# Verification of Pipelined Machines in ACL2*

Panagiotis Manolios

Department of Computer Sciences, University of Texas at Austin
pete@cs.utexas.edu
http://www.cs.utexas.edu/users/pete

**Abstract.** We describe the ACL2 techniques used in a new approach to the verification of pipelined machines. Our notion of correctness is based on WEBs (Well-founded Equivalence Bisimulations) [16, 18] and implies that the pipelined machine and the machine defined by the instruction set architecture have the same computations up to finite stuttering. We verify various variants of Sawada's simple machine [22, 21], including machines with exceptions, interrupts, non-determinism, and ALUs described in part at the netlist level. Our proofs contain no intermediate abstractions and are almost automatic, *e.g.*, the verification of the base machine does not require any user supplied theorems. To motivate the need for a new notion of correctness we show that the variant of the Burch and Dill notion of correctness [4] used by Sawada can be satisfied by incorrect machines.

## 1 Introduction

The specification used to prove a pipelined machine correct is an instruction set architecture (ISA). The ISA describes the interface between the hardware and software and contains the programmer visible components of the machine. A pipelined machine is correct if it satisfies a certain relationship with the ISA. There is no wide agreement on the "right" notion of correctness, but perhaps the most common approach is that of Burch and Dill [4]. One of the difficulties with specifying correctness is that we want to account for non-terminating behavior. If we were to restrict ourselves to terminating programs, we could say that a pipelined machine is correct if for any terminating program, both the pipelined machine and the ISA machine halt in the same final state. However, there are interesting non-terminating programs such as operating systems and transmission protocols that run on these machines and the traditional approach of stating correctness as a relationship between initial and final states cannot be used, as there is no final state. We are therefore forced to think about infinite computations.

We start with an example. Consider a simple ISA machine with instructions that are four-tuples consisting of an opcode, a target register, and two source registers. The state components of the ISA machine that are of interest are the program counter and the contents of registers ra and rb. The MA (micro

* Support for this work was provided by the SRC under contract 99-TJ-685.

architecture) machine is a pipelined machine with three stages.[1] A pipeline is analogous to an assembly line. The pipeline consists of several stages each of which performs part of the computation required to complete an instruction. When the pipeline is full many instructions are in various degrees of completion. A diagram of the MA machine appears in Fig. 1. The three stages are fetch, set-up, and write. During the fetch stage, the instruction pointed to by the PC (program counter) is retrieved from memory and placed into latch 1. During the set-up stage, the contents of the source registers (of the instruction in latch 1) are retrieved from the register file and sent to latch 2 along with the rest of the instruction in latch 1. During the write stage, the appropriate ALU (arithmetic logic unit) operation is performed and the result is used to update the value of the target register.

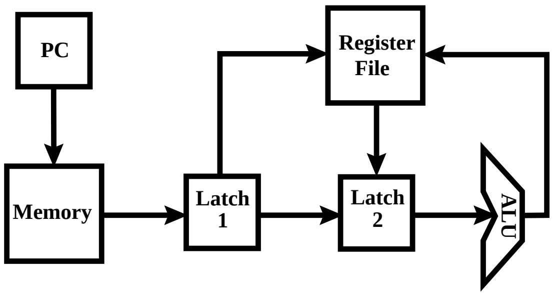


**Fig. 1.** A simple three-stage pipeline machine.

Suppose that the contents of memory are as follows.

| **Inst** | | | | |
|---|---|---|---|---|
| 0 | add | rb | ra | ra |
| 1 | add | ra | rb | ra |

When this simple two-line code fragment is executed on the ISA and MA machines, we get the following traces.

| **Clock** | **ISA** | **MA** | **Inst 0** | **Inst 1** |
|---|---|---|---|---|
| 0 | $\langle 0, \langle 1,1 \rangle\rangle$ | $\langle 0, \langle 01,01 \rangle\rangle$ | | |
| 1 | $\langle 1, \langle 1,2 \rangle\rangle$ | $\langle 1, \langle 01,01 \rangle\rangle$ | Fetch | |
| 2 | $\langle 2, \langle 3,2 \rangle\rangle$ | $\langle 2, \langle 01,01 \rangle\rangle$ | Set-up | Fetch |
| 3 | | $\langle 2, \langle 01,10 \rangle\rangle$ | Write | Stall |
| 4 | | $\langle \_, \langle 01,10 \rangle\rangle$ | | Set-up |
| 5 | | $\langle \_, \langle 11,10 \rangle\rangle$ | | Write |

[1] This machine is based on the machine described by Sawada [22, 21].

The rows correspond to steps of the machines, *e.g.*, row Clock 0 corresponds to the initial state, Clock 1 to the next state, and so on. The ISA and MA columns contain the relevant parts of the state of the machines: a pair consisting of the PC and the register file (itself a pair consisting of registers ra and rb). The contents of the register file of the ISA machine are numbers in decimal and the contents of the register file of the MA machine are bit-vectors (we show only the two low-order bits). The final two columns indicate what stage the instructions are in (only applicable to the MA machine).

In the initial state (in row Clock 0) the PCs of the ISA and MA machines contain the value 0 (indicating that the next instruction to execute is Inst 0) and both registers have the value 1. In the next ISA state (in row Clock 1), the PC is incremented and the add instruction performed, *i.e.*, register rb is updated with the value ra + ra = 2. The final entry in the ISA column contains the state of the ISA machine after executing Inst 1.

After one step of the MA machine, Inst 0 completes the fetch phase and the PC is incremented to point to the next instruction. After step 2 (in row Clock 2), Inst 0 completes the set-up stage, Inst 1 completes the fetch phase, and the PC is incremented. After step 3, Inst 0 completes the write-back phase and the register file is updated for the first time with rb set to 10 (2 in binary). However, Inst 1 is stalled during step 3 because one of its source registers is rb, the target register of the previous instruction. Since the previous instruction has not completed, the value of rb is not available and Inst 1 is stalled for one cycle. In the next cycle, Inst 1 enters the set-up stage and Inst 2 enters the fetch stage (not shown). Finally, after step 5, Inst 1 is completed and register ra is updated.

