Verification of Stack Manipulation in the SCIP Processor

J. Aaron Pendergrass
Johns Hopkins University Applied Phyics Laboratory

Abstract

This paper presents a case study in the formal verification of the hardware description level specification of a general purpose computer processor. The major contributions of this paper are a framework for modelling VHDL hardware designs in the ACL2 language, a discipline for managing the layering of abstractions when verifying a hierarchical design, and a description of the significant theorems proved.

1 Introduction

Computer systems play a central role in nearly every aspect of modern life. Unfortunately, computer systems are typically complex amalgams of software and hardware, both of which frequently contain errors. Although approaches exist to minimize the prevalence and impact of bugs, formal approaches based on mathematical reasoning are the only tools that can lead to a true understanding of system behavior and proofs of correct operation. In particular, formal correctness proofs can show that a system, when used in accordance with a well-defined set of assumptions, will conform to some stated specification. Unfortunately, formal proof is difficult: complex systems require proportionally complex models; specifications are rarely stated formally, and thus are often ambiguous and/or inconsistent; reasoning tools require dedicated expertise; and the correct assumptions are difficult to know in advance.

We present a detailed account of our application of the ACL2 theorem proving system to the verification of the design of the Scalable Configurable Instrument Processor (SCIP). The SCIP is a light-weight, low-power processor designed for use in satellite borne scientific instruments. Our key correctness theorems prove that the VHDL design of the SCIP correctly implements the basic push and pop operations for an abstract stack. Although this theorem may at first sound trivial, its proof relies on basic properties of modular bitwise arithmetic, significant results for every major functional unit of the SCIP, and unifying theorems spanning multiple clock cycles.

2 Motivation

The principal application of the SCIP is controlling scientific instruments on satellites, but the light-weight, low-power design may make the SCIP attractive in a wider array of applications, such as wearable computing.

The use of the SCIP on satellites is enough to motivate formal verification. A bug or unexpected feature in the implementation of a processor may lead to unpredictable software behavior. For the SCIP's intended application this may mean the loss or silent corruption of important scientific data, reducing the benefit of the costly satellite. Such failures are difficult to correct due to the limited communications channel between the satellite and the software authors. The high cost of failure motivates both rigorous specification of the SCIP's expected behavior, and 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 which 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 which may include more advanced features such as pipelining.

Tool choice is an important first step in formal verification. Verification tools can be roughly divided into model

checkers, which use exhaustive state space exploration to automatically prove or refute assertions phrased in a (typically temporal) logic, 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 easily 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 hardware description languges, 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 specification that we have begun to develop as part of our verification.

In contrast, theorem provers like ACL2 are designed from the ground up to support reasoning in terms of pre- and post-conditions of pieces of sequences. This makes them a natural fit for our goals. Unfortunately, the linear reasoning used by most theorem provers is not ideally suited to the parallelism of hardware description languages.

Our approach 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. Section 5 describes our approach in more depth. We chose ACL2 over other axiomatic proof systems such as Coq or Isabelle due to the abundance of literature focused on the topic of using ACL2 for hardware description verification. In fact we initially believed translation from VHDL to ACL2's lisp dialect was a solved problem. No simple tool is available for performing this translation. There is a significant body of work which served as a starting point for our own embedding of VHDL in ACL2.

3 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. [10]. 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 [8] 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. [4] describe a system for modelling VHDL in ACL2 which 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.

4 ACL2

ACL2, or A Computational Logic for Applicative Common Lisp, is both a LISP-like language and an automated term rewriting theorem prover [9]. ACL2 was principally developed by Matt Kaufmann and J. Strother Moore as the successor to Nqthm and the Boyer-Moore theorem prover[1]. 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.

4.1 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. However, as ACL2's interactive top-level frequently reminded us, "ACL2 is not Common LISP." To enable automated reasoning, ACL2 imposes several restrictions on its input language: all functions must provably terminate, statements may not have side-effects, and all functions are first-order.

To ensure termination, ACL2 requires that all functions either be non-recursive, or recur with a strictly decreasing measure function. This was not a significant burden in our work. The majority of the functions we wrote were non-recursive and those which were recursive were either structurally recursive on lists or on non-negative integers. ACL2 is able to automatically recognize these basic forms of recursion and discharge the termination proof automatically.

