A Framework for Verifying Bit-Level Pipelined Machines Based on Automated Deduction and Decision Procedures *

Panagiotis Manolios[1] and Sudarshan K. Srinivasan[2]
[1]*College of Computing*
[2]*School of Electrical & Computer Engineering*
Georgia Institute of Technology
Atlanta, Georgia 30318
[1]manolios@cc.gatech.edu, [2]darshan@ece.gatech.edu

Abstract. We describe an approach to verifying bit-level pipelined machine models using a combination of deductive reasoning and decision procedures. While theorem proving systems such as ACL2 have been used to verify bit-level designs, they typically require extensive expert user support. Decision procedures such as those implemented in UCLID can be used to automatically and efficiently verify *term-level* pipelined machine models, but these models use numerous abstractions, implement a subset of the instruction set, and are far from executable. We show that by integrating UCLID with the ACL2 theorem proving system, we can use ACL2 to reduce the proof that an executable, bit-level machine refines its instruction set architecture to a proof that a term-level abstraction of the bit-level machine refines the instruction set architecture, which is then handled automatically by UCLID. We demonstrate the efficiency of our approach by applying it to verify a complex seven stage bit-level interface pipelined machine model that implements 593 instructions and has features such as branch prediction, exceptions, and predicated instruction execution. Such a proof is not possible using UCLID and would require prohibitively more effort using just ACL2.

Keywords: verification, pipelined machines, refinement, bit-level, automated reasoning, ACL2

1. Introduction

The ever-increasing complexity of microprocessor designs and the potentially devastating economic consequences of shipping defective products has made functional verification a bottleneck in the microprocessor design cycle, requiring a large amount of time, human effort, and resources (Bentley, 2001; Semiconductor Industry Association, 2004). For example, the 1994 Pentium FDIV bug cost Intel $475 million and it is estimated that a similar bug in the current generation Intel Pentium processor would cost Intel $12 billion (Bentley, 2005).

One of the key optimizations used in these designs is pipelining. Simulation and property-based verification are the main approaches to validating such designs in industry (Bentley, 2001). The problem with simulation is that it is not exhaustive. This is also a problem with property-based veri-

* This research was funded in part by NSF grants CCF-0429924, IIS-0417413, and CCF-0438871.

fication, because the very large number of complex properties required to specify the behavior of pipelined machines makes it easy to have incomplete specifications.

Pipelined machine verification has also received a fair amount of interest from the research community. The two main approaches studied are based on the use of deductive reasoning and decision procedures. Approaches that use theorem provers such as ACL2 (Kaufmann et al., 2000b; Kaufmann et al., 2000a) can be used to verify bit-level pipelined machine models but require significant human effort from an expert user. Approaches based on decision procedures such as UCLID (Bryant et al., 2002; Lahiri and Seshia, 2004) are highly automated but their application is restricted to the verification of *term-level* models, models that abstract away the datapath, implement a small subset of the instruction set, require the use of numerous abstractions, and are far from executable.

The restriction to term-level models has severely limited the applicability of approaches based on decision procedures because to be industrially applicable, we need a firm connection to the RTL level, something that abstract term-level models do not provide. Our main contribution is to show how to attain a high degree of automation when verifying pipelined machines defined at the RTL level. We do this by combining deductive reasoning with decision procedures. Deductive reasoning, using the ACL2 theorem proving system, is used to reduce the correctness theorem for an executable, bit-level pipelined machine to a theorem about a term-level model, which can then be automatically discharged using decision procedures. We demonstrate our approach by integrating the UCLID decision procedure with the ACL2 theorem proving system and using the combined system to verify a complex seven-stage pipelined machine model defined mostly at the bit-level. The work presented in this paper extends a previous conference version (Manolios and Srinivasan, 2005c) by including a more detailed, thorough, and complete description of the techniques developed and their application.

Verification entails proving that the pipelined machine refines its instruction set architecture. The notion of refinement we use is Well Founded Equivalence Bisimulation (WEB), a compositional notion that preserves both safety and liveness properties. We take advantage of the compositionality of WEB-refinement to decompose the proof that the bit-level pipelined machine model refines its instruction set architecture into several refinement steps. The refinement steps that relate the bit-level models with the term-level models are handled by ACL2, and the step relating the term-level pipelined machine model with its instruction set architecture is handled by UCLID.

We use the ACL2 theorem-proving system because it has been successfully applied to RTL-level hardware verification efforts in industry. For example, ACL2 has been used to verify the floating point units of AMD-K5 processor (Russinoff, 1999), AMD-K7 processor (Russinoff, 1998), and IBM

Power4^{TM} processor (Sawada, 2002). ACL2 has also been used as part of the verification effort of an IBM secure co-processor (Smith et al., 1999) and an intrinsic partitioning mechanism in the AAMP7 avionics microprocessor (Greve et al., 2004).

When verifying term-level models, we found that the UCLID decision procedure is orders of magnitude faster than ACL2. For example, the verification of a simple five-stage DLX pipelined machine defined at the term-level took three seconds with UCLID, but took fifteen and a half *days* with ACL2 (Manolios and Srinivasan, 2004b).

Unfortunately, UCLID has several limitations of its own that led us to build a system integrating UCLID with ACL2. UCLID specifications are restricted to term-level models and are therefore not executable. We define systems using the ACL2 programming language, which not only results in executable models, but even allows us to simulate the models at close to C speeds (Greve et al., 2000). A related point is that since term-level models tend to contain only one instruction per instruction class, they do not capture the semantics of the instruction set architecture, which makes it impossible to reason about software. None of these restrictions apply to ACL2, so we can reason about machine code running on the pipelined machine, as we discuss later in this paper. Another difference between ACL2 and UCLID is that ACL2 is far more expressive. For example, we cannot fully state the refinement theorem in UCLID; instead we state the "core" of the refinement theorem. Even this requires that we drastically modify the UCLID models by adding external inputs, state, and combinational logic. For our models, these modifications can involve on the order of one thousand lines of code, making it difficult to guarantee that the correctness proofs involving these "polluted models" imply the correctness of the original, unpolluted models. In our system, by using the expressive power of ACL2, these problems are avoided.

The rest of the paper is organized as follows. In Section 2, we describe the seven-stage pipelined machine model, most of which is defined at the bit-level. In Section 3, we provide an overview of the refinement-based notion of correctness we use to verify the seven-stage model. In Section 4, we give a high level description of our integration of UCLID with ACL2 so as to allow the reader to better understand the various issues that arise in the refinement proof. A more detailed description will appear elsewhere. Section 5 describes in detail every major step in the refinement proof. Section 6 gives the verification statistics of the proof in terms of the running time and expert user effort required. The ACL2 and UCLID proof scripts required to reproduce our results are available upon request. In Section 7, we show how to reduce reasoning about software running on the pipelined machine to reasoning about software running on the instruction set architecture by appealing to the

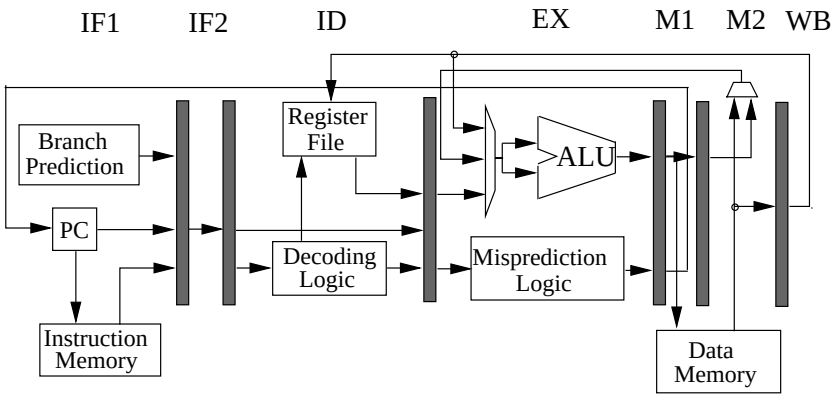

Figure 1. High-level organization of bit-level interface processor model

refinement and composition theorems. We describe related work in Section 8 and conclude in Section 9.

2. Processor Model

We demonstrate our verification approach using a complex executable seven-stage pipelined machine model, most of which is defined at the bit-level. The high-level organization of the pipeline is inspired by the Intel XScale architecture (Clark et al., 2001) and is shown in Figure 1. The model has seven pipeline stages including a 2-cycle fetch, a decode, an execute, a 2-cycle memory access, and a write back. The model has various features such as branch prediction, precise exceptions, and predicated instruction execution.

