Formal Verification of Stack Manipulation in the SCIP Processor

J. Aaron Pendergrass
High Level

- Challenges to developing capability in formal methods:
  - Perceived high barrier to entry,
  - Specialized tools and jargon,
  - Need for a compelling but attainable demonstration.

- Why we chose the SCIP processor:
  - Developed in house,
  - General purpose processor,
  - Simple design (∼5k lines VHDL),
  - No advanced processor features (pipelining, out-of-order execution, etc.),
  - For use in satellites ⇒ high reliability requirements.
**True Goal:** To prove that the physical processor does what we want,

- proof tools work on an abstract model,
- “what we want” is not formally defined.
**True Goal:** To prove that the physical processor does what we want,

but ...

- proof tools work on an abstract model,
- “what we want” is not formally defined.

Instead we prove that a model of the VHDL design meets certain correctness properties.
Approach

Embedding of VHDL in ACL2

- Focus on building a syntactic layer on ACL2 for easy translation.
- Key Goals:
  - Incremental semantic refinements,
  - Direct manual translation of existing code,
  - Target for automated translation.

ACL2 Model of SCIP Design

- Test case for modeling framework.
- Translate VHDL code, then prove axiomatic summaries of components.
VHDL Modeling Framework

Challenges

▶ Large semantic gap between VHDL and ACL2.
  ▶ VHDL processes all execute at the same time.
▶ Human checkable translation.
  ▶ Must match structure of original VHDL code.

Solution

▶ Use ACL2 (LISP) macros to wrap ACL2 implementation behind VHDL like syntax.
  ▶ Based primarily on Georgelin, et al., “A framework for VHDL combining theorem proving and symbolic simulation.”
Supported VHDL

**Entities**
- Uses `defstructure` book to generate data type predicates, accessors, updaters, etc.
- Nested components supported via copy-in/copy-out semantics.

**Processes**
- Mapped to ACL2 functions.
- Generate theorems to guarantee some safety properties (e.g., no writing to inputs).
Supported VHDL

Architectures

- Generate a single function that is the composition of all processes and subcomponent updates.
- Generate theorems to show processes are order independent.
Supported VHDL

Architectures
- Generate a single function that is the composition of all processes and subcomponent updates.
- Generate theorems to show processes are order independent.

But Wait...
- Order independence isn’t sufficient to guarantee the processes can be safely interleaved!
Supported VHDL

Architectures
► Generate a single function that is the composition of all processes and subcomponent updates.
► Generate theorems to show processes are order independent.

But Wait...  
► Order independence isn’t sufficient to guarantee the processes can be safely interleaved!
► Fine for combinatorial processes (all of SCIP).
► Problem for sequential processes with shared state.
  ► But it is easy to change the macros to generate stronger theorems for guaranteeing determinism.
Supported VHDL

Data Types

- Originally based on ACL2’s native integer type.
  - Easy for arithmetic, challenging for bit slicing operations (concatenation, truncation, etc.).
  - Simplification of VHDL’s 9 valued logic:
    U (uninitialized), X (undefined), 0 (strong drive, logic 0), 1, (strong drive, logic 1), Z (high impedance), W (weak drive, unknown value), L (weak drive, logic 0), H (weak drive, logic 1), - (don’t care).

- Used a symbolic instruction representation to avoid complex bit operations.

- Became problematic as we added type checking because data and instructions must traverse the same buses.
Supported VHDL

Data Types

- Migrated to lists of logical symbols
  - Operations such as truncation and concatenation become structurally recursive.
  - Required very little modification to existing SCIP model (mostly search and replace).
A Simple Forth Microprocessor

- Designed for managing scientific instruments on satellites.
  - Low power, light weight, low gate count.
  - 16 and 32 bit versions (16 is standard).
- No pipelining, No superscalar, No out-of-order.
- Stack based design inspired by the Forth language.
- Two stacks: parameter stack (P-stack) and return stack (R-stack).
- Instructions may specify multiple behaviors such as an ALU operation, a P-stack modification, and a return.
SCIP Instructions

Instructions Are Packed Structures

- Only 18 different kinds of instructions.
- $\sim 9356$ different opcodes.

Basic ALU Instruction
- Pop Return Stack After Execution
- Push Result On Top Of Operand Stack
- Ignored
- Addition

1011 1 01 010 00 0010

September 24, 2010
Parameter Stack Design

▶ Many instructions can include a stack operation (Push, Pop, Swap, or Nop).

