Verification of Stack Manipulation in the Scalable Configurable Instrument Processor

J. Aaron Pendergrass

This article describes a formal approach to the specification and verification of a microprocessor design and presents a case study of applying this technique to the Scalable Configurable Instrument Processor. These activities greatly increase confidence that the microprocessor correctly implements its intended functionality. In addition, the formal specification of the processor's functionality provides the basis for future work verifying the correctness of software.

INTRODUCTION

Hardware specification and verification serve two major goals: increasing confidence that hardware is correct and facilitating verification of software at the instruction-set level. Errors in hardware design can lead to serious problems when software exercises some flaw in the hardware. When the Pentium processor implemented division in a way that deviated from the IEEE-754 specification,[1] it was a serious bug because it could cause correctly implemented software to fail unexpectedly. The discovery of this bug and the resulting expense to Intel motivated Advanced Micro Devices, Inc. (AMD) to use formal verification to ensure that their implementation was correct.

Although hardware errors are of great concern, system failures caused by software errors are far more common. The overwhelming majority of software crashes and security vulnerabilities stem from differences between the assumptions made by software programmers and the actual guarantees provided by the computing hardware. Programmers frequently have only an imprecise understanding of how instructions affect the computer's state. For example, the push operation of most processors is understood to append data to the top of the stack, which in the nominal case it does, but there are many exceptional cases in which a push operation may cause an error or some other behavior for which the programmer must account. For a program to be correct, it must be impossible for such exceptional states to arise during the execution of the program, or the programmer must have correctly predicted and handled all contingencies. Many techniques, such as testing or manual code analysis, exist for gaining confidence that a program is correct. Unlike most of these techniques, formal verification requires that the programmer explicitly state all assumptions made in the code and that all possible cases be handled. Hence, formal verification gives the greatest assurance of program correctness but requires a precise and rigorous definition of the computing hardware.

This article describes our experience formally specifying and verifying aspects of the design of the Scalable Configurable Instrument Processor (SCIP), a lightweight, low-power processor designed for use in

satellite-borne scientific instruments. The theorems we state provide the first steps toward the definition of the semantics of each instruction that would enable formal verification of software written for the SCIP. Because we prove that the design of the processor satisfies the theorems, software developers may be confident that these definitions correspond to the actual behavior implemented by the processor.

Formal verification requires three components: a model of the system to be verified, a specification of the system's intended behavior, and a proof that the model implements the specification. We chose to model the SCIP at the hardware description language (HDL) level. This allows our proofs to address the logical correctness of the SCIP design but prevents us from proving lower-level properties such as that the processor layout satisfies timing requirements. To model the SCIP's HDL design, we developed a framework in our chosen proof tool, ACL2 (or A Computational Logic for Applicative Common Lisp), for representing the VHSIC Hardware Description Language (VHDL), and then we translated the SCIP's VHDL code to this framework. We specified the intended behavior of the SCIP as theorems defining the pre- and postconditions of instructions involving push and pop operations and used ACL2's automated theorem prover to develop proofs that our model satisfied these theorems. Although our theorems specify only part of the expected behavior of the SCIP, they demonstrate the feasibility of our approach and increase confidence in critical functionality of the processor. The proofs of our theorems rely on numerous lemmas defining basic properties of modular bitwise arithmetic and significant results for every major functional unit of the SCIP. Future work may expand our specification by proving additional theorems defining the behavior of the SCIP for other operations.

MOTIVATION

Why Choose the SCIP?

The SCIP is a stack-based microprocessor with 16-bit instructions and a configurable 16- or 32-bit data path. It was designed at APL to be a simple, lightweight, low-power processor suitable for implementation using a field-programmable gate array (FPGA). The SCIP uses a densely packed instruction set that allows for compact program code, a critical feature for embedded software expected to run in low-memory environments. The SCIP has been used in several APL-developed instruments on active missions including Juno[2] and the Van Allen Probes (formerly the Radiation Belt Storm Probes).[3]

We chose the SCIP as a test case for verification because of its use in satellites, as well as the simplicity of the processor's design. A bug or unexpected feature in the processor while it is deployed as part of a spacecraft may lead to unpredictable software behavior. This may mean the loss or silent corruption of important scientific data, reducing the benefit of the costly satellite. Although the SCIP is typically implemented by using a FPGA, no APL-developed spacecraft to date has included the ability to modify the FPGA programming after launch, so there would be no way to correct a hardware defect in an operational system. Even software workarounds are difficult to deploy because of the limited communications channel between the satellite and software authors. The high cost of failure motivates both rigorous specification of the SCIP's expected behavior and the need for high confidence in its correct implementation. Formal verification is the only approach capable of meeting these goals.

The SCIP's simplicity also makes the processor an attractive target for formal verification. The processor performs no pipelining, out-of-order execution, or other optimizations that may complicate instruction effects. The entire implementation is roughly 5000 lines of VHDL code, was written by a single designer, and uses a small, consistent set of idioms and VHDL features. A possible area of future work is using the formal instruction set specification as a verification tool for a more advanced revision of the SCIP that may include features such as pipelining.

Verification Tools

The choice of a verification tool is an important first step in formal verification. We chose ACL2, a theorem prover developed at the University of Texas that is based on the LISP programming language. Verification tools can be roughly divided into model checkers, which use exhaustive state space exploration to automatically prove or refute logical assertions describing either expected or forbidden sequences of program states, and theorem provers/proof assistants, which rely on axiomatic reasoning to produce proofs by applying logical inference rules.