The model is described at the bit-level except for the instruction and data memories, the register file, and combinational circuit blocks such as the ALU; these blocks have bit-level interfaces *i.e.*, their inputs and outputs are bit-vectors, but they are not necessarily defined at the bit-level internally. For example, the ALU in our machine takes bit-vector inputs, converts the inputs to integers, performs the appropriate ALU operation on these integers, and converts the result to a bit-vector, which is the output of the ALU unit. Therefore, we use the term "bit-level interface" to describe the model. In the bit-level interface model the instruction decoder, control logic, and data path logic operate on bit-vectors. The model is described using the ACL2 programming language, and unlike term-level models it is executable. In Section 7, we show an example program (a dynamic programming solution to the Knapsack problem) that executes on the model. Instructions are 32 bits in length and the model has 16 registers. The size of the data path is a parameter that can be

set to any integer value greater than one, and the verification times of our correctness proofs do not vary with the size of the data path.

The model implements 16 types of ALU instructions, a return from exception instruction, and various branch, jump, load, and store instructions. Our model has both register-register and register-immediate addressing modes and our model supports predicated instruction execution, *i.e.*, some instructions have an associated condition that depends on the processor status flags. The instructions are allowed to complete and update the programmer visible components such as the program counter, the data memory, and the register file only if the condition associated with the instruction is true. Each of the ALU, branch, load, and store instructions can be executed using 16 different conditions. ALU, load, and store instructions can also use immediate values. In all, the model implements 593 instructions.

3. Refinement

In this section, we review the required background on the theory of refinement used in this paper; for a full account see (Manolios, 2001; Manolios, 2003). Pipelined machine verification is an instance of the refinement problem: given an abstract specification, S, and a concrete specification, I, show that I refines (implements) S. In the context of pipelined machine verification, the idea is to show that MA, a machine modeled at the microarchitecture level, a low level description that includes the pipeline, refines ISA, a machine modeled at the instruction set architecture level. A refinement proof is relative to a *refinement map*, r, a function from MA states to ISA states. The refinement map shows one how to view an MA state as an ISA state, *e.g.*, the refinement map has to hide the MA components (such as the pipeline) that do not appear in the ISA. Refinement for us means that the two systems are *stuttering bisimilar*: for every pair of states w, s such that w is an MA state and $s = r.w$, one has that for every infinite path σ starting at s, there is a "matching" infinite path δ starting at w, and conversely. That σ and δ "match" implies that applying r to the states in δ results in a sequence that is equivalent to σ up to finite stuttering (repetition of states). Stuttering is a common phenomenon when comparing systems at different levels of abstraction, *e.g.*, if the pipeline is empty, MA will require several steps to complete an instruction, whereas ISA completes an instruction during every step.

The ISA and MA machines are arbitrary transition systems (TS). A TS, $\mathcal{M}$, is a triple $\langle S, \dashrightarrow, L\rangle$, consisting of a set of states, S, a left-total transition relation, $\dashrightarrow \subseteq S^2$, and a labeling function L whose domain is S and where $L.s$ (we sometimes use an infix dot to denote function application) corresponds to what is "visible" at state s.

In more detail, our notion of refinement is based on the following definition of stuttering bisimulation (Browne et al., 1988), where by $fp(\sigma,s)$ we mean that σ is a fullpath (infinite path) starting at s, and by $match(B,\sigma,\delta)$ we mean that the fullpaths σ and δ are equivalent sequences up to finite stuttering (repetition of states).

DEFINITION 1. *$B \subseteq S \times S$ is a stuttering bisimulation (STB) on TS $\mathcal{M} = \langle S, \dashrightarrow, L\rangle$ iff B is an equivalence relation and for all s,w such that sBw:*

$$\begin{array}{ll} (\text{Stb1}) & L.s = L.w \\ (\text{Stb2}) & \langle \forall \sigma : fp(\sigma,s) : \langle \exists \delta : fp(\delta,w) : match(B,\sigma,\delta)\rangle\rangle \end{array}$$

The formal definition of match follows.

DEFINITION 2. *(match) Let i range over $\mathbb{N}$. Let* INC *be the set of strictly increasing sequences of natural numbers starting at 0. The i^{th} segment of an infinite sequence σ with respect to $\pi \in$ INC, ${}^{\pi}\sigma^{i}$, is given by the sequence $\langle \sigma(\pi.i), \ldots, \sigma(\pi(i+1)-1)\rangle$. We define $match(B,\sigma,\delta) \equiv \langle \exists \pi,\xi \in INC :: corr(B,\sigma,\pi,\delta,\xi)\rangle$, where $corr(B,\sigma,\pi,\delta,\xi) \equiv \langle \forall i \in \mathbb{N} :: \langle \forall s,w : s \in {}^{\pi}\sigma^{i} \wedge w \in {}^{\xi}\delta^{j} : sBw\rangle\rangle$.*

Browne, Clarke, and Grumberg have shown that states that are stuttering bisimilar satisfy the same next-time-free temporal logic formulas (Browne et al., 1988).

LEMMA 1. *Let B be an STB on $\mathcal{M}$ and let sBw. For any* $\text{CTL}^* \setminus \text{X}$ *formula f, $\mathcal{M},w \models f$ iff $\mathcal{M},s \models f$.*

We note that stuttering bisimulation differs from weak bisimulation (Milner, 1990) in that weak bisimulation allows infinite stuttering. Distinguishing between infinite and finite stuttering is important, because (among other things) we want to distinguish deadlock from stutter.

When we say that MA refines ISA, we mean that in the disjoint union ($\uplus$) of the two systems, there is an STB that relates every pair of states w, s such that w is an MA state and $r.w = s$.

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

STB refinement is a generally applicable notion. However, since it is based on bisimulation, it is often too strong a notion and in this case refinement

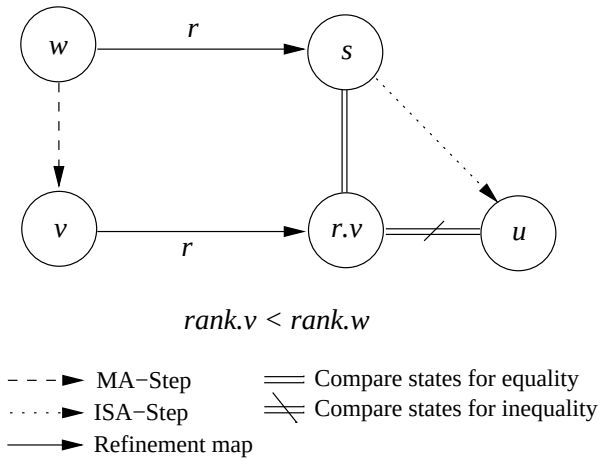

Figure 2. Diagram shows the core theorem that can be expressed in CLU logic.

based on stuttering *simulation* should be used (see (Manolios, 2001; Manolios, 2003)). The reader may be surprised that STB refinement theorems can be proved in the context of pipelined machine verification; after all, features such as branch prediction can lead to non-deterministic pipelined machines, whereas the ISA is deterministic. While this is true, the pipelined machine is related to the ISA via a refinement map that hides the pipeline; when viewed in this way, the nondeterminism is masked and we can prove that the two systems are stuttering bisimilar (with respect to the ISA visible components).

A major shortcoming of the above formulation of refinement is that it requires reasoning about infinite paths, something that is difficult to automate (Namjoshi, 1997). In (Manolios, 2001), WEB-refinement, an equivalent formulation is given that requires only local reasoning, involving only MA states, the ISA states they map to under the refinement map, and their successor states. In (Manolios and Srinivasan, 2004a), it is shown how to automate the refinement proofs in the context of pipelined machine verification. The idea is to strengthen, thereby simplifying, the refinement proof obligation; the result is the following CLU-expressible formula, where *rank* is a function that maps states of MA into the natural numbers.

THEOREM 1. $\texttt{MA} \approx_r \texttt{ISA}$ *if:*

$$\begin{aligned}\langle \forall w, v \in \texttt{MA},\ s, u \in \texttt{ISA} :: \\ & s = r.w \ \wedge\ u = \texttt{ISA-step}(s) \ \wedge \\ & v = \texttt{MA-step}(w) \ \wedge\ u \neq r.v \\ \Longrightarrow \quad & s = r.v \ \wedge\ \mathit{rank}.v < \mathit{rank}.w\rangle\end{aligned}$$

In the formula above s and u are ISA states, and w and v are MA states; `ISA-step` is a function corresponding to stepping the ISA machine once

and `MA-step` is a function corresponding to stepping the MA machine once. The proof obligation relating s and v can be thought of as the safety component, and the proof obligation that $rank.v < rank.w$ can be thought of as the liveness component. The formula is represented pictorially in Figure 2. Notice that we can state the above theorem without using the variables s, u, and v, as their values are uniquely determined by the value of w.