▶ Processor stacks are represented by a set of data registers and two index registers.

▶ On 16 bit SCIP: 16 2 byte data registers, 4 bit index registers.

▶ If enabled, overflow/underflow may trigger reading/writing main memory.
We’d like to show that the register ring actually implements a stack.

- In particular we need to show that the instructions correspond to abstract stack manipulation operations.
  - e.g., \( s \xrightarrow{\text{push}(a)} (a \cdot s) \)

Model stacks using ACL2 lists (\( \text{push} \equiv \text{cons} \)).

Focus on normal operation & detecting exception cases (overflow/underflow)
(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 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))))))

September 22, 2010
If the SCIP is valid and in stable state,

\[
\begin{align*}
&\text{(and (scip–pstack–inputs–ready–p st) (not (equal (scip–reset st) 1)))} \\
&\text{(not (rising–edge (scip–clk st))) (equal (scip–stretch st) 0))} \\
&\text{(instr–class–stack (scip–ir+ st))} \\
&\text{(equal (stack–op (scip–ir+ st)) *st_push*)} \\
&\text{(std–logic–defined–list–p (scip–ptopi+ st))} \\
&\text{(std–logic–defined–list–p (scip–poveri+ st))} \\
&\text{(integerp n) (>= n 3))} \\
&\text{(equal} \\
&\text{ (scip–get–pstack–regfile–as–list} \\
&\text{ (scip–step (scip–raise–clock (scip–step–n n st))))} \\
&\text{(let ((p (scip–ptopi+ st)) (o (scip–poveri+ st)))} \\
&\text{(cond} \\
&\text{ ((equal (std–logic–list–to–int p) (std–logic–list–to–int o))} \\
&\text{ (list (scip–pnext+ st)))} \\
&\text{ (t (cons (scip–pnext+ st) (scip–get–pstack–regfile–as–list st))))} \\
\end{align*}
\]
If the SCIP is valid and in stable state,

```
(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 (scip-ir+ st))
 (equal (stack-op (scip-ir+ st)) push*)
 (std-logic-defined-list-p (scip-ptoipi+ st))
 (std-logic-defined-list-p (scip-poveri+ st))
 (integerp n) (>= n 3))
```

then the P-stack at the next clock cycle, represented as a list...

```
(equal
 (scip-get-pstack-regfile-as-list
  (scip-step (scip-raise-clock (scip-step-n n st))))
 (let ((p (scip-ptoipi+ 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))))))))
```
If the SCIP is valid and in stable state,

\[
\text{and (scip-pstack-inputs-ready-p st) (not (equal (scip-reset st) 1))}
\]
\[
\text{(not (rising-edge (scip-clk st))) (equal (scip-stretch st) 0)}
\]

then the P-stack at the next clock cycle, represented as a list... 

\[
\text{(let ((p (scip-pto+ st)) (o (scip-poveri+ st)))}
\]
\[
\text{(cond}
\]
\[
\text{((equal (std-logic-defined-list-p (scip-pto+ st))}
\]
\[
\text{(equal (scip-get-pstack-regfile-as-list st) (scip-step (scip-raise-clock st)))}
\]
\[
\text{((equal (std-logic-list-to-int p) (std-logic-list-to-int o))}
\]
\[
\text{(list (scip-pnext+ st)))}
\]
\[
\text{(t (cons (scip-pnext+ st) (scip-get-pstack-regfile-as-list st))))}}\]

... is equal to the original pnext register cons'ed onto the original P-stack represented as a list.
Contributions

VHDL Modeling Framework

- Human readable/writable framework for modeling VHDL in ACL2.
- Automates generation of basic sanity theorems.
- Supports incremental refinements of VHDL semantics with little damage to target system model.

SCIP Processor Verification

- Demonstration of framework viability.
- Models of nearly every functional entity of the SCIP processor.
- Significant correctness proof for key SCIP functionality (stack manipulation).
Future Work

Framework Enhancement

▶ Improving auto-generated theorems (strength and proof speed).
▶ Machine translation tool.

SCIP Specific Proofs

▶ Overflow and underflow behavior.
  ▶ Proved correct detection of overflow/underflow condition.
  ▶ Memory model and axiomatic definition of page relative addressing begun.
▶ Return stack.
▶ End goal is full instruction set specification.