Term-Level Verification of a Pipelined CISC Microprocessor

Randal E. Bryant

December, 2005

CMU-CS-05-195

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This work was supported by the Semiconductor Research Corporation under contract 2002-TJ-1029.

Keywords: Formal verification, Microprocessor verification, UCLID

Abstract

By abstracting the details of the data representations and operations in a microprocessor, *term-level verification* can formally prove that a pipelined microprocessor faithfully implements its sequential, instruction-set architecture specification. Previous efforts in this area have focused on reduced instruction set computer (RISC) and very-large instruction word (VLIW) processors.

This work reports on the verification of a complex instruction set computer (CISC) processor styled after the Intel IA32 instruction set using the UCLID term-level verifier. Unlike many case studies for term-level verification, this processor was not designed specifically for formal verification. In addition, most of the control logic was given in a simplified hardware description language. We developed a methodology in which the control logic is translated into UCLID format automatically, and the pipelined processor and the sequential reference version were described with as much modularity as possible. The latter feature was made especially difficult by UCLID's limited support for modularity.

A key objective of this case study was to understand the strengths and weaknesses of UCLID for describing hardware designs and for supporting the formal verification process. Although ultimately successful, we identified several ways in which UCLID could be improved.

1 Introduction

This report describes a case study in using the UCLID verifier to formally verify several versions of the Y86 pipelined microprocessor presented in Bryant and O'Hallaron's computer systems textbook [4]. The purpose of this exercise was 1) to make sure the designs are actually correct, and 2) to evaluate the strengths and weaknesses of UCLID for modeling and verifying an actual hardware design. We ultimately succeeded in this effort, finding and correcting one serious bug in one version. We were able to prove both the safety and the liveness of our designs.

1.1 Background

Microprocessors have a succinct specifications of their intended behavior, given by their Instruction Set Architecture (ISA) models. The ISA describes the effect of each instruction on the microprocessor's *architectural state*, comprising its registers, the program counter (PC), and the memory. Such a specification is based on a sequential model of processing, where instructions are executed in strict, sequential order.

Most microprocessor implementations use forms of pipelining to enhance performance, overlapping the execution of multiple instructions. Various forms of interlocking and data forwarding are used to ensure that the pipelined execution faithfully implements the sequential semantics of the ISA. The task of formal microprocessor verification is prove that this semantic relationship indeed holds. That is, for any possible instruction sequence, the microprocessor will obtain the same result as would a purely sequential implementation of the ISA model.

Although the development of techniques for formally verification microprocessors has a history dating back over 20 years [6], the key ideas used in our verification effort are based on ideas developed by Burch and Dill in 1994 [5]. The main effort in their approach is to prove that there is some abstraction function α mapping states of the microprocessor to architectural states, such that this mapping is maintained by each cycle of processor operation. Burch and Dill's key contribution was to show that this abstraction function could be computed automatically by symbolically simulating the microprocessor as it *flushes* instructions out of the pipeline. For a single-issue microprocessor, the verification task becomes one of proving the equivalence of two symbolic simulations: one in which the pipeline is flushed and then a single instruction is executed in the ISA model, and the other in which the pipeline operates for a normal cycle and then flushes. We call this approach to verification *correspondence checking*.

Burch and Dill also demonstrated the value of *term-level modeling* for their style of microprocessor verification. With term-level modeling, the details of data representations and operations are abstracted away, viewing data values as symbolic *terms*. The precise functionality of operations of units such as the instruction decoders and the ALU are abstracted away as *uninterpreted functions*. Even such parameters as the number of program registers, the number of memory words, and the widths of different data types are abstracted away. These abstractions allow the verifier to focus its efforts on the complexities of the pipeline control logic. Although these abstractions had long been used when applying automatic theorem provers to hardware verification [6, 12], Burch and Dill were the first to show their use in an automated microprocessor verification tool.

Our research group followed Burch and Dill's lead, developing microprocessor verification tools that operated at both with bit-level [14] and term-level [15] representations. Even with term-level modeling, our tools operate by reducing the equivalence condition to a Boolean satisfiability problem, and then applying either Binary Decision Diagrams [1] or a SAT solver [17] to prove the equivalence or to generate a counterexample in the case of inequivalence. These earlier tools were restricted to the logic of *Equality with Uninterpreted Functions and Memories*, where the only operations on terms were to test them for equality and to apply uninterpreted functions, and where memories were modeled with interpreted read and write operations. The tools were also customized to the specific task of verifying microprocessors using correspondence checking.

Burch-Dill verification proves the *safety* of a pipelined processor design—that every cycle of processor operation has an effect consistent with some number of steps k of the ISA model. This includes the case where $k = 0$, i.e., that the cycle did not cause any progress in the program execution. This is indeed a possibility with our designs, when the pipeline stalls to deal with a hazard condition, or when some instructions are canceled due to a mispredicted branch. This implies, however, that a processor that deadlocks can pass the verification. In fact, a device that does absolutely nothing will pass.

To complete the verification, we must also verify *liveness*—that the processor cannot get in a state where it never makes forward progress. This issue has not been addressed in most microprocessor verification efforts, due to a perception that such bugs are highly unlikely and that this form of verification would be difficult. Velev [13] is the only published account of a liveness check using automated verification techniques. His approach involves proving a modified version of the correctness statement, stating that when the pipeline operating for n cycles has behavior consistent with completing k steps of the ISA model, where n is chosen large enough to be sure that $k > 0$. In this report, we describe a very simple and effective approach to proving liveness.

1.2 UCLID

We developed the UCLID verifier to generalize both the modeling capabilities and the range of verification tasks that the we can perform [3] at the term level. UCLID extends the underlying logic to support a limited set of integer operations, namely addition by a constant and ordered comparisons. It replaces the specialized read and write functions for modeling memories with a more general ability to define functions with a simple lambda notation. The combination of integer operations and lambda notation makes it possible to express a wide variety of memory structures, including random-access, content-addressed, queues, and various hybrids. It can also be used to model systems containing arrays of identical processes and program interpreters.

UCLID supports several different types of verification, including bounded property checking, correspondence checking, invariant checking [8] and automatic predicate abstraction [7].

The input to UCLID [11] consists of the description of a system in the UCLID modeling language, followed by a series of symbolic simulation commands to perform the actual verification. When UCLID determines that a verification condition does not hold, it generates a counterexample trace, showing an execution of the model that violates the condition.

Correspondence checking is implemented in UCLID using a combination of model construction and command script. The model consists of both the pipelined processor and a reference version that directly implements the ISA. External control signals direct which of these two components are active, and allows the state from the architectural state elements of the pipeline to be copied over to the corresponding state elements in the reference version. The command script performs two runs of the system and then invokes a decision procedure to check the final verification condition stating the conditions under which the two runs should yield identical results.

1.3 Project Objectives

An important priority for this study was to better understand the applicability of term-level modeling in general, and UCLID in particular, to hardware verification. Most uses of UCLID to date have been to verify models that were specifically constructed with UCLID's modeling in mind and for the purpose of demonstrating UCLID's abilities. By contrast, the Y86 microprocessor was designed without formal verification in mind, and hence presents a more realistic case study. Y86 also has some attributes of a complex instruction set computer (CISC) instruction set, including a byte-oriented instruction encoding and greater use of the stack for implementing procedure calls and returns. This requires a more detailed model than do the highly stylized reduced instruction set computer (RISC) processors that have been previously verified with term-level verification tools.

The Bryant and O'Hallaron textbook presents two different approaches to implementing a Y86 processor. The SEQ implementation corresponds directly to the ISA model. It executes one complete instruction per clock cycle. The PIPE implementation, of which there are seven slightly different versions, uses a 5-stage, single-issue pipeline. Having these two implementations makes our verification task easy to formulate: determine whether or not SEQ and (all seven versions of) PIPE are functionally equivalent. The task is further simplified by the fact that the two implementations share many functional elements, such as the instruction decoding logic and the ALU. They differ only in the additional pipeline registers and the control logic required by PIPE.

The control logic for both SEQ and PIPE are described in a simplified hardware description language, called *HCL*, for "Hardware Control Language." Translators had previously been written from HCL to C to construct simulation models of the two processors, as well as from HCL to Verilog to construct versions suitable for generating actual implementations by logic synthesis.

Our previous experience with formal verification tools has shown us that maintaining *model fidelity*—the assurance that the model being verified is a faithful representation of the actual design and that it is being verified over its full range of operation—is surprisingly difficult. The numerous cycles of editing the model, running the verifier, and analyzing the results often lead to errors being introduced that cause inexplicable verification errors. When there is a manual translation between the original design description and the verification model, the two descriptions can easily become inconsistent. Even worse, it is common to restrict the range of operation (e.g., by limiting the set of initial states) in order to isolate bugs. If these restrictions are not later removed, the verifier could overlook bugs that occur under more general conditions. This runs totally contrary to the aims of formal verification.

Given this background, we formulated the following objectives for this case study:

- **Modularity**
  - A common description of the function blocks should be created and then instantiated in the models for the two different processor designs.
  - We should be able to construct separate verification files for SEQ and PIPE, so that properties of the two designs can be verified directly rather than only by correspondence checking.
- **Automation and Version Control**
  - The HCL control logic should be translated automatically into UCLID format.
  - A systematic way should be created to automatically assemble the different parts of the model—the functional blocks, the control logic, the two data paths, and the connecting logic—into a single verification model.
  - Any restricted operation of the processor model should be expressed as antecedents to the verification condition, localizing the restriction to a single point in the input file. There should be no alterations of the initial state expressions for the state elements.
- **Modeling Abstraction**
  - The designs should follow the earlier simulation and Verilog models. The HCL files should be modified only to fix bugs. No tricks should be played just to improve the verifier performance.
  - We should use a "natural" term-level abstraction, using symbolic terms and uninterpreted functions wherever possible, but using the integer arithmetic capabilities of UCLID to capture the byte-oriented instruction encoding of Y86, and any special features of the arithmetic operations that are required.

