The Dissertation Committee for Cuong Kim Chau
certifies that this is the approved version of the following dissertation:

A Hierarchical Approach to Formal Modeling and Verification of Asynchronous Circuits

Committee:

Warren A. Hunt, Jr., Supervisor

Mohamed Gouda

Vladimir Lifschitz

Marly Roncken

Ivan Sutherland

A Hierarchical Approach to Formal Modeling and Verification of Asynchronous Circuits

by

Cuong Kim Chau

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2019

Dedicated to my maternal grandparents.

Acknowledgments

I would like to thank my supervisor and mentor, Warren Hunt, for his guidance, support, and encouragement. I enjoyed my time with him discussing ideas, research, industry, and life in general. He also gave me the freedom to explore ideas. I am grateful to Matt Kaufmann for his exceptional mentoring. I know I would not have got this far without his help. He was the person in our research group I felt most comfortable discussing technical and mathematical problems with. He also played a significant role in my learning to use ACL2. I thank J Moore for bringing me to the ACL2 world as well as the UT ACL2 research group. I felt very lucky to be part of this super friendly and supportive research group.

I thank Marly Roncken and Ivan Sutherland for their useful feedback, corrections, and discussions on this project. I thank Mohamed Gouda and Vladimir Lifschitz for their time, expertise, and feedback. I thank Ruben Gamboa and Marijn Heule for helping me through the other projects involving ACL2r and SAT solving, respectively.

I want to thank the people I worked with at Oracle and Centaur Technology, including David Rager, Jo Ebergen, Anna Slobodova, Jared Davis, Sol Swords, and Shilpi Goel. My ACL2 skills, especially arithmetic reasoning and defining macros, improved a lot while working with them.

Thanks to Shilpi Goel for being a good friend of mine. She was always available when I needed her help. We had a lot of fun hanging out and watching movies together. I thank Lindy Aleshire for her excellent administration help. She and Matt Kaufmann are my wonderful English teachers. It is always fun to chat with Lindy about our taste in music.

I thank the Defense Advanced Research Projects Agency (DARPA) for their support. The work presented in this dissertation was supported by DARPA under contract number FA8650-17-1-7704.

Thanks to my friends and my volleyball people for keeping me a sane and happy human being. Finally, I want to thank my family, my brother, my sister, and especially, my maternal grandparents. They have always been and will forever be with me in my life journey.

A Hierarchical Approach to Formal Modeling and Verification of Asynchronous Circuits

Publication No. ____________________

Cuong Kim Chau, Ph.D.
The University of Texas at Austin, 2019

Supervisor: Warren A. Hunt, Jr.

The self-timed (or asynchronous) approach to circuit design has demonstrated benefits in a number of different areas for its low energy consumption, high operating speed, composability, and modularity. Nonetheless, the asynchronous paradigm exposes challenges that are not found in the synchronous (or clock-driven) paradigm. For the verification task, a challenge emerges from the large number of potential operational interleavings exhibited in the asynchronous paradigm. Simply exploring all interleavings is, in general, intractable because the number of interleavings can grow exponentially.

This dissertation focuses on developing scalable methods that are capable of reasoning effectively about the interleaving problem exhibited in self-timed systems. We specify and verify finite-state-machine representations of self-timed circuit designs using the DE system, a formal hardware description language defined using the ACL2 theorem-proving system. We apply a link-joint paradigm to model self-timed circuits as networks of channels that communicate with each other locally via handshake protocols. This link-joint model has been shown to be a universal model for various self-timed circuit families. In addition, this model has a clean formalization in the ACL2 logic and provides a protocol level that abstracts away timing constraints at the circuit level.

Unlike many efforts for validating timing and communication properties of self-timed systems, we are interested in verifying functional properties. Specifically, we verify the functional correctness of self-timed systems in terms of relationships between their input and output sequences. To mitigate the consideration of all interleavings simultaneously, we address the verification problem hierarchically and avoid exploring the internal structures of verified submodules as well as their operational interleavings. The input-output relationship of a verified submodule is determined based on the communication signals at the submodule's input and output ports, while abstracting away all execution paths internal to that submodule.

Table of Contents

List of Tables

List of Figures

Part I

Preliminaries

Chapter 1

Introduction

Self-timed (or asynchronous) circuits have shown their potential advantages over clock-driven (or synchronous) circuits for low energy consumption, high operating speed, low electromagnetic interference, elimination of clock skew problems, and offering of composability and modularity [39, 78, 45, 47, 77, 3, 2, 44, 64]. On the other hand, the self-timed paradigm encounters challenges that do not occur in the clocked paradigm. Verification of large self-timed systems must deal with the large number of operational interleavings exhibited in those systems. As verification is a critical component of any complex digital design, *scalable* methods for self-timed system verification are highly desirable.

Although research on self-timed circuit design is promising, results about the verification of self-timed systems have been limited. Most research efforts in self-timed system verification have appealed to timing verification techniques to validate handshake protocols implemented in self-timed systems [41, 36, 8, 7, 4, 34, 29, 15, 37, 51]. This dissertation takes a different approach: developing a *hierarchical* (or *compositional*) methodology for verifying self-timed circuits' functionality. Our approach views self-timed circuits as networks of communication channels and functional units, while ignoring circuit-level timing constraints by relying on the timing analysis, as suggested by Park et al. [51]. Using the ACL2 theorem-proving system [33, 1, 32, 31], we develop a framework for specifying and verifying the functional correctness of those networks.

Our work focuses on developing scalable methods for reasoning about the functional correctness of self-timed systems. Our verification framework appeals to *hierarchical reasoning* and *induction* to support scalability [10]. We specify and verify self-timed circuit designs using the DE (Dual-Eval) system [25], which is defined in the ACL2 logic. DE is a formal *hardware description language* (HDL) that permits the hierarchical definition of Mealy machines. It has been used to specify and verify microprocessor designs [9, 27]. DE provides a library of verified hardware circuit generators that can be used to develop and analyze complex hardware systems [9]. A key feature of the DE system is that it supports hierarchical verification, which is critical for verifying correctness of circuit behavior at large scale. By abstracting away the internal structures of verified subsystems, hierarchical reasoning is amenable to verifying correctness of large systems. Moreover, this method enables the localization of faults to subsystems; thus permitting the resolution of faults to occur locally, while the verification procedure for bigger systems containing those subsystems still remains unaltered.

Our self-timed modeling is based on the *link-joint model* proposed by Roncken et al. [58], a universal generalization for various self-timed circuit families (e.g., Click [54], Mousetrap [62], Micropipeline [69], and GasP [70]). We use DE to model self-timed circuits as networks of (communication) links and (computation) joints that inter-operate via the link-joint model. Self-timed circuits and systems operate in a CSP-style manner; communication is coordinated locally, on a point-to-point basis. In circuit implementation terms, this means that instead of using a global-clock signal to indicate when all clocked storage elements should accept new data, self-timed storage elements (links) accept input only when they are ready and the input data are valid — and when a link accepts new input, it signals to its datum provider that the provider may proceed to calculate its next output value(s). We formally model all computation joints introduced by Roncken et al. [59]. These joints cover various computation models that are sufficient to build general-purpose computing machines such as computers.

A key issue addressed by our self-timed circuit model is allowing links and joints to proceed at their own rate. When we compose circuit modules we prove, no matter how each module might proceed internally, that their composition meets a specification consistent with the composed module's specification. Our approach scales by combining properties of verified submodules without concern for their internal structures. This is a key enabler for self-timed circuit verification, as the number of interleavings grows exponentially as the circuit size increases. When we confirm that a circuit module meets a functional (stream-oriented) specification, we consider all possible interleavings of its internal operations. Circuit interleavings are considered in a hierarchical manner; we first verify that a module meets its specification, and then we use only this specification when including this module within another module. This allows consideration of all possible circuit interleavings without an explosion of cases that must be considered. To our knowledge, we are the first to apply theorem proving with a hierarchical functional verification methodology to self-timed circuit models designed at the link-joint level.

We apply our methodology to various self-timed circuit models, including circuits performing non-deterministically arbitrated merges. The case studies presented in this dissertation are sufficiently complex to demonstrate the generality and scalability of our approach. They perform a variety of behaviors that are also present in modern microprocessors, such as pipelining, arithmetic and logical operations, bit shifting, conditional branching and merging, counting, register models, and first-come-first-served (FCFS) arbitrated merging. Given the experiments provided in this dissertation, we expect our framework can be applied to *very large scale integration* (VLSI) systems, such as self-timed microprocessors.

We present the motivation of this dissertation project in Section 1.1. A survey of existing work related to this research is discussed in Section 1.2. In Sections 1.3 and 1.4, we present the objectives and contributions of this project, respectively. The organization of this dissertation is given in Section 1.5.

1.1 Motivation

The removal of a global clock from self-timed systems shows potential benefits in many aspects of system design. This potentially reduces power consumption due to low standby power consumption on data movement, which occurs only when and where needed [72, 71, 17, 20, 46, 48]. On the other hand, the constant activity of a global clock in synchronous systems consumes power by distributing the clock signal to every part of those systems, even though some parts may not be processing any data.

Synchronous circuit designers must deal with clock skew problems that the global clock signal arrives at different components of a circuit at different times due to a variety of reasons, such as differences in physical placement, temperature, and path length. As a result, the clock rate depends on the global worst-case scenario to ensure that all data have stabilized before being sampled. The clock period need be increased to ensure correct operation in the presence of clock skew, thus yielding slower circuits. The absence of a global clock in the self-timed paradigm may yield higher performance since it eliminates clock skew problems and the operating speed is determined by actual local latencies rather than the global worst-case latency as specified in the clocked paradigm [40].

Synchronous circuits create substantial electromagnetic noise in very narrow frequency bands around the clock frequency and its harmonics; self-timed circuits generate electromagnetic interference that is much more evenly distributed across the spectrum with lower peak noise [72, 53]. This is because there is less correlation between operations in self-timed circuits; they tend to be executed at different points in time, resulting in a more distributed noise spectrum.

Another advantage of self-timed design is that it offers modularity: self-timed modules can plug together through simple handshake protocols, and they simply work without any external clock because each module acts only in its proper order [43, 38, 49, 69, 65]. Individual components in self-timed circuits have interfaces that operate correctly with arbitrary delays in the wires used for inter-module communication, i.e., interfaces use *delay-insensitive* (DI) protocols [75]. On the contrary, composing modules in clocked design requires validating global timing constraints on the composed circuit. This in turn can result in modifying the design in order to meet the global timing requirements, and consequently requiring re-verification of the entire design.

Despite the advantages mentioned above, lack of *computer-aided design* (CAD) tools and verification methods prevents the self-timed paradigm from being widely adopted by industry. This dissertation attempts to improve the verification methodology for self-timed systems. More specifically, we aim to develop a specification and mechanical verification environment for analyzing self-timed systems. Unlike many efforts in validating timing and communication properties of self-timed systems, we are interested in verifying functional properties of these systems. Our work relies on local timing analysis to justify our abstraction of self-timed circuits to finite-state-machine representations of networks of communication channels, thus ignoring circuit-level timing constraints. The expected outcome of our work is a framework for formally specifying and verifying the functional correctness of self-timed systems, which we believe is valuable to the asynchronous community. We also believe that this work provides foundational principles of self-timed circuit verification so that future work can apply and build on top of them.

1.2 Related Work

Most verification efforts that use formal techniques for analyzing self-timed circuit implementations concern circuit-level timing properties. Depending on the choice of technology (e.g., delay insensitive (DI) [68], quasi-delay insensitive (QDI) [42], bundled data [69], etc.), electrical-level timing analysis must be conducted to assure that signal propagation of ready signals is always slower than data propagation so that data are valid when sampled.

Timing verification of self-timed circuits has been investigated by several groups [41, 36, 8, 7, 4, 34, 29, 15, 37, 51]. For instance, Park et al. [51] developed the ARCtimer framework for modeling, generating, and verifying timing constraints on individual handshake components. ARCtimer uses the NuSMV model checker for its analysis. The authors' goal was to ensure that the network of logic gates and wires, along with their associated delays, meets the component's protocol requirements. In contrast, our goal concerns proving that a self-timed circuit or system meets its functional specification, while ignoring circuit-level timing constraints that can be investigated by tools like ARCtimer.

Most existing work on self-timed circuit verification has either examined circuits that do not include any data path or do not concern computations on the data paths in its verification objectives. These methods have primarily explored strategies for checking communication sequences with respect to a specification. For example, Dill [16] developed a trace theory for hierarchical verification of communication sequences in speed-independent circuits. The author focused only on control circuits, while data circuits were not involved. His method checks circuit properties by simply searching through the state-transition graph that models the circuit behavior. Although this approach is automatic, it explicitly represents and stores all possible states. This is quite inefficient when dealing with circuits consisting of a large number of states. And while the author proposed two theories for modeling and checking safety and liveness properties of speed-independent circuits, only the theory for dealing with safety properties was implemented.

The use of hierarchical verification methods in self-timed circuit contexts has also been explored by Clarke and Mishra [13], in their attempt to verify safety and liveness circuit properties automatically. Their analysis approach is based on model checking, and they investigated the correctness of a self-timed FIFO queue element specified in Computation Tree Logic (CTL). Their approach assumes a unit delay for each gate in a self-timed circuit, where our approach avoids imposing any restrictions on gate delays.

Previous applications of ACL2 to asynchronous circuit designs have focused on properties other than their functional correctness. Verbeek and

Schmaltz [74] have formalized and verified *blocking* (failing to transmit data) and *idle* (failing to receive data) conditions about delay-insensitive primitives from the Click circuit library. By using ACL2, these conditions were translated into SAT/SMT instances to confirm deadlock freedom in the self-timed circuits investigated. Peng et al. [56] presented a framework for detecting glitches that occur in synthesized clock-domain-crossing netlists but are not apparent in the original RTL specifications. The authors' approach integrates ACL2 with a SAT solver for verifying, in synthesized netlists, the glitch-free property of each state-bit associated with a corresponding flip-flop output. They demonstrated their tool on commercial designs from Oracle Microelectronics.

Loewenstein [35] formally verified some properties of the asynchronous, Sproull *counterflow pipeline processor* (CFPP) architecture using the higher-order logic, HOL theorem prover [21, 22]. He modeled the CFPP as an automaton at an abstract, architectural level. However, no connection between that high-level model and a low-level circuit design was conducted. In contrast, we are interested in verifying that the gate-level netlist description of a self-timed circuit complies with its high-level functional specification.

Srinivasan and Katti [67] applied a refinement-based method for verifying safety properties of desynchronized pipelined circuits. Their approach attempts to reduce non-determinism by adding extra sequential dependencies between controller events to circuit designs. Wijayasekara et al. [76] applied the same method for verifying the functional equivalence of NULL Convention Logic (NCL) circuits against their synchronous counterparts. While both frameworks are highly automated by using decision procedures, they provided quite limited scalability and no liveness properties were verified. Our approach, on the other hand, has been shown to be scalable by exploiting hierarchical verification and induction. We also avoid imposing any such additional restrictions on circuit designs.

1.3 Objectives

The goal of this dissertation is to develop a formal framework for hierarchically modeling and verifying functional properties of asynchronous circuit designs. This project concentrates on creating a comprehensive verification strategy by leveraging existing work in the clocked design paradigm and creating new approaches for verifying asynchronous systems. We follow the link-joint model introduced by Roncken et al. [58] in modeling self-timed systems as networks of storage links that communicate with each other via computation joints. Our modeling task attempts to represent self-timed circuits as cooperating finite-state-machines (FSMs) using the DE system — a formal HDL system already proven to be successful for synchronous circuit modeling and verification. DE provides combinational primitives as well as several latches suitable for creating links and joints. We propose to model self-timed systems fulfilling the following three facts that are absent from the synchronous paradigm.

1. Avoid a global clock signal; state-holding devices update their states based on local signaling.

2. Channels communicate with each other via local communication protocols.

3. Exhibit all possible interleavings of circuit operations due to variable delays in wires and gates.

For the verification task, our objective is to develop a methodology along with a library for verifying the functional correctness of self-timed circuit designs at scale. We are interested in developing a method that is amenable to analyzing the operational interleavings efficiently. A desired approach must be able to confirm correctness of circuit behavior under all possible orderings of circuit operations. The modularity of self-timed systems presents an opportunity for creating a modular, hierarchical verification flow. Hence our goal is to develop a hierarchical reasoning approach that is capable of verifying large systems efficiently without bothering about the internal details of their verified subsystems.

1.4 Contributions

We have generalized the DE-based, synchronous-style verification system to one that is capable of analyzing self-timed system models. This generalization advances analysis of circuit specification and verification, and provides a means to support building reliable complex hardware systems using the self-timed paradigm.

- We have provided a framework for modeling self-timed circuit designs using the link-joint paradigm. Our framework also supports simulation capability for those circuit models.

- We have develop a compositional, mechanized methodology for scalable formal verification of functional properties of self-timed circuit designs. This work has involved implementing strategies for reasoning about non-deterministic circuit behavior efficiently. Our methodology is able to deal with the non-determinism appearance in both event sequence and time at which events happen. In contrast to most of existing work of applying hierarchical reasoning to self-timed circuit and system verification, our approach verifies circuit functionality while others verify circuit timing and/or communication properties.

A successful outcome from this dissertation in connection with circuit-level implementation will support a computing specification and verification paradigm where systems can proceed at their best rate and no longer require clock signals.

1.5 Organization

The rest of this dissertation is organized as follows.

Part I: Preliminaries gives an introduction to this research, along with some background.

Chapter 2 reviews the DE system that we use to specify and verify self-timed circuit models.

Part II: Approach presents our modeling and verification methodology for self-timed circuit designs.

Chapter 3 introduces our modeling approach to self-timed circuit designs using the link-joint paradigm.

Chapter 4 introduces our hierarchical approach to functionally verifying self-timed circuit models.

Part III: Case Studies demonstrates the applicability of our methodology by modeling and verifying the functional correctness of various self-timed circuit designs.

Chapter 5 describes our modeling and verification of several circuit synthesizers that generate data-loop-free circuits.

Chapter 6 further demonstrates our framework through case studies of circuits that contain feedback loops in their data flows.

Chapter 7 describes our strategy for verifying circuits involving arbitrated merge operations. These operations are frequently used in self-timed systems in order to grant mutually exclusive accesses to shared resources. Unlike the circuits discussed in Chapters 5 and 6, circuits with arbitrated merges essentially produces non-deterministic outputs due to arbitrary arrival times of requests at arbitrated merges' inputs.

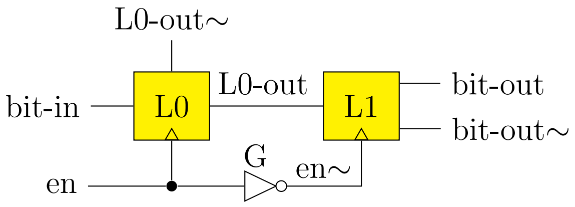
Part IV: Epilogue summarizes this research and gives some concluding remarks.

Chapter 8 concludes this dissertation and discusses potential for future research.

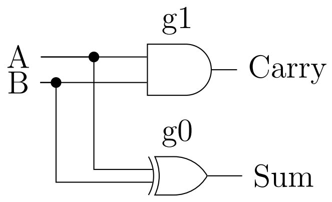
Chapter 2

DE System

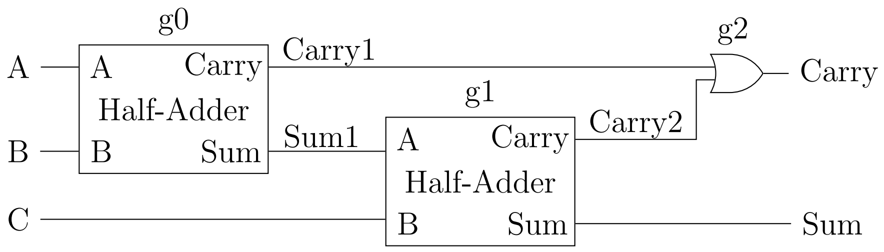
DE (Dual-Eval) is a circuit description language developed in ACL2 for describing and analyzing hierarchical Mealy machines [25]. It has previously been used to model hierarchical synchronous circuits where all state-holding primitives update their values simultaneously [25, 27, 9]. In this dissertation, we show that the DE system can be adapted for modeling and analyzing self-timed systems as well. The only extension we make is to add a single primitive to the DE primitive database that models the validity of data stored in a communication link. The benefit of using DE is that we are able to reuse the DE hierarchical circuit verification approach [25]; originally, this automated approach was used to verify the FM9001 microprocessor design [5]. We generalize the semantics of this HDL-based circuit specification and verification approach to allow the analysis of self-timed circuits whose implementations proceed at their best rate. In this chapter we will review the DE system and show how to use it model and evaluate circuit modules through concrete examples. Chapter 3 will describe our self-timed modeling using DE.



(a) Flip-flop



(b) Half-adder



(c) Full-adder

Figure 2.1: Schematic diagrams of three circuit examples

2.1 DE Description

A *well-formed DE netlist* is an ordered list of modules, where each module may include references to previously defined modules or to DE primitives. Each module definition consists of five ordered entries: a unique module name, input names, output names, internal-state names, and a list of occurrences that references previously defined submodules or DE primitives. Each occurrence in a module consists of four ordered entries: a module-unique occurrence name, a list of output names, a reference to a DE primitive or a submodule, and a list of input names. The DE system includes an ACL2 predicate that recognizes a syntactically well-formed netlist; this predicate enforces syntactic requirements on naming, arity, occurrence structure, and signal connectivity. Below is a DE netlist containing three module definitions: a flip-flop circuit built from two latches `L0` and `L1`, a half-adder, and a full-adder composed of two half-adders. Figure 2.1 offers the schematic diagrams of these three circuits.

```
(defconst *netlist*
  '((flip-flop             ;; Module's name
     (en bit-in)           ;; Inputs
     (bit-out bit-out~)   ;; Outputs
     (L0 L1)               ;; Internal states
     ;; Occurrences
     ((L0 (L0-out L0-out~)   latch (en bit-in))
      (G  (en~)              b-not (en))
      (L1 (bit-out bit-out~) latch (en~ L0-out))))

    (full-adder
     (c a b)
     (sum carry)
     ()  ;; No internal states
     ((g0 (sum1 carry1) half-adder (a b))
      (g1 (sum  carry2) half-adder (sum1 c))
      (g2 (carry)       b-or       (carry1 carry2))))

    (half-adder
     (a b)
     (sum carry)
     ()  ;; No internal states
     ((g0 (sum)   b-xor (a b))
      (g1 (carry) b-and (a b))))))
```

As the netlist example above may suggest, a module can have multiple references to a primitive or a submodule. However, for a physical realization, each reference indicates a completely new copy of the referenced primitive or submodule. In addition, the definition of a referenced module must appear after its referencing modules' definitions in the netlist. For instance, the definition *half-adder* appears after the definition of *full-adder* in the above netlist.

2.2 DE Simulator

The semantics of the DE language are given by a simulator whose *se* (*single eval*) function computes the outputs and whose *de* (*dual eval*) function computes the next state for a module from the module's current inputs and current state. The *de* simulation function operates in two passes: it first propagates values from primary inputs and internal states throughout the netlist, calculating values for every internal "wire". Once the values of all wires are known, *de* produces the module's outputs by accessing the appropriate wire values. To produce the next state, *de* makes a second pass over the entire netlist propagating previously-calculated wire values into storage elements.

Both simulation functions *se* and *de* require the following four ordered arguments: the name of the module to evaluate, its input values, its current-state value, and a well-formed DE netlist containing the definition of the module and submodules to be simulated. These simulation functions are actually defined in two sets of mutually recursive functions. The following two subsections discuss their definitions in detail.

