Verifying VIA Nano Microprocessor Components

Warren A. Hunt, Jr.
Centaur Technology, Inc.
7600-C North Capital of Texas Hwy, Suite 300
Austin, Texas 78731-1180
Email: hunt@centtech.com

***Abstract*—We verify parts of the VIA Nano X86-compatible microprocessor design using the ACL2 theorem-proving system. We translate Nano RTL Verilog into the `EMOD` hardware description language. We specify properties of the Nano in the ACL2 logic and use a combination of theorem-proving and automated techniques to verify the correctness of Nano design elements.**

I. Introduction

We have specified and verified parts of VIA's X86-compatible Nano microprocessor using the ACL2 theorem-proving system. The VIA Nano microprocessor is a full X86-64 design, including VMX, AES, DES, and SHA instructions. The current Nano is implemented in a 40-nanometer process with around 100 million transistors. The Nano design contains a security co-processor; it runs over 40 different operating system (such as Windows, MacOS, Linux, FreeBSD); and it supports four different virtual-machine implementations. The RTL Nano specification is written in Verilog, which we translate into our formalized `EMOD` hardware description language (HDL). We use this `EMOD` representation both as a specification for transistor-level circuit elements and as an implementation for more abstract properties.

The design for the Nano is represented with 570,000 lines of Verilog. This is a hierarchical description that includes specifications for all Nano circuit elements, and it can be simulated using a Verilog simulator. To verify parts of the Nano design, we first translate modules of interest into the `EMOD` formal hardware description language, which is embedded it within the ACL2 logic. We use the ACL2 logic to specify the operation of Nano hardware elements. Finally, we use the ACL2 theorem-proving system to verify the correctness of Nano design elements.

Our verification efforts have been focused on the media and floating-point units. The Nano media unit can add/subtract four pairs of floating-point numbers every clock cycle with a two-cycle latency. Depending on the size of the operands, the Nano multiplier can multiply one, two, or four pairs of operands every clock cycle with a three- or four-cycle latency. The Nano divider is implemented with a special 4-bit divider unit augmented with a microcode program.

We have verified hardware the add, subtract, multiply, divide (microcode only), compare, convert, logical, shuffle, blend, insert, extract, and min-max instructions [8]. To verify Nano components, we symbolically simulate design fragments and compare the results to specifications written in ACL2. Separately, we also verify that our ACL2 specifications implement various floating-point operations.

In this paper, we describe some of the models used by VIA to implement the Nano. We have formalized subsets of several models, and we are working to formalize the entire Nano design. The Nano is continuously updated and extended; for example, this last year the Nano was extended with 256-bit SSE instructions. The Nano will soon be offered as a multi-core, which required an internal rearrangement of many design elements. As the design is altered, we re-run our evolving set of formal verification scripts to ensure that the latest design continues to satisfy the properties we have been able to formally specify and mechanically verify. Thus, we must be able to very quickly translate existing design representations into a form suitable for our tool suite.

II. The Centaur FV Toolflow

Nano circuits are initially represented in Verilog. We translate the Nano Verilog model into our `EMOD` hardware description language; and we analyze the result of such translations by comparing them to specification functions. The relationships between these various models are shown in Figure 1.

Starting in the upper-left-hand-corner of the diagram is the `Nano ''Golden'' Model`; this is a C-language program that is used as a specification for the `VIA Nano Verilog`. The operation of the `VIA Nano Verilog` is compared to the specification using co-simulation; both models are simulated, and after each (clock cycle) step, register and memory values are compared. This procedure is the primary pre-silicon verification approach for ensuring that the Nano satisfies its specification. Once functional silicon Nano processors are available, this same kind of co-simulation is done but the actual Nano microprocessor is used in place of the Nano Verilog; results are still compared to the `Nano ''Golden'' Model`. Of course, once working microprocessors are available, they are also installed in computing systems and subjected to a wide variety of tests; the results of these tests are compared to known-good results.

