Combining Stream-based and State-based Verification Techniques for Microarchitectures

Nancy A. Day[1], Mark D. Aagaard[2], and Byron Cook[1]

[1] Oregon Graduate Institute, Beaverton, OR, USA
{nday,byron}@cse.ogi.edu
[2] Performance Microprocessor Division, Intel Corporation, Hillsboro, OR, USA
maagaard@ichips.intel.com

Abstract Algebraic verification techniques manipulate the structure of a circuit while preserving its behavior. Algorithmic verification techniques verify properties about the behavior of a circuit. These two techniques have complementary strengths: algebraic techniques are largely independent of the size of the state space, and algorithmic techniques are highly automated. It is desirable to exploit both in the same verification. However, algebraic techniques often use stream-based models of circuits, while algorithmic techniques use state-based models. We prove the consistency of stream- and state-based interpretations of circuit models, and show how stream-based verification results can be used hand-in-hand with state-based verification results. Our approach allows us to combine stream-based algebraic rewriting and state-based reasoning, using SMV and SVC, to verify a pipelined microarchitecture with speculative execution.

1 Introduction

Hardware verification techniques can be broadly grouped into those that reason about both the behavior and structure of circuits, and those that reason just about the behavior. Algebraic techniques, such as retiming (e.g., [29,18]), manipulate the structure of the circuit while preserving its behavior. They have the advantage of being largely independent of the size of the state space. Algebraic techniques often manipulate stream-based models of circuits, i.e., they treat circuits as functions (streams of values). Algorithmic verification techniques, such as model checking [10,30], verify properties about the behavior of a state-based model, i.e., a state transition system, and have the advantage of being highly automated.

In this work, we bridge the gap between these two forms of models by proving that verification results in the stream-based world correspond to correctness criteria of state-based models. We use O'Donnell's method to provide both stream- and state-based interpretations of circuit descriptions [24]. We use the notation $\{\cdot\}$ for the stream-based interpretation, and $[\![\cdot]\!]$ for the state-based interpretation.

The first contribution of this paper is proving that the behavioral equivalence of the stream-based interpretation of two models, x and y, implies that the state-based interpretation of x simulates ($\leq_s$) the state-based interpretation of y:

$$\{x\} = \{y\} \Longrightarrow [\![x]\!] \leq_s [\![y]\!]$$

We refer to this result as the *Verification Correspondence Theorem*. We use Milner's definition of simulation [20]. In order to prove the Verification Correspondence Theorem we prove a general result about the consistency between the stream- and state-based interpretations. This general result, which we call the *Interpretation Consistency Theorem*, can be used to prove the relationships between other kinds of correctness criteria, such as bisimulation [25,21].

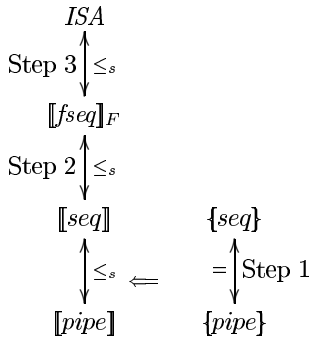


Figure 1. Verification Strategy for the Example

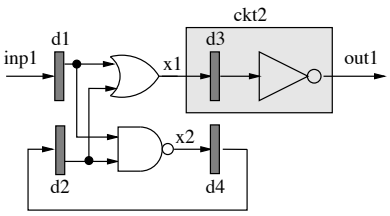
The second contribution of the paper is a demonstration of how the Verification Correspondence Theorem can be used to enable algebraic and algorithmic techniques to work hand-in-hand in microarchitecture verification. An outline of the approach we used for verifying a pipeline against an instruction set architecture (ISA) is illustrated in Figure 1. We decompose the proof that a pipelined microarchitecture with speculative execution (*pipe*) simulates an *ISA* into three parts. First, we rely on the algebraic manipulation techniques of Matthews and Launchbury to simplify the stream-based interpretation of the pipeline to an equivalent sequential (non-pipelined) model (*seq*) by retiming and removing forwarding logic ($\{pipe\} = \{seq\}$) [18]. The sequential model is less complex, making it more amenable to automated verification. The Verification Correspondence Theorem allows us to infer that the state-based interpretation of *pipe* simulates the state-based interpretation of *seq* ($[\![pipe]\!] \leq_s [\![seq]\!]$). Next, we apply the algorithmic techniques of Burch and Dill [7], but first we must overcome the problem that the model we are verifying does not have a flush signal. We construct a version of the model with a flush input (*fseq*), and the second step in the proof uses algorithmic techniques to verify the state-based model without flush ($[\![seq]\!]$) matches the model with flush when flush is false ($[\![fseq]\!]_F$)

i.e., ($[\![fseq]\!]_F \leq_s [\![seq]\!]$). In the third step, we also use algorithmic techniques to check that the state-based interpretation of the sequential model with flush simulates the *ISA* ($[\![fseq]\!]_F \leq_s ISA$). This proof requires invariants. In the third step, we use symbolic model checking [6] implemented in SMV [19] to check the invariants, and simulation-based validity checking implemented in the Stanford Validity Checker (SVC) [7,3]. From these three steps, we can conclude that the pipeline simulates the ISA ($[\![pipe]\!] \leq_s ISA$).

