

HARDWARE VERIFICATION USING COINDUCTIVE ASSERTIONS

Paul S. Miner

Submitted to the faculty of the Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the Department of Computer Science
Indiana University

June 1998

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements of the degree of Doctor of Philosophy.

Doctoral
Committee

Prof. Steven D. Johnson
(Principal Advisor)

Prof. Daniel Leivant

Prof. Jonathan W. Mills

August 13, 1997

Prof. K. Jon Barwise

To Leah and Petra

Acknowledgements

I wish to thank my advisor, Steve Johnson, for his steady support and extraordinary patience throughout the course of this research. His guidance helped make this journey a rewarding experience.

I am grateful to Daniel Leivant, Jonathan Mills, and Jon Barwise for expanding my knowledge. Each of them provided glimpses of intriguing research opportunities that I was unable to explore. I regret that I did not interact more closely with them during the course of my studies.

I had several interesting discussions about streams and coinduction with Larry Moss and Jean-Yves Marion. Larry's presentations about Hypersets at the Logic Group Seminar first inspired me to think about streams coinductively. Jean-Yves provided valuable input for the initial development of the streams library in PVS.

The Hardware Methods Laboratory provided a stimulating research environment during my two years in Bloomington. Bhaskar Bose first introduced me to the design derivation approach. Kamlesh Rath and Esen Tuna were instrumental in getting me up to speed using the design derivation system. Kathi Fisler frequently provided insightful observations. And I worked on interesting projects with both Shyam Pullela and Bob Burger. In addition to the research interactions, I will always value my

friendship with them and their families.

I have benefited from many stimulating interactions with my colleagues at NASA. In particular, Jim Caldwell, Victor Carreño, and Ben DiVito all provided encouragement and valuable feedback during this endeavor. I owe a special thanks to Ricky Butler for allowing me to pursue this opportunity and to NASA for making it possible.

Finally, to my wife Leah who walked every step of this journey with me and my daughter Petra who learned to walk as this journey neared its end, thank you for keeping me mindful of the important things in life.

Abstract

Systems of mutually recursive stream equations provide a concise model of digital hardware designs. The formal design algebra presented by Johnson illustrates the utility of this representation for design derivation. The design derivation approach has been mechanized by Bose and applied to significant case studies. However, a design optimization sometimes depends upon the semantics of the implemented function. This cannot be captured within the design algebra. Design derivation needs to be augmented with additional verification strategies.

The stream equations employed in the design derivation approach are an instance of definition by *corecursion*. Therefore, the design derivation approach may be augmented using proof by *coinduction*. Coinduction is a useful strategy for proving that two sequential hardware designs have the same behavior. This enables safe substitution of an innovative optimization into a mature design. Furthermore, coinduction is useful for establishing that a signals within a design satisfy invariant properties. Such invariants, called *coinductive assertions*, are essential to ensure the validity of aggressive design optimizations.

This dissertation illustrates how design derivation is augmented using the Prototype Verification System (PVS). A PVS library to provide the basic declarations needed for reasoning about streams is developed. Stream equations are defined using

corecursion, and equality between streams is proven using coinduction. The combination of derivation-based and theorem prover-based verification is illustrated using two significant examples. First, a fault-tolerant clock synchronization circuit, suitable for use in critical applications, is developed using this combined approach. A clever optimization that reduces the chip area is verified using coinductive proof within PVS. The second example is the development of an architecture for floating-point division. A number of optimizations that reduce the cycle time of the inner loop of the algorithm are verified using the combined approach.

Contents

Acknowledgements	iv
Abstract	vi
1 Introduction	1
1.1 Contributions	6
1.2 Outline of the Presentation	7
2 Related Research	11
2.1 Overview of Design Derivation	11
2.2 Overview of PVS	13
2.3 Research in Hardware Verification	17
Theorem-Prover based System Verification	17
Algebraic Approaches	19

2.4	Coinduction and Corecursion	21
3	Reasoning about Streams	23
3.1	Stream Basics	23
3.2	Defining Streams in PVS	33
	Proof of stream equality using coinduction	38
	Invariants on Streams	39
	Coinductive Proofs using PVS	40
	Advanced examples of stream definition	48
4	Streams as a representation of Hardware	51
4.1	Simple Examples	55
	Factorial	55
	Fibonacci	57
4.2	Coinductive Proofs about Hardware	58
	Retiming Transformation	59
	Behavioral Abstraction	60
	Temporal Abstraction	62
5	Fault-Tolerant Synchronization	67

5.1	Top-Level Properties	68
5.2	The Verified Algorithm	68
5.3	Behavioral Specification	70
5.4	Initial Derivation	75
5.5	An Optimization	82
	Verification	86
5.6	Establishing Invariants	90
5.7	Another verification of the optimization	92
5.8	VLSI implementation	93
6	Floating Point Division	95
6.1	IEEE Floating-Point Arithmetic	96
6.2	The Verified Algorithm	96
6.3	Mapping the algorithm to a design	99
6.4	Partial Remainder Formation	103
	Quotient Digit Prediction	104
	Carry-Free Adders	110
6.5	Quotient Accumulation	114
	Redundant Representation	114

On-the-fly Quotient Accumulation	117
7 Conclusions	124
7.1 Limitations of the Approach	125
7.2 Suggestions for Future Work	126
7.3 Concluding Remarks	128

List of Figures

1.1	Overview of Verification Hierarchy	3
4.1	Standard model of Sequential System	52
4.2	Simple Retiming Transformation	60
4.3	Simple Counter Circuit	61
4.4	Stuttering Counter Circuit	63
4.5	Simple Counter Circuit	64
4.6	Fair Counter	65
5.1	ASM Chart for General Algorithm	71
5.2	Initial Architecture	74
5.3	Refined sub-circuit prior to ad hoc transformation	75
5.4	Refined sub-circuit after ad hoc transformation	83
5.5	Core Circuit for Computing Convergence Function	84

5.6	Optimized Convergence Function	85
5.7	Signal Assumptions	86
5.8	Counter Sub-Circuit	90
5.9	Core Synchronization Circuit	94
6.1	Initial Partial Remainder Sub-Circuit	102
6.2	Initial Quotient Accumulation Sub-Circuit	102
6.3	Derived Architecture	106
6.4	Final Partial Remainder Sub-Circuit	109
6.5	Partial Remainder Using a Carry-Propagate Adder	111
6.6	Redundant Partial Remainder Using a Carry-Free Adder	112
6.7	Redundant Representation of Accumulated Quotient	115

Introduction

Approaches to hardware development need to include an efficient means to represent and reason about changes in an evolving hardware design. As a hardware design matures, performance and area requirements evolve, also. The resulting changes to the design typically complicate verification activities. A verification environment must be flexible enough to accommodate a changing design.

Systems of mutually recursive stream equations provide a concise model of digital hardware designs. The formal design algebra presented in Johnson's Dissertation illustrates the utility of this representation for design derivation [Joh84]. The design derivation approach has been mechanized by Bose and applied to significant case studies [Bos94]. However, a design optimization sometimes depends upon the semantics of the implemented function. This cannot be captured within the design algebra. Design derivation needs to be augmented with additional verification strategies.

The representation of hardware via stream equations may be exploited. The stream equations employed in the design derivation approach are an instance of definition by *corecursion*. Therefore, the design derivation approach may be augmented

using proof by *coinduction*. Coinduction is a useful strategy for proving that two sequential hardware designs have the same behavior. This enables safe substitution of an innovative optimization into a mature design. Furthermore, coinduction is useful for establishing that a signals within a design satisfy invariant properties. Such invariants, called *coinductive assertions*, are essential to ensure the validity of aggressive design optimizations.

As technology improves it is possible to incorporate increasingly complicated functions in hardware. Analysis of the resulting designs is infeasible without mechanized support. In this research, I have used two tools: The Prototype Verification System (PVS [ORSv95]) and the Derivational Reasoning System (DRS [BTC96]). PVS is a general purpose mechanical theorem proving system; it provides a rich specification language and a powerful interactive theorem proving environment. DRS is an interactive formal design tool that allows a designer to derive a hardware design from an abstract behavioral level description. A design is produced by a series of transformations that ensure the final artifact is consistent with the initial behavioral description. That is, the design is *correct-by-construction*. Derivation-based formalisms provide a suitable framework for managing routine design refinements, but cannot be expected to cover the possible design space. General purpose theorem proving systems, on the other hand, provide sufficient generality to capture arbitrary design refinements, but can be cumbersome for the more routine aspects of design. Formal design environments should accommodate both approaches.

Design verification spans several levels from a mathematical statement of requirements to a detailed design. The verification hierarchy, annotated with choice of verification tool, is depicted in Figure 1.1. At the top level is a mathematical statement of properties that the design should satisfy. These requirements may be expressed in several different ways, depending upon the nature of the problem. Algorithms over

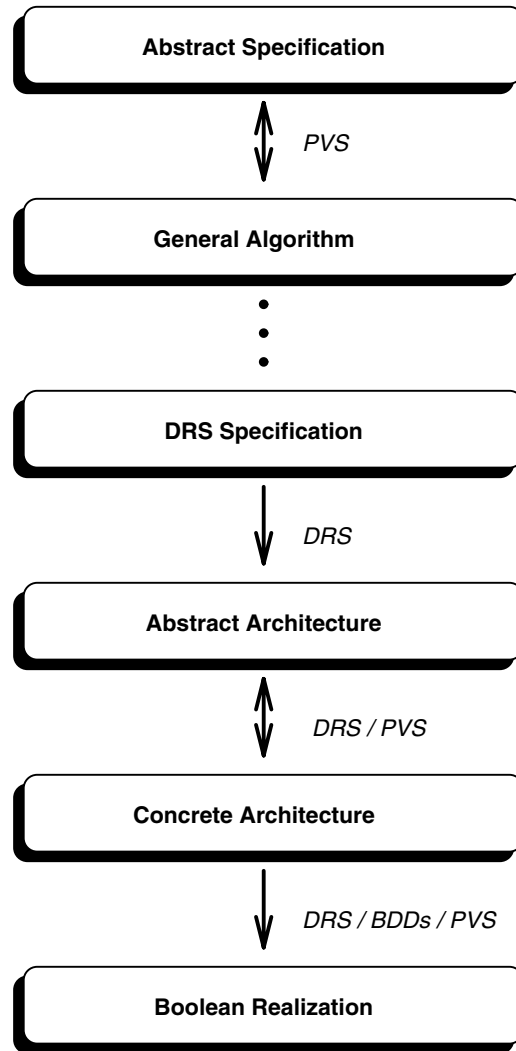


Figure 1.1: Overview of Verification Hierarchy

abstractly defined datatypes are proven to satisfy the top level requirements. Without knowledge of what the requirements may be, the only logical choice of tool for this verification activity is a general purpose theorem proving system such as PVS. General algorithms are transformed into behavioral level descriptions. In the figure, this relationship is depicted with ellipses because there is currently a semantic gap between the algorithm definition in PVS and a DRS behavioral level description. Within DRS, a behavioral description is refined until the designer is satisfied with the control structure of the design, then it is transformed into an initial architectural specification. At this stage, the architecture is refined to a concrete design. Most architectural refinements are performed within DRS. However, transformations that exploit the semantics of user defined functions may require external justification. These are verified using PVS. Finally, the abstract types in the design are instantiated with bit-vector representations. This may expose additional opportunities for refinement and optimization, so more structural transformations may be performed at this level. The resulting design is now entered into a commercial CAD tool for realization in hardware.

The case studies presented in Chapters 5 and 6 explore each level of refinement depicted in Figure 1.1. However, the primary emphasis is on the design phase between an abstract architectural description and a concrete architecture. At this stage, the principal design tool is DRS. The design is described by a system of mutually recursive stream equations. Algebraic transformations on this system of equations are used to refine the design to a target architecture. However, there are design optimizations that may not be effectively realized within a transformational system. In particular, there may be refinements that are only applicable within the context of a specific design. Such transformations cannot be justified on general principles. I

adopt the terminology of Saxe, et al. [SGGH93] and refer to these as *ad hoc* refinements. Justification of such refinements requires external verification. Thus, there is a need to augment the derivational approach with mechanical theorem proving support, thereby enabling effective formal development of efficient hardware.

In order to augment this stage of design with theorem proving support, we need to determine how to represent DRS circuit descriptions within PVS. There are two approaches. A deep embedding of DRS into PVS would consist of encoding the DRS syntax within PVS and defining a semantic interpretation. A shallow embedding consists of directly encoding DRS objects as PVS objects. Deep embeddings have the advantage of being able to support meta-level reasoning, but require working through a layer of interpretation. The intended application of this work is to use PVS as a verification engine supporting hardware design using DRS. A shallow embedding is a more direct way to achieve this goal.

The final problem is to determine how to represent DRS descriptions of sequential systems within PVS. There are several models of sequential systems in the mechanical theorem proving literature. Most use a representation similar to that advocated by Gordon [Gor86]. Signals are represented as functions from natural numbers to the signal type, hardware components are defined as relations between input and output, and hardware structure is captured by relation composition. This representation was considered for theorem-prover based justification of *ad hoc* refinements. However, within DRS, connections between combinational logic blocks are represented by function composition and the sequential aspect of the design is captured using stream equations. Feedback in the sequential circuit design is naturally represented by feedback in the stream equations. This is an instance of definition by corecursion [BM96]. Just as definition by recursion gives rise to proof by induction, the dual notion of corecursive definition gives rise to proof by coinduction. This proof technique simplifies

the verification of sequential hardware designs.

1.1 Contributions

The primary contribution of this research is the development and demonstration of a robust method for showing equivalence between two sequential circuits. The approach is based upon the observation that definition by corecursion is a natural representation for sequential hardware. This representation is especially useful in mapping an abstract behavioral level description to a register-transfer level (RTL) design. In addition, one can exploit this representation both to express and to verify properties of a design. Signals within the circuit are defined using corecursive stream equations. These signals are shown to satisfy invariant properties by coinduction. Such invariants are called coinductive assertions. Sub-circuits are shown to have equivalent behavior with respect to these coinductive assertions by exhibiting a suitable stream bisimulation.

Another contribution is the exploration of the heterogeneous reasoning required to support the application of complementary approaches to the design verification problem. Theorem proving and design derivation are just two of many tools available for design and verification. While it is possible to use a general purpose theorem proving system to support all stages of the design process, much of the design process consists of manipulation of large syntactic objects. These manipulations are routine, and do not require the deductive power of a general purpose theorem prover. In addition, most general purpose theorem provers do not handle large syntactic objects in a computationally efficient manner. Thus we need to address the interaction between design tools and general purpose verification tools.

In the work presented here, the design derivation tools represent sequential hardware using systems of stream equations. Reasoning about systems of stream equations within a general purpose theorem prover requires construction of an appropriate model. Within PVS, I have developed a theory of streams that allows definition by corecursion and proof by coinduction. This combination of design and verification tools is illustrated on two significant examples: a clock synchronization circuit and a floating-point division unit. In both cases, coinductive proofs are employed to verify circuit optimizations.

1.2 Outline of the Presentation

Chapter 2 provides an overview of the two mechanized reasoning systems employed in this research. The design derivation tool DRS provides a mechanized collection of transformations that enable a designer to develop a hardware design that is correct by construction. The mechanical theorem proving system PVS provides reasoning support for general results. It also enables exploration of verification strategies. The basic features of each tool are reviewed to provide a foundation for the discussion in the remainder of the dissertation. The chapter closes with a survey of related research.

An overview of streams, illustrated with some simple examples, is presented in Chapter 3. The primitive operations on streams are introduced and illustrated. A formulation of streams as a coinductive type in PVS is presented. The basic definitional forms are presented and coinductive proof techniques are justified. The development of a general strategy for automating coinductive proofs is illustrated using several examples from *Vicious Circles* [BM96].

Systems of stream equations provide a concise model of digital hardware. Chapter 4 illustrates the use of stream equations to represent sequential hardware design descriptions. A shallow embedding of DRS style stream equations in PVS is developed. The technique is illustrated with a few sample circuits. Chapter 4 concludes with an illustration of basic techniques for reasoning about digital hardware using coinduction. Several simple hardware verification problems are presented.

Chapter 5 presents the development of a fault-tolerant clock synchronization circuit. This development is a multi-level verification effort. It spans the spectrum from abstractly stated high-level properties to a VLSI implementation. The routine architectural refinements are performed within DRS. Two ad hoc refinements to the derived architecture are verified using coinductive proof in PVS. The first ad hoc refinement consists of a change in representation of the mechanism used to capture readings from remote clocks. An abstract storage device with a list-based representation is replaced by a pair of registers. The second refinement is a clever optimization to the core synchronization sub-circuit. Properties of the surrounding design make it possible to compute the same function using a more compact circuit. The chapter concludes with an illustration of the flexibility of the approach, by expanding the scope of the verification to explicitly include some of the surrounding design.

Chapter 6 illustrates a different class of optimization in the verification of a floating-point division unit. Performance is a key concern in the development of floating-point hardware. One way to improve performance in synchronous digital hardware design is to minimize the critical path, that is, reduce the number of combinational logic gate delays between any two clocked storage elements. There are a number of options at the designers disposal. One alternative is to re-time a design so that the gate delays are more uniformly distributed throughout the design [LS91]. Another technique is to introduce redundant representations for certain data items to

reduce the number of gate-delays required. Each optimization is illustrated within the design framework depicted in Figure 1.1. Coinductive verification provides a powerful technique for justifying these optimizations.

The focus of this dissertation is hardware design verification, that is, the goal is to establish that a design expressed in a *hardware description language* (HDL) is consistent with its specification. When design verification fails, as it did for the floating-point divide unit of the Intel Pentium(TM) microprocessor, the cost can be enormous. The bug in the Pentium cost Intel nearly \$500 million [Pra95]. Digital systems are increasingly being used for life-critical applications. A design error in a system controlling an aircraft or nuclear reactor can have dire consequences. The maiden flight of the Ariane-5 launch system ended prematurely in a spectacular fireworks display. The cause was attributed to a software design error in the inertial reference system [Ari96]. The cost of the payload lost in the mishap was estimated at approximately \$500 million [SD96]. Thus, two design errors account for a financial loss approaching one billion dollars.

We need robust techniques for design verification. Historically, the primary approach for design verification has been simulation. An executable model of the design is constructed, and this model is exercised with a series of test cases. However, even for designs of moderate complexity, it is infeasible to fully test the design with this approach. To address future design verification requirements, new techniques must be developed. In fact, Keutzer identifies design verification as *the most urgent problem in [integrated circuit (IC)] design* [Keu96].

The design derivation approach provides an exceptional framework for hardware design. The design space can be explored in a rigorous manner, secured by the knowledge that the design is correct by construction. Furthermore, the representation of hardware using corecursive stream equations allows for the augmentation of design

derivation with coinductive verification strategies. This provides an opportunity to verify aggressive optimizations that exploit the semantics of the implemented function.

2

Related Research

2.1 Overview of Design Derivation

The design derivation tools DDD [Bos91, Bos94] and DRS [BTC96] implement a formal design algebra for developing correct digital circuit descriptions. The designer interactively transforms high level behavioral specifications into a description suitable for entry into hardware synthesis tools. The top level describes the intended behavior of the circuit using a collection of mutually recursive function definitions in tail form. Each function corresponds to a control state; arguments to the functions represent the visible storage elements of the design. Transformations at this level allow the designer to modify the control structure of an architecture while preserving the functional correctness, relative to synchronization constraints. Once the control structure is determined, DRS automatically transforms the behavioral specification into an initial architectural level description. This description is refined using structural level transformations that allow the designer to reorganize the design to correspond to typical hardware design constructs. Once the architecture is refined,

the signals and operations are mapped to bit-vector representations. Depending upon the target technology (e.g. Field Programmable Gate-Array or custom VLSI), additional refinements are performed at this level to determine the physical organization of the design.

The design derivation tools represent the structure of a digital system using a collection of mutually recursive stream equations. A stream is an infinite sequence of uniformly typed values,

$$X = [x_0, x_1, x_2, \dots]$$

The stream constructor ‘*cs*’ adds an element to the front of the sequence

$$\mathbf{cs}(z, X) = [z, x_0, x_1, x_2, \dots]$$

Function ‘*cs*’ models a storage element. The first argument is the current value and the second is the input stream. Functions on values lift to sequences so that $f(X, Y)$ denotes $[f(x_0, y_0), f(x_1, y_1), \dots]$. DRS uses a system of equations to define a network of streams; recursion represents feedback in the circuit. For example, the following equation represents a loadable counter circuit.

$$COUNT = \mathbf{cs}(i, \mathbf{MUX}(S, L, \mathbf{INC}(COUNT))) \tag{2.1}$$

where i is the current integer value of the counter, *INC* and *MUX* are the increment and selection functions lifted to streams, and streams S and L are the multiplexor select signal and load input respectively. Within DRS, free variables in a system of stream equations must be bound by a system level abstraction. This abstraction defines the input signals for the circuit. The circuit’s output is a subset of the streams named in the system of equations.

Several significant hardware designs have been developed using DDD. In particular, Bose has formally derived the DDD-FM9001 [Bos94, BJ93] processor from Hunt's formal specification of the mechanically verified FM9001 microprocessor [Hun92]. Another significant derivation is the development of a Scheme Machine, which consists of a garbage-collecting memory system and a CPU that executes high-level Scheme primitives [Bur94]. During the above projects, it was recognized that a combination of design derivation with mechanized proof support would be superior to either approach individually [JWB89]. In the current study, hardware design using DRS is augmented using the PVS theorem proving system.

2.2 Overview of PVS

PVS is a general purpose specification and verification system developed at SRI International [ORS92, ORSv95]. The PVS specification language is based on higher-order logic. Specifications consist of collections of theories that may be parameterized by types and constants. Theory parameters may be constrained by conditions. PVS has predicate subtyping and dependent types. Since types may be constrained by an arbitrary predicate, typechecking is undecidable. When the typechecker cannot determine if an expression is correctly typed, it automatically generates proof obligations called Type Correctness Conditions (TCCs). PVS includes typing judgements to help reduce the frequency of TCCs. Included with PVS is the *prelude*, a collection of theories containing definitions and basic results.

There are two primitive types: `boolean` and `number`. The type `boolean` has the usual properties. The type `number` is different from the numeric types in other mechanical theorem proving systems. Properties of `number` are either declared in the

PVS prelude or hard-coded in the PVS typechecker and prover. The type `number` is declared as a nonempty, uninterpreted type. PVS recognizes all Arabic numerals as constants of type `number`. In addition, the fact that distinct numerals represent distinct numbers is hard-coded in the PVS prover. The type `real` is declared as a non-empty subtype of `number`. There are implicit typing judgements that assert that all Arabic numerals also have type `real`. The arithmetic operators and order relations are defined over the reals, and are known by the PVS decision procedures. Since PVS requires all functions to be total, the type of the division operator is constrained so that the divisor is nonzero. This uses the predicate subtyping mechanism. The type of nonzero reals is declared as follows:

```
nonzero_real: NONEMPTY_TYPE = {r: real | r /= 0} CONTAINING 1
```

The containing clause is not necessary, but it simplifies the proof of the TCC that results from this declaration. With the containing clause, this declaration generates the proof obligation: $\vdash 1 \neq 0$. This is known to the prover and is automatically discharged. If the containing clause were omitted, this declaration would generate the obligation: $\vdash \exists r. r \neq 0$. A proof of this goal requires user interaction. The usual properties of the real numbers are either expressed as axioms, or are coded in the linear arithmetic decision procedures built-in to the PVS prover. The field axioms are known by the decision procedures, as are the linear order axioms. The remaining axioms are declared in the prelude theory `real_axioms`.

The rationals are a nonempty, uninterpreted subtype of the reals (and by transitivity, also a subtype of `number`). Again, the fact that Arabic numerals have type `rational` is coded into PVS. Axioms in prelude theory `rationals` assert that the rationals are closed under addition, subtraction, multiplication, and division (with the divisor restricted to nonzero rationals). Similarly, the integers are defined as an

uninterpreted subtype of the rationals (again containing the Arabic numerals). Axioms in prelude theory `integers` assert that the integers are closed under addition, subtraction, and multiplication. Finally, the natural numbers are declared as the predicate subtype of nonnegative integers.

PVS includes the usual function type constructor, as well as constructors for tuple and record types. PVS also includes dependent forms of these. Dependent types have a number of uses within a verification system like PVS. For example, consider the floor function which truncates a real number x to the nearest integer $\leq x$. In the PVS prelude, this function is defined as follows:

```
floor(x:real): {i:int | i <= x & x < i + 1}
```

That is, `floor` is a constant with type `[x:real -> {i:int | i <= x & x < i + 1}]`. This type declaration uniquely defines the floor function. In PVS, this declaration spawns the proof obligation: $\vdash \forall x. \exists i. i \leq x < i + 1$. By encoding the desired behavior into the type, this information is available to the decision procedures. Many standard facts about the floor and ceiling functions are automatically discharged by the PVS decision procedures.

Dependent types are also quite useful in hardware verification, where many functions have regular structures that are parameterized by the width of the data word. Hanna was the first to advocate dependent types for this purpose [HD92]. PVS allows parameterized type declarations, inspired by the VERITAS⁺ system [HDL89]. The PVS prelude includes the following parameterized type declaration:

```
below(i): TYPE = {s: nat | s < i}
```

Using this type, we can declare a bit to have type `below(2)`. The type of n -bit words

can be declared as:

```
word(n) : TYPE = [below(n) -> bit]
```

Use of this parameterized type is illustrated by the following declaration for byte:

```
byte: TYPE = word(8)
```

PVS also has a mechanism for defining recursive abstract datatypes that are freely generated by a set of constructors [OS93]. The abstract data mechanism generates a theory which declares an uninterpreted type constrained by a number of axioms. These axioms correspond to the properties that would arise from the *least* solution to the recursive type equations. PVS does not yet have a corresponding mechanism for corecursive datatypes.

PVS provides an interactive theorem proving environment using a Gentzen style sequent calculus presentation of the proof goals. At the core of PVS's prover is a collection of decision procedures combined using a technique developed by Shostak [Sho78, Sho84, CLS96]. This eases some of the more burdensome aspects of interacting with a mechanical theorem prover. Most simple arithmetic results are discharged by invoking the decision procedures. The user rarely needs to direct the prover when faced with simple facts of arithmetic. PVS has a powerful conditional rewriter. There is a strategy language similar to LCF-style tactics. Thus, the user can define high-level proof procedures. There are several powerful strategies distributed with PVS that automatically verify a large number of results. For example, the built-in proof strategy (`grind`) is sufficient powerful to automatically verify many goals. The strategy (`induct-and-simplify`) is the brute force strategy for automating simple proofs by induction. PVS allows the user to prove lemmas in any order. It maintains a proof

dependency analysis to ensure that all obligations have been discharged. Included in the analysis is an enumeration of all axioms used by the proof chain.

There have been a number of exercises integrating various styles of reasoning within PVS. The most recent presentation of these efforts is reported in [ORS97].

2.3 Research in Hardware Verification

There are several competing approaches for formalizing the design process. At one end of the spectrum are general purpose mechanical theorem proving systems with expressive languages. Frequently, these systems require human interaction to produce machine checked proofs. At another extreme are a collection of fully automatic tools that can be applied to specialized verification problems. These tools include tautology checking using Binary Decision Diagrams [Bry86] and, more recently, model checking techniques for state machine verification [CES86, BCM⁺92, BCL⁺94].

Theorem-Prover based System Verification

There are several significant examples of theorem-prover based verifications in the literature. Examples include microprocessors [Hun92, SM95], dedicated hardware for fault-tolerance [BY90, SB91], and many efforts in floating-point verification [BKM96, CGZ96, RSS96, OLHA94, LO95, VCD94, Har95]. Theorem prover based verifications are not always robust in the face of changes. A small change in the design may invalidate much of the correctness proof [Wil97].

A notable effort is Hunt's verification of the FM9001 processor using *nqthm* [Hun92].

Hunt's verification strategy consists of a simulation model that animates designs expressed in a hardware description language. The verified design was subsequently fabricated in a VLSI implementation. The verification approach has subsequently been used successfully on commercial hardware devices [BKM96]. Another significant effort is the verification of the microcode of a set of instructions of the Rockwell-Collins AAMP5 microprocessor in PVS [SM95]. The AAMP5 is a modern microprocessor designed for aerospace applications. The verification activity uncovered some errors that had been seeded in the microcode to test the effectiveness of theorem proving techniques.

