Microprocessor Verification in PVS

A Methodology and Simple Example

David Cyrluk

February 10, 1994

Abstract

In the past we have verified a variety of different hardware circuits using different verification systems. In this paper we report on one such verification. The verification is that of a simple pipelined microprocessor first verified in [8].

We use this example as a vehicle for exploring a general methodology of verifying state machine systems in the PVS verification system. It is our hope that such a methodology can be extended to verify the correctness of actual commercial microprocessors.

Contents

List of Figures

Chapter 1

Introduction

Unlike traditional programs, many computer systems cannot be specified using simple input/output relations. Such systems include reactive software systems such as operating systems, concurrent processes, many hardware circuits such as microprocessors, and many protocols such as cache coherence.

The current trend is to specify such systems as state transition systems. This is the approach taken in both Unity [3] and TLA [6]. This approach has also been used in protocol verification [5], and microprocessor verification [9]. Abadi and Lamport [1] discuss this approach and under what conditions it can be used.

Over the past year we have used a general methodology based on state transition systems to verify the correctness of several hardware circuits. In this paper we show how one can follow this methodology to verify state machine systems in the PVS verification system [7]. We illustrate this methodology on two examples—a simple example, part of a trip meter, and a more complex example, a pipelined microprocessor.

The methodology consists of specifying both the specification and implementation of the system using state machines. The relationship between the two machines is specified using an abstraction function[1] that relates implementation states to specification states. To show that the implementation machine implements the specification machine amounts to showing that the traces generated by the two machines are the same relative to this abstraction function.

There are several possible variations of this methodology. The relative timing of the two machines might be different. The machines might be nondeterministic. We might want to hierarchically decompose the proof of correctness. Also it may be desirable to vary the methodology to take into account the specific type of system we are verifying. For example, in the case of hardware verification, we might want to specialize the methodology to deal with the combinational part of the circuit separately, or to deal with control and data paths differently.

In this paper we first present the general methodology in its simplest form, illustrating it with a very simple example. We then explore one variation of the simple methodology in detail: how to handle timing differences between the specification and implementation machines. We present two alternatives for dealing with this problem. The first alternative

[1] Abadi and Lamport [1] call this a *refinement mapping*.

is to “slow down” the specification machine. The second alternative is to “speed up” the implementation machine. We illustrate these two alternatives by applying them to the verification of a pipelined microprocessor [8]. It turns out that the choice of alternative does make a difference as to naturalness of specification and ease of proof.

At the end of the paper we discuss other possible variations of this methodology, paying particular attention to their appropriateness for hardware verification. A familiarity with PVS might be useful, but is not necessary to understand this report.

Chapter 2

The General Methodology

We now provide a general methodology for specifying state machine systems in PVS and verifying that one state machine correctly implements another. The methodology consists of three steps:

1. Provide the *specification* state machine.
2. Provide the *implementation* state machine.
3. Prove that the implementation machine satisfies the specification machine.

2.1 The State Machines

The state machines are specified by providing the following information:

- The type of the *machine state*
- The type of the *input*
- The type of the *output*
- The *next-state* function
- The *output* function

The state type is typically a record consisting of all the state variables. In a hardware circuit the state variables would be the registers, memory, and other clocked components of the circuit.

The input to the state machine is modeled as an input stream: an infinite sequence of inputs. Multiple inputs can be represented as a single input stream of a tuple of the different inputs.

The machine's behavior is specified by providing a next-state function. This function takes the current input and current state, and returns the next state. The specification of this function may be quite complicated. It typically requires the specification of other constants, functions, and theories. It is usually defined in terms of auxiliary functions that return the values of the state variables in the next state, given the current input and state.

Finally, the output function must be provided. This function takes a machine state and returns the corresponding output for that state. This corresponds to a *Moore* machine. In the case of a *Mealy* machine, the output will also be a function of the input. (For definitions and properties of these machines the reader is referred to [4].)

A boilerplate PVS specification of a state machine is presented in Figure 2.1.

```
state_machine : THEORY

BEGIN

  machine_state: TYPE = <state type>

  input: TYPE = <input type>

  output: TYPE = <output type>

  next_state: [machine_state, input -> machine_state]
  % Specification of behavior of the next state function

  output:  [machine_state -> output]
  % Specification of behavior of the output function

END state_machine
```

Figure 2.1: A state-machine template

The user will provide two state machines, a specification state machine and an implementation state machine. Generally the specification machine is designed to capture the intended behavior of the system the user is trying to verify. The implementation machine is designed to provide an *efficient* implementation that respects that behavior.

Thus the specification machine will typically be simpler, more abstract and less efficient than the implementation machine. Ideally it should be an obvious encoding of an informal requirement. The implementation machine will typically not be obviously correct, but more efficient, complicated, and concrete than the specification machine. The verification task is to show that the implementation machine does not introduce any behaviors not allowed by the specification machine.

2.2 The Correctness Condition

Once the specification and implementation state machines have been provided, the user must prove that the implementation state machine *satisfies* the specification state machine.

Before defining this correctness condition we provide some background. We model the input of a state machine as an infinite sequence of objects of the input type. Similarly, the output is an infinite sequence of objects of the output type, and the execution history is an infinite sequence of states of the machine state type.

In PVS, infinite sequences are functions from natural numbers to some type. A predefined sequence type built into PVS is shown in Figure 2.2.

```
sequences[t: TYPE] : THEORY

 BEGIN

  sequence : TYPE = [nat->t]

  nth(seq: sequence, n: nat): t = seq(n)

  suffix(seq:sequence, n:nat): sequence =
    (LAMBDA (i:nat): seq(i+n))

  first(seq: sequence): t = nth(seq, 0)

  rest(seq: sequence): sequence = suffix(seq, 1)

                                          ⋮

  every(p: PRED[t], seq: sequence): bool =
    (FORALL (n:nat): p(nth(seq, n)))

  seq_induction : LEMMA
    (FORALL (p: pred[t]), (seq: sequence) :
         p(nth(seq,0)) AND
         (FORALL (n : nat) : p(nth(seq, n)) IMPLIES p(nth(seq, n + 1)))
      IMPLIES every(p, seq))

 END sequences
```

Figure 2.2: Theory `sequences`

Now we define the execution trace of a state machine as the sequence of machine states that results from iterating the next-state function for that machine.

The execution trace is constructed by providing a function that for any natural number, n, returns the state that the machine will be in after iterating the next-state function n times starting at some initial state, and given an input sequence.

The function `trace`, when given a next-state function, an initial state, and an input sequence, generates the execution trace for the machine defined by the next-state function. The PVS theory that defines these functions is presented in Figure 2.3.

Now, given two state machines we would like to say that one implements the other just in case, for any input sequence and any initial state, the output sequences of the two machines are identical. However, this is not quite enough. The problem is that we have not specified a way of relating the implementation state to the specification state. For example, in the above definition, what does it mean for the implementation machine and specification machine to start from the same initial state?

```
traces [state: TYPE, input: TYPE]: THEORY

BEGIN

  i, n: VAR nat

  next_type: TYPE = function[state, input -> state]

  next: VAR next_type

  s: VAR state

  input_sequence: TYPE = sequence[input]

  l: VAR input_sequence

  next_to_the_n (next, s, l, n): RECURSIVE state =
    IF n = 0 THEN s
       ELSE next_to_the_n(next, next(s, first(l)), rest(l), n-1)
    ENDIF
  MEASURE (LAMBDA next, s, l, n: n)

  trace(next, s, l): sequence[state] =
    (LAMBDA i: next_to_the_n(next, s, l, i))

END traces
```

Figure 2.3: Theory `traces`

Thus, the user must provide an *abstraction* function that when given an implementation state returns a specification state. We denote this function as `Abst`. This function must be surjective. This is to ensure that if a specification trace starts at some initial state, an implementation trace is able to start at a corresponding state.

Given the `Abst` function, we can now state formally the condition that for any input sequence and any initial state, the output sequences of the two machines are identical.

Let s be an implementation state, *ins* an input sequence, `I` the implementation next-state function, `A` the specification next-state function, `Ioutput` the implementation output function, `Aoutput` the specification output function, and `map` a function that given a function and a sequence returns a new sequence by applying the given function to every element in the given sequence. Then equation (2.1) is a formal statement that the output sequences of the two machines correspond.

$$\forall s, ins : \mathtt{map}(\mathtt{Ioutput}, \mathtt{trace}(\mathtt{I}, s, ins)) = \mathtt{map}(\mathtt{Aoutput}, \mathtt{trace}(\mathtt{A}, \mathtt{Abst}(s), ins)) \qquad (2.1)$$

This states that the implementation trace starting with state s yields the same output as the specification state starting with state $\mathtt{Abst}(s)$. Note that while the specification machine starts at the *abstraction* of the state that the implementation machine starts at, both machines get their input from the same input stream.

In many examples, such as those involving microprocessors, we are not concerned with the output behavior of the state machines, but only that the execution traces of the two machines correspond. That is, that taking the abstraction of the implementation trace starting in state s is identical to the specification trace starting in state $\texttt{Abst}(s)$. This is stated formally in equation 2.2.

$$\forall s, ins : \texttt{map}(\texttt{Abst}, \texttt{trace}(\texttt{I}, s, ins)) = \texttt{trace}(\texttt{A}, \texttt{Abst}(s), ins) \tag{2.2}$$

A graphical view of formula 2.2 is shown in Figure 2.4(a).

