Verification of Executable Pipelined Machines with Bit-Level Interfaces

Panagiotis Manolios
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30318
Email: manolios@cc.gatech.edu

Sudarshan K. Srinivasan
School of Electrical & Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia 30318
Email: darshan@ece.gatech.edu

***Abstract*— We show how to verify pipelined machine models with bit-level interfaces by using a combination of deductive reasoning and decision procedures. While decision procedures such as those implemented in UCLID can be used to verify pipelined machines, the models are at the *term level*: they abstract away the datapath, require the use of numerous abstractions, implement a small subset of the instruction set, and are far from executable. In contrast, we focus on verifying executable machines with bit-level interfaces. Such proofs have previously required substantial expert guidance and the use of deductive reasoning engines. 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. In this way, we exploit the strengths of ACL2 and UCLID to prove theorems that are not possible to even state using UCLID and that would require prohibitively more effort using just ACL2.**

I. Introduction

Successful approaches to pipelined machines verification can be roughly classified as being based on the use of theorem provers or decision procedures. Theorem proving systems such as ACL2 [14] have been used to reason about pipelined machine models at various levels of abstraction, ranging from the term-level to bit- and cycle-accurate models, but they typically require extensive expert user support. Approaches based on decision procedures such as UCLID [4], [17] are fast and highly automated but are restricted to term-level models, which employ numerous abstractions and are far from being executable, let alone bit- and cycle-accurate.

We describe an approach to bit-level pipelined machine verification that uses the tool UCLID to reason about term-level models and the theorem prover ACL2 to relate the term-level models to bit-level models and to establish the correctness of the various abstractions used in the term-level models. To this end, we have integrated the UCLID decision procedure with ACL2, and have used the combined system to show that an executable complex pipelined machine model, mostly defined at the bit-level, refines its instruction set architecture. The proof requires minimal expert user support compared to previous approaches and the verification times are in the order of minutes.

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

Our proofs are based on WEB-refinement, a theory of refinement that is compositional and preserves safety and liveness properties [20]. Essential use is made of both these features of WEB-refinement. Compositionality allows us to reduce the proof that the bit-level machine refines its instruction set architecture into several refinement steps, some of which are handled using ACL2 and some of which are handled using UCLID. That WEB-refinement preserves both safety *and liveness* is used to prove that code running on the pipelined machine is correct, by first proving that the pipelined machine refines the instruction set architecture and then proving that the software running on the instruction set—not on the pipelined machine—is correct.

We chose to use ACL2 because it is an industrial-strength mechanical theorem prover that has been successfully used for hardware verification. Some of ACL2's commercial applications include floating-point unit verification of the AMD-K5 processor [25], AMD Athlon^{TM} processor [26], and IBM Power4^{TM} processor [29]. The verification of separation properties in Rockwell avionics microprocessors [7], and verification of an IBM secure co-processor [32] also used ACL2.

We combined ACL2 with UCLID [4], [17] because UCLID implements a decision procedure for formulas expressed in a decidable fragment of first order logic called CLU, which has been shown to be capable of describing and verifying pipelined machines at the term level. The CLU logic contains the boolean connectives, uninterpreted functions, equality, counter arithmetic, ordering, and restricted lambda expressions.

For problems expressible in CLU, UCLID tends to drastically outperform ACL2, *e.g.*, to complete the proof of correctness of a simple five-stage DLX pipeline defined at the term-level, UCLID took about 3 seconds, while ACL2 required $15\frac{1}{2}$ days [22]. Unfortunately, as we now outline, UCLID also has some severe limitations, which is why we need the power of a theorem proving system such as ACL2. Since UCLID models are defined at the term-level, they are not executable. In contrast, ACL2 can be made to simulate processors at close to C speeds [8]. In addition, term-level models generally contain only one instruction per instruction class and do not capture the semantics of the instruction set architecture, *e.g.*,