We use formal verification to augment the already extensive simulation being performed. Formal verification has found errors not detected during testing and in commercial usage that are generally very subtle. Of course, if such bugs were easy to find, they would have been uncovered by simulation. Our formal verification process begins by translating the Nano

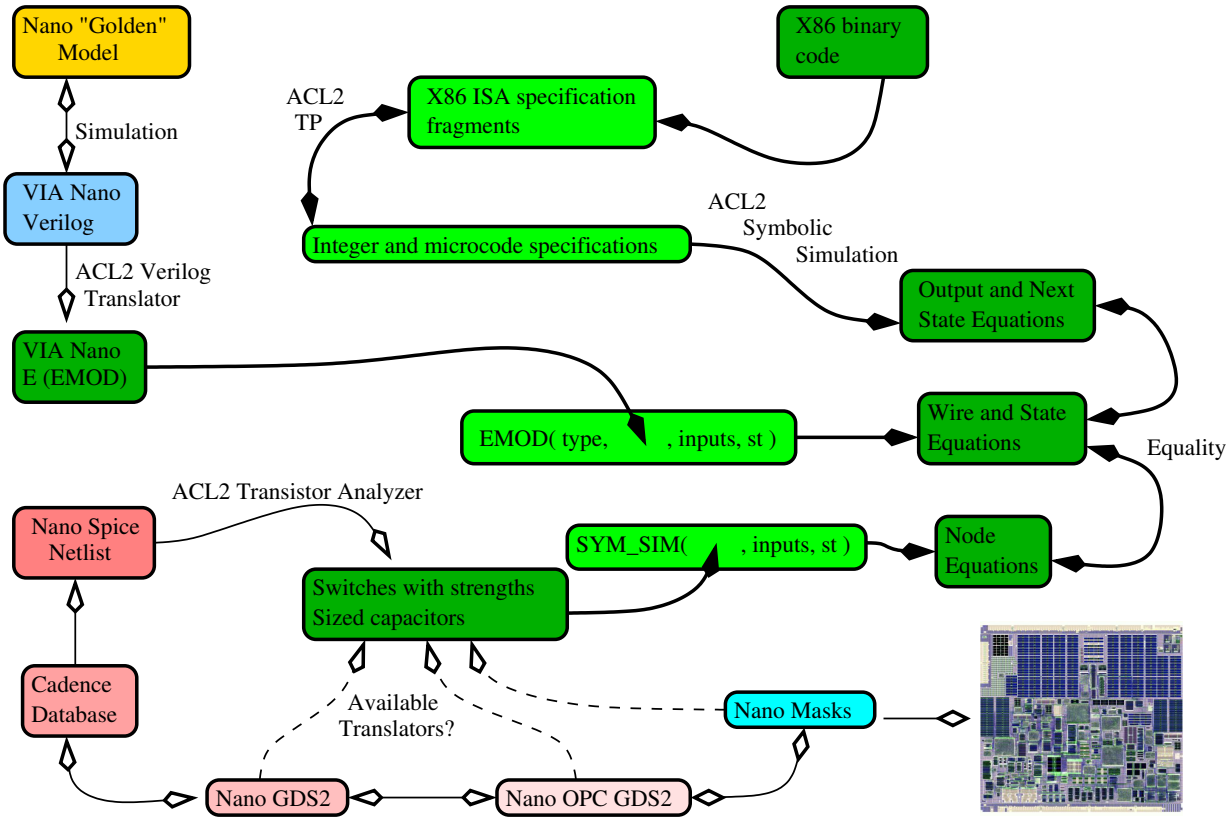

Fig. 1. Verification relationships between the models

Verilog into the `EMOD` HDL, and then symbolically simulating it to get ``Wire and State Equations`. For more complex specifications, we generally write ACL2 code that is designed to mimic the behavior of the source Verilog, and then compare the results produced. In some cases, we compare our `Integer and microcode specifications` to even more abstract specifications, such as when we verified the Nano media-unit [8] instructions.

In addition to the kind of Verilog verification so far discussed, we also verify transistor-level circuit implementations. A large part of the Nano implementation is custom-designed, transistor-level circuits. In fact, almost all of the Nano design is full custom, except for a number of auto-place-and-route blocks that primarily implement control logic. As show in Figure 1, we verify transistor-level models by translating their Spice-level circuit representations into a `Switches with strengths -- Sized capacitors` form. Using `SYM_SIM`, we symbolically simulate the resulting circuit models and compare the resulting `Node Equations` values with the expected `Wire and State Equations` results.

At a high level, our current verification efforts could be described as co-simulation with symbolic test vectors. Boolean data is represented with Boolean variables instead of with specific Boolean values. In some cases, such as exist in the execution cluster, the specification of correctness is relatively straight forward, although voluminous and detailed. In other cases, such as with the bus interface, the specification is much more complex and subtle because of the very large number interactions with other units. Before we attempt to explain our use of `EMOD`, we provide a simple embedding example.

III. A Simple Embedded Language

We now illustrate the embedding of a very simple language within the ACL2 logic. This language, based on IF trees, is defined by two functions: a recognizer (the permitted syntax) for IF expressions and an evaluator (the semantics) for IF expressions.