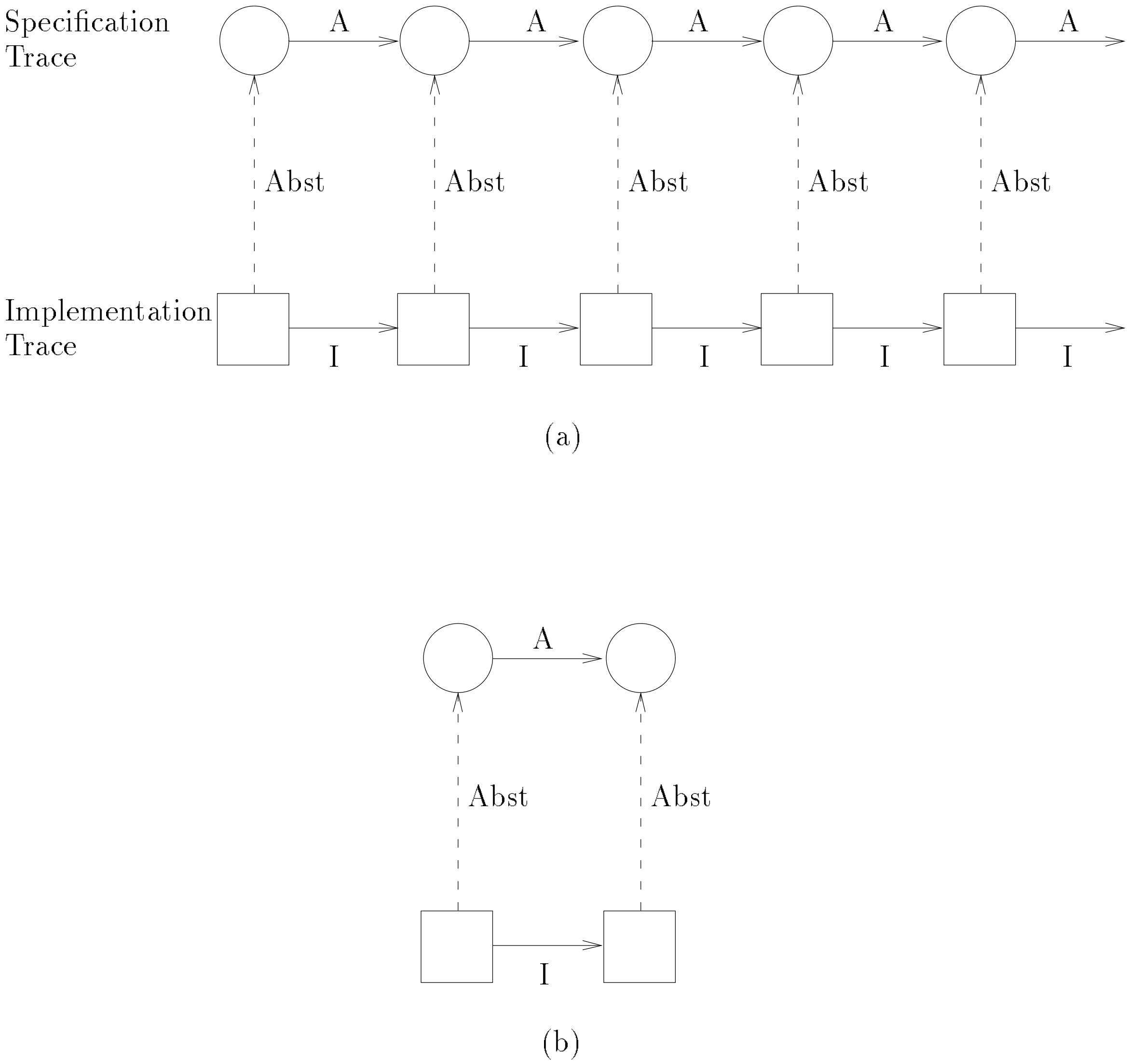

Figure 2.4: Commutes

Formulas 2.1 and 2.2 are higher-order formulas. However, in order to prove them it is sufficient to prove the following two first-order formulas:

Commutes: $\forall s, input : \texttt{Abst}(\texttt{I}(s, input)) = \texttt{A}(\texttt{Abst}(s), input)$

Output: $\forall s : \texttt{Ioutput}(s) = \texttt{Aoutput}(\texttt{Abst}(s))$

where *input* is an arbitrary element in the input stream. The **Commutes** formula corresponds to the commutativity of the diagram in Figure 2.4(b).

Note that the `Commutes` formula is sufficient to prove equation 2.2. Note also that equation 2.1 follows trivially from equation 2.2 and the `Output condition`.

We have captured the notion of correctness stated in equations 2.1 and 2.2, as well as the sufficient first-order conditions that imply these equations in a PVS theory. The theory is parameterized by the specification state machine, the implementation state machine, and the abstraction function, which is required to be surjective. It contains as the two theorems the correctness conditions stated above. Its `assuming` section consists of the two first-order formulas sufficient to prove the correctness condition. This theory is presented in Figure 2.5.

Whenever this theory is instantiated, it will generate as *typecheck correctness conditions* (*TCCs*) three verification conditions:

1. The surjectiveness of `Abst`
2. The **Commutes** lemma
3. The **Output** lemma

There is no need for the user to prove the general higher-order verification conditions, as these are proved once and for all in the generic `trace_equiv` theory.

2.3 Adding Time Explicitly

In the above presentation there was no explicit mention of time. Time was implicit in there being a sequence of states, but no explicit variable representing time was needed.

However, there is a style of specification in which an explicit time variable can be used. Saxe, et al [8] use the explicit time style, for example. We will need to make use of the explicit time style when we slow down the specification machine. In the explicit time style execution traces are explicitly represented as functions from time to machine states. The next-state function (or relation) is specified using axioms that relate machine states at successive times.

The high level statement of correctness can then be stated differently than in the previous section, but ultimately, the proof of correctness still comes down to showing the commutativity of diagram 2.4(b). For further details see [8] or the discussion later in this paper.

```
trace_equiv [Astate, Istate, input, output: TYPE,
             A: [Astate, input -> Astate],
             I: [Istate, input -> Istate],
             Abst: (surjective?[Istate, Astate]),
             Aoutput: [Astate -> output],
             Ioutput: [Istate -> output]]: THEORY

BEGIN

ASSUMING

  Commutes: ASSUMPTION
     FORALL (s: Istate), (inp: input): Abst(I(s, inp)) = A(Abst(s), inp)

  Output: ASSUMPTION
     FORALL (s: Istate): Ioutput(s) = Aoutput(Abst(s))

ENDASSUMING

USING traces[Astate, input], traces[Istate, input]

  r : VAR Astate
  s : VAR Istate
  inp : VAR input
  is : VAR sequence[input]

  Atrace(r, is): sequence[Astate] =
    traces[Astate, input].trace(A, r, is)

  Itrace(s, is): sequence[Istate] =
    traces[Istate, input].trace(I, s, is)

  trace_equiv: THEOREM map(Abst, Itrace(s, is)) = Atrace(Abst(s), is)

  output_equiv: THEOREM
    map(Ioutput, Itrace(s, is)) = map(Aoutput, Atrace(Abst(s), is))

END trace_equiv
```

Figure 2.5: Theory trace_equiv

Chapter 3

The Trip Meter

We now illustrate the above methodology on a simple example. We begin with an informal description of the trip meter, its specification state machine, and its implementation state machine. We then proceed to specify these machines in PVS and verify the correctness of the implementation machine.

3.1 Informal Specification

The input to the trip meter is a stream of numbers. The output of the trip meter is a stream of boolean values. At any time, the output will be `true` if and only if the meter is tripped. Very loosely, the meter is tripped if and only if the current value of the input is either above a given *high trip* threshold, or below a given *low trip* threshold. In addition, the meter has some hysteresis. If in the previous state the meter was tripped, then the meter will remain tripped even if the input value moves below the high trip threshold or above the low trip threshold by only a small amount.

Let `HI` be the high trip threshold, `LO` the low trip threshold, `D` the amount of hysteresis in the meter, and *input* the current input value. Then the conditions that determine when the meter is tripped can be given more precisely as follows:

1. If $input > \mathtt{HI}$, then the meter is tripped.
2. If $input < \mathtt{LO}$, then the meter is tripped.
3. If $input > \mathtt{HI} - \mathtt{D}$ and in the previous state the meter was tripped due to condition 1 or 3, then the meter remains tripped.
4. If $input < \mathtt{LO} + \mathtt{D}$ and in the previous state the meter was tripped due to condition 2 or 4, then the meter remains tripped.
5. Otherwise, the meter is not tripped.

In addition, `HI`, `LO`, and `D` must satisfy the following constraints:

1. The hysteresis value is nonnegative, that is $\mathtt{D} \geq 0$.

2. The high trip threshold is greater than the low trip threshold, that is $\mathtt{HI} > \mathtt{LO}$.

3. The areas of hysteresis do not overlap, that is $\mathtt{HI} - \mathtt{D} \geq \mathtt{LO} + \mathtt{D}$.

We now turn the above requirements into a PVS specification of a state machine following the general form presented above.

3.2 The Specification State Machine

Our goal in writing the PVS *code* for the specification state machine is to translate the above requirements in as transparent a manner as possible. We are not concerned here with efficiency or even implementability, but solely with accuracy and clarity.

We capture the state of the machine with two state variables, `HT` and `LT`. If the meter is tripped because of the input being high, then `HT` is `true`. If the meter is tripped because of the input being low, then `LT` is `true`. If the meter is untripped, then both `HT` and `LT` are `false`. It is never the case that both `HT` and `LT` are `true`. In PVS this state is represented by a record containing the two fields, `HT` and `LT`.

`A` is the next-state function. It updates the state according to the informal description of the trip meter given above. The output of the meter is given by the boolean-valued function `AOutput`, which is `true` in a given state if and only if the machine is either *low tripped* or *high tripped*. The complete specification can be found in Figure 3.1.

In `trips_spec` the definition of `Astate` states that the machine state is a record consisting of state variables, `HT` and `LT`, such that it is never the case that both `HT` and `LT` are `true`. This type declaration will cause *TCCs* to be generated ensuring that the invariant will be maintained by any function that returns an `Astate`. The rest of the theory should be self-explanatory.

3.3 The Implementation State Machine

Our goal in generating an implementation state machine is to come up with something that can be implemented and hopefully implemented efficiently. The trip meter example is such a simple example that the specification above is a reasonable implementation. To illustrate our general methodology we provide a separate implementation that could actually also be used as a specification. In fact in this case it is possible to reverse the roles of the specification and implementation machines and still complete the proof of correctness. This is not the case generally though. Usually the implementation machine is more concrete and complex and allows *fewer* behaviors (traces) than the specification machine, and thus their roles cannot be reversed.

The implementation of the trip meter differs from the specification only in the representation of the machine state. In the implementation, instead of representing the state as a pair of boolean state variables we represent the state as a single state variable that can take on one of three values—`Ht`, `Lt`, or `Ut`—depending on whether the meter is high tripped, low tripped, or untripped, respectively. The complete implementation can be found in Figure 3.2.

```
trips_spec : THEORY

BEGIN

  Any_Astate : TYPE = [# HT, LT : bool #]

  Astate : TYPE = { r : Any_Astate | NOT (HT(r) AND LT(r)) }
                  % The state type.

  input : TYPE = number

  output : TYPE = bool

  r : VAR Astate
  inp : VAR input

  LO : number % The low trip threshold.
  HI : number % The high trip threshold.
  D : {x : number | x >= 0} % The amount of hysteresis.

  LO_HI : AXIOM LO < HI % The high trip threshold should be greater
                        % than the low trip threshold.

  D_LO_HI : AXIOM 2*D <= HI - LO % The low trip area and high trip
                                 % area do not overlap.

  lt(r, inp) : bool = % Function for updating the LT state variable.
   (inp < LO OR (inp < LO + D AND LT(r)))

  ht(r, inp) : bool = % Function for updating the LT state variable.
   (inp > HI OR (inp > HI - D AND HT(r)))

  A(r, inp): Astate = % The next state function.
   {# LT := lt(r, inp),  HT := ht(r, inp) #}

  AOutput(r): output = (HT(r) OR LT(r)) % The aoutput function.

END trips_spec
```

Figure 3.1: Theory `trips_spec`

Note that this theory includes `trips_spec` in its `USING`. This is so that it has access to the input and output types defined in `trips_spec` as well as to the constants, `HI`, `LO`, and `D`. Also note that the type of `Istate` is an enumeration type instead of a record. The rest of the theory should be self-explanatory.

```
trips_impl: THEORY

BEGIN

USING trips_spec

  Istate : TYPE = { Ht, Lt, Ut }

  s : VAR Istate
  inp : VAR input

  I(s, inp) : Istate =
   (IF (inp > HI OR (s = Ht AND inp > HI - D))
        THEN Ht
      ELSIF (inp < LO OR (s = Lt AND inp < LO + D))
        THEN Lt
      ELSE Ut ENDIF)

  IOutput(s): output = (s /= Ut)

END trips_impl
```

Figure 3.2: Theory `trips_imp`

3.4 Proving Correctness

We now wish to prove that `trips_impl` correctly implements `trips_spec`. To do this we define the abstraction function, `Abst`, and then generate the three verification conditions described in Section 2.2, and then prove those verification conditions.