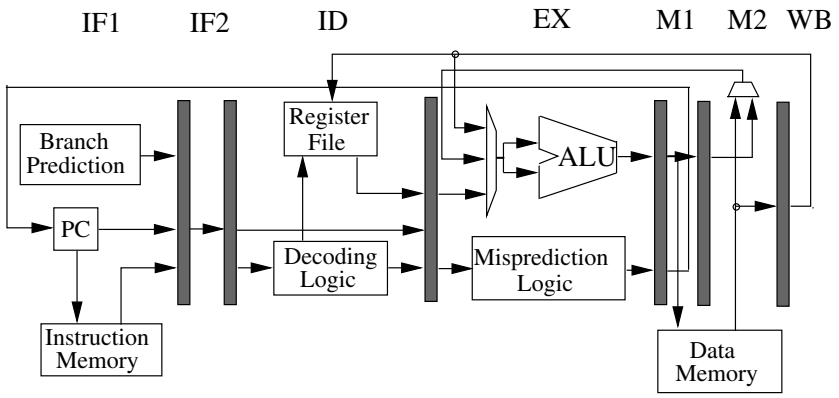

Fig. 1. High-level organization of bit-level interface processor model

ALU instructions are treated as uninterpreted functions. This makes it impossible to reason about software, *e.g.*, to prove any interesting theorems about machine code, we need to know that an add instruction adds. ACL2 has none of these restrictions, so we can reason about machine code running on the pipelined machine, as we show later in this paper. Another major difference between the two systems is that ACL2 is far more expressive than the UCLID specification language. In fact, UCLID cannot even state the refinement theorem. We can state the "core" of the refinement theorem, but even then we have to drastically modify the UCLID models, *e.g.*, by adding external inputs and extra state and combinational logic (giving rise to what we term "polluted models"). It is not at all clear that the correctness proofs involving the polluted models imply the correctness of the unpolluted models. As we show in the sequel, by using the expressive power of ACL2, this problem can be avoided.

The paper is organized as follows. In Section II, we describe the executable pipelined machine model. In Section III, we describe the notion of correctness based on refinement that we use for checking the pipelined machine model. In Section IV, we give an overview of our integration of UCLID with ACL2; a more detailed description will appear elsewhere. Section V-A describes in detail the refinement-based proof of correctness of the bit-level machine using the combined system obtained from integrating UCLID with ACL2. Section VI gives the verification statistics of the proof in terms of the running time and expert user effort required. In Section VII, we demonstrate the ability of ACL2 to use the refinement theorem to efficiently reason about programs compiled for the pipelined machine model. We describe related work in Section VIII and conclude in Section IX.

II. Processor Model

The pipelined machine model we use is inspired by the Intel XScale architecture [6] and is shown in Figure 1. The model is described using the ACL2 programming language and can execute assembly-level programs. In Section VII, we show an example program (a dynamic programming solution to the Knapsack problem) that executes on the model. The model is defined at the bit-level, except for the register file, the instruction and data memories, and some combinational blocks such as the ALU, which have bit-level interfaces, *i.e.*, they are defined as functions whose inputs and outputs are bit vectors, but are otherwise unconstrained. This is why we use the term "bit-level interface" to describe the model.

The model has the following 7 pipeline stages: a 2-cycle fetch, a decode, an execute, a 2-cycle memory access, and a write back. It has features such as branch prediction, ALU exceptions, and predicated instruction execution. It implements 16 ALU instructions, 15 branch instructions, various jump, load, and store instructions, and a return from exception instruction. Each ALU instruction can be conditionally executed based on 16 different conditions. Note that these are actual instructions that can be executed by the model. The model implements both register-register and register-immediate addressing modes. Instructions are bit-vectors of size 32 and there are 16 registers. The word size is a parameter that can be set to any positive integer; the size selected does not affect the verification times. In the bit-level interface model, the processor control logic and the data path logic are defined to operate on bit-vectors, except for some combinational blocks such the ALU. The ALU takes bit-vector inputs, converts them to integers, performs the appropriate ALU operation on the integers, and converts the result back to a bit-vector.

III. Refinement

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.

What does it mean for the MA to refine its ISA? It 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. Of course, reasoning about infinite paths is difficult to automate, and in [20], 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.

