 3.16) could not be found. Hence, using Method III to construct $PTermS$, while typically incorporates less number of $Terms$, is not proved (or disproved) to result in an *optimum Solution* for all problems. Nonetheless, in all the examples where Method I and Method III ran to completion, Method III provided *optimum Solutions*.

3.2.2.4 Method IV: All $CTermS$ Intersections

This method initially populates $PTermS^4$ with $CTermS$. It then considers *only* the intersection between any two $PTerms$ to be another $PTerm$. Formally, define $PTermS_o^4$ to be the *smallest* set (in cardinality) that satisfies the following two conditions:

1. $PTermS_o^4 \supseteq CTermS$.
2. $\forall Term_{pi}, Term_{pj} \in PTermS_o^4 : Term_{pi} \cap Term_{pj} \in PTermS_o^4$.

$$PTermS^4 = PTermS_o^4 \cup SourceS \tag{3.33}$$

It is easy to show that $PTermS^4 \subseteq PTermS^3$ and thus Method IV exhibits the shortest algorithm runtime among all the four methods. Nonetheless, counter examples showing that $PTermS^4$ may not satisfy the definition of $PTermS$ (Def. 3.16) in some cases do exist. Examples are explained in Sec. 3.2.6. Sec. 3.2.6 also provides some techniques to help check whether a *Solution* returned by the algorithm when using Method IV is indeed *optimum*. Possible correction techniques are explained as well.

The number of *potential Terms* provided by Step I is, at worst, exponential. In particular,

$$
\begin{aligned}
&PTermS^i \leq 2^{|SourceS|} - 1 \quad \forall i \in \{1, 2, 3, 4\} \\
&PTermS_o^4 \leq min\left(\left(2^{|SourceS|} - 1\right), \left(2^{|TargetS|} - |TargetS| - 1\right)\right)
\end{aligned} \tag{3.34}
$$

Nonetheless, in practice, the size of $PTermS$ is much smaller (see Table 3.2). The actual size depends on the overlapping between the different $Target$ set contents.

3.2.3 Step II: Construct the Partial Solutions

The search $Space$ (i.e., the possible $Solutions$), at this point, consists of all combinations of all possible PS choices of all $PTermS$. This step aims at excluding PSs that are not needed in an $OptSoln$. A cost metric is thus needed to differentiate between several PSs of the same $Term$ and to eliminate expensive PSs from the search $Space$. nAJ provides such a metric as shown in the following theorems:

Theorem 3.4. *Let $Soln_1$ and $Soln_2$ be two Solutions. Let also, $Soln_1/\{Term_t\} = Soln_2/\{Term_t\}$ (i.e., $\forall i = 1, 2, \ldots |TermS| \wedge i \neq t : Soln_1[Term_i] = Soln_2[Term_i]$), $Soln_1[Term_t] = PS_{t1}$, and $Soln_2[Term_t] = PS_{t2}$. Then, if $\left(nAJ(PS_{t1})\Big|_{Soln_1} \geq nAJ(PS_{t2})\Big|_{Soln_2}\right)$, then $Cost(Soln_1) \geq Cost(Soln_2)$. Greater and equal operators are ordered respectively.*

Proof. It follows from Def. 3.15 of nAJ that:

$$Cost(Soln_1) = Cost(Soln_1/\{Term_t\}) + nAJ(Term_t)\Big|_{Soln_1} \tag{3.35}$$

$$Cost(Soln_2) = Cost(Soln_2/\{Term_t\}) + nAJ(Term_t)\Big|_{Soln_2} \tag{3.36}$$

Since $Soln_1/Term_t = Soln_2/Term_t$, therefore, $Cost(Soln_1/Term_t) = Cost(Soln_2/Term_t)$. This concludes the proof. ■

Corollary 3.5. *Let PS_1 and PS_2 be two PSs of $Term_t$. Then, if for all possible combinations of other $Term$ PS choices $nAJ(PS_1) > nAJ(PS_2)$, then any $OptSoln$ will not use PS_1.*

Corollary 3.6. *Let PS_1 and PS_2 be two PSs of $Term_t$. Then, if for all possible combinations of other Term PS choices, $nAJ(PS_1) \geq nAJ(PS_2)$, then an $OptSoln$ can be found that does not use PS_1.*

Proof of both Corollaries 3.5 and 3.6 follows from Theorem 3.4 as well as Def. 3.11 of $OptSoln$.

It is easy to show that the $Cost$ function (Def. 3.10) *cannot* be used instead of nAJ in Theorem 3.4 to identify expensive PSs. In other words, let $Soln_1/\{Term_t\} = Soln_2/\{Term_t\}$, $Soln_1[Term_t] = PS_{t1}$, and $Soln_2[Term_t] = PS_{t2}$. Then, if $\left(Cost(PS_{t1})\Big|_{Soln_1} \geq Cost(PS_{t2})\Big|_{Soln_2}\right)$, then the following inequality does *not* necessarily hold: $Cost(Soln_1) \geq Cost(Soln_2)$.

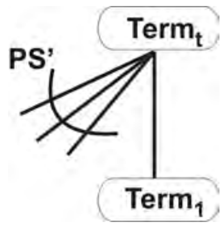
Following is a list of proven rules to be considered while constructing the $PTermS$ PSs. The rules help identify and exclude PSs that are not needed while searching for an $OptSoln$. Lemma 3.7 will be useful to prove the rules.

Lemma 3.7. *Use $s_i\Big|_{Soln_1}$ as in Eq. 3.13. Let $Term_1 \in Cone(Term_t)\Big|_{Soln_1}$. Then, if $s[Term_1]\Big|_{Soln_1/\{Term_t\}} = 0$, then, $s[Term_i]\Big|_{Soln_1/\{Term_t\}} = 0$ $\forall Term_i \in Cone(Term_1)\Big|_{Soln_1}$.*

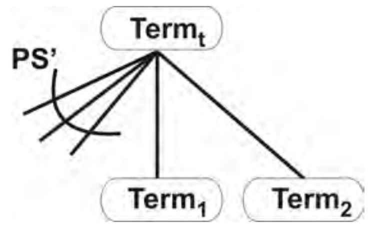
Proof. By s_i definition in Eq. 3.13, $s[Term_1]\Big|_{Soln_1/\{Term_t\}} = 0$ if $nUsed[Term_1]\Big|_{Soln_1/\{Term_t\}} > 0$. Hence, in the absence of $Term_t$ (i.e., $Soln_1/\{Term_t\}$) $Term_1$ is still used at least once. From Def. 3.7 of $nUsed$ and Def. 3.13 of $Cone$, it follows that all $Terms \in Cone(Term_1)\Big|_{Soln_1}$ will also still be used at least once in the absence of $Term_t$ (i.e., through $Term_1$). That concludes the proof. ■

Theorem 3.8. Rule I *Adding a whole redundant Term to a PS always causes it to be more expensive (in terms of nAJ). Formally, let $Term_t, Term_1, Term_2 \in TermS$, $Term_2 \subset Term_1 \subseteq Term_t$. Let PS_{t1} and PS_{t2} be two PSs of $Term_t$. Let both PS_1 and PS_2 be the same except that PS_1 contains $Term_1$, while PS_2 contains $Term_1$ and $Term_2$. Then, an optimum Solution will not use PS_{t2}.*

Proof. Let $Soln_1$ and $Soln_2$ be two $Solution$s such that: $Soln_1/\{Term_t\} = Soln_2/\{Term_t\}$, $Soln_1[Term_t] = PS_{t1}$, and $Soln_2[Term_t] = PS_{t2}$. Let PS' be the maximal common subset of PS_{t1} and PS_{t2}. Let also $|PS'| = n' \geq 0$. Following the theorem text (see Fig. 3.4):



(a) PS_{t1}



(b) PS_{t2}

Figure 3.4: Rule I.

$$\begin{aligned} PS_{t1} &= PS' \cup \{Term_1\} \\ PS_{t2} &= PS' \cup \{Term_1, Term_2\} \end{aligned} \tag{3.37}$$

From Def. 3.15 of nAJ:

$$\begin{aligned} nAJ(PS_{t1})\Big|_{Soln_1} &= C_1 \\ &+ s_1\Big|_{Soln_1/\{Term_t\}} \times nAJ_o(Term_1)\Big|_{Soln_1} \\ &+ \sum_{i=1}^{|Cone(Term_1)-Cone(PS')|} s_i\Big|_{Soln_1/\{Term_t\}} \times nAJ_o(Term_i)\Big|_{Soln_1} \end{aligned} \tag{3.38}$$

$$\begin{aligned} nAJ(PS_{t2})\Big|_{Soln_2} &= C_2 + 1 \\ &+ s_1\Big|_{Soln_2/\{Term_t\}} \times nAJ_o(Term_1)\Big|_{Soln_2} \\ &+ s_2\Big|_{Soln_2/\{Term_t\}} \times nAJ_o(Term_2)\Big|_{Soln_2} \\ &+ \sum_{i=1}^{|(Cone(Term_1)\cup Cone(Term_2))-Cone(PS')|} s_i\Big|_{Soln_2/\{Term_t\}} \times nAJ_o(Term_i)\Big|_{Soln_2} \end{aligned} \tag{3.39}$$

where C_l accounts for PS' contribution to $nAJ(PS_{tl})\Big|_{Soln_l}$ ($l \in \{1, 2\}$), as follows:

$$C_l = n' + \sum_{i=1}^{|Cone(PS')|} s_i\Big|_{Soln_l/\{Term_t\}} \times nAJ_o(Term_i)\Big|_{Soln_l} \tag{3.40}$$

Since $Soln_1/\{Term_t\} = Soln_2/\{Term_t\}$, it follows that $C_1 = C_2$. Therefore, $nAJ(PS_{t2})\Big|_{Soln_2} - nAJ(PS_{t1})\Big|_{Soln_1} \geq 1$. The proof then follows from Corollary 3.5.

■

Consider $Term$ $ABCG$. $PS_1 = \{A, BCG\}$ is always cheaper than $PS_2 = \{A, BCG, BC\}$. Hence, PS_2 should be excluded from the search $Space$.

Theorem 3.9. Rule II *Using a Term in a PS is always the same or cheaper (in terms of nAJ) than using all its constituent Terms. Formally, let* $Term_t, Term_c, Term_{a1}, \ldots Term_{an} \in TermS$, $Term_c \subseteq Term_t$, *and* $Term_c = \bigcup_{i=1}^{n} Term_{ai}$. *Let* PS_{t1} *and* PS_{t2} *be two* PSs *of* $Term_t$. *Let both* PS_{t1} *and* PS_{t2} *be the same except that* PS_{t2} *contains* $Term_c$, *while* PS_{t1} *instead contains Terms* $Term_{a1}, \ldots Term_{an}$. *Then, an* $OptSoln$ *can be found that does not use* PS_{t1}.

Proof. Informally, the idea behind the theorem is, if $Term_t$ needs a set of $Term$s in its implementation, then it hurts nothing to join these $Term$s in one $Term$ ($Term_c$) and use $Term_c$ instead. This is the same or cheaper than using the constituent $Term$s directly, since $Term_c$ may be used by other $Term$s and its $Cost$ will then be shared.

Formally, define $PS_{c1} = \{Term_{a1}, \ldots Term_{an}\}$. Let PS' be the maximal common subset of PS_{t1} and PS_{t2}. Let also $|PS'| = n' \geq 0$. Following the theorem text:

$$\begin{aligned} PS_{t1} &= PS' \cup PS_{c1} \\ PS_{t2} &= PS' \cup \{Term_c\} \end{aligned} \tag{3.41}$$

The theorem can be proved if it is proved that for each $Soln_1$ where $Soln_1[Term_t] = PS_{t1}$, there exists another $Soln_2$ such that $Soln_2[Term_t] = PS_{t2}$ and $Cost(Soln_2) \leq Cost(Soln_1)$. To prove the latter, it is sufficient to prove the following: For each $Soln_1$ where $Soln_1[Term_t] = PS_{t1}$, there exists another $Soln_2$ such that $Soln_2/\{Term_t, Term_c\} = Soln_1/\{Term_t, Term_c\}$, $Soln_2[Term_t] = PS_{t2}$ and $Cost(Soln_2) \leq Cost(Soln_1)$. The proof hereafter will be concerned with the last statement.

$Term_t$ and $Term_c$ may be referred to as T_t and T_c for brevity. Notice that the theorem does not specify a particular PS choice for $Term_c$. Hence, in general, if there are k PSs for $Term_c$ in the search $Space$ (call them PS_{c1}, PS_{c2}, $\ldots PS_{ck}$) then define the following two sets of $Solutions$:

$$\begin{aligned} Soln_1S = \{Soln_{1i} \,|&Soln_{1i}[Term_t] = PS_{t1} \wedge Soln_{1i}[Term_c] = PS_{ci} \\ &\wedge Soln_{1i}/\{T_c\} = Soln_{1j}/\{T_c\} \quad \forall Soln_{1i}, Soln_{1j} \in Soln_1S\} \end{aligned} \tag{3.42}$$

$$\begin{aligned} Soln_2S = \{Soln_{2i} \,|&Soln_{2i}[Term_t] = PS_{t2} \wedge Soln_{2i}[Term_c] = PS_{ci} \\ &\wedge Soln_{2i}/\{T_c\} = Soln_{2j}/\{T_c\} \quad \forall Soln_{2i}, Soln_{2j} \in Soln_2S\} \end{aligned} \tag{3.43}$$

Note that, by definition,

$$Soln_i/\{Term_t, Term_c\} = Soln_j/\{Term_t, Term_c\} \quad \forall Soln_i, Soln_j \in (Soln_1S \cup Soln_2S) \tag{3.44}$$

For illustration, and without loss of generality, three particular PS_{ci}s are shown in Fig. 3.5 when used in $Soln_1S$ and $Soln_2S$ $Solution$s. Note that $PS_{c2} \cap PS_{c1} = \emptyset$ and $\emptyset \subset PS_{c3} \cap PS_{c1} \subset PS_{c1}$.

The theorem can be proved (i.e., PS_{t1} can be omitted from the search $Space$) if the following statement can be proved (for all $Soln_1S$ and $Soln2S$ $Solution$s):

$$\exists Soln_{2i} \in Soln_2S : Cost(Soln_{2i}) \leq min_{j=1}^{|Soln_1S|} Cost(Soln_{1j}) \tag{3.45}$$

Informally, if a $Solution$ exists where PS_{t2} is used and which $Cost$ is the same or lower than all $Solution$s that use PS_{t1} instead, then PS_{t1} can be omitted from the search $Space$. The claim is $Soln_{21}$ does satisfy the above condition. To prove, extend Def. 3.15 of the $nAddedJoins$ to more than one $Term$ (namely, $Term_t$ and $Term_c$) and similar to Eq. 3.35, the following holds for any $Soln_i$:

$$Cost(Soln_i) = nAJ(Term_t, Term_c)\Big|_{Soln_i} + Cost(Soln_i/\{Term_t, Term_c\}) \tag{3.46}$$

From Eq. 3.44, it follows that, to prove the statement of 3.45, it suffices to prove the following:

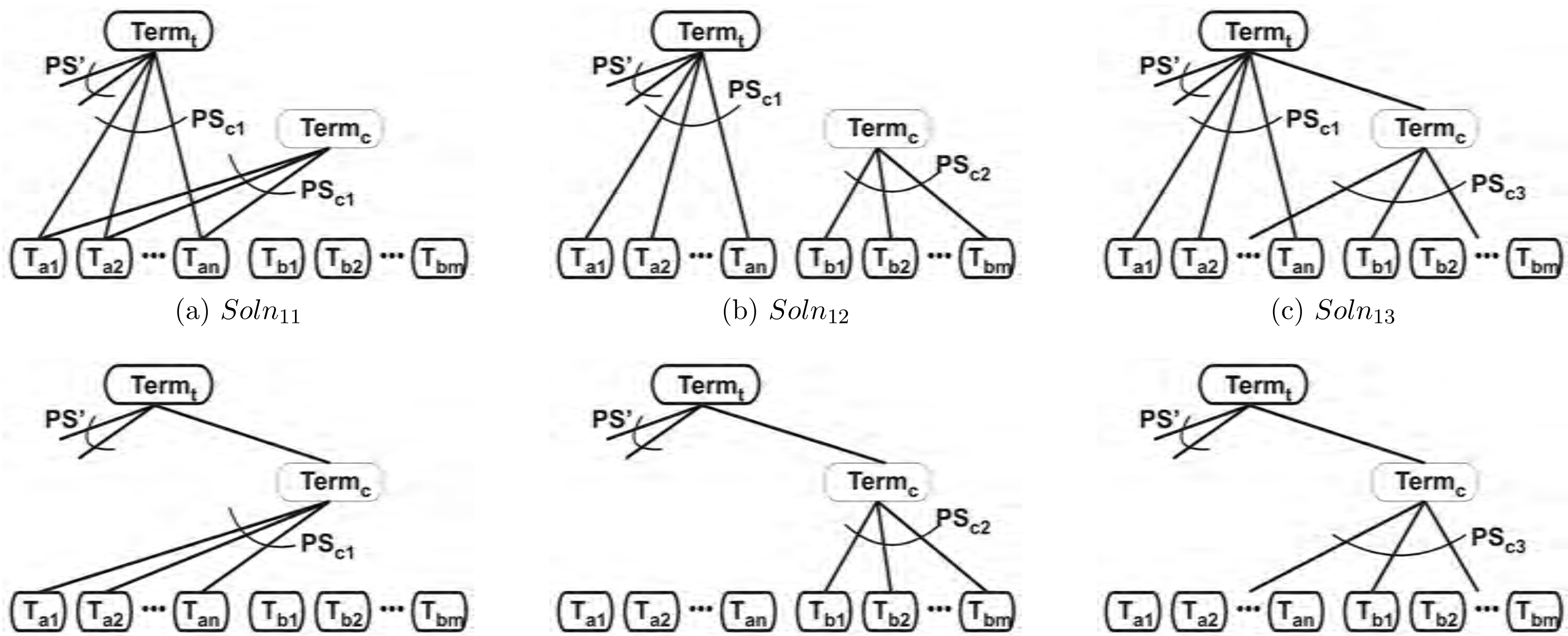
$$\exists Soln_{2i} \in Soln_2S : nAJ(Term_t, Term_c)\Big|_{Soln_{2i}} \leq min_{j=1}^{|Soln_1S|} nAJ(Term_t, Term_c)\Big|_{Soln_{1j}} \tag{3.47}$$

$nAJ(Term_t, Term_c)$ in the different $Soln_{1,2}S$ $Solution$s can be defined as follows (refer to Fig. 3.5):

$$\begin{aligned} nAJ(Term_t, Term_c)\Big|_{Soln_{1i}} &= C + n - 1 \\ &+ \sum_{j=1}^{|Cone(PS_{c1})-Cone(PS')|} s_j\Big|_{Soln_{1i}/\{T_t,T_c\}} \times nAJ_o(Term_j) \\ &+ u[Term_c] \times (|PS_{ci}| - 1) \\ &+ u[Term_c] \times \sum_{j=1}^{|Cone(PS_{ci})-(Cone(PS_{c1})\cup Cone(PS'))|} s_j\Big|_{Soln_{1i}/\{T_t,T_c\}} \times nAJ_o(Term_j) \end{aligned} \tag{3.48}$$

$$\begin{aligned} nAJ(Term_t, Term_c)\Big|_{Soln_{21}} &= C + n - 1 \\ &+ \sum_{j=1}^{|Cone(PS_{c1})-Cone(PS')|} s_j\Big|_{Soln_{21}/\{T_t,T_c\}} \times nAJ_o(Term_j) \end{aligned} \tag{3.49}$$

where u_c (or $u[Term_c]$), s_j (or $s[Term_j]$) and C are defined as in Equations 3.12, 3.13 and 3.40, respectively.



(a) $Soln_{11}$ (b) $Soln_{12}$ (c) $Soln_{13}$

(d) $Soln_{21}$ (e) $Soln_{22}$ (f) $Soln_{23}$

Figure 3.5: Rule II.

It is clear from Equations 3.48 and 3.49 that $Soln_{21}$ indeed meets the existential condition of 3.47. In particular,

$$nAJ(Term_t, Term_c)\Big|_{Soln_{21}} \leq min_{j=1}^{|Soln_1S|} nAJ(Term_t, Term_c)\Big|_{Soln_{1j}} \tag{3.50}$$

That concludes the proof. Note that in Equations 3.48 and 3.49, $u[Term_t]$ is implicitly set to one. In other words $Term_t$ is, and without loss of generality, assumed to be *useful* in all $Soln_1S$ and $Soln_2S$ *Solution*s. From Eq. 3.44, and from Def. 3.8 of *usefulness*, it is clear that if $Term_t$ is *useful* in one *Solution* in $Soln_1S \cup Soln_2S$, then it is also *useful* in all of them. Proving the theorem in case $Term_t$ is not *useful* is trivial. Since, in that case $Term_t$ has no effect on the *Cost* of the $Soln_{1,2}S$ *Solution*s. In other words,

$$\forall Soln_{1i} \in Soln_1S, Soln_{2i} \in Soln_2S : Cost(Soln_{1i}) = Cost(Soln_{2i})$$

which meets the existential condition of 3.45. ■

Consider *Term ABCG*. $PS_1 = \{A, BCG\}$ is always the same or cheaper than $PS_2 = \{A, BC, G\}$. Hence, PS_2 can be excluded from the search *Space*.

Theorem 3.10. Rule III *Using a Source in a PS is always the same or cheaper (in terms of nAJ) than any other non−Source Term. Formally, let* $Term_1, Term_2, Term_t \in TermS$, $Term_1 \in SourceS$, *and* $Term_2 \notin SourceS$. *Let also* $Term_1, Term_2 \subseteq Term_t$. *Let* PS_{t1} *and* PS_{t2} *be two PSs of* $Term_t$. *Let both* PS_{t1} *and* PS_{t2} *be the same except that* PS_{t1} *contains* $Term_1$, *while* PS_{t2} *contains* $Term_2$, *instead. Then, an OptSoln can be found that does not use* PS_{t2}.

Proof. Let $Soln_1$ and $Soln_2$ be two *Solution*s such that: $Soln_1/\{Term_t\} = Soln_2/\{Term_t\}$, $Soln_1[Term_t] = PS_{t1}$ and $Soln_2[Term_t] = PS_{t2}$. Let PS' be the maximal common subset of PS_{t1} and PS_{t2}. Let also $|PS'| = n' > 0$. Following the theorem text and Lemma 3.7:

$$\begin{aligned} PS_{t1} &= PS' \cup \{Term_1\} \\ PS_{t2} &= PS' \cup \{Term_2\} \end{aligned} \tag{3.51}$$

$$nAJ(PS_{t1})\Big|_{Soln_1} = C + s_1\Big|_{Soln_1/\{Term_t\}} \times nAJ_o(Term_1)\Big|_{Soln_1}$$
$$+ s_1\Big|_{Soln_1/\{Term_t\}} \times \sum_{i=1}^{|Cone(Term_1)-Cone(PS')|} s_i\Big|_{Soln_1/\{Term_t\}} \times nAJ_o(Term_i)\Big|_{Soln_1} \quad (3.52)$$
$$nAJ(PS_{t2})\Big|_{Soln_2} = C + s_2\Big|_{Soln_2/\{Term_t\}} \times nAJ_o(Term_2)\Big|_{Soln_2}$$
$$+ s_2\Big|_{Soln_2/\{Term_t\}} \times \sum_{i=1}^{|Cone(Term_2)-Cone(PS')|} s_i\Big|_{Soln_2/\{Term_t\}} \times nAJ_o(Term_i)\Big|_{Soln_2} \quad (3.53)$$

where C reflects the contribution of PS' to $nAJ(PS_{t1})\Big|_{Soln_1}$ (or equivalently to, $nAJ(PS_{t2})\Big|_{Soln_2}$), and computed as in Eq. 3.40. Since $Term_1$ is a *Source*, therefore, $nAJ_o(Term_1)\Big|_{Soln_1} = 0$ (Eq. 3.11). Also, $Cone(Term_1) - Cone(PS') = \emptyset$. Hence, $nAJ(PS_{t1})\Big|_{Soln_1} = C$. From which, $nAJ(PS_{t2})\Big|_{Soln_2} \geq nAJ(PS_{t1})\Big|_{Soln_1}$. The proof then follows from Corollary 3.6. ■

Consider $Term\ ABCG$ in Example 3.2. $PS_1 = \{BCG, A\}$ is always the same or cheaper than $PS_2 = \{BCG, AB\}$. Hence, PS_2 can be excluded from the search *Space*.

Definition 3.18. Target-image term or TITerm $Term_i$ is a $TITerm$ if

$$(Term_i \in PTermS) \wedge (\exists Target_j \in TargetS : Target_j = Term_i)$$

Also, define $TITermS$ to be the set of all *Target*-image *Term*s. For Example 3.2 and SG_1 of Fig. 3.3: $Term\ BCG$ (with $TermID$ of 8) is a $TITerm$, since it is an image of $Target\ BCG$ (with $TermID$ of 1) associated with $ONode\ X_1$.[3]

Theorem 3.11. Rule IV *Using a $TITerm$ in a PS is always the same or cheaper (in terms of nAJ) than any other $non-TITerm$. Formally, let $Term_1, Term_2, Term_t \in TermS$, $Term_1 \in TITermS$, and $Term_2 \notin TITermS$. Let also $Term_1, Term_2 \subset Term_t$. Let PS_{t1} and PS_{t2} be two PSs of $Term_t$. Let both PS_{t1} and PS_{t2} be the same except that PS_{t1} contains $Term_1$, while PS_{t2} contains $Term_2$, instead. Then, an OptSoln can be found that does not use PS_{t2}.*

Proof. Let $Soln_1$ and $Soln_2$ be two $Solution$s such that: $Soln_1/\{Term_t\} = Soln_2/\{Term_t\}$, $Soln_1[Term_t] = PS_{t1}$ and $Soln_2[Term_t] = PS_{t2}$. Following the theorem text, $nAJ(PS_{t1,2})$ can be expressed the same as in Equations 3.52 and 3.53 used in the proof of Theorem 3.10, respectively.

[3] $TermID$s are listed in Table 3.1.

Since $Term_1$ is a $TITerm$, then by Def. 3.18, $\exists Target_j \in TargetS : Target_j = Term_1$. Based on PS construction Rule II (i.e., Theorem 3.9), $Soln_1[Target_j] = Soln_2[Target_j] = \{Term_1\}$. Hence, $nUsed[Term_1]\Big|_{Soln_1} \geq 1$. It is realized from the Theorem text that $Target_j \neq Term_t$, and, therefore, $nUsed[Term_1]\Big|_{Soln_1/\{Term_t\}} \geq 1$. From s_i definition in Eq. 3.13, it follows $s_1\Big|_{Soln_1/\{Term_t\}} = 0$, and hence $nAJ(PS_{t1})\Big|_{Soln_1} = C$. From which, $nAJ(PS_{t2})\Big|_{Soln_2} \geq nAJ(PS_{t1})\Big|_{Soln_1}$. The proof then follows from Corollary 3.6. ■

Definition 3.19. AddedCoverage (or for short ACov) $ACov(Term_i, PS_t) : TermS \times 2^{TermS} \rightarrow 2^{INodeS}$. A function that returns the letters (i.e., $INode$s) covered by $Term_i \in PS_t$ and not covered by any other $Term$ in PS_t. Formally, $ACov(Term_i, PS_t) = Term_i - \bigcup_{j=1, j\neq i}^{|PS_t|} Term_j$.

Definition 3.20. Redundant PS PS_t is called a *redundant PS* if:

$$\exists Term_i \in PS_t : |ACov(Term_i, PS_t)| = 0 \vee (|ACov(Term_i, PS_t)| = 1 \wedge Term_i \notin SourceS)$$

Also, $Term_i$ will be called a *redundant* $Term$ in PS_t.

Corollary 3.12. *An OptSoln exists that does not use redundant PSs.*

Proof. The proof follows directly from Rules I and III (i.e., Theorems 3.8 and 3.10, respectively). ■

Algorithm 1 takes into account all the four rules while constructing the PSs. It takes five arguments:

- $Term_t$: the $Term$ to be constructed.
- $PSTerms$: the contents (thus far) of the PS being constructed.
- *Required*: a subset of $Term_t$, consisting of the $INode$s that have not yet been covered in the current PS. Initially, *Required* consists of all the $INodeS$ in $Term_t$.
- *RTermS* (or *Relevant Terms*): a set of $Term$s from which a PS of $Term_t$ can be built. *RTermS* are initialized with

$$\{Term_i \in PTermS \,|Term_i \subseteq Term_t \wedge Term_iID \neq Term_tID\}$$

By Def. 3.5 of PS, a $PTerm$ cannot be used to construct itself. Also, $Target$s cannot be used to construct any $Term$. Nonetheless, $Target$-image $Term$s (Def. 3.18) can construct their corresponding $Target$s.

- $ERTermS$ (or *Essential Relevant Terms*): a set initialized with $(SourceS \cup TITermS) \cap RTermS$.

Algorithm 1 runs (recursively) on each $Term_t \in (TargetS \cup PTermS)$. For each $Term_t$, it is initially called with $Required = Term_t$, $PSTermS = \emptyset$, and the appropriate $RTermS$ and $ERTermS$. PS and $PSTermS$ may be used interchangeably in the algorithm description.

Lines 1 - 15 check whether a single $Source$ or a single $TITerm$ exists that can cover all the letters (i.e., $INode$s) in *Required*. If this is the case, the $Source$ or the $TITerm$ is added to the current $PSTermS$, and the algorithm returns without further need to search for cheaper PSs (Rules III and IV).

If there is no single $Source$ or $TITerm$ that can cover all the letters in *Required*, the algorithm tries to cover them using all possible non-*redundant* combinations of the $Term$s in $RTermS$. First, Lines 17 - 20 check whether indeed a PS can be found using the current set of $RTermS$. If yes, the first $Term$ in $RTermS$ (call it $RTerm_i$) is picked and removed from $RTermS$. Lines 23 - 27 check whether adding $RTerm_i$ to the current PS will cause any redundancy (see Def. 3.20 of *redundant* PSs). If it causes redundancy, the next $RTerm$ is picked instead. If not, the algorithm will find all possible PSs in which $RTerm_i$ is used. To do that, the algorithm creates a new set of $Required_1$, $PSTermS_1$, and $RTermS_1$ structures that are modified copies of *Required*, $PSTermS$, and $RTermS$, respectively, based on the fact that $RTerm_i$ is used (Lines 28 - 30). If adding $RTerm_i$ to the current PS covers all the letters in *Required* (Line 31) then $PSTermS_1$ is a complete PS. The PS is stored (Line 32) and the algorithm picks the next $RTerm$. If $PSTermS_1$ is not yet complete (i.e., $Required_1$ is not empty), the algorithm iteratively calls $FindPSs$ (Line 36). However, adding $RTerm_i$ to $PSTermS_1$ typically renders *redundant* (Def. 3.20) some of the $Terms$ in $RTermS_1$. Hence, line 35 filters out such redundant $RTerms$ (and also applies Rule II) before iteratively calling $FindPSs$ algorithm.

As an upper bound, Algorithm 1 will have to visit all possible combinations of $RTermS$. Hence, its complexity is $O(2^{|RTermS|})$, and the number of PSs per $Term_t(\notin SourceS)$ is bounded by:

$$
\begin{aligned}
|PSS[Term_t]| &\leq 2^{|RTermS(of\ Term_t)|} - 1 \\
|RTermS(of\ Term_t)| &\leq 2^{|Term_t|} - 1
\end{aligned}
\tag{3.54}
$$

Nonetheless, in practice, the algorithm is much faster than (and the number of PSs is

Algorithm 1 FindPSs($Term_t$, $Required$, $RTermS$, $ERTermS$, $PSTermS$)

$ReuiredIsCoveredByAnETerm = 0$
for each $ERTerm_i \in ERTermS$ **do**
  if $ERTerm_i \supseteq Required$ **then** // *i.e., A Source or a* $TITerm$ *can cover Required - Rules III, IV*
    if Adding $ERTerm_i$ causes the PS to be redundant **then** // *Def. 3.20*
      return
    end if
    $ReuiredIsCoveredByAnETerm = 1$
    $CoveringERTerm = ERTerm_i$
  end if
end for
if $ReuiredIsCoveredByAnETerm$ **then**
  $PSTermS = PSTermS \cup CoveringERTerm$
  AddThisPS ($PSTermS$,$Term_t$)
  return
end if
while $|RTermS| > 0$ **do**
  $RTermSUnion = \bigcup_{i=1}^{|RTermS|} RTerm_i$
  if $RTermSUnion \not\supseteq Required$ **then** // *A PS cannot be constructed from the remaining RTermS*
    return
  end if
  Take and remove the first $Term$ from $RTermS$, $RTerm_i$
  $ERTermS = ERTermS - RTerm_i$
  $ACov = RTerm_i \cap Required$
  if $|ACov| > 1 \vee (|ACov| = 1 \wedge |Required| = 1)$ **then**
    if Adding $RTerm_i$ causes the PS to be redundant **then**
      continue
    end if
    $Required_1 = Required - RTerm_i$
    $PSTermS_1 = PSTermS \cup RTerm_i$
    $RTermS_1 = RTermS$
    if $|Required_1| = 0$ **then** // *i.e., all letters covered*
      AddThisPS ($PSTermS_1$,$Term_t$)
      continue
    end if
    Filter $RTermS_1$ because of adding $RTerm_i$
    FindPSs($Term_t$, $Required_1$, $RTermS_1$, $ERTermS$, $PSTermS_1$)
  end if
end while
return

much less than) exponential. This is because not all $RTerm$ combinations are PSs. Also, applying Rules I, II, III, and IV as well as $RTermSUnion$ check (in Line 18) eliminate substantial part of the $RTerm$ combinations. Table 3.3 shows the reduction in the search $Space$ due to applying the four rules of Step II for sample problems. For Example 3.2, the PSs computed by Algorithm 1 are listed in Table 3.1.