ACL2 functions must always be pure (mathematical) functions from their inputs to their outputs. This requirement 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 its result. ACL2 does support a notion of a "single threaded object" which allows a limited form of side-effecting by preventing aliasing [2]. 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, is 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 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.

4.2 Proofs in ACL2

Theorems in ACL2 are introduced using the top-level `defthm` event form. Theorems are given as standard S-expressions with an implicit universal quantification over all free variables. Example 1 shows a common form of such a 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.

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))))
```

The key to effective use of ACL2 is understanding how the proof engine decides which rules to use under what 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 which refers to the term by name can no longer be applied (since the term's name has been replaced by its body).

Example 4.2 demonstrates this problem. Our model of VHDL types includes a function, `std-logic-list-append`, for appending to lists of logical values. This function just calls the ACL2 `append` function with the arguments in reverse order (to match the VHDL & operator). Theorems about `std-logic-list-append` rely on facts both about list appending in general, and on the properties of logical values. In Example 4.2 this prevents ACL2 from applying `evenp-std-logic-list-to-int` because satisfying its second hypothesis requires unfolding the definition of `std-logic-list-append` to apply `append-car`. This unfolding prevents the application of `std-logic-list-p-append` which is necessary to satisfy the first hypothesis of `evenp-std-logic-list-to-int`.

Example 2 `evenp-std-logic-list-append` follows directly from the preceeding three theorems but applying the first theorem requires unfolding the definition of `std-logic-list-append` which prevents ACL2 from applying the latter two theorems.

```
(defthm append-car
    (implies (and l1 (true-listp l1) (true-listp l2))
        (equal (car (append l1 l2) (car l1)))))
(defthm std-logic-list-p-append
    (implies (and (std-logic-list-p l1)
                  (std-logic-list-p l2))
        (std-logic-list-p (std-logic-list-append l2 l1))))
(defthm evenp-std-logic-list-to-int
    (implies (and (std-logic-list-p l1)
                  (equal (car l1) 0))
        (evenp (std-logic-list-to-int l1))))
(defthm evenp-std-logic-list-append
    (implies (and l1 (std-logic-list-p l1)
                  (std-logic-list-p l2)
                  (equal (car l1) 0))
        (evenp (std-logic-list-to-int
                    (std-logic-list-append l2 l1)))))
```

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 which pattern match on function bodies to reassemble the original invocation [12]. 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 `std-logic-list-append` to `append`, we explicitly lift the needed theorems and disable the definition of `std-logic-list-append`. Thus to solve the problem of Example 4.2 we would define a lifted version of `append-car` called `std-logic-list-append-car`

which states essentially the same theorem in terms of `std-logic-list-p` and `std-logic-list-append` This approach requires a fair amount of additional boiler plate code for lifting "obvious" theorems, but prevents excessive case splitting, and reduces the prover's reliance on explicit hints. In Section 5.4 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 processor.

5 Modeling Architecture

The first critical task facing any attempt at formal verification is modeling the system to be verified in the language of the verification tool. Any confidence gained by performing formal verification is limited by the fidelity and accuracy of the model. To represent the SCIP's design in ACL2's LISP dialect, we build on the work of Georgelin et al. [4]. We use ACL2/LISP macros to provide a syntactic layer that is nearly line-for-line comparable 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.

The key goals of our modeling system are enabling both automated and by-hand translation of VHDL code, and allowing 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 Section 5.4 we describe our motivations for altering the core datatypes used by the SCIP model. This change was straightforward because of the syntactic abstractions we used to build the model initially.

Our modeling system employs 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.

5.1 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 and theorems necessary for instantiating and reasoning about entities in ACL2. In particular `defentity` uses `defstructure`[3] 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 updates 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 processor does not use input-output ports, our macro system does not currently support them.

5.2 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.

Example 3 A side by side comparison of VHDL (on left) and its ACL2 equivalent (on 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)
```

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 inputs 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.

5.3 Architectures

VHDL architectures are a syntactic construct which group the internal signals, subcomponents, 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 3 defines a step function which is the composition of the two processes and a single step of the component `child`. `Defarchitecture` also intrudoces theorems which show that the final state is independent of the order in which processes are composed. A future enhancement to our modeling system may allow `defprocess` forms in the body of `defarchitecture` to ease translation by more closely approximating the syntax of VHDL.

