Formal Verification of Pipelined Y86-64 Microprocessors with UCLID5

Randal E. Bryant

October, 2018
CMU-CS-18-122

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This work was supported, in part, by the Semiconductor Research Corporation under contract 2637.001, and the National Science Foundation under STARSS grant 1525527.

Abstract

This work reports on the verification of a complex instruction set computer (CISC) processor, named Y86-64, styled after the Intel64 instruction set using the UCLID5 verifier. We developed a methodology in which the control logic is translated into UCLID5 format automatically, and the pipelined processor and the sequential reference version were described with as much modularity as possible. This work provides confidence in the processor designs presented in the Bryant-O'Hallaron textbook on computer systems, and it also provides a case study for the capabilities and performance of UCLID5.

Keywords: Formal verification, Microprocessor verification, UCLID5

1 Introduction

This report describes a case study in using the UCLID5 verifier [14] to formally verify several variants of the Y86-64 pipelined microprocessor presented in third edition of Bryant and O'Hallaron's computer systems textbook [5]. The purpose of this exercise was 1) to make sure the designs are actually correct, and 2) to evaluate the capabilities of UCLID5 for modeling and verifying hardware designs. UCLID5 supports a variety of data types and abstraction techniques. We explore how these features affect its ability to verify the different processor variants, in terms of both modeling capabilities and performance. We succeeded in this effort, showing that the different pipeline processors will generate the same results as does the sequential reference model for all possible programs.

1.1 Background

The traditional approach to verifying a hardware design is to run many simulations. For microprocessors, this involves writing a suite of test programs that will exercise its many operations and check the results. For a pipelined implementation, programs testing different sequences of operations are required to check the many possible interactions that can occur with instructions being processed simultaneously. Special attention must be paid to how the processor handles different exceptional conditions, such as invalid instructions and out-of-bounds memory accesses. Microprocessor developers find it challenging to design a comprehensive suite of tests that truly exercise the many corner cases.

An alternative approach is to use formal verification tools, generating a mathematical proof (or at least something close to a proof) that the processor will operate correctly for all possible programs. Unlike human-generated proofs, which are prone to both fundamental, conceptual errors, as well as simple omissions, formal verification tools use automated methods to systematically ensure that the rigor of their reasoning. Although some formal verification tools mimic the style of proofs constructed by humans, breaking the task down into devising and proving a number of lemmas and theorems, more automated tools rely on symbolic forms of simulation and reasoning to exhaustively analyze all possible system behaviors [7]. Using formal verification eliminates the reliance on test suites. It eliminates the nagging doubt that some test case may have been overlooked, allowing a design flaw to escape detection.

Microprocessors have 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 employed to ensure that the pipelined execution faithfully implements the sequential semantics of the ISA. The task of formal microprocessor verification is to prove that this semantic relationship holds for all possible programs. 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 decades [10], the key ideas used in our verification effort are based on ones described by Burch and Dill in 1994 [6]. The main requirement for 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. Most pipelined processor designs already have some mechanism for flushing instructions, because this is required to bring the pipeline to a quiescent state when dealing with exceptional conditions, such as halting or handling an interrupt. 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*.

Prior to Burch and Dill, much of the work in automated formal hardware verification required that systems be modeled in terms of their precise, bit-level operations. This limited the size and complexity of the systems that could be verified. Burch and Dill demonstrated the value of employing data abstractions, using *term-level modeling* for their 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 bit widths and formats of different data types can be 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 [10, 15], Burch and Dill were the first to show their use in an automated microprocessor verification tool.

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. In this report, we describe a simple and effective approach for proving liveness that builds on the safety property to

show that the pipeline does not stall indefinitely.

1.2 UCLID5

The UCLID5 verifier is the most recent of a series of formal verification tools developed at Carnegie Mellon University [2] and at University of California, Berkeley [14]. It provides both a *modeling language* with which the user describes a system to be verified, and a *command language* with which the user creates a *verification script*, describing how the system state is to be initialized, how the system is to be operated, and what verification conditions should be checked. In our case, the modeled system consists of a combination of a pipelined microprocessor and the sequential reference implementation, while the verification script describes the steps required to carry out Burch-Dill correspondence checking.

UCLID5 is designed to support models involving combinations of (synchronous) hardware and software. Hardware is expressed in terms of state machines—computing a next state in terms of the current state, and then transitioning to that next state. Software is expressed in terms of sequences of operations, each updating some part of the system state. For verifying pipelined microprocessors, only the hardware modeling aspects of UCLID5 were used.

UCLID5 provides a number of different data types, each supporting different operations. When modeling hardware, these types have many different uses:

Uninterpreted: Suitable for the term-level modeling demonstrated by Burch and Dill. Such terms can be tested for equality. They also support *uninterpreted functions*, viewed by the verifier as having an arbitrary, but consistent functionality. For example, for an uninterpreted function f over two arguments, we can assume when $x_1 = x_2$, and $y_1 = y_2$, that $f(x_1, y_1) = f(x_2, y_2)$. Uninterpreted functions can be used in modeling a number of hardware blocks in the microprocessors, where their detailed functionality is not required, as long as they behave consistently in both the pipeline and the sequential reference model.

Integer: The mathematical type of integers, supporting arithmetic operations and comparisons. Although no hardware design truly supports unbounded integers, they can be useful for modeling abstracted representations of hardware designs.

Bit vector: Representing fixed-width groups of bits, with defined arithmetic, logical, and comparison operations. This is the most precise way to model data in a hardware design, but, as shall be seen, it can incur a high cost in verification effort.

Enumerated: Finite collections of objects. These can be useful for representing register identifiers, operation codes, and other data encoded in hardware with small bit fields.

Boolean: Single Boolean signal. These, of course, are widely used in hardware designs.

Tuples and records: Aggregations of other data types. We did not make use of these.

Arrays: These are useful for modeling register files, data memories, and other memory arrays.

One strength of UCLID5 is that these different data types can be combined in many different ways. A function can be defined having multiple arguments, each with different types, and yielding a value of yet another type. For example, we will model the logic determining whether a branch should or should not be taken as an uninterpreted function yielding a Boolean value, having as arguments an enumerated function code type and an uninterpreted condition code type. Similarly, arrays can be defined with arbitrary index and data types.

As we will see, these data types present a number of choices in creating a formal model, yielding different levels of abstraction. As a general rule, it is best to use the most abstract model possible that still captures the properties of the system that guarantee its correctness. We will see that the different variants of the pipeline require different levels of abstraction in their verification.

Given a combination of model and verification script, UCLID5 generates a set of *verification conditions*, expressed as formulas in a logic that supports the multiple data types—known as *theories*—used in the model. Typically, these formulas are negations of the properties that the user is trying to verify. It then invokes a *satisfiability modulo theories* (SMT) solver. UCLID5 can make use of several different SMT solvers. For this work, we used the Z3 solver developed at Microsoft Research [8].

When invoked by UCLID5, the SMT solver can return three different answers. First, it can determine that the formula is *unsatisfiable*. Given the negated form of the formula, this indicates that the desired verification condition holds. Second, it can determine that the formula is *satisfiable*. It provides concrete values for all of the data elements (including the uninterpreted functions) appearing in the formula such that the formula holds. This typically implies that some verification condition failed. UCLID5 then uses the concrete values to generate a *counterexample*: a sequence of actions that could have occurred in the model that would violate a verification condition. These counterexamples are important indications that either 1) there is a true error in the design, 2) that the model was inaccurate or too abstract, or 3) that the verification condition was not expressed properly. Finally, the verifier can declare the formula is *indeterminate*, indicating that, while it could find no satisfying solution, it also could not prove that the formula is unsatisfiable. This typically indicates that the model is too complex or requires more sophisticated reasoning than the SMT solver can provide.

1.3 Outline of Report

The Bryant-O'Hallaron textbook [5] presents the Y86-64 instruction set and a pipelined implementation of a Y86-64 processor. It also gives homework problems that require modifications to that pipeline, yielding a total of seven variants. This report describes the steps we took to verify all of these variants. Section 2 provides a brief description of the Y86-64 architecture and its implementations. Section 3 describes how we constructed UCLID5 models of the processors. Section 4 describes the efforts required to verify these models. Section 5 explores some of the performance

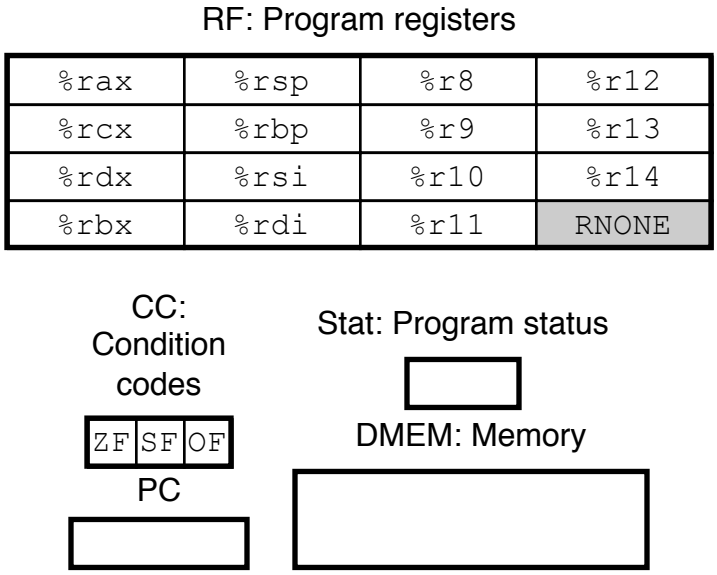

Figure 1: **Y86-64 programmer-visible state.** As with x86-64, programs for Y86-64 access and modify the program registers, the condition code, the program counter (PC), and the data memory. The additional exception status word is used to handle exceptional conditions. (From [5, Fig. 4.1].)

aspects of UCLID5 when verifying these processor designs. Section 6 describes related work, including our efforts to verify an earlier version of these processor designs with an earlier version of UCLID [1].

