ACL2VHDL Translator:
A Simple Approach to Fill the Semantic Gap

Jun Sawada
IBM Austin Research Laboratory
11501 Burnet Road, MS 904-6H012
Austin, TX 78758
E-mail: sawada@austin.ibm.com
Overview of the Talk

- Language Translation for Hardware Verification
- Our Approach
- Bit Vector Library
- Translation of ACL2 to VHDL
- Application
- Discussion
Two Approaches in Hardware Verification by ACL2

- Proof about abstract models written in the ACL2 language.
  - Pro: Direct. Easy.
  - Cons: Is the model a correct representation of actual HW?
- Proof on hardware written in an HDL, such as Verilog and VHDL.
  - Pro: We get results on actual HW, essential for industry.
  - Cons: Tedious proof about the low-level details of HW.
  - Cons: May require to change the proof when HW changes.
  - Cons: Need a translation from HDL to ACL2. This is not easy.
Two Approaches for Language Translation

• Deep Embedding (e.g DUAL-EVAL)
  – Define an ACL2 interpreter of an HDL.
  – Analyze the evaluation process by the interpreter.
  – Proof can be tedious and confusing because of two-step reasoning.

• Shallow Embedding
  – Use a language translator from an HDL to the ACL2 language.
  – Analyze the result of translation.
  – Translate is more likely to be incorrect: semantic gap.
Typical Problems in Language Translation

- Data types that are not isomorphic.
  - E.g. In ACL2, NIL is both false and an empty list, unlike ML.
    - When translating ACL2 to ML, how to translate NIL?
- Some languages are not well-defined.
  - E.g. C arrays of size bigger than $2^{32}$. 
Why difficult to translate VHDL?

- Many language features make it difficult to write a complete translator between VHDL and ACL2.
  - Entity and Architecture.
  - Delayed actions.
  - Generics.
  - Sequential and Concurrent Assignment.
    - e.g. Incrementer with input x and output inc in concurrent assignment.

```vhdl
carry(32) <= '1';
carry(0 to 31) <= x and carry(1 to 32);
inc(0 to 31) <= din xor carry(1 to 32);
```
How it has been done?

• Typically people write translators from HDL to ACL2 (with some restriction.)
  – Georgelin, Borrione, Ostier 2002
  – Russinoff 1998

• It is laborious to write a complete translator.
Our Approach

• A New Approach

  – Define a translator from ACL2 functions to VHDL functions.
  – Translatable ACL2 functions are defined in terms of a bit vector library.
  – VHDL-level verification tools use the result of translation.
  – Translator does not handle anything like delays, ports, and clocks.

• Why?

  – We only translate a subset of ACL2 language which can be mapped directly to VHDL without loss of semantics.
  – Use ACL2 only for the analysis of algorithms and specifications of HW.
  – Proof on algorithms does not need to change, even if hardware changes.
  – VHDL verification tools are responsible for handling delays, clocks, anything that are related to the actual implementation of hardware.
Overview of the Talk

- Language Translation for Hardware Verification
- Our Approach
- Bit Vector Library
- Translation of ACL2 to VHDL
- Application
- Discussion
IHS : ACL2’s Bit Vector Library

- IHS (Integer Hardware Specification) Library by Bishop Brock.
  - A bit is represented by a 0 and 1.
  - A bit vector is represented by an integer.
  - Speedy simulations with many supporting lemmas.
  - Not adequate for the language translation between ACL2 and HDL.
  - 0 and 1 represent both bit and bit vectors.
  - An integer can represent bit vector of many different length.
    - e.g. No way to tell a 32-bit bit vector from a 64-bit bit vector.
  - Hard to define some functions
    - e.g. a function returning the most significant bit of a bit vector.
A new bit vector library: BV Library

- A bit is defined as (bit 0) or (bit 1).
- A bit vector is defined as (BV val size).
  - val is the integer value of the bit vector and size is the length.
- Many basic operations are defined as functions.
- BV library is built on top of IHS.
  - e.g. Bit concatenation function bv\& is defined in terms of logapp from the IHS library.

