Centaur Verification Approach

Jared Davis, Warren Hunt, Jr., Anna Slobodova, Sol Swords
Bob Boyer, Gary Byers, Matt Kaufmann, Robert Krug

November, 2010

Computer Sciences Department
University of Texas
1 University Way, M/S C0500
Austin, TX 78712-0233

hunt@cs.utexas.edu
TEL: +1 512 471 9748
FAX: +1 512 471 8885

Centaur Technology, Inc.
7600-C N. Capital of Texas Hwy
Suite 300
Austin, Texas 78731

hunt@centtech.com
TEL: +1 512 418 5797
FAX: +1 512 794 0717

Outline

Introduction

We have verified add, sub, multiply, divide (microcode), compare, convert, logical, shuffle, blend, insert, extract, min-max instructions from Centaur's 64-bit, X86-compatible, Nano^{TM} microprocessor.

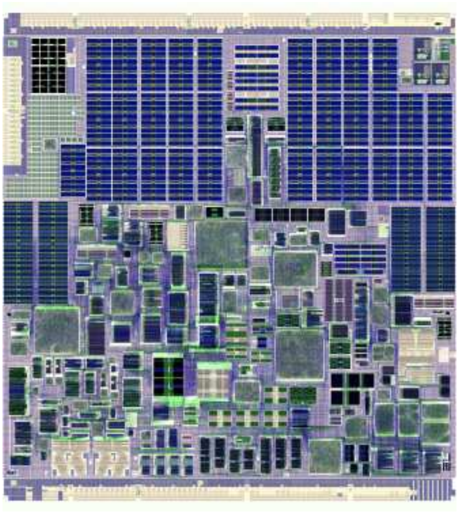
- Media unit implements over 100 X86 SSE and X87 instructions.
- Multiplier implements scalar & packed X86, X87, and FMA.

For our verifications, we use a combination of AIG- and BDD-based symbolic simulation, case splitting, and theorem proving.

- We create a theorem for each instruction to be verified.
- We use ACL2 to mechanically verify each proposed theorem.

We discuss our verification approach for formally verifying execution-unit instructions for the Centaur Nano^{TM} – the Nano^{TM} is used by Dell, HP, Lenovo, OLPC, and Samsung.

VIA Nano^{TM} X86-64 Microprocessor



Contemporary Example

- Full X86-64 design including VMX
- 40-nanometer design of 97.5M transistors
- AES, DES, SHA, and random-number generator hardware
- Built-in security processor
- Runs 40 operating systems and four VMs

Centaur Technology

Centaur Technology, Inc., is a whole-owned subsidiary of VIA.

- Entire X86 processor design team is in Austin, Texas
- 100+ people specify, design, validate, bring up, test, build burn-in fixtures and programs – everything but chip manufacturing
  - Roughly 20 people write RTL
  - Around 20 work in validation
  - Approximately 25 work in design
  - About 30 work in test, manufacturing, bring up
  - Three systems support
  - Ten or so group leads, flat management
  - Three support (payroll, benefits, reception, etc.)
  - FV group is about 4 FTEs – high ratio!

Extremely efficient organization, flat management, everyone expected to pull their own weight and then some...

Centaur 64-bit Design Comments

X86 designs are complicated, and to be cost and performance competitive, they are necessarily full custom.

- Low cost, small size, low power
- 64-bit (Intel EMT64-compatible) architecture
- Virtual Machine (Intel VMX-compatible) design
- Latest SSEx instructions
- 64-bit EA, 48-bit Virtual Address
- 40-bit Physical Address

Targeted at low-power, low-cost applications: netbooks, low-power workstations, and embedded designs.

Core Technology: ACL2

Our work is based on the ACL2 logic and its mechanical theorem prover.

- First-order predicate calculus with recursion and equality.
- Atomic data objects
  - Complex rationals: `5`, `-12`, `3/4`, `\#C(3 4)`
  - Characters: `#\a`, `#\8`, `#\Tab`
  - Strings: `"abc"`, `"aBc"`, `"ABC"`
  - Symbols: `X`, `DEF`, `|abc|`, `|54-fifty4|`
- Data constructor
  - Pairs: `(CONS 7 "ghi")`, `'(7 . "ghi")`
  - Sophisticated quotation and abbreviation mechanisms
- Functions – subset of Common Lisp
  - 31 primitive functions
  - 200+ defined functions
  - Guards defined for all functions