All of the experimental results in this report were measured using UCLID5 version 0.9.5 [9, 16] and with Z3 version 4.5.0 as the SMT solver. All times are the total number of CPU seconds on an eight-core 2.20 GHz Intel Xeon E5-1660. Z3 gains a small benefit from multi-core parallelism, and so these run times are slightly shorter than the total amount of CPU time used.

2 The Y86-64 Processor

The Y86-64 instruction set architecture adapts many of the features of the Intel64 instruction set (known informally as "x86-64"), 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. Whereas the x86-64 ISA has 16 program registers. Y86-64 supports only 15, eliminating register `%r15`.[1] This reduction allows a four-bit field to encode either the register or the case where no register is addressed, e.g., as a destination identifier for a register update. We refer to this sixteenth value as `RNONE`. As will be seen, this value creates special challenges in both modeling and verification. We refer to the fifteen registers collectively as the register file RF. Of these registers, only the stack pointer `%rsp` 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 a status register Stat to indicate whether the program is executing normally, or that an exception has occurred. Exceptional conditions include when an invalid instruction or data memory address is referenced, an invalid instructions is fetched, or a halt instruction is executed.

Figure 2 illustrates the instructions in the Y86-64 ISA. These instruction range between one and ten bytes long. Simple instructions such as `halt` and `nop` (No Operation) require only a single byte. The x86-64 data movement instruction `movq` is split into four cases: `rrmovq` for register-register, `irmovq` for immediate-register, `rmmovq` for register-memory, and `mrmovq` for memory to register. Memory referencing uses a register plus displacement address computation.

Figure 2 shows three instruction types: `OPq`, `jXX`, and `cmovXX` that represent families of related instructions, according to the value of the field labeled `fn`, as illustrated in Figure 3. The `OPq` instruction shown in the figure represents four different arithmetic and logical operations. 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, with different branch conditions. Branching is based on the setting of the condition codes by the arithmetic instructions. The `cmovXX` instruction shown in the figure represents seven different conditional move instructions. These instructions have register `rA` as source operand and `rB` as destination. As can be seen, the `rrmovq` instruction is a special case of a conditional move, where the move condition always holds.

The `pushq` and `popq` instructions push and pop 8-byte words onto and off of the stack. As with x86-64, pushing involves first decrementing the stack pointer by eight 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 eight.

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 instruction is not part of the standard Y86-64 instruction set, but is given to implement

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

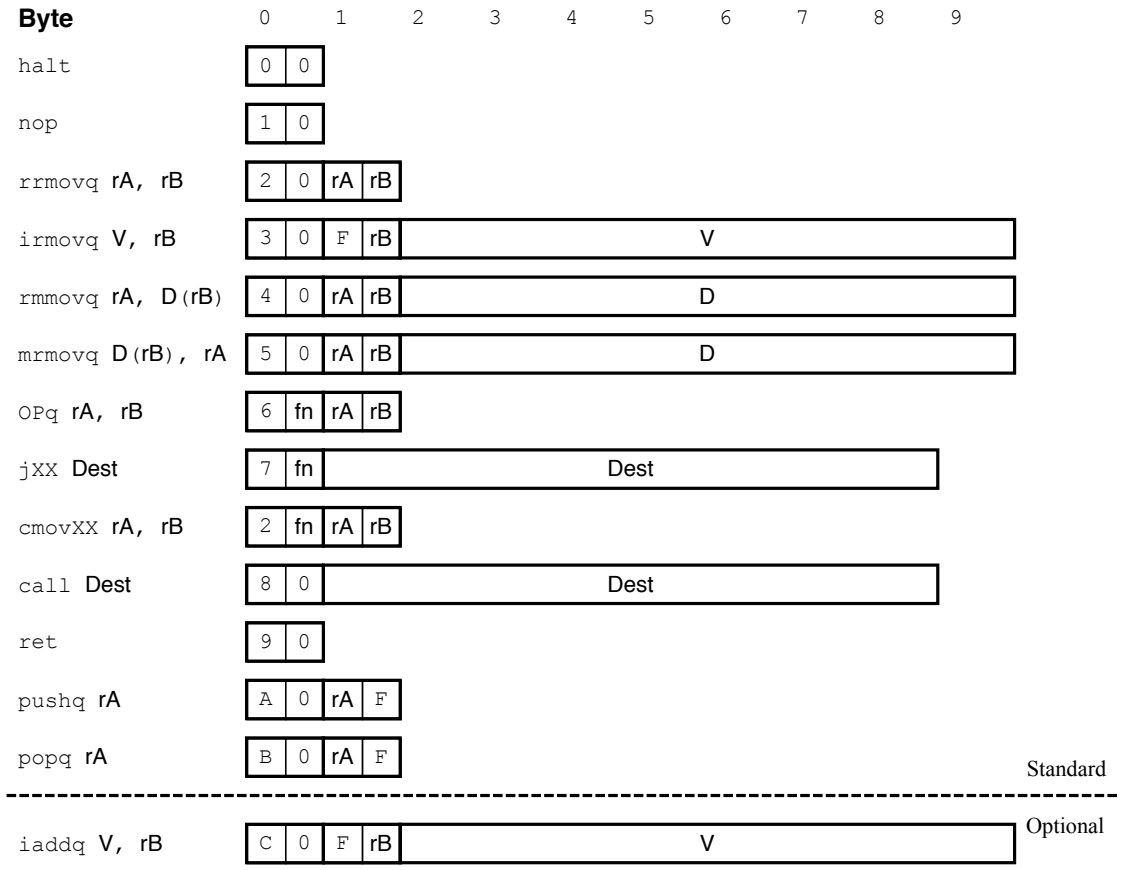

Figure 2: **Y86-64 instructions.** Instruction encodings range between one and ten bytes. An instruction consists of a one-byte instruction specifier, possibly a one-byte register specifier, and possibly an eight-byte constant word. All numeric values are shown in hexadecimal. (From [5, Fig. 4.2].)

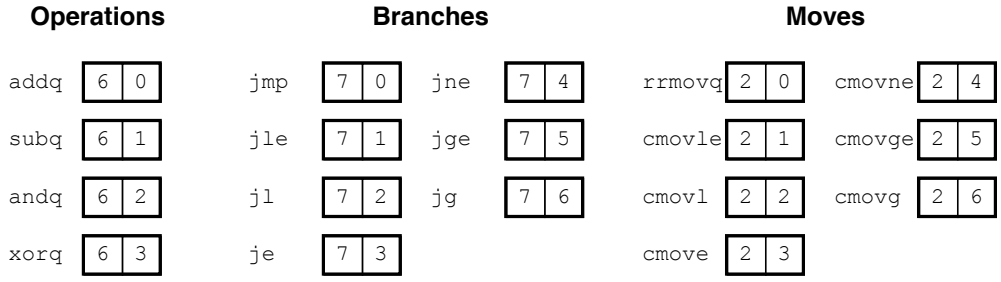

Figure 3: **Y86-64 function codes.** Function codes specify the ALU operation, the jump condition, or the conditional move condition. All numeric values are shown in hexadecimal. (From [5, Fig. 4.3].)

as a homework exercise in [5]. One of our variants of Y86-64 also implements this instruction. The `iaddq` instruction adds an immediate value to the value in its destination register.

We see that Y86-64 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.
- Condition codes control conditional branching and conditional moves.
- Some instructions (`pushq` and `popq`) 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.

2.2 Sequential Implementation

Figure 4 illustrates SEQ, a sequential implementation of the Y86-64 ISA, where each cycle of execution carries out the complete execution of a single instruction. The only state elements are those that hold the Y86-64 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, ten bytes are fetched from memory (not all are used), and address of the next sequential instruction is computed by incrementing the PC. 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.

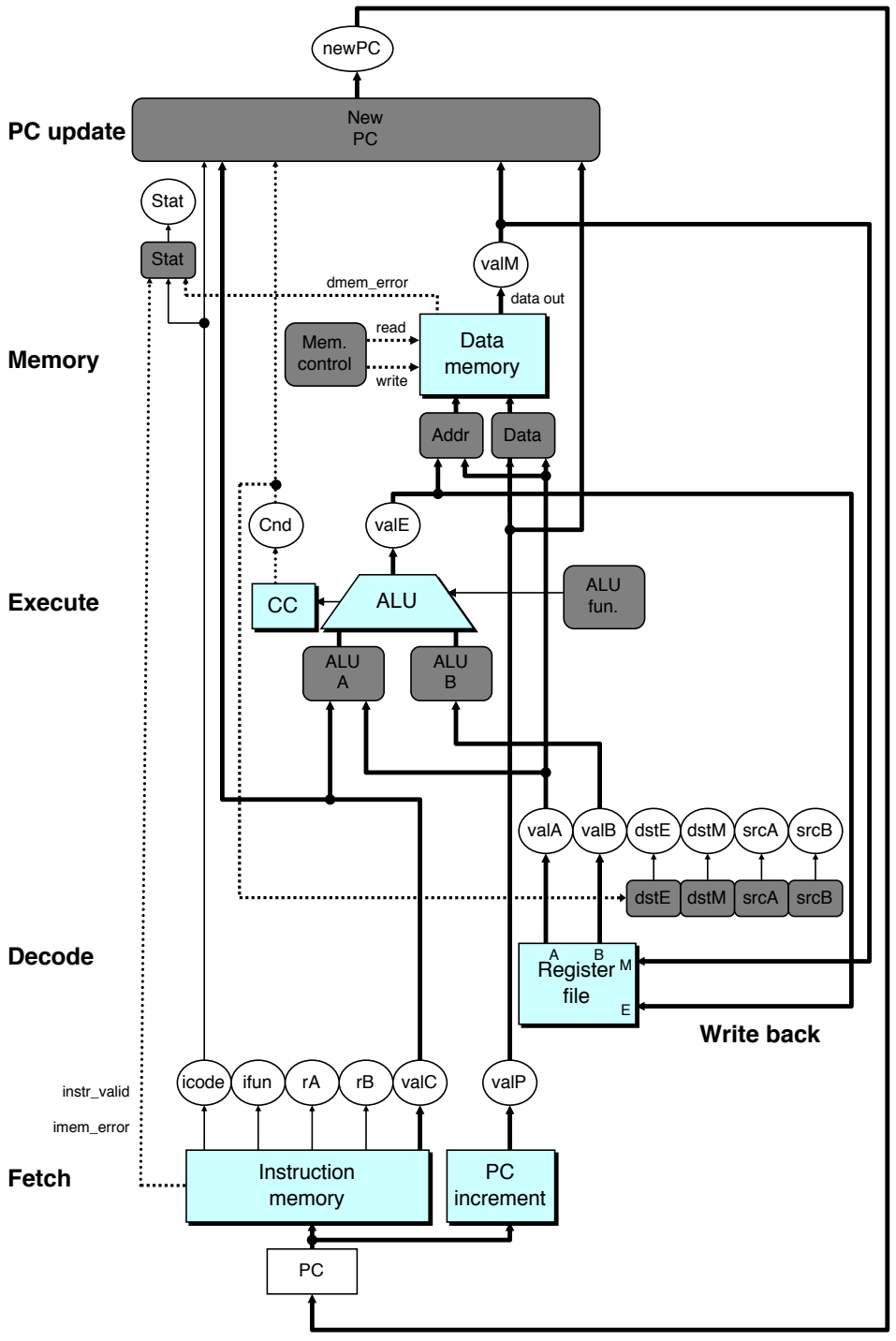

Figure 4: **Hardware structure of** SEQ**.** This design was used as the sequential reference version. (From [5, Fig. 4.23].)

