Centaur Verification Approach

Jared Davis, Warren Hunt, Jr., Anna Slobodova, Sol Swords

August, 2011

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.

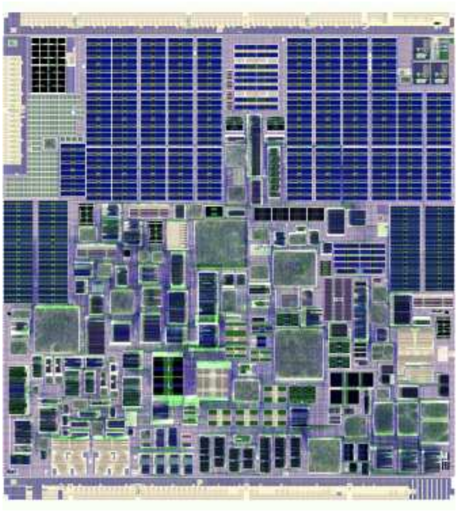
VIA Isaiah ^{TM} – X86-64 Microprocessor

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

- 64-bit (Intel EMT64-compatible) architecture
- Latest SSEx instructions
- Complex micro-architecture for performance
- Up to three instructions can be issues each cycle
- Lots of microcode
- Low cost, small size, low power, AND high performance
- 64-bit EA, 48-bit Virtual Address, 40-bit Physical Address

Requires full custom design
Targeted at low-power, low-cost products:
netbooks, low-power workstations, and embedded designs.
Used by Dell, Lenovo, OLPC, HP, and others...

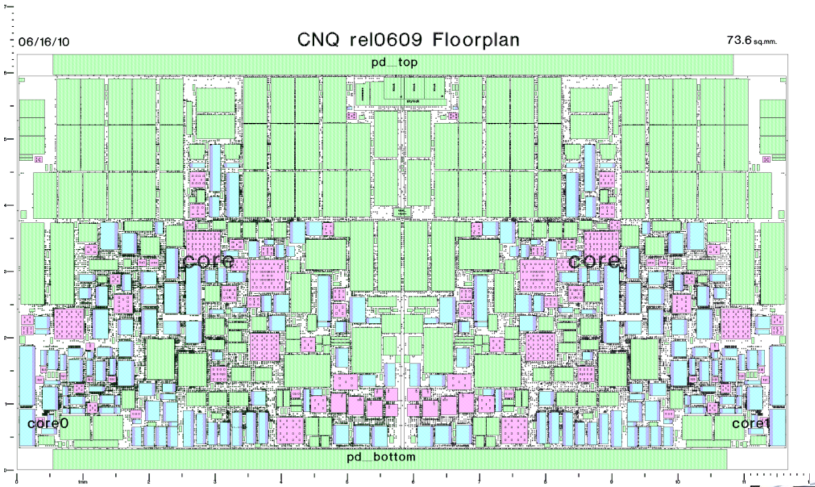
VIA Nano^{TM} Microprocessor



Contemporary Example

- Full X86-64 compatible two-core design
- 40nm technology, 97.6 million transistors per core (195.7)
- AES, DES, SHA, and random-number generator hardware
- Built-in security processor
- Runs 40 operating systems, four VMs

The Dual-Processor VIA Processor



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...

Core Technology: ACL2

- 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
- Efficient execution – models are often validated by co-simulation
- In use commercially by AMD, Centaur, and Rockwell-Collins
- Critical feature is the overall capacity of the ACL2 system

The ACL2 Theorem Prover

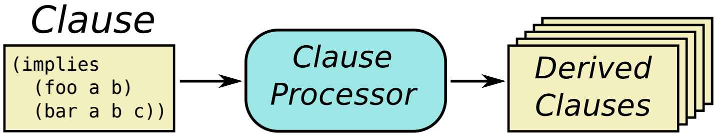
Associated with the ACL2 Logic, is the ACL2 theorem prover.

- Rewriter-based theorem-prover
- Simplifier include:
  - Clausification
  - Simplification
  - Linear arithmetic solver
  - Term type analysis
- Destructor elimination, generalization, induction
- Proof checker
  - Allows a manual proof, with each step checked
  - If successful, the proof developed can be re-used
  - Good to use, when it's not obvious why something fails.
- Symbolic simulation based on BDDs and AIGs
- Clause processors
  - Extensions to the ACL2 proving process
  - Clause processors can be verified, and then become part of ACL2
  - Symbolic simulation proof method included as a clause processor

What is a clause processor?