Fibonacci Function Example

```
(defun fib (x)
  (declare (xargs :guard (natp x)))
  (mbe :logic
       (if (zp x)
           0
         (if (= x 1)
             1
           (+ (fib (- x 2)) (fib (- x 1)))))
       :exec
       (if (< x 2)
           x
         (+ (fib (- x 2)) (fib (- x 1))))))
```

Any such function can be memoized.

```
(memoize 'fib :condition '(< 40 x))
```

Equivalent Function Proof Statement

```
(defun f1 (fx-1 fx n-more)
  (declare (xargs :guard (and (natp fx-1)
                              (natp fx)
                              (natp n-more))))
  (if (zp n-more)
      fx
    (f1 fx (+ fx-1 fx) (1- n-more))))

(defun fib2 (x)
  (declare (xargs :guard (natp x)))
  (if (zp x)
      x
    (f1 0 1 (1- x))))

(defthm fib2-is-fib
  (implies (natp x)
           (equal (fib2 x)
                  (fib x))))
```

Symbolic Simulation Proof Examples

An obvious observation about the factorial function.

```
(def-gl-thm fib-in-range
   :hyp (and (natp x)
             (<= 4 x) (<= x 6))
   :concl (or (equal (fib x) 3)
              (equal (fib x) 5)
              (equal (fib x) 8))
   :g-bindings ‘((x ,(g-number (list (list 0 1 2 3)))))
   :rule-classes nil)
```

A simple arithmetic fact.

```
(def-gl-thm 4-5-6-is-less-than-7-8-9
   :hyp (and (natp x)  (natp y)
             (<= 4 x)  (<= 7 y)
             (<= x 6)  (<= y 9))
   :concl (< x y)
   :g-bindings ‘((x ,(g-number (list (list 0 1 2 3 4))))
                 (y ,(g-number (list (list 5 6 7 8 9)))))
   :rule-classes nil)
```

Symbolic Simulation in ACL2

We have developed a verified framework for ACL2 that provides a means for symbolic simulation.

- Defined functions can be mechanically generalized.
- Each mechanically defined generalized function is automatically verified.
- Such generalized functions, given finite sets, can be symbolically executed.
- Our framework allows the results of symbolic simulation of ACL2 functions to be used as a part of a proof.

Our work provides a symbolic-simulation capability for the entire ACL2 logic.

A Simple Embedded Language

To illustrate embedding a HDL within ACL2, we define the semantics of a Boolean logic based on `IF` trees.

```
(defun if-termp (term)
   (declare (xargs :guard t))
   (if (atom term)
       (eqlablep term)
     (let ((fn (car term))
           (args (cdr term)))
       (and (consp args)
            (consp (cdr args))
            (consp (cddr args))
            (null  (cdddr args))
            (eql fn 'if)
            (if-termp (car args))
            (if-termp (cadr args))
            (if-termp (caddr args))))))
```

```
(defun if-evl (term alist)
   (declare
    (xargs :guard
           (and (if-termp term)
                (eqlable-alistp alist))))
   (if (atom term)
       (cdr (assoc term alist))
     (if (if-evl (cadr term) alist)
         (if-evl (caddr term) alist)
       (if-evl (cadddr term) alist))))
```

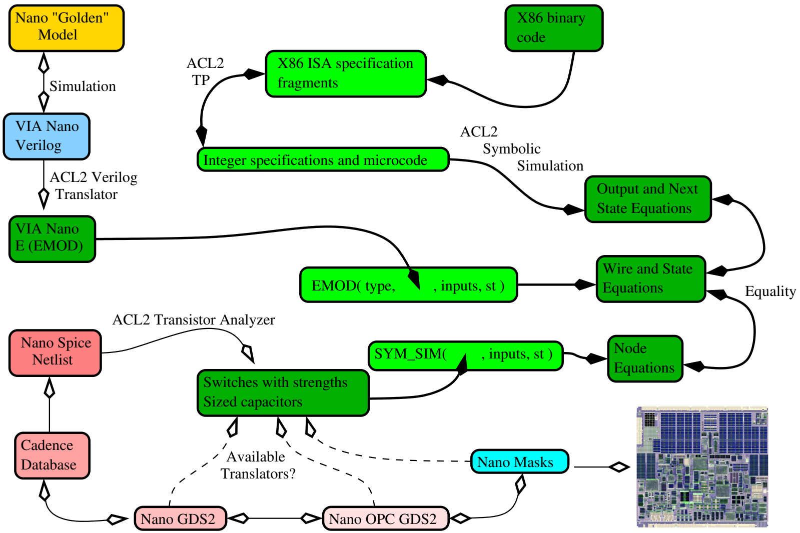
Example IF Tree and Verification by Symbolic Execution

```
(to-if '(implies (and x y) (or x y)))
  ==>
'(IF (IF X Y NIL) (IF X T Y) T)
```