Here are some syntactically, well-formed examples in language. Note that these expressions are “quoted”; that is, they are ACL2 (and Lisp) data constants.

```
'(IF c a b)        '(IF 1 2 3)

'(IF r (IF c T NIL) q)
```

We can check whether these forms are indeed acceptable using our syntactic recognizer function `if-termp`, which takes a single argument and recognizes whether this argument is a valid IF-expression. If `term` is an `atom` then it must be recognized by the `eqlablep` predicate, which recognizes atoms that are numbers, symbols, or characters. Otherwise, this predicate requires an object of the form `(IF a b c)`, where the argument recursively recognized by `if-termp`.

```
(defun if-termp (term)
   (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))))))
```

The function `if-evl` evaluates the `term` argument, recognized by `if-termp`, using assignments of values to variables as given in `alist`.

```
(defun if-evl (term alist)
   (if (atom term)
       (cdr (assoc term alist))
     (if (if-evl (cadr term) alist)
         (if-evl (caddr term) alist)
       (if-evl (cadddr term) alist))))
```

For instance, by binding the variables `1`, `2`, and `3` to themselves, we get:

```
(IF-EVL '(IF 1 2 3)
        '((1 . 1) (2 . 2) (3 . 3)))
  ==>
2
```

Given these two functions, we have defined the syntax and semantics of our IF-expression language. This is a very simple language embedding; we use the same technique to embed our hardware description language with ACL2.

We can now prove theorems about descriptions involving formulas our IF-expression language. For instance, we can prove:

```
(let ((if-expr  '(IF A B C))
      (bindings (list (cons 'A a)
                      (cons 'B b)
                      (cons 'C c))))
(implies
 (and (if-termp if-expr)
      (eqlable-alistp bindings))
 (equal (if-evl if-expr bindings)
        (if a b c))))
```

This shows for any `a`, `b`, and `c`, that the evaluation of the expression `'(IF A B C)` with the bindings shown is the same as `(if a b c)`.

IV. Our Verification Approach

We verify Verilog circuit descriptions by translating them into a HDL-form that ACL2 can process. We then use our ACL2-based definition of our HDL to symbolically simulate these translations, and we compare the simulation results to ACL2 specifications.

A. Our Verilog-to-EMOD Translator

We have written a translator that converts a Verilog design description into the `EMOD` language. This translation is meant to be principally a syntactic transformation; however, because of the complexity of Verilog it involves a number of semantic transformations.

To implement the translation of Verilog into `EMOD`, we adopt a program-transformation-like [17] style: to begin with, the entire parse tree for the Verilog sources is constructed; we then apply a number of rewriting passes to the tree which result in simpler Verilog versions of each module. The final conversion into `EMOD` is really almost incidental, with the resulting `EMOD` modules differing from our most-simplified Verilog modules only in minor syntactic ways. Since each rewriting pass produces well-formed Verilog modules, we can simulate the original and simplified Verilog modules against each other, either at the end of the simplification process or at any intermediate point.

- We instantiate modules to eliminate parameters introducing new modules for each instantiation size.
- Wires and registers in Verilog can have varying widths, and we resolve all such expressions to constants.
- We reduce the variety of operators we need to deal with by simply rewriting some operators away. In particular, we perform rewrites such as:

  `a && b` → `(|a) & (|b)`,
  `a != b` → `|(a ^ b)`, and
  `a < b` → `~(a >= b)`.

  This process eliminates all logical operators, equality comparisons, negated reduction operators, and standardizes all inequality comparisons.
- We annotate every expression with its type (sign) and width. The rules for determining widths are subtle, and if they are not properly implemented then, signals might be inappropriately kept or dropped.
- We introduce explicit wires to hold the intermediate values in expressions.
- Verilog allows for implicit truncations in assignment statements; for instance, one can assign the result of a five-bit addition `a + b` to a three-bit bus (collection of wires), `w`. We make these truncations explicit by introducing a new wire for the intermediate result. We replace expressions like `a + b` with basic module instances.

We have left out many minor transformations like naming any unnamed instances, eliminating supply wires, and minor optimizations. Together, our simplifications leave us with a new list of modules where only simple gate and module instances are used. From this we can produce either `EMOD` or simplified Verilog. The simplified Verilog can be co-simulated with the original Verilog as a translation sanity check.