Currently, we use separate tools to accomplish each of the tasks in our verification strategy and chain these steps together in a paper proof. Our Verification Correspondence Theorem is designed to facilitate the integration of algebraic and algorithmic reasoning tasks within a theorem proving environment. Towards that end, we have chosen an approach applicable to shallowly embedded models thereby allowing us to use existing algebraic techniques and theorem proving infrastructure. A shallow embedding in a host language or logic means that the primitives of the model are functions of the host language - there is no separate representation of the abstract syntax of the modeling language in the host language [5]. A deep embedding involves an extra level of indirection in proof. Stream- and state-based interpretations of a model are created by providing two different definitions of the base functions for constructing signals. The approach does not require syntactic analysis of the model – it is not a translation – but instead uses the evaluation mechanism of the host language to calculate the two interpretations. It has the advantage that the two interpretations have the same meaning for all constructs except the base functions. We witnessed the benefit of having a shallow embedding in our proof of the Interpretation Consistency Theorem because we could take direct advantage of an existing framework for proving relationships between multiple interpretations [23]. A further benefit of a shallow embedding is that we avoid building the infrastructure of a translator or special-purpose support for the modeling language. Also, the language can use many convenient features of the host language in constructing a model, such as higher-order functions to generate circuits. Our result relating stream- and state- based interpretations could potentially be applied to models written in languages such as Hawk [11,17], DDD [15], Ruby [16], Lava [4,9], Lustre [14] and stream-based methods for describing hardware in higher-order logic.

2 Model Descriptions

The base functions for constructing signals are: a family of lift operators (`lift1`, `lift2`, ...) and `delay`. Functions over combinational logic are lifted to functions over signals using `lift`, and latches are introduced using `delay`. The initial value of the latch is supplied as a parameter to the delay operator. Each delay has a string label naming the latch. Circuits are described using a set of possibly mutually recursive equations. Figure 2 shows a simple Boolean circuit. A delay is graphically represented using a narrow shaded box. An advantage of this method of description of hardware is that it facilitates modularity because of the ability to hide internal state.




```
ckt1 inp1 = out1 where
  d1 = delay "d1" False inp1
  d2 = delay "d2" False
         (delay "d4" True x2)
  x1 = lift2 (or) d1 d2
  x2 = lift (not) (lift2 (and) d1 d2)
  out1 = ckt2 x1

ckt2 inp = out where
  out = lift (not)
           (delay "d3" False inp)
```

Figure 2. Example circuit

These base functions form the core of Hawk, a microarchitecture description language shallowly embedded in the pure functional programming language Haskell [27]. Pipelines are constructed using these base functions and Haskell language features such as complex datatypes and higher-order functions. Figure 3 shows a graphical representation of a Hawk pipeline. Hawk uses the *transaction* abstraction to describe a structured collection of data [2,1,28]. A transaction contains the opcode, source and destination registers, source and destination values, program counter, and speculative program counter for an instruction. Some units of the pipeline operate on transactions. For example, a register file takes a transaction as input, reads the source values from the register file, and outputs a transaction with these values in the source value fields. A bypass unit ($\diamond$) checks if the source registers for the incoming transaction match the destination register of the instruction forwarded, and if so, sets the source register values to the value of the destination register. The streams and transactions of Hawk have been used to model superscalar out-of-order microprocessors [11].

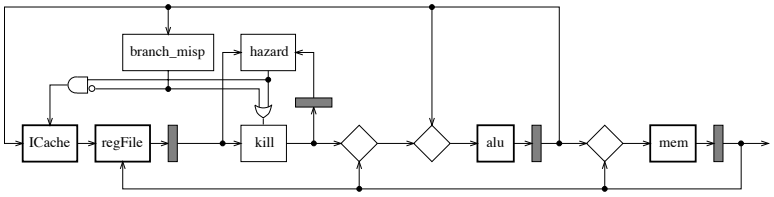


Figure 3. Hawk pipeline [18]

3 Stream-based Interpretation and Verification

There are two common, and isomorphic, stream-based interpretations of signals: infinite lists of values and functions from time to values. Our results apply to

both representations. Definitions 1-6 are the definitions of the base functions for the two stream-based interpretations. The stream-based interpretations ignore the label (*label*), which will be used in the state-based interpretation (Section 4).

Streams as infinite lists:

Definition 1 $\{\texttt{delay}\}\ label\ i\ inp \triangleq i : inp$

Definition 2 $\{\texttt{lift1}\}\ f\ inp \triangleq \texttt{map}\ f\ inp$

Definition 3 $\{\texttt{lift2}\}\ f\ inp_1\ inp_2 \triangleq \texttt{zipWith}\ f\ inp_1\ inp_2$

$where\ \texttt{zipWith}\ z\ (a : as)\ (b : bs) \triangleq (z\ a\ b) : \texttt{zipWith}\ z\ as\ bs$

$\texttt{zipWith}\ \_\ \_\ \_ \triangleq []$

Streams as functions of time:

Definition 4 $\{\texttt{delay}\}\ label\ i\ inp \triangleq \lambda t.\ \texttt{if}\ t = 0\ \texttt{then}\ i\ \texttt{else}\ inp\ (t-1)$

Definition 5 $\{\texttt{lift1}\}\ f\ inp \triangleq \lambda t.\ f\ (inp\ t)$

Definition 6 $\{\texttt{lift2}\}\ f\ inp_1\ inp_2 \triangleq \lambda t.\ f\ (inp_1\ t)\ (inp_2\ t)$

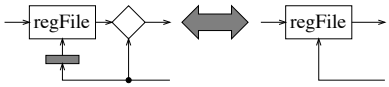


Figure 4. Register Bypass Law [18]

Matthews and Launchbury derived and verified a set of algebraic rules for simplifying Hawk microarchitecture models using an interpretation of signals as infinite lists [18]. These laws allow the components of a model to be rearranged in behavior-preserving ways to result in an equivalent but simpler model. For example, their *register-bypass law* (Figure 4) states that a write-before-read register file followed by a bypass, with a delay on its writeback line, has equivalent behavior to a write-before-read register file by itself. Application of this rule allows for the removal of forwarding logic once a sufficient amount of retiming has been done in a pipeline. They used Isabelle [26] both to verify the laws and to transform the pipeline of Figure 3 into the simpler one of Figure 5. While the validity of these rules was proved using coinductive techniques, for the most part the use of the rules only requires rewriting.