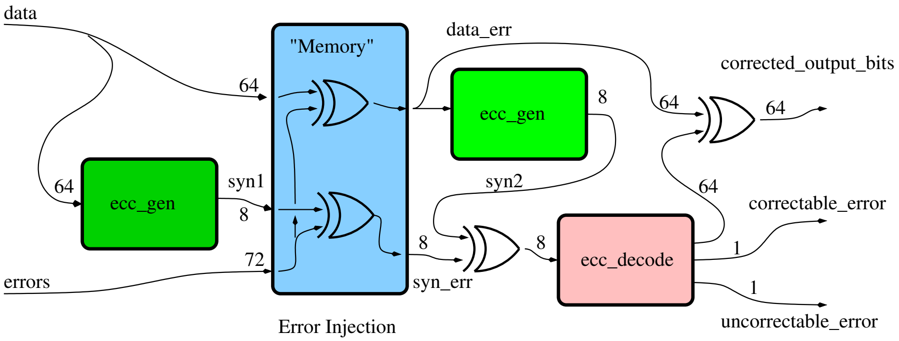
Our language of IF trees only contains one logical connective.

```
(def-gl-thm if-evl-example
  :hyp (and (booleanp a) (booleanp b))

  :concl (if-evl '(IF (IF X Y NIL) (IF X T Y) T)
                 `((NIL . nil)
                   (T   . t)
                   (X   . ,a)
                   (Y   . ,b)))

   :g-bindings `((a ,(g-boolean 0))
                 (b ,(g-boolean 1))))
```

The Centaur Verification Tool Relationships



ECC Example



Model to analyze the ECC circuitry.

- Syndrome unit produces error-correcting code
- ECC unit decodes syndrome to produce 1-hot, correction position

Verilog for ECC Model

```
module ecc_model (data,                      // Input Data
                  errors,                    // Error Injection
                  corrected_output_bits,     // Output Data
                  correctable_error,         // Corrected?
                  uncorrectable_error);      // Can't be corrected

   ecc_gen gen1 (syn1, data);      // Generate syndrome bits for "memory"

   assign         data_err = data ^ errors[63:0];   // Fault injection
   assign         syn_err  = syn1 ^ errors[71:64];  // Fault injection

   ecc_gen gen2 (syn2, data_err);  // Syndrome bits for "memory" output

   assign         syn_backwards_xor = syn_err ^ syn2;  // Compute syndrome

   ecc_decode make_outs (bit_to_correct,        // One-Hot output correction
                         correctable_error,     // Correctable error?
                         uncorrectable_error,   // UnCorrectable error?
                         syn_backwards_xor);    // Syndrome input

   assign         corrected_output_bits = bit_to_correct ^ data_err;
endmodule
```

E-Language for ECC Model

```
(:n  |*ecc_model*|

 :i (|data[0]| |data[1]| |data[2]| |data[3]| |data[4]|
     |data[5]| |data[6]| |data[7]| |data[8]| |data[9]| ...)

 :o (|corrected_output_bits[0]| |corrected_output_bits[1]|
     |corrected_output_bits[2]| |corrected_output_bits[3]|
     |corrected_output_bits[4]| |corrected_output_bits[5]|
     |corrected_output_bits[6]| |corrected_output_bits[7]|
     |corrected_output_bits[8]| |corrected_output_bits[9]| ...)

 :occ ((:full-i #@53# :full-o #@54# :u |_gen_3|
                 :op #.*vl_64_bit_buf* :o #@55# . #@56#)
       (:full-i #@57# :full-o #@58# :u |_gen_4|
                 :op #.*vl_8_bit_buf* :o #@59# . #@60#)
       (:full-i #@61# :full-o #@62# :u |_gen_5|
                 :op #.*vl_64_bit_pointwise_xor*
                 :o #@63# . #@64#)
       (:full-i #@19# :full-o #@20#
                 :u |gen1|
                 :op #.|*ecc_gen*|
                 :o #@21#
                 :i #@22#)  ...   ))
```