5.4 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 enabled us to quickly 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 which led us to reimplement our underlying data model using lists of logical values. We briefly describe these challenges before describing our new approach in greater detail.

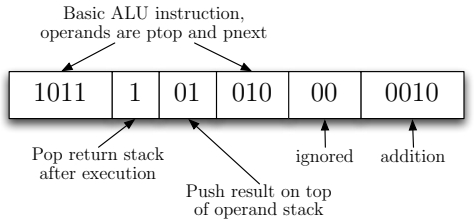

Figure 1: The SCIP instruction to add the top two elements of the `pstack`, push the result on top of the `pstack`, and perform a return.

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 only represent (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 our theorems were only valid for well defined inputs, which is not necessarily a reasonable assumption.

VHDL vectors are fixed width whereas ACL2 integers are arbitrary precision. This led to the need to explicitly coerce any computed value 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 Section 6, the instruction set of the SCIP processor uses a packed bit field to specify several primitive operations in each instruction word. Figure 1 shows an example instruction which 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 Figure 1 we represented as the list: `(alu a+b next push return)`. This made decoding trivial, but meant that instructions were a different data type from numeric values which prevented us from employing typing rules for any port or signal which 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 the 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 are represented as ACL2 lists beginning with the least significant bit. 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 implement 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 implement basic arithmetic and logical operations such as incrementing, decrementing, and, or, not, and logical shifts on fixed length lists of std-logic values. We prove 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.

6 The SCIP Processor

The SCIP's execution model is based on the FORTH programming language model. The processor 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 section 7 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`, `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 (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

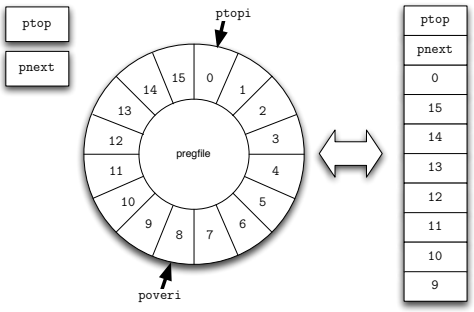

Figure 2: The pstack is assembled from the ptop register, the pnext register, and the elements of the pregfile starting at ptopi and counting down (mod 16) to poveri.

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 Figure 1 includes an arithmetic operation, a pstack operation (push), and a bit indicating that the top of the rstack should be popped in to 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 1 return operation, 4 stack operations and 4 arithmetic operations, the SCIP has 32 return+stack+arithmetic operations. In total the SCIP has 18 different instruction forms totaling over 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.

7 Proving Correctness

Although our long term goal is for a complete verification of the instruction set semantics of the SCIP processor, in this paper we focus on proving that the SCIP's implementation of push and pop operations are equivalent to applying the cons and cdr (prepend and tail) operations on the parameter stack represented as a list. 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.

7.1 Standard Operation

We must show that at the beginning of the clock cycle following 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 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 dotted lines represent data flow and originate from rectangles which show each data update. From Figure 3, the need for at least three internal steps is clear: the first step sets pstack[ptopi_plus1] and pstack[ptopi_minus1], the second step updates pstack[ptopi_next], and the third step updates ptopi_n. The hypotheses of these theorems provide similar guarantees about the state of the processor. 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. The next two hypotheses indicate that the current instruction includes the stack operation in question (push or pop). The predicate