Comparing the partial traces of the ISA and MA machines and thinking about how to relate them makes it clear that we should stick to one representation of numbers. Below, the partial traces of the ISA and MA machines appear in the first two columns, with numbers represented in decimal.

$$\begin{array}{cccccc}
\textbf{ISA} & \textbf{MA} & & \textbf{MA} & & \textbf{MA} \\
\langle 0, \langle 1,1 \rangle\rangle & \langle 0, \langle 1,1 \rangle\rangle & & \langle 0, \langle 1,1 \rangle\rangle & & \langle 0, \langle 1,1 \rangle\rangle \\
\langle 1, \langle 1,2 \rangle\rangle & \langle 1, \langle 1,1 \rangle\rangle & \longrightarrow & \langle 0, \langle 1,1 \rangle\rangle & \longrightarrow & \langle 1, \langle 1,2 \rangle\rangle \\
\langle 2, \langle 3,2 \rangle\rangle & \langle 2, \langle 1,1 \rangle\rangle & & \langle 0, \langle 1,1 \rangle\rangle & & \langle 2, \langle 3,2 \rangle\rangle \\
 & \langle 2, \langle 1,2 \rangle\rangle & \text{Commit} & \langle 1, \langle 1,2 \rangle\rangle & \text{Remove} & \\
 & \langle _, \langle 1,2 \rangle\rangle & \text{PC} & \langle 1, \langle 1,2 \rangle\rangle & \text{Stutter} & \\
 & \langle _, \langle 3,2 \rangle\rangle & & \langle 2, \langle 3,2 \rangle\rangle & &
\end{array}$$

Notice that the PC differs in the two traces and this occurs because the pipeline, initially empty, is being filled and the PC points to the next instruction to fetch. If the PC were to point to the next instruction to commit (*i.e.*, the next instruction to complete), then we would get the trace shown in column 3. Notice that in column 3, the PC does not change from 0 to 1 until Inst 0 is committed in which case the next instruction to commit is Inst 1. We now have a trace that is the same as the ISA trace except for stuttering; after removing the stuttering we have, in column 4, the ISA trace.

To state correctness we use a *refinement map*, a function that maps MA states to ISA states. In the above example we mapped MA states to ISA states by transforming bit-vectors into decimal numbers and by transforming the PC. Proving correctness amounts to relating MA states with the ISA states they map to under the refinement map and proving a WEB (Well-founded Equivalence Bisimulation). Proving a WEB guarantees that MA states and related ISA states have related computations up to finite stuttering. This is a strong notion of equivalence, *e.g.*, a consequence is that the two machines satisfy the same $CTL^*\backslash X$ properties.[2] This includes the class of next-time free safety and liveness (including fairness) properties, *e.g.*, one such property is that the MA machine cannot deadlock (because the ISA machine cannot deadlock).

Why "up to finite stuttering"? Because we are comparing machines at different levels of abstraction: the pipelined machine is a low-level implementation of the high-level ISA specification. When comparing systems at different levels of abstraction, it is often the case that the low-level system requires several steps to match a single step of the high-level system.

Why use a refinement map? Because data can be represented in different ways, *e.g.*, the MA machine represents numbers in binary whereas the ISA machine uses a decimal representation. In addition, there may be components in one system that do not appear in the other, *e.g.*, the MA machine has latches but the ISA machine does not. Yet another reason is that components present in both systems may have different behaviors, as is the case with the PC above. Notice that the refinement map affects how MA and ISA states are related, not the behavior of the MA machine.

The theory of refinement maps and WEBs is presented in Section 2. In Section 3 we discuss supporting ACL2 books. In Section 4 we discuss the deterministic variants of Sawada's machine [21, 22][3]. The variants include machines with exceptions, fleshed-out ALUs, and combinations of these features. We also discuss the use of macros to automate the proof process. In Section 5 we discuss the non-deterministic variants and macros. Conclusions and related work appear in Section 6.

A more detailed explanation of how refinement maps and WEBs relate to the correctness of pipelined machines is presented in a companion paper [13]. Some key observations to keep in mind as you read the paper follow. Note that we prove the same theorem for each of the pipelined machine variants, including the variant with interrupts and exceptions. Other approaches [21] introduce new notions of correctness to deal with such features. Our characterization of correctness allows us to prove an MA machine correct with respect to an ISA machine by looking only at single steps of the machines. For the examples we consider, this leads to dramatically shorter proofs (as already mentioned, some proofs are automatic) and does not require intermediate abstractions. A practical and noteworthy feature of WEBs is their compositionality; this allows us to decompose the proof of the correctness of the complicated machines into man-

[2] $CTL^*$ is a braching-time temporal logic; $CTL^*\backslash X$ is $CTL^*$ without the next-time operator $X$.

[3] Sawada uses the machine to explain issues of correctness and proof techniques. It is a toy version of the FM9801, the final machine verified in Sawada's thesis.

ageable steps. All of the ACL2 definitions and theorems described in this paper are available from the author's Web page [14].

# 2 Well-founded Equivalence Bisimulation

## 2.1 Preliminaries

$\mathbb{N}$ denotes the natural numbers. $\langle Qx : r : b\rangle$ denotes a quantified expression, where $Q$ is the quantifier, $x$ is the bound variable, $r$ is the range of $x$ (true if omitted), and $b$ is the body. Function application is sometimes denoted by an infix dot "." and is left associative. Function composition is denoted by $\circ$. The disjoint union operator is denoted by $\uplus$. For a relation $R$, we abbreviate $\langle s, w\rangle \in R$ by $sRw$. A *well-founded structure* is a pair $\langle W, \prec\rangle$ where $W$ is a set and $\prec$ is a binary relation on $W$ such that there are no infinitely decreasing sequences on $W$ with respect to $\prec$. Spacing is used to reinforce binding: more space indicates lower binding.

**Definition 1** *(Transition System)*
A Transition System (TS) is a structure $\langle S, \dashrightarrow, L\rangle$, where $S$ is a non-empty set of states, $\dashrightarrow \;\subseteq S \times S$ is a left-total (every state has a successor) *transition relation*, and $L$ is a *labeling function* which maps each state to a label.

For TS $M$, state $s$ (of $M$), and temporal logic formula $f$, $M, s \models f$ denotes that $f$ holds at state $s$ of model $M$. A *path*, $\sigma$, is a sequence of states such that for adjacent states $s$ and $u$, $s \dashrightarrow u$. A path, $\sigma$, is a *fullpath* if it is infinite. $fp.\sigma.s$ denotes that $\sigma$ is a fullpath starting at $s$.

## 2.2 Well-Founded Equivalence Bisimulation

**Definition 2** *(Well-Founded Equivalence Bisimulation (WEB* [16, 18]*))*
$B$ is a well-founded equivalence bisimulation on TS $M = \langle S, \dashrightarrow, L\rangle$ iff:

1. $B$ is an equivalence relation on $S$; and
2. $\langle \forall s, w \in S : sBw : L.s = L.w\rangle$; and
3. There exists function $rank : S \times S \to W$, with $\langle W, \prec\rangle$ well-founded, and
$$\begin{array}{l} \langle \forall s, u, w \in S : sBw \;\;\wedge\;\; s \dashrightarrow u : \\ \qquad \langle \exists v : w \dashrightarrow v : uBv\rangle \;\;\vee \\ \qquad (uBw \;\;\wedge\;\; rank(u,u) \prec rank(s,s)) \;\;\vee \\ \qquad \langle \exists v : w \dashrightarrow v : sBv \;\;\wedge\;\; rank(u,v) \prec rank(u,w)\rangle\rangle \end{array}$$

We call a pair $\langle rank, \langle W, \prec\rangle\rangle$ satisfying condition 3 in the above definition, a *well-founded witness*. The third WEB condition guarantees that related states have the same computations up to stuttering. If states $s$ and $w$ are in the same class and $s$ can transit to $u$, then one of the following holds.