The above notion of refinement is *compositional* and a complete compositional reasoning framework based our notion of refinement is given in [23]. For example, one can prove the following theorem, where $r; q$ denotes functional composition, *i.e.*, $(r; q)(s) = q(r(s))$.

Theorem 1: (Composition)
If $\mathcal{M} \approx_r \mathcal{M}'$ and $\mathcal{M}' \approx_q \mathcal{M}''$ then $\mathcal{M} \approx_{r;q} \mathcal{M}''$.

In [21], it is shown how to automate the proof of WEB-refinement in the context of pipelined machine verification. The idea is to strengthen, thereby simplifying, the WEB-refinement proof obligation; the result is a CLU-expressible formula that guarantees that the two machines satisfy the same formulas expressible in the temporal logic $\mathrm{CTL}^* \setminus \mathsf{X}$, over the state components visible at the instruction set architecture level. $\mathrm{CTL}^* \setminus \mathsf{X}$ is a very expressive temporal logic, allowing one to express both safety and liveness properties. The CLU-expressible formula that implies WEB-refinement follows, where *rank* is a function that maps states of MA into the natural numbers.

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

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 is the safety component, and the proof obligation that $\mathit{rank}(v) < \mathit{rank}(w)$ is the liveness component.

IV. Integrating UCLID with ACL2

Our integration of the UCLID decision procedure with ACL2 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 [2].

We initially considered building a system that given an ACL2 conjecture generates a corresponding UCLID specification, which is then handed to the UCLID decision procedure,

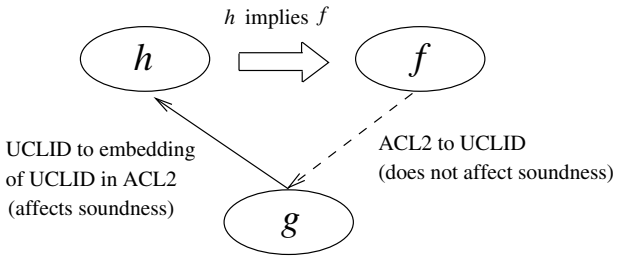

Fig. 2. An overview of our integration of UCLID with ACL2. To prove f, an ACL2 theorem, we generate, using any translator (not necessarily trusted), g, a UCLID statement. If UCLID proves g, then we generate an equivalent ACL2 theorem, h, using a trusted translator, and prove in ACL2 that h implies f.

but this approach has several problems. Since the ACL2 language is more expressive than CLU, there is no general way to define the translation. This means that any such attempt will have to use abstractions, but the right abstractions will most likely be problem-dependent. We can imagine adding an analysis phase that tries to find good abstractions, but besides not being complete, this will complicate the translation process, thereby increasing the possibility of soundness errors. For these reasons, we decided to build a system that allows users to define their own translators, but which cannot be rendered unsound by the use of erroneous translators.

The approach we use is depicted in Figure 2. If f is the ACL2 conjecture to be checked, we first translate it to a UCLID specification, g. We have defined such a translator that is suitable for our needs, but any user-provided translator will do. Recall that the soundness of our system does not depend on this translation step. If UCLID claims that g is valid, then we assert an equivalent ACL2 statement, say h. We have a method of embedding the UCLID logic in ACL2, and can therefore translate any UCLID theorem to an equivalent ACL2 theorem. This translation step does affects soundness. It is what formally connects ACL2 and UCLID, but this translation is fixed and completely general. What remains is to show, with ACL2, that h implies f.

Our approach is general in that it can handle all of the UCLID constructs described in this paper. In addition, we have limited the sources of unsoundness. That is, only our embedding of UCLID into ACL2 has to be trusted; faulty user provided translators cannot render our system unsound. To see this, note that the proof of f depends only on the validity of g and the soundness of our UCLID embedding in ACL2—if we assume that ACL2 and UCLID are sound. Let's consider three illustrative scenarios. First, suppose that f is not valid and it correctly gets translated to g. Then UCLID will produce a counterexample which can be used to debug the ACL2 model. Second, suppose that f is not valid, but the user provided translator incorrectly generates a valid formula, g. In this case, our translator will generate an equivalent ACL2 formula h, but ACL2 will not be able to prove $h \Rightarrow f$, as it is not true. Finally, suppose that f is valid, but the (untrusted) translation incorrectly generates g. If g is not valid, then UCLID will generate a counterexample, but the user will determine that