Model checkers are popular in hardware design circles and can be used to demonstrate the absence of particular faults in a system (e.g., that certain events always occur in a particular order). Model checkers are well suited to hardware design because, like HDLs, their modeling languages tend to be designed around the concept of interacting state machines executing in parallel. The most common criticism of model checkers is that they are susceptible to a state explosion problem on large models. Although modern model checkers may scale well enough to handle the SCIP, they are still not particularly well suited to the development of axiomatic specifications, a key goal of this effort.

In contrast, theorem provers like ACL2 are designed from the ground up to support reasoning in terms of pre- and postconditions of sequences. This makes them a natural fit for our goals. Unfortunately, the linear rea-

soning used by most theorem provers is not ideally suited to the parallelism of HDLs. Our solution to this is to show that, although the VHDL processes may execute in parallel, their effects are independent, and thus they can be treated as independent functions. We chose ACL2 over other axiomatic proof systems such as Coq or Isabelle because of the abundance of literature focused on the topic of using ACL2 for hardware description verification. There is a significant body of work representing HDLs in ACL2 that served as a starting point for our own embedding of VHDL.

RELATED WORK

ACL2 and its predecessor, Nqthm, have a long history of use in hardware design verification. Probably the best-known example is the verification of the kernel of the AMD floating point division algorithm by Moore et al.[4] This work focused on proving that the algorithm correctly implemented floating point division as defined in the IEEE floating point standard. It did not attempt to verify the HDL specification of the algorithm.

Hunt and Brock[5] introduce an HDL with semantics formally defined in ACL2 and use it to specify and verify the design of the FM9001 processor. Their HDL can be mechanically translated into a preexisting HDL for synthesis.

Georgelin et al.[6] describe a system for modeling VHDL in ACL2 that uses macros to provide syntactic constructs similar to the original VHDL. We build on their work by introducing a more faithful model of VHDL types and support for hierarchically nested components.

ACL2

ACL2 is both a LISP-like language and an automated term rewriting theorem prover.[7] ACL2 was principally developed by Matt Kaufmann and J. Strother Moore as the successor to Nqthm and the Boyer–Moore theorem prover.[8] ACL2 is an attractive tool for hardware design verification because of its high level of automation, familiar syntax, history of application in the field, and active user community.

Programming in ACL2

ACL2's input language is a LISP dialect, which makes it straightforward for anyone with a LISP background to write simple functions in ACL2. Like LISP, ACL2's syntax is based on S-expressions. An S-expression is either a symbol such as `foo` or a list of S-expressions enclosed in parentheses such as `(foo bar baz)`. Symbols evaluate to some value determined by the current environment (notably, symbols that are numbers, such as `12`, always evaluate to themselves). Lists are evaluated by applying the function named by the first element to the result of evaluating the other elements in the list. Thus, the S-expression `(+ 1 (* 4 5))` evaluates to `21` by first evaluating `(* 4 5)` to produce `20` and then evaluating `(+ 1 20)`. To enable automated reasoning, ACL2 imposes several restrictions on its input language: all functions must provably terminate, statements may not modify program state, and functions cannot be passed as arguments to other functions.

To ensure termination, ACL2 requires that all functions either be nonrecursive or recur with a strictly decreasing measure function. For example, ACL2 can automatically prove that a recursive function terminates if the function operates on a list that is shortened at each recursive call site. All recursive functions we used to model the SCIP followed this pattern, and thus ACL2 was easily able to prove termination.

ACL2 functions must always be pure functions from their inputs to their outputs. In most programming languages, variables represent a box in which a value can be stored; at any time, the value in the box may be retrieved or replaced with a different value. In ACL2, as in mathematics, a variable is a name given to an unknown value. As a result, there is no assignment statement and hence no ability for a program to modify external state. This guarantees that the rewriting system can consider function invocations without concern for the order or context in which they are called. In particular, it implies that it is safe to replace the invocation of a function with the function's body. We avoid side effects by writing our models as functions from the complete current state of a hardware unit to the complete next state (including unchanged values).

Higher-order functions (i.e., functions that accept functions as arguments or return functions) are a prominent and popular feature of most LISP-like languages. Unfortunately, to allow for greater automation of theorem proving, ACL2 does not support higher-order functions. This restriction makes it difficult or impossible to generalize interfaces in ways common to LISP; functions such as `mapcar`, which applies a function to each element of a list, are not expressible in ACL2.

ACL2 supports LISP-style macros, which allow the programmer to introduce new syntactic forms and control the order of term evaluation. Some applications of higher-order functions can be simulated by using macros. Our framework for modeling VHDL in ACL2 relies heavily on macros to allow a nearly line-for-line translation without exposing the difficulty of mapping between the differing semantics of the two languages.

Proofs in ACL2

Theorems in ACL2 are introduced using the top-level `defthm` event form. Theorems are given as S-expressions with an implicit universal quantification over all free variables. ACL2 attempts to prove a theo-

Example 1. A common structure for ACL2 `defthm` events: given hypotheses `(h1 x y z)` and `(h2 x y z)`, the form `(f x y z)` can be replaced by `(g x y z)`

```
(defthm my-theorem
        (implies (and (h1 x y z) (h2 x y z))
                 (equal (f x y z) (g x y z))))
```

rem by applying a series of rewrite rules to transform the theorem into something that is trivially true. Example 1 is a common form of an ACL2 theorem. If given such an event, ACL2 would attempt to use the definitions of `h1`, `h2`, `f`, and `g` and any other currently active theorems or definitions to show that for any choice of `x`, `y`, and `z` satisfying both `(h1 x y z)` and `(h2 x y z)`, `(f x y z)` is equal to `(g x y z)`. If successful, ACL2 introduces a new rewrite rule, which it may use in future proofs to replace `(f x y z)` with `(g x y z)`. Every theorem and function introduced in ACL2 affects the way ACL2 attempts to prove future theorems.