Prior to the design error in the Pentium processor, there were only a few activities focussed on design verification of floating-point hardware. The first efforts targeted verified implementations of binary non-restoring algorithms. Leeser, O'Leary, et al. present a verification, using *Nuprl*, of a binary non-restoring square root algorithm and its implementation [OLHA94, LO95]. Verkest, et al. present a similar verification, using *nqthm*, of a binary non-restoring division algorithm [VCD94]. In response to the flaw in the Pentium, several researchers investigated theorem prover based verifications of SRT division hardware. Clarke, German, and Zhao used the ANALYTICA theorem prover to verify Taylor's [Tay81] radix-4 SRT division circuit [CGZ96]. Their verification includes an abstract representation of the lookup table and a proof that it defines all necessary values for the quotient selection logic. Rueß, Srivas, and Shankar [RSS96] generalize this work using PVS. They present a general verification of arbitrary radix SRT division algorithms, instantiate their theory with Taylor's radix-4 SRT division circuit, and verify a description of the hardware. Included in their work is a technique to verify a concrete representation of the lookup table. Miner and Leathrum provide a further generalization to the class of subtractive division algorithms and relate this class of algorithms to a floating-point standard [ML96]. This

provides the entry point for the derivation activities presented in Chapter 6. Recently, the microcode for the floating point division and square root algorithms of the AMD5_K86TM microprocessor has been mechanically verified using the *ACL2* theorem prover [BKM96].

Within a general purpose theorem proving system, there are several ways to represent hardware artifacts. One of the most common is the relational style advocated by Gordon [Gor86]. Signals are represented as functions from discrete points in time to the signal type. Circuit components are defined as relations between signals; hardware structure is defined by relation composition. Bronstein and Talcott present a string-functional semantics for reasoning about hardware using *nqthm* [BT90]. This supports describing hardware structure using a functional style similar to that used in DRS. Paulin-Mohring has used a coinductive representation of streams in COQ to explore their application in hardware verification [Pau95]. She illustrated the approach using Gordon's multiplier circuit [Gor86]. She did not explore use of bisimulations for showing circuit equivalence, but suggested it as an interesting possibility. Coupet-Grimal and Jakubiec have exploited the constructive nature of COQ to extract circuits from proofs [CJ96]. In particular, they follow the example of Hanna, Daeche, and Longley's VERITAS⁺ system [HDL89] and use dependent types to construct parameterized circuit modules.

Algebraic Approaches

A number of algebraic approaches have been proposed for formal reasoning about digital hardware design. In addition to the design derivation tools discussed above, there is the Ruby design algebra [She88] and the work of Harman and Tucker [HT96]

Harman and Tucker present an algebraic model of synchronous digital hardware. The general hardware representation is by iterated maps. Let T represent time, A be set of states, W be a set of input values, $[T \rightarrow W]$ be a set of input streams, and X be a set of output values. A sequential system is modeled by function $F : T \times A \times [T \rightarrow W] \rightarrow A \times X$, where:

$$\begin{aligned} F_1(0, a, w) &= a. \\ F_1(t + 1, a, w) &= \text{next_state}(F_1(t, a, w), w(t)), \\ F_2(t, a, w) &= \text{out}(F_1(t, a, w)). \end{aligned}$$

When the set of states A is finite, this is a description of a Moore machine [Moo56]. It provides a rigorous mathematical description of synchronous hardware, and lends itself to inductive verification. However, this representation has the complication that the current state is represented as a function of the initial state and past history of the input stream. Frequently, the initial state of a hardware device is irrelevant. Also, the current state of a device only depends upon a limited past history of the input stream. My experiences in proving hardware using this representation is that it distracts from the main verification goal. This structure is similar to that used in the PVS proofs of the clock synchronization circuit presented in [MPJ94]. In that verification effort, more time was spent dealing with quirks of the model than in verifying the refinement. The verification presented in [MPJ94] has been redone using coinduction and is presented in Chapter 5.

Ruby is a relational algebra for digital hardware design. Circuit components are defined as relations between signals; systems are built up via relational composition. The primary difference with other representations is that time is infinite in both directions, eliminating the distraction of an irrelevant initial state. The language was introduced by Jones and Sheeran [JS90], and is particularly well suited to refining

circuits with a regular structure. The Ruby design philosophy is similar to the underlying design philosophy for DRS, in that it supports a transformational approach to circuit design. It has been demonstrated on a number of circuit designs, including a pipelined systolic correlator [She88]. Recently, Sharp and Rasmussen developed a strongly typed version called T-Ruby [SR97]. It is based upon Pure Ruby [Ros90], which consists of four basic constructs. In T-Ruby, all circuits and combinators are built up from the pure core. Rasmussen has developed a proof system called RubyZF using the Isabelle theorem prover [Ras96]. RubyZF enables mechanical verification of all T-Ruby transformations, and provides integrated reasoning support for verifying proof obligations generated by T-Ruby.

The synchronous data-flow language LUSTRE has been used effectively in the design and verification of synchronous hardware [HCRP91, TP90]. The language was originally developed for the specification and verification of real-time systems and was inspired by the dataflow language Lucid [WA85]. Lustre is a declarative language where expressions represent sequences of values. Systems are specified using a collection of equations. The verification tool Lesar provides a state exploration technique similar to model checking.

2.4 Coinduction and Corecursion

This dissertation does not address foundational issues of coinduction or corecursion, but rather focuses on a specific application. There are a number of foundational treatments in the literature. One such basis is Aczel's Special Final Coalgebra Theorem [Acz88]. Another is Barwise and Moss' Corecursion Theorem [BM96]. Recently, Moss and Danner have further explored foundational issues and have suggested that

corecursion may be justified by reduction to recursion [MD97]. Research in constructive type theory has also addressed the notion of types of infinite objects. The first treatments were by Mendler et al. [MPC86, Men87], with subsequent investigations by Coquand [Coq93] and Leclerc and Paulin-Mohring [LP93].

Applications of coinduction to verification have their root in process calculi such as CCS [Mil89]. Park appears to be among the first to explore the use of greatest fixed points as a verification strategy [Par80, Par81]. Other applications include programming language semantics [MT91, Pit94]. Giménez used coinductive types in COQ for the verification of an alternating bit protocol [Gim95]. Recently, Jacobs has explored coinductive verification techniques for class refinement in an object-oriented setting [Jac97].

There have been a number of recent efforts seeking to provide mechanized reasoning support for coinductive verification techniques. Paulson built up a framework for inductive and coinductive datatypes in Isabelle-HOL [Pau97]. Inductive types are expressed as least fixed points and coinductive types are expressed as greatest fixed points. Leclerc and Paulin-Mohring have explored coinductive types using the COQ system [LP93]. Recently, Hensel and Jacobs have provided mechanized support for coinductive reasoning in PVS by developing a theory of final coalgebras [HJ97a, HJ97b].

3

Reasoning about Streams

Many approaches to formal verification of hardware represent signals as streams, i.e. functions from time to value, where time ranges over the natural numbers. These previous verification activities have not exploited the fact that streams are a coinductively generated type and hence also allow proof by coinduction. This chapter introduces some basic properties of streams, and illustrates the use of the mechanized reasoning system PVS to reason about them.

3.1 Stream Basics

A *stream* over A is an infinite sequence

$$x = [x_0, x_1, x_2, \dots]$$

where each $x_i \in A$. The constructor \mathbf{cs} adds an element to the front of the sequence. Take $a \in A$ and x as above, then

$$\mathbf{cs}(a, x) = [a, x_0, x_1, x_2, \dots]$$

The natural model for streams over A is $\mathbf{N} \rightarrow A$, with the following definition for \mathbf{cs} :

$$\mathbf{cs}(a, x)(n) = \begin{cases} a & \text{if } n = 0 \\ x(n-1) & \text{otherwise} \end{cases}$$

Streams may also be viewed coinductively, that is, as a greatest fixed-point of a monotone operator. An operator $F : \mathcal{P}(\mathcal{U}) \rightarrow \mathcal{P}(\mathcal{U})$ is monotone if $F(X) \subseteq F(Y)$ whenever $X \subseteq Y$. The greatest fixed-point of monotone F , defined by

$$\nu Z.F(Z) = \bigcup \{Z \mid Z \subseteq F(Z)\}$$

is the largest set Z such that $Z = F(Z)$.

Fix a class \mathcal{U} and $\mathbf{cs} : A \times \mathcal{U} \rightarrow \mathcal{U}$ for some set A . Define monotone operator $\mathbf{CS}_A : \mathcal{P}(\mathcal{U}) \rightarrow \mathcal{P}(\mathcal{U})$ by

$$\mathbf{CS}_A(X) = \{\mathbf{cs}(a, x) \mid a \in A \ \& \ x \in X\}$$

The set A^∞ of streams over A is defined as the greatest fixed-point of \mathbf{CS}_A .

$$\begin{aligned} A^\infty &\hat{=} \nu Z.\mathbf{CS}_A(Z) \\ &= \{\mathbf{cs}(a, x) \mid a \in A \ \& \ x \in A^\infty\} \end{aligned}$$

The constructor \mathbf{cs} and universe \mathcal{U} must be selected so that whenever $A \neq \emptyset$ then also $A^\infty \neq \emptyset$ and furthermore, there exist functions $\mathbf{hd} : A^\infty \rightarrow A$ and $\mathbf{tl} : A^\infty \rightarrow A^\infty$

so that for $a \in A$ and $x \in A^\infty$:

$$\begin{aligned} \text{hd}(\text{cs}(a, x)) &= a \\ \text{tl}(\text{cs}(a, x)) &= x \\ \text{cs}(\text{hd}(x), \text{tl}(x)) &= x \end{aligned}$$

Using the standard model of streams, that is, setting $\mathcal{U} = \mathbf{N} \rightarrow A$, defining cs as above, and defining $\text{hd}(x) = x(0)$ and $\text{tl}(x) = \lambda n. x(n + 1)$, these equations are easily verified. An elegant model of streams is found in the universe of non-well-founded sets, where it is possible to define $\text{cs}(a, x) = \langle a, x \rangle$ [BM96].

A trivial consequence of the fact that $A^\infty = \bigcup \{Z \mid Z \subseteq \text{CS}_A(Z)\}$ is the following proof principle:

Coinduction Principle: To show that $x \in A^\infty$, exhibit a Z such that $x \in Z$ and $Z \subseteq \text{CS}_A(Z)$.

We use the coinduction principle to build towards a general mechanism for defining streams.

Theorem 1 For $g : A \rightarrow A$ and $a \in A$, if

$$\text{iter}_g(a) = \text{cs}(a, \text{iter}_g(g(a))) \tag{3.1}$$

then $\text{iter}_g(a) \in A^\infty$

Proof: Let

$$Z = \{\text{iter}_g(a) \mid a \in A\}$$

We need to show that $Z \subseteq \text{CS}_A(Z)$. Given $a' \in A$, $\text{iter}_g(a') \in Z$. From equation (3.1), $\text{iter}_g(a') = \text{cs}(a', \text{iter}_g(g(a')))$. Since $a' \in A$ and $\text{iter}_g(g(a')) \in Z$, we have $\text{iter}_g(a') \in \text{CS}_A(Z)$. Since the choice of a' was arbitrary, $Z \subseteq \text{CS}_A(Z)$.

Since for any $a \in A$, $\text{iter}_g(a) \in Z$ and $Z \subseteq \text{CS}_A(Z)$, we use the coinduction principle to conclude $\text{iter}_g(a) \in A^\infty$. \dashv

The next question is whether a solution to equation 3.1 is unique. This requires a principle of equality. Two streams are equal if and only if they are indistinguishable at every finitely accessible point. The following proof principle provides a useful tool for demonstrating equality between streams.

Definition A *stream bisimulation* is a relation R between A^∞ and A^∞ so that whenever $s R t$, then

$$\text{hd}(s) = \text{hd}(t) \text{ and } \text{tl}(s) R \text{tl}(t)$$

Coinduction principle If $x, y \in A^\infty$ and $x R y$ for some bisimulation R , then

$$x = y.$$

A proof of this principle in the universe of non-well-founded sets is on page 84 of [BM96]. Jacobs and Rutten use the fact that streams are a final coalgebra to justify this principle [JR97]. Here we directly show that two streams in a stream bisimulation are indistinguishable.

Theorem 2 For $n \in \mathbf{N}$ if $x, y \in A^\infty$ and $x R y$ for some bisimulation R , then $\text{hd}(\text{tl}^n(x)) = \text{hd}(\text{tl}^n(y))$.

Proof: By induction on n .

Base case: Since R is a bisimulation and $x R y$, we have

$$\text{hd}(\text{tl}^0(x)) = \text{hd}(x) = \text{hd}(y) = \text{hd}(\text{tl}^0(y))$$

Induction step: The induction hypothesis asserts that if $x', y' \in A^\infty$ and $x' R y'$ then $\text{hd}(\text{tl}^n(x')) = \text{hd}(\text{tl}^n(y'))$. Since R is a bisimulation and $x R y$, we have $\text{tl}(x) R \text{tl}(y)$. Using the induction hypothesis, we get $\text{hd}(\text{tl}^n(\text{tl}(x))) = \text{hd}(\text{tl}^n(\text{tl}(y)))$. Simplification yields $\text{hd}(\text{tl}^{n+1}(x)) = \text{hd}(\text{tl}^{n+1}(y))$ \dashv

Corollary 1 For any stream bisimulation R ,

$$R \subseteq \{(x, y) \mid \forall (n \in \mathbf{N}). \text{hd}(\text{tl}^n(x)) = \text{hd}(\text{tl}^n(y))\}$$

If we define a monotone operator on $\mathcal{U} = A^\infty \times A^\infty$ by:

$$F(R) = \{(x, y) \mid \text{hd}(x) = \text{hd}(y) \text{ and } \text{tl}(x) R \text{tl}(y)\}$$

then

$$\nu R.F(R) = \{(x, y) \mid \forall (n \in \mathbf{N}). \text{hd}(\text{tl}^n(x)) = \text{hd}(\text{tl}^n(y))\}$$

Now we can use the coinduction principle to show that equation 3.1 has a unique solution.

Theorem 3 Suppose $I(a) = \text{cs}(a, I(g(a)))$, then $I(a) = \text{iter}_g(a)$

Proof: By coinduction using relation

$$R = \{(I(a), \text{iter}_g(a)) \mid a \in A\}$$

It is clear that $I(a) R \text{iter}_g(a)$ for all $a \in A$. All that remains is to show that R is a stream bisimulation.

Take $x, y \in A^\infty$ such that $x R y$. That is, there is an $a \in A$ such that $x = I(a)$ and $y = \text{iter}_g(a)$. There are two properties to be shown

1. $\text{hd}(x) = \text{hd}(I(a)) = a = \text{hd}(\text{iter}_g(a)) = \text{hd}(y)$
2. Using equations (3.1) and the hypothesis above, we conclude that

$$\text{tl}(x) = \text{tl}(I(a)) = \text{tl}(\text{cs}(a, I(g(a)))) = I(g(a))$$

and

$$\text{tl}(y) = \text{tl}(\text{iter}_g(a)) = \text{tl}(\text{cs}(a, \text{iter}_g(g(a)))) = \text{iter}_g(g(a))$$

Since $g(a) \in A$, $\text{tl}(x) R \text{tl}(y)$. +

Thus, equation (3.1) uniquely determines a stream for each $a \in A$ and any function $g : A \rightarrow A$. `lter` provides a mechanism for defining several streams over A . The next requirement is a functional to map a function $f : A \rightarrow B$ to every element of a stream over A .

Theorem 4 For $s \in A^\infty$ and $f : A \rightarrow B$, if

$$\text{map}_f(s) = \text{cs}(f(\text{hd}(s)), \text{map}_f(\text{tl}(s))) \tag{3.2}$$

then $\text{map}_f(s) \in B^\infty$.

Proof: Let

$$Z = \{\mathbf{map}_f(s) \mid s \in A^\infty\}$$

We need to show that $Z \subseteq \mathbf{CS}_B(Z)$. Given any $s \in A^\infty$, $\mathbf{map}_f(s) \in Z$. We know that since $s \in A^\infty$ there are $a \in A$ and $s' \in A^\infty$ such that $s = \mathbf{cs}(a, s')$. By the definition of \mathbf{map} we have $\mathbf{map}_f(s) = \mathbf{cs}(f(a), \mathbf{map}_f(s'))$. This is an element of $\mathbf{CS}_B(Z)$, so $Z \subseteq \mathbf{CS}_B(Z)$.

Since $\mathbf{map}_f(s) \in Z$ and $Z \subseteq \mathbf{CS}_B(Z)$, we use the coinduction principle to conclude $\mathbf{map}_f(s) \in B^\infty$. +

A coinductive proof suffices to show that \mathbf{map} is unique.

Theorem 5 For $s \in A^\infty$, given $f : A \rightarrow B$, if $m : A^\infty \rightarrow B^\infty$ satisfies

$$m(s) = \mathbf{cs}(f(\mathbf{hd}(s)), m(\mathbf{tl}(s)))$$

then $m(s) = \mathbf{map}_f(s)$

Proof: By coinduction using relation

$$R = \{(m(s), \mathbf{map}_f(s)) \mid s \in A^\infty\}$$

It is clear that $m(s) R \mathbf{map}_f(s)$ for all $s \in A^\infty$. All that remains is to show that R is a stream bisimulation.

Take $x, y \in A^\infty$ such that $x R y$. That is, there is an $s \in A^\infty$ such that $x = m(s)$ and $y = \mathbf{map}_f(s)$. There are two properties to be shown

1. $\mathbf{hd}(x) = \mathbf{hd}(m(s)) = f(\mathbf{hd}(s)) = \mathbf{hd}(\mathbf{map}_f(s)) = \mathbf{hd}(y)$

2. Using equations (3.2) and the hypothesis above, we conclude that

$$\text{tl}(x) = \text{tl}(m(s)) = \text{tl}(\text{cs}(f(\text{hd}(s)), m(\text{tl}(s)))) = m(\text{tl}(s))$$

and

$$\text{tl}(y) = \text{tl}(\text{map}_f(s)) = \text{tl}(\text{cs}(f(\text{hd}(s)), \text{map}_f(\text{tl}(s)))) = \text{map}_f(\text{tl}(s))$$

Since $\text{tl}(s) \in A^\infty$, $\text{tl}(x) R \text{tl}(y)$. +

Given $f : A \rightarrow B$, $g : A \rightarrow A$. Consider the function $\text{corec}_{(f,g)} : A \rightarrow B^\infty$ defined by

$$\text{corec}_{(f,g)}(a) \hat{=} \text{map}_f(\text{iter}_g(a)) \tag{3.3}$$

This function provides a useful tool for providing solutions to corecursive stream equations.

Theorem 6

$$\text{corec}_{(f,g)}(a) = \text{cs}(f(a), \text{corec}_{(f,g)}(g(a))) \tag{3.4}$$

Proof:

$$\begin{aligned} \text{corec}_{(f,g)}(a) &= \text{map}_f(\text{iter}_g(a)) && \text{; by equation (3.3)} \\ &= \text{map}_f(\text{cs}(a, \text{iter}_g(g(a)))) && \text{; by equation (3.1)} \\ &= \text{cs}(f(a), \text{map}_f(\text{iter}_g(g(a)))) && \text{; by equation (3.2)} \\ &= \text{cs}(f(a), \text{corec}_{(f,g)}(g(a))) && \text{; by equation (3.3)} \end{aligned}$$

+

Suppose we want to find a solution for c_a in the equation:

$$c_a = \mathbf{cs}(a, c_a) \tag{3.5}$$

We can use **corec**. Assume that $A = B$, then define

$$c_a \hat{=} \mathbf{corec}_{(id_A, id_A)}(a) \tag{3.6}$$

Lemma 1 $c_a = \mathbf{cs}(a, c_a)$

Proof:

$$\begin{aligned} c_a &= \mathbf{corec}_{(id_A, id_A)}(a) && \text{; by equation (3.6)} \\ &= \mathbf{cs}(id_A(a), \mathbf{corec}_{(id_A, id_A)}(id_A(a))) && \text{; by equation (3.4)} \\ &= \mathbf{cs}(a, \mathbf{corec}_{(id_A, id_A)}(a)) && \text{; definition of } id_A \\ &= \mathbf{cs}(a, c_a) && \text{; by equation (3.6)} \end{aligned}$$

⊢

Corec may be used to define a solution to equation 3.1

Fact 1 $\mathbf{iter}_g(a) = \mathbf{corec}_{(id, g)}(a)$

Proof: By coinduction, using bisimulation $\{(\mathbf{iter}_g(a), \mathbf{corec}_{(id, g)}(a)) \mid a \in A\}$. ⊢

Inspection of the above bisimulation shows that it is defined in a simple manner from the equality to be proven. When a bisimulation is of this form, it is trivial to show that the streams being shown equal are in the relation. This technique for constructing bisimulations is quite common in coinductive proofs. A bisimulation constructed in

this manner is a *trivial bisimulation*. The proof steps are mechanical and can be automated.

As an additional example of verifying stream equivalence using coinduction, consider the following identity. This is frequently given as an alternative definition for `iter`.

Fact 2

$$\text{iter}_g(a) = \text{cs}(a, \text{map}_g(\text{iter}_g(a))) \quad (3.7)$$

Proof: By coinduction using bisimulation $\{(\text{iter}_g(a), \text{cs}(a, \text{map}_g(\text{iter}_g(a)))) \mid a \in A\} \dashv$

Barwise & Moss introduce a few other stream equations and some exercises for practicing coinductive proofs. These will be used as illustrative test cases for the PVS streams theories presented in the next section. The definitions and identities are:

$$\text{map}_f(c_a) = c_{f(a)} \quad (3.8)$$

$$\text{zip}(\text{cs}(a, s), \text{cs}(b, t)) \hat{=} \text{cs}(a, \text{cs}(b, \text{zip}(s, t))) \quad (3.9)$$

$$\text{zip}(c_a, c_a) = c_a \quad (3.10)$$

$$\text{iter}_f(a) = \text{zip}(\text{iter}_{f \circ f}(a), \text{map}_f(\text{iter}_{f \circ f}(a))) \quad (3.11)$$

$$\text{dmap}_f(s) \hat{=} \text{cs}(f(\text{hd}(s)), \text{dmap}_f(\text{dmap}_f(\text{tl}(s)))) \quad (3.12)$$

recalling that `iterg(a)`, `mapf(s)`, and `ca` satisfy equations (3.1), (3.2), and (3.5), respectively.

3.2 Defining Streams in PVS

Developing support for mechanized reasoning about streams is complicated by the fact that it must be developed within the logic of some mechanized proof system. Traditionally, streams over A have been represented by $[\mathbf{N} \rightarrow A]$. The initial efforts to reason about streams in PVS [MPJ94] used this representation. However, this choice of representation led to emphasis on proofs by induction, even though we were modeling objects that were defined (within DDD) using stream equations. As shown in the previous section, corecursive stream equations naturally lead to proofs using coinduction.

The PVS streams library is modeled after the PVS abstract datatype mechanism [OS93]. The principal definition of the stream codatatype is parameterized by nonempty type α . The type of streams over α is defined as an uninterpreted nonempty type. As in the PVS abstract datatype mechanism, the type is constrained by axioms about uninterpreted constructor and accessor functions. The PVS abstract datatype mechanism supports recursive function definition and proofs by induction. Similarly, the streams library provides support for corecursive function definition and proofs by coinduction. The streams library is available electronically [WWW].

The following initial fragment from PVS theory `Stream_cdt` provides the declaration of type `Stream[alpha]`, and defines the type signature for `Stream` constructor `cs` and accessors `hd` and `tl`.

```
Stream_cdt[alpha: TYPE+]: THEORY
  BEGIN

  Stream: TYPE+

  cs: [alpha, Stream -> Stream]
  hd: [Stream -> alpha]
```

```
t1: [Stream -> Stream]
```

The constructor `cs` and accessors `hd` and `t1` are constrained by the usual axioms. In the following axioms, `S` has type `Stream[alpha]` and `a` has type `alpha`.

```
Stream_cs_eta : AXIOM cs(hd(S), t1(S)) = S
```

```
hd_cs          : AXIOM hd(cs(a, S)) = a
```

```
t1_cs          : AXIOM t1(cs(a, S)) = S
```

Since type `Stream` is uninterpreted, there is no general mechanism in PVS for proving equality between streams. We need to introduce an axiom that defines what we mean by stream equality. Axiom `Stream_eq` defines two streams be equal if and only if they are indistinguishable at every finitely accessible point.

```
nth((S : Stream), (n : nat)) : alpha = hd(iterate(t1,n)(S))
```

```
Stream_eq      : AXIOM (X = Y) <=>
                  FORALL n: nth(X, n) = nth(Y, n)
```

Function `iterate`, used in the definition of function `nth`, is defined in the PVS prelude. For any type T , $x : T$, and function $g : T \rightarrow T$,

$$\text{iterate}(g, n)(x) = g^n(x).$$

Thus, function `nth` returns the n^{th} element of a stream.

The final base definition for the PVS Streams library is the corecursion combinator:

$$\text{corec} : ((\alpha \rightarrow \beta) \times (\alpha \rightarrow \alpha)) \rightarrow (\alpha \rightarrow \beta^\infty)$$

Given $f : \alpha \rightarrow \beta$, $g : \alpha \rightarrow \alpha$ and $a : \alpha$, we axiomatize the behavior of the corecursion combinator via the following PVS declaration:

```
corec_def: AXIOM
  corec(f, g)(a) = cs(f(a), corec(f, g)(g(a)))
```

Given this definition, and the previous axioms, we prove the following two facts. The proofs consist of rewriting with `corec_def` and then rewriting with either `hd_cs` or `tl_cs`.

```
hd_corec: FACT
  hd(corec(f, g)(a)) = f(a)

tl_corec: FACT
  tl(corec(f, g)(a)) = corec(f, g)(g(a))
```

For each function $F : \alpha \rightarrow \beta^\infty$ that is defined using `corec`, I adopt the convention that the following three facts are also introduced and proven within PVS:

```
F_def: FACT
  F(a) = cs(f(a), F(g(a)))

hd_F: FACT
  hd(F(a)) = f(a)

tl_F: FACT
  tl(F(a)) = F(g(a))
```

These proof obligations are easily discharged. With the exception of `F_def`, no theorems about `F` are proven directly from the corecursive definition. Establishing these three facts about streams defined using `corec` greatly simplifies subsequent coinductive proofs.

The functions f and g used in corecursive definitions are often complicated. If we allow function definitions using `corec` to be expanded by some of the powerful proof strategies within PVS, it is possible that much of the proof time could be consumed by trivial details. Thus, we desire some mechanism to *hide* the corecursion combinator from the standard automatic proof strategies within PVS. The following definitional scheme suffices:

$$F(a): \{B : \text{Stream}[\text{beta}] \mid B = \text{corec}(f, g)(a)\}$$

This style of definition exploits the PVS dependent type mechanism. In this declaration, function $F : \alpha \rightarrow \beta^\infty$ is both uninterpreted and completely defined. Since there is no body, the function cannot be expanded or rewritten. The only access to the definition of this function is via the PVS proof command (`typepred "F(a)"`), for some `a:alpha`. The equation `F_def` is proven using the `typepred` command followed by rewriting with `corec_def`.

As an illustration of this template, here is the corecursive definition of `map`:

```
map((f : [alpha-> beta]), (S : Stream[alpha]))
  : {B : Stream[beta] |
      B = corec((lambda S : f(hd(S))),
                (lambda S : tl(S)) )(S)}

map_def : FACT
  map(f, S) = cs(f(hd(S)), map(f, tl(S)))

hd_map  : FACT
  hd(map(f, S)) = f(hd(S))

tl_map  : FACT
  tl(map(f, S)) = map(f, tl(S))
```

Facts `hd_map` and `tl_map` are trivial consequences of `map_def`, `hd_cs`, and `tl_cs`. The

only remaining obligation is the proof of `map_def`. It follows from the dependent type constraints and `corec_def`.

