Inverting the Abstraction Mapping: A Methodology for Hardware Verification

(To be presented at Formal Methods in Computer Aided Design (FMCAD'96) Palo Alto, CA, November 1996.)

David Cyrluk*

Dept. of Computer Science, Stanford University, Stanford CA 94305 and
Computer Science Laboratory, SRI International, Menlo Park, CA 94025
cyrluk@cs.stanford.edu

Abstract. Abstraction mappings have become a standard approach to verifying the correctness of processors. When used in a straightforward manner this approach suffers from generating extremely large intermediate terms that have to be simplified.

In an interactive theorem prover the complete expansion of the abstraction mapping is not even possible. Yet, with human guidance it is interactive theorem proving that is applied to examples too large to be handled by automated methods.

We present a methodology for verifying the correctness of processors that aims to limit the size of intermediate terms generated in an interactive proof and to manage the complexity of the proof search.

The main idea of this methodology is that, instead of expanding the abstraction mapping and thereby introduce large implementation terms into the specification, we try to identify sub-terms of implementation terms that can be replaced by specification state variables. This is done by inverting the equations defining the abstraction mapping so to rewrite implementation terms into abstract state variables.

This method has been successfully applied to the verification of the DLX processor. It has also been applied to the verification of the ALU pipeline and Saxe pipeline and has simplified their proofs.

Finally, lessons learned from this methodology can help develop better heuristics employed by automatic methods.

1 Introduction

The use of abstraction mappings [1] for the verification of microprocessors and other sequential hardware circuits is commonplace [3, 7, 11]. Both automated stand-alone tools [3] and automated proof strategies for use in interactive theorem provers [6] have been developed based on the use of abstraction mappings. In [5] we developed a language *GTL*2 that is appropriate for specifying the correctness of sequential hardware circuits using abstraction mappings.

* This research was partially supported by SRI International, DARPA contract NAG2-703, and NSF grants CCR-8917606, CCR-8915663. Discussions with Deepak Kapur helped immensely in getting this paper written.

A common theme of all of these approaches is to expand the definitions of the specification machine, the implementation machine, and the abstraction mapping to generate large *if-then-else* expressions. These expressions are then compared by performing case analysis on the conditions appearing in them.

In large examples these expressions can get huge. In the PVS verification system [10], the complete expansion of the abstraction mapping for this machine causes PVS to run out of memory! There are over 100 embedded *if-then-else* expressions in the expansion of the abstraction of the register file. It takes over 10 screens to print the entire expansion.

In spite of this size, the automated tool finishes the verification in a matter of seconds. In PVS, larger and more complicated processor verifications have been carried out [11]. How can an automated tool that generates such large expressions verify their equivalence so quickly? Are there systematic methodologies that a user of an interactive proof system can make use of to structure the verification of large circuits?

Since large realistic processors are still beyond the capabaility of completely automatic tools, a methodology for structuring large verifications is needed.

In this paper we propose a methodology that can give the verifier of large processors some guidance in structuring an interactive proof attempt. We also look at whether this methodology can give insight into what makes the automated tools work so effectively and possibly how they might be improved.

At the highest level, the methodology rests on the observation that the proof of correctness of processors using the abstraction mapping approach comes down to identifying in the implementation machine certain structures that correspond to state elements in the abstract machine. The main example is that the implementation register file along with the pipeline bypass logic behaves like the abstract register file.

The aim of this paper is to translate this very high-level idea into a concrete proof methodology. The main idea is instead of using the abstraction mapping to introduce into the specification machine large implementation terms, we invert the abstraction mapping—hoping to replace implementation terms with simpler specification state variables.

In developing and implementing this methodology several insights became apparent. One is that in choosing conditions to perform case analysis on, we should select conditional formulas in the following order: first formulas that determine when the specification machine stalls, then formulas that appear in the specification machine, and then formulas that appear in the implementation machine. This observation should also be useful to finite state methods that rely on BDDs to encode state; the BDD variables should also be ordered as above. We will discuss additional insights in the conclusion.

In the next section we present the basic framework in which our methodology operates. Section 3 illustrates our methodology on a simple example. Section 4 discusses how our methodology is applied to the verification of the DLX processor. We end with comparisons to other work and conclusions.

2 Preliminaries

The basic framework is illustrated by the conventional commutative diagram shown in Figure 1.