3.4.1 The Verification Conditions

The verification conditions are generated automatically by instantiating the PVS theory `trace_equiv`. The PVS theory `trips_verif`, found in Figure 3.3 does all this.

The verification conditions generated are:

```
verif_TCC1:
   OBLIGATION surjective?(Abst)

verif_TCC2:
   OBLIGATION
  FORALL (s: Istate), (inp: number): Abst(I(s, inp)) = A(Abst(s), inp)

verif_TCC3:
   OBLIGATION FORALL (s: Istate): IOutput(s) = AOutput(Abst(s))
```

In addition to the above verification conditions PVS generated other proof obligations. Most of these were simple lemmas needed to verify type correctness. One, however, is worth noting:

```
trips_verif: THEORY

BEGIN

USING trips_spec, trips_impl, trace_equiv

  Abst((s:Istate)):  Astate =
   (IF Lt = s THEN {# LT := TRUE, HT := FALSE #}
       ELSIF Ht = s THEN  {# HT := TRUE, LT := FALSE #}
       ELSE  {# HT := FALSE, LT := FALSE #}
    ENDIF)

  verif: THEORY = trace_equiv[number, bool, Astate, A, AOutput,
                              Istate, I, IOutput, Abst]

END trips_verif
```

Figure 3.3: Theory `trips_verif`

```
% Subtype TCC generated for
   % {# % The next state function.
   %    LT := lt(r, inp), HT := ht(r, inp) #}
A_TCC1: OBLIGATION (FORALL inp, r: NOT (ht(r, inp) AND lt(r, inp)))
```

This *TCC* states that the result of `A`, the specification next-state function, satisfies the type invariant for `Astate`. In many other verification systems it is the user's obligation to generate the appropriate lemmas to ensure that the state invariants hold; PVS generates these automatically as a product of typechecking. If needed, this invariant can be used in the other proof obligations.

3.4.2 Proving the Verification Conditions

All the *TCC*s generated were easily proven. Here we briefly describe the proofs of the three verification conditions described in section 2.2.

3.4.2.1 Surjectiveness of Abst

This theorem is proven by performing case analysis on the different possible specification states. For each possible specification state, r, an appropriate implementation state, s, is provided such that $\texttt{Abst}(s) = r$.

3.4.2.2 Commutes

This theorem is proven by rewriting the definitions of `Abst, A, I, lt,` and `ht.` The axioms constraining `HI,` `LO`, and `D` are provided. This is then asserted to the ground decision procedures, which proves the theorem automatically.

3.4.2.3 Output

This theorem is proven by rewriting `Abst`, `AOutput`, and `IOutput`. The result is asserted to the ground decision procedures, which completes the proof.

Chapter 4

Timing Differences

We have just presented a methodology for verifying the correctness of state machine implementations and illustrated it on a very simple example. Unfortunately, many state machine verification problems are not as simple. In this section we extend our methodology to deal with one possible complication: the specification state machine and implementation state machine might run at different *rates*. For instance, in the trip meter example, a possible implementation could introduce an intermediate state that would record whether the input is in the strictly high tripped area, hysteresis high tripped area, strictly untripped area, hysteresis low tripped area, or strictly low tripped area. Then from this state the machine would go to the appropriate next-state, just as in the machine specified earlier.

Now, it is no longer possible to use the methodology just presented to verify the correctness of this new implementation. The reason is that the new implementation now runs half as fast as the specification. That is, there is no specification state that corresponds to the intermediate implementation state. Figure 4.1 illustrates this situation.

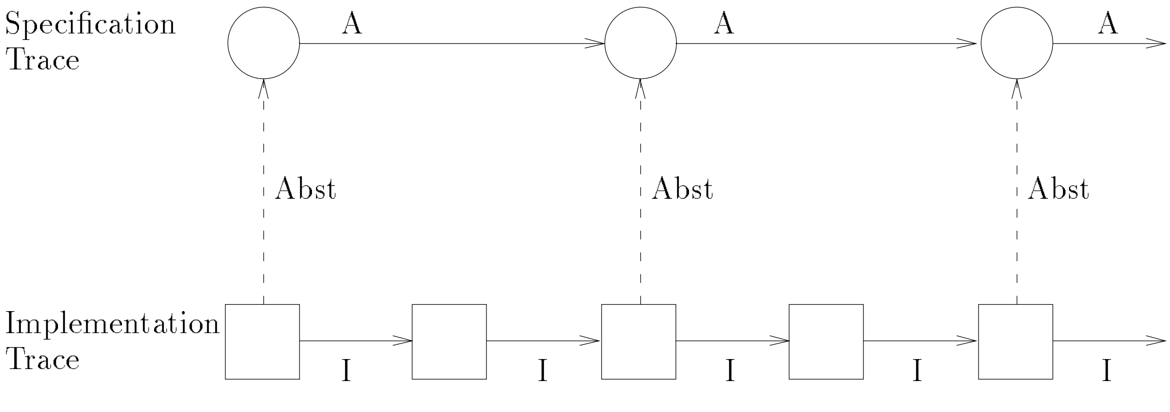

Figure 4.1: Timing Differences

There are two possible ways to extend the original methodology to deal with this.

- Speed the implementation machine up. This is illustrated in Figure 4.2.
- Slow the specification machine down. This is illustrated in Figure 4.3.

4.1 Speeding Up the Implementation Machine

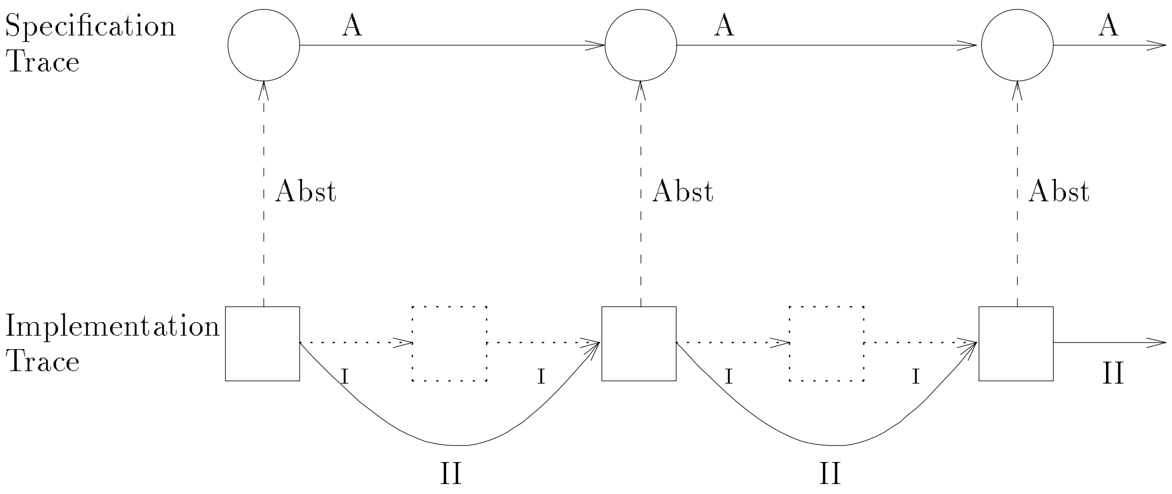

Figure 4.2: Speeding up the Implementation Machine

We call the approach in which we "speed up" the implementation machine, the *visible-state* approach. Before describing the *visible-state* approach we provide some terminology. In Figure 4.1 we can distinguish between two types of implementation states. We call those states that have a corresponding specification state *visible* states. Those states that do not have a corresponding specification state (or, in Figure 4.3, whose corresponding specification state is a stuttering state) we call *non-visible* states.

Now, the main idea in the *visible-state* approach is to define a *visible* next-state function in the implementation machine. The visible next-state function *combines* the transitions from a *visible* state through zero or more *non-visible* states to the next *visible* state into one transition. (In Figure 4.2 this function is `II`.)

One way to define the *visible* next-state function is for the user to provide a function which, when given the current implementation state and input, will return the number of stuttering states the machine will cycle through. (In Figure 4.1 this function would be the constant function, 2.) We call this function `num_non_visible`. The PVS declaration for `num_non_visible` is:

```
num_non_visible: [Istate, input -> nat]
```

Then the new next-state function can be defined as follows:

```
visible_I(s, inp): Istate =
   next_to_the_n(I, s, (LAMBDA i: inp), num_non_visible(s, inp)).
```

Note that in the `visible_I` definition the current input is repeated as the input for each of the stuttering states. An alternative would be to have each stuttering transition consume an input. One drawback with this alternative is that the specification machine would also have to consume the same number of inputs, and thus would have to have access to the `num_non_visible` function. This would either require the use of an explicit time variable or would require the `num_non_visible` function to be defined solely in terms of the abstract machine state and input.

An alternative to using a `num_non_visible` function is to define the `visible_I` function in terms of a predicate that characterizes the visible states. Very often the proof of correctness needs to make use of such a predicate anyway. Nevertheless, using a `num_non_visible` predicate is useful in that it provides guidance to the theorem prover on how to direct its search.

The rest of the *visible-state* approach follows the basic methodology except that `visible_I` is used in place of `I`.

4.2 Slowing the Specification Machine Down

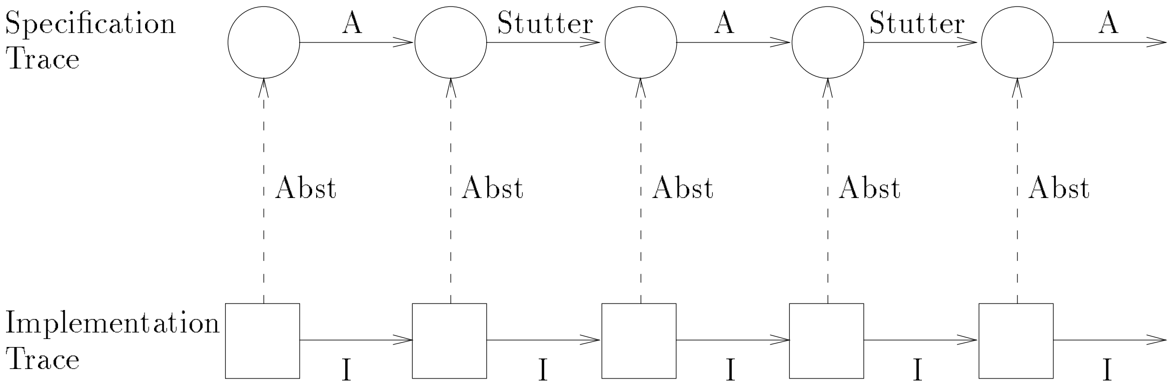

Figure 4.3: Stuttering the Specification Machine

We now describe two ways to modify the basic methodology so as to slow the specification machine down. Both methods slow the specification machine down by *stuttering* the specification machine, (i.e. add stuttering transitions that do not modify the specification state). In one method the **commutes** condition is modified to take stuttering into account. In the second method the specification next-state function is modified to have no effect (*stutter*) whenever the implementation machine transitions to a non-visible state.