the counterexample is spurious, indicating that their translator is faulty. If g is valid, then UCLID will prove it so and our translator will provide an equivalent ACL2 theorem h, but h will be of limited use in proving f. Of course, since f is valid, $h \Rightarrow f$ is also valid, and ACL2 may be able to prove this regardless, but for the examples we consider, the ACL2 proof will take a very long time.

We now give a high-level description of our embedding of the CLU logic and the UCLID specification language in ACL2. The full details of the embedding are rather technical and will be presented elsewhere. The CLU syntax and semantics and the UCLID specification language are described in [4], and [30], 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 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 which 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 disciple, while ACL2 is untyped. Another issue is the embedding of lambda expressions, which is not straight-forward because 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 states elements, each of which has an *initial* and a *next* "state function," which specify the possible behaviors of the system. The initial and next-state functions are defined using CLU expressions which can now 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. The 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 way we handle this is first to closure-convert [18] and lambda-lift [12] the relevant lambda expressions: we extract the free state variables of each lambda term, and alter the term to take an additional argument that packages up their current values. Secondly, we perform a defunctionalisation step [24] 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.

[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.

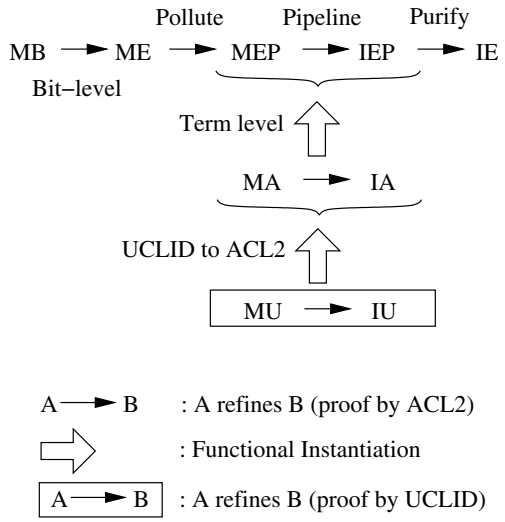

Fig. 3. Proof outline that uses ACL2 and UCLID to show that MB refines IE.

V. Refinement Proof

We now describe the proof that bit-level interface pipelined machine (MB) refines the executable instruction set architecture (IE). Both machines are defined in ACL2 and the proof is carried out using our system, which integrates UCLID with ACL2. We start with an overview of the proof first.

A. Proof Overview

An outline of the proof that MB refines IE, both of which are defined in ACL2, is shown in Figure 3. We make essential use of the compositionality of WEB-refinement to reduce the proof to a sequence of simpler refinement proofs. The first refinement proof is used to move from bit-vectors to integers. We do this by proving with ACL2 that MB refines ME, an executable pipelined machine that is similar to MB, but which

operates on integers, not bit vectors. Now, 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 IE, which can be thought of as a purification step that removes the pollution introduced earlier.

The proof that MEP refines IEP cannot be directly handled with UCLID, *e.g.*, the models are executable. Therefore, several abstractions are employed, resulting in machines MA and IA which abstract MEP and IEP, respectively. MA and IA are term-level models and we prove that MA refines IA, using our system as outlined in Figure 2. The proof that MA refines IA corresponds to f, the ACL2 theorem to be proved. We use our ACL2 to UCLID (untrusted) translator to generate a UCLID theorem, g, which states that MU refines IU. MU and IU are the UCLID analogs of MA and IA, respectively. Once UCLID proves this theorem, we generate with our (trusted) translation from UCLID to ACL2, the corresponding ACL2 theorem, h, and prove that h implies f. The ACL2 proofs that one refinement step implies another use functional instantiation, a proof technique supported by ACL2 that allows us one 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.