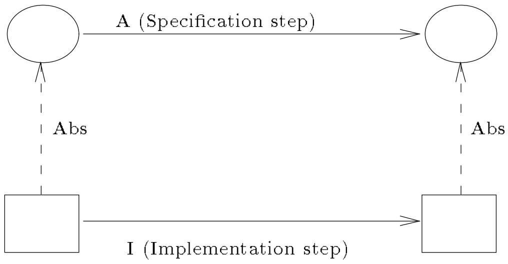

Fig. 1. The commutes diagram

Let V be a state variable of the specification machine and s an arbitrary implementation state. We denote by V_{spec} the term that results from following the left upper path through the commutative diagram. Similarly, V_{impl} denotes the term resulting from following the lower right path:

$$V_{spec} = \mathrm{A}(\mathrm{Abs}(s)), \tag{1}$$

$$V_{impl} = \mathrm{Abs}(\mathrm{I}(s)). \tag{2}$$

Common to both the stand-alone tools approach and the interactive theorem proving approach is the representation of the specification and implementation machines as transition systems. These transition systems are represented as equations that specify how the state variables are updated by the execution of one machine cycle. These equations are implementation equations in *GTL*2 [5] and can be stated in the following form.

$$\bigwedge_j V_j(\mathrm{next}(s)) = \mathrm{next_j}(s) \tag{3}$$

where V_j are state variables, next is the next state function (either `I` or `A` in Figure 1) and $\mathrm{next_j}$ is a function that specifies how state variable V_j changes.

Additionally, the abstraction mapping is given as equations that equate state variables in the specification machine with terms involving state variables of the implementation machine. These equations will be of this form:

$$\bigwedge_j V_j(\mathrm{Abs(s)}) = \mathrm{Abs_j}(s), \tag{4}$$

An example abstraction mapping for a three stage pipeline might be:

$$PC_A(\mathrm{Abs}(s)) = PC_I(s),$$

$$REG_A(\mathrm{Abs}(s)) = REG_I(\mathrm{next}(\mathrm{next}(\mathrm{next}(s)))).$$

In practice these equations have the effect of introducing implementation machine terms into V_{spec}. In the above three stage pipeline, where the parts of the abstraction mapping involves executing the implementation machine for three cycles, the terms introduced by the abstraction mapping can get quite large.

Now let us consider inverting the equations describing the abstraction mapping. That is, instead of equations of the form 4 we reverse the equations as follows:

$$\bigwedge_j \mathrm{Abs_j}(s) = V_j(\mathrm{Abs}(s)). \tag{5}$$

The abstraction mapping for our three stage pipeline would then become:

$$PC_I(s) = PC_A(\mathrm{Abs}(s)),$$

$$REG_I(\mathrm{next}(\mathrm{next}(\mathrm{next}(s)))) = \mathrm{REG_A}(\mathrm{Abs}(s)).$$

Now the term V_{spec} will only contain specification terms. The inverse abstraction mapping tells us how to transform V_{impl} into V_{spec}, i.e. how to transform terms involving implementation state variables into equivalent specification state variables. The hope is that since the resulting terms, V_{spec} and V_{impl} will be much smaller the verification should be easier and more manageable.

It turns out that this is only partially true. The difficulty is that there is no straightforward way to obtain the inverse abstraction mapping. That is, Equation 5 is not sufficient to invert the abstraction mapping. In particular, in different contexts there may be many terms equivalent to $\mathrm{Abs_j(s)}$. For example, consider the case in our three stage pipeline example when the pipe is filled with three store instructions. In this case, since store instructions do not change the register file, the register file remains unchanged after executing the machine three cycles. Thus, in a context in which the pipe is filled with store instructions we would have to know that we could replace $REG_I(s)$ with $REG_A(\mathrm{Abs}(s))$. Thus, to be complete, new equations that can rewrite all possible implementation terms that are equivalent to $REG_I(\mathrm{next}(\mathrm{next}(\mathrm{next}(s))))$ in different contexts need to be generated.

While in general it is difficult to generate such a complete set of rewrite rules, we have found this method greatly simplifies the verification of the processor verification benchmarks to which we have applied it. When used in the context of an interactive theorem prover, like PVS, the idea of inverting the abstraction mapping provides a powerful methodology to guide the human interaction.

On the other hand, while at this time the difficulty of generating a complete set of equations makes this approach problematic for a completely automatic tool, the ideas presented here can help to provide a better understanding as to how those automatic tools work, why their heuristics are effective, and perhaps how those heuristics can be improved.

In the next section we illustrate our method through the complete verification of the simplest one-stage pipeline.