Note that the notion of WEB-refinement is parameterized by the refinement map used. In this paper, we use the commitment refinement map (Manolios, 2000), where MA states are mapped to ISA states by invalidating all partially executed instructions in the pipeline, undoing any effect they had on the programmer-visible components, and projecting out the programmer-visible components. In previous work, we have explored the use of and impact of refinement maps on pipelined machine verification, *e.g.*, in (Manolios and Srinivasan, 2005b), we present a new class of refinement maps that can provide several orders of magnitude improvements in verification times over the standard flushing-based refinement maps for term-level models.

WEB-refinement is *compositional* and a complete compositional reasoning framework based on our notion of refinement is given in (Manolios and Srinivasan, 2005a). For example, one can prove the following theorem, where $r;q$ denotes functional composition, *i.e.*, $(r;q)(s) = q(r.s)$.

THEOREM 2. *(Composition)*
If $\mathcal{M} \approx_r \mathcal{M}'$ *and* $\mathcal{M}' \approx_q \mathcal{M}''$ *then* $\mathcal{M} \approx_{r;q} \mathcal{M}''$.

Another useful compositional theorem is the following.

THEOREM 3. *(Composition)*

$$\frac{\mathtt{MA} \approx_r \cdots \approx_q \mathtt{ISA} \qquad \mathtt{ISA} \parallel P \vdash \varphi}{\mathtt{MA} \parallel P \vdash \varphi}$$

The above theorem states that to prove MA $\parallel P \vdash \varphi$ (think of this as saying that MA, a pipelined machine, executing program P satisfies property φ, a CTL$^* \setminus$ X property over the ISA visible components), it suffices to prove MA $\approx$ ISA and ISA $\parallel P \vdash \varphi$: that MA refines ISA and that ISA, executing P, satisfies φ. This is a powerful rule as it allows us to reduce correctness proofs about programs executing on the MA to proofs about programs executing on the ISA.

4. Integrating UCLID with ACL2

Our integration of the UCLID decision procedure with the ACL2 theorem proving system is coarse-grained, meaning that the user has to invoke the decision procedure explicitly. This allows us to avoid the well-known difficulties associated with the fine-grained integration of decision procedures into heuristic theorem provers (Boyer and Moore, 1988). In this section, we will give a high-level description of our shallow embedding of the CLU logic and the UCLID specification language in ACL2. A description of the embedding is useful in understanding the various issues that arise in the refinement proof. However, this paper is about an application of the integration of UCLID with ACL2 to verify a complex pipelined machine. The full details of the integration and the embedding of CLU and the UCLID specification language in ACL2 are rather technical and will be presented elsewhere.

The CLU syntax and semantics and the UCLID specification language are described in (Bryant et al., 2002), and (Seshia et al., 2003a), respectively. The UCLID specification language is based on CLU, but extends it with features such as macros and convenient commands for expressing symbolic simulation. UCLID specifications are therefore more high-level than the corresponding CLU specifications, which means that UCLID specifications are semantically closer to ACL2 expressions, which is why we chose to interface ACL2 with UCLID instead of just CLU. We can then use the UCLID symbolic simulation engine (implemented in the UCLID tool) to generate the CLU formulas corresponding to the UCLID specification.

We first give an overview of how we embed CLU into ACL2. The CLU logic contains the boolean connectives, uninterpreted functions and predicates, equality, counter arithmetic, ordering, and restricted lambda expressions. Booleans, integers, equality, ordering, successor, and predecessor functions in CLU are mapped to the corresponding entities in ACL2.[1] CLU's uninterpreted functions (UFs) and uninterpreted predicates (UPs) are modeled in ACL2 using constrained functions. ACL2 has an *encapsulation* mechanism that allows one to safely introduce functions about which only a set of constraints is known. To model UFs, we use constrained functions which have the property that if their inputs are integers, then their outputs are integers also. Similarly, UPs are modeled as functions that given integer inputs return booleans. The embedding of UFs and UPs highlights one of the issues with embedding CLU into ACL2, which is that the CLU logic obeys a statically monomorphic type discipline, while ACL2 is untyped. Another issue is the embedding of lambda expressions, which is not straightforward because

[1] In fact, models of the CLU logic are only required to satisfy a small set of axioms over equality, $<$, and the successor and predecessor functions. Therefore, CLU could be used to reason about other domains, say strings. Our system allows users to do this by explicitly providing the intended domain.

ACL2 is first-order. We use fresh, lambda-lifted top level ACL2 functions to translate CLU lambda expressions.

We now consider the full UCLID specification language. UCLID models contain a set of state elements, whose behavior is specified with initial and next state functions. The initial and next state functions are defined using CLU expressions extended with syntactic sugar and can also refer to state variables. Notice that the value of a UCLID state variable can be given by a CLU lambda expression. To map UCLID specifications into ACL2, we use the CLU to ACL2 embedding. The resulting ACL2 models have state elements corresponding to the UCLID state elements and for each state element, we define a pair of initial-state and next-state functions. A major problem with the translation is how to handle state elements that are themselves functions or predicates. As ACL2 is a first-order language, the value of a variable cannot be a function. Therefore, we cannot directly translate UCLID lambda expressions to ACL2 functions. The way we handle this is first to closure-convert (Landin, 1964) and lambda-lift (Jouannaud, 1985) the relevant lambda expressions, *i.e.*, we extract the free state variables of each lambda term, and alter the term to take an additional argument, which is an "environment" that packages up their current values. The resulting lambda term that takes an "environment" as input along with its other input arguments is called a closure. Secondly, we perform a defunctionalisation step (Reynolds, 1998) on the resulting closures. That is, we statically know the call sites for each (functional) state variable. Such a call must be to the lambda expression produced by either the state variable's initial-state function, or its next-state function: there are only two choices. Thus, we express the "code" part of the state-element's closure with a closure-converted ACL2 function that can query its extra "environment" argument (which captures the values of the preceding state) to determine if the state is the initial state or a non-initial state. If the former, the code executes the body of the initial-state closure; if the latter, the body of the next-state closure. The result is an *evaluator* function when applied to the arguments of the lambda and the "environment" evaluates to an expression corresponding to the original lambda expression.

4.1. TRANSLATION EXAMPLES

In this section, we show two examples of translations from the UCLID specification language to ACL2. The first example is a part of a module that defines a simple instruction set architecture (ISA). The second example is a part of a UCLID specification.

Part of a model of a simple ISA machine is shown in Figure 3. The model consists of a `DEFINE` section that defines macros and an `ASSIGN` section that describes the initial and next values of state elements. The macros can be

```
MODULE spec
...
DEFINE
  inst := imem0(sPC);
  scond := GetCond(inst);
  is_condition_true :=
        Condit(scond , sNZCV_Flags);
...
  val := case
        MemToReg : ReadData;
        default  : Result;
    esac;
ASSIGN
  init[sPC] := pc0;
  next[sPC] :=
    case
      initi :  pc0;
      project_impl : project_pc;
      (isa & is_ReturnFromException) : sEPC;
      (isa & is_alu_exception) : ALU_Exception_Handler;
      (isa & is_taken_branch) : TargetPC;
      isa : pcadd(sPC);
      default : sPC;
    esac;
...
```

Figure 3. Example of a module defined in the UCLID specification language

thought of as wires in the context of hardware systems. One state element corresponding to a program counter (`sPC`) is shown in the figure. For the state element, an `init` function and a `next` function are defined, which describe the initial and the next value of the state element, respectively. Figure 4 shows the ACL2 specification obtained by translating from the UCLID model.

The UCLID module shown in Figure 3 is translated to two functions `spec-initialize-u` (not shown in the figure due to space limitations) and `spec-simulate-u` (shown in the figure), which are used to compute the initial and next values of state elements in the module, respectively. The macro definitions are sequential, meaning that the definition of a macro can depend on previously defined macros. Therefore the UCLID macros are translated to nested let bindings (`let*`). The `init` and `next` functions of state element `sPC` are translated to the ACL2 functions `initspc-u` and `nextspc-u`, respectively.