2.3 Pipelined Implementation

Figure 5 illustrates a five-stage pipeline, called PIPE, implementing the Y86-64 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 are 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 variants of PIPE. The basic implementation STD is illustrated in the figure and described in detail in [5]. The others are presented in the book as homework exercises, where our variants 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 5. 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 `iaddq` instruction listed as optional in Figure 2. Verification is performed against a variant of SEQ that also implements this instruction.

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 multiplexer selecting between the two sources. This requires splitting the execution of the `popq` instruction into two cycles: one to update the stack pointer and one to read from memory.

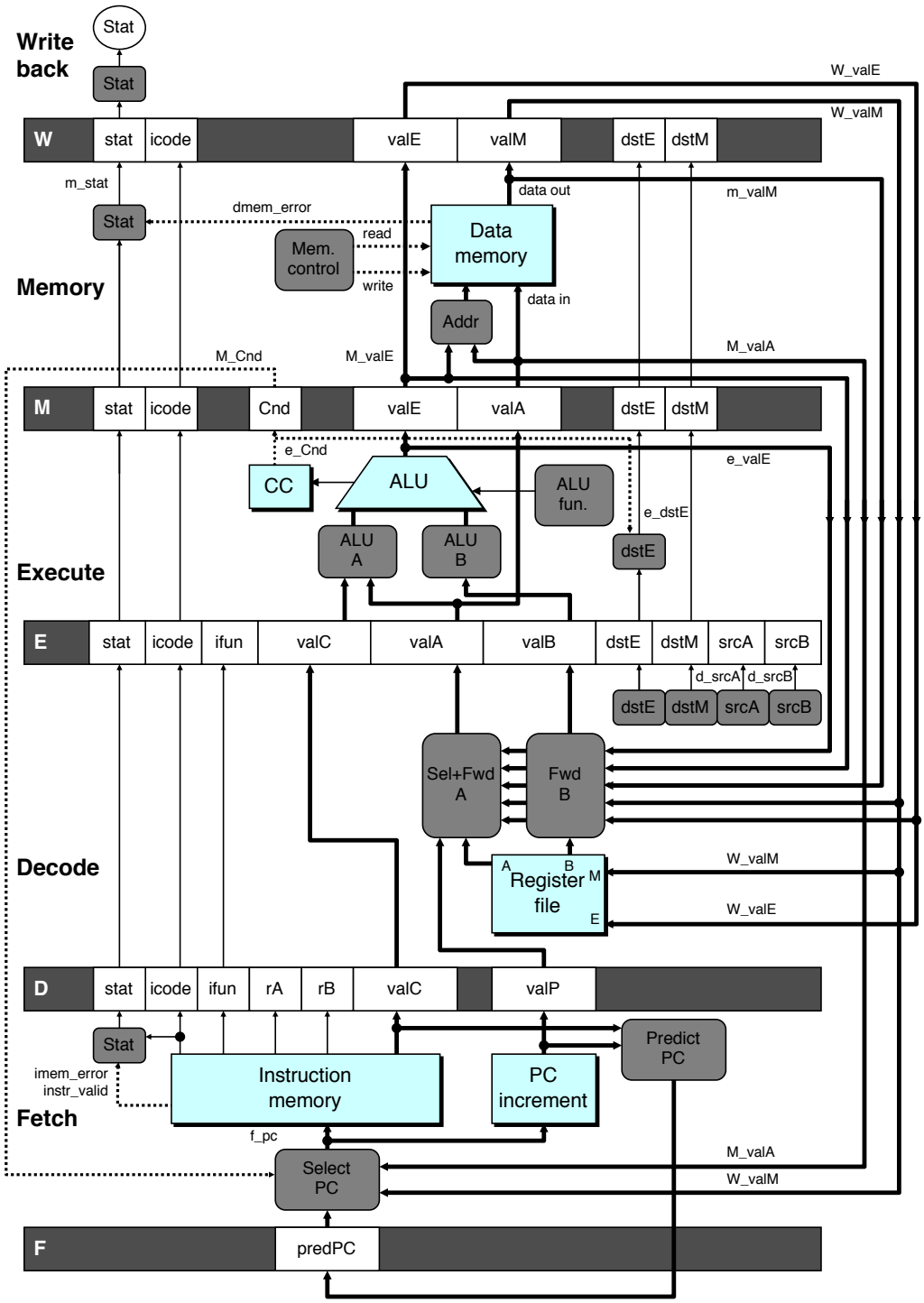

Figure 5: **Hardware structure of PIPE, the pipelined implementation to be verified.** Some of the connections are not shown. (From [5, Fig. 4.52].)

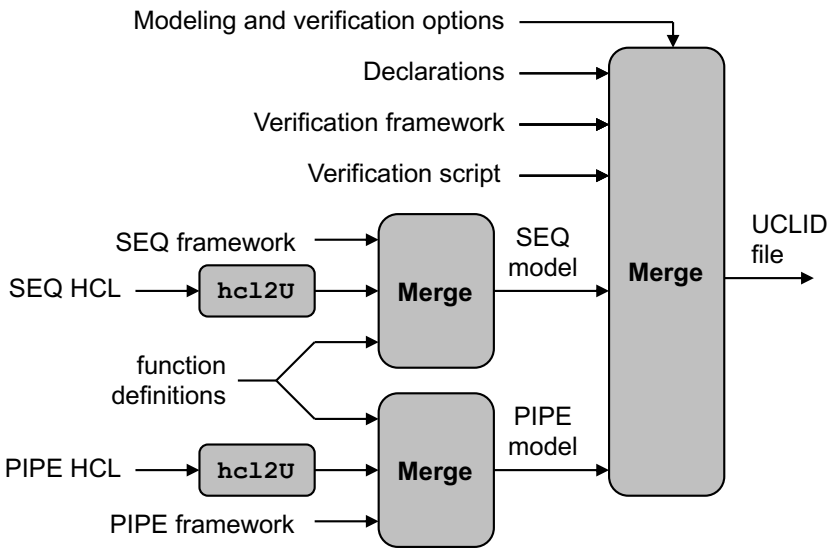

Figure 6: **Generating complete verification file.** The process extracts the control logic directly from their HCL descriptions and can generate UCLID5 files with different modeling and verification choices.

2.4 Verification Task

For our verification, the SEQ processor will serve as a reference version of the Y86-64 ISA. Of course, there is a chance that SEQ is incorrect, but this is much more easily tested by conventional methods (e.g., simulation) than are the more complex pipelined implementations. Our task is then to determine whether or not SEQ and (all seven variants 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 specific to PIPE.

The control logic for both SEQ and PIPE is described in a simple hardware description language, called *HCL*, for "Hardware Control Language." Translators had previously been written from HCL to C to construct simulation models of the processors, from HCL to Verilog to construct versions suitable for generating implementations by logic synthesis, and from HCL to the earlier version of UCLID in our previous verification effort. By generating the control logic directly from a common representation, we maintain consistency between the simulation models, the synthesizable hardware descriptions, and the formal verification.

A). HCL description

```
## Select input A to ALU
word aluA = [
        icode in { IRRMOVQ, IOPQ } : valA;
        icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ } : valC;
        icode in { ICALL, IPUSHQ } : -8;
        icode in { IRET, IPOPQ } : 8;
        # Other instructions don't need ALU
];
```

B). Generated UCLID5 code

```
define gen_aluA() : common.word_t =
    (if ((icode == IRRMOVQ || icode == IOPQ)) then valA else if ((icode ==
        IIRMOVQ || icode == IRMMOVQ || icode == IMRMOVQ)) then valC else
      if ((icode == ICALL || icode == IPUSHQ)) then CONSTM8() else if ((
        icode == IRET || icode == IPOPQ)) then CONST8() else CONST8());
```

Figure 7: **Automatically generated** UCLID5 **code.** Each signal definition in HCL is translated into a UCLID5 macro.

3 Generating UCLID5 Models

Figure 6 shows the overall framework we developed for generating UCLID5 files, each describing a model and verification script to be evaluated. Generating models for the two processors involves combining control logic, expressed in HCL, with frameworks describing the operations of the functional blocks and the connections between them. Definitions of the functional blocks are expressed in a single file and duplicated in the two models, to ensure they will be consistent. These processor models are then merged with files defining the common data types, the overall system model, and the verification script. Finally, different options for the models and the verification are selected to generate a file for a specific verification task. All of these steps were performed by programs: a translator HCL2U translating HCL signal definitions into UCLID5 macro definitions, and a Python program to perform the merging and option selection.