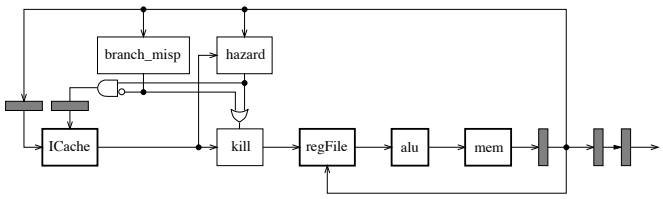


Figure 5. Sequential model resulting from algebraic transformation [18]

4 State-based Interpretation and Verification

We use O'Donnell's method of creating a state-based interpretation of a circuit by providing alternative definitions of `delay` and the family of `lift` operators [24]. O'Donnell had multiple interpretations for simulation and generating netlists from a shallowly embedded gate-level hardware description language. The user provides explicit string labels for the delay elements. The addition of the labels provides a means of identifying state-holding elements and detecting feedback loops within a shallow embedding. We extend his method slightly to handle more complex datatypes, such as transactions, in order to apply the technique to microarchitecture verification.

In the state-based interpretation, evaluating a circuit results in a state-transition system: an initial state, a set of next-state equations (one for each latch), and an observation function. A next-state equation matches a string name with combinational circuitry. The observation function is a function that, given a state, computes the values for the output signals. We use N_M, I_M, and O_M to be the next-state equations, initial state and observation function for state model M. The next-state equations and observation functions are represented using a simple datatype for quantifier-free first-order logic. As in the stream interpretation, evaluating a circuit with a combinational loop will not terminate.

Definitions 7 and 8 show the state-based interpretations of `delay` and `lift2`. The initial value (i) for a delay element is provided as one of the arguments. The user supplies a name for the element in the label argument (*label*).

Definition 7 (State-based interpretation of delay).

$[\![\texttt{delay}]\!]\ \mathit{label}\ i\ \mathit{inp} \triangleq$
$\quad \lambda \mathit{seen}.\ \texttt{if}\ \mathit{label} \in \mathit{seen}$
$\qquad \texttt{then}\ (\emptyset, \emptyset, \mathit{label})$
$\qquad \texttt{else let}\ (N, I, O) = \mathit{inp}\ (\mathit{seen} \cup \{\mathit{label}\})\ \texttt{in}$
$\qquad\quad (\{(\mathit{label}, O)\} \cup N,\ \{(\mathit{label}, i)\} \cup I,\ \mathit{label})$

Definition 8 (State-based interpretation of lift2).

$[\![\texttt{lift2}]\!]\ f\ \mathit{inp}_1\ \mathit{inp}_2 \triangleq$
$\quad \lambda \mathit{seen}.\ \texttt{let}\ (N_1, I_1, O_1) = \mathit{inp}_1\ \mathit{seen}$
$\qquad\quad (N_2, I_2, O_2) = \mathit{inp}_2\ \mathit{seen}\ \texttt{in}$
$\qquad (N_1 \cup N_2,\ \ I_1 \cup I_2,\ \ f\ O_1 O_2)$

The initial state, next-state equations and observation function for the simple circuit of Figure 2 produced by the state-based interpretation is:

$$\begin{aligned} \text{State variables} &= \texttt{d1}, \texttt{d2}, \texttt{d3}, \texttt{d4} \\ I_{ckt} &= \texttt{d3} \leftarrow \texttt{False},\ \texttt{d1} \leftarrow \texttt{False},\ \texttt{d2} \leftarrow \texttt{False},\ \texttt{d4} \leftarrow \texttt{True} \\ N_{ckt} &= \texttt{d3} \leftarrow \texttt{d1} \vee \texttt{d2},\ \texttt{d1} \leftarrow \texttt{inp1},\ \texttt{d2} \leftarrow \texttt{d4},\ \texttt{d4} \leftarrow \neg(\texttt{d1} \wedge \texttt{d2}) \\ O_{ckt} &= \neg\texttt{d3} \end{aligned}$$

The state-based interpretation is calculated by passing as arguments a symbolic representation of the inputs along with an empty list as the list of *seen* labels to

the circuit. The calculation follows the data flow backwards through the fanin of each circuit element. When traversing the example circuit, we encounter `d3`, `d1`, and `d4` each once, and `d2` twice. We see `d2` in the fanin of `d3`, and again after having passed through `d2` and `d4`. At each latch, if the label has not been seen, the method retrieves the set of next-state equations, initial state, and combinational circuitry for the inputs. The next-state equation for a latch is the combinational circuitry of the input signal. The combinational output of a delay is just a term representing the name of the delay. If the label has been seen at a latch, as in the case of seeing `d2` a second time when calculating the input for `d4`, the method does not traverse through `d2` again, because it has already been traversed.

The next-state equations resulting from a state-based interpretation of a circuit are used to create input for existing state-based verification tools. We provide links to two such tools: the SMV model checker [19], and SVC, a tautology checker for a subset of first-order logic [3]. Depending on the logic of the tool, we can generate either interpreted or uninterpreted functions for some operators. To link with SMV, we generate an SMV input file. To link with SVC we symbolically simulate the next-state equations in Haskell, and use a previously implemented tight link between Haskell and SVC to create the internal SVC representations of the terms [12].