B. The EMOD HDL

Our EMOD-language analysis approach permits the hierarchical verification of cooperating finite-state machines. We have been investigating such languages for over 20 years. Our initial attempt was the HEVAL language [2]; this combinational-only language was embedded in the NQTHM logic [4]. This led us to the development of the DUAL-EVAL HDL which was used as the target for the FM9001 microprocessor verification [3]. As we were the designers of the FM9001, we actually created and verified a DUAL-EVAL description of the FM9001 before translating it into LSI Logic's NDL language for implementation [12].

The DE HDL [6] was our first HDL embedded into the ACL2 [11] logic. Later, we extended DE by adding parameters and busses; we called this the DE2 [7] language. To validate a data-network circuit, the logic was represented in DE2 and then this design fragment was verified using ACL2 [14]. Our latest effort is the EMOD HDL, which is used as a target for Nano circuits. Other groups [5] have pursued a similar approach using HOL [16] to provide the formal semantics. Intel has done extensive formal verification of the Intel®Core i7™ processor architecture [10]. AMD is also using formal verification to aid the verification of their processors [15].

The semantics of EMOD are specified by a deeply-embedded interpreter written in the ACL2 logic; this interpreter permits multiple signal evaluation styles: BDDs, AIGs, definedness, dependency, and delay. We believe EMOD is the first formally-specified language to support multiple interpretations of HDL descriptions within a single system, and EMOD is the first formally-defined HDL to be used in a commercial design flow.

Although EMOD language circuit descriptions have the form of a HDL, its structure allows it to be accessed and updated much like a database. Annotations may be attached to every module definition and occurrence; such annotations include information such as signaling conventions, functional requirements, warnings, and clock disciplines. Thus, we eventually imagine that a post-silicon design engineer may interrogate an EMOD-language design with database-like queries to determine properties that were specified and proven by pre-silicon designers. And, a post-silicon engineer may exhaustively establish properties using the speed of fabricated circuits, and then add these properties to the evolving EMOD-based design (database).

In support of commercial design verification, we have defined edge-triggered and level-sensitive, state-holding primitives, and using these primitives, a user may define and verify multi-clock (derived from one master clock) circuits. Verification of gated-clock circuits is supported, indeed, required for the Nano design style. Verifying bi-directional, tri-state busses and pass-transistor circuits requires four-valued equations to be used, and since our transistor-level circuit analyzer targets our four-valued logic, mixed transistor-gate-level designs may also be verified.

C. Our Circuit Models

We formally verify fragments of the Nano by translating them from Verilog to our formal EMOD language, and then performing symbolic analysis. Instead of trying to write a formal semantics for Verilog, we choose to formally define a simpler language and then analyze the results of our translator, which is labeled ACL2 Verilog Translator in Figure 1. The EMOD language contains mechanisms that allow us to represent all of the interface and module names that appear in Verilog design representations, and we verify EMOD circuit representations using ACL2.

We verify EMOD circuit representations to more abstract specifications that we write in ACL2. As depicted in Figure 1, we write Integer and microcode specifications in ACL2, and then symbolically simulate these specifications [1]; this produces, either as AIGs or BDDs, results that we compare to the symbolic simulation of EMOD circuit representations. Sometimes, independently of the Nano design, we may write an even more abstract X86 ISA specification fragments, such as for the floating-point operations, and compare our Integer specifications to these higher-level specifications. For instance, we have such X86 ISA specification fragments for the basic floating-point operations; these specifications are independent from the Nano; they conform to the IEEE floating-point specifications [9].

In a large number of cases, there are custom implementations for various Nano circuits; these circuits are implemented at the transistor level. To verify such transistor-level circuit descriptions, we use our ACL2 Transistor Analyzer which converts a Spice-level circuit representation [13] into a model that has Switches with strengths and Sized capacitors. This kind of model can be symbolically simulated using the SYM_SIM circuit simulator, and we compare the results of such simulations to higher-level, symbolic simulations.