3.2.4 Step III: Collect $Space$ Metrics and Remove Higher nAJ Partial Solutions

Theorem 3.3 narrowed down the search $Space$ by confining the number of candidate $Term$s. Furthermore, Theorems 3.4 through 3.12 reduced their possible corresponding PSs. At this point the search $Space$ of the problem consists of all the remaining possible PS choices of all the candidate $Term$s. This step aims at further pruning out the search $Space$ by computing the different PS upper and lower bound nAJ values and eliminating expensive PSs. The value of $nAJ(PS_t)$ is $Solution$-dependent (e.g., a $Solution$ that provides sharing to the constituent $Term$s of PS_t will reduce its nAJ, and vice versa). Nonetheless, through calculating the maximum and minimum possible sharing (in any $Solution$ in the search $Space$) of the PS_t constituent $Term$s (called $nUsedMax[Term_i]$ and $nUsedMin[Term_i]$, respectively), the lower and upper bounds of $nAJ(PS_t)$ (called, $nAJMin(PS_t)$ and $nAJMax(PS_t)$, respectively) can be computed. Comparing such bounds of different PSs, some PSs can be found too expensive and thus omitted from the search $Space$. This step is iterative. Omitting some $Term$ PSs can affect the max/min usage (sharing) of the $Term$s constituting these PSs. This, in turn, affects the nAJ lower/upper bounds of other PSs that use these $Term$s, allowing for further reduction. At the end of each iteration, more areas of the search $Space$ can be eliminated. When the algorithm can do no more eliminations, it goes to the next step.

Following are the definitions of the basic data structures and functions associated with the search $Space$ (also referred to as *metrics*):

Definition 3.21. PSS $PSS[Term_t]\Big|_{S_k}$ is the set of $Term_t$ PSs in the search $Space$, S_k.

Definition 3.22. Usable Term A $Term$ is *usable* in a search $Space$ if it is *useful* (Def. 3.8) in at least one $Solution$ in that $Space$. Formally, $Term_i$ is *usable* in search $Space$ S_k if:

$$\exists Soln_i \in S_k : Term_i \in UsefulTermS(Soln_i)$$

Definition 3.23. nUsedMax A vector of numbers where $TermID$s are used as indices. $nUsedMax[Term_i]\Big|_{S_k}$ provides an upper bound on the maximum possible sharing of $Term_i$ in any $Solution$ in the search $Space$, S_k. Formally,

$$nUsedMax[Term_i]\Big|_{S_k} \geq \max_{j=1}^{|S_k|} nUsed[Term_i]\Big|_{Soln_j} \forall Soln_j \in S_k$$

$nUsedMax[Term_i]\Big|_{S_k}$ is recursively defined as the number of $Term_t$s in the search $Space$, S_k, that satisfy the following two conditions:

1. $\exists PS_t \in PSS[Term_t]\Big|_{S_k} : Term_i \in PS_t$.
2. $Term_t$ is *usable* in S_k.

Table 3.1 shows the initial values of $nUsedMax$ of different $Term$s in Example 3.2. At the end of each iteration, some PSs are omitted from the search $Space$, and hence, the value of $nUsedMax$ of some $Term$s will decrease.

Definition 3.24. Essential Term or ETerm $Term_t$ is an *essential* $Term$ in a search *Space* if it is *useful* in all that *Space Solution*s. Formally, $Term_i$ is an $ETerm\Big|_{S_k}$ if

$$\forall Soln_i \in S_k : Term_i \in UsefulTermS(Soln_i)$$

All $Target$s are $ETerm$s in all $Space$s. $ETermS\Big|_{S_k}$ is defined to be the set of all $ETerm$s in $Space$ S_k.

Definition 3.25. Essential Child or EChild $Term_i$ is said to be an *essential* child of $Term_t$ in search $Space$ S_k iff all the following conditions are satisfied:

1. $\forall PS_t \in PSS[Term_t]\Big|_{S_k} : Term_i \in PS_t$.
2. $Term_t$ is *usable* in S_k.

Also, define $EChildren[Term_t]\Big|_{S_k}$ to be all $EChild$ $Term$s of $Term_t$ in search $Space$ S_k.

Definition 3.26. nUsedMin A vector of numbers where $TermID$s are used as indices. $nUsedMin[Term_i]\Big|_{S_k}$ provides a lower bound on the minimum possible sharing of $Term_i$ in any $Solution$ in the search $Space$, S_k. Formally,

$$nUsedMin[Term_i]\Big|_{S_k} \leq \min_{j=1}^{|S_k|} nUsed[Term_i]\Big|_{Soln_j} \forall Soln_j \in S_k$$

$nUsedMin[Term_i]\Big|_{S_k}$ is recursively defined as the number of $Term_t$s in the search $Space$, S_k, that satisfy the following two conditions:

1. $Term_i$ is an $EChild$ of $Term_t$ in S_k.
2. $Term_t$ is an $ETerm$ in S_k.

The calculation of $nUsedMin$ in a search $Space$ starts by the fact that all $Target$s are *essential* $Term$s ($ETerm$s) in any search $Space$. Propagation of *essentiality* then takes place. If $Term_t$ is an $ETerm$, then all its $EChildren$ will also be $ETerm$s (increasing their $nUsedMin$ by 1).

Table 3.1 shows the initial values of $nUsedMin$ of different $Term$s in Example 3.2. At the end of each iteration, more PSs are omitted and more $Term$s become $ETerm$s, and hence, their $nUsedMin$ increase.

Definition 3.27. nAJMax(PS) $nAJMax(PS_t)\Big|_{S_k}$ is an upper bound on the maximum value of $nAJ(PS_t)$ in all $Solution$s of the search $Space$ S_k. Formally, $nAJMax(PS_t)\Big|_{S_k} \geq \max_{j=1}^{|S_k|} nAJ(PS_t)\Big|_{Soln_j}$.

$nAJ(PS_t)$ is maximized in a $Solution$ when the $Solution$ provides minimum sharing to the constituent $Term$s of PS_t. Calculation of the exact maximum value of $nAJ(PS_t)$ in all $Solution$s of a given search $Space$ can be computation expensive. On the other extreme, a very conservative approximation for the upper bound can be easily computed but will provide too little selectivity (i.e., to find and omit expensive PSs). Between these two extremes, $nAJMax(PS_t)\Big|_{S_k}$ can be computed as follows. Let PS_{t1} be a PS of $Term_t$, then:

$$\begin{aligned} nAJMax(PS_{t1})\Big|_{S_k} = &|PS_{t1}| - 1 + \\ &\sum_{i=1}^{|PS_{t1}|} s_i\Big|_{max,S_k} \times nAJMax_o(Term_i)\Big|_{S_k} \end{aligned} \tag{3.55}$$

$$\begin{aligned} nAJMax_o(PS_{t1})\Big|_{S_k} = &|PS_{t1}| - 1 + \\ &\sum_{i=1}^{|PS_{t1}|} nAJMax_o(Term_i)\Big|_{S_k} \end{aligned} \tag{3.56}$$

$$nAJMax_o(Term_t)\Big|_{S_k} = \max_{i=1}^{\left|PSS[Term_t]\Big|_{S_k}\right|} nAJMax_o(PS_{ti})\Big|_{S_k} \tag{3.57}$$

where

$$s_i\Big|_{max,S_k} = \begin{cases} 1 & nUsedMin[Term_i]\Big|_{S_k/\{Term_t\}} = 0 \\ 0 & nUsedMin[Term_i]\Big|_{S_k/\{Term_t\}} > 0 \end{cases} \tag{3.58}$$

where $nUsedMin[Term_i]\Big|_{S_k/\{Term_t\}}$ = the number of $Term_f$s (where $Term_f \neq Term_t$) that satisfy the following two conditions:

1. $Term_i$ is an $EChild$ of $Term_f$ in S_k.
2. $Term_f$ is an $ETerm$ in S_k.

Note that the above definition of $nAJMax(PS_t)\Big|_{S_k}$ will provide a value that is the same or greater than the exact maximum value of $nAJ(PS_t)$ in all $Solution$s of S_k.

Definition 3.28. nAJMin(PS) $nAJMin(PS_t)\Big|_{S_k}$ is a lower bound on the minimum value of $nAJ(PS_t)$ in all $Solution$s of the search $Space$ S_k. Formally, $nAJMin(PS_t)\Big|_{S_k} \leq \min_{j=1}^{|S_k|} nAJ(PS_t)\Big|_{Soln_j}$.

$nAJ(PS_t)$ is minimized in a $Solution$ when the $Solution$ provides maximum sharing to the constituent $Term$s of PS_t. $nAJMin(PS_t)\Big|_{S_k}$ can be computed as follows. Let PS_{t1} be a PS of $Term_t$, then:

$$nAJMin(PS_{t1})\Big|_{S_k} = |PS_{t1}| - 1 + \sum_{i=1}^{|PS_{t1}|} s_i\Big|_{min,S_k} \times nAJMin_o(Term_i)\Big|_{S_k} \tag{3.59}$$

$$nAJMin_o(PS_{t1})\Big|_{S_k} = |PS_{t1}| - 1 \tag{3.60}$$

$$nAJMin_o(Term_t)\Big|_{S_k} = \min_{i=1}^{\left|PSS[Term_t]\Big|_{S_k}\right|} nAJMin_o(PS_{ti})\Big|_{S_k} \tag{3.61}$$

where

$$s_i\Big|_{min,S_k} = \begin{cases} 1 & nUsedMax[Term_i]\Big|_{S_k/\{Term_t\}} = 0 \\ 0 & nUsedMax[Term_i]\Big|_{S_k/\{Term_t\}} > 0 \end{cases} \tag{3.62}$$

where $nUsedMax[Term_i]\Big|_{S_k/\{Term_t\}}$ = the number of $Term_f$s (where $Term_f \neq Term_t$) that satisfy the following two conditions:

1. $\exists PS_f \in PSS[Term_f]\Big|_{S_k} : Term_i \in PS_f$.
2. $Term_f$ is *usable* in S_k.

Note that the above definition of $nAJMin(PS_t)\Big|_{S_k}$ will provide a value that is the same or less than the exact minimum value of $nAJ(PS_t)$ in all $Solution$s of S_k.

More restricted conditions, yet easier to check than those of Corollaries 3.5 and 3.6 are stated in the following corollaries:

Corollary 3.13. *Let* PS_1 *and* PS_2 *be two* PSs *of* $Term_t$ *in Space* S_k*. Then, if* $nAJMin(PS_1) > nAJMax(PS_2)$*, then any* $OptSoln$ *will not use* PS_1*.*

Corollary 3.14. *Let* PS_1 *and* PS_2 *be two* PSs *of* $Term_t$ *in Space* S_k*. Then, if* $nAJMin(PS_1) \geq nAJMax(PS_2)$*, then an* $OptSoln$ *can be found that doesn't use* PS_1*.*

Theorem 3.15. Rule V *A* $Term$ *that is used at most once in any Solution of a given search Space can be omitted from that search Space. Formally, if* $nUsedMax[Term_c]\Big|_{S_k} = 1$*, then an* $OptSoln$ *can be found without using* $Term_c$*.*

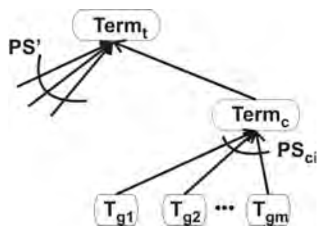
Proof. The proof is a special case of Rule II (Theorem 3.9). Informally, the idea behind the theorem is, if $Term_t$ is the only $Term$ (remaining) in the search $Space$ that may need a certain set of $Terms$ in its implementation, then it saves nothing to join these $Terms$ in one $Term$ ($Term_c$) and use $Term_c$ instead. It saves nothing because $Term_c$ is not shared with any other $Term$.

Formally, let $Term_t$ be the only $Term$ in S_k that *may* use $Term_c$ (note that $nUsedMax[Term_c]\Big|_{S_k} = 1$). Without loss of generality, define PS_{t1} to represent the form of any PS of $Term_t$ that uses $Term_c$, as follows:

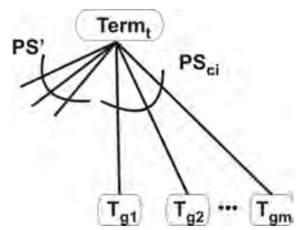
$$PS_{t1} = PS' \cup \{Term_c\} \tag{3.63}$$

The theorem can be proved if it is proved that for each $Soln_1$ where $Soln_1[Term_t] = PS_{t1}$, there exists another $Soln_2$ such that $Soln_2[Term_t] = PS_{t2}$ where $Term_c \notin PS_{t2}$ and $Cost(Soln_2) = Cost(Soln_1)$. To prove the latter statement, it is sufficient to prove the following: For each $Soln_1$ where $Soln_1[Term_t] = PS_{t1}$, there exists another $Soln_2$ such that $Soln_2/\{Term_t\} = Soln_1/\{Term_t\}$, $Soln_2[Term_t] = PS_{t2} = PS' \cup PS_{ci}$ (where $Soln_1[Term_c] = PS_{ci}$), and $Cost(Soln_2) = Cost(Soln_1)$. The proof hereafter will be concerned with the last statement. PS_{t1} and PS_{t2} are depicted in Fig. 3.6 (note that $Term_c$ is not *useful* in $Soln_2$). From Def. 3.15 of nAJ:

$$\begin{aligned} nAJ(PS_{t1})\Big|_{Soln_1} &= C + s_c\Big|_{Soln_1/\{Term_t\}} \times nAJ_o(Term_c) \\ &+ \sum_{i=1}^{|Cone(Term_c)-Cone(PS')|} s_i\Big|_{Soln_1/\{Term_t\}} \times nAJ_o(Term_i) \end{aligned} \tag{3.64}$$



(a) PS_{t1}



(b) PS_{t2}

Figure 3.6: Rule V.

where C reflects the contribution of PS' to $nAJ(PS_{t1})\Big|_{Soln_1}$ (or equivalently to, $nAJ(PS_{t2})\Big|_{Soln_2}$), and is computed as in Eq. 3.40. From Lemma 3.7, definition of nAJ_o in Eq. 3.11 and Def. 3.13 of $Cone$, it follows:

$$\begin{aligned} nAJ(PS_{t1})\Big|_{Soln_1} &= C + s_c\Big|_{Soln_1/\{Term_t\}} \times (|PS_{ci}| - 1) \\ &+ s_c\Big|_{Soln_1/\{Term_t\}} \times \sum_{i=1}^{|Cone(PS_{ci})-Cone(PS')|} s_i\Big|_{Soln_1/\{Term_t\}} \times nAJ_o(Term_i) \end{aligned} \quad (3.65)$$

$$\begin{aligned} nAJ(PS_{t2})\Big|_{Soln_2} &= C + (|PS_{ci}| - 1) \\ &+ \sum_{i=1}^{|Cone(PS_{ci})-Cone(PS')|} s_i\Big|_{Soln_2/\{Term_t\}} \times nAJ_o(Term_i) \end{aligned} \quad (3.66)$$

Since $s_c\Big|_{Soln_1/\{Term_t\}} = 1$, therefore, $nAJ(PS_{t2})\Big|_{Soln_2} = nAJ(PS_{t1})\Big|_{Soln_1}$. That concludes the proof. ■

Definition 3.29. Rule V Transformation

Let $nUsedMax[Term_c]\Big|_{S_k} = 1$ and $Term_t$ be the only $Term$ in S_k that *may* use $Term_c$. Define $OldPSS \subset PSS[Term_t]\Big|_{S_k}$ to be the set of all $Term_t$ PSs remaining in S_k that use $Term_c$. Formally,

$$OldPSS = \left\{ PS_{ti} \,\middle|\, PS_{ti} \in PSS[Term_t]\Big|_{S_k} \wedge Term_c \in PS_{ti} \right\} \quad (3.67)$$

Let $PSS[Term_c]\Big|_{S_k} = \{PS_{c1}, \ldots, PS_{cn}\}$. Then, the following transformation will be referred to as Rule V transformation: Replace each $PS_{ti} \in OldPSS$ with n PSs (PS_{ti1}, …, PS_{tin}), where PS_{tij} is defined as follows: If

$$PS_{ti} = PS' \cup \{Term_c\}$$

then,

$$PS_{tij} = PS' \cup PS_{cj}$$

The transformation has the potential of rendering many PS_{tij}s *redundant* (see Def. 3.20), and thus will be omitted from the search $Space$. This, in turn, updates the $nUsedMin$ and $nUsedMax$ structures of these PS constituent $Term$s. Hence, the transformation can result in affecting $nAJMin$ and $nAJMax$ of other PSs that are using these $Term$s allowing for more $Space$ reduction using Corollary 3.14.

Algorithm 2 iteratively collects and updates the search $Space$ metrics. It makes use of Corollaries 3.13 and 3.14 and Rule V (Theorem 3.15) and its transformation (Def. 3.29) to refine the search $Space$. It incorporates the following data structures:

- $nAJMin_o/Max_o[Term_t]\Big|_{S_k}$: a vector that stores $nAJMin_o/Max_o$ of all $Term_t$ in search $Space$ S_k, respectively.
- $PSnAJMin/Max[Term_t][PS_{ti}]\Big|_{S_k}$ *and* $PSnAJMin_o/Max_o[Term_t][PS_{ti}]\Big|_{S_k}$: two two dimensional structures that store $nAJMin/Max$ and $nAJMin_o/Max_o$ of all PS_{ti} of all $Term_t$ in search $Space$ S_k, respectively.
- $UT\Big|_{S_k}$: a set of $Term$s whose (or whose PS) $nAJMin_{(o)}/Max_{(o)}$ need to be updated. The $Term$s are ordered within the set by their cardinalities starting from the largest to the smallest. UT is initialized with $(TargetS \cup PTermS - SourceS)$.
- $UPSMin/Max[Term_t]\Big|_{S_k}$: a set of PSs of $Term_t$ whose $nAJMin_{(o)}/Max_{(o)}$ need to be updated, respectively. They are initialized with $PSS[Term_t]\Big|_{S_k}$.
- $PSR\Big|_{S_k}$: a set of PSs that are scheduled to be removed from the search $Space$, S_k.

At this point, the current search $Space$ consists of all the remaining possible PS choices of $(TargetS \cup PTermS)$. The suffix $\Big|_{S_k}$ will be omitted in Algorithm 2, since it is implied that all data structures and functions are calculated for the current search $Space$.

Algorithm 2 starts with UT initialized with $(TargetS \cup PTermS - SourceS)$. Line 2 picks the smallest $Term$ in UT, $Term_t$. Lines 4 to 7 check whether $Term_t$ is used only once in the search $Space$ and, if this is the case, apply Rule V transformation. The procedure in Line 5 also updates UT and $UPSMin/Max$ with the $Term$s and PSs (respectively) whose nAJ need to be updated in a next iteration due to the transformation. Lines 8 and 9 store the old values of $Term_t$ $nAJMax_o, nAJMin_o$, and $EChildren$ before doing any update. Lines 10 through 13 (Lines 14 through 17) update $nAJMin_{(o)}(nAJMax_{(o)})$ of the PSs of $Term_t$ specified in $UPSMin[Term_t](UPSMax[Term_t])$, respectively. Lines 18

Algorithm 2 Collect *Space* Metrics and Remove Higher nAJ Partial Solutions

```
while |UT| ≥ 1 do
  Get and remove the last element in UT, Term_t
  if nUsedMax[Term_t] ≥ 1 then // Term_t is usable
    if nUsedMax[Term_t] = 1 then
      Apply Rule V transformation
      continue
    end if
    OldnAJMin_o/Max_o = nAJMin_o/Max_o[Term_t]
    OEChildren = EChildren[Term_t]
    for each PS_ti in UPSMin[Term_t] do
      Update PSnAJMin[Term_t][PS_ti] and PSnAJMin_o[Term_t][PS_ti]
      Remove PS_ti from UPSMin[Term_t]
    end for
    for each PS_ti in UPSMax[Term_t] do
      Update PSnAJMax[Term_t][PS_ti] and PSnAJMax_o[Term_t][PS_ti]
      Remove PS_ti from UPSMax[Term_t]
    end for
    for all PS_ti and PS_tj of Term_t do
      if PSnAJMin[Term_t][PS_ti] ≥ PSnAJMax[Term_t][PS_tj] then
        PSR.insert(PS_ti)
      end if
    end for
    if |PSR| ≥ 1 then // Some PSs are to be removed
      Remove PSs And Update nUsedMax
      if nUsedMin[Term_t] ≥ 1 then // ETerm
        NEChildren = EChildren[Term_t] − OEChildren
        Update nUsedMin Because Of NEChildren
      end if
    end if
    Calculate and store NewnAJMin_o/Max_o of Term_t
    Compare them with OldnAJMin_o/Max_o respectively
    if NewnAJMax_o ≠ OldnAJMax_o then
      Determine which PSs (of other Terms) whose nAJMax need to be updated.
      Update UT and UPSMax accordingly
    end if
    if NewnAJMin_o ≠ OldnAJMin_o then
      Determine which PSs (of other Terms) whose nAJMin need to be updated.
      Update UT and UPSMin accordingly
    end if
  end if
end while
return
```

through 22 apply Corollary 3.14 to prune out expensive PSs. PSs to be removed are stored in PSR. The procedure of Line 24 propagates the effect of removing a PS, PS_t, of $Term_t$ to $nUsedMax$ of some (or all) of PS_t constituent $Term$s (and possibly their corresponding constituent $Term$s as well - see Def. 3.23 of $nUsedMax$). This in turn can affect $nAJMin_{(o)}$ of other PSs that use these $Term$s. The affected $Term$s and PSs are added to UT, and $UPSMin$, respectively, so that they are updated in a following iteration of the algorithm. Removing PSs from $Term_t$ may not only affect $nUsedMax$ of the constituting $Term$s, but also may add to $EChildren[Term_t]$. If $Term_t$ is an $ETerm$, and it gained new $EChildren$ in this iteration, then its new $EChildren$ will also become $ETerm$s. This is handled in Lines 25 through 28 of Algorithm 2. The procedure of Line 27 propagates the effect of *essentiality* to the $nUsedMin$ of the new $EChildren$ of $Term_t$ (and of their corresponding $EChildren$ as well - see Def. 3.26 of $nUsedMin$). This, in turn, can affect $nAJMax_{(o)}$ of other PSs that use these $Term$s. Again, the affected $Term$s and PSs are added to UT, and $UPSMax$, respectively, so that they are updated in a future iteration. The final part of Algorithm 2 (i.e., Lines 30 through 39) checks if any change has occurred to the values of $nAJMax_o$ and $nAJMin_o$ of $Term_t$. If so, it determines which $Term$s and PSs (that use $Term_t$) are affected by these changes. UT, $UPSMax$ and $UPSMin$ are updated accordingly. Algorithm 2 will continue to iterate until UT is empty (i.e., no more $Term$s need to be updated).

3.2.5 Step IV: Divide, Refine the Search *Space* and Find an Optimum Solution

In case there are more than one *Solution* still left in the search *Space*, this step aims at finding an *OptSoln* from the set of remaining *Solution*s. It does so through iterative division and refining of the search *Space*. Choosing a certain PS for a $Term$ (and omitting the other PSs from the search *Space*) does affect $nUsedMax$ and $nUsedMin$ of the constituent $Term$s. This, in turn, can affect $nAJMax_{(o)}$ and $nAJMin_{(o)}$ of other PSs that use these $Term$s, allowing for possible expensive PS elimination (through Corollary 3.14). Hence, instead of exploring all *Solution*s in the current search *Space*, Step IV divides the search *Space* into mutually exclusive sub-*Space*s (based on mutually exclusive PS choices for what is referred to as *Selection* $Term$s). Each sub-*Space* is then refined and possibly recursively divided until only one *Solution* is left in that sub-*Space*. The *Cost* of each remaining *Solution* in each sub-*Space* is computed, compared and an *OptSoln* is returned. *Space* division and pruning substantially reduces the total amount of *Solution*s explored.

Algorithm 3 is used to implement Step IV. It makes use of the following data structures for each search *Space* (besides $nUsedMax[Term_t]\Big|_{S_k}$, $nUsedMin[Term_t]\Big|_{S_k}$, $PSS[Term_t]\Big|_{S_k}$, $nAJMin_o/Max_o[Term_t]\Big|_{S_k}$, and $PSnAJMin_{(o)}/Max_{(o)}[Term_t][PS_{ti}]\Big|_{S_k}$).

- $STermS\Big|_{S_k}$: a vector of *Selection Terms*. These are the *essential Terms* (see Def. 3.24 of $ETermS$) of S_k. They are also the $Term$s on whose PS choices a *Space* division may occur. $STermS$ of the whole search *Space* (S_o) is initialized with $TargetS$.
- $STerm_c\Big|_{S_k}$: the current $STerm$ on whose PS choices S_k may be divided into sub-*Space*s.
- $STP\Big|_{S_k}$: the index of $STerm_c$ in $STermS$.
- $PSSelect\Big|_{S_k}$: a vector that keeps track of each decision (i.e., PS choice) made for each $STerm$ in S_k.

Algorithm 3 is initially called with the *whole* search *Space* as an input *Space* (S_k). $STermS\Big|_{S_k}$ is initialized with $TargetS$. By definition, any *Solution* in S_k must construct all $STermS\Big|_{S_k}$. Starting with the $STerm_c$ pointed to by STP (initially 1), the algorithm checks whether a *Space* division is required or not. In case $STerm_c$ has only one PS, call it SPS (Lines 6 - 10), SPS is chosen for $STerm_c$ and that choice (also referred to as a decision or selection) is stored in $PSSelect$ of S_k (Line 8). Furthermore, since each $STerm$ is an $ETerm$, and by Def. 3.24 of $ETermS$, therefore, all the $Term$s in SPS are also $ETerm$s in S_k. Thus, they are all appended to $STermS$ of S_k (if they were not already there) so that the algorithm decides for their PS choices at a later point (Line 9). Also, in that case there is no need for a *Space* division. The algorithm increments STP of S_k to move to the next $STerm_c$ (Line 10). On the other hand, if $STerm_c$ has n PSs in S_k, with $n > 1$ (Lines 4 - 5), then the current *Space* S_k will be divided into n child sub-*Space*s (Lines 20 - 26). Each sub-*Space*, S_j, will initially copy all the S_k metric structures (including $PSSelect$ and $STermS$ - Line 21). Then, each sub-*Space*, S_j, will have a mutually exclusive PS choice of $Term_c$, PS_{cj}. The PS choice of each sub-*Space* is stored in its corresponding $PSSelect$ (Line 22). Since each sub-*Space* now only sees one PS for $Term_c$, therefore, each $Term$ in that PS is an $ETerm$ of the corresponding sub-*Space*. These new $ETerm$s are now appended to $STermS$ (Line 23) so that the algorithm decides for their PS choices at a later point. As mentioned at the beginning of this section, such PS selections affect the *Space* metrics and typically lead to further search *Space* reduction in each sub-*Space*

Algorithm 3 Find the *optimum Solution* in this *Space* (*Space* S_k)

$STP\_is\_updated = 0$
while ($!STP\_is\_updated$) **do**
  $STerm_c\Big|_{S_k} = STermS[STP\Big|_{S_k}]\Big|_{S_k}$
  if $\left(\left|PSS[STerm_c]\Big|_{S_k}\right| > 1\right)$ **then** // *A Space division is required*
    $STP\_is\_updated = 1$
  else if $\left(\left|PSS[STerm_c]\Big|_{S_k}\right| == 1\right)$ **then** // *No Space division is required*
    Let SPS be the only PS of $STerm_c$ in S_k
    $PSSelect[STerm_c]\Big|_{S_k} = SPS$
    Append each $Term_i \in SPS$ (and $\notin STermS\Big|_{S_k}$) to $STermS\Big|_{S_k}$
    $STP\Big|_{S_k}++$
  else if $\left(\left|PSS[STerm_c]\Big|_{S_k}\right| == 0\right)$ **then** // *No Space division is required*
    $STP\Big|_{S_k}++$
  end if
  if $STP\Big|_{S_k} > \left|STermS\Big|_{S_k}\right|$ **then** // *All STermS have been decided for*
    Calculate the *Cost* of this *Soln* (i.e., $PSSelect\Big|_{S_k}$) and compare with *OptCost*
    Update *OptSoln* if necessary
    return
  end if
end while
for each PS_{cj} in $PSS[STerm_c]\Big|_{S_k}$ **do** // *Divide S_k into sub-Spaces*
  Create a new *Space* ($S_j = S_k$)
  $PSSelect[STerm_c]\Big|_{S_j} = PS_{cj}$
  Append each $Term_i \in PS_{cj}$ (and $\notin STermS\Big|_{S_j}$) to $STermS\Big|_{S_j}$
  Refine this search *Space* based on this selection (S_j, $STP\Big|_{S_j}$)
  Find the optimum *Solution* in this *Space* (S_j)
end for
return

(Line 24). The procedure of Line 24 is very similar to the one in Algorithm 2, except that UT is initialized with only one $Term$, namely, $STerm_c$. After refining the sub-*Space*, S_j, Algorithm 3 is called iteratively to continue the divide and prune process. Iterations continue until a sub-*Space* is created that has only one *Solution* left (Lines 14 - 18). A search *Space*, S_j, is reduced to one *Solution* if all its $STermS$ have been decided for, or formally, when the following holds: $\forall STerm_i \in \left(STermS\Big|_{S_j} - SourceS \right) : \left| PSS[STerm_i]\Big|_{S_j} \right| = 1$. Once there is only one *Solution* left, its $Cost$ is calculated and compared to $OptCost$. The procedure repeats for all sub-*Space*s and the algorithm returns an *OptSoln*.

In the worst case, Steps III and IV (i.e., Algorithms 2 and 3, respectively) will need to visit every possible *Solution* left in the search *Space* (from Step II) before returning an *OptSoln*, a number which is exponential ($\leq \prod_{i=1}^{|PTermS \cup TargetS|} |PSS[Term_i]|$). Nonetheless, the number of visited *Solution*s, in practice, is much smaller due to the *Space* reduction techniques employed in these steps. Table 3.3 shows the reduction in the search *Space* after running Steps III and IV for sample problems.

3.2.6 *OptSoln* Check

Let the minimum *Cost Solution* returned by Step IV be denoted as $OptSoln^i$, where i is the index of the method used to construct the *potential Terms* (also referred to as $PTermS^i$) in Step I (Sec. 3.2.2). The algorithm is proven to return the minimum *Cost Solution* ($OptSoln^i$) among those *Solution*s that can *only* use terms from $PTermS^i$. In the case when the *potential Terms* are constructed using Method I, it is proven (Theorem 3.3) that $\exists OptSoln_i \in OptSolnS : PTermS^1 \supseteq UsefulTermS(OptSoln_i)$. Hence, passing $PTermS^1$ (computed by Method I) to the algorithm, is proven to result in, indeed, an *optimum Solution* to the given problem.

Method IV, on the other hand, provides a substantially smaller number of *potential Terms* than Method I which enhances the algorithm runtime. However, as shown below, in some problems there may not be an $OptSoln_i \in OptSolnS$ such that $PTermS^4 \supseteq UsefulTermS(OptSoln_i)$. Hence, in such a case, the minimum *Cost Solution* returned by the algorithm (i.e., $OptSoln^4$) may have higher *Cost* than the *optimum Solution*. This can happen when an *optimum Solution* requires a *Term* that is not in the given *potential Terms*.

Therefore, the following two criteria were developed to help check whether the $OptSoln^i$ returned by the algorithm (when given $PTermS^i, i \neq 1$) is indeed an *optimum Solution* for a given problem. The criteria help define if a $Term$ is missing from the given $PTermS^i$, and what the missing $Term$ is. The checks are not required when the *potential* $Terms$ are constructed using Method I. Furthermore, there is no proof that these checks are complete (although found very useful in practice as illustrated below and in Sec. 3.3 - the Results). Failing Check I (introduced below) is a sufficient condition to show that the returned $OptSoln^i$ is *not* an *optimum* $Solution$ for the problem, and that indeed one or more terms are missing from the corresponding $PTermS^i$. On the other hand, failing Check II does not necessarily mean that the returned $OptSoln^i$ is not an *optimum* $Solution$ for the problem.

Algorithm 4 shows a pseudo-code for the whole CNG algorithm (including using the checks). The checks are used to iterate over the algorithm with added terms to $PTermS^i$ in each iteration. Iterations stop when an $OptSoln^i$ is found that passes both checks.

3.2.6.1 Check I: Sharing Check

If more than one $Term$ (call them constituting $Terms$) appear *together* implementing more than one *useful* $Term$ in $OptSoln^i$, then, this is a sufficient condition that a $Term$ $PTerm_m$ is missing from $PTerm^i$. $PTerm_m$ is the union of these constituent $Terms$. It is

Algorithm 4 CNG ($INodeS$, $TargetS$, $PTermConstructionMethod$)

Step I: Construct the Potential $Terms$ using Method $PTermConstructionMethod$
$done = 0$
while ($done = 0$) **do**
  Step II: Construct the Partial Solutions
  Step III: Collect Space Metrics and Remove Higher nAJ Partial Solutions
  Step IV: Divide, Refine the Search Space and Find an Optimum $Solution$
  if ($PTermConstructionMethod$ = Method I) **then**
    $done = 1$
  else
    C_1 = Check I ($OptSoln^i$) *// PASS/FAIL, also possibly updates* $NewPTermS$
    C_2 = Check II ($OptSoln^i$) *// PASS/FAIL, also possibly updates* $NewPTermS$
    if ($C_1 \wedge C_2$) **then** *//* $OptSoln^i$ *passes both checks*
      $done = 1$
    else
      // The checks found possibly missing $PTerms$ *(i.e.,* $|NewPTermS| > 0$*)*
      $PTermS^i = PTermS^i \cup NewPTermS$
    end if
  end if
end while
return

easy to show that another $Solution$ that would be the same as $OptSoln^i$ except that it uses $PTerm_m$ instead of joining its constituent $Terms$ each time they are needed would have a lower $Cost$. The following theorem formalizes the argument:

Theorem 3.16. Check I *Let* $Term_j, Term_k \in UsefulTerms(OptSoln^i)$. *Let* $OptSoln^i[Term_j] \cap OptSoln^i[Term_k] = S$. *Being a set of* $Terms$*, possibly empty, let* $S = \{Term_{s1}, Term_{s2}, \ldots\}$. *If* $|S| > 1$ *then Check I fails. Define* $PTerm_m = \bigcup_{l=1}^{|S|} Term_{sl}$. *Also, define:* $PTermS^{i'} = PTermS^i \cup \{PTerm_m\}$. *The following holds:*

1. $OptSoln^i$ *is not an optimum* $Solution$ *for the problem.*
2. *Passing* $PTermS^{i'}$ *to the algorithm instead of* $PTermS^i$ *will produce* $OptSoln^{i'}$ *(instead of* $OptSoln^i$*) such that:* $Cost(OptSoln^{i'}) < Cost(OptSoln^i)$.

Proof. The description of $OptSoln^i$ provided in the theorem text implies that $PTerm_m \notin PTermS^i$. Since, if $PTerm_m$ was indeed in $PTermS^i$, then, according to Rule II, the algorithm would have used it to construct (at least) $Term_j$ and $Term_k$ instead of using its constituent $Terms$ (i.e., $\{Term_{s1}, Term_{s2}, \ldots\}$)[4].