The proofs justifying definition by corecursion have the same structure. To simplify the handling of these proof obligations, I have defined a PVS strategy to discharge goals of this form. Consider the corecursive definition of `iter`:

```
iter((g : [alpha -> alpha]), (a : alpha)) :
  { A : Stream[alpha] | A = corec(id[alpha], g)(a) }

iter_def : FACT
  iter(g, a) = cs(a, iter(g, g(a)))
```

The PVS justification of `iter_def` uses the proof strategy (`def-corec`):

Proof: (Taken verbatim from the `*pvs*` proof buffer)

```
iter_def :

  |-----
  {1}   (FORALL (a: alpha, g: [alpha -> alpha]):
        iter(g, a) = cs(a, iter(g, g(a))))

Rule? (def-corec)
id rewrites id[alpha](a!1)
  to a!1
Justifying CoRecursive definition,
Q.E.D.
```

In addition, we define the appropriate supporting facts concerning the head and tail of `iter`.

```
hd_iter : FACT
  hd(iter(g, a)) = a

tl_iter : FACT
  tl(iter(g, a)) = iter(g, g(a))
```

These facts are also simply discharged within PVS. This template is followed for all stream definitions within the PVS streams library.

For a more complicated example of stream definition using the corecursion combinator, consider equation (3.9) from section 3.1. The challenge (from Barwise and Moss [BM96]) is to show that there is a function $\text{zip}: A^\infty \times A^\infty \rightarrow A^\infty$ that satisfies the equation:

$$\text{zip}(X,Y) = \text{cs}(\text{hd}(X), \text{cs}(\text{hd}(Y), \text{zip}(\text{tl}(X), \text{tl}(Y))))$$

At first glance, it does not appear that zip can be defined using corec . However, one of the laws for zip is

$$\text{zip}(X,Y) = \text{cs}(\text{hd}(X), \text{zip}(Y, \text{tl}(X)))$$

This leads us to a correct definition of zip using corec

$$\text{zip}(X,Y) : \{Z \mid Z = \text{corec}(\text{lambda } X,Y: \text{hd}(X), \\ \text{lambda } X,Y: (Y, \text{tl}(X)))(X,Y)\}$$

From this, we easily establish that zip satisfies equation (3.9).

Proof of stream equality using coinduction

Using axiom `Stream_eq` (page 34), we can prove stream equality using function `nth` and induction. However, section 3.1 demonstrated proofs of stream equality using the coinduction principle.

The PVS definition of the coinduction principle for proving stream equality is given in theorem `co_induct` below:

```

Bisimulation: TYPE =
  {R : PRED[[Stream[alpha], Stream[alpha]]] |
    FORALL X, Y:
      R(X, Y) =>
        hd(X) = hd(Y) & R(tl(X), tl(Y))}

co_induct: THEOREM
  (EXISTS (R: Bisimulation): R(X, Y)) => X = Y

```

Theorem `co_induct` is a trivial consequence of Axiom `Stream_eq` and Theorem 2. It has be proven using PVS. The PVS theories are available electronically [WWW].

Invariants on Streams

Proof by coinduction is also used to establish invariant properties of a stream. Relations are lifted to streams, yielding streams over the booleans. As an example, suppose we are given two boolean streams A and B . We lift implication to these streams yielding a boolean valued stream satisfying the equation

$$(A \Rightarrow B) = \text{cs}(\text{hd}(A) \Rightarrow \text{hd}(B), \text{tl}(A) \Rightarrow \text{tl}(B))$$

In this example, the symbol ' \Rightarrow ' is overloaded. In sub-term $\text{hd}(A) \Rightarrow \text{hd}(B)$ it has (PVS) type `[bool, bool -> bool]` and has type `[Stream[bool], Stream[bool] -> Stream[bool]]` in the other occurrences.

One way to make the assertion that $\forall n : n^{\text{th}}(A, n) \Rightarrow n^{\text{th}}(B, n)$ is to identify stream $(A \Rightarrow B)$ with stream `const(true)`. If this identity is true, it is easily proven using the coinduction principle for showing stream equality. This identification is sufficiently useful that we have defined the abbreviation:

```
Invariant(A : Stream[bool]) : bool = (A = const(true))
```

With this construct we define a coinduction principle for verifying that streams satisfy invariant properties.

```

CoInductive_Assertion: TYPE =
  { (P : PRED[Stream[bool]]) |
    FORALL A: P(A) => hd(A) & P(tl(A)) }

co_induct: THEOREM
  (EXISTS (P : CoInductive_Assertion): P(A)) => Invariant(A)

```

This is a simple consequence of the coinduction principle for stream equality. Alternatively, one could define:

```

Invariant(A : Stream[bool]) : bool = FORALL (n:nat): nth(A,n)

```

and then prove this coinduction principle using the same proof structure as shown for the coinduction principle for stream equality.

Coinductive Proofs using PVS

In this section we revisit some of the stream equations given in section 3.1 and illustrate approaches for coinductive proofs using PVS. First, consider equation (3.7).

Fact 3 $\text{iter}(g, a) = \text{cs}(a, \text{map}(g, \text{iter}(g, a)))$

Proof: The initial PVS step strips off the universal quantifiers.

```

iter_map_def :
  |-----
{1}   (FORALL (a: alpha, g: [alpha -> alpha]):
      iter(g, a) = cs(a, map(g, iter(g, a))))
Rule? (SKOLEM!)
Skolemizing,

```

this simplifies to:

```
iter_map_def :
  |-----
{1}   iter(g!1, a!1) = cs(a!1, map(g!1, iter(g!1, a!1)))
```

Rewriting with theorem `co_induct` yields

```
iter_map_def :
  |-----
{1}   (EXISTS (R: Bisimulation[alpha]):
      R(iter(g!1, a!1), cs(a!1, map(g!1, iter(g!1, a!1))))))
```

Instantiate `R` with a trivial bisimulation

```
Rule? (inst +
      "{(X,Y:Stream[alpha])|
       EXISTS (a:alpha) :
         X = iter(g!1, a) &
         Y = cs(a, map(g!1, iter(g!1, a)))}")
this yields 2 subgoals:
```

The first subgoal is to show that the two streams are in the provided relation, the second is to show that the relation is a bisimulation. After beta reduction, the first subgoal simplifies to

```
iter_map_def.1 :
  |-----
{1}   EXISTS (a: alpha):
      iter(g!1, a!1) = iter(g!1, a)
      & cs(a!1, map(g!1, iter(g!1, a!1)))
      = cs(a, map(g!1, iter(g!1, a)))
```

The `(inst?)` command asks PVS to guess a value to provide for the existentially quantified variable. In this case, PVS correctly guesses `a!1`, resulting in a goal that is easily proven using propositional simplification.

```

Rule? (inst? )
Found substitution:
a gets a!1,
Instantiating quantified variables,
this simplifies to:
iter_map_def.1 :

  |-----
{1}  iter(g!1, a!1) = iter(g!1, a!1)
      & cs(a!1, map(g!1, iter(g!1, a!1)))
      = cs(a!1, map(g!1, iter(g!1, a!1)))

```

```

Rule? (prop)
Applying propositional simplification,

```

This completes the proof of `iter_map_def.1`.

The second subgoal is to show that the given relation is a bisimulation. After the proof steps `(skosimp*)(replace*)(hide -1 -2)` the resulting goal is:

```

iter_map_def.2 :
  |-----
[1]  hd(iter(g!1, a!2))
      =
      hd(cs(a!2, map(g!1, iter(g!1, a!2))))
      & EXISTS (a: alpha):
      tl(iter(g!1, a!2)) = iter(g!1, a)
      &
      tl(cs(a!2, map(g!1, iter(g!1, a!2))))
      = cs(a, map(g!1, iter(g!1, a)))

```

This splits into the two standard cases for showing that a relation is a bisimulation.

The first subgoal is to prove that the heads are equal

```

Rule? (split)
Splitting conjunctions,
this yields 2 subgoals:

```

`iter_map_def.2.1 :`

```

|-----
{1}   hd(iter(g!1, a!2))
      =
      hd(cs(a!2, map(g!1, iter(g!1, a!2))))

```

Rewriting with `hd_cs` and `hd_iter` discharges this goal. Leaving us with:

`iter_map_def.2.2 :`

```

|-----
{1}   EXISTS (a: alpha):
      tl(iter(g!1, a!2)) = iter(g!1, a)
      &
      tl(cs(a!2, map(g!1, iter(g!1, a!2))))
      = cs(a, map(g!1, iter(g!1, a)))

```

In this branch, we rewrite with `map_def`, `tl_cs`, `hd_iter`, and `tl_iter` giving us the following goal:

```

|-----
{1}   EXISTS (a: alpha):
      iter(g!1, g!1(a!2)) = iter(g!1, a)
      & cs(g!1(a!2), map(g!1, iter(g!1, g!1(a!2))))
      = cs(a, map(g!1, iter(g!1, a)))

```

The commands `(inst?)` and `(prop)` finish off the proof. ⊢

Most of the steps in the proof given above are common to many coinductive proofs. I have defined a PVS strategy named `co-induct` that handles some of the routine aspects of a coinductive proof. In particular, it automatically discharges the membership subgoal when presented with a trivial bisimulation. In addition, it reduces the proof obligation that the given relation is a bisimulation into the two standard subgoals: the heads are equal, and the tails are in the relation.

The following PVS proof of equation (3.8) illustrates the (`co-induct`) strategy.

Fact 4 $\text{map}(f, \text{const}(a)) = \text{const}(f(a))$

Proof:

`map_const :`

```
|-----
{1}  (FORALL (a: alpha, f: [alpha -> beta]):
      map(f, const(a)) = const(f(a)))
```

```
Rule? (CO-INDUCT
      "{(X,Y:Stream[beta]) |
       EXISTS f, a:
         X=map(f,const(a)) &
         Y=const(f(a))}")
```

this yields 2 subgoals:

`map_const.1 :`

```
|-----
{1}  hd(map(f!2, const(a!2)))
      = hd(const(f!2(a!2)))
```

Rule? (`postpone`)

Postponing `map_const.1`.

`map_const.2 :`

```
|-----
{1}  EXISTS (f: [alpha -> beta]), (a: alpha):
      tl(map(f!2, const(a!2)))
        = map(f, const(a))
        & tl(const(f!2(a!2))) = const(f(a))
```

Rule?

The first subgoal is proven by rewriting with `hd_map` and `hd_const`. The second subgoal is proven by rewriting with `tl_map` and `tl_const`, followed by `(inst?)` and `(assert)`. +

The information necessary to construct a trivial bisimulation is present in the initial goal. PVS provides some lisp functions that help in collecting information from formulae in the sequent and in generating terms for use in later proof steps. Using these, I developed a more advanced PVS strategy named `simple-co-induct` that automatically generates a candidate bisimulation and then invokes strategy `co-induct`. Consider equation (3.10) from section 3.1.

Fact 5 `zip(const(a),const(a)) = const(a)`

Proof: The following sequence of commands suffice to prove equation (3.10) using PVS and the streams library.

```
(""
 (SIMPLE-CO-INDUCT)
 (("1" (REWRITE "hd_zip"))
 ("2" (REWRITE "tl_zip")
 (REWRITE "tl_const")
 (INST?) (ASSERT))))
```

+

Coinductive proofs are automated further. Recall that the template for introducing corecursive stream definitions includes two facts for simplifying the head and tail of a given stream. These are installed as automatic rewrites and combined with some of the existing brute force strategies within PVS to provide powerful strategies for determining coinductive proofs using PVS. The most potent strategy for discovering coinductive proofs is `co-induct-and-simplify`. It is sufficiently powerful to prove

the preceding equations automatically. The following PVS proof of equation (3.3) demonstrates the utility of strategy `(co-induct-and-simplify)`

Fact 6 $\text{corec}(f,g)(a) = \text{map}(f,\text{iter}(g,a))$

Proof: (Taken verbatim from `*pvs*` `buffer`)

`corec_alt_def :`

```

|-----
{1}   (FORALL (a: alpha,
            f: [alpha -> beta],
            g: [alpha -> alpha]):
      corec(f, g)(a) = map(f, iter(g, a)))

```

Rule? (CO-INDUCT-AND-SIMPLIFY)

```

hd_corec rewrites hd(corec(f!2, g!2)(a!2))
  to f!2(a!2)
hd_iter rewrites hd(iter(g!2, a!2))
  to a!2
hd_map rewrites hd(map(f!2, iter(g!2, a!2)))
  to f!2(a!2)
tl_corec rewrites tl(corec(f!2, g!2)(a!2))
  to corec(f!2, g!2)(g!2(a!2))
tl_iter rewrites tl(iter(g!2, a!2))
  to iter(g!2, g!2(a!2))
tl_map rewrites tl(map(f!2, iter(g!2, a!2)))
  to map(f!2, iter(g!2, g!2(a!2)))

```

Showing equivalence by exhibiting trivial bisimulation,
Q.E.D.

The strategy `co-induct-and-simplify` is also useful for establishing equations (3.11) and (3.12). To prove equation (3.11), we first introduce the following lemma.

Lemma 2 $\text{map}(f,\text{iter}(f \circ f,a)) = \text{iter}(f \circ f,f(a))$

Proof: Using PVS strategy `co-induct-and-simplify`. +

Fact 7 $\text{iter}(f,a) = \text{zip}(\text{iter}(f \circ f,a),\text{map}(f,\text{iter}(f \circ f,a)))$

Proof: Strategy (co-induct-and-simplify) leaves us with the goal:

```

|-----
{1}   zip(map(f!2, iter(f!2 o f!2, a!2)),
        iter(f!2 o f!2, (f!2 o f!2)(a!2)))
      =
        zip(iter(f!2 o f!2, f!2(a!2)),
            map(f!2, iter(f!2 o f!2, f!2(a!2))))

```

Rewriting twice with lemma 2 and once with “o” completes the proof. ⊢

The challenge of equation (3.12) is to prove that such a function exists. Define $F(f,S) \hat{=} f(\text{hd}(S))$ and $G(f,S) \hat{=} (f \circ f, \text{tl}(S))$, then define

$$\text{dmap}(f,S) : \{ S1 \mid S1 = \text{corec}(F,G)(f,S) \}$$

From this corecursive definition, it is trivial to establish

```

dmap_def: LEMMA
  dmap(f,S) = cs(f(hd(S)),dmap(f o f,tl(S)))

```

```

hd_dmap : LEMMA
  hd(dmap(f,S)) = f(hd(S))

```

```

tl_dmap : LEMMA
  tl(dmap(f,S)) = dmap(f o f,tl(S))

```

All that remains is to show that this function `dmap` defines a stream that satisfies equation (3.12).

Fact 8 $\text{dmap}(f,S) = \text{cs}(f(\text{hd}(S)),\text{dmap}(f,\text{dmap}(f,\text{tl}(S))))$

Proof: Using `(co-induct-and-simplify)` we get the following goal:

```

|-----
{1}  dmap(f!2,
      cs(f!2(hd(tl(S!2))),
         dmap(f!2 o f!2, tl(tl(S!2))))))
=
      cs((f!2 o f!2)(hd(tl(S!2))),
         dmap(f!2 o f!2, dmap(f!2 o f!2, tl(tl(S!2))))))

```

Rewriting with `dmap_def` and “o” completes the proof. +

Advanced examples of stream definition

Thus far, the streams we have examined had a very simple form. In the study of streams there are some more complicated examples. This section presents corecursive definitions for two: `filter` and `sharp`.

The `filter` operator takes a predicate and a stream as arguments and returns a stream constructed from those elements of the argument stream that satisfy the given predicate. `Filter` satisfies the equation:

```

filter(P)(S) = IF P(hd(S)) THEN cs(hd(S), filter(P)(tl(S)))
              ELSE filter(P)(tl(S)) ENDIF

```

`Sharp` collapses a stream by removing any repeated values so that the resulting stream is such that no two adjacent positions have the same value. Its defining equation is:

```

sharp(S) = IF hd(S) /= hd(tl(S)) THEN cs(hd(S), sharp(tl(S)))
          ELSE sharp(tl(S)) ENDIF

```

These functionals are well-behaved if the input stream satisfies a fairness condition. Suppose only finitely many elements of a stream satisfy the given predicate, then the natural result of `filter` is finite. Similarly, the natural result of `sharp` is finite if the argument is ever a constant stream. However, there is no computable function to test whether the output of either of these functionals is finite. For our purposes, any solution to the above equations is sufficient. By addressing the special cases appropriately, we have been able to define streams that satisfy the above equations using the corecursion combinator. The next two subsections present the corecursive definitions of `filter` and `sharp`. The proofs that they satisfy the defining equations are available electronically [WWW].

Definition of filter

Let F and G be defined as follows:

```

F(P)(S): alpha =
  IF (EXISTS n: P(nth(S, n)))
  THEN nth(S, least({n | P(nth(S, n))}))
  ELSE non(P) ENDIF

G(P)(S): Stream[alpha] =
  iterate(tl, i)(S)
  WHERE
  i = IF (EXISTS n: P(nth(S, n)))
      THEN least({n | P(nth(S, n))}) + 1
      ELSE 0 ENDIF

```

The term `non(P)` returns an arbitrary element of type `alpha` such that $\neg P(\text{non}(P))$.

We now define

$$\text{filter}(P)(S) : \{S1 \mid S1 = \text{corec}(F(P), G(P))(S)\}$$

This has been proven to satisfy the above equation using PVS. Whenever `filter(P)` is applied to a stream where no position satisfies `P`, it returns the stream `const(non(P))`.

Definition of sharp

The definition of sharp is simpler. Let `G` be defined by

```
G(S): Stream[alpha] =
  iterate(tl[alpha], i)(S)
  WHERE
  i = IF (EXISTS n: nth(S, n) /= hd(S))
        THEN least({n | nth(S, n) /= hd(S)})
        ELSE 0 ENDIF
```

then we define

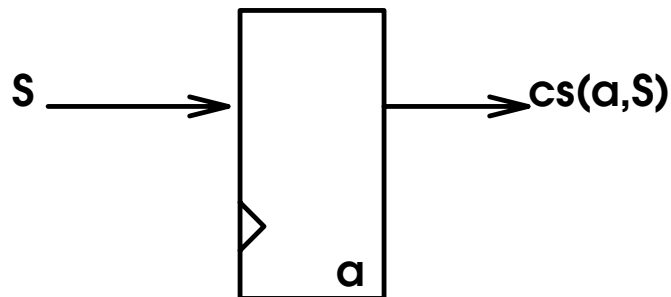
$$\text{sharp}(S) : \{S1 \mid S1 = \text{corec}(\text{hd}, G)(S)\}$$

This has been proven in PVS to satisfy the above defining equation for sharp. When `sharp` is given a constant stream, it behaves as the identity function.

4

Streams as a representation of Hardware

A system of stream equations provides a concise formal description of synchronous digital hardware [Joh84]. The key observation is that the stream constructor `cs` models the standard unit delay (or D-type flipflop) used in digital system design. The first argument to `cs` denotes the current contents of the register and the second argument is its input stream. In the examples that follow, `reg` (for register) is synonymous with `cs`.



It is well established that a synchronous digital system is determined by two

functions. An output function that maps the input and current state to the output, and a next-state function that maps the input and current state to the next state. This standard model of sequential systems is often referred to as the Huffman model [Huf54]¹ and is depicted in Figure 4.1. Using the specification style developed

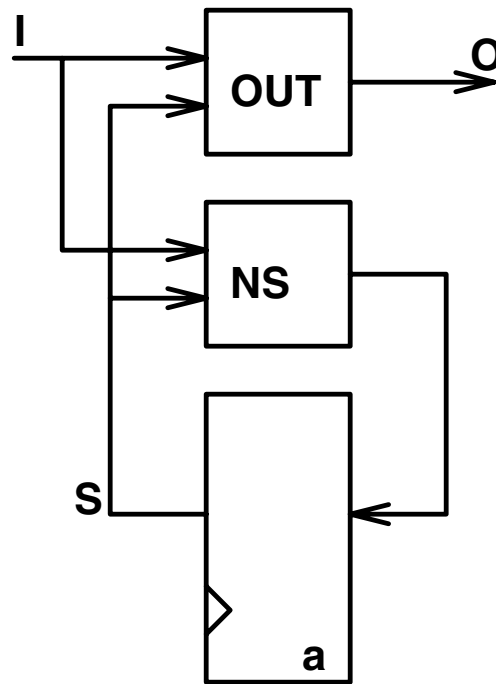


Figure 4.1: Standard model of Sequential System

by Johnson [Joh84], this standard system is described by the equations

$$S = \text{cs}(a, \text{NS}(I, S))$$

$$O = \text{OUT}(I, S)$$

Where functions NS and OUT are the next-state and output functions lifted to streams using a two argument version of `map`.

¹Huffman's paper describes asynchronous sequential systems decomposed into combinational logic and delay elements. Here we assume a synchronous system.

It is simple to formally define this standard model of a sequential system using the corecursion combinator. In this system there are three signal types. Let TI denote the type of the input, TO the type of the output, and TS the type of the internal state. Given the following PVS declarations:

```
I: Var Stream[TI]
O: Var Stream[TO]
s: Var TS
out: Var [TI, TS -> TO]
ns : Var [TI, TS -> TS]
```

A generic sequential system is defined by the following corecursive declaration:

```
sequential_system(out,ns)(I,s):
  { 0 | 0 = corec(lambda I,s:out(hd(I),s),
                 lambda I,s:(tl(I),ns(hd(I),s))
                 )(I,s)}
```

Since this is defined using the corecursion combinator, the following facts are easily established within PVS:

```
sequential_system_def: LEMMA
  sequential_system(out,ns)(I,s) =
    cs(out(hd(I),s),sequential_system(out,ns)(tl(I),ns(hd(I),s)))
```

```
hd_sequential_system: LEMMA
  hd( sequential_system(out,ns)(I,s)) = out(hd(I),s)
```

```
tl_sequential_system: LEMMA
  tl( sequential_system(out,ns)(I,s))
    = sequential_system(out,ns)(tl(I), ns(hd(I),s))
```

With these definitions, we can set up the proof obligations necessary to show equivalence between two sequential systems, even if the internal state is different.

Sometimes, however, we wish to explore properties of the internal state of a system. To enable this, we use `corec` and the next state function to define a single internal stream equation.

```
S: Var Stream[TS]
state(ns)(I,s): { S | S = corec(lambda I,s:s,
                                lambda I,s:(tl(I),ns(hd(I),s))
                                )(I,s)}

state_def: LEMMA
  state(ns)(I,s) = cs(s,state(ns)(tl(I),ns(hd(I),s)))

hd_state: LEMMA
  hd(state(ns)(I,s)) = s

tl_state: LEMMA
  tl(state(ns)(I,s)) = state(ns)(tl(I),ns(hd(I),s))
```

With this definition, we prove that we have streams that satisfy the defining equations for a sequential system.

```
sequential_system_equations: THEOREM
  LET S = state(ns)(I,s),
      O = sequential_system(out,ns)(I,s),
      OUT = lambda I,S: map2(out,I,S),
      NS = lambda I,S: map2(ns,I,S)
  IN
  S = cs(s,NS(I,S)) &
  O = OUT(I,S)
```

The PVS strategy (`co-induct-and-simplify`), presented in Chapter 3, proves the equations for `S` and `O`.

By restricting the input and state to finite types, these equations define a Mealy machine [Mea55]. By further restricting this model so that function `out` only depends on the state, these equations define a Moore machine [Moo56]. However, we are

generally interested in countably infinite types such as the integers, rationals, and abstract data types. This allows use of standard properties of these abstract types in refining the digital system architectures. Once the data path is determined, these types are refined to bit-vector representations.

4.1 Simple Examples

To illustrate the utility of specification using corecursion and verification using coinduction, I present a number of simple examples. The first two examples come from Johnson's dissertation and illustrate how systems of stream equations are defined using the PVS streams library. That is, the PVS streams library supports a shallow embedding of formal representations of hardware architectures defined using systems of stream equations. Subsequent examples illustrate how to use corecursion and coinduction to represent and reason about some standard abstractions in hardware verification.

Factorial

A system of stream equations describing a hardware architecture to compute the factorial function is [Joh84]:

$$X = \text{cs}(x, \text{DCR}(X))$$

$$Y = \text{cs}(1, \text{MPY}(X, Y))$$

$$\text{READY} = \text{ZERO?}(X)$$

$$\text{VALUE} = Y$$

At the first instant that *READY* is asserted, the *VALUE* signal contains $x!$. We use `map` to lift the decrement, multiply, and zero? functions to streams. The stream equation defining X is defined using `iter` and the defining equation for Y is defined using `state`. A definition of this system of equations using the PVS streams library is:

```
dcr(x): nat = x ~ 1;
mpy(x,y): nat = x*y;

factorial_stream_equations: LEMMA
  LET X = iter(dcr,x),
      Y = state[nat,nat,nat](mpy)(X,1),
      DCR = lambda (Z:Stream[nat]): map(dcr,Z),
      MPY = lambda (W,Z:Stream[nat]): map2(mpy,W,Z),
      ZERO? = lambda (Z:Stream[nat]): map[nat,bool](zero?,Z),
      READY = ZERO?(X),
      VALUE = Y
  IN
    X = cs(x,DCR(X)) &
    Y = cs(1,MPY(X,Y)) &
  READY = ZERO?(X) &
  VALUE = Y
```

In PVS, `~` defines cut-off subtraction. That is, when $x < y$, $x \sim y = 0$. Simple coinductive proofs suffice to show that the definitions of X and Y satisfy the given equations. Of course, this example does not involve simultaneous recursion, so it is simple to define in PVS. Systems of mutually recursive stream equations are more difficult to define using the streams library. The next example illustrates one technique for defining mutually recursive stream equations.

Fibonacci

Another simple example from Johnson's dissertation is a circuit that computes the Fibonacci sequence. The circuit is formally defined by the system of stream equations:

$$\begin{aligned}
 X &= \text{cs}(x, \text{DCR}(X)) \\
 Y &= \text{cs}(1, Z) \\
 Z &= \text{cs}(1, \text{ADD}(Y, Z)) \\
 \text{READY} &= \text{ZERO?}(X) \\
 \text{VALUE} &= Y
 \end{aligned}$$

The definitions of X , READY , and VALUE are done in the same manner as for the factorial circuit. We need a mechanism to define the mutual recursion captured by equations Y and Z . The method is to construct a stream of pairs and then project out to a pair of streams that satisfy the set of mutually recursive stream equations. The Streams library includes a collection of definitions to support transposing a stream of pairs to a pair of streams. Let $S : \text{Stream}[[T1, T2]]$, function L applied to S returns a stream over $T1$, and $R(S)$ has type $\text{Stream}[T2]$. With these functions, we define the Fibonacci stream equations as follows:

```

dcr(x:nat)   : nat       = x ~ 1;
zero?(x:nat) : bool      = (x = 0);
yz(y,z:nat) : [nat, nat] = (z, y + z);

```

```

fib_stream_equations: LEMMA
  LET X = iter(dcr,x),
      YZ = iter(yz,(y,z)),
      Y = L(YZ),
      Z = R(YZ),
      DCR = lambda (X:Stream[nat]):
              map(dcr,X),
      ADD = lambda (Y,Z:Stream[nat]):

```

```

        map2(lambda (x,y:nat): x+y, Y, Z),
ZERO? = lambda (Z:Stream[nat]):
        map[nat,bool](zero?, Z),
READY = ZERO?(X),
VALUE = Y
IN
    X = cs(x,DCR(X)) &
    Y = cs(y, Z) &
    Z = cs(z, ADD(Y, Z)) &
READY = ZERO?(X) &
VALUE = Y

```

Once again, the strategy (*co-induct-and-simplify*) was sufficient to prove that each of the streams X , Y , and Z satisfy the corresponding recursive stream equations.

4.2 Coinductive Proofs about Hardware

Melham [Mel88] defines four basic abstractions used in hardware verification. *Structural* abstraction occurs when we hide the internal structure of a device and only reason about its black-box behavior. *Behavioral* abstraction occurs when we only partially specify a device's behavior. This allows one to constrain the environment in which the device is intended to operate. *Data* abstraction is similar to structural abstraction in that it allows suppression of information about how data is represented. *Temporal* abstraction is used to relate two different views of the passage of time. A common example is microprocessor verifications where one view is that of a sequence of instructions. Each instruction is a function that transforms the state of the device. At a lower level, each instruction may require several cycles to complete.

Corecursion and coinduction play a role in at least two of these abstractions. Coinductive proofs are useful for demonstration that a sequential system satisfies

some behavioral specification. Corecursion is also useful for defining simple temporal abstractions. In addition, coinduction is useful for verification that two sequential systems have the same behavior. In this section, each of these is illustrated. First, a simple hardware transformation is proven to preserve the behavior of a sequential circuit. Then a behavioral abstraction is demonstrated. The chapter concludes with an illustration of simple temporal abstractions.

