Combining ACL2 and an Automated Verification Tool to Verify a Multiplier

Jun Sawada and Erik Reeber
IBM Austin Research Laboratory
University of Texas at Austin

August 16, 2006

Introduction

- Implemented prototype mechanism for extending ACL2 with external tools
- Integrated IBM's SixthSense Verification Tool to ACL2
  - Use SixthSense to verify smaller properties automatically.
  - Use ACL2 to prove problems too difficult to verify with SixthSense.
- Applied the technique to the verification of an industrial multiplier design written in VHDL.

Outline

- Prototype External Tool Mechanism
- ACL2SIX: Extending ACL2 with SixthSense
- Multiplier Design
- Booth Encoder Verification
- Compression Verification
- Conclusion

Prototype External Tool Mechanism

- A new ACL2 hint that extends the ACL2 theorem prover with functions that implement
  - new theorem proving procedures
  - external tool interfaces
- Extension is dynamic
  - Implemented as program-mode functions
- Prototype modifies ACL2 source
  - Only 57 lines of modification
- To-do list entry contains additional features
  - Allows users to declare trusted clause-processors

:External Example

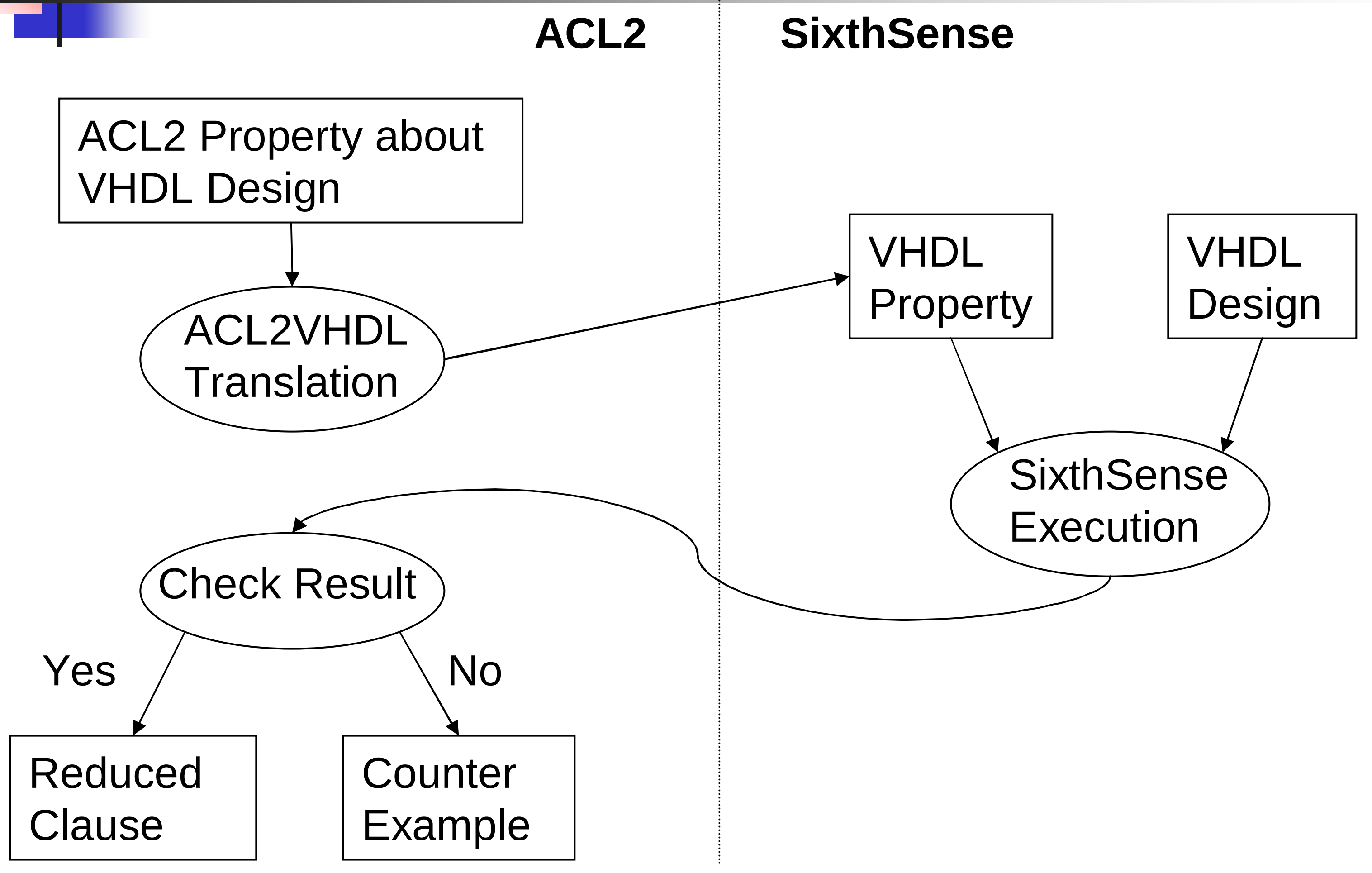
```
(defun generalize-expr (clause expr new-var state)
  (cond
   ((or (not (symbolp new-var))
        (var-in-expr-listp new-var clause))
    (mv (list "ERROR: Target must be a new variable~%”)
        nil
        state))
   (t
    (mv nil
        (list (substitute-expr-list expr new-var clause))
         state))))
```

```
(thm (implies (and (natp a) (natp (foo)))
              (equal (nthcdr a (nthcdr (foo) x))
                     (nthcdr (+ a (foo)) x)))
     :hints (("Goal" :external (generalize-expr '(foo) 'b))
             ("Goal'" :induct (nthcdr b x))))
```