Let $OptSoln^i[Term_j] = PS_j' \cup S$ and $OptSoln^i[Term_k] = PS_k' \cup S$. Define $Solution$ $Soln_1$ such that: $Soln_1 / \{Term_j, Term_k, PTerm_m\} = OptSoln^i / \{Term_j, Term_k, PTerm_m\}$, $Soln_1[Term_j] = PS_j' \cup \{PTerm_m\}$, $Soln_1[Term_k] = PS_k' \cup \{PTerm_m\}$, and $Soln_1[PTerm_m] = S$. Note that $OptSoln^i[PTerm_m]$ does not matter since $PTerm_m$ is not *useful* in $OptSoln^i$. It is easy to show that $Cost(OptSoln^i) - Cost(Soln_1) = |S| - 1$. That concludes the first half of the proof.

On the other hand, if $PTermS^{i'}$ is passed to the algorithm instead of $PTermS^i$, then applying Rule II will result in a $Solution$ with the same $Cost$ of $Soln_1$ mentioned above or less. That concludes the second half of the proof. ■

Following is an example where Method IV fails to provide an *optimum Solution* for the problem (i.e., $Cost(OptSoln^4) > OptCost$). $OptSoln^4$ fails Check I. Nonetheless, the correction in the second iteration of Algorithm 4 results in an *optimum Solution*.

Example 3.30. Find an *optimum* control network implementation for the following register-to-register data communications: $INodeS = \{A, B, C, D, E, F, G, H, I, J, L, M\}$,

[4]To avoid confusion with Rule V, note also that if $PTerm_m$ was to be used in $OptSoln^i$, it would have been used more than once (i.e., to construct at least $Term_j$ and $Term_k$). This implies that Rule V does not apply in this case.

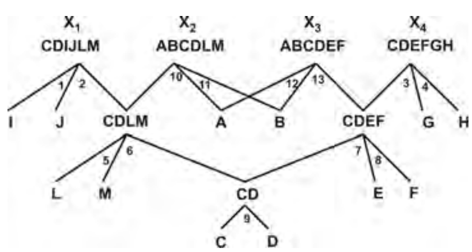
$ONodeS = \{X_1, X_2, X_3, X_4\}$, Target of X_1 (or for short, X_1)$=\{C, D, I, J, L, M\}$, $X_2 = \{A, B, C, D, L, M\}$, $X_3 = \{A, B, C, D, E, F\}$, and $X_4 = \{C, D, E, F, G, H\}$.

The $OptCost$ for this problem is 12. First and second iterations of CNG running this problem using Method IV are depicted in Fig. 3.7. Method IV first iteration returns a $Solution$, $OptSoln^4$, with $Cost = 13$. The $Solution$ returned fails Check I, since $\left|OptSoln^4[ABCDLM] \cap OptSoln^4[ABCDEF]\right| = |\{A, B\}| = 2 > 1$. According to Theorem 3.16, $OptSoln^4$ is not an *optimum Solution* for the problem, and $Term\ AB$ is missing from $PTermS^4$. In the second iteration, $Term\ AB$ is added to $PTermS^4$ and the $Cost$ returned is, indeed, the $OptCost$ (i.e., 12).

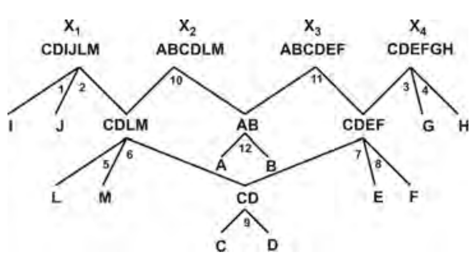
3.2.6.2 Check II: Redundancy Check

If only a subset of a $Term$ is *useful* in a PS of $OptSoln^i$, then, this *may* indicate that this useful subset of the $Term$ is missing in $PTermS^i$. It *may* also indicate that replacing the $Term$ with its useful sub-$Term$ in that PS results in a better $Solution$.

Formally, let $Term_t \in UsefulTermS(OptSoln^i)$ and $OptSoln^i[Term_t] = PS_t$. Let also $Term_i \in PS_t$. Define $PTerm_m = AddedCoverage(Term_i, PS_t)$ (see Def. 3.19 of $ACov$). Then, if $PTerm_m \subset Term_i$ and $PTerm_m \notin PTermS^i$, then, Check II fails.



(a) First iteration $OptSoln^4$.
$Cost = 13$.



(b) Second iteration $OptSoln^4$.
$Cost = 12$.

Figure 3.7: First and second iterations for Example 3.30 using Method IV.

Define $PTermS^{i'} = PTermS^{i} \cup \{PTerm_m\}$. The following holds:

1. $OptSoln^i$ *may* not be an *optimum Solution* for the problem.
2. Passing $PTermS^{i'}$ to the algorithm instead of $PTermS^i$ *may* produce $OptSoln^{i'}$ (instead of $OptSoln^i$) such that: $Cost(OptSoln^{i'}) < Cost(OptSoln^i)$.

Note that failing Check II does not necessarily imply that $OptSoln^i$ is not indeed *optimum*. In fact, Example 3.32 introduced in Sec. 3.3.4 shows that in some cases it reduces the $Cost$ if $Term_i$ (rather than its subset, $PTerm_m$) is used in PS_t even if $Term_i$ is overlapping with other terms in PS_t (while $PTerm_m$ is not). This can happen, for example, if $Term_i$ is needed for other $Terms$ in $OptSoln^i$ and thus can be shared while $PTerm_m$ is not.

Following is an example where Method IV fails to provide an *optimum Solution* for the problem (i.e., $Cost(OptSoln^4) > OptCost$). $OptSoln^4$ fails Check II. Nonetheless, the correction in the second iteration of Algorithm 4 results in an *optimum Solution*.

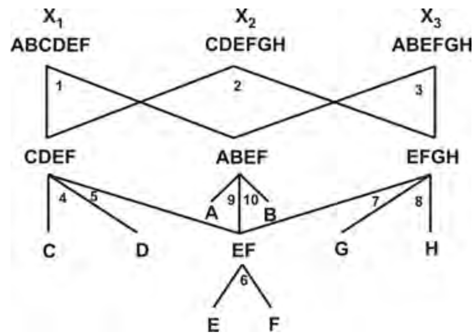
Example 3.31. Find an *optimum* control network implementation for the following register-to-register data communications: $INodeS = \{A, B, C, D, E, F, G, H\}$, $ONodeS = \{X_1, X_2, X_3\}$, $X_1 = \{A, B, C, D, E, F\}$, $X_2 = \{C, D, E, F, G, H\}$, and $X_3 = \{A, B, E, F, G, H\}$.

The $OptCost$ for this problem is 9. Minimum *Cost Solutions* returned by the first and second iterations of CNG using Method IV are depicted in Fig. 3.8. Method IV first iteration returns a *Solution*, $OptSoln^4$, with $Cost = 10$. The *Solution* returned fails Check II, since, for example, $ACov(ABEF, PS_{X_1}) = AB \subset ABEF$ and $AB \notin PTermS^4$, where $PS_{X_1} = OptSoln^4[X_1]$. This suggests that $OptSoln^4$ *may not be* an *optimum Solution* for the problem. In that example, this is indeed the case. In the second iteration, $Term\ AB$ is added to $PTermS^4$ and the $Cost$ returned is the $OptCost$ (i.e., 9).

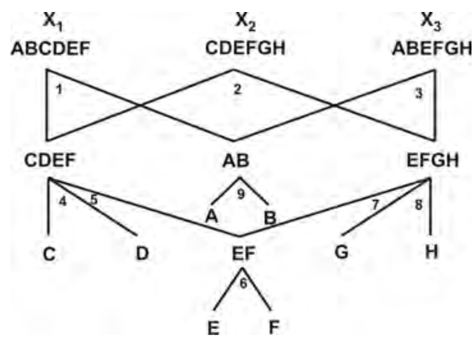
3.3 Results

3.3.1 CNG Tool

The algorithm has been coded in C++ within a tool called CNG. Multi-core parallel programming using OpenMP [56] has been employed whenever possible. A pseudo-code for the main CNG steps is listed in Algorithm 4. CNG accepts an input file with the required register-to-register communications. It returns an $OptSoln$ and the $OptCost$. Another tool, PreCNG, was developed to take an ISCAS benchmark in *verilog* and automatically finds



(a) First iteration $OptSoln^4$.
$Cost = 10$.



(b) Second iteration $OptSoln^4$.
$Cost = 9$.

Figure 3.8: First and second iterations for Example 3.31 using Method IV.

the register-to-register communications. These communications are then expressed in *eqn* and *verilog* formats as well as another format that CNG accepts.

3.3.2 Case Study: The MiniMIPS

MIPS (Microprocessor without Interlocked Pipeline Stages) is a 32-bit architecture, first designed by Hennessy [46]. MiniMIPS is an 8-bit subset of MIPS. It is fully described in [1]. A block diagram of the original clocked MiniMIPS is shown in Fig. 2.5. Its synchronous elasticization is described in Sec. 2.2.

The required register-to-register communication in the MiniMIPS are passed to CNG. CNG generates the elastic control network shown in Fig. 3.9.

Generating a control network for the MiniMIPS using the direct approach of [9] would result in a network with 25 J_2s and 25 F_2s. A hand optimized version of its control network is shown in Fig. 2.6. The hand optimized version utilizes 14 J_2s and 14 F_2s. Comparing to the hand optimized version and to the direct approach of [9], CNG generates a network

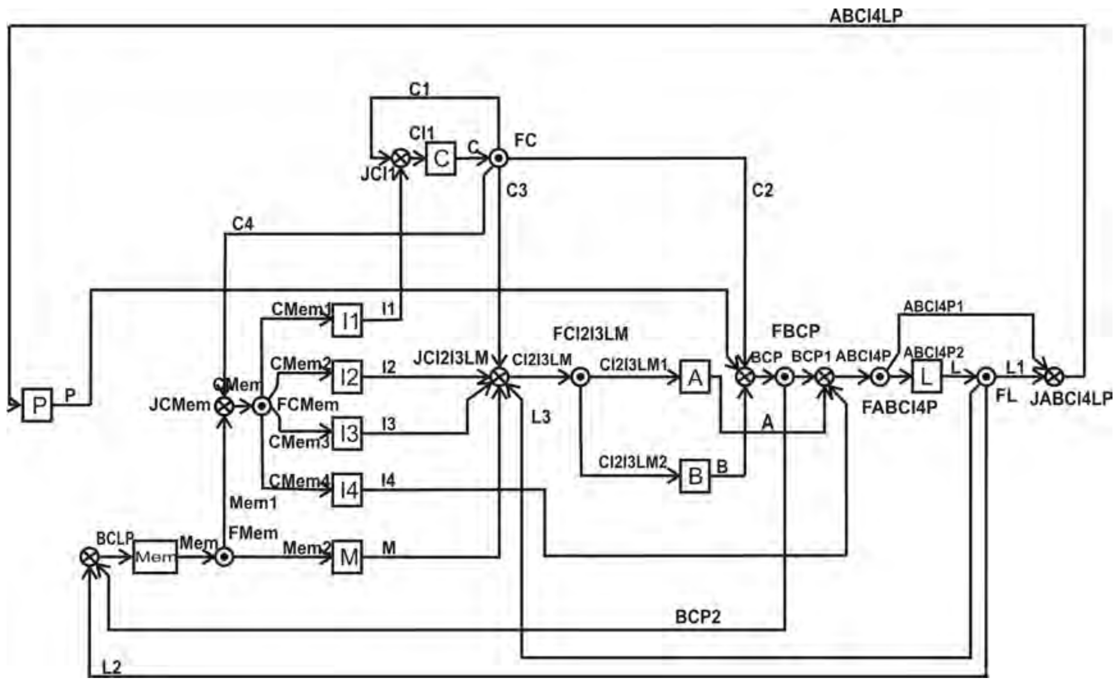


Figure 3.9: CNG-optimized control network of the elastic clocked MiniMIPS.

with only 12 J_2s and 12 F_2s, for 14.3% and 52% reductions, respectively.

3.3.3 Different $PTermS$ Construction Methods

Table 3.2 shows $PTermS$ size for some ISCAS benchmarks and other problems. For all the listed examples Method IV kept $PTermS$ size below 100. Reduction of $PTermS$ size from Method I to Method IV substantially reduces the algorithm runtime.

Table 3.3 shows the reduction in the search *Space* size after applying each CNG step for different $PTermS$ construction methods. Step IV (Sec. 3.2.5) does iteratively divide and refine the search *Space* until each sub-*Space* contains only one *Soln*. The *Cost* of each remaining *Soln* of each sub-*Space* are then computed and compared to return $OptSoln^i$. The last column (titled "After Step IV") lists the total number of these remaining *Soln*s (i.e., the *Soln*s whose *Cost*s are computed and compared). In all the examples of Table 3.3, Method IV returns $OptSoln^4$ after Step III.

3.3.4 CNG vs. Other Synthesis Tools/Flows

Following is a brief description of other approaches that may be used to construct the control network of elastic circuits (besides CNG). For the following approaches, PreCNG is used to take an ISCAS benchmark and automatically formulate the register-to-register

Table 3.2: $\left|PTermS^i\right|$ of different $PTermS$ Construction Methods.

Problem	$\lvert SourceS \rvert$	$\lvert TargetS \rvert$	$\lvert PTermS^1 \rvert$	$\lvert PTermS^2 \rvert$	$\lvert PTermS^3 \rvert$	$\lvert PTermS^4 \rvert$
Example 3.2	7	5	31	31	20	14
MiniMIPS	12	12	46	22	21	17
s27	7	4	64	10	10	9
s298	17	20	162	88	41	33
s344	24	26	8,223	2,064	1,106	38
s349	24	26	8,223	2,064	1,106	38
s382	24	27	16,583	193	67	37
s386	13	13	4,096	71	32	21
s400	24	27	16,583	193	67	37
s420	34	17	131,088	65,554	155	50
s444	24	27	16,583	193	67	37
s510	25	13	1,420	46	37	30
s526	24	27	16,488	4,156	132	45
s641	54	43	3,014,686	23,593	493	85
s713	54	42	3,014,686	23,593	493	85
s820	23	24	1,105,919	9,483	330	46
s832	23	23	1,105,919	9,483	330	46
s1488	14	25	16,383	517	79	32

Table 3.3: Search *Space* reduction (in terms of number of *Solns*) for different methods.

Problem	M	Total (with Rule I applied)	After Step I	After Step II	After Step III	After Step IV
Example 2	M1	3.04×10^{49}	1.44×10^{20}	3.01×10^{8}	42	2
	M2		1.10×10^{20}	3.01×10^{8}	42	2
	M3		1.56×10^{11}	6,912	12	2
	M4		9.12×10^{5}	8	1	1
MiniMIPS	M1	7.05×10^{97}	1.28×10^{34}	1.13×10^{14}	234	2
	M2		6.33×10^{8}	72	6	2
	M3		1.06×10^{8}	24	4	2
	M4		3.07×10^{4}	4	1	1
s27	M1	7.94×10^{78}	2.72×10^{77}	1.64×10^{33}	1	1
	M2		$1,000$	1	1	1
	M3		$1,000$	1	1	1
	M4		108	1	1	1
s298	M1	double overflow $> 1.7 \times 10^{308}$	3.04×10^{257}	7.38×10^{111}	1	1
	M2		1.43×10^{109}	2.48×10^{42}	1	1
	M3		1.31×10^{37}	5.57×10^{8}	1	1
	M4		1.63×10^{22}	2.88×10^{3}	1	1

communication requirements in forms accepted by these approaches (e.g., *eqn* and *verilog* formats).

3.3.4.1 Basic Flow

A direct flow is provided in [9, 3]. In that approach, for each register that is receiving data communications from multiple registers, one multi-input join is connected to this register controller input. Similarly, for each register that is sending data communications to multiple registers, one multi-output fork is connected to this register controller output. This approach, however, could be inefficient in terms of the total number of joins and forks used, increasing the elastic control network area and power overheads.

3.3.4.2 Berkeley ABC

ABC [54] is a synthesis tool from Berkeley. The control network problem may be formulated as an equation, with the join components replaced by logical ANDs. In that sense, every $Target$ is an output of a logical AND of all the $INode$s going to that $Target$. Formally: $\forall Target_i \in TargetS : Target_i = AND_{j=1}^{|Target_i|} INode_j$, where $INode_j \in Target_i$. The following script (courtesy of Alan Mishchenko, one of ABC authors) is used to minimize the number of 2-input AND gates (which would correspond to minimizing 2-input join components) in a given control network:

```
read_eqn connection.eqn; st; ps
clp; fx; resyn2; ps; write_eqn out.eqn
```

connection.eqn is the file containing the required register-to-register communications (in standard *eqn* format).

Note that, from Theorem 3.2, minimizing the number of 2-input join components in a control network will equivalently minimize the *total* number of 2-input join and 2-output fork components in that network.

3.3.4.3 Synopsys® Design Compiler®

Design Compiler® (DC) is a synthesis tool from Synopsys®. Similar to the control network problem formulation with ABC, the required connections can be passed to DC as a *verilog* input file. To minimize the total number of 2-input AND gates (corresponding to 2-input join components) a cell library composed of only one cell, a 2-input AND gate, is passed to the tool. DC Ultra™ is asked to minimize the control network area through the following commands:

```
set_max_area 0
compile_ultra -area_high_effort_script
```

Table 3.4 compares the results of the different approaches over several ISCAS-89 benchmarks and other problems. For each approach column, it shows the *Cost* (i.e., the total number of J_2s required to implement the control network) and the Worse% with respect to CNG. In all complete benchmark runs in this chapter, DC and ABC produce a network with the same or more number of join (and fork) components than CNG. In s614, for example, ABC produces a network with 11.3% more joins than CNG (69 vs. 62). In s1238, DC produces a network with 10.9% more joins than CNG (51 vs. 46). Method IV is used in CNG. Multiple rows per problem reflects the number of CNG iterations. The CNG column also shows the runtime required by each problem. In all the listed ISCAS problems, the total runtime (i.e., including all iterations) is less than 1 second. The machine used has Intel® Core™ i7 2.80GHz processor. ISCAS problems bigger than s1488 require impractically long runtime. This motivates using better data structures, problem division algorithms and/or heuristics to cut runtime for bigger problems (see Appendix A). The CNG column also includes *nSol* sub-column. *nSol* gives the number of *Solutions* left in the search *Space* after applying the reductions of Steps I to IV. This is the number of *Soln*s whose *Cost*s have to be calculated and compared to return the *OptSoln*. In most of the listed ISCAS problems, only one *Solution* is left after applying the algorithm reductions. This shows the reduction efficiency of Steps I to IV.

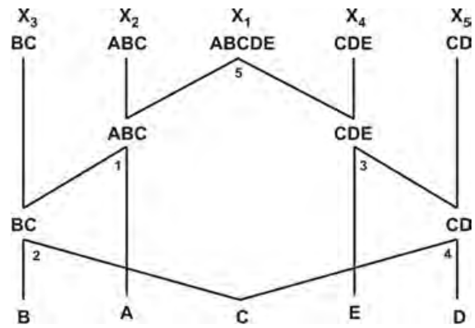
The following example, *ProOverlap_n_m*, is locally developed based on observations of DC and ABC synthesis of some of the ISCAS-89 benchmarks.

Example 3.32. *ProOverlap_5_1* Find an *optimum* control network implementation for the following register-to-register data communications: $INodeS = \{A, B, C, D, E\}$, $ONodeS = \{X_1, X_2, X_3, X_4, X_5\}$, $X_1 = \{A, B, C, D, E\}$, $X_2 = \{A, B, C\}$, $X_3 = \{B, C\}$, $X_4 = \{C, D, E\}$, and $X_5 = \{C, D\}$.

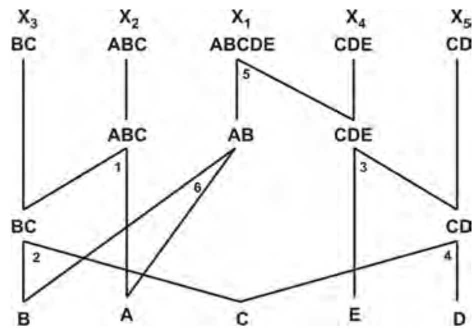
Fig. 3.10 shows CNG vs. Design Compiler® (DC) *Solutions* for that problem. CNG produces a control network with one less join (and one less fork) than DC. The difference occurs because CNG implements $Target\ X_1$ (i.e., $ABCDE$) as follows: $OptSoln_{CNG}[ABCDE] = \{ABC, CDE\}$. On the other hand, $OptSoln_{DC}[ABCDE] = \{AB, CDE\}$. In $OptSoln_{CNG}[ABCDE]$, $Term\ ABC$ covers three $INode$s (i.e., A, B, and C) while only A and B are needed (since $Term\ CDE$ is also covering C). $INode$s A and

Table 3.4: CNG *Cost* vs. other synthesis tools/flows. Worse percentages are calculated with respect to CNG results.

Problem	CNG			Flow of [9, 3]		ABC		Design Compiler®	
	Cost	runtime	*nSol*	*Cost*	Worse%	*Cost*	Worse%	*Cost*	Worse%
MiniMIPS	12	$< 1s$	1	25	108.3%	12	0%	12	0%
s27	6	$< 1s$	1	17	183.3%	6	0%	6	0%
s298	22	$< 1s$	1	66	200%	23	4.5%	22	0%
s344	30	$< 1s$	1	95	216.7%	32	6.7%	30	0%
s382	22 22	$< 1s$	1 1	148	572.7%	22	0%	22	0%
s349	30	$< 1s$	10	95	216.7%	32	6.7%	30	0%
s386	15	$< 1s$	1	116	673.3%	15	0%	15	0%
s400	22 22	$< 1s$	1 1	148	572.7%	22	0%	22	0%
s420	33	$< 1s$	1	169	412.1%	34	3.0%	33	0%
s444	22 22	$< 1s$	1 1	148	572.7%	22	0%	22	0%
s510	25 25	$< 1s$	1 1	90	260%	28	12%	26	4%
s526	29	$< 1s$	1	140	382.8%	30	3.4%	29	0%
s641	62 62	$< 1s$	1 1	457	637.1%	69	11.3%	68	9.7%
s713	62 62	$< 1s$	1 1	444	616.1%	68	9.7%	68	9.7%
s820	34 33 33	$< 1s$	10 160 212	189	472.7%	33	0%	33	0%
s832	34 33 33	$< 1s$	10 160 212	189	472.7%	33	0%	33	0%
s838	65	$< 1s$	1	593	812.3%	66	1.5%	65	0%
s953	37 36 36	$< 1s$	12 16 20	299	730.6%	36	0%	37	2.8%
s1196	46 46 46	$< 1s$	4 114 114	355	671.7%	48	4.3%	51	10.9%
s1238	46 46 46	$< 1s$	4 114 114	355	671.7%	48	4.3%	51	10.9%
s1488	21 21	$< 1s$	3 3	241	1047.6%	22	4.8%	22	4.8%
Overlap_9_1(2)	9	$< 1s$	1	28	211.1%	13	44%	12	33%
Overlap_25_25(2)	625	$< 1s$	1	4500	620%	925	48%	900	44%
Overlap_51_51(2)	2601	$20s$	1	35700	1272.5%	4081	57%	3825	47%
Overlap_n_m	$m\times$ n	- -	1 1	$m\times$ $\frac{n^2+4n-5}{4}$	$\frac{n^2-5}{4n}$ $\times 100\%$	- -	- -	$m\times$ $\frac{3(n-1)}{2}$	$\frac{n-3}{2n}$ $\times 100\%$



(a) CNG - $Cost = 5$.

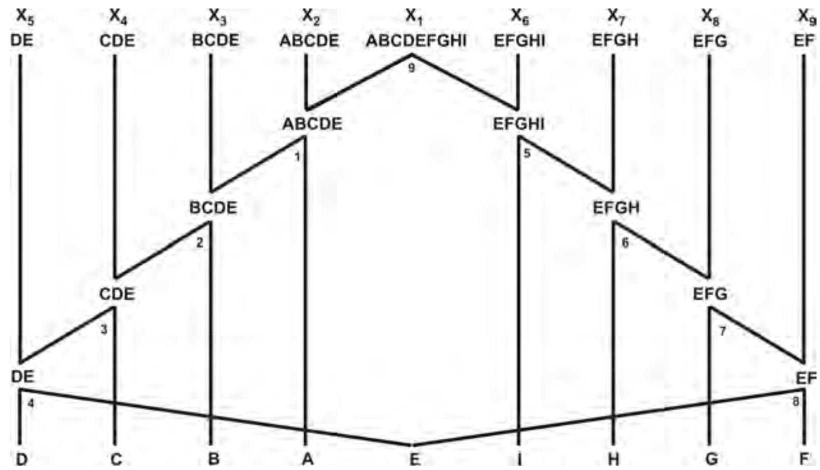


(b) DC - $Cost = 6$.

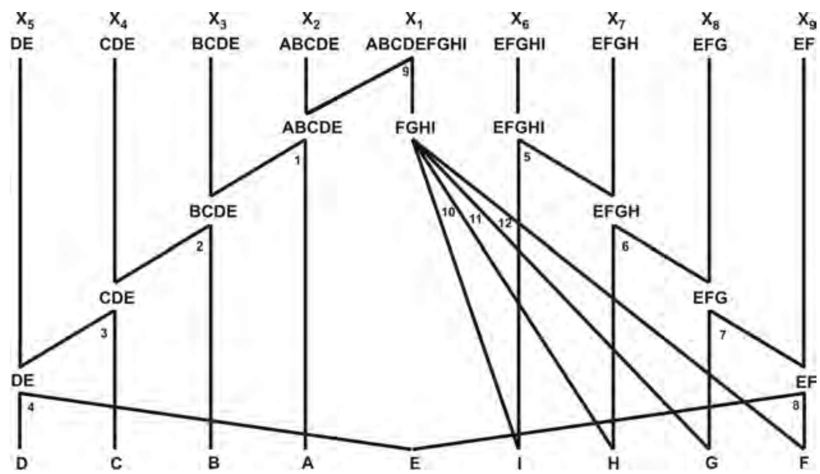
Figure 3.10: *ProOverlap*_5_1 example: CNG vs. DC.

B could be covered by $Term\ AB$ instead. Thus, it may seem that using $Term\ ABC$ in $OptSoln[ABCDE]$ is adding redundancy. However, $Term\ ABC$ is shared in the $Solution$ (it is a $TITerm$ that must be constructed any way to construct $Target\ X_2$ (i.e., ABC) - see Def. 3.18 and Theorem 3.11). $Term\ AB$, on the other hand, is not shared by any other $Term$ in the $Solution$, and thus must be built solely to construct $ABCDE$. That adds the 1-join overhead of DC comparing to CNG. Using Def. 3.19 of $AddedCoverage$, it seems that DC misses the *optimum Solution* because it does not allow for using $Term_i$ in PS_t if $AddedCoverage(Term_i, PS_t) \neq Term_i$. In other words, it seems that DC does not allow for overlapping between the constituent terms of any PS. ABC seems to exhibit similar behavior.

It can be easily shown that Example *ProOverlap*_5_1 can be scaled based on two parameters (n and m), as follows: Define $n = |X_1|$. Also, define m to be the replication factor of the structure (i.e., how many times the structure is replicated). n must be an odd number. For Example *ProOverlap*_5_1, $n = 5$ and $m = 1$. Fig. 3.11 shows CNG vs. DC $Solutions$ for *ProOverlap*_9_1. $Cost(OptSoln_{CNG}) = 9$ while $Cost(OptSoln_{DC}) = 12$ (and $Cost(OptSoln_{ABC}) = 13$). In terms of any odd n and m, the following were verified for



(a) CNG - $Cost = 9$.



(b) DC - $Cost = 12$.

Figure 3.11: *ProOverlap_9_1* example: CNG vs. DC.

numerous values of n and m:

$$Cost(OptSoln_{CNG}) = m \times n \tag{3.68}$$

$$Cost(OptSoln_{DC}) = m \times \frac{3(n-1)}{2} \tag{3.69}$$

That is, $Cost(OptSoln_{DC})$ is $\frac{n-3}{2n}$ worse than $Cost(OptSoln_{CNG})$ (independent of m). The DC to CNG $Cost$ overhead increases as n increases with a limit of %50 as n goes to inf. ABC seems to produce worse results than DC for this specific set of *ProOverlap_n_m* examples. Example *ProOverlap_n_m* was built upon observations of the DC and ABC *Solution*s for some of the ISCAS-89 benchmarks.

CHAPTER 4

LAZY AND HYBRID SELF PROTOCOL IMPLEMENTATIONS[1]

Synchronous elasticization converts an ordinary clocked circuit into Latency-Insensitive (LI). The conversion involves the generation of a handshake control network that reflects the register-to-register communication in the original circuit. The Synchronous Elastic Flow (SELF) is an LI protocol used over the control network channels. This chapter investigates alternative implementations of the SELF protocol that can reduce the control network area and power consumption.

The SELF protocol can be implemented with eager or lazy evaluation in the data steering network. Eager implementation of the SELF protocol enjoys no combinational cycles and also may have performance advantages in some designs when compared to lazy implementations. However, eager protocols are more expensive in terms of area and power consumption. The LI control network area and power consumption may become prohibitive in some cases [3]. Measurements of the MiniMIPS processor fabricated in a 0.5 μm node (see Chapter 2) show that elasticization with an eager SELF implementation results in area, dynamic, and leakage power penalties of 29%, 13%, and 58.3%, respectively.

Lazy SELF implementations may be an attractive solution. Unfortunately the standard implementation suffers from combinational cycles that make it an unreliable design [9, 45]. This work defines a larger design space that can be employed to implement lazy channel protocols and to verify correctness of these protocols both independently and when combined with the standard eager protocol.

A formal investigation of a complete set of lazy SELF protocol specifications is reported. This includes introducing new lazy join and fork structures, which are verified along with the existing designs. A novel hybrid implementation flow is then introduced that combines the advantages of both eager and lazy implementations. The hybrid SELF essentially

[1]This is a revised and extended version of a paper originally published in [49].

avoids some of the redundancy of the eager implementation without any performance loss. Moreover, it is combinational cycle free. The hybrid SELF network is demonstrated with the design of the elastic MiniMIPS processor. The hybrid implementation achieves the same runtime as an *all* eager implementation with a reduction of 31.8%, 26.0%, and 30.8% in the control network area, dynamic, and leakage power consumption, respectively.

An overview of the SELF protocol was given in Sec. 2.1. The notion of a *control buffer* is introduced in order to gain understanding of the design and verification of control network components, such as joins and forks. A linear control buffer simply breaks the control signals in a channel into left and right channels. Such a buffer will have two inputs: the $Valid$ on the left channel and $Stall$ on the right channel, and two outputs: the $Stall$ on the left channel and $Valid$ on the right.

4.1 SELF Channel Protocol Verification

All join and fork components are verified to be conformant to the SELF channel protocol. The correctness requirements for the channel protocol are adapted from the general elastic component conditions consisting of persistence, freedom from deadlock, and liveness [10]. A fourth constraint is added here that disallows glitching on the control wires.

1. *Persistence.* No $R \rightarrow I$ transition may occur.
2. *Deadlock freedom.* For each component in the verification, at least two states can be reached from any other reachable state [57].
3. *Liveness.* The liveness condition is one of data preservation. Lazy control buffers must have the same number of tokens transferred on all their channels. This functional requirement is a special case of the liveness condition in [10]. This is implemented by creating token counters on all the lazy control buffer channels and verifying that they are always equivalent.
4. *Glitch Free.* No $S\uparrow$ signal transition may occur in state I. The specification of the idle protocol state I in Fig. 2.2 does not constrain the behavior of the $Stall$ signal. This allows glitching on the control wires to occur. If the $Stall$ signal is not allowed to rise in the idle state then glitching will not occur. This requirement is not explicit in the SELF specifications. However, it can be observed that this transition is not possible in published Elastic Buffer (EB) or Elastic Half Buffer (EHB) designs [9, 58]. If control wire glitching is possible, then the composition of some forks and joins may not be compliant with the channel protocol. For example, the Karnaugh map of $LF01$, one

of the two lazy forks proven to be SELF compliant (Sec. 4.3.1.2), is shown in Fig. 4.1. Transition A occurs when S_{r2} rises in the idle state. While this glitching transition is valid according to the channel specification, it results in V_{r1} falling, which produces an illegal $R \rightarrow I$ transition on channel r_1. Since this transition can never happen unless channel r_2 can make an $S\!\uparrow$ transition glitch, this condition is added to the verification suite.

4.2 SELF Control Network Design

A truth table can be created to specify the permissible behaviors for the control buffer left $Stall$ and right $Valid$ signals that conform to the SELF channel protocol of Sec. 2.1. Such a truth table shows the flexibility in design choices that can be made. The same procedure is performed for the lazy fork and join components.

4.3 Fork Components

4.3.1 Lazy Fork

The Lazy Fork ($LFork$) does not propagate valid data from its root to its branches until *all* branches are ready to store the data. A sample lazy fork is shown in Fig. 4.2 [8, 9] (which maps to $LF00$ introduced later in the chapter). In Fig. 4.2, if any of the lazy fork branches stalls, it forces all the other branches into the idle state.

4.3.1.1 Lazy Fork Synthesis

The truth table for a lazy fork is shown to be purely combinational. Thus it is easily represented with the Karnaugh Map (KM) shown in Fig. 4.3. The KM has two don't care terms m_0 and m_1 giving four possible designs. Each implementation is denoted as LFm_0m_1 (e.g., $LF00$, $LF01$, etc.). Table 4.1 maps previsouly published lazy fork implementations to those of this work.

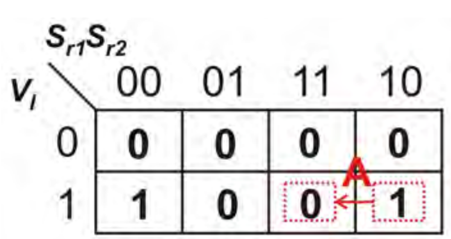


Figure 4.1: V_{r1} of $LF01$.

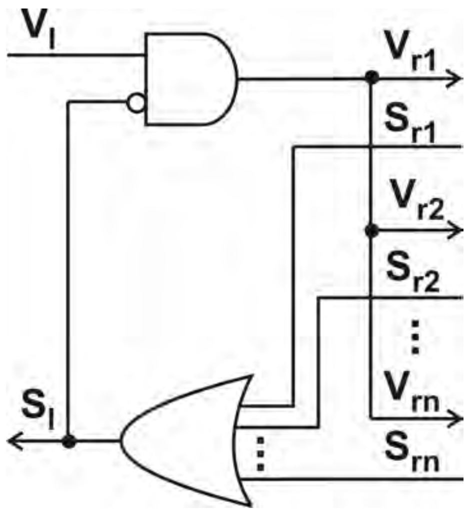


Figure 4.2: A 1-to-n lazy fork (maps to $LF00$).