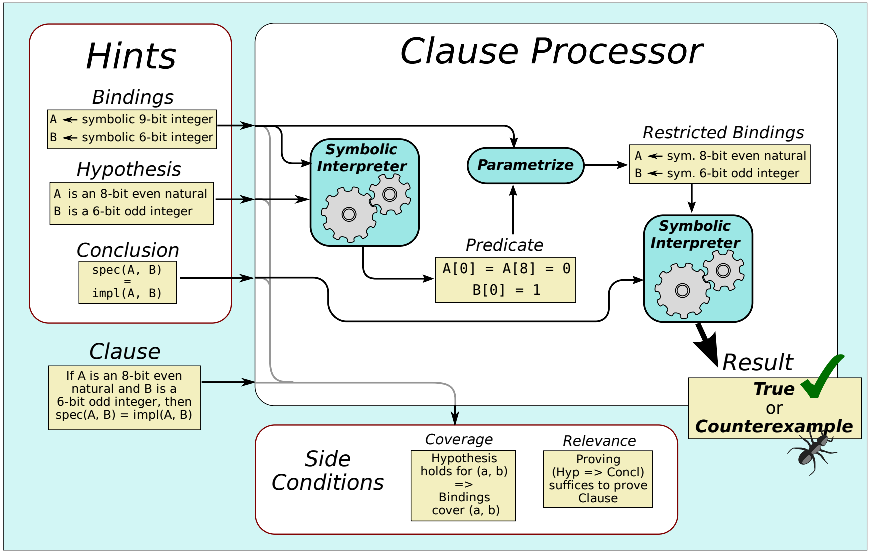
From ACL2 documentation: “A simplifier at the level of goals, where a goal is represented as a clause.”

- User function that takes goal clause and produces a list of clauses
- Soundness contract: proving that new clauses suffices to prove goal
- May be verified or not (requires trust tag)

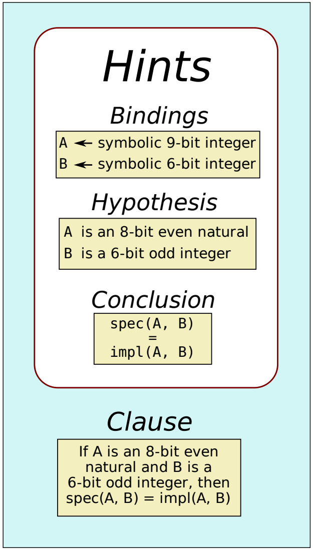


The GL system is a symbolic simulator for nearly all of the ACL2 logic by encoding data (objects) as BDDs or AIGs and symbolic versions of all primitive and user-defined functions

GL Clause Processor Flow

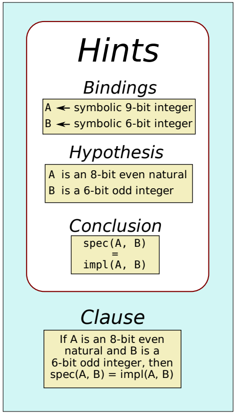

Hints
Bindings
A ← symbolic 9-bit integer
B ← symbolic 6-bit integer
Hypothesis
A is an 8-bit even natural
B is a 6-bit odd integer
Conclusion
spec(A, B) = impl(A, B)
Clause
If A is an 8-bit even natural and B is a 6-bit odd integer, then spec(A, B) = impl(A, B)
Clause Processor
Symbolic Interpreter
Parametrize
Restricted Bindings
A ← sym. 8-bit even natural
B ← sym. 6-bit odd integer
Predicate
A[0] = A[8] = 0
B[0] = 1
Symbolic Interpreter
Result
True or Counterexample
Side Conditions
Coverage
Hypothesis holds for (a, b) => Bindings cover (a, b)
Relevance
Proving (Hyp => Concl) suffices to prove Clause