1. The transition can be matched with no stutter, in which case, $u$ is matched by a step from $w$.

2. The transition can be matched but there is stutter on the left (from $s$), in which case, $u$ and $w$ are in the same class and the rank function decreases (to guarantee that $w$ is forced to take a step eventually).
3. The transition can be matched but there is stutter on the right (from $w$), in which case, there is some successor $v$ of $w$ in the same class as $s$ and the rank function decreases (to guarantee that $u$ is eventually matched).

To prove a relation is a WEB, note that reasoning about single steps of $\dashrightarrow$ suffices. In addition we can often get by with a rank function of one argument. The example WEB in Fig. 2 demonstrates this, as the function *tag* has one argument.

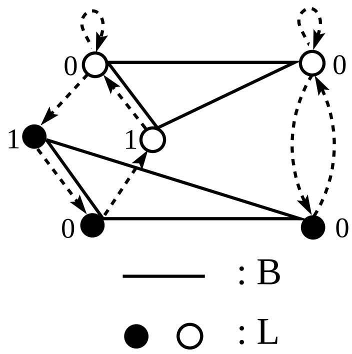


**Fig. 2.** The graph denotes a transition system, where circles denote states, the color of the circles denotes their label, and the transition relation is denoted by a dashed line. States related by the equivalence relation $B$ are joined by a solid line. To check that $B$ is a WEB, let $rank(u, v) = tag$ of $v$, and use the well-founded witness $\langle rank, \langle \mathbb{N}, < \rangle \rangle$.

**Theorem 1** (cf. [2, 18]) *If $B$ is a WEB on TS $M$ and $sBw$, then $s$ and $w$ have the same fullpaths up to stuttering and for any $CTL^* \backslash X$ formula $f$, $M, s \models f$ iff $M, w \models f$.*

A consequence of Theorem 1 is that states related by a WEB satisfy the same next-time free formulae of *LTL* (Linear Temporal Logic) and the same safety and progress properties (up to stuttering). This is a strong notion of equivalence; that we can use it profitably depends on the use of refinement maps.

### 2.3 Refinement and Composition

In this section, we define a notion of refinement and show that WEBs can be used in a compositional fashion. The compositionality of WEBs allows us to prove the correctness of pipelined machines in stages.

**Definition 3** *(Refinement)*
Let $M = \langle S, \dashrightarrow, L\rangle$, $M' = \langle S', \dashrightarrow', L'\rangle$, $r : S \to S'$. We say that $M$ is a *refinement* of $M'$ with respect to refinement map $r$ if there exists an equivalence relation, $B$, on $S \uplus S'$ (the disjoint union of $S$ and $S'$) such that $sB(r.s)$ for all $s \in S$ and $B$ is a WEB on the TS $\langle S \uplus S', \dashrightarrow \uplus \dashrightarrow', \mathcal{L}\rangle$, where $\mathcal{L}.s = L'(r.s)$ for $s$ an $S$ state and $\mathcal{L}.s = L'(s)$ otherwise.

The pipelined machine discussed in the introduction is a refinement of the ISA machine with respect to a refinement map that transforms bit-vectors into decimal numbers and transforms the PC as previously described.

**Theorem 2** *(Composition* ([15], cf. [16])*)*
*If $\langle S, \dashrightarrow, L\rangle$ is a refinement of $\langle S', \dashrightarrow', L'\rangle$ with respect to $r$, and $\langle S', \dashrightarrow', L'\rangle$ is a refinement of $\langle S'', \dashrightarrow'', L''\rangle$ with respect to $q$, then $\langle S, \dashrightarrow, L\rangle$ is a refinement of $\langle S'', \dashrightarrow'', L''\rangle$ with respect to $q \circ r$.*

We will use the above theorem to prove correctness in stages. An advantage of proving correctness in this way is that we can limit the difference between the machine descriptions from one stage to the next. This allows the proofs go through automatically because there is enough structural similarity between machines that ACL2 can decide equivalence forthwith. Yet another advantage is that changes to the lower-level machines can be localized.

## 3 Supporting Books

We use several general-purpose books to support automation. These include the standard `"top-with-meta"` and `"ihs"` books for reasoning about arithmetic as well as the books `"nth-thms"`, `"alist-thms"`, and `"defun-weak-sk"` for reasoning about `nth` and `update-nth`, alists, and quantification, respectively.

To prove correctness, we define interpreters for the ISA and MA machines and prove a WEB on the disjoint union of the machines, under some refinement map. Machine states are represented as lists and components of states are accessed and updated with `nth` and `update-nth`, respectively. The proof requires that we compare components of stepped states and much of this can be done automatically with rewrite rules that simplify and normalize `nth` and `update-nth` expressions. The book `"nth-thms"` contains the rewrite rules we found useful for this purpose and is based on the approach taken by Greve, Wilding, and Hardin on page 131 of reference [6]. We also use alists (*e.g.*, register files are represented as lists of register name, value pairs) and the book `"alist-thms"` contains some simple rules, similar to those in `"nth-thms"`, for reasoning about alists.

The book `"defun-weak-sk"` is used to reason about existential quantification. Recall that the macro `defun-sk` is used to implement quantification in ACL2 by introducing witness functions and constraints. For example, the quantified formula $\langle \exists x :: P(x,y)\rangle$ can be rendered in ACL2 as the function `EP` with the constraints `(P x y)` $\Rightarrow$ `(EP y)` and `(EP y)` $=$ `(P (W y) y)`. To see that this corresponds to quantification, notice that the first constraint gives us one direction of the argument: it says that if any value of `x` makes `(P x y)` true (*i.e.*, if $\langle \exists x :: P(x,y) \rangle$) then `(EP y)` is true. This constraint allows us to establish an existentially quantified formula by exhibiting a witness, but the constraint can be satisfied if `EP` always returns `t`. The second constraint gives us the other direction. It introduces the witness function `W` and requires that `(EP y)` is true iff `(P (W y) y)` is true. As a result, if `(EP y)` is true, then some value of `x` makes `(P x y)` true. As is mentioned in the ACL2 documentation [11], this idea was known to Hilbert. An ACL2 script corresponding to the above follows.[4] Notice that the constraints on `EP` are the constraints on `EP-witness` (which corresponds to our `W`).

```
(defstub P(x y) t)
(defun-sk EP (y)
  (exists (x) (P x y)))
:props EP
:props EP-witness
```