SixthSense

- We use the :external extension mechanism to integrate ACL2 with *SixthSense*
- IBM internal verification tool
- Operates on a finite-state machine described in VHDL.
- Uses transformation-based verification approach
  - BDDs & SAT Solvers
  - Re-timing engine
  - Semi-formal counter-example search engine
- It formally proves safety properties of FSMs
- When a property is found invalid, it returns a counter example as a waveform.

ACL2SIX Flow Chart

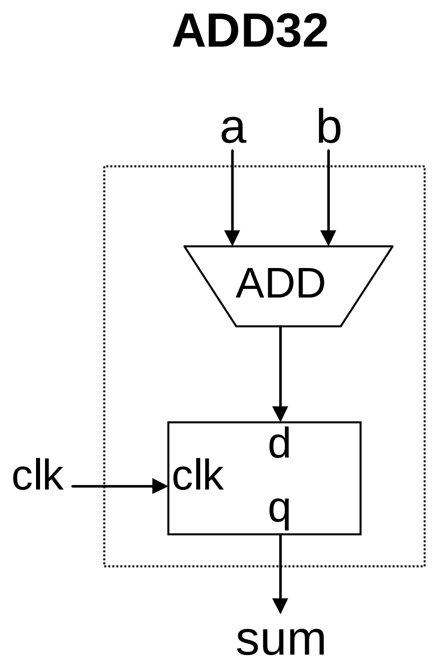


ACL2SIX Extension

- There is no ACL2 model of hardware design!
- VHDL signals are represented in ACL2 logic with function stubs `sigbit` and `sigvec`:

  ```
  (sigbit entity signame cycle phase)
  (sigvec entity signame (lbit hbit) cycle phase)
  ```
- ACL2SIX translates these stubs to the appropriate signals in the VHDL design.
- Besides `sigbit` and `sigvec`, only ACL2VHDL primitives, such as `bv+`, `bv-and`, and `bv-or` can be used in the verified property.