3.1 Generating UCLID5 from HCL

Figure 7 shows an example of how the control logic described in UCLID is translated into UCLID5. An HCL file contains a series of signal definitions, each defining how some logic block operates. For example, Figure 7A describes the logic block in SEQ labeled "ALU A" in Figure 4, defining the input to the A input of the ALU. HCL supports *case expressions* indicating a sequence of possible choices, and the result that should be returned for the first matching choice. It also supports a set

membership test. The overall expression then specifies that the input to the ALU should be one of the following, depending on the instruction code field of the current instruction: `valA`, the result of reading from the register file; `valC`, the data field extracted from the instruction; or constant values $+8$ or -8.

The HCL2U translator generates a UCLID5 macro for each signal definition. The translation is straightforward, but no attempt is made to make the result readable by humans. As the example shows, case expressions are translated into nested sequences of if-then-else expressions. Set membership testing is expanded into a disjunction of equality tests. Note also that HCL does not require a default case. In translating into nested if-then-else expressions, the final "then" value is replicated as the final "else" value. The code references macros `CONSTM8` and `CONST8`, which are defined elsewhere to encode the values used to represent -8 and $+8$.

Data Types and Functions

As mentioned earlier, a general rule is to use the most abstract model possible for a particular verification task. This ensures that the model captures all possible behaviors while reducing the potential for the SMT solver to waste its time tracking irrelevant details about the system. On the other hand, we were interested in exploring the different data types and modeling capabilities of UCLID5 and their impact on the verifier run times. In addition, the different variants of PIPE require different levels of precision in modeling the ALU operation, different restrictions on the initial pipeline state, and different numbers of flushing steps. The framework of Figure 6 allowed us to generate and test the many different combinations these choices created.

Some aspects of the modeling were common across all verifications:

- Instructions were modeled with an uninterpreted data type. The extraction of the different fields from an instruction were described by uninterpreted functions. Since the same functions are used in the SEQ and PIPE model, no further details about the exact instruction decoding are required for verification.

- Fields having a small number of possible values, such as instruction codes, function codes, register identifiers, and exception codes were modeled as enumerated types.

- All program data and addresses were modeled as a single type `word_t`. This type was defined to be either uninterpreted, integer, or 64-bit vector, as will be discussed later.

- The instruction memory was modeled as an uninterpreted function, mapping an address (of type `word_t`) to an instruction. This reflects the assumption that the program resides in a protected region of memory and therefore will not be modified during program execution. Indeed, PIPE is not designed to correctly execute self-modifying code.

- The condition codes are modeled with an uninterpreted data type, with uninterpreted functions describing how these are updated by an ALU operation, and how the setting of the condition codes determines the outcome of a branch or conditional move decision. This logic is the same in SEQ and PIPE, and so can be modeled at an abstract level.

For other aspects of the modeling, multiple options were explored:

- Data type `word_t` was defined to be type uninterpreted, integer, or 64-bit vector. Some pipeline variants could be verified with the data being uninterpreted, while others required more precise modeling.

- For the cases of integer and bit-vector data, there were multiple possible choices regarding the modeling of the ALU function, the PC increment logic, and comparison operations.

- The main memory could be modeled either using an array data type, or by treating the memory state as an uninterpreted value, using uninterpreted functions to represent the read and write operations. Because the pipelined implementations performs all memory operations in program order, the uninterpreted version sufficed. However, we also wanted to explore the performance implications of a more precise model and so tested both approaches.

The different levels of abstraction form a partial ordering among processor models. Informally speaking, one model is more abstract than another if it permits a wider range of behaviors. In particular, uninterpreted data types are more abstract than concrete ones: an uninterpreted value can be instantiated as an integer, a bit-vector, a real number, etc. Similarly, an uninterpreted function is more abstract than a precise, mathematical function.

The choice of data type—uninterpreted, integer, or bit vector—forms a partial order, with uninterpreted being more abstract than integers and bit vectors, but with the latter two being incomparable.[2]

For modeling ALU operations, we considered a number of alternatives, based on the required levels of precision for the different variants of the pipeline. These are described by the UCLID5 procedure definitions in Figures 8 and 9:

Uninterpreted: The ALU is defined as an uninterpreted function yielding a word as a function of the operation code (of type `op_t`) and two words. This sufficed for verifying pipeline variants STD, FULL, STALL, and LF.

[2]It may seem, offhand, that integers could be an abstraction of bit vectors. Every integer x can be mapped to an n-bit vector by the function $h(x) = x \bmod 2^n$. This mapping preserves the behavior of many standard arithmetic operations, including addition, multiplication, and negation. However, it does not preserve the behavior of equality and ordering operations. In addition, there are no integer operations corresponding to bit-wise logical operations.

A.) Uninterpreted

```
procedure alu_operate(op: op_t, valA: word_t, valB : word_t)
  returns (val : word_t)
  {
    val = common.base_alufun(op, valA, valB);
  }
```

B.) ALU Add zero

```
procedure alu_operate(op: op_t, valA: word_t, valB: word_t)
  returns (val : word_t)
  {
    case
      (op == ALUADD && valB == CONST0()) : { val = valA; }
      default : { val = common.base_alufun(op, valA, valB); }
    esac;
  }
```

C.) ALU Increment/Decrement

```
axiom (forall (x : word_t) ::
  common.base_alufun(ALUADD,
                     common.base_alufun(ALUADD,
                                        x,
                                        CONST8()),
                     CONSTM8())
    == x);
```

D.) ALU Add

```
procedure alu_operate(op: op_t, valA: word_t, valB: word_t)
  returns (val : word_t)
  {
    case
      (op == ALUADD) : { val = valA + valB; }
      default : { val = common.base_alufun(op, valA, valB); }
    esac;
  }
```

Figure 8: **Abstracted ALU models.** Some Y86-64 variants require partial interpretations of the ALU function.

```
procedure alu_operate(op: op_t, valA: word_t, valB: word_t)
  returns (val : word_t)
  {
    case
      (op == ALUADD) : { val = valA + valB; }
      (op == ALUSUB) : { val = valA - valB; }
      (op == ALUAND) : { val = valA & valB; }  // Bit vector only
      (op == ALUXOR) : { val = valA ^ valB; }  // Bit vector only
      default : { val = common.base_alufun(op, valA, valB); }
    esac;
  }
```

Figure 9: **Precise ALU model.** Bit-vector representations allow precise modeling of the ALU. Integer representations can precisely model addition and subtraction.

ALU Add zero: This version captures the property that $x+0 = 0$, but otherwise has uninterpreted behavior. This was required for verifying pipeline variants NT and BTFNT, where the ALU is used to pass the branch target through the execute stage in case the branch is taken.

ALU Increment/Decrement: This version attempts to capture the property that $(x+8)+-8 = x$. This property is required for verifying pipeline variant SW, due to the way it manipulates the stack pointer in its implementation of the `popq` instruction. This property cannot be expressed by modifying the definition of the ALU, since it describes a requirement on multiple applications of the function. Instead, it is expressed in UCLID5 as an axiom—an assertion that is imposed on the otherwise uninterpreted ALU function and provided as a constraint to the SMT solver. As will be discussed, however, we found that the SMT solver could not make effective use of this axiom, and so the verification with this level of abstraction was unsuccessful.

ALU Add: This version fully interprets the ALU behavior in the case of addition but uses an uninterpreted function otherwise. It can only be used when modeling words as either integers or bit vectors. It suffices for verifying all variants of the pipeline.

Precise: The operation of the ALU is modeled as precisely as UCLID5 permits: addition and subtraction are modeled precisely, as are the bit-wise logical operations when bit vectors are used. In addition, the program counter incrementing and the comparison operation (used in variant BTFNT to determine whether branch target is greater or less than the current PC) are modeled precisely. This version provided the opportunity to test how UCLID5 performs when given more precise models than are required.

The combination of data type and ALU abstraction can be visualized with the partial-order diagram shown in Figure 10, with the abstraction level increasing from top to bottom. In this