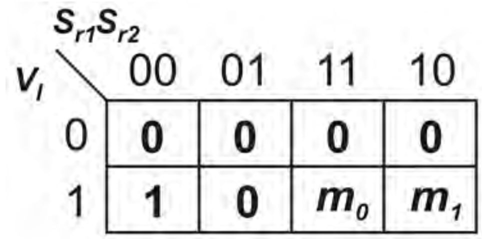


Figure 4.3: Lazy fork specifications (V_{r1}).

Table 4.1: Mapping between published and this work lazy forks and joins.

$Fork$ [8]	$LF00$	$Join$ [8]	$LJ0000$
$Fork$ [9]	$LF00$	$Join$ [9]	$LJ0000$
$LFork$ [45]	$LF00$	$LJoin$ [45]	$LJ0000$
$LKFork$[1][45]	$LF01$	$LKJoin$[1][45]	$LJ1111$

[1] $LKFork$ and $LKJoin$ are part of the contribution of this dissertation.

The hand translation of the fork as a control buffer may still result in illegal channel behavior on one or more of the channels due to the interactions between branches of the fork and join. Thus a rigorous verification methodology is employed to prove correctness of the designs. Indeed, verification shows that two of the four possible designs do not fully obey the SELF channel protocol.

4.3.1.2 Lazy Fork Verification

The setup of Fig. 4.4 is used to verify correctness of the fork designs. The root channel (A) as well as the branches ($A1$ and $A2$) are connected to three elastic buffers (EBs) as well as data token counters (TCs). This work employs the EB implementation published in [9]. The counters track the number of clock cycles that the channel is in the transfer state T. The structure is modeled and passed to a symbolic model checker, NuSMV [59].

All constituent blocks are connected *synchronously* in NuSMV. Synchronous connection

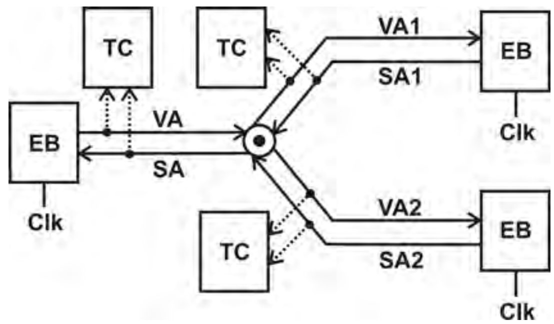


Figure 4.4: Lazy fork verification setup.

guarantees that all modules advance in lock-step. Logic delays are then executed in internal cycles of the verification engine. All combinational logic is modeled to have zero delay. The clock generator is modeled to have a unit delay for each phase. For example, following is the $LF00$ model:

```
MODULE LF00(Vl,Sr1,Sr2)
DEFINE Sl := Sr1 | Sr2 ; DEFINE Vr1 := Vl & (!Sr1) & (!Sr2) ; ...
```

The four SELF compliance checks of Sec. 4.1 are applied to each design as follows: (The properties are expressed in the Property Specification Language (PSL) [60] unless otherwise specified.)

1. *Persistence*. For each channel (i.e., A, $A1$ and $A2$) it is verified that no $R \rightarrow I$ transition occurs:

   ```
   DEFINE R_A := VA & SA ; -- Retry on channel A
   DEFINE I_A := !VA ; -- Idle on channel A
   PSLSPEC never {[*]; R_A; I_A};
   ```

   Out of the 4 lazy fork implementations only $LF00$ and $LF01$ pass this check.

2. *Deadlock freedom*. At least two states are verified as reachable from all other reachable states [57]. For example, inside the $LF00$ module the following properties verify that two states are always reachable: (The properties are specified in the Computation Tree Logic (CTL) syntax [61].)

   ```
   SPEC AG EF (Vr1=1 & Vr2 =1 & Sl=0);
   SPEC AG EF (Vr1=0 & Vr2 =0 & Sl=0);
   ```

   Note that a state in $LF00$ is defined by the three variables: V_{r1}, V_{r2} and S_l. All four lazy fork implementations pass this check.

3. *Liveness* is calculated through *data token preservation*. Let the number of data tokens transferred at the fork root channel and the two branch channels be: d_l, d_{r1} and d_{r2},

respectively. (d_i is, equivalently, the number of clock cycles where channel i is in the Transfer state (T) (i.e., $V_i \& !S_i$).) The number of data tokens transferred at a lazy fork root channel must always be the same as those at its branches. (i.e., the following requirement must always hold: $d_{ri} - d_l = 0$ for $i \in \{1, 2\}$.) The following code is used to model a token counter for channel i. The model counts on the negative edge of the clock.

```
MODULE TokenCounter (Clk,Vi,Si)
VAR Count:  0..31;
ASSIGN
init (Count) := 0;
next (Count) := case
(Clk=1)&(next(Clk)=0)&(Vi=1)&(Si=0)&(Count < 31):  Count + 1;
1:  Count;
esac;
```

NuSMV only supports finite data types. Without loss of generality, the upper limit of the *Count* variable is chosen to be a sufficiently large number (32 in this case). For each branch define and check the following property:

```
DEFINE TokenCountError_A1 := case (dl != dr1):1; 1:0; esac;
PSLSPEC never {[*]; TokenCountError_A1};
```

All the four lazy fork implementations pass this check.

4. *No glitching.* This verifies that the *Stall* signal does not rise in the idle state:

```
DEFINE I0_A := !VA & !SA ; -- Idle0 on A
DEFINE I1_A := !VA & SA ; -- Idle1 on A
PSLSPEC never {[*]; I0_A; I1_A};
```

All lazy fork implementations pass this check.

Hence, among the four possible lazy fork implementations, only $LF00$ and $LF01$ conform to the SELF specification.

4.3.1.3 Lazy Fork Characterization

To help characterize the different fork implementations as well as their combinations with lazy joins in a network, the following definitions are introduced:

Definition 4.1. *C_{Fr}, Fork Reflexive Characterization Set C_{Fr}* is a set of characterization elements (c_{Fr}), where: $c_{Fr} \in \{I, N, 0, 1\}$.

1. $c_{Fr} = I$ (or *inverting*) in a 2-output fork iff V_{ri} is a function of S_{ri}, and iff, for some constant V_l and S_{rj}, $V_{ri} = !S_{ri}$, where $i, j \in \{1, 2\}$ and $i \neq j$.
2. $c_{Fr} = N$ (or *noninverting*) in a 2-output fork iff V_{ri} is a function of S_{ri}, and iff, for some constant V_l and S_{rj}, $V_{ri} = S_{ri}$, where $i, j \in \{1, 2\}$ and $i \neq j$.
3. $c_{Fr} = 0$ (or *constant zero*) in a 2-output fork iff V_{ri} is a function of S_{ri}, and iff, for some constant V_l and S_{rj}, $V_{ri} = 0$, where $i, j \in \{1, 2\}$ and $i \neq j$.
4. $c_{Fr} = 1$ (or *constant one*) in a 2-output fork iff V_{ri} is a function of S_{ri}, and iff, for some constant V_l and S_{rj}, $V_{ri} = 1$, where $i, j \in \{1, 2\}$ and $i \neq j$.

Table 4.2 illustrates C_{Fr} computation of $LF00$. From the table, C_{Fr} of $LF00$ is $\{I, 0\}$. Similarly C_{Fr} of $LF01$ is $\emptyset$. This is because in $LF01$ (see Fig. 4.5), V_{ri} is not a function of S_{ri}. Sec. 4.6.1 will show that this property gives an advantage to $LF01$ since it can reduce the number of combinational cycles in the control network substantially.

Definition 4.2. C_{Ft}*, Fork Transitive Characterization Set* C_{Ft} is a set of characterization elements (c_{Ft}), where: $c_{Ft} \in \{I, N, 0, 1\}$.

1. $c_{Ft} = I$ (or *inverting*) in a 2-output fork iff V_{ri} is a function of S_{rj}, and iff, for some constant V_l and S_{ri}, $V_{ri} = !S_{rj}$, where $i, j \in \{1, 2\}$ and $i \neq j$.
2. $c_{Ft} = N$ (or *noninverting*) in a 2-output fork iff V_{ri} is a function of S_{rj}, and iff, for some constant V_l and S_{ri}, $V_{ri} = S_{rj}$, where $i, j \in \{1, 2\}$ and $i \neq j$.
3. $c_{Ft} = 0$ (or *constant zero*) in a 2-output fork iff V_{ri} is a function of S_{rj}, and iff, for some constant V_l and S_{ri}, $V_{ri} = 0$, where $i, j \in \{1, 2\}$ and $i \neq j$.

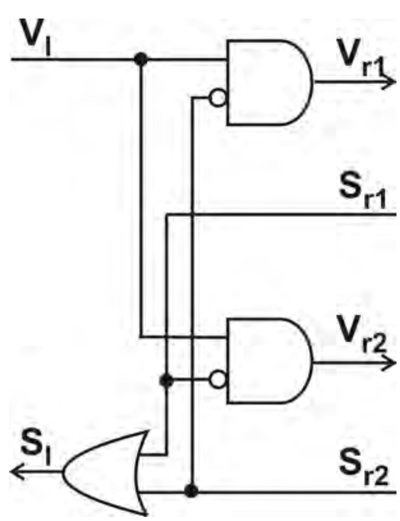


Figure 4.5: A 2-output $LF01$ implementation.

4. $c_{Ft} = 1$ (or *constant one*) in a 2-output fork iff V_{ri} is a function of S_{rj}, and iff, for some constant V_l and S_{ri}, $V_{ri} = 1$, where $i, j \in \{1, 2\}$ and $i \neq j$.

Table 4.3 illustrates C_{Ft} computation of $LF00$. From the table, C_{Ft} of $LF00$ is $\{I, 0\}$. Similarly, C_{Ft} of $LF01$ is also $\{I, 0\}$.

4.3.2 Eager Fork

The Eager Fork ($EFork$), unlike the lazy, even if not all its branches are ready to receive, will immediately pass the (valid) data token from its root to the branches that are ready. The $EFork$ will stall (if needed) until all the stalled branches (if any) receive the data token as well. This gives the earliest possible data transfer to the branches that are ready to receive data. Hence, the $EFork$ can result in performance advantage over lazy forks in some systems. This will also be illustrated in the case study of Sec. 4.7.1. Due to the necessary pipelining that occurs in the control signals, the $EFork$ incorporates one flip-flop per branch. The control flip-flop is clocked every cycle to sample changes. Moreover, eager forks have higher logic complexity comparing to lazy. This makes the $EFork$ expensive in terms of both area and power consumption. Fig. 2.4 shows an n output extension of the $EFork$ proposed in [9].

4.3.2.1 Eager Fork Verification

Similar to the lazy fork verification of Sec. 4.3.1.2, the $EFork$ is also verified against the four SELF compliance checks. Since the $EFork$ allows its ready branches to transfer tokens while stalled waiting for the other branches to be ready, the data token preservation requirement is: $0 \leq d_{ri} - d_l \leq 1$ for $i \in \{1, 2\}$. Indeed, the $EFork$ passes all the checks and, hence, is compliant with the SELF protocol.

Table 4.2: C_{Fr} computation of $LF00$.

V_l	S_{r2}	$S_{r1} \rightarrow V_{r1}$	c_{Fr}
0	0	$0 \rightarrow 0$ $1 \rightarrow 0$	**0**
0	1	$0 \rightarrow 0$ $1 \rightarrow 0$	**0**
1	0	$0 \rightarrow 1$ $1 \rightarrow 0$	**I**
1	1	$0 \rightarrow 0$ $1 \rightarrow 0$	**0**

Table 4.3: C_{Ft} computation of $LF00$.

V_l	S_{r1}	$S_{r2} \rightarrow V_{r1}$	c_{Ft}
0	0	$0 \rightarrow 0$ $1 \rightarrow 0$	**0**
0	1	$0 \rightarrow 0$ $1 \rightarrow 0$	**0**
1	0	$0 \rightarrow 1$ $1 \rightarrow 0$	**I**
1	1	$0 \rightarrow 0$ $1 \rightarrow 0$	**0**

4.4 Lazy Join

The lazy join has to wait for all its input branch channels to carry valid data before data is transferred on the output channel. A sample lazy join is shown in Fig. 2.3 (which maps to $LJ0000$ introduced later in the chapter).

4.4.1 Lazy Join Synthesis

The synthesis of a lazy join as a control buffer is performed similar to the lazy fork. The KM is shown in Fig. 4.6. There are 16 possible implementations.

4.4.2 Lazy Join Verification

Similar to the lazy fork verification in Sec. 4.3.1.2, the structure of Fig. 4.7 is used to verify the different lazy join implementations. The following properties are checked:

1. *Persistence*: All the 16 lazy joins pass this check.
2. *Deadlock freedom*: All the 16 joins pass.
3. *Data token preservation*: All the 16 joins pass.
4. *Glitch Free*: Out of the 16 lazy joins, only 6 pass.

Only the following lazy join designs pass verification: $LJ0000$, $LJ0010$, $LJ0011$, $LJ1010$, $LJ1011$, $LJ1111$. Among the 6 SELF-compliant joins, $LJ1111$ (Fig. 4.8) has the simplest logic allowing for more efficient area utilization during synthesis. Results of Sec. 4.7.2 confirms the observation.

4.4.3 Lazy Join Characterization

To help characterize the different join implementations as well as their combinations with lazy forks in a network, the following definitions are introduced:

S_r \ $V_{l1}V_{l2}$	00	01	11	10
0	m_0	m_1	0	1
1	m_2	m_3	1	1

Figure 4.6: Lazy join specifications (S_{l1}).

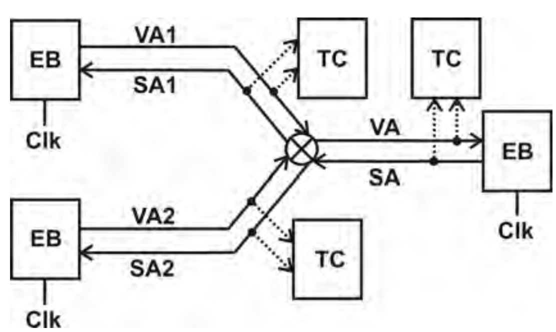


Figure 4.7: Lazy join verification setup.

Definition 4.3. *C_{Jr}, Join Reflexive Characterization Set* C_{Jr} is a set of characterization elements (c_{Jr}), where: $c_{Jr} \in \{I, N, 0, 1\}$.

1. $c_{Jr} = I$ (or *inverting*) in a 2-input join iff S_{li} is a function of V_{li}, and iff, for some constant S_r and V_{lj}, $S_{li} =!V_{li}$, where $i, j \in \{1, 2\}$ and $i \neq j$.
2. $c_{Jr} = N$ (or *noninverting*) in a 2-input join iff S_{li} is a function of V_{li}, and iff, for some constant S_r and V_{lj}, $S_{li} = V_{li}$, where $i, j \in \{1, 2\}$ and $i \neq j$.
3. $c_{Jr} = 0$ (or *constant zero*) in a 2-input join iff S_{li} is a function of V_{li}, and iff, for some constant S_r and V_{lj}, $S_{li} = 0$, where $i, j \in \{1, 2\}$ and $i \neq j$.
4. $c_{Jr} = 1$ (or *constant one*) in a 2-input join iff S_{li} is a function of V_{li}, and iff, for some constant S_r and V_{lj}, $S_{li} = 1$, where $i, j \in \{1, 2\}$ and $i \neq j$.

Similar to Table 4.2, C_{Jr} of $LJ0000$, for example, can be computed to be $\{N, 0\}$. $LJ1011$ has a C_{Jr} of $\emptyset$. This is because in $LJ1011$ (see Fig. 4.9) S_{li} is not a function of V_{li}. Sec. 4.6.1 will show that this property gives an advantage to $LJ1011$ since it can reduce the number of combinational cycles in the control network substantially.

Definition 4.4. *C_{Jt}, Join Transitive Characterization Set* C_{Jt} is a set of characterization elements (c_{Jt}), where: $c_{Jt} \in \{I, N, 0, 1\}$.

1. $c_{Jt} = I$ (or *inverting*) in a 2-input join iff S_{li} is a function of V_{lj}, and iff, for some constant S_r and V_{li}, $S_{li} =!V_{lj}$, where $i, j \in \{1, 2\}$ and $i \neq j$.
2. $c_{Jt} = N$ (or *noninverting*) in a 2-input join iff S_{li} is a function of V_{lj}, and iff, for some constant S_r and V_{li}, $S_{li} = V_{lj}$, where $i, j \in \{1, 2\}$ and $i \neq j$.

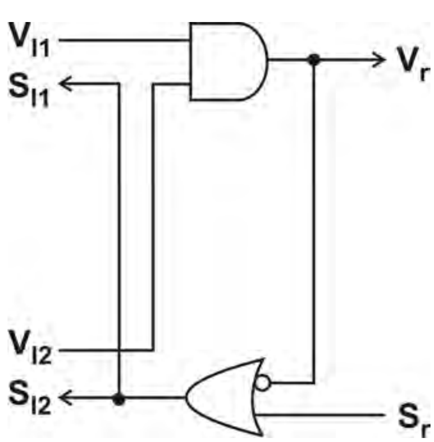


Figure 4.8: A 2-input $LJ1111$ implementation.

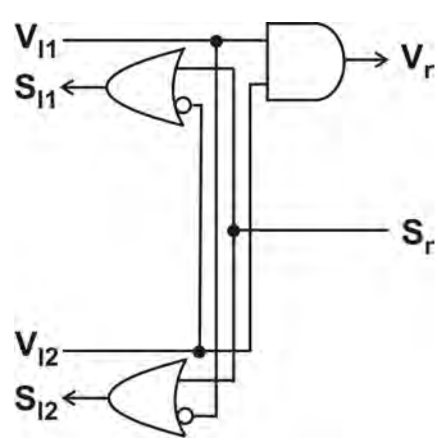


Figure 4.9: A 2-input $LJ1011$ implementation.

3. $c_{Jt} = 0$ (or *constant zero*) in a 2-input join iff S_{li} is a function of V_{lj}, and iff, for some constant S_r and V_{li}, $S_{li} = 0$, where $i, j \in \{1, 2\}$ and $i \neq j$.
4. $c_{Jt} = 1$ (or *constant one*) in a 2-input join iff S_{li} is a function of V_{lj}, and iff, for some constant S_r and V_{li}, $S_{li} = 1$, where $i, j \in \{1, 2\}$ and $i \neq j$.

Similar to Table 4.3, C_{Jt} of $LJ0000$, for example, can be computed to be $\{I, 0, 1\}$.

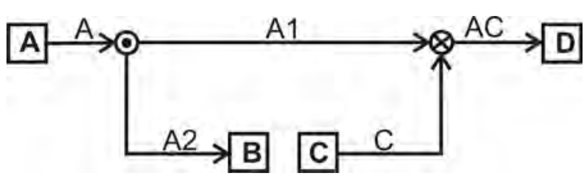
4.5 Lazy SELF Networks

Unlike eager forks, lazy forks have no state holding elements (e.g., flip-flops). Hence, arbitrary connections of lazy joins and forks in a control network typically result in combinational cycles. These cycles can cause deadlock or oscillation due to logical or transient instability:

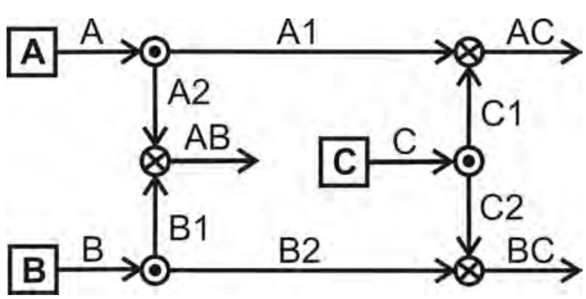
4.5.1 Deadlock - D

A combinational cycle can cause a deadlock if under some input sequence its internal signals can get stuck at certain values. For example, consider a structure in which a fork output channel is feeding a join (Fig. 4.10a). This structure is a basic building block of typical elastic control networks. Fig. 4.11 shows a circuit implementation of Fig. 4.10a using $LF00$ and $LJ1111$.

It can be easily shown that if *VA* is zero, *VA1* and *VAC* must also be zero. This will force *SA1* to be one, *SA* to be one and *VA1* to be zero. Apparently, the loop shown in



(a)



(b)

Figure 4.10: Sample fork join combinations.

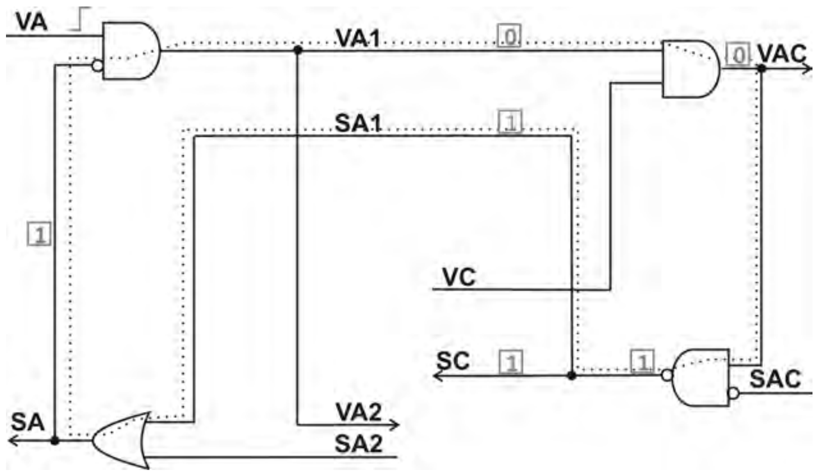


Figure 4.11: $LF00$ and $LJ1111$ combination.

dotted lines forms a latch, since all its wires can simultaneously carry controlling values to the gates they are driving in the loop. Hence, after a zero on *VA*, the system will deadlock. *VA2*, *VAC*, *SC* and *SA* will be stuck at zero, zero, one and one, respectively.

In general, for the common structure of Fig. 4.10a, the following can be readily proved. Let C_{Jr1} (C_{Fr1}) and C_{Jt1} (C_{Ft1}) be the join (fork) reflexive and transitive characteristic sets of the lazy join (fork) used, $LJ1$ ($LF1$), respectively. Then, the connection of Fig. 4.10a will result in deadlock if the following condition holds: $C_{Jr1} = \{1, I\}$ and $C_{Fr1} = \{I, 0\}$. To illustrate, since $C_{Fr1} = \{I, 0\}$, therefore, for all the possible values of $LF1$ inputs, $VA1$ is either 0 or the inverse of $SA1$. Similarly, since $C_{Jr1} = \{1, I\}$, therefore, for all the possible values of $LJ1$ inputs, $SA1$ is either 1 or the inverse of $VA1$. Hence, once $VA1$ is 0 or $SA1$ is 1, the loop formed by $VA1$ and $SA1$ will stuck at these values.

Similarly, a deadlock will occur in the connection of Fig. 4.10b if the following condition holds: $C_{Jt1} = \{1, I\}$ and $C_{Ft1} = \{0, I\}$.

4.5.2 Oscillation Due to Logical Instability - LI

A loop is logically unstable if it has an odd number of inverting elements. Under some input sequence, it can behave as a ring oscillator.

For example, consider again the structure of Fig. 4.10a. Fig. 4.12 shows a circuit implementation of that structure using $LF00$ and $LJ0000$.

Assume the elastic buffer C in Fig. 4.12 holds a bubble (i.e., its output $Valid$ signal is zero), while A holds data. Assume also that *SA2* is zero (B is not stalled). This connection will form a loop (shown in dotted lines in Fig. 4.12). The loop is logically unstable since it

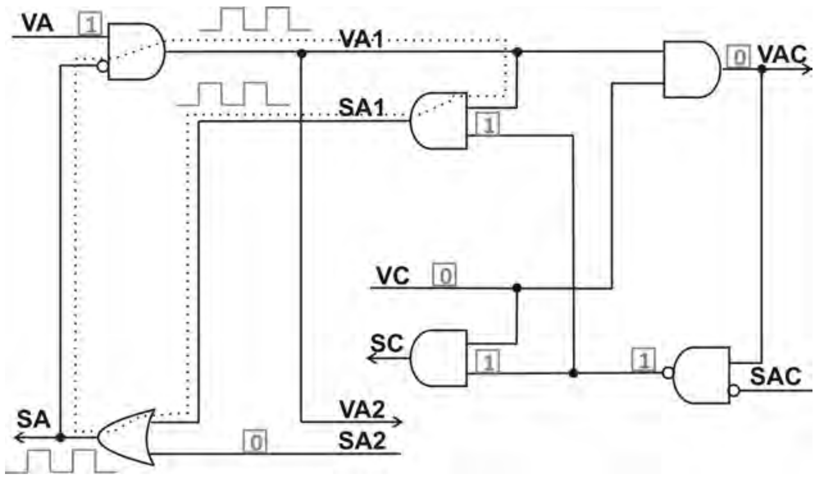


Figure 4.12: $LF00$ and $LJ0000$ combination.

has an odd number of inverting elements. This results in an oscillation inside the loop as well as on the SA wire.

In general, for the common structure of Fig. 4.10a, the following can be readily proved. Let C_{Jr1} (C_{Fr1}) and C_{Jt1} (C_{Ft1}) be the join (fork) reflexive and transitive characteristic sets of the lazy join (fork) used, $LJ1$ ($LF1$), respectively. Then, the connection of Fig. 4.10a will result in logical instability if any of the following condition holds:

- $I \in C_{Jr1}$ and $N \in C_{Fr1}$.
- $N \in C_{Jr1}$ and $I \in C_{Fr1}$.

4.5.3 Oscillation Due to Transient Instability - TI

Even if a combinational loop does have an even number of inverting elements it can still cause oscillation in an elastic control network. Since the loop has more than one input, both logic one and zero values can be simultaneously injected at different places in the loop. The one and zero values can then race around the loop causing oscillation.

Table 4.4 shows the different lazy fork-join combinations characteristics. The table refers to the network structures of Fig. 4.10.

Research is still in progress to investigate whether the oscillation due to transient instability can be avoided by forcing network-specific timing constraints on the control network. However, a simpler solution, not only for transient instability, but also for deadlock and logical instability, is to use eager forks when needed to cut such combinational cycles. This will be discussed in Sec. 4.6.

Table 4.4: Lazy fork-join combination characterization. All other combinations (2 forks × 10 joins) are noncompliant with the SELF protocol.

	Join	0000	0010	0011	1010	1011	1111
Fork	$\frac{C_r}{C_t}$	$\frac{N,0}{I,0,1}$	$\frac{N,0,1}{I,0,1}$	$\frac{N,0,1}{I,0,1}$	$\frac{N,0,1}{I,1}$	$\frac{\emptyset}{I,1}$	$\frac{I,1}{I,1}$
00	$\frac{I,0}{I,0}$	**LI**	**LI**	**LI**	**LI**	**D**	**D**
01	$\frac{\emptyset}{I,0}$	**TI**	**TI**	**TI**	**D**	**D**	**D**

The following logic is used for the root's $Stall$ signal in all of the lazy forks investigated in this work: $S_l = S_{r1}|S_{r2}$. Similarly, the lazy join elements use $V_r = V_{l1}\&V_{l2}$. Other implementations for these signals that consider flexibility allowed by lazy control buffers is not presented here. However, note that designs with additional logic will increase the probability of combinational loops in component composition.

4.6 Hybrid SELF Protocol

Two lazy forks and six lazy joins, as well as the traditional eager fork, have been proven to be compliant with the SELF channel protocol. Therefore, eager and lazy forks (and joins) can be *correctly* connected together as long as no combinational cycles are formed [10]. Eager forks exhibit no cycles and can achieve better runtime in some systems. However, they consume more power and area than lazy forks. Hence, this work introduces a hybrid SELF implementation, that uses both eager and lazy forks, has no cycles, and achieves the same runtime as an *all* eager implementation. Hybrid implementation should keep minimal number of eager forks in the control network that are necessary for the following reasons:

4.6.1 Cycle Cutting

Lazy fork-join combinations can result in combinational cycles that cause oscillation or deadlock. These cycles can be avoided by replacing lazy forks with eager in places where cycles exist. Cycles can be easily identified either by hand analysis of the control network or through synthesis tools (e.g., `report_timing -loops` command in Design Compiler™ [53]).

$LF01$ enjoys the property that there is no internal path in the fork that connects any of its branch $Stall$s to its corresponding $Valid$. This reduces the number of combinational cycles substantially. Similarly, $LJ1011$ enjoys the property that there is no internal path in the join that connects any of its input channel $Valid$ signals to its corresponding $Stall$. This also reduces the number of cycles substantially. Hence, the fork-join combination of $LF01 - LJ1011$ results in the minimum number of combinational cycles among all the other lazy fork-join combinations. This, in turn, minimizes the need to use eager forks to cut the cycles, resulting in minimizing the total area and power consumption of the hybrid control network.

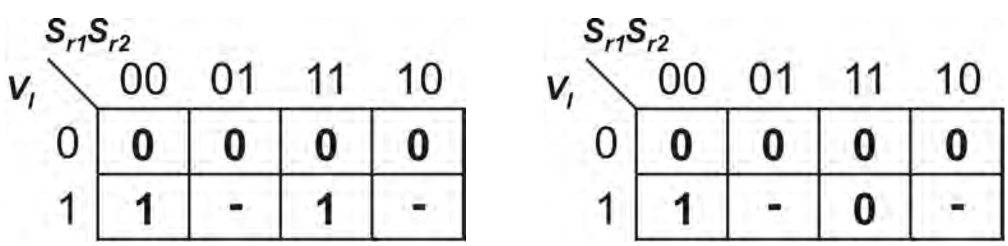
4.6.2 Runtime Boosting

Eager forks can enjoy better performance than lazy due to the early start they provide for ready branches (Sec. 4.3.2). However, this section shows that under some constrained input behavior, a lazy fork can replace an eager fork without any performance loss. In that context, the term $LFork$ will be used to refer to the lazy forks $LF00$ and/or $LF01$.

A 2-output $EFork$ operation will reduce to the KM of Fig. 4.13a if the $EFork$ flip-flops are initialized to logic one and if the following input combinations are avoided (a proof will be provided in Sec. 5.1):

1. $(V_l = 1)\&(S_{r1} = 0)\&(S_{r2} = 1)$.
2. $(V_l = 1)\&(S_{r1} = 1)\&(S_{r2} = 0)$.

The KM of the lazy forks $LF00$ and $LF01$, with the above input combinations avoided, is shown in Fig. 4.13b. Comparing Fig. 4.13a and Fig. 4.13b, it is apparent that, under these conditions, the $EFork$ will behave exactly the same as the lazy forks, except in the case when both branches are stalled simultaneously. One might add a conservative constraint by avoiding such an input as well. However, as the following verification will confirm, when both branches are stalled, the lazy forks will have both branches in the *Idle* (I) state, while the $EFork$ will keep them in the *Retry* (R) state. Since there is no data transfer occurring



(a) $EFork$ (b) $LFork$

Figure 4.13: V_{r1} (or V_{r2}) of the $EFork$ and $LFork$ under some constrained input behavior.

in either states (i.e., I or R), there is no performance advantage of the $EFork$ comparing to the $LFork$ in such a case. Hence, the above stated conditions are sufficient to replace an $EFork$ with $LF00$ or $LF01$ without any performance loss. The conditions will, thus, be referred to as *performance equivalence* conditions, or, for short, *equivalence* conditions.

To verify this argument, the verification setup of Fig. 4.14 is employed. The whole structure is modeled in the symbolic model checker, NuSMV. The input and output channels of both the $EFork$ and $LFork$ are connected to terminal Elastic Buffers (EBs). The EBs are initialized in random states. The $EFork$ input and two output channels are named: L_E (read $Left\_Eager$), $R1\_E$ (read $Right1\_Eager$), and $R2\_E$ (read $Right2\_Eager$), respectively. Similarly, the $LFork$ input and 2 output channels are named: L_L, $R1\_L$, and $R2\_L$, respectively. V and S are prepended to the channel names to indicate the $Valid$ and $Stall$ signals of these channels, respectively.

All the blocks as well as the clock generator are connected synchronously inside NuSMV. The clock changes phase with each unit verification cycle. The Transfer state on the $EFork$ input and output channels are defined as follows:

```
DEFINE L_E_T := VL_E & !SL_E;
DEFINE R1_E_T := VR1_E & !SR1_E;
DEFINE R2_E_T := VR2_E & !SR2_E;
```

Similarly, for the $LFork$:

```
DEFINE L_L_T := VL_L & !SL_L;
```

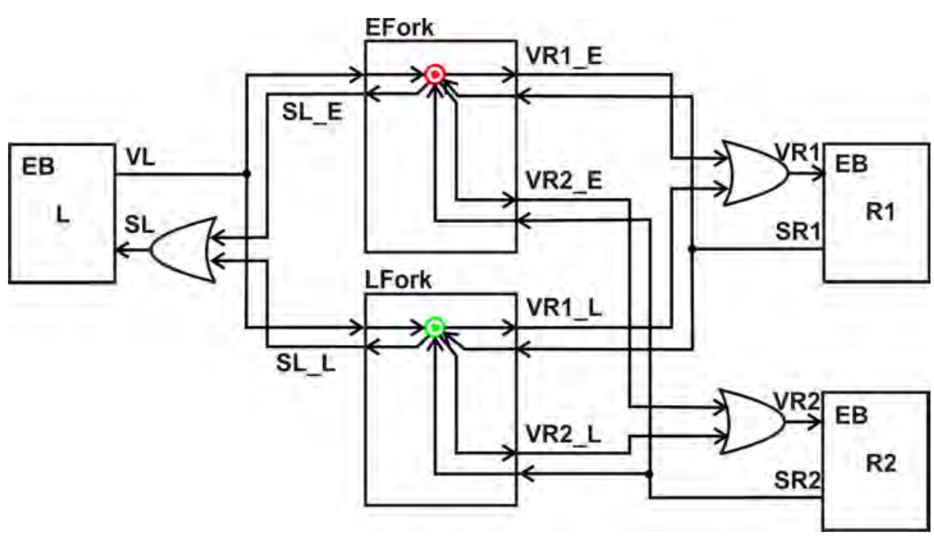


Figure 4.14: $EFork$-$LFork$ performance equivalence verification setup.

```
DEFINE R1_L_T := VR1_L & !SR1_L;
DEFINE R2_L_T := VR2_L & !SR2_L;
```

A performance mismatch may occur if any of the channels in the $EFork$ transfers data while the corresponding channel in the $LFork$ does not. Hence, a channel (i.e., L, $R1$, or $R2$) TOKEN_MISMATCH can be defined as follows:

```
DEFINE L_TOKEN_MISMATCH := (L_E_T xor L_L_T);
DEFINE R1_TOKEN_MISMATCH := (R1_E_T xor R1_L_T);
DEFINE R2_TOKEN_MISMATCH := (R2_E_T xor R2_L_T);
```

A TOKEN_MISMATCH is defined to be the ORing of any channel mismatch:

```
DEFINE TOKEN_MISMATCH := L_TOKEN_MISMATCH | R1_TOKEN_MISMATCH |
R2_TOKEN_MISMATCH;
```

The performance equivalence conditions are defined as following:

```
DEFINE C_1 := !(VL & (SR1 xor SR2));
```

Constraint C_1 is forced by using the NuSMV reserved word INVAR which semantically defines an invariant:

```
INVAR C_1;
```

The performance equivalence property is then verified using PSLSPEC:

```
PSLSPEC never TOKEN_MISMATCH;
```

The property is proven true by the model checker. There is no clock cycle in which any of the $EFork$ channels is in the Transfer state while the corresponding channel in the $LFork$ is not transferring data as well. Hence, under the stated performance equivalence conditions, the $EFork$ and $LFork$ will transfer exactly the same number of tokens, thus, achieving the same performance. The results can be easily extended to n-output forks with $n > 2$, based on the fact that an n-output fork is logically equivalent to concatenated $(n-1)$ 2-output forks.