4.2.1 Modifying the Commutes Condition

This approach is illustrated in Figure 4.4. Here we modify the **commutes** lemma to take stuttering into account:

Commutes:

$$\forall s, input : \neg\texttt{stutter}(\texttt{Abst}(\texttt{I}(s, input))) \supset \texttt{Abst}(\texttt{I}(s, input)) = \texttt{A}(\texttt{Abst}(s), input)$$

Commutes_stutter:

$$\forall s, input : \texttt{stutter}(\texttt{Abst}(\texttt{I}(s, input))) \supset \texttt{Abst}(\texttt{I}(s, input)) = \texttt{Abst}(s)$$

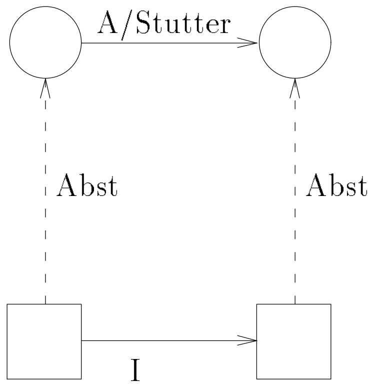

Figure 4.4: Commutes with stuttering

The above two conditions state that if the current implementation transition corresponds to a stuttering transition then the specification state remains unchanged, otherwise the specification state is updated as normal.

In addition to the change in the **commutes** condition, we also would need to prove that the machine does not stutter indefinitely. That is that the implementation machine won't loop indefinitely before reaching a new specification state. (This corresponds to proving total correctness, not just partial correctness.) This can be stated as:

Terminates:

$$\forall s, \mathit{input\_seq} : \exists n : \neg \mathtt{stutter}(\mathtt{next\_to\_the\_n}(\mathtt{I}(s, \mathit{input\_seq}), n))$$

Unfortunately, in our approach this is a little difficult to do directly since the decision as to whether the specification machine stutters or not is based on the implementation machine state at that time. As we have stated our approach there is no common clock that the two machines can refer to. Thus the specification has no way to determine the corresponding implementation state.

4.2.2 Stuttering the Specification Next-State Function

Stuttering the specification machine is more easily accomplished when an explicit time variable is used. This is because the decision as to whether the specification machine stutters or not is based on the implementation machine state at that time. Without an explicit time variable there is no common time that the two machines can refer to. Thus the specification has no way to determine the corresponding implementation state. Explicitly quantifying over time allows the specification machine to refer to the implementation state.

For example, let $\mathtt{S} : \mathtt{time} \rightarrow$ state, and $\mathtt{next\_S}$ be the original next-state function (without stuttering) then the following equation defines the new next-state relationship (with stuttering):

```
S(t+1) = IF stutter(t) THEN S(t)
            ELSE next_S(input_seq(t+1), S(t))
```

where `stutter` is a new predicate defined in terms of the implementation state variables.

In this approach, instead of explicitly representing the state using a record, we found it convenient to implicitly represent the state as a collection of functions from time. For example, in the trip meter example, the specification state would be represented by the following two functions:

```
LT: [time -> bool]
HT: [time -> bool]
```

The next-state function would also be represented implicitly as constraints relating the values of the state variables in consecutive time instances. For example, in the trip meter example, the next-state function would be replaced by the following two equations.

```
LT(t+1) = is(t+1) < LO OR (is(t+1) < LO + D AND LT(t))
HT(t+1) = is(t+1) > HI OR (is(t+1) > HI - D AND HT(t))
```

where `is` is an input sequence.

Now it is easy to add stuttering:

```
LT(t+1) = IF stutter(t) THEN LT(t)
             ELSE is(t+1) < LO OR (is(t+1) < LO + D AND LT(t))
HT(t+1) = IF stutter(t) THEN HT(t)
             ELSE is(t+1) > HI OR (is(t+1) > HI - D AND HT(t)).
```

In addition to adding the stuttering predicate, the abstraction function would have to be modified to indicate how a stuttering implementation state maps to a specification state. Note that this mapping might be different for different stuttering states.

In addition, because we no longer have explicit next-state functions, our definition of correctness needs to be changed. For details on this approach see [8], as well as the discussion and example found later in this paper.

Chapter 5

A Pipelined Microprocessor

We now illustrate how the two different approaches can be applied to a somewhat complicated example. This example is a pipelined microprocessor initially verified by Saxe, et al [8]. Figure 5.1 shows the non-pipelined version of the microprocessor which will be used as the specification. Figure 5.2 shows the pipelined version which will be used as the implementation. The reader is referred to [8] for an informal description of these circuits.

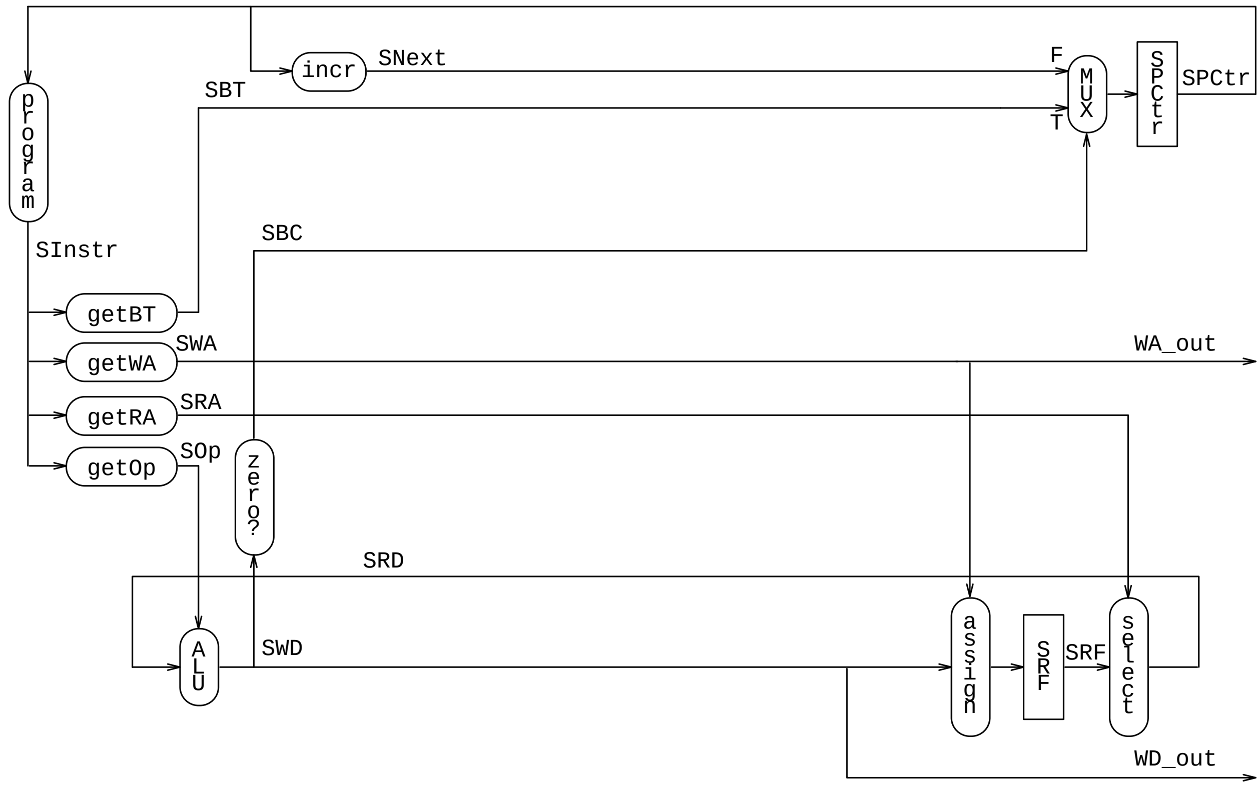

Figure 5.1: The non-pipelined microprocessor

In this *stripped-down* microprocessor two types of instructions are distinguished—normal instructions and branch instructions. Branch instructions are "detected" by performing a

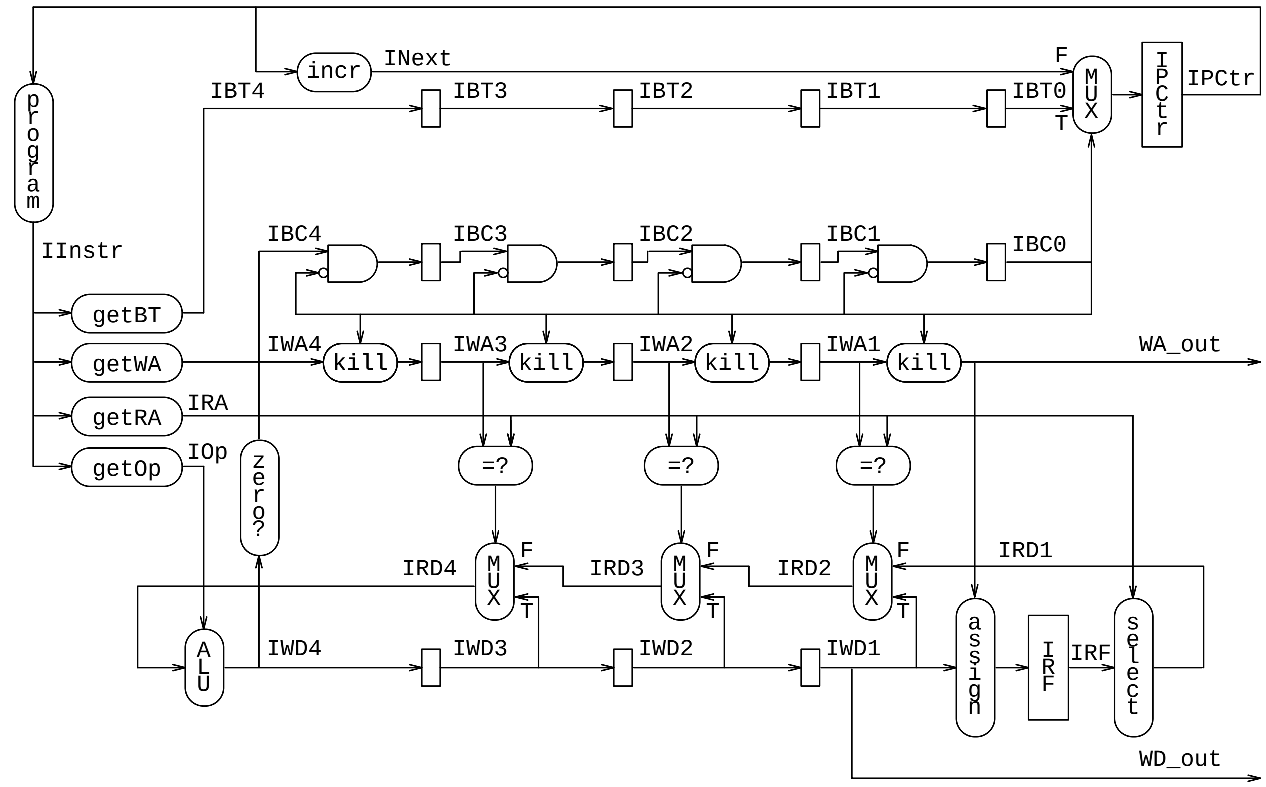

Figure 5.2: The pipelined microprocessor

`zero?` test on the output of the ALU. (Wire `SBC` in the specification, and wires `IBC_` in the implementation.)

The implementation machine differs from the specification machine in that it adds four pipeline stages. The pipeline is added by introducing four sets of four registers to the specification circuit. The registers along the `IBC_` path delay the decision on whether to branch by four cycles. The registers along the `IBT_` path delay the target of the branch for four cycles also. The `IWA_` path is necessary so that in case of a branch the four instructions that have already been executed will not change the register file. The **`kill`** elements along the `IWA_` path return the address **`null`** if their conditional input (`IBC0`) is high, otherwise they pass their input address through unchanged. When data is written to the null address the register file is unchanged. The `IWD_` path with the corresponding muxs and `=?` comparators implement the register bypass logic. If an instruction needs the value of a register that a previous instruction has written, but which hasn't been written in the register file yet (due to the pipeline delay), then this logic forwards the value of that register directly back to the ALU. (See [8] or the specification below for more details.)