3 A Simple Example

Our simple one-stage pipeline has *stall*, *dstn*, and *src* inputs. Its internal state consists of a register file, *SREG*. Two functions operate on register files: apply and update. Apply reads from a register file at a source address. Update writes to a register file. The specification of the pipeline is that if the *stall* input is not asserted then the value of *SREG* is updated at the *dstn* input to the current value of *SREG* at the *src* input.

The above specification is given by the following equation:

$$\begin{aligned} SREG(\text{step}(as)) = \text{cond}(&\text{NOT } stall(as), \\ &\text{update}(SREG(as), \\ &\qquad dstn(as), \\ &\qquad \text{f}(\text{apply}(SREG(as),\ src(as)))), \\ &SREG(as)). \end{aligned} \tag{S1}$$

The implementation consists of adding one pipeline stage. This is done by adding the registers *STALLD*, *SRCD*, *DSTND*, and *WBREG*. *STALLD*, *SRCD*, and *DSTND* store the previous cycle's inputs. The write back register *WBREG* stores the contents of the implementation register file, *IREG*, at the previous cycle's *src* input. This value is what will be written to the register file. Note that the equation, I2, that describes how *WBREG* is updated contains the pipeline bypass logic.

The implementation is given by the following equations:

$$\begin{aligned} IREG(\text{next}(is)) = \text{cond}(&STALLD(is), \\ &IREG(is), \\ &\text{update}(IREG(is),\ DSTND(is), \\ &\qquad WBREG(is))), \end{aligned} \tag{I1}$$

$$\begin{aligned} WBREG(\text{next}(is)) = \text{f}(\text{cond}(&STALLD(is), \\ &\text{apply}(IREG(is),\ src(is)), \\ &\text{cond}(src(is) = DSTND(is), \\ &\qquad WBREG(is), \\ &\qquad \text{apply}(IREG(is),\ src(is))))), \end{aligned} \tag{I2}$$

$$SRCD(\text{next}(is)) = src(is), \tag{I3}$$

$$DSTND(\text{next}(is)) = dstn(is), \tag{I4}$$

$$STALLD(\text{next}(is)) = stall(is). \tag{I5}$$

Finally, the abstraction mapping is given by these equations:

$$SREG(\text{Abs}(is)) = IREG(\text{next}(is)), \tag{Abs1}$$

$$src(\text{Abs}(is)) = src(is), \tag{Abs2}$$

$$dstn(\mathrm{Abs}(is)) = dstn(is), \tag{Abs3}$$
$$stall(\mathrm{Abs}(is)) = stall(is). \tag{Abs4}$$

The relationship between apply and update is given by the following equation:

$$\begin{aligned}
&\mathrm{apply}(\mathrm{update}(reg,\ value,\ dstn_addr),\\
&\quad src_addr)\\
&= \mathrm{cond}(src_addr = dst_addr,\\
&\qquad value,\\
&\qquad \mathrm{apply}(reg,\ src_addr)).
\end{aligned} \tag{APP_UPD}$$

3.1 Traditional Method

In this section we illustrate the verification of the above pipeline using the traditional method similar to methods discussed in [3, 6]. This method simply expands the definitions of the specification machine, implementation machine, and abstraction mapping. The resulting terms which contain embedded conditionals are then compared by case splitting on those conditionals and simplified using reasoning about equalities and updates.

In the above example let $SREG_{spec} = \mathrm{SREG}(\mathrm{step}(\mathrm{Abs}(is)))$ and $SREG_{impl} = \mathrm{SREG}(\mathrm{Abs}(\mathrm{next}(is)))$ then the correctness condition corresponding to the commutative diagram is:

$$SREG_{spec} = SREG_{impl}. \tag{6}$$

After rewriting we have:

$$\begin{aligned}
&SREG_{spec} =\\
&\quad \mathrm{cond}(\mathrm{NOT}\ stall(is),\\
&\qquad \mathrm{update}(\mathrm{cond}(STALLD(is),\\
&\qquad\qquad IREG(is),\\
&\qquad\qquad \mathrm{update}(IREG(is),\ DSTND(is),\ WBREG(is))),\\
&\qquad\quad dstn(is),\\
&\qquad\quad \mathrm{f}(\mathrm{apply}(\mathrm{cond}(STALLD(is),\\
&\qquad\qquad\quad IREG(is),\\
&\qquad\qquad\quad \mathrm{update}(IREG(is),\ DSTND(is),\ WBREG(is))),\\
&\qquad\qquad src(is)))),\\
&\qquad \mathrm{cond}(STALLD(is),\\
&\qquad\quad IREG(is),\\
&\qquad\quad \mathrm{update}(IREG(is),\ DSTND(is),\ WBREG(is)))),
\end{aligned}$$