ACL2SIX Example



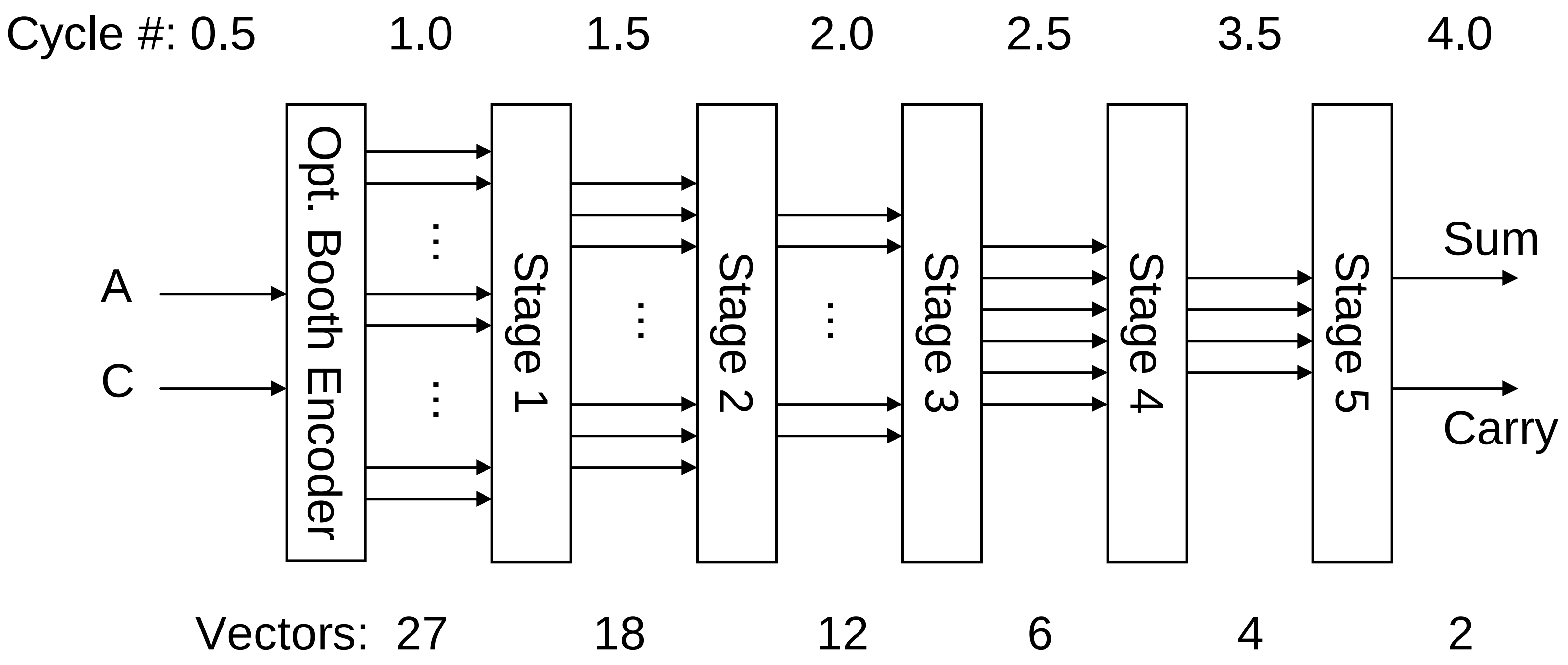
```
(defun add32 ()
 '(add32
   (port
    (clk :in std_ulogic)
    (a :in std_ulogic_vector (0 31))
    (b :in std_ulogic_vector (0 31))
    (sum :out std_ulogic_vector (0 31)))
   (extra-assigns (clk "c0")))

(defthm adder-adds
 (implies
  (and (integerp n) (<= 1 n))
  (equal
   (bv+ (sigvec (add32) a (0 31) (1- n) 2)
        (sigvec (add32) b (0 31) (1- n) 2))
   (sigvec (add32) sum (0 31) n 2)))
 :hints
 (("Goal" :external
          (acl2six ((:cycle-expr n)
                    (:ignore-init-cycles 1))))))
```

Booth Multiplier

- 53bit x 54bit multiplier
- Used to compute double-precision floating-point multiplication
- Written in VHDL
- Output consists of two vectors, whose sum is equal to its product.
- Uses Booth-encoding algorithm, with a number of carry-save adder stages.
- Sixthsense cannot verify entire system, or even a single stage of the multiplier.

Multiplier Dataflow



A X C = Sum + Carry

Multiplier Correctness Theorem

```
(defthm multiplier-correct
 (implies
  (and (integerp n)
       (<= 7 n))
  (equal (bv+ (Sum-output n 1)
              (Carry-output n 1))
         (bv (* (bv-val (A-input (- n 4) 2))
                (bv-val (C-input (- n 4) 2)))
             108))))
```

- `Bv+` computes the binary sum.
- `(bv i n)` returns the `n`-bit vector representing `i`.
- Input `A-input` and `C-input` defined using `sigvec`.
- Similarly with Output `Sum-output` and `Carry-output`.

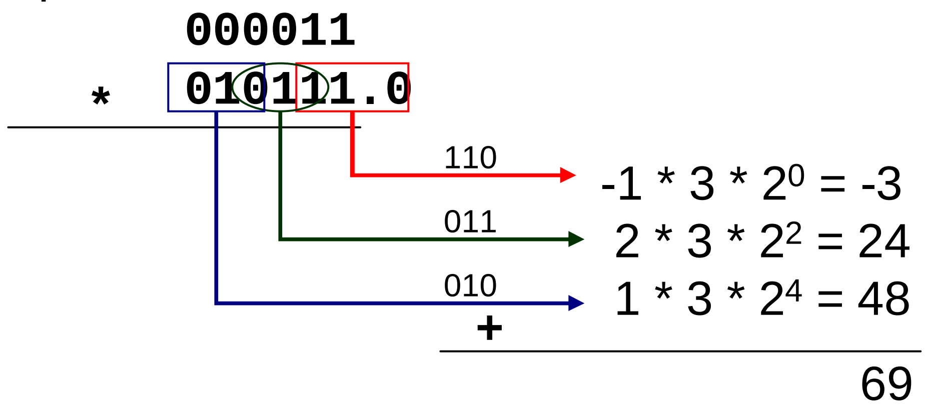
Booth Encoder

- Reduces the multiplication to summation
  - Half as many partial-products of the grade-school method.
  - Two’s Complement Notation
  - Looks at three bits at a time

