A Flexible Formal Verification Framework for Industrial Scale Validation

Anna Slobodová, Jared Davis, Sol Swords
Centaur Technology
Austin, Texas, USA
{anna, jared, sswords}@centtech.com

Warren Hunt, Jr.
Centaur Technology and UT Austin
Austin, Texas, USA
hunt@cs.utexas.edu

***Abstract*—In recent years, leading microprocessor companies have made huge investments to improve the reliability of their products. Besides expanding their validation and CAD tools teams, they have incorporated formal verification methods into their design flows. Formal verification (FV) engineers require extensive training, and FV tools from CAD vendors are expensive. At first glance, it may seem that FV teams are not affordable by smaller companies. We have not found this to be true. This paper describes the formal verification framework we have built on top of publicly-available tools. This framework gives us the flexibility to work on myriad different problems that occur in microprocessor design.**

I. Introduction

With the increasing complexity of microprocessors, the design-process bottleneck has shifted from the design to its validation. Pre- and post-silicon validation teams continue to grow relative to the size of the design teams, and more and more companies are embracing formal methods as a complement to traditional, simulation-based validation. However, formal verification is still exotic and commercial FV tools remain costly. Even with a site license for a large organization, leasing each FV tool might cost over $100,000 per seat per year, and even with conservative use, many licenses may be needed to cover the needs of the designers. A different approach is to hire FV experts, generally PhD-level engineers, that can build in-house FV tools. Both solutions are generally too expensive for small or medium-sized companies. However, as we describe in this paper, there is a way for a company of any size to take advantage of formal verification. With a handful of FV engineers and publicly available software, FV tools can be developed and deployed.

The verification framework described in this paper is in daily use at Centaur Technology—a foundry-less company that designs the Via Nano, a low-power, high-performance, fully X86-compatible microprocessor. Centaur has about one hundred employees and tens of contractors, and carries out all phases of a fully-custom design process, including post-silicon system validation. Four years ago, the founders of the company decided to start a FV pilot project and challenged two of us to formally verify the Nano's microcode implementation of floating-point division. This was done successfully with the ACL2 [1], [2] theorem prover.

Centaur responded with an even more challenging problem: the verification of the Register-Transfer Level (RTL) implementation of their floating-point addition (FADD) and subtraction hardware. During the verification of the floating-point addition/subtraction hardware, a bug was discovered that FV experts dream about; this bug only occurs when performing floating-point addition/subtraction with extended precision for exactly one pair of numbers (modulo commutativity). This problem would be practically impossible to uncover by simulation. This showed a clear advantage of formal methods as the proof can be run in under one hour while the time required for a full simulation is not practical. The FADD unit also implements integer-to-floating-point conversions, floating-point-to-integer conversions, and many media and logical instructions. Later, we went on to verify almost hundred of these data-manipulation instructions [3], [4].

After this experience, the company decided to create a small FV team that today counts for three regular employees and several external consultants. Even though the absolute size of our FV team is small, to the best of our knowledge Centaur's relative investment in FV is larger than at any other company.

Our goal is to find efficient ways to help designers to achieve their goals. We try to focus on problems that provide relatively large returns on small investments. We have been primarily concerned with the functional correctness of the RTL and transistor-level designs. Our efforts include equivalence-checking of the design models at different levels of abstraction, property-checking, static analysis, consistency checking, and a variety of derived problems. Sometimes we are only able to provide a partial solution, either because of the time pressure, or because of the character or scale of the problem.

We make heavy use of open source software. A key part of our framework is the ACL2 theorem prover. ACL2 implements a mathematical logic which is also a functional subset of Common Lisp. We use the ACL2 language to describe circuit models, and its associated theorem prover to prove properties about them. In addition, because ACL2 is a programming language, we use it to write our own verification tools, specify their correctness and prove them correct. This extensibility is one of the most valuable features of ACL2. We can modify existing tools and tailor them for the needs of the company. The use of ACL2 for design verification is nothing new [5], but our development of tools extending ACL2 for hardware

validation is extensive.

ACL2 is a trustworthy tool. It has been hardened by many groups and individuals in academia and industry for more than a decade, and it won the ACM Software System Award in 2005. Because of this quality, we generally start by trying to carry out proofs in ACL2 alone. When this does not suffice, we may appeal to special-purpose tools outside ACL2, which generally provide more verification capacity for particular problems. Sometimes these tools can emit results that can be verified by ACL2, e.g., the Berkeley ZZ SAT-solver [6] can be set to emit a proof that we can check. For other tools, e.g. ABC [7], we are forced to trust the results. In such cases, we tag the results so it is clear which proofs rely on the correctness of these external tools.