The second example (Figure 5) shows the control section of the UCLID specification that describes a formula to be checked for validity. At the beginning of the `EXEC` section all the state elements in the UCLID specification are initialized to values defined by the `init` functions using an implicit ini-

```
(defun initspc-u (pc0) pc0)
(defun nextspc-u
   (initi pc0 project_impl project_pc isa
    is_returnfromexception sepc is_alu_exception
    alu_exception_handler is_taken_branch targetpc spc)
   (cond
    (initi pc0)
    (project_impl project_pc)
    ((and isa is_returnfromexception) sepc)
    ((and isa is_alu_exception) alu_exception_handler)
    ((and isa is_taken_branch) targetpc)
    (isa (pcadd spc))
    (t spc)))
...
(defun spec-simulate-u
   (spec st initi pc0 project_impl project_pc isa
    alu_exception_handler nzcv_flags0 project_nzcv
    epc0 isexception0)
   (let* ((spc (g 'spc spec))
          ...)
     (let* ((inst (imem0 spc))
            (scond (getcond inst))
            (is_condition_true (condit scond snzcv_flags))
            ...
            (val (cond (memtoreg readdata) (t result))))
       (spec-state
           (nextspc-u
              initi pc0 project_impl project_pc
              isa is_returnfromexception sepc
              is_alu_exception alu_exception_handler
              is_taken_branch targetpc spc)
           ...)
```

Figure 4. Translation to ACL2 of the UCLID module shown in Figure 3.

tialize command. The first command is a symbolic simulation that assigns all state elements to the values described by their `next` functions. After some computation, the `Good_MA_V` and the `Rank_W` variables are assigned CLU expressions. Finally, the formula to be checked for validity is given in the `decide` command.

The translation of the UCLID control section to ACL2 is shown in Figure 6. The whole control section is translated to a `defthm` construct, the construct used to state theorems in ACL2. The implicit UCLID initialize command at the beginning of the `EXEC` is translated to the `initialize-u` function. The UCLID simulate command is translated to the `simulate-u` function. The resulting states of the model obtained after initialization and simulation are stored using `let*` bindings and are used for further simulation

```
CONTROL
...
EXEC
simulate(1);
...
Good_MA_V :=  ( Equiv_MA_0 | Equiv_MA_1 | Equiv_MA_2 | ...);
...
Rank_W := Rank;
...
decide(
      Good_MA_V &
      ((~((S_pc0  = I_pc0 ) &
           (S_rf0 (a1) = I_rf0 (a1)) &
           (S_dmem0(a1)  = I_dmem0(a1))
...);
```

Figure 5. Example of a control section in UCLID.

steps. The `Good_MA_V` and the `Rank_W` variables are assigned values in the `let*` bindings. Finally, the formula to be checked is given.

5. Refinement Proof

In this section, we describe in detail the proof that the seven-stage bit-level interface pipelined machine model (MB) refines its executable instruction set architecture (IE). Both models MB and IE are described using the ACL2 programming language. We use the combined system obtained from integrating UCLID with ACL2 for the proof. We first give an outline of the proof.

5.1. Proof Overview

An outline of the proof that MB refines IE, both of which are defined in ACL2, is shown in Figure 7. We make essential use of the compositionality of WEB-refinement to reduce the proof to a sequence of simpler refinement proofs.

In the ACL2 models IE and MB, memory is modeled using association lists mapping addresses to data values. In the other models, memory is modeled using evaluator functions, because as described in Section 4, this allows us to relate ACL2 models with UCLID models, where memory is modeled using restricted lambda expressions.

The first refinement proof shows that MB refines MM, a bit-level interface pipelined machine model that is similar to MB except that MM's memories are modeled using evaluator functions. The second refinement proof is used to move from bit-vectors to integers. We do this by proving (with ACL2) that MM refines ME, an executable pipelined machine that is similar to MM, but which operates on integers, not bit-vectors.

```
(defthm web_core-u
  (implies
   (and (integerp pc0)  (integerp epc0) ...)
     (let*
       ((st0 (initialize-u nil nil nil nil nil pc0 ...))
        (st1 (simulate-u st0 bpstate0 nil nil nil pc0 ...))
        ...
        (good_ma_v
          (or ... (or equiv_ma_0 equiv_ma_1)
              equiv_ma_2 ...)
        ...
        (rank_w
          (rank-u (g 'mwwrt (g 'impl st21))
                  zero
                  (g 'mmwrt (g 'impl st21))
                  (g 'emwrt (g 'impl st21))
                  (g 'dewrt (g 'impl st21))
                  (g 'fdwrt (g 'impl st21))
                  (g 'ffwrt (g 'impl st21))))
        ...)
       (and good_ma_v
         (or ...
             (and (equal s_pc0 i_pc0)
                  (equal (read-srf-u a1 s_st0)
                         (read-prf-u a1 i_st0)))
                  (equal (read-sdmem-u a1 s_st0)
                         (read-pdmemhist_2-u a1 i_st0)))
        ...)
```

Figure 6. Translation of the UCLID control section shown in Figure 5 to ACL2.

Recall that our goal is to move towards refinement steps that can be handled by UCLID, and, as mentioned previously, this requires that we "pollute" the models by adding extra inputs and logic in order to state the "core" refinement theorems. The refinement step from ME to MEP, a polluted version of ME, does exactly this.

The pipeline is dealt with next, when MEP is shown to refine IEP, a polluted version of IE. As we will see shortly, both ACL2 and UCLID are used for this refinement proof.

What remains is to show that IEP refines IM, which can be thought of as a purification step that removes the pollution introduced earlier, and that IM refines IE, which transforms the memory models based on lambda expressions in IM to the association lists based memory models in IE.

The proof that MEP refines IEP cannot be directly handled with UCLID, *e.g.*, the models use arithmetic operations on integers that are not expressible in the CLU logic or the UCLID specification language. Therefore, several abstractions are employed, resulting in machines MA and IA which abstract

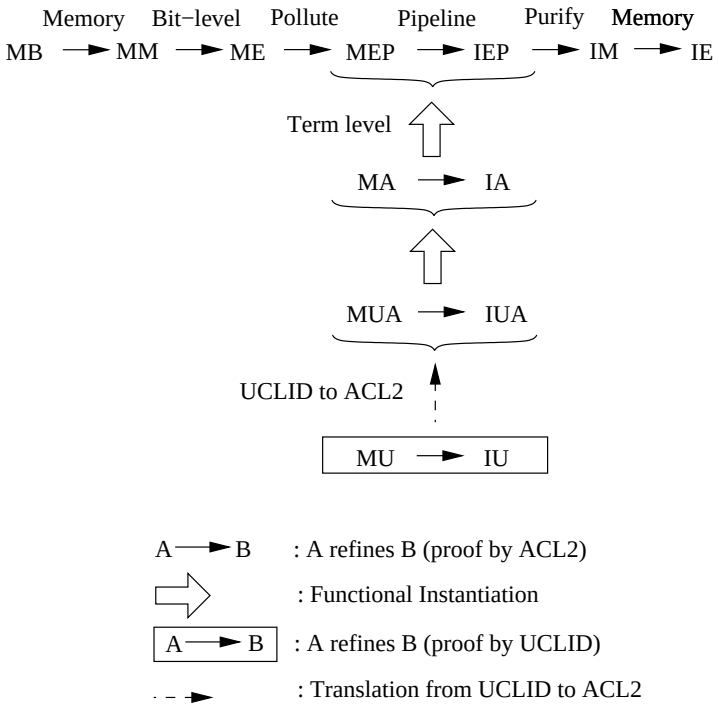

Figure 7. Proof outline that uses ACL2 and UCLID to show that MB refines IE.

MEP and IEP, respectively. MA and IA are term-level models and we prove that MA refines IA by translating this proof obligation to a UCLID theorem stating that MU refines IU. MU and IU are the UCLID analogs of MA and IA, respectively. MUA and IUA are obtained by translating MU and IU to ACL2 using our trusted translator. The UCLID theorem that MU refines IU is also translated to ACL2 using our trusted translator resulting in an ACL2 theorem stating that MUA refines IUA, which is then used to prove that MA refines IA. Several of the ACL2 refinement proofs use functional instantiation, an ACL2 proof technique that allows one to lift theorems involving constrained functions to theorems involving functions satisfying the constraints. This is how we use UCLID proofs, which contain UFs and UPs, to prove theorems about defined functions and predicates.

We now describe in detail aspects of the refinement proof. Note that the various models are very large (for example, the MA model is about 2,500 lines of ACL2 code) and given the limited space, it is not possible to fully describe these models.

5.2. MEMORY MODELS

In this section, we describe the parts of the refinement proof that relate the machine models MB and IE with machine models MM and IM, respectively. MB and IE use association lists to model memory whereas MM and IM use the evaluator function to model memory.

Typically, memories are thought of as arrays of data values. In ACL2, we use association lists (lists of key/value pairs that are also referred to as alists) to model memories using *set* to write into a memory location and *get* to read the value of a memory location. The data memory, the instruction memory, and the register file in machine models MB and IE are modeled using alists.

In UCLID, memories are modeled using restricted lambda expressions, which are essentially functions that map addresses to data values. The lambda expressions are restricted so that they only take integers as arguments and there is no way to express recursive definitions. A read from memory is an application of the lambda expression corresponding to the memory to the address, and a memory update results in a new lambda expression corresponding to the new memory.