As these goals indicate, our objective in this project is not simply to verify a particular microprocessor, but to formulate a set of "best practices" for using UCLID, and to identify ways in which UCLID could be easier and more effective to use.

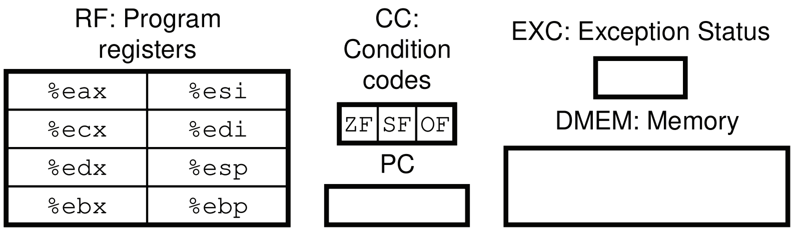

Figure 1: **Y86 programmer-visible state.** As with IA32, programs for Y86 access and modify the program registers, the condition code, the program counter (PC), and the data memory. The additional exception status word is used to handle exceptional conditions.

2 The Y86 Processor

The Y86 instruction set architecture adapts many of the features of the Intel IA32 instruction set (known informally as "x86"), although it is far simpler. It is not intended to be a full processor implementation, but rather to provide the starting point for a working model of how microprocessors are designed and implemented.

2.1 Instruction Set Architecture

Figure 1 illustrates the architectural state of the processor. As with x86, there are eight program registers, which we refer to collectively as the register file RF. Of these registers, only the stack pointer `%esp`[1] has any special status. There are three bits of condition codes, referred to as CC, for controlling conditional branches. There is a program counter PC, and a data memory DMEM. We also introduce an exception status register EXC to indicate the program status. With the formal verification model, the only two exceptional conditions are when an invalid instruction is encountered or when a halt instruction is executed.

Figure 2 illustrates the instructions in the Y86 ISA. These instruction range between one and six bytes long. Simple instructions such as `nop` (No Operation) and `halt` require only a single byte. The x86 data movement instruction is split into four cases: `rrmovl` for register-register, `irmovl` for immediate-register, `rmmovl` for register-memory, and `mrmovl` for memory to register. Memory referencing uses a register plus displacement address computation.

The `OPl` instruction shown in the figure represents four different arithmetic operations, with the operation encoded in the field labeled `fn`. These instructions have registers `rA` and `rB` as source operands and `rB` as destination.

The `jXX` instruction shown in the figure represents seven different branch instructions, where the branch condition is encoded in the field labeled `fn`. Branching is based on the setting of the

[1]We following the naming and assembly code formatting conventions used by the GCC compiler, rather than Intel notation.

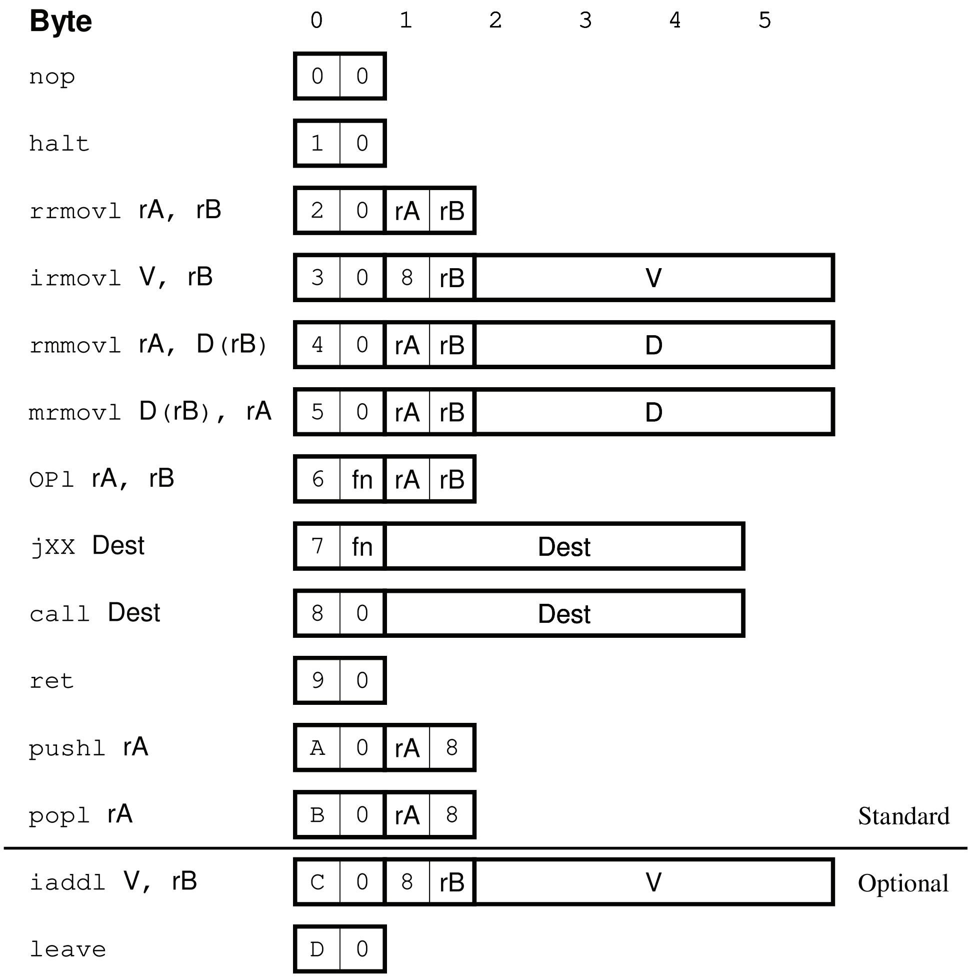

Figure 2: **Y86 instruction set.** Instruction encodings range between 1 and 6 bytes. An instruction consists of a one-byte instruction specifier, possibly a one-byte register specifier, and possibly a four-byte constant word. Field `fn` specifies a particular integer operation (`OPl`) or a particular branch condition (`jXX`). All numeric values are shown in hexadecimal. (From [4, Fig. 4.2]).

condition codes by the arithmetic instructions.

The `pushl` and `popl` instructions push and pop 4-byte words onto and off of the stack. As with IA32, pushing involves first decrementing the stack pointer by four and then writing a word to the address given by the stack pointer. Popping involves reading the top word on the stack and then incrementing the stack pointer by four.

The `call` and `ret` instructions implement procedure calls and returns. The `call` instruction pushes the return address onto the stack and then jumps to the destination. The `ret` instruction pops the return address from the stack and jumps to that location.

The final two instructions are not part of the standard Y86 instruction set, but given to implement as homework exercises in [4]. One of our versions of Y86 implements these instructions. The `iaddl` instruction adds an immediate value to the value in its destination register. The `leave` instruction prepares the stack frame for procedure return. It is equivalent to the two instruction sequence

```
rrmovl %ebp, %esp
popl %ebp
```

where `%ebp` is the program register used as a frame pointer.

We see that Y86 contains some features typical of CISC instruction sets:

- The instruction encodings are of variable length.
- Arithmetic and logical instructions have the side effect of setting condition codes.
- The condition codes control conditional branching.
- Some instructions (`pushl` and `popl`) both operate on memory and alter register values as side effects.
- The procedure call mechanism uses the stack to save the return pointer.

On the other hand, we see some of the simplifying features commonly seen in RISC instruction sets:

- Arithmetic and logical instructions operate only on register data.
- Only simple, base plus displacement addressing is supported.
- The bit encodings of the instructions are very simple. The different fields are used in consistent ways across multiple instructions.

