Correctness of Pipelined Machines *

Panagiotis Manolios

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

Abstract. The correctness of pipelined machines is a subject that has been studied extensively. Most of the recent work has used variants of the Burch and Dill notion of correctness [4]. As new features are modeled, *e.g.*, interrupts, new notions of correctness are developed. Given the plethora of correctness conditions, the question arises: what is a reasonable notion of correctness? We discuss the issue at length and show, by mechanical proof, that variants of the Burch and Dill notion of correctness are flawed. We propose a notion of correctness based on WEBs (Well-founded Equivalence Bisimulations) [16, 19]. Briefly, our notion of correctness implies that the ISA (Instruction Set Architecture) and MA (Micro-Architecture) machines have the same observable infinite paths, up to stuttering. This implies that the two machines satisfy the same $CTL^*\backslash X$ properties and the same safety and liveness properties (up to stuttering).

To test the utility of the idea, we use ACL2 to verify several variants of the simple pipelined machine described by Sawada in [22, 23]. Our variants extend the basic machine by adding exceptions (to deal with overflows), interrupts, and fleshed-out 128-bit ALUs (one of which is described in a netlist language). In all cases, we prove the same final theorem. We develop a methodology with mechanical support that we used to verify Sawada's machine. The resulting proof is substantially shorter than the original and does not require any intermediate abstractions; in fact, given the definitions and some general-purpose books (collections of theorems), the proof is automatic. A practical and noteworthy feature of WEBs is their compositionality. This allows us to prove the correctness of the more elaborate machines in manageable stages.

1 Introduction

The complexity of computing systems has made mechanical verification the preferred way to reason about such systems. In reasoning about computing systems, we start by modeling the system and stating properties of interest. This is inherently an informal process; therefore, care must be taken to get it right. Computing systems vary as do approaches to modeling them, but notions of correctness often do not. For example, there are many different sorting algorithms, but there is one notion of correctness, *viz.*, that the output is an ordered permutation of

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

the input. This correctness criterion can be thought of as a constraint satisfied only by sorting algorithms. If we give you an algorithm and a proof that it satisfies this constraint, then you do not have to look at the internals of the algorithm to use it with confidence to sort. On the other hand, if the notion of correctness is that the output is ordered, you have to look at the internals to determine if the algorithm sorts and cannot use it with confidence otherwise, *e.g.*, a trivial algorithm that always returns the empty sequence satisfies this constraint. The right notion of correctness allows you to ignore the details of a system, but the wrong notion is useless; therefore, getting the notion of correctness right is of the utmost importance. Similarly, in the case of pipelined machine verification, a notion of correctness can be thought of as a constraint that, given an ISA (Instruction Set Architecture) specification, is satisfied only by "correct" MA (Micro-Architecture) machines. If the constraint allows trivial implementations, it is as useless for specifying the correctness of pipelined machines as the notion of ordered output is for specifying the correctness of sorting algorithms.

Previous approaches to pipelined machine verification have not been stated in these terms. For example, one finds that there are restrictions placed on the types of pipelined machines that can be checked (*e.g.*, machines with a flush instruction), or that the notion of correctness relates inconsistent MA states (*e.g.*, states not reachable from a flushed state) to ISA states, or that there are various conditions, often separated into safety and liveness conditions, that need to be checked, or that as new features are added, new notions of correctness are used. We explored the situation in detail for the BD (Burch and Dill [4]) variant of correctness used by Hunt and Sawada in [24, 25, 22, 23] because of the availability of proof scripts and because of the ubiquity of the BD approach to pipelined machine verification. We found that trivial machines satisfy this notion of correctness; a mechanical proof establishing this is described near the beginning of Section 3.2. We must point out that the actual machines verified in the above referenced work are not trivial machines. The machines are remarkably complicated and we consider their verification an impressive body of work.