B. Reasoning about Bit-Level Interface Designs

In this section we describe the first part of the refinement proof, where we show that the pipelined, bit-level machine MB refines 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. MB and ME are very similar in structure and do not stutter with respect to each other. Therefore, we can in fact prove that the two systems are bisimilar.

The refinement map from MB to ME converts unsigned and signed bit-vectors in MB to naturals and integers. For the proof, we developed a bit-vector library in ACL2 based. For example, we defined and developed a theory of rules for functions to convert bit-vectors to numbers and vice-versa. The functions include $n\text{-}ubv$ (which converts naturals to unsigned bit-vectors), $ubv\text{-}n$ (which converts unsigned bit-vectors to naturals), $i\text{-}sbv$ (which converts integers to signed bit-vectors), and $sbv\text{-}i$ (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. $natp(a) \wedge natp(n) \wedge len(n\text{-}ubv(a)) \leq n$
   $\Rightarrow \quad ubv\text{-}n(extend\text{-}n(n\text{-}ubv(a), n)) = a$
2. $integerp(a) \wedge natp(n) \wedge len(i\text{-}sbv(a)) \leq n$
   $\Rightarrow \quad sbv\text{-}i(sign\text{-}extend\text{-}n(i\text{-}sbv(a), n)) = a$
3. $bvp(x) \wedge natp(a) \wedge (a < len(x)) \quad \Rightarrow \quad bitp(nth(a, x))$

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 and 2 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.

C. 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 and logic. That proofs 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 [19], [21], 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 main 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. To define the commitment and the projection functions in UCLID, we use two inputs, $commit\_impl$, which is an input to the pipelined machine model and $project\_impl$, which is an input to the ISA model. The inputs are used to modify the logic of the models to define the refinement maps.

It is not clear that proving the modified processor model is correct implies that the original processor model is correct. For example, it is possible that external input modifies the normal operation of the pipelined machine and hides a bug that exists in the original machine. Therefore, we check in ACL2 that if the external inputs in a polluted pipelined machine model are set to values for normal operation, 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.

D. 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.

Memories in UCLID can be modeled using lambda expressions and such memories can be matched with ACL2 memories rather straightforwardly. 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 [16]. However, it is much more difficult to use if the abstraction has to be mechanically verified. To mechanically verify this abstraction, we have to show how to instantiate the UFs corresponding to read and write operations, in order to obtain our executable model. This requires the use of a 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 [16]. 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 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 straight-forward, 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 use the opcode as an argument; when we instantiate the UCLID model, we use a function that checks the opcode and performs the appropriate action. For example, the UCLID model only has one ALU operation, but the executable model first checks the opcode and then performs the appropriate operation.

We end by pointing out that executable models have other advantages. We can use them to more easily understand UCLID counterexamples, which can be thousands of lines and are quite difficult to understand. Also, while the refinement proof established that the pipelined machine behaves like the instruction set, how do we know that the instruction set is correct? Executable models allows us to run test programs. In our case, while executing a simple program, we found a bug in the instruction decoder. The decoder was reading the 32-bit instructions in the reverse order.

E. Abstract Models

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. UCLID proves the refinement theorem and our (trusted) translator returns an equivalent ACL2 theorem, now about the models generated by our translator, IU and MU. Using functional instantiation, as outline previously, ACL2 is able to complete the proof automatically.

VI. Verification Statistics

The proof times 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 [27] (variant 4). All the experiments were run on a 3.06 GHz Intel Xeon, 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. 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.

VII. Program Verification

Unlike abstract models, executable models describe the semantics of instructions, which can be used to reason about

Proof Step	Proof Time (secs)	User Effort (man-weeks)
MU → IU	157	3
MA → IA	91	2
MEP → IEP	36	2
IEP → IE	4	1
ME → MEP	21	2
MB → ME	182	3

TABLE I

VERIFICATION TIMES AND EXPERT USER EFFORT REQUIRED FOR THE REFINEMENT PROOFS.

```
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)
```

Fig. 4. Pseudo code for solving the Knapsack problem.