Lambda expressions in UCLID are mapped in ACL2 to evaluator functions and an "environment" variable as follows. Memory is modeled as a list *st* of elements containing an address, data, and a number of other fields that we call the condition fields. The condition fields together are used to define the memory update condition and memory is updated only if the memory update condition holds. During every step of the machine, the list *st* is updated with the address, the data value, and the condition fields. The evaluator function is used to read memory. It takes an address and the list *st* as input and returns the data corresponding to the address, which is defined to be the first write to the address where the update condition holds. If no such write exists, the initial value is returned.

In order to illustrate how the evaluator function can be used to model memories, we give a simple example. Consider a register file, `rf`, in a machine that is updated every time the signal `rfupdate` is **true**. First, we define the "environment" as the list `st`, whose every element is a list that has the fields `addr`, `data`, and `rfupdate` corresponding with the register file address, the data to be written to the register file, and the signal `rfupdate`, respectively. The list `st` is updated every time the machine model is stepped. The evaluator function for the register file `rf` is defined as shown in Figure 8.

In Figure 8, `car` and `cdr` return the first element of a list, and the rest of a list, respectively. The functions `addr-st`, `data-st`, and `rfupdate-st` are functions that return the fields `addr`, `data`, and `rfupdate` of an element of `st`, respectively. The `evalrf` function returns the data field of the first element in `st`, whose addr field is equal to `addr` and whose `rfupdate` field is **true**. If no such element is found, `evalrf` returns the initial value

```
(defun evalrf (addr st)
  (if (endp st)
      (rf0 addr)
    (if (and (rfupdate-st (car st))
             (equal (addr-st (car st)) addr))
        (data-st (car st))
      (evalrf addr (cdr st)))))
```

Figure 8. ACL2 code of the evaluator function for the register file example.

given by an Uninterpreted Function (`rf0`). Notice that `st` is like the register file, except that every element has an additional field `rfupdate`; also, `evalrf` is the read function for the register file.

For the proof that IM refines IE, the refinement map maps all the states in IM that are not memories to corresponding states in IE. Since memories in IM are modeled using evaluators, the refinement map maps memories in IM to alists in IE by starting with an empty alist and updating it using the address and data fields of every element in the IM memory with a **true** memory update condition. IM and IE are very similar in structure and do not stutter with respect to each other. Therefore, we actually prove that IM and IE are bisimilar.

To prove that MB refines MM we prove that MB and MM are bisimilar by defining a refinement map from MM to MB that is similar to the refinement map from IM to IE.

5.3. Reasoning about Bit-Level Interface Designs

In this section we describe the part of the refinement proof that relates the bit-level pipelined machine model MM to ME, a pipelined machine operating on integers. This proof is carried out exclusively using ACL2 and is parameterized with respect to the word size, *i.e.*, our proof remains the same regardless of the word size of the machines involved. Since MM and ME do not stutter with respect to each other, we prove that the two systems are bisimilar.

The refinement map from MM to ME converts unsigned and signed bit-vectors in MM to naturals and integers, respectively. For the proof, we developed a bit-vector library in ACL2. For example, we defined and developed a theory of rules for functions to convert bit-vectors to numbers and vice-versa. The functions include *n-ubv* (which converts naturals to unsigned bit-vectors), *ubv-n* (which converts unsigned bit-vectors to naturals), *i-sbv* (which converts integers to signed bit-vectors), and *sbv-i* (which converts signed bit-vectors to integers). The library required about four days for an expert ACL2

user to develop. For the refinement proof, we required theorems such as the following.

1. $natp(a) \wedge natp(n) \wedge len(n\text{-}ubv(a)) \leq n$
   $\Rightarrow$ $ubv\text{-}n(extend\text{-}n(n\text{-}ubv(a), n)) = a$
2. $integerp(a) \wedge natp(n) \wedge len(i\text{-}sbv(a)) \leq n$
   $\Rightarrow$ $sbv\text{-}i(sign\text{-}extend\text{-}n(i\text{-}sbv(a), n)) = a$
3. $bvp(x) \wedge natp(a) \wedge (a < len(x)) \Rightarrow bitp(nth(a, x))$
4. $bvp(a) \wedge bvp(b) \wedge (len(a) = len(b))$
   $\Rightarrow$ $(ubv\text{-}n(a) = ubv\text{-}n(b)) \Leftrightarrow (a = b)$
5. $bvp(a) \wedge bvp(b) \wedge (len(a) = len(b))$
   $\Rightarrow$ $(sbv\text{-}i(a) = sbv\text{-}i(b)) \Leftrightarrow (a = b)$

In the above theorems, $len(x)$ is the length of the bit-vector x, $natp(a)$ denotes that a is a natural number, $integerp(a)$ denotes that a is an integer, $bvp(a)$ denotes that a is a bit-vector, $bitp(a)$ denotes that a is a bit, $nth(n, x)$ corresponds to the n^{th} element of list x, $extend\text{-}n(b, n)$ extends the unsigned bit-vector b to a length of n, and $sign\text{-}extend\text{-}n(b, n)$ sign extends the signed bit-vector b to a length of n. Theorems 1, 2, 4, and 5 are used to reason about the refinement map and Theorem 3 is useful for reasoning about the instruction decoder, which generates control signals from the bit-vector corresponding to instructions. The theorems described above were essential for our proof, but our proofs required many other theorems all of which are included in our bit-vector library. Also, the bit-vector library was developed based on what was required for the refinement proof, and can be easily extended with more bit-vector operations and rules.

5.4. Pollution and Purification of Models

Due to the limited expressiveness of the UCLID specification language, to define refinement maps, we have to modify (pollute) the machine models by adding external inputs, logic, and history variables (variables that record previous values of state elements). That theorems about polluted models imply something about the original models requires proof.

Refinement maps are used to map implementation states to specification states. We use the commitment refinement map for this purpose (Manolios, 2000; Manolios and Srinivasan, 2004a), where a pipelined machine state is related to an instruction set architecture state by invalidating all the partially executed instructions in the pipeline and rolling back the programmer-visible components so that they correspond with the last committed instruction. To define the refinement map, two functions are required. One is the commitment function that commits the pipelined machine state and the other is the

projection function that projects the programmer visible components of the pipelined machine state to the ISA state.

The pipelined machine model can essentially be thought of as a function that describes the operational semantics of the pipelined machine. Both the pipelined machine model and the commitment function are two different functions that modify the state of the pipelined machine. Using the UCLID specification language, it is not possible to define two different functions, both of which modify the same state elements. Therefore, to implement the commitment function, we modify the pipelined machine model by adding an external input *commit_impl*, history variables—variables that record the history of some of the state elements—and some extra logic resulting in what we call the polluted pipelined machine model. When the input *commit_impl* is **false**, the polluted pipelined machine model corresponds to the operational semantics of the pipelined machine, and when *commit_impl* is **true**, the polluted pipelined machine model corresponds to the commitment function. Similarly, we modify the ISA model by adding an input *project_impl* and some control logic to implement the projection function.

It is not clear that proving the polluted pipelined machine model is correct implies that the original pipelined machine model is correct. It is possible that an external input or a history variable modifies the normal operation of the pipelined machine and hides a bug that exists in the original machine. For example, consider a buggy variant of the seven-stage model that does not stall the program counter when the pipeline is stalled. To define the commitment refinement map for the seven-stage pipelined machine model, we use a history variable *stallp* that records the value of the stall signal from the previous step of the machine. We abstract and pollute the pipelined machine model so that we can define the commitment refinement map. In the process of pollution, we stall the program counter if the next value of *stallp* is **true**. Now, we can prove using UCLID that the polluted pipelined machine model refines the ISA model. But, in fact, this does not tell us anything about the original pipelined machine model because the polluted model uses the *stallp* history variable in the program counter logic, whereas the original model has no history variables.

Therefore, we check in ACL2 that if the external inputs in a polluted pipelined machine model are set to values corresponding to the operational semantics of the pipelined machine, then the unpolluted executable model (ME) refines the polluted executable model (MEP). Similarly, for the purification step, we check that the polluted executable ISA model (IEP) refines the purified ISA model (IE). ME and MEP do not stutter with respect to each other and neither do IE and IEP. Therefore, we can prove a bisimulation result. The bug in the program counter logic of the buggy variant of the seven-stage model will be caught using our method when we try to prove that the original pipelined machine model refines the polluted model.

5.5. Relating Executable Models and Term-Level Models

In this section, we give an overview of the proof that MEP refines IEP. This refinement step deals with the pipeline and uses UCLID. However, in order to use UCLID, we have to show a relationship between executable machines and term-level machines. The difficulty is in mechanically verifying the various abstractions employed, which are used to deal with memories, branch prediction, instruction classes, etc. Below we describe two abstraction techniques that are very hard to mechanically verify. Both these abstraction techniques– one for memories and the other for branch predictors– are widely used in term-level modeling.