In this paper, we describe our framework and its relation to the design flow (Section II). Then we give examples of different validation problems that occur in the process of designing a microprocessor, and describe some of our solutions (Section III). We have made large improvements in the robustness and capacity of our tools since we started to use them at Centaur. We conclude by summarizing these improvements, new features of our system, and our future plans (Section IV).

II. Methodology and Framework

All of the tools in our verification framework are tied together in the ACL2 system. Some of these components are shown in Figure 1. Many of our tools (e.g., our Verilog Translator, BDD library, and symbolic simulators) are written directly as ACL2 programs. We also have ACL2 interfaces for connecting to tools written in C/C++, such as the ABC system [7] and the Berkeley ZZ SAT-solver and model-checker by Niklas Een. The Berkeley ZZ model-checker is SAT-based and includes an efficient implementation of bounded model checking (BMC) [8], an interpolation-based model checker [9], and Property-Directed Reachability (PDR) [10], [11] (Section II-C).

Design can enter our system in two ways. RTL written in Verilog is translated into a formal model by Verilog Translator (Section II-A). Transistor Analyzer (Section II-B) extracts finite-state machines from transistor-level design. Our verification engine is build on top of AIG- and BDD-based symbolic simulator. Its seamless connection to the theorem prover is provided by GL-system (Section II-E).

A. Verilog toolkit

A first step in analyzing circuits is to load them into our verification system. We currently have two paths for loading modules, one for RTL and one for transistor-level designs. The Nano's RTL model is around 600,000 lines of Verilog. To load it, we have developed a Verilog-processing tool named *VL-Translator (VL)*, which has been recently released as a part of the ACL2-Books repository [12].

VL-Translator is in many ways similar to a synthesis tool without optimization. It parses the Verilog code, resolves all

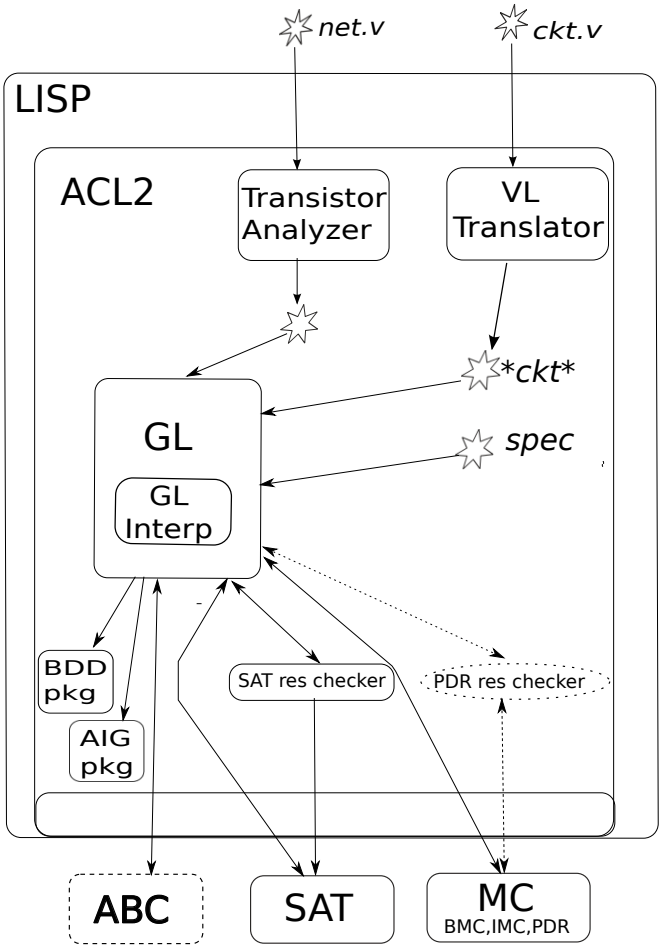

Fig. 1. FV Framework

of the connections between modules, recognizes flops and latches, and so forth. Expressions like `a + b` are eliminated by "synthesizing" an adder module of the appropriate size out of gates. Unlike a real synthesis tool, no attention is paid to making the circuit perform well. Instead, the focus is on ensuring that the translated modules are "conservative" with respect to Verilog's 4-valued semantics. For instance, the adders we synthesize completely unrealistically include X-detection circuitry so that if any bit of either input is X (unknown) or Z (undriven), the whole result is X; this is the behavior Verilog mandates for the + operator.