Assembly Code	Machine Code
storei r1 0	3886026752
movi r6 0	3818938368
addi r6 r6 1	3800457217
movi r10 0	3818954752
movi r7 0	3818942464
mov r14 r3	3787710467
mov r15 r4	3787780100
addi r10 r10 1	3800735745
load r12 r14	3854483470
load r13 r15	3854553103
movi r0 21	3818913813
sub r11 r6 r12	3779506188
bn r0	1249902592
add r11 r11 r1	3767250945
load r11 r11	3854282763
add r11 r11 r13	3767250957
movi r0 21	3818913813
sub r9 r11 r7	3779825671
bn r0	1249902592
mov r7 r11	3788206091
movi r0 7	3818913799
sub r11 r5 r10	3779440650
bnz r0	0444596224
add r11 r1 r6	3766595590
store r11 r7	3852972039
movi r0 2	3818913794
sub r11 r2 r6	3779244038
bnz r0	0444596224
add r11 r1 r2	3766595586
load r9 r2	3853684738

TABLE II

ASSEMBLY-LEVEL PROGRAM AND MACHINE CODE FOR THE KNAPSACK PROBLEM.

programs and compilers for the pipelined machines. For example, one can prove the correctness of a program compiled for the machine or show the correctness of compiler optimizations. We describe a simple example in ACL2 to demonstrate the ability to use the executability of the pipelined machine model and the refinement theorem that relates the pipelined machine to its instruction set architecture to efficiently reason about programs, which cannot be done with UCLID models as they are not executable.

The program that we consider is one that solves the Knapsack problem, a commonly arising optimization problem. We have a knapsack with capacity `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 with exceeding its capacity? A dynamic programming solution to the knapsack problem, in pseudo-code, is shown in Figure 4.

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 require to prove the property that `K(T)` is the maximum value achievable with a knapsack of capacity `T`. We can prove using ACL2 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, stalls etc., and it is 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.

VIII. RELATED WORK

We now selectively review previous work on pipelined machine verification that is directly related to our work. Burch and Dill showed how to automatically compute the abstraction function using flushing [5] and gave a decision procedure for the logic of uninterpreted functions with equality and boolean connectives. Another, more efficient decision procedure was given in [3]. The work was further extended in [4], where a decision procedure for the CLU logic that exploits optimized encoding schemes [31] is given. The decision procedure is implemented in UCLID, which has been used to verify out-of-order microprocessors [16] and which we use to verify the models presented in this paper. An early, pioneering body of work on the use of theorem proving for the verification of microprocessors is the CLI stack work [10], [11], [1]. More recent theorem proving approaches include [28], [9]. The notion of correctness for pipelined machines that we use was first proposed in [19], and is based on WEB-refinement [20].

The first proofs of correctness for pipelined machines based on WEB-refinement were carried out using the ACL2 theorem proving system [14], [15]. 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 [19], whereas it follows directly from the WEB-refinement approach that deadlock (or any other liveness problem) is ruled out. In [21], it is shown how to automatically verify safety *and liveness* properties of pipelined machines using WEB-refinement.

IX. 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. The proof was completed in a few minutes of CPU time and required minimal expert user support. The 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. In this way, we exploit the strengths of ACL2 and UCLID and establish proofs that are not possible to prove using UCLID and that would require considerably more effort using just ACL2. For future work, we plan to apply this approach to a wider class of pipelined machines.

References

[1] W. R. Bevier, W. A. Hunt, Jr., J. S. Moore, and W. D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, December 1989.

[2] R. S. Boyer and J. S. Moore. Integrating decision procedures into heuristic theorem provers: a case study of linear arithmetic. In *Machine intelligence 11*, pages 83–124. Oxford University Press, Inc., 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] R. E. Bryant, S. K. Lahiri, and S. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In E. Brinksma and K. Larsen, editors, *Computer-Aided Verification–CAV 2002*, volume 2404 of *LNCS*, pages 78–92. Springer-Verlag, 2002.

[5] 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.

[6] L. Clark, E. Hoffman, J. Miller, M. Biyani, Y. Liao, S. Strazdus, M.Morrow, K. Velarde, and M. Yarch. An embedded 32-bit microprocessor core for low-power and high-performance applications. *IEEE Journal of Solid-State Circuits*, 36(11):1599–1608, 2001.