When the pipeline is being flushed due to a branch instruction, one of the `IBC_` registers will be set to `T`. When this is the case, we say that the pipeline is *stalled*.

In addition to this implementation [8] gives a third implementation which is a more efficient version of this one obtained by retiming.

5.1 Speeding Up the Implementation Machine

We now describe the verification of the microprocessor using the *visible-state* approach.

5.1.1 Overview

We first give an overview of the verification. In section 5.1.2 we will go into more details of the PVS specification and verification.

5.1.1.1 Relating the Two Machines

In order to prove the implementation machine correct we need to specify how the two machines are related. To relate the two machines we need to specify which implementation states are visible, and the abstraction mapping that for any implementation state returns the corresponding specification state.

In the pipelined machine the visible states are those states in which the pipeline is not *stalled*:

```
visible(s) = NOT IBC0(s) AND NOT IBC1(s) AND NOT IBC2(s) AND NOT IBC3(s)
```

where `s` is an implementation state.

In PVS, if `Istate` is the type of a machine state, then we can define the type of a visible state as follows:

```
visible_state: TYPE = {s: Istate | visible(s)}
```

The `num_non_visible` function is:

```
num_non_visible(vs: visible_state) = IF IBC4(vs) THEN 5 ELSE 1 ENDIF
```

That is, in the case of a branch instruction, the next visible state is 5 state transitions later (after the pipeline has flushed), but in the case of a non-branch instruction, the very next state is visible.

Now the next-visible-state function can be defined:

$\texttt{visible\_I(vs: visible\_state) = I}^{\texttt{num\_non\_visible(vs)}}\texttt{(vs)}$[1].

Note that as part of typechecking, PVS will automatically generate lemmas to be proven that guarantee that the `visible_I` function preserves the `visible` predicate. Proving these lemmas correspond to verifying the pipeline invariants in [9].

We now need to specify the abstraction function that maps an implementation state to its corresponding specification state. The specification machine state is defined by the values of the program counter, `SPCtr`, and the register file, `SRF`. The abstraction function, `ABS` specifies how `SPCtr` and `SRF` are obtained from a visible implementation state. The definition of `ABS` is captured by the following equations:

$$\texttt{SPCtr}(\texttt{ABS}(\texttt{vs})) = \texttt{IPCtr}(\texttt{vs})$$

[1] This notation is just a shorthand for the `next_to_the_n` function introduced earlier.

$$\texttt{SRF(ABS(vs))} = \texttt{IRF(I}^3\texttt{(vs))}$$

In English, the abstraction mapping states that the program counters of the two machines are updated at the same time, but the register file of the specification machine corresponds to the register file of the implementation machine three (not necessarily visible) states later.

5.1.1.2 The Proof of Correctness

The proof consists of proving the following six lemmas:

Lemma 1 (Pipeline Invariant): $\texttt{visible(vs)} \supset \texttt{visible(I}^{\texttt{num\_non\_visible(vs)}}\texttt{(vs))}$

Proof The proof first does case analysis based on the expansion of `num_non_visible`. The remainder of the proof consists of simply rewriting the functions according to their definitions.

Lemma 2: $\texttt{IRD4(vs)} = \texttt{SRD(ABS(vs))}$

Proof The proof follows by performing case analysis on the possible values of `IRD4`. The value of `IRD4` depends on whether internal forwarding is needed or not; that is whether $\texttt{IRA} = \texttt{IWA3}$ or $\texttt{IRA} = \texttt{IWA2}$ or $\texttt{IRA} = \texttt{IWA1}$ or is not equal to any of them. While all the proofs were directed by a user, the decision on how to perform case analysis is one of the few places where *intelligent* human interaction is needed. (Another place where intelligent human interaction is needed is in the decision to make Lemma 2 a separate lemma.)

Lemma 3: $\texttt{IBC4(vs)} \supset \texttt{IPCtr(vs)} = \texttt{SPCtr(ABS(vs))}$

Lemma 4: $\neg\texttt{IBC4(vs)} \supset \texttt{IPCtr(vs)} = \texttt{SPCtr(ABS(vs))}$

Lemma 5: $\texttt{IBC4(vs)} \supset \texttt{IRF(vs)} = \texttt{SRF(ABS(vs))}$

Lemma 6: $\neg\texttt{IBC4(vs)} \supset \texttt{IRF(vs)} = \texttt{SRF(ABS(vs))}$

Proofs The proofs of lemmas 3 through 6 all follow the same form. First they all make use of Lemma 1. Lemmas 5 and 6 need Lemma 2. Secondly they expand the `ABS` function, and then the next-state functions the appropriate number of times. All these proofs make heavy use of rewriting.

Except for Lemma 2, all these lemmas could be generated automatically. Lemma 1 is generated by the PVS typechecker. The definition of the specification state generates two cases; one case for proving the program counters are equal and one case for proving that the register files are equal. Finally, the definition of the `num_non_visible` function can direct the theorem prover to split these two cases each in two: one case for a branch instruction, one for a non-branch instruction.

5.1.2 The PVS Specification

We now present in more detail the PVS specifications of the specification machine, implementation machine and the abstraction mapping between the two machines.

Figure 5.3 shows the specification machine. This is obtained in a straightforward manner from the presentation in [8].

```
mux [data: TYPE]: THEORY

BEGIN

  mux((flg: bool), (a:data), (b: data)): data =
    (IF flg THEN b ELSE a ENDIF)

END mux

register_file [addr: TYPE, data: TYPE] : THEORY

BEGIN

  null: addr

  register_file: TYPE = function[addr -> data]

  select((a: addr), (r: register_file)): data =
    r(a)

  assign((a: addr), (r: register_file), (d: data)): register_file =
    IF a = null then r ELSE r WITH [(a) := d] ENDIF

END register_file

abstract [prog_addr: TYPE, instruction: TYPE,
          reg_addr: TYPE, data: TYPE, alu_op: TYPE]: THEORY

BEGIN

USING mux[prog_addr], register_file[reg_addr, data]

  non_null_addr: TYPE = {a: reg_addr | a /= null}

  input: TYPE

  abs_state: TYPE = [# pc: prog_addr,
                       reg: register_file
                    #]

  as1, as2: VAR abs_state

  program: FUNCTION[prog_addr -> instruction]
  incr: FUNCTION[prog_addr -> prog_addr]
  getBT: FUNCTION[instruction -> prog_addr]
  getWA: FUNCTION[instruction -> non_null_addr]
  getRA: FUNCTION[instruction -> non_null_addr]
  getOp: FUNCTION[instruction -> alu_op]
```

Figure 5.3: Specification Machine (continues)

```
  getRA_null1: LEMMA (FORALL (x:instruction) : (getRA(x) = null) = FALSE)
  getRA_null2: LEMMA (FORALL (x:instruction) : (null = getRA(x)) = FALSE)
  getWA_null1: LEMMA (FORALL (x:instruction) : (getWA(x) = null) = FALSE)
  getWA_null2: LEMMA (FORALL (x:instruction) : (null = getWA(x)) = FALSE)

  alu: FUNCTION[data, alu_op -> data]
  zero?: FUNCTION[data -> bool]

  step_pc((pc: prog_addr), (reg: register_file)): prog_addr =
    mux(zero?(alu(select(getRA(program(pc)), reg), getOp(program(pc)))),
        incr(pc),
        getBT(program(pc)))

  step_reg((pc: prog_addr), (reg: register_file)): register_file =
    assign(getWA(program(pc)),
           reg,
           alu(select(getRA(program(pc)), reg), getOp(program(pc))))

  step((s: abs_state), (inp: input)): abs_state =
    {# pc := step_pc(pc(s), reg(s)),
       reg := step_reg(pc(s), reg(s)) #}

END abstract
```

Figure 5.3: Specification Machine

The specification state consists of only the clocked components in the specification circuit: the program counter and the register file. The combinational part of the circuit is represented using function application. This is done as a matter of convenience. Another possibility, which will be illustrated in section 5.2, associates a separate function with each wire of the circuit. Equations are then added which describe how the wires and the combinational elements are connected. The state remains unchanged, but the next-state function, `step`, refers to the wires in the rest of the circuit.

Figure 5.4 shows the PVS specification of the implementation machine. Again the implementation state consists of all the registers in the implementation machine. These include the program counter and register file, as well as all the registers added for pipelining.

The *visible* implementation state is a subtype of the implementation state, that satisfies the condition that none of the `IBC_` registers is high.

```
real [prog_addr: TYPE, instruction: TYPE,
      reg_addr: TYPE, data: TYPE, alu_op:TYPE]: THEORY
BEGIN
```

Figure 5.4: Implementation Machine (continues)

```
USING traces, abstract[prog_addr, instruction, reg_addr, data, alu_op],
      mux[data], mux[prog_addr], register_file[reg_addr, data]

  kill(b: bool, d: reg_addr): reg_addr =
   IF b THEN null ELSE d ENDIF

  real_state: TYPE = [# pc: prog_addr,
                        reg: register_file,
                        IBT3, IBT2, IBT1, IBT0: prog_addr,
                        IBC3, IBC2, IBC1, IBC0: bool,
                        IWA3, IWA2, IWA1: reg_addr,
                        IWD3, IWD2, IWD1: data
     #]

  visible(s: real_state): bool =
     NOT IBC0(s) AND NOT IBC1(s) AND NOT IBC2(s) AND NOT IBC3(s)

  visible_real_state: TYPE =
      {s: real_state | visible(s)}


  s: VAR real_state

  vs: VAR visible_real_state

  inp: VAR input

  next_pc(pc: prog_addr, ibt0: prog_addr, ibc0: bool): prog_addr =
    mux(ibc0, incr(pc), ibt0)

  next_reg(reg: register_file, ibc0: bool, iwa1: reg_addr,
           iwd1: data): register_file =
    assign(kill(ibc0, iwa1), reg, iwd1)

  next_ibt3(pc: prog_addr): prog_addr =
    getBT(program(pc))

  next_ibtx(ibtx_1: prog_addr): prog_addr =
    ibtx_1

  IRD1(s: real_state): data = select(getRA(program(pc(s))), reg(s))

  IRD2(s: real_state): data =
    mux((IWA1(s) = getRA(program(pc(s)))),
        IRD1(s),
        IWD1(s))
```