(defun bv\& (a b)
  (declare (xargs :guard (and (bvp a) (bvp b)))))
  (bv (logapp (bv-size b) (bv-val b) (bv-val a))
      (+ (bv-size a) (bv-size b))))
Some Basic Functions in the BV library.

- (b-not b): Bit negate.
- (bv-not bv): Bit vector negate.
- (msb bv): Returns the most significant bit of bv.
- (bits bv i j): Sub range of a bit vector from i’th bit to j’th.
- (bv& bv0 bv1): Concatenation.
- (bv+ bv0 bv1): Addition.
- (bv<< bv sh): Shift to left.
- (bv-gt? bv0 bv1): Greater-than relation.
- (bv-if b0 bv0 bv1): if-then-else.
BV Library Summary

- Quite powerful library to specify functions on bit and bit vectors.
- A floating-point instructions of a PowerPC™ media unit has been specified.
- Many lemmas from the IHS library are or can be imported to the BV library.
  - However, need more work to expand it.
ACL2VHDL Translator

- Translates ACL2 function defined in terms of the functions from the BV library, let and let*.
  - No if-statement. Use bv-if or b-if.
  - No recursive functions.

- Conversion Process:
  - Parsing.
  - Type Inference.
  - VHDL code generation.
Conversion Tricks

- We need to implement a type inference system, because ACL2 language is dynamically typed, but VHDL is statically typed.
- Code generation is simple mapping as all BV library functions are re-defined in VHDL.
- Name conflicts in the let expressions are resolved by renaming.
- Addition of 32-bit integer types in ACL2, since VHDL integers are 32-bit integers.
Carry Generation in ACL2

(defun lc8 (v8)
  (declare (xargs :guard (and (bvp v8)
                              (equal (bv-size v8) 8)))))

(b&& (bv-and-all (bits v8 1 7))
     (bv-and-all (bits v8 2 7))
     (bv-and-all (bits v8 3 7))
     (bv-and-all (bits v8 4 7))
     (bv-and-all (bits v8 5 7))
     (bv-and-all (bits v8 6 7))
     (bitn 7 v8)
     *b1*)))
Translated Carry Function

function lc8(v8 : std_ulogic_vector)
    return std_ulogic_vector is
    variable result : std_ulogic_vector(0 to 7);
begin
    result := (and_reduce(bits(v8,1,7)) &
    (and_reduce(bits(v8,2,7)) &
    (and_reduce(bits(v8,3,7)) &
    (and_reduce(bits(v8,4,7)) &
    (and_reduce(bits(v8,5,7)) &
    (and_reduce(bits(v8,6,7)) &
    (bitn(7,v8) &
    b2bv(b1))))))))));

    return result;
end lc8;
Carry look ahead signal in ACL2

(defun gc8 (v32)
  (declare (xargs :guard (and (bvp v32)
                              (equal (bv-size v32) 32)))))
  (b&& (bv-and-all (bits v32 8 31))
       (bv-and-all (bits v32 16 31))
       (bv-and-all (bits v32 24 31))))
function gc8(v32 : std_ulogic_vector)
  return std_ulogic_vector is
  variable result : std_ulogic_vector(0 to 2);
begin
  result := (and_reduce(bits(v32,8,31)) &
    (and_reduce(bits(v32,16,31)) &
      b2bv(and_reduce(bits(v32,24,31)))))�;
  return result;
end gc8;
Carry and Increment

(defun carry32 (v32)
  (declare (xargs :guard (and (bvp v32)
                                (equal (bv-size v32) 32)))
    (let ((lc_0_7 (lc8 (bits v32 0 7)))
          (lc_8_15 (lc8 (bits v32 8 15)))
          (lc_16_23 (lc8 (bits v32 16 23)))
          (lc_24_31 (lc8 (bits v32 24 31)))
          (gc (gc8 v32)))
      (bv&& (bv-if (bitn 0 gc) lc_0_7 (bv 0 8))
             (bv-if (bitn 1 gc) lc_8_15 (bv 0 8))
             (bv-if (bitn 2 gc) lc_16_23 (bv 0 8))
             lc_24_31)))