Once Verilog modules have been fully simplified by VL, they can be easily converted into *E modules* [13], [14], a simple, hierarchical representation that we use as the basis for our symbolic simulator. VL can skip over modules that have problems and produces good warnings and error messages. It is able to analyze and translate the entire Nano design in about twenty minutes of single-threaded execution on a 2.93GHz Intel® Core-2 Xeon® machine. The fast build time of our formal model allows us to run regressions of a substantial subset of our proofs every night, and to work with a current model on a daily basis. Previously translated models can be

loaded into ACL2 in a matter of seconds.

Verilog translation is a mundane necessity, but it is also difficult due to the scale of the language. It is tricky at times, e.g., Verilog's rules for expression types and sizes are quite complex. Since VL already dealt with parsing, sizing, etc., it did not take much effort to build a *linting* tool on top of it. This linting tool looks for things like signal-name typos, duplicate or undefined modules, duplicate wire assignments, unused/undriven wires, multiply driven wires, and size mismatches in assignments. It also looks for patterns that sometimes point to potential problems. For instance, skipped wires in the assignment

$$somevld = write3vld \mid write3vld \mid write1vld \mid write0vld$$

would be flagged as suspicious since $write2vld$ is skipped and $write3vld$ is repeated. While linting is not really formal verification, this reuse of VL took little effort to develop and has really paid off. During a recent re-partitioning of the existing design, it was used to check for thousands of potential wire mismatches and type errors.

Another tool built atop VL is a web-based *module browser* for visualizing the RTL sources. The module browser allows a user to follow a wire through multiple levels of the hierarchy, and it is in many ways a better way to view RTL design than by searching through the Verilog files using an editor. It also visualizes hierarchy of modules as a graph where nodes are labeled by instances of the used submodules.

B. Transistor analyzer

The other way to load circuits into our system is with our transistor analyzer. The Nano's transistor-level netlist can be printed as either a Spice or Verilog file, either of which can be read by our transistor analyzer. When reading Verilog, we can leverage VL to additionally allow RTL-level constructs to be included. To begin, we use an algorithm described by Bryant [15] to derive an And-Inverter graph (AIG) [16], [17] representation of the local update function for each wire in terms of inputs to the respective channel-connected component. This is a switch-level model of the circuit, but we can use it to develop a cycle-based model that can be compared against an RTL design.

The major steps of the process are shown in Figure 2. The cycle-level model that we wish to produce consists of a set of update functions for the wires of the circuit, each in terms of the primary inputs and the previous values of the state-holding wires, which are a small subset of the full set of wires. These update functions represent each wire's new value after a clock cycle; they are computed by composing together some number of phase-level update functions, each representing a time period in which the clock and inputs are held constant for a long enough time for the circuit state to settle. The update functions for these phases are in turn composed of several unfoldings of a unit-time update function, which itself is the composition of the switch-level update functions computed by Bryant's algorithm. More details on the process of obtaining these update functions follow.

We begin by addressing state-holding elements in the circuit. We look for combinational feedback loops among the switch-level update functions of the circuit, and then break the feedback arc of a loop by inserting a delay element (a clockless flop) on that wire. This removes the zero-delay combinational feedback loops while still allowing each state-holding wire to depend on its previous value.

The phase-level update functions that we wish to eventually derive represent each wire's update over a longer time period, in which the primary inputs and clocks are held constant until the circuit settles to a steady state. One can express this steady state in terms of the held values of the inputs and clock and the initial values on the state-holding wires. However, to do this correctly, we need to resolve timing races. For example, let us consider a circuit representing a latch. If the enable signal for the latch transitions from 1 (transparent) to 0 (opaque) and its data input is also changing in the same phase, then its next state depends on whether the updated data arrives before the enable signal falls. It is difficult to recover accurate timing information from a netlist alone, so instead we rely on the circuit designers to provide timing information. (This is no additional work to the designers since they already provide "tick delays" for Verilog simulators, so they can reliably simulate a transistor-level design.) We represent each 1-tick delay as a clockless flop. Therefore, when we compose together our switch-level update functions to get flop-to-flop update functions, these are effectively the updates for a single tick-delay.