2.2.1 Output Evaluator

Here we present the definitions of the output evaluator *se* and its mutually recursive peer function *se-occ*. Function *se* evaluates a module and returns its output values. *se* evaluates primitives using the *se-primp-apply* function. If argument `fn` identifies a defined module, its definition is extracted from `netlist` for further evaluation. Function *se-occ* evaluates occurrences and appends the newly computed outputs of the current occurrence onto a growing list of name-value pairs. Before *se-occ* is called by *se* to evaluate the occurrences of a module, two association lists are created binding input and state names to their respective values.

```
(mutual-recursion
 (defun se (fn ins st netlist)
   (if (primp fn)  ;;  fn is a primitive.
     (se-primp-apply fn ins st)
     ;; Extract the module definition and evaluate its outputs
     (let ((module (assoc-eq fn netlist)))
       (if (atom module)
           nil
         (let* ((md-ins      (md-ins   module))
                (md-outs     (md-outs  module))
                (md-st       (md-st    module))
                (md-occs     (md-occs  module))
                (wire-alist (pairlis$ md-ins ins))
                (st-alist   (pairlis$ md-st st)))
           (assoc-eq-values
            md-outs
            (se-occ md-occs wire-alist st-alist
                    (delete-to-eq fn netlist)))))))
 (defun se-occ (occs wire-alist st-alist netlist)
   (if (atom occs)
       wire-alist
     (let* ((occ       (car occs))
            (occ-name  (occ-name occ))
            (occ-outs  (occ-outs occ))
            (occ-fn    (occ-fn   occ))
            (occ-ins   (occ-ins  occ))
            (ins       (assoc-eq-values occ-ins  wire-alist))
            (st        (assoc-eq-value  occ-name st-alist))
            (new-vals  (se occ-fn ins st netlist))
            (new-alist (pairlis$ occ-outs new-vals))
            (new-wire-alist (append new-alist wire-alist)))
       (se-occ (cdr occs) new-wire-alist st-alist netlist)))))
```

Function $\textit{assoc-eq}(x, \textit{alist})$ returns the first key-value pair of the association list *alist* whose key is x, or `nil` if no such pair exists. Function *assoc-eq-value* returns only the value from the pair produced by *assoc-eq*. Function *assoc-eq-values* returns a list of values and is defined recursively in terms of *assoc-eq-value*. For example,

```
(assoc-eq 'B '((A . 5) (B . 3) (B . 4))) = '(B . 3),

(assoc-eq-value 'B '((A . 5) (B . 3) (B . 4))) = 3,

(assoc-eq-values '(B A) '((A . 5) (B . 3) (B . 4))) = '(3 5).
```

Function $\textit{pairlis\$}(x, y)$ zips together two lists x and y.

```
(pairlis$ '(B A C) '(1 3 2)) = '((B . 1) (A . 3) (C . 2))

(pairlis$ '(1 3 2) nil) = '((1) (3) (2))
```

Function $\textit{delete-to-eq}(fn, \textit{netlist})$ returns a subnetlist by walking through *netlist* until passing module fn and removing all modules that it passes (including fn). Primitive evaluation is simply function application as is shown in the definition of function *se-primp-apply*. For the DE system, we use the ACL2 constants `t` and `nil` to represent Boolean true and false, respectively. The new primitive `link-cntl` is added to the DE system for modeling the validity of data stored in communication links. We will discuss the use of this primitive in Chapter 3 when we present our self-timed circuit modeling approach.

```
(defun se-primp-apply (fn ins st)
  (case fn
    (b-and     (list (f-and (car ins) (cadr ins))))
    (b-buf     (list (f-buf (car ins))))
    (b-if      (list (f-if (car ins) (cadr ins) (caddr ins))))
    (b-not     (list (f-not (car ins))))
    (b-or      (list (f-or (car ins) (cadr ins))))
    (b-xor     (list (f-xor (car ins) (cadr ins))))
    (ff        (list (f-buf (car st))
                     (f-not (car st))))
    (latch     (list (f-if (car ins)
                           (cadr ins)
                           (car st))
                     (f-if (car ins)
                           (f-not (cadr ins))
                           (f-not (car st)))))
    (link-cntl (list (f-buf (car st))))
    (vdd       (list t))
    (vss       (list nil))
    (wire      (list (car ins)))
     ;; [ ... elided entries ... ]
    (otherwise nil)))
```

The primitive evaluation functions used in *se-primp-apply* are defined in four-valued logic. For instance, below are the definitions of f-*and*, f-*buf*, and *f-if*. The latter two are defined in terms of function *3v-fix* that coerces a non-Boolean value to an "unknown" value.

```
(defun f-and (a b)
  (if (or (equal a nil) (equal b nil))
      nil
    (if (and (equal a t) (equal b t))
        t
      *x*)))  ;; Constant *x* represents an "unknown" value.

(defun 3vp (x)
  (or (equal x t)
      (equal x nil)
      (equal x *x*)))

(defun 3v-fix (x)
  (if (3vp x) x *x*))

(defun f-buf (x)
  (3v-fix x))

(defun f-if (c a b)
  (if (equal c t)
      (3v-fix a)
    (if (equal c nil)
        (3v-fix b)
      *x*)))
```

2.2.2 State Evaluator

The state evaluator *de* and its mutually recursive peer function *de-occ* are defined in a manner similar to *se* and *se-occ*. However, *de* calls *de-occ* with an embedded call to *se-occ* so that the output values of all occurrences are first computed; *de-occ* then goes through each occurrence the second time and binds the occurrence (state) name to a (possibly empty) next state.

```
(mutual-recursion
 (defun de (fn ins st netlist)
   (if (primp fn)
       (de-primp-apply fn ins st)
     (let ((module (assoc-eq fn netlist)))
       (if (atom module)
           nil
         (let* ((md-ins      (md-ins   module))
                (md-st       (md-st    module))
                (md-occs     (md-occs  module))
                (wire-alist  (pairlis$ md-ins ins))
                (st-alist    (pairlis$ md-st st))
                (new-netlist (delete-to-eq fn netlist)))
           (assoc-eq-values
            md-st
            (de-occ md-occs
                    (se-occ md-occs wire-alist st-alist new-netlist)
                    st-alist
                    new-netlist)))))))

 (defun de-occ (occs wire-alist st-alist netlist)
   (if (atom occs)
       st-alist
     (let* ((occ      (car occs))
            (occ-name (occ-name occ))
            (occ-fn   (occ-fn   occ))
            (occ-ins  (occ-ins  occ))
            (ins      (assoc-eq-values occ-ins  wire-alist))
            (st       (assoc-eq-value  occ-name st-alist))
            (new-st-alist
             (update-alist occ-name
                           (de occ-fn ins st netlist)
                           st-alist)))
       (de-occ (cdr occs) wire-alist new-st-alist netlist)))))
```

Function *de-primp-apply* returns the next state for each primitive.

```
(defun de-primp-apply (fn ins st)
  (case fn
    ((ff latch) (list (f-if (car ins) (cadr ins) (car st))))
    (link-cntl  (list (f-sr (car ins) (cadr ins) (car st))))
    (otherwise  nil)))
```

The next state of a link-control primitive is specified by function *f-sr* that operates like set-reset latch updates.

```
(defun f-sr (s r st)
  (cond ((and (equal s nil) (equal r nil))
          (3v-fix st))
         ((and (equal s nil) (equal r t))  ;; Reset
          nil)
         ((and (equal s t) (equal r nil))  ;; Set
          t)
         (t *x*)))
```

Below is an example of evaluating the outputs and next state for the *flip-flop* module using the *se* and *de* functions, respectively. The semantics of `latch` is given as follows: when the *enable* signal is on, `latch` will propagate the input value to the output and update its internal state with the input value; otherwise it will report the current state to the output and its state remains unchanged. The single quotation marks require the evaluator to use the inputs as given, thus the expression `'(nil nil)` provides a list of two Boolean values: false, false. In this example, we instantiate en := `nil`, *bit-in* := `nil`, $L0$:= `'(t)`, and $L1$:= `'(nil)`. Note that we use the ACL2 (LISP-prefix) syntax to describe the definitions and formulas in this chapter. The later chapters may also use the infix syntax when convenient.

```
(se 'flip-flop '(nil nil) '((t) (nil)) *netlist*) = '(t nil)

(de 'flip-flop '(nil nil) '((t) (nil)) *netlist*) = '((t) (t))
```

To model the effect of inputs changing over time, the DE simulator is used repeatedly to evaluate a circuit netlist description whenever any primary input changes. We have defined the *de-n* function that returns an updated state after applying *de* *n* times. Through the repeated use of the DE simulator, Mealy machines are modeled advancing forward in time.

```
(defun de-n (fn inputs-seq st netlist n)
  (if (zp n)  ;; n is not a positive integer.
      st
    (de-n fn
          (cdr inputs-seq)
          (de fn (car inputs-seq) st netlist)
          netlist
          (- n 1))))
```

In synchronous circuits, storage elements update their values simultaneously at every global clock tick, where the clock rate is fixed. Hence the duration represented by two consecutive *de* evaluations of a synchronous module is fixed to model exactly one clock cycle. In self-timed circuits, however, storage elements update their values whenever their local communication conditions are met; and hence the duration represented by two consecutive *de* evaluations of a self-timed module varies.

2.3 Value and State Lemmas

The DE system provides a hierarchical approach to analyze DE circuit descriptions. In particular, we prove the following two lemmas in a hierarchical manner for every DE module: a *value lemma* characterizing a module's outputs and a *state lemma* characterizing a module's next state — and for other than the lowest-level modules, these two lemmas are proved by automatic application of the value and state lemmas of submodules, without referencing the internal details of the submodules. A purely combinational module requires only a value lemma. For example, here are the value lemmas for combinational modules *half-adder* and *full-adder*,

```
(defthm half-adder$value
  (implies (half-adder& netlist)
           (equal (se 'half-adder inputs st netlist)
                  (half-adder$outputs inputs st))))
```

```
(defthm full-adder$value
  (implies (full-adder& netlist)
           (equal (se 'full-adder inputs st netlist)
                  (full-adder$outputs inputs st))))
```

where the *outputs* functions *half-adder$outputs* and *full-adder$outputs* are defined as symbolic logical expressions characterizing the outputs of *half-adder* and *full-adder*, respectively.

```
(defun half-adder$outputs (inputs st)
  (declare (ignorable st))
  (let ((a (car inputs))
        (b (cadr inputs)))
    (list (f-xor a b)     ;; Sum
          (f-and a b))))  ;; Carry
```

```
(defun full-adder$outputs (inputs st)
  (declare (ignorable st))
  (let ((c (car inputs))
        (a (cadr inputs))
        (b (caddr inputs)))
    (list
     ;; Sum
     (car
      (half-adder$outputs
       (list (car (half-adder$outputs (list a b) ()))
             c)
       ()))
     ;; Carry
     (f-or
      (cadr (half-adder$outputs (list a b) ()))
      (cadr
       (half-adder$outputs
        (list (car (half-adder$outputs (list a b) ()))
              c)
        ())))))))
```

Note, we avoid exploring the internal structure of *half-adder* when proving the value lemma for *full-adder*; we apply the hierarchical reasoning by using the *half-adder*'s value lemma instead.

For each DE module, we introduce a predicate *module&*(*netlist*) that checks if that module and all of its referenced submodule(s) are defined in *netlist*. See the definitions of *half-adder&* and *full-adder&* below for examples.

```
(defun half-adder& (netlist)
  (equal (assoc 'half-adder netlist)
         (caddr *netlist*)))
```

```
(defun full-adder& (netlist)
  (let ((subnetlist (delete-to-eq 'full-adder netlist)))
    (and (equal (assoc 'full-adder netlist)
                (cadr *netlist*))
         (half-adder& subnetlist))))
```

For a module that contains state-holding elements, e.g. *flip-flop*, we prove both the value and state lemmas for that module.

```
(defthm flip-flop$value
  (implies (flip-flop& netlist)
           (equal (se 'flip-flop inputs st netlist)
                  (flip-flop$outputs inputs st))))

(defthm flip-flop$state
  (implies (flip-flop& netlist)
           (equal (de 'flip-flop inputs st netlist)
                  (flip-flop$step inputs st))))
```

The *outputs* and *step* functions *flip-flop$outputs* and *flip-flop$step* are defined as symbolic logical expressions characterizing the outputs and next state of *flip-flop*, respectively.

```
(defun flip-flop$outputs (inputs st)
  (let ((en      (car inputs))
        (bit-in  (cadr inputs))
        (L0.data (caar st))
        (L1.data (caadr st)))
    (list (f-if (f-not en)
                (f-if en bit-in L0.data)
                L1.data)
          (f-if (f-not en)
                (f-not (f-if en bit-in L0.data))
                (f-not L1.data)))))
```

```
(defun flip-flop$step (inputs st)
  (let ((en      (car inputs))
        (bit-in  (cadr inputs))
        (L0.data (caar st))
        (L1.data (caadr st)))
    (list  ;; L0's next state
          (list (f-if en bit-in L0.data))
           ;; L1's next state
          (list (f-if (f-not en)
                      (f-if en bit-in L0.data)
                      L1.data)))))
```

From the state lemma, we prove the following *multi-step state lemma* for *flip-flop* by induction,

```
(defthm flip-flop$de-n
  (implies (flip-flop& netlist)
           (equal (de-n 'flip-flop inputs-seq st netlist n)
                  (flip-flop$run inputs-seq st n))))
```

where *flip-flop$run* is defined recursively in terms of *flip-flop$step*.

```
(defun flip-flop$run (inputs-seq st n)
  (if (zp n)
      st
    (flip-flop$run (cdr inputs-seq)
                   (flip-flop$step (car inputs-seq) st)
                   (- n 1))))
```

Once the state and multi-step state lemmas are proved, we use only the *step* and *run* functions in reasoning about the module behavior. We reuse much of the DE machinery developed years ago, however avoiding the requirement to update state based on a single clock input. Recent use of this technology in clocked system verification is being made by Centaur Technology to verify properties of their contemporary x86-compatible microprocessor designs [63, 26]. We extend DE with a single link-control primitive that models the validity of data stored in a link. Thus, instead of advancing the values held by state-holding primitives with a clock, we allow the design to proceed at its own rate moderated by *oracle* values — extra input values modeling non-determinacy — that can cause any part of the logic to delay an arbitrary amount. Because circuit propagation speeds are unknown, we are obliged to consider all possible interleavings of circuit operations.

Of course, the consideration of all possible interleavings places an additional substantial burden on the verification methodology. To manage this complexity, we pursue our proofs in a hierarchical manner. For example, when we prove the correctness of a sequential circuit, we abstract away all of the internal delays and interleavings in the specification of sub-circuits; and we demonstrate that no matter when calculations occur internally, the external interface obeys its specification. Details of our verification methodology will be discussed in Chapter 4.

Part II

Approach

Chapter 3

Modeling

Self-timed circuits can be viewed as networks of communication channels (links) and handshake components (joints). Various circuit families (such as Micropipeline [69], GasP [70], Mousetrap [62], and Click [54]) have been proposed for designing self-timed circuits. Previous work on the design of these circuit families treated the links as merely communication channels of nothing but wires and put all the logic and storage in the joints. This made these circuit families much harder to exchange or combine because they provided various interfaces. Roncken et al. [58] proposed a different point of view that unified the existing families. In particular, their approach put more emphasis on links in which data are stored along with a validity signal, while joints are storage-free circuits that implement flow control and data operations. Under this point of view, the differences between the existing circuit families are seen only in the links, while the joints become identical. This allows different types of links to be interchangeable. Henceforth, we refer this point-of-view approach as the *link-joint model*. Because of its universality, we choose to follow this link-joint paradigm in modeling self-timed circuits with DE [1]. It is sufficiently

[1]This chapter is based on our previous publication [10]. The author of this dissertation did most of the technical work and wrote the first draft of that paper.

general to develop models of conventional and network processors. We show that this model has a clean formalization in the ACL2 logic and provides a protocol level that abstracts away timing constraints at the circuit level.

3.1 Link-Joint Model

Here we describe the link-joint model that we use to represent self-timed circuits with the DE language. Links are communication channels in which data are stored along with a validity signal. Joints are combinational circuits performing data operations and implementing flow control. Joints are the meeting points that coordinate links and share link data. A self-timed system can be viewed as a directed graph with links as edges and joints as nodes: the input or output of a link connects to exactly one joint each, so the link serves as a directed edge between those two joints. Figure 3.1 shows an example of a simple self-timed circuit using the link-joint model. This circuit consists of a joint associated with an input link L_0 and an output link L_1. A joint can have more than one input and/or output link(s) connected to it [58].

Links receive *fill* or *drain* commands from, and report their full/empty states and data to, their connected joints. A *full* link carries valid data, while an *empty* link holds data that are not yet or no longer valid. When an empty link receives a fill command at its input end, it changes its state to full. A full link will change to the empty state only if it receives a drain command at its output end. We model the full/empty state of a link using the `link-cntl`

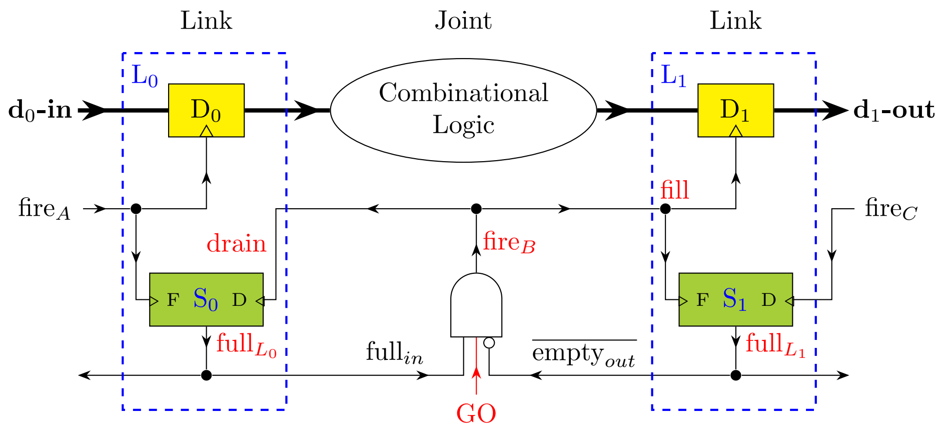


Figure 3.1: A diagram of a link-joint circuit is shown. It has two links, $L0$ and $L1$, and three joints A, B, and C. Only joint B is shown in its entirety. The upper and lower boxes in each link represent link data and link full/empty status, respectively.

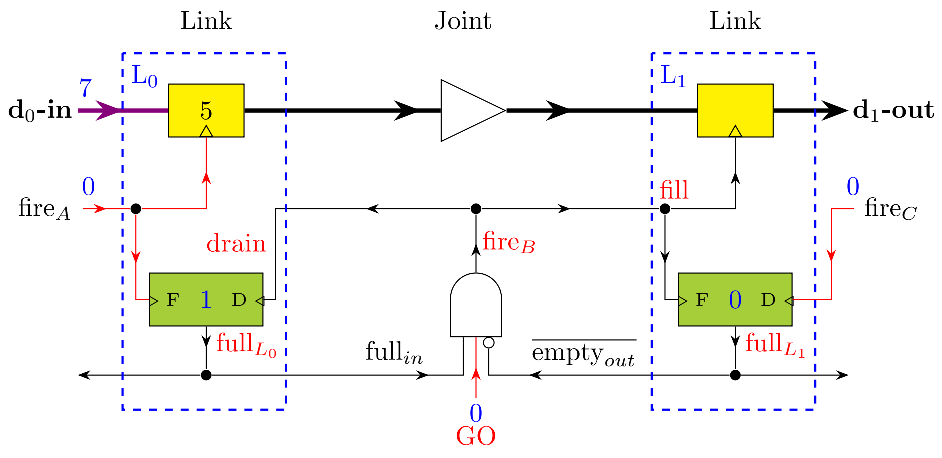
primitive that is specified in the DE primitive database. This primitive helps abstract away implementation details of link and joint control circuitry at the electrical circuit level. We refer the interested reader to Roncken et al.'s work [58] for examples of link and joint control circuitry.

Joints receive the full/empty states of their links and issue the fill and drain commands when their communication conditions are satisfied. Primitive joints are storage-free and they perform data computation and drain and fill storage links. The control logic of a joint is an AND function of the conditions necessary for it to act. A joint can have multiple such AND functions to guard different actions, which are usually mutually exclusive. To enable a joint action, all input and output links of that action must be full and empty, respectively, as illustrated by the AND gate in Figure 3.1. Enabled joints (that is, when at least one action is enabled) may fire in any order due to arbitrary delays in wires and gates. We model this non-deterministic circuit behavior by associating each joint with a so-called *go* signal as an extra input to the AND function in the control logic of that joint. In case a joint has multiple such AND functions, they may share the same *go* signal as long as at most one function can fire at a time. The value of the *go* signal will indicate whether the corresponding joint will fire when it is enabled. In our framework, when applying the `de` function that computes the next state of a self-timed circuit, only enabled joints with enabled *go* signals will fire. When a joint acts, the following three tasks will execute in parallel: [2]

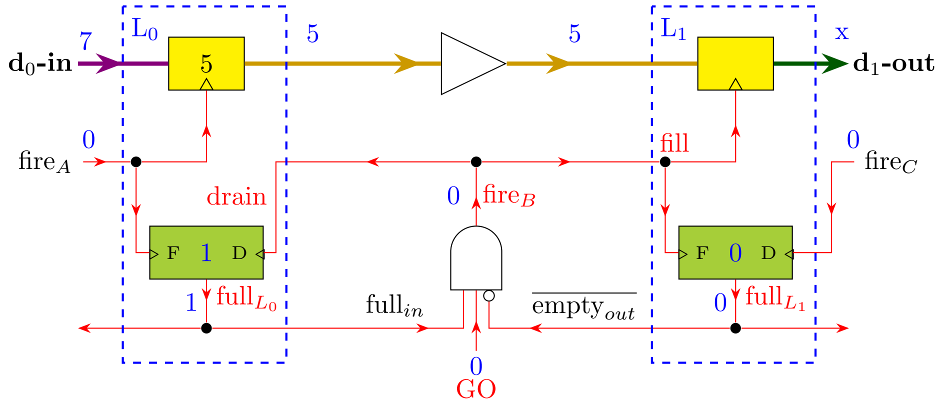
- using data from full input links, compute results to transfer to empty output links;
- *fill* (possibly a subset of) the empty output links, leaving them full; and
- *drain* (possibly a subset of) the full input links, leaving them empty.

Figures 3.2 and 3.3 illustrate how the self-timed module in Figure 3.1 progresses given concrete values of current inputs and state. The Combinational Logic oval, which represents the data computation of the joint in Figure 3.1, is a storage-free amplifier in these illustrations.

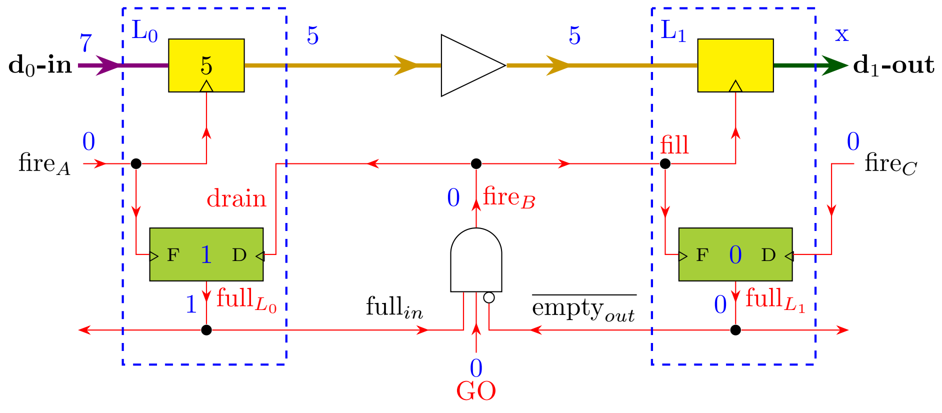
[2]Park et al. [51] used ARCtimer for generating and validating timing constraints in joints in order to guarantee that when a joint acts, its fire pulse is enabled for long enough for the three actions to execute properly. Our work assumes that we have a valid circuit that satisfies necessary circuit-level timing constraints, as might be guaranteed by ARCtimer.



Current inputs and state



First pass: wire evaluation



Second pass: state evaluation. The state remains unchanged.