4.1 Circuit Structure

Because the labels make the structure of the circuit explicit, two circuits with the same behavior in the stream-based interpretation may result in different state-based interpretations. For example, the circuit of Figure 6 has the same observational behavior as that of Figure 2 but the state-based interpretation has the additional state-holding element `d5`. The equivalence of the stream-based interpretation of Figures 2 and 6, along with the Interpretation Consistency Theorem ensure both state-based interpretations have equivalent observational behavior.

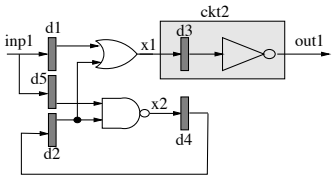


Figure 6. Equivalent to Figure 2

4.2 Valid Labeling of Delays

A danger in our method is that the user can accidently use the same label on multiple delays in the circuit. This can lead to an inconsistent interpretation of the model in the state-based world. This *invalid* labeling can occur in two different ways: one that we can detect in the state-based interpretation of `delay` (Definition 7) and one that requires an external design-rule-check for unique labels on all delays.

In Figure 7, the label c is used for two different delays. We can detect this kind of invalid labeling because there will be two conflicting next-state equations

for the same label. As a side note, we have an optimized version of `delay` that uses the assumption that the circuit has a valid labeling to avoid redundant traversal.

In Figure 8, both occurrences of c appear on the same "branch" of the circuit. When the c closest to a is encountered, the first c is already on the *seen* list, and so the traversal algorithm does not ever encounter a. The state-based interpretation of this circuit matches that one shown in Figure 9.

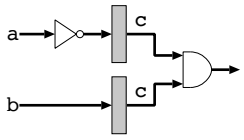


Figure 7. Incorrect labeling 1

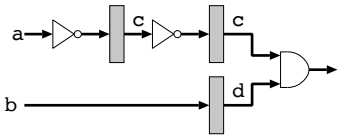


Figure 8. Incorrect labeling 2

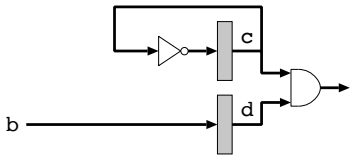


Figure 9. State-based interpretation of Figure 8

4.3 Related Work

Although the aims are the same, our work differs from the signal to state machine translations performed for Lustre by LUSTRE-V2 [14], or for Ruby by T-Ruby [31] and others. Whereas they analyze the program structure and compile a state machine, we want the advantages of working with a shallow embedding and therefore determine the different representations using multiple definitions of base functions.

Singh also uses the technique of alternative definitions for base functions, which he calls *non-standard* interpretations, to provide simulation and test pattern generator interpretations (among others) for Ruby [32]. His work applies only to combinational logic and feedback loops are not allowed.

The original version of the Haskell-based hardware description language Lava was a monadic embedding in Haskell [4]. It used monads to thread the process of generating unique names for state-holding elements through the circuit description in a hidden manner. A difficulty with this approach is that normal function application cannot be used, but rather a new function application operator must be defined.

The recent version of Lava [9] uses Claessen and Sands notion of "observable sharing" [8], which does not require labels for delays, to provide multiple

interpretations of a shallowly embedded language. Their approach is to use a language extension to Haskell that allows them to discover sharing within the heap. The advantage to their approach is that the model does not have to name each state-holding element. The disadvantage is that observable sharing is an impure language feature that precludes the applicability of their approach to pure programming languages and formalisms such as higher-order logic. For generality, we have followed O'Donnell's approach of explicit labeling. In addition to the advantage of generality, we find that explicit labeling has a pragmatic advantage: labels allow the user to provide the names used in the state-based verification. This technique has advantages for debugging and allows the verifier to manage the complexity of many automated verification techniques more easily through means such as controlling variable order. The disadvantage of explicit labeling is that the user has to write these labels and inconsistent labeling is possible. O'Donnell describes methods for generating names semi-automatically, such as using the circuit hierarchy to create structured labels.

5 Formal Connection Between State and Stream Interpretations

In this section, we formalize and verify the connection between state- and stream-based verification. Our main result of the section is the Verification Correspondence Theorem (Theorem 1), which says that if two models, x and y, are equivalent in the stream-based interpretation, then the state-based interpretation of x simulates ($\leq_s$) the state-based interpretation of y.

Theorem 1 (Verification Correspondence).
$\forall x, y.\ \{\!|x|\!\} = \{\!|y|\!\} \Longrightarrow [\![x]\!] \leq_s [\![y]\!]$

For simulation ($\leq_s$), we use Milner's definition [20], restated as Definition 9, where $x \leq_s y$ means that there exists a relation R, such that if R holds for two states q_x and q_y, then R must hold after taking one step of x and y.

Definition 9 (Simulation).
$x \leq_s y \triangleq \exists R.\ \forall q_x, q_y.\ R(q_x, q_y) \Longrightarrow R\ (N_x\ q_x, N_y\ q_y)$

Our proof of the Verification Correspondence Theorem assumes that the two models, x and y, have a valid labeling, as described in Section 4.2. The proof is described in Section 5.2 and relies upon our Interpretation Consistency Theorem. Our Interpretation Consistency Theorem shows that O'Donnell's method produces consistent interpretations in the state- and stream-based worlds; its proof is sketched in Section 5.1.