The key to effective use of ACL2 is understanding how the proof engine decides which rules to use under which circumstances. A common stumbling block is ACL2's difficulty in moving between levels of abstraction: once ACL2 expands the definition of a term, any theorem that refers to the term by name can no longer be applied (because the term's name has been replaced by its body).

Example 2 demonstrates this problem. Our model makes heavy use of lists of bits, called bitlists, such as `(1 0 0)` or `(0 0 1)`. The function `bitlist-append` uses ACL2's built-in append function to append two bitlists—for example, `(bitlist-append '(0 0 1) '(1 0 1))` evaluates to `(0 0 1 1 0 1)`. The function `bitlist-to-int` converts bitlists to integers, with the bits interpreted from least to most significant—for example, `(bitlist-to-int '(0 0 1))` evaluates to `4`, `(bitlist-to-int '(1 0 1))` evaluates to `5`, and `(0 0 1 1 0 1)` evaluates to `44`. The theorem `evenp-bitlist-append` of Example 2 states that if the first element of a bitlist is zero, then appending any other bitlist to it will yield a result representing an even number. Because the proof of this theorem relies both on facts about list appending in general and on facts that are unique to bitlists, ACL2 is unable to prove it as written.

Example 2. The last theorem, `evenp-bitlist-append`, follows directly from the previous two, but applying `append-car` requires unfolding the definition of `bitlist-append`, which prevents ACL2 from applying the theorem `evenp-bitlist-to-int`

```
(defthm append-car
    (equal (car (append x y)) (car x)))

(defthm evenp-bitlist-to-int
    (implies (equal (car b) 0)
         (evenp (bitlist-to-int b))))

(defthm evenp-bitlist-append
    (implies (equal (car b1) 0))
         (evenp (bitlist-to-int (bitlist-append b1 b2)))))
```

The first theorem of Example 2, `append-car`, states that the first element of the value returned by the append function is the same as the first element of its first argument (the function `car` returns the first element of a list). The second theorem, `evenp-bitlist-to-int`, states that if the first element of a bitlist is zero, then the numerical interpretation of the bitlist is even. The final theorem, `evenp-bitlist-append`, states that if the first element of a bitlist is zero and another bitlist is appended to it, then the result will represent an even number. This is true because, by the first theorem, we know that the first element of the resulting bitlist will be zero, and by the second theorem, we know that the integer interpretation of a bitlist starting with zero is even. Unfortunately, applying the first theorem requires that ACL2 rewrite the call `(bitlist-append b2 b1)` with its definition `(append b1 b2)`, which no longer matches the statements of `bitlist-append-thm` or `evenp-bitlist-to-int`. This prevents ACL2 from automatically proving the theorem as stated. The actual theorems also require hypotheses that all variables have appropriate types; however, even with these hypotheses, the theorem cannot be proved because of ACL2's rewriting strategy.

The solution to this problem is to carefully control the set of rules that ACL2 will use to prove new theorems, called the "current theory." One approach is to carefully introduce rules that pattern match on function bodies to reassemble the original invocation.[9] The key to this approach is strategically enabling and disabling theorems during subproofs. We found this approach somewhat contrary to the goal of automated proof finding and instead focused on a strategy of disciplined abstraction levels.

Rather than allow ACL2 to "simplify" instances of `bitlist-append` to `append`, we explicitly lift the needed theorems and disable the definition of `bitlist-append`. Thus, to solve the problem of Example 2, we would define a lifted version of `append-car` called `bitlist-append-car`, which states essentially the same theorem in terms of `bitlist-append`. This approach requires a fair amount of additional boilerplate code for lifting "obvious" theorems but prevents excessive case splitting and reduces the prover's reliance on explicit hints. In the *Data Types* section, we discuss another benefit of this approach: because of the strict layering, we were able to replace the underlying data model of our framework without requiring significant changes to the model of the SCIP.

MODELING ARCHITECTURE

The first component of formal verification is a model of the system to be verified in the language of the verification tool. Because our proofs are developed on the basis of this model and not the original VHDL source code, any confidence gained by performing formal verification is limited by the model's fidelity and accuracy. To represent the SCIP's design in ACL2's LISP dialect, we build on the work of Georgelin et al.[6] We use ACL2/LISP macros to provide a syntactic layer that is nearly comparable, line for line, to the original VHDL source code. These macros expand to ACL2 functions that implement the VHDL behavior and theorems that guarantee the validity of the model.

Our modeling system has two main goals:

1. To enable either automated or manual translation of VHDL code to ACL2
2. To allow for independent auditing to ensure that the VHDL model and the ACL2 model for a particular system correspond

To support these goals, we focused on providing VHDL-like syntactic constructs in ACL2. This approach allowed us to incrementally improve the faithfulness of the modeling system's semantics without breaking the existing translation of the SCIP's design. In the *Data Types* section, we describe our motivations for altering the core data types used by the SCIP model. This change was straightforward because of the syntactic abstractions we used to build the model initially. Because of time constraints, we were unable to develop an automated translation tool targeting our modeling system. Instead we relied on a manual translation of the SCIP's source code; proof of the theorems stated in the *Proving Correctness* section required translation of roughly 4000 lines of the SCIP's VHDL design. The major entity that was not translated was the SCIP's arithmetic/logical unit because it was not relevant to the theorems.

Our modeling system uses three top-level macros, `defentity`, `defprocess`, and `defarchitecture`, for describing VHDL entities, processes, and architectures, respectively. Example 3 illustrates the use of these macros to represent VHDL code. There is some divergence between the ACL2 macro-based syntax and the original VHDL; most notably, internal signals and components are described in the `defentity` block, and processes are defined at the top level and then explicitly listed in the `defarchitecture` block.