Retiming Transformation

A retiming transformation is used to distribute registers throughout a design in order to minimize the longest critical path delay. A shorter maximum critical path in a design means that the maximum cycle time for the circuit is less, thus allowing a faster clock rate.

Such transformations have been extensively studied [LS91], and have made their way into practice. We use them here to illustrate the utility and general nature of coinductive proofs. This example moves a unary combinational function f through a multiplexor and a register where the register output is fed back into the mux. The transformation is depicted in Figure 4.2.

The representation of this transformation using the PVS Streams library is:

$$\text{map}(f, \text{MUX_REG}(S, A, a)) = \text{MUX_REG}(S, \text{map}(f, A), f(a))$$

This transformation is verified using PVS strategy (`co-induct-and-simplify`). A simple sequence of DRS transformations also suffices to justify this replacement. There is no need to perform a coinductive proof each time a retiming transformation is required. This is further illustrated in one of the derivation examples in Chapter 6.

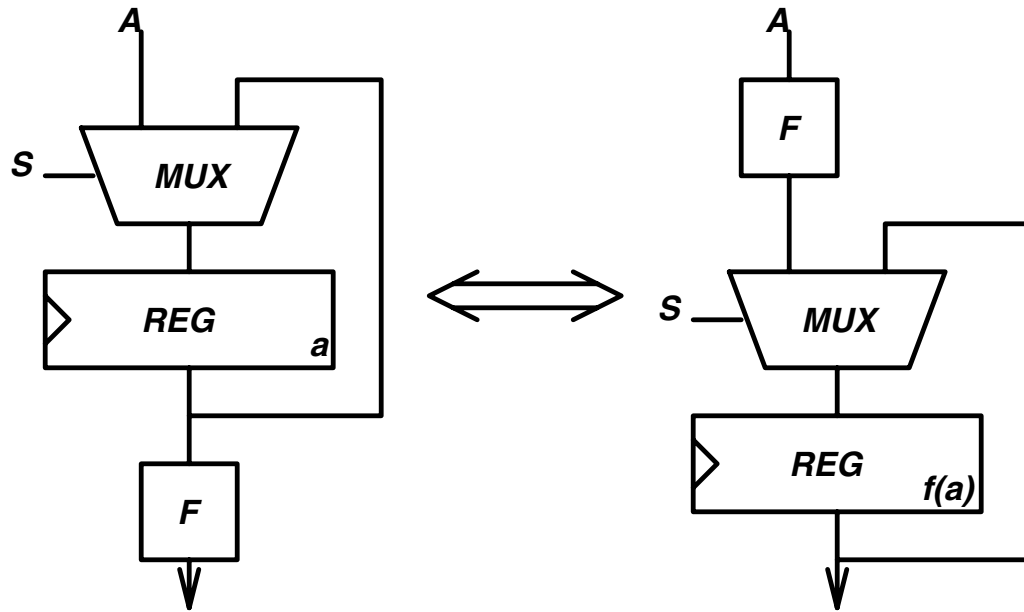


Figure 4.2: Simple Retiming Transformation

Behavioral Abstraction

In this example, we illustrate how to verify that a sequential system satisfies a behavioral specification using coinduction. Although this example may seem contrived, its relevance becomes apparent in Chapter 5. The behavioral specification describes a device that periodically behaves as a counter. The informal requirement is that so long as a reset signal is not asserted, the device operates as a counter, that is, the value on its output signal increases by one each clock tick. Nothing is asserted about its value when the reset signal is asserted. We define this behavior using the predicate `Invariant` from the streams library and the PVS dependent type mechanism. We define the type:

```
C(R): TYPE =
  {I : Stream[int] | Invariant(NOT R => EQ(t1(I), INC(I))) }
```

This type is parameterized by a stream of booleans that corresponds to a reset signal.

Whenever the reset signal is not asserted, a signal of type $C(R)$ behaves as a counter. Otherwise, its behavior is unspecified.

A counter circuit is depicted in Figure 4.3. This circuit has two argument streams.

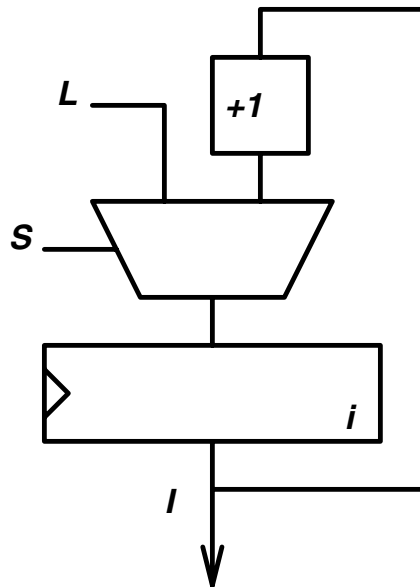


Figure 4.3: Simple Counter Circuit

The first is a stream of booleans, S , that controls the select line of the multiplexor. The second is a stream of integers, L , that is directly loaded into the register whenever the select line is low. Finally, when the select signal is high, this circuit stores the result of adding one to the current value, i , of the counter. Its corecursive definition in PVS is:

```
COUNT(S, L, i):
  { I | I = corec(lambda S, L, i : i,
                 lambda S, L, i:
                   (tl(S), tl(L),
                    mux(hd(S),hd(L),inc(i))))
                 )(S, L, i)}
```

The usual supporting lemmas are easily discharged. It is also possible to define this

circuit using `state` to define the stream equation. However, this requires that we explicitly pair up the input streams, and modify the iterated function so that its type signature matches. In general, it is easier to define systems of stream equations using `corec` directly.

We still need to show that the counter circuit satisfies the required behavioral property. That is, it has type:

```
C(R): TYPE = {I | Invariant(NOT R => EQ(tl(I), INC(I)))}
```

for some boolean valued stream `R`. The multiplexor select line, `S`, seems a likely candidate, but has the opposite truth value of what we want. If we let `R` be `NOT S`, we are able to establish the invariant property.

```
COUNT_Invariant: LEMMA
  Invariant( NOT NOT S => EQ(tl(COUNT(S, L, i)), INC(COUNT(S, L, i))))

JUDGEMENT COUNT HAS_TYPE
  [S:Stream[bool], Stream[int],int -> C(NOT S)]
```

The judgement statement asserts that function `COUNT` applied to a boolean stream `S`, an integer stream, and an integer always has type `C(NOT S)`. The type correctness condition generated by this judgement is alpha convertible to `COUNT_Invariant`, and is easily discharged by (`co-induct-and-simplify`).

Temporal Abstraction

Assume we have a microprocessor architecture defined so that each instruction corresponds to a state transition. An implementation of this instruction set may require a different number of clock cycles for each instruction. The following temporal

abstraction schema can be used to relate the design of the microprocessor to its instruction level specification. Function f maps each element of stream S into the number of cycles required to correspond to the next high-level state.

```
temp_abs(f)(S):
  { S1 | S1 = corec(lambda S: hd(S),
                   lambda S: iterate(tl,f(hd(S)))(S)
                   )(S)}
```

By iterating the tl operator by $f(hd(S))$ cycles we relate stream S to a specification with a coarser granularity.

Another use for temporal abstractions is to relate circuits operating at different clock rates. Bronstein and Talcott [BT90] use the stuttering counter depicted in Figure 4.4 to illustrate part of their approach for pipeline verification. The goal is to

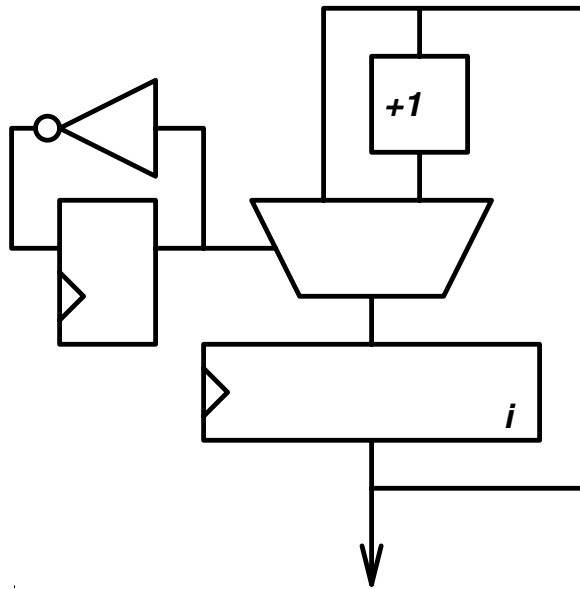


Figure 4.4: Stuttering Counter Circuit

show that a temporal abstraction of the stuttering counter is equivalent to a simple counter (Figure 4.5). The simple counter is just $iter(inc, i)$. One possible correctness criteria is to relate the simple counter to every other tick of the stuttering counter.

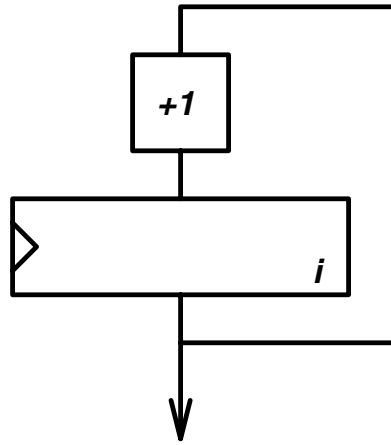


Figure 4.5: Simple Counter Circuit

Lemma 3 *Let the stuttering counter be defined by the following stream equations:*

$$\begin{aligned} \text{oscillate} &= \text{reg}(\text{false}, \neg \text{oscillate}) \\ \text{stutter} &= \text{reg}(i, \text{mux}(\text{oscillate}, \text{stutter}, \text{stutter} + 1)) \end{aligned}$$

then

$$\text{temp_abs}(\lambda x.2)(\text{stutter}) = \text{iter}(\text{inc}, i)$$

Proof: In PVS by strategy (co-induct-and-simplify). +

Alternatively, the **sharp** function defined in Chapter 3 is designed to eliminate stuttering on a stream. A correctness result using **sharp** is:

Lemma 4 *Let the stuttering counter be defined by the following stream equations:*

$$\begin{aligned} \text{oscillate} &= \text{reg}(\text{false}, \neg \text{oscillate}) \\ \text{stutter} &= \text{reg}(i, \text{mux}(\text{oscillate}, \text{stutter}, \text{stutter} + 1)) \end{aligned}$$

then

$$\text{sharp}(\text{stutter}) = \text{iter}(\text{inc}, i)$$

Proof: By coinduction in PVS. Strategy (`simple-co-induct`) generates a suitable bisimulation. Automatic rewrites, excluding `tl_sharp`, complete the goal for the heads and simplify the tail case to a point where two rewrites with `sharp`'s defining equation complete the proof. \dashv

A generalization of the stuttering counter replaces the oscillating control line with an arbitrary fair control line. This is depicted in Figure 4.6 with the underlying

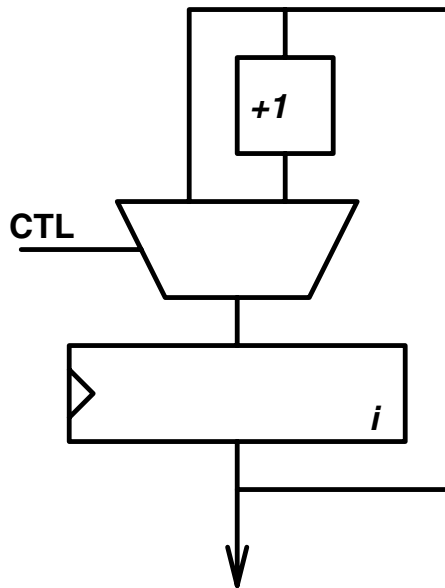


Figure 4.6: Fair Counter

assumption that CTL is asserted infinitely often.

One way to assert fairness on a boolean valued stream uses the `filter` function. For boolean valued stream `B`, define

$$\text{Fair}(B) : \text{bool} = \text{Invariant}(\text{filter}(\text{id})(B))$$

Lemma 5 *Let the (fair) stuttering counter be defined by the following stream equation:*

$$\mathbf{stutter} = \mathbf{reg}(i, \mathbf{mux}(\mathbf{ctl}, \mathbf{stutter}, \mathbf{stutter} + 1))$$

If $\mathbf{Fair}(\mathbf{ctl})$ then

$$\mathbf{sharp}(\mathbf{stutter}) = \mathbf{iter}(\mathbf{inc}, i)$$

Proof: Strategy (**simple-co-induct**) generates a suitable bisimulation. The only difficult part of the proof is coordinating **filter** and **sharp** in the tails case. \dashv

This last example warrants further investigation. It should be possible to simplify proof efforts that require synchronization between **filter** and **sharp**.

Fault-Tolerant Synchronization

In the early explorations of the design framework depicted in Figure 1.1 (page 3), we developed a hardware design of a verified fault-tolerant clock synchronization algorithm using a combination of formal design techniques [MPJ94]. These techniques included using standard DDD transformations, an ad hoc refinement verified using PVS, and BDD-based tautology checking. The resulting hardware description was suitable for realization using a field-programmable gate-array.

However, for the PVS verification of the ad hoc refinement presented in [MPJ94], the streams library had not yet been developed. Signals were represented by functions from time to the signal carrier type. The proofs were by induction, and were cluttered with several irrelevant proof obligations. In contrast, the verification of this replacement using coinduction focuses attention on the underlying reasons that the replacement is justified. In this chapter, the ad hoc refinement is verified using coinduction, as is a clever optimization to the circuit. The first refinement substitutes a pair of registers in place of a collection of clock readings captured using a list data structure. Similar changes in data representation are common when transforming an

abstractly defined algorithm to a concrete hardware description. The second refinement is a clever optimization to the derived architecture. The verification of each refinement exploits properties of the surrounding circuitry.

5.1 Top-Level Properties

In a distributed computing system, one approach to ensure coordinated activity is to maintain synchronization of the clocks of the distributed processors. In a fault-tolerant computer architecture, the synchronization algorithm must also tolerate a bounded number of failures. The property that a synchronization algorithm must ensure is that:

For any two clocks C_p and C_q that are nonfaulty at time t

$$|C_p(t) - C_q(t)| \leq \delta$$

In addition, a clock must maintain an accurate measure of the passage of time. Clock synchronization consists of periodically exchanging values and executing an averaging function so that the above properties are guaranteed.

5.2 The Verified Algorithm

Schneider [Sch87] demonstrates that many fault-tolerant clock synchronization algorithms can be treated as refinements of a general paradigm. Shankar [Sha92] and Miner [Min93] have provided mechanically checked proofs of Schneider's paradigm. Miner's verification is the top-level specification for the circuit developed here.

A generalized view of the algorithm employed by each participant is:

```
do forever {  
    exchange clock values;  
    determine adjustment for this interval;  
    determine local time to apply correction;  
    when time, apply correction }
```

Schneider's paradigm is parameterized by

- R —the nominal duration of a synchronization interval. Each clock is adjusted with a frequency approximately equal to $1/R$.
- N —the number of clocks participating in the protocol, $N > 0$
- F —the number of faults tolerated
- A mechanism for exchanging clock values. The relationship between N and F depends on this mechanism. For Byzantine faults, $N > 3F$. θ denotes a collection of readings from clocks in the system. In the mechanically verified theory, θ is a function from clock indices to clock readings.
- cfn —a convergence function to compute an adjustment from this collection of readings. The following three properties are sufficient to ensure synchronization.
 - *Translation Invariance*: The function depends only on the relative magnitude of the readings, not the absolute magnitude.
 - *Precision Enhancement*: For any two good clocks with similar estimates of other clock's values, the result of computing the convergence function is similar.

- *Accuracy Preservation*: If the readings from good clocks are sufficiently similar, then the computed value of the convergence function is close to all good clocks.

The fault-tolerant midpoint convergence function,

$$cfn_{MID}(\theta) = \left\lfloor \frac{\theta_{(F+1)} + \theta_{(N-F)}}{2} \right\rfloor,$$

where $\theta_{(m)}$ = the m th largest value in collection θ

employed in the Welch and Lynch [WL88] clock synchronization algorithm, possesses the required properties of a convergence function [Min93].

5.3 Behavioral Specification

An abstract view of the behavioral specification is given in Figure 5.1 using the ASM chart notation from [PW87]. This corresponds to a top-level DRS specification. It establishes the control of the machine and identifies some of the registers, but leaves architectural components abstractly specified. In particular, there is not an explicit representation for either θ or cfn . The choice for cfn is the fault-tolerant midpoint convergence function presented in section 5.2.

This presentation assumes a standard mechanism for exchanging readings. At a fixed offset into each synchronization interval, each participant in the protocol transmits a signal. Upon receipt of that signal, each clock captures its estimate for the transmitting clock. By analyzing delays in the communication mechanism we can arrive at an estimate of what the value of the local counter, LC , should be when a clock receives a signal from another clock that is perfectly synchronized. This is a

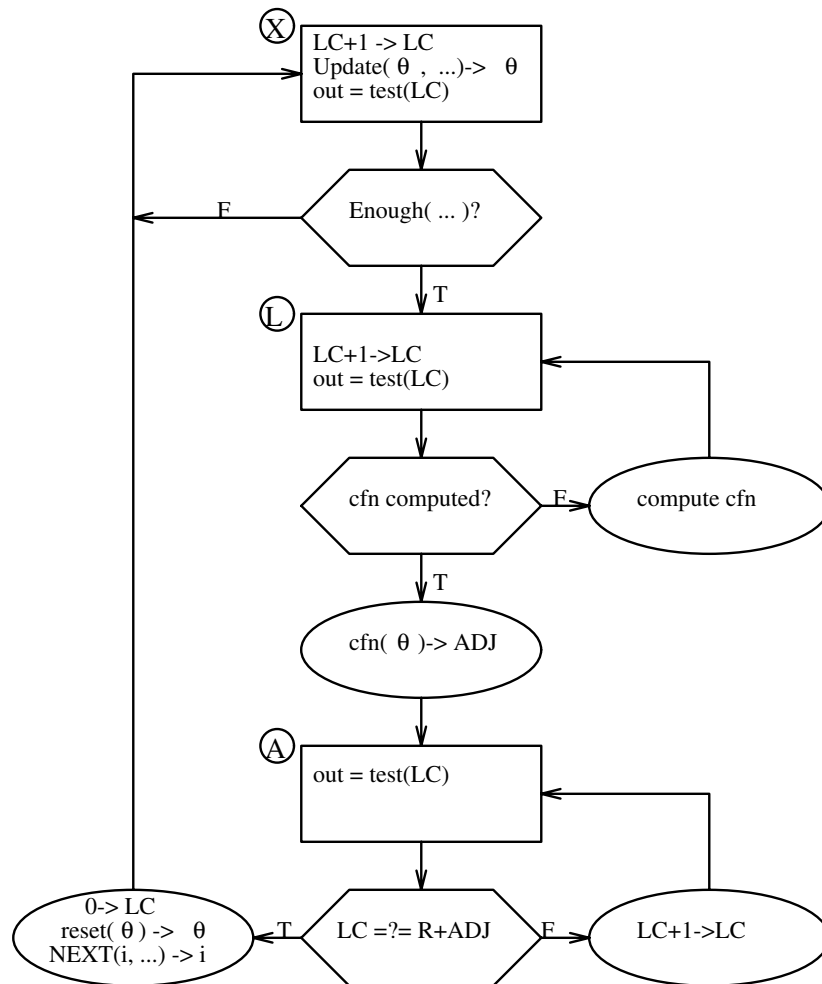


Figure 5.1: ASM Chart for General Algorithm

constant value denoted by Q . Thus, the value captured when a signal is received from a remote clock is $LC - Q$.

The next step is to provide specifications for the functions manipulating θ . The arguments to function **Update** include θ , a collection of N signals, and a reading, $LC - Q$. Since the convergence function chooses elements of θ based on their relative magnitude, it is desirable to capture these readings in an ordered manner. **Update** maintains an array of flags, indicating which signals have been received during the current interval. It also maintains a list of readings using the following function:

```
(define update-readings
  (lambda (theta reading pop-count)
    (let ((new-cnt (- pop-count (length theta))))
      (append theta (make-list new-cnt reading)))))
```

Argument **pop-count** (population count) represents the number of signals received from distinct clocks during this interval. Since the argument for **reading**, $LC - Q$, is increasing with each tick of the local counter, this function captures the readings of the remote clocks in a sorted list.

The DRS specification of the convergence function is:

```
(define cfn
  (lambda (theta)
    (floor_divby2 (add (index (f+1) theta)
                      (index (n-f) theta)))))

(define floor_divby2 (lambda (a) (floor (div a 2))))
```

This is a direct translation of the specification given in Section 5.2. Since we have

been careful to capture the readings in sorted order, `index` simply selects the $(F + 1)$ th and $(N - F)$ th elements of the sorted list.

This specification of the synchronization algorithm is quite general. Any convergence function that satisfies the required properties may be used. For many of the published convergence functions, it is unnecessary to alter the mechanism for capturing readings of remote clocks in a sorted list. Any of the following convergence functions could be used instead of the fault-tolerant midpoint (the names are from Schneider [Sch87]): Ego-centric Average, Fast Convergence Algorithm, or Fault-tolerant Average. Other algorithms, such as that by Srikanth and Toueg [ST87], do not fit as cleanly into this specification.

Other portions of the algorithm depend upon the choice of convergence function. For the fault-tolerant midpoint convergence function, there are enough readings to compute the convergence function whenever there have been signals from at least $N - F$ other clocks. Thus, for this design, predicate ‘`Enough(...)?`’ simply corresponds to a population count of received synchronization messages greater than $N - F$. Similarly, standard hardware constructs are sufficient for computing the fault-tolerant midpoint within a single tick. Therefore, for this design, predicate ‘`cfn computed?`’ is always true.

During normal operation, the interval index i is incremented at the end of each interval. However, at system startup or when multiple faults occur, it may be appropriate to clear this index. This abstractly defined function is given the name `NEXT` in the ASM chart. For a system that enables recovery from transient faults, `NEXT` also includes a majority vote of indices received from other clocks. The elapsed time in clock ticks from the start of the protocol is $iR + LC$.

Now that the architectural components of this behavioral specification have been

defined, it is transformed into an initial structural description. The next step is to refine the architecture. The initial architecture constructed from this behavioral specification is given in Figure 5.2.

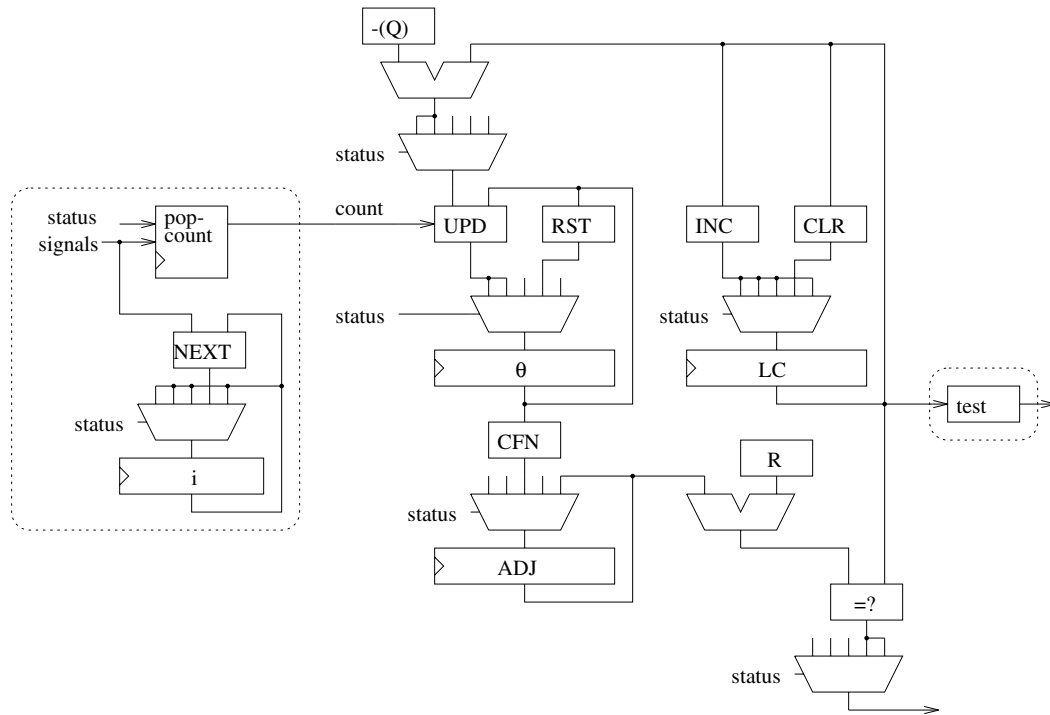


Figure 5.2: Initial Architecture

The areas enclosed by dotted lines in Figure 5.2 remains unchanged throughout the remainder of this development. Function `test` generates signals to other clocks in the system. Correct implementation requires knowledge of delays in the underlying communication mechanism. Therefore, we defer implementing this portion of the design until we build a complete system. Similarly, there are compelling reasons to defer implementing the population count module and the interval index counter. In particular, the function `NEXT` includes a vote of indices received from other clocks in the system. This function can either be realized in software or can be merged with the voter from a Byzantine agreement module.

5.4 Initial Derivation

Since we know that the convergence function only requires two readings, it is not necessary to generate estimates for all clocks in the system. We would like to replace this portion of the design with a more efficient realization. The first step is to isolate the portion of the design to be transformed. This sub-circuit, after some simplification to the control structure, is depicted in Figure 5.3. The stream equations

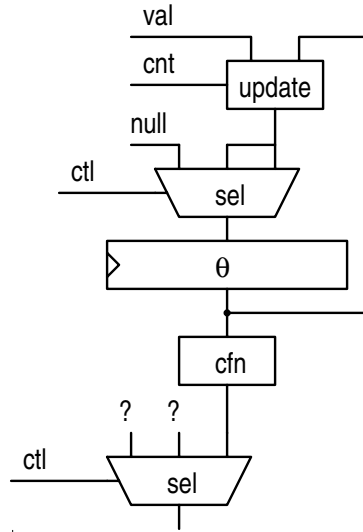


Figure 5.3: Refined sub-circuit prior to ad hoc transformation

corresponding to this sub-circuit (after expanding the definition of `cfn`) are:

$$\begin{aligned} \theta &= \text{reg}(\text{?}, \text{sel}(\text{ctl}, \text{null}, \text{update}(\theta, \text{val}, \text{cnt}), \text{update}(\theta, \text{val}, \text{cnt}))) \\ \text{sel-cfn} &= \text{sel}(\text{ctl}, \text{?}, \text{?}, \left\lfloor \frac{\text{index}(\theta, F + 1) + \text{index}(\theta, N - F)}{2} \right\rfloor) \end{aligned}$$

It is not necessary to provide storage for readings from all the participants in the protocol. The only readings required for computation of the convergence function are the $(F + 1)$ th and $(N - F)$ readings to arrive. The collection of clock readings, θ , is represented by a sorted list. We replace this list, which would require N registers,

with two registers, one for each value required in the computation of the convergence function.

The first step is to isolate the terms $\text{index}(\theta, F + 1)$ and $\text{index}(\theta, N - F)$, preserving the context in which these values are required.

$$\begin{aligned}\theta &= \text{reg}(\text{?}, \text{sel}(\text{ctl}, \text{null}, \text{update}(\theta, \text{val}, \text{cnt}), \text{update}(\theta, \text{val}, \text{cnt}))) \\ \text{idx-f1} &= \text{sel}(\text{ctl}, \text{?}, \text{?}, \text{index}(\theta, F + 1)) \\ \text{idx-nf} &= \text{sel}(\text{ctl}, \text{?}, \text{?}, \text{index}(\theta, N - F)) \\ \text{sel-cfn} &= \text{sel}(\text{ctl}, \text{?}, \text{?}, \left\lfloor \frac{\text{idx-f1} + \text{idx-nf}}{2} \right\rfloor)\end{aligned}$$

The next step is to introduce stream equations defining the new sub-circuit:

$$\begin{aligned}\text{f1} &= \text{reg}(\text{?}, \text{sel}(\text{ctl}, \text{false}, (\text{cnt} \geq (F + 1)), (\text{cnt} \geq (F + 1)))) \\ \theta_{(F+1)} &= \text{reg}(\text{?}, \text{mux}(\text{f1}, \text{val}, \theta_{(F+1)})) \\ \text{nf} &= \text{reg}(\text{?}, \text{sel}(\text{ctl}, \text{false}, (\text{cnt} \geq (N - F)), (\text{cnt} \geq (N - F)))) \\ \theta_{(N-F)} &= \text{reg}(\text{?}, \text{mux}(\text{nf}, \text{val}, \theta_{(N-F)})) \\ \theta &= \text{reg}(\text{?}, \text{sel}(\text{ctl}, \text{null}, \text{update}(\theta, \text{val}, \text{cnt}), \text{update}(\theta, \text{val}, \text{cnt}))) \\ \text{idx-f1} &= \text{sel}(\text{ctl}, \text{?}, \text{?}, \text{index}(\theta, F + 1)) \\ \text{idx-nf} &= \text{sel}(\text{ctl}, \text{?}, \text{?}, \text{index}(\theta, N - F)) \\ \text{sel-cfn} &= \text{sel}(\text{ctl}, \text{?}, \text{?}, \left\lfloor \frac{\text{idx-f1} + \text{idx-nf}}{2} \right\rfloor)\end{aligned}$$

The desired modification is to substitute

$$\text{sel}(\text{ctl}, \text{?}, \text{?}, \theta_{(N-F)})$$

for

$$\text{sel}(\text{ctl}, ?, ?, \text{index}(\theta, N - F))$$

in the right-hand side of the defining equation for `idx-nf`. A similar substitution is needed for `idx-f1`.