We wish to use quantification and encapsulation in the following way. We prove that a set of constrained functions satisfy a quantified formula. We then use functional instantiation [1, 12] to show that a set of functions satisfying these constraints also satisfy the (analogous) quantified formula. We want this proof obligation to be generated by macros but have found that the constraints generated by the quantified formulae complicate the design of such macros. The following observation has allowed us to simplify the process. The quantified formulae are established using witness functions, as is often the case. Therefore, only the first constraint generated by `defun-sk` is required for the proof. We defined the macro `defun-weak-sk` which generates only this constraint, *e.g.*, executing the ACL2 script

```
(defstub P(x y) t)
(defun-weak-sk E (y)
  (exists (x) (P x y)))
:props E
```

shows that the only constraint on `E` is `(P x y)` $\Rightarrow$ `(E y)`. By functional instantiation, any theorem proved about `E` also holds when `E` is replaced by `EP` (since `EP` satisfies the constraint on `E`). We use `defun-weak-sk` in our scripts and at the very end we prove the `defun-sk` versions of the main results by functional instantiation (a step taken to make the presentation of the final result independent of our macros).

## 4 Deterministic Machines

### 4.1 `ISA` Definition

The first machine we define is `ISA`. The main function is `ISA-step`, a function that steps the `ISA` machine, *i.e.*, it takes an `ISA` state and returns the next

[4] `Props` shows all of the properties in the ACL2 world that are associated with a symbol.

ISA state. For all the machines, an instruction is a five-tuple consisting of the symbol Inst, an opcode, a target register, and two source registers. We have the following definitions in the book "Inst".

```
(defun Inst (opcode rc ra rb) (list 'Inst opcode rc ra rb))

(defmacro Inst-opcode () 1)
(defmacro Inst-rc () 2)
(defmacro Inst-ra () 3)
(defmacro Inst-rb () 4)
```

An ISA state is a four-tuple as shown below. The following definitions are in the book "ISA".

```
(defun ISA-state (pc regs mem)
  (list 'ISA pc regs mem))

(defun ISA-p (x)
  (equal (car x) 'ISA))

(defmacro ISA-pc () 1)
(defmacro ISA-regs () 2)
(defmacro ISA-mem () 3)
```

The definition of the step function, ISA-step, of the ISA machine is outlined below. The instruction referenced by pc (the program counter) is fetched from memory. Based on the opcode, the appropriate instruction is performed. The code for subtraction is not shown since it is similar to the code for addition. Recall that regs, the register file, is an alist. The functions value-of and update-valuation are defined in the "alist-thms" book and are used to access and update alists.

```
(defun add-rc (ra rb rc regs)
  (update-valuation rc
                    (+ (value-of ra regs)
                       (value-of rb regs))
                    regs))

(defun ISA-add (rc ra rb ISA)
  (ISA-state (1+ (nth (ISA-pc) ISA))
             (add-rc ra rb rc (nth (ISA-regs) ISA))
             (nth (ISA-mem) ISA)))
...

(defun ISA-default (ISA)
  (ISA-state (1+ (nth (ISA-pc) ISA))
             (nth (ISA-regs) ISA)
             (nth (ISA-mem) ISA)))
```

```
(defun ISA-step (ISA)
  (let ((inst (value-of (nth (ISA-pc) ISA) (nth (ISA-mem) ISA))))
    (let ((op (nth (Inst-opcode) inst))
          (rc (nth (Inst-rc) inst))
          (ra (nth (Inst-ra) inst))
          (rb (nth (Inst-rb) inst)))
      (cond ((equal op 0) ; add
             (ISA-add rc ra rb ISA))
            ((equal op 1) ; sub
             (ISA-sub rc ra rb ISA))
            (t (ISA-default ISA))))))
```

### 4.2 MA Definition

ISA is the specification for MA, a three stage pipelined machine. It corresponds to the simple machine described by Sawada [21, 22] and is the simplest pipelined machine we consider. An MA state is a six-tuple consisting of the symbol MA, a program counter, a register file, a memory, and two latches. The first latch is a six-tuple consisting of the symbol latch1, a flag which indicates if the latch is valid, an opcode, the target register, and two source registers. The second latch is a six-tuple consisting of the symbol latch2, a flag as before, an opcode, the target register, and the values of the two source registers. The definition of MA-step follows. The definitions in this section are in the book "MA".

```
(defun MA-step (MA)
  (MA-state (step-pc MA)
            (step-regs MA)
            (nth (MA-mem) MA)
            (step-latch1 MA)
            (step-latch2 MA)))
```

Step-pc is defined below. A stall occurs if both latches are valid and the target register of the instruction in latch2 is one of the source registers.[5]

```
(defun step-pc (MA)
  (if (stall-condp MA) (nth (MA-pc) MA) (1+ (nth (MA-pc) MA))))
(defun stall-condp (MA)
  (let ((latch1 (nth (MA-latch1) MA))
        (latch2 (nth (MA-latch2) MA)))
    (and (nth (latch2-validp) latch2)
         (nth (latch1-validp) latch1)
         (bor (equal (nth (latch1-ra) latch1)
                     (nth (latch2-rc) latch2))
              (equal (nth (latch1-rb) latch1)
                     (nth (latch2-rc) latch2))))))
```

[5] (Bor x y) translates to (if x 't y). We use this macro because (or x y) translates to (if x x y) and two occurrences of x can slow down the theorem prover.

To step the register file, we check that latch2 is valid and that an arithmetic operation is to be performed. If so, we update the target register with the result obtained from the ALU.

```
(defun ALU-output (op val1 val2)
  (if (equal op 0)
      (+ val1 val2)
    (- val1 val2)))
(defun step-regs (MA)
  (let ((latch2 (nth (MA-latch2) MA)))
    (if (and (nth (latch2-validp) latch2)
             (bor (equal (nth (latch2-op) latch2) 0)
                  (equal (nth (latch2-op) latch2) 1)))
        (update-valuation (nth (latch2-rc) latch2)
                          (ALU-output (nth (latch2-op) latch2)
                                      (nth (latch2-ra-val) latch2)
                                      (nth (latch2-rb-val) latch2))
                          (nth (MA-regs) MA))
      (nth (MA-regs) MA))))
```

If there is no stall, we step latch 1 by fetching the instruction in memory pointed to by the pc.

```
(defun step-latch1 (MA)
  (let ((latch1 (nth (MA-latch1) MA))
        (inst (value-of (nth (MA-pc) MA) (nth (MA-mem) MA))))
    (cond ((stall-condp MA)
           latch1)
          (t (latch1 t
                     (nth (Inst-opcode) inst)
                     (nth (Inst-rc) inst)
                     (nth (Inst-ra) inst)
                     (nth (Inst-rb) inst))))))
```