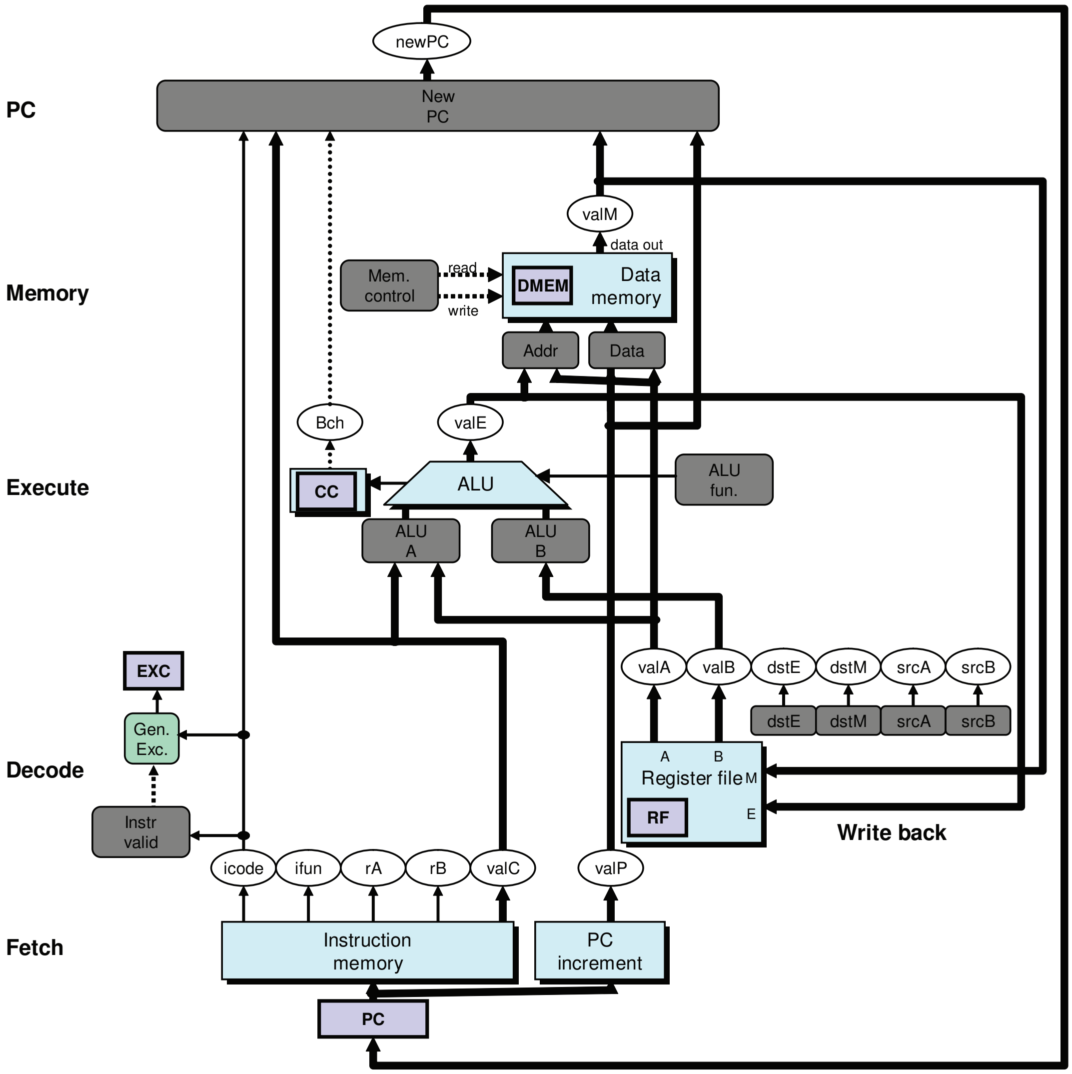

Figure 3: **Hardware structure of** SEQ**.** This design was used as the sequential reference version.

2.2 Sequential Implementation

Figure 3 illustrates SEQ, a sequential implementation of the Y86 ISA, where each cycle of execution carries out the complete execution of a single instruction. The only state elements are those that hold the Y86 architectural state. The data path also contains functional blocks to decode the instruction, to increment the PC, to perform arithmetic and logical operations (ALU). The control logic is implemented by a number of blocks, shown as shaded boxes in the figure. Their detailed functionality is described in HCL, a simple language for describing control logic.

The overall flow during a clock cycle occurs from the bottom of the figure to the top. Starting with the current program counter value, six bytes are fetched from memory (not all are used), and the PC is incremented to the next sequential instruction. Up to two values are then read from the register file. The ALU operates on some combination of the values read from the registers, immediate data from the instruction, and numeric constants. It can perform either addition or the operation called for by an arithmetic or logical instruction. A value can be written to or read from the data memory, and some combination of memory result and the ALU result is written to the registers. Finally, the PC is set to the address of the next instruction, either from the incremented value of the old PC, a branch target, or a return address read from the memory.

2.3 Pipelined Implementation

Figure 4 illustrates a five-stage pipeline, called PIPE, implementing the Y86 instruction set. Note the similarities between SEQ and PIPE—both partition the computation into similar stages, and both use the same set of functional blocks. PIPE contains additional state elements in the form of pipeline registers, to enable up to five instructions to flow through the pipeline simultaneously, each in a different stage. Additional data connections and control logic is required to resolve different *hazard* conditions, where either data or control must pass between two instructions in the pipeline.

There are a total of seven versions of PIPE. The basic implementation STD is illustrated in the figure and described in detail in [4]. The others are presented in the book as homework exercises, where our versions are the official solutions to these problems. They involve adding, modifying, or removing some of the instructions, forwarding paths, branch prediction policies, or register ports from the basic design.

STD This is the standard implementation illustrated in Figure 4. Data hazards for arguments required by the execute stage are handled by forwarding into the decode stage. A one-cycle stall in the decode stage is required when a load/use hazard is present, and a three-cycle stall is required for the return instruction. Branches are predicted as taken, with up to two instructions canceled when a misprediction is detected.

FULL Implements the `iaddl` and `leave` instructions listed as optional in Figure 2. Verification is performed against a version of SEQ that also implements these two instructions.

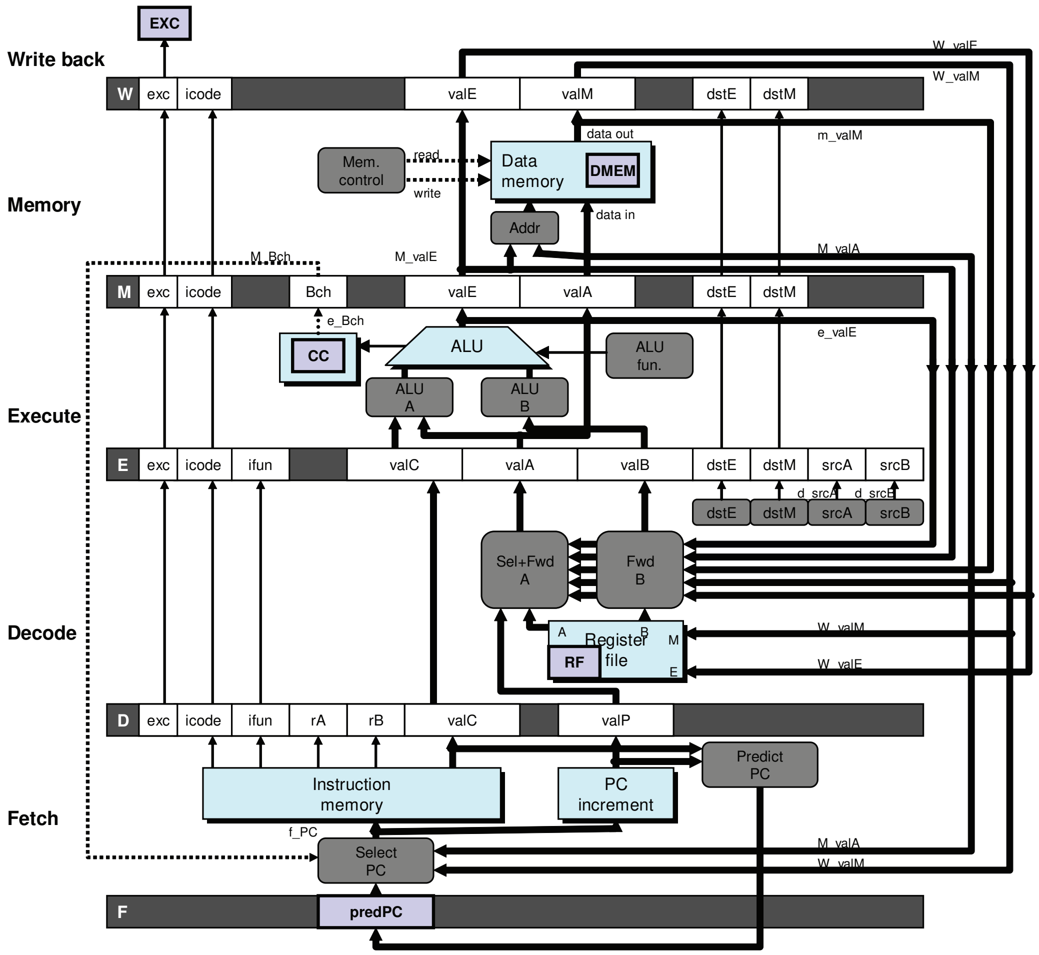

Figure 4: **Hardware structure of PIPE, the pipelined implementation to be verified.** Some of the connections are not shown.

STALL No data forwarding is used by the pipeline. Instead, an instruction stalls in the decode stage for up to three cycles whenever an instruction further down the pipeline imposes a data hazard.

NT The branch prediction logic is modified to predict that branches will not be taken, unless they are unconditional. Up to two instructions are canceled if the branch was mispredicted.

BTFNT Similar to NT, except that branches to lower addresses are predicted as being taken, while those to higher addresses are predicted to not be taken, unless they are unconditional. Up to two instructions must be canceled if the branch is mispredicted.

LF An additional forwarding path is added between the data memory output and the pipeline register feeding the data memory input. This allows some forms of load/use hazards to be resolved by data forwarding rather than stalling.

SW The register file is simplified to have only a single write port, with a multiplexor selecting between the two sources. This requires splitting the execution of the `popl` instruction into two cycles: one to update the stack pointer and one to read from memory.

3 Generating UCLID Models

3.1 Data Types

Our first task was to determine an appropriate level of abstraction for the term-level model. UCLID supports five different data types:

Truth Boolean values, having possible values 0 and 1.

Term Term values. These are assumed to be values from some arbitrary, infinite domain. A limited set of integer operations on terms is also supported.

Enumerated The user can declare a data type consisting of an enumerated set of values. UCLID implements these using term values.

Function These are mappings from terms to terms. They are either uninterpreted, defined by lambda expressions, or one of the integer operations.

Predicate These are mappings from terms to Boolean values. They are either uninterpreted, defined by lambda expressions, or a test for integer equality or ordering.

We wish to formulate a model that maximizes the use of term-level abstraction, in order to maximize the generality of the model and the performance of the verifier. To achieve this goal, it is better to encode a signal as a symbolic term rather than an enumerated type. It is better to use uninterpreted functions and predicates rather than lambda expressions or integer operations. However, if we overdo the level of abstraction, then we can get verification errors that would not be possible in the actual system. For example, the values of an enumerated type are guaranteed to be distinct, whereas the verifier will consider two different symbolic term constants as possibly having the same value.

Based on this goal, we formulated the following representations:

- An enumerated type was introduced for the instruction code (`icode`) fields. Their values are based on the instruction names shown in Figure 2. For example, `IHALT` is the instruction code for the `halt` instruction, while `IOPL` and `IJXX` are the instruction codes for the arithmetic and logical operations and the jump instructions, respectively.

- All other fields in the instruction are modeled as symbolic terms: the function code (for arithmetic and logical operations and for jump instructions), the register identifiers, and the constant data.