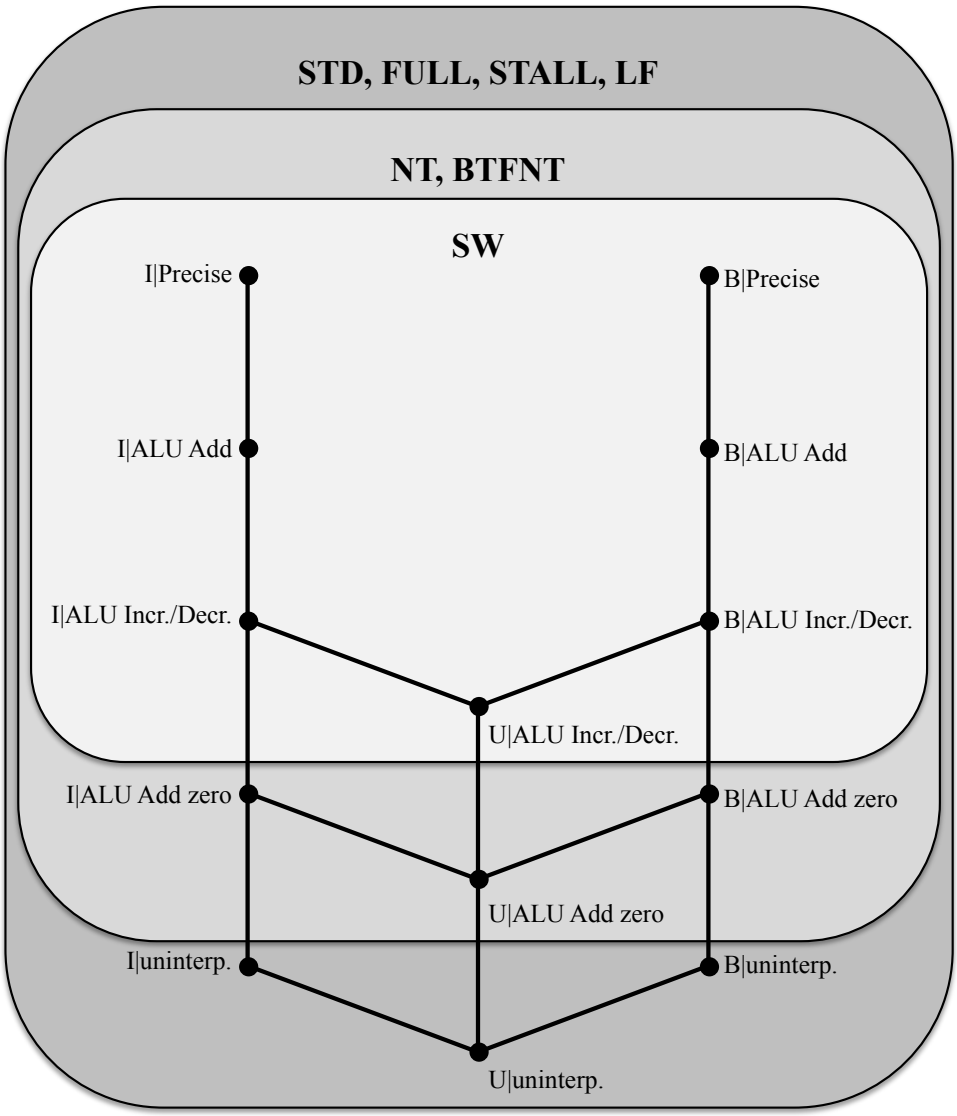

Figure 10: **Levels of abstraction for data and ALU modeling.** Data can be modeled as uninterpreted terms (U), integers (I), or bit vectors (B). ALU and other data operations can be modeled with levels of precision ranging from uninterpreted to precise arithmetic. Different variants of the pipeline require different levels of modeling precision.

```
procedure do_execute()
  modifies aluA, aluB, alufun, e_valE, set_cc, e_valA,
           cc, e_Cnd, e_dstE;
{
  aluA = gen_aluA();
  aluB = gen_aluB();
  alufun = gen_alufun();
  call (e_valE) = alu_operate(alufun, aluA, aluB);
  set_cc = gen_set_cc();
  if (set_cc) {
    cc = common.cc_fun(alufun, aluA, aluB);
  }
  e_valA = gen_e_valA();
  e_Cnd = cond_fun(E_ifun, cc);
  e_dstE = gen_e_dstE();
}
```

Figure 11: UCLID5 **description of execute stage operation in pipeline.** Expressions of the form `gen_`*XXX* reference definitions generated by HCL2U

diagram, we see that uninterpreted data (center) is more abstract than either integer (left) or bit-vector (right) representations. Each vertical chain represents different levels of abstraction in the ALU model, ranging from uninterpreted functions (most abstract) to a precise modeling of the ALU (least abstract). The two most precise ALU models only apply to integer and bit-vector data. The nested gray boxes indicate which models were suitable for which variants of the pipeline.

We see that, in principle, an uninterpreted data type should suffice for all of the variants. However, given that the SMT solver could not make effective use of the axiom of Figure 8C, we can see that variant SW required modeling the data as either integer or bit vector, and with a precise interpretation of addition.

3.2 Model Construction

As is seen in Figures 4 and 5, the processor designs consist of state elements, blocks with fixed functionality, and blocks with functionality specified in HCL. These components are organized as stages. In the case of PIPE, there are pipeline registers separating the stages.

UCLID5 supports two execution models with respect to how state is updated. For modeling software, it follows a *serial* model in which assignment statements are executed in sequence, each updating some portion of the system state. For modeling hardware, it follows a *state-machine* model in which there are two values associated with each state variable: its *present* value and its *next* value. The latter is specified by appending a single quote to the name of the variable.

For example, when state variable `var` is used in an expression, `var` refers to its current value, and `var'` refers to its next value. These are known as the "unprimed" and "primed" versions of the variable, respectively. Assignment statements define how the next state values are computed based on the current values of the state variables, plus possibly the next-state values of other state variables. Conceptually, the state machine operates by first computing all next-state values and then synchronously updating all state variables so that their next values become their present values.

Although it may seem natural to model the processors as state machines, the need to identify whether an expression references the present or next value of a state variable proved problematic when translating the control logic expressed in HCL. For example, the HCL code shown in Figure 7A defines how signal `aluA` is computed in SEQ in terms of values that are computed earlier in the same clock cycle, e.g., `icode`, `valA`, and `valC`. The generated UCLID5 code should therefore refer to these signals in their primed forms. The corresponding block in the HCL for PIPE, on the other hand, performs the identical computation, but in terms of pipeline register states `E_icode`, `E_valA`, and `E_valC`. The generated UCLID5 code should therefore refer to these signals in their unprimed forms. Other blocks reference signals that would require the generated UCLID5 code to include a mixture of primed and unprimed variables.

Instead of trying to generate HCL code that references the appropriate form of each state variable, we adopted a mixed strategy, where a processor is described as a state machine, but with the next-state computation expressed in terms of procedures. When using this style, UCLID5 treats assignments within procedures to be to the next-state values for the variables, but makes use of the ordering of assignments to determine whether a reference to a state variable in an expression is to its current or next value. So, for example, within a procedure, the sequence

```
varA = varA + varB;
varB = varA + varB;
```

is equivalent to the state-machine computation

```
varA' = varA + varB;
varB' = varA' + varB;
```

With this approach, we had to make sure the ordering of procedure calls and assignments respected the flow of signals through the combinational logic of the circuit.

To model the processors in UCLID5, we wrote a procedure for each stage invoking procedures, instantiating macros, and using UCLID5 statements to compute the values of the signals in that stage and possibly update state elements. Figure 11 shows an example of such a procedure, defining the operation of the pipeline execute stage. Comparing this procedure with the execute stage logic in Figure 5, we can see that it describes how the different signals should be computed. An expression of the form `gen_`*XXX*`()` instantiates the macro generated by HCL2U for signal *XXX*. We also see that the signal `e_valE` is computed according to one of the ALU functions defined in Figures 8 and 9. The condition codes are updated according to the uninterpreted function

`common.cc_fun`. The ordering of these computations follows the propagation of the signals in the stage.

To assemble the model for an entire processor, the stage procedures must be invoked in an order that reflects the order in which combinational logic signals would flow among the stages. For the case of SEQ, this follows the steps of instruction execution: fetch, decode, execute, memory, and writeback. For the case of PIPE, the pipeline registers cause the normal instruction flow to advance only one stage at a time. However, there are several cases where a signal is computed in one stage and then flows combinationally to a computation in another. As an example from Figure 5, the signal `m_valM` carries the data read in the memory stage. This feeds into the forwarding logic in the decode stage.

Overall, these combinational dependencies impose the following ordering constraints among the stages (in some cases, only for specific pipeline variants):

writeback → decode: To multiplex two data sources from writeback to the register port and forwarding logic of decode in variant SW. (See [5, Problem 4.58].)

memory → execute: For forwarding memory data in variant LF. (See [5, Fig. 4.70].) Also, to disable updating of the condition code register when a memory exception occurs.

memory → decode: For forwarding data from the memory stage.

execute → decode: For forwarding data from the execute stage.

These individual constraints impose an overall set of constraints: writeback → decode and memory → execute → decode. Based on this, we ordered the stage computations as writeback, fetch, memory, execute, and decode. These were then followed by procedures to update the control logic and to update the pipeline registers.

Figure 12 shows the UCLID5 representation of a pipeline register, in this case one capable of storing data of type `word_t`. This register is a generalization of the one shown in Figure 4.65 of [5]. It supports a number of different modes, required both for operating the pipeline and for performing Burch-Dill verification. The possible new state values for the register are:

Input value: This is the normal register operation.

Old value: This occurs when the pipeline stalls.

Empty value: This occurs when a bubble is injected into the stage.

Initial value: This is the initial state for verification, typically an uninterpreted value to indicate that it can be arbitrary.

```
procedure word_register(initialize : boolean,
                        stall : boolean,
                        bubble : boolean,
                        in_value : word_t,
                        empty_value : word_t,
                        init_value : word_t,
                        old_value : word_t)
  returns (val : word_t)
{
  if (initialize) {
    val = init_value;
  } else {
    if (stall) {
      val = old_value;
    } else {
      if (bubble) {
        val = empty_value;
      } else {
        val = in_value;
      }
    }
  }
}
```

Figure 12: UCLID5 **definition of pipeline register.** On each step, a register can either initialize, stall, inject a bubble, or load its input.

Two special requirements for Burch-Dill verification require extending the frameworks for SEQ and PIPE. First, there must be a way to transfer the values of the architectural state elements from PIPE to SEQ prior to running one step of SEQ's normal operation. This was implemented by defining an input signal `proj_impl` to the SEQ framework that will cause the values for the program counter, register file, data memory, and status register to be loaded from a set of module inputs to become the SEQ state.

As mentioned earlier, Burch-Dill verification requires introducing a flushing mechanism into the pipeline. Flushing requires stopping the fetching of new instructions while completing those already in the pipeline. Toward this end, the pipeline framework was augmented with an input control signal `force_flush`. Flushing is implemented by injecting a bubble into the decode stage for each cycle until the pipeline is empty. This causes `nop` instructions to be dynamically injected, while setting the status register to `SBUB`, indicating a pipeline bubble. This bubble-injection capability had already incorporated into the pipeline's control logic to handle cases encountered in normal program execution, such as waiting until a return address can be popped off the stack when executing a `ret` instruction.

A few points about flushing are a bit subtle and required several iterations to get working correctly. First, flushing should not cause a stall in the fetch stage, causing the program counter (shown in the F pipeline register of Figure 5 as `predPC`) to stay at a fixed value. That would prevent a `ret` or `jXX` from setting the program counter to the return or jump destination. Second, variant SW involves stalling the pipeline for a cycle to dynamically convert a `popq` instruction into a two-instruction sequence. Flushing should be disabled for one cycle when this occurs. Even with these subtleties, the extensions required to support flushing are very small.

Overall, the model for each variant of PIPE required around 650 lines of UCLID5 code to describe the pipeline framework and 930–1030 lines of code generated from the HCL files (the HCL files were 360–400 lines each.) The model for SEQ required around 220 lines of UCLID5 code for the framework and 81 lines generated from the HCL (with the HCL file being 217 lines.)

3.3 Modifications to Processors

Our initial plan was to use unmodified versions of the HCL files in our verifications. Unfortunately, this proved to be difficult, and so we made minor modifications, as described below.