5.1 Consistency of State and Stream Interpretations

The hardware modeling language we use, Hawk, is implemented as a shallow embedding in Haskell. The fundamental type for describing hardware in Hawk

is a `signal`: wires are `signal`s and gates are functions from `signal`s to `signal`. Our stream-based interpretation of a signal is a function of time. Our state-based interpretation is a state transition system that consists of an initial state, next-state function, and observation function.

We define *consistency* between state- and stream-based interpretations of a model x to mean that: for all k, iterating the state-based interpretation k times produces the same output as sampling the stream-based interpretation at time k (Theorem 2). The function `StsToStream` iterates a state transition system to produce a stream (Definition 10).

Theorem 2 (Interpretation consistency).
$\forall x : \texttt{signal}.\ \texttt{StsToStream}\ [\![\texttt{x}]\!] = \{\texttt{x}\}$

Definition 10.
$\texttt{StsToStream}\ (N, I, O)\ t \triangleq O\ (\texttt{StsToStream}'(N, I)\ t)$
where $\texttt{StsToStream}'\ (N, I)\ 0 \triangleq I$
$\texttt{StsToStream}'\ (N, I)\ t \triangleq N\ (\texttt{StsToStream}'\ (N, I)\ (t-1))$

In Hawk, `signal` is a parameterized type (e.g., `signal` α). To simplify the proof of the Interpretation Consistency Theorem, we treat signals as an unparameterized type (`signal`) over an uninterpreted type T. We also restrict ourselves to `lift1`, which lifts functions of one argument (e.g., an inverter). Extending the proof to a parameterized signal type and multi-argument `lift`s would not pose any technical challenges.

Haskell is based on the second-order polymorphic lambda calculus [33]. Our hardware descriptions are programs in an extension of the lambda calculus that includes two new constants: `delay` and `lift`. The proof of our Interpretation Consistency Theorem uses Mitchell and Meyer's work on *logical relations* [23]. The logical relations framework facilitates proving theorems that relate the meaning of a lambda-calculus program in interpretations that assign different meanings to constants, as we do with `delay` and `lift`.

Mitchell and Meyer's "Fundamental Theorem of Second-Order Logical Relations" (Theorem 3) states that if a relation, ϕ, is a *constant preserving logical relation* between two interpretations, then the meanings of any program in the two interpretations are related by ϕ. The use of recursion in our hardware descriptions gives us the additional obligation to show that the relation is strict and continuous [22].

Theorem 3 (Fundamental Thm of Second-Order Logical Relations). *For all interpretations* $[\![\cdot]\!]$ *and* $\{\cdot\}$ *and for all* ϕ *such that* ϕ *is a logical relation over* $[\![\cdot]\!]$ *and* $\{\cdot\}$*: if* ϕ *preserves the constants of the language then, for all terms* e *of type* τ *in the language:* $\phi_\tau([\![e]\!], \{e\})$

A logical relation is actually a family of relations: one for each possible type in the language. Mitchell and Meyer introduce and verify a logical relation for the standard types and constants in lambda calculus. Mitchell [22] extends this relation to include recursion.

Our Interpretation Consistency Theorem describes a relationship between the state- and stream-based interpretations of expressions of type `signal`, which is captured in Definition 11. We prove the Interpretation Consistency Theorem by 1) extending Mitchell's logical relation with Definition 11 for type `signal` and including `delay` and `lift` as new constants in the lambda calculus; then 2) proving that this new family of relations is a strict and continuous, constant-preserving logical relation.

Definition 11. $\phi_{\texttt{signal}}(e_1, e_2) \triangleq (\texttt{StsToStream}\ e1 = e2)$

The three proof obligations of strictness, continuity, and logicality are solved quite easily because we expressed $\phi_{\texttt{signal}}$ in the lambda calculus. Because $\phi_{\texttt{signal}}$ fully evaluates both of its arguments, $\phi_{\texttt{signal}}$ is strict. Only continuous expressions can be expressed in our formalism, and therefore ϕ is continuous. Logicality simply means that ϕ's treatment of function application and lambda-abstraction is compatible with the lambda calculus, which it clearly is. To prove that ϕ is constant preserving, we prove that, for all types τ and all constants c of type τ, $\phi_\tau([\![c]\!], \{c\})$.

Theorems 4 and 5 state that ϕ preserves the meaning of `lift` and `delay`. Of the two, `delay` has the more interesting proof, so we do not describe the verification of `lift`.

Theorem 4 (Consistency of `lift`).
$\phi_{(T\to T)\to\texttt{signal}\to\texttt{signal}}([\![\texttt{lift}]\!], \{\texttt{lift}\})$

Theorem 5 (Consistency of `delay`).
$\phi_{\texttt{string}\to\texttt{T}\to\texttt{signal}\to\texttt{signal}}([\![\texttt{delay}]\!], \{\texttt{delay}\})$

The correctness of `delay` relies on four properties:

1. The value of a label (i.e., a delay element) in a clock cycle is equivalent to evaluating the combinational logic feeding the delay element in the previous clock cycle.
2. If a latch is not used in the circuit (e.g. it has no fanout), then adding the equation for the latch to the next-state equations of the model does not affect the behavior of the model.
3. When evaluating a model, if a delay is encountered whose label is already in the set of next-state equations for the model, then the combinational logic feeding that delay is equivalent to the next-state equation already computed for the label.
4. Every latch in the circuit has an equation in the set of next-state equations of the model.

The first property is proved by induction over the values produced from a delay. The second property relies on the correctness of adding a next-state equation for a label to the set of next-state equations of a circuit. The third and fourth properties are related to the two forms of invalid labelling described in Section 4.2. Duplicate labels on different branches of a fanin cone (Figure 7) conflicts with the third property and duplicate labels on the same branch of a fanin cone (Figure 8) conflicts with the fourth property.