If latch 1 is valid and there is no stall, latch 1 and the contents of the register file are used to step latch 2.

```
(defun step-latch2 (MA)
  (let ((latch1 (nth (MA-latch1) MA)))
    (if (nth (latch1-validp) latch1)
        (latch2 (not (stall-condp MA))
                (nth (latch1-op) latch1)
                (nth (latch1-rc) latch1)
                (value-of (nth (latch1-ra) latch1)
                          (nth (MA-regs) MA))
                (value-of (nth (latch1-rb) latch1)
                          (nth (MA-regs) MA)))
      (update-nth (latch2-validp) nil (nth (MA-latch2) MA)))))
```

### 4.3 Refinement Map Definitions

To prove that MA is a correct implementation of ISA, we prove a WEB on the (disjoint) union of the machines, with MA states labeled by the appropriate refinement map. Once the required notions are defined, the macros implementing our proof methodology can be used to prove correctness without any user supplied theorems. In this section, we present the definitions which make the statement of correctness precise. The definitions in this section are in the book "MA-ISA".

The following function is a recognizer for "good" MA states.

```
(defun good-MA (MA)
  (and (rationalp (nth (MA-pc) MA))
       (let ((latch1 (nth (MA-latch1) MA))
             (latch2 (nth (MA-latch2) MA))
             (NMA (committed-MA MA)))
         (case (shift-pc latch1 latch2)
           (0 t)
           (1 (MA-= (MA-step NMA) MA))
           (otherwise (MA-= (MA-step (MA-step NMA)) MA))))))
```

First, we require that pc, the program counter, is a rational. Such type restrictions are common. Next, we require that MA states are reachable from flushed states (states with invalid latches). The reason for this restriction is that otherwise MA states can be inconsistent (unreachable), *e.g.*, consider an MA state whose first latch contains an add instruction, but where there are no add instructions in memory. We check for this by stepping the *committed state*, the state obtained by invalidating all partially completed instructions and altering the program counter so that it points to the next instruction to commit.

```
(defun committed-MA (MA)
  (let ((pc (nth (MA-pc) MA))
        (regs (nth (MA-regs) MA))
        (mem (nth (MA-mem) MA))
        (latch1 (nth (MA-latch1) MA))
        (latch2 (nth (MA-latch2) MA)))
    (MA-state
     (- pc (shift-pc latch1 latch2))
     regs
     mem
     (update-nth (latch1-validp) nil latch1)
     (update-nth (latch2-validp) nil latch2))))
```

The program counter is decremented depending on the number of valid latches.

```
(defun b-to-num (x)
  (if x 1 0))
(defun shift-pc (latch1 latch2)
  (+ (b-to-num (nth (latch1-validp) latch1))
     (b-to-num (nth (latch2-validp) latch2))))
```

Finally, we note that `MA-=` relates two `MA` states if they have the same pc, regs, mem, and if their latches match, *i.e.*, they have the same latch 1 or both states have an invalid latch 1 and similarly with latch 2. Note that `committed-MA` performs a kind of reverse flushing: instead of stepping the machine in an attempt to complete pending instructions and reach a flushed state, `committed-MA` invalidates pending instructions and adjusts the program counter accordingly. To make sure that a state is not inconsistent, we check that it is reachable from the corresponding committed state. Now that we have made precise what the `MA` states are, the refinement map is:

```
(defun MA-to-ISA (MA)
  (let ((MA (committed-MA MA)))
    (ISA-state (nth (MA-pc) MA)
               (nth (MA-regs) MA)
               (nth (MA-mem) MA))))
```

The final definition required is that of the well-founded witness. The function `MA-rank` serves this purpose by computing how long it will take an `MA` state to commit an instruction. An `MA` state will commit an instruction in the next step if its second latch is valid. Otherwise, if its first latch is valid it will be ready to commit an instruction in one step. Otherwise, both latches are invalid and it will be ready to commit an instruction in two steps.

```
(defun MA-rank (MA)
  (let ((latch1 (nth (MA-latch1) MA))
        (latch2 (nth (MA-latch2) MA)))
    (cond ((nth (latch2-validp) latch2) 0)
          ((nth (latch1-validp) latch1) 1)
          (t 2))))
```

### 4.4 Proof of Correctness

To complete the proof it seems we have to: define the machine corresponding to the disjoint union of `ISA` and `MA`, define a WEB that relates a (good) `MA` state `s` to `(MA-to-ISA s)`, define the well-founded witness, and prove that indeed the purported WEB really is a WEB. We have implemented macros which automate this. The macros are useful not only for this example, but also for the verification of the rest of the deterministic machines we present in this paper and can be used to show a WEB between other types of deterministic systems. (Non-deterministic versions are described in Section 5.) The proof of correctness is completed with the following three macro calls (in the book `"MA-ISA"`).

```
(generate-full-system isa-step isa-p ma-step ma-p
                      ma-to-isa good-ma ma-rank)
(prove-web isa-step isa-p ma-step ma-p ma-to-isa ma-rank)
(wrap-it-up isa-step isa-p ma-step ma-p
            good-ma ma-to-isa ma-rank)
```

The first macro, `generate-full-system`, generates the definition of `B`, the purported WEB as well as `R`, the transition relation of the disjoint union of the `ISA` and `MA` machines. The macro translates to the following. (Some declarations and forward-chaining theorems used to control the theorem prover have been elided.)

```
(progn
  (defun wf-rel (x y) ...
    (and (ISA-p x)
         (MA-p y)
         (good-MA y)
         (equal x (MA-to-ISA y))))
  (defun B (x y) ...
    (bor (wf-rel x y)
         (wf-rel y x)
         (equal x y)
         (and (MA-p x)
              (MA-p y)
              (good-MA x)
              (good-MA y)
              (equal (MA-to-ISA x) (MA-to-ISA y)))))
  (defun rank (x)  ...
    (if (MA-p x) (MA-rank x) 0))
  (defun R (x y) ...
    (cond ((ISA-p x) (equal y (ISA-step x)))
          (t (equal y (MA-step x)))))
  ...)
```

What is left is to prove that `B`—the reflexive, symmetric, transitive closure of `wf-rel`—is a WEB with well-founded witness `rank`. We do this in two steps. First, the macro `prove-web` is used to prove the "core" theorem (as well as some "type" theorems not shown).

```
(defthm B-is-a-wf-bisim-core
  (let ((u (ISA-step s))
        (v (MA-step w)))
    (implies (and (wf-rel s w)
                  (not (wf-rel u v)))
             (and (wf-rel s v)
                  (e0-ord-< (MA-rank v) (MA-rank w))))))
```