We propose a notion of correctness based on WEBs (Well-founded Equivalence Bisimulations) that amounts to: when viewed appropriately, the pipelined machine and the ISA machine have the same infinite behaviors, up to stuttering. We account for stuttering because we are comparing systems at different levels of abstraction and an atomic step of one system may not be atomic in the other system. We say "viewed appropriately" because the two systems may represent data in different ways, *e.g.*, we show the equivalence of a machine that operates on integers with a machine that operates on bit-vectors. Another reason for the qualifier is that an MA state has more components than an ISA state, *e.g.*, the pipeline. Even when components overlap, they may exhibit different behaviors, *e.g.*, if the pipeline is non-empty, the program counter of an MA state will point several instructions ahead of the last completed instruction. Therefore, we use a *refinement map* or an *abstraction function* that relates MA states to corresponding ISA states. There are many ways in which to choose a refinement map. For example, BD approaches require that the MA machine have a flush instruction and relate an MA state with the ISA state obtained by flushing the MA state and retaining only the programmer visible components. A description of the BD notion of correctness and a more thorough discussion of the issues is given in Section 3, but, briefly, the main problem is that the MA state obtained from flushing can look very different from the original MA state. To make this point very clear, let PRIME be the system whose single behavior is the sequence of primes and let NAT be the system whose single behavior is the sequence of natural numbers. We do not consider NAT and PRIME equivalent, but using the refinement map from NAT to PRIME that maps i to the i^{th} prime, we can indeed prove the peculiar theorem that NAT and PRIME have the same behaviors. The moral is that we must be careful to not bypass the verification problem with the use of such refinement maps. Refinement maps that leave the important part of the state untouched are best. Refinement maps based on flushing modify important programmer visible components such as the register file; therefore, we find their use objectionable. As further evidence that something strange is happening, note that flushing-based refinements map inconsistent (unreachable) MA states to reachable ISA states. (See Section 3 for details.) To overcome these obstacles we use refinement maps that—to the extent possible—do not alter components of MA states: we map an MA state to the ISA state obtained by retaining the programmer visible components of the committed part of the state. This can be thought of as invalidating the partially executed instructions, which leaves the register file intact, but may change the program counter.

In Section 2, we define stuttering bisimulation, WEBs, and present the relevant theorems. In Section 3, we describe several variants of a simple pipelined machine described by Sawada in [22, 23]. This machine was designed to explain the issues in pipelined machine verification; it is a toy version of the final machine verified in Sawada's thesis. Since this is the first paper to use WEBs to verify pipelined machines, it seemed sensible to compare our results to existing work. We describe eleven proofs of correctness involving twelve variants of this machine. The variants include exceptions, fleshed-out ALUs, interrupts, and combinations of these features. In all cases, we proved the same final theorem.

Deciding what theorem to prove is only part of the story. The other part is to get the theorem mechanically checked with limited human interaction. In Section 4, we present a methodology for automating proofs of pipelined machine correctness. We have implemented our methodology in ACL2 (see [12, 11, 10, 1]) and have used it to verify the various previously mentioned machines. Given the descriptions of the machines, the refinement maps, and a few definitions, we automatically generate further definitions and proof obligations. If all the proof obligations are discharged by ACL2, correctness has been established. If not, at least a proof outline has been provided. All of the machines we verified were handled using this methodology and no intermediate abstractions were required. Some proof was required, for example, we show that a netlist description of a circuit is actually a 128-bit adder, but we did not have to prove any invariants about the pipeline. We present conclusions and discuss related work in Section 5.

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 $\prec$. We abbreviate $((s \prec w) \quad \vee \quad (s = w))$ by $s \preccurlyeq w$. From highest to lowest binding power, we have: parentheses, binary relations (*e.g.*, sBw), equality ($=$) and set membership ($\in$), conjunction ($\wedge$) and disjunction ($\vee$), implication ($\Rightarrow$), and finally, binary equivalence ($\equiv$). 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* σ is a sequence of states such that for adjacent states s, u, $s \dashrightarrow u$. A path σ is a *fullpath* if it is infinite. *fp*.σ.s denotes that σ is a fullpath starting at s.

2.2 Stuttering Bisimulation

We define a variant of stuttering bisimulation [2]. This notion is similar to the notion of a bisimulation [20, 18], but allows for finite stuttering, and therefore differs from weak bisimulation [18]. The idea is to partition the state space of a transition system into equivalence classes such that states in the same class have the same infinite paths up to stuttering. We show that the equivalence class obtained by relating ISA states to MA states, where the label of an MA state is determined by a refinement map, is a stuttering bisimulation.

Definition 2 *(match)*
Let i range over $\mathbb{N}$. Let INC be the set of strictly increasing sequences of natural numbers starting at 0; rigorously, $INC = \{\pi : \pi : \mathbb{N} \rightarrow \mathbb{N} \quad \wedge \quad \pi.0 = 0 \quad \wedge \quad \langle \forall i : i \in \mathbb{N} : \pi.i < \pi(i+1)\rangle\}$. The ith segment of fullpath σ with respect to $\pi \in INC$, ${}^{\pi}\sigma^{i}$ is given by the sequence $\sigma(\pi.i), \ldots, \sigma(\pi(i+1)-1)$. For $B \subseteq S \times S$, $\pi, \xi \in INC$, $i, j \in \mathbb{N}$, and fullpaths σ and δ, we abbreviate $\langle \forall s, t : s \in {}^{\pi}\sigma^{i} \wedge t \in {}^{\xi}\delta^{j} : sBt\rangle$ by ${}^{\pi}\sigma^{i}\, B\, {}^{\xi}\delta^{j}$. For $B \subseteq S \times S$, $\pi, \xi \in INC$:

$$match(B, \sigma, \delta) \;\equiv\; \langle \exists \pi, \xi : \pi, \xi \in INC : \langle \forall i : i \in \mathbb{N} : {}^{\pi}\sigma^{i} \;\; B \;\; {}^{\xi}\delta^{i}\rangle\rangle$$

Definition 3 *(Equivalence Stuttering Bisimulation (ESTB))*
B is an equivalence stuttering 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. $\langle \forall s, w \in S : sBw : \langle \forall \sigma : \mathit{fp}.\sigma.s : \langle \exists \delta : \mathit{fp}.\delta.w : \mathit{match}(B, \sigma, \delta)\rangle\rangle\rangle$

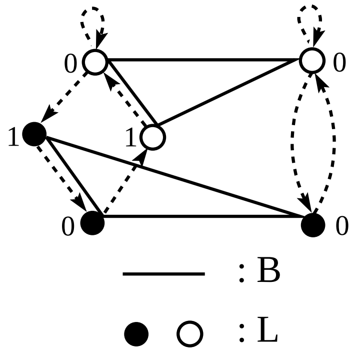

Fig. 1. The graph denotes a transition system, where circles denote states, the color of the circles denotes their label, and the transition relation is denoted by a dashed line. States related by the equivalence relation B are joined by a solid line. Notice that B is ESTB as related states have the same infinite paths, up to stuttering. To check that B is a WEB, let $\mathit{rank}(u, v)$ = tag of v, and use the well-founded witness $\langle \mathit{rank}, \langle \mathbb{N}, <\rangle\rangle$.

An example of an ESTB appears in Fig. 1. ESTBs are the appropriate notion of correctness because if we show that MA and ISA states are related by an ESTB, then it follows that the states have the same behaviors. However, ESTBs are very difficult to prove mechanically because one has to reason about infinite sequences. We would much rather reason about single steps. This is the motivation for the following definition.

Definition 4 *(Well-Founded Equivalence Bisimulation (WEB* [16, 19]*))*
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 $\mathit{rank} : S \times S \to W$, with $\langle W, \prec\rangle$ well-founded, and
$\langle \forall s, u, w \in S : sBw \;\wedge\; s \dashrightarrow u :$
$\quad \langle \exists v : w \dashrightarrow v : uBv\rangle \;\vee$
$\quad (uBw \;\wedge\; \mathit{rank}(u, u) \prec \mathit{rank}(s, s)) \;\vee$
$\quad \langle \exists v : w \dashrightarrow v : sBv \;\wedge\; \mathit{rank}(u, v) \prec \mathit{rank}(u, w)\rangle\rangle$

We call a pair $\langle rank, \langle W, \prec \rangle \rangle$ satisfying condition 3 in the above definition, a *well-founded witness*. An example of a WEB appears in Fig. 1. Note that to prove a relation is a WEB, reasoning about single steps of $\dashrightarrow$ suffices.

Notice that the difference between WEBs and ESTBs is in their third conditions. The third WEB condition can be thought of saying that given states s and w in the same class, such that s can step to u, u is either matched by a step from w, or u and w are in the same class and the rank function decreases (to guarantee that w is forced to take a step), or some successor v of w is in the same class as s and the rank function decreases (to guarantee that u is eventually matched). We have the following theorems.

Theorem 1 (cf. [19, 16]) *B is an ESTB on TS M iff B is a WEB on M.*

Theorem 1 says that the notion of stuttering bisimulation is exactly captured by the notion of WEB. The significance is that any stuttering bisimulation can be proved using WEBs.

Theorem 2 (cf. [2, 19]) *If B is a WEB on TS M and sBw, then for any $CTL^* \backslash X$ formula f, $M, s \models f$ iff $M, w \models f$.*

Theorem 2 says that states related by a WEB satisfy the same next-time free formulae of the branching-time logic CTL^*. As a consequence, they satisfy the same LTL (Linear Temporal Logic) formulae and the same safety and progress properties (up to stuttering). This is a strong notion of equivalence and that we can use it profitably depends, in part, on the use of refinement maps, which we now discuss.