5.2 Verification Correspondence

To prove that two models, x and y with equivalent stream-based behavior simulate each other (Verification Correspondence Theorem), we assume $\{\!|x|\!\} = \{\!|y|\!\}$ and prove $[\![x]\!] \leq_s [\![y]\!]$.

Applying the Interpretation Consistency Theorem to both sides of the assumption produces:

$$\texttt{StsToStream}\ [\![x]\!] = \texttt{StsToStream}\ [\![y]\!]$$

Using the definition of `StsToStream`, stream equality, and induction, we know that the observations, O_x and O_y, of iterating x and y a total of t times are equal.

$$\forall t.\ O_x(N_x^t\ I_x) = O_y(N_y^t\ I_y)$$

Next we unfold the definition of $\leq_s$ (Definition 9) and provide a witness for the simulation relation. We define two states to be related iff both are reachable in the same number of steps and the observations at those states are equal:

$$R(q_x, q_y) = \exists t.\ (q_x = N_x^t\ I_x) \wedge (q_y = N_y^t\ I_y) \wedge (O_x\ q_x = O_y\ q_y)$$

The proof is completed using skolemization, substitution and reasoning about function composition.

6 Example Verification

To demonstrate the use of the Verification Correspondence Theorem, we prove that the state-based interpretation of the pipeline in Figure 3 (*pipe*) simulates the *ISA*, which is a state-based model, i.e.:

$$[\![pipe]\!] \leq_s ISA$$

The state-based *ISA* is shown in Figure 10. It takes a single input, *stutter*, that we use to make the *ISA* run at the same rate as the pipeline. Figure 11 shows the decomposition of our proof. To achieve this result, we chain together proof steps involving two intermediate models, *seq*, and *fseq*. The Verification Correspondence Theorem allows us to use both stream-based algebraic techniques and state-based algorithmic methods in our proof.

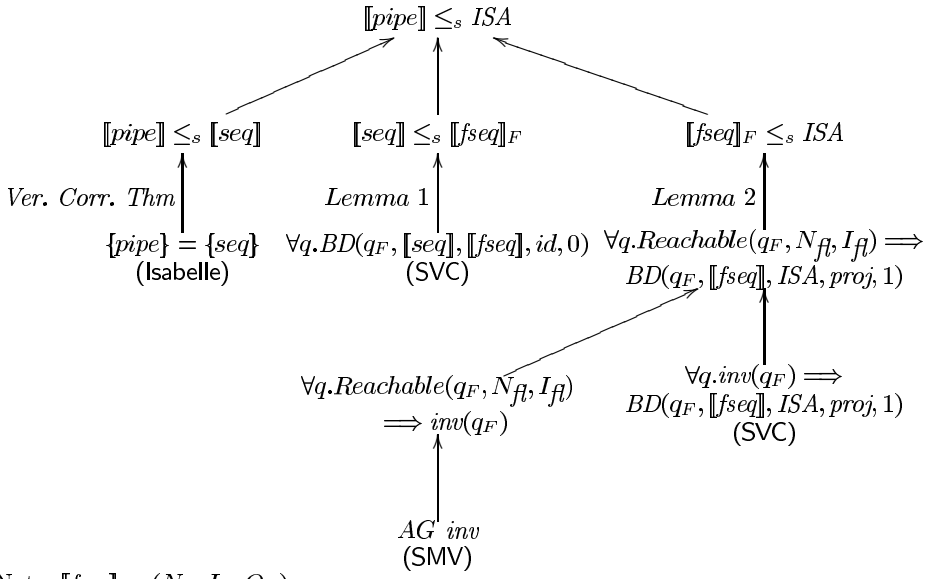
The first step in the proof was previously completed by Matthews and Launchbury using algebraic transformations in the theorem prover Isabelle. They showed that the pipeline of Figure 3 (*pipe*) is observationally equivalent to the sequential model of Figure 5 (*seq*), that is they proved that the stream interpretations of the pipeline and sequential models are equivalent:

$$\{\!|pipe|\!\} = \{\!|seq|\!\}$$

N_{ISA} =

```
let (opc, s1 , s2, dest, imm) = read instrMem pc
    src1 = read regs s1
    src2 = read regs s2 in
mem ← if ((¬stutter) ∧ isStore opc)
        then (write mem src1 src2 )
        else mem
pc ← if stutter
        then pc
        else (if (isIBrz opc)      /* indirect branch if zero */
              then (if (src1 = 0) then src2 else (incpc pc))
              else (if (isBrz opc)      /* offset branch if zero */
                    then (if (src1 = 0) then (incpcby pc imm) else (incpc pc))
                    else (incpc pc)))
regs ← if (¬stutter ∧ (isAlu opc ∨ isLoad opc))
        then (write regs dest (if (isLoad opc)
                                  then (read mem src1 )
                                  else (alu opc src1 src2 )))
        else regs
instrMem ← instrMem
```

Figure 10. ISA



Note: $[\![fseq]\!] = (N_{fl}, I_{fl}, O_{fl})$

q_F means the state q with the *flush* input False

$[\![fseq]\!]_F$ means the model *fseq* with the *flush* input False

BD is the Burch-Dill correctness criteria

Figure 11. Proof tree for the example

From this and our Verification Correspondence Theorem, we can conclude the pipeline simulates the sequential model, i.e., $[\![pipe]\!] \leq_s [\![seq]\!]$.