- Symbolic constants were introduced to designate special term values: register identifiers `RESP` (stack pointer) and `RNONE` (indicating no register accessed and shown encoded as numeric value 8 in Figure 2), ALU operation `ALUADD`, and data constant `CZERO` (UCLID does not have any designated numeric constants).

- An enumerated type was introduced for the exception status values. These are named `EAOK` (normal operation), `EHLT` (halted), and `EINS` (invalid instruction exception). We added a fourth possibility `EBUB` to indicate a bubble at some stage in the pipeline. This value will never arise as part of the architectural state.

- Program addresses are considered to be integers. The instruction decoding logic increments the PC using integer arithmetic.

- The register file is implemented as a function mapping register identifiers to data values. A lambda expression describes how the function changes when a register is written.

- The set of condition codes is modeled as a single symbolic term. Uninterpreted functions are introduced to specify how the value should be updated. An uninterpreted predicate determines whether a particular combination of branch condition and condition code value should cause a jump. With this level of abstraction, we ignore the number of condition code bits as well as their specific interpretations.

- The data memory is modeled as a single symbolic term, with uninterpreted functions modeling the read and write operations. This abstraction is adequate for modeling processors in which memory operations occur in program order [2].

Some aspects of UCLID make it difficult to make the best use of its data abstraction capabilities.

- Symbolic constants of enumerated type are not supported. This makes it impossible to declare lambda expressions having enumerated types as arguments.

- Uninterpreted functions yielding enumerated types are not supported. This requires a workaround to model instruction decoding, where the instruction code is extracted from the high order bits of the first instruction byte. Instead, we must use a workaround where the instruction decoder consists of a lambda expression that yields different instruction code values depending on the integer value given by an uninterpreted function.

3.2 Modularity

UCLID has limited support for modularity:

- Portions of the system can be grouped into modules. However, nesting of modules is not supported, and so the hierarchy is very shallow.

- Functional blocks can be abstracted as lambda expressions. Lambda expressions can yield results of the three scalar types: Boolean, term, and enumerated. However, their arguments must be terms or Booleans, limiting the class of blocks for which they are useful.

- The DEFINE section of module and overall descriptions make it possible to introduce intermediate values in the next state expression. This makes it possible to define blocks of combinational logic separately, using the intermediate signals as the connection points.

Besides the above noted restrictions, other aspects of UCLID make modularity difficult:

- The signal definitions in the DEFINE section must be ordered such that any signal is defined before it is used. Although this is a natural requirement for an imperative programming language, it makes it harder to decompose the logic into independent blocks, since it induces an ordering restriction on the block definitions.

We achieved our goals for modularity and automation by partitioning the description into multiple files, and then merging them automatically via the process illustrated in Figure 5. As this figure indicates, we create models for both PIPE and SEQ and then merge them with a framework that supports correspondence checking. Several of the files are identical for the two models, ensuring that the two models remain consistent.

To generate a model for either the SEQ or PIPE processor, our code merges a general framework, defining the state variables, with declarations of the connection points for the function blocks and the overall logic describing how each state variable should be updated. This logic comes from three sources: the HCL descriptions of the control logic, translated to UCLID by the program HCL2U, descriptions of the functional blocks, and descriptions of the connections between them. These

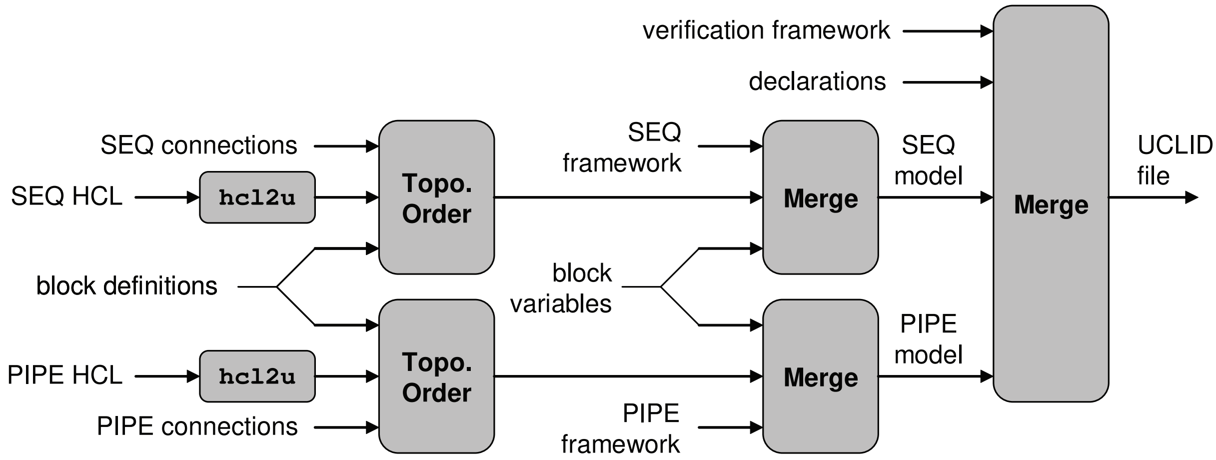

Figure 5: **Generating complete verification file.** The merging steps ensure that a common set of definitions is used for both the SEQ and PIPE models.

definitions must occur in an appropriate order, and so they are merged using topological sorting. Other merging is done using the C compiler preprocessor CPP, with `#include` statements describing the nesting of files. The models for PIPE and for SEQ are declared as separate modules. This makes it possible to have distinct signals with the same names in the two modules.

The blocks labeled HCL2U translate the HCL descriptions of the control logic into UCLID signal definitions. This translation is fairly straightforward, since the two formats use similar semantic models. Both HCL and UCLID formats use case expressions to define data values (integers in the case of HCL, and terms or enumerated types in the case of UCLID) and Boolean expressions to define Boolean values. Both require each signal to be defined in its entirety as a single expression, a style both formats inherit from the original SMV model checker [9]. By contrast, hardware description languages such as Verilog follow more of an imperative programming style, where assignment statements for a signal can appear in multiple locations in a program, with conditional constructs determining which, if any, are executed for a particular combination of state and input values. To extract a network representation from such a description, logic synthesis tools must symbolically simulate the operation of the system and derive the next state expression for each signal.

Figure 6A provides a sample HCL description to illustrate its translation into UCLID. This code describes how the A input to the ALU is selected in SEQ based on the instruction code. The HCL case expression, delimited by square brackets, consists of a list of expression pairs. The first element in the pair evaluates to a Boolean value, while the second gives the resulting data value for the expression. Expressions are evaluated in sequence, where the value returned is given by the first case that evaluates to 1. We can see in this expression that the possible values for this ALU input are `valA`, read from the register file, `valC`, read from the instruction sequence, or constant values $+4$ or -4.

A). HCL description

```
## Select input A to ALU
int aluA = [
        icode in { IRRMOVL, IOPL } : valA;
        icode in { IIRMOVL, IRMMOVL, IMRMOVL } : valC;
        icode in { ICALL, IPUSHL } : -4;
        icode in { IRET, IPOPL } : 4;
        # Other instructions don't need ALU
];
```

B). Generated UCLID code

```
(* $define aluA *)
aluA :=
    case
      (icode = IRRMOVL|icode = IOPL) : valA;
      (icode = IIRMOVL|icode = IRMMOVL|icode = IMRMOVL) : valC;
      (icode = ICALL|icode = IPUSHL) : pred^4(CZERO);
      (icode = IRET|icode = IPOPL) : succ^4(CZERO);
      default : succ^4(CZERO);
    esac;
(* $args icode:valA:valC *)
```

Figure 6: **Automatically generated UCLID code.** The semantic similarity of the two representations allows a direct translation.

```
(* $define dmem_new *)
dmem_new := case
   dmem_wrt : dmem_write_fun(dmem_old, dmem_addr, dmem_din);
   default: dmem_old;
esac;
(* $args dmem_wrt:dmem_old:dmem_addr:dmem_din *)

(* $define dmem_dout *)
dmem_dout := dmem_read_fun(dmem_old, dmem_addr);
(* $args dmem_old:dmem_addr *)
```

Figure 7: **Sample block description.** These provide abstract model of data memory, using uninterpreted functions to abstract read and write functions.

Figure 6 shows the UCLID code generated by HCL2U for this HCL expression. Observe that the overall structure is the same—a case expression where expressions are evaluated in sequence to find the first one evaluating to 1. The differences are mostly syntactic. Instead of square brackets, UCLID uses the keywords `case` and `esac`. UCLID does not support a set membership test, and so these are translated into a disjunction of equality tests. UCLID insists on having a default case, and so HCL2U simply repeats the final expression as a default. We see, though, that the two formats have different ways of modeling integer constants. UCLID does not directly support integer constants. Instead, we translate them into an increment (`succ`) or decrement `pred` relative to the symbolic constant `CZERO`.

We see also that the generated UCLID code contains comments at the beginning and end declaring the name of the signal generated and its arguments. These declarations are used by the topological ordering code.

Figure 7 shows an example of a functional block definition, describing the state and output of the data memory. These descriptions were generated by hand, along with the comments declaring the name of the signal generated and the arguments. As this example indicates, the block descriptions are fairly straightforward. Note the use of uninterpreted functions `dmem_write_fun` and `dmem_read_fun` to describe the updating of the data memory and the extraction of a value by a read operation.

Figure 8 shows how the connections between functional blocks, control logic, and the state variables are defined in both the SEQ and PIPE models. Since the two models are declared as separate models, we can use the same signals in both, such as the connections to the functional blocks, as well as other internal signals such as `mem_addr`. Again, we can see the comments declaring the names of the generated signal and its arguments.