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 one to prove the correctness of a pipelined machine in stages, where at each stage, one can focus on a specific aspect of the machine.

Definition 5 *(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.

Consider `MAnet`, a pipelined machine which represents numbers as bit-vectors and with an ALU described in a netlist language, and `MA`, a pipelined machine which represents numbers directly and whose ALU is described in terms of the familiar arithmetic operators. The labeling function of these two systems is the identity function. To show that `MAnet` refines `MA` we exhibit a refinement map which maps `MAnet` states to `MA` states by turning bit-vectors into numbers and prove that the resulting equivalence relation is a WEB. A consequence is that related `MAnet` and `MA` states have the same behaviors modulo the representation of numbers.

Theorem 3 *(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 $q \circ r$.

Consider the following application of this theorem. Suppose we prove that `MAnet` is a refinement of `MA`, as discussed above. Suppose we also show that `MA` is a refinement of `ISA`, the non-pipelined specification. Using the composition of WEBs theorem, we have that `MAnet` is a refinement of `ISA`. An advantage to proving correctness in stages is that if the difference between the machine descriptions from one stage to the next is not too great, we may be able to have the proofs go through automatically. This is because there is enough structural similarity in the two machines that ACL2 can decide equivalence on its own. In our experiments with various versions of Sawada's small machine, this is what we observed. Yet another advantage is that changes to the lower-level machines can be localized. For instance, if we change the adder from a serial adder to a carry-lookahead adder, then the resulting proof obligations are isolated to showing that the carry-lookahead adder is equivalent to the serial adder.

3 Pipelined Machine Verification

In this section, we describe various versions of Sawada's simple pipelined machine [22, 23] and the correctness criteria proved. We discuss correctness, the deterministic variants, and finally the non-deterministic ones.

Machines are modeled as functions in ACL2, *e.g.*, the first machine we define is `ISA` and this amount to defining `ISA-step`, a function that given an `ISA` state returns the next state. For all the machines, an instruction is a four-tuple consisting of an opcode, a target register, and two source registers. We give informal descriptions of the machines in this paper, since the formal descriptions in ACL2 are available from the author's Web page [14].

3.1 Correctness

In the introduction, we made the case that any notion of correctness can be thought of as a constraint. Pipelined machines that satisfy the constraint are "correct" implementations of the ISA, with respect to this notion of correctness. We can judge the merits of a notion of correctness by checking that no obviously incorrect machine satisfies the related constraint. We argued that the refinement maps should be understandable, *e.g.*, if we map MA states to ISA states then there should be a clear relationship between related states. Applying these criteria to the BD variant under consideration, we find that in both respects, this notion of correctness is flawed.

The notion of correctness that we use is simple to state: we show that the pipelined machine is a *refinement* of the ISA specification. Notice that any notion of correctness has to account for stuttering, *e.g.*, a pipelined machine requires several cycles to fill the pipeline, whereas an ISA machine executes an instruction per cycle; any notion of correctness also has to account for refinement, *e.g.*, a pipelined machine may represent numbers as bit-vectors, whereas the ISA machine may represent them directly. Our notion of correctness is the strongest notion that we can think of which accounts for stuttering and refinement. We will discuss the verification of a number of machines, but, regardless of the proof details, we always prove the same theorem, *viz.*, that a pipelined machine has the same infinite paths (up to stuttering and refinement) as an ISA machine.

3.2 Deterministic Machines

The deterministic machines are named `ISA`, `MA`, `ISA128`, `MA128`, `MA128serial`, and `MA128net`. `ISA` and `MA` correspond to the machines in [23] and the simple machines in [22]. We start with descriptions of `ISA` and `MA` and compare the BD approach to correctness with ours.

ISA An `ISA` state is a three-tuple consisting of a program counter (pc), a register file, and a memory. The next state of an `ISA` state is obtained by fetching the instruction pointed to by the pc from memory, checking the opcode, and performing the appropriate instruction. The possible instructions are addition, subtraction, and noop. In the case of addition, the target register is updated with the sum of the values in the source registers and the program counter is incremented. Subtraction is treated similarly. In the case of a noop, only the program counter is incremented. `ISA` is the specification for `MA`.

MA `MA` is a three-stage pipelined machine. An `MA` state is a five-tuple consisting of a pc, a register file, a memory, and two latches. The three stages are the fetch stage, the set-up stage, and the write-back stage. During the fetch stage, instructions are fetched from memory and stored in the first latch; during the set-up stage, the instruction in the first latch is passed to the second latch, but with the values of the source registers; during the write-back stage the values and the opcode are fed to the ALU which performs the appropriate instruction and the result is written into the target register, if the instruction was not a noop. The `MA` machine can execute one instruction per cycle once the pipeline is full, except when there are successive arithmetic instructions where the second instruction uses the target register of the first instruction as a source register. In this case, the machine is stalled for one cycle in order for the target register to be updated.