GL Clause Processor: Inputs

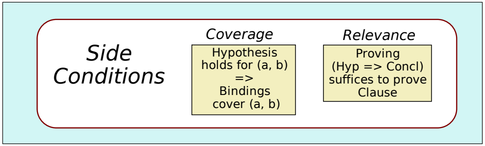

Hints
Bindings
A ← symbolic 9-bit integer
B ← symbolic 6-bit integer
Hypothesis
A is an 8-bit even natural
B is a 6-bit odd integer
Conclusion
spec(A, B) = impl(A, B)
Clause
If A is an 8-bit even natural and B is a 6-bit odd integer, then spec(A, B) = impl(A, B)

- Clause: the goal to be proved
- Hypothesis, conclusion, bindings: hints to the clause processor
- Bindings associate a symbolic object to each free variable in the clause
- Hypothesis gives "type"/"shape" constraints on variables
- Conclusion may further restrict variables (may itself be an `IMPLIES` term).

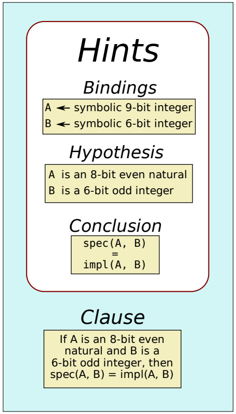
GL Clause Processor: Side Conditions


Hints
Bindings
A ← symbolic 9-bit integer
B ← symbolic 6-bit integer
Hypothesis
A is an 8-bit even natural
B is a 6-bit odd integer
Conclusion
spec(A, B) = impl(A, B)
Clause
If A is an 8-bit even natural and B is a 6-bit odd integer, then spec(A, B) = impl(A, B)

- Coverage:
  - Symbolic simulation (if successful) proves: *The conclusion holds of input vector x if x is a possible value of the symbolic inputs used in the simulation.*
  - To relate this to the hypothesis, must show: *If input vector x satisfies the hypothesis, then it is a possible value of the symbolic inputs.*


Side Conditions
Coverage
Hypothesis holds for (a, b) => Bindings cover (a, b)
Relevance
Proving (Hyp => Concl) suffices to prove Clause

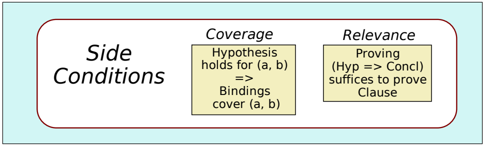
GL Clause Processor: Side Conditions


Hints
Bindings
A ← symbolic 9-bit integer
B ← symbolic 6-bit integer
Hypothesis
A is an 8-bit even natural
B is a 6-bit odd integer
Conclusion
spec(A, B) = impl(A, B)
Clause
If A is an 8-bit even natural and B is a 6-bit odd integer, then spec(A, B) = impl(A, B)

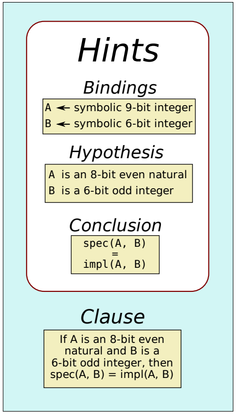
- Relevance:
  - Clause, hypothesis, conclusion are independent clause processor inputs
  - Symbolic simulation (with coverage) effectively proves

$$hypothesis \Rightarrow conclusion$$

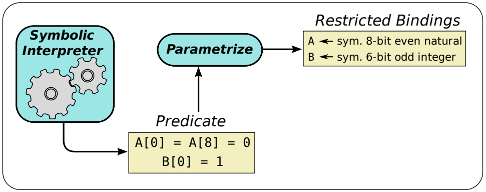
  - Therefore, prove that this implies the clause and we're done.
  - Typically trivial by construction.


Side Conditions
Coverage
Hypothesis holds for (a, b) => Bindings cover (a, b)
Relevance
Proving (Hyp => Concl) suffices to prove Clause

GL Clause Processor: Parametrization

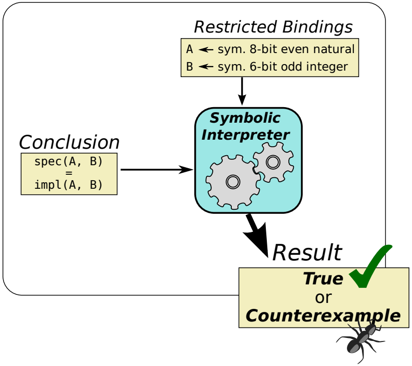

Hints
Bindings
A ← symbolic 9-bit integer
B ← symbolic 6-bit integer
Hypothesis
A is an 8-bit even natural
B is a 6-bit odd integer
Conclusion
spec(A, B) = impl(A, B)
Clause
If A is an 8-bit even natural and B is a 6-bit odd integer, then spec(A, B) = impl(A, B)

- Symbolic bindings may cover more than is accepted by the hypothesis - often better symbolic simulation performance is achievable if inputs cover less
- Symbolically simulating the hypothesis on the inputs yields a symbolic predicate
- Parametrization by that predicate yields new symbolic objects with coverage restricted to the space recognized by the hypothesis.