Memories in UCLID can be modeled using lambda expressions and such memories can be matched with ACL2 memories as described in Section 5.2. However, in cases where reads and writes are in order—*e.g.*, this is the case for the data memory of our machine—memory can be modeled as an integer variable using two UFs, one to read and one to write. This modeling style leads to faster verification times than the approach using lambdas (Lahiri et al., 2002). However, it is much more difficult to use if the abstraction has to be mechanically verified. To mechanically verify this abstraction, we have to encode the memory state as an integer and define the read and write operations for this encoding of the memory, in order to obtain our executable model. This is possible using Gödel encoding scheme, as shown below.

$$((a_1 \ . \ d_1) \ (a_2 \ . \ d_2) \ \ldots \ (a_n \ . \ d_n)) \ \rightarrow \ p_1^{p_3^{a_1+1} p_4^{a_2+1} \ldots p_{n+2}^{a_n+1}} \ p_2^{p_3^{d_1+1} p_4^{d_2+1} \ldots p_{n+2}^{d_n+1}}$$

In the above equation, the data memory is an alist whose address elements are a_1, $a_2, \ldots, a_n$ and whose data elements are d_1, $d_2, \ldots, d_n$. The i^{th} prime is denoted p_i. Any finite memory can now be represented as a single integer, but there are several problems with the above approach. For example, the theorem proving effort required to show that this scheme works is non-trivial, *e.g.*, it requires that we prove the prime decomposition theorem. In addition, the above encoding scheme cannot be used for infinite memories, as there is no bijection between the set of infinite memories and the natural numbers. Therefore, we find that the time savings attained by abstracting the data memory with an integer are not worth the added theorem proving effort required to justify this abstraction.

Branch predictors in UCLID can also be modeled using an integer variable that represents the state of the branch predictor and three UFs that take the branch predictor state as input and return the next state of the branch predictor, a prediction for the branch direction, and a prediction for the branch target (Lahiri et al., 2002). To show that the above correctly abstracts an executable implementation, for example a Branch Target Buffer (BTB), we

are required to model the environment of the BTB using an integer. However, since the next state of the BTB depends on the entire processor state, we have to encode the state of the processor with one integer. We can do this using Gödel encoding schemes, as described above, provided the memories are finite, but the effort required would be considerable. Therefore, we use an alternate abstraction, where we simply model the branch predictor choices using non-determinism. Justifying this abstraction is straightforward, thus the ACL2 verification effort is drastically simplified. In addition, the UCLID verification times are comparable to the verification times required by the standard approach.

A final abstraction that we briefly mention concerns the instruction set. The UCLID models only have one instruction per instruction class, whereas the executable models have the full instruction set. This turned out to be surprisingly easy to deal with because the UCLID models abstract the instructions by using uninterpreted functions (UFs) that take the opcode as argument and collapse the instructions corresponding to various values of the opcode to one instruction. When we instantiate the UCLID model, we replace the uninterpreted functions that take the opcode as input with functions that check the value of the opcode and perform the appropriate operation. For example, the UCLID model only has one ALU operation, but the executable model first checks the opcode to determine whether it is an add or a subtract etc., and then performs the appropriate operation.

Executable models have other advantages. We can use them to debug designs more easily. For example, using UCLID counterexamples one can determine the sequence of instructions that leads to a bug, but it might be more difficult to determine what the actual bug is. Using our executable models, we can use test inputs that simulate the same sequence of instructions to track the bug in the design. Note that counterexamples generated by UCLID correspond to counterexamples for the refinement relation between ACL2 models MUA and IUA, and for the refinement relation between ACL2 models MA and IA.

Executable models also allow proofs of properties that might not be theorems in the abstract models. In addition, while the refinement proof established that the pipelined machine model (MB) behaves like the instruction set architecture model (IE), how do we know that the instruction set architecture model (IE) is correct? Executable models allows us to run test programs. In our case, while executing a simple program, we found three bugs in the instruction set architecture model (IE), which are described below.

– Instructions are 32 bits with the least significant bit and the most significant bit corresponding to the 0^{th} bit and the 31^{st} bit of the instruction, respectively. The bug was in the functions that implement the instruction decoder, which were reading the 32-bit instruction in the reverse order.

For example, if a decoder function was supposed to read the 5^{th} bit, it was instead reading the 26^{th} bit (31 - 5) of the instruction.

– The register file is updated by both ALU and load instructions. A decoder function that takes the instruction as input is used to determine if that instruction updates the register file. The function was buggy in that it did not signal that the register file should be updated if the input was a load instruction.

– The processor has 4 flags that are used to store various properties of the result obtained from the previous instruction. For example, if the previous instruction was an ALU instruction whose result was zero, then the *Z* flag is set. The *update_nzcv* function that takes the result and the previous value of the flags is used to update the processor flags. The *update_nzcv* was buggy in that the two input arguments to the function were swapped.

Since the decoder functions and the *update_nzcv* function are abstracted using uninterpreted functions (UFs) in the term-level models, none of the above bugs could have been caught during the verification of the term-level models using UCLID.

The bugs described above bring up an important aspect of automatic term-level verification using decision procedures such as UCLID. Recall that in order to use such methods, one must abstract away the ALU, the decoding logic, etc. using UFs shared by both the MA and ISA. While these abstractions drastically reduce the complexity of the verification problem, they also lead to ISA models that are structurally similar to MA models. MA models tend to have next state relations for each of the components of the MA machine and this way of specifying the MA model makes sense because they are inherently parallel machines whose every component is continuously updated. ISA models defined in UCLID tend to have the same structure as their corresponding MA models. This is what allows them to share the same UFs as their MA models, but it also is what makes it easy to mask the kinds of errors reported above. Notice, however, that ISA models are inherently sequential and, conceptually, the simplest way to define them is to just have a big case statement that checks the type of the next instruction and executes code corresponding to the semantics of this instruction. If we define ISA models in this way, we have a much better chance of catching errors, as the semantic gap between the MA and ISA models is now larger. Using our approach, we can in fact define such an ISA machine and can prove that it is refined by IE.

MA and IA are term-level models, and we are finally at the point where we can invoke UCLID, which is optimized to automatically and efficiently reason about such models. MA, IA, and the refinement theorem that relates these models are translated to the UCLID specification language. The resulting UCLID models are MU and IU. UCLID proves the refinement theorem and our (trusted) translator returns an equivalent ACL2 theorem, now about the models generated by our translator, IUA and MUA. Using functional instantiation, as outlined previously, ACL2 is able to complete the proof automatically.

6. Verification Statistics

The verification times for the proofs and the expert user effort required in terms of man-weeks for each intermediate step in the refinement proof is shown in Table I. In the "Proof Step" column in the table, A $\rightarrow$ B means that system A refines system B. For all the proof steps, except MU $\rightarrow$ IU, we used the ACL2 theorem proving system (version 2.9). For MU $\rightarrow$ IU, we used the UCLID decision procedure (version 1.0) coupled with the siege SAT solver (Ryan, 2004) (variant 4). All the experiments were run on a 3.06 GHz Intel Xeon machine, with a cache size of 512 KB. The user effort required for the proof steps is an estimate of the effort that would be required for an expert user of both the UCLID tool and the ACL2 theorem proving system to apply this verification approach to verify another pipelined machine design of similar complexity. The times reported above do not include the time required to learn UCLID and ACL2 and do not include the time required for the integration, which took several months.

7. Reasoning about Programs

An advantage of executable models over abstract models is that we are able to reason about programs running on pipelined machines and even about compilers that generate code for pipelined machines. We describe a simple example in ACL2 that demonstrates our ability to use the executability of our pipelined machine model and the refinement theorem that relates the pipelined machine to its instruction set architecture to efficiently reason about programs running on the pipelined machine model, something that significantly extends the kind of analysis one can perform when restricted to using term-level models.

The program that we consider is one that solves the Knapsack problem, a commonly arising optimization problem. We have a knapsack with capacity

Table I. Verification times and expert user effort required for the refinement proofs.

Proof Step	Proof Time (secs)	User Effort (man-weeks)
MU → IU	84	3
MA → IA	3	2
MEP → IEP	5	2
IEP → IM	4	1
IM → IE	90	1
ME → MEP	2	2
MM → ME	233	3
MB → MM	12	1

```
K(0) := 0
for c = 1 to T
  max := 0
  for j = 1 to n
    if C(j) ≤ c
      x := K(c-C(j)) + V(j)
      if x > max
        max := x
  K(c) := max
return K(T)
```

Figure 9. Pseudo code for solving the Knapsack problem.