One difference between `MA` as defined above and the version given by Sawada (`SMA`) is that `SMA` has an extra input signal which determines whether the machine can fetch an instruction. With the use of this signal, `SMA` can be flushed, whereas we have no way of flushing `MA`.

Proof of SMA The proof of SMA given in [23, 22] uses a variant of the BD notion of correctness. The main theorem proved is that if the SMA starts in SMA_0, a flushed state, takes n steps to arrive at state SMA_n, also a flushed state, then there is some number m such that stepping the projection of SMA_0 m steps results in the projection of SMA_n. The projection of an SMA state is the ISA state obtained from the program counter, register file, and memory of the SMA state. Notice that for any system which never reaches a flushed state, the above condition is trivially true. Since MA is such a system, proving that it satisfies this condition is of no use.

The proof of SMA also includes a "liveness" component: it is proved that any SMA state can be flushed. Taking these two conditions together, we can ask: are there any pipelined machines for which we can prove the above theorems, but which are obviously not correct? The answer is yes and using Sawada's proof scripts, we provide a mechanically checked proof that a pipelined machine which does nothing satisfies this notion of correctness. This proof and any other mechanical proof that we claim to have completed can be found on the author's Web page [14].

Flushing Proof of MA Even though there is no way to flush MA, we can use flushing to prove that MA is a refinement of ISA. This digression allows us to discuss issues related to refinement maps. One approach is to modify MA so that it can be flushed (which would give us SMA), but this is a different machine. The approach we take is to define an auxiliary function that flushes an MA state. Using this function we show that MA is a refinement of ISA. Notice that in contrast to the proof of SMA, 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 entirely satisfactory and this has to do with the use of flushing as a refinement map.

When we define systems at different levels of abstraction, we often find that there are inconsistent states, *e.g.*, consider an MA state in which the first latch contains an instruction that is not in memory. This is usually dealt with by considering only "good" (reachable) MA states. The flushing-based refinement imposes no such restriction because pipelined machines are self-stabilizing: from any state they eventually reach a good state and flushing guarantees this. As a result, the refinement map—which is supposed to show us how to view MA states as ISA states—relates inconsistent MA states with consistent ISA states (all ISA states are "good"). We find such a refinement objectionable.

Proof of MA The approach we take to pipelined machine verification in the rest of this paper is to prove a WEB where the refinement map relates pipelined machine states to the ISA states obtained by retaining the programmer visible components of the committed part of the pipelined state. The definition of the refinement map is based on the function `committed-MA` which takes an MA state and returns the `MA` state obtained by invalidating all partially completed instructions and moving the program counter back based on the number of partially completed instructions. As mentioned previously, "good" MA states are the ones reachable form a committed state. The function `good-MA` recognizes `MA` state s if `committed-MA`.s, stepped the appropriate number of times, is s. Notice that as with flushing, this function is easy to define because we can use the definition of `MA`. In fact, it is simpler to define than the flushing operation because we are not trying to get the machine into a special state: we are just stepping it. Notice that the use of `good-MA` allows us to avoid defining an invariant (an error prone process), hence, we maintain this methodological feature of the BD approach. We call this approach the "commitment approach."

ISA128 An `ISA128` state is a four-tuple consisting of a program counter (pc), a register file, a memory, and an exception flag. The next state of an `ISA128` state is obtained by fetching the instruction pointed to by the pc from memory, checking the opcode, and performing the appropriate instruction. The possible instructions are addition, multiplication, and noop. In the case of addition, the program counter is incremented and the target register is modified to contain *sum*, the sum of the values in the source registers if the sum is less than 2^{128}. Otherwise, if an overflow occurs, the exception flag is checked; if it is off, then the program counter is incremented and the target register is assigned $sum \pmod{2^{128}}$; if the exception flag is on, the 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. (A function about which we know only that it satisfies some specified properties is called a constrained function. An uninterpreted function is a special case of a constrained function.) Multiplication is treated similarly. In the case of a noop, only the program counter is incremented. `ISA128` is the specification for `MA128`, `MA128serial`, and `MA128net`.