and

$$\begin{aligned}
&SREG_{impl} =\\
&\quad \mathrm{cond}(\mathrm{stall}(is),\\
&\qquad \mathrm{cond}(STALLD(is),\\
&\qquad\quad IREG(is),\\
&\qquad\quad \mathrm{update}(IREG(is),\ DSTND(is),\ WBREG(is))),
\end{aligned}$$

$$
\begin{aligned}
&\text{update}(\text{cond}(STALLD(is),\\
&\qquad\qquad IREG(is),\\
&\qquad\qquad \text{update}(IREG(is),\ DSTND(is),\ WBREG(is))),\\
&\qquad dstn(is),\\
&\qquad \text{cond}(STALLD(is),\\
&\qquad\qquad \text{f}(\text{apply}(IREG(is),\ src(is))),\\
&\qquad\qquad \text{cond}(src(is) = DSTND(is),\\
&\qquad\qquad\qquad \text{f}(WBREG(is)),\\
&\qquad\qquad\qquad \text{f}(\text{apply}(IREG(is),\ src(is))))))).
\end{aligned}
$$

If we do case analysis on $src(is) = DSTND(is)$ and $stall(is)$ both $SREG_{spec}$ and $SREG_{impl}$ reduce to the same term. Note that we might also have decided to do case analysis on $STALLD(is)$ which would have been unnecessary.

3.2 Inverting the abstraction mapping

We now redo the same proof but use the equations for the abstraction mapping in the reverse direction:

$$IREG(\text{next}(is)) = SREG(\text{Abs}(is)), \tag{Abs1\_rev}$$
$$src(is) = src(\text{Abs}(is)), \tag{Abs2\_rev}$$
$$dstn(is) = dstn(\text{Abs}(is)), \tag{Abs3\_rev}$$
$$stall(is) = stall(\text{Abs}(is)). \tag{Abs4\_rev}$$

Applying these rules we get:

$$
\begin{aligned}
&SREG_{spec} =\\
&\quad \text{cond}(\text{NOT } stall(\text{Abs}(is)),\\
&\qquad \text{update}(SREG(\text{Abs}(is)),\ dstn(\text{Abs}(is)),\\
&\qquad\qquad \text{f}(\text{apply}(SREG(\text{Abs}(is)),\ src(\text{Abs}(is))))),\\
&\qquad SREG(\text{Abs}(is))),
\end{aligned}
$$

and (with expanding Abs once before applying the rewrite rules)

$$
\begin{aligned}
&SREG_{impl} = IREG(\text{next}(\text{next}(is))) =\\
&\quad \text{cond}(stall(\text{Abs}(is)),\\
&\qquad SREG(\text{Abs}(is)),\\
&\qquad \text{update}(SREG(\text{Abs}(is)),\ dstn(\text{Abs}(is)),\\
&\qquad\qquad \text{f}(\text{cond}(STALLD(is),\\
&\qquad\qquad\qquad \text{apply}(IREG(is),\ src(\text{Abs}(is))),\\
&\qquad\qquad\qquad \text{cond}(src(\text{Abs}(is)) = DSTND(is),\\
&\qquad\qquad\qquad\qquad WBREG(is),\\
&\qquad\qquad\qquad\qquad \text{apply}(IREG(is),\ src(\text{Abs}(is)))))))).
\end{aligned}
$$

There is a slight advantage over the traditional method in that the terms generated are smaller. In a realistic example with a five stage pipeline the terms would be considerably smaller.

Unfortunately we also have a huge disadvantage: the above equation no longer simplifies. No rewrite rules are applicable and no amount of case analysis can reduce the above equation to true.

The reason for this is that when we inverted the abstraction mapping we no longer have a confluent set of rewrite rules. Much like in the Knuth-Bendix completion procedure [8], we need to add equations to make the resulting system canonical.