Figure 3.2: Link-Joint circuit evaluation when GO = 0

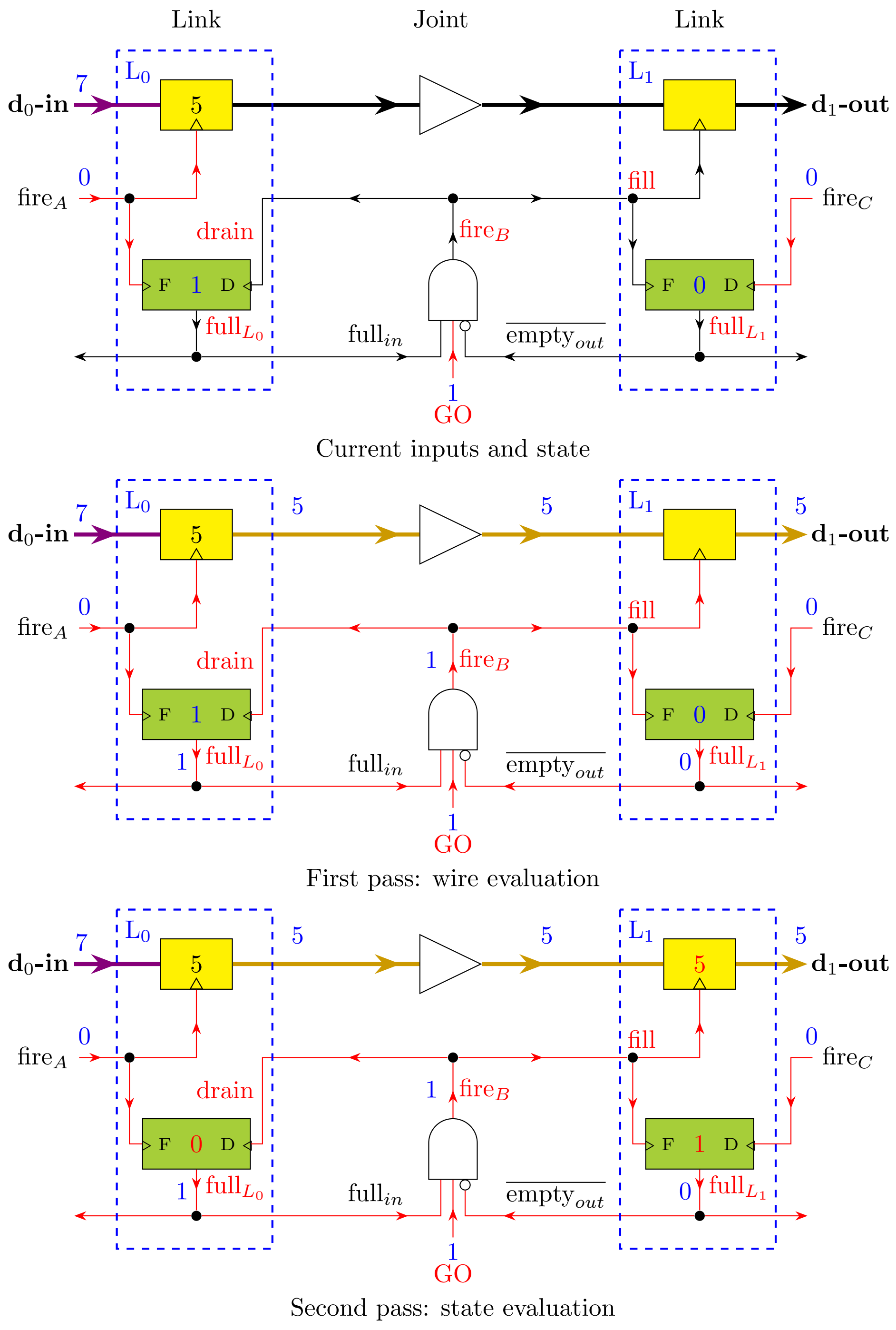


Figure 3.3: Link-Joint circuit evaluation when GO = 1

Before describing our DE description of a link, we first introduce some functions and macros that are used in our definitions of DE module generators. Macro *list∗* builds a list of objects from given elements and a tail. For example,

```
(list* '3 '5 '(4 2 3)) = '(3 5 4 2 3).
```

Function *si* returns a symbol that its name is combined with an index. Function $sis(s, i, n)$ returns a list of n symbols starting from index i. For example,

```
(si 'sym 5) = 'SYM_5
(sis 'x 2 3) = '(X_2 X_3 X_4).
```

The ACL2 macro, *module-generator*, is used to create DE modules with parameterized data sizes. The following form defines a circuit generator that can produce a link of any data size. Notice that there are two state-holding devices residing in a link: one stores the link's full/empty status and one stores the link data.

```
(module-generator
 link* (data-size)  ;; Generator's name and its parameter
 (si 'link data-size)                            ;; Module's name
 (list* 'fill 'drain (sis 'data-in 0 data-size))  ;; Inputs
 (list* 'status (sis 'data-out 0 data-size))      ;; Outputs
 '(s d)                                           ;; Internal states
 ;; Occurrences
 (list
  ;; Link status
  '(s (status) link-cntl (fill drain))
  ;; Link data
  (list 'd
        (sis 'data-out 0 data-size)
        (si 'latch-n data-size)  ;; Submodule reference
        (list* 'fill (sis 'data-in 0 data-size)))))
```

Module *latch-n* consists of a list of one-bit latches sharing the same enable signal.

```
(module-generator
 latch-n* (n)  ;; Generator's name and its parameter
 (si 'latch-n n)            ;; Module's name
 (list* 'EN (sis 'D 0 n))  ;; Inputs
 (sis 'Q 0 n)               ;; Outputs
 (sis 'G 0 n)               ;; States
 (latch-n-body 0 n))        ;; Occurrences
```

The occurrences of *latch-n* are generated by the recursive function *latch-n-body*.

```
(defun latch-n-body (m n)
  (if (zp n)
      nil
    (cons
     ;; Occurrence
     (list (si 'G m)                          ;; Occurrence's name
           (list (si 'Q m) (si 'Q~ m))  ;; Outputs
           'latch                             ;; Primitive reference
           (list 'EN (si 'D m)))         ;; Inputs
     (latch-n-body (+ m 1) (- n 1)))))
```

Our DE description of a self-timed module allows links and joints to appear in any order in the module's occurrence list, except that each link must be declared before its input and output joints so that when the module is being evaluated, the *se* function called in the first pass will extract the links' full/empty states and data and provide these values as inputs for the corresponding joints; the *de* function will make the second pass to update the link's full/empty states and data using the joints' output values calculated from the first pass. Below is the generator for the self-timed module shown in Figure 3.1, where D_0 and D_1 are n-bit latches, and the Combinational Logic oval is an n-bit storage-free amplifier (`v-buf` below).

```
(module-generator
 link-joint* (n)
 (si 'link-joint n)
 (list* 'fireA 'fireC
        (append (sis 'd0-in 0 n) '(go)))
 (list* 'l0-status 'l1-status (sis 'd1-out 0 n))
 '(l0 l1)
 ;; Occurrences
 (list
   ;; Link L0
  (list 'l0
        (list* 'l0-status (sis 'd0-out 0 n))
        (si 'link n)
        (list* 'fireA 'fireB (sis 'd0-in 0 n)))
    ;; Link L1
  (list 'l1
        (list* 'l1-status (sis 'd1-out 0 n))
        (si 'link n)
        (list* 'fireB 'fireC (sis 'd1-in 0 n)))

   ;; Joint B
  '(jb-cntl (fireB) joint-cntl (l0-status l1-status go))
  (list 'jb-op
        (sis 'd1-in 0 n)
        (si 'v-buf n)
        (sis 'd0-out 0 n))))
```

Module *joint-cntl* implements the control logic, which is an AND function, of a joint.

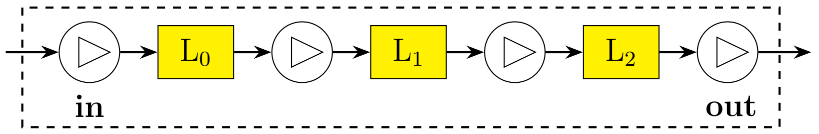
```
'(joint-cntl
  (full-in full-out go)
  (act)
  ()  ;; No internal states
  ((g0 (empty-out-) b-not (full-out))
   (g1 (ready) b-and (full-in empty-out-))
   (g2 (b-go) b-bool (go))  ;;  b-bool converts a non-Boolean to t.
   (joint-act (act) b-and (ready b-go))))
```

Module *v-buf* is composed of a list of one-bit buffers generated by function *v-buf-body*.

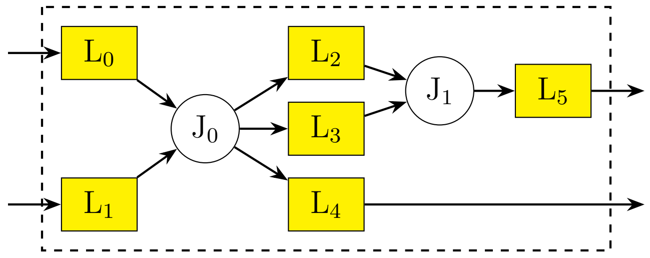
```
(defun v-buf-body (m n)
  (if (zp n)
      nil
    (cons (list (si 'g m)
                (list (si 'y m))
                'b-buf
                (list (si 'x m)))
          (v-buf-body (+ m 1) (- n 1)))))
```

```
(module-generator
 v-buf* (n)
 (si 'v-buf n)
 (sis 'x 0 n)
 (sis 'y 0 n)
 ()
 (v-buf-body 0 n))
```

As an example, we use the list `'(((t) ((nil) (nil))) ((nil) ((t) (nil))))` to represent the state of module *link-joint_2* where link $L0$ is full and its data value is `'(nil nil)`, and link $L1$ is empty and its data value is `'(t nil)`. Note that when a link is empty, its data are invalid.



(a) Complex joint: a queue of length three, $Q3$



(b) Complex link

Figure 3.4: Example of a complex joint and a complex link. The figure displays only the data flow; it omits both the flow control of the joints and the link states for the sake of simplicity. Circles represent joints, rectangles represent links. The primitive joints shown in (a) are buffers. Recall that primitive joints are storage-free.

3.2 Self-Timed Module Modeling

We construct self-timed modules using links and joints as described in the previous section. In principle, we can place a link or a joint at each module's input/output port, provided that the link-joint topology is preserved: links are connected via joints, and joints are connected via links. We generally model self-timed modules as *complex joints*, as illustrated in Part III. A module is a complex joint if only joints appear at its input and output ports (Figure 3.4a). A complex joint can replace any other joint in the system that has the same configuration of inputs and outputs, without violating the link-joint topology. We also demonstrate the potential advantage of *complex links* for significantly improving the verification time of a circuit; see Section 5.3 of Chapter 5. A module is a complex link if only links appear at its input and output ports (Figure 3.4b). It is typical that self-timed modules receive and send data via different links, using separate input and output communication signals.

3.2.1 Complex Joint

Our generator for the complex joint $Q3$ in Figure 3.4a, a FIFO queue of three links, is defined as follows.

```
(module-generator
 queue3* (data-size)
  ;; Module's name
 (si 'queue3 data-size)
  ;; Inputs
 (list* 'full-in 'empty-out-
        (append (sis 'data-in 0 data-size)
                (sis 'go 0 4)))  ;; Four GO signals
  ;; Outputs
 (list* 'in-act 'out-act
        (sis 'data-out 0 data-size))
  ;; Internal states
 '(l0 l1 l2)
  ;; Occurrences
 (list
   ;; LINKS
   ;; L0
  (list 'l0
        (list* 'l0-status (sis 'd0-out 0 data-size))
        (si 'link data-size)
        (list* 'in-act 'trans1-act (sis 'd0-in 0 data-size)))
```

```
;; L1
(list 'l1
      (list* 'l1-status (sis 'd1-out 0 data-size))
      (si 'link data-size)
      (list* 'trans1-act 'trans2-act (sis 'd1-in 0 data-size)))
;; L2
(list 'l2
      (list* 'l2-status (sis 'd2-out 0 data-size))
      (si 'link data-size)
      (list* 'trans2-act 'out-act (sis 'd2-in 0 data-size)))
;; JOINTS
;; In
(list 'in-cntl
      '(in-act)
      'joint-cntl
      (list 'full-in 'l0-status (si 'go 0)))
(list 'in-op
      (sis 'd0-in 0 data-size)
      (si 'v-buf data-size)
      (sis 'data-in 0 data-size))
;; Transfer data from L0 to L1
(list 'trans1-cntl
      '(trans1-act)
      'joint-cntl
      (list 'l0-status 'l1-status (si 'go 1)))
(list 'trans1-op
      (sis 'd1-in 0 data-size)
      (si 'v-buf data-size)
      (sis 'd0-out 0 data-size))
;; Transfer data from L1 to L2
(list 'trans2-cntl
      '(trans2-act)
      'joint-cntl
      (list 'l1-status 'l2-status (si 'go 2)))
```

```
  (list 'trans2-op
        (sis 'd2-in 0 data-size)
        (si 'v-buf data-size)
        (sis 'd1-out 0 data-size))
  ;; Out
  (list 'out-cntl
        '(out-act)
        'joint-cntl
        (list 'l2-status 'empty-out- (si 'go 3)))
  (list 'out-op
        (sis 'data-out 0 data-size)
        (si 'v-buf data-size)
        (sis 'd2-out 0 data-size)))
```

The following *module-generator* form illustrates our modeling of a self-timed module consisting of self-timed submodules. Specifically, we model a queue of seven links by connecting two three-link queues via a link (Figure 3.5). Note that we use a link to connect two three-link queues in order to maintain the link-joint topology.

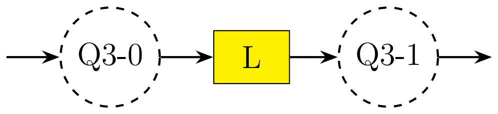


Figure 3.5: Data flow of $Q7$: a FIFO queue of seven links composed of two instances of $Q3$. Dashed circles represent complex joints.

```
(module-generator
 queue7* (data-size)
 (si 'queue7 data-size)
 (list* 'full-in 'empty-out-
        (append (sis 'data-in 0 data-size)
                (sis 'go 0 8)))  ;; Eight GO signals
 (list* 'in-act 'out-act
        (sis 'data-out 0 data-size))
 '(l q3-0 q3-1)
 (list
  ;; Link L
  (list 'l
        (list* 'l-status (sis 'd-out 0 data-size))
        (si 'link data-size)
        (list* 'q3-0-out-act 'q3-1-in-act (sis 'd-in 0 data-size)))
  ;; Complex joint Q3-0
  (list 'q3-0
        (list* 'in-act 'q3-0-out-act
               (sis 'd-in 0 data-size))
        (si 'queue3 data-size)
        (list* 'full-in 'l-status
               (append (sis 'data-in 0 data-size)
                       (sis 'go 0 4))))
  ;; Complex joint Q3-1
  (list 'q3-1
        (list* 'q3-1-in-act 'out-act
               (sis 'data-out 0 data-size))
        (si 'queue3 data-size)
        (list* 'l-status 'empty-out-
               (append (sis 'd-out 0 data-size)
                       (sis 'go 4 4))))))
```

3.2.2 Complex Link

In contrast to a complex joint, a complex link inputs the *act* signals from external joints and outputs the status of the links at its interface. For instance, below is our DE description of the complex link $Q4'$, a FIFO queue of length four. The data flow of $Q4'$ is shown in Figure 3.6.

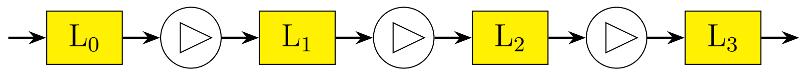


Figure 3.6: Data flow of $Q4'$. Note that links L_0 and L_3 are placed at the input and output ports, respectively. Thus $Q4'$ is a complex link.

```
(module-generator
 queue4-l* (data-size)
 (si 'queue4-l data-size)
 (list* 'in-act 'out-act
        (append (sis 'data-in 0 data-size)
                (sis 'go 0 3)))
 (list* 'ready-in- 'ready-out
        (sis 'data-out 0 data-size))
 '(l0 l1 l2 l3)
 (list
  ;; LINKS
  ;; L0
  (list 'l0
        (list* 'l0-status (sis 'd0-out 0 data-size))
        (si 'link data-size)
        (list* 'in-act 'trans1-act (sis 'data-in 0 data-size)))
  ;; L1
  (list 'l1
        (list* 'l1-status (sis 'd1-out 0 data-size))
        (si 'link data-size)
        (list* 'trans1-act 'trans2-act (sis 'd1-in 0 data-size)))
  ;; L2
  (list 'l2
        (list* 'l2-status (sis 'd2-out 0 data-size))
        (si 'link data-size)
        (list* 'trans2-act 'trans3-act (sis 'd2-in 0 data-size)))
```

```
;; L3
(list 'l3
      (list* 'l3-status (sis 'data-out 0 data-size))
      (si 'link data-size)
      (list* 'trans3-act 'out-act (sis 'd3-in 0 data-size)))
;; JOINTS
;; Transfer data from L0 to L1
(list 'trans1-cntl
      '(trans1-act)
      'joint-cntl
      (list 'l0-status 'l1-status (si 'go 0)))
(list 'trans1-op
      (sis 'd1-in 0 data-size)
      (si 'v-buf data-size)
      (sis 'd0-out 0 data-size))
;; Transfer data from L1 to L2
(list 'trans2-cntl
      '(trans2-act)
      'joint-cntl
      (list 'l1-status 'l2-status (si 'go 1)))
(list 'trans2-op
      (sis 'd2-in 0 data-size)
      (si 'v-buf data-size)
      (sis 'd1-out 0 data-size))
;; Transfer data from L2 to L3
(list 'trans3-cntl
      '(trans3-act)
      'joint-cntl
      (list 'l2-status 'l3-status (si 'go 2)))
(list 'trans3-op
      (sis 'd3-in 0 data-size)
      (si 'v-buf data-size)
      (sis 'd2-out 0 data-size))
;; Input port's status
'(in-status (ready-in-) wire (l0-status))
```

```
  ;; Output port's status
  '(out-status (ready-out) wire (l3-status))))
```

Two complex links can communicate with each other via joints, as illustrated in the data flow of the complex link $Q8'$ shown in Figure 3.7. As we see, two instances of complex link $Q4'$ are connected via a buffer joint. The DE description of $Q8'$ is given below.

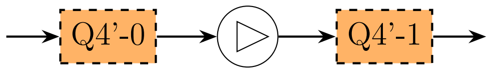


Figure 3.7: Data flow of $Q8'$. Dashed rectangles represent complex links.

```
(module-generator
 queue8-l* (data-size)
 (si 'queue8-l data-size)
 (list* 'in-act 'out-act
        (append (sis 'data-in 0 data-size)
                (sis 'go 0 7)))
 (list* 'ready-in- 'ready-out
        (sis 'data-out 0 data-size))
 '(q4-l0 q4-l1)
 (list
  ;; LINKS
  ;; Complex link Q4-L0
  (list 'q4-l0
        (list* 'ready-in- 'q4-l0-ready-out
               (sis 'q4-l0-data-out 0 data-size))
        (si 'queue4-l data-size)
        (list* 'in-act 'trans-act
               (append (sis 'data-in 0 data-size)
                       (sis 'go 1 3))))
;; Complex link Q4-L1
(list 'q4-l1
      (list* 'q4-l1-ready-in- 'ready-out
             (sis 'data-out 0 data-size))
      (si 'queue4-l data-size)
      (list* 'trans-act 'out-act
             (append (sis 'q4-l1-data-in 0 data-size)
                     (sis 'go 4 3))))

;; JOINT
;; Transfer data from Q4-L0 to Q4-L1
(list 'trans-cntl
      '(trans-act)
      'joint-cntl
      (list 'q4-l0-ready-out 'q4-l1-ready-in- (si 'go 0)))
(list 'trans-op
      (sis 'q4-l1-data-in 0 data-size)
      (si 'v-buf data-size)
      (sis 'q4-l0-data-out 0 data-size)))
```

Chapter 4

Verification

We develop a methodology for verifying the functional correctness of self-timed circuits (and systems) in terms of the relationships between their input and output sequences. We consider self-timed circuits that involve both data operations and flow control, while most existing work has concerned only flow control. Those efforts have mainly explored strategies for validating timing and communication properties, while our approach concerns functional properties of a self-timed system as a whole. Our approach supports scalability via *hierarchical reasoning* and *induction*. We would like to emphasize that although we aim to verify the multi-step input-output relationship for each self-timed module, our hierarchical reasoning is applied only at one-step updates. Once the one-step update on the output sequence is established, the multi-step input-output relationship can then be proved by induction. In order to specify the input-output relationship at one step, we introduce a set of *extraction functions* for each sequential module. An extraction function $extract(st)$ returns a sequence of values computed from valid data residing in state st. We use such a function to abstract away state transitions internal to its corresponding module. Applying *extract* to the *step* function, i.e. $extract(step(inputs, st))$, will compute the one-step update on the abstracted state given the current inputs *inputs* and current state *st*. Recall that *step* symbolically specifies the module's next state in one (*de*) step (see $flip\text{-}flop\$step$ in Chapter 2 for an example). To establish the multi-step input-output relationship by induction, we prove the following key lemma, which is called the *single-step-update* property,

$$extract(step(inputs, st)) = extracted\text{-}step(inputs, st) \tag{4.1}$$

where *extracted-step* is the specification for the one-step update on the abstracted state. An important property of *extracted-step* is that its definition avoids exploring the module's internal operations and their possible interleavings. To illustrate our definition of *extracted-step*, let us consider an example where a module has one input port and one output port, and let *in-act* and *out-act* denote the communication signals at the input and output ports respectively. The value of *in-act* indicates whether the module is currently accepting a new input data item; and the value of *out-act* indicates whether the module is currently reporting a valid output data item (`t` indicates *yes*, and `nil` indicates *no* for both signals). Our definition of *extracted-step* is then defined as follows,

$$extracted\text{-}step(inputs, st) :=$$
$$\begin{cases} extract(st), \ \textit{if } in\text{-}act = nil \land out\text{-}act = nil \\ [op(inputs.data)] \mathbin{++} extract(st), \ \textit{if } in\text{-}act = t \land out\text{-}act = nil \\ remove\text{-}last(extract(st)), \ \textit{if } in\text{-}act = nil \land out\text{-}act = t \\ [op(inputs.data)] \mathbin{++} remove\text{-}last(extract(st)), \ \textit{otherwise} \end{cases} \tag{4.2}$$

where

- $++$ is the concatenation operator,
- *remove-last*(l) returns list l except for its last element,
- *op* performs the module's functionality on the input data; in other words, *op* is the functional specification for the module.

Note that the parameter *inputs* we mention in the definitions and formulas presented in this dissertation consists of both input data and input control signals, including *go* signals for every joint. We write *inputs.data* to denote the data part of the inputs. As we see, *extracted-step* is defined in terms of *extract*; and while it depends on the values of *in-act* and *out-act*, it avoids considering the internal structure of the module. It is critical that *step* and *extract* are defined hierarchically so that the single-step-update property (4.1) can be proved hierarchically. A naive approach that expands the definitions of the *step* and *extract* functions of submodules when proving (4.1) may lead to a computational explosion. Our proofs cover all possible interleavings of circuit operations by considering all combinations of *go* signals' values. Much of our proof process is stylized. We automate the proof process by introducing proof idioms via macros and by developing lemma libraries.