Entities

In VHDL, entities represent the interface to architectural components in terms of input and output ports. In ACL2, the `defentity` macro introduces the functions

Example 3. A side-by-side comparison of VHDL (on the left) and its ACL2 equivalent (on the right) using our `defentity`, `defprocess`, and `defarchitecture` macros

```
entity myent is
  port (
    in1   : in   unsigned(15 downto 0);
    in2   : in   unsigned(15 downto 0);
    out1 : out unsigned(15 downto 0);
    out2 : out unsigned(15 downto 0));
end myent;
architecture rtl of myent is
signal sig1 : unsigned(15 downto 0);
component child_typ
port(
      cin  : in   unsigned(15 downto 0);
      cout: out unsigned(15 downto 0)
);
end component;
begin
  child: child_typ
  port map(
            cin  => in2,
            cout=> out2
  );
  proc1: process (in1, in2)
  begin
    sig1 <= in1(15 downto 8) & in2(7 downto 0);
  end process proc1
  end rtl
```

```
(defentity myent
    :inputs      ((in1   std-logic :array 16)
                  (in2   std-logic :array 16))
    :outputs     ((out1 std-logic :array 16)
                  (out2 std-logic :array 16))
    :signals     ((sig1  std-logic :array 16))
    :components  ((child-typ child
                    ((cin   . in2)
                     (cout . out2))))
)
(defprocess myent proc1 (in1 in2)
    (example-set-output sig1
        (std-logic-list-append (downto in1 15 8)
                               (downto in2 7 0)))
    )
(defarchitecture myent
    myent-proc1
    myent-step-child
    )
(myent-arch-order-independence-thm)
```

and theorems necessary for instantiating and reasoning about entities in ACL2. In particular, `defentity` uses `defstructure`[10] to introduce a type predicate for the new entity and accessors and mutators for the inputs, outputs, signals, and components listed.

In Example 3, `defentity` is used to introduce an entity called `myent`, with input ports `in1` and `in2`, output ports `out1` and `out2`, a signal `sig1`, and a subcomponent `child`. The `defentity` macro will introduce a number of functions including a state predicate: `myent-state-p`, accessors such as `myent-in1`, and mutators such as `myent-set-in1`.

Unlike in VHDL, the internal signals and subcomponents of an entity must be listed as arguments to `defentity`. To simulate VHDL's latching behavior, each internal signal of the entity corresponds to two distinct fields of the structure generated by `defentity`; the first field has the same name as the signal and contains the initial signal value, the second field is named by appending a "+" character to the signal name and is assigned the computed next value for the signal. The `defentity` macro introduces a function ⟨`entity`⟩`-update-state`, which is used to update the signal fields.

As in VHDL, the component definition must include a mapping between the input and outputs of the child and the signals of the parent entity. This mapping is used by `defentity` to generate an update function for the child component, which uses copy-in-copy-out semantics to provide the child's inputs, step the child, and map the outputs into the parent's state.

Additionally, `defentity` introduces macros for generating theorems that specify which ports are read or written by a form. These macros are used by `defprocess` and `defarchitecture` to ensure that processes depend only on input ports and the input half of internal signals, and write only to output ports and the output half of internal signals. Because the SCIP does not use input-output ports, our macro system does not currently support them.

Processes

A VHDL process describes how the values of the input ports and internal signals of an entity are combined to compute the values of output ports and to update internal signals. Processes correspond roughly to functions in a traditional programming language, and thus we use ACL2 functions to model VHDL processes.

The `defprocess` form in Example 3 generates a function called `myent-proc1`. This function applies the body of the `defprocess` form to an argument representing the state of a `myent` instance. The process is used in the `defarchitecture` block later to define the single-step behavior of the `myent` entity.

For convenience, the input ports and signals values on the input state are bound to appropriately named local variables; this allows free variables in the body of a `defprocess` to be resolved as port/signal names as in VHDL. To guarantee that the body depends only on the input ports and signals of the entity and updates only output ports and signals, `defprocess` uses the macros introduced by `defentity` for introducing port preservation and independence theorems.

VHDL processes may be either combinatorial or sequential. A combinatorial process defines its outputs as an arithmetic or logical combination of its inputs. Sequential processes are more like functions in a traditional programming language; they consist of a series of statements that are executed one after another and may make use of intermediate state by assigning values to local variables. Our framework supports only combinatorial processes because these map well onto ACL2's notion of variables and because the SCIP does not make use of sequential processes. Although our framework could directly support sequential processes, it has no way to represent the interleaving of sequential actions and hence would force sequencing of process executions. This would cast into doubt the correctness of proofs for sequential processes that make temporary updates to shared input-output signals.

Architectures

VHDL architectures follow a syntactic construct that groups the internal signals, subcomponents, concurrent statements, and processes of an entity into a complete description of the component's behavior. The `defarchitecture` macro is intended to indicate a similar grouping of functional units. The `defarchitecture` block in Example 3 defines a step function that is the composition of the two processes and a single step of the component `child`. `Defarchitecture` also introduces theorems that show that the final state is independent of the order in which processes are composed. This is an important theorem because VHDL processes are evaluated in parallel, while our framework evaluates the processes sequentially. Because our framework supports only combinatorial processes, proving that the result of a step is independent of the order in which the processes are evaluated suffices to show that their parallel execution is equivalent to the chosen serialization. If our framework supported sequential processes, a more detailed theorem showing all possible interleavings of processes would be needed.

Data Types

Initially our VHDL models used ACL2 numeric types for vectors of VHDL logical values and a symbolic representation for the SCIP's compound instructions to simplify decoding logic. This approach made it easy for us to model the SCIP using the `defentity`, `defprocess`,

and `defarchitecture` macros described but made the correctness theorems more difficult to prove. We encountered three main problems that led us to reimplement our underlying data model by using lists of logical values. We briefly describe these challenges before describing our new approach in greater detail.