4.6.3 Eager to Hybrid Conversion Flow

An automatic flow to identify which eager forks satisfy the performance equivalence conditions will be provided in Chapter 5. For the sake of illustration, a simulation-based analysis will be used in this section. In that approach, a closed *eager* control network is simulated and all the fork $Valid$ and $Stall$ patterns are collected and analyzed. An example will be shown in the MiniMIPS case study in Sec. 4.7. Starting with an elastic control network (generated manually or through automatic tools like CNG - Chapter 3), the following flow generates a hybrid SELF implementation (H) of that network:

1. Define the set of all forks in the control network, Φ.
2. Construct a pure eager implementation of the control network, E_1, such that each fork $F \in \Phi$ is an eager fork. Define the set of forks, Φ_p, that do not meet the performance equivalence conditions. Φ_p are the forks that must be implemented as eager to achieve the same runtime as a pure eager implementation of the control network.
3. Construct an intermediate hybrid network, H_1, such that: each fork $F \in \Phi - \Phi_p$ is a lazy fork, and each fork $F \in \Phi_p$ is an eager fork.
4. In H_1, identify the set of forks, Φ_c, that need to be replaced by eager forks to cut the combinational cycles.
5. Build a final hybrid network,H, such that: each fork $F \in \Phi - \Phi_p - \Phi_c$ is lazy, and each $F \in \Phi_p \cup \Phi_c$ is eager.

4.7 MiniMIPS Case Study and Results

MIPS (Microprocessor without Interlocked Pipeline Stages) is a 32-bit architecture with 32 registers, first designed by Hennessey [46]. The MiniMIPS is an 8-bit subset of MIPS, fully described in [1]. Elasticizing the MiniMIPS was illustrated in Sec. 2.2.1. A block diagram of the original clocked MiniMIPS and the hand-optimized elastic version are shown in Figures 2.5 and 2.6, respectively.

4.7.1 Eager Versus Lazy SELF Implementations

Beside their combinational cycle problems, lazy forks can suffer inferior performance comparing to eager when the branch *Stall* patterns do not match. Eager forks provide the earliest possible start for the ready branches (Sec. 4.3.2). To measure this advantage, a different number of bubbles are inserted at the register file outputs (i.e., before registers A and B of Fig. 2.6, simultaneously). Table 4.5 compares the number of clock cycles required by a lazy and by an eager implementations of the MiniMIPS control network to complete the testbench program of [1]. For the lazy protocol, the LF01-LJ0000 combination is used. The behavioral simulations used some timing constraints to avoid possible oscillations. Table 4.5 shows that running the same testbench program on an elastic MiniMIPS processor implemented with lazy SELF takes 32.7% and 58.8% longer runtime than an eager implementation in case of one and three bubbles in the register file path, respectively.

Table 4.5: Time required (in terms of #cycles) by lazy and eager protocols to finish the testbench program in [1]. Bubbles are inserted at the register file outputs.

Fork-join combination	0 Bubbles	1 Bubble	3 Bubbles
Lazy protocol: $LF01$-$LJ0000$	98	195	389
Eager protocol: $EFork$-$LJ0000$	98	147	245
Clocked MiniMIPS	98	-	-

The runtime advantage of the eager versus lazy designs is illustrated in the following example (taken from the MiniMIPS control network of Fig. 2.6). Fig. 4.15 shows a simplified part of the MiniMIPS control network. One bubble is added before the A register, and another one before the B register, labeled $b1$ and $b2$, respectively. Consider the clock cycle when VA and VB go low. $SC1$ will go high through join $JABCI4P$. In FC (assuming $SC2$ is low), VC is high and $SC1$ is high. A lazy FC will invalidate the data at $C2$ (i.e., deassert $VC2$) until $SC1$ goes low again. Hence, no new data token can be written at register $b1$ or $b2$ until the stall condition on $C1$ is removed (i.e., $SC1$ goes low again). On the other hand, an eager FC will validate the data on $C2$ (i.e., assert $VC2$) for the first clock cycle giving $C2$ branch an early start. Hence, new data tokens can be written immediately in registers $b1$ and $b2$ in the following cycle.

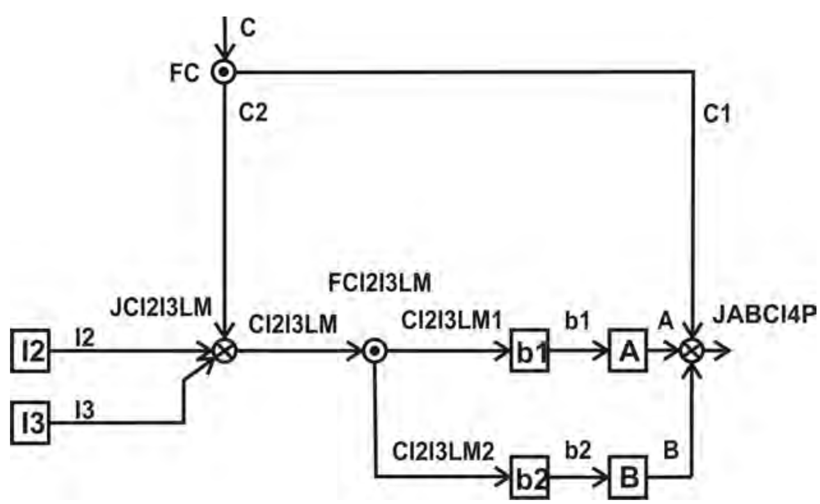


Figure 4.15: A sample structure where eager protocol will have runtime advantage over lazy.

4.7.2 Eager Versus Hybrid SELF Implementations

The hybrid SELF implementation attempts to achieve the same performance of the eager SELF with less area and power consumption. This is done by replacing as many eager forks by lazy as possible. Without loss of generality, both eager and hybrid implementations will be applied to the CNG-generated elastic MiniMIPS control network of Fig. 3.9. This control network achieves the same register-to-register communications as the hand-optimized one in Fig. 2.6 but with two fewer joins and two fewer forks. Furthermore, zero to three bubbles (i.e., EBs that hold no valid data) are inserted at the register file output (i.e., at the inputs of A and B registers, simultaneously). In practice, this might be done, for example, to accommodate a high latency register file without affecting the functionality of the whole system.

The flow of Sec. 4.6.3 will be followed to construct the hybrid implementation. Starting with an all eager implementation of the closed control network of Fig. 3.9 (call it E_1), the sample testbench program of [1] is run. The simulation waveforms of each eager fork in the network are analyzed. $EFork$s whose input behavior does not meet the performance equivalence conditions (of Sec. 4.6.2) are then identified. These are the forks that must be implemented as eager in the (to-be) hybrid control network in order to maintain the same performance as the all eager network. The set of these forks will be called Φ_p.

Analysis of the simulation waveforms of the MiniMIPS case (with 0 to 3 bubbles at the register file output) shows that all forks except FC and FL receive $Valid$ and $Stall$ patterns that meet the performance equivalence conditions. Hence, all the forks except FC and FL can be safely implemented as lazy forks without any performance loss. For FC, repetitive $Stall$ patterns similar to those shown in Fig. 4.16 are observed. The numbered columns in Fig. 4.16 represent the clock cycles. The red 0s and 1s are the branch $Stall$ signal values at the corresponding clock cycles. It is obvious that the $Stall$ patterns at $C1$ and $C3$ meet the conditions of Sec. 4.6.2 (they do not stall at all). Hence, branches $C1$ and $C3$ can be safely connected through a lazy fork (call it FC_1_3). Similarly, the $Stall$ patterns at branches $C2$ and $C4$ meet the replacement conditions (their $Stall$ patterns match). Hence, branches $C2$ and $C4$ can also be connected through another lazy fork (call it FC_2_4). To maintain the same runtime as an *all* eager implementation, FC_1_3 and FC_2_4 must be connected through an eager fork (call it FC_i) since their corresponding $Stall$ patterns do not match. The resultant hybrid FC implementation is shown in Fig. 4.17. EF and LF in the figure refer to eager and lazy forks, respectively. Similarly, based on the simulation waveform

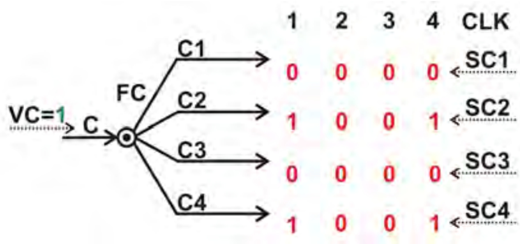


Figure 4.16: *Stall* patterns at the branches of FC in the presence of bubbles.

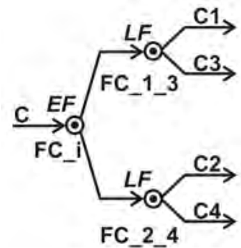


Figure 4.17: Hybrid implementation of FC.

analysis, branches 1 and 2 of FL could be connected through a lazy fork (FL_1_2). FL_1_2 must be connected eagerly to the third branch of FL to maintain the runtime of an *all* eager implementation.

As stated in Sec. 4.6.3, a hybrid network (call it H_1) is now constructed. All forks of H_1 are implemented as lazy except those in set Φ_p (i.e., that do not meet the equivalence conditions). H_1 typically involves combinational cycles formed by the connection of lazy forks and joins. To cut the cycles in H_1, more forks have to be implemented as eager (call this set of forks Φ_c). The number of forks in Φ_c depend on the lazy fork and join combination used. Some lazy fork-join combinations exhibit more cycles than others and, hence, require more eager fork replacements. For example, when the lazy combination $LF01 - LJ1011$ is used, only 2 extra forks have to be implemented as eager to cut the cycles, namely, FL_1_2 and FC_2_4. The MiniMIPS control network is implemented using all the correct 12 lazy fork-join combinations (with some eager fork replacements). The network is also implemented with an *all* eager control network.

Table 4.6 shows the synthesis results. The Artisan academic library for IBM® 65nm library is used for physical design. The MiniMIPS control network has been synthesized separately from the data path. All area and power numbers in Table 4.6 are for *the control network only*. All combinations have passed post synthesis simulation (with 0 to 3 bubbles). The MiniMIPS testbench program in [1] is used to validate correctness. Column 1 in Table 4.6 lists the different combinations (sorted by their area). Column 2 lists the set of all forks that have to be implemented as eager (to both maintain the performance and cut the cycles). The column also shows the ratio of the number of $EFork$s used to the total number of forks in the network. For counting the forks, it is assumed that an n-output fork counts as $n - 1$ concatenated 2-output forks. Unsurprisingly, $E - LF01 - LJ1011$ needs the least number of eager fork replacements (see Sec. 4.6.1), tying with $E - LF00 - LJ1011$

Table 4.6: Area, power, and runtime of the MiniMIPS control network using different hybrid (eager/lazy) SELF implementations.

Combination	nEForks/nForks: Eager Forks Used	nCycles	Area (μm^2)	Power @ 4ns $\frac{P_{dyn}}{P_{leakage}}$ (μW)			Runtime (Cycles)		
				0 B	1 B	3 B	0 B	1 B	3 B
$E-LF00-LJ1011$	4/12:*SOME BRANCHES OF FC, FL*	0	513.0	$\frac{58.187}{1.980}$	$\frac{164.284}{1.990}$	$\frac{122.720}{1.992}$	98	147	245
$E-LF01-LJ1111$	6/12:*FC, FL, FBCP*	0	575.4	$\frac{65.626}{2.339}$	$\frac{188.094}{2.307}$	$\frac{140.389}{2.278}$	98	147	245
$E-LF01-LJ1011$	4/12:*SOME BRANCHES OF FC, FL*	0	588.0	$\frac{58.187}{2.640}$	$\frac{183.991}{2.536}$	$\frac{134.636}{2.542}$	98	147	245
$E-LF01-LJ0000$	6/12:*FC, FL, FBCP*	0	634.2	$\frac{65.626}{2.739}$	$\frac{194.001}{2.663}$	$\frac{143.822}{2.599}$	98	147	245
$E-LF00-LJ1111$	8/12:*FC, FL, FBCP, FMem, FABCI4P*	0	639.0	$\frac{74.475}{2.525}$	$\frac{206.882}{2.514}$	$\frac{155.145}{2.499}$	98	147	245
$E-LF01-LJ0011$	6/12:*FC, FL, FBCP*	0	646.8	$\frac{65.626}{2.738}$	$\frac{192.545}{2.672}$	$\frac{143.065}{2.617}$	98	147	245
$E-LF01-LJ1010$	6/12:*FC, FL, FBCP*	0	649.8	$\frac{64.710}{2.761}$	$\frac{197.261}{2.691}$	$\frac{145.481}{2.631}$	98	147	245
$E-LF01-LJ0010$	6/12:*FC, FL, FBCP*	0	653.4	$\frac{65.635}{2.685}$	$\frac{191.208}{2.642}$	$\frac{142.149}{2.598}$	98	147	245
$E-LF00-LJ0000$	8/12:*FC, FL, FBCP, FMem, FABCI4P*	0	683.4	$\frac{74.933}{2.825}$	$\frac{196.338}{2.762}$	$\frac{148.919}{2.713}$	98	147	245
$E-LF00-LJ0011$	8/12:*FC, FL, FBCP, FMem, FABCI4P*	0	695.4	$\frac{74.933}{2.790}$	$\frac{198.957}{2.742}$	$\frac{150.580}{2.699}$	98	147	245
$E-LF00-LJ0010$	8/12:*FC, FL, FBCP, FMem, FABCI4P*	0	698.4	$\frac{74.475}{2.853}$	$\frac{202.539}{2.838}$	$\frac{152.374}{2.811}$	98	147	245
$E-LF00-LJ1010$	8/12:*FC, FL, FBCP, FMem, FABCI4P*	0	704.4	$\frac{73.101}{2.887}$	$\frac{205.521}{2.867}$	$\frac{153.914}{2.844}$	98	147	245
$EFork-LJ0000$	12/12:*ALL*	0	752.4	$\frac{86.158}{2.914}$	$\frac{221.921}{2.875}$	$\frac{168.807}{2.842}$	98	147	245

in this specific network. Column 3 lists the number of combinational cycles in the control network (after eager fork replacements), which is zero for all of them. Column 4 lists the synthesis area. $E-LF00-LJ1011$ requires minimum area among all with 31.8% reduction comparing to an all eager implementation. $E-LF01-LJ1111$ comes second. Note that even though $E-LF01-LJ1111$ uses more $EFork$s than $E-LF01-LJ1011$, it requires less area. This can be attributed to the logic simplicity of $LJ1111$ (Fig. 4.8) in comparison with $LJ1011$ (Fig. 4.9), making it easier to optimize the former during synthesis.

Column 5 lists the dynamic and leakage power consumption reported by the synthesis tool. Power is calculated with different number of bubbles inserted at the output of the register file. To accurately estimate the power, the synthesized netlist is simulated and an *saif* file is generated. That file is then read by the synthesis tool to calculate the power. Synthesis and simulation are done at 4 ns clock period for all the implementations. $E-LF00-LJ1011$ consumes the least power among all with up to 32.5% and 32.1% dynamic and leakage power reduction comparing to an eager implementation. $E-LF01-LJ1011$ comes second.

Finally, column 6 lists the required runtime (in terms of number of clock cycles) to finish the testbench program in [1]. The 12 hybrid networks all achieve the same runtime as the *all* eager implementation.

The elastic MiniMIPS constructed using the hybrid control network implementations listed in Table 4.6 can tolerate 0 - 3 bubbles in the register file path, and still achieve the same runtime as the *all* eager implementation. A direct comparison with the ordinary clocked MIPS cannot be established since inserting bubbles in the latter will change some channel latencies causing it to fail. For the normally clocked MiniMIPS to handle bubbles (or variable latency interfaces) over its channels, several changes in the datapath may be required (e.g., implementing FSMs at channel receiver ends to wait until valid data arrive, some mechanism to propagate this information to the rest of the system, a stalling mechanism, etc.). On the other hand, and by its definition, synchronous elasticization inherently achieves such a goal.

Table 4.7 shows the cost of achieving this required elasticity using the SELF protocol in an *all* eager and a hybrid ($E-LF00-LJ1011$) implementations. The results in the table are synthesis numbers for the whole MiniMIPS (not just the control network). Since the normally clocked MiniMIPS cannot directly tolerate register file bubbles, therefore and for the sake of comparison, no bubbles are added in either the normally clocked or the

Table 4.7: Elasticity area and power overheads of an *all* eager and a hybrid (eager/lazy) SELF implementations of the MiniMIP processor.

Implementation		Area		P_{dyn} @ 4ns		P_{leak}	
		μm^2	over.%	μW	over.%	μW	over.%
Normally clocked MiniMIPS	Flip-flop based	2617.2		446.247		8.850	
	Latch based	2642.4		380.466		9.504	
Elastic clocked MiniMIPS	*All* eager ($EFork - LJ0000$)	3385.2	28.1%	474.465	24.7%	12.681	33.4%
	Hybrid ($E - LF00 - LJ1011$)	3136.2	18.7%	437.977	15.1%	11.686	23.0%

elastic MiniMIPS (even though the elastic MiniMIPS *can* tolerate the register file bubbles). Two implementations for the clocked MiniMIPS are listed. The first is flip-flop (FF) based. In the second one, each FF is replaced by a master-slave latch pair. The latches used in both the latch based and the elastic MiniMIPS are selected from manually synthesized and optimized templates that are protected during synthesis with set_size_only attributes. The FF based design is completely synthesized by DC. In this specific design and cell library, the latch based design consumed more area and leakage power but less dynamic power. Without loss of generality, overhead percentages (over. %) of elastic versions are with respect to the latch based design. Please note that if more bubbles (or variable latency interfaces) are required in the MiniMIPS, more lazy forks (in the hybrid implementation) may need to be replaced by $EFork$s to keep the same runtime as the *all* eager implementation, resulting in more area and power.

CHAPTER 5

UTILIZING THE ULTRA SIMPLE FORK AND CONTROLLER MERGING[1]

This chapter introduces two more area and power reduction techniques in synchronous elastic control networks, namely, utilizing the novel Ultra Simple Fork ($USFork$) and controller merging. The two techniques are fully automated and have been integrated in a tool called HGEN.

Last chapter introduced the concept of replacing expensive eager forks with lazy in places where eagerness does not provide any runtime advantage. Though the technique was shown to substantially reduce the area and power of a control network, the idea of hybrid (eager and lazy) control network can be further exploited. The flow of Sec. 4.6.3 showed that some of the eager forks are kept in the lazy-eager hybrid network for the sole purpose of cutting the combinational cycles (formed by lazy forks and joins). This motivates the search for a new fork structure that is, unlike lazy forks, does not form combinational cycles when combined with lazy joins in any arbitrary connection. Similar to lazy forks, the new sought design should also be cheap in area and power, and under similar constrained input behavior can also be substituted for eager forks without any performance loss.

Sec. 5.1 introduces the Ultra Simple Fork ($USFork$). As the name implies, the $USFork$ implementation has no logic gates - just wired connections. The $EFork$ transition diagram is computed and the conditions under which an $EFork$ can be replaced by a $USFork$ without any performance loss are formally driven. The transformation guarantees that, under such conditions, the $USFork$ will schedule exactly the same state transitions as the $EFork$ over all its channels, thus maintaining the same runtime. Unlike lazy SELF implementations, utilizing the $USFork$ does not create combinational cycles when connected to lazy joins. In essence, the proposed approach selectively replaces the *redundant* $EFork$s in a control network with $USFork$s resulting in a hybrid network where both $EFork$s and $USFork$s are

[1]This is a revised and extended version of a paper originally published in [52].

used. The resultant network has the same runtime as the *all* eager network with reduced area and power consumption.

The second contribution of this chapter is automatically merging equivalent controllers. Sec. 5.2 investigates the conditions under which multiple SELF controllers can be merged into one controller. The transformation reduces the control network area and power overhead and is limited only by the physical placement constraints. SELF controller clustering has previously been reported in [50]. However, their approach requires both the control network and its environment to have static (and known) latencies. On the other hand, the approach proposed in this work can handle situations where the environment abstract is not available or required to be flexible. It can also handle designs with variable latency units.

The above two transformations have been integrated in a fully automated tool, HGEN (Sec. 5.4). Hybrid GENerator (HGEN) selectively replaces redundant $EFork$s with $USFork$s and, optionally, merges equivalent controllers. HGEN uses IBM® 6thSense tool [51] as an embedded verification engine. Comparing to the methodology used in published work on a MiniMIPS processor case study, HGEN shows up to 36.9% and 31.3% savings in area and power, respectively, due to utilizing $USFork$s. If the physical placement allows for controller merging, the resultant control network shows up to 62.8% and 54.1% savings in area and power, respectively. HGEN also shows *at least* 32% saving in the number of $EFork$s in s382 ISCAS benchmark. More reduction is possible if the physical placement allows for controller merging. Thanks to the advance in synchronous verification technology, HGEN runs within seconds or a few minutes (for all this chapter examples). This makes the proposed approach suitable for tight time-to-market constraints.

5.1 Eager to Ultra Simple Fork Transformation

An overview of the SELF protocol was given in Sec. 2.1. An Elastic Buffer (EB) block diagram and the protocol state transition graph are drawn in Figures 2.1 and 2.2, respectively.

5.1.1 Eager SELF Protocol

An eager SELF implementation uses eager forks ($EFork$s) and lazy joins. Study of lazy joins (and forks) are given in Chapter 4. Fig. 5.1 shows a 2-output-channel $EFork$ proposed in [9]. Once a ($Valid$) data token is available at an $EFork$ stem, it will immediately pass it to all its branches that are ready to receive (i.e., their corresponding $Stall$ signals are low).

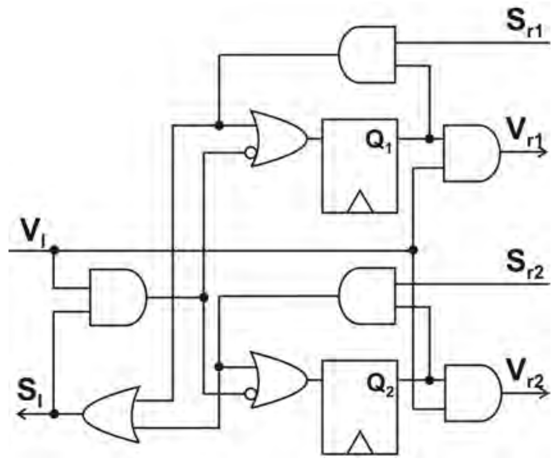


Figure 5.1: A 2-output-channel $EFork$.

Meanwhile, the $EFork$ will *Stall* until all its branches receive the data token. This gives an early start to the branches that are ready.

5.1.2 Eager Fork State Diagram

A 2-output-channel $EFork$ has 3 terminal channels, namely, L ($Left$), R_1 ($Right_1$), and R_2 ($Right_2$). L consists of signals V_l and S_l. Similarly, R_1 consists of V_{r1} and S_{r1}, and R_2 of V_{r2} and S_{r2}. In order to compute the state diagram of the $EFork$, the behavior allowed by the SELF protocol over the fork 3 channels must be taken into account. Hence, the desired state diagram is obtained by composing the simple (2 flip-flop based) 4-state diagram of the $EFork$ circuit of Fig. 5.1 with the SELF transition diagram of Fig. 2.2 (over the three terminal channels). The $EFork$ state table and diagram are depicted in Table 5.1 and Fig. 5.2, respectively. In this diagram, the inputs V_l, S_{r1}, and S_{r2} are part of the state vector (along with the flip-flop outputs, Q_1 and Q_2). To simplify the notation, the state vector takes the following format: $<Q_1,Q_2,L,R_1,R_2>$, where L, R_1, and R_2 carry the corresponding channel status (i.e., I, T, or R). States with dot inside are reset states. Some of the transitions (and states) are not allowed (or reached) because of the SELF protocol constraints, and hence, omitted from the diagram. Most of the transition labels are omitted from Fig. 5.2 for brevity.

5.1.3 Input Behavior Constraints

In a 2-output-channel $EFork$, the input vector, I, is a 3-tuple of signals $< V_l, S_{r1}, S_{r2} >\in \{0,1\}^3$. Subscript n is added to I and the 3 signals to denote the value at clock cycle n. S^I is

Table 5.1: The $EFork$ state table.

Current State						Next State Inputs			Next State					
s_i	Q_1	Q_2	L	R_1	R_2	V_l	S_{r1}	S_{r2}	s_i	Q_1	Q_2	L	R_1	R_2
s_0	1	1	I	I	I	0	-	-	s_0	1	1	I	I	I
						1	0	0	s_1	1	1	T	T	T
						1	0	1	s_3	1	1	R	T	R
						1	1	0	s_4	1	1	R	R	T
						1	1	1	s_2	1	1	R	R	R
s_1	1	1	T	T	T	0	-	-	s_0	1	1	I	I	I
						1	0	0	s_1	1	1	T	T	T
						1	0	1	s_3	1	1	R	T	R
						1	1	0	s_4	1	1	R	R	T
						1	1	1	s_2	1	1	R	R	R
s_2	1	1	R	R	R	0	-	-	Illegal Transition					
						1	0	0	s_1	1	1	T	T	T
						1	0	1	s_3	1	1	R	T	R
						1	1	0	s_4	1	1	R	R	T
						1	1	1	s_2	1	1	R	R	R
s_3	1	1	R	T	R	0	-	-	Illegal Transition					
						1	0	0	s_5	0	1	T	I	T
						1	0	1	s_6	0	1	R	I	R
						1	1	0	s_5	0	1	T	I	T
						1	1	1	s_6	0	1	R	I	R
s_4	1	1	R	R	T	0	-	-	Illegal Transition					
						1	0	0	s_7	1	0	T	T	I
						1	0	1	s_7	1	0	T	T	I
						1	1	0	s_8	1	0	R	R	I
						1	1	1	s_8	1	0	R	R	I
s_5	0	1	T	I	T	0	-	-	s_0	1	1	I	I	I
						1	0	0	s_1	1	1	T	T	T
						1	0	1	s_3	1	1	R	T	R
						1	1	0	s_4	1	1	R	R	T
						1	1	1	s_2	1	1	R	R	R
s_6	0	1	R	I	R	0	-	-	Illegal Transition					
						1	0	0	s_5	0	1	T	I	T
						1	0	1	s_6	0	1	R	I	R
						1	1	0	s_5	0	1	T	I	T
						1	1	1	s_6	0	1	R	I	R
s_7	1	0	T	T	I	0	-	-	s_0	1	1	I	I	I
						1	0	0	s_1	1	1	T	T	T
						1	0	1	s_3	1	1	R	T	R
						1	1	0	s_4	1	1	R	R	T
						1	1	1	s_2	1	1	R	R	R
s_8	1	0	R	R	I	0	-	-	Illegal Transition					
						1	0	0	s_7	1	0	T	T	I
						1	0	1	s_7	1	0	T	T	I
						1	1	0	s_8	1	0	R	R	I
						1	1	1	s_8	1	0	R	R	I

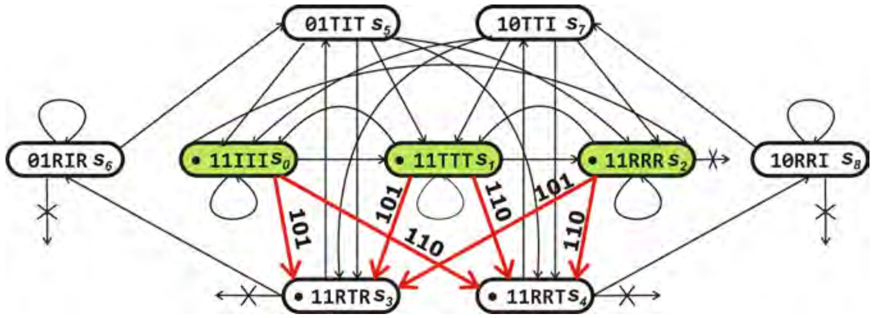


Figure 5.2: The $EFork$ state diagram.

defined to be an infinite sequence of input vectors ordered by the clock index. Hence, $S^I[n] = I_n$. The total input behavior, B_T^I, is defined to be the set of all input sequences. Some of the input sequences are not allowed by the SELF protocol. For example, the following sequence will cause an R to I transition on the L channel: $<< 1,0,0 >,< 1,1,1 >,< 0,1,1 >,.. >$. The set of all sequences that are excluded for violating the SELF protocol will be denoted as E_P^I. Nonetheless, in this section, some of the sequences will also be excluded due to other constraints. Under Constraint C_i, the allowed input behavior, B_{Ci}^I, is, thus, given by the following equation:

$$B_{Ci}^I = B_T^I - (E_P^I \cup E_{Ci}^I) \tag{5.1}$$

where E_{Ci}^I is the set of sequences excluded from the input behavior for violating constraint C_i. The words property and constraint will be used interchangeably as long as the context is clear. In this work notation, *constraint* x constrains the input behavior such that *property* x holds. Properties (and constraints) will be specified using the Property Specification Language (PSL) syntax [60] unless mentioned otherwise.

Definition 5.1. Protocol Equivalence Two forks are said to be SELF protocol equivalent (or, for short, just protocol equivalent), if given the same input sequences, their terminal channels go through the same SELF state transitions.

Theorem 5.1. *The $EFork$ of Fig. 5.1 is protocol equivalent to the $USFork$ of Fig. 5.3 if the fork input behavior is constrained such that the following property is true in the former: ALWAYS $s_0|s_1|s_2$, where s_i is 1 if the $EFork$ is in state s_i $\forall i \in \{0,1,2\}$ (Refer to Fig. 5.2).*

Proof. Figures 5.4 and 5.5 show the Karnaugh maps of V_{r1} (or V_{r2}) and S_l, respectively, in states s_0 - s_2. By using simple logic optimization, the following equations can be obtained:

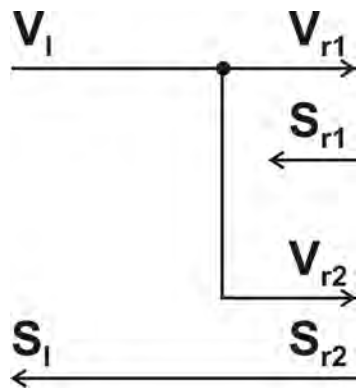


Figure 5.3: A 2-output-channel $USFork$.

$$V_{r1} = V_l,\ V_{r2} = V_l,\ S_l = S_{r1} \text{ or } S_l = S_{r2} \tag{5.2}$$

The $USFork$ of Fig. 5.3 exactly implements these equations. ■

Notice that the choice to connect S_l to either S_{r1} or S_{r2} in Fig. 5.3 is irrelevant. The reason is, as will be shown in Theorem 5.2, under the input constraint specified in Theorem 5.1, S_{r1} and S_{r2} are always identical. They may differ only when V_l is zero, in which case the L channel is in the idle (I) state whatever the value of S_l.

Definition 5.2. Equivalent Constraints Referring to Equation 5.1, two constraints C_i and C_j are said to be equivalent if $B^I_{Ci} = B^I_{Cj}$ (i.e., the allowed input behavior under constraint i is the same as the allowed input behavior under constraint j).

In other words, two properties i and j (also referred to as constraints) are equivalent if constraining the input behavior such that property i holds, will also cause property j to hold, and vice versa.

Similarly, n properties (also referred to as constraints) are equivalent if $\forall i, j \in \{1, 2, .., n\}$: property i and property j are equivalent.

V_l \ $S_{r1}S_{r2}$	00	01	11	10
0	0	0	0	0
1	1	-	1	-

Figure 5.4: V_{r1} (same for V_{r2}) in states s_0 to s_2.

V_l \ $S_{r1}S_{r2}$	00	01	11	10
0	-	-	-	-
1	0	-	1	-

Figure 5.5: S_l in states s_0 to s_2.

Theorem 5.2. *The following three properties (also referred to as constraints) are equivalent:*

1. *ALWAYS* $s_0|s_1|s_2$*, where* s_i *is 1 if the* $EFork$ *is in state* s_i $\forall i \in \{0, 1, 2\}$*.*
2. *NEVER* $V_l\&(S_{r1}$ *xor* $S_{r2})$*.*
3. *ALWAYS* V_{r1} *xnor* V_{r2}*.*

Proof. It will be proved that constraining the input behavior such that any one property holds will cause the other two to hold as well.

C. 1 If the input behavior is constrained such that $EFork$ operates in s_0 to s_2 only (i.e., C. 1 holds), then, as shown in s_0 to s_2 entries in Table 5.1, S_{r1} never differs from S_{r2} while V_l is one (C. 2), and V_{r1} is always the same as V_{r2} (C. 3).

C. 2 States s_0 to s_4 are reset states. However, if the input behavior is constrained such that S_{r1} is always the same as S_{r2} while V_l is one, then the $EFork$ can reset only in any of the states s_0 to s_2, exclusively. Besides, it will stay in these states since all the red transitions in Fig. 5.2 will not fire. Hence, C. 1 will be satisfied, and subsequently, C. 3 will be satisfied as well.

C. 3 If the input behavior is constrained such that only those input sequences that cause V_{r1} to be always the same as V_{r2} are allowed, then the $EFork$ will never move to any of the states s_5 to s_8 (where V_{ri}s differ). Moreover, the $EFork$ will not reset in states s_3 or s_4 since all the input sequences that go through them must also go through states s_5 to s_8 (no other transition is permitted). And the latter sequences are excluded by the constraint. Hence, forcing C. 3 will cause the $EFork$ to reset and operate in states s_0 to s_2 only. Therefore, both C. 1 and C. 2 will be satisfied.