Both substitutions are justified by exploiting properties of the surrounding design. It is known that the stream `cnt` is non-decreasing between reset events. This fact, coupled with the definition of the function `update` ensures that the length of the list θ is non-decreasing between reset events. In addition, the list grows by appending new values to the end of the list. Therefore, once the length of θ is at least n , $\text{index}(\theta, n)$ is invariant until the next (active low) reset. The primary justification for both replacements is the following result:

Theorem 7 For $I, J, K \in \mathbf{Z}^\infty$, $\text{reset} \in \mathbf{B}^\infty$, $\text{cnt} \in \mathbf{N}^\infty$, $b \in \mathbf{B}$, $i \in \mathbf{Z}$, $l \in \text{list}[\mathbf{Z}]$, if

$$\text{Invariant}(\text{IF reset THEN EQ}(\text{tl}(\text{cnt}), 0) \text{ ELSE } \text{tl}(\text{cnt}) \geq \text{cnt} \text{ ENDIF})$$

$$b = (\text{length}(l) \geq \text{idx})$$

$$b \Rightarrow (i = \text{index}(l, \text{idx}))$$

$$\text{length}(l) \leq \text{hd}(\text{cnt}),$$

and the stream equations are defined by

$$\text{load} = \text{reg}(b, \text{mux}(\neg \text{reset}, \text{false}, \text{cnt} \geq \text{idx}))$$

$$\theta_{(\text{idx})} = \text{reg}(i, \text{mux}(\text{load}, I, \theta_{(\text{idx})}))$$

$$\theta = \text{reg}(l, \text{mux}(\neg \text{reset}, \text{null}, \text{update}(\theta, I, \text{cnt}))),$$

then

$$\begin{aligned} & \text{mux}(\neg\text{reset}, J, \text{mux}(\text{load}, K, \text{index}(\theta, \text{idx}))) \\ &= \text{mux}(\neg\text{reset}, J, \text{mux}(\text{load}, K, \theta_{(\text{idx})})) \end{aligned}$$

Proof: The proof is by coinduction. A suitable bisimulation is suggested by the statement of the theorem. Let $X \mathcal{R} Y$ if there exist $I, J, K \in \mathbf{Z}^\infty$, $\text{reset} \in \mathbf{B}^\infty$, $\text{cnt} \in \mathbf{N}^\infty$, $b \in \mathbf{B}$, $i \in \mathbf{Z}$, $l \in \text{list}[\mathbf{Z}]$, such that

$$\text{Invariant}(\text{IF } \text{reset} \text{ THEN } \text{EQ}(\text{tl}(\text{cnt}), 0) \text{ ELSE } \text{tl}(\text{cnt}) \geq \text{cnt} \text{ ENDIF}),$$

$$\begin{aligned} b &= (\text{length}(l) \geq \text{idx}), \\ b &\Rightarrow (i = \text{index}(l, \text{idx})), \\ \text{length}(l) &\leq \text{hd}(\text{cnt}), \\ \text{load} &= \text{reg}(b, \text{mux}(\neg\text{reset}, \text{false}, \text{cnt} \geq \text{idx})), \\ \theta_{(\text{idx})} &= \text{reg}(i, \text{mux}(\text{load}, I, \theta_{(\text{idx})})), \\ \theta &= \text{reg}(l, \text{mux}(\neg\text{reset}, \text{null}, \text{update}(\theta, I, \text{cnt}))), \\ X &= \text{mux}(\neg\text{reset}, J, \text{mux}(\text{load}, K, \text{index}(\theta, \text{idx}))), \text{ and} \\ Y &= \text{mux}(\neg\text{reset}, J, \text{mux}(\text{load}, K, \theta_{(\text{idx})})) \end{aligned}$$

Relation \mathcal{R} was constructed to contain the streams in the theorem. All that remains is to show that \mathcal{R} is a bisimulation. Suppose we have streams X and Y such that $X \mathcal{R} Y$. There are two cases to consider:

Heads: $\text{hd}(X) = \text{hd}(Y)$.

The goal is to show that

$$\text{hd}(\text{mux}(\neg\text{reset}, J, \text{mux}(\text{load}, K, \text{index}(\theta, \text{idx}))))$$

$$= \text{hd}(\text{mux}(\neg\text{reset}, J, \text{mux}(\text{load}, K, \theta_{(\text{idx})})))$$

By rewriting with the knowledge we have about the heads of all streams involved, this simplifies to:

$$\begin{aligned} & \text{mux}(\neg\text{hd}(\text{reset}), \text{hd}(J), \text{mux}(b, \text{hd}(K), \text{index}(l, \text{idx}))) \\ &= \text{mux}(\neg\text{hd}(\text{reset}), \text{hd}(J), \text{mux}(b, \text{hd}(K), i)) \end{aligned}$$

By expanding the definition of `mux`, this simplifies to showing that

$$\neg\text{hd}(\text{reset}) \wedge b \Rightarrow \text{index}(l, \text{idx}) = i$$

This is a direct consequence of the constraints on the state variables.

Tails: $\text{tl}(X) \mathcal{R} \text{tl}(Y)$.

Since $X \mathcal{R} Y$, we have $I', J', K' \in \mathbf{Z}^\infty$, $\text{reset}' \in \mathbf{B}^\infty$, $\text{cnt}' \in \mathbf{N}^\infty$, $b' \in \mathbf{B}$, $i' \in \mathbf{Z}$, $l' \in \text{list}[\mathbf{Z}]$, such that

Invariant(IF `reset'` THEN EQ(`tl(cnt)'`,0) ELSE `tl(cnt)'` \geq `cnt'` ENDIF),

$$b' = (\text{length}(l') \geq \text{idx}),$$

$$b' \Rightarrow (i' = \text{index}(l', \text{idx})),$$

$$\text{length}(l') \leq \text{hd}(\text{cnt}'),$$

$$\text{load} = \text{reg}(b', \text{mux}(\neg\text{reset}', \text{false}, \text{cnt}' \geq \text{idx})),$$

$$\theta_{(\text{idx})} = \text{reg}(i', \text{mux}(\text{load}, I', \theta_{(\text{idx})})),$$

$$\theta = \text{reg}(l', \text{mux}(\neg\text{reset}', \text{null}, \text{update}(\theta, I', \text{cnt}'))),$$

$$X = \text{mux}(\neg\text{reset}', J', \text{mux}(\text{load}, K', \text{index}(\theta, \text{idx}))), \text{ and}$$

$$Y = \text{mux}(\neg\text{reset}', J', \text{mux}(\text{load}, K', \theta_{(\text{idx})}))$$

The tail of a stream has the same structure, so if we rewrite with the stream tails, we get:

$$\begin{aligned}
\text{tl}(\text{load}) &= \text{reg}(\text{mux}(\neg\text{hd}(\text{reset}'), \text{false}, \text{hd}(\text{cnt}') \geq \text{idx}), \\
&\quad \text{mux}(\neg\text{tl}(\text{reset}'), \text{false}, \text{tl}(\text{cnt}') \geq \text{idx})), \\
\text{tl}(\theta_{(\text{idx})}) &= \text{reg}(\text{mux}(b', \text{hd}(I'), i'), \\
&\quad \text{mux}(\text{tl}(\text{load}), \text{tl}(I'), \text{tl}(\theta_{(\text{idx})}))), \\
\text{tl}(\theta) &= \text{reg}(\text{mux}(\neg\text{hd}(\text{reset}'), \text{null}, \text{update}(l', \text{hd}(I'), \text{hd}(\text{cnt}'))), \\
&\quad \text{mux}(\neg\text{tl}(\text{reset}'), \text{null}, \text{update}(\text{tl}(\theta), \text{tl}(I'), \text{tl}(\text{cnt}')))), \\
\text{tl}(X) &= \text{mux}(\neg\text{tl}(\text{reset}'), \text{tl}(J'), \\
&\quad \text{mux}(\text{tl}(\text{load}), \text{tl}(K'), \text{index}(\text{tl}(\theta), \text{idx}))),
\end{aligned}$$

and

$$\text{tl}(Y) = \text{mux}(\neg\text{tl}(\text{reset}'), \text{tl}(J'), \text{mux}(\text{tl}(\text{load}), \text{tl}(K'), \text{tl}(\theta_{(\text{idx})})))$$

In these equations $\text{tl}(\text{load})$, $\text{tl}(\theta_{(\text{idx})})$, and $\text{tl}(\theta)$ are just identifiers, so they are replaced with load , $\theta_{(\text{idx})}$, and θ , respectively. Thus we instantiate the existentially quantified variables with $\text{tl}(I')$, $\text{tl}(J')$, $\text{tl}(K')$, $\text{tl}(\text{reset}')$, $\text{tl}(\text{cnt}')$, $\text{mux}(\neg\text{hd}(\text{reset}'), \text{false}, \text{hd}(\text{cnt}') \geq \text{idx})$, $\text{mux}(b', \text{hd}(I'), i')$. and $\text{mux}(\neg\text{hd}(\text{reset}'), \text{null}, \text{update}(l', \text{hd}(I'), \text{hd}(\text{cnt}')))$. To complete the proof, all that remains is to show that the new state variables satisfy the appropriate invariants. That is:

$$\begin{aligned}
&\text{mux}(\neg\text{hd}(\text{reset}'), \text{false}, \text{hd}(\text{cnt}') \geq \text{idx}) \\
&= (\text{length}(\text{mux}(\neg\text{hd}(\text{reset}'), \text{null}, \text{update}(l', \text{hd}(I'), \text{hd}(\text{cnt}')))) \geq \text{idx}), \\
&\text{mux}(\neg\text{hd}(\text{reset}'), \text{false}, \text{hd}(\text{cnt}') \geq \text{idx}) \\
&\Rightarrow (\text{mux}(b', \text{hd}(I'), i')
\end{aligned}$$

$$\begin{aligned}
&= \text{index}(\text{mux}(\neg \text{hd}(\text{reset}'), \text{null}, \text{update}(l', \text{hd}(I'), \text{hd}(\text{cnt}'))), \text{idx}), \\
&\text{and} \\
&\text{length}(\text{mux}(\neg \text{hd}(\text{reset}'), \text{null}, \text{update}(l', \text{hd}(I'), \text{hd}(\text{cnt}')))) \\
&\leq \text{hd}(\text{tl}(\text{cnt}'))
\end{aligned}$$

When a reset occurs, the new list is set to null and these properties all follow trivially. During normal operation (e.g. not reset), the first time $\text{hd}(\text{cnt}') \geq \text{idx}$, both i and $\text{index}(l, \text{idx})$ are set to $\text{hd}(I')$. After that point, both remain the same until the next reset. \dashv

The above proof was originally developed using PVS. The strategy (**co-induct-and-simplify**) automatically guessed a suitable bisimulation. The main goals of the coinductive proof are completed by the strategy, and all that remains are sub-goals requiring that the invariants are preserved. These are discharged using some basic results about lists. The automated strategy focussed attention directly on the goals required to show that the state invariants are preserved.

The following argument illustrates how this result justifies the two replacements. When $\text{idx} = N - F$,

$$\text{mux}(\neg \text{reset}, I, \text{mux}(\text{nf}, J, K)) = \text{sel}(\text{ctl}, I, J, K)$$

Therefore, Theorem 7 directly justifies the replacement. For the case when $\text{idx} = F + 1$

$$\text{mux}(\neg \text{reset}, I, \text{mux}(\text{f1}, J, K)) = \text{sel}(\text{ctl}, I, \text{mux}(\text{f1}, J, K), K)$$

Thus, we need to first transform

$$\text{idx-f1} = \text{sel}(\text{ctl}, ?, ?, \text{index}(\theta, F + 1))$$

to

$$\text{idx-f1} = \text{sel}(\text{ctl}, ?, \text{mux}(\text{f1}, ?, \text{index}(\theta, F + 1)), \text{index}(\theta, F + 1))$$

by instantiating the middle “?” with $\text{mux}(\text{f1}, ?, \text{index}(\theta, F + 1))$. This then allows us to apply Theorem 7.

At this point we substitute $\theta_{(F+1)}$ for idx-f1 , $\theta_{(N-F)}$ for idx-nf , remove stream θ , and simplify, to give us:

$$\begin{aligned} \text{f1} &= \text{reg}(?, \text{sel}(\text{ctl}, \text{false}, (\text{cnt} \geq (F + 1)), (\text{cnt} \geq (F + 1)))) \\ \theta_{(F+1)} &= \text{reg}(?, \text{mux}(\text{f1}, \text{val}, \theta_{(F+1)})) \\ \text{nf} &= \text{reg}(?, \text{sel}(\text{ctl}, \text{false}, (\text{cnt} \geq (N - F)), (\text{cnt} \geq (N - F)))) \\ \theta_{(N-F)} &= \text{reg}(?, \text{mux}(\text{nf}, \text{val}, \theta_{(N-F)})) \\ \text{cfn} &= \left\lfloor \frac{\theta_{(F+1)} + \theta_{(N-F)}}{2} \right\rfloor \\ \text{sel-cfn} &= \text{sel}(\text{ctl}, ?, ?, \text{cfn}) \end{aligned}$$

A circuit diagram corresponding to this system of stream equations is given in Figure 5.4.

5.5 An Optimization

Torres-Pomales [TP95] discovered a more efficient realization of the core synchronization circuit. The optimization trades time for space in a clever manner. Verification of this optimization was previously presented in [MJ96]. Figure 5.5 illustrates the core circuit computing the convergence function from Section 5.4. The signal val is the output of a counter. The signals f1 and nf are boolean valued signals that

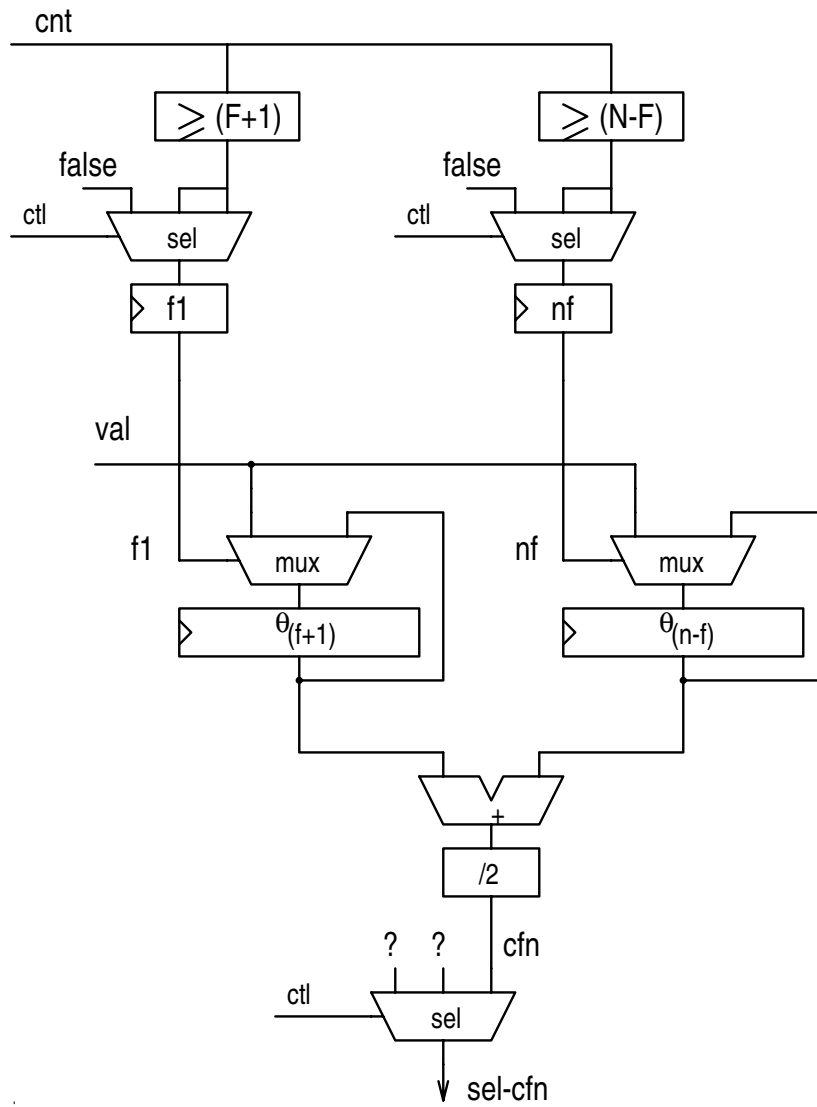


Figure 5.4: Refined sub-circuit after ad hoc transformation

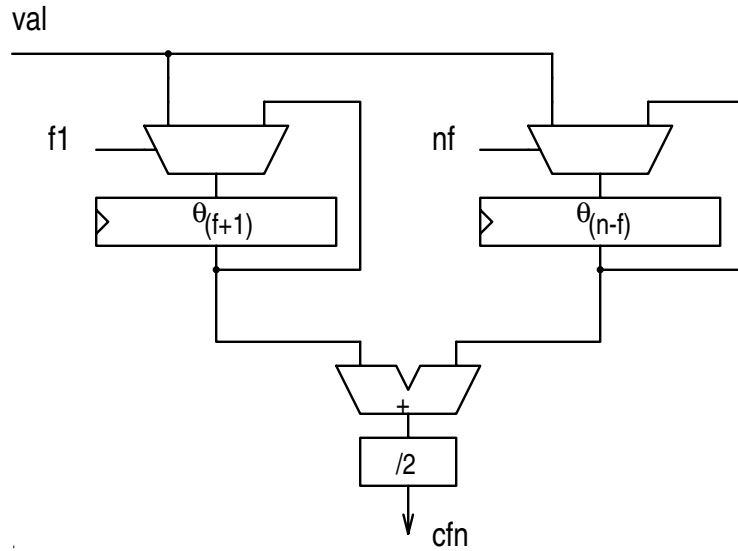


Figure 5.5: Core Circuit for Computing Convergence Function

indicate receipt of a synchronization signal from at least $F + 1$ and $N - F$ distinct participants in the protocol, respectively.

Torres-Pomales recognized that he could exploit the time interval between the $(F + 1)$ th and $(N - F)$ th signals to partially compute the convergence function. His optimization consists of capturing the $(F + 1)$ th reading as before, but he then increments the captured value every other clock tick until the signal from the $(N - F)$ th clock arrives. At this point the stored value is exactly the required value of the convergence function.

This next section outlines a technique to transform the previous design into Torres-Pomales' design. The optimized convergence function is depicted in Figure 5.6. This optimization requires assumptions about the input signals. We already know that the integer stream **val** is the output of a counter, so it increases by 1 each tick of the clock during the interval between resets. We also know that the Boolean streams **f1** and **nf** follow the behavior shown in Figure 5.7. The next section outlines a proof

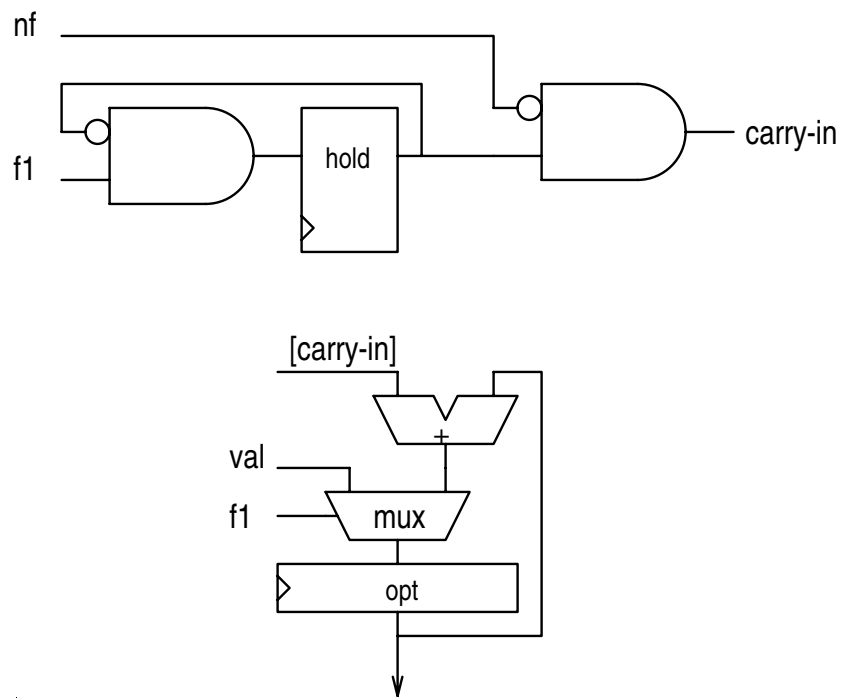


Figure 5.6: Optimized Convergence Function

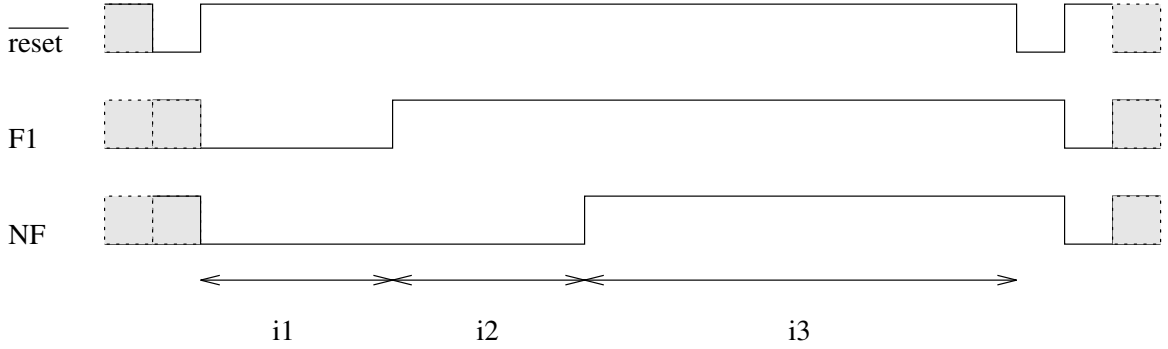


Figure 5.7: Signal Assumptions

that this refinement is correct.

Verification

The original circuit (Figure 5.5) is described by the following collection of stream equations:

$$\begin{aligned}\theta_{(F+1)} &= \text{reg}(?, \text{mux}(\text{f1}, \text{val}, \theta_{(F+1)})) \\ \theta_{(N-F)} &= \text{reg}(?, \text{mux}(\text{nf}, \text{val}, \theta_{(N-F)})) \\ \text{cfn} &= \left\lfloor \frac{\theta_{(F+1)} + \theta_{(N-F)}}{2} \right\rfloor\end{aligned}$$

The optimized circuit (Figure 5.6) is described by these stream equations:

$$\begin{aligned}\text{hold} &= \text{reg}(?, \text{f1} \wedge \neg \text{hold}) \\ \text{carry-in} &= \text{hold} \wedge \neg \text{nf} \\ \text{opt} &= \text{reg}(?, \text{mux}(\text{f1}, \text{val}, (\text{opt} + [\text{carry-in}])))\end{aligned}$$

In these equations, I adopt a notational convention from [GKP89]. For a boolean valued expression φ , $[\varphi] \hat{=} \text{IF } \varphi \text{ THEN } 1 \text{ ELSE } 0 \text{ ENDIF}$.