Figure 5.4: Implementation Machine (continues)

```
IRD3(s: real_state): data =
  mux((IWA2(s) = getRA(program(pc(s)))),
      IRD2(s),
      IWD2(s))

IRD4(s: real_state): data =
  mux((IWA3(s) = getRA(program(pc(s)))),
      IRD3(s),
      IWD3(s))

next_ibcx(ibcx_1: bool, ibc0: bool): bool =
  ibcx_1 AND (NOT ibc0)

next_iwax(iwax_1: reg_addr, ibc0: bool): reg_addr =
  kill(ibc0, iwax_1)

next_iwdx(iwdx_1: data): data =
  iwdx_1


next(s: real_state, inp: input): real_state =
  {# pc := next_pc(pc(s), IBT0(s), IBC0(s)),
     reg := next_reg(reg(s), IBC0(s), IWA1(s), IWD1(s)),
     IBT3 := next_ibt3(pc(s)),
     IBT2 := next_ibtx(IBT3(s)),
     IBT1 := next_ibtx(IBT2(s)),
     IBT0 := next_ibtx(IBT1(s)),
     IBC3 := next_ibcx(zero?(alu(IRD4(s), getOp(program(pc(s))))), IBC0(s)),
     IBC2 := next_ibcx(IBC3(s), IBC0(s)),
     IBC1 := next_ibcx(IBC2(s), IBC0(s)),
     IBC0 := next_ibcx(IBC1(s), IBC0(s)),
     IWA3 := next_iwax(getWA(program(pc(s))), IBC0(s)),
     IWA2 := next_iwax(IWA3(s), IBC0(s)),
     IWA1 := next_iwax(IWA2(s), IBC0(s)),
     IWD3 := next_iwdx(alu(IRD4(s), getOp(program(pc(s))))),
     IWD2 := next_iwdx(IWD3(s)),
     IWD1 := next_iwdx(IWD2(s))
   #}

pc_next: LEMMA pc(next(s, inp)) = next_pc(pc(s), IBT0(s), IBC0(s))

reg_next: LEMMA reg(next(s, inp)) = next_reg(reg(s), IBC0(s), IWA1(s), IWD1(s))

IBT3_next: LEMMA IBT3(next(s, inp)) = next_ibt3(pc(s))

IBT2_next: LEMMA IBT2(next(s, inp)) = next_ibtx(IBT3(s))

IBT1_next: LEMMA IBT1(next(s, inp)) = next_ibtx(IBT2(s))
```

Figure 5.4: Implementation Machine (continues)

```
  IBT0_next: LEMMA IBT0(next(s, inp)) = next_ibtx(IBT1(s))

  IBC3_next: LEMMA IBC3(next(s, inp)) = next_ibcx(zero?(alu(IRD4(s),
                                                  getOp(program(pc(s))))),
                                                IBC0(s))

  IBC2_next: LEMMA IBC2(next(s, inp)) = next_ibcx(IBC3(s), IBC0(s))

  IBC1_next: LEMMA IBC1(next(s, inp)) = next_ibcx(IBC2(s), IBC0(s))

  IBC0_next: LEMMA IBC0(next(s, inp)) = next_ibcx(IBC1(s), IBC0(s))

  IWA3_next: LEMMA IWA3(next(s, inp)) = next_iwax(getWA(program(pc(s))), IBC0(s))

  IWA2_next: LEMMA IWA2(next(s, inp)) = next_iwax(IWA3(s), IBC0(s))

  IWA1_next: LEMMA IWA1(next(s, inp)) = next_iwax(IWA2(s), IBC0(s))

  IWD3_next: LEMMA IWD3(next(s, inp)) = next_iwdx(alu(IRD4(s), getOp(program(pc(s)))))

  IWD2_next: LEMMA IWD2(next(s, inp)) = next_iwdx(IWD3(s))

  IWD1_next: LEMMA IWD1(next(s, inp)) = next_iwdx(IWD2(s))

  num_non_visible(vs: visible_real_state, inp: input): nat =
    IF zero?(alu(IRD4(vs), getOp(program(pc(vs)))))
       THEN 5
       ELSE 1
    ENDIF

  visible_next(vs, inp): visible_real_state =
     next_to_the_n(next, vs, (LAMBDA (i:nat): inp), num_non_visible(vs, inp))

END real
```

Figure 5.4: Implementation Machine

Note that in the implementation we decided that it would be useful to name some of the wires in the circuit explicitly. Namely the wires, `IRD4`, `IRD3`, `IRD2`, and `IRD1`. The next-implementation-state function, `next`, refers to the wire `IRD4`. This allowed us to use smaller expressions to relate the values of the wires to each other.

Also note that we introduced a series of auxiliary lemmas. These are the lemmas, `pc_next`, `reg_next`, ..., `IWD1_next`. These lemmas provide rewrite rules that give the values of the state variables for the next state in terms of the values of the state variables for the previous state. They can be proven or derived automatically from the next-state function. The reason that they are useful is so that the next-state function need never be expanded directly in the proof. Since the implementation-state function is so large expanding it unnecessarily can be very costly.

The end of the module gives the definitions of the `num_non_visible` function, and the `visible_next` function.

Figure 5.5 shows the abstraction mapping from an implementation state to the corresponding specification state. Note that the abstraction mapping only needs to map *visible* implementation states onto specification states.

```
verification [prog_addr: TYPE, instruction: TYPE,
              reg_addr: TYPE, data: TYPE, alu_op:TYPE]: THEORY

BEGIN

USING trace_equiv,
      real[prog_addr, instruction, reg_addr, data, alu_op],
      abstract[prog_addr, instruction, reg_addr, data, alu_op]

  vs: VAR visible_real_state

  as: VAR abs_state

  inp: VAR input

  someinput: input

  ABS(vs): abs_state =
    {# pc := pc(vs),
       reg := reg(next_to_the_n(next, vs, (LAMBDA (i:nat): someinput), 3)) #}

  verif: THEORY = trace_equiv[input, abs_state,
                              abs_state, step,
                              identity[abs_state],
                              visible_real_state, visible_next,
                              ABS, ABS]

END verification
```

Figure 5.5: Abstraction Mapping

As in the trip meter example, the verification conditions are generated automatically by instantiating the `trace_equiv` theory. We are not concerned with the output of the machine, so we use the identity function as the output function.

5.1.3 The Proof of Correctness

The theorems generated as *TCC*s are:

```
% Subtype TCC generated for
   % next_to_the_n(next, vs, (LAMBDA (i: nat): inp), num_non_visible(vs, inp))
visible_next_TCC1: OBLIGATION
  (FORALL inp, vs:
```

```
    NOT
      IBC0(next_to_the_n(next, vs,
                         (LAMBDA (i: nat): inp),
                         num_non_visible(vs, inp)))
      AND NOT
        IBC1(next_to_the_n(next, vs,
                           (LAMBDA (i: nat): inp),
                           num_non_visible(vs, inp)))
        AND NOT
          IBC2(next_to_the_n(next, vs,
                             (LAMBDA (i: nat): inp),
                             num_non_visible(vs, inp)))
          AND NOT
            IBC3(next_to_the_n(next, vs,
                               (LAMBDA (i: nat): inp),
                               num_non_visible(vs, inp))))

% Subtype TCC generated for
   % ABS
verif_TCC1: OBLIGATION surjective?[visible_real_state, abs_state](ABS)

% Assuming TCC generated for
   % trace_equiv[input, abs_state, abs_state, step,
   %             identity[abs_state], visible_real_state,
   %             visible_next, ABS, ABS]
verif_TCC2: OBLIGATION
  FORALL (s: visible_real_state), (inp: input):
    ABS(visible_next(s, inp)) = step(ABS(s), inp)

% Assuming TCC generated for
   % trace_equiv[input, abs_state, abs_state, step,
   %             identity[abs_state], visible_real_state,
   %             visible_next, ABS, ABS]
verif_TCC3: OBLIGATION
  FORALL (s: visible_real_state): ABS(s) = identity[abs_state](ABS(s))
```

The first *TCC* corresponds to Lemma 1 in section 5.1.1. The proof is straightforward. It just involves expanding the subtype predicate for the initial visible state, `vs`. Then expanding the `num_non_visible` function, then case splitting on the condition as to whether the instruction is a branch instruction or not. In each case the implementation machine is "run" for the appropriate number of steps by rewriting. The result is sent to the ground decision procedures which completes the proof.

The second *TCC* states that the `ABS` function is surjective. Actually this *TCC* is not provable. When `ABS` is expanded we are left with the requirement to prove that for any specification state there exists an implementation state that when executed for three cycles has the same register file as in the specification state. This is not provable given our current specification. We thus must take this as a condition that must be satisfied in order for the implementation to be correct. What this condition states, in effect, is that the machine must be run for three cycles before we can say anything useful about it. This corresponds to having the condition, $\mathtt{t} \geq 3$, as an antecedent to the correctness condition in the previous approach.

The third *TCC* is the commutes condition. It corresponds to lemmas 3-6 in section 5.1.1. It is proven by splitting the proof into the following four lemmas:

1. Show the `pc`s correspond for the case when the instruction is not a branch instruction (Lemma 4).

2. Show the `regfile`s correspond for the case when the instruction is not a branch instruction (Lemma 6).

3. Show the `pc`s correspond for the case when the instruction is a branch instruction (Lemma 3).

4. Show the `regfile`s correspond for the case when the instruction is a branch instruction (Lemma 5).

The proof of each of these cases requires expanding the `visible_next` function and the `step` function. Depending on which case we are proving this will require a varying number of expansions of the `next` function.

In the cases involving the program counter this is sufficient. In the cases involving the register file the following lemma is needed:

```
IRD4(vs) = select(getRA(program(pc(ABS(vs)))), reg(ABS(vs))).
```

This is Lemma 2 in section 5.1.1 with `SRD(ABS(vs))` expanded. This lemma states that the value being fed into the *ALU* in the implementation machine is the same as the value being fed into the *ALU* in the corresponding specification machine. This lemma is proven by doing case analysis on whether `IRA` = `IWA3` or `IRA` = `IWA2` or `IRA` = `IWA1` or is not equal to any of them. That is, on where `IRD4` is getting its value from.

5.2 Slowing Down the Specification Machine

5.2.1 Overview

We now briefly describe how we verified the microprocessor using the *stuttering* approach. Rather than give all the details we just highlight the differences between the verification using the *stuttering* approach and the verification using the *visible-state* approach. The reader is referred to section 5.2.2 and [8] for more details.

5.2.1.1 Relating the Two Machines

To relate the two machines we need to define when the specification machine stutters, and the abstraction mapping from the implementation machine to the specification machine.

The specification machine stutters exactly when the implementation machine goes through non-visible states. That is:

```
stutter(t) = IBC0(t) OR IBC1(t) OR IBC2(t) OR IBC3(t)
```

where `t` refers to time.

Note that this is the negation of the definition of `visible` defined in the previous section.

The abstraction mapping is similar to the abstraction mapping given in the previous section. There are several differences though. The only one we will note here is that the abstraction mapping for the *stuttering* approach needs to map the non-visible as well as the visible implementation states into a corresponding specification state.

5.2.1.2 The Proof of Correctness

The proofs are all very straightforward. The pattern they follow is to symbolically execute one step in the implementation machine, doing case analysis on the conditions that determine the next state. The execution is done by using the equations that make up the implementation machine and the equation that define the abstraction mapping as rewrite rules. PVS's builtin strategies can complete much of the proof. Again, the only intelligent human interaction needed is that involved in recognizing the need for and in proving Lemma 2 above.

Because *every* implementation state is related to a specification state, there is no need to prove a pipeline invariant. Also, while in the *visible-state* approach the proof was broken down into cases of whether the instruction was a branch instruction or not, in the *stuttering* approach the proof was broken down into separate cases for each different stuttering transition.