V. ECC Circuit Analysis

We present a memory error-detection-and-correction circuit (ECC) and its analysis. This circuit detects and corrects single-bit memory errors; it also detects double-bit memory errors. A descendant of this circuit is used in the VIA Nano, and we verified its operation using the procedures outlined above. As shown in Figure 2, the circuit is composed of two identical syndrome generators and an ECC element that drives 64 exclusive-OR gates. The "Memory" block is a model we developed to model the operation of the real memory; this block is modeled with 72 exclusive-OR gates, which allows, using the 72 error inputs, to model any number of inversion failures. We have three verification goals:

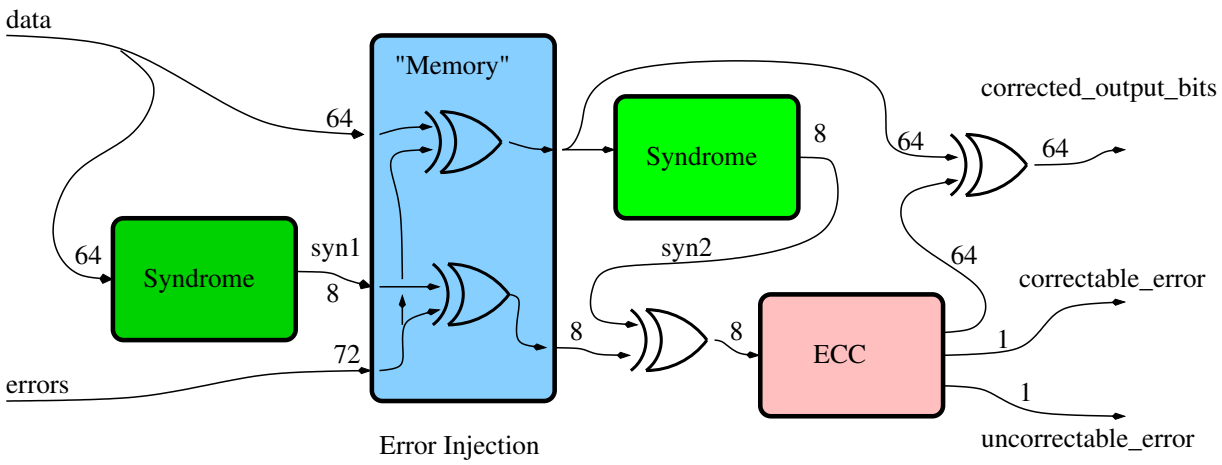

Fig. 2. Error-Correction-Circuitry Diagram

- when there are no memory errors, the output is correct, and no error is indicated;
- when there is one memory error, the output is correct, the `correctable_error` output bit is set, and the `uncorrectable_error` is not set; and
- when there are two memory errors, we only check that the `uncorrectable_error` is set.

We approach this verification considering all of the possible combinations of memory bits and errors that are possible:

- no memory errors: 2^{64} = 18446744073709551616;
- one error: $(2^{64} * 72)$ = 1328165573307087716352; and
- $(2^{64} * (72 * 71)/2)$ = 47149877852401613930496 for when there are two errors.

In the two-error case, the error positions are symmetric. We encode the possible errors explicitly by a one- or two-hot encoding on the `error` input vector; we will later see that our specification functions model these errors.

Below is the Verilog source we will attempt to validate. We constructed this model so we could check that modules `ecc_gen` and `ecc_decode` perform the intended operation. Of course, the operation of this circuit model also depends on the exclusive-OR operations (gates) that are part of the circuit. The exclusive-OR gates in the memory are just part of our model; these gates allow us to model "bit-flips" in the memory.

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

   input  [63:0]  data;     // Data inputs
   wire   [63:0]  data;
   input  [71:0]  errors;   // Error injection bits
   wire   [71:0]  errors;

   output [63:0]  corrected_output_bits; // Output
   wire   [63:0]  corrected_output_bits;
   output         correctable_error;     // Good?
   wire           correctable_error;
   output         uncorrectable_error;   // Bad
   wire           uncorrectable_error;

   wire [7:0]  syn1;      // from first ecc_gen
   wire [7:0]  syn2;      // from second ecc_gen
   wire [63:0] data_err; // Possibly flawed data
   wire [7:0]  syn_err;  // Memory syndrome bits
   wire [63:0] bit_to_correct; // correct outputs

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

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

   // Thus, using the "errors" input we can create
   // faults that could be considered memory errors.

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

   wire [7:0] syn_backwards_xor;
   // Compute syndrome
   assign syn_backwards_xor = syn_err ^ syn2;

   ecc_decode make_outs (bit_to_correct,
                         correctable_error,
                         uncorrectable_error,
                         syn_backwards_xor);

   // Finally, correct the output.
   assign corrected_output_bits
                  = bit_to_correct ^ data_err;
endmodule
```

We now present the ACL2 commands used to define and verify our example ECC circuit. Some details are omitted, but we attempt to supply sufficient detail so a reader can understand the process. After placing ourselves in the directory containing the Verilog above, we start our ACL2-based analysis system and execute the commands below. The `defmodules` command reads the Verilog source and converts it into the `EMOD` language. The `find-input` commands collect and group the input wire names. Similarly, the `find-output` commands collect the output wire names. We use these command because it allows rearrangement of the

module interface without it concerning our effort.