MA128, MA128serial, and MA128net `MA128` is a three-stage pipelined machine. An `MA128` state is a six-tuple consisting of a program counter, a register file, a memory, two latches, and an exception flag. As with `MA`, the three stages are the fetch stage, the set-up stage, and the write-back stage. If an overflow occurs during an arithmetic operation, then the partially executed instructions are invalidated and the interrupt handler is called. The resulting state is constrained to be flushed (*i.e.*, both latches are invalid). We prove that `MA128` is a refinement of `ISA128` using the commitment approach.

`MA128serial` is the same as `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 [11]. We used the commitment approach to prove that `MA128serial` refines `MA128`. Since the ALU of `MA128` operates on bit-vectors, the refinement map used maps the bit-vectors in the register file and the second latch to numbers. By Theorem 3 (composition), we get that `MA128serial` is a refinement of `ISA128`. Although the use of the composition theorem here was not essential, it was nice to be able to break up the proof into these two logically separate concerns. ACL2 can take advantage of the structural similarity between `MA128serial` and `MA128`, hence the proof of correctness is pretty fast. This is covered in more detail later.

`MA128net` is the same as `MA128`, except that the ALU is defined in terms of an adder described in a netlist language. The netlist adder is a 128-bit adder and is described in terms of primitive functions on bits. 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`.

3.3 Non-deterministic Machines

We now consider the non-deterministic versions of the six deterministic machines described above. The names of these machines are: `ISAint`, `MAint`, `ISA128int`, `MA128int`, `MA128intserial`, and `MA128intnet`. They are elaborations of the similarly named deterministic machines, except that they can be interrupted. Whereas the next state of the deterministic machines is a function of the current state (even in the presence of exceptions), the next state of the machines described in this section also depends on the interrupt signal, which is free. Therefore, the machines in this section are non-deterministic.

The approach in [22] to dealing with interrupts is different. There, the correctness criterion is: 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.

In our approach, we have to show that the pipelined machine is a refinement of the specification, as before. Note that this is the same final theorem we proved in the deterministic case, as WEBs can be used to relate non-deterministic systems. Therefore, our notion of correctness cannot be satisfied by a pipelined machine that ignores interrupts. Another advantage is that our proof obligation is still about single steps of the machines, as opposed to finite behaviors. The problem with the finite behaviors approach was highlighted above: when comparing finite executions of a pipelined machine and of its specification, there are executions with different lengths; how does one relate interrupts in one execution with interrupts in the other?

ISAint An `ISAint` state is a four-tuple consisting of a program counter (pc), a register file, a memory, and an interrupt register. The next state of an `ISAint` state is obtained by first checking the interrupt register. If non-empty, the interrupt handler is called. The interrupt handler is a constrained function of the register file, memory, and interrupt register and returns a state with the same pc and register file, but may modify memory, and clears the interrupt register. If the interrupt register is empty, we check if an interrupt has been raised. If so, we record the interrupt type in the interrupt register. If not, we proceed by fetching the instruction pointed to by the pc from memory, checking the opcode, and performing the appropriate instruction. The possible instructions are addition, multiplication, and noop. In the case of addition, the target register is modified to contain the sum of the values in the source registers and the program counter is incremented. Multiplication is treated similarly. In the case of a noop, only the program counter is incremented. `ISAint` is the specification for `MAint`.

MAint `MAint` is a three-stage pipelined machine. An `MAint` state is a six-tuple consisting of a pc, a register file, a memory, two latches, and an interrupt register. The three stages are the fetch stage, the set-up stage, and the write-back stage, as before. The next state of an `MAint` state is obtained by first checking the interrupt register. If non-empty, partially executed instructions are aborted and the interrupt handler is called. Otherwise we check if an interrupt has been raised, in which case we abort partially executed instructions and set the interrupt register. Otherwise, execution proceeds in a fashion similar to the execution of `MA`. The refinement map used to show that `MAint` is a refinement of `ISAint` is almost identical to the one used to show that `MA` is a refinement of `ISA`, except that the interrupt register is also retained.

ISA128int As the name implies this is the ISA-level specification of 128-bit ALU machine with exceptions and interrupts. Interrupts are given priority, and this machine is defined the way you would expect: it is similar to `ISAint`, except that arithmetic operations are checked for overflows, in which case the exception handler is called. This machine is the specification used for the machines `MA128int`, `MA128intserial`, and `MA128intnet`.

MA128int, MA128intserial, and MA128intnet `MA128int`, `MA128intserial`, and `MA128intnet` are three-stage pipelined machines analogous to `MA-128`, `MA128serial`, and `MA128net`, respectively, but with exceptions. As before, we show that `MA128int` is a refinement of `ISA128int`, that `MA128intserial` is a refinement of `MA128int` (where the refinement map converts bit-vectors to integers) and finally that `MA128intnet` is a refinement of `MA128intserial`. By the composition theorem, we get that all of these machines are refinements of `ISA128int`.