■

Definition 5.3. Equivalence Constraint A constraint on the input behavior that causes the $EFork$ to be protocol equivalent to the $USFork$ is called an equivalence constraint.

Thus, each of the three constraints of Theorem 5.2 is an equivalence constraint. When the context is clear, an equivalence constraint will also be referred to as an equivalence *condition*. Following, it will be proved that *any* of these three conditions allow us to find the *maximum* number of candidate $EFork$s in a network that can be replaced by $USFork$s.

Definition 5.4. Minimal Equivalence Constraint An equivalence constraint is minimal if it allows for maximum behavior of the inputs beyond which an $EFork$ will fail to be protocol equivalent to a $USFork$.

Theorem 5.3. *Each of the three constraints of Theorem 5.2 is minimal.*

Proof. If C. 1 is not minimal, then the $EFork$ is allowed to operate in other states beside s_0 to s_2 and still be protocol equivalent to the $USFork$. However, this is not the case. In states s_5 to s_8, the $EFork$ V_{r1} and V_{r2} differ. Thus, the $EFork$ R_1 and R_2 channels will be in protocol states that cannot be provided (or scheduled) by the $USFork$ (where V_{r1} is tied to V_{r2} - Fig. 5.3). Similarly, if the $EFork$ operates in states s_3 or s_4, it has no other legal transition but to move to one of the states s_5 to s_8 (which as was argued break the protocol equivalence). Hence, C. 1 is a minimal constraint.

Since the three constraints are equivalent (from Theorem 5.2), therefore, they constrain the input behavior similarly. It follows that, since C. 1 is minimal, C. 2 and C. 3 are minimal as well. ■

To check for $EFork$ replacements, the $EFork$ can be checked against *any* of the three properties. However, without loss of generality, only property 3 will be used, hereafter. Would two branches of an $EFork$ satisfy property 3, the $EFork$ can be correctly replaced by a $USFork$. Being a minimal *condition* for equivalence (as proven in Theorem 5.3), it maximizes the chance of finding candidate $EFork$s for replacement.

Replacing an $EFork$ with a $USFork$ cannot create combinational cycles, since there are no internal paths inside the $USFork$ that connects $Valid$ to $Stall$ ports (or vice versa). This is an advantage over lazy forks where such internal paths do exist. Besides, since (under the mentioned *conditions*) the $USFork$ is protocol equivalent to the $EFork$, they both schedule the same protocol state transitions over their terminal channels. Hence, they will both have the same runtime. Finally, replacing an $EFork$ with a $USFork$ should never degrade the control network maximum frequency. It can actually boost it since the $USFork$ cuts from *all* the $EFork$ internal path delays (by removing the logic gates), and it does not add any new paths.

5.1.4 Verification

To verify Theorems 5.1 and 5.2, the setup of Fig. 5.6 is used. The whole structure is modeled and passed to a symbolic model checker, NuSMV [59]. The $EFork$ and $USFork$ inputs (i.e., V_l, S_{r1}, and S_{r2}) are driven from Protocol Terminals (PTs). A PT can simply be an EB controller initialized in a random state. It can also be implemented as a SELF channel with protocol constraints forced on its $Valid$ and $Stall$ signals. In this section the first approach is used, the other will be used later in the chapter. The outputs of the $EFork$

and $USFork$ have suffixes of $\_E$ and $\_US$, respectively. They are ORed together to form the corresponding signals over the three terminal channels (i.e., L, R_1, and R_2). $Valid$ and $Stall$ signals on channel L will be denoted as VL and SL, respectively. Same for the other channels. For example, $VR1$ is the ORing of $VR1\_E$ and $VR1\_US$.

The shown blocks as well as a clock generator are all connected synchronously in NuSMV. The clock changes phase with every verification cycle. The I, T, and R states of the $EFork$ L channel (denoted as L_E) are defined as follows:

```
DEFINE L_E_I := !VL_E;
DEFINE L_E_T := VL_E & !SL_E;
DEFINE L_E_R := VL_E & SL_E;
```

And on the USFork:

```
DEFINE L_US_I := !VL_US;
DEFINE L_US_T := VL_US & !SL_US;
DEFINE L_US_R := VL_US & SL_US;
```

The other states of the other 2 channels are defined similarly for both $EFork$ and $USFork$. The $EFork$ states of operation are also defined as follows:

```
-- s0 = 11III
DEFINE S0_E := EFork.q1 & EFork.q2 & L_E_I & R1_E_I & R2_E_I;
-- s1 = 11TTT
DEFINE S1_E := EFork.q1 & EFork.q2 & L_E_T & R1_E_T & R2_E_T;
```

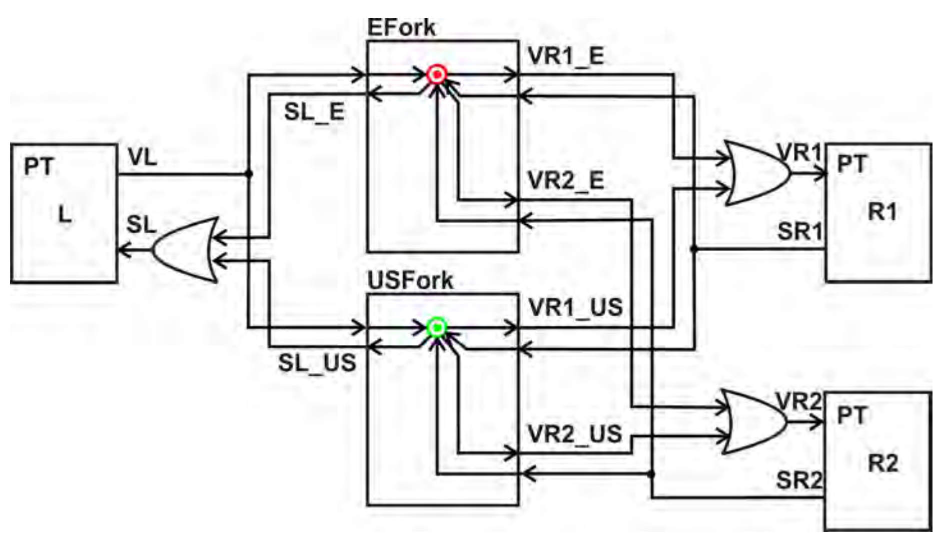


Figure 5.6: $EFork$-$USFork$ equivalence verification setup.

```
-- s2 = 11RRR
DEFINE S2_E := EFork.q1 & EFork.q2 & L_E_R & R1_E_R & R2_E_R;
-- s3 = 11RTR
DEFINE S3_E := EFork.q1 & EFork.q2 & L_E_R & R1_E_T & R2_E_R;
-- s4 = 11RRT
DEFINE S4_E := EFork.q1 & EFork.q2 & L_E_R & R1_E_R & R2_E_T;
```

Mismatches over the three channels are defined as follows:

```
DEFINE L_MISMATCH := (L_E_I xor L_US_I) | (L_E_T xor L_US_T) | (L_E_R xor
L_US_R);
DEFINE R1_MISMATCH := (R1_E_I xor R1_US_I) | (R1_E_T xor R1_US_T) | (R1_E_R
xor R1_US_R);
DEFINE R2_MISMATCH := (R2_E_I xor R2_US_I) | (R2_E_T xor R2_US_T) | (R2_E_R
xor R2_US_R);
DEFINE MISMATCH := L_MISMATCH | R1_MISMATCH | R2_MISMATCH;
```

Finally, the three constraints (or properties) are defined as follows (without temporal qualifiers):

```
DEFINE C_1 := S0_E | S1_E | S2_E;
DEFINE C_2 := !(VL & (SR1 xor SR2));
DEFINE C_3 := VR1_E xnor VR2_E;
```

A constraint is forced through the NuSMV INVAR reserved word, and a property is verified using PSLSPEC. In the following code, only one constraint is forced at a time. To verify Theorem 5.1:

```
INVAR C_1;
PSLSPEC never MISMATCH; -- True
```

Similarly, Theorem 5.2 Constraint. 1 is verified as follows:

```
INVAR C_1;
PSLSPEC always C_2; -- True
PSLSPEC always C_3; -- True
```

And Theorem 5.2 Constraint. 2:

```
INVAR C_2;
PSLSPEC always C_1; -- True
PSLSPEC always C_3; -- True
```

And Constraint. 3:

```
INVAR C_3;
PSLSPEC always C_1; -- True
PSLSPEC always C_2; -- True
```

5.1.5 Multi-output-channel $EFork$s

Theorem 5.6 extends the results of the previous theorems to multi-output-channel $EFork$s.

Lemma 5.4. *An n-output-channel $EFork$ is protocol equivalent to concatenated (n-1) 2-output-channel $EForks$.*

Proof. Proof is trivial and omitted for brevity. ■

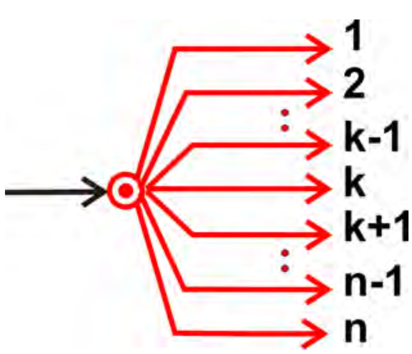
Lemma 5.5. *An n-output-channel $USFork$ is protocol equivalent to concatenated (n-1) 2-output-channel $USForks$.*

Proof. Proof is trivial and omitted for brevity. ■

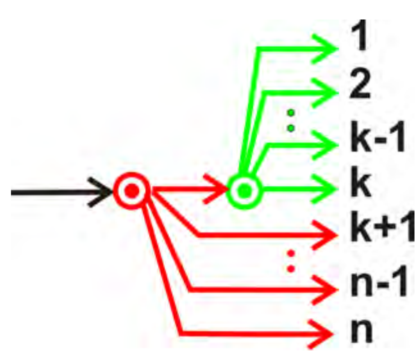
Theorem 5.6. *If, in Fig. 5.7, $\forall i, j \in \{1, 2, .., k\}$ the following property holds: ALWAYS (V_{ri} xnor V_{rj}), then the hybrid fork ($HFork$) of Fig. 5.7b is protocol equivalent to the eager fork ($EFork$) of Fig. 5.7a.*

Proof. The proof follows from Lemmas 5.4 and 5.5 and Theorems 5.1 and 5.2, and was omitted for brevity. ■

Red forks in Fig. 5.7 are $EFork$s while green are $USFork$s.



(a) Eager fork ($EFork$).



(b) Hybrid fork ($HFork$).

Figure 5.7: Eager to hybrid transformation of multi-output forks.

5.2 Elastic Buffer Controller Merging

In a typical control network, some Elastic Buffer Controllers (EBCs) may activate their corresponding latches at similar schedules. This can allow for possible merging of these controllers into one controller that feeds them all (as much as the physical placement permits). In this section and the following, a framework is provided for finding and merging such controllers in any control network; including open networks (i.e., when the environment abstract is not available or required to be flexible) as well as networks incorporating variable latency units.

Definition 5.5. Functional Equivalence Two structures are said to be functionally equivalent, if given the same input sequences, they produce the same output sequences.

Theorem 5.7. *If the n $EBCs$ of Fig. 5.8a are initialized in the same state and the environment behavior is constrained such that the following two properties (also referred to as constraints) are true $\forall i, j \in \{1, 2, ..n\}, i \neq j$:*

1. *ALWAYS (V_{li} xnor V_{lj}).*
2. *ALWAYS (S_{ri} xnor S_{rj}).*

Then, the structure of Fig. 5.8b is functionally equivalent to the one in Fig. 5.8a.

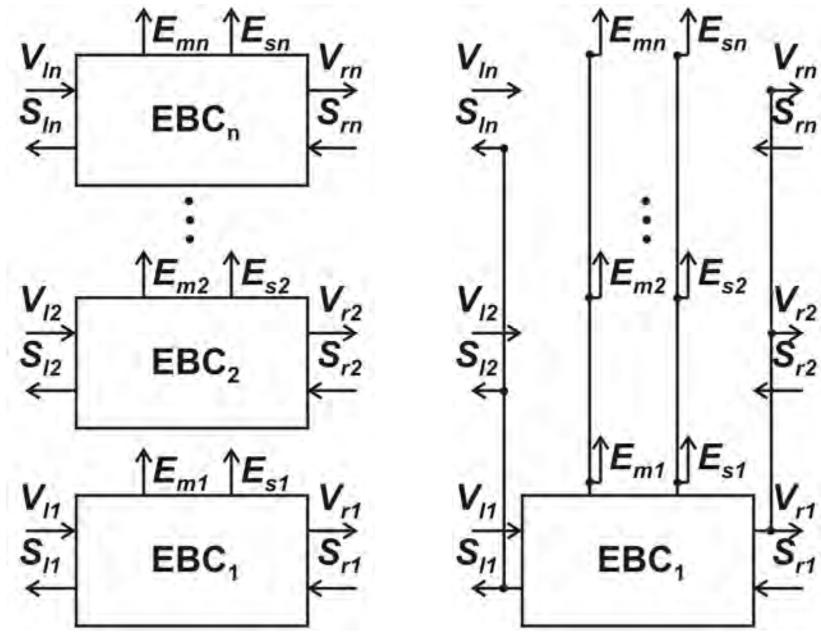
Proof. Trivial. It is easy to show under the conditions of the theroem, that the following properties will also hold: ALWAYS (V_{ri} xnor V_{rj}), ALWAYS (S_{li} xnor S_{lj}), ALWAYS (E_{mi} xnor E_{mj}), and ALWAYS (E_{si} xnor E_{sj}). ■

EBC merging is limited only by the physical placement constraints. Authors of [50] proposed a technique in which a maximum diameter per cluster of merged EBCs is specified. The same technique can be readily integrated in this approach.

5.3 Verification Models of Different Control Network Components

An elastic control network needs to be verified as a whole to check if the required conditions for using $USFork$s or merging EBCs are met. Two frameworks were particularly useful in this work, namely, 6thSense and NuSMV. This section will try to cover both frameworks as space allows.

6thSense uses a standard VHDL to model a circuit and is particularly designed for synchronous circuit verification. Most of the control network models will be omitted since they are intuitive.



(a) Before (b) After

Figure 5.8: *EBC* merging.

NuSMV model checker has its own input language and supports both synchronous and asynchronous circuit verification. To mimic a synchronous behavior in NuSMV, the network components (e.g., joins and forks), including a clock generator, are connected synchronously. All combinatorial logic are modeled with zero delay (using DEFINE reserved word), and the clock generator changes phase with every verification cycle. An NuSMV model for a clock generator is as follows:

```
MODULE ClkGenerator
VAR Clk:boolean;
ASSIGN init(Clk) := 0; next (Clk) := !Clk;
```

and for a D-FF (with a reset value of 1):

```
MODULE DFF1(Clk,D)
VAR Q:boolean;
ASSIGN
init(Q):= 1;
next(Q):= case
(Clk=0) & (next(Clk))=1:  D;
1:  Q; esac;
```

5.3.1 n-Input Join

An NuSMV model for an n-input extension of the $LJ1111$ join structure of Fig. 4.8 is as follows:

```
MODULE LJoinn(Vl1,Vl2,..Vln,Sr)
DEFINE Vr:= Vl1 & Vl2 & ...  Vln;
DEFINE Sl1:= !(Vr & !Sr); ...  DEFINE Sln:= !(Vr & !Sr);
```

5.3.2 n-Output Fork

An NuSMV model for the n-output $EFork$ of Fig. 2.4 is as follows:

```
MODULE EForkn(Clk,Vl,Sr1,Sr2,...Srn)
VAR DFF_1:  DFF1(Clk,d1); ...  DFF_n:  DFF1(Clk,dn);
DEFINE d1 := (Sr1 & q1) | !(Vl & Sl) ;
DEFINE q1 := DFF_1.Q; DEFINE Vr1 := Vl & q1; ...
DEFINE dn := (Srn & qn) | !(Vl & Sl) ;
DEFINE qn := DFF_n.Q; DEFINE Vrn := Vl & qn;
DEFINE Sl := (Sr1 & q1) | (Sr2 & q2) | ..  (Srn & qn);
```

$USFork$ transformation Condition 3 of Theorem 5.2 is verified for each two branches in the $EFork$ to determine if they can be replaced by a $USFork$. Hence, in an n-output $EFork$ F and $\forall i, j \in \{1, 2, .., n\}, i \neq j$, the following properties are specified. In NuSMV:

```
DEFINE F_i_j_MISMATCH := Vri xor Vrj ;
PSLSPEC never F_i_j_MISMATCH;
```

And, in 6thSense (*bil* file):

```
[ fail; F_i_j; "F_i_j" ] <= Vri xor Vrj ;
```

5.3.3 Elastic Buffer Controller

Similarly, the EBC model immediately follows the FSM or the circuit implementation of [9]. The EBC merging condition of Theorem 5.7 is verified for each two EBCs in the network to determine if they can be merged. Hence, for a control network with n EBCs and $\forall i, j \in \{1, 2, .., n\}, i \neq j$, the following properties are specified. In NuSMV:

```
DEFINE EBC_i_j_MISMATCH := (Vli xor Vlj) | (Sri xor Srj) ;
PSLSPEC never EBC_i_j_MISMATCH;
```

And in 6thSense (*bil* file) as:

```
[ fail; EBC_i_j; "EBC_i_j" ] <= (Vli xor Vlj) or (Sri xor Srj) ;
```

5.3.4 SELF Input Channel

A SELF input channel (see Fig. 5.9) is the control channel corresponding to a data input (or group of data inputs) to the design. The $Valid$ signal of this channel Vi is an input to the design and the *Stall* (Si) is an output. Vi will be defined as an input with the SELF protocol constraints applied. In particular, SELF prohibits a transition from R to I states on any channel. This constraint on the input behavior is expressed in NuSMV as:

```
DEFINE InputChannel_i_Constraint := !(Vi) | !(Si) | Vi_next;
INVAR InputChannel_i_Constraint;
```

and in 6thSense (*bil* file) as:

```
[ constraint; InputChannel_i_Constraint ] <= not(Vi) or not(Si) or Vi_next;
```

In both cases, Vi is a one clock delayed version of Vi_next. Vi_next is, then, considered as the *virtual* input that the verification engine exhaustively randomizes.

5.3.5 SELF Output Channel

Similarly, a SELF output channel (see Fig. 5.9) is the control channel corresponding to a data output (or group of data outputs) from the design. The $Valid$ signal of this channel Vi is an output from the design and the *Stall* (Si) is an input. The SELF protocol does not explicitly set constraints on the possible sequence of values over the input *Stall* signal. However, it can be easily inferred from the EB specifications in [9] or the EHB (elastic half buffer) in [58] that a transition from $I0$ ($!V\&!S$) to $I1$ ($!V\&S$) states cannot happen on any SELF channel. Hence, the following constraint is applied to the SELF output channel. In NuSMV:

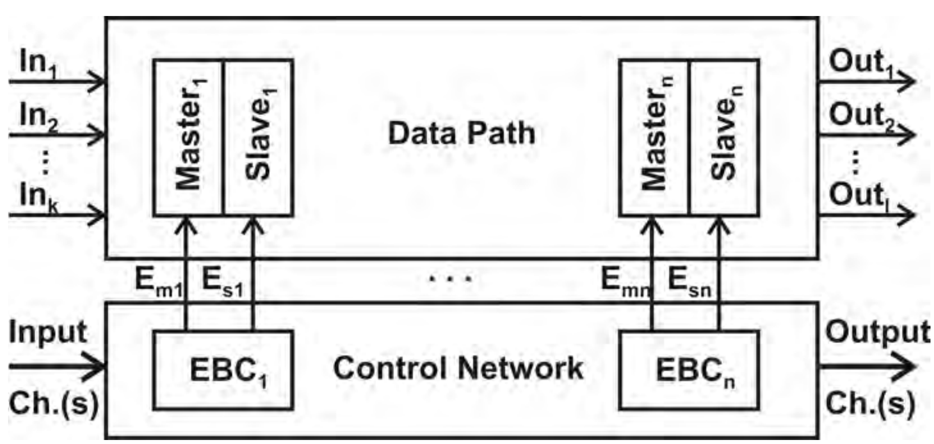


Figure 5.9: Illustration of elastic control network input and output channels.

```
DEFINE OutputChannel_i_Constraint := Vi | Si | !(Si_next);
INVAR OutputChannel_i_Constraint;
```

and in 6thSense as:

```
[ constraint; OutputChannel_i_Constraint ] <= Vi or Si or not(Si_next);
```

Again, *Si* is a one clock delayed version of the input *Si_next*.

5.3.6 Variable Latency Unit

Fig. 5.10 [9] shows a block diagram of a variable latency unit (VLU) and a variable latency controller (VLC). The VLC model follows the figure directly and omitted for brevity. The VLU model would depend on the actual unit design. Nonetheless, to be able to verify the control network, it suffices to know the minimum and maximum latency values of that unit (whatever its functionality is). Hence, for each VLU, a model is used that randomly picks the next latency value from a range of values [min,max] specified by the designer for that VLU.

5.4 HGEN Tool

To automate the transformations described in this chapter, HGEN was developed. HGEN (Hybrid network GENerator) is a fully automated tool that takes a *verilog* description of a control network and returns a *verilog* description of the minimized version. The tool currently uses 6thSense as the verification engine. Support for NuSMV is left for future versions. HGEN models the input *verilog* control network into VHDL. It adds the proper

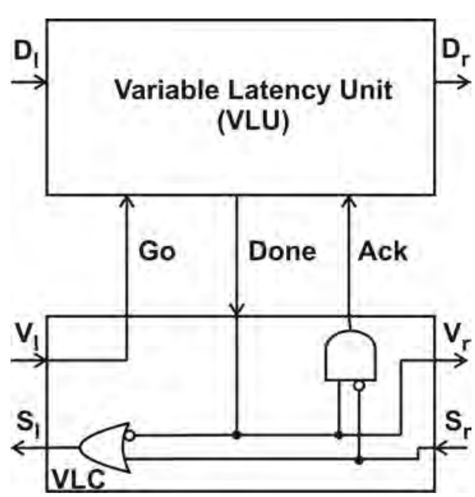


Figure 5.10: A variable latency unit and a controller.

constraints for the SELF channels. The $EFork$ to $USFork$ transformation conditions are verified for each fork in the network. Similarly, the EB controller merging conditions are checked for each two EB controllers. HGEN automatically generates the suitable models for the variable latency units (based on the min and max latencies provided by the user in a configuration file). It generates a report with the $EFork$ branches that have been transformed into $USFork$, and the merged EB controllers. *-nm* (*no merge*) option can be used to prevent HGEN from merging equivalent EB controllers (i.e., to only check for and do $EFork$ to $USFork$ transformations). The option is useful for doing the EBC merge after having some insight over the place and route information. HGEN currently supports all the network components described in Sec. 5.3 and more. Other component models (e.g., elastic half buffer and early evaluation components [43]) can be readily integrated.

5.5 Results

For all the designs in this section, CNG tool (Chapter 3) is used to automatically generate their initial elastic control networks. HGEN is then run to do the transformations described in this chapter. In all the designs the runtime is within seconds or a few minutes. The machine used has AMD Athlon™ 64 X2 Dual Core 3.2GHz processor. Area and power are synthesis numbers. DC Ultra™ [53] technology and IBM® 65 nm library were used.

5.5.1 The MiniMIPS Processor

For the sake of comparison with previous optimization techniques in this dissertation, the MiniMIPS processor is used as one of this chapter case studies. The MiniMIPS is an 8-bit subset of the 32-bit MIPS (Microprocessor without Interlocked Pipeline Stages) [46, 1]. A block diagram of the original clocked MiniMIPS is shown in Fig. 2.5. The MiniMIPS synchronous elasticization is described in Sec. 2.2. The CNG-generated elastic control network is in Fig. 3.9.

To illustrate the capability of the proposed approach, the MiniMIPS is studied in three different settings:

5.5.1.1 Register File Bubbles

In this setting the control network is closed. One to three bubble stages are inserted at the two outputs of the register file (shown in dotted rectangles in Fig. 5.11). In practice this can be done to accommodate a high latency register file or because of long wires. The resultant control network *verilog* is passed to HGEN twice (once to do $EFork$ to $USFork$ conversion only, and the second to merge equivalent EBCs as well). Table 5.2 shows the synthesis results. The last two rows in Table 5.2 are entries for the cases when controller merging is enabled. For the sake of comparison, Table 5.2 also includes entries from Chapter 4 for the *all* eager network as well as two implementations that were found to be the most area efficient among the MiniMIPS hybrid ($EFork$-$LFork$) implementations, namely, LF01-LJ1111 and LF00-LJ1011. The $EFork$-$USFork$ hybrid networks are implemented using the area and power efficient lazy joins LJ0000 and LJ1111. Column 1 in Table 5.2 lists the different combinations (sorted by their area). Column 2 lists the set of all forks that have to be implemented as eager (to both maintain the performance and cut the cycles (for the case of lazy forks)). The column also shows the ratio of the number of $EFork$s used to the total number of forks in the network. Since $USFork$s do not produce combinational cycles, therefore, $EFork$s are only used when their *eagerness* provide runtime advantage. Hence, comparing to $EFork$-$LFork$ hybrid combinations, $EFork$-$USFork$ hybrid combinations require fewer number of $EFork$s, thus minimizing the area and power of the control network. Column 3 lists the number of Elastic Buffer Controllers (EBCs) in the network. HGEN verification found that 6 out of the 10 EBCs in the MiniMIPS elastic network (in this setting) can be merged into other EBCs. The EBCs in the following groups can be merged

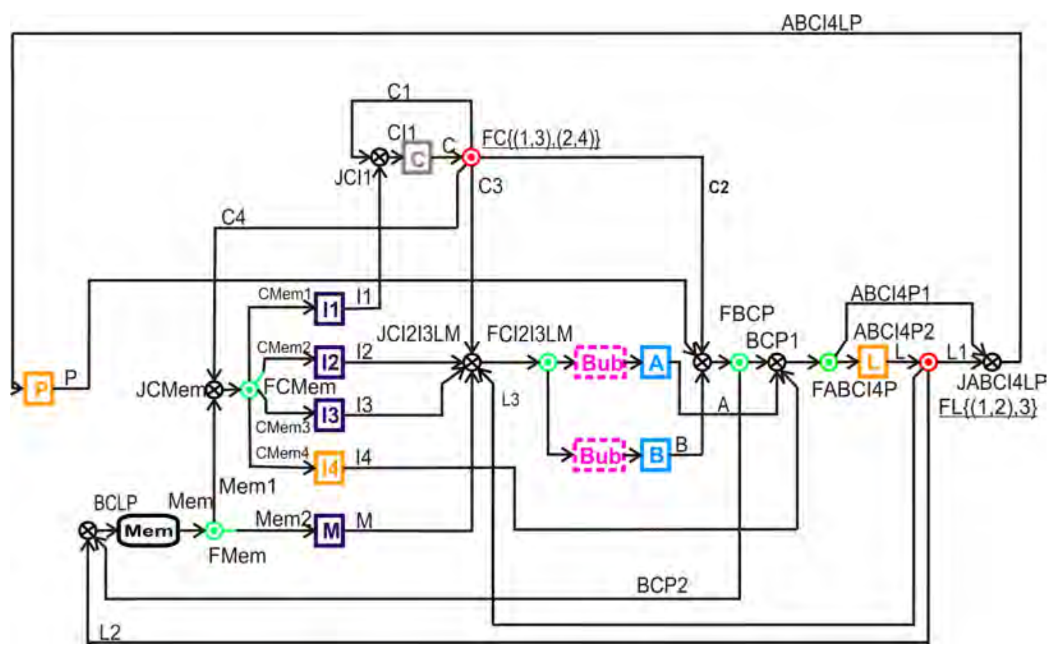


Figure 5.11: Control network of the elastic clocked MiniMIPS with register file bubbles.

Table 5.2: Area, power, and runtime of the MiniMIPS control network using different hybrid (eager/ultra-simple) SELF implementations with and without EBC merging.

Combination	n$EForks$/nForks: Eager Forks Used	nEBCs	nCyc.	Area (μm^2)	Power @ 4ns $\frac{P_{dyn}}{P_{leakage}}$ (μW) 0 B	1 B	3 B	Runtime (Cyc.) 0 B	1 B	3 B
$EFork-LJ0000$	12/12:*All*	10	0	752.4	$\frac{86.158}{2.914}$	$\frac{221.921}{2.875}$	$\frac{168.807}{2.842}$	98	147	245
$E-LF01-LJ1111$	6/12:*FC, FL, FBCP*	10	0	575.4	$\frac{65.626}{2.339}$	$\frac{188.094}{2.307}$	$\frac{140.389}{2.278}$	98	147	245
$E-LF00-LJ1011$	4/12:*Some branches of FC, FL*	10	0	513.0	$\frac{58.187}{1.980}$	$\frac{164.284}{1.990}$	$\frac{122.720}{1.992}$	98	147	245
$E-USFork-LJ0000$	2/12:*Some branches of FC, and of FL*	10	0	503.4	$\frac{53.992}{2.159}$	$\frac{153.789}{2.119}$	$\frac{114.595}{2.077}$	98	147	245
$E-USFork-LJ1111$	2/12:*Some branches of FC, and of FL*	10	0	474.6	$\frac{53.992}{1.965}$	$\frac{152.360}{1.955}$	$\frac{113.663}{1.940}$	98	147	245
$E-USFork-LJ0000\_m$	2/12:*Some branches of FC, and of FL*	4	0	288.6	$\frac{26.892}{1.276}$	$\frac{95.391}{1.281}$	$\frac{68.643}{1.279}$	98	147	245
$E-USFork-LJ1111\_m$	2/12:*Some branches of FC, and of FL*	4	0	279.6	$\frac{27.350}{1.256}$	$\frac{101.754}{1.240}$	$\frac{72.667}{1.223}$	98	147	245

together (EBCs of the same group are drawn with the same color in Fig. 5.11; no EBC for Mem): $\{(C), (I4, L, P), (I1, I2, I3, M), (A, B)\}$. The two bubble EBCs before A and B, respectively, can be merged as well; however, the two bubble areas are not included in the results. Column 4 lists the number of combinational cycles in the control network (after eager fork replacements), which is zero for all of them. Columns 5 and 6 list the synthesis area and power consumption, respectively. Comparing to the *all* eager implementation, the $EFork$-$USFork$-$LJ1111$ (or, for short, E-$USFork$-$LJ1111$) hybrid network (without EBC merging), in the case of 1 bubble, for example, shows up to 36.9% and 31.3% savings in the control network area and power, respectively. If the physical placement allows for controller merging, the resultant control network (with EBC merging) shows up to 62.8% and 54.1% savings in area and power, respectively. Finally, column 7 lists the required runtime (in terms of number of clock cycles) to finish the testbench program in [1]. Since all the transformations in this dissertation preserve the runtime, all the settings listed achieve the same runtime as the *all* eager implementation.

Row 1 of Table 5.3 contrasts the results of this setting (in case of 1 bubble in the register file path) with the other settings studied in this section.

5.5.1.2 Variable Latency ALU

In this setting, the control network is closed, and there are no bubbles at the register file outputs. The ALU is modeled with a variable latency unit that finishes an operation within one or two clk cycles. Row two of Table 5.3 shows the results. In this setting, 9 out of the 12 $EFork$s can be replaced by $USFork$s. This achieves 32.3% area reduction, and 30.5% and 25.9% dynamic, and leakage power savings, respectively. Similarly, the table also shows 63.1%, 63.0%, and 55.6% reductions in area, dynamic and leakage power, respectively, in case the physical placement allows for merging 7 out of the 10 EBCs.

5.5.1.3 Off-Chip Memory with Unknown Latency

In this setting, the control network is open at the memory interface. The memory interface is modeled in HGEN by one input and one output SELF channels. In practice this can be done if the actual latency of the memory is unknown or required to be flexible. Row three of Table 5.3 shows the results. In this setting, 7 out of the 12 $EFork$s can be replaced by $USFork$s. This achieves 25.6% area reduction, and 22.8% and 22.2% dynamic and leakage power savings, respectively. Similarly, the table also shows 47.7%, 45.0%, and

Table 5.3: HGEN results for the elastic MiniMIPS control network. Power is computed at 4 ns clock period.

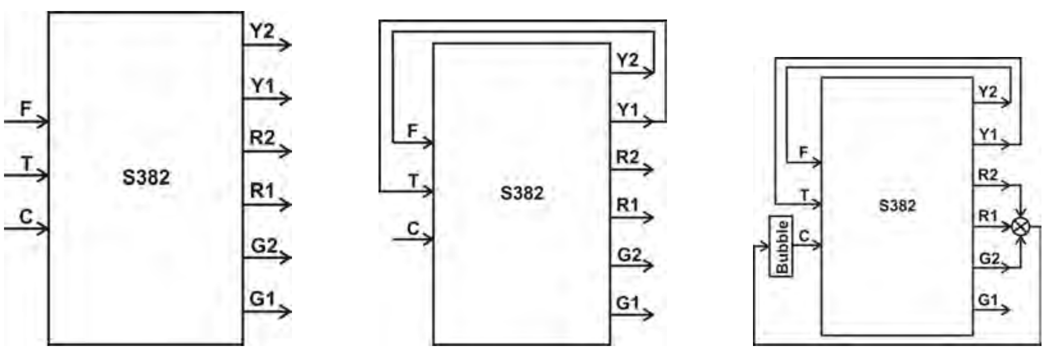
			The Original Control Network				HGEN Step 1				HGEN Step 2			
Design	#I	#O	Total # $EForks$	Total # $EBCs$	Area (μm^2)	P (μW) $\frac{P_{dyn}}{P_{leakage}}$	# Repl. $EForks$	Area (μm^2)	P (μW) $\frac{P_{dyn}}{P_{leakage}}$	$\frac{\#Prop.}{Time(s)}$	# Merg. $EBCs$	Area (μm^2)	P (μW) $\frac{P_{dyn}}{P_{leakage}}$	$\frac{\#Prop.}{Time(s)}$
MiniMIPS - 1	0	0	12	10	752.4	$\frac{221.921}{2.875}$	10	474.6	$\frac{152.360}{1.955}$	$\frac{19}{0.52}$	6	279.6	$\frac{101.754}{1.240}$	$\frac{65}{0.64}$
MiniMIPS - 2	0	0	12	10	754.2	$\frac{108.3}{2.7}$	9	510.6	$\frac{75.3}{2.0}$	$\frac{19}{0.7}$	7	278.4	$\frac{40.1}{1.2}$	$\frac{64}{0.85}$
MiniMIPS - 3	1	1	12	10	754.2	$\frac{110.5}{2.7}$	7	561.0	$\frac{85.3}{2.1}$	$\frac{19}{20.56}$	5	394.8	$\frac{60.8}{1.6}$	$\frac{64}{10.46}$

40.7% reductions in area, dynamic, and leakage power, respectively, in case the physical placement allows for merging 5 out of the 10 EBCs.