The topological sorter extracts definitions from multiple files and orders them in a way that the definition of a signal occurs before any use of that signal. We implemented this with a simple Perl

A). Connections in SEQ.

```
(* $define dmem_addr *)
dmem_addr := mem_addr;
(* $args mem_addr *)

(* $define dmem_din *)
dmem_din := mem_data;
(* $args mem_data *)

(* $define dmem_wrt *)
dmem_wrt := mem_write;
(* $args mem_write *)

(* $define valM *)
valM := dmem_dout;
(* $args dmem_dout *)

(* $define dmem_old *)
dmem_old := DMEM;
(* $args DMEM *)
```

B). Connections in PIPE.

```
(* $define dmem_addr *)
dmem_addr := mem_addr;
(* $args mem_addr *)

(* $define dmem_din *)
dmem_din := M_valA;
(* $args M_valA *)

(* $define dmem_wrt *)
dmem_wrt := mem_write;
(* $args mem_write *)

(* $define m_valM *)
m_valM := dmem_dout;
(* $args dmem_dout *)

(* $define dmem_old *)
dmem_old := DMEM;
(* $args DMEM *)
```

Figure 8: **Connecting to functional blocks.** By renaming signals, we can instantiate different versions of the functional blocks in the two models.

A.) Uninterpreted

```
alu_out := case
  1 : alu_fun(alu_cntl, aluA, aluB);
  default : alu_fun(alu_cntl, aluA, aluB);
esac;
```

B.) Pass Through

```
alu_out := case
  alu_cntl = ALUADD & aluB = CZERO : aluA;
  default : alu_fun(alu_cntl, aluA, aluB);
esac;
```

C.) Incr./Decr.

```
alu_out := case
  alu_cntl = ALUADD & aluA = succ^4(CZERO) : succ^4(aluB);
  alu_cntl = ALUADD & aluA = pred^4(CZERO) : pred^4(aluB);
  default : alu_fun(alu_cntl, aluA, aluB);
esac;
```

Figure 9: **Different ALU models.** Some Y86 versions require partial interpretations of the ALU function.

script. It relies on the annotations written as comments in the UCLID source, as shown in Figures 6B, 7, and 8 to determine the expression dependencies.

Our method for generating the UCLID file proved very successful in practice. A more ideal approach would be to modify UCLID to support greater modularity as well as file inclusion, but we found the more ad hoc approach of file inclusion, simple translators, and Perl scripts to be satisfactory. We were able to follow a principle that each piece of logic had a single description, and that any modification to this logic would be propagated automatically to any place it was used.

3.3 Modeling Issues

For the most part, our original plan for modeling the processor proved successful. We were able to use uninterpreted functions and predicates extensively in instruction decoding, ALU and condition code operation, and data memory access. There were two places, however, where a more detailed level of modeling was required.

First, we had to be more precise in that handling of register arguments for different instruction types. Our implementation relies on a special register identifier RNONE to indicate when no register argument is present in the pipeline, either as the source or the destination of instruction data. The control logic tests for this value when determining when to inject stall cycles into the pipeline and when and where to forward data from one instruction to another. The behavior of PIPE might not match that of SEQ if either an instruction flowing through the pipeline has a register ID field set to RNONE when it should be a normal program register, or it is not set to RNONE when it should be. The original designs for PIPE and SEQ were designed assuming that the Y86 assembler would generate the register ID fields as is shown in Figure 2, and so there was no provision for handling invalid cases.

We solved this problem by adding logic to detect when an instruction had an improper register ID, causing it to generate an illegal instruction exception, both in SEQ and in PIPE. We consider this to be an example of a positive benefit of formal verification—it forces designers to consider all of the marginal operating conditions and specify just what the processor should do for every possible instruction sequence, not just those that are expected to occur in actual programs. Given the possibility that a processor could encounter arbitrary code sequences either inadvertently (e.g., an incorrect jump target) or due to malice (e.g., code inserted by a virus), it is important to have predictable behavior under all conditions.

Second, some versions of the processor required a more precise modeling of the ALU. Figure 9 shows different versions of the ALU function. The simplest is to model it as an uninterpreted function, having as arguments the function code and the two data inputs (A). This high level of abstraction suffices for three versions: STD, LF, and STALL.

The versions that change the branch prediction policy NT and BTFNT pass the branch target through the A input to the ALU, setting the B input to 0 and the function code to addition. The ALU function must detect this as a special case and pass the A input to the ALU output. The code for this is shown in Figure 9B.

Version SW breaks the `popl` instruction into two steps. Effectively, it dynamically converts the instruction `popl rA` into the two-instruction sequence

```
iaddl $4, %esp
mrmovl -4(%esp), rA
```

This particular ordering of incrementing and then reading, rather than reading and then incrementing, is used to correctly handle the instruction `popl %esp`. We must indicate in the ALU model the effect of having the ALU function be addition and the A input be either $+4$ or -4. Given this information, UCLID can use its ability to reason about the successor and predecessor operations to recognize that $(x + 4) - 4 = x$.

Figure 9C shows how the ALU function can be modified to correctly handle this property. It need only detect the identity when the data is passing through the B input of the ALU, since this is the input used by both the incrementing and the decrementing instruction.

Unfortunately, as we will see with the experimental data, these seemingly small modifications of the ALU function greatly increase the run time of the verifier.

4 Verification

The conventional approach to verifying a pipelined microprocessor design is to perform extensive logic simulation, first running on simple tests, then running on typical programs, and then systematically running tests that were systematically generated to exercise many different combinations of instruction interactions. We had previously done all of these forms of testing on the base designs for SEQ and PIPE, using an instruction set simulator as the reference model. The main untested portions of the designs were the exception handling logic and the mechanisms introduce to enable a flushing of the pipeline. Not surprisingly, these were the portions that required the most effort to get working correctly. In addition, a fair amount of effort was required to deal with modeling issues, such as the different versions of the ALU function and the handling of register identifiers described in the previous section.

4.1 Burch-Dill Pipeline Verification

Our task is to prove that SEQ and PIPE would give the same results on any instruction sequence. We do this using the method developed by Burch and Dill [5], in which two symbolic simulation runs are performed, and then we check for consistency between the resulting values of the architectural state elements.

The overall verification process can be defined in terms of the following simulation operations on models of PIPE or SEQ. We describe these operations as if they were being performed by a conventional simulator. The symbolic simulation performed by UCLID can be viewed as a method to perform a number of conventional simulations in parallel.

Init(s)**:** Initialize the state of the PIPE to state s. This state specifies the values of the architectural state elements as well as the pipeline registers.

Pipe: Simulate the normal operation of PIPE for one cycle.

Flush(n)**:** Simulate n steps of PIPE operating in flushing mode. Instruction fetching is disabled in this mode, but any instructions currently in the pipeline are allowed to proceed.

Seq: Simulate the normal operation of SEQ for one cycle.

Xfer: Copy the values of the architectural state elements in PIPE over to their counterparts in SEQ.

SaveS(s)**:** Save the values of the state elements in SEQ as a state s. UCLID provides *storage variables* to record the symbolic value of a state variable at any point during its execution.

The key insight of Burch and Dill was to recognize that simulating a flushing of the pipeline provides a way to compute an abstraction function α from an arbitrary pipeline state to an architectural state. In particular, consider the sequence

$$\text{Init}(P_0),\ \text{Flush}(n),\ \text{Xfer},\ \text{SaveS}(A)$$

It starts by setting the pipeline to some initial state P_0. Since this is a general pipeline state, it can have some partially executed instructions in the pipeline registers. It simulates a flushing of the pipeline for n steps, where n is chosen large enough to guarantee that all partially executed instructions have been completed. Then it transfers the architectural state to SEQ and saves this as state A. We then say that $\alpha(P_0) = A$. That is, it maps a pipeline state to the architectural state that results when all partially executed instructions are executed.

Our task in correspondence checking is to prove that the operations of PIPE and SEQ remain consistent with respect to this abstraction function. Checking involves performing the following two simulation sequences:

$$\begin{aligned}
\sigma_a &\doteq \text{Init}(P_0),\ \text{Pipe},\ \text{Flush}(n),\ \text{Xfer},\ \text{SaveS}(S_P) \\
\sigma_b &\doteq \text{Init}(P_0),\ \text{Flush}(n),\ \text{Xfer},\ \text{SaveS}(S_0),\ \text{Seq},\ \text{SaveS}(S_1)
\end{aligned}$$

Sequence σ_a captures the effect of one step of PIPE followed by the abstraction function, while sequence σ_b captures the effect of the abstraction function followed by a possible step of SEQ. Sequence σ_b starts with PIPE initialized to the same arbitrary state as in σ_a. It executes the steps corresponding to the abstraction function, saves that state as S_0, runs one step of SEQ, and saves that state as S_1.

The correspondence condition for the two processors can then be stated that for any possible pipeline state P_0 and for the two sequences, we should have:

$$S_P = S_1 \ \lor\ S_P = S_0 \qquad (1)$$

The left hand case occurs when the instruction fetched during the single step of PIPE in sequence σ_a causes an instruction to be fetched that will eventually be completed. If the design is correct, this same instruction should be fetched and executed by the single step of SEQ in sequence σ_b. The right hand case occurs when either the single step of PIPE does not fetch an instruction due to a stall condition in the pipeline, or an instruction is fetched but is later canceled due to a mispredicted branch. In this case, the verification simply checks that this cycle will have no effect on the architectural state.

Our above formulation holds for a single-issue pipeline such as PIPE. It can readily be extended to superscalar pipelines by running multiple cycles of SEQ in sequence σ_b [15].

4.2 Implementing Burch-Dill Verification with UCLID

Implementing Burch-Dill verification involves a combination of constructing a model that supports the basic operations listed above and creating a script that executes and compares the results of sequences σ_a and σ_b. This is documented via a simple pipelined data path example in the reference manual [11]. Our verification framework includes control signals that allow PIPE to be operated in either normal or flushing mode, that allow the SEQ state elements to import their values from the corresponding elements in PIPE, and that allow SEQ to be operated. As the execution proceeds, we capture the values of state variables in UCLID storage variables.