Along with simple properties of cond, the following three rules can be added to finish the proof:

$$\begin{aligned}&\text{cond}(STALLD(is),\\ &\quad IREG(is),\\ &\quad \text{update}(IREG(is),\ DSTND(is),\ WBREG(is)))\\ &\qquad\qquad = SREG(\text{Abs}(is))\end{aligned} \tag{ABS1$'$}$$

$$\begin{aligned}&\text{cond}(src\_addr = dst\_addr,\\ &\quad value,\\ &\quad \text{apply}(reg,\ src\_addr))\\ &\qquad = \text{apply}(\text{update}(reg,\ value,\ dstn\_addr),\\ &\qquad\qquad src\_addr)\end{aligned} \tag{APP\_UPD\_REV}$$

$$\begin{aligned}&\text{cond}(p,\ \text{apply}(then,\ addr),\\ &\quad \text{apply}(else,\ addr))\\ &\qquad = \text{apply}(\text{cond}(p,\ then,\ else),\ addr)\end{aligned} \tag{PUSH\_COND}$$

Using APP_UPD_REV

$$\begin{aligned}&\text{cond}(src(\text{Abs}(is)) = DSTND(is),\\ &\quad WBREG(is),\\ &\quad \text{apply}(IREG(is),\ src(\text{Abs}(is))))\end{aligned}$$

rewrites to

$$\begin{aligned}&\text{apply}(\text{update}(IREG(is),\ DSTND(is),\ WBREG(is)),\\ &\quad src(\text{Abs}(is)))\end{aligned}$$

Thus $SREG_{impl}$ is rewritten to

$$\begin{aligned}&\text{cond}(stall(\text{Abs}(is)),\\ &\quad SREG(\text{Abs}(is)),\\ &\quad \text{update}(SREG(\text{Abs}(is)),\ dstn(\text{Abs}(is)),\\ &\qquad \text{f}(\text{cond}(STALLD(is),\\ &\qquad\quad \text{apply}(IREG(is),\ src(\text{Abs}(is))),\\ &\qquad\quad \text{apply}(\text{update}(IREG(is),\ DSTND(is),\ WBREG(is)),\\ &\qquad\qquad src(\text{Abs}(is))))))).\end{aligned}$$

PUSH_COND will lift the embedded applys so that ABS1$'$ rewrites $SREG_{impl}$ to:

$$
\begin{aligned}
&\mathrm{cond}(\mathit{stall}(\mathrm{Abs}(\mathit{is})),\\
&\qquad \mathit{SREG}(\mathrm{Abs}(\mathit{is})),\\
&\qquad \mathrm{update}(\mathit{SREG}(\mathrm{Abs}(\mathit{is})),\ \mathit{dstn}(\mathrm{Abs}(\mathit{is})),\\
&\qquad\qquad \mathrm{f}(\mathrm{apply}(\mathit{SREG}(\mathrm{Abs}(\mathit{is})),\ \mathit{src}(\mathrm{Abs}(\mathit{is})))))).
\end{aligned}
$$

Now case analysis on $\mathit{stall}(\mathrm{Abs}(\mathit{is}))$ finishes the proof. Note also that this is the only choice for case analysis.

The basic lesson from this example that we will apply to more complicated examples is that if we can generate the necessary rules, then by inverting the abstraction mapping we can limit our case analysis to conditions that appear in the specification machine. This should lead to a big savings since there are usually much fewer conditions in the specification than in the implementation.

Put another way, we divide the proof of correctness into two parts. The first part is the inversion of the abstraction mapping and the generation of the new rules. This requires some understanding of the implementation and, as will be seen, may require some user guidance. The second part is more automatic and essentially uses our core hardware strategy from [6] to rewrite using the inverted abstraction mapping, the machine definitions, and the new rules, and then perform case analysis on the resulting terms. If we have generated enough rules the case analysis need only be on specification conditions.

As will be seen in Section 4 we can generate different instances of the inverse abstraction mapping lazily by simplifying the inverse abstraction mapping at the same time that we perform case analysis on the specification machine

3.3 Generating new equations

We now look at how the rules needed above can be generated.

One possible method for generating new rules is to run a conditional Knuth-Bendix completion procedure on the implementation equations, reverse abstraction mapping, and domain equations. We have done this by hand on several small examples, but the cost of blindly running a completion procedure on large examples may quickly exceed the potential benefit.

However, the ideas behind completion can help. Let us look more closely at our simple example. Rule ABS1$'$ is generated by noticing that the addition of the rule Abs1_rev causes $\mathit{IREG}(\mathrm{next}(\mathit{is}))$ to rewrite to two terms, the left and right hand sides of equation ABS1$'$. Thus ABS1$'$ is added to eliminate that source of non-confluence.

We could have continued in this fashion generating many conditional rewrite rules. Instead we note that the goal of our rewriting process is to replace implementation terms with corresponding specification terms. At a high level this corresponds to recognizing in the implementation circuit instances of the specification circuit.

In processor verification examples this corresponds to recognizing instances of the register file distributed (sometimes in a hidden fashion) in the implementation machine. In this example the bypass logic combined with the implementation register file behave as one larger register file. In more complicated examples this observation still holds, although it is somewhat obscured by the presence of additional logic that might stall the pipeline or provide branch prediction, etc.