Figure 4.1 displays our verification flow for a self-timed module. Our goal is to prove the gate-level netlist representation of a self-timed circuit correctly implements its functional specification. Our approach introduces two intermediate levels in connecting the low-level netlist to the high-level functional specification. Thus, we specify a self-timed module at four levels of ab-

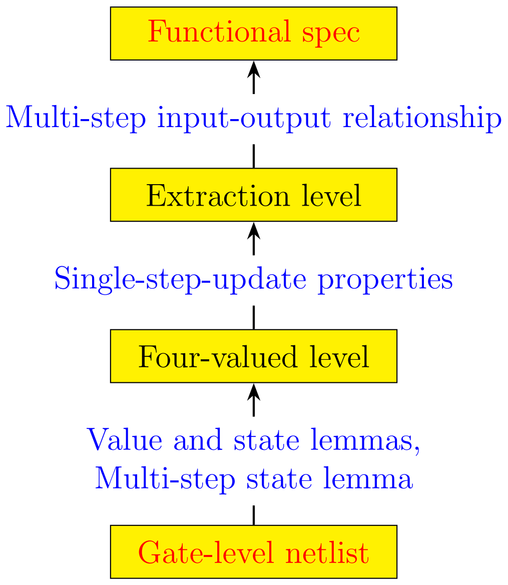


Figure 4.1: Verification flow

straction: netlist, four-valued, extraction, and functional. Functions *outputs*, *step*, and *run*, discussed in Section 2.3, serve as the specification at the four-valued level, while the extracted next-state function *extracted-step*, described above, serves as the specification for the extraction level. Figure 4.2 depicts the verification steps in our process of connecting these four levels together. The value and state lemmas and multi-step state lemma confirm the equivalence between the gate-level netlist and the four-valued level of a module. The single-step-update properties connect the four-valued level with the extraction level, while the multi-step input-output relationship links the four-valued level with the functional level through the extraction level. Our functional correctness proof combines the multi-step state lemma with the multi-step input-output relationship to justify that the netlist level implements the func-

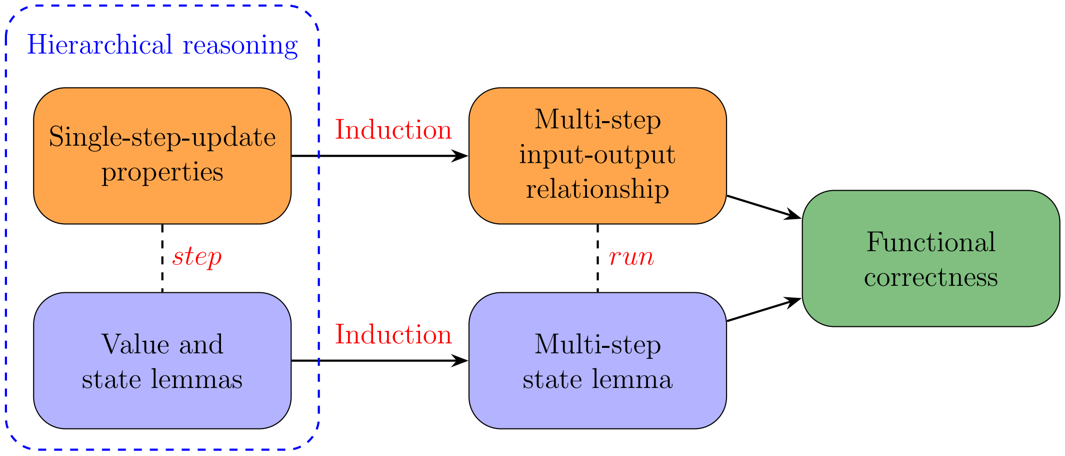


Figure 4.2: Verification steps

tional level. The following sections will describe each step in our verification procedure in further detail.

4.1 Value and State Lemmas

The first verification step in our procedure is to prove the correspondence between the netlist level and the four-valued level for a module. In particular, we prove a value lemma, a state lemma, and a multi-step state lemma for each sequential module as described in Chapter 2. A combinational-logic-only module lacks an internal state and hence requires only a value lemma. Since reasoning about module behavior through the low-level DE interpreter is complicated, we use those lemmas to "lift" a link-joint representation to the realm of pure ACL2 functions that abstract away the DE interpreter and netlist description. Recall that a value lemma characterizes a module's outputs, and a state lemma characterizes a module's next state. Value and state lemmas are proved in a hierarchical manner as presented in Chapter 2. We currently specify the *outputs* and *step* functions manually for each module. However, the definitions of these functions can be mechanically produced by the ACL2's *simplification process*. Possible future work might consider applying the ACL2 simplifier to generate these functions automatically.

For parameterized-data-width module synthesizers whose DE descriptions or occurrences are defined recursively, we apply induction in proving the value and state lemmas for those synthesizers. For instance, synthesizer *latch-n∗* defined in Chapter 3 contains the recursively defined occurrence generator *latch-n-body*; we prove, by induction, the following two lemmas for *latch-n-body* in support of proving the value and state lemmas for *latch-n∗*. These two lemmas characterize the wire and state evaluations for *latch-n-body* using the simulation functions *se-occ* and *de-occ*, respectively.

```
(defthm latch-n-body$value
  (implies (natp m)
           (equal (assoc-eq-values
                   (sis 'Q m n)
                   (se-occ (latch-n-body m n)
                           wire-alist st-alist netlist))
                  (fv-if (assoc-eq-value 'EN wire-alist)
                         (assoc-eq-values (sis 'D m n) wire-alist)
                         (strip-cars
                          (assoc-eq-values (sis 'G m n) st-alist))))))

(defthm latch-n-body$state
   (implies (and (natp m)
                 (subsetp (sis 'G m n) (strip-cars st-alist)))
            (equal (assoc-eq-values
                    (sis 'G m n)
                    (de-occ (latch-n-body m n)
                            wire-alist st-alist netlist))
                   (pairlis$
                    (fv-if (assoc-eq-value 'EN wire-alist)
                           (assoc-eq-values (sis 'D m n) wire-alist)
                           (strip-cars
                            (assoc-eq-values (sis 'G m n) st-alist)))
                    nil))))
```

Function *strip-cars*(x) collects up all first components of *cons* pairs in list x. For example,

```
      (strip-cars '((1 2 3) (4 5 6) (7 8 9))) = '(1 4 7)

      (strip-cars '((2) (1) (5))) = '(2 1 5)
```

Function *fv-if* performs if-then-else tests and returns a corresponding bit-vector. It is defined in terms of function *f-if*.

```
(defun fv-if (c a b)
  (if (atom a)
      nil
    (cons (f-if c (car a) (car b))
          (fv-if c (cdr a) (cdr b)))))
```

The value and state lemmas for a sequential self-timed module with a parameterized data size have the following form.

```
(defthm module$value
  (implies (and (module& netlist data-size)
                (module$input-format inputs data-size)
                (module$st-format st data-size))
           (equal (se (si 'module data-size)
                      inputs st netlist)
                  (module$outputs inputs st data-size))))
```

```
(defthm module$state
  (implies (and (module& netlist data-size)
                (module$input-format inputs data-size)
                (module$st-format st data-size))
           (equal (de (si 'module data-size)
                      inputs st netlist)
                  (module$step inputs st data-size))))
```

Predicate *module&* is the module recognizer (see *half-adder&* and *full-adder&* in Chapter 2 for examples). Predicate *module$input-format* imposes constraints on the inputs. For example, function *queue3$input-format* described below checks the following conditions on the argument *inputs* of $Q3$ (Figure 3.4a). Notice that the last condition requires the input data to be a bit (Boolean) vector when the *full-in* signal is high.

$$\begin{aligned}
&queue3\$input\text{-}format(inputs, data\text{-}size) := \\
&\quad booleanp(inputs.full\text{-}in) \wedge \\
&\quad booleanp(inputs.empty\text{-}out\text{-}) \wedge \\
&\quad len(inputs.data) = data\text{-}size \wedge \\
&\quad (\neg inputs.full\text{-}in \vee bvp(inputs.data))
\end{aligned}$$

Predicate *module$st-format*$(st, data\text{-}size)$ requires the data of each link in state *st* to be stored as a list of *data-size* singletons. This link data condition is checked via the predicate *link$st-format*$(link, data\text{-}size)$. For example, if the data of a link L are stored as the list `'((t) (x) (nil))`, this link would satisfy the condition *link$st-format*$(L, 3)$. Below is the definition of

$queue3\$st\text{-}format$.

$$\begin{aligned}&queue3\$st\text{-}format(st, data\text{-}size) :=\\&\quad link\$st\text{-}format(st.L0, data\text{-}size) \wedge\\&\quad link\$st\text{-}format(st.L1, data\text{-}size) \wedge\\&\quad link\$st\text{-}format(st.L2, data\text{-}size)\end{aligned}$$

4.2 Multi-Step State Lemma

The multi-step state lemma for a module characterizes the state update for that module after a parameterized-multi-step execution. This lemma is proved by induction after establishing the state lemma. We automate this verification step by introducing a macro called *simulate-lemma*. An application of this macro in one of the following forms will automatically generate the proof for the corresponding multi-step state lemma.

```
(simulate-lemma <name>)
(simulate-lemma <name> :clink t)  ;; Applied for complex links
```

As an example, the following multi-step state lemma for $Q3$ will be proved automatically by executing the term `(simulate-lemma queue3)`, given that the state lemma is already established.

```
(defthm queue3$de-n
  (implies (and (queue3& netlist data-size)
                (queue3$input-format-n inputs-seq data-size n)
                (queue3$st-format st data-size))
           (equal (de-n (si 'queue3 data-size)
                        inputs-seq st netlist n)
```

```
                 (queue3$run inputs-seq st data-size n))))
```

Functions *queue*3$*input-format-n* and *queue*3$*run* are recursively defined in terms of functions *queue*3$*input-format* and *queue*3$*step*, respectively.

```
(defun queue3$input-format-n (inputs-seq data-size n)
  (if (zp n)
      t
    (and (queue3$input-format (car inputs-seq) data-size)
         (queue3$input-format-n (cdr inputs-seq)
                                data-size
                                (- n 1)))))

(defun queue3$run (inputs-seq st data-size n)
  (if (zp n)
      st
    (queue3$run (cdr inputs-seq)
                (queue3$step (car inputs-seq) st data-size)
                data-size
                (- n 1))))
```

4.3 Single-Step-Update Properties

The key step in our verification effort is to map the four-valued level to the extraction level. The purpose of introducing the extraction level is to abstract away the internal structure of a module as well as its operational interleavings when reasoning about the module behavior. Our trick introduces a set of extraction functions for each sequential module as described earlier. Precisely, we define one extraction function for each sequential module with deterministic outputs; for circuits with non-deterministic outputs presented later in Chapter 7, we define multiple extraction functions for each of those systems. We currently define these functions manually, depending on module design. The key property of these functions is to abstract away state transitions internal to their corresponding modules; such a function will return the same sequence for any two input states of the corresponding module if one of those states can reach the other through merely internal transitions. We formalize this property through single-step-update properties of the form (4.1). After this step, we expand only the definition of the specification function at the extraction level, i.e. function *extracted-step*, when reasoning about the module behavior. Recall that *extracted-step* performs case analysis only on the communication signals at the module's interface, while the internal transitions do not affect its calculation (see definition (4.2) for an example). Thus, *extracted-step* is a clean specification function that allows us to avoid reasoning about the complex *step* function. Induction may be involved in this verification step when circuits contain feedback loops in their data flows (see Chapter 6). For example, this appears in the context of reasoning about the recursively defined algorithmic specification functions for those circuits. Our proof of (4.1) imposes certain conditions on the inputs and internal state. We specify those conditions manually for each module. For example, the following single-step-update property for $Q3$ requires the inputs and state to satisfy the predicates $queue3\$input\text{-}format$ and $queue3\$valid\text{-}st$, respectively.

$$\begin{aligned}
&queue3\$input\text{-}format(inputs, data\text{-}size) \land queue3\$valid\text{-}st(st, data\text{-}size)\\
&\Rightarrow queue3\$extract(queue3\$step(inputs, st, data\text{-}size)) =\\
&\quad queue3\$extracted\text{-}step(inputs, st, data\text{-}size)
\end{aligned} \tag{4.3}$$

Predicate $queue3\$valid\text{-}st$ is defined as follows,

$$\begin{aligned}
&queue3\$valid\text{-}st(st, data\text{-}size) :=\\
&\quad link\$valid\text{-}st(st.L0, data\text{-}size) \land\\
&\quad link\$valid\text{-}st(st.L1, data\text{-}size) \land\\
&\quad link\$valid\text{-}st(st.L2, data\text{-}size)
\end{aligned}$$

where the predicate $link\$valid\text{-}st(L, data\text{-}size)$ checks the following conditions for link L.

- the link data satisfies the predicate $link\$st\text{-}format(L, data\text{-}size)$;
- the link status is either full or empty; and
- when the link is full, its data value must be a bit vector of length $data\text{-}size$.

The next three chapters will describe the definitions of functions *extract* and *extracted-step* through specific circuit models. Here we present the extraction function for module $Q3$ that extracts data from links that are full at state st, while preserving the queue order,

$$queue3\$extract(st) := extract\text{-}valid\text{-}data([st.L0, st.L1, st.L2]) \tag{4.4}$$

where the projection function $extract\text{-}valid\text{-}data(l)$ returns the list generated by mapping over the links in l, collecting the data item of each full link and ignoring each empty link. For example, suppose links L_0 and L_2 are full, and link L_1 is empty in state st; then $extract\text{-}valid\text{-}data([st.L_0, st.L_1, st.L_2])$ will return $[d_0, d_2]$, where d_i is the data item of link L_i.

4.4 Multi-Step Input-Output Relationship

This step connects the four-valued level with the functional level via the extraction level. Our approach validates functional correctness of self-timed circuit models in terms of the functional relationships between their input and output sequences. More specifically, we verify the following multi-step input-output relationship for each sequential module,

$$\begin{aligned} & extract(run(inputs\text{-}seq, st, n)) \mathbin{++} out\text{-}seq = \\ & op\text{-}map(in\text{-}seq) \mathbin{++} extract(st) \end{aligned} \tag{4.5}$$

where

- $inputs\text{-}seq$ consists of both input data and input control signals, including go signals for every joint in the module;
- $in\text{-}seq$ is the sequence of input data extracted from $inputs\text{-}seq$ that are accepted by the module;

- *out-seq* is the sequence of data items that are reported to the output; and

- *op-map* performs the module's functionality over a data sequence.

The above property considers a general case where the initial and final states may contain valid data. When there are no valid data residing in these two states, we obtain the following corollary.

$$out\text{-}seq = op\text{-}map(in\text{-}seq)$$

We prove property (4.5) by induction, using single-step-update properties as supporting lemmas. Since these supporting lemmas impose some constraints on the module's state as discussed in the previous section, our induction proof for (4.5) requires that those constraints are invariant. For instance, the following property indicates that $queue3\$valid\text{-}st$ is an invariant.

$$queue3\$input\text{-}format(inputs, data\text{-}size) \wedge queue3\$valid\text{-}st(st, data\text{-}size)$$
$$\Rightarrow queue3\$valid\text{-}st(queue3\$step(inputs, st, data\text{-}size), data\text{-}size)$$

We automate this verification step through the following macro application, given the state invariants for the corresponding module are already certified.

```
(in-out-stream-lemma <module-name>)
```

4.5 Functional Correctness

The last step in our verification procedure combines the multi-step state lemma and multi-step input-output relationship to certify that the netlist level implements the functional level. Our main theorem, stated below, is generated automatically from an application of macro *in-out-stream-lemma* mentioned in the previous verification step. This theorem is a direct corollary of the multi-step input-output relationship that is stated in terms of the *de-n* function, while that relationship is formalized in terms of the *run* function as presented in equation (4.5),

$$\begin{aligned}&extract(de\text{-}n(module\text{-}name, inputs\text{-}seq, st, netlist, n)) \mathbin{++} out\text{-}seq\text{-}netlist = \\ &op\text{-}map(in\text{-}seq\text{-}netlist) \mathbin{++} extract(st)\end{aligned}$$

where *in-seq-netlist* and *out-seq-netlist* are specified at the netlist level, while *in-seq* and *out-seq* in property (4.5) are specified at the four-valued level. To be more specific, the former two are defined in terms of the *se* and *de* functions, while the latter two are defined in terms of the *outputs* and *step* functions. We are of course obliged to prove the equivalence between *in-seq-netlist* and *in-seq*, and between *out-seq-netlist* and *out-seq*. Our framework introduces the macro *seq-gen* that automatically generates the definitions of those sequences as well as their equivalence proofs. For example, consider the following application of *seq-gen*,

```
(seq-gen queue3 in in-act
         0  ;; 0 is the index of the in-act signal from the output list.
         (queue3$data-in inputs data-size))
```

where function *queue3$data-in* returns the data part of the inputs. Executing the term above will generate the following events in which the theorem *queue3$in-seq-lemma* confirms the equivalence between *queue3$in-seq-netlist* and *queue3$in-seq*.

```
(progn
  (defun queue3$in-seq (inputs-seq st data-size n)
    (if (zp n)
        nil
      (let* ((inputs (car inputs-seq))
             (in-act (queue3$in-act inputs st data-size))
             (data (queue3$data-in inputs data-size))
             (seq (queue3$in-seq (cdr inputs-seq)
                                 (queue3$step inputs st data-size)
                                 data-size
                                 (- n 1))))
        (if (equal in-act t)
            (append seq (list data))
          seq))))

  (defun queue3$in-seq-netlist (inputs-seq st netlist data-size n)
    (if (zp n)
        nil
      (let* ((inputs (car inputs-seq))
             (outputs (se (si 'queue3 data-size)
                          inputs st netlist))
             ;; Extract in-act at index 0 of outputs
             (in-act (nth 0 outputs))
             (data (queue3$data-in inputs data-size))
             (seq (queue3$in-seq-netlist (cdr inputs-seq)
                                        (de (si 'queue3 data-size)
                                            inputs st netlist)
                                        netlist
                                        data-size
                                        (- n 1))))
        (if (equal in-act t)
            (append seq (list data))
          seq))))

  (defthm queue3$in-seq-lemma
    (implies
     (and (queue3& netlist data-size)
          (queue3$input-format-n inputs-seq data-size n)
          (queue3$st-format st data-size))
     (equal (queue3$in-seq-netlist inputs-seq st netlist data-size n)
            (queue3$in-seq inputs-seq st data-size n)))))
```

Similarly, the following application of *seq-gen* produces the corresponding events for $Q3$'s output sequence,

```
(seq-gen queue3 out out-act
         1  ;; 1 is the index of the out-act signal from the output list.
         (queue3$data-out st)
         :netlist-data (nthcdr 2 outputs))
```

where $queue3\$data\text{-}out(st)$ and $nthcdr(2, outputs)$ return the output data computed at the four-valued and netlist levels, respectively. Here are the events generated by executing the term above.

```
(progn
  (defun queue3$out-seq (inputs-seq st data-size n)
    (if (zp n)
        nil
      (let* ((inputs (car inputs-seq))
             (out-act (queue3$out-act inputs st data-size))
```

```
           (data (queue3$data-out st))
           (seq (queue3$out-seq (cdr inputs-seq)
                                (queue3$step inputs st data-size)
                                data-size
                                (- n 1))))
      (if (equal out-act t)
          (append seq (list data))
        seq))))

(defun queue3$out-seq-netlist (inputs-seq st netlist data-size n)
  (if (zp n)
      nil
    (let* ((inputs (car inputs-seq))
           (outputs (se (si 'queue3 data-size)
                        inputs st netlist))
           ;; Extract out-act at index 1 of outputs
           (out-act (nth 1 outputs))
           (data (nthcdr 2 outputs))
           (seq (queue3$out-seq-netlist (cdr inputs-seq)
                                        (de (si 'queue3 data-size)
                                            inputs st netlist)
                                        netlist
                                        data-size
                                        (- n 1))))
      (if (equal out-act t)
          (append seq (list data))
        seq))))

(defthm queue3$out-seq-lemma
  (implies
   (and (queue3& netlist data-size)
        (queue3$input-format-n inputs-seq data-size n)
        (queue3$st-format st data-size))
   (equal (queue3$out-seq-netlist inputs-seq st netlist data-size n)
          (queue3$out-seq inputs-seq st data-size n)))))
```

The next three chapters will demonstrate the applicability of our verification framework through a sequence of increasingly complex self-timed circuit models, including data-loop-free circuits (Chapter 5), iterative circuits (Chapter 6), and circuits performing non-deterministically arbitrated merges (Chapter 7). The source code for these case studies is available at `https://github.com/acl2/acl2/tree/master/books/projects/async`. We will discuss only the single-step-update properties and multi-step input-output relationship in our case studies. The other verification steps are totally routine, so we will skip them to keep our presentation concise.

Part III

Case Studies

Chapter 5

Data-Loop-Free Circuits

This chapter demonstrates our verification framework through case studies of several data-loop-free self-timed circuits [1]. We handle parameterized families of implementations: we mechanically generate circuit descriptions with data buses of arbitrary width, and we verify such parameterized HDL circuit generators. We describe our hierarchical verification process over data-loop-free circuits. These circuits are simple, yet illustrate our verification process entirely. We also show the advantage of using complex links in substantially reducing the verification time of a self-timed module. In the next chapter, we will show that our framework is also applicable to iterative circuits, i.e., circuits with feedback loops in their data flows.

5.1 Example 1: A FIFO Circuit Model

Our first example is the FIFO queue model of length three, $Q3$, as mentioned in Chapters 3 and 4. This circuit model contains three links and four joints that pass data items from its input to its output (Figure 3.4a). Let

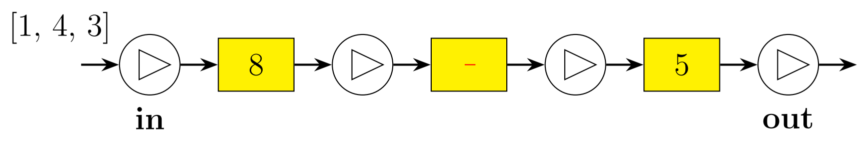
[1]This chapter is based on our previous publication [10]. The author of this dissertation did most of the technical work and wrote the first draft of that paper.

in-act denote the *fire* signal from the AND gate (not illustrated) in the control logic of joint **in**. Module $Q3$ accepts a new data item each time the *in-act* signal fires. We define *in-seq*, the *accepted input sequence*, as the sequence of data items that have passed joint **in**. Similarly, let *out-act* denote the *fire* signal from the AND gate (not illustrated) in the control logic of joint **out**. We define *out-seq*, the *valid output sequence*, as the sequence of data items that have passed through joint **out** when *out-act* fires. Note, *in-act* cannot fire when L_0 is full, and *out-act* cannot fire when L_2 is empty.

The extraction function $queue3\$extract$ simply extracts valid data from $Q3$ as defined in (4.4). We introduce the extracted next-state function $queue3\$extracted\text{-}step$, an instance of function *extracted-step* defined in (4.2), to extract valid data at the next state of $Q3$. This function is defined in terms of $queue3\$extract$ as shown below,

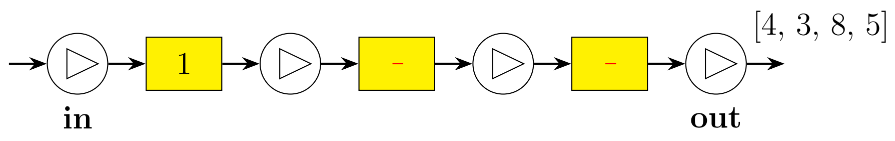
$$queue3\$extracted\text{-}step(inputs, st, data\text{-}size) :=$$
$$\begin{cases} queue3\$extract(st), & \textit{if } in\text{-}act = nil \land out\text{-}act = nil \\ [inputs.data] \mathbin{++} queue3\$extract(st), & \textit{if } in\text{-}act = t \land out\text{-}act = nil \\ remove\text{-}last(queue3\$extract(st)), & \textit{if } in\text{-}act = nil \land out\text{-}act = t \\ [inputs.data] \mathbin{++} remove\text{-}last(queue3\$extract(st)), & \textit{otherwise} \end{cases}$$

where signals *inputs.data*, *in-act*, and *out-act* are produced by functions that are defined in terms of parameter *data-size*. Note that function $queue3\$extracted\text{-}step$ avoids considering internal operations of $Q3$; it considers only the values of $queue3\$extract$ and the *in-act* and *out-act* signals at $Q3$'s input and output ports respectively, thus reducing the complexity of extracting valid data from $Q3$'s next state to four cases.



$[1,4,3] \mathbin{++} [8,5]$

(a)



$[1] \mathbin{++} [4,3,8,5]$

(b)

$$[1] \mathbin{++} [4,3,8,5] = [1,4,3] \mathbin{++} [8,5]$$

Figure 5.1: An example illustrating the multi-step input-output relationship for $Q3$

We verify the functional correctness of $Q3$ by proving that after an n-step execution from its initial state, *the concatenation of the valid data in the final state and the output sequence is identical to the concatenation of the input sequence and the valid data in the initial state.* Let us represent invalid data as _. Given that the input sequence consumed by $Q3$ is $[1,4,3]$, and the initial content of $Q3$ was $[8,\_,5]$ (Figure 5.1a), $Q3$ will deliver the valid data from its initial state first then followed by the input sequence. If $Q3$ delivers all data from the initial state and the input sequence, the output sequence must be $[1,4,3,8,5]$. Specifically, the output sequence is the concatenation of the input sequence and the valid data of the initial state. However, suppose at some time the content of $Q3$'s state is $[1,\_,\_]$, then the output

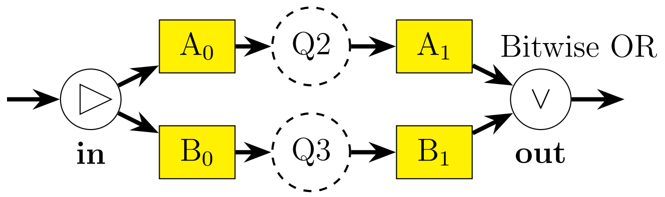


Figure 5.2: Data flow of module C: a circuit that performs bitwise OR in joint **out**. Dashed circles represent complex joints, $Q2$ and $Q3$.

sequence must be $[4, 3, 8, 5]$ (Figure 5.1b). In this case, the following equation states the relationship between the input and output sequences of $Q3$: $[1] \mathbin{++} [4, 3, 8, 5] = [1, 4, 3] \mathbin{++} [8, 5]$. This relation is formalized as follows.

$$\begin{aligned}&\mathit{queue3\$extract}(\mathit{queue3\$run}(\mathit{inputs\text{-}seq}, \mathit{st}, \mathit{data\text{-}size}, n)) \mathbin{++} \mathit{out\text{-}seq} = \\ &\mathit{in\text{-}seq} \mathbin{++} \mathit{queue3\$extract}(\mathit{st})\end{aligned} \tag{5.1}$$

It trivially follows that $\mathit{out\text{-}seq} = \mathit{in\text{-}seq}$ when the initial and final states contain no valid data. Recall that *in-seq* and *out-seq* contain only data and are devoid of control information. Moreover, *in-seq* is extracted from the inputs in *inputs-seq* that are accepted by $Q3$, specifically when: L_0 is empty, the link providing the input data is full, and the corresponding *go* signal is active. Our ACL2 proof of (5.1) uses induction and the single-step-update property (4.3) as a key supporting lemma.