5.2.2 The PVS Specification

Figure 5.6 shows the specification machine presented in the "explicit time" style. This is obtained in a straightforward manner from the presentation in [8]. Time is represented by the natural numbers. The wires in the circuit are represented as functions from time to the types that the wires carry. Hardware elements are represented as functions from the types on their input lines to the types on their output lines. If necessary, axioms can be given that constrain these functions. The muxs are expanded to their equivalent `IF_THEN_ELSE_` form. The register file is represented by the two functions, `select` and `assign`.

```
signal [T : TYPE] : THEORY
BEGIN

signal : TYPE = [nat -> T]
a, b : VAR signal
t : VAR nat
x : VAR T
delay(a, x) : signal = (LAMBDA t: (IF t=0 THEN x ELSE a(t-1) ENDIF))
END signal

spec : THEORY
BEGIN
USING signal, mux

  instruction, data, address, operator, counter, file : TYPE
  null: address
  goodaddress: TYPE = {a: address | a/=null}
  flag : TYPE = bool

  m, n, t : VAR nat

  SPCtr, SBT, SNext : signal[counter]
  SRF : signal[file]
  incr: [counter -> counter]
  program : [counter -> instruction]
  getBT : [instruction -> counter]
  getWA : [instruction -> goodaddress]
  getRA : [instruction -> goodaddress]
  getOp : [instruction -> operator]

  ALU : [operator, data -> data]

  select : [file, address -> data]
  assign : [file,address,data -> file]

  zero? : [data -> flag]
  SBC, stalled? : signal[bool]

  SInstr : signal[instruction]
  WA_out : signal[address]
  SWA, SRA : signal[goodaddress]
  SRD, SWD, WD_out: signal[data]
  SOp : signal[operator]
```

Figure 5.6: Specification Machine (continues)

```
  SNextstate(t) : bool =
   (SPCtr(t+1) = (IF stalled?(t) THEN SPCtr(t)
                  ELSE mux(SBC(t), SBT(t), SNext(t)) ENDIF)
    AND
    SRF(t+1) = (IF stalled?(t) THEN SRF(t)
                ELSE assign(SRF(t), SWA(t), SWD(t)) ENDIF))

  SInv(t) : bool =
   (SInstr(t) = program(SPCtr(t)) AND
    SBT(t) = getBT(SInstr(t)) AND
    SWA(t) = getWA(SInstr(t)) AND
    SRA(t) = getRA(SInstr(t)) AND
    SOp(t) = getOp(SInstr(t)) AND
    SBC(t) = zero?(SWD(t)) AND
    SWD(t) = ALU(SOp(t), SRD(t)) AND
    SRD(t) = select(SRF(t), SRA(t)) AND
    SNext(t) = incr(SPCtr(t)) AND
    WA_out(t) = (IF stalled?(t) THEN null ELSE SWA(t) ENDIF) AND
    ((NOT stalled?(t)) IMPLIES (WD_out(t) = SWD(t))))

  SM : bool =
   (FORALL t: t>= 3 IMPLIES SNextstate(t) AND SInv(t))

END spec
```

Figure 5.6: Specification Machine

The function `stalled?` is the *stuttering* function that we described earlier. The definition of this function cannot be provided until the implementation machine is given.

The circuit is specified using two predicates, `SNextstate` and `SInv`. They are both functions of time. The first one states the constraints on the values of wires leaving clocked components. The second one states the constraints on wires between combinational elements. The predicate `SM` is the conjunction of `SNextstate` and `SInv`. It states the constraints that a machine must satisfy for it to meet the specification. After we give the implementation machine and abstraction mapping, our proof of correctness only involves showing that `SM` holds in that theory.

Figure 5.7 shows the implementation machine done in this style. Note that while the specification machine was presented as a single predicate, `SM`, the implementation machine is represented as a collection of equations. This is for practical reasons. The implementation machine could also have been presented as a predicate, but then the predicate would have to have been expanded explicitly and the resulting equations explicitly used as rewrite rules. By just giving the equations we can tell PVS to automatically use these equations as rewrite rules.

Note that the equations that describe the behavior of registers require that at `t` = 0 the value of the register is some unspecified value such as the `null` address, or `somedata`, `somecounter`, or `somefile`.

```
impl : THEORY
BEGIN
USING spec, signal

  IPCtr : signal[counter]
  IBC0: signal[bool]
  IRF: signal[file]
  IRD4: signal[data]

  somecounter: counter

  m, n, t : VAR nat
  ad, ad1, ad2: VAR address
  rf, rf1, rf2: VAR file
  A, B: VAR bool
  a, b, c, d: VAR data

  kill(ad, A) : address = (IF A THEN null ELSE ad ENDIF)

  INext(t): counter = incr(IPCtr(t))
  IInstr(t): instruction = program(IPCtr(t))
  IBT4(t): counter = getBT(IInstr(t))
  IBT3(t): counter = (IF t>0 THEN IBT4(t-1) ELSE somecounter ENDIF)
  IBT2(t): counter = (IF t>0 THEN IBT3(t-1) ELSE somecounter ENDIF)
  IBT1(t): counter = (IF t>0 THEN IBT2(t-1) ELSE somecounter ENDIF)
  IBT0(t): counter = (IF t>0 THEN IBT1(t-1) ELSE somecounter ENDIF)
  IWA4(t) : address = getWA(IInstr(t))
  IWA3(t): address = (IF t>0 THEN kill(IWA4(t-1), IBC0(t-1))
                          ELSE null ENDIF)
  IWA2(t): address = (IF t>0 THEN kill(IWA3(t-1), IBC0(t-1))
                          ELSE null ENDIF)
  IWA1(t): address = (IF t>0 THEN kill(IWA2(t-1), IBC0(t-1))
                          ELSE null ENDIF)
  IOp(t) : operator = getOp(IInstr(t))

  somedata: data

  IWD4(t): data = ALU(IOp(t), IRD4(t))
  IWD3(t):data = (IF t>0 THEN IWD4(t-1) ELSE somedata ENDIF)
  IWD2(t):data = (IF t>0 THEN IWD3(t-1) ELSE somedata ENDIF)
  IWD1(t):data = (IF t>0 THEN IWD2(t-1) ELSE somedata ENDIF)

  IRA(t): address = getRA(IInstr(t))
  IRD1(t): data = select(IRF(t), IRA(t))
  IRD2(t): data = mux((IWA1(t) = IRA(t)), IWD1(t), IRD1(t))
  IRD3(t): data = mux((IWA2(t) = IRA(t)), IWD2(t), IRD2(t))
  IRD4_defn: AXIOM
    IRD4(t) =  mux((IWA3(t) = IRA(t)), IWD3(t), IRD3(t))
```

Figure 5.7: Implementation Machine (continues)

```
  IBC4(t): bool = zero?(IWD4(t))
  IBC3(t): bool = (IF t>0 THEN (IBC4(t-1) AND NOT IBC0(t-1)) ELSE FALSE ENDIF)
  IBC2(t): bool = (IF t>0 THEN (IBC3(t-1) AND NOT IBC0(t-1)) ELSE FALSE ENDIF)
  IBC1(t): bool = (IF t>0 THEN (IBC2(t-1) AND NOT IBC0(t-1)) ELSE FALSE ENDIF)
  IBC0_defn: AXIOM IBC0(t) =
    (IF t>0 THEN (IBC1(t-1) AND NOT IBC0(t-1)) ELSE FALSE ENDIF)

  WA_out_defn: AXIOM WA_out(t)  = kill(IWA1(t), IBC0(t))
  WD_out_defn: AXIOM WD_out(t) = IWD1(t)

  somefile : file
  IRF_defn: AXIOM IRF(t) =
    (IF t>0 THEN assign(IRF(t-1), kill(IWA1(t-1), IBC0(t-1)), IWD1(t-1))
        ELSE somefile ENDIF)
  IPCtr_defn: AXIOM IPCtr(t) =
    (IF t>0 THEN mux(IBC0(t-1), IBT0(t-1), INext(t-1))
        ELSE somecounter ENDIF)

  assign_ax: AXIOM
    assign(rf, null, a) = rf

  select_assign: AXIOM
    select(assign(rf, ad, a), ad1) =
      (IF ((ad=ad1) AND ad1/=null) THEN a ELSE select(rf, ad1) ENDIF)
END impl
```

Figure 5.7: Implementation Machine

Figure 5.8 shows the abstraction mapping from an implementation state to the specification state. First note that the definition of `stalled?` is finally given in the `stalled?_abs` axiom. This equation states that the machine is stalled whenever one of `IBC0`, `IBC1`, `IBC2`, or `IBC3` are true. When the machine is stalled it is in the process of clearing the pipeline after a branch instruction has been detected.

Unlike in the *visible-state* approach, the abstraction mapping must specify the value of every wire in the specification machine. This is because in this approach the specification machine is not used as axioms, but as the theorem to be proven.

Note that the mapping consists of a set of equations, many of which do not mention implementation wires, and turn out to be the same equations as those found in the specification. This set of equations, taken as a whole, specifies for each specification wire, an equivalent term involving only implementation wires. A requirement on this mapping is that the set of equations must be a conservative extension of the implementation. See [8] for details.

The key equations that relate the implementation state to the specification state are `SRF_abs` and `SPCtr_abs`. Together they state that the effects of the instruction at the program counter at time `t` won't be reflected in the register file until time $t + 3$. Note also that the axiom `SPCtr_abs` must state the relationship between the specification program counter and the implementation machine for every stuttering state.

```
abst: THEORY
BEGIN
USING spec, signal, impl
  m, n, t : VAR nat
  SRF_abs: AXIOM SRF(t) = IRF(t)

  SPCtr_abs: AXIOM
    SPCtr(t) =
     (IF t>=3 THEN
        (IF IBC0(t-3) THEN IBT0(t-3)
           ELSIF IBC1(t-3) THEN IBT1(t-3)
           ELSIF IBC2(t-3) THEN IBT2(t-3)
           ELSIF IBC3(t-3) THEN IBT3(t-3)
           ELSE IPCtr(t-3) ENDIF)
        ELSE somecounter ENDIF)

  SNext_abs: AXIOM
    SNext(t) = incr(SPCtr(t))
  SInstr_abs: AXIOM
    SInstr(t) = program(SPCtr(t))
  SBT_abs: AXIOM SBT(t) = getBT(SInstr(t))
  SWA_abs: AXIOM SWA(t) = getWA(SInstr(t))
  SRA_abs: AXIOM SRA(t) = getRA(SInstr(t))
  SOp_abs: AXIOM SOp(t) = getOp(SInstr(t))

  SRD_abs: AXIOM SRD(t) = select(IRF(t), SRA(t))
  SWD_abs: AXIOM SWD(t) = ALU(SOp(t), SRD(t))
  SBC_abs: AXIOM SBC(t) = zero?(SWD(t))

  stalled?_abs: AXIOM
    stalled?(t) =
      (IF t>=3 THEN
        (IF IBC0(t-3) THEN TRUE
          ELSIF IBC1(t-3) THEN TRUE
          ELSIF IBC2(t-3) THEN TRUE
          ELSE IBC3(t-3) ENDIF)
        ELSE FALSE ENDIF)

  correctness: THEOREM SM

  SPCtr_corr: LEMMA
   t>=3 IMPLIES
    SPCtr(t+1) = (IF stalled?(t) THEN SPCtr(t)
                    ELSIF SBC(t) THEN SBT(t) ELSE SNext(t) ENDIF)
```

Figure 5.8: Abstraction Mapping (continues)

```
  SRF_corr: LEMMA
   t>=3 IMPLIES
    SRF(t+1) = (IF stalled?(t) THEN SRF(t)
                  ELSE assign(SRF(t), SWA(t), SWD(t)) ENDIF)

  WA_out_corr: LEMMA
   t>=3 IMPLIES WA_out(t) = (IF stalled?(t) THEN null ELSE SWA(t) ENDIF)

  WD_out_corr: LEMMA
   t>=3 IMPLIES (NOT stalled?(t)) IMPLIES (WD_out(t) = SWD(t))

  SRD_corr: LEMMA
    t>=3 IMPLIES SRD(t) = IRD4(t-3)

END abst
```

Figure 5.8: Abstraction Mapping

5.2.3 The Proof of Correctness

Given the above theories, the proof of correctness involves showing that the equations defining the specification machine hold, given the equations defining the implementation machine and the equations giving the abstraction mapping from the implementation machine to the specification machine. The statements of correctness can be found at the end of Figure 5.8.