5.5.2 S382

S382 (see Fig. 5.12) is one of the ISCAS benchmarks. It has 3 input channels: F, T, and C, and 6 output channels: Y2, Y1, R2, R1, G2, and G1, and 21 EBCs. Table 5.4 shows the results of running HGEN over s382 in 3 different *incremental* settings:

1. All the 9 input/output channels are left open.
2. Y2 is connected to F, and Y1 is connected to T. The other 5 input/output channels are left open.
3. Y2 is connected to F, and Y1 is connected to T. R2, R1, and G2 are connected to C through a 3-input join followed by a bubble. Output channel G1 is left open.



(a) Settings - 1. (b) Settings - 2. (c) Settings - 3.

Figure 5.12: S382.

Table 5.4: HGEN results for s382 benchmark.

Design	#I	#O	Total # $EForks$	Total # EBCs	# Repl. $EForks$	# Merg. EBCs	$\frac{\#Prop.}{Time(s)}$
s382 - 1	3	6	25	21	8	7	$\frac{255}{20.1}$
s382 - 2	1	4	25	21	9	8	$\frac{255}{375.22}$
s382 - 3	0	1	25	22	18	17	$\frac{255}{152.29}$

Intuitively, the input behavior of setting 3 is a subset of 2, which in turn, is a subset of 1. Hence, the number of $EFork$s that can be replaced by $USFork$s is the same or increases from setting 1 to setting 3. Though the proposed approach handles open and closed control networks, however, this example shows that the chance of finding candidate $EFork$s for replacement increases as more knowledge of the environment is available. In s382, the reduction in the number of $EFork$s is 32%, 36%, and 72% in settings 1, 2, and 3, respectively.

Finally, Table 5.5 shows HGEN results for other ISCAS benchmarks verified in *totally open* control network settings (i.e., no abstract for the environment is provided). The results emphasize the speed of the tool. Further savings in the number of $EFork$s and EBCs can be achieved with more knowledge of the environment model.

Table 5.5: HGEN results for other ISCAS benchmarks - in *open* network settings.

Design	#I	#O	Total # $EFork$s	Total # EBCs	# Repl. $EFork$s	# Merg. EBCs	$\frac{\#Prop.}{Time(s)}$
s27	4	1	3	3	1	1	$\frac{7}{0.79}$
s298	3	6	25	14	2	2	$\frac{131}{5.37}$
s344	9	11	32	15	2	2	$\frac{177}{2.61}$
s386	7	7	15	6	2	2	$\frac{40}{1.49}$
s1488	8	19	32	6	5	5	$\frac{102}{4.56}$

CHAPTER 6

CONCLUSION AND FUTURE WORK

Several optimization algorithms, tools and flows have been introduced in this dissertation to minimize the area and power overhead of elastic control networks without sacrificing performance. That included:

- minimizing the total number of join and fork control steering units in the control network.
- replacing the area and power expensive eager forks with lazy forks under some performance equivalence conditions.
- utilizing a novel Ultra Simple Fork ($USFork$) implementation. The $USFork$ has two advantages over lazy forks: it is composed of simpler logic (just wires) and does not form combinational cycles in the control network.
- merging equivalent Elastic Buffer Controllers (EBC)s.

The dissertation also introduced a fully automated control network verification (and transformation) framework (HGEN). HGEN automatically verifies the conditions under which an $EFork$ can be replaced by a lazy fork or a $USFork$, and the conditions under which several EBCs can be merged in a control network. HGEN supports different types of synchronous elastic control networks. That includes open networks (i.e., when the environment abstract is not available or required to be flexible) as well as networks incorporating variable latency units.

The MiniMIPS processor was studied as a running case study throughout the dissertation. Table 6.1 shows the area, power, and runtime of the most relevant control network implementations in this work. Results are synthesis numbers (of the control network only) using the Artisan academic library for IBM® 65 nm process. Runtime is measured in the number of clock cycles required to finish the testbench program in [1]. The table starts with the non-optimized version generated using the direct approach proposed in [9, 3] (Row 1). Every following row shows the effect of applying one of the optimization techniques proposed in this dissertation. Comparing the last row to the first, the optimization techniques of this

Table 6.1: Summary of results for some of the different MiniMIPS control network implementations introduced in this dissertation. One bubble is inserted at each of the register file two outputs.

Network Generator	Reference Chapter	Combination	n$EForks$/nForks: Eager Forks Used	nEBCs	nCyc.	Area (μm^2)	P @ 4ns $\frac{P_{dyn}}{P_{leakage}}$ (μW)	Runtime (Cyc.)
[9, 3]	Ch. 1	$EFork-LJ0000$	25/25:*ALL*	10	0	1044.0	$\frac{324.424}{4.018}$	147
Hand optimized	Ch. 2	$EFork-LJ0000$	14/14:*ALL*	10	0	799.2	$\frac{235.941}{2.990}$	147
CNG	Ch. 3	$EFork-LJ0000$	12/12:*ALL*	10	0	752.4	$\frac{221.921}{2.875}$	147
	Ch. 4	$E-LF01-LJ1011$	4/12:*Some branches of FC, FL*	10	0	588.0	$\frac{183.991}{2.536}$	147
	Ch. 4	$E-LF01-LJ1111$	6/12:*FC, FL, FBCP*	10	0	575.4	$\frac{188.094}{2.307}$	147
	Ch. 4	$E-LF00-LJ1011$	4/12:*Some branches of FC, FL*	10	0	513.0	$\frac{164.284}{1.990}$	147
	Ch. 5	E-$USFork$-LJ0000	2/12:*Some branches of FC, and of FL*	10	0	503.4	$\frac{153.789}{2.119}$	147
	Ch. 5	E-$USFork$-LJ1111	2/12:*Some branches of FC, and of FL*	10	0	474.6	$\frac{152.360}{1.955}$	147
	Ch. 5	E-$USFork$-LJ0000_m	2/12:*Some branches of FC, and of FL*	4	0	288.6	$\frac{95.391}{1.281}$	147
	Ch. 5	E-$USFork$-LJ1111_m	2/12:*Some branches of FC, and of FL*	4	0	279.6	$\frac{101.754}{1.240}$	147

dissertation accumulatively achieve an area, dynamic, and leakage power reduction (in the control network) of 73.2%, 68.6%, and 69.1%, respectively. Charts illustrating the area and dynamic power of different MiniMIPS synchronous elastic control network implementations are shown in Figures 6.1 and 6.2, respectively. In both charts, except for the first two bars in each, the control network is automatically generated by CNG tool.

The elastic MiniMIPS constructed using the hybrid control network implementations listed in Table 6.1 can tolerate bubbles in the register file path, and still achieve the same runtime as the *all* eager implementation. A direct comparison with the ordinary clocked MIPS cannot be established since inserting bubbles in the latter will cause it to fail as it is designed for static latencies only. For the normally clocked MiniMIPS to handle bubbles (or variable latency interfaces) over its channels, several changes in the datapath may be required (e.g., implementing FSMs at channel receiver ends to wait until valid data arrive, some mechanism to propagate this information to the rest of the system, a stalling mechanism, etc.). On the other hand, and by its definition, the SELF protocol inherently achieves this goal.

Table 6.2 shows the cost of achieving this required elasticity using an *all* eager and a set of hybrid SELF implementations. The results in the table are synthesis numbers for

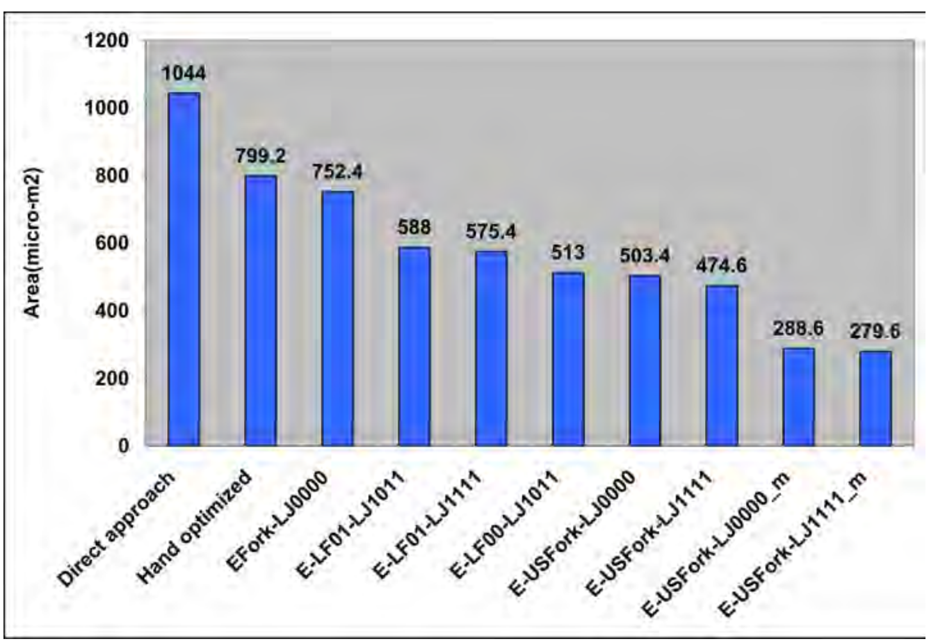


Figure 6.1: A chart of the MiniMIPS control network area in different synchronous elastic implementations.

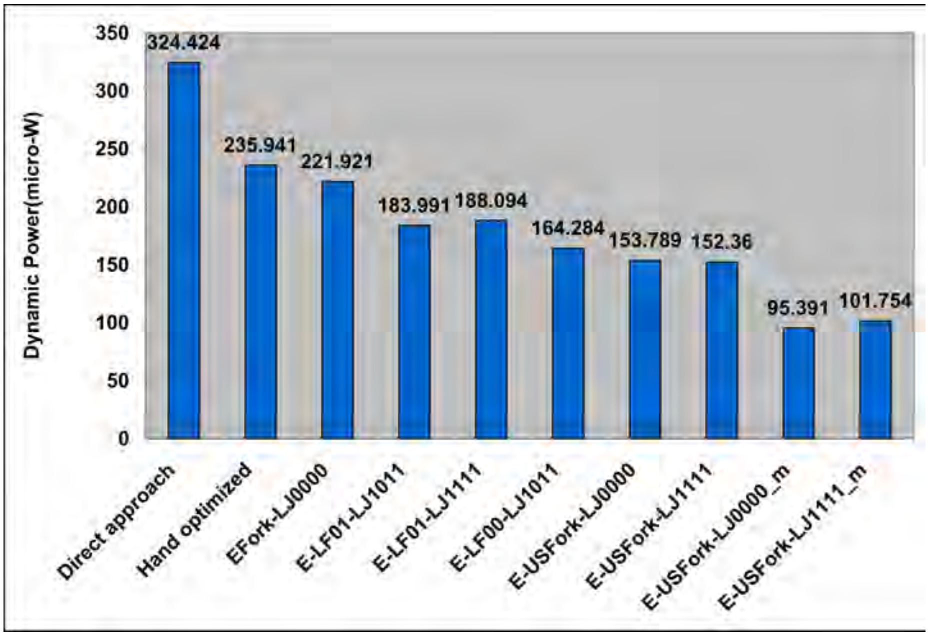


Figure 6.2: A chart of the MiniMIPS control network dynamic power in different synchronous elastic implementations.

the whole MiniMIPS (not just the control network). Since the normally clocked MiniMIPS cannot directly tolerate register file bubbles, therefore and for the sake of comparison, no bubbles are added in either the normally clocked or the elastic MiniMIPS (even if the elastic MiniMIPS *can* tolerate the register file bubbles). Two implementations for the clocked MiniMIPS are listed. The first is flip-flop (FF) based. In the second one, each FF is replaced by a master-slave latch pair. The latches used in both the latch based and the elastic MiniMIPS are selected from manually synthesized and optimized templates that are protected during synthesis with set_size_only attributes. The FF based design is completely synthesized by DC. In this specific design and cell library, the latch based design consumed more area and leakage power but less dynamic power. Without loss of generality, overhead percentages (over. %) of elastic versions are with respect to the latch based design. Please note that if more bubbles (or variable latency interfaces) are required in the MiniMIPS, more lazy forks (in the hybrid implementation) may need to be replaced by $EFork$s to keep the same runtime as the *all* eager implementation, and some of the EBCs may not be mergeable any more, resulting in more area and power.

The optimization techniques have also been applied to several ISCAS benchmarks showing similar significant reductions in area and power. For the case of s382, for example,

Table 6.2: Elasticity area and power overheads of different hybrid SELF implementations of the MiniMIP processor.

Implementation		Area		P_{dyn} @ 4ns		P_{leak}	
		μm^2	over.%	μW	over.%	μW	over.%
Normally clocked	Flip-flop based	2617.2		446.247		8.850	
	Latch based	2642.4		380.466		9.504	
Elastic clocked	*All* eager ($EFork-LJ0000$)	3385.2	28.1%	474.465	24.7%	12.681	33.4%
	Hybrid ($E-LF00-LJ1011$)	3136.2	18.7%	437.977	15.1%	11.686	23.0%
	Hybrid ($E-USFork-LJ1111$)	3106.8	17.6%	435.334	14.4%	11.620	22.3%
	Hybrid ($E-USFork-LJ1111\_m$)	2889.6	9.4%	408.840	7.5%	10.838	14.0%

CNG generates a control network with only 22 2-input join (J_2) and 25 2-output fork (F_2) components compared to a control network of 148 J_2s and 151 F_2s generated through a direct unoptimized approach. Furthermore, HGEN verifies that at least 32% of the $EFork$s in the CNG-generated s382 control network can be replaced by $USFork$s, reducing area and power without any performance loss. More reduction is possible if the physical placement allows for controller merging.

The impact of this work will broaden the class of circuits that can be elasticized with acceptable overhead (circuits that designers would otherwise find it too expensive to elasticize). The impact will also enable designers to deepen the level of elastic granularity in their designs to enjoy the full benefit of elasticity at a reasonable cost.

6.1 Future Work

Though the optimization algorithms introduced in this work were applied to basic join and fork structures, nonetheless, we do not see any major obstacles for extending the work to advanced structures like early evaluation joins and anti-token propagation [32]. Other tool-specific future work is listed below:

6.1.1 CNG

The CNG algorithm described in Chapter 3 is based on continuous reduction of the search *Space* until an *optimum Solution* is returned. Indeed, the *Space* reduction steps are so efficient that in 18 out of the 25 problems listed in Table 3.4, only one *Solution* is left in the search *Space* (i.e., the *OptSoln*). The CNG runtime is also less than 1 second for all the listed 20 ISCAS-89 benchmarks. Nonetheless, since the search *Space* is exponential in the problem input size, for ISCAS problems bigger than s1488, the tool (as described in Chapter 3) requires impractically long runtime. This motivates the search for better data structures, algorithms for dividing the problem into a set of smaller ones, and/or heuristics to cut the runtime. Chapter 3 laid the foundation for the theoretical background of CNG. With its plenty of theorems, numerous ideas for good heuristics can be devised as well as integration of well known search heuristic methods (e.g., simulated annealing, genetic algorithms, etc. [62]). Appendix A shows some preliminary heuristics that were briefly explored. This is an area for future research.

6.1.2 HGEN

HGEN replaces $EFork$s with $USFork$s when the former eagerness is not adding any performance advantage (i.e., redundant). Similarly, it also merges EBCs when they schedule their corresponding latches at similar times. Nonetheless, the conditions used in either cases (i.e., $EFork$ to $USFork$ conversion and equivalent EBC merging) are rather conservative. In both cases, the equivalence conditions were based on cycle-by-cycle equivalence. For example, in $EFork$ to $USFork$ conversion, conditions are employed that guarantee the different branches have matched *Stall* patterns in *all* clock cycles (when the left $Valid$ is one). Nonetheless, it can be true in some networks that even if the *Stall* patterns are not matched in all clock cycles, yet still, the eagerness is not required. Consider, for example, the case when both branches which have mismatched *Stall* patterns are not on any critical (architectural) cycle in the network. Hence, delaying passing the data token to one of them (rather than the earliest start provided by the $EFork$) may not enhance the overall network performance as the bottleneck is somewhere else. Hence, finding the equivalence conditions (for both aforementioned transformations) that preserves the overall network performance rather than the local cycle-by-cycle equivalence is left for future work. This can allow for more relaxed conditions that would provide higher chances for $EFork$ replacements and EBC merging, further reducing the area and power. The future work can make very much use of HGEN since the tool provides a fully automated framework for synchronous elastic control network verification and transformation. The idea of overall network performance (expressed as throughput) is well formulated in the literature (see, for example, [63, 43]).

APPENDIX A

HEURISTICS TO CUT CNG RUNTIME FOR BIG PROBLEMS

For all the 20 (out of 28) ISCAS-89 problems listed in Table 3.4, CNG required less than 1 second to finish. However, for ISCAS problems bigger than s1488, the tool (as described in Chapter 3) requires impractically long runtime. This motivates using better data structures, problem division algorithms, and/or heuristics to cut runtime for bigger problems. Based on the numerous theorems listed in Chapter 3, several heuristics may be devised. This is an open area for research. Following are some heuristics that were *briefly* explored:

- **H1** Limit the maximum number of PSs per $Term$ to value m. H1(m) will be used as a shortcut for applying H1 with a maximum number of PSs $= m$ per $Term$. m can be defined as a constant value or a function of the $Term$ cardinality. It can also be defined as a function of the $Term$ *essentiality*; giving more choices for $Term$s that are known to be used in the *OptSoln* (i.e., *essential* $Terms$ - see Definitions 3.18, 3.24, and 3.26).
- **H2** Restrict overlapping of $Term$s in any PS; allow a $Term$ to overlap with other $Term$s in a PS only if it is a $TITerm$ (see Def. 3.18). $Term_i$ is overlapping in PS_t if $ACov(Term_i, PS_t) \neq Term_i$ (see Def. 3.19).
- **H3** Relax the PS elimination condition of Corollary 3.14 to the following condition: Let PS_{t1} and PS_{t2} be two PSs of $Term_t$ in *Space* S_k. Then, eliminate PS_{t1} from the search *Space* if $nAJMin_o(PS_{t1})\Big|_{S_k} \geq nAJMin_o(PS_{t2})\Big|_{S_k}$.
- **H4** Generate a good *Solution* in a short time using any combination of H1 - H3, and use it as an initial seed for well known search heuristic methods (e.g., simulated annealing, genetic algorithms, etc. [62]).

By their definition, heuristics do not guarantee an *optimum Solution*, nonetheless, good heuristics give good *Solution*s in a short runtime in most cases [62]. Only a small subset of the above heuristics has been tried. For the sake of demonstration, Table A.1 shows *sample* results for applying some of the heuristics above over the rest of the ISCAS-89 benchmarks that were not covered in Table 3.4. The table shows that with even preliminary application of *simple* heuristics on the listed examples, on the average, ABC from Berkeley generates a control network with a number of joins (and forks) that is 3.02% worse than CNG and DC is slightly (0.53%) better than CNG. The sample results show the potential of even *simple* heuristics in both the quality of the *Solution* and the runtime.

Refining the above heuristics, devising new set based on the CNG theorem of Chapter 3, as well as integration of well known search heuristic methods (e.g., simulated annealing, genetic algorithms, etc. [62]) are kept for future work.

Table A.1: CNG *Cost* vs. other synthesis tools/flows using heuristics. Worse percentages are calculated with respect to CNG results.

Problem	$\|SourceS\|$	$\|TargetS\|$	$\|PTermS^4\|$	CNG			Flow of [9, 3]		ABC		Design Compiler®	
				Cost	Heuristics used	runtime	*Cost*	Worse%	*Cost*	Worse%	*Cost*	Worse%
s5378	214	228	493	367	H1(30), H3	0.7s	2085	468.12%	367	0.00%	359	-2.18%
s9234	247	250	672	414	H1(30), H2, H3	2.2s	3010	627.05%	411	-0.72%	419	1.21%
s1423	91	79	322	175 171	H1(1) H1(2)	1.3s 108.4s	2156	1160.82%	172	0.58%	166	-2.92%
s13207	700	790	1051	900 893	H1(30), H3 H1(2)	8.5s 1613.0s	3931	340.20%	902	1.01%	941	5.38%
s15850	611	684	1534	1129	H1(30), H3	101.3s	16203	1335.16%	1186	5.05%	1186	5.05%
s35932	1763	2048	3781	3635	H1(30), H3	5.7s	5547	52.6%	3938	8.34%	3695	1.65%
s38417	1664	1742	4087	2838	H1(1)	29.49s	32609	1049.01%	3235	13.99%	2699	-4.90%
s38584	1464	1730	5086	3428	H1(20), H2, H3	145.4s	18714	445.92%	3287	-4.11%	3170	-7.53%

APPENDIX B

ELIMINATING NEGATIVE SLACK IN SYNCHRONOUS ELASTIC CONTROL NETWORKS

CNG tool described in Chapter 3 produces a control network with minimal total number of 2-input joins and 2-output forks. Nonetheless, it is not guaranteed that the generated network has the minimum possible critical path delay. Normally this is not a problem since the critical delay of the datapath is usually larger than that of the control network. Nonetheless, this appendix introduces a systematic flow (referred to as CNGT) of structural transformations of the control network (of *basic* synchronous elastic circuits) that reduces the network delay to meet tight timing constraints. CNGT iteratively targets paths that have negative slacks at the cost of possibly adding some hardware until meeting a specified clock period constraint. The flow is validated by proving that the two versions of the control network (i.e., before and after the transformations) are functionally equivalent. It has been applied to the MiniMIPS processor and s298 ISCAS-89 benchmark. In the former, it removed a total negative slack of 1.3 ns with an area improvement of 6.2%. In the latter, it removed 5.3 ns with an area penalty of only 0.4%. Though the CNG-generated control network can be implemented synchronously or asynchronously, however, CNGT (in its current form) is applicable to synchronous elasticity only.

B.1 Proposed Structural Transformations

A path, p_i, in a synchronous elastic control network is defined the same way as in the data path. A path is a concatenation of signals. It starts at a Q-output of a synchronizing element (e.g., a flip-flop or a latch), and it ends at a D-input of a synchronizing element. A path, p_i, is called a violator, v_i, if its delay violates one of the timing constraints. This flow focuses on maximum delay constraints. A path is considered a violator if its delay exceeds some maximum delay constraints (usually a clock period with setup and propagation delays and time borrowing taken into account). The difference between a time constraint and the path delay is known as slack. If the slack is negative, the path is a violator. The total negative slack is defined to be the sum of the negative slacks in all the violators of the design (i.e., the control network in this case). It is usually represented with a positive number. The purpose of the presented flow is to reduce the total negative slack to zero at a certain clock period constraint. Following are some proposed structural transformations that help reducing violator delays:

B.1.1 Combining Joins and Input $Valid$s Reorder

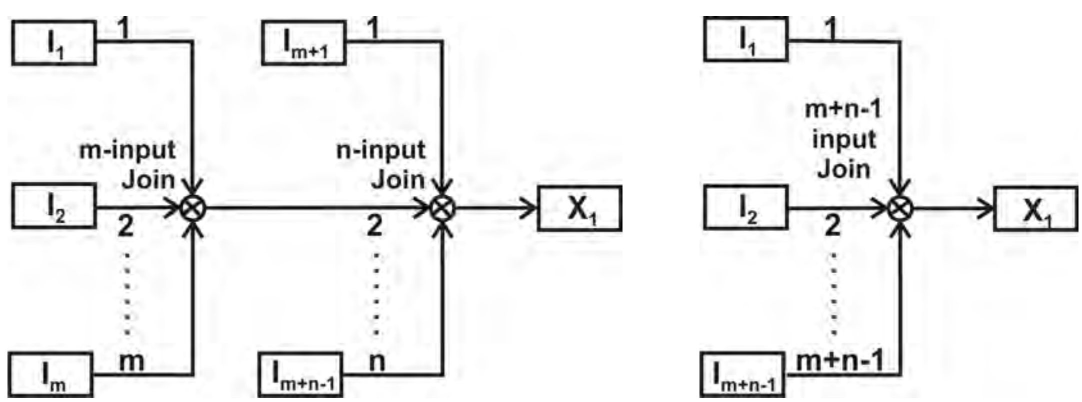
Concatenated m-input-channel and n-input-channel joins can be combined into an (m+n-1)-input-channel join, as shown in Fig. B.1. The combination preserves the control network functionality. It also reduces the delay of the $Valid$ output signal, V_r.

Combining reduces the amount of logic gates between the latest input $Valid$ signal and the join $Valid$ output, V_r. It allows for an optimization inside the combined join that takes into account the relative arrival times of the different input $Valid$ signals moving critical signals closer to the output.

Similarly, local optimization inside the combined (m+n-1)-input join can reduce the delays of the $Stall$ output signals (i.e., S_{l1}, S_{l2}, etc.).

B.1.2 Combining Forks and Input $Stall$s Reorder

Similarly, a concatenated m-output-channel and n-output-channel forks can be combined into (m+n-1)-output-channel fork. The combination preserves the control network functionality. It also reduces the delay of the $Stall$ output signal, S_l. Reasons are the same



(a) Before (b) After

Figure B.1: Combining concatenated n-input and m-input joins.

as in Sec. B.1.1 but with respect to the *Stall* signals. Also, local optimization inside the combined (m+n-1)-output-channel fork can reduce the delays of the $Valid$ output signals (i.e., V_{r1}, V_{r2}, etc.).

B.1.3 Rolling Back a Fork

If concatenated joins and forks are, respectively, combined, then any path would pass through a concatenation of interleaving multi-input (or output) joins (or forks).

Rolling back a fork moves a fork back in a path, such that it can combine with forks preceding it in that path. Further, this allows the joins before and after it to be combined as well. Rolling back a fork preserves the control network functionality (see the verification in Sec. B.4). It has the potential of cutting from the path delay because of the combining action that takes place in both joins and forks that surround this fork. However, in some cases the transformation can introduce more violators. Quantifying the effect of rolling back a fork is deferred to Sec. B.2.

Example B.1. Let A, B, C, D, X_1, X_2, X_3, and X_4 be eight registers in the original ordinary clocked design. The following registers pass data to X_1: A, B, and C, and to X_2: A, B, and D, and to X_3: A, and to X_4: B. A possible control network of the LI version of this design is shown in Fig. B.2a.

Let Vx and Sx be the $Valid$ and $Stall$ signals of control channel x, respectively. Assume that the following path is a violator in Fig. B.2a: (from A), VA, $VA2$, VAB, $VAB1$, $VABC$ (to X_1). This path passes through two 2-output forks and two 2-input joins. Rolling fork FAB back to the inputs of join JAB is shown in Fig. B.2b. This allows for combining the preceding and following joins and forks as shown in Fig. B.2c. The path from A to X_1 now incorporates only one 3-output fork and one 3-input join. Hence, rolling back fork FAB allows for delay optimization in the 3-output fork and in the 3-input join, reducing that violator delay.

In general, rolling back an n-output fork through an m-input join is shown in Fig. B.3, where I_{ij} is the jth output of an n-output fork whose input is I_i. The m n-output forks that produce I_{ij}s are omitted from Fig. B.3b for simplicity. I_is and X_is in Fig. B.3 could be any control channels (i.e., not necessarily directly connected to controllers). Rolling back some (not all) of the branches of an n-output fork through an m-input join also has delay reduction effects for some of the paths. However, in the context of this work, when a fork

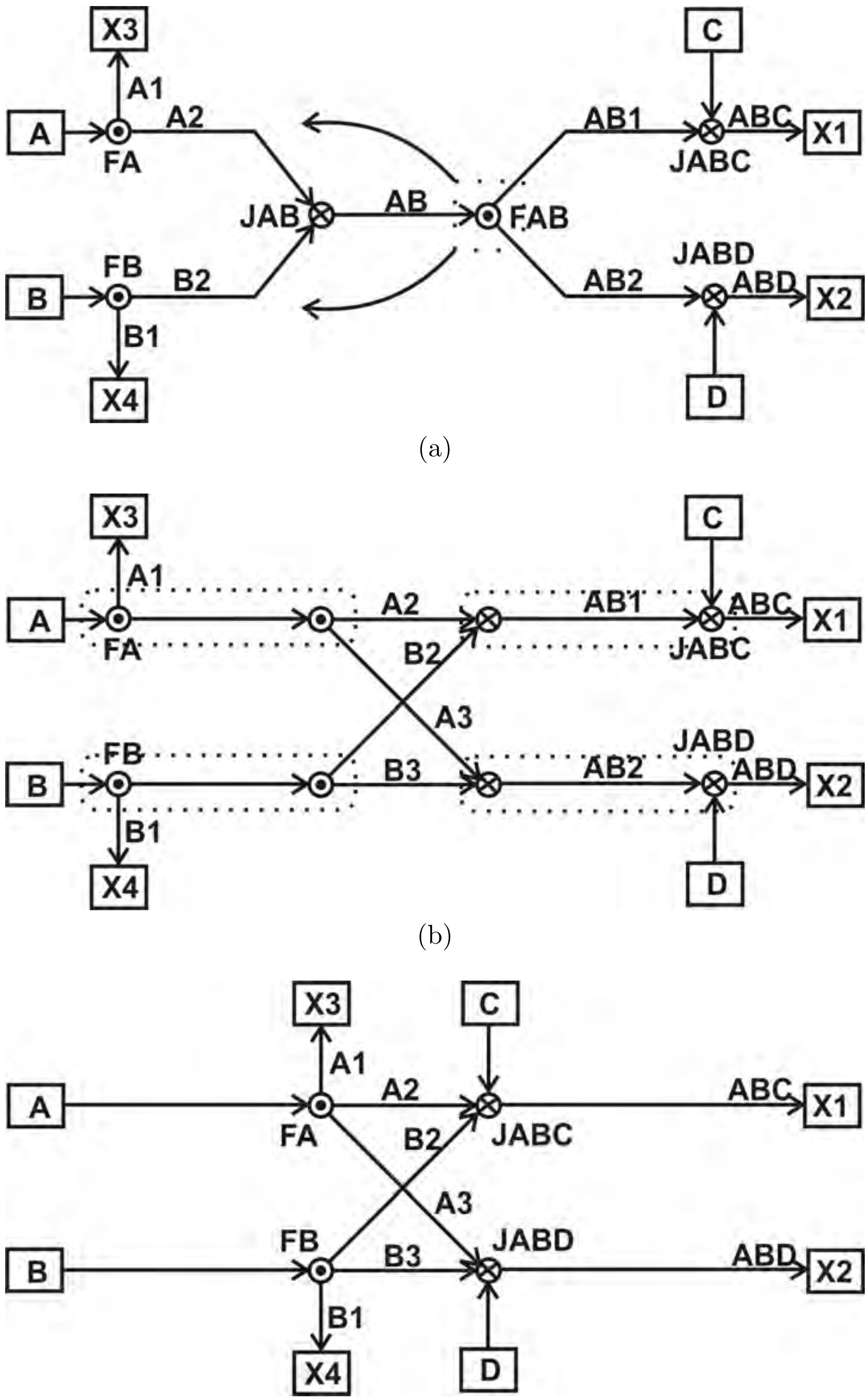
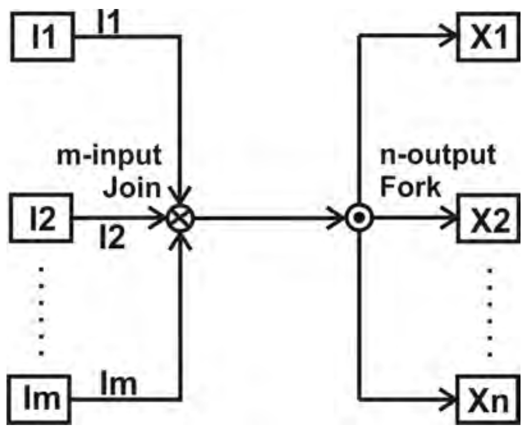
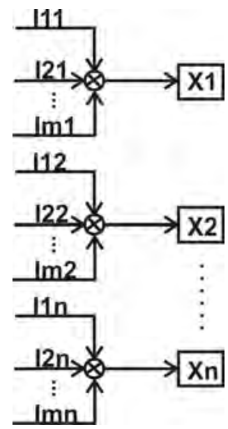


Figure B.2: Steps of rolling back fork *FAB*.



(a) Before



(b) After

Figure B.3: Rolling back an n-output fork through an m-input join

is rolled back, all its branches are rolled.

B.2 *Gain* Function

Rolling back a fork would usually decrease the delay of the associated paths because of the combining action that takes place in the preceding and following joins and forks. However, in some cases, it may increase the negative slack of some violators. To quantify these effects on a certain fork F_i, a *heuristic Gain* function is defined, $Gain(F_i)$. $Gain(F_i)$ evaluates to a number that should be proportional to the reduction in the total negative slack of the network if fork F_i is rolled back.

To compute the $Gain$ of a certain fork, F_i, the different path types that can pass through this fork need, first, be examined. Following is a list of six path types along with the rolling back effect on each. The argument will make use of the network of Fig. B.2, where fork FAB is to be rolled back. The work is applicable to eager fork and lazy join implementations (see, for example, Figures 2.4 and 2.3, respectively).

B.2.1 Type I

A path of this type will have the fork V_l and any of the V_{ri} as part of it (i.e., it passes through the fork in the $Valid$ direction).

Let us consider a path of type I passing through fork FAB in Fig. B.2a. A path cannot start nor end in a join, since a join does not have any synchronizing elements. A path can only start or end either in an elastic controller or in a fork (since eager forks incorporate flip-flops). Hence, a type I path, that passes through fork FAB, will end either at the $Valid$

input of X_1 controller (i.e., through join $JABC$), or at the $Valid$ input of X_2 controller (i.e., through join $JABD$), or at the *Stall* input of C controller (i.e., $VAB1$, then through join $JABC$ to SC), or at the *Stall* input of D controller (i.e., $VAB2$, then through join $JABD$ to SD). In all these four cases, rolling back fork FAB will reduce the delay of the path end points, respectively. Delay reduction is due to the fork combination (FA with FAB, and FB with FAB) and the join combination (JAB with $JABC$, and JAB with $JABD$), as shown in Fig. B.2c.

B.2.2 Type II

A path of this type will have any of the fork S_{ri} and S_l as part of it (i.e., it passes through the fork in the *Stall* direction).

Let us consider a path of type II passing through fork FAB in Fig. B.2a. This path will end either at the *Stall* input of A or B controllers, or at the D-input of any of the two registers R_1 and R_2 in forks FA or FB. In all these cases, the path delays are the same or less after rolling back fork FAB.