In attempting the verifications, we encountered difficulties with instructions having register identifier `RNONE` as a source or destination operand. Such cases cannot be generated by our Y86-64 assembler, and therefore these cases had never been part of the simulation tests. But, referring to Figure 2, it is indeed possible to have instructions with fields rA or rB set `0xF`, the code for `RNONE`. Since our UCLID5 models had `RNONE` as one of the possible values for the enumerated type of register identifiers, these cases also arise during verification.

Three different options for how to handle these cases were considered:

- *Ignore them.* We could restrict the verification to exclude cases where RNONE was an operand for an instruction. This could be done in the code for the fetch stage, using UCLID5 `assume` statements to exclude instructions with improper register operands. But, it also ignores the fact that these instructions could genuinely arise in program binaries.

- *Ban them.* We could modify the fetch-stage logic to generate an invalid instruction exception when it encounters an instruction having an invalid register operand. This would be a good engineering choice, since it's generally a good idea to eliminate ambiguous cases. However, it would have required significant modifications to the logic in the fetch stage of the pipeline.

- *Define them.* We instead chose to define RNONE as a "pseudo-register" that contains constant value 0. Writes to it have no effect. This approach could be implemented by minor modifications of the HCL for the pipelines to make sure that data forwarding does not occur when the destination register has identifier RNONE.

In future iterations of the processor designs, we will revisit these choices. Overall, the "ban them" approach seems the most rigorous.

4 Verification

Complete verification of a processor includes: 1) Burch-Dill verification of the correspondence between the pipeline and a reference version, 2) verification of the invariance of any restrictions on the initial state, and 3) verification of liveness. The first step requires the most effort, both human and computer.

4.1 Burch-Dill Pipeline Verification

Our task is to prove that SEQ and PIPE would give the same results on every possible instruction sequence. Burch and Dill's approach [6] involves performing two symbolic simulations and then checking 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 UCLID5 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 both the architectural state elements and 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. Typically, n is set large enough to ensure that any partially executed instructions have completed.

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 state s. These are recorded in additional state variables in the UCLID5 model.

SaveAP(s)**:** Save the values of the *architectural* state elements in PIPE as a state s.

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

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

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 S. We then say that $\alpha(P_0) = S$. 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:

$$\sigma_a \doteq \text{Init}(P_0),\ \text{Pipe},\ \text{Flush}(n),\ \text{SaveAP}(S^a) \quad (1)$$
$$\sigma_b \doteq \text{Init}(P_0),\ \text{Flush}(n),\ \text{Xfer},\ \text{SaveS}(S_0^b),\ \text{Seq},\ \text{SaveS}(S_1^b) \quad (2)$$

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^b, runs one step of SEQ, and saves that state as S_1^b. In terms of abstraction function α, we can see that the three state values correspond to:

$$\begin{aligned} S^a &= \alpha(Pipe(P_0)) \\ S_0^b &= \alpha(P_0) \\ S_1^b &= Seq(\alpha(P_0)) \end{aligned}$$

Step	pipe_A	pipe_B	SEQ
0	Operate	—	—
$1, 2, \ldots, n$	Flush	Flush	—
$n+1$	SaveAP	—	Xfer
$n+2$	—	—	SaveS + Operate
$n+3$	—	—	SaveS
$n+4$	Check		

Figure 13: **Steps in performing Burch-Dill verification.** State machines representing two copies of PIPE and one copy of SEQ are operated in parallel. The modeled pipeline requires up to n steps to flush. Entries '—' indicate steps where the indicated state machine remains in its current state.

where function Seq (respectively, $Pipe$) computes the state transformation by one step of SEQ, (resp., PIPE.)

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:

$$S^a = S^b_1 \ \vee \ S^a = S^b_0 \tag{3}$$

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.

4.2 Implementing Burch-Dill Verification with UCLID5

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 [16], although we chose to organize the sequence of control steps differently. 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 as UCLID5 variables and then verify assertions about these variables.

Our symbolic simulation captures the two sequences shown as Equations 1 and 2. These are expressed as parallel runs of two copies of the PIPE model, labeled `pipe_A` and `pipe_B`, plus one copy of SEQ, as shown in the table of Figure 13. Both copies of PIPE start with the same initial

```
invariant correspondence :
(
  step > nflush+3
  && pipe_state_ok0
) ==>
  ((S_stat_b0 == SAOK ==> S_pc_a == S_pc_b0)
    && S_rf_a == S_rf_b0
    && S_cc_a == S_cc_b0
    && S_mem_a == S_mem_b0
    && S_stat_a == S_stat_b0) ||
   ((S_stat_b0 == SAOK ==> S_pc_a == S_pc_b1)
    && S_rf_a == S_rf_b1
    && S_cc_a == S_cc_b1
    && S_mem_a == S_mem_b1
    && S_stat_a == S_stat_b1);
```

Figure 14: **Verification Condition.** This check ensures that PIPE operation is consistent with SEQ operation.

state, using symbolic constants and uninterpreted functions to encode the set (or more typically, a superset) of the possible pipeline states that the pipeline may encounter. This is an important distinction between the symbolic simulation of Burch-Dill and traditional simulation-based testing. Burch-Dill requires operating the machine over all possible states, but only for a short simulation sequence. Traditional simulation involves starting the system in a reset state and then running it for many cycles in order to exhibit its range of possible states.

When modeling hardware, UCLID5 supports a state-machine model, where new values of the state variables are computed, and then all state elements are updated simultaneously. One consequence is that changing a control signal (e.g., to start flushing) takes a full cycle in order for the changed new state to become the current value of the signal. For a verification that requires n flushing steps, the overall sequencing of symbolic simulation requires $n + 4$ steps. It exploits a feature of the UCLID5 language, where, at any step an individual module can remain in its current state (indicated by an entry '—' in the table of Figure 13), or it can operate for one step. During the simulation, the (symbolic) state of either `pipe_A` or SEQ is recorded, and the correctness conditions are expressed in terms of these values.

Figure 14 shows the UCLID5 representation of the correctness condition. It is expressed as an invariant, meaning that it should hold for every step of the symbolic simulation, but the antecedent expression `step > nflush+3` implies that the correspondence need only hold for steps $n + 4$ and beyond. The antecedent condition `pipe_state_ok0` describes restrictions that may be imposed on the initial pipeline state, as will be discussed in the next section. The consequent expression captures the correctness condition of Equation 3. Note, however, that the consistency condition for the PC is imposed only for steps in which the processor starts in normal execution

```
define E_ok() : boolean =
    (E_stat == SAOK ==> (E_icode != IHALT && E_icode != IBAD))
 && (E_stat == SBUB ==> (E_dstM == RNONE  && E_dstE == RNONE && E_icode == INOP))
 && (E_stat == SHLT ==> (E_dstM == RNONE  && E_dstE == RNONE && E_icode == IHALT))
 && (E_stat == SINS ==> (E_dstM == RNONE  && E_dstE == RNONE && E_icode == INOP))
 && (E_stat == SADR ==> (E_dstM == RNONE  && E_dstE == RNONE && E_icode == INOP));

define M_ok() : boolean =
    (M_stat == SAOK ==> (M_icode != IHALT && M_icode != IBAD))
 && (M_stat == SBUB ==> (M_dstM == RNONE  && M_dstE == RNONE && M_icode == INOP))
 && (M_stat == SHLT ==> (M_dstM == RNONE  && M_dstE == RNONE && M_icode == IHALT))
 && (M_stat == SINS ==> (M_dstM == RNONE  && M_dstE == RNONE && M_icode == INOP))
 && (M_stat == SADR ==> (M_dstM == RNONE  && M_dstE == RNONE && M_icode == INOP));

define W_ok() : boolean =
    (W_stat == SAOK ==> (W_icode != IHALT && W_icode != IBAD))
 && (W_stat == SBUB ==> (W_dstM == RNONE  && W_dstE == RNONE && W_icode == INOP))
 && (W_stat == SHLT ==> (W_dstM == RNONE  && W_dstE == RNONE && W_icode == IHALT))
 && (W_stat == SINS ==> (W_dstM == RNONE  && W_dstE == RNONE && W_icode == INOP));

define ret_ok() : boolean =
    (E_icode == IRET ==> (D_stat == SBUB))
 && (M_icode == IRET ==> (D_stat == SBUB && E_stat == SBUB))
 && (W_icode == IRET && W_stat == SAOK ==>
       (D_stat == SBUB && E_stat == SBUB && M_stat == SBUB));

define pipe_ok() : boolean =
  D_ok() && E_ok() && M_ok() && W_ok() && ret_ok();
```

Figure 15: **Pipeline consistency predicate.** Some models required restrictions on the initial pipeline state.

mode, as indicated by status code SAOK. One design limitation of PIPE is that it does not set the program counter correctly when an exception occurs. For example, if an address exception occurs in the memory stage, the program counter will already have been incremented one or more times. Rather than redesigning the pipeline to properly support exceptions, we took the more straightforward route of qualifying the correctness condition.

4.3 Restricting the Initial Pipeline State and Invariant Checking

The matching condition of Equation 3 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 regular program register, even though the pipeline register contains a `nop` instruction. There can also be inter-instruction inconsistencies, for example when a `ret` instruction is followed by other instructions

```
define sw_ok() : boolean =
    (E_dstM == RNONE || E_dstE == RNONE)
 && (M_dstM == RNONE || M_dstE == RNONE)
 && (W_dstM == RNONE || W_dstE == RNONE);
```

Figure 16: **Pipeline single-write predicate.** The SW model also requires this restriction on the initial pipeline state.

in the pipeline rather than by at least three bubbles. These impossible states can be excluded by the verifier, but only if we can also prove that they can never actually occur.

A restriction on the pipeline state can be expressed as a predicate I over possible pipeline states. We say that restriction I is a *pipeline invariant* if it holds under all possible operating conditions. To ensure I is invariant, we must show that it holds when the processor is started in any reset state, 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 SaveP(s) saves the values of all 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 restrictions as simple as possible, since they place an additional burden on the user to formulate and to prove them to be invariant.