As an example, the following code is used to capture the matching condition of Equation 1 as a formula `match`:

```
match0 := (S0_pc = SP_pc) & (S0_rf(a1) = SP_rf(a1))
          & (S0_cc = SP_cc) & (S0_dmem = SP_dmem) & (S0_exc = SP_exc);

match1 := (S1_pc = SP_pc) & (S1_rf(a1) = SP_rf(a1))
          & (S1_cc = SP_cc) & (S1_dmem = SP_dmem) & (S1_exc = SP_exc);

match := match1 | match0;
```

In these formulas, a storage variable with name of the form SP_x refers to the value of state element x in state S_P. Similarly, names of the form S0_x and S1_x refers to the values of state element x in states S_0 and S_1, respectively.

Our method of comparing the states of the register files must work around the fact that UCLID does not support testing of equality between functions. Instead, we state the test $f = g$ for functions f and g as $\forall x[f(x) = g(x)]$. We do this by introducing a symbolic constant `a1` and testing that the two register file functions yield the same values for this constant. Since symbolic constants are implicitly universally quantified, we can therefore test that the register files have identical contents.

The matching condition of Equation 1 must hold for all possible states of the pipeline P_0. In practice, however, many possible pipeline states would never arise during operation. There could be an intra-instruction inconsistency, for example, if a register identifier is set to `RNONE` when it should denote a program register. There can also be inter-instruction inconsistencies, for example when a `ret` instruction is followed by other instructions in the pipeline rather than by at least three bubbles. These inconsistent states can be excluded by the verifier, but only if we can also prove that they can never actually occur.

We define a *pipeline invariant* to be any restriction I on the pipeline state that can be shown to hold under all possible operating conditions. To ensure I is invariant, we must show that it holds when the processor is first started, and that it is preserved by each possible processor operation. The former condition usually holds trivially, since the pipeline will be empty when the processor is first started, and so we focus our attention on proving the latter *inductive* property. We do this by executing the following simulation sequence:

$$\text{Init}(P_0),\ \text{Pipe},\ \text{SaveP}(P_1)$$

where $\text{SaveP}(s)$ saves the values of the pipeline state elements as a state s. We must then prove $I(P_0) \Rightarrow I(P_1)$.

We would like to keep our pipeline invariants as simple as possible, since they place an additional burden on the user to formulate and prove them to be invariant.

We found that several versions of PIPE could be verified without imposing any invariant restrictions. This is somewhat surprising, since no provisions were made in the control logic for handling conditions where data will not be properly forwarded from one instruction to another, or where pipeline bubbles can cause registers to be updated. The design is more robust than might be expected. One explanation for this is that the two simulation sequences σ_a and σ_b start with the same pipeline state. Any abnormalities in this initial state will generally cause the same effect in both simulations. Problems only arise when the inconsistent instructions initially in the pipeline interfere with the instruction that is fetched during the first step of σ_a.

Some versions of PIPE required imposing initial state restrictions, based on invariant properties, as illustrated in Figure 10. These expressions describe consistency properties by referencing storage variables that captured parts of the initial pipeline state. First, we found it necessary to account for the different exception conditions, and how they would affect the instruction code and the destination register fields. Figure 10A shows such an example *stage invariant*, imposing a consistency on the memory stage. Similar conditions were defined for the decode, execute, and write-back stages.

The property shown in Figure 10B describes the requirement that any `ret` instruction passing through the pipeline will be followed by a sequence of bubbles, owing to the 3-cycle stall that this instruction causes. The conjunction of this invariant with the stage invariants has the effect of imposing restrictions on the destination register and instruction code fields of the following instructions.

The property shown in Figure 10C is specific to the SW implementation. It constrains each pipeline stage to have at most one destination register. This property is implicitly assumed to hold in our control logic for SW.

Verifying that these invariants are inductive takes at most 10 CPU seconds, and so it is not a serious burden for the verification. We also found it important to actually carry out the verification. Some of our earlier formulations of invariants proved not to be inductive, and in fact were overly restrictive when used to constrain the initial pipeline state.

Figure 11 shows four formulations of the correctness condition used for different versions of PIPE. They differ only in what antecedent is used to constrain the data values and the initial state. All of them include the requirement that `RESP`, the register identifier for the stack pointer, be

A.) Constraints among state variables in memory stage

```
M_ok0 :=
    (pipe.M_exc = EAOK =>
       pipe.M_icode != IHALT & pipe.M_icode != IBAD)
&   (pipe.M_exc = EBUB | pipe.M_exc = EINS =>
       pipe.M_dstM = RNONE & pipe.M_dstE = RNONE & pipe.M_icode = INOP)
&   (pipe.M_exc = EHLT =>
       pipe.M_dstM = RNONE & pipe.M_dstE = RNONE & pipe.M_icode = IHALT);
```

B.) Constraints on pipeline state following `ret` instruction.

```
(* Check that ret instruction is followed by three bubbles *)
ret_ok0 :=
    (pipe.E_icode = IRET => pipe.D_exc = EBUB)
&   (pipe.M_icode = IRET => pipe.D_exc = EBUB & pipe.E_exc = EBUB)
&   (pipe.W_icode = IRET
     => pipe.D_exc = EBUB & pipe.E_exc = EBUB & pipe.M_exc = EBUB);
```

C.) Constraints on destination registers for SW implementation

```
(* Check that at most one register write in each pipeline stage *)
swrite0 :=
    (pipe.E_dstE = RNONE | pipe.E_dstM = RNONE)
&   (pipe.M_dstE = RNONE | pipe.M_dstM = RNONE)
&   (pipe.W_dstE = RNONE | pipe.W_dstM = RNONE);
```

Figure 10: **Pipeline state invariants.** These formulas express consistency conditions for the initial pipeline state.

A.) Arbitrary initial pipeline state:

```
(* Verification of all instructions with unrestricted initial state *)
  (RESP != RNONE  => match)
```

B.) Arbitrary initial pipeline state with `REBP` value constraint.

```
(* Verification of all instructions with unrestricted initial state *)
  (REBP != RNONE & RESP != RNONE  => match)
```

C.) Initial pipeline state satisfies return invariant:

```
(* Verification of all instructions with restricted initial state.
   Restriction accounts for stalling after ret instruction *)
  ((W_ok0 & M_ok0 & E_ok0 & D_ok0 & RESP != RNONE & ret_ok0)
    => match)
```

D.) Initial pipeline state satisfies single-write invariant:

```
(* Verification of all instructions with restricted initial state.
   Restriction accounts single-write property of SW pipeline *)
  ((W_ok0 & M_ok0 & E_ok0 & D_ok0 & swrite0 & ret_ok0 & RESP != RNONE)
      => match)
```

Figure 11: **Verification conditions.** Different Y86 versions require different verification condition.

Version	Flush Steps	ALU Model	Condition	Time (minutes)
STD	5	Uninterp.	Arbitrary	3.9
FULL	5	Uninterp.	Arbitrary + REBP value	4.6
STALL	7	Uninterp.	Return Invariant	27.0
NT	5	Pass through	Arbitrary	22.3
BTFNT	5	Pass through	Arbitrary	24.3
LF	5	Uninterp.	Return Invariant	4.0
SW	6	Incr./Decr.	Single Write	465.3

Figure 12: **Verification times for different Y86 versions.** All times are the total number of CPU minutes for the complete run on an 2.2 GHz Intel Pentium 4.

distinct from RNONE, the value indicating that no register is to be read or written. Since both of these values are declared as symbolic constants, UCLID would otherwise consider the case where they are the same, and this causes numerous problems with the data forwarding logic. The first verification condition (A) does not constrain the initial state. It suffices for verifying versions STD, NT, and BTFNT. For verifying FULL, we must impose an additional data constraint that REBP, the register identifier for the base pointer, is distinct from RNONE. The third formulation (C) requires register consistency within each stage as well as that `ret` instructions are followed by bubbles. This condition is required for verifying versions STALL and LF. The fourth (D) imposes all of these constraints, plus the single-write property. This version is used for verifying SW.

Figure 12 documents the different combinations of n, the number of steps to flush the pipeline, the ALU function, and the verification condition required to verify each version of PIPE. We see that most versions could be flushed in five steps: one due to a possible stall when a load/use hazard occurs, and four to move the instruction initially in the decode stage through the decode, execute, memory, and write-back stages. Version STALL requires seven steps, since a data hazard can require up to three stall cycles. Version SW requires six steps due to the additional decode cycle it uses.

The figure also shows the CPU times for the verification running on a 2.2 GHz Intel Pentium 4. The times are expressed in CPU minutes, and include both the user and system time for the complete UCLID run. Running the SAT solver ZCHAFF [10] accounted for 80–90% of the total CPU time. We can see that the verification times range from several minutes to almost eight hours. We investigate the factors that influence performance later in this report.

4.3 Errors

We discovered only one error in the HCL descriptions of the different processor versions. The SW version handled the following instruction sequence improperly:

ALU	Flush Steps		
	5	6	7
Uninterp.	3.9	6.6	11.2
Pass Through	22.3	45.1	76.2
Incr./Decr.	40.9	92.9	216.3

Figure 13: **Verification times for STD.** The times depend heavily on the number of flushing steps and the type of ALU model.

```
popl %esp
ret
```

This instruction sequence causes a combination of control conditions as discussed in [4, pp. 349–350], where the load/use hazard between the two instructions occurs at the same time the control logic is preparing to inject bubbles following the `ret` instruction. This combination was carefully analyzed and handled in the other versions of PIPE, but it takes a slightly different form in SW where the `popl` instruction is split into two steps. This bug was easily corrected in the HCL file.