Comparing `B-is-a-wf-bisim-core` with the definition of WEBs, we see that `B-is-a-wf-bisim-core` does not contain quantifiers and it mentions neither `B` nor `R`. This is on purpose as we use "domain-specific" information to construct a simplified theorem that is used to establish the main theorem. To that end we removed the quantifiers and much of the case analysis. For example, in the definition of WEBs, $u$ ranges over successors of $s$ and $v$ is existentially quantified over successors of $w$, but because we are dealing with deterministic systems, `u` and `v` are defined to be *the* successors of `s` and `w`, respectively. Also, `wf-rel` is not an equivalence relation as it is not reflexive, symmetric, or transitive. Finally, we ignore the second disjunct in the third condition of the definition of WEBs because `ISA` does not stutter. The justification for calling this the "core" theorem is that we have proved in the book `"det-encap-wfbisim"` that a constrained system which satisfies a theorem analogous to `B-is-a-wf-bisim-core` (and some "type" theorems) also satisfies a WEB. Using functional instantiation we can now prove `MA` correct. The use of this domain-specific information makes a big difference, *e.g.*, when we tried to prove the theorem obtained by a naive translation of the WEB definition (sans quantifiers), ACL2 ran out of memory after 30 hours, yet the above theorem is now proved in about 11 seconds.

The final macro call generates the events used to finish the proof. We present the generated events germane to this discussion below. The first step is to show that `B` is an equivalence relation. This theorem is proved by functional instantiation of a theorem in the book `"det-encap-wfbisim"`.

```
(defequiv B
  :hints (("goal" :by (:functional-instance
                       encap-B-is-an-equivalence ...))))
```

The second WEB condition, that related states have the same label, is taken care of by the refinement map. We show that `rank` is a well-founded witness.

```
(defthm rank-well-founded
  (e0-ordinalp (rank x)))
```

We use functional instantiation and `B-is-a-wf-bisim-core` as described above to prove the following.

```
(defun-weak-sk exists-w-succ-for-u-weak (w u)
  (exists (v) (and (R w v) (B u v))))
(defun-weak-sk exists-w-succ-for-s-weak (w s)
  (exists (v)
    (and (R w v)
         (B s v)
         (e0-ord-< (rank v) (rank w)))))
(defthm
 B-is-a-wf-bisim-weak
 (implies (and (B s w)
               (R s u))
          (or (exists-w-succ-for-u-weak w u)
              (and (B u w)
                   (e0-ord-< (rank u) (rank s)))
              (exists-w-succ-for-s-weak w s)))
 :hints
 (("goal" :by (:functional-instance b-is-a-wf-bisim-sk ...)))
 :rule-classes nil)
```

We use `defun-weak-sk` for these definitions and for the proofs in the book `"det-encap-wfbisim"` for the reasons outlined in Section 3. To make it easier for the general ACL2 community to understand the results, we state them in terms of the built-in macro `defun-sk`. The proof is a trivial functional instantiation, since the single constraint generated by `defun-weak-sk` is one of the constraints generated by `defun-sk`.

```
(defun-sk exists-w-succ-for-u (w u)
  (exists (v) (and (R w v) (B u v))))
(defun-sk exists-w-succ-for-s (w s)
  (exists (v)
    (and (R w v)
         (B s v)
         (e0-ord-< (rank v) (rank w)))))
...
(defthm
  B-is-a-wf-bisim
  (implies (and (B s w)
                (R s u))
           (or (exists-w-succ-for-u w u)
               (and (B u w)
                    (e0-ord-< (rank u) (rank s)))
               (exists-w-succ-for-s w s)))
  :hints
  (("goal"
     :by (:functional-instance B-is-a-wf-bisim-weak
           (exists-w-succ-for-u-weak exists-w-succ-for-u)
           (exists-w-succ-for-s-weak exists-w-succ-for-s))))
  :rule-classes nil))
```

### 4.5 Comparison With Original Proof

The proof given by Sawada [22] uses a variant of the Burch and Dill notion of correctness. The main theorem proved is that if the pipelined machine starts in $MA_0$, a flushed state, takes $n$ steps to arrive at state $MA_n$, also a flushed state, then there is some number $m$ such that stepping the projection of $MA_0$ $m$ steps results in the projection of $MA_n$. The projection of an `MA` state is the `ISA` state obtained from the program counter, register file, and memory of the `MA` state. Since this notion of correctness requires a pipelined machine that can be flushed, the machine defined by Sawada has an extra input signal which can be used to flush the machine. We have no way (and no need) to flush `MA`.

Sawada also proves the "liveness" theorem that any pipelined state can be flushed. These two theorems constitute his notion of correctness. We ask the informal question, "Are there any pipelined machines that are obviously incorrect but satisfy this notion of correctness?" If you consider deadlock an abhorrent behavior, the answer if yes. Using Sawada's proof scripts, we provide a mechanically checked proof that the trivial pipelined machine with a next-state function which invalidates the latches and keeps the PC, memory, and register file intact satisfies this notion of correctness.[6] The proof is straightforward. The first theorem is established by choosing $m$ to be 0. The second theorem holds because the next state function invalidates the latches; therefore, the next state of any state is flushed.

**Flushing Proof of `MA`** We can use flushing to prove that `MA` is a refinement of `ISA` as follows. We define a function that flushes an `MA` state and use this function to show that `MA` is a refinement of `ISA`. Notice that, in contrast to the proof by Sawada, there is no trivial pipelined machine that will satisfy this notion of correctness. This is because proving a WEB between a pipelined machine and `ISA` implies that any `ISA` behavior can be matched by the pipelined machine; since `ISA` has non-trivial behaviors so does the pipelined machine. Even so, this proof is not satisfactory because the relationship between `MA` states and `ISA` states is not clear, *e.g.*, inconsistent `MA` states are related to `ISA` states. A full discussion appears in the companion paper [13].

**Theorem Proving Effort** Finally, we compare the complexity of the proofs. We consider only the books required for the proof; this includes neither the books used to define the machines nor supporting general-purpose books. We consider the books containing our macros to be part of the supporting books because they were designed for general-purpose use and have been used with the dozen or so proofs described in this paper. Since we proved the correctness of `MA` without the use of any intermediate abstractions, invariants, or user-supplied theorems, the size of the book containing the definitions required for the proof is about 3K; the size of the files containing Sawada's proof is about 94K. The time required for our proof (including the loading of related books) is about 30 seconds on a 600MHz Pentium III; the time required for Sawada's proof is about 460 seconds (on the same machine).