We found that several versions of PIPE could be verified without imposing any 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.

Other variants required imposing the restrictions expressed in Figure 15, which we refer to as the "consistency" property. These describe restrictions at each pipeline stage according to the possible status values for the stage. For example, when the execute stage has a regular instruction (status `SAOK`), it cannot be processing a `halt` instruction, nor can can it be processing an invalid instruction (expressed in UCLID5 as case `IBAD` for the enumerated type encoding possible instructions.) When the stage has a bubble, a `halt` instruction, an invalid instruction, or an address exception, the destination register identifiers must be `RNONE`, and the stage must have an appropriate instruction code. Similar restrictions hold for the other stages.

A final restriction in Figure 15 states that whenever a `ret` instruction is in the pipeline, there

Variant	Flush Steps	ALU Model	Initial State Restriction	Time (secs.)		
				Uninterp.	Integer	Bit vector
STD	5	Uninterpreted	None	56.5	62.6	173.8
FULL	5	Uninterpreted	None	69.3	49.4	168.6
STALL	7	Uninterpreted	Consistency	352.2	374.9	TOUT
NT	5	Add Zero	None	52.4	39.1	233.1
BTFNT	5	Add zero	None	42.4	76.1	190.5
LF	5	Uninterpreted	Consistency	53.9	83.5	194.2
SW	6	ALU Incr./Decr.	Con. + Single Write	Indeterm.	Indeterm.	Indeterm.
SW	6	ALU Add	Con. + Single Write	Impossible	14.3	TOUT

Figure 17: **Verification requirements and times for different PIPE variants and data types** The different variants require different modeling abstractions and verification procedures. TOUT: timed out after 3600 seconds; Indeterm.: UCLID5 could not establish the validity of the correctness condition; Impossible: the data type does not support the necessary operations.

must be bubbles in the preceding stages.

Finally, Figure 16 shows a pipeline state restriction that is specific to the SW variant, stating that at most one of the two destination registers in a stage can have a value other than RNONE. We refer to this as the "single write" property.

Considering all possible combinations of pipeline variant, data type, ALU model, pipeline state restriction, and memory model, we performed 88 different runs of UCLID5 to ensure all of the possible restrictions were inductive. The run times for these verifications ranged from 2.6 to 3.1 seconds.

4.4 Verification Results

Figure 17 summarizes the performance of UCLID5 when performing correspondence checking for the different variants of PIPE. As has been discussed, the different variants required different flush schedules, different ALU models, and different initial state restrictions. We show the performance for all three data types: uninterpreted, integer, and bit vector.

In each case, we attempted to verify the model with the most abstract possible model of the ALU. Unfortunately, the SMT solver was unable to make use of the axiom of Figure 8C, stating that the formulas arising from the models of variant SW using the "ALU Incr./Decr." version of the ALU were "indeterminate." Instead, we had to use the more precise "Add ALU," which precluded using uninterpreted data. Note also that the bit-vector models required significantly greater time, exceeding a one-hour time limit for two of the cases.

We explore the performance of UCLID5 for correspondence checking in more detail in Section

5. The results in Figure 17 indicate that each of our seven variants could be verified in less than six minutes of elapsed time. This indicates that the complexity of the Y86-64 pipelines is well within the range of practical verification for UCLID5.

4.5 Liveness Verification

As mentioned earlier, the fact that our correctness condition (Equation 3) 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. Detecting whether or not a processor "makes progress" is a bit subtle. For example, it cannot be based on simply ensuring that the PC gets incremented, since it is possible for the processor to execute an instruction that jumps to itself. Likewise, when the processor encounters an exception condition, the specification states that it should set the appropriate status flag and cease execution.

We devised a simple method for proving liveness that can be performed on the PIPE model alone. We added an additional state variable `completion_count` that counts the number of instructions that have been completed by the pipeline. This value is incremented every time an instruction passes through the writeback stage, as indicated by the status for the stage being something other than `SBUB`. (Status `SBUB` indicates a bubble in the stage.)

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 `SAOK`, as given by the following correctness condition:

```
invariant live :
   (step >= 5 && pipe_state_ok0 && pipe_stat0 == SAOK)
      ==> (impl.completion_count > 0);
```

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 pipeline variants. We must start the pipeline with the pipeline satisfying the consistency conditions of Figure 15. Otherwise, if the pipeline started with "hidden" jump instructions—instructions that cause a branch misprediction but have their status set to `SBUB`, then five cycles of execution would not suffice to ensure the completion 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.

Considering all possible combinations of pipeline variant, data type, ALU model, memory

Data	Flush Steps					
	5	6	7	8	9	10
Uninterpreted	48.4	48.1	87.5	84.6	110.0	79.8
Integer	58.0	89.8	86.0	80.8	87.1	117.8
Bit vector	169.2	216.0	576.9	1250.1	386.3	520.2

Figure 18: **Verification times for STD.** The times generally, but not always, depend on the number of flushing steps and the data type.

model, and initial state restriction, we performed 140 runs of UCLID5 to verify that all of these satisfied the liveness conditions. Run times ranged from 3.0 to 4.2 seconds.

5 Performance Analysis

We have seen that of the three tests—correspondence checking, pipeline state invariance, and liveness checking—only correspondence checking requires a significant amount of computational effort. We therefore focus on correspondence checking in our performance analysis.

As Figure 17 indicates, the run times for the verifications can vary widely. In attempting to identify the sources of these variations, we conducted a number of experiments. We found the performance defied any simple characterization. SMT solvers employ a number of heuristics in their search, and so their performance can depend on many performance tuning parameters that may or may not be ideal for a given verification run.

Overall, though, we can see from Figure 17 that 1) bit-vector modeling requires more computational effort than either uninterpreted or integer data, and 2) models having longer flushing requirements tend to have longer verification times. Figure 18 explores these two factors in verifications of variant STD, with different numbers of flushing steps and different data types. In all cases, the ALU is modeled by an uninterpreted function, the data memory is uninterpreted, and no restrictions are placed on the initial pipeline state. Surprisingly, even for this single variant, the run times do not follow a simple pattern. Clearly, bit-vector representations of data require more time than either uninterpreted or integer representations. The performance difference between integer and uninterpreted data is inconsistent, and not particularly large. Runs with more flushing steps tend to require more time than those with shorter ones, but even that does not always hold.

It was disappointing that several of the models using bit-vector representations of the data could not be verified, even with a timeout limit of one hour. For variant SW, this left only the model with integer data as having both the expressive power and a successful run. Bit-vector models are the most authentic representation of hardware, but they are much more difficult for SMT solvers.

As a test of bit-vector modeling, we explored the role of the bit vector width. Since the models

Arithmetic	Bit Width				
	4	8	12	16	20
Add ALU	40.7	55.6	75.7	1610.9	TOUT
Precise	53.0	109.8	177.7	486.2	TOUT

Figure 19: **Verification of variant SW with different bit vector widths.** The SMT solver does not scale well with respect to bit width.

require bit vectors of width 64, we considered whether using a smaller word size would make bit-vector modeling feasible. We attempted modeling variant SW with bit widths ranging down to 4 bits (the smallest that could represent values -8 and $+8$.) The performance is shown in Figure 19. This table shows two models for the ALU: modeling addition precisely, and modeling all four ALU operations precisely. Recall that the latter model includes precise models for the program counter incrementation. As the table shows, the verification is straightforward for small bit widths but quickly becomes problematic. Interestingly, the more precise model can lead to shorter run times. Both, however, timeout for vector widths of 20 and above.

As further evaluations of the impact of abstraction on the verification performance, Figures 20 and 21 quantify the performance impact of different choices of memory and arithmetic abstraction, respectively. Both are based on the ratio of the verification time for a more precise model versus the time for a more abstract model, while holding all other parameters constant. We would expect this ratio to be at least 1.0. Both display the results as histograms, using logarithmically-scaled bucket sizes. The ratios labeling the histogram buckets in the two figures are the (geometric) mean values for the buckets.

Figure 20 considers the data memory model—either as an array (precise) or as an uninterpreted function (abstract.) It shows the results for 54 combinations of pipeline variant, data type (either integer or uninterpreted), ALU model, and initial pipeline state restriction. As the histogram shows, there is quite a wide range of ratios, with many of them having ratios less than 1.0. The geometric mean of the ratios is 1.00, indicating that, on average, the choice of memory model had no impact on the runtime.

Figure 21 considers the level of precision when modeling integer arithmetic. The more abstract case models only the ALU's ability to perform addition, while keeping other ALU functions, as well as the program counter incrementing, and integer comparison operations uninterpreted. The more precise case models all of these aspects precisely. Overall, there were 22 combinations of pipeline variant, memory model, and initial state restriction. Again, we see a wide range of ratios, with many less than 1.0. The geometric mean was 1.02, indicating that, on average, the level of precision in the arithmetic model has little impact on the runtime.