By using equation APP_UPD_REV we are in essence looking for instances of register files in the implementation machine. In this case the addition of the one rule PUSH_COND allowed us to eventually apply APP_UPD_REV.

In general it is not always so easy to generate a simple rule that will allow us to syntactically recognize applications of APP_UPD_REV. The particular form of the implementation equations I1-I5 were important for the above example to work so simply.

As rule I2 was written it is easy to recognize that it can be rewritten by equation APP_UPD_REV and PUSH_COND to the following rule before the proof is even begun:

$$\begin{aligned}\mathrm{WBREG}(\mathrm{next}(is)) = \mathrm{f}(\mathrm{cond}(&\mathrm{STALLD}(is),\\ &\mathrm{apply}(\mathrm{IREG}(is), \mathrm{src}(is)),\\ &\mathrm{apply}(\mathrm{update}(\mathrm{IREG}(is),\\ &\qquad\mathrm{DSTND}(is),\\ &\qquad\mathrm{WBREG}(is)),\\ &\quad\mathrm{src}(is))))\end{aligned} \tag{I2$'$}$$

However, I2 could originally have been written in such a way as to make it difficult to transform it into I2$'$. There are several possible approaches to dealing with this difficulty:

One is to canonize the right hand sides of implementation equations into a form that can be matched against the left hand side of rule APP_UPD_REV. A possible canonical form can be generated by pushing uninterpreted symbols out of cond terms and re-ordering the conditionals that appear so that equality between addresses appear last.

Another possibility is to build into the matching routine enough understanding of cond and updates so that it can match the left hand side of APP_UPD_REV.

In an interactive prover such as PVS the most reasonable method is to have the user help with the identification. In our simple example the user could actually have rewritten equation I2 as the simpler equation I2$'$.

An intelligent user could have gone even further and defined a new function stall_update with the following definition:

$$\begin{aligned}&\mathrm{stall\_update}(\mathit{stall?}, \mathit{reg}, \mathit{addr}, \mathit{val}) =\\ &\quad \mathrm{cond}(\mathit{stall?}, \mathit{reg}, \mathrm{update}(\mathit{reg}, \mathit{addr}, \mathit{val}))\end{aligned}$$

The presence of this function could have aided the user (or automated tool) in recognizing that the right hand sides of equations S1, I1, and I2 were all applications of stall_update.

4 The DLX Processor

We now illustrate how this methodology is applied to the DLX processor. The example that we verified was translated from the description used by Burch and Dill in [3]. Our verification differs from theirs in two respects. Firstly, we deal with stuttering slightly differently. Where they stall the specification machine based on a predicate indicating whether the implementation machine is in a visible state, we use an oracle to specify how many cycles the implementation needs to move from one visible state to the next.

Secondly, their verification is done using their automated tool, whereas our verification is done in PVS, an interactive theorem prover. In some ways this difference makes our task both easier and more difficult. It is easier since the user can guide the theorem prover. It is more difficult because the intermediate goals generated by the prover must be manageable and understandable to the user. Furthermore, the general purpose nature of the theorem prover sometimes results in inefficiencies that a special purpose tool can avoid. In practice this means that the automated tool can generate much larger terms before blowing up.

A separate verification of the DLX was done in [2]. This verification was done by hand and represents an extreme in human guidance. This proof was done by decomposing the pipeline into small stepwise refinements that were separately verified.

Our approach falls in between these two extremes. It provides a guide and methodology for human guidance, but automates most of the proof and as will be seen requires little more insight from the user than was required in our simple toy example.

4.1 Setup

We now give some details of the specification and verification of DLX.

The specification state consists of four state variables: `pc, imem, reg, dmem`. The implementation state consists of seventeen state variables.

The DLX is a five stage pipeline. The default execution of the pipeline assumes that branches are not taken. If it turns out that a branch is taken then the pipeline takes an extra cycle to correct for assuming the wrong next program counter. Additionally the pipeline does not assume a load delay slot. Thus, the pipeline needs to be stalled if the instruction following a load depends on the destination of the load.