VHDL's standard logic type includes nine different values: U (uninitialized), X (strong drive, undefined value), 0, 1, Z (high impedance), W (weak drive, undefined value), L (weak drive, logically 0), H (weak drive, logically 1), and - (don't care), whereas ACL2 integers may represent only (sequences of) zeros and ones. This was rarely significant, as the SCIP's design tends to rely solely on the logical interpretation of values. However, the inability to faithfully represent "undefined" and "uninitialized" meant that our theorems were valid only for well-defined inputs, which is not necessarily a reasonable assumption.

The length of a VHDL vector is fixed by its declaration, whereas ACL2 integers are unbounded. This led to the need to explicitly coerce any computed value by using the modulus function. The implementation of modulus in ACL2 is not particularly transparent and is difficult for a beginning user to manipulate in theorems. We learned later that a more powerful set of theorems for working with modular arithmetic is included in the ACL2 distribution, but by that time we had completed our reimplementation.

A related challenge is that ACL2 integers are poorly suited to bit slicing operations common in VHDL. As we discuss further in the *Stack Design in the SCIP* section, the instruction set of the SCIP uses a packed bit field to specify several primitive operations in each instruction word. Figure 1 shows an example instruction that performs an addition of `ptop` and `pnext`, pushes the result on top of the `pstack`, and performs a return.

Initially we modeled instructions as lists of symbols describing each operation performed. This avoided complex bit slicing logic for decoding. For example, the instruction in Fig. 1 was represented as the list `(alu a+b next push return)`. This made decoding trivial but meant that instructions were a different

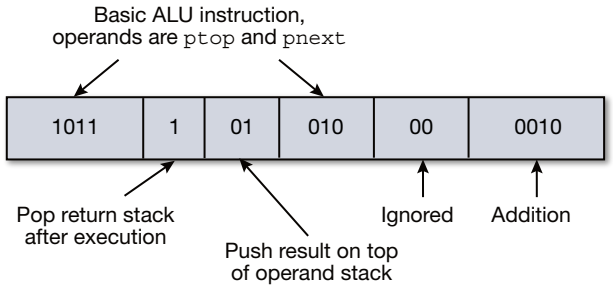

Figure 1. The SCIP instruction to add the top two elements of the `pstack`, push the result on top of the `pstack`, and perform a return. ALU, arithmetic logic unit.

data type from numeric values, which prevented us from using typing rules for any port or signal that could contain either instructions or data. This was the most significant challenge we faced with our original typing model; as the complexity and scope of our proofs grew, it became impossible to maintain consistency without the ability to coerce a number into an instruction. In our new model, the necessary bit slicing logic is simple, and so we use the faithful representation of instructions as bit vectors.

Most of the modifications required to switch our model of the SCIP to this new data model were automated search-and-replace operations. The new data model represents VHDL's std-logic type directly in ACL2 and VHDL's vector types as ACL2 lists beginning with the least significant bit. The representation of vectors of std-logic types is similar to the bitlists described in the *Proofs in ACL2* section. This implementation allows us to use structural recursion and existing list manipulation primitives such as `car`, `cdr`, `append`, etc., to implement the common bit slicing operations of VHDL. We implemented conversion functions `int-to-std-logic-list` and `std-logic-list-to-int` for converting between non-negative ACL2 integers and std-logic lists. We also implemented basic arithmetic and logical operations such as incrementing, decrementing, and, or, not, and logical shifts on fixed-length lists of std-logic values. We proved that these operations have the expected algebraic properties and correspond with operations on non-negative integers (modulo 2 to the length of the list). In keeping with our policy of disciplined level separation, the SCIP model relies only on these theorems and not directly on the implementation of the data types.

STACK DESIGN IN THE SCIP

The SCIP maintains two internal stacks: a parameter stack (`pstack`) used to provide operands for instructions and store results and a return stack (`rstack`) used to store the return address of call instructions. Because many instructions rely on or manipulate these stacks, the major proofs presented in the *Proving Correctness* section focus on showing that the VHDL implementation of the SCIP conforms to the abstract properties of a stack in normal operation and behaves predictably in exceptional situations (such as underflow or overflow).

The `pstack` and `rstack` are implemented as 16-element arrays of word-size registers (`pregfile` and `rregfile`, respectively) combined with two 4-bit index registers (`ptopi` and `poveri` for the `pstack` and `rtopi` and `roveri` for the `rstack`) indicating the index of the top element contained in the array and the overflow point (bottom element). Note that this scheme naturally forms a ring because incrementing the index registers will wrap around from the last

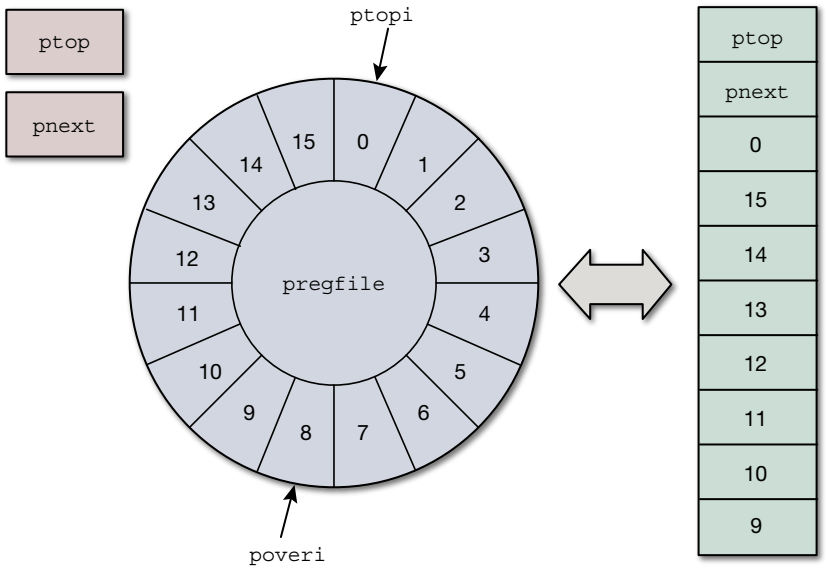

Figure 2. The `pstack` is assembled from the `ptop` register, the `pnext` register, and the elements of the `pregfile` starting at `ptopi` and counting down (mod 16) to `poveri`.