4 Proof Decomposition

To reduce the amount of guidance that has to be manually given to the theorem prover, we develop a methodology with mechanical support which we use to verify the various variants of Sawada's machine. We outline the approach in this section by discussing the proof that `MA` is a refinement of `ISA`.

A user of the ACL2 theorem prover often uses books provided by others. Books are files of ACL2 definitions and theorems, *e.g.*, there are books for reasoning about arithmetic, data structures, floating-point arithmetic, and the like.

Books can be certified and then included (*i.e.*, loaded) in future sessions without having to re-admit the definitions and theorems. Books are also used to structure proofs, by placing related definitions and theorems in the same book. The proof that `MA` is a refinement of `ISA` uses some general-purpose books, *e.g.*, a book about arithmetic and books that we have developed for proving that a concrete system is a refinement of an abstract system.

The books specific to the proof are `"ISA"`, `"MA"`, and `"MA-ISA"`. `"ISA"` and `"MA"` contain the definitions of `ISA-step` and `MA-step`, the functions to step (*i.e.*, compute the next state of) the `ISA` and `MA` machines, respectively. The `"MA-ISA"` book contains the definition of the refinement map, `MA-to-ISA`, and the function recognizing "good" `MA` states, `good-MA`. These functions are described in Section 3.2. Also included is the definition of a rank function, `MA-rank`. The rank function is very simple: it is either 0, 1, or 2 based on how many steps it will take `MA` to commit an instruction. The rest of `"MA-ISA"` consists of three macro calls. These macros are defined in the books we developed for reasoning about WEBs. Only these definitions are required; the proof obligation is generated by the macros and discharged by ACL2. No user provided theorems, intermediate abstractions, or invariant proofs are needed. The books containing the verification of the other machines are similar. Some of them require auxiliary lemmas, but the lemmas are about the serial adder, the netlist description language, and the like.

We now explain what the three macro calls mentioned above do. The first, `generate-full-system`, is given seven arguments. They are the names of the functions to step and recognize `ISA` and `MA` states, `MA-to-ISA`, `good-MA`, and `MA-rank`. This macro, as the name implies, generates the system to be verified. Recall that in order to prove a WEB, we need one system; this system is the union of the `MA` and `ISA` systems. The function `R`, corresponding to the transition relation ($\dashrightarrow$ in the definition of WEBs) of the system is defined. `R` is a function of two arguments that holds between `ISA` states and their successors and `MA` states and their successors. The function `wf-rel` is defined; this is a function of two arguments that holds if the first argument is an `ISA` state, the second argument is a good `MA` state and `MA-to-ISA` of the `MA` state equals the `ISA` state. `B` is defined to be the reflexive, symmetric, transitive closure of `wf-rel` and corresponds to the equivalence relation in the definition of WEBs. `Rank` is defined to return 0 on `ISA` states and `MA-rank` otherwise and corresponds to the rank function in the definition of WEBs. The function `take-appropriate-step` is defined to return `ISA-step` of its argument, if an `ISA` state, or `MA-step` of its argument otherwise. Several other functions corresponding to existentially quantified formulae are also defined.

The second macro called is `prove-web` and it is given five arguments. They are the names of the functions to step and recognize `ISA` and `MA` states and `MA-rank`. This macro generates the following proof obligation:

```
(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))))))
```

Most of ACL2's effort goes into establishing the above theorem. This theorem says that if `u` is the `ISA-step` of `s`, `v` is the `MA-step` of `w`, and `s` and `w` are related by `wf-rel`, but `u` and `v` are not, then `wf-rel` relates `s` and `v` and the `MA-rank` of `v` is less than that of `w`. `E0-ord-<` is the less than relation on ACL2 ordinals. (The proofs in this paper depend only on the natural numbers, an initial segment of the ordinals.) Compare this theorem with the definition of WEBs. We used domain-specific information to remove 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 neither reflexive, symmetric, nor transitive, but `wf-rel` relates `ISA` and `MA` states and this is enough for us to automatically deduce the theorem for the induced equivalence relation. Finally, we ignore the second disjunct in the third condition of the definition of WEBs because `ISA` does not stutter. 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, ACL2 ran out of memory after 30 hours, yet the above theorem is now proved in about 11 seconds. `Prove-web` also generates some "type" theorems that we do not discuss further.