Encoding Table

100	→ -2 * y
101	→ -1 * y
110	→ -1 * y
111	→ 0 * y
000	→ 0 * y
001	→ 1 * y
010	→ 1 * y
011	→ 2 * y

Example: 23 * 3



$-1 * 3 * 2^0 = -3$

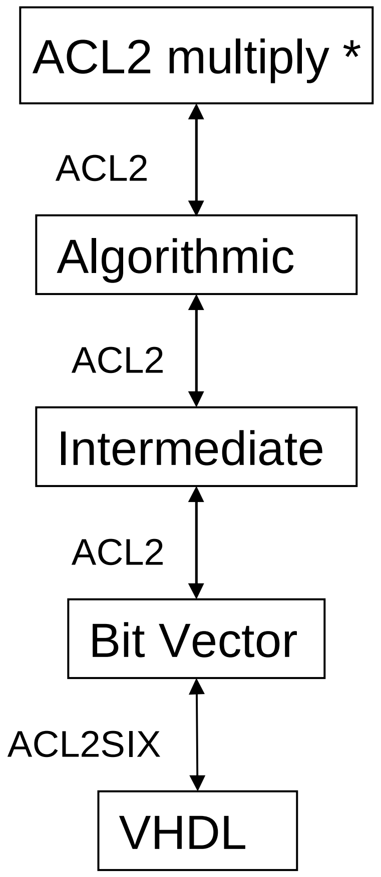
$2 * 3 * 2^2 = 24$

$1 * 3 * 2^4 = 48$

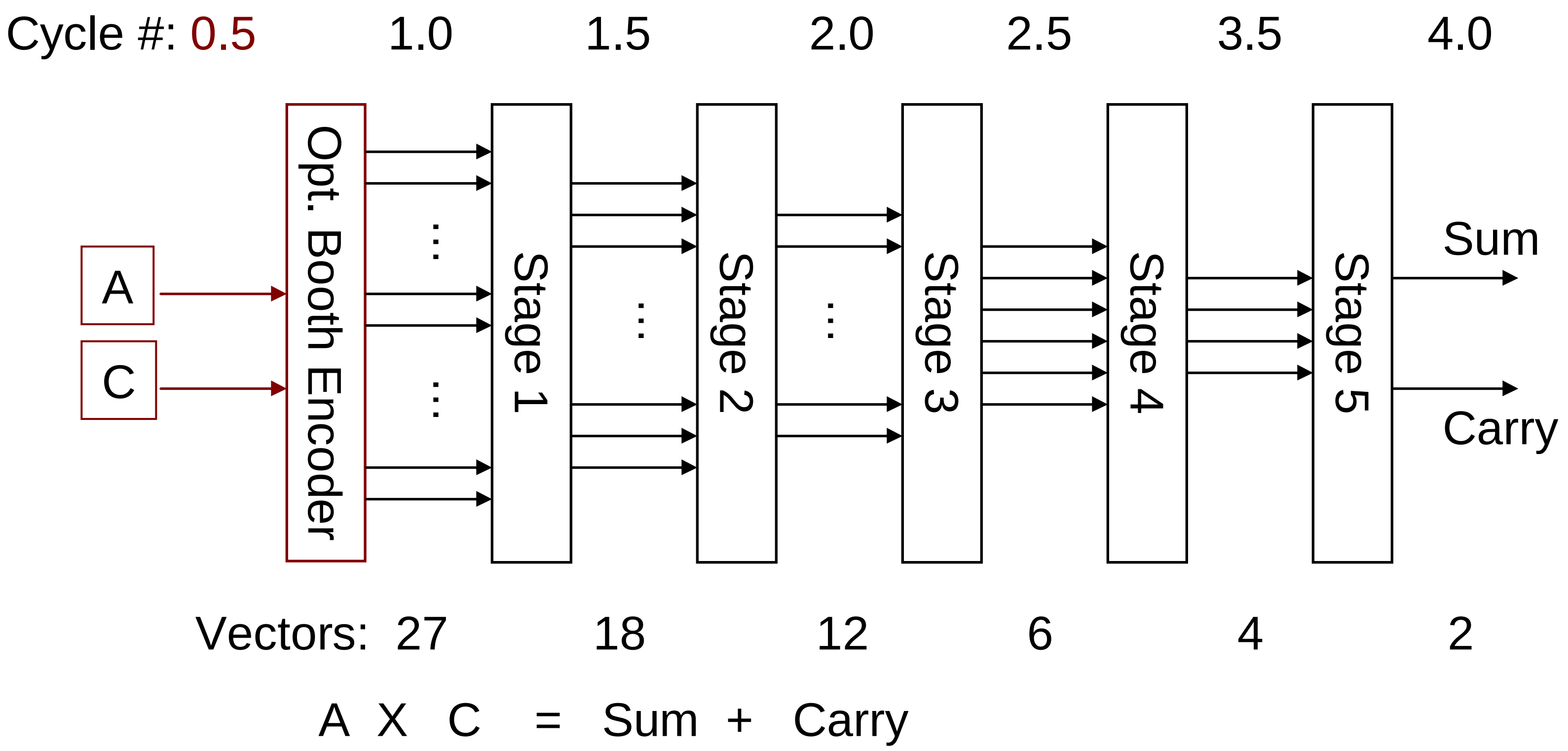
69

Levels of Booth Encoder Models

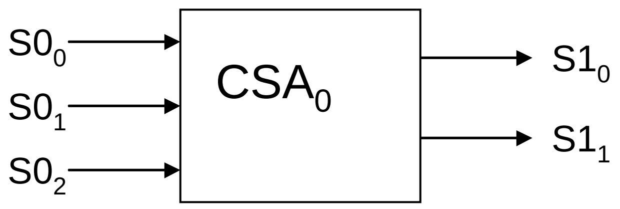
- Algorithmic ACL2 Model
  - Algorithms of n-bit Booth Encoder
  - 19 lines of ACL2
  - Verified to implement a multiplier by induction
- Intermediate ACL2 Model
  - Stepping stone between algorithmic and bit vector models
- Bit Vector ACL2 Model
  - Only using subset of ACL2 that is translatable to VHDL
- VHDL Model
  - High-performance industrial design
  - Optimized to decrease # wires
  - Equivalent to Bit Vector Model, by SixthSense



Multiplier Dataflow



Compression Algorithm



$$S0_0 + S0_1 + S0_2 = S1_0 + S1_1$$

- 3-to-2 Carry-Save Adder (CSA) takes 3 inputs and produces 2 outputs, preserving the sum.
- 4-to-2 CSA reduces 4 inputs to 2.
- Compression Stage 1 consists of nine 3-to-2 CSAs.
- Verifying sum-preservation on a single CSA can be done by SixthSense, but not nine CSAs combined.

Compression Verification