(15th) element of the array to the first (0th) element. In addition to these arrays, the SCIP includes dedicated registers to hold the top two elements of the `pstack` (`ptop` and `pnext`) and the top element of the rstack (`rtop`). Figure 2 shows conceptually how the `pstack` is constructed from these registers.

The SCIP can also be configured to store additional elements from the stacks in main memory by setting the stack-enabled (`stackenb`) bit in the processor control register (`pcr`). If this bit is set, then overflow or underflow of the on-processor stack registers will cause the SCIP to enter a special overflow (`resp. underflow`) mode for two clock cycles while a stack element is written to (`resp. read from`) main memory. While in this special mode, the SCIP does not execute user instructions.

Another important feature of the SCIP design is that the instruction set consists of a small number of instruction classes, each of which is really a packed bitfield structure specifying a number of different primitive operations and options. For example, the instruction in Fig. 1 includes an arithmetic operation, a `pstack` operation (push), and a bit indicating that the top of the `rstack` should be popped into the PC to perform a return. All arithmetic and logical operations take `ptop` as their first operand, but the second operand may come from an immediate value, a register, or `pnext`. Similarly, the result of an operation may be placed on the `pstack` or in a register.

This expressiveness leads to a multiplicative explosion in instruction set size: rather than having one return operation, four stack operations, and four arithmetic operations, the SCIP has 32 return+stack+arithmetic operations. In total, the SCIP has 18 different instruction forms totaling more than 9000 unique opcodes. To verify the instruction set's correctness on a per-instruction basis would be infeasible. Instead, we focused on specifying and verifying the effects of primitive operations, such as stack manipulations, with the intent to later verify that the cumulative effect of an instruction is consistent with the composition of the effects of its component operations.

PROVING CORRECTNESS

Although our long-term goal is for a complete verification of the instruction set semantics of the SCIP, in this article we focus on proving that the SCIP's implementation of push and pop operations is equivalent to applying the `cons` and `cdr` (prepend and tail) operations on the parameter stack represented as a list. These theorems rigorously define the abstract semantics of the relevant instructions and could be used in future efforts to prove the correctness of software targeting the SCIP. We separate the problem into two cases: verifying that stack updates are performed correctly when no overflow or underflow occurs and verifying that overflow and underflow conditions are correctly handled.

Standard Operation

We must show that, at the beginning of the clock cycle after an instruction specifying a push operation, the stack contains the element pushed, followed by the elements of the previous stack. Similarly, for pop operations, the new stack must be the old stack with the top element removed. The exact statement of the theorem proved for the case of push operation is shown in Example 4. Note that this theorem is concerned only with the portion of the `pstack` contained in the register file and does not describe the updates to the `ptop` and `pnext` registers.

This theorem shows that, after the execution of a push operation, the `pstack` is defined by the `cons` of the new element onto the original `pstack`. The theorem for pop operations, `scip-pop-pstack-cdr` (not shown), is analogous with the new `pstack` defined by the `cdr` of the original. Figure 3 shows pictorially how the new value of `ptopi` is computed; solid lines represent control flow and originate from diamonds, which represent conditionals or guards, while dashed lines represent data flow and originate from rectangles, which show each data update. From Fig. 3, the need for at least three internal steps is clear: the first step sets the `ptopi_plus1` and `ptopi_minus1` signals of the `pstack` (`pstack[ptopi_plus1]` and `pstack [ptopi_minus1]`, respectively), the second step

Example 4. Statement of the principal correctness theorem for the `pstack` push operation

```
(defthm scip-push-pstack-cons
   (implies
      (and (scip-pstack-inputs-ready-p st) (not (equal (scip-reset st) 1))
           (not (rising-edge (scip-clk st))) (equal (scip-stretch st) 0)
           (instr-class-stack (scip-ir+ st)) (equal (stack-op (scip-ir+ st)) *st_push*)
           (std-logic-defined-list-p (scip-ptopi+ st)) (std-logic-defined-list-p (scip-poveri+ st))
           (integerp n) (>= n 3))
      (equal
         (scip-get-pstack-regfile-as-list (scip-step (scip-raise-clock (scip-step-n n st))))
         (let ((p (scip-ptopi+ st)) (o (scip-poveri+ st)))
           (cond
             ((equal (std-logic-list-to-int p) (std-logic-list-to-int o))
                     (list (scip-pnext+ st)))
             (t      (cons (scip-pnext+ st) (scip-get-pstack-regfile-as-list st))))))))
```

updates the `ptopi_next(pstack[ptopi_next])` signal of the `pstack`, and the third step updates the processor's `ptopi_n` signal.