`T` and a set of `n` items, each of which has a cost, `C(·)`, and a value, `V(·)`, associated with it. The value of the knapsack is the sum of the values of the items in it, where we allow multiple instances of the same item. Similarly, the cost of the knapsack is the sum of the costs of the items in it. What is the maximum value our knapsack can attain without exceeding its capacity? A dynamic programming solution to the knapsack problem, in pseudo-code, is shown in Figure 9.

The assembly-level program and the machine code program of the Knapsack problem for the bit-level interface pipelined machine model MB is shown in Table II. To show that the program works correctly, we are required to prove the property that `K(T)` is the maximum value achievable with a knapsack of capacity `T`.

Using ACL2, we can prove that the machine code for MB satisfies the correctness property of the Knapsack solution. As we have seen, MB is a complex bit-level pipelined machine with branch prediction, forwarding logic,

Table II. Assembly-level program and machine code for the Knapsack problem.

Assembly Code	Machine Code	Assembly Code (Cont.)	Machine Code (Cont.)
storei r1 0	3886092288	add r11 r11 r13	3768299533
movi r6 0	3815792640	movi r0 20	3815768084
addi r6 r6 1	3800457217	sub r9 r11 r7	3780874247
movi r10 0	3815809024	bn r0	1249902592
movi r7 0	3815796736	mov r7 r11	3787157515
add r14 r3 r10	3767787530	movi r0 5	3815768069
add r15 r4 r10	3767857162	sub r11 r5 r10	3780489226
addi r10 r10 1	3800735745	bnz r0	444596224
load r12 r14	3854483470	add r11 r1 r6	3767644166
load r13 r15	3854553103	store r11 r7	3853234183
movi r0 20	3815768084	movi r0 2	3815768066
sub r11 r6 r12	3780554764	sub r11 r2 r6	3780292614
bn r0	1249902592	bnz r0	444596224
add r11 r11 r1	3768299521	add r11 r1 r2	3767644162
load r11 r11	3854282763	load r9 r11	3854274571

stalls etc. This makes it difficult to reason about even simple programs executing on MB. It is much simpler to show the correctness of programs running on IE, the high-level non-pipelined model. Our theory of refinement allows us to do exactly this, but notice that the preservation of liveness plays a crucial role, *e.g.*, were we to use a notion of refinement that did not preserve liveness, then a proof that the program runs correctly on IE does not rule out the possibility of livelock on MB.

Having reduced the problem of reasoning about code running on MB to code running on IE, the final concern is how to reason about code running on IE. To prove partial correctness, at the very least we are required to define a sufficient collection of program invariants. In fact, a method that requires only this is due to Moore (Moore, 2003), who shows how to use ACL2 to automatically generate the verification conditions required to show that the program invariants imply partial correctness. There are also extensions that allows us to prove total correctness results (Matthews and Vroon, 2004).

8. Related Work

Previous work on pipelined machine verification can be roughly classified into automatic approaches based on decision procedures and approaches that

use deductive reasoning. An early work on the use of automatic decision procedures was by Burch and Dill who showed how to automatically compute the abstraction function using flushing (Burch and Dill, 1994) and gave a decision procedure for the logic of uninterpreted functions with equality and boolean connectives. Another, more efficient decision procedure was given in (Bryant et al., 1999) that exploits positive equality. The work was further extended in (Bryant et al., 2002), where a decision procedure for the CLU logic that exploits optimized encoding schemes (Seshia et al., 2003b) is given. The decision procedure is implemented in UCLID, which has been used to verify out-of-order microprocessors (Lahiri et al., 2002) and which we use to verify the models presented in this paper. Recently, there is interest in abstracting bit-level designs to UCLID specifications, but these methods only work on very simple examples and have some severe limitations, *e.g.*, they do not handle memories (Andraus and Sakallah, 2004). We also expect that recent advances in decision procedures (Ganzinger et al., 2004; de Moura, 2005) will drastically reduce the verification times of term-level pipelined machine models.

An early, pioneering body of work on the use of theorem proving for the verification of microprocessors is the CLI stack work (Hunt, 1989; Hunt, 1994; Bevier et al., 1989). Another notable use of theorem proving in the context of hardware verification used ACL2 to reason about Motorola's CAP digital signal processor (Brock and Hunt, 1997). Sawada and Hunt have used the ACL2 theorem proving system to verify the FM9801 Microarchitecture. Their work is based on computing an intermediate abstraction of the pipelined machine state called MAETT that keeps track of completed and in-flight instructions. Using the MAETT abstraction, they check that each of the instructions in the pipeline executes correctly. Hosabettu et al., (Hosabettu et al., 1998; Hosabettu et al., 1999) use the PVS theorem prover to verify pipelined processors. Their work is based on the use of completion functions that specifies the effect of completing an instruction in the pipeline on the programmer visible components. The abstraction function is computed by using a composition of completion functions, one for every partially executed instruction in the pipeline. Arons and Pnueli (Arons and Pnueli, 2000) have also used the PVS theorem prover to verify a machine with speculative instruction execution. In (Kroning, 2001), data consistency and liveness of pipelined machine models is verified using the PVS theorem prover. The models are synthesizable and are described very close to the gate-level.

The notion of correctness for pipelined machines that we use was first proposed in (Manolios, 2000), and is based on WEB-refinement (Manolios, 2001). The first proofs of correctness for pipelined machines based on WEB-refinement were carried out using the ACL2 theorem proving system (Kaufmann et al., 2000b; Kaufmann and Moore, 2004). The advantage of using a theory of refinement over using the Burch and Dill notion of correctness,

even if augmented by a "liveness" criterion, is that deadlock may avoid detection with the Burch and Dill approach (Manolios, 2000), whereas it follows directly from the WEB-refinement approach that deadlock (or any other liveness problem) is ruled out. In (Manolios and Srinivasan, 2004a), it is shown how to automatically verify safety *and liveness* properties of pipelined machines using WEB-refinement.

9. Conclusions and Future Work

We have shown how to verify executable pipelined machine models with bit-level interfaces using our integration of the UCLID decision procedure with the ACL2 theorem proving system. This has allowed us to overcome the major limitation of approaches based on decision procedures, namely that they only work for abstract term-level models and do not provide a firm connection with RTL models. Theorem proving approaches can reason about RTL-level designs, but tend to require heroic human effort. With our approach, the proof required only minutes of CPU time and the human theorem proving effort required was modest. Our proofs are based on WEB-refinement, a theory of refinement that is compositional and preserves both safety and liveness properties. We also demonstrated that we can decompose the proof that code running on the pipelined machine is correct by first showing that the pipelined machine refines the instruction set architecture and then showing that the software running on the instruction set architecture is correct. For future work, we plan to apply this approach to a wider class of pipelined machines and to determine what other domains can benefit from our work.

References

Andraus, Z. S. and K. A. Sakallah: 2004, 'Automatic Abstraction and Verification of Verilog Models'. In: S. Malik, L. Fix, and A. B. Kahng (eds.): *Design Automation Conference–DAC'04*. pp. 218–223.

Arons, T. and A. Pnueli: 2000, 'A Comparison of Two Verification Methods for Speculative Instruction Execution'. In: *Tools and Algorithms for the Construction and Analysis of Systems–TACAS'00*, Vol. 1785 of *Lecture Notes in Computer Science*. pp. 487–502.

Bentley, B.: 2001, 'Validating the Intel Pentium 4 microprocessor'. In: *38th Design Automation Conference–DAC'01*. pp. 253–255.

Bentley, B.: 2005, 'Validating a Modern Microprocessor'. See URL `http://-www.cav2005.inf.ed.ac.uk/bentley_CAV_07_08_2005.ppt`.

Bevier, W. R., W. A. Hunt, Jr., J. S. Moore, and W. D. Young: 1989, 'An Approach to Systems Verification'. *Journal of Automated Reasoning* **5**(4), 411–428.

Boyer, R. S. and J. S. Moore: 1988, 'Integrating Decision Procedures Into Heuristic Theorem Provers: A Case Study of Linear Arithmetic'. In: *Machine intelligence 11*. Oxford University Press, Inc., pp. 83–124.

Brock, B. and W. A. Hunt, Jr.: 1997, 'Formally Specifying and Mechanically Verifying Programs for the Motorola Complex Arithmetic Processor DSP'. In: *1997 IEEE International Conference on Computer Design*. pp. 31–36.

Browne, M., E. M. Clarke, and O. Grumberg: 1988, 'Characterizing Finite Kripke Structures in Propositional Temporal Logic'. *Theoretical Computer Science* **59**.