The above two stall conditions are defined in the DLX processor with the following equations:

```
branch_taken(s) =
        (NOT bubble_id(rs)) AND
        (   opcode_class(opcode_of(instr_id(rs))) = J
         OR (    opcode_class(opcode_of(instr_id(rs))) = BEQZ
             AND next_a(rs) = 0)),
stall_issue(s) =
        (NOT bubble_ex(s)) AND
        opcode_class(opcode_ex(s)) = LOAD AND
        (NOT bubble_id(s)) AND
        (   rf1_of(instr_id(s)) = dest_ex(rs)
         OR rf2_of(instr_id(s)) = dest_ex(s)).
```

Thus the following predicate defines when an implementation state is visible:

```
visible(s) = NOT branch_taken(s) AND NOT stall_issue(s)
```

This is the negation of the stalling condition used in [3]. In their approach the user also needs to provide this information.

The user also needs to provide an oracle that specifies how many cycles the machine needs to execute to reach the next visible state:

```
num_cycles(vs) =
  IF stall_issue(next(vs))
     THEN IF branch_taken(next(next(vs)))
             THEN 3
             ELSE 2
          ENDIF
  ELSIF branch_taken(next(vs))
     THEN 2
     ELSE 1
  ENDIF.
```

The visible predicate and the num_cycles function are important aides in guiding the proof of correctness. They provide a definition of the context in which specification states and implementation states need to be equal. They also specify when the specification machine stutters. Thus even though they are technically part of the implementation machine they are also implicitly part of the specification machine and provide additional sources of case analysis.

An additional source of up-front user guidance is in transforming the following definitions. a and b are the two inputs to the alu. The bypass logic is embedded in their definition.

```
a(next(rs)) =
  IF rf1_of(instr_id(rs)) = reg0
     THEN 0
  ELSIF rf1_of(instr_id(rs)) = dest_ex(rs)
     THEN result_mem(rs)
  ELSIF rf1_of(instr_id(rs)) = dest_mem(rs)
     THEN result_wb(rs)
     ELSE reg(rs)(rf1_of(instr_id(rs)))
  ENDIF,
b(next(rs)) =
  IF opcode_class(opcode_of(instr_id(rs))) = ALU_IMMED
     THEN short_immed_of(instr_id(rs))
  ELSIF rf2_of(instr_id(rs)) = reg0
     THEN 0
  ELSIF rf2_of(instr_id(rs)) = dest_ex(rs)
     THEN result_mem(rs)
  ELSIF rf2_of(instr_id(rs)) = dest_mem(rs)
     THEN result_wb(rs)
     ELSE reg(rs)(rf2_of(instr_id(rs)))
  ENDIF
```

to

```
a(next(rs)) = select(assign(assign(next_reg(rs), dest_mem(rs),
                                   next_result_wb(rs)),
                            dest_ex(rs), next_result_mem(rs)),
                     rf1_of(instr_id(rs))),
b(next(rs)) =
   IF opcode_class(opcode_of(instr_id(rs))) = ALU_IMMED
     THEN short_immed_of(instr_id(rs))
     ELSE
       select(assign(assign(next_reg(rs),
                             dest_mem(rs),
                             next_result_wb(rs)),
                      dest_ex(rs),
                      next_result_mem(rs)),
               rf2_of(instr_id(rs)))
   ENDIF
```

where `select` and `assign` are defined as:

```
select(reg, addr) = IF addr = 0
                      THEN 0
                      ELSE apply(reg, addr)
                    ENDIF,
assign(reg, addr, data) = IF addr = 0
                            THEN reg
                            ELSE update(reg, addr, data)
                          ENDIF.
```

This is the same type of transformation required in our simple example where equation `I2` was transformed into equation I2′.

The inverse abstraction mapping is given as:

```
pc(vs) = pc(ABS(vs))
reg(next(next(next(next(vs))))) = reg(Abs(vs))
dmem(next(next(next(vs)))) = dmem(Abs(vs))
imem(vs) = imem(Abs(vs))
```

4.2 Proof

Now the proof proceeds essentially in the manner specified by our microprocessor strategy defined in [6]. Case analysis is performed on the `num_cycles` function. This yields different goals for each branch in this function. In each goal the two terms corresponding to the two paths in the commutative diagram are expanded much like in our simple example. Case analysis is then performed on the conditions in the specification side of the commutative diagram.

A method is used lazily to generate additional rules corresponding to specializations of the inverse abstraction mapping in different contexts. The inverse abstraction mapping is kept as a hypothesis in the proof goal. As the proof goal is split, the left hand side of the inverse abstraction mapping is further simplified in each context. This simplified inverse mapping is then used to rewrite terms in the rest of the proof goal that are equivalent to an abstract state variable.