We then compute the phase-level update functions for each flop in terms of the primary inputs and flops. This is done by unfolding the single-tick update function under stable inputs until a fixpoint is reached. The unfolding process starts at a symbolic initial state. Each step represents a unit delay. The values of primary inputs are held constant while the states are allowed to update. During this process, we check for combinational equivalence between the nth and $n+1$st

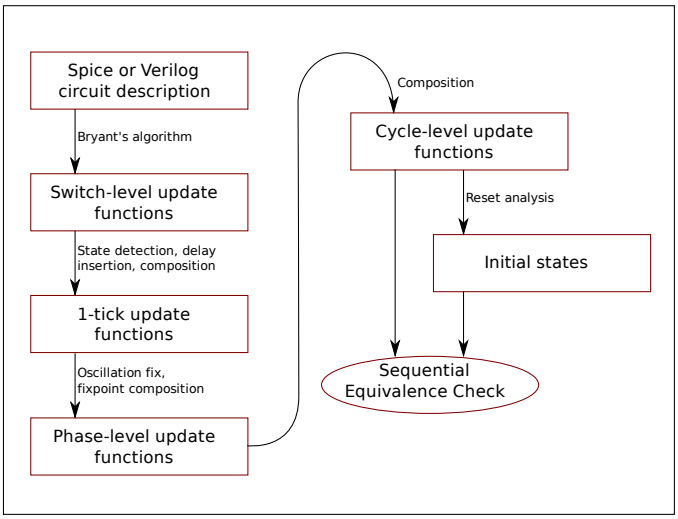

Fig. 2. Transistor analyzer/equivalence checker flow

unfolding. When two consecutive unfoldings are combinationally equivalent, then a fixpoint has been reached. Since the unfolding is much easier than checking equivalence, we may, for optimization purpose, skip the latter for several steps, and check whether we reached a fixpoint every few steps only.

There is one tricky point in the fixpoint computation described above. For some settings of the inputs and initial states, the circuit may oscillate and never arrive at a stable state, and therefore there may not be a fixpoint. While such an oscillation should not occur in the actual operation of the circuit, sometimes they may occur starting from an unreachable state. We resolve this problem by identifying an oscillation condition, then modifying the original update function for the affected state bits by setting them to X when the oscillation condition occurs. This modification is conservative, and will only produce X-values in symbolic simulation if such an oscillation occurs in a reachable state.

In order to compute the single-clock-cycle update function, we first compute the fixpoint, as described above, for each of several phases that make up a clock cycle. These phases are determined by the relative timing of the clocks and input signals. For example, if the inputs arrive by some setup time before the positive clock edge and are held for some hold time after that edge, but may be unpredictable in between, then an appropriate sequence of phases would be:

1) Clock low, inputs set to symbolic variables (setup time)
2) Clock high, inputs remain set to the same symbolic variables (hold time)
3) Clock high, inputs set to X
4) Clock low, inputs remain set to X.

We then compose these phases together into the single-clock-cycle update functions.

In order to prove this cycle-level model sequentially equivalent to an RTL specification, we first search for a reset state. A reset sequence for both the circuit and its RTL specification can be found by using a bounded model-checking search from an initial state where all state-holding elements are set to X. This search finds an earliest reachable state in which all state-holding elements are non-X. The counterexample produced by the BMC run is a reset sequence for both the circuit and its specification, and we run this reset sequence on both models to find corresponding starting states. We then use the sequential equivalence checking capabilities of ABC and Berkeley ZZ to prove equivalence between the circuit and RTL models.

Our transistor-to-RTL equivalence checking tool is in an early stage of development, but it is able to work with transistor-level designs containing hundreds of thousands of transistors and sophisticated dynamic logic schemes. As this tool matures, we expect it to replace the expensive and inconclusive equivalence-checking tools now used. It will also give designers the comfort of in-house tool support and customization.

C. External tools

The ACL2 system provides a way to interface with external tools. In order to protect soundness, the ACL2 system requires a *trust tag* to be declared before an external tool may be called. This mechanism allows us to keep track of the results obtained outside ACL2. We use it to connect to ABC and Berkeley ZZ. Our main uses of ABC are for sequential equivalence checking and to simplify AIGs during transistor analysis, where the size of AIGs grows very fast and affects our ability to check whether we have reached a fixpoint. At this point we do not have any ability to verify that the AIGs returned from ABC after simplification are functionally equivalent to the original AIGs. Therefore, whenever we use ABC, our final theorems are reliable only if ABC returns reliable results.