The fact that this bug had not been detected previously exposed a weakness in our simulation-based verification strategy. We had attempted to systematically simulate the processor for every possible pair of instructions where the first generates a register value and the second reads this value, possibly with a `nop` instruction between them. Unfortunately, we had not considered cases where the `ret` instruction is the reader of `%esp`. This is a general problem with simulation—it can be difficult to identify potential weaknesses in the testing plan.

4.4 Performance Analysis

We saw in Figure 12 that the CPU times to verify the different Y86 versions varied greatly, with the longest being 119 times greater than the shortest. Moreover, we see that the longest one required nearly eight hours, which proved a serious hindrance to completing the verification. Four factors could contribute to the varying times for these verifications:

1. the invariant restrictions imposed on the initial pipeline state.
2. the number of cycles required to flush the pipeline,
3. the level of abstraction used for modeling the ALU function, and
4. the pipeline characteristics of the model,

To better evaluate the relative effects, we ran several experiments holding some parameters fixed while varying others.

First, we determined that the invariant restrictions have little effect on the running time by verifying STD using all three formulations of the correctness condition show in Figure 11. All three formulas are valid, although the single-write invariant is not inductive, since only the SW version obeys the single-write property. The verification times were 3.9, 3.5, and 3.7 minutes, respectively. Given that SAT uses a heuristic search-based algorithm, which even effects the performance when proving a formula is unsatisfiable, this variation is negligible.

To evaluate the second and third factors, we ran a number of verifications of STD using different ALU models and numbers of flush cycles. These times are shown in Figure 13. We can see that both of these factors have a major influence on the verification times. First, looking at each row of the table, we see that the verification time increases by a factor of 1.7–2.3 with each additional flushing step. In other words, the verification time grows exponentially with the number of flushing steps. This is consistent with other uses of term-level verification for Burch-Dill verification [16] and for bounded property checking [3]. The propositional formulas grow in size by factors of 1.6–1.8 per additional flush cycle, and the times for the SAT solver grow slightly worse than linearly.

Looking at the columns in the table of Figure 13, we see that the seemingly small changes in the ALU function shown in Figure 9 can have a major effect on the verification times. Making the ALU pass values through when there is an addition by 0 increases the verification times by a factor of 5.7–6.9. Making the ALU properly handle the incrementing and decrementing by four increases the verification times by a factor of 10.4–19.3. This demonstrates the fragility of UCLID's decision procedure. A small change in the HCL files—only a few lines of code—can have a large impact on the verification times. By conditionally testing the ALU input value, we lose the ability to exploit positive equality [2] in the decision procedure. Although changing the ALU function has little effect on the total number of symbolic constants that are required, the fraction that can be treated as unique values shifts from around 80% to around 40% of the total. Essentially, the constants representing all of the values in the data path must be encoded in a more general, and more costly form. The underlying propositional formulas grow by factors of 2.0–2.3 for the pass-through case and by factors of 2.4–2.7 for the increment/decrement case, but these formulas are significantly more challenging for the SAT solver.

The number of flushing steps and the ALU model account for a large portion of the variation between the different verification times. Figure 14 attempts to factor out these effects to show the effect of the pipeline structure. Each row compares the verification time for one of the Y86 versions from Figure 12 to the verification of STD with the matching number of flushing steps and ALU model from Figure 13. We see that the verification times for four of the versions: FULL, NT, BTFNT, and LF then have additional factors of only 1.0–1.2. On the other hand, STALL, which resolves all data hazards by stalling, has an additional factor of 2.4, while SW, which splits the `popl` instruction into separate increment and read instructions, has an additional factor of 5.0. Clearly, these changes in how the instructions flow through the pipeline have a major impact

Version	Time (minutes)	STD Time (minutes)	Ratio
STD	3.9	3.9	1.0
FULL	4.6	3.9	1.2
STALL	27.0	11.2	2.4
NT	22.3	22.3	1.0
BTFNT	24.3	22.3	1.1
LF	4.0	3.9	1.0
SW	465.3	92.9	5.0

Figure 14: **Isolating pipeline structure effects.** The verification time for each version is compared to the applicable time from Figure 13

on the complexity of the correctness formulas generated. This is not reflected in the sizes of the propositional formulas: only the formula for SW is significantly larger (1.5X) than the matching STD formulas, but the formulas for SW and STALL are clearly more challenging for the SAT solver.

4.5 Liveness Verification

As mentioned earlier, the fact that our correctness condition (Equation 1) includes the case where the pipeline makes no progress in executing the program implies that our verification would declare a device that does absolutely nothing to be a valid implementation. To complete the verification, we must also prove that the design is live. That is, if we operate the pipeline long enough, it is guaranteed to make progress.

We devised a simple method for proving liveness that can be performed on the PIPE model alone. We added an additional state variable `Icount` that counts the number of instructions that have been completed by the pipeline. This value is incremented every time an instruction passes through the write-back stage, as indicated by the exception status for the stage being something other than `EBUB`. Incrementing is enabled only when the architectural exception status `EXC` has value `EAOK`.

We created a verification script for PIPE that starts with an arbitrary pipeline state, runs the pipeline for five cycles, and checks that the value of the instruction counter changes. More specifically, that the value of the counter will change if and only if the initial exception status is `EAOK`, as given by the following correctness condition:

```
RNONE != RESP & M_ok0 & E_ok0 & D_ok0
=>
(P_exc0 = EAOK & pipe.Icount != icount0
```

```
| P_exc0 != EAOK & pipe.Icount = icount0)
```

Five cycles is enough to handle the worst case condition of the pipeline starting out empty. We succeeded in verifying this condition for all seven versions, each taking less than one CPU second. We must start the pipeline with the decode, execute, and memory stages satisfying our intra-instruction consistency requirements. Otherwise, if the pipeline started with "hidden" jump instructions—instructions that cause a branch misprediction but have their exception status set to EBUB, then five cycles of execution would not suffice to ensure the instruction counter changes value.

Our liveness check makes use of the safety check provided by Burch-Dill verification. The safety check implies that any progress made by the pipeline will be consistent with that of the ISA model. We only need to show that progress is made at some nonzero rate.

5 Lessons Learned

So far, we have presented the final outcome of our verification effort, giving the final models, formulas, and CPU times required for verifying our different Y86 versions. This is typical of the way experimental results are presented in research papers on formal verification. What this fails to capture is the substantial work required to build the models, and to find and fix bugs in the design, the model, or the specification. In this section, we attempt to capture some of the ideas and practices we developed and used to make the process more tractable.

One principle we followed with great success was to use as much automation as possible in constructing the model and in running the verification. Figure 5 demonstrated the automation used in model construction. We see that relatively simple tools—Perl scripts, format translators, and the C preprocessor—can be strung together for this task. This process meant that any change to any part of the model only had to be done in one place.

When running a tool such as UCLID, it is important to realize that there are many aspects of the verification that can cause it to fail:

- *Hardware design*: The actual design of one of the two processors is incorrect. These are the bugs we are trying to find.
- *Model*: The logic describing the state variables, data path, or additional logic for flushing and transferring state contains an error. Such errors arise partly because there is no reliable way to construct a verification model directly from the hardware description language representation of a circuit. Even then, some of the additional logic is unique to our verification effort. We found it especially challenging to get the flushing and exception handling mechanisms to work in combination.
- *Abstraction*: Data types or functional models have been chosen that do not capture all of the properties required to ensure proper operation.

- *Specification*: Some aspect of the symbolic execution or the correctness formula contains an error.

As the list above shows, actual bugs in the design are only one of several possible sources of error. One of the challenges of verification is to determine why a particular verification run has failed.

5.1 Guiding the Verifier

Compared to debugging a design by simulation, uncontrolled use of a formal verification tool such as UCLID can be very frustrating. We highlight a few ways we experienced and dealt with these frustrations.

First, with simulation, the designer initially develops simulation patterns that direct the testing toward a specific aspect of the design. He or she can therefore focus attention on one part of the design at a time, performing a number of tests and modifications before moving on to another aspect of the design. In implementing a microprocessor, the designer typically implements and verifies a small number of instructions at a time. Furthermore, he or she typically focuses on handling the normal operations of the system initially, and then on how it handles exceptional cases later. By contrast, UCLID generates counterexamples based on whatever satisfying assignment the SAT solver generates. From one run to another, there will seldom be a correlation between the counterexamples, making it hard to focus on one aspect of the design.

Second, with simulation, the designer begins each simulation run by simulating an initialization sequence to put the system into some known initial state. From that point on, he or she tries to always keep the system in a reliable state to avoid wasting effort on problems that would only occur if the system somehow got into an inconsistent state. By contrast, Burch-Dill verification requires the system to start in a general state. We saw that we can impose restrictions on the initial state, if these restrictions can be shown to be inductive, but this requires the user to start from a general case and narrow it down, rather than the more intuitive approach of starting from simple cases and generalizing them. It would require a substantial effort to formulate a precise invariant restricting the system to states that can only arise during actual system operation. Even the most restrictive invariant we used (Figure 10C) includes pipeline states where data and control hazards will not be handled properly. As a consequence, the verifier will often generate counterexamples that clearly will not happen in actual operation. The user must decide whether such counterexamples point to an underlying design or model error, or whether a more restrictive initial state condition should be used.

Finally, compared to the elegant user interfaces that have been developed for visualizing circuit operation during simulation, the user interface provided by UCLID is quite primitive. Tracing through a single counterexample can easily take 30 minutes or more, and this process must often be restarted from scratch from one counterexample to the next.