This is enough to finish automatically all but two cases. In the remaining two cases the conditions that are hidden in the definitions of `stall_issue` and `branch_taken` are brought out and case analysis is performed on them. This still does not completely finish the proof. Goals of the following form are left:

```
  next_dest_ex(s!1) ≠ rf1_of(apply(imem(Abs(s!1)), pc(Abs(s!1))))
  AND assign(reg_file_term,
             next_dest_ex(s!1),
             next_result_wb(next(next(s!1)))) = reg(Abs(s!1))
IMPLIES
   select(assign(reg_file_term,
                 next_dest_ex(s!1),
                 next_result_mem(next(s!1))),
          rf1_of(apply(imem(Abs(s!1)), pc(Abs(s!1)))))
  = select(reg(Abs(s!1)), rf1_of(apply(imem(Abs(s!1)),
           pc(Abs(s!1))))).
```

Goals of this form are obviously true, but currently have to be left to the user to complete. Note that the above goal would have been automatically proven if we had a rule of the form

```
dstn_addr ≠ src_addr IMPLIES
    select(assign(reg_file_term, dstn_add, value1),
           src_addr)
     = select(assign(reg_file_term, dstn_addr, value2),
              src_addr).
```

This is another rule that is needed to complete the inversion of the abstraction mapping. It indicates an incompleteness of our lazy method of inverting the abstraction mapping. Our method only looks at possible boolean contexts, not at functional contexts. Thus rules that describe how `reg(Abs(s!1))` behaves inside different `select`s and `assign`s were never generated.

5 Conclusions, Future Work and Comparison with Other Work

We have presented a methodology for structuring the human guidance of processor proofs in an interactive prover. We have seen how this methodology while incomplete allows us to automate much of the verification and provide guidance where human input is needed.

The methodology also gives us insight why the automatic methods work well. In the method of [3] they generate extremely large *if-then-else* terms represented in a DAG structure. A large part of this *if-then-else* term is devoted to different instances of the abstraction mapping being fully expanded. Their method also employs a heuristic to rewrite conditional terms into register file terms [4]. As our proof indicates if this is done only a relatively few number of case analysis (relative to the size of the whole if-then-else) is needed to finish the proof. Our method also indicates that if the specification is not written right this heuristic may not be useful.

Our method also indicates that a good heuristic is to choose to split on specification conditions before splitting on conditions found in the expanded abstraction mapping. It would be interesting to see how well this heuristic works when in employed in an automatic procedure.

Additionally as the automatic methods tackle larger problems they are likely to hit a point where some form of human guidance is needed. Such guidance is likely to be similar to the type that users of the Boyer-Moore theorem prover use—namely providing additional lemmas. The methodology in this paper can give ideas on what additional lemmas are likely to be helpful.

Finally this is just the first experiments in using this methodology. With more experience we hope to be able to develop additional ideas on how to more fully automate the inversion of the abstraction mapping.

References

1. Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.
2. E. Boerger and S. Mazzanti. A correctness proof for pipelining in risc architectures. Dimacs technical report, DIMACS, 1996. (To appear).
3. J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In David Dill, editor, *Computer-Aided Verification, CAV '94*, volume 818 of *Lecture Notes in Computer Science*, pages 68–80, Stanford, CA, June 1994. Springer-Verlag.
4. Jerry Burch. Personal Communication.
5. D. Cyrluk and P. Narendran. Ground temporal logic—a logic for hardware verification. In David Dill, editor, *Computer-Aided Verification, CAV '94*, volume 818 of *Lecture Notes in Computer Science*, pages 247–259, Stanford, CA, June 1994. Springer-Verlag.
6. D. Cyrluk, S. Rajan, N. Shankar, and M. K. Srivas. Effective theorem proving for hardware verification. In Kumar and Kropf [9], pages 203–222.
7. David Cyrluk. Microprocessor verification in PVS: A methodology and simple example. Technical Report SRI-CSL-93-12, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993.
8. Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 6, pages 243–320. Elsevier and MIT press, Amsterdam, The Netherlands, and Cambridge, MA, 1990.
9. Ramayya Kumar and Thomas Kropf, editors. *Theorem Provers in Circuit Design (TPCD '94)*, volume 910 of *Lecture Notes in Computer Science*, Bad Herrenalb, Germany, September 1994. Springer-Verlag.
10. Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.
11. Mandayam K. Srivas and Steven P. Miller. Formal verification of the AAMP5 microprocessor. In Michael G. Hinchey and Jonathan P. Bowen, editors, *Applications of Formal Methods*, Prentice Hall International Series in Computer Science, chapter 7, pages 125–180. Prentice Hall, Hemel Hempstead, UK, 1995.