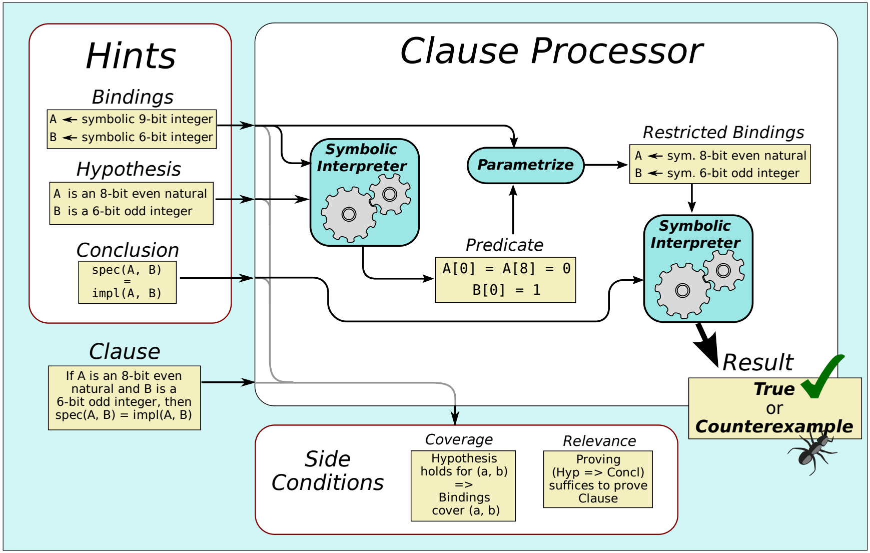


GL Clause Processor: Simulation

- Symbolically execute the conclusion to determine whether it holds on the space represented by the restricted bindings
- Result: often `T` or a set of counterexamples
- May fail or produce an ambiguous result (stack depth overrun, unimplemented primitive)



GL Clause Processor Flow: Recap


Hints
Bindings
A ← symbolic 9-bit integer
B ← symbolic 6-bit integer
Hypothesis
A is an 8-bit even natural
B is a 6-bit odd integer
Conclusion
spec(A, B) = impl(A, B)
Clause
If A is an 8-bit even natural and B is a 6-bit odd integer, then spec(A, B) = impl(A, B)
Clause Processor
Symbolic Interpreter
Parametrize
Restricted Bindings
A ← sym. 8-bit even natural
B ← sym. 6-bit odd integer
Predicate
A[0] = A[8] = 0
B[0] = 1
Symbolic Interpreter
Result
True
or
Counterexample
Side Conditions
Coverage
Hypothesis holds for (a, b) => Bindings cover (a, b)
Relevance
Proving (Hyp => Concl) suffices to prove Clause

Symbolic Simulation in ACL2

Thus, 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 (almost all of) the ACL2 logic.
We use both BDDs and AIGs to support our symbolic-simulation capability; we have verified our BDD package and we verify the result of our AIG-based SAT checker.

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))))))
```

Aside: most ACL2 functions can be memoized.

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

Population Count, by S. Anderson: Bit Twiddling Hacks

Let's consider a *simple* problem; find the population of 1's in an integer.

```
int popcount_bits ( int v ) {
  v = v - ((v >> 1) & 0x55555555);
  v = (v & 0x33333333) + ((v >> 2) & 0x33333333);
  v = ((v + (v >> 4) & 0xF0F0F0F) * 0x1010101) >> 24;
  return( v ); }
```

Question: is this code correct?

```
int popcount ( unsigned int v ) {
  return (( v == 0 ) ? 0 : (v & 1) + popcount( v / 2 )); }
```

By exhaustive simulation, we can attempt to validate our program with respect to another program.