[7] D. Greve, R. Richards, and M. Wilding. A summary of intrinsic partitioning verification. In *Fifth International Workshop on the ACL2 Theorem Prover and Its Applications*, 2004.

[8] D. Greve, M. Wilding, and D. Hardin. High-speed, analyzable simulators. In Kaufmann et al. [13], pages 113–135.

[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] W. A. Hunt, Jr. Microprocessor design verification. *Journal of Automated Reasoning*, 5(4):429–460, 1989.

[11] W. A. Hunt, Jr. *FM8501: A Verified Microprocessor*, volume 795. Springer-Verlag, 1994.

[12] J.-P. Jouannaud, editor. *Functional Programming Languages and Computer Architecture*, number 201, Nancy, France, Sept. 1985.

[13] M. Kaufmann, P. Manolios, and J. S. Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, June 2000.

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

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

[16] S. Lahiri, S. Seshia, and R. Bryant. Modeling and verification of out-of-order microprocessors using UCLID. In *Formal Methods in Computer-Aided Design (FMCAD'02)*, volume 2517 of *LNCS*, pages 142–159. Springer-Verlag, 2002.

[17] S. K. Lahiri and S. Seshia. The UCLID decision procedure. In *Computer Aided Verification, CAV'04*, volume 3114 of *LNCS*, pages 475–478, July 2004.

[18] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[19] P. Manolios. Correctness of pipelined machines. In W. A. Hunt, Jr. and S. D. Johnson, editors, *Formal Methods in Computer-Aided Design–FMCAD 2000*, volume 1954 of *LNCS*, pages 161–178. Springer-Verlag, 2000.

[20] P. Manolios. *Mechanical Verification of Reactive Systems*. PhD thesis, University of Texas at Austin, August 2001. See URL `http://-www.cc.gatech.edu/~manolios/publications.html`.

[21] P. Manolios and S. Srinivasan. Automatic verification of safety and liveness for XScale-like processor models using WEB-refinements. In *Design Automation and Test in Europe, DATE'04*, pages 168–175, 2004.

[22] P. Manolios and S. Srinivasan. A suite of hard ACL2 theorems arising in refinement-based processor verification. In M. Kaufmann and J. S. Moore, editors, *Fifth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2004)*, November 2004. See URL `http://www.cs.utexas.edu/users/-moore/acl2/workshop-2004/`.

[23] P. Manolios and S. Srinivasan. A complete compositional reasoning framework for the efficient verification of pipelined machines. In *ICCAD-2005, International Conference on Computer-Aided Design*, 2005. to appear.

[24] J. C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998. Reprinted from the proceedings of the 25th ACM National Conference (1972), with a foreword.

[25] D. M. Russinoff. A mechanically checked proof of correctness of the AMD-K5 floating-point square root microcode. *Formal Methods in System Design*, 14:75–125, 1999.

[26] D. M. Russinoff and A. Flatau. RTL verification: A floating-point multiplier. In Kaufmann et al. [13], pages 201–231.

[27] L. Ryan. Siege homepage. See URL `http://www.cs.sfu.ca/~loryan/personal`.

[28] 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/`.

[29] J. Sawada. Formal verification of divide and square root algorithms using series calculation. In M. Kaufmann and J. S. Moore, editors, *Proceedings of the ACL2 Workshop 2002*. 2002.

[30] S. Seshia, S. Lahiri, and R. Bryant. A user's guide to uclid version 1.0, 2003. See URL `http://www.cs.cmu.edu/ uclid/-userguide.ps`.

[31] S. A. Seshia, S. K. Lahiri, and R. E. Bryant. A hybrid SAT-based decision procedure for separation logic with uninterpreted functions. In *Design Automation Conference (DAC 03)*, pages 425–430, 2003.

[32] S. Smith, R. Perez, S. Weingart, and V. Austel. Validating a high-performance, programmable secure coprocessor. In *22nd National Information Systems Security Conference*, Oct. 1999.