When we use the Berkeley ZZ SAT-solver, we have the ability to verify the result returned. If the AIG represents a satisfiable condition, we will get a satisfying assignment that can be checked by ACL2. If Berkeley ZZ claims that the formula is unsatisfiable, it is also possible to ask that Berkeley ZZ return a resolution-style proof of its claim; this proof is then used as a *hint* for the ACL2 theorem prover to prove that the formula does indeed always evaluates to false. We developed a proof-checker for this output in ACL2 that was also verified [18]. Since this checking process is decoupled from the implementation of any external SAT-solver, we can use any SAT-solver as long it can emit a similar resolution-style proof. During proof development, we generally turn off our proof-checking procedure and use the SAT-solver with ACL2's trust-tag mechanism; we also forgo its use when debugging.

The Berkeley ZZ system is tightly connected to our system. In fact, it can be compiled as a shared object library that our underlying (ACL2) Lisp directly loads. The Berkeley ZZ system contains several model-checkers; their input consists of an initial state I, a transition function represented as an association T of update functions to state variables, and a property P. All sets and functions are represented as AIGs. At this time, we use the Berkeley ZZ model-checkers with trust tags. However, since the PDRmodel checker returns either a counterexample (in case of failure), or (in case of a valid property) it can return a stronger property Q that is an inductive invariant, we plan to add a proof-checker support similar to that of the SAT-solver-checker. For a valid property, we only need to to check three conditions:

1) whether the initial state satisfies the (new) strengthened property ($I \implies Q$),
2) whether the new property implies the original property ($Q \implies P$)
3) whether the new property is inductive:

$$(Q \wedge T) \implies Q'$$

where Q' is the property with state variables substituted by their update functions.

The first two conditions are generally easy to check using BDDs, or we can use our verified SAT-solver. We can check the third condition with our verified SAT-checker mechanism.

Our system can accommodate additional external tools. We use two main criteria for choosing external tools:

- Are the results of the tool verifiable, and is it easier to check them than to compute them?
- Does the tool provide superior techniques that would be hard to reproduce in the ACL2 system?

Whenever a tool helps us to solve more problems we are open to consider its addition to our verification framework.

D. BDD library

Using ACL2, we have implemented a library for working with Binary Decision Diagrams (BDDs) [19]. The library relies on ACL2's built-in hash-consing and memoization schemes [20] for performance, but its operations are all implemented quite simply and can therefore be reasoned about. Each operation is proven correct with respect to a definition of BDD evaluation. That is, for any Boolean operation $\times$ implemented in the library, we prove

$$\forall x \in B^n : (f \otimes g)(x) = f(x) \times g(x)$$

where f and g are BDDs, $\otimes$ is a Boolean operation over BDDs, and $\times$ the respective Boolean operation. The BDD library plays an important role in many of our proofs about arithmetic circuits.

E. GL

Verification of a complex software or hardware system can become overwhelming, even to a knowledgeable verification expert. In addition, when the main idea of the proof is clear, numerous low-level lemmas may need to be proven in support of the top-level theorem. We use the ACL2 theorem prover to orchestrate such efforts; users provide milestones that guide ACL2 to the final goal. These milestones come in the form of lemmas about the design, and their formulation may require a deep knowledge of Nano design details. Changes in the design may invalidate previously proven lemmas and subsequently require new lemmas to be proven.

In our verification efforts, we find that many claims involve functions defined only over a finite domain. When such a claim is made, it can often be discharged by symbolic simulation of the functions with minimal manual intervention of the user. The motivation behind the *G system* [21] was to simplify the proofs of theorems with finite-input domains. The G system allowed the symbolic simulation of all ACL2 functions; this was implemented by representing the input domain as BDDs and simulating ACL2 functions with a corresponding BDD-based symbolic simulator. The G system rendered proofs of many lemmas into symbolic executions. The G system was implemented in Common Lisp and hence outside ACL2 logic; therefore, there was no way to prove its soundness, and G was used with a trust tag.

In 2010, Swords re-implemented the capabilities of the G system using only ACL2 code. This new system is called GL – for *G* in the *Logic* – and used ACL2 to prove its implementation is correct [4]. GL allows any ACL2 function to be either translated into its symbolic counterpart, or be symbolically interpreted on the fly. We generally use GL-based symbolic simulation as a decision-procedure. GL can be used for any function admitted to ACL2 whose input is constrained to a finite domain.

The GL system has became a core element of our verification framework, and this framework takes full advantage of our ability to manipulate AIGs and BDDs. GL allows us to write our specifications at a fairly high level of abstraction; for example, we often use integers instead of bit-vectors.