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))))))))
```

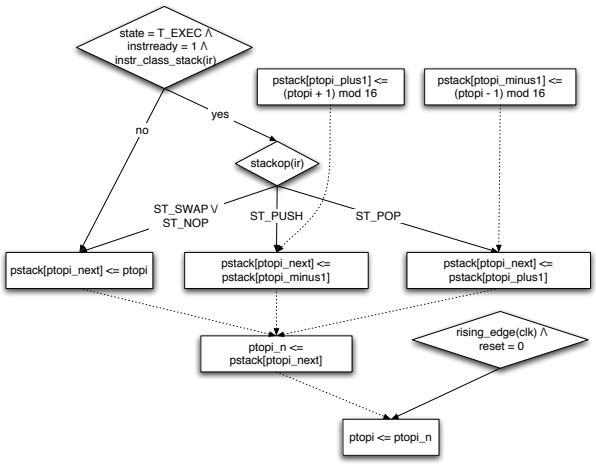

Figure 3: The control and data flow used to calculate the value for `ptopi` at the next clock cycle.

`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 final hypothesis of `scip-pop-pstack-cdr` disallows 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 can not 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.

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 two above but is otherwise analogous.

7.2 Handling Overflow and Underflow

If a push operation would cause the `ptopi` register to advance to the value of the `poveri` register the SCIP processor 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 based on 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 is 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 ALU to decrement this value, writing this value onto the `wbus`, and setting the high bits of the $\texttt{psp}_\texttt{n}$ signal to the low 15 bits of the `wbus` padded with a 0, and finally setting `psp` to $\texttt{psp}_\texttt{n}$ on the next rising clock edge. Figure 4 illustrates this process.

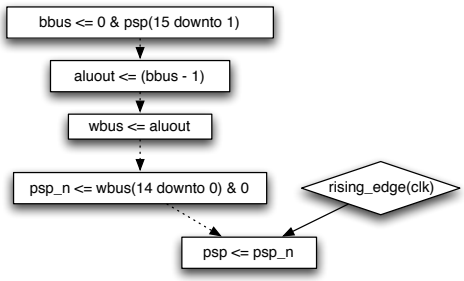

Figure 4: During the first overflow cycle, the new psp is computed by subtracting two from the original value.

The result is that the psp register is decreased by 2 (modulo 2^{16}) prior to the overflowed value being 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 prior to the update to psp. The symmetry is necessary to ensure that overflowed data is fetched properly during the next underflow event. Further, because the read request is issued during the first underflow repair cycle, the data is 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 can not be meaningfully defined anyway.

8 Conclusion

Hardware verification serves two major goals: increase confidence in hardware correctness, and facilitate verification of software at the instruction set level.

In this paper we have described our experience precisely specifying an aspect of the SCIP processor'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. This work increases our confidence in the correctness of the SCIP's design, lays the foundation for a complete specification of the SCIP's expected behavior, and for a verification that the design meets these expectations. Such a specification would give us a rigorous understanding of the SCIP's instruction set semantics, and a guarantee that the SCIP's design properly implements this specification. This could in turn serve as the basis for proofs of software behavior which would ultimately enable high confidence in the overall system behavior.

References

[1] BM79 Robert S Boyer and J Strother Moore. A computational logic, 1979.

[2] Robert S. Boyer and J. Strother Moore. Single-threaded objects in ACL2. In *PADL '02: Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages*, pages 9–27, London, UK, 2002. Springer-Verlag.

[3] Bishop Brock. Defstructure for ACL2. Technical report, 1997.

[4] P Georgelin, D Borrione, and P Ostier. A framework for vhdl combining theorem proving and symbolic simulation. In *In ACL2 Workshop*, 2002.

[5] David M Goldschlag. Mechanically verifying concurrent programs with the boyer-moore prover. *IEEE Transactions on Software Engineering SE-16*, (9), 1990.

[6] D Greve, R Richards, and M Wilding. A summary of intrinsic partitioning verification. In *Proc. Fifth Int?l Workshop ACL2 Prover and Its Applications*, 2004.

[7] W Hunt. Fm8501: A verified microprocessor. Technical report, 1985.

[8] Warren A Hunt and Bishop Brock. A formal hdl and its use in the fm9001 verification. In *Proceedings of the Royal Society*, 1992.

[9] Matt Kaufmann and J. Strother Moore. ACL2: An industrial strength version of nqthm, 1996.

[10] J Strother Moore, Tom Lynch, and Matt Kaufmann. A mechanically checked proof of the correctness of the kernel of the amd5k86 floating-point division algorithm. *IEEE Transactions on Computers*, 47, 1996.

[11] D M Russinoff. Specification and verification of gate-level vhdl models of synchronous and asynchronous circuits. Technical report, Computational Logic, Inc, 1994.

[12] Bill Young. Reverse abstraction in ACL2. In *Fifth International Workshop on the ACL2 Theorem Prover and its Applications (ACL2 '04)*, November 2004.