(defun inc2 (v32)
  (declare (xargs :guard (and (bvp v32)
                                (equal (bv-size v32) 32)))
    (bv-xor v32 (carry32 v32)))
Carry and Increment in VHDL

function carry32(v32 : std_ulogic_vector) return std_ulogic_vector is
    variable lc_0_7 : std_ulogic_vector(0 to 7);
    variable lc_8_15 : std_ulogic_vector(0 to 7);
    variable lc_16_23 : std_ulogic_vector(0 to 7);
    variable lc_24_31 : std_ulogic_vector(0 to 7);
    variable gc : std_ulogic_vector(0 to 2);
    variable result : std_ulogic_vector(0 to 31);
begin
    lc_0_7 := lc8(bits(v32,0,7));
    lc_8_15 := lc8(bits(v32,8,15));
    lc_16_23 := lc8(bits(v32,16,23));
    lc_24_31 := lc8(bits(v32,24,31));
    gc := gc8(v32);
Carry and Increment in VHDL : Continued

result := (ite(bitn(0,gc),lc_0_7,bv(X"0",8)) &
            (ite(bitn(1,gc),lc_8_15,bv(X"0",8)) &
             (ite(bitn(2,gc),lc_16_23,bv(X"0",8)) &
              lc_24_31)));

    return result;
end carry32;

function inc2(v32 : std_ulogic_vector)
    return std_ulogic_vector is
    variable result : std_ulogic_vector(0 to 31);
begin
    result := (v32 xor carry32(v32));
    return result;
end inc2;
A Simple Incrementer

- In ACL2
  (defun inc (v0)
    (declare (xargs :guard (and (bvp v0)
                                (equal (bv-size v0) 32))))
    (bv+ v0 (bv 1 32)))

- In VHDL translation:

  function inc(v0 : std_ulogic_vector)
  return std_ulogic_vector is
  variable result : std_ulogic_vector(0 to 31);
  begin
    result := (v0 + bv(X"1",32));
    return result;
  end inc;
Application

- Floating-Point instruction verification.
- Multiplier Verification
Verification of Floating-Point Instruction

- We wrote a specification of floating-point instructions for a media unit.
- ACL2VHDL was used to the spec to VHDL.
- Run a SAT procedure to verify precision conversion instructions implemented in hardware.

\[ \text{ACL2 spec of instructions} \]
\[ \text{ACL2VHDL} \]
\[ \text{VHDL ref. model} \]
\[ \text{HW VHDL} \]
\[ \text{SAT} \]
\[ \text{Test Driver} \]
\[ \text{yes/no} \]
Multiplier Verification

• Sandip Ray worked on the integration of ACL2 and SixthSense, an IBM internal tool for VHDL verification.

• Working on a multiplier built in terms of Booth encoder and Carry Save Adders.

• We proved the correctness of multiplier algorithm in ACL2, provided that the hardware satisfies a number of proof obligations.

• The proof obligations are translated, checked by SixthSense and then imported back into ACL2 as a theorem.

• This is an on-going work.
Multiplier Verification Flow

ACL2 world

ACL2 Property

ACL2VHDL

Translated property in VHDL

ACL2 Theorem

SixthSense

Test Driver

VHDL world
Discussion

• Bare-bone translator against full-fledged translators.
  – ACL2VHDL is a simple translator, but flexible for wide applications.
  – No knowledge about time, unit, port, etc.
• Recursive functions are useful or not?
  – VHDL has a limited form of recursive functions.
  – if-statement needs to be added to the conversion.
  – Currently, recursive functions can be used in verification with extra steps.
    – First verify theorems using recursive functions.
    – Define and prove there is an non-recursive function equivalent to it.
    – Use the non-recursive version functions for HW verification.