```
; Convert the Verilog ECC model and its inferior
; components into the EMOD language.

(defmodules *ecc* :start-files (list "ecc_model.v")
                  :search-path '("."))

; By name, collect the input data and error inputs.

(defconst *ecc_model/data*
          (find-input "data"   64  |*ecc_model*|))
(defconst *ecc_model/errors*
          (find-input "errors" 72  |*ecc_model*|))

; By name, collect the output and the correctable
; and uncorrectable error indications.

(defconst *ecc_model/corrected-output-bits*
  (find-output
   "corrected_output_bits" 64 |*ecc_model*|))
(defconst *ecc_model/correctable_error*
  (find-output
   "correctable_error" 1 |*ecc_model*|))
(defconst *ecc_model/uncorrectable_error*
  (find-output
   "uncorrectable_error" 1 |*ecc_model*|))
```

The two functions below allow us to form the inputs by name. With the function `create-input`, we construct an association list pairing names with their values, and then we generate an appropriate pattern. This frees us from being concerned about the position of the arguments in the `|*ecc_model*|` model. The next two functions below perform a similar function for the output; that is, we construct three outputs based on the output wire names.

```
(defun create-input (data errors)
  (b* ((alist
        (make-fast-alist
         (ap (pairlis$ *ecc_model/data*   data)
             (pairlis$ *ecc_model/errors* errors))))
       (pat (gsal (gpl :i |*ecc_model*|)
                  alist 'fail))
       (- (fast-alist-free alist)))
      pat))

(defun alist-extract (keys alist)
  (declare (xargs :guard t))
  (if (atom keys)
      nil
    (cons (cdr (hons-get (car keys) alist))
          (alist-extract (cdr keys) alist))))

(defun get-output (output)
  (b* ((alist (pal (gpl :o |*ecc_model*|)
                   output nil))
       (corrected_output_bits
        (alist-extract
         *ecc_model/corrected-output-bits*
         alist))
       (correctable_error
        (car (alist-extract
              *ecc_model/correctable_error*
              alist)))
       (uncorrectable_error
        (car (alist-extract
              *ecc_model/uncorrectable_error*
              alist))))
      (mv corrected_output_bits
          correctable_error
          uncorrectable_error)))
```

We next specify our error-correction circuit. We define the `q-not-nth` function that (symbolically) inverts a bit of `x` at position `n`. If `n` is larger than the length of the list `x`, no change is made. The next three functions specify the operation of our `|*ecc_model*|` when there are no memory errors, when one error is introduced, and when two errors are inserted.

```
(defun q-not-nth (n x)
  ;; Invert bit N of X.
  (if (atom x)
      nil
    (if (zp n)
        (cons (q-not (car x)) (cdr x))
      (cons (car x)
            (q-not-nth (1- n) (cdr x))))))

(defun no-problems ()
  ;; Check output correctness if no errors injected.
  (b* ((data   (qv-list 0 1 64))
       (errors (make-list 72 :initial-element nil))
       (inputs (create-input data errors))
       ((mv & o) (emod 'two |*ecc_model*|
                       inputs nil))
       ((mv corrected_output_bits
            correctable_error
            uncorrectable_error)
        (get-output o)))
      (and (equal corrected_output_bits data)
           (not   correctable_error)
           (not   uncorrectable_error))))

(defun one-bit-error-predicate (bad-bit)
  ;; Check output correctness if one error injected.
  (b* ((data     (qv-list 0 1 64))
       (err-bits (make-list 72
                            :initial-element nil))
       (errors   (q-not-nth bad-bit err-bits))
       (inputs   (create-input data errors))
       ((mv & o) (emod 'two |*ecc_model*|
                       inputs nil))
       ((mv corrected_output_bits
            correctable_error
            uncorrectable_error)
        (get-output o)))
      (and (equal corrected_output_bits data)
           (equal correctable_error (< bad-bit 64))
           (not   uncorrectable_error))))

(defun two-bit-error-predicate (x y)
  ;; For two-bit errors, we only check that the
  ;; uncorrectable error is signaled.
  (if (eql x y)
      ;; If only one error bit is injected.
      (one-bit-error-predicate x)
    (b* ((data     (qv-list 0 1 64))
         (err-bits (make-list
                    72 :initial-element nil))
         (errors   (q-not-nth
                    x (q-not-nth
                       y err-bits)))
         (inputs   (create-input data errors))
         ((mv & o) (emod 'two |*ecc_model*|
                         inputs nil))
         ((mv & & uncorrectable_error)
          (get-output o)))
        uncorrectable_error)))
```

Finally, we introduce functions that generate input suitable to check all one- and two-bit errors. Thus, these functions provide the top-level requirements for the ECC circuit.