ACL2 Specification for ECC Model

```
(defn our-one-bit-error-predicate (bad-bit)
  ;; Check output correctness if one error injected.
  (declare (xargs :guard (natp bad-bit)))
  (let* ((data   (qv-list 0 1 64))
         (errors (q-not-nth bad-bit
                            (make-list 72 :initial-element nil)))
         (inputs (ap data errors)))
    (equal (mv-let (s o)
             (emod 'two |*ecc_model*| inputs nil)
             (declare (ignore s))
             (list :corrected-bits
                   (take 64 o)
                   :correctable_error
                   (nth 64 o)
                   :uncorrectable_error
                   (nth 65 o)))
           (list :corrected-bits
                 data
                 :correctable_error
                 (< bad-bit 64)
                 :uncorrectable_error
                 NIL))))
```

Centaur Formal Verification Toolflow

We begin, by translating Nano's Verilog specification into our formally-defined, **E**-language HDL.

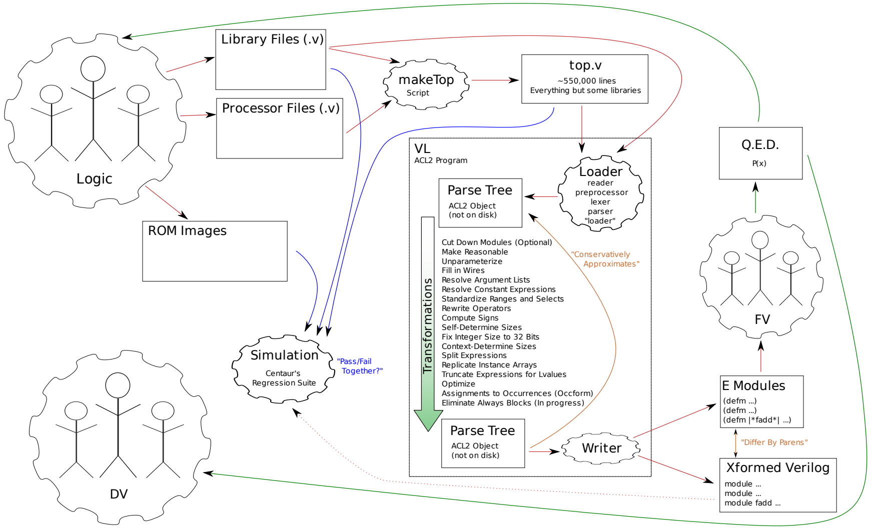
- Verilog is simplified into *single-assignment* form.
- Create environment suitable for media unit verification.
- We extract its *equation* by symbolic simulation.
- We specialize this equation to the instruction of interest.
- We then, as appropriate, convert this equation into BDDs.

The specification is written in ACL2.

- Integer operations are used to specify media-unit instructions.
- Such operations are symbolically simulated and specialized.
- These specification are proven to implement floating-point operations.

Finally, the results of both paths are compared.

The Verilog-to-E Translator



Use of the **E** Language

We have developed a formalized HDL in support of industrial design.

- Deeply embedded **E** language in ACL2 logic.
- Language descriptions are represented as Lisp constants.
- ACL2 theorem-proving system used to verify **E** descriptions.

The **E** language is formal.

- Syntax of **E** language is recognized by ACL2 predicate.
- Semantics given by interpreter.
    - Multiple evaluators defined: BDD, four-valued BDD, AIG, four-valued AIG, dependency, and delay.
    - Symbolic simulation for all modes (except delay).

The **E** Language is in everyday industrial use at Centaur.

E-Language Features

The **E** language is deeply embedded in ACL2, and it is:

- hierarchical, and
- occurrence-oriented.

We use the **E** language much like a database; it includes:

- HDL descriptions
- Hierarchical state representation
- Signal sense and direction
- Clock discipline
- Properties
- Annotations

E-language has multiple symbolic simulators

- BDD and AIG (both two- and four-valued) simulators
- Symbolic information-flow simulator
- Delay estimator