### 4.6 The Remaining Deterministic Machines

The remaining deterministic machines are named `ISA128`, `MA128`, `MA128serial`, and `MA128net`. `ISA128` is an ISA-level machine that is the specification for the remaining pipelined machines. `ISA128` differs from `ISA` as follows. It has a multiply instruction but no subtract instruction and the arithmetic operations are mod $2^{128}$. It deals with exceptions as follows: an `ISA128` state has an extra field containing an exception flag. If an overflow occurs, the exception flag is checked;

[6] More insidious machines also satisfy this notion of correctness (but we do not provide mechanical proofs). Examples include machines that sometimes deadlock (in a flushed state) and machines that spend some of their time performing computations other than the ones expected.

if it is off, arithmetic mod $2^{128}$ is performed; if the exception flag is on, an exception handler is called. The exception handler is a constrained function of the program counter, register file, and memory that returns a new program counter, register file, memory, and exception flag.

`MA128` is a three-stage pipelined machine similar to `MA` but designed to accommodate the `ISA128` specification. Overflows are dealt with as follows: if an overflow occurs during an arithmetic operation, then the partially executed instructions are invalidated[7] and the exception handler is called. The resulting state is constrained to be flushed (*i.e.*, both latches are invalid). The proof that `MA128` is a refinement of `ISA128` is very similar to proof that `MA` is a refinement of `ISA`. The same function names are used and the functions `good-MA` and `MA-rank` are exactly the same as before while the functions `committed-MA`, `MA-=`, and `MA-to-ISA` differ only in their treatment of the exception flag. As before, no user supplied theorems are required.

`MA128serial` is similar to `MA128`, except that the ALU is defined in terms of a serial adder and a multiplier based on the adder. The adder, multiplier, and proof of their correctness are taken from Kaufmann, Manolios, and Moore [10]. We prove that `MA128serial` is a refinement of `MA128` and with Theorem 2, we conclude that it is also a refinement of `ISA128`. The refinement map used maps the bit-vectors in the register file and the second latch to numbers because the ALU of `MA128serial` operates on bit-vectors, whereas `MA128` operates on integers. For the proof to go through, we need four theorems, but they are general results about `nfix` and the relationship between `convert-regs` (used to convert bit-vectors to numbers in the register file) and `value-of` and `update-valuation` (the functions we use to access and update alists).

`MA128net` is a version of `MA128` with an ALU defined in terms of a 128-bit adder described in a netlist language (*i.e.*, described in terms of Boolean functions). We have a function that generates an adder of any size and we prove that the adder generated is correct by relating it to the serial adder. We prove that `MA128net` is a refinement of `MA128serial`, hence by composition, a refinement of `ISA128`.

## 5 Non-Deterministic Machines

We have defined non-deterministic analogues of the deterministic machines presented in the previous section. The non-determinism arises because the next state of the machines is a function of the current state and an interrupt signal, which is free. Recall that the macros used in the previous section were designed for deterministic machines, therefore, we have analogous macros for the non-deterministic case. The differences between the macros arise from the differences in the next state function mentioned above and in the generation of the transition relation `R`, which, due to the non-determinism, requires quantification. For example, the

[7] The function `committed-MA` in the book `"MA128"` is used to perform the invalidation. It is interesting that this function, originally conceived for verification purposes, is used to define a more complicated machine.

following is generated by the macro `generate-full-system` when applied to the non-deterministic analogues of the `ISA` and `MA` machines of the previous section.

```
(defun R-int (x y int) ...
  (cond ((ISA-p x) (equal y (ISA-step x int)))
        (t (equal y (MA-step x int)))))
(defun-sk R (x y)
  (exists (int) (R-int x y int)))
```

The non-deterministic machines differ from the deterministic machines as follows. Their state contains an interrupt register which is used to record interrupts. Their next state is obtained by first checking the interrupt register. If non-empty, the interrupt handler is called (in MA-level machines, partially executed instructions are aborted before the call). The interrupt handler is a constrained function of the register file, memory, and interrupt register and returns a state with the same program counter and register file, an empty interrupt register, and a new (possibly modified) memory. If the interrupt register is empty, we check if an interrupt has been raised. If so, the interrupt type is recorded in the interrupt register (and with MA-level machines, partially executed instructions are aborted). If not, we proceed as in the deterministic case.

Non-determinism has led to new notions of correctness in the literature. For example, to deal with interrupts, a notion of correctness (still based on the Burch and Dill notion, but different from the one used for deterministic machines) is presented by Sawada [21]: if $M_0$ is a flushed state and if taking $n$ steps where the interrupts at each step are specified by the list $l$ results in a flushed state $M_n$, then there is a number $n'$ and a list $l'$ such that stepping the projection of $M_0$ $n'$ steps with interrupt list $l'$ results in the projection of $M_n$. Notice that a machine which always ignores interrupts satisfies this specification and is therefore considered correct.

In contrast, since WEBs apply to non-deterministic systems, our notion of correctness in the presence of interrupts remains the same, *i.e.*, we prove the pipelined machine is a refinement of its ISA specification. As a consequence, a pipelined machine which ignores interrupts cannot be proven correct. Another advantage is that our proof obligation is still about single steps of the machines, as opposed to finite behaviors. As before, this makes the proof much simpler as no intermediate abstractions are required.

## 6 Conclusions

Some of the early work on pipelined machine verification was based on skewed abstraction functions [25, 5, 26]. The Burch and Dill notion of correctness, based on flushing and commuting diagrams, was introduced later [4]. Theorem-proving approaches include the work by Sawada and Hunt [23, 24, 21, 22]. They use an intermediate abstraction called MAETT to verify some very complicated machines. There are other theorem proving approaches as well [9, 8, 27]. Model-checking approaches include the use of symmetry reductions and compositional model-checking [17] and the use of assume-guarantee reasoning [7]. In addition, decision procedures for Boolean logic with equality and uninterpreted function symbols [3, 19] have been used to verify pipelined machines [3].

Our notion of correctness is motivated by the problems with the ubiquitous Burch and Dill notion of correctness. We show with mechanical proof, that as used by Sawada [21, 22], the Burch and Dill notion can be satisfied by pipelined machines which deadlock. In contrast, our notion is based on WEBs and only pipelined machines which have the same behaviors (under a refinement map) as the instruction set architecture specification satisfy this notion of correctness. We verified various extensions of Sawada's simple pipelined machine [21, 22]. Our extensions include exception handling, interrupts, and ALUs described in part at the netlist level; all of the proofs are available from the author's Web page [14]. Our proofs are very simple and in many cases automatic. The reasons for the simplicity include:

1. We prove a theorem about a constrained system that is then invoked with functional instantiation. This allows us to bypass much of the case analysis and the reasoning about quantifiers that would otherwise be required.
2. We implemented macros that generate the disjoint union of the ISA and MA machines and that generate proof obligations which take advantage of the analysis mentioned immediately above.
3. Our notion of correctness can be proved by reasoning about single steps of the machines (as opposed to reasoning about an arbitrary number of steps). This helps us avoid the use of intermediate abstractions.
4. We used the compositionality of WEBs to decompose proofs.
5. We used functional instantiation to deal with exceptions and interrupts. As a result, our proofs apply for any specific exception and interrupt handlers.