```
if ( popcount_bits( i ) != popcount( i ) ) {
  ...  return 1;  ...}
```

Population Count Example, continued

Let's model this C-program in Lisp (ACL2).   System DEMO!

```
(defmacro &   (x y) `(logand ,x ,y))
(defmacro >>  (x y) `(ash ,x (- ,y)))
(defmacro 32* (x y) `(mod (* ,x ,y) *2^32*))

(defun fast-logcount-32 (v)
 (declare (xargs :guard (natp v)))
 (let*
  ((v (- v  (& (>> v 1)   #x55555555)))
   ;v = v - (  (v >> 1) & 0x55555555);

   (v (+ (& v #x33333333)   (& (>> v 2)   #x33333333)))
   ;v =  (v & 0x33333333) + (  (v >> 2) & 0x33333333);

   (c (>> (32* (& (+ v   (>> v 4))  #xF0F0F0F)   #x1010101)    24))
   ;c =           ( (v + (v >> 4) & 0xF0F0F0F) * 0x1010101) >> 24;
   )
   c))
```

Population Count Example, continued

```
(def-gl-thm fast-logcount-32-correct
  :hyp (unsigned-byte-p 32 x)
  :concl (equal (fast-logcount-32 x)
                (logcount x))
  :g-bindings `((x ,(g-int 0 1 33))))
```

The proof completes in 0.09 seconds and results in the ACL2 theorem:

```
(defthm fast-logcount-32-correct
  (implies (unsigned-byte-p 32 x)
           (equal (fast-logcount-32 x)
                  (logcount x)))
  :hints ((gl-hint ...)))
```

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))))
```

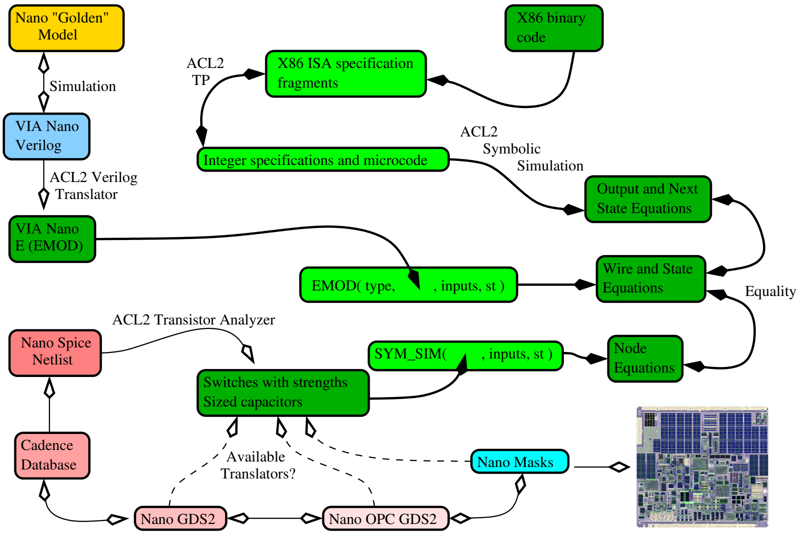
Example IF Tree and Verification by Symbolic Execution

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

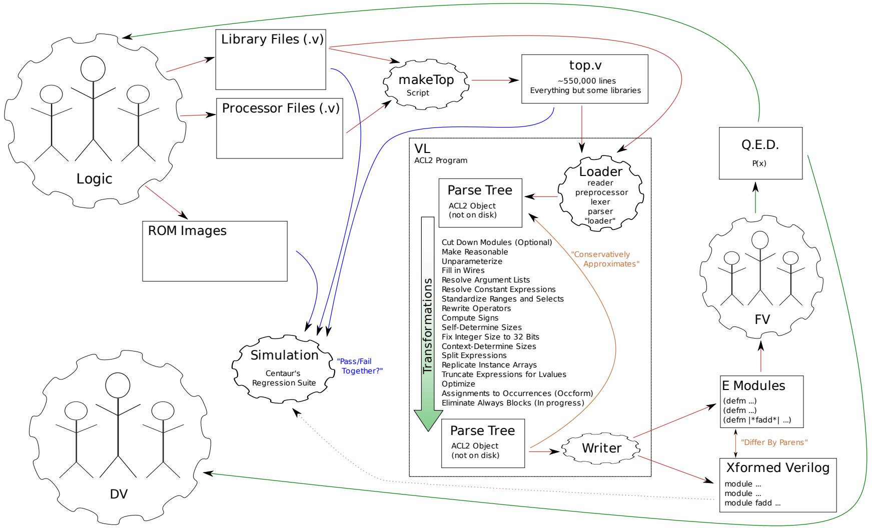
Our language of IF trees only contains one logical connective.

```
  (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))))
```