5.2 Example 2: Hierarchical Reasoning

The next example illustrates how we apply hierarchical reasoning to verify a circuit C that contains $Q2$ and $Q3$ as submodules (Figure 5.2), where $Q2$ is a FIFO queue of two links. In terms of input-output relationship, C

simply performs the bitwise OR operation on paired input data. Joint **in** splits the input data items into two, equal-sized fields, a and b. The operation of C over a data sequence is specified as follows,

```
$c\$op\text{-}map(seq) :=$
  **if** $(seq = NULL)$
  **then** $[]$  // an empty list
  **else**
    **let** $in := first(seq)$
    **return** $[v\text{-}or(in.a, in.b)]$ ++ $c\$op\text{-}map(rest(seq))$
```

where $v\text{-}or(a, b)$ performs the bitwise OR operation over fields a and b. We then define an extraction function that computes the future output sequence from the current state, st, as described below,

```
$c\$extract(st)$
  :=
**let** $data\text{-}seq_0 := extract\text{-}valid\text{-}data([st.A_0])$ ++ $queue2\$extract(st.Q2)$ ++
                $extract\text{-}valid\text{-}data([st.A_1])$,
    $data\text{-}seq_1 := extract\text{-}valid\text{-}data([st.B_0])$ ++ $queue3\$extract(st.Q3)$ ++
                $extract\text{-}valid\text{-}data([st.B_1])$
**return** $c\$op\text{-}map(data\text{-}seq_0 \otimes data\text{-}seq_1)$
```

where $\otimes$ is the *pairlis$* operation that zips together two lists, e.g., $[1, 3, 5] \otimes [2, 4, 6] = [[1, 2], [3, 4], [5, 6]]$. A key invariant for the state of C is that the number of valid data of queue ($A_0 \rightarrow Q2 \rightarrow A_1$) equals the number of valid data of queue ($B_0 \rightarrow Q3 \rightarrow B_1$). We formalize this invariant as follows.

$$
\begin{aligned}
&c\$inv(st)\\
&\quad :=\\
&\Big(len\big(extract\text{-}valid\text{-}data([st.A_0, st.A_1]) \mathbin{++} queue2\$extract(st.Q2)\big)\\
&\quad =\\
&\;len\big(extract\text{-}valid\text{-}data([st.B_0, st.B_1]) \mathbin{++} queue3\$extract(st.Q3)\big)\Big)
\end{aligned}
$$

Then the following property of C holds, assuming $c\$inv(st)$.

$$
\begin{aligned}
&c\$extract(c\$run(inputs\text{-}seq, st, data\text{-}size, n)) \mathbin{++} out\text{-}seq =\\
&c\$op\text{-}map(in\text{-}seq) \mathbin{++} c\$extract(st)
\end{aligned}
\tag{5.2}
$$

This functional property states that the output sequence is computed by performing the bitwise OR on the members of the input sequence. Our ACL2 proof of (5.2) follows the same proof strategy as we used previously in (5.1); we use induction with the single-step-update property (described below) as a supporting lemma,

$$
\begin{aligned}
&c\$extract(c\$step(inputs, st, data\text{-}size)) =\\
&c\$extracted\text{-}step(inputs, st, data\text{-}size)
\end{aligned}
\tag{5.3}
$$

where the definition of $c\$extracted\text{-}step$ is given as follows.

$$c\$extracted\text{-}step(inputs, st, data\text{-}size) :=$$
$$\begin{cases} c\$extract(st), & \textit{if } in\text{-}act = nil \land out\text{-}act = nil \\ [v\text{-}or(inputs.a, inputs.b)] \mathbin{++} c\$extract(st), & \textit{if } in\text{-}act = t \land out\text{-}act = nil \\ remove\text{-}last(c\$extract(st)), & \textit{if } in\text{-}act = nil \land out\text{-}act = t \\ [v\text{-}or(inputs.a, inputs.b)] \mathbin{++} remove\text{-}last(c\$extract(st)), & \textit{otherwise} \end{cases}$$

Property (5.3) holds when $c\$inv(st)$ holds. In this case, in order to prove (5.2) by using induction and (5.3), we need to prove that $c\$inv(st')$ holds for any possible next state st' that can be reached from the current state st, given that $c\$inv(st)$ holds. In other words, we must prove that $c\$inv$ is indeed an invariant.

Let us emphasize an important fact: our proof of the invariant $c\$inv$ and (5.3) avoids considering the internal details of $Q2$ and $Q3$. Instead, we use their single-step-update properties to prove $c\$inv$ and (5.3). In other words, we employ a hierarchical strategy to prove single-step-update properties for a self-timed module.

5.3 Example 3: Complex Links

In this example, we demonstrate the benefit of specifying a module as a complex link. This approach aids our proof of both a self-timed system's invariant as well as its single-step-update property. First let us introduce a "wig-wag" circuit WW, illustrated in Figure 5.3. The **branch** joint in WW accepts input data and places them alternately into links L_0 and L_1. The

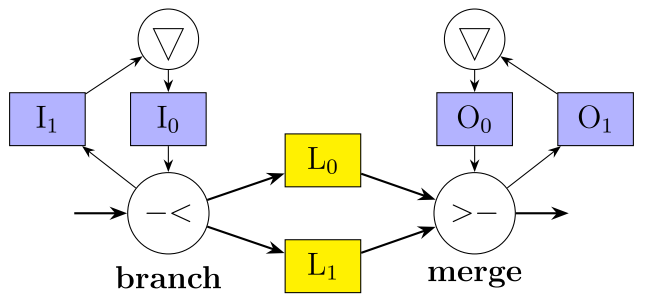


Figure 5.3: Data flow of a wig-wag circuit, WW

merge joint takes data alternately from links L_0 and L_1 and delivers them as outputs. Links I_0 and O_0 hold binary values that retain the alternation state. Joints **branch** and **merge** have two mutually exclusive actions each. When the **branch** joint fires, it fills either link L_0 or link L_1 according to the input alternation state (0 or 1, respectively). Likewise, the **merge** joint will drain either L_0 or L_1 when it fires, according to the output alternation state. In addition, joint **branch** delivers the negated value of I_0 to I_1. This new alternation value returns to joint **branch** via I_0. The same mechanism applies in the **merge** joint. An interesting property of WW is that its functionality is equivalent to $Q2$, but potentially has higher performance because of its lower latency:

- WW can input data into either L_0 or L_1, while $Q2$ can input data only into L_0;
- WW can output data from either L_0 or L_1, while $Q2$ can output data only from L_1.

Given that the full/empty states of I_0 and I_1 must differ, we define the select signal of the **branch** joint as follows.

```
$branch\text{-}select(st) :=$
  **if** $full(st.I_0.s)$
  **then** $st.I_0.d$
  **else** $st.I_1.d$
```

Similarly, the definition of the select signal of the **merge** joint is given below.

```
$merge\text{-}select(st) :=$
  **if** $full(st.O_0.s)$
  **then** $st.O_0.d$
  **else** $st.O_1.d$
```

We verify the correctness of WW by proving the same property we used to specify $Q2$: the concatenation of valid internal data in the final state with the output sequence is identical to the concatenation of the input sequence with the initial valid data.

$$\begin{aligned} &ww\$extract(ww\$run(inputs\text{-}seq, st, data\text{-}size, n)) \mathbin{++} out\text{-}seq = \\ &in\text{-}seq \mathbin{++} ww\$extract(st) \end{aligned} \tag{5.4}$$

The extraction function for WW is defined as follows.

$$
\begin{aligned}
& ww\$extract(st) := \\
& \quad \textbf{if } (merge\text{-}select(st) = 0) \\
& \quad \textbf{then } data([st.L_1, st.L_0]) \\
& \quad \textbf{else } data([st.L_0, st.L_1])
\end{aligned}
$$

Our proof of (5.4) requires the initial state st to satisfy the following condition.

$$
\begin{aligned}
& ww\$inv(st) := \\
& \quad st.I_0.s \neq st.I_1.s \wedge \\
& \quad st.O_0.s \neq st.O_1.s \wedge \\
& \quad \Big(\big(st.L_0.s = st.L_1.s \wedge \\
& \quad\ \ branch\text{-}select(st) = merge\text{-}select(st)\big) \vee \\
& \quad\ \ \big(full(st.L_0.s) \wedge empty(st.L_1.s) \wedge \\
& \quad\ \ branch\text{-}select(st) = 1 \wedge merge\text{-}select(st) = 0\big) \vee \\
& \quad\ \ \big(empty(st.L_0.s) \wedge full(st.L_1.s) \wedge \\
& \quad\ \ branch\text{-}select(st) = 0 \wedge merge\text{-}select(st) = 1\big)\Big)
\end{aligned}
$$

Assuming the current state st satisfies the $ww\$inv$ condition, we prove the single-step-update property of WW as an auxiliary lemma for proving (5.4). In addition, in order to prove (5.4) by induction, we also prove that $ww\$inv$ is an invariant.

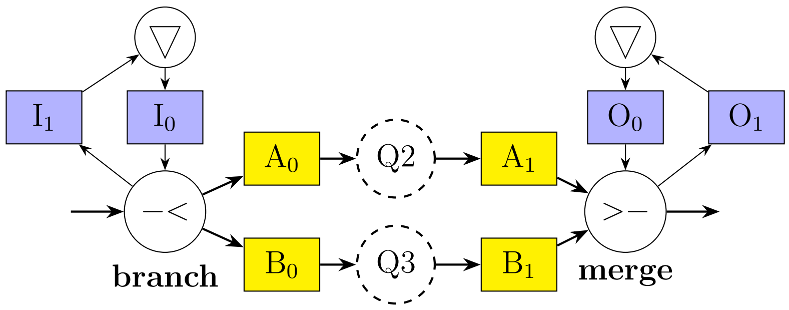


Figure 5.4: Data flow of a round-robin circuit, $RR1$

Now let us consider an extended version of WW in which links L_0 and L_1 are replaced by the queues ($A_0 \rightarrow Q2 \rightarrow A_1$) and ($B_0 \rightarrow Q3 \rightarrow B_1$), as shown in (Figure 5.4). We call this circuit round-robin $RR1$. Recall that $Q2$ and $Q3$ are complex joints. The verification time of $RR1$ is about 32.5 minutes, while verifying WW takes only 9 seconds on a contemporary laptop. Why? There are many case splits required to prove the invariant as well as the single-step-update property for $RR1$. It takes 6.3 minutes to prove the invariant and nearly 25.6 minutes to prove the single-step-update property. Most case splits arise from considering the full/empty states of four links A_0, A_1, B_0, and B_1 along with the case splits in the correlation between the numbers of valid data items in $Q2$ and $Q3$.

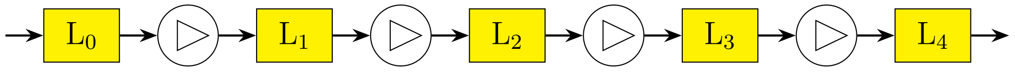


Figure 5.5: Data flow of $Q5'$

To reduce the number of case splits in this problem, we abstract two queues ($A_0 \rightarrow Q2 \rightarrow A_1$) and ($B_0 \rightarrow Q3 \rightarrow B_1$) as two complex links. We call these two links $Q4'$ (Figure 3.6) and $Q5'$ (Figure 5.5) respectively. We

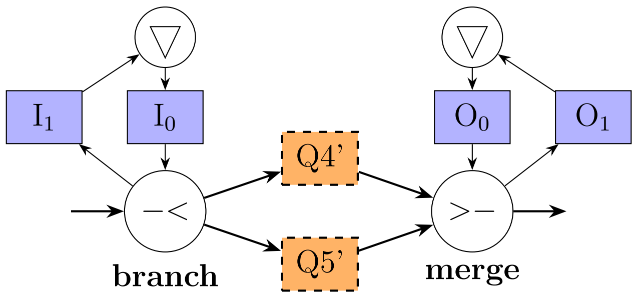


Figure 5.6: Data flow of a round-robin circuit, $RR2$. Dashed rectangles represent complex links.

follow the same procedure of formalizing the relationship between input and output sequences for $Q4'$ and $Q5'$, as described in previous examples. The verification times of $Q4'$ and $Q5'$ are 4 and 8 seconds respectively. By using $Q4'$ and $Q5'$ as submodules we construct an alternative round-robin circuit, which we call $RR2$, as depicted in Figure 5.6. The verification time of $RR2$ is a mere 22 seconds, which shows the benefit of using a hierarchical verification approach with complex links.

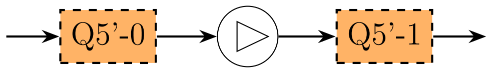


Figure 5.7: Data flow of $Q10'$

To demonstrate the scalability of our approach, we specify and verify a larger round-robin circuit model, which we call $RR3$. $RR3$ replaces $Q4'$ and $Q5'$ with longer queues $Q8'$ (Figure 3.7) and $Q10'$ (Figure 5.7), respectively. $Q8'$ is a complex link representing a queue of eight links and is constructed by concatenating two instances of $Q4'$ via a buffer joint. Similarly, $Q10'$ is constructed by concatenating two instances of $Q5'$. Our proof of $RR3$ is exactly the same as that of $RR2$ and its verification time is also 22 seconds. The verification times for $Q8'$ and $Q10'$ are 3 seconds each. We claim that our compositional proof for a round-robin circuit is independent of the queue size. To further support this claim, we apply our framework to the round-robin circuit model $RR4$ that substitutes $Q8'$ and $Q10'$ in $RR3$ with two 40-link queues $Q40'$s. The complex link $Q40'$ is modeled from two instances of $Q20'$, which in turn is a complex link constructed from two instances of $Q10'$. Our proof for $RR4$ just takes 25 seconds, which is about the same as $RR2$ and $RR3$. Table 5.1 reports the verification times of the self-timed circuits discussed in this chapter, along with the number of *go* signals in these circuits and the number of *go* signals actually affecting our hierarchical reasoning method.

Table 5.1: Proof times for the self-timed circuits discussed in this chapter. All experiments used an Apple laptop with a 2.9 GHz Intel Core i7 processor, 4MB L3 cache, and 8GB memory. The proof time for a module excludes proof times for its submodules.

Circuit	**Proof time**	**# *go* signals**	**# *go* signals affecting reasoning**
$Q2$	2s	3	3
$Q3$	3s	4	4
$Q4'$	4s	3	3
$Q5'$	8s	4	4
$Q8'$	3s	7	1
$Q10'$	3s	9	1
$Q20'$	3s	19	1
$Q40'$	3s	39	1
C	11s	9	6
WW	9s	4	4
$RR1$	32.5m	11	8
$RR2$	22s	11	4
$RR3$	22s	20	4
$RR4$	25s	82	4

Chapter 6

Iterative Circuits

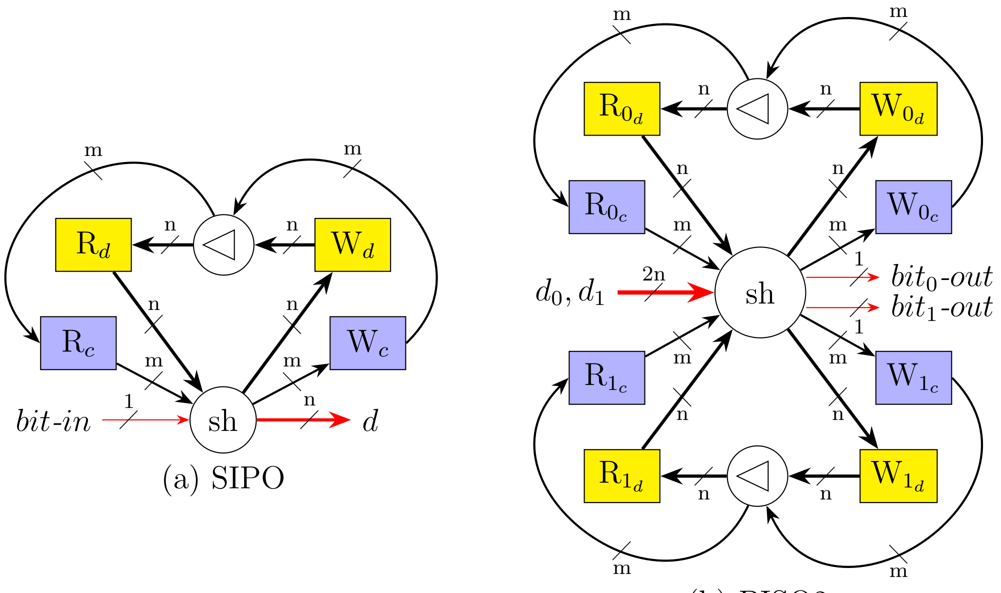
In the previous chapter we described our verification process over data-loop-free circuits. We now further demonstrate the robustness and scalability of our method by showing that it can be applied to iterative circuits as well. The difference between verifying data-loop-free and iterative circuits using our approach falls into proving their single-step-update properties, which involves reasoning about their functional specifications: the specification for a data-loop-free circuit is given by a function without recursion, while we define a recursive function for an iterative circuit. To exhibit our approach to iterative circuits, we use it to specify and verify three types of iterative circuits: shift registers, serial adders, and circuits computing the greatest-common-divisor (GCD) of two natural numbers. These relatively simple examples are sufficiently complex to demonstrate the generality of our approach.

6.1 Example 1: Shift Register

We formalize two shift register models: serial-in, parallel-out (SIPO); and parallel-in, serial-out (PISO). These two models include counters that keep track of the number of shift operations performed. The value of a counter is initially set to the size of the corresponding shift register's internal data. This value will be decremented by one every time a shift operation occurs. Our SIPO shift register will shift in an input bit to the *most-significant-bit* (MSB) position of its internal data when the counter value is non-zero. It will report its internal data (a bit-vector) to the output when the counter value is zero; at this point the counter value will be reset back to the size of the internal data. Similarly, our PISO shift register will input a bit-vector when its counter value is zero and reset the counter value back to the size of the internal data; it will shift out the *least significant bit* (LSB) from its internal data only if the counter value is non-zero.

We construct shift registers from links and joints. Here we discuss our link-joint model of the SIPO shift register. The PISO shift register is constructed in a similar manner. The data flow of our SIPO shift register model forms a feedback loop as shown in Figure 6.1a. Links R_d and W_d store internal data, while links R_c and W_c hold counter values. Joint *sh* performs two mutually exclusive actions, depending on the value of R_c. Both actions require that R_d and R_c are full, and that W_d and W_c are empty. If the value of R_c is non-zero, *sh* will shift in *bit-in* to the MSB position of the data reported from R_d and store the result in W_d. In the meantime, *sh* will also subtract R_c's value by one and store the result in W_c. When R_c's value is zero, *sh* will output R_d's data and reset the value in W_c back to the size of R_d's data.

Figure 6.1b displays the data flow of a complex joint PISO2 that consists of two PISO shift registers sharing the same communication signal at



(a) SIPO

(b) PISO2

Figure 6.1: Data flows of (a) a SIPO shift register and (b) a double PISO2 shift register. m and n denote the counter and data sizes, respectively.

their input ports. This means that they must accept their corresponding inputs at the same time but can report their outputs at different times. We use PISO2 in our design of the serial adder described in the next section. Joint *sh* in PISO2 has three actions: one accepts two input operands d_0 and d_1, the other two report outputs from d_0 and d_1 accordingly.

We now present the single-step-update property and multi-step input-output relationship for SIPO. Those properties for PISO2 are formalized in a similar manner. Below is the single-step-update property for SIPO.

$$\begin{aligned}&\textit{sipo-sreg\$extract}(\textit{sipo-sreg\$step}(\textit{inputs}, \textit{st}, \textit{data-size})) = \\ &\textit{sipo-sreg\$extracted-step}(\textit{inputs}, \textit{st})\end{aligned} \tag{6.1}$$

The extraction function *sipo-sreg$extract* returns the valid data stored in SIPO, given that R_d and R_c are either both full or both empty, W_d and W_c are either both full or both empty, and R_d's status is different from W_d's status.

```
$\textit{sipo-sreg\$extract}(st) :=$
  **if** $full(st.R_d.s)$
  **then** $\textit{nthcdr}(\textit{v-to-nat}(\textit{strip-cars}(st.R_c.d)), \textit{strip-cars}(st.R_d.d))$
  **else** $\textit{nthcdr}(\textit{v-to-nat}(\textit{strip-cars}(st.W_c.d)), \textit{strip-cars}(st.W_d.d))$
```

Function $\textit{nthcdr}(n, l)$ removes the first n elements from the list l, and function $\textit{v-to-nat}(v)$ converts the bit-vector v into its natural number representation. For example, $\textit{v-to-nat}([1, 0, 1, 1]) = 13$. Note that in our bit-vector representation, lower-order bits are stored at lower indices. Therefore, the four-bit vector $[1, 0, 1, 1]$ represents the binary number 1101_2, which is 13 in decimal.

The extracted next-state function *sipo-sreg$extracted-step* is given as follows. Note, signals *in-act* and *out-act* cannot be both active at the same time since they are associated with two mutually exclusive actions of joint sh.

$$\mathit{sipo\text{-}sreg\$extracted\text{-}step}(\mathit{inputs}, \mathit{st}) :=$$
$$\begin{cases} [], & \mathit{if}\ \mathit{out\text{-}act} = t \\ \mathit{sipo\text{-}sreg\$extract}(\mathit{st}) \mathbin{++} [\mathit{inputs.bit\text{-}in}], & \mathit{if}\ \mathit{in\text{-}act} = t \\ \mathit{sipo\text{-}sreg\$extract}(\mathit{st}), & \mathit{otherwise} \end{cases}$$

Property (6.1) holds when the following predicate is satisfied.

$$\mathit{sipo\text{-}sreg\$inv}(\mathit{st}) :=$$
$$\mathit{st}.R_d.s = \mathit{st}.R_c.s \wedge$$
$$\mathit{st}.W_d.s = \mathit{st}.W_c.s \wedge$$
$$\mathit{st}.R_d.s \neq \mathit{st}.W_d.s \wedge$$
$$(\mathit{empty}(\mathit{st}.R_c.s) \vee \mathit{v\text{-}to\text{-}nat}(\mathit{strip\text{-}cars}(\mathit{st}.R_c.d)) \leq \mathit{len}(\mathit{st}.R_d.d)) \wedge$$
$$(\mathit{empty}(\mathit{st}.W_c.s) \vee \mathit{v\text{-}to\text{-}nat}(\mathit{strip\text{-}cars}(\mathit{st}.W_c.d)) \leq \mathit{len}(\mathit{st}.W_d.d))$$

Once property (6.1) is established and *sipo-sreg$inv* is proved to be invariant, the following multi-step input-output relationship can be certified by induction.

$$[\mathit{sipo\text{-}sreg\$extract}(\mathit{sipo\text{-}sreg\$run}(\mathit{inputs\text{-}seq}, \mathit{st}, \mathit{data\text{-}size}, n))] \mathbin{++} \mathit{out\text{-}seq} =$$
$$\mathit{pack\text{-}rev}(\mathit{data\text{-}size}, \mathit{sipo\text{-}sreg\$extract}(\mathit{st}) \mathbin{++} \mathit{rev}(\mathit{in\text{-}seq}))$$

Function *rev* reverses a list. Function *pack-rev* is defined as follows,

```
$pack\text{-}rev(n, l) :=$
  if $n \leq 0 \vee l = []$
  then $[]$
  else if $len(l) \leq n$
  then $[l]$
  else $pack\text{-}rev(n, nthcdr(n, l)) \mathbin{++} [take(n, l)]$
```

where $take(n, l)$ collects the first n elements of the list l. Below is an example of applying *pack-rev* to concrete arguments.

$$pack\text{-}rev(3, [1, 2, 3, 4, 5, 6, 7]) = [[7], [4, 5, 6], [1, 2, 3]]$$

6.2 Example 2: Serial Adder

A serial adder is a digital circuit that performs binary addition via bit additions, from LSB to MSB, one at a time. Bit addition is performed using a 1-bit full-adder. Unlike our previous design [12], the new serial adder model, *s-add* (Figure 6.2), is not associated with any external counter; the counters are entirely internal to the shift registers. This eases the compositional reasoning task for the serial adder. In our new design, two input operands are stored in the double PISO2 shift register *piso*2 and the accumulated sum is stored in the SIPO shift register *sipo*. Link C_i stores a carry-in bit for bit addition and is initially set to 0. The carry-out result from a bit addition will be used as

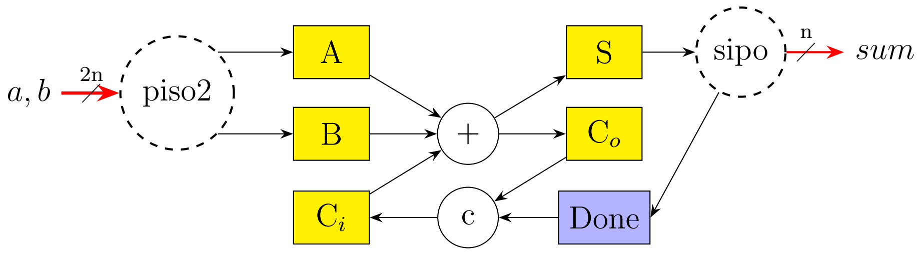


Figure 6.2: Data flow of a serial adder *s-add*. Unless specified, every arrow carries one-bit data.

the carry-in for the next bit addition, except for the final bit addition on two current operands. In our serial adder model, joint c will copy the carry-out bit in link C_o to link C_i when the binary value reported from link *Done* is 0; otherwise joint c will reset the value in link C_i back to 0. The value stored in link *Done* indicates whether the serial adder has finished its computation on the current operands. When joint *sipo* receives a sum bit from link S, it detects, by consulting its counter value, if this bit is the result from the bit addition on the MSBs of two current operands. If yes, it will issue a high signal to link *Done*; otherwise link *Done* will get a low signal.