The proofs are all very straightforward. The pattern they follow is to execute one step in the implementation machine, doing case analysis on the conditions that determine the next state. The execution is done by using the equations in the implementation and abstract theories as rewrite rules. All this can be done automatically using PVS's builtin strategies. The only *intelligent* human interaction needed is to do case analysis on the possible values of `IRD4`. The value of `IRD4` depends on whether internal forwarding is needed or not; that is whether `IRA` = `IWA3` or `IRA` = `IWA2` or `IRA` = `IWA1` or is not equal to any of them. In order for the proof to go through *efficiently* the user needs to direct the theorem prover at the appropriate time to split the proof according to these cases.

5.3 Comparing the Two Approaches

In this section we use the pipeline example of Saxe et al. as a reference point in comparing the two approaches. We compare the two approaches based on naturalness of specification and ease of proof.

5.3.1 Specification

The approaches discussed in this paper allow a very straightforward encoding of the original circuits. It should be possible to automatically translate circuit descriptions into the *logical specifications* used in these approaches. (Bronstein [2] describes such a translation into the

Boyer-Moore logic.) The approaches are all flexible enough to allow the user to encode the circuits in different ways that may facilitate an easier proof of correctness.

One difference is that in the *stuttering* approach the state of each machine consists of all the wires, not just the registers. This requires a more complete abstraction mapping between the implementation machine and specification machine.

The main difference between the two approaches is in how they deal with stuttering. In the *stuttering* approach, the specification must be changed to explicitly add stuttering. This can be done in a mechanical fashion, but however it is done, the specification is not complete until the implementation is defined.

On the other hand, the *visible-state* approach does not require modifying the specification at all. However, due to this, it suffers other draw backs. Most importantly, when there are external synchronous inputs, it must somehow consume them at the same rate in the two machines. (As there were no external inputs in our example, this was not an issue there.) Again there are two approaches for doing this. One way is to have the specification machine consume the "correct" number of inputs in each cycle—one input for each stalled cycle. (This is the approach taken in [9].) As mentioned earlier this potentially runs into the same problems as in the *stuttering* approach.

The approach we took is to have the implementation machine only consume one input for each visible-state transition. Unfortunately, this may not always be reasonable. For example, in the verification of microprocessors with interrupts (such as [9]), the number of non-visible states depends on the input interrupt signal. This interrupt can occur during non-visible transitions as well as during visible transitions. The next specification state depends on whether, and maybe even when, this interrupt occurs, and thus must have access to the input stream even during the non-visible state transitions.

This discussion would seem to indicate an advantage for the *stuttering* approach. However, we disagree. In situations where the behavior of the machine depends on what happens during these stuttering states, then the appropriate method for specifying such a system is to explicitly modify the specification machine to contain as much information as is needed to completely specify its functional behavior. If this is done, then the number of stuttering transitions that are necessary would be dependent on the specification state and input alone. Thus there would be no need for the specification machine to refer to the implementation state and thus there would be no need to introduce an explicit time variable, and the *visible-state* approach could be used.

5.3.2 Proof

Even though the underlying framework—and thus the ultimate statement of correctness for the two approaches—is different, the actual proof obligations are the same. Thus again, the main difference between the two approaches is in how they deal with stuttering, and how this affects the proof of correctness.

From the users point of view, we found the two proofs very similar. The only part of the proof that required any sort of intelligence was in dealing with the internal forwarding logic. However, from a mechanical point of view, the proofs were not identical.

The total amount of cpu time taken to prove the microprocessor correct was considerably less using the *stuttering* approach. This is due to the amount of rewriting that was needed to expand the next-state function.

In the *stuttering* approach, only one implementation next-state function was expanded for each branch in the proof. However, in the *visible-state* approach, there are many non-visible next state transitions for one visible next-state transition. Consider the following analysis.

In the cases where the instruction is a branch instruction, expanding the `visible_I` function requires expanding the `I` function five times. In addition, for the cases involving the register file, the `I` function must be expanded an additional three times (from the relationship stated in the `Abst` function). Thus, in the worst case, the `I` function must be expanded eight times. This involves considerably more rewriting and results in larger terms in the proof.

On the other hand, in the *stuttering* approach, the proof of correctness must take into account the relationship between the non-visible implementation states and their corresponding stuttering specification states. This adds additional cases to the proof.

One other drawback to the *stuttering* approach is in dealing with termination of rewriting. In the original approach rewriting terminates when there are no more `I` functions to rewrite. Since we are always dealing with a finite number of state transitions this will always terminate. In the *stuttering* approach there is no explicit next-state function, but rather an equation relating the state variables at time t with the state variables at time $t-1$. Thus without intelligent rewriting we could get infinite terms involving $t-1, t-2\cdots$. PVS is able to avoid infinite rewriting by rewriting from $t+1$ to t only when it can prove that t is a natural number (≥ 0). It will not rewrite to $t-1$ since it might be the case that $t-1 \not\geq 0$.

In either case the difficulty in automating the proof of correctness is in not doing case analysis willy-nilly, but being able to intelligently combine similar cases. We saw this in the proofs of correctness of the microprocessor. The user had to separately deal with the cases of internal forwarding. Once this was done the remainder of the proof was essentially mechanical.

5.3.3 Verification of Saxe et al.

The verification done in [8] uses the explicit-time style and *stuttering* approach to relating the two machines. Our comments concerning the *stuttering* approach apply equally well to the verification reported in [8]. The main difference between that verification and ours is in the theorem prover. Saxe et al. use the Larch Prover (LP). The main difference between LP and PVS is that LP is based on rewrite rules and is intended to be used in a mainly automatic fashion, while PVS is a sequent based prover and is intended to be used in a highly interactive manner. In spite of these differences the proofs of correctness in the two systems are pretty much the same. If anything, perhaps there is more user guidance in the proof in [8]. The automatic rewriting mechanisms in PVS as well as some of the builtin strategies allow the PVS proof to proceed in a highly automatic fashion.

5.3.4 Further Variations

We have dealt rather extensively with two types of timing differences between the specification and implementation machines. (Machines running at different rates and differences between the relative updating of state variables—the `pc` and `regfile` in the example.) We now discuss further complications that might arise in practical verification problems.

5.3.4.1 More Timing Differences

It might not be the case that one machine runs strictly slower than the other, but that at different times each machine might stutter (or even at the same time). For example the specification machine might take two cycles to execute an instruction, while the implementation machine takes three cycles.

This can be dealt with in the explicit time approach by stuttering both machines. In this case the actual amount of time to execute the instruction would have to be six cycles (the least common multiple of two and three).

In the other approach both machines would have their own next-visible-state function. In this case each machine would only consume one input per instruction.

5.3.4.2 Nondeterminism

In both our examples, the next-state function was deterministic. That is, given the current input and current state there was only one possible next state. A state machine may be nondeterministic though. There might be more than one possible next state for any given input and current state. This can be modeled by using a next-state *predicate* instead of a next-state function. However, in this case, rewriting could no longer be exploited as effectively in the proofs of correctness.

However, if only the specification machine were nondeterministic, then in the explicit time approach rewriting may still be fully exploited. This is because in this approach the specification machine is used as the statement of correctness, not as hypothesis to be used in the proof of correctness. Thus if both the implementation machine and level map can be specified equationally, then rewriting would still be fully exploited. This would not be the case in the *visible-state* approach.

5.3.4.3 Hierarchical Decomposition

Very often it is much easier to decompose the proof of correctness into simpler stages. Here we would verify that a simpler implementation machine correctly implements the specification machine. Once this is done, the simpler implementation machine becomes the new specification machine. We can then verify a more complicated implementation with respect to this machine. This process can be repeated as often as necessary. At the end we would have verified the correctness of the final implementation with respect to the initial specification.

5.3.4.4 Functional Composition

Very often we wish to develop our machines compositionally. For example in proving a microprocessor correct we would want to verify the correctness of the actual cpu independent of its memory system; treating the memory as a black box that works correctly. In addition we would verify that the memory system is correct independent of the cpu; assuming that the cpu works correctly. Given these two verifications we would need to verify that the cpu-memory combination works correctly.

However the actual interaction of the two systems might create problems. For example, there might be certain timing requirements on both the cpu and memory contained in the separate proofs of correctness. We would need to show that neither system violates those conditions when put together. One way of doing such things is by showing that each system does not violate some invariant of the other system. Unity [3] explores some of these issues. In regard to stating these invariants, the explicit timing approach might be more natural. However more work needs to go into this.

Chapter 6

Conclusions

Many verification problems can be stated as equivalences of state machines. We have presented a general methodology for verifying the equivalence of state machines in PVS. Several aspects of PVS facilitate using this methodology. Namely, the parameterized theories and their `ASSUMING` sections allow us to automatically generate verification conditions that prove the general equivalence of the state machines. In addition, the subtype mechanism in PVS allows us to easily state invariants on the states and have the lemmas necessary for proving those invariants automatically generated.

We have described two approaches in detail for handling timing differences between the specification machine and implementation machine. We applied the two approaches to a specific example—the verification of a pipelined microprocessor.

We compared the two approaches on several criteria. We found the explicit time approach to lead in some sense to smaller proofs, but perhaps at the expense of specification style. Finally, we presented ideas on how to extend the methodology to more complicated problems.

Bibliography

[1] Martín Abadi and Leslie Lamport. The existence of refinement mappings. In *Third Annual Symposium on Logic in Computer Science*, pages 165–175. IEEE, Computer Society Press, July 1988.

[2] Alexandre Bronstein. *MLP: String-Functional Semantics and Boyer-Moore Mechanization for the Formal Verification of Synchronous Circuits.* PhD thesis, Stanford University, 1989.

[3] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison Wesley, 1988.

[4] John Hopcroft and Jeffrey Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, 1979.

[5] Simon S. Lam and Udaya Shankar. A relational notation for state transition systems. *IEEE Transactions on Software Engineering*, 16(7):755–775, July 1990.

[6] Leslie Lamport. The temporal logic of actions. Technical Report 79, Digital Systems Research Center, 130 Lytton Avenue; Palo Alto, California 94301, December 1991.

[7] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *Automated Deduction - CADE-11, 11th International Conference on Automated Deduction, Lecture Notes in Artifical Intelligence*, pages 748–752. Springer Verlag, June 1992.

[8] James B. Saxe, Stephen J. Garland, John V. Guttag, and James J. Horning. Using transformations and verification in circuit design. Technical Report 78, Digital Systems Research Center, 130 Lytton Avenue; Palo Alto, California 94301, September 1991.

[9] Mandayam Srivas and Mark Bickford. Formal verification of a pipelined microprocessor. *IEEE Software*, 7(5):52–64, September 1990.