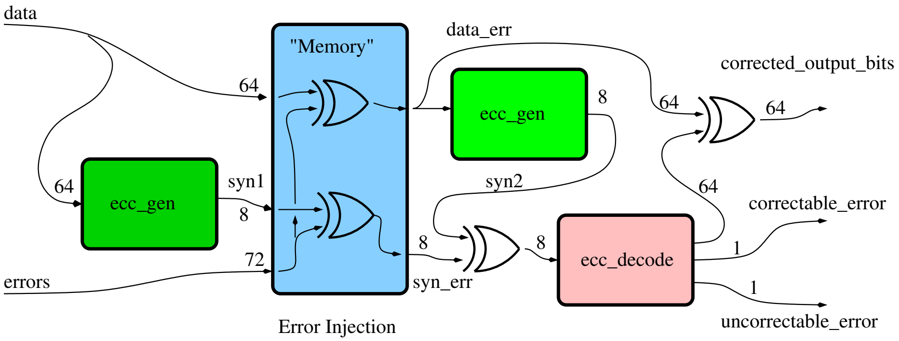
The Centaur Verification Tool Relationships



The Verilog-to-E Translator



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.

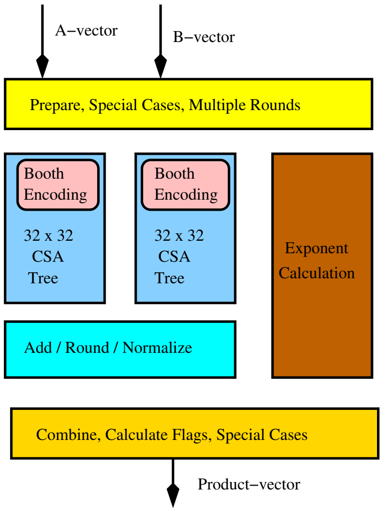
- Verilog is simplified into *single-assignment* form.
- Create environment suitable for media unit verification.
- We extract its *equation* by symbolic simulation.
- We specialize this equation to the instruction of interest.
- We then, as appropriate, convert this equation into BDDs.

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 Centaur Nano Multiplier Units



Many multiplier configurations

- two copies of diagram
- signed and unsigned: 8x8, 16x16, 32x32, 64x64
- packed-integer multiply: 16x16, 32x32
- packed-integer multiply-and-add
- floating-point: X87 and SSE2 with single, double, and extended precisions
- floating-point multiply-add

Nano Multiplier Unit Characteristics

Accommodates execution of most micro-operations that involve multiplication:

- 8-, 16-, and 32-bit packed signed, unsigned integer multiplications and multiply-add
- 64-bit integer multiplication
- Different flavors of x87 floating-point multiplication
- SSE2 single and double precision packed floating-point multiplication
- floating-point multiply add

Performance and latency:

- Fixed latency depends on micro-operation
- Back-to-back pipelined instructions
- Special operands including denormal numbers

Centaur Processor Multiplier Unit Proofs

Nano multiplier proofs broken into pieces to accommodate capabilities of our symbolic simulator — multiple passes complicate decomposition!

- 2 proofs for 8-, 16- and 32-bit multiplication:
  - sources to partial products
  - partial products to write-back
- 3 proofs for 64-bit integer multiplication:
  - sources to partial products
  - 2 proofs for partial products to mantissa products
- 3 proofs for SSE2 single precision FP packed multiplication:
  - sources to partial products
  - partial products to mantissa product
  - mantissa product to rounded result
- 4 proofs for x87 and SSE2 double precision FP packed multiplication:
  - sources to partial products
  - 2 proofs for partial products to mantissa product
  - mantissa product to rounded result

- Partial proofs using GL clause processor
- Composition of theorems obtained in previous step using ACL2

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
  - 640,000 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.