```
(defun all-one-bit-errors (x)
  (and (or (one-bit-error-predicate x)
           (cw "one-bit-error ~x0~%" x))
       (if (zp x)
           t
         (all-one-bit-errors (1- x)))))

(defun all-two-bit-errors-help (x y)
  (and (or (two-bit-error-predicate x y)
           (cw "two-bit-error ~x0 ~x1~%" x y))
       (if (zp x)
           t
         (all-two-bit-errors-help (1- x) y))))

(defun all-two-bit-errors (y)
  (if (zp y)
      t
    (and (all-two-bit-errors-help (1- y) y)
         (all-two-bit-errors (1- y)))))

(defun all-zero-one-two-bit-errors (z)
  (and (or (no-problems)
           (cw "no-problems ~%"))
       (all-one-bit-errors z)
       (all-two-bit-errors z)))

(time$ (all-zero-one-two-bit-errors 71))
```

This is not the most efficient way to investigate all such errors, but it is straightforward. We could have introduced additional symbolic variables to indicate input-error positions, and then performed one symbolic computation. However, in spite of the fact that over 5000 symbolic executions of the EEC circuit are performed, it takes less than 30 seconds to consider all of the combinations. When we considered this problem, the ECC circuit designers wanted a quick answer, and this was a simple way to check their intent. But, we realized a few days later that our specification, and therefore, the circuit had an error – our `one-bit-error-predicate` only checks that a flawed data bit is detected, but it does not check if one of the redundant check bits (positions 64 to 71) is itself flawed.