We wish to prove that $\mathbf{opt} = \mathbf{cfn}$, given some assumptions about the input streams $\mathbf{f1}$, \mathbf{nf} , and \mathbf{val} . In order to simplify the statement of the theorem (and the bisimulation used to prove it), we introduce the following stream valued functions (where $A, B \in \mathbf{B}^\infty$, $I \in \mathbf{Z}^\infty$, $i, j \in \mathbf{Z}$, and $a \in \mathbf{B}$):

$$\begin{aligned} \Theta(A, I, i) &= \mathbf{reg}(i, \mathbf{mux}(A, I, \Theta(A, I, i))) \\ \mathbf{Cfn}(A, B, I, i, j) &= \left\lfloor \frac{\Theta(A, I, i) + \Theta(B, I, j)}{2} \right\rfloor \\ \mathbf{Hold}(A, a) &= \mathbf{reg}(a, A \wedge \neg \mathbf{Hold}(A, a)) \\ \mathbf{Opt}(A, B, I, i) &= \mathbf{reg}(i, \mathbf{mux}(A, I, [B] + \mathbf{Opt}(A, B, I, i))) \end{aligned}$$

Theorem 8 *Let $R, f1, nf \in \mathbf{B}^\infty$, $val \in \mathbf{Z}^\infty$, $i \in \mathbf{Z}$, and $b \in \mathbf{B}$ such that the following properties hold:*

$$\mathit{Invariant}(R \Rightarrow \mathit{EQ}(\mathbf{tl}(val), val + 1)),$$

$$\mathit{Invariant}(\mathit{IF} R \mathit{ THEN } \neg \mathbf{tl}(f1) \mathit{ ELSE } f1 \Rightarrow \mathbf{tl}(f1) \mathit{ ENDIF}),$$

$$\mathit{Invariant}(\mathit{IF} R \mathit{ THEN } \neg \mathbf{tl}(nf) \mathit{ ELSE } nf \Rightarrow \mathbf{tl}(nf) \mathit{ ENDIF}),$$

$$\mathit{Invariant}(nf \Rightarrow f1),$$

$$\neg \mathbf{hd}(f1),$$

then

$$\mathbf{Cfn}(f1, nf, val, i, i) = \mathbf{Opt}(f1, \mathbf{Hold}(f1, b) \wedge \neg nf, val, i)$$

Proof: The proof is by coinduction on streams. Define relation \mathcal{R} between integer-valued streams such that $X \mathcal{R} Y$, if there exist $R, f1, nf \in \mathbf{B}^\infty$, $val \in \mathbf{Z}^\infty$, $i, j \in \mathbf{Z}$, and $b \in \mathbf{B}$ such that

Invariant($R \Rightarrow \text{EQ}(\text{tl}(\text{val}), \text{val} + 1)$),

Invariant(IF R THEN $\neg \text{tl}(f1)$ ELSE $f1 \Rightarrow \text{tl}(f1)$ ENDIF),

Invariant(IF R THEN $\neg \text{tl}(nf)$ ELSE $nf \Rightarrow \text{tl}(nf)$ ENDIF),

Invariant($nf \Rightarrow f1$),

$\text{hd}(f1) \wedge \neg \text{hd}(nf) \Rightarrow (\text{hd}(\text{val}) = j + 1) \wedge (b = \text{odd?}(i + j))$,

$X = \text{Cfn}(f1, nf, \text{val}, i, j)$, and

$Y = \text{Opt}(f1, (\text{Hold}(f1, b) \wedge \neg nf), \text{val}, \lfloor \frac{i + j}{2} \rfloor)$

The two streams that we wish to prove equivalent are contained in relation \mathcal{R} . Since $\neg \text{hd}(f1)$, there are no constraints on j or b . All that remains is to prove that \mathcal{R} is a bisimulation. Suppose we have streams X and Y such that $X \mathcal{R} Y$. There are two cases to consider

Heads: ($\text{hd}(X) = \text{hd}(Y)$)

$$\begin{aligned} \text{hd}(X) &= \text{hd}\left(\left\lfloor \frac{\Theta(f1', \text{val}', i') + \Theta(nf', \text{val}', j')}{2} \right\rfloor\right) \\ &= \left\lfloor \frac{\text{hd}(\Theta(f1', \text{val}', i')) + \text{hd}(\Theta(nf', \text{val}', j'))}{2} \right\rfloor \\ &= \left\lfloor \frac{i' + j'}{2} \right\rfloor \\ &= \text{hd}(Y) \end{aligned}$$

Tails: ($\text{tl}(X) \mathcal{R} \text{tl}(Y)$)

First, simplify the tails of both X and Y .

$$\text{tl}(X) = \text{tl}(\text{Cfn}(f1', nf', \text{val}', i', j'))$$

$$\begin{aligned}
&= \text{Cfn}(\text{tl}(fI'), \text{tl}(nf'), \text{tl}(val')), \\
&\quad \text{mux}(\text{hd}(fI'), \text{hd}(val'), i'), \\
&\quad \text{mux}(\text{hd}(nf'), \text{hd}(val'), j')
\end{aligned}$$

$$\begin{aligned}
\text{tl}(Y) &= \text{tl}(\text{Opt}(fI', (\text{Hold}(fI', b') \wedge \neg nf'), val', \left\lfloor \frac{i' + j'}{2} \right\rfloor)) \\
&= \text{Opt}(\text{tl}(fI'), \\
&\quad (\text{Hold}(\text{tl}(fI'), (\text{hd}(fI') \wedge \neg b')) \wedge \neg \text{tl}(nf')), \\
&\quad \text{tl}(val'), \\
&\quad \text{mux}(\text{hd}(fI'), \text{hd}(val'), [b' \wedge \neg \text{hd}(nf')] + \left\lfloor \frac{i' + j'}{2} \right\rfloor)
\end{aligned}$$

This provides us with the instantiations for the existentially quantified variables. We must now show that all the invariant properties are satisfied. The stream invariants are trivially satisfied. For the state invariants, we are left with the following goals:

$$\begin{aligned}
&\text{mux}(\text{hd}(fI'), \text{hd}(val'), [b' \wedge \neg \text{hd}(nf')] + \left\lfloor \frac{i' + j'}{2} \right\rfloor) \\
&= \left\lfloor \frac{\text{mux}(\text{hd}(fI'), \text{hd}(val'), i') + \text{mux}(\text{hd}(nf'), \text{hd}(val'), j')}{2} \right\rfloor
\end{aligned}$$

and

$$\begin{aligned}
&\text{hd}(\text{tl}(fI')) \wedge \neg \text{hd}(\text{tl}(nf')) \\
&\Rightarrow (\text{hd}(\text{tl}(val')) = \text{mux}(\text{hd}(nf'), \text{hd}(val'), j') + 1) \wedge \\
&\quad ((\text{hd}(fI') \wedge \neg b') = \\
&\quad \quad \text{odd}?(\text{mux}(\text{hd}(fI'), \text{hd}(val'), i' + \text{mux}(\text{hd}(nf'), \text{hd}(val'), j'))))
\end{aligned}$$

These goals are proven by expanding with all the definitions and using the

invariant properties at our disposal. +

5.6 Establishing Invariants

The verification presented above is only valid if the input signals satisfy the corresponding invariants. These are also established using coinductive proof. The signal val is generated by the sub-circuit shown in Figure 5.8. To justify the invariant

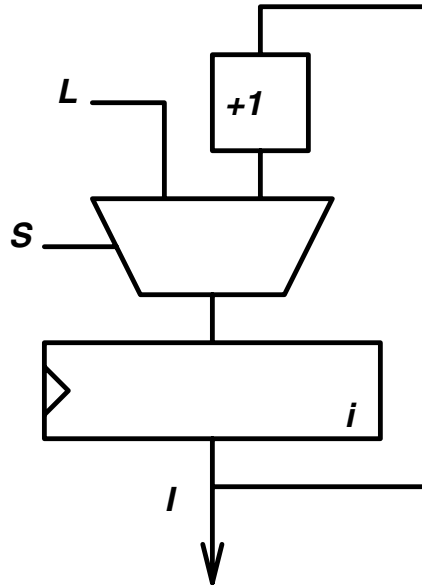


Figure 5.8: Counter Sub-Circuit

property on val in Theorem 8, we use the following result:

Lemma 6 For $R \in \mathbf{B}^\infty$, $L \in \mathbf{Z}^\infty$, and $i \in \mathbf{Z}$, if

$$val = \text{reg}(i, \text{mux}(R, L, val + 1))$$

then

Invariant($R \Rightarrow EQ(\text{tl}(\text{val}), \text{val} + 1)$),

Proof: In PVS, using strategy (co-induct-and-simplify). \dashv

So for the counter sub-circuit, the only remaining environmental constraint is that it be reset by the same control signal as the circuits generating streams $f1$ and nf . These streams have additional invariant properties that they must satisfy. These are established by the following result:

Lemma 7 For $R \in \mathbf{B}^\infty$, $\text{cnt} \in \mathbf{N}^\infty$, $b_1, b_2 \in \mathbf{B}$, if

Invariant(IF R THEN $EQ(\text{tl}(\text{cnt}), 0)$ ELSE $\text{tl}(\text{cnt}) \geq \text{cnt}$ ENDIF),

$$b_1 \Rightarrow \text{hd}(\text{cnt}) \geq (F + 1),$$

$$b_2 \Rightarrow \text{hd}(\text{cnt}) \geq (N - F),$$

$$b_2 \Rightarrow b_1$$

$$f1 = \text{reg}(b_1, \text{mux}(\neg R, \text{false}, (\text{cnt} \geq (F + 1))))), \text{ and}$$

$$nf = \text{reg}(b_2, \text{mux}(\neg R, \text{false}, (\text{cnt} \geq (N - F))))$$

then

Invariant(IF R THEN $\neg \text{tl}(f1)$ ELSE $f1 \Rightarrow \text{tl}(f1)$ ENDIF),

Invariant(IF R THEN $\neg \text{tl}(nf)$ ELSE $nf \Rightarrow \text{tl}(nf)$ ENDIF), and

Invariant($nf \Rightarrow f1$)

Proof: A suitable coinductive assertion for each conjunct is automatically generated by PVS strategy (co-induct-and-simplify), the constraints on cnt , b_1 , and b_2 are sufficient for brute force strategy (grind) to complete the proof. \dashv

In this case, justifying the invariants produces another obligation. Namely, that input stream `cnt` is nondecreasing between reset events. This obligation is outside the scope of the existing design, so it is not addressed here. A similar coinductive verification suffices.

5.7 Another verification of the optimization

In Theorem 8, the verification of Torres-Pomales' optimization is localized to just the affected sub-circuits. The inputs to this sub-circuit are constrained by coinductive assertions. An alternate approach is to include the surrounding circuitry in the statement of the theorem.

Theorem 9 For $R \in \mathbf{B}^\infty$, $\text{cnt} \in \mathbf{N}^\infty$, $L \in \mathbf{Z}^\infty$, $i, j, k \in \mathbf{Z}$, and $b, b_1, b_2 \in \mathbf{B}$, if

Invariant(IF R THEN EQ($\text{tl}(\text{cnt}), 0$) ELSE $\text{tl}(\text{cnt}) \geq \text{cnt}$ ENDIF),

$$b_1 \Rightarrow \text{hd}(\text{cnt}) \geq (F + 1),$$

$$b_2 \Rightarrow \text{hd}(\text{cnt}) \geq (N - F),$$

$$b_2 \Rightarrow b_1,$$

$$b_1 \wedge \neg b_2 \Rightarrow (k = j + 1) \wedge (b = \text{odd?}(i + j)),$$

$$fl = \text{reg}(b_1, \text{mux}(\neg R, \text{false}, (\text{cnt} \geq (F + 1))))), \text{ and}$$

$$nf = \text{reg}(b_2, \text{mux}(\neg R, \text{false}, (\text{cnt} \geq (N - F))))$$

and

$$val = \text{reg}(k, \text{mux}(\neg R, L, val + 1))$$

then

$$\text{Cfn}(f1, nf, val, i, j) = \text{Opt}(f1, \text{Hold}(f1, b) \wedge \neg nf, val, \left\lfloor \frac{(i+j)}{2} \right\rfloor)$$

Proof: Essentially the same as that for Theorem 8. The only real difference is that some invariant properties are encoded in the state variables, as opposed to the stream variables. PVS strategy (**co-induct-and-simplify**) guesses an appropriate bisimulation, and the verification reduces to showing that the state invariants are preserved. \dashv

One advantage of Theorem 8 is that the verification is independent of the number of clocks participating in the protocol. The replacement can also be justified by including the counter sub-circuit, but leaving input streams $f1$ and nf abstractly specified. By pushing these signals into another portion of the design, we develop a core circuit that may be used in a number of different fault-tolerant architectures.

5.8 VLSI implementation

After incorporating the above refinements, the circuit was further refined using some basic transformations on stream equations. The resulting circuit, depicted in Figure 5.9, has been fabricated and tested. Bob Burger participated in the development of the layout, simulation, and initial testing of the VLSI implementation.

There were two phases of testing. The initial tests were conducted using a Logic Engine (LE) as the test environment [WC91]. Test signals were generated using a Scheme interface to the LE written by M. Esen Tuna. The test cases varied the length of the synchronization interval, varied the arrival times of signals from remote clocks, and exercised the full ranges of the counters on the chip. The circuit performed as expected on all test cases.

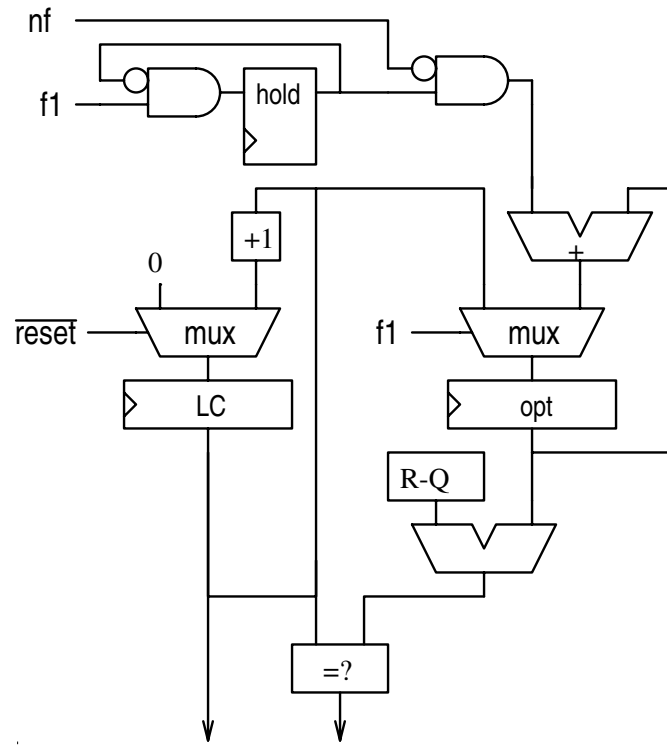


Figure 5.9: Core Synchronization Circuit

The second phase of testing consisted of incorporating the design in Torres-Pomales’ four node synchronization system. Three of the nodes were Torres-Pomales’ original design [TP93], the fourth was the VLSI implementation described in this chapter. The system ran at 10Mhz. We ran the same suite of tests as Torres-Pomales had performed on his implementation of the optimized synchronization circuit [TP95]. There are three main test cases: initialization, recovery from a single transient fault, and recovery from massively correlated transient faults. The VLSI implementation performed perfectly on all tests. The system maintained synchronization within 300 nanoseconds.

6

Floating Point Division

In contrast to the clock synchronization circuit, where a driving factor in the design was reducing the required chip area, current hardware for floating point arithmetic seeks to improve performance. Often, by using redundant encodings for intermediate results, cycle time can be reduced, thus increasing the speed of the operation. The tradeoff is that these redundant representations consume more chip area. This chapter presents the formal development of an IEEE compliant floating point division unit. The same verification structure is in force. At the top level of the verification hierarchy is a formal definition of an IEEE standard for floating point arithmetic. An algorithm for performing floating point division is verified with respect to the standard using PVS. This algorithm forms the behavioral specification for the floating point division unit. A hardware architecture is derived using DRS. Four local optimizations are explored. The first is derived within DRS, with the exception of a few combinational correctness conditions. The other optimizations are verified using coinduction. Two different approaches are illustrated on the final optimization.

6.1 IEEE Floating-Point Arithmetic

The top level in this development is a formalization of the IEEE-854 Standard for Radix-Independent Floating-Point Arithmetic using PVS [IEE87, Min95]. This formalization encompasses most of the standard, including definition of the four rounding modes. The standards require support for addition, subtraction, multiplication, division, and square root with the requirement that each operation

shall be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then that result rounded according to one of the modes ... [IEE85, IEE87]

This formalization has been extended to include a verification of a standard algorithm for providing IEEE compliant rounding [ML96]. The next section presents a generalized subtractive division algorithm which, when composed with the rounding algorithm, provides IEEE compliant division.

6.2 The Verified Algorithm

The class of subtractive division algorithms consists of several generalizations of the long division algorithm taught in elementary school. By carefully relaxing the precision in selecting each quotient digit, it is possible to develop algorithms with more efficient hardware realizations. Errors introduced by the imprecision are corrected by subsequent iterations of the algorithm. Ercegovic and Lang present several interrelated factors that affect the design of subtractive division algorithms [EL94]. These include:

1. Radix r —If b is the radix of the underlying floating-point number system, the division radix is chosen so that $r = b^k$ for some positive integer k .
2. Quotient digit set— $q_i \in \{-a, -a + 1, \dots, -1, 0, 1, \dots, a - 1, a\}$ where $\lceil \frac{r-1}{2} \rceil \leq a < r$.
3. Representation of the partial remainder.
4. Range restriction of the divisor.
5. Quotient-digit selection function.

Of these, all but the third are defined as parameters to the PVS formalization of the general division algorithm. Variations in the representation of the partial remainder are best addressed at the hardware architecture level. This is discussed in Section 6.4. The formal parameters for theory *general_division* are

```

r  : {i : posint | i > 1},
a  : {i : posint | ceiling((r-1)/2) <= i & i < r},
b  : {i : posint | 1 < i & i <= r},
(IMPORTING divide_types[r, a, b])
qs : qs_type

```

The first three parameters are defined using predicate subtypes of the positive integers. In addition, the types for a and b are constrained by the value of r . The type signature for function qs depends on all of the previous parameters. The declaration for type qs_type is imported from theory *divide_types*. Theory *divide_types* declares constant $\rho = \frac{a}{r-1}$ and the following types:

```

D_type      : TYPE = {d : real      | 1 <= d & d < b}

```

```

p_type(D)    : TYPE = {p : real    | abs(p) <= rho*D}
dividend(D)  : TYPE = {x : posreal | 1/r <= x & x < rho*D}
qs_type      : TYPE =
    [D : D_type, p : p_type(D) ->
     {q : subrange(-a,a) | abs(r*p - q*D) <= rho*D}]

```

Constant ρ denotes the redundancy factor of the quotient-digit set. It represents a trade-off in design complexity between the quotient selection function and generation of divisor multiples. The type of the divisor is constrained by D_type , which is defined to include the numeric range of the significand of a normalized floating point number. Parameterized type $p_type(D)$ encodes an invariant, dependent on divisor D , that the partial remainder must satisfy during execution of the algorithm. The type of the dividend, also parameterized by D , is constrained to ensure that the dividend satisfies the constraints imposed on the partial remainder. Finally, the type of the quotient selection function, qs_type , is restricted to functions that, given a divisor D and a partial remainder p such that $|p| \leq \rho \cdot D$, return a digit q such that $-a \leq q \leq a$ and $|r \cdot p - q \cdot D| \leq \rho \cdot D$.

The subtractive division algorithms, given divisor D and dividend X , are characterized by the following recurrence equations for the quotient q and the partial remainder p :

$$\begin{aligned}
 p_i &= r \cdot p_{i-1} - qd_i \cdot D \\
 q_i &= r \cdot q_{i-1} + qd_i
 \end{aligned}$$

where $q_0 = 0$, $p_0 = X$, and qd_i is the quotient digit selected for iteration i . A PVS specification of these equations is

```

divide((D : D_type), (X : dividend(D)))(n : nat) :

```

```

RECURSIVE [# p : p_type(D), q: integer #] =
  IF n=0 THEN (# p := X, q := 0 #)
    ELSE (# p := r*p(divide(D,X)(n-1)) - qd_n*D,
          q := r*q(divide(D,X)(n-1)) + qd_n #)
    WHERE qd_n = qs(D,p(divide(D,X)(n-1))) ENDIF
MEASURE n

```

By using a record type for the range of the function, the definition is a direct translation of the recurrence equations. In addition, by declaring the partial remainder to be of type $p_type(D)$, PVS automatically generates a proof obligation to ensure that the invariant is satisfied. This obligation is proven using the type constraints on the quotient selection function.

6.3 Mapping the algorithm to a design

In order to derive division hardware, we first have to convert the algorithm into tail-recursive form. We know that such a form exists, because the algorithm consists of a simple recursion on the natural numbers. The following algorithm suffices:

```

drs_div((n : nat), (D : D_type), (pr : p_type(D)), (qa : int)):
  RECURSIVE [# p : p_type(D), q : integer #] =
    IF n = 0 THEN
      (# p := pr, q := qa #)
    ELSE
      drs_div(n - 1, D,
              r * pr - q_sel(D, pr) * D,

```



```

                r * qa + q_sel(D, pr))
    ENDIF
    MEASURE n

```

The correctness result is:

Theorem 10 $drs_div(n, D, X, 0) = divide(D, X)(n)$

To prove this, we introduce the following result:

Lemma 8 For $n \in \mathbf{N}$, $Y : \alpha$ and $h : \alpha \rightarrow \alpha$, let

$$F_h(Y)(n) = \begin{cases} Y & \text{if } n = 0 \\ h(F(Y)(n-1)) & \text{otherwise} \end{cases}$$

and

$$G_h(Y)(n) = \begin{cases} Y & \text{if } n = 0 \\ G(h(Y))(n-1) & \text{otherwise} \end{cases}$$

then

$$F(Y)(n) = G(Y)(n)$$

Proof: Simple proofs by induction give us:

$$F_h(Y)(n) = h^n(Y)$$

$$G_h(Y)(n) = h^n(Y)$$

Both equations were verified in PVS using built-in strategy (*induct-and-simplify*), which automatically generates simple inductive proofs. \dashv

Proof: (of Theorem 10) Define function h_D

$$\begin{aligned} h_D((\# p := p_1, q := q_1 \#)) &\hat{=} (\# p := r \cdot p_1 - qs(D, p_1) \cdot D, \\ &q := r \cdot q_1 + qs(D, p) \#) \end{aligned}$$

and

$$f_{(D,X)} \hat{=} (\# p := X, q := 0 \#)$$

then induction gives us:

$$\text{divide}(D, X)(n) = F_{h_D}(f_{(D,X)})(n)$$

and for any p_1, q_1

$$\text{drs_div}(n, D, p_1, q_1) = G_{h_D}((\# p := p_1, q := q_1 \#))(n)$$

By lemma 8, $G_{h_D}(f_{(D,X)})(n) = F_{h_D}(f_{(D,X)})(n)$, therefore

$$\text{drs_div}(n, D, X, 0) = \text{divide}(D, X)(n)$$

\dashv

This algorithm was manually translated into a DRS behavioral level specification. This is then immediately transformed into a structural level description using DRS. Figure 6.1 depicts the initial partial remainder formation sub-circuit and figure 6.2 depicts the initial quotient accumulation sub-circuit. The stream equations

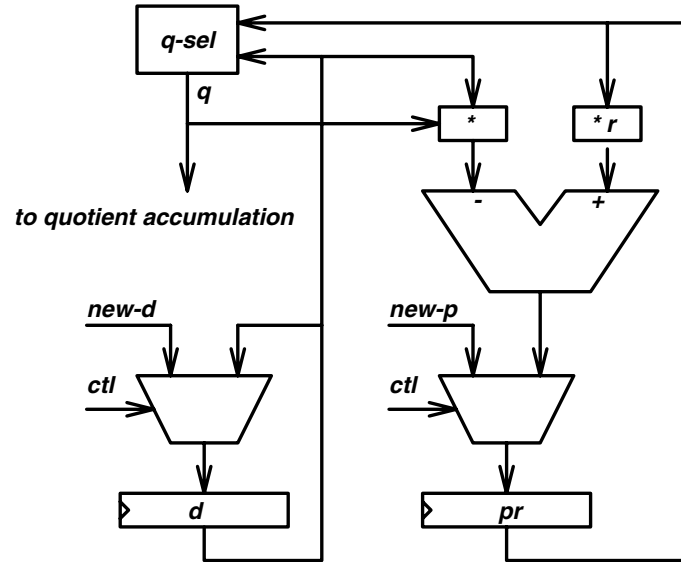


Figure 6.1: Initial Partial Remainder Sub-Circuit

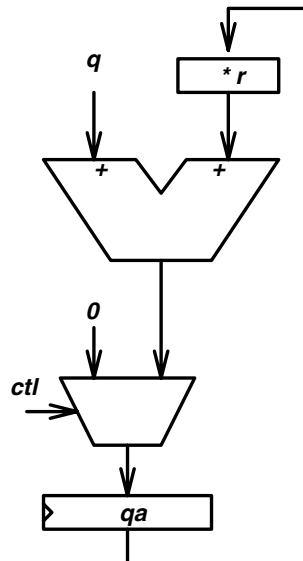


Figure 6.2: Initial Quotient Accumulation Sub-Circuit

corresponding to the partial remainder sub-circuit are:

$$\begin{aligned}d &= \text{reg} (?, \text{sel}(\text{ctl}, \text{new-d}, d)) \\ \text{pr} &= \text{reg} (?, \text{sel}(\text{ctl}, \text{new-p}, ((r \times \text{pr}) - (q \times d)))) \\ q &= \text{q-sel}(d, \text{pr})\end{aligned}$$

The quotient accumulation sub-circuit consists of the stream equation:

$$\text{qa} = \text{reg} (?, \text{sel}(\text{ctl}, 0, ((r \times \text{qa}) + q)))$$

Function `reg` is a standard unit delay element and `sel` is a selector function controlled by signal `ctl`. The symbol `?` denotes *don't care*. Section 6.4 presents two techniques to optimize the partial remainder sub-circuit and section 6.5 presents two optimizations to the quotient accumulation sub-circuit.

6.4 Partial Remainder Formation

There are two common optimizations for the partial remainder formation sub-circuit of subtractive division algorithms. The first predicts the next quotient digit using an approximation of the next partial remainder. This is accomplished by pipelining the operation and computing the next quotient digit in parallel with computing the next remainder.

Another common optimization is to use a redundant representation of the partial remainder. This eliminates the propagation delay due to computing the carry-chain in the adder for the inner loop of the algorithm.

Quotient Digit Prediction

The derivation of the architecture employing quotient digit prediction consists primarily of standard DRS transformations. Coinductive proof techniques are not necessary to complete the refinement. The correctness conditions generated by DRS reflect modifications to the combinational logic of the sub-circuit; these are discharged directly using arithmetic decision procedures. The derivation proceeds from the stream equations defining the partial remainder sub-circuit:

$$\begin{aligned} d &= \text{reg} (?, \text{sel}(\text{ctl}, \text{new-d}, d)) \\ \text{pr} &= \text{reg} (?, \text{sel}(\text{ctl}, \text{new-p}, ((r \times \text{pr}) - (q \times d)))) \\ q &= \text{q-sel}(d, \text{pr}) \end{aligned}$$

The first step is to replace **new-p** with the arithmetically equivalent term $(\text{new-p} - 0)$.

$$\begin{aligned} d &= \text{reg} (?, \text{sel}(\text{ctl}, \text{new-d}, d)) \\ \text{pr} &= \text{reg} (?, \text{sel}(\text{ctl}, (\text{new-p} - 0), ((r \times \text{pr}) - (q \times d)))) \\ q &= \text{q-sel}(d, \text{pr}) \end{aligned}$$

Next, distribute the selector through the subtraction operations, and introduce names **pp** and **dz** for the arguments to the subtraction operator.

$$\begin{aligned} d &= \text{reg} (?, \text{sel}(\text{ctl}, \text{new-d}, d)) \\ \text{pr} &= \text{reg} (?, (\text{pp} - \text{dz})) \\ q &= \text{q-sel}(d, \text{pr}) \\ \text{pp} &= \text{sel}(\text{ctl}, \text{new-p}, (r \times \text{pr})) \\ \text{dz} &= \text{sel}(\text{ctl}, 0, (q \times d)) \end{aligned}$$

In anticipation of the next transformation, we introduce the name **d1** for the signal loading the **d** register.

$$\begin{aligned}
 \mathbf{d} &= \text{reg}(\mathbf{?}, \mathbf{d1}) \\
 \mathbf{pr} &= \text{reg}(\mathbf{?}, (\mathbf{pp} - \mathbf{dz})) \\
 \mathbf{q} &= \text{q-sel}(\mathbf{d}, \mathbf{pr}) \\
 \mathbf{pp} &= \text{sel}(\text{ctl}, \text{new-p}, (\mathbf{r} \times \mathbf{pr})) \\
 \mathbf{dz} &= \text{sel}(\text{ctl}, 0, (\mathbf{q} \times \mathbf{d})) \\
 \mathbf{d1} &= \text{sel}(\text{ctl}, \text{new-d}, \mathbf{d})
 \end{aligned}$$

Expand with the definitions of **d** and **pr** in stream **q**.

$$\begin{aligned}
 \mathbf{d} &= \text{reg}(\mathbf{?}, \mathbf{d1}) \\
 \mathbf{pr} &= \text{reg}(\mathbf{?}, (\mathbf{pp} - \mathbf{dz})) \\
 \mathbf{q} &= \text{q-sel}(\text{reg}(\mathbf{?}, \mathbf{d1}), \text{reg}(\mathbf{?}, (\mathbf{pp} - \mathbf{dz}))) \\
 \mathbf{pp} &= \text{sel}(\text{ctl}, \text{new-p}, (\mathbf{r} \times \mathbf{pr})) \\
 \mathbf{dz} &= \text{sel}(\text{ctl}, 0, (\mathbf{q} \times \mathbf{d})) \\
 \mathbf{d1} &= \text{sel}(\text{ctl}, \text{new-d}, \mathbf{d})
 \end{aligned}$$

Re-time stream equation **q**, distributing the application of **q-sel** through the registers and simplify the resulting application **q-sel**(**?**, **?**) to **?**.

$$\begin{aligned}
 \mathbf{d} &= \text{reg}(\mathbf{?}, \mathbf{d1}) \\
 \mathbf{pr} &= \text{reg}(\mathbf{?}, (\mathbf{pp} - \mathbf{dz})) \\
 \mathbf{q} &= \text{reg}(\mathbf{?}, \text{q-sel}(\mathbf{d1}, (\mathbf{pp} - \mathbf{dz}))) \\
 \mathbf{pp} &= \text{sel}(\text{ctl}, \text{new-p}, (\mathbf{r} \times \mathbf{pr})) \\
 \mathbf{dz} &= \text{sel}(\text{ctl}, 0, (\mathbf{q} \times \mathbf{d}))
 \end{aligned}$$

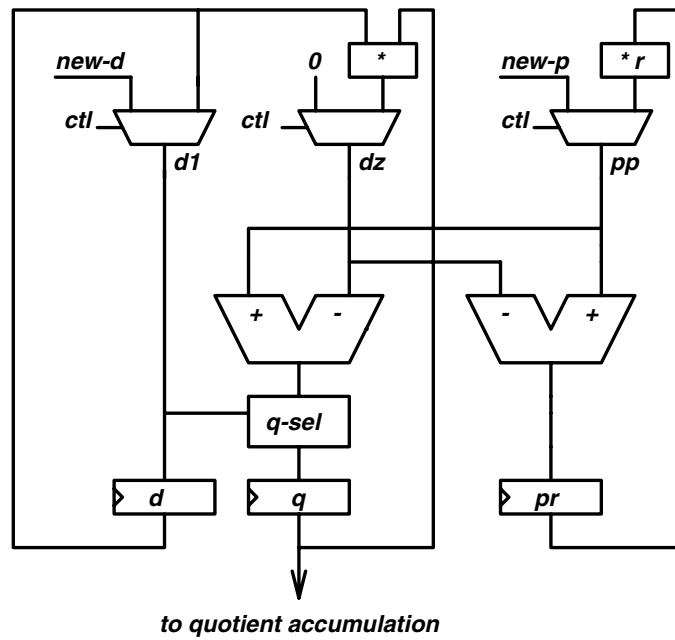


Figure 6.3: Derived Architecture

$$d1 = \text{sel}(\text{ctl}, \text{new-d}, d)$$

This system of stream equations corresponds to the derived architecture depicted in Figure 6.3. Thus far, the derivation steps have not depended on the radix, or the realization of the quotient selection function. The remainder of the derivation assumes a radix-4 algorithm ($r = 4$), and Taylor’s lookup table for the computation of quotient digits.