- Use SixthSense to sum preservation of CSA
  - e.g., $S1_0 + S1_1 = S0_0 + S0_1 + S0_2$
- Make a rewrite rule to help simplification.
  - e.g., $S1_0 = S0_0 + S0_1 + S0_2 - S1_1$
- Chain of rewriting (with assoc. rules).

$S1_0 + S1_1 + S1_2 + \ldots\ldots + S1_{17}$

$\Rightarrow S0_0 + S0_1 + S0_2 - S1_1 + S1_1 + S1_2 + \ldots\ldots + S1_{17}$

$\Rightarrow S0_0 + S0_1 + S0_2 + S1_2 + \ldots\ldots + S1_{17}$

…

$\Rightarrow S0_0 + S0_1 + S0_2 + S0_3 + \ldots\ldots + S0_{26}$

Multiplier Verification

- Combine with Booth Encoder verification
    - $S5_0 + S5_1 = A * C$
- Analysis
    - No bugs
    - Increased assurance
    - Can re-run proof if multiplier is modified
        - Low-level modifications only are seen by SixthSense!
    - About one month of human effort
        - Sixthsense: 7 work days
        - ACL2: 14 work days

Conclusion

- Added prototype mechanism for extending ACL2 with external tools
- Integrated SixthSense and ACL2
  - Avoided most of the VHDL semantics
  - Improved automation in verification of VHDL designs
  - Provided counter-example generation
- Applied to multiplier verification
  - All low-level details are verified automatically by SixthSense.
  - Beyond scope of SixthSense alone