```
(equal correctable_error (< bad-bit 64))
```

This was a problem of there being an error in both the circuit and the specification; subsequently, this error was fixed.

In specification for the ECC circuit we just described, we did not symbolically co-simulate a corresponding ACL2 specification; we directly specified what we expected as answers. Thus, as pictured in Figure 1, instead of comparing `Output and Next State Equations` to `Wire and State Equations`, we just inspected the latter. For our proofs about the Nano media unit [8], we wrote ACL2 specifications that we believed correctly specified it operation. We later verified that our media-unit specifications were valid by proving that they implemented our IEEE floating-point specification.

VI. Conclusion

We have verified parts of the VIA Nano Verilog design using the ACL2 theorem prover. Much of the verification "work" is done with symbolic simulation, and we use the ACL2 theorem prover both to verify high-level properties and to orchestrate the various verification techniques we use. All of our proofs are carried out with the ACL2 theorem prover, and the BDD and AIG algorithms we use have also been verified using the ACL2 theorem prover.

Beyond the straightforward mechanisms described here, we often use additional verification techniques. The circuit descriptions we verify include state-holding elements, and we must either initialize such state-holding elements with suitable initial values or perform additional symbolic simulation that forces such storage elements into suitable (symbolic) states. We usually simulate a circuit for multiple steps, as it requires several clock cycles for such such circuits to complete their operations. With sequential circuits, it is necessary to specify the clocking discipline; that is, when and in what phases the clocks arrive is critical to circuit operation. For instance, for the verification work on the Nano media unit, we must correctly orchestrate 26 clock inputs. We use input parametrization, with appropriate choice of input space partitioning, to allow verifications where otherwise we might fail to create desired output equations – generally, we construct AIGs and then, through an iterative BDD construction procedure, we compare the results produced to their specifications. We have developed a general procedure for symbolically simulating any specification written in ACL2. Using this procedure, we symbolically evaluate ACL2-based specifications and compared to their results to an `EMOD` simulation. Separately, we prove desired correctness properties about such ACL2 specifications.

Most of our overall effort has been directed to verifying execution-cluster properties, much in the same way that Intel has done with the Nehalem family [10]. AMD has also been using ACL2 to verify elements from their Athlon processors [15]. We have begun to explore the use of our formal verification tools for other parts of the Nano design; for instance, we have recently been investigating the instruction decoder because a problem manifested itself that was not discovered by other tools; this was due to a lack of capacity, as the state machines being compared were too large for available commercial tools.

Our application of one formal system, specifically ACL2, may be broader than any single formal verification tool in use by other projects. We use ACL2 to read and translate the Verilog and to model the behavior of Nano circuits at the transistor level; this part of our verification flow has become more important as we experience the limitations of commercially available tools. We specify high-level operations, such as floating-point operations, in a manner that is independent of the specific operation of the Nano; these specifications are general and would likely be valid for many microprocessors. We are expanding the use of formal verification on future Nano microprocessors.

Acknowledgment

The author would like to thank Jared Davis, Anna Slobodova, and Sol Swords, for the work that they have done to make this effort possible, and for their contributions to this paper.

REFERENCES

[1] Robert S. Boyer and Warren A. Hunt, Jr.: "Symbolic Simulation in ACL2", with Robert S. Boyer, in the Proceedings of the Eighth International Workshop on the ACL2 Theorem Prover and its Applications, May, 2009.

[2] Bishop C. Brock and Warren A. Hunt, Jr.: "The Formalization of a Simple HDL", *Proceedings of the IFIP TC10/WG10.2/WG10.5 Workshop on Applied Formal Methods for Correct VLSI Design*, Elsevier Science Publishers, 1989.

[3] Bishop C. Brock and Warren A. Hunt, Jr.: "The `DUAL-EVAL` Hardware Description Language and Its Use in the Formal Specification and Verification of the FM9001 Microprocessor", in *Formal Methods in Systems Design*, Volume 11, pp. 71–105, Kluwer Academic Publishers, 1997.

[4] Robert S. Boyer and J Strother Moore: "A Computational Logic Handbook", Academic Press", 1988.

[5] Mike Gordon, "Relating event and trace semantics of Hardware Description Languages", in The Computer Journal, Volume 45, No. 1, 2002.

[6] Warren A. Hunt, Jr.: "The DE Language", in Computer-Aided Reasoning ACL2 Case Studies, edited by Matt Kaufmann, Panagiotis Manolios, and J Strother Moore, Kluwer Academic Publishers, 2000.

[7] Warren A. Hunt, Jr. and Erik Reeber: "Formalization of the DE2 Language", in Correct Hardware Design and Verification Methods (CHARME 2005), *Lecture Notes in Computer Science*, No. 3725, pp 20–34, Springer-Verlag, 2005.

[8] Warren A. Hunt, Jr. and Sol Otis Swords: "Centaur Technology Media Unit Verification", *Proceedings of the 21st International Conference on Computer Aided Verification (CAV 2009)*, In Computer-Aid Verification (CAV) 2009, LNCS No. 5643, pp 353–367, Springer-Verlag, June, 2009.

[9] IEEE Computer Society: IEEE Standard for Floating-Point Arithmetic. IEEE Std 754^{TM}-2008 edn.

[10] Roope Kaivola, Rajnish Ghughal, Naren Narasimhan, Amber Telfer Jesse Whittemore, Sudhindra Pandav, Anna Slobodová, Christopher Taylor, Vladimir Frolov, Erik Reeber and Armaghan Naik, *Replacing Testing with Formal Verification in Intel CoreTM i7 Processor Execution Engine Validation*. In Computer-Aid Verification (CAV) 2009, LNCS No. 5643, pp 414–429, Springer-Verlag, June, 2009.

[11] Matt Kaufmann, Panagiotis Manolios and J Strother Moore. Computer-Aided Reasoning: An Approach. Kluwer Academic Press, Boston, Massachusetts, 2000.

[12] LSI LOGIC. 1.5-Micron Array-Based Products Databook. LSI Logic Corporation, Milpitas, CA. 1990.

[13] T. Quarles, A. R. Newton, D. O. Pederson, A. Sangiovanii-Vincentelli. SPICE 3B1 User's Guide. Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, California. April, 1987.

[14] Erik Reeber and Warren A. Hunt, Jr., *A SAT-Based Decision Procedure for the Subclass of Unrollable List Formulas in ACL2 (SULFA)*. In the Third International Joint Conference on Automated Reasoning, Springer Verlag, Volume 4130, pp. 453–467.

[15] David Russinoff, *A Case Study in Formal Verification of Register-Transfer Logic with ACL2: the Floating-point Adder of the AMD Athlon (TM) Processor*. In: Formal Methods in Computer-Aided Design. (2000) 22–55

[16] Konrad Slind and Michael Norrish, *A Brief Overview of HOL4*. In TPHOL, pp. 28-32, 2008.

[17] Eelco Visser, *A Survey of Strategies in Rule-Based Program Transformation Systems*. In the Journal of Symbolic Computation, Volume 40, Issue 1, pp. 831–873, July, 2005.