The hypotheses of our theorems provide guarantees about the state of the processor analogous to the guards shown in the diamonds of Fig. 3. The compound predicate `scip-pstack-inputs-ready-p` guarantees that the other inputs to this computation, `state`, `ir`, and `ptopi`, are held constant until the next clock rise. The next three hypotheses,

```
(not (equal (scip-reset st) 1))
(not (rising-edge (scip-clk)) st)
(equal (scip-stretch st) 0),
```

ensure that the processor is mid clock cycle, not operating on stretched cycles, and will not reset its state on the next clock rise. This is essentially the normal operating state of the processor. The next two hypotheses indicate that the current instruction includes the stack operation in question (push or pop). The predicate `std-logic-defined-list-p` is a type predicate to guarantee that all the bits of the `ptopi` and `poveri` registers have well-defined logical values (i.e., are neither X nor U). The two hypotheses `(integerp n)` and `(>= n 3)` are used to force the SCIP to step enough times for the `pstack` logic to update `ptopi_n`. The `scip-pop-pstack-cdr` theorem has an extra hypothesis to disallow the case in which the `ptopi` and `poveri` registers are equal; in this case, the stack is considered empty, and thus the new stack after the pop operation cannot be defined in terms of the original. In contrast, `scip-push-pstack-cons` can handle this case because pushing onto an empty stack yields a stack of one element.

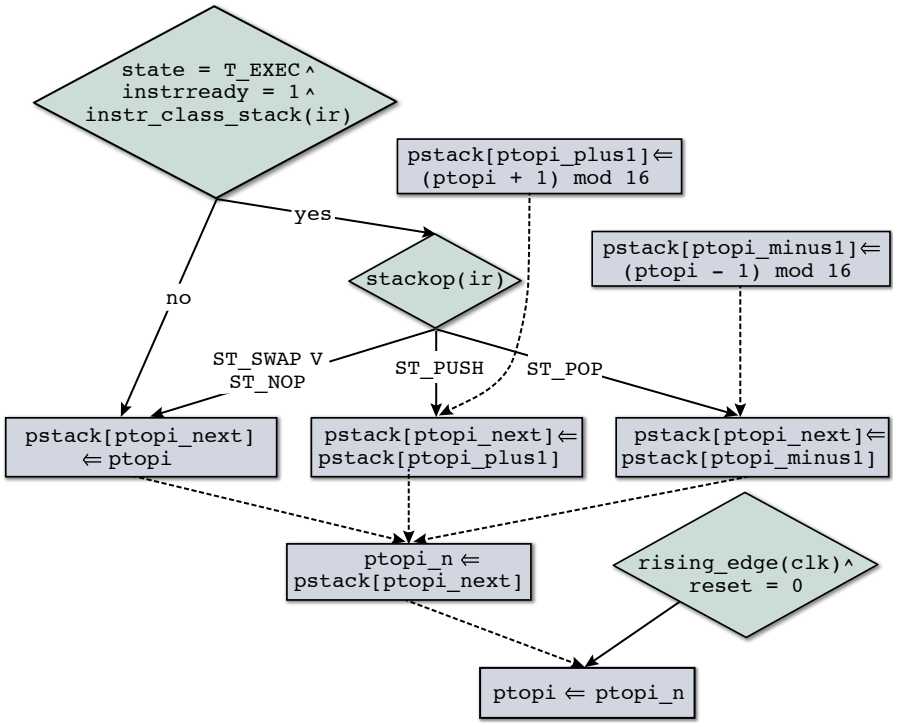

Figure 3. The control and data flow used to calculate the value for `ptopi` at the next clock cycle.

We proved a third theorem showing that the `pstack` register file is unchanged if the current instruction is not a stack operation or specifies either a no-op or swap operation. This theorem is simpler than the other two but is otherwise analogous.

Handling Overflow and Underflow

If a push operation would cause the `ptopi` register to

advance to the value of the poveri register, the SCIP suspends execution of the user program for two clock cycles, during which it stores the element of the stack pointed to by poveri to main memory on the basis of the value of the psp register, and increments poveri. Analogously, if a pop operation causes the ptopi register to decrease to poveri + 1, the processor will insert cycles to read an element of the stack from main memory. We have not yet completed a model of main memory, and thus we cannot show that the correct data are fetched. We have shown that the processor correctly identifies the overflow or underflow, enters the desired state, writes the correct values to the output ports for reading and writing data from/to main memory, and updates the psp register appropriately.

These proofs are more complex than the proofs of normal operation because they must describe behavior spanning four clock cycles: the cycle during which overflow/underflow is detected, two repair cycles, and the beginning of the next cycle of normal execution. One of the most complex operations of this procedure is calculating the new value of the psp register. In the case of overflow, the first repair cycle is used to compute the new psp value by placing its high 15 bits on bbus, using the arithmetic logic unit (ALU) to decrement this value, writing this value onto the wbus, and setting the high bits of the psp_n signal to the low 15 bits of the wbus padded with a 0, and finally setting psp to psp_n on the next rising clock edge. Figure 4 illustrates this process.

The result is that the psp register is decreased by 2 (modulo 2^{16}) before the overflowed value is stored in memory at the address referenced by psp. The underflow case is essentially the inverse operation and is performed during the second repair cycle so that the read request is issued before the update to psp. The symmetry is necessary to ensure that overflowed data are fetched properly during the next underflow event. Further, because the read request is issued during the first underflow repair cycle, the data are available when normal execution is resumed after the second repair cycle. Note that if psp is even (word aligned), the overflow procedure is a subtraction by 2, which is the desired effect. However, if psp is odd, it will subtract 3 before storing the overflowed value. The only way for psp to take on an odd value is via direct manipulation by a user program. If the user stores an odd value to psp, the overflow/underflow protocol will still function properly because the first overflow will fix psp to an even value, and the result of underflow after an explicit update to psp without an intervening overflow cannot be meaningfully defined anyway.