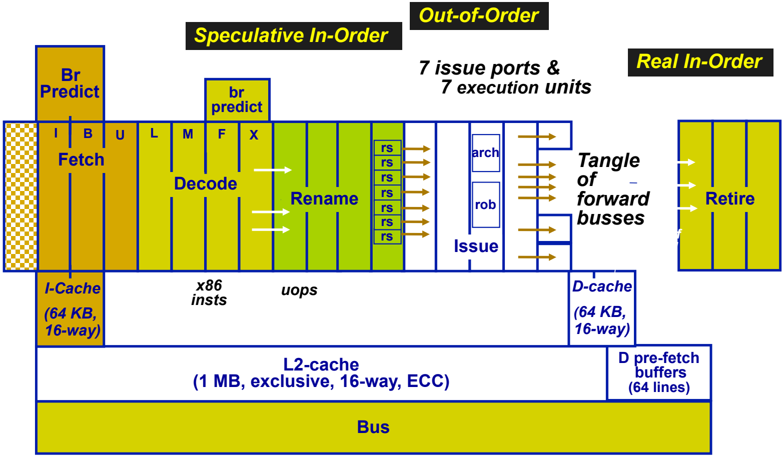
E-Language Example

```
(defm *simple-ff-latch*
  `(:i (clk a)
    :o (o)
    :s (l- l+)
    :c (clk)
    :cd ((t) (nil))
   :occs
    ((:u l+ :o (n n~) :op ,*latch+*  :i (clk a))
     (:u o1 :o (clk~) :op ,*not1*    :i (clk))
     (:u l- :o (o o~) :op ,*latch+*  :i (clk~ n)))))
```

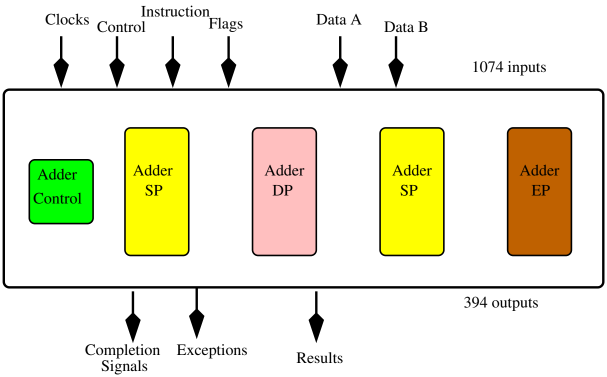
Simple two-latch, flip-flop

- Interface: **:i**, **:o**, and **:s** fields.
- Clock: **:c** and **:cd** fields.
- Occurrences are in a list, but treated as a set
- Multi-phase and gated clocking supported (and used by Centaur)

Centaur CN Pipeline

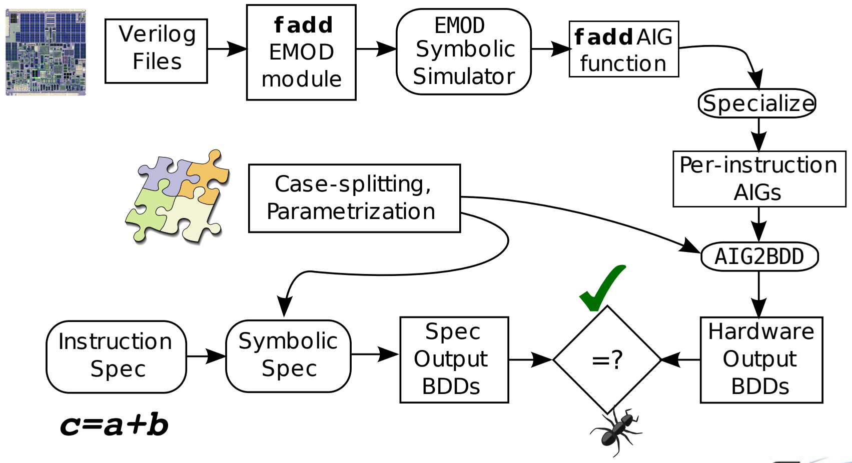


Centaur Nano^{TM} Media Unit – FADD



- 33,700 line Verilog description of 680 modules
- Modules represent 432,322 transistors
- Unit has 374 outputs and 1074 inputs (26 clocks)
- Implements over 100 media instructions
- Two-cycle-latency for floating-point additions/subtractions

The Centaur Media-Unit, Verification Tool Flow



Symbolic Simulation of the Media Unit

Using the **E**-language model, we perform a four-valued, AIG-based symbolic simulation of entire design for eight half-cycles.