At this point, we exploit the fact that the lookup table only requires an approximation of the partial remainder and replace the adder feeding the quotient selection function with one that sums only eight bits. The signal and function names introduced for the remainder of this derivation were selected to correspond to signal names in the Clarke, German, Zhao verification of Taylor’s circuit [CGZ96].

The subtraction feeding the register `pr` is replaced with a full precision alu (`dalu`),

controlled by the sign bit of the selected quotient digit. This replacement is verified using PVS strategy (`grind`), the definition of signal `dz` and knowledge that the divisor `d` is nonnegative. The PVS statement of the correctness conditions from this derivation are available in the electronic repository [WWW].

```

d = reg (?, d1)
pr = reg (?, dalu((q < 0), pp, |dz|))
q = reg (?, q-sel(d1, (pp - dz)))
pp = sel(ctl, new-p, (r × pr))
dz = sel(ctl, 0, (q × d))
d1 = sel(ctl, new-d, d)

```

Next, expand the definition of `q-sel`, revealing an invocation of a lookup table that only uses an approximation of the partial remainder, further truncated to have only six bits of precision to the right of the binary point.

```

d = reg (?, d1)
pr = reg (?, dalu((q < 0), pp, |dz|))
q = reg (?, lookup(d1, approx-6(approx((pp - dz))))))
pp = sel(ctl, new-p, (r × pr))
dz = sel(ctl, 0, (q × d))
d1 = sel(ctl, new-d, d)

```

A two stage verification in PVS suffices to justify replacing the approximation with a *guess* alu (`galu`), which computes an 8-bit approximation of the next partial remainder. Each verification stage in PVS is discharged using the (`grind`) proof strategy.

Each argument to `galu` is truncated six bits to the right of the radix point. Signal `pp` requires four bits to the left of the radix point, for a total of ten bits. However, the leading two bits of `pp` have no effect on the result, which, by the invariant property of the partial remainder (page 99), only requires two bits to the left of the radix point. Thus, the `galu` need only sum eight bits.

```

d = reg(?, d1)
pr = reg(?, dalu((q < 0), pp, md))
q = reg(?, lookup(d1, galu((q < 0), approx-6(pp), approx-6(md))))
pp = sel(ctl, new-p, (r × pr))
dz = sel(ctl, 0, (q × d))
d1 = sel(ctl, new-d, d)
md = |dz|

```

The derivation proceeds by introducing names for some of the signals, and distributing the absolute value function through the definition of signal `md`. The stream equations for the final architecture depicted in figure 6.4 are:

```

d = reg(?, d1)
pr = reg(?, dalu(qsign, pp, md))
q = reg(?, lookup(d1, pr1))
pp = sel(ctl, new-p, (r × pr))
d1 = sel(ctl, new-d, d)
md = sel(ctl, 0, (|q| × d))
pr1 = galu(qsign, approx-6(pp), approx-6(md))
qsign = (q < 0)

```

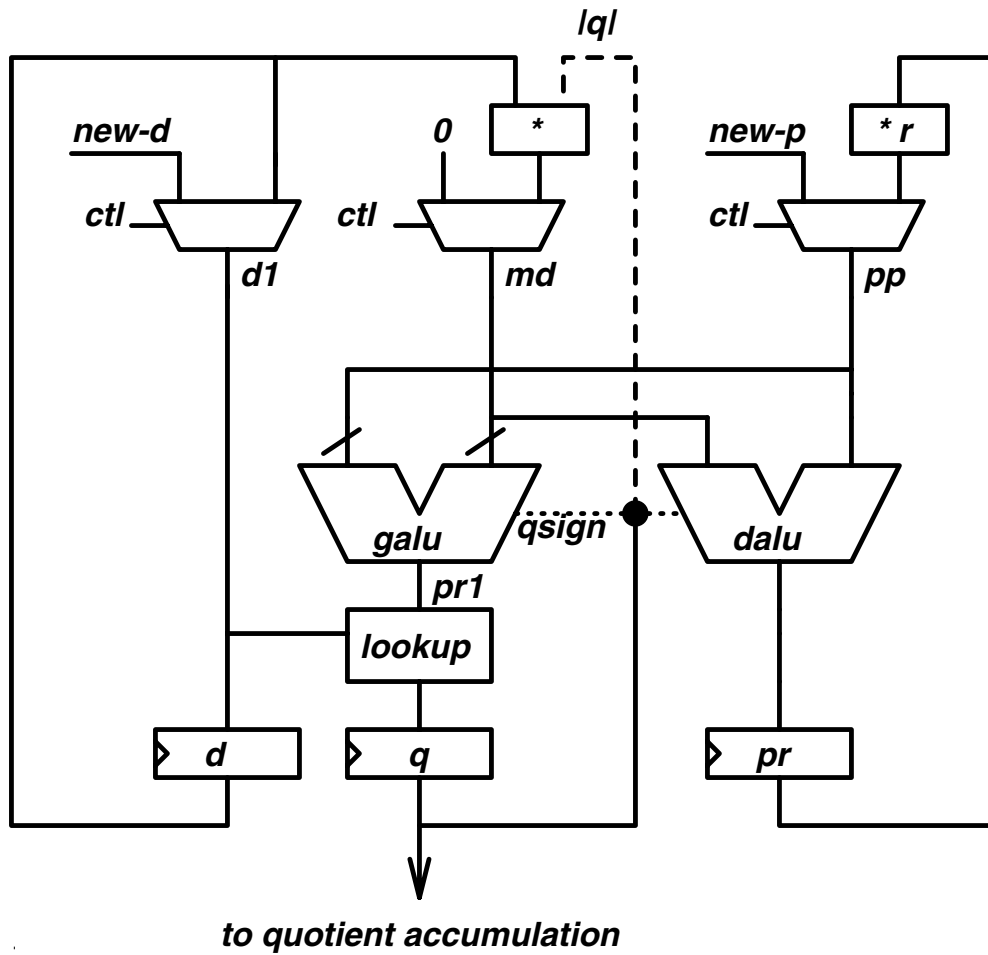


Figure 6.4: Final Partial Remainder Sub-Circuit

Several of the functions in this system of stream equations do not require any special hardware to compute. Signal `qsign` corresponds to the sign bit of `q`. Also, if `q` is represented using a sign-and-magnitude format, `|q|` corresponds to the magnitude bits. Multiplying by radix-`r` is implemented using a hard-wired shift operation. In addition, once the design is mapped to a boolean representation, the computation of signal `md` is optimized. From the definition of the lookup table, `q` is restricted so that $|q| \leq 2$. Therefore, `md` is either 0, `d`, or $2 \times d$. This multiplication is typically realized using a hardwired shift operation, so the computation of the multiple of the divisor is realized using a selector. The critical path in this circuit is the cycle from `pr` to `pr`. The delay is dominated by the computation of the carry chain in the `dal`. The next optimization illustrates how to eliminate the carry-chain in the computation of the partial remainder.

Carry-Free Adders

For extended precision floating point numbers, a more efficient optimization is to use a redundant representation of the partial remainder. The most common redundant representation is to store the partial remainder in a carry-save format. This allows for using a faster clock, because the delay computing the combinational logic is independent of the word length. The standard carry-propagate adder takes two inputs and produces a single output. Carry-free adders take three inputs (`x`, `y`, and `z`) and produce two outputs (`c` and `s`) such that $x + y + z = c + s$.

Within DRS we want to replace the stream equation defining `pr` (illustrated in figure 6.5)

$$\text{pr} = \text{reg}(\text{?, sel}(\text{ctl}, \text{new-p}, ((r \times \text{pr}) - (q \times d))))$$

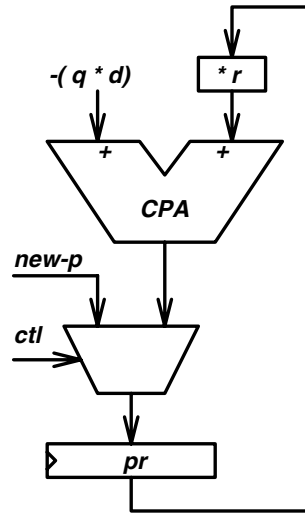


Figure 6.5: Partial Remainder Using a Carry-Propagate Adder

with a redundant representation of the partial remainder (figure 6.6).

The starting point of this derivation is the following pair of streams, denoting all cycles in the logic determining the partial remainder. The goal is to minimize the combinational logic delay.

$$\begin{aligned} pr &= \text{reg}(\text{?}, \text{sel}(\text{ctl}, \text{new-p}, ((r \times pr) - (q \times d)))) \\ q &= \text{q-sel}(d, pr) \end{aligned}$$

The first step is to introduce the stream equations corresponding to the carry-free adder accumulation of the partial remainder.

$$\begin{aligned} pr &= \text{reg}(\text{?}, \text{sel}(\text{ctl}, \text{new-p}, ((r \times pr) - (q \times d)))) \\ q &= \text{q-sel}(d, pr) \\ pr1 &= \text{reg}(\text{?}, \text{sel}(\text{ctl}, \text{new-p}, \text{cfa1}(-((q \times d)), (r \times pr1), (r \times pr2)))) \\ pr2 &= \text{reg}(\text{?}, \text{sel}(\text{ctl}, 0, \text{cfa2}(-((q \times d)), (r \times pr1), (r \times pr2)))) \end{aligned}$$

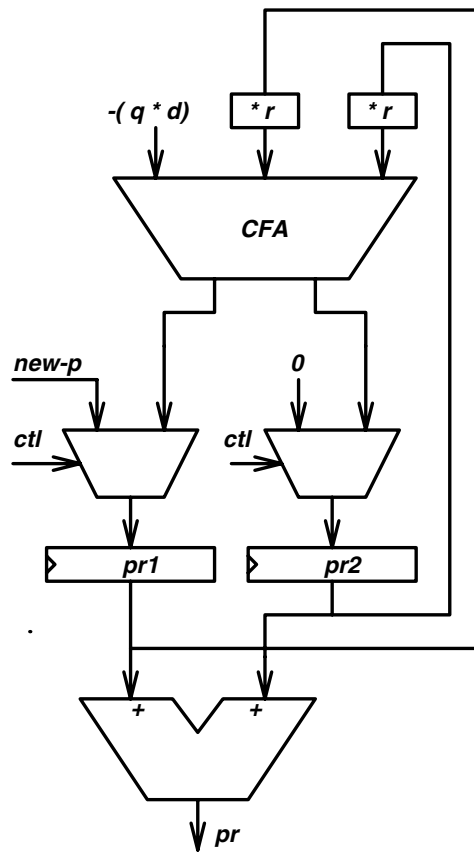


Figure 6.6: Redundant Partial Remainder Using a Carry-Free Adder

Then, replace pr with $(\text{pr1} + \text{pr2})$.

$$\begin{aligned} \text{q} &= \text{q-sel}(\text{d}, (\text{pr1} + \text{pr2})) \\ \text{pr1} &= \text{reg}(\text{?}, \text{sel}(\text{ctl}, \text{new-p}, \text{cfa1}(-((\text{q} \times \text{d})), (\text{r} \times \text{pr1}), (\text{r} \times \text{pr2})))) \\ \text{pr2} &= \text{reg}(\text{?}, \text{sel}(\text{ctl}, 0, \text{cfa2}(-((\text{q} \times \text{d})), (\text{r} \times \text{pr1}), (\text{r} \times \text{pr2})))) \end{aligned}$$

This replacement is justified by the following result:

Theorem 11 *For rational i, j , if*

$$\begin{aligned} \text{pr} &= \text{reg}(i + j, \text{sel}(\text{ctl}, \text{new-p}, ((\text{r} \times \text{pr}) - (\text{q} \times \text{d}))), \\ \text{pr1} &= \text{reg}(i, \text{sel}(\text{ctl}, \text{new-p}, \text{cfa1}(-((\text{q} \times \text{d})), (\text{r} \times \text{pr1}), (\text{r} \times \text{pr2})))) \textit{, and} \\ \text{pr2} &= \text{reg}(j, \text{sel}(\text{ctl}, 0, \text{cfa2}(-((\text{q} \times \text{d})), (\text{r} \times \text{pr1}), (\text{r} \times \text{pr2})))) \end{aligned}$$

then

$$\text{pr} = (\text{pr1} + \text{pr2})$$

Proof: Verified in PVS using strategy `(co-induct-and-simplify)`. +

This verification exploits the fact that the head of pr is a don't care. We are able to replace it with any value, and preserve the behavior of the system. The term $i + j$ is the obvious choice, and suggests the state invariant for the automatically generated bisimulation.

The derivation proceeds by replacing the full precision carry-propagate adder feeding the quotient selection function with an 8-bit approximation, using arguments similar to those for quotient digit prediction. The final system of stream equations

is:

$$\begin{aligned} q &= \text{lookup}(d, (\text{approx-6}(\text{pr1}) + \text{approx-6}(\text{pr2}))) \\ \text{pr1} &= \text{reg}(\text{?}, \text{sel}(\text{ctl}, \text{new-p}, \text{cfa1}(-((q \times d)), (r \times \text{pr1}), (r \times \text{pr2})))) \\ \text{pr2} &= \text{reg}(\text{?}, \text{sel}(\text{ctl}, 0, \text{cfa2}(-((q \times d)), (r \times \text{pr1}), (r \times \text{pr2})))) \end{aligned}$$

6.5 Quotient Accumulation

Just as in the partial remainder formation, the carry chain is the critical path in the quotient accumulation sub-circuit. One technique is to accumulate the magnitudes of the positive and negative quotient digits in separate registers using a shift-append operation. After the final quotient digit is received, the absolute value of the accumulated negative digits is subtracted from the accumulated positive digits.

A more recent strategy, *on-the-fly* conversion [EL94], maintains a decremented copy of the accumulated quotient, so that the correct quotient is computed using a shift-append operation. Each of these optimizations is verified using coinduction.

Redundant Representation

The first technique for improving the performance of the quotient accumulation sub-circuit was presented in Taylor [Tay81]. The quotient digits are represented using a sign and magnitude representation. For a radix- r algorithm, $|q| < r$. Depending upon the sign of the quotient digit, the magnitude is appended to either a positive or negative shift register. This shift-append operation has a critical path that is independent of the word length of the floating point numbers. The naive implementation

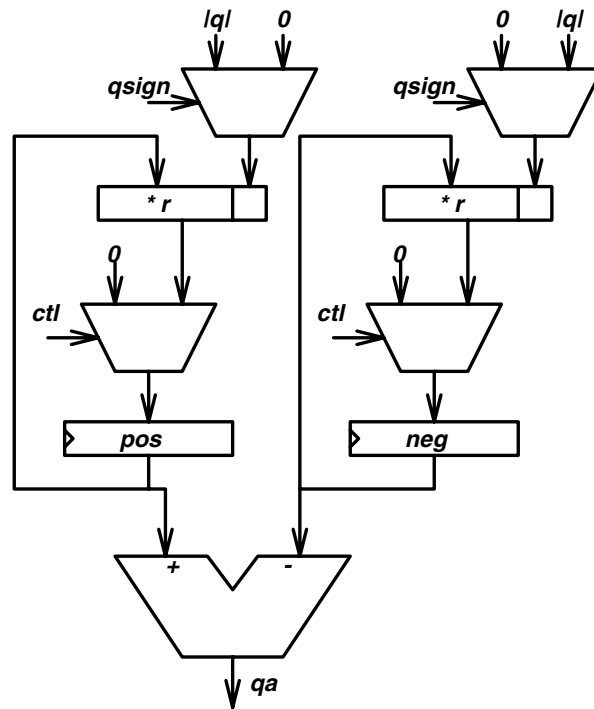


Figure 6.7: Redundant Representation of Accumulated Quotient

uses a carry-propagate adder with a critical path of $O(\log_2 n)$ gate-delays for an n -bit word.

The goal is to replace stream equation (shown in figure 6.2)

$$qa = \text{reg} (?, \text{sel}(\text{ctl}, 0, ((r \times qa) + q)))$$

with the system of equations (shown in figure 6.7)

$$\text{neg} = \text{reg} (?, \text{sel}(\text{ctl}, 0, \text{shift-append}(\text{neg}, \text{mux}((q < 0), 0, |q|))))$$

$$\text{pos} = \text{reg} (?, \text{sel}(\text{ctl}, 0, \text{shift-append}(\text{pos}, \text{mux}((q < 0), |q|, 0))))$$

$$qa = (\text{pos} - \text{neg})$$

The replacement is justified by the following result:

Theorem 12 For $i, j \in \mathbf{Z}$, if

$$\mathbf{qa} = \mathbf{reg}(i - j, \mathbf{sel}(\mathbf{ctl}, 0, ((\mathbf{r} \times \mathbf{qa}) + \mathbf{q}))),$$

$$\mathbf{neg} = \mathbf{reg}(j, \mathbf{sel}(\mathbf{ctl}, 0, \mathbf{shift-append}(\mathbf{neg}, \mathbf{mux}((\mathbf{q} < 0), 0, |\mathbf{q}|))), \text{ and}$$

$$\mathbf{pos} = \mathbf{reg}(i, \mathbf{sel}(\mathbf{ctl}, 0, \mathbf{shift-append}(\mathbf{pos}, \mathbf{mux}((\mathbf{q} < 0), |\mathbf{q}|, 0))))$$

then

$$\mathbf{qa} = (\mathbf{pos} - \mathbf{neg})$$

Proof: Verified in PVS using strategy (co-induct-and-simplify). +

This approach eliminates the carry chain while accumulating the quotient, but still requires a full precision, carry-propagate operation to deliver the final quotient. The standard technique for computing the final quotient within the constraints of the shorter cycle time is to allow two clock cycles for the carry to propagate. This requires modifying the controlling state-machine to introduce an additional delay, before the result is used.

The next approach computes the accumulated quotient as each quotient digit arrives, and completely eliminates the need for a carry-propagate adder in the quotient accumulation logic.

On-the-fly Quotient Accumulation

This approach for quotient accumulation is due to Ercegovac and Lang [EL94]. The idea is to maintain two registers, one containing the current value of the accumulated quotient, qa , and the other containing $qa - 1$. In order to efficiently accumulate the quotient digits as they arrive, we define two functions:

$$\begin{aligned} \mathbf{qa-step}(q, qa, qm) &= \begin{cases} r \cdot qm + (r + q) & , \text{ if } q < 0 \\ r \cdot qa + q & , \text{ otherwise} \end{cases} \\ \mathbf{qm-step}(q, qa, qm) &= \begin{cases} r \cdot qa + (q - 1) & , \text{ if } q > 0 \\ r \cdot qm + (r + q - 1) & , \text{ otherwise} \end{cases} \end{aligned}$$

Since $|q| < r$, these functions are implemented by appending the computed radix- r digit to an appropriately selected shifted value. This is a constant time operation. It is independent of the word length. In addition, these functions satisfy the following equations:

$$\begin{aligned} \mathbf{qa-step}(q, qa, qa - 1) &= (r \cdot qa) + q \\ \mathbf{qm-step}(q, qa, qa - 1) &= (r \cdot qa) + q - 1 \\ &= \mathbf{qa-step}(q, qa, qa - 1) - 1 \end{aligned}$$

Two different design paths for on-the-fly conversion are presented. The first uses derivation strategies to refine the original sub-circuit. The second approach uses a coinductive proof to directly replace the original sub-circuit with a compact realization of on-the-fly conversion.

Derived on-the-fly conversion

This section illustrates how to use a derivation system to refine the initial quotient accumulation sub-circuit to a more efficient design using on-the-fly conversion.

$$qa = \text{reg} (?, \text{sel}(\text{ctl}, 0, ((r \times qa) + q)))$$

The first step is to replace $((r \times qa) + q)$ with $qa\text{-step}(q, qa, (qa - 1))$ and introduce the name qm for $(qa - 1)$.

$$qa = \text{reg} (?, \text{sel}(\text{ctl}, 0, qa\text{-step}(q, qa, qm)))$$

$$qm = (qa - 1)$$

Expanding with the definition of qa , re-timing, and lifting the selector yields

$$qa = \text{reg} (?, \text{sel}(\text{ctl}, 0, qa\text{-step}(q, qa, qm)))$$

$$qm = \text{reg} ((? - 1), \text{sel}(\text{ctl}, (-1), (qa\text{-step}(q, qa, qm) - 1)))$$

The three previous transformations strictly preserve equality, so we still have the knowledge that $qm = qa - 1$. Using the properties of $qa\text{-step}$ and $qm\text{-step}$ (page 117) we replace $(qa\text{-step}(q, qa, qm) - 1)$ with $qm\text{-step}(q, qa, qm)$

$$qa = \text{reg} (?, \text{sel}(\text{ctl}, 0, qa\text{-step}(q, qa, qm)))$$

$$qm = \text{reg} ((? - 1), \text{sel}(\text{ctl}, (-1), qm\text{-step}(q, qa, qm)))$$

Inspection of the ctl stream justifies replacing term $(? - 1)$ with $?$, resulting in the following pair of stream equations defining the quotient accumulation logic.

$$qa = \text{reg} (?, \text{sel}(\text{ctl}, 0, qa\text{-step}(q, qa, qm)))$$

$$\mathbf{qm} = \text{reg}(\text{?}, \text{sel}(\text{ctl}, (-1), \mathbf{qm}\text{-step}(\mathbf{q}, \mathbf{qa}, \mathbf{qm})))$$

This is a very efficient circuit for quotient accumulation, but it includes some unnecessary logic. There is no need to reset the \mathbf{qm} register to (-1) when we initiate a division operation. Since the division algorithm is always invoked with positive operands, there will always be at least one positive quotient digit before there is a negative quotient digit. Therefore, the \mathbf{qm} register does not need to be initialized. Justifying this optimization requires the use of coinductive verification techniques.

Coinductive verification

Theorem 13 *Let ctl be a stream of control signals; let \mathbf{q} be a stream of integers, such that after each reset (controlled by ctl), the first nonzero digit is positive; and let \mathbf{qa} and \mathbf{qm} be defined by the following stream equations:*

$$\mathbf{qa} = \text{reg}(i, \text{sel}(\text{ctl}, 0, ((\mathbf{r} \times \mathbf{qa}) + \mathbf{q})))$$

$$\mathbf{qm} = \text{reg}(j, \mathbf{qm}\text{-step}(\mathbf{q}, \mathbf{qa}, \mathbf{qm}))$$

such that if there has been a positive value on \mathbf{q} since the last reset, then $j = i - 1$.

Then

$$\text{sel}(\text{ctl}, 0, ((\mathbf{r} \times \mathbf{qa}) + \mathbf{q})) = \text{sel}(\text{ctl}, 0, \mathbf{qa}\text{-step}(\mathbf{q}, \mathbf{qa}, \mathbf{qm}))$$

This theorem allows the direct replacement of the initial quotient accumulation sub-circuit :

$$\mathbf{qa} = \text{reg}(\text{?}, \text{sel}(\text{ctl}, 0, ((\mathbf{r} \times \mathbf{qa}) + \mathbf{q})))$$

with the optimized on-the-fly quotient accumulation

$$\begin{aligned} \mathbf{qa} &= \mathbf{reg}(\?, \mathbf{sel}(\mathbf{ctl}, 0, \mathbf{qa}\text{-step}(\mathbf{q}, \mathbf{qa}, \mathbf{qm}))) \\ \mathbf{qm} &= \mathbf{reg}(\?, \mathbf{qm}\text{-step}(\mathbf{q}, \mathbf{qa}, \mathbf{qm})) \end{aligned}$$

Proof: By coinduction. Define integer stream relation \mathcal{R} such that $I \mathcal{R} J$, if there exists

- a stream of control signals, \mathbf{ctl} ;
- a stream of integers q , such that the first nonzero digit after a reset is positive;
- integers i and j , such that if there has been a positive digit on q since the last reset, $j = i - 1$;
- streams qa and qm such that

$$\begin{aligned} qa &= \mathbf{reg}(i, \mathbf{sel}(\mathbf{ctl}, 0, ((r \times qa) + q))) \\ qm &= \mathbf{reg}(j, \mathbf{qm}\text{-step}(q, qa, qm)) \end{aligned}$$

such that

$$\begin{aligned} I &= \mathbf{sel}(\mathbf{ctl}, 0, ((r \times qa) + q)) \text{ and} \\ J &= \mathbf{sel}(\mathbf{ctl}, 0, \mathbf{qa}\text{-step}(q, qa, qm)) \end{aligned}$$

By inspection, we have

$$\mathbf{sel}(\mathbf{ctl}, 0, ((r \times qa) + q)) \mathcal{R} \mathbf{sel}(\mathbf{ctl}, 0, \mathbf{qa}\text{-step}(q, qa, qm))$$

To complete the coinductive proof, it is necessary to show that \mathcal{R} is a bisimulation. Suppose $I \mathcal{R} J$. There are two cases

Heads: $[\text{hd}(I) = \text{hd}(J)]$ The goal is to show that

$$\text{hd}(\text{sel}(ctl, 0, ((r \times qa) + q))) = \text{hd}(\text{sel}(ctl, 0, \text{qa-step}(q, qa, qm)))$$

After expanding the definition of **qa-step**, the only non-trivial case is when $\text{hd}(q) < 0$. The goal is to show $j = i - 1$ ($\text{hd}(qm) = j$ and $\text{hd}(qa) = i$). Since $\text{hd}(q) < 0$, there must have been a positive quotient digit previously, so this goal is satisfied by the constraints on i and j .

Tails: $[\text{tl}(I) \mathcal{R} \text{tl}(J)]$

$$\begin{aligned} \text{tl}(I) &= \text{sel}(\text{tl}(ctl), 0, ((r \times \text{tl}(qa)) + \text{tl}(q))) \text{ and} \\ \text{tl}(J) &= \text{sel}(\text{tl}(ctl), 0, \text{qa-step}(\text{tl}(q), \text{tl}(qa), \text{tl}(qm))) \end{aligned}$$

If we satisfy the appropriate type constraints, then $\text{tl}(I) \mathcal{R} \text{tl}(J)$. The key step is to show that if there has been a positive value on q , or if $\text{hd}(q) > 0$, then $\text{hd}(\text{tl}(qm)) = \text{hd}(\text{tl}(qa)) - 1$. This follows directly from the definition of **qm-step**.
 \dashv

The only difficult aspect about checking the above proof using PVS, is that the stream semantics provide current state and future behavior. There is no access to the past history of a stream. Thus, we cannot directly state the constraint on the stream of quotient digits. The trick is to introduce another stream to remember whether q has had a positive value. Let **R** and **S** have type `Stream[boolean]`. Then,

```
RS(R,S): TYPE =
  { A:Stream[boolean] |
    Invariant(IF R AND (S OR A)
              THEN tl(A)
              ELSE NOT tl(A)
              ENDIF)}
```

defines the type of boolean valued streams that remember whether an event indicated by stream S has occurred since the last reset signaled by stream R . This type is named RS by way of analogy with an RS flip flop. It is reset by an (active-low) signal R , and set by signal S . Once set, it remains asserted until the next reset event. Thus, the type $RS(R, q > 0)$ defines a stream that is asserted if there has been a positive value on stream q since the last reset event signaled by stream R .

We use this type, and PVS' dependent type system to constrain the behavior of stream Q . Let CTL be a stream of control signals, and Q be a stream of integers. Then,

```
(B : RS(map(comp?,CTL),Q > const[int](0)) |
  Invariant(Q < const[int](0) => B))
```

defines a boolean valued stream, B , that remembers the occurrence of a positive value on Q since the beginning of the current execution of the algorithm, with the additional constraint that the first occurrence of a positive value on Q cannot have been preceded by a negative occurrence. The PVS statement of Theorem 13 is:

```
FORALL
  (CTL : Stream[state]),
  (Q   : Stream[integer]),
  (B   : RS(map(comp?,CTL),Q>const[int](0)) |
    Invariant(Q<const[int](0) => B)),
  (i   : integer),
  (j   : integer | hd(B) => j = i - 1):
LET
  ZERO = const[int](0),
  QA   = QA(CTL,Q,i),
  QM   = QM(B,Q,QA,j)
IN
SEL(CTL,ZERO,map2(shift_add,QA,Q)) =
  SEL(CTL,ZERO,map3(qa_step,Q,QA,QM))
```

The PVS proof begins with the strategy (*co-induct-and-simplify*), which guesses an appropriate bisimulation, and simplifies the resulting goals. The proof currently

requires manual intervention to complete, but all the steps could be mechanized. The only intellectual effort required is providing the appropriate constraint on Q and defining the guarded state invariant on the initial values. This information should be present in the designers informal arguments justifying the replacement.