We have also exhibited a clear, compositional path from the verification of term-level descriptions of pipelined machines to the verification of low-level descriptions (*e.g.*, netlist descriptions). For example, in the proof that `MA128` is a refinement of `ISA128`, the definition of the ALU is disabled; the result is that ACL2 treats the ALU as an uninterpreted function. However, to verify that `MA128net` is a refinement of `MA128`, we enable the definition of the ALU and prove theorems relating a netlist description of the circuit to a serial adder which is then related to addition on integers. This allows us to relate term-level descriptions of machines to lower-level descriptions in a compositional way, *e.g.*, it would not be too much work to use the floating-point multiplier of Russinoff and Flatau [20].

For future work, we plan to look at more complicated machines, namely machines with deeper pipelines, out-of-order execution, and richer instruction sets.

## Acknowledgements

J Moore provided guidance, inspiration, and a proof of the equivalence between the serial adder and the netlist adder.

## References


[1] R. S. Boyer, D. Goldschlag, M. Kaufmann, and J. S. Moore. Functional instantiation in first order logic. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 7–26. Academic Press, 1991.

[2] M. Browne, E. Clarke, and O. Grumberg. Characterizing finite Kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59, 1988.

[3] R. E. Bryant, S. German, and M. N. Velev. Exploiting positive equality in a logic of equality with uninterpreted functions. In N. Halbwachs and D. Peled, editors, *Computer-Aided Verification–CAV '99*, volume 1633 of *LNCS*, pages 470–482. Springer-Verlag, 1999.

[4] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *Computer-Aided Verification (CAV '94)*, volume 818 of *LNCS*, pages 68–80. Springer-Verlag, 1994.

[5] D. Cyrluk. Microprocessor verification in PVS: A methodology and simple example. Technical Report SRI-CSL-93-12, SRI, Dec. 1993.

[6] D. Greve, M. Wilding, and D. Hardin. High-speed, analyzable simulators. In M. Kaufmann, P. Manolios, and J. S. Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, pages 113–135. Kluwer Academic Press, June 2000.

[7] T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Assume-guarantee refinement between different time scales. In N. Halbwachs and D. Peled, editors, *Computer-Aided Verification–CAV '99*, volume 1633 of *LNCS*, pages 208–221. Springer-Verlag, 1999.

[8] R. Hosabettu, G. Gopalakrishnan, and M. Srivas. A proof of correctness of a processor implementing Tomasulo's algorithm without a reorder buffer. In L. Pierre and T. Kropf, editors, *Correct Hardware Design and Verification Methods, 10th IFIP WG10.5 Advanced Research Working Conference, (CHARME '99)*, volume 1703 of *LNCS*, pages 8–22. Springer-Verlag, 1999.

[9] R. Hosabettu, M. Srivas, and G. Gopalakrishnan. Proof of correctness of a processor with reorder buffer using the completion functions approach. In N. Halbwachs and D. Peled, editors, *Computer-Aided Verification–CAV '99*, volume 1633 of *LNCS*. Springer-Verlag, 1999.

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

[11] M. Kaufmann and J. S. Moore. ACL2 homepage. See URL `http://www.cs.-utexas.edu/users/moore/acl2`.

[12] M. Kaufmann and J. S. Moore. Structured theory development for a mechanized logic. *Journal of Automated Reasoning*, 2000. To appear, See URL `http://www.cs.utexas.edu/users/moore/publications/acl2-papers.-html#Foundations`.

[13] P. Manolios. Correctness of pipelined machines. In W. A. Hunt, Jr. and S. D. Johnson, editors, *Formal Methods in Computer-Aided Design–FMCAD 2000*, LNCS. Springer-Verlag, 2000.

[14] P. Manolios. Homepage of Panagiotis Manolios, 2000. See URL `http://www.cs.-utexas.edu/users/pete`.

[15] P. Manolios. Well-founded equivalence bisimulation. Technical report, Department of Computer Sciences, University of Texas at Austin, 2000. In preparation.

[16] P. Manolios, K. Namjoshi, and R. Sumners. Linking theorem proving and model-checking with well-founded bisimulation. In N. Halbwachs and D. Peled, editors,

*Computer-Aided Verification–CAV '99*, volume 1633 of *LNCS*, pages 369–379. Springer-Verlag, 1999.

[17] K. L. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In A. J. Hu and M. Y. Vardi, editors, *Computer Aided Verification (CAV '98)*, volume 1427 of *LNCS*, pages 110–121. Springer-Verlag, 1998.

[18] K. S. Namjoshi. A simple characterization of stuttering bisimulation. In *17th Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 1346 of *LNCS*, pages 284–296, 1997.

[19] A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel. Deciding equality formulas by small domain instantiations. In N. Halbwachs and D. Peled, editors, *Computer-Aided Verification–CAV '99*, volume 1633 of *LNCS*, pages 455–469. Springer-Verlag, 1999.

[20] D. M. Russinoff and A. Flatau. Rtl verification: A floating-point multiplier. In M. Kaufmann, P. Manolios, and J. S. Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, pages 201–231. Kluwer Academic Press, June 2000.

[21] J. Sawada. *Formal Verification of an Advanced Pipelined Machine*. PhD thesis, University of Texas at Austin, Dec. 1999. See URL `http://www.cs.utexas.edu/-users/sawada/dissertation/`.

[22] J. Sawada. Verification of a simple pipelined machine model. In M. Kaufmann, P. Manolios, and J. S. Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, pages 137–150. Kluwer Academic Press, 2000.

[23] J. Sawada and W. A. Hunt, Jr. Trace table based approach for pipelined microprocessor verification. In *Computer Aided Verification (CAV '97)*, volume 1254 of *LNCS*, pages 364–375. Springer-Verlag, 1997.

[24] J. Sawada and W. A. Hunt, Jr. Processor verification with precise exceptions and speculative execution. In A. J. Hu and M. Y. Vardi, editors, *Computer Aided Verification (CAV '98)*, volume 1427 of *LNCS*, pages 135–146. Springer-Verlag, 1998.

[25] M. Srivas and M. Bickford. Formal verification of a pipelined microprocessor. *IEEE Software*, pages 52–64, Sept. 1990.

[26] M. K. Srivas and S. P. Miller. Formal verification of an avionics microprocessor. Technical Report CSL-95-04, SRI International, 1995.

[27] P. J. Windley and M. L. Coe. A correctness model for pipelined microprocessors. In *Theorem Provers in Circuit Design*, volume 901 of *LNCS*, pages 33–52. Springer-Verlag, 1994.