Consider the following binary addition function.

```
$add(c_i, a, b) :=$
  **if** $(a = NULL)$
  **then** $[c_i]$
  **else**
    **let** $a_i := first(a),$
        $b_i := first(b),$
        $c_o := or3(and(a_i, b_i), and(a_i, c_i), and(b_i, c_i))$
    **return** $[xor3(c_i, a_i, b_i)] ++ add(c_o, rest(a), rest(b))$
```

We prove that *s-add* implements the above addition algorithm, except that it excludes the last carry-out bit from the final result and the initial carry-in bit is set to 0. Thus, it computes an n-bit sum from two n-bit input operands a and b, as specified in the following function.

$sum(a, b) := remove\text{-}last(add(0, a, b))$

Below is the multi-step input-output relationship for *s-add*,

$$\begin{aligned}&\textbf{let } st_f := s\text{-}add\$run(inputs\text{-}seq, st, data\text{-}size, n)\\&\textbf{return } s\text{-}add\$extract(st_f, data\text{-}size) ++ out\text{-}seq =\\&\qquad s\text{-}add\$op\text{-}map(in\text{-}seq) ++ s\text{-}add\$extract(st, data\text{-}size)\end{aligned} \tag{6.2}$$

where function *s-add\$op-map* performs the functionality of *s-add* over a data sequence.

```
$s\text{-}add\$op\text{-}map(seq) :=$
  **if** $(seq = NULL)$
  **then** $[]$  // an empty list
  **else**
    **let** $in := first(seq)$
    **return** $[sum(in.a, in.b)] ++ s\text{-}add\$op\text{-}map(rest(seq))$
```

From (6.2), it trivially follows that $out\text{-}seq = s\text{-}add\$op\text{-}map(in\text{-}seq)$ when the initial and final states of *s-add* contain no valid data. Theorem (6.2) involves the extraction function $s\text{-}add\$extract(x, data\text{-}size)$ that computes the future output sequence from state x; notice that at most two pairs of operands can reside in the internal state of *s-add*.

```
$s\text{-}add\$extract(st, data\text{-}size) :=$
  **let** $A.valid\text{-}d :=$ **if** $full(st.A.status)$ **then** $[st.A.data]$ **else** $[]$,
      $B.valid\text{-}d :=$ **if** $full(st.B.status)$ **then** $[st.B.data]$ **else** $[]$,
      // The full/empty states of $C_i$ and $C_o$ must be different.
      // See the definition of s-add$inv.
      $c :=$ **if** $full(st.C_i.status)$ **then** $st.C_i.data$ **else** $st.C_o.data$,
      $S.valid\text{-}d :=$ **if** $full(st.S.status)$ **then** $[st.S.data]$ **else** $[]$,
      // Extract valid data from the first shift register of $piso2$
```

```
    $piso0.valid\text{-}d := piso2\$extract_0(st.piso2)$,
    // Extract valid data from the second shift register of $piso2$
    $piso1.valid\text{-}d := piso2\$extract_1(st.piso2)$,
    // Extract valid data in $sipo$
    $sipo.valid\text{-}d := sipo\$extract(st.sipo)$,
    $in0 := A.valid\text{-}d \mathbin{++} piso0.valid\text{-}d$,
    $in1 := B.valid\text{-}d \mathbin{++} piso1.valid\text{-}d$,
    $out := sipo.valid\text{-}d \mathbin{++} S.valid\text{-}d$
return
  cases:
  $len(in0) + len(out) = 2 * data\text{-}size :$   // $st$ contains two operand pairs.
    cases:
    $len(out) < data\text{-}size :$
      $[sum(piso0.valid\text{-}d, piso1.valid\text{-}d), out \mathbin{++} [xor3(c, st.A.data, st.B.data)]]$
    $len(out) = data\text{-}size : [sum(in0, in1), out]$
    otherwise: $[S.valid\text{-}d \mathbin{++} remove\text{-}last(add(c, in0, in1)), sipo.valid\text{-}d]$
  $len(in0) + len(out) = data\text{-}size :$   // $st$ contains one operand pair.
    cases:
    $len(out) = 0 : [sum(in0, in1)]$
    $len(out) < data\text{-}size : [out \mathbin{++} remove\text{-}last(add(c, in0, in1))]$
    otherwise: $[out]$
```

otherwise: [] `// ` *st* `contains no data.`

Theorem (6.2) is proved by induction after proving the following single-step-update property.

$$s\text{-}add\$extract(s\text{-}add\$step(inputs, st), data\text{-}size) = \\ s\text{-}add\$extracted\text{-}step(inputs, st, data\text{-}size) \tag{6.3}$$

The extracted next-state function *s-add$extracted-step* extracts the future output sequence from the next state in terms of the *s-add$extract* function, as defined below,

$$s\text{-}add\$extracted\text{-}step(inputs, st, data\text{-}size) := \\ \begin{cases} s\text{-}add\$extract(st, data\text{-}size), & \textit{if } in\text{-}act = nil \wedge out\text{-}act = nil \\ [sum(inputs.a, inputs.b)] \mathbin{++} s\text{-}add\$extract(st, data\text{-}size), & \textit{if } in\text{-}act = t \wedge out\text{-}act = nil \\ remove\text{-}last(s\text{-}add\$extract(st, data\text{-}size)), & \textit{if } in\text{-}act = nil \wedge out\text{-}act = t \\ [sum(inputs.a, inputs.b)] \mathbin{++} remove\text{-}last(s\text{-}add\$extract(st, data\text{-}size)), & \textit{otherwise} \end{cases}$$

where *in-act* and *out-act* denote the communication signals associated with the input and output ports of *s-add* respectively. Note that *s-add* and *piso*2 share the same *in-act* signal; and *s-add* and *sipo* share the same *out-act* signal. *s-add* accepts a new input data item each time the *in-act* signal fires. Similarly, *s-add* reports a data item to the output when *out-act* fires.

Lemma (6.3) is proved in a hierarchical manner using the single-step-update properties of submodules *piso*2 and *sipo*. (6.3) holds only when *st* satisfies the following condition.

```
$s\text{-}add\$inv(st, data\text{-}size) :=$
  let $A.valid\text{-}d :=$ if $full(st.A.status)$ then $[st.A.data]$ else $[]$,
      $B.valid\text{-}d :=$ if $full(st.B.status)$ then $[st.B.data]$ else $[]$,
      $S.valid\text{-}d :=$ if $full(st.S.status)$ then $[st.S.data]$ else $[]$,
      $piso0.valid\text{-}d := piso2\$extract_0(st.piso2)$,
      $piso1.valid\text{-}d := piso2\$extract_1(st.piso2)$,
      $sipo.valid\text{-}d := sipo\$extract(st.sipo)$,
      $in0 := A.valid\text{-}d \mathbin{++} piso0.valid\text{-}d$,
      $in1 := B.valid\text{-}d \mathbin{++} piso1.valid\text{-}d$,
      $out := sipo.valid\text{-}d \mathbin{++} S.valid\text{-}d$
  return
    $st.C_i.status \neq st.C_o.status \land$
    $(empty(st.C_i.status) \lor empty(st.Done.status)) \land$
    $(empty(st.C_o.status) \lor st.S.status \neq st.Done.status) \land$
    $(empty(st.S.status) \lor (full(st.C_o.status) \land empty(st.Done.status))) \land$
    $(empty(st.C_i.status) \lor (len(in0) \neq 0 \land len(in0) \neq data\text{-}size) \lor st.C_i.data = 0) \land$
    $(empty(st.Done.status) \lor$
     if $len(sipo.valid\text{-}d) = 0 \lor len(sipo.valid\text{-}d) = data\text{-}size$
     then $st.Done.data = 1$
     else $st.Done.data = 0) \land$
```

$len(in0) = len(in1) \wedge$

$len(in0) + len(out) \in \{0, data\text{-}size, 2 * data\text{-}size\} \wedge$

$piso2\$inv(st.piso2) \wedge$ `// piso2's state invariant`

$sipo\$inv(st.sipo)$ `// sipo's state invariant`

Since we prove that *s-add$inv* is an invariant, our induction proof for (6.2) is applicable to any initial state that satisfies *s-add$inv*. Once property (6.3) is established, we expand only the definition of *s-add$extracted-step* when reasoning about *s-add*'s behavior. Notice that *s-add$extracted-step* performs case analysis only on the communication signals *in-act* and *out-act* at *s-add*'s input and output ports respectively, while ignoring how the internal operations may proceed.

6.3 Example 3: GCD

We now illustrate the advantage of our hierarchical reasoning method in substituting functionally equivalent submodules. We show that, without altering the proofs, the functionality of a module still remains unchanged when replacing its submodules with functionally equivalent ones. We experiment with iterative self-timed circuit models that compute the greatest-common-divisor (GCD) of two natural numbers. We prove that those GCD circuit models implement the following algorithm.

```
$gcd\text{-}alg(a, b) :=$
  **if** $(a = 0)$ **then** $b$
  **else if** $(b = 0)$ **then** $a$
  **else if** $(a = b)$ **then** $a$
  **else if** $(a < b)$ **then** $gcd\text{-}alg(b - a, a)$
  **else** $gcd\text{-}alg(a - b, b)$
```

gcd-alg is formalized in ACL2 to serve as the functional specification for *gcd*. By proving the following properties (where d is any common divisor of a and b), we show that *gcd-alg* correctly computes the GCD of two natural numbers a and b.

$$a \ \mathbf{mod} \ gcd\text{-}alg(a, b) = 0,$$

$$b \ \mathbf{mod} \ gcd\text{-}alg(a, b) = 0,$$

$$\big((a > 0) \vee (b > 0)\big) \wedge (a \ \mathbf{mod} \ d = 0) \wedge (b \ \mathbf{mod} \ d = 0)$$
$$\Rightarrow 0 < gcd\text{-}alg(a, b) \wedge d \leq gcd\text{-}alg(a, b)$$

6.3.1 GCD1

We first describe our verification of a GCD circuit model, called *gcd*1, whose loop computation is performed by a storage-free joint [11]. We then show that, without the need to rework proofs, replacing that circuit with a functionally equivalent sequential circuit still preserves the functionality of

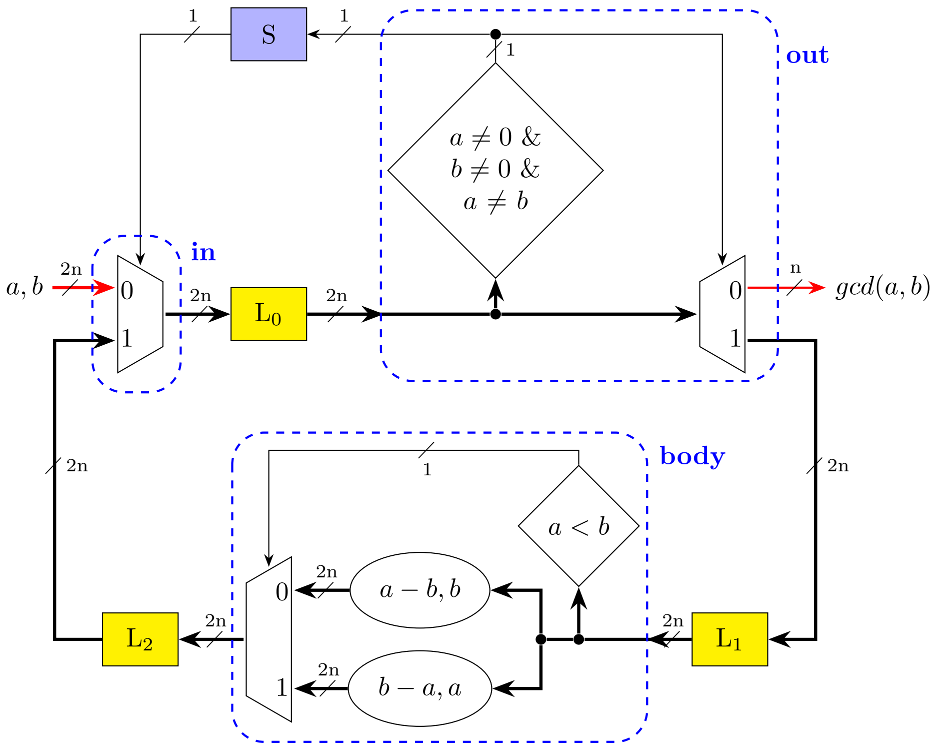


Figure 6.3: Data flow of $gcd1$. n is the number of bits in each operand. Dashed, rounded-corner rectangles identify joints.

the GCD circuit model. Figure 6.3 displays the data flow, based on Sparso's design [64], of $gcd1$. Link S holds a binary value that acts as the mux *select* signal for joint **in**. The two operands stored in link L_0 will be passed to joint **out** to check if they are non-zero and do not have the same value — we call this condition the GCD condition. If the GCD condition is true, the two operands will enter the body of the loop and be stored in link L_1. Otherwise, the circuit will report the result and be ready to accept new inputs after the result has been accepted (because in this case 0 will be stored in link S). In the case the two operands have entered the loop and been stored in link L_1, the greater operand will be updated as described in the *gcd-alg* algorithm. The updated operand and the remaining operand will then be stored in link L_2.

When joint **in** fires, it drains either link L_2 or the link providing the external input [1], according to the value of the mux select signal. Thus, joint **in** has two mutually exclusive actions. In a similar manner, joint **out** also has two mutually exclusive actions: when it fires, it fills link S and either link L_1 or the link accepting the final output [2], depending on the value of the demux select signal. Joint **body** updates the operands as described below.

```
$gcd\text{-}body\$op(a, b) :=$
  **if** $(a < b)$ **then** $swap(a, b)$
  **return** $(a - b, b)$
```

The subtraction operations in joint **body** are executed by two combinational-logic ripple-carry subtractors. We confirm that $gcd1$ satisfies the following multi-step input-output property,

$$gcd1\$extract(gcd\$run(inputs\text{-}seq, st, data\text{-}size, n)) +\!+\; out\text{-}seq = gcd\$op\text{-}map(in\text{-}seq) +\!+\; gcd1\$extract(st) \tag{6.4}$$

where *gcd$op-map* is recursively defined in terms of *gcd-alg*, in the same manner as presented in the definition of *s-add$op-map*; the extraction function

[1]The link that provides the external input is represented only by the small arrow at the left of the drawing labeled a, b in Figure 6.3.

[2]The link that accepts the final output is represented only by the small arrow at the right of the drawing labeled $gcd(a, b)$ in Figure 6.3.

$gcd1\$extract(st)$ extracts the future output sequence from state st as defined below.

$$gcd1\$extract(st) := gcd\$op\text{-}map(extract\text{-}valid\text{-}data([st.L_0, st.L_1, st.L_2]))$$

Note that the order of data items to be extracted does not affect our correctness proof for $gcd1$, because we impose a condition that there be at most one operand pair in the system at any time. In particular, we require the following condition, which we prove is an invariant.

$$\begin{aligned}
&gcd1\$inv(st) := \\
&\quad \textbf{if } full(st.S.s) \wedge (st.S.d = 0) \\
&\quad \textbf{then } len(gcd1\$extract(st)) = 0 \\
&\quad \textbf{else } len(gcd1\$extract(st)) = 1
\end{aligned}$$

This invariant is necessary to maintain the first-in-first-out relationship between input and output sequences. The single-step-update property for $gcd1$, that is used as a supporting lemma for our induction proof of (6.4), is shown below,

$$\begin{aligned}
&gcd1\$extract(gcd\$step(inputs, st, data\text{-}size)) = \\
&gcd1\$extracted\text{-}step(inputs, st, data\text{-}size)
\end{aligned} \tag{6.5}$$

where the extracted next state function $gcd1\$extracted\text{-}step$ is given as follows.

$$gcd1\$extracted\text{-}step(inputs, st, data\text{-}size) :=$$
$$\begin{cases} gcd1\$extract(st), & \text{if } in\text{-}act = nil \wedge out\text{-}act = nil \\ [gcd\text{-}alg(inputs.a, inputs.b)] \mathbin{++} gcd1\$extract(st), & \text{if } in\text{-}act = t \wedge out\text{-}act = nil \\ remove\text{-}last(gcd1\$extract(st)), & \text{if } in\text{-}act = nil \wedge out\text{-}act = t \\ [gcd\text{-}alg(inputs.a, inputs.b)] \mathbin{++} remove\text{-}last(gcd1\$extract(st)), & otherwise \end{cases}$$

Equation (6.5) holds when $gcd1\$inv(st)$ holds. Since we already proved that $gcd1\$inv$ is an invariant, our induction proof for (6.4) still applies as long as the initial state of $gcd1$ satisfies $gcd1\$inv$.

6.3.2 GCD2: Combinational-to-Sequential Substitution

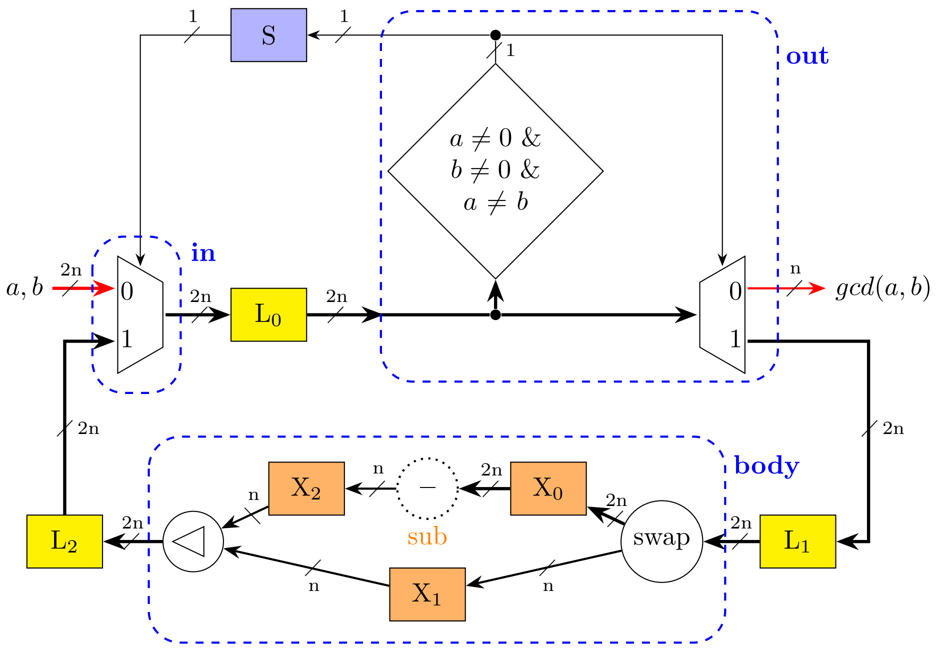


Figure 6.4: Data flow of $gcd2$. Three circles inside complex joint **body** also represent joints. The dotted circle performs the subtraction.

Now let us consider another GCD circuit model, called *gcd*2, whose joint **body** is a sequential circuit (see Figure 6.4). While this complex joint contains three links, it uses only one combinational-logic ripple-carry subtractor. On the other hand, the storage-free joint **body** in *gcd*1 includes two ripple-carry subtractors executing in parallel (see Figure 6.3). We prove that complex joint **body** in *gcd*2 follows the same specification *gcd-body* as described in Section 6.3.1 for primitive joint **body** in *gcd*1. We impose a state invariant, *gcd-body*2\$*inv*, on joint **body** of *gcd*2 that when link X_1 is empty, both links X_0 and X_2 are also empty; and when X_1 is full, either X_0 or X_2 is full.

$$
\begin{aligned}
&\textit{gcd-body}2\$\textit{inv}(st) := \\
&\quad \textit{len}(\textit{extract-valid-data}([st.X_0, st.X_2])) = \textit{len}(\textit{extract-valid-data}([st.X_1]))
\end{aligned}
$$

Under the condition imposed by *gcd-body*2\$*inv*, the extraction function for joint **body** is given as follows.

```
$\textit{gcd-body}2\$\textit{extract}(st, \textit{data-size}) :=$
  **if** $\textit{empty}(st.X_1.s)$
  **then** $[]$
  **else if** $\textit{full}(st.X_0.s)$
  **then** $[\textit{sub}(\textit{take}(\textit{data-size}, \textit{strip-cars}(st.X_0.d)),$
             $\textit{nthcdr}(\textit{data-size}, \textit{strip-cars}(st.X_0.d)))$
        $++\ \textit{strip-cars}(st.X_1.d)]$
  **else** $[\textit{strip-cars}(st.X_2.d)\ ++\ \textit{strip-cars}(st.X_1.d)]$
```

$gcd\text{-}body2\$extract(st, data\text{-}size)$ extracts the future output sequence from state st. Once joint **body** is verified, $gcd2$ can then be verified in the same manner as described for $gcd1$. In other words, our verification framework allows substitution of functionally equivalent submodules without altering the proofs. The step function of $gcd2$ must incorporate the step function of its joint **body** because this joint contains state-holding devices. Furthermore, our approach introduces a set of extraction functions for a sequential circuit. Therefore, when a combinational submodule is substituted by a sequential submodule, the extraction functions for a larger module comprising that sequential submodule must take into account the extraction functions for that submodule. In the case of $gcd2$, its extraction function also considers valid data stored in complex joint **body**, as defined below.

$$\begin{aligned} &gcd2\$extract(st, data\text{-}size) := \\ &\quad gcd\$op\text{-}map(extract\text{-}valid\text{-}data([st.L_0, st.L_1, st.L_2]) \mathbin{++} \\ &\qquad\qquad gcd\text{-}body2\$extract(st.body, data\text{-}size)) \end{aligned}$$

As a consequence, the state invariant of $gcd2$ also includes the state invariant of **body**.

$$
\begin{aligned}
&gcd2\$inv(st, data\text{-}size) := \\
&\quad (\textbf{if } full(st.S.s) \wedge (st.S.d = 0) \\
&\quad\;\; \textbf{then } len(gcd2\$extract(st, data\text{-}size)) = 0 \\
&\quad\;\; \textbf{else } len(gcd2\$extract(st, data\text{-}size)) = 1) \\
&\quad \wedge\, gcd\text{-}body2\$inv(st.body)
\end{aligned}
$$

6.3.3 GCD3: Sequential-to-Sequential Substitution

We now demonstrate the fact that substitutions among functionally equivalent sequential modules do not affect our proofs at all. This also applies for substitutions among functionally equivalent combinational modules. We experiment with replacing the complex joint **body** in *gcd*2 with a functionally equivalent complex joint that performs subtraction via a sequential serial subtractor. We call the corresponding GCD circuit model *gcd*3. Note, *gcd*2 and *gcd*3 differ only in the subtractor design: the former contains a combinational-logic ripple-carry subtractor, while the latter contains a functionally equivalent, sequential serial subtractor. The modeling and verification of our sequential serial subtractor model are absolutely similar to those of the serial adder model mentioned in Section 6.2. The reader may also think of using two serial subtractors in place of the two ripple-carry subtractors in *gcd*1. However this violates the link-joint topology because our serial subtractor model is a complex joint, while the two ripple-carry subtractors in *gcd*1 are just part of the data operations of joint **body**.