These two sets of experiments highlight two important features about the Z3 SMT solver. First, run times can vary widely with just minor variations in modeling choices. As mentioned early, the many heuristic choices and tuning parameters in SMT solvers makes their search methods operate

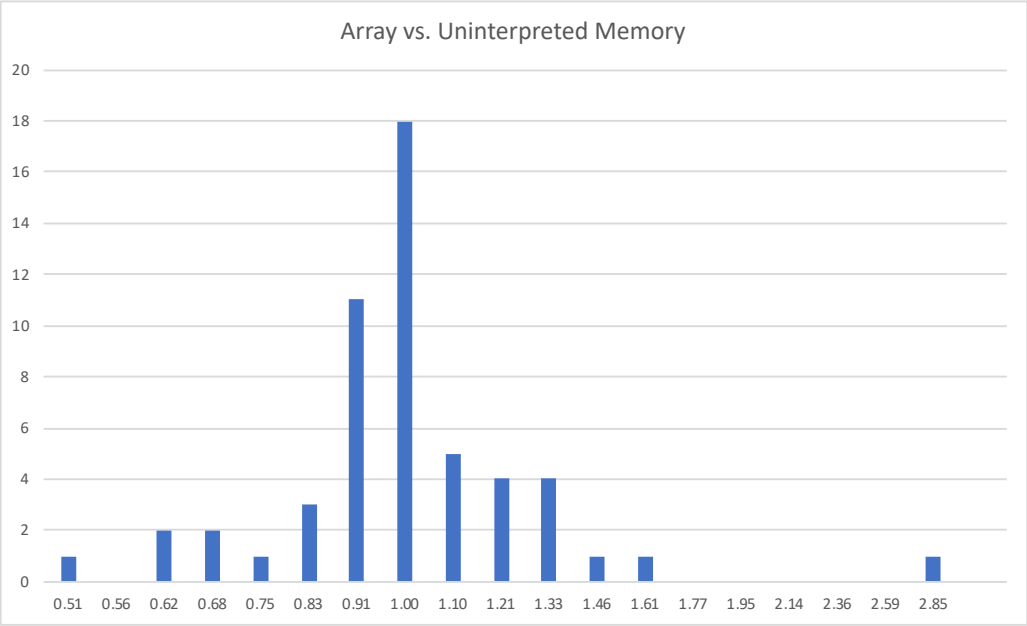

Figure 20: **Performance Impact of Memory Abstraction.** The histogram is based on the ratios of modeling the data memory as an array vs. with uninterpreted functions for 54 modeling combinations.

unpredictably. Second, Z3 is generally successful at exploiting modeling abstractions without external guidance. This is an attractive feature for a tool to be used by nonexperts.

6 Related Work

Burch and Dill's initial paper generated considerable interest in the verification research community. Our research group followed Burch and Dill's lead, developing microprocessor verification tools that operated at both with bit-level [17] and term-level [18] representations. We extended the ideas to superscalar [18] and even out-of-order processors [11, 12].

6.1 Earlier Verification of Y86

We performed an exercise similar to the one described in this report, based on an earlier set of processor models [3] and an earlier version of UCLID [2]. It is instructive to compare these two efforts.

The three editions of the textbook [3, 4, 5] have presented processor designs based on simplified version of the x86 instruction set. The first two editions featured "Y86," modeled after the 32-bit

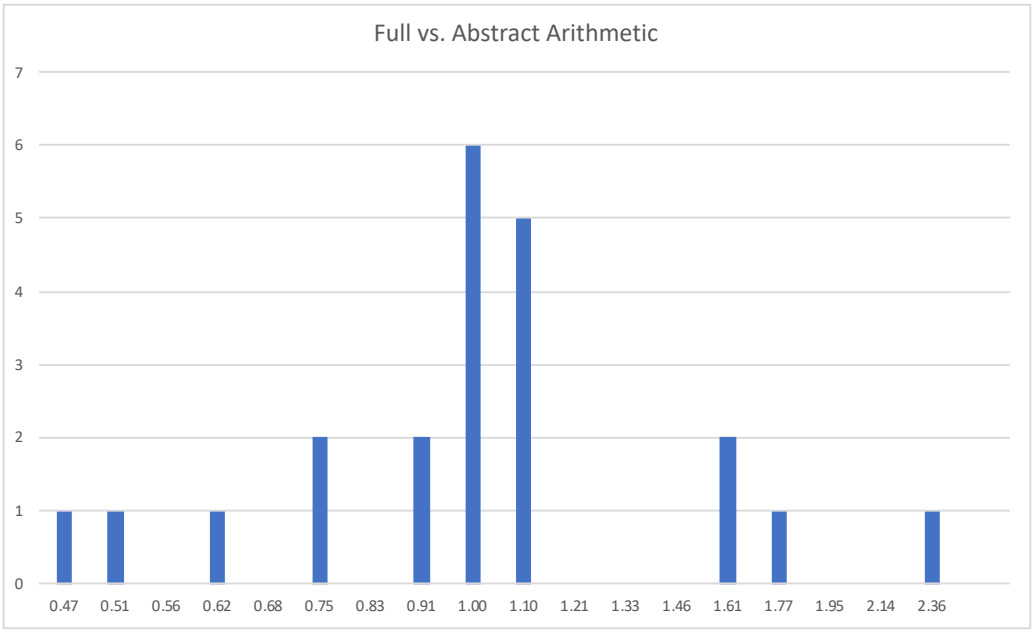

Figure 21: **Performance Impact of Arithmetic Abstraction.** The histogram is based on the ratios of modeling with precise integer arithmetic vs. only the addition property of the ALU for 22 modeling combinations.

IA32 instruction set, while the third featured "Y86-64," based on the 64-bit extension of IA32 commonly referred to as "x86-64." All used the same general instruction formats, as well as sequential and pipelined implementations.

In terms of verification, the biggest change was between the first and second editions. In the first, there was no attempt to support exceptions—if the processor encountered an invalid address or an invalid instruction, it had no defined behavior. In our original verification effort, we introduced rules for handling exceptions. We made appropriate extensions to the pipeline state and the control logic directly in the UCLID framework. In writing the second edition, these ideas were incorporated into the design presented in the textbook, including HCL descriptions of the exception-handling logic. The exercise of performing formal verification induced us to tighten the specification for future versions.

The original version of UCLID had considerably less expressive power than UCLID5. It supported only a limited integer data type that included addition by constant values. It did not allow the function types, procedures, macros, and modularity supported by UCLID5. As a result, greater preprocessing was required to construct the verification models than is shown in Figure 6.

In performing correspondence checking, the run times[3] for the earlier effort ranged from 235 seconds for STD up to 27,920 (7.75 hours) for SW. Even accounting for the difference in processor performance, we can see that the Z3 SMT solver provides a more powerful and more expressive platform for verification than did the one we implemented in UCLID.

Overall, many aspects of the previous exercise carried over to the efforts reported here. The seven different variants each required the same numbers of flushing steps, as well as similar initial state restrictions. There were more options in data types and ALU models, but the requirements for the different variants were similar.

6.2 Real-World Adoption

Surprisingly, there is little evidence of real-world processor verification that adopts both aspects of Burch and Dill's work—the combination of term-level modeling and the use of flushing to compute an abstraction function. Perhaps the closest is the use of the ISA-Formal system at ARM [13]. Their tool uses commercially available formal verification tools to perform correspondence checking, but they rely on bit-vector models of the hardware (extracted from Verilog representations) and they generate their abstraction functions manually. Their explanations for these choices are instructive.

ARM's choice of bit-vector modeling stems largely from the current industrial practice of using register-transfer level (RTL) languages for describing hardware systems. Since RTL is the basis for all synthesis, simulation, and testing tools, it is the natural choice for system designers. It would be difficult to maintain both a term-level and an RTL model of a system and to be certain

[3] Run times were measured as the sum of the user and system CPU time on a 2.2 GHz Intel Pentium 4.

the two are consistent. Our approach had the advantage that we had a unified representation of the control logic (in HCL) from which either Verilog or UCLID5 models could be generated. The ARM researchers also note that many of the logic blocks they use are difficult to characterize and vary from one model to another. This reduces their ability to abstract these blocks as uninterpreted functions, as we were able to do. Given that system designers are unlikely to adopt a more abstract representation of hardware, the best course for making formal verification practical seems to be to improve the performance of bit-vector decision procedures.

ARM's choice of manually generated abstraction functions is also understandable. They developed a system to extract Verilog models of instruction behaviors from their formal representation of the ARM instruction set. Verifying one instruction at a time allowed them to incrementally verify parts of the system as it evolved and to schedule the many verification runs in a large computer cluster.

7 Conclusions

We did not find any design errors in any of the seven processor variants. This is not surprising—our earlier effort found one error in one variant, and this was subsequently corrected. The design of the exception-handling logic for the second edition was taken largely from the UCLID models. The extension from 32 bits to 64 in the third edition required no changes to the conceptual design. In addition, we had an extensive suite of simulation tests, checking both individual instructions and possible hazards between them. Nonetheless, completing a formal verification eliminates the nagging doubt that there was some bug that escaped detection via simulation.

We found that UCLID5 is up to the task of verifying designs of this complexity. Its state machine model is a good match to synchronous hardware, and our inclusion of procedures within the state-machine model eliminated the need to track whether each variable should be referenced in its unprimed or primed form. We found that its data types met our needs, although the weak performance of Z3 on bit-vector models was disappointing.

We especially appreciated the counterexample generation capability of UCLID5. Although debugging a verification run proved difficult—far more so than debugging via simulation—it was helpful to be able to trace through the model and see what values passing through the pipeline were causing the formula to be invalid.

Although the computer industry has adopted a number of tools for formally verifying the individual components of a system, the ability to verify the overall correctness of a complex system remains a challenge. We continue to see processor verification as one of the great challenges for formal verification research.

References

[1] R. E. Bryant. Term-level verification of a pipelined CISC microprocessor. Technical Report CMU-CS-05-195, Carnegie Mellon University Computer Science Department, December 2005.

[2] 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.

[3] R. E. Bryant and D. R. O'Hallaron. *Computer Systems: A Programmer's Perspective, First Edition*. Prentice-Hall, 2003.

[4] R. E. Bryant and D. R. O'Hallaron. *Computer Systems: A Programmer's Perspective, Second Edition*. Prentice-Hall, 2011.

[5] R. E. Bryant and D. R. O'Hallaron. *Computer Systems: A Programmer's Perspective, Third Edition*. Prentice-Hall, 2016.

[6] 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.

[7] Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[8] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '08)*, LNCS 4963, pages 337–340. Springer, 2008.

[9] UCLID distribution. Available at `https://github.com/uclid-org/uclid`, 2018.

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

[11] 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.

[12] 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.

[13] Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi. End-to-end verification of processors with ISA-formal. In S. Chaudhuri and A. Farzan, editors, *Computer Aided Verification (CAV, '16)*, LNCS 9780, pages 42–58. Springer, 2016.

[14] S. A. Seshia and P. Subramanyam. UCLID5: Integrating modeling, verification, synthesis, and learning. In *Formal Methods and Models for System Design (MEMOCODE '18)*, 2018.

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

[16] P. Subramanyan and S. A. Seshia. *Getting Started with Uclid5 Alpha Release, version 0.9.5*. University of California, Berkeley, 2018.

[17] 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.

[18] 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.