Bryant, R. E., S. German, and M. N. Velev: 1999, 'Exploiting Positive Equality in a Logic of Equality with Uninterpreted Functions'. In: N. Halbwachs and D. Peled (eds.): *Computer-Aided Verification–CAV'99*, Vol. 1633 of *LNCS*. pp. 470–482.

Bryant, R. E., S. K. Lahiri, and S. Seshia: 2002, 'Modeling and Verifying Systems using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions'. In: E. Brinksma and K. Larsen (eds.): *Computer-Aided Verification–CAV'02*, Vol. 2404 of *LNCS*. pp. 78–92.

Burch, J. R. and D. L. Dill: 1994, 'Automatic Verification of Pipelined Microprocessor Control'. In: *Computer-Aided Verification–CAV'94*, Vol. 818 of *LNCS*. pp. 68–80.

Clark, L., E. Hoffman, J. Miller, M. Biyani, Y. Liao, S. Strazdus, M.Morrow, K. Velarde, and M. Yarch: 2001, 'An Embedded 32-bit Microprocessor Core for Low-Power and High-Performance Applications'. *IEEE Journal of Solid-State Circuits* **36**(11), 1599–1608.

de Moura, L.: 2005, 'Yices Homepage'. See URL `http://fm.csl.sri.com/yices`.

Ganzinger, H., G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli: 2004, 'DPLL(T): Fast Decision Procedures'. In: R. Alur and D. Peled (eds.): *Computer Aided Verification–CAV'04*, Vol. 3114 of *LNCS*. pp. 175–188.

Greve, D., R. Richards, and M. Wilding: 2004, 'A Summary of Intrinsic Partitioning Verification'. In: *Fifth International Workshop on the ACL2 Theorem Prover and Its Applications*.

Greve, D., M. Wilding, and D. Hardin: 2000, 'High-Speed, Analyzable Simulators'. in (Kaufmann et al., 2000a), pp. 113–135.

Hosabettu, R., M. Srivas, and G. Gopalakrishnan: 1998, 'Decomposing the Proof of Correctness of a Pileplined Microprocessors'. In: A. J. Hu and M. Y. Vardi (eds.): *Computer-Aided Verification–CAV'98*, Vol. 1427 of *LNCS*.

Hosabettu, R., M. Srivas, and G. Gopalakrishnan: 1999, 'Proof of Correctness of a Processor with Reorder Buffer Using the Completion Functions Approach'. In: N. Halbwachs and D. Peled (eds.): *Computer-Aided Verification–CAV'99*, Vol. 1633 of *LNCS*.

Hunt, Jr., W. A.: 1989, 'Microprocessor Design Verification'. *Journal of Automated Reasoning* **5**(4), 429–460.

Hunt, Jr., W. A.: 1994, *FM8501: A Verified Microprocessor*, Vol. 795. Springer-Verlag.

Jouannaud, J.-P. (ed.): 1985, 'Functional Programming Languages and Computer Architecture', No. 201. Nancy, France:.

Kaufmann, M., P. Manolios, and J. S. Moore (eds.): 2000a, *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers.

Kaufmann, M., P. Manolios, and J. S. Moore: 2000b, *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers.

Kaufmann, M. and J. S. Moore: 2004, 'ACL2 homepage'. See URL `http://www.cs.utexas.edu/users/moore/acl2`.

Kroning, D.: 2001, 'Formal Verification of Pipelined Microprocessors'. Ph.D. thesis, Universität des Saarlandes.

Lahiri, S., S. Seshia, and R. Bryant: 2002, 'Modeling and Verification of Out-of-order Microprocessors using UCLID'. In: *Formal Methods in Computer-Aided Design–FMCAD'02*, Vol. 2517 of *LNCS*. pp. 142–159.

Lahiri, S. K. and S. Seshia: 2004, 'The UCLID Decision Procedure'. In: *Computer Aided Verification–CAV'04*, Vol. 3114 of *LNCS*. pp. 475–478.

Landin, P. J.: 1964, 'The Mechanical Evaluation of Expressions'. *The Computer Journal* **6**(4), 308–320.

Manolios, P.: 2000, 'Correctness of Pipelined Machines'. In: W. A. Hunt, Jr. and S. D. Johnson (eds.): *Formal Methods in Computer-Aided Design–FMCAD'00*, Vol. 1954 of *LNCS*. pp. 161–178.

Manolios, P.: 2001, 'Mechanical Verification of Reactive Systems'. Ph.D. thesis, University of Texas at Austin. See URL `http://www.cc.gatech.edu/~manolios/-publications.html`.

Manolios, P.: 2003, 'A Compositional Theory of Refinement for Branching Time'. In: D. Geist and E. Tronci (eds.): *12th IFIP WG 10.5 Advanced Research Working Conference–CHARME'03*, Vol. 2860 of *LNCS*. pp. 304–318.

Manolios, P. and S. Srinivasan: 2004a, 'Automatic Verification of Safety and Liveness for XScale-Like Processor Models Using WEB-Refinements'. In: *Design Automation and Test in Europe–DATE'04*. pp. 168–175.

Manolios, P. and S. Srinivasan: 2004b, 'A Suite of Hard ACL2 Theorems Arising in Refinement-Based Processor Verification'. In: M. Kaufmann and J. S. Moore (eds.): *Fifth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2004)*. See URL `http://www.cs.utexas.edu/users/moore/acl2/-workshop-2004/`.

Manolios, P. and S. Srinivasan: 2005a, 'A Complete Compositional Reasoning Framework for the Efficient Verification of Pipelined Machines'. In: *International Conference on Computer-Aided Design–ICCAD'05*. pp. 863–870.

Manolios, P. and S. Srinivasan: 2005b, 'Refinement Maps for Efficient Verification of Processor Models'. In: *Design Automation and Test in Europe–DATE'05*. pp. 1304–1309.

Manolios, P. and S. Srinivasan: 2005c, 'Verification of Executable Pipelined Machines with Bit-Level Interfaces'. In: *International Conference on Computer-Aided Design–ICCAD'05*. pp. 855–862.

Matthews, J. and D. Vroon: 2004, 'Partial Clock Functions in ACL2'. In: M. Kaufmann and J. S. Moore (eds.): *Fifth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2004)*. See URL `http://www.cs.utexas.edu/users/-moore/acl2/workshop-2004/`.

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

Moore, J. S.: 2003, 'Inductive Assertions and Operational Semantics'. In: *Advanced Research Working Conference on Correct Hardware Design and Verification Methods–CHARME'03*, Vol. 2860 of *Lecture Notes in Computer Science*. pp. 289–303.

Namjoshi, K. S.: 1997, 'A Simple Characterization of Stuttering Bisimulation'. In: *17th Conference on Foundations of Software Technology and Theoretical Computer Science*, Vol. 1346 of *LNCS*. pp. 284–296.

Reynolds, J. C.: 1998, 'Definitional Interpreters for Higher-Order Programming Languages'. *Higher-Order and Symbolic Computation* **11**(4), 363–397. Reprinted from the proceedings of the 25th ACM National Conference (1972), with a foreword.

Russinoff, D. M.: 1998, 'A Mechanically Checked Proof of IEEE Compliance of a Register-Transfer-Level Specification of the AMD-K7 Floating-Point Multiplication, Division, and Square Root Instructions'. *London Mathematical Society Journal of Computation and Mathematics* **1**, 148–200.

Russinoff, D. M.: 1999, 'A Mechanically Checked Proof of Correctness of the AMD-K5 Floating-Point Square Root Microcode'. *Formal Methods in System Design* **14**, 75–125.

Ryan, L.: 2004, 'Siege homepage'. See URL `http://www.cs.sfu.ca/ ~loryan/-personal`.

Sawada, J.: 2002, 'Formal Verification of Divide and Square Root Algorithms using Series Calculation'. In: M. Kaufmann and J. S. Moore (eds.): *Proceedings of the ACL2 Workshop 2002*.

Semiconductor Industry Association: 2004, 'International Technology Roadmap for Semiconductors'. See URL `http://public.itrs.net/`.

Seshia, S., S. Lahiri, and R. Bryant: 2003a, 'A User's Guide to UCLID version 1.0'. See URL `http://www.cs.cmu.edu/ uclid/userguide.ps`.

Seshia, S. A., S. K. Lahiri, and R. E. Bryant: 2003b, 'A Hybrid SAT-Based Decision Procedure for Separation Logic with Uninterpreted Functions'. In: *Design Automation Conference–DAC'03*. pp. 425–430.

Smith, S., R. Perez, S. Weingart, and V. Austel: 1999, 'Validating a High-Performance, Programmable Secure Coprocessor'. In: *22nd National Information Systems Security Conference*.