The final macro called is `wrap-it-up` and it is given the same seven arguments given to the first macro call. What this macro does is a little bit more interesting. It generates the proof obligations required to show the WEB by generating the following three theorems (compare with the definition of WEBs):

```
(defequiv B)

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

(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))))
```

That is, `B` is an equivalence relation, `rank` is a well-founded witness, and the third condition in the definition of WEBs. (`E0-ordinalp` is a predicate that recognizes ACL2 ordinals and recall that `R` is the transition relation.) What is interesting is how the final theorem is proved. It is proved using a proof rule of ACL2 called *functional instantiation* (see [13]). In one of the books that support reasoning about WEBs, we proved that from a theorem similar to `b-is-a-wf-bisim-core`, a theorem similar to `b-is-a-wf-bisim` follows. The theorem proved was about a constrained system, *i.e.*, a system about which we have some constraints, but not a definition. Using functional instantiation, we can prove `b-is-a-wf-bisim` by showing that the system defined by `MA` and `ISA` satisfies the constraints of the constrained system. In this way, we prove a decomposition theorem once about a constrained system that allows us to reduce our problem to one with no quantifiers and minimal case analysis, and we can use the decomposition on any system that satisfies the constraints.

We end this section by describing 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). In our 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.

5 Conclusions

Various approaches to pipelined machine verification appear in the literature. In [4], a notion of correctness based on flushing and commuting diagrams is presented. Another approach using skewed abstraction functions is outlined in [26, 5, 27]. There are approaches based on model-checking, *e.g.*, in [17], compositional model checking and symmetry reductions are used and in [6] assume-guarantee reasoning at different time scales is discussed. There are theorem-proving based approaches, *e.g.*, in [24, 25, 22, 23], an intermediate abstraction called MAETT is used to verify some very complicated machines. Another theorem-proving approach is presented in [8, 9, 7], where "completion functions" are used to decompose the abstraction function. In [28], a related notion of correctness based on state and temporal abstraction is used to verify a pipelined machine. In addition, there is work on decision procedures, *e.g.*, [3, 21] present decision procedures for boolean logic with equality and uninterpreted function symbols and in [3] their decision procedure is used to verify pipelined machines.

We have presented an approach to pipelined machine correctness based on WEBs and argued that only machines which truly implement the instruction set architecture satisfy our notion of correctness. In contrast, we showed with mechanical proof that the Burch-Dill variant of correctness used in [22, 23] can be satisfied by trivial, clearly wrong, implementations. We verified various extensions to the simple pipelined machine presented in [22, 23]. Our extensions include exception handling, interrupts, and ALUs described in part at the netlist level. In every case, we proved the same final theorem. In addition, we showed how to use the compositionality of WEBs and ACL2's functional instantiation to relate term-level descriptions to netlist-level descriptions. All of the proofs were done within one logical system and we thus avoided the semantic gaps that could otherwise result. To automate the proofs, we developed a methodology with mechanical support that is used to generate and discharge the proof obligations. The main proof obligation generated contains no quantifiers and has minimal case analysis, but once proved is used to automatically infer the WEB. This is done with the functional instantiation of a decomposition theorem that is part of the mechanical support for WEBs that we provide. All of the proofs are available from the author's Web page [14].

Using our methodology and notion of correctness, we were able to prove Sawada's example ("Proof of MA" in Section 3.2) without the use of any intermediate abstractions or invariants. The size of the file containing the proof, which does not include the machine definitions, 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).

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 has been a great resource and inspiration. He also provided a proof of the equivalence between the serial adder and the netlist adder. Rob Sumners and Kedar Namjoshi read the paper and made many useful suggestions.

References

[1] B. Brock, M. Kaufmann, and J. S. Moore. ACL2 theorems about commercial microprocessors. In M. Srivas and A. Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD'96)*, pages 275–293. Springer-Verlag, 1996.

[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] 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.

[7] 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.

[8] R. Hosabettu, M. Srivas, and G. Gopalakrishnan. Decomposing the proof of correctness of a pileplined microprocessors. In A. J. Hu and M. Vardi, editors, *Computer-Aided Verification–CAV '98*, volume 1427 of *LNCS*. Springer-Verlag, 1998.

[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, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Press, June 2000.

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

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

[13] 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`.

[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] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1990.

[19] 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.

[20] D. Park. Concurrency and automata on infinite sequences. In *Theoretical Computer Science*, volume 104 of *LNCS*, pages 167–183. Springer-Verlag, 1981.

[21] 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.

[22] 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/`.

[23] 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.

[24] 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.

[25] 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.

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

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

[28] 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.