Before verifying the functional equivalence between *gcd*2 and *gcd*3, we first certify the functional equivalence between their submodules **body**s. This step is an analogy to verifying the functional equivalence between *gcd*1 and *gcd*2 described in Section 6.3.2. Here we prove that substituting a combinational subtractor with a functionally equivalent sequential subtractor preserves the functionality of joint **body**. That is, we prove that joint **body** in *gcd*3 meets the specification *gcd-body*\$*op*. The step function and extraction function of this joint now incorporate the step function and extraction function of the serial subtractor, respectively. Here we show the definition of the extraction function as an illustration.

```
$\mathit{gcd\text{-}body}3\$\mathit{extract}(st, \mathit{data\text{-}size}) :=$
  if $\mathit{empty}(st.X_1.s)$
  then $[]$
  else if $\mathit{full}(st.X_0.s)$
  then $[\mathit{sub}(\mathit{take}(\mathit{data\text{-}size}, \mathit{strip\text{-}cars}(st.X_0.d)),$
            $\mathit{nthcdr}(\mathit{data\text{-}size}, \mathit{strip\text{-}cars}(st.X_0.d)))$
        $++\ \mathit{strip\text{-}cars}(st.X_1.d)]$
  else if $\mathit{full}(st.X_2.s)$
  then $[\mathit{strip\text{-}cars}(st.X_2.d)\ ++\ \mathit{strip\text{-}cars}(st.X_1.d)]$
  else $[\mathit{first}(\mathit{s\text{-}sub}\$\mathit{extract}(st.\mathit{sub}, \mathit{data\text{-}size}))\ ++\ \mathit{strip\text{-}cars}(st.X_1.d)]$
```

In addition, the state invariant of joint **body** also includes the state invariant of the serial subtractor.

$$\begin{aligned}
&\mathit{gcd\text{-}body3\$inv}(st, \mathit{data\text{-}size}) := \\
&\quad \mathit{len}(\mathit{extract\text{-}valid\text{-}data}([st.X_0, st.X_2]) \mathbin{++} \mathit{s\text{-}sub\$extract}(st.sub, \mathit{data\text{-}size})) \\
&\quad = \mathit{len}(\mathit{extract\text{-}valid\text{-}data}([st.X_1])) \\
&\quad \land\ \mathit{s\text{-}sub\$inv}(st.sub, \mathit{data\text{-}size})
\end{aligned}$$

Once joint **body** is verified, the corresponding GCD module can then be verified in the same manner. Recall that the internal operations of a verified module are abstracted away by using the single-step-update properties for that module. Those properties enable our framework to expand only the definitions of the extracted next state functions, which avoid exploring the module's internal structure, when reasoning about the module behavior. See (4.2) for an example of an extracted next state function. Since our hierarchical proofs do not rely on implementation details of verified submodules, our proofs for both *gcd*2 and *gcd*3 are exactly the same, regardless of how their submodules **body**s are implemented.

As we see that our verification process for iterative circuits presented in this chapter is quite similar to that of data-loop-free circuits discussed in Chapter 5; this further demonstrates the generality and applicability of our framework. Moreover, our case study of the GCD circuit models illustrates the flexibility of our framework that, without the need to rework proofs, the functionality of a module is maintained under functionally equivalent submodule

Table 6.1: Proof times for the self-timed circuits discussed in this chapter

Circuit	**Proof time**	**# *go* signals**	**# *go* signals affecting reasoning**
$SIPO$	10s	2	2
$PISO2$	2.2m	5	5
$s\text{-}add$	2.3m	9	7
$s\text{-}sub$	2.5m	9	7
$gcd\text{-}body1$	2s	1	1
$gcd\text{-}body2$	4s	3	3
$gcd\text{-}body3$	9s	11	4
$gcd1$	8s	3	3
$gcd2$	11s	5	4
$gcd3$	11s	13	4

substitutions. While we are required to update the step function, the extraction functions, and possibly the state invariants for a module when replacing its combinational submodules with functionally equivalent sequential submodules (or in the opposite direction), the proof process for that module still remains unchanged. Table 6.1 reports the verification times of the self-timed circuits discussed in this chapter.

Chapter 7

Arbitrated Merge

All the circuits discussed in the previous two chapters have the first-in-first-out input-output property. For each of those circuits, even though there can be lots of interleavings of link updates happening internally due to non-deterministic delays, the order in the output sequence is completely determined. This chapter will discuss another well-known type of self-timed circuit that produces non-deterministic output sequences, that is, circuits allocating mutually exclusive access to shared resources. These circuits require storing additional information about which request is granted first when there are two requests arriving at approximately the same time. A *mutual-exclusion circuit* or *arbiter* is commonly used in self-timed systems to provide mutually exclusive access to a shared resource on a first-come-first-served (FCFS) basis [60]. Since the arrival times of requests are variable, the grant outcomes are essentially non-deterministic.

In this chapter we present our modeling and verification method for self-timed circuits involving the non-deterministically arbitrated operation mentioned above. In particular, we first describe our formalization of two *arbitrated merge* joint models that provide mutually exclusive access to their output links from their two corresponding input links [59]. We then present our strategy for proving the relationships between the input and output sequences for modules containing such arbitrated merge joints. We believe that having arbitrated merge operations formalized and verified enhances the applicability of our verification system to a broad collection of self-timed circuit models.

7.1 Arbitrated Merge Joint

We formalize two arbitrated merge joint models:

1. A simple model, *arb-merge*1, that always transfers data *randomly* from one of the two full input links to the empty output link if both input links are full "nearly" at the same time.

2. A more complicated model, *arb-merge*2, that supports *fairness* by memorizing its current selection and using this information as an indicator for exclusively servicing the other input link next.

We model *arb-merge*1 as a storage-free joint, to which we add an oracle signal called *select* to perform random selections when necessary. Although quite simple, this arbitrated merge model serves our purpose of illustrating the handling of non-determinism in our verification framework [11]. We prove that *arb-merge*1 follows the specification described in Table 7.1.

We design *arb-merge*2 as a complex joint. Figure 7.1 displays the data flow of *arb-merge*2. When two inputs are available nearly at the same

Table 7.1: Specification of *arb-merge*1

Conditions					Actions			
$in_0.s$	$in_1.s$	$out.s$	$select$	go	$dr(in_0)$	$dr(in_1)$	$fi(out)$	$out.d$
full	empty⋆	empty	_	1	yes	no	yes	$in_0.d$
empty⋆	full	empty	_	1	no	yes	yes	$in_1.d$
full	full	empty	0	1	yes	no	yes	$in_0.d$
full	full	empty	1	1	no	yes	yes	$in_1.d$

fi: fill, dr: drain.
⋆: an *empty* input link of a joint in our modeling is equivalent to a *not full* input link at the lower circuit level.

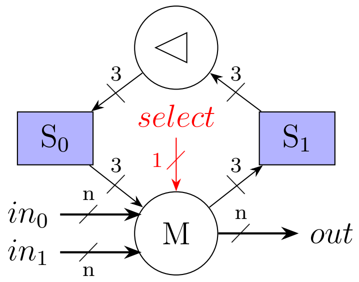


Figure 7.1: Data flow of *arb-merge*2

time, *arb-merge*2 will *randomly* select one of them to service first. In that case, *arb-merge*2 will memorize its selection and use this information as an indicator for exclusively servicing the other input next. Once the other input is serviced, the selection information will be erased from the merge and the process will start over. We also use an oracle signal *select* to perform random selections in *arb-merge*2.

We model *arb-merge*2's operation in two phases: it first grants access for one of the two input links, and then transfers data from the granted input link to the output link. Each of the two internal links S_0 and S_1 of the merge

Table 7.2: Specification of joint M shown in Figure 7.1. In each of the cases listed in the table, it is also required that S_0 and S_1 must be full and empty respectively; and when M acts, it also fills S_1 and drains S_0.

Conditions						**Actions**				
$in_0.s$	$in_1.s$	$out.s$	$S_0.d$	$select$	go_M	$dr(in_0)$	$dr(in_1)$	$fi(out)$	$S_1.d$	$out.d$
full	empty⋆	_	_0_	_	_	no	no	no	010	x
empty⋆	full	_	_0_	_	_	no	no	no	110	x
full	full	_	_0_	0	_	no	no	no	011	x
full	full	_	_0_	1	_	no	no	no	111	x
full	_	empty	010	_	1	yes	no	yes	000	$in_0.d$
_	full	empty	110	_	1	no	yes	yes	000	$in_1.d$
full	_	empty	011	_	1	yes	no	yes	110	$in_0.d$
_	full	empty	111	_	1	no	yes	yes	010	$in_1.d$

_: any value, x: do nothing.
⋆: an *empty* input link of a joint in our modeling is equivalent to a *not full* input link at the lower circuit level.

contains three-bit data:

- The lowest-order bit (memoir bit): the merge will memorize its selection only if this bit is on.
- The middle bit (grant bit) indicates whether the merge already grants access or not: 0 means *no*, 1 means *yes*.
- The highest-order bit (selection bit) indicates which input link to be served. The merge consults this bit only when the grant bit is on.

Table 7.2 specifies *arb-merge*2's operation in granting access and transferring data. The first four cases specify how the merge grants access for the input links. The last four cases specify the merge action when transferring data from the granted input link to the output link. We show two theorems validating the first and last cases in Table 7.2; the other cases are similar. Here is the theorem for the first case.

```
**let** $st' := de(si(\text{`}arg\text{-}merge2, data\text{-}size), inputs, st, netlist)$
**return** $arb\text{-}merge2\&(netlist, data\text{-}size) \wedge$
           $arb\text{-}merge2\$input\text{-}format(inputs, data\text{-}size) \wedge$
           $arb\text{-}merge2\$st\text{-}format(st) \wedge$
           $st.S_0.d[1] = 0 \;\wedge$   // The grant bit in link $S_0$ is 0.
           $inputs.full\text{-}in_0 = t \wedge$
           $\neg inputs.full\text{-}in_1 \wedge$
           $full(st.S_0.s) \wedge$
           $empty(st.S_1.s)$
           $\Rightarrow empty(st'.S_0.s) \wedge$
              $full(st'.S_1.s) \wedge$
              $st'.S_1.d = [0, 1, 0] \wedge$
              // Do not drain link $in_0$
              $\neg arb\text{-}merge2\$act_0(inputs, st, data\text{-}size) \wedge$
              // Do not drain link $in_1$
              $\neg arb\text{-}merge2\$act_1(inputs, st, data\text{-}size) \wedge$
              // Do not fill link $out$
              $\neg arb\text{-}merge2\$act(inputs, st, data\text{-}size)$
```

The following theorem validates the last case in Table 7.2.

```
let $st' := de(si(\text{‘}arg\text{-}merge2, data\text{-}size), inputs, st, netlist)$
return $arb\text{-}merge2\&(netlist, data\text{-}size) \wedge$
       $arb\text{-}merge2\$input\text{-}format(inputs, data\text{-}size) \wedge$
       $arb\text{-}merge2\$st\text{-}format(st) \wedge$
       $st.S_0.d = [1, 1, 1] \wedge$
       $inputs.go_M = t \wedge$
       $inputs.full\text{-}in_1 = t \wedge$
       $\neg inputs.empty\text{-}out\text{-} \wedge$
       $full(st.S_0.s) \wedge$
       $empty(st.S_1.s)$
       $\Rightarrow empty(st'.S_0.s) \wedge$
           $full(st'.S_1.s) \wedge$
           $st'.S_1.d = [0, 1, 0] \wedge$
           // Do not drain link $in_0$
           $\neg arb\text{-}merge2\$act_0(inputs, st, data\text{-}size) \wedge$
           // Drain link $in_1$
           $arb\text{-}merge2\$act_1(inputs, st, data\text{-}size) = t \wedge$
           // Fill link $out$
           $arb\text{-}merge2\$act(inputs, st, data\text{-}size) = t \wedge$
           $out.d = inputs.d_1$
```

7.2 Experiments

We experiment with several self-timed circuit designs involving arbitrated merges. For each circuit model described in this section, we experiment with both arbitrated merge models $arb\text{-}merge1$ and $arb\text{-}merge2$; and the properties reported here are also applied for both models. These experiments further demonstrate the stability of our hierarchical reasoning approach in substituting functionally equivalent submodules. Due to variable arrival times of the two input sequences of an arbitrated merge, the corresponding output sequence can be any possible interleaving of the two input sequences. We use the membership relation ($\in$) and the interleaving operation ($\otimes$) for establishing the multi-step input-output relationships for self-timed circuits performing arbitrated merge operations. For example, the output sequence from an arbitrated merge can be expressed as a member of all possible interleavings of the two input sequences, as follows: $out\text{-}seq \in (in_0\text{-}seq \otimes in_1\text{-}seq)$. The interleaving operation $\otimes$ computes all interleavings of its two input sequences, e.g.,

$$\begin{aligned}[5,1,2] \otimes [a,b] = [&[5,1,2,a,b],[5,1,a,2,b],\\ &[5,1,a,b,2],[5,a,1,2,b],\\ &[5,a,1,b,2],[5,a,b,1,2],\\ &[a,5,1,2,b],[a,5,1,b,2],\\ &[a,5,b,1,2],[a,b,5,1,2]].\end{aligned}$$

Verifying the multi-step input-output relationships for circuits performing arbitrated merges involves developing a library that supports reasoning about the membership relation and the interleaving operation. In this library, referred to as *mem-interl-lib*, we prove lemmas about the preservation of the membership under the concatenation operation with the presence of the interleaving operation. For example,

$$\begin{aligned} x \in (y \otimes z) \Rightarrow (x \mathbin{++} x1) \in ((y \mathbin{++} x1) \otimes z) \wedge \\ (x \mathbin{++} x1) \in (y \otimes (z \mathbin{++} x1)). \end{aligned} \tag{7.1}$$

We will present more key lemmas from *mem-interl-lib* when we discuss our experiments below. As we did for the self-timed circuits mentioned in the previous two chapters, our strategy for proving the multi-step relationship is based on single-step-update properties. For circuits involving arbitrated merges, we introduce two extraction functions to extract two valid input streams for each arbitrated merge; and we prove a single-step-update property for each extraction function, as shown below,

$$extract_0(step(inputs, st)) = extracted_0\text{-}step(inputs, st),$$

$$extract_1(step(inputs, st)) = extracted_1\text{-}step(inputs, st)$$

where $extract_i$ is an extraction function, and $extracted_i\text{-}step$ is the corresponding specification for the one-step update on the output sequence. Each $extracted_i\text{-}step$ function is defined in the same manner as presented in definition (4.2). Note that there are two mutually exclusive actions at each arbitrated merge; and each $extracted_i\text{-}step$ function depends on either one of the two actions.

7.2.1 Example 1

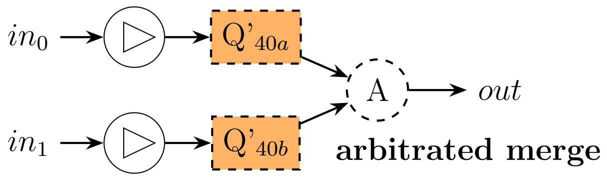


Figure 7.2: Data flow of *interl*

Our first example considers a circuit that connects two 40-link queues to the two input ports of an arbitrated merge (Figure 7.2). Note that these queues are complex links. Let in_0-seq and in_1-seq represent two accepted input sequences connected to Q'_{40a} and Q'_{40b}, respectively. We prove that for any interleaving x of two data sequences remaining in the final state, the concatenation of x and the output sequence must be a member of $(seq_0 \otimes seq_1)$; where seq_0 is the concatenation of in_0-seq and the valid data sequence in Q'_{40a} at the initial state, and seq_1 is the concatenation of in_1-seq and the valid data sequence in Q'_{40b} at the initial state. Formally, we prove the following property,

$$\begin{aligned}
&\textbf{let } st_f := interl\$run(inputs\text{-}seq, st, n),\\
&\forall x \in \big(interl\$extract_0(st_f) \otimes interl\$extract_1(st_f)\big).\\
&\quad (x \mathbin{++} out\text{-}seq) \in \Big(\big(in_0\text{-}seq \mathbin{++} interl\$extract_0(st)\big) \otimes\\
&\qquad\qquad\qquad \big(in_1\text{-}seq \mathbin{++} interl\$extract_1(st)\big)\Big)
\end{aligned} \tag{7.2}$$

where $interl\$extract_0$ and $interl\$extract_1$ extract valid data from Q'_{40a} and Q'_{40b}, respectively. When the initial and final states have no valid data, we obtain the corollary $out\text{-}seq \in (in_0\text{-}seq \otimes in_1\text{-}seq)$. We prove property (7.2) by induction after proving two single-step-update properties (one for each extraction function) and lemma (7.1). Note that the single-step-update properties for *interl* are proved in a hierarchical manner. Specifically, these properties are proved by applying the single-step-update properties of submodules Q'_{40a} and Q'_{40b} accordingly, without exploring the operations internal to these submodules.

7.2.2 Example 2

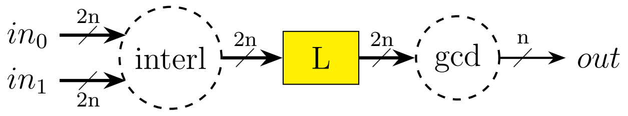


Figure 7.3: Data flow of *igcd*

The next example further illustrates hierarchical reasoning: the verification of a self-timed module that contains self-timed submodules with and without arbitration. We model a module, called *igcd*, that connects the output port of *interl* to the input port of *gcd* via a link (Figure 7.3). Notice that each item in each input stream of *igcd* carries a pair of operands for a GCD operation. We verify the correctness of *igcd* by proving that this circuit produces a sequence of GCDs over any interleaving of two input sequences. Our approach uses three extraction functions that compute the GCDs of the valid data residing in $interl.Q'_{40a}$, $interl.Q'_{40b}$, and the concatenation of L and *gcd*, respectively.

$$\begin{aligned}
&\mathit{igcd\$extract}_0(st) := \mathit{gcd\$op\text{-}map}(\mathit{interl\$extract}_0(st.\mathit{interl})),\\
&\mathit{igcd\$extract}_1(st) := \mathit{gcd\$op\text{-}map}(\mathit{interl\$extract}_1(st.\mathit{interl})),\\
&\mathit{igcd\$extract}_2(st) :=\\
&\quad \mathit{gcd\$op\text{-}map}(\mathit{extract\text{-}valid\text{-}data}([st.L])) \mathbin{++} \mathit{gcd\$extract}(st.\mathit{gcd})
\end{aligned}$$

We formalize the multi-step input-output relationship for *igcd* in terms of function $\mathit{prepend\text{-}rec}(x, y)$ that prepends each list in x to y. For example,

$$\mathit{prepend\text{-}rec}([[1,2],[-3,6,4]],[a,b]) = [[1,2,a,b],[-3,6,4,a,b]].$$

Below is the multi-step input-output relationship for *igcd* that we formalize.

$$\begin{aligned}
&\textbf{let } st_f := \mathit{igcd\$run}(\mathit{inputs\text{-}seq}, st, n),\\
&\forall x \in \big(\mathit{igcd\$extract}_0(st_f) \otimes \mathit{igcd\$extract}_1(st_f)\big).\\
&\quad (x \mathbin{++} \mathit{igcd\$extract}_2(st_f) \mathbin{++} \mathit{out\text{-}seq}) \in\\
&\qquad \mathit{prepend\text{-}rec}\Big(\big(\mathit{gcd\$op\text{-}map}(\mathit{in}_0\text{-}\mathit{seq}) \mathbin{++} \mathit{igcd\$extract}_0(st)\big) \otimes\\
&\qquad\qquad \big(\mathit{gcd\$op\text{-}map}(\mathit{in}_1\text{-}\mathit{seq}) \mathbin{++} \mathit{igcd\$extract}_1(st)\big),\\
&\qquad\qquad \mathit{igcd\$extract}_2(st)\Big)
\end{aligned}$$

Our proof applies the following lemma in *mem-interl-lib*.

$$e \in x \Rightarrow (e \mathbin{++} e1) \in \mathit{prepend\text{-}rec}(x, e1)$$

Again, we apply our hierarchical approach to prove the single-step-update properties for *igcd* from the single-step-update properties of submodules *interl* and *gcd* accordingly.

7.2.3 Example 3

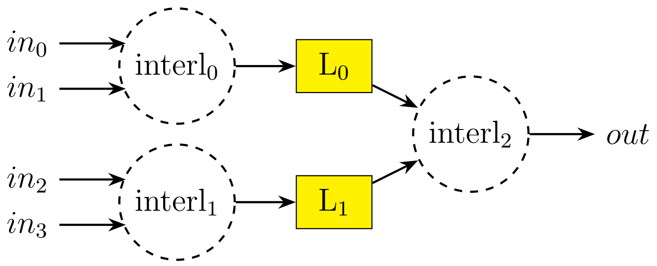


Figure 7.4: Data flow of *comp-interl*

We continue illustrating our hierarchical reasoning method via a circuit model, *comp-interl*, that composes three instances of *interl* as displayed in Figure 7.4. Loosely speaking, we prove that the output sequence from *comp-interl* is an interleaving of its four input sequences. Our method introduces six extraction functions for *comp-interl*, two for each instance of *interl*.

$$comp\text{-}interl\$extract_0(st) := interl\$extract_0(st.interl_0),$$

$$comp\text{-}interl\$extract_1(st) := interl\$extract_1(st.interl_0),$$

$$comp\text{-}interl\$extract_2(st) := interl\$extract_0(st.interl_1),$$

$$comp\text{-}interl\$extract_3(st) := interl\$extract_1(st.interl_1),$$

$$comp\text{-}interl\$extract_4(st) :=$$
$$extract\text{-}valid\text{-}data([st.L_0]) \mathbin{++} interl\$extract_0(st.interl_2),$$

$$\begin{aligned}&\textit{comp-interl\$extract}_5(st) := \\ &\quad \textit{extract-valid-data}([st.L_1]) \mathbin{++} \textit{interl\$extract}_1(st.\textit{interl}_2).\end{aligned}$$

Our correctness theorem for *comp-interl* is stated in terms of the nested interleaving operator, $\otimes_2$, where $x \otimes_2 y$ interleaves each list in x with each list in y. For example,

$$\begin{aligned}[l_1, l_2, l_3] \otimes_2 [l_4, l_5] = {} & (l_1 \otimes l_4) \mathbin{++} (l_1 \otimes l_5) \mathbin{++} (l_2 \otimes l_4) \mathbin{++} \\ & (l_2 \otimes l_5) \mathbin{++} (l_3 \otimes l_4) \mathbin{++} (l_3 \otimes l_5).\end{aligned}$$

We apply the following property from our library, *mem-interl-lib*, in proving the multi-step input-output relationship for *comp-interl*.

$$\begin{aligned}x \in (y \otimes_2 z) \Rightarrow {} & (x \mathbin{++} x1) \in (\textit{prepend-rec}(y, x1) \otimes_2 z) \wedge \\ & (x \mathbin{++} x1) \in (y \otimes_2 \textit{prepend-rec}(z, x1))\end{aligned}$$

Here is the multi-step input-output relationship for *comp-interl*.

$$\begin{aligned}
&\textbf{let } st_f := \mathit{comp\text{-}interl\$run}(\mathit{inputs\text{-}seq}, st, n),\\
&\forall x \in \Big(\mathit{prepend\text{-}rec}\big(\mathit{comp\text{-}interl\$extract}_0(st_f) \otimes \mathit{comp\text{-}interl\$extract}_1(st_f),\\
&\qquad\qquad \mathit{comp\text{-}interl\$extract}_4(st_f)\big)\\
&\quad \otimes_2\\
&\quad \mathit{prepend\text{-}rec}\big(\mathit{comp\text{-}interl\$extract}_2(st_f) \otimes \mathit{comp\text{-}interl\$extract}_3(st_f),\\
&\qquad\qquad \mathit{comp\text{-}interl\$extract}_5(st_f)\big)\Big).\\
&\quad (x \mathbin{++} \mathit{out\text{-}seq}) \in\\
&\quad \Big(\mathit{prepend\text{-}rec}\Big(\big(\mathit{in}_0\text{-}\mathit{seq} \mathbin{++} \mathit{comp\text{-}interl\$extract}_0(st)\big)\otimes\\
&\qquad\qquad \big(\mathit{in}_1\text{-}\mathit{seq} \mathbin{++} \mathit{comp\text{-}interl\$extract}_1(st)\big),\\
&\qquad\qquad \mathit{comp\text{-}interl\$extract}_4(st)\Big)\\
&\quad \otimes_2\\
&\quad \mathit{prepend\text{-}rec}\Big(\big(\mathit{in}_2\text{-}\mathit{seq} \mathbin{++} \mathit{comp\text{-}interl\$extract}_2(st)\big)\otimes\\
&\qquad\qquad \big(\mathit{in}_3\text{-}\mathit{seq} \mathbin{++} \mathit{comp\text{-}interl\$extract}_3(st)\big),\\
&\qquad\qquad \mathit{comp\text{-}interl\$extract}_5(st)\Big)\Big)
\end{aligned}$$

Table 7.3 reports the verification times of the self-timed circuits discussed in Section 7.2. The proof times were the same for both arbitrated merge models, except for the case of module *interl* where it took 5 seconds using *arb-merge*1 but 9 seconds using *arb-merge*2. This makes sense because *arb-merge*1 is a combinational circuit, while *arb-merge*2 contains state-holding devices, specifically, two links S_0 and S_1. In the latter case, the *step*

Table 7.3: Proof times for the self-timed circuits discussed in Section 7.2

Arbiter	**Circuit**	**Proof time**	**# *go* signals**	**# *go* signals affecting reasoning**
*arb-merge*1	*interl*	5s	81	3
	igcd	12s	84	5
	comp-interl	23s	243	9
*arb-merge*2	*interl*	9s	82	3
	igcd	12s	85	5
	comp-interl	23s	246	9

function for *interl* also takes into account the *step* function for *arb-merge*2; this would take more time to reason about the *step* function for *interl*. On the other hand, the implementation detail of an arbitrated merge does not affect the verification times of modules *igcd* and *comp-interl* because the merge is hidden by *interl* in these two modules.