Conclusions

In this dissertation I have explored multiple levels of interaction between design derivation and mechanical theorem proving. The requirements for the design are expressed in the specification language of the theorem proving system. Algorithms that satisfy the requirements are verified within the proof system. The verified algorithms are translated into a behavioral specification for a design derivation system. Within the derivation system the design is refined to an architecture that meets certain design constraints. Refinements outside the scope of the derivation tools are verified externally using a theorem proving system.

The two main examples illustrated how general characteristics of a design are preserved through several layers of design refinement. The core synchronization circuit was designed to be independent of the number of participants in the protocol. The implementation was tested in a four clock system, but could just as easily be used in a ten clock system. In addition, the duration of the synchronization interval was preserved as a parameter throughout the derivation. For the VLSI implementation, the synchronization interval may range from 20 to 20,000 cycles of the underlying

oscillator. For the floating point division hardware, most of the design steps were independent of the quotient digit selection logic. For the partial remainder formation sub-circuits, the designs were refined so that this function was implemented by a lookup table, but the development of the quotient accumulation logic was independent of this function.

The two main examples also demonstrated the utility of the derivational framework for exploring design options. For the clock synchronization circuit, the derivational framework made it possible to isolate the sub-circuit affected by Torres-Pomales' optimization, thus preserving much of the previous design effort. For the division designs, there are two designs each of the partial remainder sub-circuit and the quotient accumulation sub-circuit. Any of the four possible combinations of these sub-circuits provide a verified design.

7.1 Limitations of the Approach

The current framework is limited because the interface between the tools is manual translation. This was unavoidable, because both tools underwent changes during the course of this work. In addition, I have no direct link to the development of either tool. At best, I can make suggestions and hope that they are reflected in the next release.

Many of the verifications required that signals within the design be annotated with invariant properties. These requirements were maintained manually by way of comments. There is nothing in the derivation tools to enforce these design constraints

I used the streams library to define a shallow embedding of DRS stream equations

in PVS. That is, I established an equivalence between objects in DRS and objects defined in PVS. Shallow embedding eliminates potential for verifying some meta-results about DRS stream equations within PVS. In particular, the shallow embedding uses PVS tuples to represent DRS tuples. PVS tuples are distinct from repeated pairing. Since PVS does not allow quantification over types, we cannot express properties about all tuple types.

Another limitation is that the interface between PVS and DRS at the behavioral level is still somewhat sketchy. Recursive functions in PVS must terminate, because PVS requires all functions to be total. Hardware designs are generally non-terminating. While it is certainly possible to develop structures in PVS that faithfully define the semantics of DRS behavioral level descriptions, it would be better if we could maintain a syntactic similarity.

7.2 Suggestions for Future Work

One obvious suggestion is to overcome the limitations enumerated in the previous section. However, many of the suggested improvements involve modifications to the tools themselves. Without input to future modifications to these tools, it is difficult to explore these options.

In Chapter 5, I presented two proofs of Torres-Pomales' optimization. One was localized to just the modified sub-circuit, with appropriate constraints placed on the input signals. The other included some of the surrounding circuitry in the statement of the replacement theorem. The general strategy I have followed is to provide just enough context to justify the replacement. An interesting question is whether a sufficient amount of context can be automatically generated at the same time DRS

generates the correctness condition. Alternatively, the replacement command within DRS could be modified to include a context parameter, to guide generation of the correctness condition.

Many of the replacements were justified using the strategy (*co-induct-and-simplify*). This suggests that it may be possible to add a function to DRS to attempt an automatic verification of a replacement using a trivial bisimulation. This could eliminate the overhead of generating the PVS theories for justifying the replacement. Also, it may be possible to discharge some of the combinational replacement conditions using a tool such as the Stanford Validity Checker [BDL96]. For the replacements that require more interaction, failed proof attempts often suggest appropriate invariants. A coinductive verifier might be improved by adding some of the recent techniques for semi-automatic generation of invariants [GS96].

This dissertation focused on structural level refinements and how theorem proving support can augment derivational design at the structural level. More work needs to be done to link behavioral level descriptions to verified algorithms. This requires development of appropriate representation for reactive systems. Perhaps the theory developed by Jacobs and Hensel will enable more useful models of reactive systems within PVS [HJ97a, HJ97b].

At the end of chapter 4, I suggested one way to assert a fairness constraint on a boolean valued stream. More research is needed to determine how to express and verify other properties on streams. More case studies are needed to determine what properties arise in practice.

7.3 Concluding Remarks

The problem currently facing the application of formal methods in hardware design lies not in creating the first design correctly, but rather in correctly incorporating improvements and optimizations. Often a first design is conservative, with straightforward mappings from algorithms into hardware. Formal methods research has primarily addressed this phase of the design cycle. Subsequent iterations in the design life-cycle involve modifications to improve some aspect of the design. Such modifications may have no obvious relationship to earlier iterations of the design. Frequently, a design modification inadvertently violates some assumptions made in the earlier design, thus we need some mechanism to both verify refinements and remember design constraints.

For formal design approaches to succeed in practice, it is essential that they be able to support the ingenuity of the designer. This requires that formal techniques be designed so that they do not unnecessarily restrict the available design options. The representation of hardware structures using systems of stream equations provides a solid foundation upon which to build a formal framework for hardware design. By viewing sequential circuits as corecursive equations, coinductive proof techniques are effective for verifying arbitrary circuit optimizations. This technique combined with the deductive capabilities of a theorem proving system such as PVS provides a robust method for proving equivalence between two sequential systems.

Bibliography

- [Acz88] Peter Aczel. *Non-Well-Founded Sets*. Number 14 in CSLI Lecture Notes. CSLI Publications, 1988.
- [Ari96] Ariane 5 flight 501 failure: Report by the inquiry board, July 19, 1996. <http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html>.
- [BCL⁺94] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, April 1994.
- [BCM⁺92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.
- [BDL96] Clark Barrett, David Dill, and Jeremy Levitt. Validity checking for combinations of theories with equality. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design, FMCAD '96*, volume 1166 of *Lecture Notes in Computer Science*, pages 187–201, Palo Alto, CA, November 1996. Springer-Verlag.
- [BJ93] Bhaskar Bose and Steven D. Johnson. DDD-FM9001: Derivation of a verified microprocessor. An exercise in integrating verification with formal derivation. In *Proceedings of IFIP Conference on Correct Hardware Design and Verification Methods*. Springer, 1993.
- [BKM96] Bishop Brock, Matt Kaufmann, and J Strother Moore. ACL2 theorems about commercial microprocessors. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design, FMCAD '96*, volume 1166 of *Lecture Notes in Computer Science*, pages 275–293, Palo Alto, CA, November 1996. Springer-Verlag.

- [BM96] Jon Barwise and Lawrence Moss. *Vicious circles: On the Mathematics of Non-Wellfounded Phenomena*. Number 60 in CSLI Lecture Notes. CSLI Publications, 1996.
- [Bos91] Bhaskar Bose. DDD - A Transformation System for Digital Design Derivation. Technical Report 331, Computer Science Dept. Indiana University, May 1991.
- [Bos94] Bhaskar Bose. *DDD-FM9001: Derivation of a verified microprocessor*. PhD thesis, Computer Science Department, Indiana University, USA, 1994.
- [Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.
- [BT90] Alexandre Bronstein and Carolyn L. Talcott. Formal verification of pipelines based on string-functional semantics. In L. J. M. Claesen, editor, *Formal VLSI Correctness Verification: VLSI Design Methods-I*, pages 349–366. North Holland, 1990.
- [BTC96] Bhaskar Bose, M. Esen Tuna, and Venkatesh. Choppella. A tutorial on digital design derivation using DRS. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design, FMCAD '96*, volume 1166 of *Lecture Notes in Computer Science*, pages 270–274, Palo Alto, CA, November 1996. Springer-Verlag.
- [Bur94] Robert G. Burger. The Scheme machine. Technical Report 413, Indiana University, Computer Science Department, August 1994. 59 pages.
- [BY90] William R. Bevier and William D. Young. Machine checked proofs of the design and implementation of a fault-tolerant circuit. Contractor Report 182099, NASA, November 1990. Authors' affiliation: Computational Logic, Inc., Austin, TX.
- [CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2), 1986.
- [CGZ96] E. M. Clarke, S. M. German, and X. Zhao. Verifying the SRT division algorithm using theorem proving techniques. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 111–122. Springer-Verlag, 1996.

- [CJ96] Solange Coupet-Grimal and Line Jakubiec. COQ and hardware verification: A case study. In J. Von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics, 9th International Conference, TPHOLs '96*, volume 1125 of *Lecture Notes in Computer Science*, pages 125–140, 1996.
- [CLS96] David Cyrluk, Patrick Lincoln, and N. Shankar. On Shostak's decision procedure for combinations of theories. In M. A. McRobbie and J. K. Slaney, editors, *Automated Deduction—CADE-13*, volume 1104 of *Lecture Notes in Artificial Intelligence*, pages 463–477, New Brunswick, NJ, July/August 1996. Springer-Verlag.
- [Coq93] Thierry Coquand. Infinite objects in type theory. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs, International Workshop TYPES '93*, volume 806 of *Lecture Notes in Computer Science*, pages 62–78, 1993.
- [EL94] Miloš D. Ercegovic and Tomás Lang. *Division and Square Root: Digit-Recurrence Algorithms and Implementations*. Kluwer Academic Publishers, 1994.
- [Gim95] Eduardo Giménez. An application of co-inductive types in COQ: Verification of the alternating bit protocol. In Stefano Berardi and Mario Coppo, editors, *Types for Proofs and Programs, International Workshop TYPES '95*, volume 1158 of *Lecture Notes in Computer Science*, pages 135–152, 1995.
- [GKP89] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, 1989.
- [Gor86] M. J. C. Gordon. Why higher order logic is a good formalism for specifying and verifying hardware. In G.J. Milne and P.A. Subrahmanyam, editors, *Formal Aspects of VLSI Design: Proceedings of the 1985 Edinburgh Conference on VLSI*, pages 153–177. North Holland, 1986.
- [GS96] Susanne Graf and Hassen Saïdi. Verifying invariants using theorem proving. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 196–207. Springer-Verlag, 1996.
- [Har95] John Harrison. Floating point verification in HOL. In E.T. Schubert, P.J. Windley, and J. Alves-Foss, editors, *Proceedings 8th International*

- Workshop on Higher Order Logic Theorem Proving and its Applications*, volume 971 of *Lecture Notes in Computer Science*, pages 186–199, Aspen Grove, UT, USA, September 1995. Springer Verlag.
- [HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.
- [HD92] F. K. Hanna and N. Daeche. Dependent types and formal synthesis. In C.A.R. Hoare and M.J.C. Gordon, editors, *Mechanized Reasoning in Hardware Design*, pages 121–135. Prentice-Hall, 1992.
- [HDL89] F. K. Hanna, N. Daeche, and M. Longley. VERITAS⁺: A specification language based on type theory. In M. Lesser and G. Brown, editors, *Proceedings of the MSI Workshop on Hardware Specification, Verification and Synthesis: Mathematical Aspects, Cornell University*, volume 408 of *Lecture Notes in Computer Science*, pages 358–379. Springer-Verlag, July 1989.
- [HJ97a] Ulrich Hensel and Bart Jacobs. Coalgebraic theories of sequences in PVS. Technical Report CSI-R9708, Computing Science Institute, Catholic University of Nijmegen, July 1997. An electronic version may be retrieved from <http://www.cs.kun.nl/~bart/sequences.html>.
- [HJ97b] Ulrich Hensel and Bart Jacobs. Proof principles for datatypes with iterated recursion. Technical Report CSI-R9703, Computing Science Institute, Catholic University of Nijmegen, March 1997. To appear in LNCS proceedings of Category Theory and Computer Science 1997.
- [HT96] N. A. Harman and J. V. Tucker. Algebraic models of microprocessors: Architecture and organization. *Acta Informatica*, 33(5):421–456, 1996.
- [Huf54] D.A. Huffman. The synthesis of sequential switching circuits. *Journal of the Franklin Institute*, 257(3):161–190, 1954.
- [Hun92] Warren A. Hunt. A formal HDL and its use in the FM9001 verification. In C.A.R. Hoare and M.J.C. Gordon, editors, *Mechanized Reasoning in Hardware Design*. Prentice-Hall, 1992.
- [IEE85] *Standard for Binary Floating-Point Arithmetic*, 1985. ANSI/IEEE Std 754-1985.
- [IEE87] *Standard for Radix-Independent Floating-Point Arithmetic*, 1987. ANSI/IEEE Std 854-1987.

- [Jac97] Bart Jacobs. Behaviour-refinement of coalgebraic specifications with coinductive correctness proofs. In *TAPSOFT '97: Theory and Practice of Software Development*, Lecture Notes in Computer Science. Springer-Verlag, 1997. To appear.
- [Joh84] Steven D. Johnson. *Synthesis of Digital Design from Recursion Equations*. The MIT Press, Cambridge, 1984. ACM Distinguished Dissertation.
- [JR97] Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 1997. To appear, Available electronically from <http://www.cs.kun.nl/~bart/PAPERS/JR.ps.Z>.
- [JS90] Geraint Jones and Mary Sheeran. Circuit design in Ruby. In Jørgen Staunstrup, editor, *Formal Methods for VLSI Design*, pages 13–70. Elsevier Science Publishers B.V., 1990.
- [JWB89] Steven D. Johnson, R.M. Wehrmeister, and B. Bose. On the interplay of synthesis and verification: Experiments with the FM8501 processor description. In Claesen, editor, *Applied Formal Methods for Correct VLSI Design*, pages 385–404. Elsevier, 1989. IMEC 1989.
- [Keu96] Kurt Keutzer. The need for formal methods for integrated circuit design. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design, FMCAD '96*, volume 1166 of *Lecture Notes in Computer Science*, pages 1–18, Palo Alto, CA, November 1996. Springer-Verlag.
- [LO95] M. Leeser and J. O’Leary. Verification of a subtractive radix-2 square root algorithm and implementation. In *Proceedings International Conference on Computer Design 1995 (ICCD '95)*, pages 526–531, October 1995.
- [LP93] François Leclerc and Christine Paulin-Mohring. Programming with streams in COQ—a case study: The sieve of Eratosthenes. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs, International Workshop TYPES '93*, volume 806 of *Lecture Notes in Computer Science*, pages 191–212, 1993.
- [LS91] Charles E. Leiserson and James B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1):5–35, 1991.
- [MD97] Lawrence S. Moss and Norman Danner. On the foundations of corecursion. *Logic Journal of the IGPL*, 5(2):231–257, 1997.

- [Mea55] G. H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34:1045–1079, September 1955.
- [Mel88] T. Melham. Abstraction mechanisms for hardware verification. In Graham Birtwistle and P. A. Subramanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 267–292. Kluwer Academic Publishers, 1988.
- [Men87] N. P. Mendler. Recursive types and type constraints in second-order lambda calculus. In *Proceedings of the Second Symposium on Logic in Computer Science*, pages 30–36, 1987.
- [Mil89] Robin Milner. *Communication and Concurrency*. Series in Computer Science. Prentice/Hall International, 1989.
- [Min93] Paul S. Miner. Verification of fault-tolerant clock synchronization systems. Technical Paper 3349, NASA, Langley Research Center, Hampton, VA, November 1993.
- [Min95] Paul S. Miner. Defining the IEEE-854 floating-point standard in PVS. Technical Memorandum 110167, NASA, Langley Research Center, Hampton, VA, June 1995.
- [MJ96] Paul S. Miner and Steven D. Johnson. Verification of an optimized fault-tolerant clock synchronization circuit. In Mary Sheeran and Satnam Singh, editors, *Designing Correct Circuits*, Electronic Workshops in Computing. Springer-Verlag, September 1996. <http://www.springer.co.uk/ewic/workshops/DCC96>.
- [ML96] Paul S. Miner and James F. Leathrum, Jr. Verification of IEEE compliant subtractive division algorithms. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design, FMCAD '96*, volume 1166 of *Lecture Notes in Computer Science*, pages 64–78, Palo Alto, CA, November 1996. Springer-Verlag.
- [Moo56] E. F. Moore. Gedanken-experiments on sequential machines. In Claude Shannon and John McCarthy, editors, *Automata Studies*, number 34 in *Annals of mathematics studies*, pages 129–153. Princeton University Press, 1956.
- [MPC86] N. P. Mendler, P. Panangaden, and R. L. Constable. Infinite objects in type theory. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 249–255, 1986.

- [MPJ94] Paul S. Miner, Shyamsundar Pullela, and Steven D. Johnson. Interaction of formal design systems in the development of a fault-tolerant clock synchronization circuit. In *Proceedings 13th Symposium on Reliable Distributed Systems*, pages 128–137, Dana Point, CA, October 1994.
- [MT91] R. Milner and M. Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87:209–220, 1991.
- [OLHA94] J. O’Leary, M. Leeser, J. Hickey, and M. Aagaard. Non-restoring integer square root: A case study in design by principled optimization. In T. Kropf and R. Kumar, editors, *Proc. 2nd International Conference on Theorem Provers in Circuit Design (TPCD94)*, volume 901 of *Lecture Notes in Computer Science*, pages 52–71, Bad Herrenalb, Germany, September 1994. Springer Verlag. published 1995.
- [ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.
- [ORS97] Sam Owre, John Rushby, and N. Shankar. Integration in PVS: Tables, types, and model checking. In Ed Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems TACAS ’97*, volume 1217 of *Lecture Notes in Computer Science*, pages 366–383, Enschede, The Netherlands, April 1997. Springer-Verlag.
- [ORSv95] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.
- [OS93] Sam Owre and Natarajan Shankar. Abstract datatypes in PVS. Technical Report SRI-CSL-93-9R, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993. Extensively revised June 1997.
- [Par80] David Park. On the semantics of fair parallelism. In Bjørner, editor, *Abstract Software Specifications*, volume 86 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.
- [Par81] David Park. Concurrency and automata on infinite sequences. In *Proceedings of the 5th GI Conference*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer-Verlag, 1981.

- [Pau95] Christine Paulin-Mohring. Circuits as streams in COQ: Verification of a sequential multiplier. In Stefano Berardi and Mario Coppo, editors, *Types for Proofs and Programs, International Workshop TYPES '95*, volume 1158 of *Lecture Notes in Computer Science*, pages 216–230, 1995.
- [Pau97] Lawrence C. Paulson. Mechanizing coinduction and corecursion in higher-order logic. *Journal of Logic and Computation*, 7:175–204, 1997.
- [Pit94] A.W. Pitts. A co-induction principle for recursively defined domains. *Theoretical Computer Science*, 124(02):195–220, 1994.
- [Pra95] Vaughn Pratt. Anatomy of the Pentium bug. In P.D. Mosses, M. Nielsen, and M.I. Schwartzbach, editors, *TAPSOFT '95: Theory and Practice of Software Development*, volume 915 of *Lecture Notes in Computer Science*, pages 97–107, Aarhus, Denmark, May 1995. Springer Verlag.
- [PW87] Franklin P. Prosser and David E. Winkel. *The Art of Digital Design*. Prentice-Hall Inc., second edition, 1987.
- [Ras96] Ole Rasmussen. Ensuring correctness of Ruby transformations. In Mary Sheeran and Satnam Singh, editors, *Designing Correct Circuits*, Electronic Workshops in Computing. Springer-Verlag, September 1996. <http://www.springer.co.uk/ewic/workshops/DCC96>.
- [Ros90] Lars Rossen. Ruby algebra. In Geraint Jones and Mary Sheeran, editors, *Designing Correct Circuits*, pages 297–312. Springer-Verlag, 1990.
- [RSS96] H. Rueß, N. Shankar, and M. K. Srivas. Modular verification of SRT division. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 123–134. Springer-Verlag, 1996.
- [SB91] Mandayam Srivas and Mark Bickford. Verification of the FtCayuga fault-tolerant microprocessor system: Volume 1: A case study in theorem prover-based verification. Contractor Report 4381, NASA, July 1991. Authors' affiliation: ORA Corporation, Ithaca, NY.
- [Sch87] Fred B. Schneider. Understanding protocols for Byzantine clock synchronization. Technical Report 87-859, Department of Computer Science, Cornell University, Ithaca, NY, August 1987.
- [SD96] Pierre Sparaco and Michael Dornheim. Ariane probe seeks cause of sudden pitch and yaw. *Aviation Week & Space Technology*, 144(24):18–21, June 10, 1996.

- [SGGH93] James B. Saxe, Stephen J. Garland, John V. Guttag, and James J. Horning. Using transformations and verification in circuit design. *Formal Methods in System Design*, 3(3):181–210, 1993.
- [Sha92] Natarajan Shankar. Mechanical verification of a generalized protocol for Byzantine fault-tolerant clock synchronization. In *Second International Symposium on Formal Techniques in Real Time and Fault Tolerant Systems*, volume 571 of *Lecture Notes in Computer Science*, pages 217–236. Springer-Verlag, January 1992.
- [She88] Mary Sheeran. Retiming and slowdown in Ruby. In George J. Milne, editor, *The Fusion of Hardware Design and Verification*, pages 289–308. North-Holland, 1988.
- [Sho78] Robert E. Shostak. An algorithm for reasoning about equality. *Communications of the ACM*, 21(7):583–585, July 1978.
- [Sho84] Robert E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, January 1984.
- [SM95] Mandayam K. Srivas and Steven P. Miller. Applying formal verification to a commercial microprocessor. In Steven D. Johnson, editor, *CHDL '95: 12th Conference on Computer Hardware Description Languages and their Applications*, pages 493–502, Chiba, Japan, August 1995. Proceedings published in a single volume jointly with ASP-DAC '95, CHDL '95, and VLSI '95, IEEE Catalog no. 95TH8102.
- [SR97] Robin Sharp and Ole Rasmussen. The T-Ruby design system. *Formal Methods in System Design*, 1997. In press, contact osr@id.dtu.dk.
- [ST87] T.K. Srikanth and S. Toueg. Optimal clock synchronization. *Journal of the ACM*, 34(3):626–645, July 1987.
- [Tay81] George S. Taylor. Compatible hardware for division and square root. In *Proceedings 5th Symposium on Computer Arithmetic*, pages 127–134, 1981.
- [TP90] Ghislaine Thuau and Daniel Pilaud. Using the declarative language LUSTRE for circuit verification. In Geraint Jones and Mary Sheeran, editors, *Designing Correct Circuits*, pages 313–331. Springer-Verlag, 1990.
- [TP93] Wilfredo Torres-Pomales. A hardware implementation of a provably correct design of a fault-tolerant clock synchronization circuit. Technical

- Memorandum 109001, NASA, Langley Research Center, Hampton, VA, July 1993.
- [TP95] Wilfredo Torres-Pomales. An optimized implementation of a fault-tolerant clock synchronization circuit. Technical Memorandum 109176, NASA, Langley Research Center, Hampton, VA, February 1995.
- [VCD94] D. Verkest, L. Claesen, and H. De Man. A proof of the nonrestoring division algorithm and its implementation on an ALU. *Formal Methods in System Design*, 4:5–31, January 1994.
- [WA85] William W. Wadge and Edward A. Ashcroft. *LUCID, The Dataflow Programming Language*. Academic Press, 1985.
- [WC91] Robert M. Wehrmeister and Ingo Cyliax. *Logic Engine User Manual*. Indiana University Computer Science Department, Bloomington, Indiana, June 1991. URL <http://www.cs.indiana.edu/classes/c421/man>.
- [Wil97] Matthew M. Wilding. Robust computer system proofs in PVS. In C. Michael Holloway and Kelly J. Hayhurst, editors, *Fourth NASA Langley Formal Methods Workshop*, pages 177–184, September 1997. NASA Conference Publication 3356.
- [WL88] J. Lundelius Welch and N. Lynch. A new fault-tolerant algorithm for clock synchronization. *Information and Computation*, 77(1):1–36, April 1988.
- [WWW] A web directory containing electronic sources of the PVS theories and DRS derivations presented in this dissertation is reachable from <http://www.cs.indiana.edu/~psm/> If you have difficulty finding something, please e-mail psm@cs.indiana.edu.

Curriculum Vitae

Education

- Ph.D. 1998 Indiana University (Computer Science, Minor in Philosophy, Certificate of Advanced Study in Logic)
- M.S. 1992 College of William and Mary (Computer Science)
THESIS: *Verification of Fault-Tolerant Clock Synchronization Systems*
- B.S. 1986 Old Dominion University (Computer Science)

Experience

- 1989– now Research Engineer, NASA Langley Research Center, Hampton, VA
- 1986–1989 Scientific Programmer, Planning Research Corporation—Kentron, Inc., Hampton, VA
- 1986–1986 Summer Intern, NASA Langley Research Center, Hampton, VA

Refereed Conference Publications

7. Steven D. Johnson and Paul S. Miner. Integrated Reasoning Support in System Design: Design Derivation and Theorem Proving. In proceedings IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'97), 1997. To Appear.
6. Paul S. Miner and James F. Leathrum, Jr. Verification of IEEE Compliant Subtractive Division Algorithms. In proceedings of the International Conference on Formal Methods in Computer-Aided Design, pages 64–78, Palo Alto, California, Springer Verlag Lecture Notes in Computer Science 1166, November 1996.
5. Paul S. Miner and Steven D. Johnson. Verification of an Optimized Fault-Tolerant Clock Synchronization Circuit. In Proceedings of the Third Workshop on Designing Correct Circuits, Båstad, Sweden. Springer Verlag, Electronic Workshops in Computing, September 1996.
4. Victor A. Carreño and Paul S. Miner. Specification of the IEEE-854 Floating-Point Standard in HOL and PVS. In supplemental proceedings International Workshop on Higher Order Logic Theorem Proving and its Applications, September 11-14, 1995.

3. Ricky Butler, James Caldwell, Victor Carreño, Michael Holloway, Paul Miner, and Ben Di Vito. NASA Langley's Research and Technology Transfer Program in Formal Methods. in 10th Annual Conference on Computer Assurance (COMPASS 95), Gaithersburg, MD, June 1995.
2. Paul S. Miner, Shyamsundar Pullela, and Steven D. Johnson. Interaction of Formal Design Systems in the Development of a Fault-Tolerant Clock Synchronization Circuit. In Proceedings of the 13th Symposium on Reliable Distributed Systems, pages 128–137, Dana Point, California, October 1994.
1. Steven D. Johnson, Paul S. Miner, and Albert Camilleri. Studies of the Single-pulsar in Various Reasoning Systems. In Second International Conference on Theorem Provers in Circuit Design (TPCD 94), pages 209-227, September 1994.

Other publications and reports

8. Ricky W. Butler, Paul S. Miner, Mandayam K. Srivas, Dave A. Greve and Steven P. Miller. A Bitvectors Library for PVS. NASA Technical Memorandum 110274, August 1996.
7. Paul S. Miner. Formal Specification of IEEE Floating-Point Arithmetic using PVS. Second IFAC Workshop on Safety and Reliability in Emerging Control Technologies. Daytona Beach, Florida, November 1-3, 1995.
6. Paul S. Miner. Defining the IEEE-854 Floating-Point Standard in PVS. NASA Technical Memorandum 110167, June 1995.
5. Paul S. Miner. Verification of Fault-Tolerant Clock Synchronization Systems. NASA Technical Paper 3349, November 1993. (Revised version of Master's Thesis)
4. Paul S. Miner, Peter A. Padilla, and Wilfredo Torres. A provably correct design of a fault-tolerant clock synchronization circuit. In 11th Digital Avionics Systems Conference. October 1992.
3. Paul S. Miner. An extension to Schneider's general paradigm for fault-tolerant synchronization. NASA Technical Memorandum 107634, June 1992
2. Paul S. Miner. A verified design of a fault-tolerant clock synchronization system: Preliminary investigations. NASA Technical Memorandum 107568, March 1992
1. Paul S. Miner. Test and Evaluation of the Generalized Gate Logic System Simulator. NASA Technical Memorandum 4251. 1991