We found that the best way to use UCLID was to adopt the practices that make debugging by

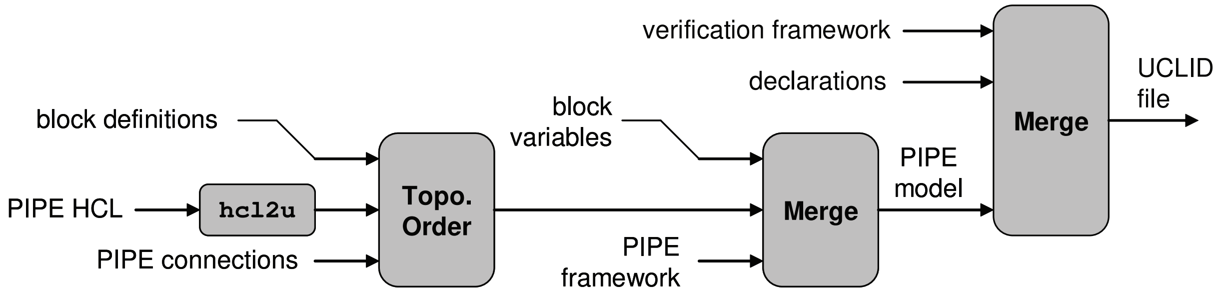

Figure 15: **Generating PIPE verification file.** This model can be used to verify assertions about individual instructions.

simulation more tractable. In particular, we followed an approach where we constrained the verifier to focus initially on simple cases and then slowly generalized these cases until we achieved full verification. We did this by imposing preconditions on the correctness formula that narrow the range of conditions modeled. We start by artificially constraining the pipeline to simple initial states, and considering only limited instruction possibilities. For example, the following formula restricts the verifier to consider only the case of a `rrmovl` instruction moving through an initially empty pipeline:

```
((W_ok0 & W_exc0 = EBUB & M_ok0 & M_exc0 = EBUB
  & E_ok0 & E_exc0 = EBUB & D_ok0 & D_exc0 = EBUB
  & SP_icode0 = IRRMOVL)
   => match)
```

This formula uses stage invariants such as we showed for the memory stage in Figure 10A. It then further restricts each stage to start with a bubble, and to only consider the case where the instruction being fetched has the appropriate instruction code value. Any counterexample for this case should be fairly easy to trace. It will show a single `rrmovl` instruction moving through the pipeline.

If verification fails when the initial pipeline state is constrained to have all stages empty, then we can simplify things further by focusing independently on SEQ and PIPE. Figure 15 illustrates the process used to generate a verification model for PIPE operating on its own. A similar process can be used to generate a verification model for SEQ. We see that this model is generated using the same files and tools as is used to construct the complete Y86 verification model (Figure 5). Thus, anything we prove about this model, or any changes we make, will reflect back to the main verification.

Figure 16 shows how we can specify the behavior of individual instructions for the two different processors. This example shows a precise specification of the effect of executing a `rrmovl` instruction. In the formula for SEQ, each storage variable with name of the form S_x0 captures the value of state variable x initially, while those named S_x1 capture the values after one step of

A.) Verifying `rrmovl` instruction in SEQ

```
(* Effect of RRMOVL instruction *)
 (S_exc0 = EAOK & S_icode0 = IRRMOVL & S_rB0 != RNONE & S_rA0 != RNONE =>
   (S_pc1 = succ^2(S_pc0) & S_cc0 = S_cc1 & S_exc1 = EAOK
    & S_rf1(S_rB0) = alu_fun(ALUADD, S_rf0(S_rA0), CZERO)
    & (a1 != S_rB0 & a1 != RNONE => S_rf1(a1) = S_rf0(a1))
 ))
```

B.) Verifying `rrmovl` instruction in PIPE

```
 (D_empty0 & E_empty0 & M_empty0 & W_empty0) =>
(* Effect of RRMOVL instruction *)
 (P_exc0 = EAOK & P_icode0 = IRRMOVL & P_rB0 != RNONE & P_rA0 != RNONE =>
   (P_pc1 = succ^2(P_pc0) & P_cc0 = P_cc1 & P_exc1 = EAOK
    & P_rf1(P_rB0) = alu_fun(ALUADD, P_rf0(P_rA0), CZERO)
    & (a1 != P_rB0 & a1 != RNONE => P_rf1(a1) = P_rf0(a1))
 ))
```

Figure 16: **Examples of verification test runs.** Many modeling inconsistencies were detected by running the same instruction through the SEQ and PIPE descriptions independently.

operation. In the formula for PIPE these variables capture the values in the initial state and after one step of normal operation followed by seven flush cycles. Running either of these verifications requires less than 10 seconds of CPU time, and will quickly uncover any discrepancies between the two models.

Once instructions have been verified flowing through an empty pipeline, we can relax the initial state conditions a little bit at a time, allowing more general conditions in each stage. By this means we can quickly isolate a problematic interaction between stages. We found it worthwhile to try to simplify the conditions as much as possible before examining the counterexamples in detail.

6 Conclusions

We succeeded in verifying seven different versions of a pipelined, Y86 microprocessor. One version contained a bug in the control logic that caused incorrect results for some assembly language programs. More significantly, we learned several important lessons about how to use UCLID and how it could be improved.

6.1 Appraisal of UCLID

Overall, we found UCLID to be a usable tool. The modeling language is reasonably expressive and flexible. With the aid of additional tools, we were able to set up an automated system for generating models. The general ability to support different forms of symbolic execution and testing of validity conditions allows the user to guide the verification in a manner similar to the way designs are debugged by simulation. A few aspects of UCLID could clearly be improved. These are mainly in the modeling language and in the counterexample facility. Some aspects most in need of improvement are described below.

In the modeling language, it would be helpful if the three scalar data types: Boolean, term, and enumerated, could be used more interchangeably. Currently, uninterpreted function arguments must be terms, while lambda arguments must be terms or Booleans. Uninterpreted functions yield terms, while uninterpreted predicates yield Booleans, but there is no counterpart generating enumerated types. Lambda Expressions can generate terms, Booleans, or enumerated types. The type restrictions force the user to select data types partly on how they can be used, rather than on what is appropriate for the system being modeled, or to use some of the tricks and workarounds. What we would really like is to have all three types treated equally. Any of the three should be usable as arguments, and any of the three should be possible results of uninterpreted functions or lambda expressions.

We also found the limited support for modularity to be somewhat burdensome. It forced us to use a workaround of syntactic replication, and required us to write additional scripts to topologically sort signal definitions. Better support for modularity would be especially useful in translating from hardware description language (HDL) representations of a system. For example, it would be convenient to write the HDL in a highly modular way, so that converting it to a term-level representation would involve mainly selecting the representations for different signals and then designating some of the modules to become uninterpreted functions.

The second major area for improvement is in the run-time support, and especially in the counterexample reporting facility. Some of the interface features found in simulators would be useful in helping the user understand counterexamples. Just as simulators dump their trace information to a file that the user can then repeatedly examine with visualization tools, it would help to be able to examine a single counterexample trace multiple times with the user specifying which signal values to display. It should be possible to observe not just state variables, but also the simulator control signals as well as internal signals that have been identified by the user beforehand.

6.2 Prospects for Industrial Use

Although this exercise revealed some aspects of what it would be like to apply UCLID to an actual industrial microprocessor design, we are still far short of showing that this tool can be used in a real-life design. Our Y86 design was very limited in its data types, the number of instructions, and details such as exception handling. Even with improvements to the modeling language and the run-time support described above, it is not clear whether UCLID can scale to real-life designs. Even if we restrict ourselves to linear pipelines such as the one used here, a complete microprocessor design would be considerably more complex. It is not clear whether the performance of UCLID would scale up to such complexity. We saw already that simple changes to add more detail to the ALU model result in much longer execution times. Adding more stages to the pipeline also increases verification time due to the larger number of steps required for flushing. Improvements to the core decision procedure may be required before UCLID is ready for industrial usage.

References

[1] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[2] 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)*, LNCS 1633, pages 470–482, 1999.

[3] R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In E. Brinksma and K. G. Larsen, editors, *Computer-Aided Verification (CAV '02)*, LNCS 2404, pages 78–92, 2002.

[4] R. E. Bryant and D. R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Prentice-Hall, 2002.

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

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

[7] S. K. Lahiri and R. E. Bryant. Deductive verification of advanced out-of-order microprocessors. In *Computer-Aided Verification (CAV '03)*, LNCS 2725, pages 341–354, 2003.

[8] S. K. Lahiri, S. A. Seshia, and R. E. Bryant. Modeling and verification of out-of-order microprocessors in UCLID. In M. D. Aagaard and J. W. O'Leary, editors, *Formal Methods in Computer-Aided Design (FMCAD '02)*, LNCS 2517, pages 142–159, 2002.

[9] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1992.

[10] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *38th Design Automation Conference (DAC '01)*, pages 530–535, 2001.

[11] S. A. Seshia, S. K. Lahiri, and R. E. Bryant. *A User's Guide to UCLID Version 1.0*. Carnegie Mellon University, 2003. Available at `http://www.cs.cmu.edu/~uclid`.

[12] Mandayam Srivas and Mark Bickford. Formal verification of a pipelined microprocessor. *IEEE Software*, 7(5):52–64, 1990.

[13] M. N. Velev. Using positive equality to prove liveness for pipelined microprocessors. In *Asia and South Pacific Design Automation Conference*, pages 316–321, 2004.

[14] M. N. Velev and R. E. Bryant. Bit-level abstraction in the verification of pipelined microprocessors by correspondence checking. In *Formal Methods in Computer-Aided Design (FMCAD '98)*, LNCS 1522, pages 18–35, 1998.

[15] M. N. Velev and R. E. Bryant. Superscalar processor verification using efficient reductions of the logic of equality with uninterpreted functions. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods (CHARME '99)*, LNCS 1703, pages 37–53, 1999.

[16] M. N. Velev and R. E. Bryant. Formal verification of superscalar microprocessors with multicycle functional units, exceptions and branch predication. In *37th Design Automation Conference (DAC '00)*, pages 112–117, 2000.

[17] M. N. Velev and R. E. Bryant. Effective use of Boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors. In *38th Design Automation Conference (DAC '01)*, pages 226–231, 2001.