We use automated verification techniques applicable to state-based methods to complete the remainder of the proof. For this example, we choose to use SMV and SVC, but other methods could be applied.

Burch and Dill showed that for simple pipelines (such as our sequential one) a simulation relation R can be automatically determined by flushing the pipeline (fl) for some number of steps (n), and projecting relevant state-holding elements (p), i.e., $R(q_x, q_y) \triangleq (\exists n.q_y = (p \circ fl^n)(q_x))$. Substituting this definition of R into the definition of simulation, reduces the task of showing model x simulates model y to showing that for some n:

$$\forall q.N_y((p \circ fl^n)\, q) = (p \circ fl^n)(N_x\, q)$$

This is often calling the Burch-Dill commuting diagram. We abbreviate this expression using $BD(q, N_x, N_y, p, n) \triangleq (N_y((p \circ fl^n)\, q) = (p \circ fl^n)(N_x\ q))$. The Burch-Dill result is that:

Lemma 1 (Burch-Dill implies simulation).
$\forall x, y.(\forall q.BD(q, N_x, N_y, p, n)) \Longrightarrow x \leq_s y$

The flush (fl) operation flushes the pipeline using the next state function N and is defined as:

$$fl\ q \triangleq Nq_T$$

where q_T is the state q with the flush input having value True. To apply the Burch-Dill method, we needed to introduce another intermediate model, called *fseq*, which is a version of the sequential model that takes a flush input. Determining the appropriate behavior for completing a flush and resuming normal operation in the midst of potential hazards and branch mispredictions was one of the more difficult parts of this verification effort. To ensure that *fseq* has exactly the same behavior as *seq* when the flush signal is low, we verified that:

$$\forall q.BD(q_F, [\![seq]\!], [\![fseq]\!], id, 0)$$

where id is the identity function, and q_F is the state q with the flush input having value False. We used SVC to carry out this proof. Based on Lemma 1, from this we can conclude that $[\![seq]\!] \leq_s [\![fseq]\!]_F$, where $[\![fseq]\!]_F$ is the state-based interpretation of the *fseq* model with the flush input having value False.

The third branch in the proof shows that $[\![fseq]\!]_F \leq_s ISA$. Again, we use Burch and Dill's result with a projection function (*proj*) that mimics the calculation of the current pc in *fseq* based on the flush, hazard, and mispredict inputs to map the *fseq*'s pc to the *ISA*. We found that to prove the commuting diagram, we needed a number of invariants (*inv*) capturing aspects of the reachable state space. Reachability is defined as $Reachable(q, N, I) \triangleq \exists t.q = N^t I$. We proved,

$$\forall q.Reachable(q_F, N_{fl}, I_{fl}) \Longrightarrow BD(q_F, [\![fseq]\!], [\![ISA]\!], proj, 1)$$

in two steps. First, we used SMV to prove the invariants for limited word size and register file size. Second, we use SVC with the ALU and branch prediction units

uninterpreted, to prove that the commuting diagram holds, under the invariants. These two steps are related to simulation using the following lemma, which is easily proven by incorporating reachability into the simulation relation:

Lemma 2 (Burch-Dill with reachability implies simulation).
$\forall x, y.(\forall q.\mathit{Reachable}(q, N_x, N_y) \Longrightarrow \mathrm{BD}(q, N_x, N_y, p, n)) \Longrightarrow x \leq_s y$

7 Conclusions and Future Work

The first contribution of this paper is the Verification Correspondence Theorem, which relates stream-based algebraic verification results to Milner's state-based simulation correctness criteria. This theorem allows reasoning over both interpretations of models to contribute to a verification result. The value of the Verification Correspondence Theorem derives from the complementary strengths of the two approaches: algebraic techniques can handle large state spaces, and algorithmic techniques are largely automated. It has wide application to stream-based modeling languages such as Hawk, DDD, Ruby, Lava, Lustre, and descriptions of hardware in higher-order logic.

The proof of the Verification Correspondence Theorem required a general result about the correspondence between O'Donnell's stream- and state-based interpretations of models. We plan to investigate how this general result, which we called the Interpretation Consistency Theorem, can be used for non-deterministic models, and stronger correctness criteria such as bisimulation. We also plan to explore how to import state-based results into the stream-based algebraic world, i.e., reversing the implication in the Verification Correspondence Theorem.

Our second contribution is showing the practical application of the Verification Correspondence Theorem in the verification of a pipelined microarchitecture with speculative execution. This example verification pulled together proof steps carried out in the Isabelle theorem prover, the SMV model checker, and SVC. Previously, Matthews and Launchbury's microarchitectural algebra had no connection to standard state-based techniques for processor verification. We are working on additional connections to techniques such as symbolic trajectory evaluation [30].

Our result is a key ingredient towards creating an integrated theorem-proving verification environment where both stream- and state-based verification techniques work hand-in-hand. We have chosen an approach applicable to shallowly embedded models to simplify the application of existing algebraic techniques and theorem proving infrastructure. Currently, the proof steps in our example are chained together on paper using rules such as transitivity of simulation. Mechanization of these steps within a theorem proving environment would help for proof management and security. The first step in creating an integrated environment is to mechanize the proofs of the Interpretation Consistency and Verification Correspondence Theorems. By working within a theorem proving environment, we could also use our result about the consistency between interpretations to link stream-based algebraic results with state-based algebraic results. For example, recent work by Gordon [13] uses algebraic manipulation of

state-based models to reduce the size of the state space before applying algorithmic techniques such as model checking.