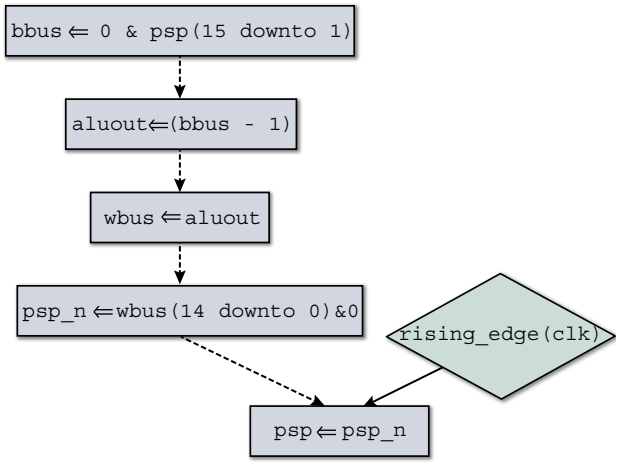

Figure 4. During the first overflow cycle, the new psp is computed by subtracting 2 from the original value.

CONCLUSION

The SCIP has been used in numerous APL-developed flight instruments and will be used for several in the future. Its simplicity makes it an attractive target for verification efforts, and its utility makes it a key APL technology. Seen as a single component, the SCIP may seem like a curiosity—a processor employing an unusual processor architecture without pipelining or other advanced processor optimizations. But in reality, it resides comfortably at the center of a layered system that allows APL flight instruments to provide a critical contribution to APL and the wider space community. The SCIP provides a foundation for the execution of a multitasking real-time operating system,[11] which in turn provides a foundation for a reusable commanding and telemetry management library,[12] which in turn provides a foundation for the development of instrument-specific software routines, allowing APL-developed instruments to leverage a significant amount of heritage design from one instrument to the next.[13] Each layer of this architecture relies on the correctness of the layers below to function properly. Formal proof of the correctness of lower layers provides ripple benefits, as the assumptions made by each subsequent layer can then be verified with respect to the guarantees proved by the lower layer.

In this article, we have described our experience precisely specifying one aspect of the SCIP's behavior: parameter stack manipulation. We have described the framework we developed for representing the SCIP's VHDL design in the language of ACL2 and highlighted the more significant theorems we proved. The theorems we proved showing the exact state transitions caused by stack manipulation instructions are the first steps toward a complete verification of the SCIP's implementation and formalization of its instruction set. Developers would rely on the guarantees provided by these and other analogous theorems when developing or verifying critical software targeting the SCIP, such as the operating system, compiler, libraries, and application software.

Developing and verifying a complete specification for even a small processor like the SCIP is a significant undertaking. However, much of the challenge comes in modeling the system in a form amenable to specification and proof. Once this groundwork is laid, the major obstacles stem from the need to understand exactly what the specification should be, and the discovery that elements of the models may be inconsistent with the true implementation. Given the breadth of the lemmas required to prove the properties presented here, future proofs of the correctness of the SCIP's design should follow naturally.

ACKNOWLEDGMENTS: John Hayes and Andrew Harris provided valuable insights on the SCIP design for both the execution of the described work and the writing of this article.

REFERENCES

[1]IEEE Computer Society, *IEEE Standard for Floating-Point Arithmetic*, IEEE Standard 754-2008, doi: 10.1109/IEEESTD.2008.4610935 (29 Aug 2008).

[2]NASA's Juno website, http://www.nasa.gov/mission_pages/juno/main/index.html (accessed 29 May 2013).

[3]NASA's Van Allen Probes website, http://www.nasa.gov/mission_pages/rbsp/main/index.html (accessed 29 May 2013).

[4]Moore, J. S., Lynch, T. W., and Kaufmann, M., "A Mechanically Checked Proof of the AMD5$_K$86 Floating-Point Division Algorithm," *IEEE Trans. Comput.* **47**(9), 913–926 (1998).

[5]Hunt, W. A., and Brock, B., "A Formal HDL and Its Use in the FM9001 Verification," *Philos. Trans. Phys. Sci. Eng.* **339**(1652), pp. 35–47 (1992).

[6]Georgelin, P., Borrione, D., and Ostier, P., "A Framework for VHDL Combining Theorem Proving and Symbolic Simulation," in *Proc. ACL2 Workshop*, Grenoble, France (2002).

[7]Kaufmann, M., and Moore, J. S., "ACL2: An Industrial Strength Version of Nqthm," in *Proc. 11th Annual Conf. on Computer Assurance (COMPASS '96)*, Gaithersburg, MD, pp. 23–34 (1996).

[8]Boyer, R. S., and Moore, J. S., *A Computational Logic*, Academic Press, New York, 1979.

[9]Young, B., "Reverse Abstraction in ACL2," in *Proc. Fifth International Workshop on the ACL2 Prover and Its Applications*, Austin, TX (2004).

[10]Brock, B., *Defstructure for ACL2*, technical report, University of Texas at Austin, http://www.cs.utexas.edu/~moore/publications/others/defstructure-brock.ps (1997).

[11]Hayes, J. R., *Multitasking in FRISC3 Forth*, Technical Memorandum TCE-88-291, JHU/APL, Laurel, MD (1988).

[12]Hayes, J. R., *MESSENGER Instrument Common Flight Software Specification*, Technical Memorandum SRI-03-011, JHU/APL, Laurel, MD (2003).

[13]Hayes, J. R., *Instrument Software Reuse*, Technical Memorandum SRI-05-001, JHU/APL, Laurel, MD (2005).

The Author

J. Aaron Pendergrass is a member of the Senior Professional Staff in the Cyber Technology Branch of the Asymmetric Operations Department. His work focuses on enhancing software assurance via formal methods for mechanized program analysis, automating reverse engineering, and dynamic software integrity measurement. His e-mail address is aaron.pendergrass@jhuapl.edu.

The *Johns Hopkins APL Technical Digest* can be accessed electronically at **www.jhuapl.edu/techdigest**.