Consider, as an example, the following path in Fig. B.2a: (from X_1), $SABC$, $SAB1$, SAB, $SA2$, SA,(to A). The path incorporates two 2-output forks and two 2-input joins. After rolling back, in Fig. B.2c, the path is reduced to only one 3-output fork and one 3-input join.

B.2.3 Type III

A path of this type will have the fork V_l and any of the R_i register D-inputs as part of it (i.e., it is a path coming in the $Valid$ direction and ends inside the fork). Rolling back a fork is likely to decrease the delay of this type of paths.

An example of this type in Fig. B.2a is: (from A), VA, $VA2$, VAB, $(FAB/R_1/D)$. It can be easily shown that rolling back fork FAB will decrease the delay at that path endpoint.

B.2.4 Type IV

A path of this type will have any of the R_i register Q-outputs (inside the fork) and S_l as part of it (i.e., it starts inside the fork and propagates in the *Stall* direction). Rolling back a fork is likely to decrease the delay of this type of paths.

An example of this type in Fig. B.2a is: (from $FAB/R_1/Q$), SAB, $SA2$, SA, (to A). It can be shown that rolling back fork FAB will decrease the delay at that path endpoint.

B.2.5 Type V

A path of this type will have any of the R_i register Q-outputs (inside the fork) and the corresponding V_{ri} as part of it (i.e., it is a path starting inside the fork and propagating in the $Valid$ direction). Rolling back a fork is likely to increase the delay of this type of paths.

An example of this type in Fig. B.2a is: (from $FAB/R_1/Q$), $VAB1$, $VABC$, (to X_1). It can be easily shown that rolling back fork FAB will increase the delay at that path endpoint.

B.2.6 Type VI

A path of this type will have any of the fork S_{ri} and any of the R_i register D-inputs as part of it (i.e., it is a path coming in the *Stall* direction and ends inside the fork). Rolling back a fork is likely to increase the delay of this type of paths.

An example of this type in Fig. B.2a is: (from X_1), $SABC$, $SAB1$, (to $FAB/R_1/D$). It can be easily shown that rolling back fork FAB will increase the delay at that path endpoint.

The *Gain* function of a certain fork, F_i, is defined as follows:

$$Gain(F_i) = \sum_{j=1}^{|Violators|} r_j.w_j \tag{B.1}$$

where $|Violators|$ is the number of violators. r_j is a number proportional to the delay reduction in violator, v_j, caused by rolling back fork F_i. w_j is the weight of violator v_j.

One approach of choosing violator weights (i.e., w_j), is to give each violator a weight based on its negative slack. This approach will give priority to worst slack violator fixing. Another approach is to choose a value of 1 for all violator weights, giving all of them the same priority. The results reported in this appendix are based on the latter approach.

The value of r_j is technology and topology dependent. It also depends on the synthesis tool optimization algorithms. Accurate evaluation of these values are kept for future work. A value of 1 is chosen for each violator that is of type I, II, III or IV, and -1 for each violator that is of type V or VI, and 0 otherwise (i.e., if F_i is not in the violator path).

B.3 The Proposed Flow

A chart of the proposed flow is shown in Fig. B.4. The flow starts by running the `CNG` tool (Chapter 3) to generate a control network with minimal total number of 2-input joins and 2-output forks. The flow then takes as an input a target clock frequency for the control

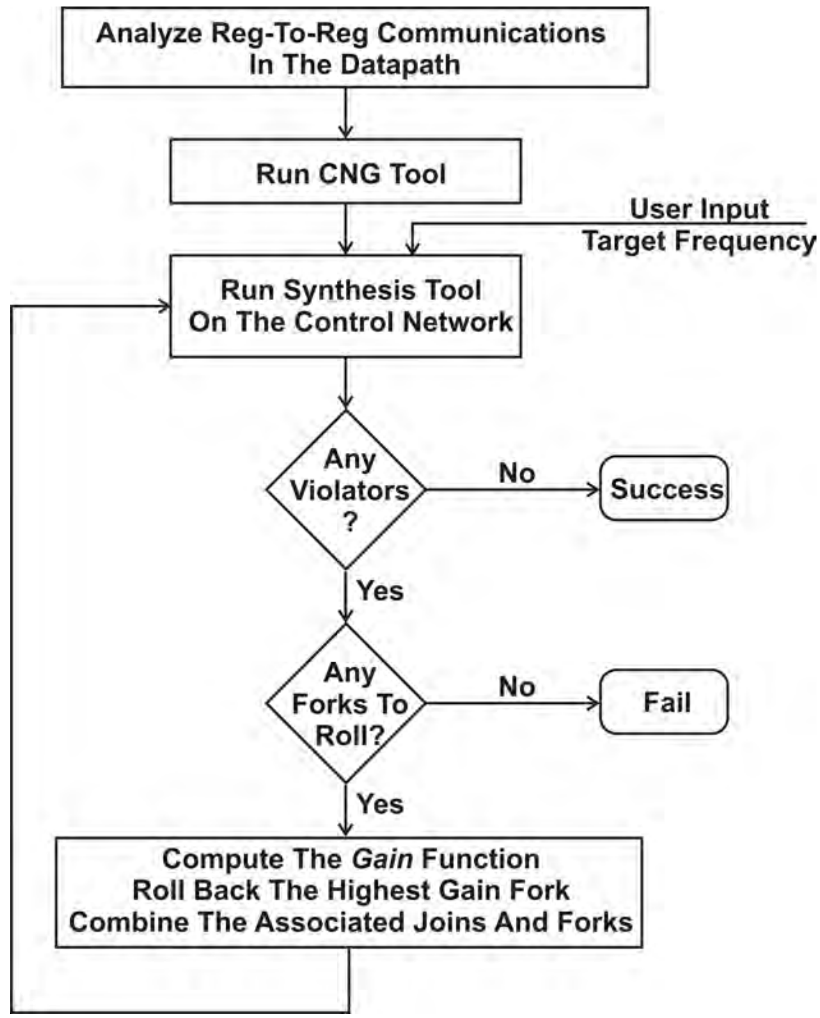


Figure B.4: The proposed flow.

network. The network is synthesized and checked against the timing constraints. If there is no violation, the flow exits successfully. If there are timing violations, the reported violators (by the synthesis tool) are analyzed. The *Gain* function is computed for all the forks in the design. The fork with the highest *Gain* is chosen to be rolled back. The new network is now passed to the synthesis tool again. The loop continues until the network meets the timing constraint (i.e., success) or there are no more forks available to be rolled back (i.e., fail).

B.3.1 Synthesis Considerations

Only the control network part of the design is synthesized. The data path is abstracted out. The *EB* controller implementation of [9] is used. In the controllers, a value of zero is set for the output port delays of the master and slave latch enables (i.e., E_m and E_s, respectively). This allows E_m and E_s to change as late as the clock positive edge but not later. It also ensures maximum possible time borrowing (for E_m) without touching the data path performance (i.e., no time borrowing from the data path will take place). A more accurate value for E_m and E_s port delays should be the enable setup times, which are library dependent.

One of the strongest motivations behind the latency insensitive paradigm is to tackle long wire delay problems [15, 16, 17]. Besides, it facilitates communication between different IP cores on a chip. Hence, the logic in the LI control network is expected to be highly distributed, where wire delays are substantial contributors in the violator slacks. It is planned to include a metric for wire delays in the *Gain* function proposed in Sec. B.2 in future work. The wire delay metric will be based on back-annotated place and route information. Hence, the choice of rolling back a fork will take into account the added (or removed) wire delay expenses. For this same reason, the hierarchy is kept during synthesis (i.e., the logical positions of joins and forks are kept and only local optimizations inside the joins and forks are allowed). This way it will be possible to back annotate the wire delays into this flow calculations and into the synthesis tool.

Example B.2. Given the control network of Fig. B.5, find a functionally equivalent network that can be clocked with 370 ps clock.

The original control network of Fig. B.5 is synthesized with Design Compiler® (DC) [53] for clock period constraint of 370 ps. DC reports an area of 1304.4 μm^2, 23 violators, and a total negative slack of 1.4 ns. All reported violators are then analyzed and the *Gain* function is calculated for all the network forks.

Table B.1 shows the analysis results. Since fork $FABDE$ has the highest *Gain* of 38, it is chosen to be rolled back. $FABDE$ is preferred over $FABE$, because 4 of the violators that pass through both of them in the *Valid* direction (i.e., type I), pass only through $FABDE$ in the *Stall* direction. An example of such violators is: (start from $FA/R_2/Q$), $VA2$, $VABE$, $VABE2$, $VABDE$, $VABDE2$, (through join $JABCDE$), $SABDE2$, $SABDE$, (end at SD). Besides, two violators end at the internal registers of $FABE$ coming in the *Stall* direction (i.e., Type VI).

Hence, $FABDE$ is rolled back and the new control network is synthesized again with the same timing constraints (i.e., 370 ps clock period). DC reports an area of 1174.2 μm^2,

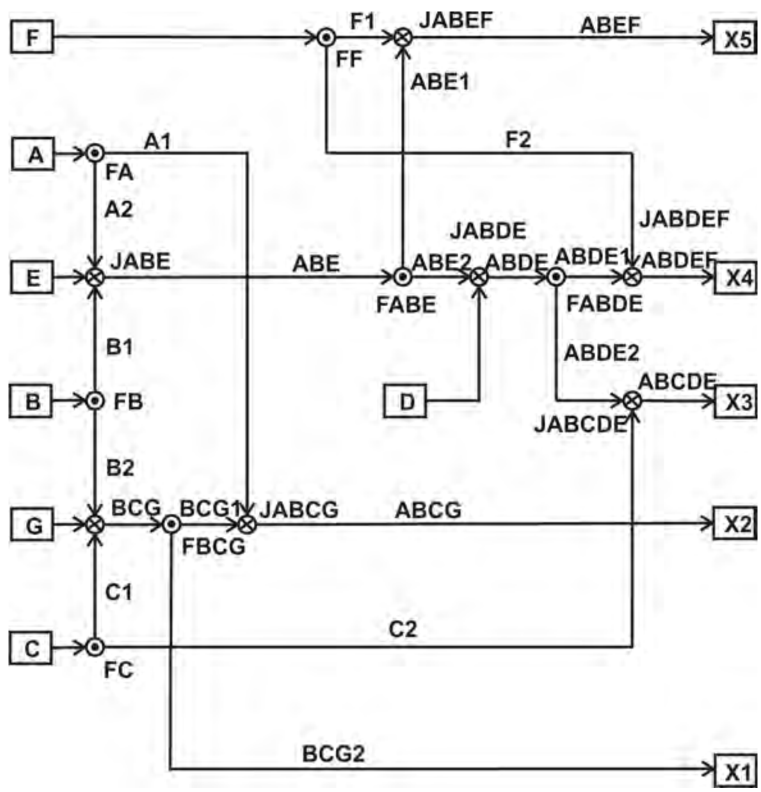


Figure B.5: Control network of Example B.2.

Table B.1: Iteration 1 for Example B.2.

	$FBCG$	$FABE$	$FABDE$
Type I	0	21	21
Type II	0	13	17
Type III	0	0	0
Type IV	0	0	0
Type V	0	0	0
Type VI	0	2	0
Gain	0	32	38

9 violators and total negative slack of only 0.1 ns. Violators are similarly analyzed. $FABE$ is rolled back. Then, the network is synthesized. DC reports an area of 1195.8 μm^2 and no violations. Hence, the flow eliminated the whole negative slack (1.4 ns) in three iterations, with an area gain (i.e., decrease) of 8.3%. Results are summarized in Table B.2. Rolling back a fork involves adding redundant forks and joins to the design. However, this is compensated, in part, by join and fork combinations that take place. Besides, rolling a fork back makes it easier for DC to meet the timing constraints. This, in turn, seems to help DC optimizes the area more efficiently. The last column in Table B.2 shows the area of the

Table B.2: Example B.2 results.

#	Total Neg. Slack (ns)	Area (μm^2) @T=0.37 ns	Fork To Roll Back	Its *Gain*	Area (μm^2) @T=400 ns
1	1.4	1304.4	$FABDE$	38	852
2	0.1	1174.2	$FABE$	16	859
3	0.0	1195.8			940.8

control network in the different iterations when they are synthesized with 400 ns timing constraint (i.e., virtually no constraints). In that case, rolling the fork back costs an area degradation (i.e., increase) of 10.4%.

B.4 Verification

The correctness of the proposed structural transformations of Sec. B.1 is verified using a symbolic model checker, NuSMV [59]. It is verified that the control networks before and after the transformations are functionally equivalent. In other words, there is no sequence of inputs to the control network that produces different outputs in the two versions of the control network. In this section the correctness of rolling back a fork (Sec. B.1.3) is verified. Other transformations (i.e., of Sections B.1.1 and B.1.2) can be similarly verified.

Fig. B.3 showed rolling back an n-output fork through an m-input join. For brevity, the case of n=2 and m=2 is verified. Higher values of n and m have also been verified. The setup of Fig. B.6 is used. Elastic buffer controllers $I1$ and $I2$ are connected to controllers $X1$ and $X2$ through two versions of the control network. The one on the top (designated 'Before') is the control network before doing any transformations. The one on the bottom (designated 'After') is the control network after rolling back fork $FI1I2$ through join $JI1I2$. Green lines represent the $Valid$ signals of the control channels. Red lines represent the $Stall$s. Suffixes $\_B$ and $\_A$ are used to designate the outputs of the control network before and after the transformation, respectively. The inputs coming from the controllers (i.e., $VI1$, $VI2$, $SX1$, and $SX2$) are applied to both networks simultaneously. The corresponding two network outputs (i.e., $VX1$, $VX2$, $SI1$, and $SI2$) are ORed together, respectively, and then passed to the controllers. For example, $VX1\_B$ and $VX1\_A$ are ORed and passed to the input $Valid$ pin of controller $X1$. The different components of Fig. B.6 are connected synchronously in NuSMV similar to [57]. Synchronous connection guarantees that all components of the design advance synchronously. The delay of each component is then encoded in individual counters in terms of the global time unit used by NuSMV. Without loss of generality, all combinational logic are assumed to have zero delay. NuSMV verification models for joins, forks, etc. are similar to those presented in Sec. 5.3.

The following PSL [60] properties are used to check the functional equivalence of the two versions of the control network (i.e., before and after the transformation):

```
DEFINE VX1_MISMATCH := VX1_B xor VX1_A ;
PSLSPEC never VX1_MISMATCH;
-- Similarly check VX2, SI1, SI2.
```

All the properties are proven true by NuSMV which guarantees functional equivalence between the two versions of the control network. It also proves the correctness of the transformation (rolling back a fork).

B.5 Case Studies and Results

This section presents two case studies: the MiniMIPS processor and the s298 ISCAS-89 benchmark. Results are synthesis numbers. Design Compiler® (DC) is used as a synthesis tool with an ARM® 65 nm library. DC Ultra™ is run with -timing_script to ensure the highest performance optimization effort. To minimize the area, set_max_area is set to zero.

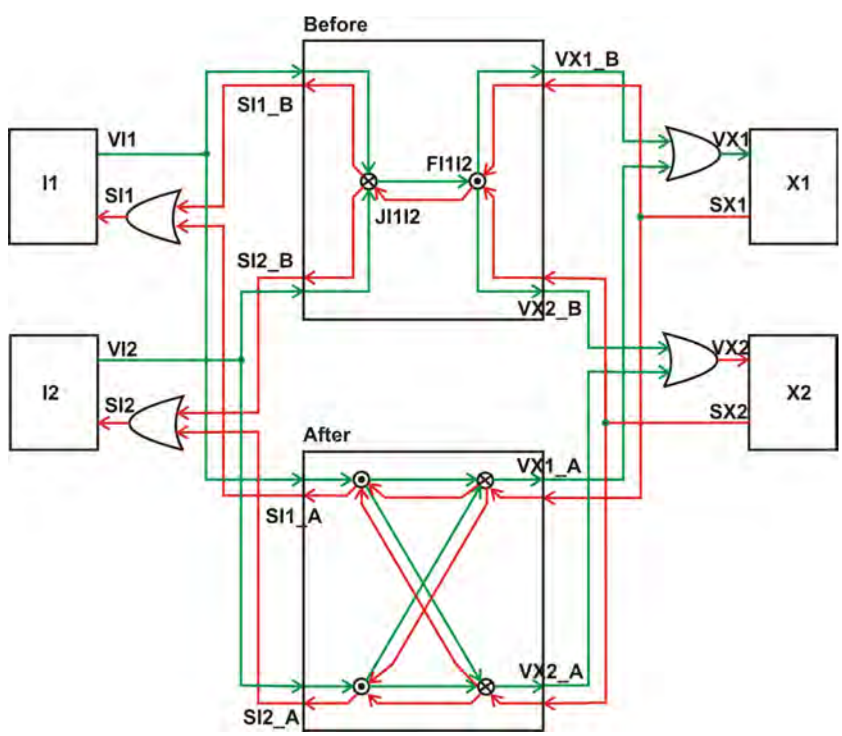


Figure B.6: Verification setup for rolling back a fork.

B.5.1 MiniMIPS

MIPS (Microprocessor without Interlocked Pipeline Stages) is a 32-bit architecture, first designed by Hennessy [46]. MiniMIPS is an 8-bit subset of MIPS. A block diagram of the original clocked MiniMIPS is shown in Fig. 2.5. The MiniMIPS synchronous elasticization is described in Sec. 2.2. The CNG-generated elastic control network is in Fig. 3.9. The MiniMIPS control network (with elastic buffer controllers for the register file and for the memory) is passed to the CNGT flow in order to meet a clock period constraint of 370 ps. The results are shown in Table B.3. The flow eliminated, in only one iteration, the whole negative slack (1.3 ns), with an area gain (i.e., decrease) of 6.2%. As argued in Example B.2, rolling back a fork involves adding redundant forks and joins to the design. However, this is compensated, in part, by join and fork combinations that take place. Besides, rolling a fork back makes it easier for DC to meet the timing constraints. This, in turn, seems to help DC optimizes the area more efficiently. The last column in Table B.3 shows the area of the control network in the different iterations when they are synthesized with 400 ns timing constraint (i.e., virtually no constraints). In that case, rolling the fork back costs an area degradation (increase) of 6.5%.

B.5.2 S298

S298 is an ISCAS-89 benchmark. It is a traffic light controller. S298 has a total of 23 synchronization points (14 registers + 3 inputs + 6 outputs). After analyzing all the register-to-register communications in the data path, the required connections are passed to the CNG tool. The resultant control network is shown in Fig. B.7. The s298 control network is passed to the CNGT flow in order to meet a clock period constraint of 500 ps. The results are shown in Table B.4. CNGT eliminated, in 3 iterations, the whole negative slack (5.3 ns), with an area degradation (i.e., increase) of only 0.4%.

Table B.3: MiniMIPS results.

#	Total Neg. Slack (ns)	Area (μm^2) @T=0.37 ns	Fork To Roll Back	Its *Gain*	Area (μm^2) @T=400 ns
1	1.3	1350	$FABCI4P$	35	953.4
2	0.0	1266			1015.2

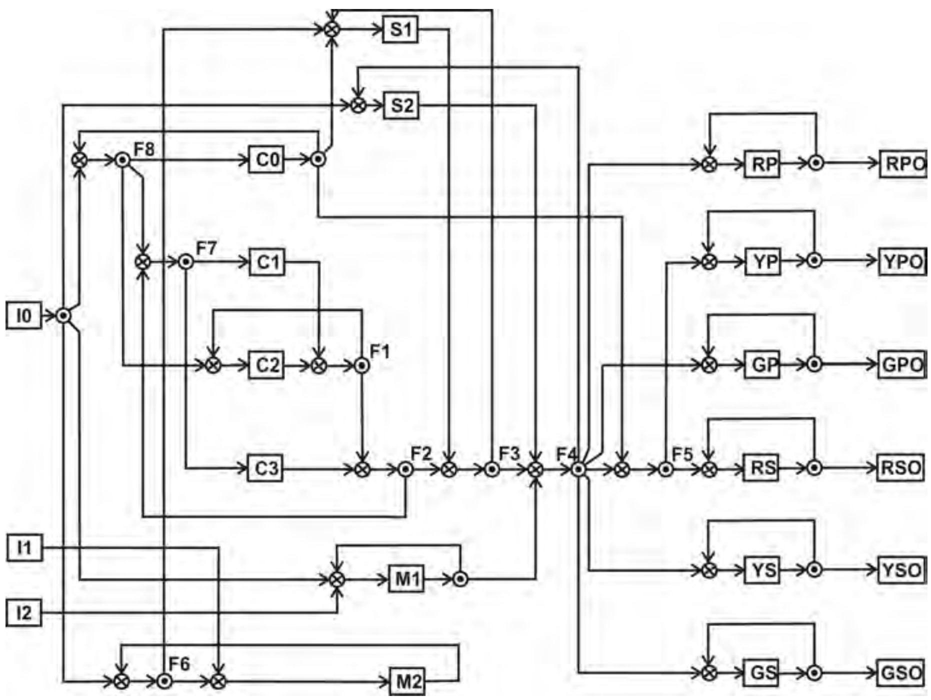


Figure B.7: Control network of the synchronous elastic version of s298.

Table B.4: S298 results.

#	Total Neg. Slack (ns)	Area (μm^2) @T=0.5 ns	Fork To Roll Back	Its *Gain*	Area (μm^2) @T=400 ns
1	5.3	2657.4	$F5$	70	1991.4
2	2.2	2799	$F3$	42	1977
3	0.4	2392.8	$F4$	36	1989.6
4	0.0	2668.8			2374.2

REFERENCES

[1] N. Weste and D. Harris, *CMOS VLSI design: a circuits and systems perspective.* Addison Wesley, 2004.

[2] L. Carloni, K. Mcmillan, and A. L. Sangiovanni-VincentelliR, "Theory of latency insensitive design," in *IEEE Transactions on CAD of Integrated Circuits and Systems*, vol. 20, no. 9, Sep. 2001, pp. 1059–1076.

[3] J. Carmona, J. Cortadella, M. Kishinevsky, and A. Taubin, "Elastic circuits," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 28, no. 10, pp. 1437–1455, Oct. 2009.

[4] C. J. Myers, *Asynchronous Circuit Design.* John Wiley & Sons, New York, 2001.

[5] A. J. Martin, "The limitations to delay-insensitivity in asynchronous circuits," in *Proceedings of the sixth MIT conference on Advanced research in VLSI.* Cambridge, MA, USA: MIT Press, 1990, pp. 263–278. [Online]. Available: http://dl.acm.org/citation.cfm?id=101415.101434

[6] J. Cortadella, A. Kondratyev, L. Lavagno, and C. Sotiriou, "Desynchronization: Synthesis of asynchronous circuits from synchronous specifications," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 25, no. 10, pp. 1904–1921, Oct. 2006.

[7] N. Andrikos, L. Lavagno, D. Pandini, and C. Sotiriou, "A fully-automated desynchronization flow for synchronous circuits," in *Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE*, Jun. 2007, pp. 982–985.

[8] H. M. Jacobson, P. N. Kudva, P. Bose, P. W. Cook, S. E. Schuster, E. G. Mercer, and C. J. Myers, "Synchronous interlocked pipelines," in *8th International Symposium on Asynchronous Circuits and Systems*, Apr. 2002, pp. 3–12.

[9] J. Cortadella, M. Kishinevsky, and B. Grundmann, "Synthesis of synchronous elastic architectures," in *ACM/IEEE Design Automation Conference*, Jul. 2006, pp. 657–662.

[10] S. Krstic, J. Cortadella, M. Kishinevsky, and J. O'Leary, "Synchronous elastic networks," in *Formal Methods in Computer Aided Design, 2006. FMCAD '06*, Nov. 2006, pp. 19–30.

[11] L. Carloni, K. McMillan, and A. Sangiovanni-Vincentelli, "Latency insensitive protocols," in *The 11th International Conference on Computer-Aided Verification*, Jul. 1999.

[12] ITRS report 2009 edition. Available online at. http://www.itrs.net/Links/2009ITRS/2009Chapters_2009Tables/2009_Design.pdf.

[13] A. Nieuwoudt, J. Kawa, and Y. Massoud, "Crosstalk-induced delay, noise, and interconnect planarization implications of fill metal in nanoscale process technology," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 18, no. 3, pp. 378 –391, Mar. 2010.

[14] S. Shah, N. Mansouri, and A. Nunez-Aldana, "Pre-layout estimation of interconnect lengths for digital integrated circuits," in *Electronics, Communications and Computers, 2006. CONIELECOMP 2006. 16th International Conference on*, Feb. 2006, p. 38.

[15] M. Bohr, "Interconnect scaling-the real limiter to high performance ULSI," in *Electron Devices Meeting, 1995., International*, Dec. 1995, pp. 241–244.

[16] R. Ho, K. Mai, H. Kapadia, and M. Horowitz, "Interconnect scaling implications for CAD," in *Computer-Aided Design, 1999. Digest of Technical Papers. 1999 IEEE/ACM International Conference on*, 1999, pp. 425–429.

[17] L. Carloni and A. Sangiovanni-Vincentelli, "Coping with latency in SoC design," *Micro, IEEE*, vol. 22, no. 5, pp. 24–35, Sep./Oct. 2002.

[18] ——, "Performance analysis and optimization of latency insensitive systems," in *Design Automation Conference, 2000. Proceedings 2000. 37th*, 2000, pp. 361 –367.

[19] D. Bufistov, J. Julvez, and J. Cortadella, "Performance optimization of elastic systems using buffer resizing and buffer insertion," in *Computer-Aided Design, 2008. ICCAD 2008. IEEE/ACM International Conference on*, 10-13 2008, pp. 442 –448.

[20] P. Cocchini, "Concurrent flip-flop and repeater insertion for high performance integrated circuits," in *Computer Aided Design, 2002. ICCAD 2002. IEEE/ACM International Conference on*, 10-14 2002, pp. 268 – 273.

[21] R. Collins and L. Carloni, "Topology-based optimization of maximal sustainable throughput in a latency-insensitive system," in *Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE*, Jun. 2007, pp. 410 –415.

[22] L. Carloni, K. McMillan, A. Saldanha, and A. Sangiovanni-Vincentelli, "A methodology for correct-by-construction latency insensitive design," in *Computer-Aided Design, 1999. Digest of Technical Papers. 1999 IEEE/ACM International Conference on*, 1999, pp. 309 –315.

[23] T. Kam, M. Kishinevsky, J. Cortadella, and M. Galceran-Oms, "Correct-by-construction microarchitectural pipelining," in *Computer-Aided Design, 2008. ICCAD 2008. IEEE/ACM International Conference on*, Nov. 2008, pp. 434 –441.

[24] M. Galceran-Oms, J. Cortadella, D. Bufistov, and M. Kishinevsky, "Automatic microarchitectural pipelining," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, Mar. 2010, pp. 961 –964.

[25] M. Galceran-Oms, Ph.D. dissertation.

[26] J. You, Y. Xu, H. Han, and K. S. Stevens, "Performance evaluation of elastic GALS interfaces and network fabric," *Electronic Notes in Theoretical Computer Science*, vol. 200, no. 1, pp. 17 – 32, 2008, proceedings of the Third International

Workshop on Formal Methods for Globally Asynchronous Locally Synchronous Design (FMGALS 2007). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1571066108000868

[27] D. Gebhardt and K. S. Stevens, "Elastic flow in an application specific network-on-chip," in *in: Third International Workshop on Formal Methods in Globally Asynchronous Locally Synchronous Design (FMGALS 07), Elsevier Electronic Notes in Theoretical Computer Scinece*, 2007.

[28] L. Benini and G. De Micheli, "Networks on chip: a new paradigm for systems on chip design," in *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*, 2002, pp. 418 –419.

[29] ——, "Networks on chips: a new SoC paradigm," *Computer*, vol. 35, no. 1, pp. 70 –78, Jan. 2002.

[30] A. Gotmanov, M. Kishinevsky, and M. Galceran-Oms, "Evaluation of flexible latencies: Designing synchronous elastic H.264 CABAC decoder." in *The Problems in design of micro- and nano-electronic systems*, Oct. 2010.

[31] L. Benini, G. De Micheli, A. Lioy, E. Macii, G. Odasso, and M. Poncino, "Automatic synthesis of large telescopic units based on near-minimum timed supersetting," *Computers, IEEE Transactions on*, vol. 48, no. 8, pp. 769 –779, Aug. 1999.

[32] J. Cortadella and M. Kishinevsky, "Synchronous elastic circuits with early evaluation and token counterflow," in *Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE*, Jun. 2007, pp. 416 –419.

[33] S. Suhaib, D. Mathaikutty, D. Berner, and S. Shukla, "Validating families of latency insensitive protocols," *Computers, IEEE Transactions on*, vol. 55, no. 11, pp. 1391 –1401, Nov. 2006.

[34] I. Blunno, J. Cortadella, A. Kondratyev, L. Lavagno, K. Lwin, and C. Sotiriou, "Handshake protocols for de-synchronization," in *Asynchronous Circuits and Systems, 2004. Proceedings. 10th International Symposium on*, Apr. 2004, pp. 149 – 158.

[35] S. Furber and P. Day, "Four-phase micropipeline latch control circuits," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 4, no. 2, pp. 247 –253, Jun. 1996.

[36] G. Birtwistle and K. S. Stevens, "The family of 4-phase latch protocols," in *Asynchronous Circuits and Systems, 2008. ASYNC '08. 14th IEEE International Symposium on*, Apr. 2008, pp. 71–82.

[37] K. Stevens, Y. Xu, and V. Vij, "Characterization of asynchronous templates for integration into clocked CAD flows," in *Asynchronous Circuits and Systems, 2009. ASYNC '09. 15th IEEE Symposium on*, 17-20 2009, pp. 151 –161.

[38] O. Roig, J. Cortadella, M. Peiia, and E. Pastor, "Automatic generation of synchronous test patterns for asynchronous circuits," in *Design Automation Conference, 1997. Proceedings of the 34th*, Jun. 1997, pp. 620 –625.

[39] F. te Beest, A. Peeters, K. van Berkel, and H. Kerkhoff, "Synchronous full-scan for asynchronous handshake circuits," *Journal of Electronic Testing*, vol. 19, pp. 397–406, 2003, 10.1023/A:1024687809014. [Online]. Available: http://dx.doi.org/10.1023/A:1024687809014

[40] O. Petlin and S. Furber, "Scan testing of micropipelines," in *VLSI Test Symposium, 1995. Proceedings., 13th IEEE*, Apr.-3 May 1995, pp. 296 –301.

[41] L. P. Carloni, "The role of back-pressure in implementing latency-insensitive systems," in *Electronic Notes in Theoretical Computer Science*, 2006, pp. 61–80.

[42] C.-H. Li, R. Collins, S. Sonalkar, and L. P. Carloni, "Design, implementation, and validation of a new class of interface circuits for latency-insensitive design," in *Fifth ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, 2007.

[43] J. Julvez, J. Cortadella, and M. Kishinevsky, "Performance analysis of concurrent systems with early evaluation," in *Computer-Aided Design, 2006. ICCAD '06. IEEE/ACM International Conference on*, Nov. 2006, pp. 448 –455.

[44] M. Galceran-Oms, J. Cortadella, and M. Kishinevsky, "Speculation in elastic systems," in *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*, Jul. 2009, pp. 292 –295.

[45] E. Kilada, S. Das, and K. Stevens, "Synchronous elasticization: Considerations for correct implementation and MiniMIPS case study," in *VLSI System on Chip Conference (VLSI-SoC), 2010 18th IEEE/IFIP*, Sep. 2010, pp. 7 –12.

[46] J. L. Hennessy, N. P. Jouppi, J. Gill, F. Baskett, A. Strong, T. R. Gross, C. Rowen, and J. Leonard, "The MIPS machine." in *COMPCON'82*, 1982, pp. 2–7.

[47] E. Kilada and K. Stevens, "Control network generator for latency insensitive designs," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, Mar. 2010, pp. 1773 –1778.

[48] ——, "Theory and implementation of CNG: a control network generator for elastic circuits," in *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, submitted.

[49] E. Kilada and K. S. Stevens, "Design and verification of lazy and hybrid implementations of the SELF protocol," in *VLSI-SoC: forward-looking trends in IC and system design (best papers of VLSI-SoC 2010)*, J. L. Ayala, D. Atienza, and R. Reis., Eds. Springer, 2011, to appear.

[50] J. Carmona, J. Júlvez, J. Cortadella, and M. Kishinevsky, "A scheduling strategy for synchronous elastic designs," *Fundam. Inform.*, vol. 108, no. 1-2, pp. 1–21, 2011.

[51] IBM® SixthSense. http://domino.research.ibm.com/comm/research_projects.nsf/pages/sixthsense.index.html.

[52] E. Kilada and K. Stevens, "Synchronous elasticization at a reduced cost: Utilizing the ultra simple fork and controller merging," in *Computer-Aided Design, 2011. ICCAD 2011. IEEE/ACM International Conference on*, Nov. 2011.

[53] Synopsys® Design Compiler®. http://www.synopsys.com/Tools/Implementation/RTLSynthesis.

[54] R. K. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool." in *22nd International Conference on Computer Aided Verification. CAV'10.*, 2010, pp. 24–40.

[55] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design*. Morgan Kaufmann Publishers, 2004.

[56] B. Chapman, G. Jost, and R. van van der Pas, *Using OpenMP: Portable Shared Memory Parallel Programming*. MIT Press, Cambridge, Mass., 2008.

[57] V. Vakilotojar and P. Beerel, "RTL verification of timed asynchronous and heterogeneous systems using symbolic model checking," in *Design Automation Conference 1997. Proceedings of the ASP-DAC '97. Asia and South Pacific*, 28-31 1997, pp. 181 –188.

[58] G. Hoover and F. Brewer, "Synthesizing synchronous elastic flow networks," in *Design, Automation and Test in Europe, 2008. DATE '08*, 10-14 2008, pp. 306 –311.

[59] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV 2: An opensource tool for symbolic model checking." in *Proc. of 14th Conf. on Computer Aided Verification (CAV 2002)*, vol. 2404, Jul. 2002.

[60] "IEEE standard for property specification language (PSL)," *IEEE Std 1850-2010 (Revision of IEEE Std 1850-2005)*, pp. 1 –171, 6 2010.

[61] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Trans. Program. Lang. Syst.*, vol. 8, no. 2, pp. 244–263, 1986.

[62] "Heuristic methods," in *Complexity and approximation*, G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M.Protasi, Eds. Springer-Verlag Berlin Heidelberg, 1999.

[63] J. Carmona, J. Julvez, J. Cortadella, and M. Kishinevsky, "Scheduling synchronous elastic designs," in *Application of Concurrency to System Design, 2009. ACSD '09. Ninth International Conference on*, Jul. 2009, pp. 52 –59.