Part IV

Epilogue

Chapter 8

Conclusions

8.1 Summary

This dissertation presents a framework for formally modeling and verifying self-timed circuits using the ACL2 theorem-proving system. Our goal is to develop a methodology that is capable of verifying the functional correctness of self-timed circuit designs at scale. This project also provides a library for analyzing self-timed systems in ACL2. Our modeling approach is based on the link-joint model to represent self-timed systems as networks of links communicating with each other locally via handshake components, which are called joints. We show that the existing DE system already proven to be successful for synchronous circuits is adaptable for handling self-timed systems by reasoning with *go* signals as well as state-holding elements that have their own gating. This hierarchical HDL provides combinational primitives as well as several latches suitable for creating links and joints. We extend DE to represent self-timed circuits by adding a single link-control primitive that coordinates the means to update the state of a link. Using the extended DE system to describe circuit implementations, we show how those implementations can be *lifted* into more abstract four-valued models that have required properties; subsequently, these four-valued models are shown to have single-step-update properties that can be derived from the single-step-update properties of sub-circuits. This method demonstrates the verification of Mealy machines hierarchically; it can be used to verify both clocked and self-timed circuit models.

We have discussed the specification and verification of self-timed circuits represented with a formally-defined, hierarchical HDL. These circuits may have loops, arbitrated merges, and may be parameterized by data width. Hierarchical verification is a key methodology supporting efficiency of our correctness proofs.

The sequential nature of circuits can be coarse-grained, such as in a globally-clocked circuit, or it can be fine-grained, such as in a self-timed circuit. Generally the number of reachable states in a self-timed circuit is much greater than its equivalent clocked circuit — this *state explosion* increases the difficulty of analysis and verification, but it may simplify the design and implementation by avoiding the *global timing closure* problem.

We model the interleavings of event-ordering in self-timed circuits by associating each joint with an external *go* signal that, when disabled, prevents a joint from *firing*. A joint action will fire only when all of its input-output conditions and its externally-provided (from an oracle) *go* signal are valid. Thus, when we undertake the verification of a circuit composed of combinational-logic joints and state-holding links, we are modeling all of the possible interleavings of circuit activity. All proofs about our self-timed circuit models reported here are carried out respecting this circuit advancement freedom.

Note, the non-determinism considered in the proofs presented is at the *level* of state-holding links and combinational joints. Our model makes no mention of the actual, real-time delay of corresponding circuit implementations, although physical circuit implementations must provide non-zero delay through links, which is of course unavoidable in actual circuits. Even so, to implement self-timed circuits successfully, such as those discussed in this dissertation, it would be necessary to use traditional, circuit-level, timing-analysis tools to make sure that internally-produced *done* signals arrive after all bundled data arrives at the receiving (storage) links. Such timing analysis can be carried out only after a design has been committed to a specific technology. Our specification and verification approach can also be used for purely DI and QDI circuits. In fact, we believe our analysis approach may extend to more general computational models, such as Hoare's CSP [24].

We believe this is the first approach using theorem proving with a hierarchical functional verification methodology for self-timed circuit models specified at the link-joint level. As the hierarchical approach we are employing for self-timed circuits represents a generalization of earlier efforts using DE, we believe we will be able to verify arbitrarily large, general-purpose, self-timed circuit designs. In fact, we believe our approach is sufficiently general that it may be used to model and analyze combined synchronous and self-timed systems [55, 73, 23, 6, 57]; for instance, certifying the correctness of data exchange between two synchronous systems over an asynchronous interconnect fabric.

8.2 Future Work

This section discusses ideas for extending this research.

We currently have no mechanism to check if our self-timed circuit models actually follow the link-joint topology: links are connected via joints, and joints are connected via links. Potential future work may develop a syntactic checker that detects link-joint topology violations in self-timed circuit designs. We imagine such a checker can operate in a hierarchical manner by adding some annotations to each module informing the checker the component type, i.e. either link or joint, at each module's input/output port. That way we can validate this information for each submodule separately and then use it to verify the link-joint topology compliance of larger modules composing of those submodules.

It would make our framework more practical if one can implement tools that convert our DE descriptions of self-timed modules to other HDL formats commonly used by designers (such as Verilog) and/or vice versa. A challenge emerging from this work is to ensure the equivalence between any two formats. In other words, we may need to guarantee the correctness of implemented transformations. Some proof obligations may be required to assure the reliability of those transformations.

There is a lot of room for increasing automation of our framework through the further introduction of macros. For instance, the value and state lemmas proof process can be done automatically as this step is quite routine:

the *outputs* and *step* functions at the four-valued level of a module can be produced using the ACL2 simplifier. We believe the APT tool developed at Kestrel Institute [14, 28] would be suitable for generating value and state lemmas mechanically.

Another possible direction for future work is to extend our methodology to other types of self-timed circuits, for instance self-timed, parameterized FFT circuits. Very high performance FFT implementations are well served by self-timed implementations, especially when used in multi-gigahertz direct signal conversion for 5G transceivers. Self-timed circuits with non-deterministic outputs are also interesting as they potentially impose great challenges for circuit analysis. Studying such systems could enhance our framework by developing strategies and libraries that support reasoning about new operations pertinent to those systems. This would benefit the ACL2 community as well, particularly for those who also use those operations in their work.

The DE system has been used to specify and verify synchronous microprocessor designs, such as the FM9001 microprocessor design [9]. An interesting project may apply the extended DE system along with our methodology to modeling self-timed microprocessors and verifying their functional properties. One could consider modeling the self-timed version of FM9001 and verifying its functional correctness in terms of input and output sequences, using the hierarchical verification technique presented in this dissertation. Such work can benefit from the previous work on the synchronous FM9001 specification and verification [5, 9], especially reusing combinational modules of the existing FM9001 model in designing primitive joints for a self-timed FM9001 design. Modeling a register file in the link-joint paradigm can follow the idea of storing two copies of registers in two links, as applied in our shift registers' designs (see Section 6.1). A memory model can also follow that same modeling idea. A further goal in this direction would be applying our framework to self-timed *pipelined* microprocessor designs, which are an active research area in the asynchronous community [69, 18, 52, 66, 35, 20, 40, 19, 30, 50, 61].

We also expect to consider the verification of mixed self-timed, synchronous circuits. For instance, we wish to verify the correctness of data exchange between two synchronous systems over an asynchronous interconnect fabric. Such an advance could contribute to the use of self-timed networks to reduce the use of inter-clock-domain synchronizers.

Bibliography

[1] ACL2 User Manual. http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=ACL2____TOP.

[2] P. A. Beerel, R. O. Ozdag, and M. Ferretti. *A Designer's Guide to Asynchronous VLSI*. Cambridge University Press, 2010.

[3] P. A. Beerel and M. E. Roncken. Low Power and Energy Efficient Asynchronous Design. *Low Power Electronics (JOLPE-2007)*, 3(3):234–253, 2007.

[4] M. Bozga, H. Jianmin, O. Maler, and S. Yovine. Verification of Asynchronous Circuits using Timed Automata. *Electronic Notes in Theoretical Computer Science*, 65(6):47–59, 2002.

[5] B. C. Brock and W. A. Hunt Jr. The DUAL-EVAL Hardware Description Language and Its Use in the Formal Specification and Verification of the FM9001 Microprocessor. In *Formal Methods in System Design*, volume 11, pages 71–104. Kluwer Academic Publishers, 1997.

[6] J. Carmona, J. Cortadella, M. Kishinevsky, and A. Taubin. Elastic Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(10):1437–1455, 2009.

[7] S. Chakraborty, D. I. Dill, and K. Y. Yun. Min-Max Timing Analysis and an Application to Asynchronous Circuits. In *Proc of the IEEE*, volume 87, pages 332–346, 1999.

[8] S. Chakraborty, K. Y. Yun, and D. L. Dill. Practical Timing Analysis of Asynchronous Circuits Using Time Separation of Events. In *Proc of the IEEE 1998 Custom Integrated Circuits Conference (Cat. No.98CH36143)*, pages 455–458, 1998.

[9] C. Chau. Extended Abstract: Formal Specification and Verification of the FM9001 Microprocessor Using the DE System. In *Proc of the Fourteenth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2017)*, pages 112–114, 2017.

[10] C. Chau, W. A. Hunt Jr., M. Kaufmann, M. Roncken, and I. Sutherland. Data-Loop-Free Self-Timed Circuit Verification. In *Proc of the Twenty Fourth IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC-2018)*, pages 51–58, 2018.

[11] C. Chau, W. A. Hunt Jr., M. Kaufmann, M. Roncken, and I. Sutherland. A Hierarchical Approach to Self-Timed Circuit Verification. In *Proc of the Twenty Fifth IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC-2019)*, 2019. To appear.

[12] C. Chau, W. A. Hunt Jr., M. Roncken, and I. Sutherland. A Framework for Asynchronous Circuit Modeling and Verification in ACL2. In *Proc*

of the Thirteenth Haifa Verification Conference (HVC-2017), pages 3–18, 2017.

[13] E. Clarke and B. Mishra. Automatic Verification of Asynchronous Circuits. In *Proc of the Workshop on Logic of Programs*, pages 101–115, 1983.

[14] A. Coglio, M. Kaufmann, and E. Smith. A Versatile, Sound Tool for Simplifying Definitions. In *Proc of the Fourteenth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2017)*, pages 61–77, 2017.

[15] K. Desai, K. S. Stevens, and J. O'Leary. Symbolic Verification of Timed Asynchronous Hardware Protocols. In *Proc of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI-2013)*, pages 147–152, 2013.

[16] D. L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. MIT press, 1989.

[17] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, S. Temple, and J. V. Woods. The Design and Evaluation of an Asynchronous Microprocessor. In *Proc of the IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD-1994)*, pages 217–220, 1994.

[18] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, and J. V. Woods. AMULET1: A Micropipelined ARM. In *Proc of COMPCON '94*, pages 476–485, 1994.

[19] S. B. Furber, J. D. Garside, and D. A. Gilbert. AMULET3: A High-Performance Self-Timed ARM Microprocessor. In *Proc of the IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD-1998)*, pages 247–252, 1998.

[20] S. B. Furber, J. D. Garside, S. Temple, J. Liu, P. Day, and N. C. Paver. AMULET2e: An Asynchronous Embedded Controller. In *Proc of the Third International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC-1997)*, pages 290–299, 1997.

[21] M. Gordon. HOL: A Machine Oriented Formulation of Higher Order Logic. Technical report, University of Cambridge, Computer Laboratory, 1985.

[22] M. J. C. Gordon. HOL: A Proof Generating System for Higher-Order Logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128. Springer, Boston, MA, 1988.

[23] E. Grass, B. Sarker, and K. Maharatna. A Dual-Mode Synchronous/Asynchronous CORDIC Processor. In *Proc of the Eighth International Symposium on Asynchronous Circuits and Systems (ASYNC-2002)*, pages 76–83, 2002.

[24] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[25] W. A. Hunt Jr. The DE Language. In M. Kaufmann, P. Manolios, and J S. Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, chapter 10, pages 151–166. Springer US, 2000.

[26] W. A. Hunt Jr., M. Kaufmann, J S. Moore, and A. Slobodova. Industrial Hardware and Software Verification with ACL2. *Philosophical Transactions of the Royal Society*, 375(2104), 2017.

[27] W. A. Hunt Jr. and E. Reeber. Applications of the DE2 Language. In *Proc of the Sixth International Workshop on Designing Correct Circuits (DCC-2006)*, 2006.

[28] Kestrel Institute. Automated Program Transformations. `https://www.kestrel.edu/home/projects/apt/`, 2019.

[29] P. Joshi, P. A. Beerel, M. Roncken, and I. Sutherland. Timing Verification of GasP Asynchronous Circuits: Predicted Delay Variations Observed by Experiment. In D. Dams, U. Hannemann, and M. Steffen, editors, *Lecture Notes in Computer Science*, chapter 17, pages 260–276. Springer Berlin Heidelberg, 2010.

[30] H. K. Kapoor. Formal Modelling and Verification of an Asynchronous DLX Pipeline. In *Proc of the Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM-2006)*, pages 118–127, 2006.

[31] M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Press, Boston, MA, 2000.

[32] M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Press, Boston, MA, 2000.

[33] M. Kaufmann and J S. Moore. ACL2 Home Page. `http://www.cs.utexas.edu/users/moore/acl2/`, 2019.

[34] H. Kim, P. A. Beerel, and K. Stevens. Relative Timing Based Verification of Timed Circuits and Systems. In *Proc of the Eighth International Symposium on Asynchronous Circuits and Systems (ASYNC-2002)*, pages 115–124, 2002.

[35] P. N. Loewenstein. Formal Verification of Counterflow Pipeline Architecture. In *Proc of the Eighth International Conference on Higher Order Logic Theorem Proving and Its Applications (TPHOLs-1995)*, pages 261–276, 1995.

[36] O. Maler and A. Pnueli. Timing Analysis of Asynchronous Circuits using Timed Automata. In *Proc of the IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME-1995)*, pages 189–205, 1995.

[37] J. V. Manoranjan and K. S. Stevens. Qualifying Relative Timing Constraints for Asynchronous Circuits. In *Proc of the Twenty Second In-*

ternational Symposium on Asynchronous Circuits and Systems (ASYNC-2016), pages 91–98, 2016.

[38] A. J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing*, 1(4):226–234, 1986.

[39] A. J. Martin, S. M. Burns, T. K. Lee, D. Borkovic, and P. J. Hazewindus. The First Asynchronous Microprocessor: The Test Results. *ACM SIGARCH Computer Architecture News*, 17(4):95–98, 1989.

[40] A. J. Martin, A. Lines, R. Manohar, M. Nystrom, P. Penzes, R. Southworth, U. Cummings, and T. K. Lee. The Design of an Asynchronous MIPS R3000 Microprocessor. In *Proc of the Seventeenth Conference on Advanced Research in VLSI (ARVLSI-1997)*, pages 164–181, 1997.

[41] T. M. McWilliams. Verification of Timing Constraints on Large Digital Systems. In *Proc of the Seventeenth Design Automation Conference (DAC-1980)*, pages 139–147, 1980.

[42] C. Mead and L. Conway. *Introduction to VLSI Systems.* Addison-Wesley, 1980.

[43] D. E. Muller. Asynchronous Logics and Application to Information Processing. In *Proc of Symposium on Application of Switching Theory in Space Technology*, pages 289–297, 1963.

[44] C. J. Myers. *Asynchronous Circuit Design.* Wiley, 2001.

[45] C. D. Nielsen and A. J. Martin. Design of a Delay-Insensitive Multiply-Accumulate Unit. *Integration*, 15(3):291–311, 1993.

[46] L. S. Nielsen. *Low-power Asynchronous VLSI Design.* PhD thesis, Department of Information Technology, Technical University of Denmark, 1997.

[47] L. S. Nielsen, C. Niessen, J. Sparso, and K. van Berkel. Low-Power Operation Using Self-Timed Circuits and Adaptive Scaling of the Supply Voltage. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2(4):391–397, 1994.

[48] L. S. Nielsen and J. Sparso. An 85 μW Asynchronous Filter-Bank for a Digital Hearing Aid. In *Proc of the IEEE International Solid State circuits Conference*, pages 108–109, 1998.

[49] C. Niessen, C. H. van Berkel, M. Rem, and R. W. J. J. Saeijs. VLSI Programming and Silicon Compilation; A Novel Approach from Philips Research. In *Proc of the IEEE International Conference on Computer Design: VLSI (ICCD-1988)*, pages 150–166, 1988.

[50] S. M. Nowick and M. Singh. High-Performance Asynchronous Pipelines: An Overview. *IEEE Design & Test of Computers*, 28(5):8–22, 2011.

[51] H. Park, A. He, M. Roncken, X. Song, and I. Sutherland. Modular Timing Constraints for Delay-Insensitive Systems. *Computer Science and Technology*, 31(1):77–106, 2016.

[52] N. C. Paver. *The Design and Implementation of an Asynchronous Microprocessor*. PhD thesis, Department of Computer Science, The University of Manchester, 1994.

[53] N. C. Paver, P. Day, C. Farnsworth, D. L. Jackson, W. A. Lien, and J. Liu. A Low-Power, Low-Noise, Configurable Self-Timed DSP. In *Proc of the Fourth International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC-1998)*, pages 32–42, 1998.

[54] A. Peeters, F. te Beest, M. de Wit, and W. Mallon. Click Elements: An Implementation Style for Data-Driven Compilation. In *Proc of the Sixteenth IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC-2010)*, pages 3–14, 2010.

[55] A. Peeters and K. van Berkel. Synchronous Handshake Circuits. In *Proc of the Seventh International Symposium on Asynchronous Circuits and Systems (ASYNC-2001)*, pages 86–95, 2001.

[56] Y. Peng, I. W. Jones, and M. R. Greenstreet. Finding Glitches Using Formal Methods. In *Proc of the Twenty Second IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC-2016)*, pages 45–46, 2016.

[57] M. Roncken, C. Cowan, B. Massey, S. M. Gilla, H. Park, R. Daasch, A. He, Y. Hei, W. Hunt Jr., X. Song, and I. Sutherland. Beyond Carrying Coal To Newcastle: Dual Citizen Circuits. In A. Mokhov, editor, *This*

Asynchronous World Essays dedicated to Alex Yakovlev on the occasion of his 60th birthday, pages 241–261. Newcastle University, 2016.

[58] M. Roncken, S. M. Gilla, H. Park, N. Jamadagni, C. Cowan, and I. Sutherland. Naturalized Communication and Testing. In *Proc of the Twenty First IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC-2015)*, pages 77–84, 2015.

[59] M. Roncken, I. Sutherland, C. Chen, Y. Hei, W. Hunt Jr., C. Chau, S. M. Gilla, H. Park, X. Song, A. He, and H. Chen. How to Think about Self-Timed Systems. In *Proc of the Fifty First IEEE Asilomar Conference on Signals, Systems, and Computers (Asilomar-2017)*, pages 1597–1604, 2017.

[60] C. L. Seitz. System Timing. In C. Mead and L. Conway, editors, *Introduction to VLSI Systems*, chapter 7, pages 218–262. Addison-Wesley, 1980.

[61] J. Simatic, A. Cherkaoui, F. Bertrand, R. P. Bastos, and L. Fesquet. A Practical Framework for Specification, Verification, and Design of Self-Timed Pipelines. In *Proc of the Twenty Third IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC-2017)*, pages 65–72, 2017.

[62] M. Singh and S. M. Nowick. MOUSETRAP: High-Speed Transition-Signaling Asynchronous Pipelines. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 15(6):684–698, 2007.

[63] A. Slobodova, J. Davis, S. Swords, and W. Hunt Jr. A Flexible Formal Verification Framework for Industrial Scale Validation. In *Proc of the Ninth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE-2011)*, pages 89–97, 2011.

[64] J. Sparso and S. Furber. *Principles of Asynchronous Circuit Design - A Systems Perspective*. Springer US, 2001.

[65] J. Sparso and J. Staunstrup. Delay-insensitive multi-ring structures. *INTEGRATION, the VLSI Journal*, 15(3):313–340, 1993.

[66] R. F. Sproull, I. E. Sutherland, and C. E. Molnar. The Counterflow Pipeline Processor Architecture. *IEEE Design & Test*, 11(3):48–59, 1994.

[67] S. K. Srinivasan and R. S. Katti. Desynchronization: Design for Verification. In *Proc of the Eleventh International Conference on Formal Methods in Computer-Aided Design (FMCAD-2011)*, pages 215–222, 2011.

[68] J. Staunstrup and M. R. Greenstreet. Designing Delay-Insensitive Circuits using Synchronized Transitions. In *IMEC IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, 1989.

[69] I. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, 1989.

[70] I. Sutherland and S. Fairbanks. GasP: A Minimal FIFO Control. In *Proc of the Seventh International Symposium on Asynchronous Circuits and Systems (ASYNC-2001)*, pages 46–53, 2001.

[71] K. van Berkel, R. Burgess, J. Kessels, A. Peeters, M. Roncken, F. Schalij, and R. van de Wiel. A Single-Rail Re-implementation of a DCC Error Detector Using a Generic Standard-Cell Library. In *Proc of the Second Working Conference on Asynchronous Design Methodologies*, pages 72–79, 1995.

[72] K. van Berkel, R. Burgess, J. Kessels, M. Roncken, F. Schalij, and A. Peeters. Asynchronous Circuits for Low Power: A DCC Error Corrector. *IEEE Design & Test of Computers*, 11(2):22–32, 1994.

[73] K. van Berkel, A. Peeters, and F. te Beest. Adding Synchronous and LSSD Modes to Asynchronous Circuits. In *Proc of the Eighth International Symposium on Asynchronous Circuits and Systems (ASYNC-2002)*, pages 161–170, 2002.

[74] F. Verbeek and J. Schmaltz. Verification of Building Blocks for Asynchronous Circuits. In *Proc of the Eleventh International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2013)*, pages 70–84, 2013.

[75] T. Verhoeff. Delay-insensitive codes - an overview. *Distributed Computing*, 3(1):1–8, 1988.

[76] V. M. Wijayasekara, S. K. Srinivasan, and S. C. Smith. Equivalence Verification for NULL Convention Logic (NCL) Circuits. In *Proc of the Thirty Second IEEE International Conference on Computer Design (ICCD-2014)*, pages 195–201, 2014.

[77] T. Williams, N. Patkar, and G. Shen. SPARC64: A 64-b 64-Active-Instruction out-of-Order-Execution MCM Processor. *IEEE Journal of Solid State Circuits*, 30(11):1215–1226, 1995.

[78] T. E. Williams and M. A. Horowitz. A Zero-Overhead Self-Timed 160-ns 54-b CMOS divider. *IEEE Journal of Solid State Circuits*, 26(11):1651–1661, 1991.