Acknowledgments

We thank Robert Jones, John Launchbury, John Matthews, and John O'Leary, as well as members of PacSoft at OGI, for discussions on this topic. We also thank the FMCAD reviewers and Per Bjesse for their helpful suggestions of improvements. John Matthews supplied three figures describing the Matthews and Launchbury work. Support for this work was provided by Intel, NSF (EIA-98005542), USAF Air Materiel Command (F19628-96-C-0161), and the Natural Science and Engineering Research Council of Canada (NSERC).

References

1. M. D. Aagaard and M. E. Leeser. Reasoning about pipelines with structural hazards. In *Theorem Provers in Circuit Design (TPCD)*. pages 13-32, Springer, 1994.
2. S. Bainbridge, A. Camilleri, and R. Fleming. Theorem proving as an industrial tool for system level design. In *Theorem Provers in Circuit Design (TPCD)*, pages 253–274. Elsevier Science Publishers, 1992.
3. C. Barrett, D. Dill, and J. Levitt. Validity checking for combinations of theories with equality. In *FMCAD*, volume 1166 of *LNCS*, pages 187–201. Springer, 1996.
4. P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware design in Haskell. In *ACM Int. Conf. on Functional Programming (ICFP)*, pages 174–184. ACM Press, 1998.
5. R. Boulton, A. Gordon, M. Gordon, J. Harrison, J. Herbert, and J. V. Tassel. Experience with embedding hardware description languages in HOL. In *Theorem Provers in Circuit Design (TPCD)*. pages 129-156, Elsevier, 1992.
6. J. R. Burch, E. Clarke, K. McMillan, D. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*, June 1990.
7. J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *Computer Aided Verification (CAV)*, volume 818 of *LNCS*, pages 68–80. Springer, 1994.
8. K. Claessen and D. Sands. Observable sharing for functional circuit description. In *Asian Computing Science Conference*, 1999.
9. K. Claessen and M. Sheeran. A tutorial on Lava: A hardware description and verification system. April 9, 2000.
10. E. Clarke, E. Emerson, and A. Sistla. Automatic verification of finite-state concurrent systems using temporal logic. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.
11. B. Cook, J. Launchbury, and J. Matthews. Specifying superscalar microprocessors in Hawk. In *Workshop on Formal Techniques for Hardware*, 1998.
12. N. A. Day, J. Launchbury, and J. Lewis. Logical abstractions in Haskell. In *Proceedings of the 1999 Haskell Workshop*. Utrecht University Department of Computer Science, Technical Report UU-CS-1999-28, October 1999.

13. M. Gordon. Reachability programming in Hol98 using BDDs. To appear in *13th International Conference on Theorem Proving and Higher Order Logics (TPHOLs)*, August, 2000.
14. 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.
15. S. D. Johnson, B. Bose, and C. D. Boyer. A tactical framework for digital design. In *VLSI Specification, Verification and Synthesis*, pages 349–384. Kluwer, 1988.
16. G. Jones and M. Sheeran. Circuit design in Ruby. In *Formal Methods for VLSI Design*, pages 13–70. Elsevier Science Publications, 1990.
17. J. Matthews, B. Cook, and J. Launchbury. Microprocessor specification in Hawk. In *International Conference on Computer Languages*, 1998.
18. J. Matthews and J. Launchbury. Elementary microarchitecture algebra. In *Computer Aided Verification (CAV)*, volume 1633 of *LNCS*, pages 288–300. Springer, 1999.
19. K. L. McMillan. *Symbolic Model Checking*. PhD thesis, Carnegie Mellon University, May 1992.
20. R. Milner. An algebraic definition of simulation between programs. In *Proceedings of the 2nd International Joint Conference on Artificial Intelligence*, pages 481–489. The British Computer Society, 1971.
21. R. Milner. *Communication and Concurrency*. Prentice Hall, New York, 1989.
22. J. C. Mitchell. *Foundations for programming languages*. MIT Press, 1996.
23. J. C. Mitchell and A. R. Meyer. Second-order logical relations (extended abstract). In *Logic of Programs*, volume 193 of *LNCS*, pages 225–236. Springer, 1985.
24. J. O'Donnell. Generating netlists from executable functional circuit specifications in a pure functional language. In *Functional Programming Glasgow*, Workshops in Computing, pages 178–194. Springer, 1992.
25. D. Park. Concurrency and automata on infinite sequences. In *5th GI Conference on Theorectical Computer Science*, volume 104 of *LNCS*. Springer, 1981.
26. L. C. Paulson. Introduction to Isabelle. Technical Report 280, University of Cambridge, Computer Lab, 1993. Latest edition: 24 November 1997.
27. J. Peterson and K. Hammond, editors. *Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.4)*. Yale University, Department of Computer Science, RR-1106, February 1997.
28. J. Sawada and W. Hunt. Trace table based approach for pipelined microprocessor verification. In *Computer Aided Verification (CAV)*, volume 1254 of *LNCS*, pages 364–375. Springer, 1997.
29. J. B. Saxe, S. J. Garland, J. V. Guttag, and J. J. Horning. Using transformations and verification in circuit design. In *Designing Correct Circuits*, 1992.
30. C.-J. H. Seger and R. E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design*, 6:147–189, March 1995.
31. R. Sharp. T-Ruby: A tool for handing Ruby expressions. August, 1996.
32. S. Singh. Implementation of a nonstandard interpretation system. In *Functional Programming, Glasgow*, Workshops in Computing, pages 206–224. Springer, 1989.
33. P. Wadler. Theorems for free! In *Fourth International Conference on Functional Programming Languages and Computer Architecture (FPCA89)*, pages 347–359, 1989.