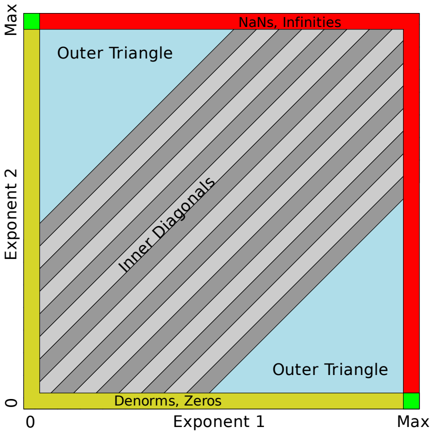
- AIGs specialized for the instruction under investigation
- AIGs are converted to BDDs
  - For some instructions, a property may be too big to verify directly, so case splitting employed
  - For each case, BDD approximated until exact
  - For each case, compared to symbolic simulation of specification
- Cases are shown to be exhaustive

The Centaur Media-Unit, Case-Splitting Approach

Floating-point add/subtract unit is too big to verify all at once.

- Case split by exponent differences
- Separately, account for special cases (e.g., NaNs, Infinity)
- For each case, generate symbolic inputs that cover the specified set of inputs
  - BDDs are parametrized
  - Approach used for all FP sizes

Used ACL2 to orchestrate proofs and to ensure entire input space covered and verified.



Centaur Nano Media-Unit, Verification

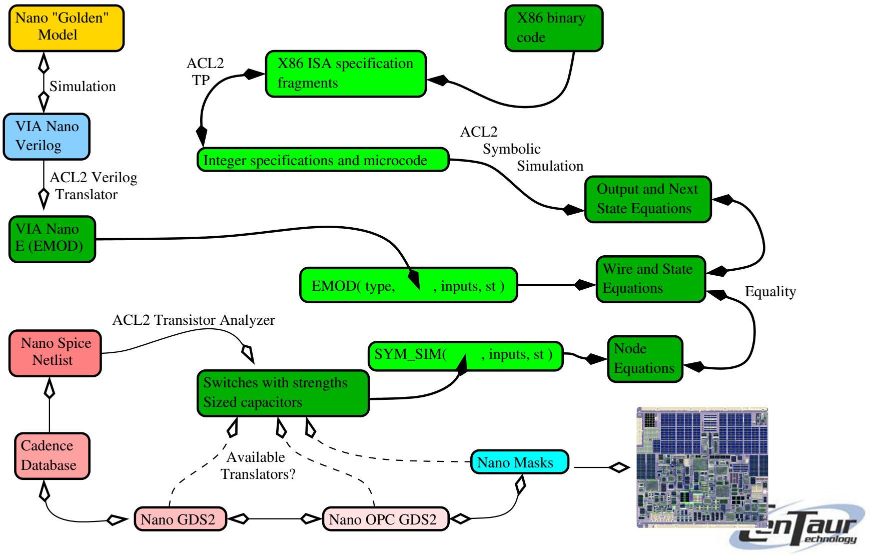
We attempted to verify single, double, and extended precision addition/subtraction operations.

- Single precision (32-bit) results and flags OK.
- Double precision (64-bit) results and flags OK.
- Extended precision (80-bit) results had an error.
  - Exactly one pair of numbers returned an incorrect answer
  - Sort of like a *perfect storm*; a 64-bit cancellation
  - Answer returned was twice as big as it should have been.

A fix was developed, and this bug was eliminated. We checked the correctness of the new design – it took less than an hour.

Robert Krug proved that our Boolean-based adder/subtracter specification is correct.

Redux: The Centaur Verification Tool Relationships



Verification of Full-Custom Circuitry

The Nano design is largely custom – meaning designers implement transistor-level circuits to satisfy Verilog specifications.

- Equivalence checking used to validate transistor-level circuits.
- Not all modules can be checked by equivalence checking
    - Module sub-divided
    - Individual submodules checked
    - Composition of submodules verified with ACL2

When automatic verification not possible, Verilog is further partitioned so as to permit automatic equivalence check.

Capability useful because vendor tools do not have adequate capacity.

Conclusion

ACL2 is in everyday commercial use at Centaur Technology.

- Each night, entire design is translated
  - 570,000 lines of Verilog translated to **E**
  - Unable to translate some modules – working to finish translation
- New ACL2 containing all **E**-based modules is built each day.
  - Entire translation and build time about 15 minutes
  - Human verifiers get newest design version each morning
- Each night we recheck our proofs on the new model

Extended ACL2:

- by deeply embedding the **E** HDL, transistor-level HDL,
- with AIG and BDD algorithms, which we mechanically verified, and
- by providing generalized symbolic simulation of all ACL2 functions,

It is possible to use a theorem prover to support
an industrial hardware verification flow.