GL claims have a form

$$hypothesis \implies conclusion$$

where the hypothesis identifies a finite input domain, and the conclusion a conjecture we would like proven. An example of hypothesis is an ACL2 statement below for the assumption: `a` is a 8-bit integer, and `b` is a 8-bit natural number, and `flgs` is a Boolean vector of length 3.

```
(and (integerp a)
     (<= -128 a)
     (< a 128)
     (integerp b)
     (<= 0 b)
     (<  b 256)
     (boolean-listp flgs)
     (= (length flgs) 3))
```

GL operates in two main modes. In BDD-mode, the mapping of bits within the theorem's variables to BDD variable indices is controlled by the user; each variable is bound to a symbolic object. Sufficient coverage of such bindings is automatically checked; that is, for any values of a, b and $flgs$ that satisfy the hypothesis, there must be a possible evaluation of the symbolic objects that results in those values. The hypothesis is used as a condition when parametrization [22] is performed; this restricts subsequent symbolic computations to only inputs satisfying the hypothesis. When using AIGs instead of BDDs, GL can query the SAT solver either with or without checking the UNSAT proof. The GL system has been recently released as a part of the ACL2 books repository [12].

III. Examples of Problems And Solutions

In this section, we will point to several examples of problems that were raised in course of the design process. Some of them are classic problems that have been described in earlier publications (e.g., verification of arithmetic circuits), other might appear new to the reader. Our main goal is to illustrate how our framework allows us to quickly respond to various requests, which is important in the time-sensitive context of an industrial design process.

A. RTL verification of arithmetic circuits

Our first formal verification effort at Centaur was in the area of the functional correctness of the RTL design [3], [23]. In particular, we focused on the execution units that perform arithmetic and logical operations, such as add, subtract, multiply, divide, bitwise operations, conversions, and string manipulations. The micro-operations that are actually implemented by the hardware are often closely related to counterparts on

the architectural (ISA) level, which are documented in publicly available *Intel(R) 64 and IA-32 Architecture Software Developers Manual* (available at Intel's web site http://www.intel.com). Therefore, clarifying these specifications and writing them formally was easier than for some other parts of the design. The Nano design poses many of the same challenges faced by the pioneers of formal verification of arithmetic circuits in industry: Intel [24]–[26], AMD [27]–[30] and IBM [31]. Thanks to our previous experience [25], [32]–[35] and published knowledge in this area, we were adequately prepared to tackle this kind of problem.

After verifying the unit responsible for floating-point addition, conversions and logical operations [14], we worked on integer and later on floating-point multipliers. Centaur's design is fully compatible with the 64-bit extension of IA-32(R) architecture. It implements a whole variety of integer-, x87 floating-point-, and packed (Single Instruction Multiple Data) integer and floating-point multiply instructions.

Verification of multipliers has been previously done at Intel, AMD and IBM [25], [28], [31], [34]. While the approach taken at AMD appears to be most rigorous, it also requires the most manual intervention from a very sophisticated user. It uses ACL2 – every lemma is an ACL2 formula. The main correctness theorem is obtained from a huge number of low-level lemmas derived from the design and composed into higher-level theorems by a knowledgeable ACL2-user and mathematician. Although it hasn't been explicitly stated, the proof does not seem very robust with respect to the changes to design. That means a limited portability from project to project. The approach taken by Intel and IBM, similarly to our approach, heavily relies on symbolic simulation that allows a fair amount of automation. It requires sequential decomposition of the design, which adds a layer of complexity because one needs to find appropriate properties for signals at the decomposition boundaries. The strength of Intel and IBM's approaches lie in very efficient symbolic simulators. Their weakness is in the way the lemmas are composed into a top-level theorem. While Intel relies on a light-weight theorem prover that has only a loose connection to symbolic simulator. We are not aware of any mechanical theorem prover used in IBM proofs. Although the top-level multiplier proofs can be done by hand, there is always space for a human error.

There are similarities and differences of the previous approaches and ours. Our proofs of multipliers are done completely within ACL2. We use GL to prove the low-level lemmas about symbolic values computed by concrete signals over a fixed number of steps. Even though BDD-based GL has been satisfactory, we made a use of the verified-SAT (see Section II-C) as as well. By composing these lemmas together, we proved that the RTL design correctly implements the Radix-2^k Booth Encoding algorithm within the specified latency. Next step was to prove that this algorithm indeed implements integer multiplication. To complete the floating-point multiplication proof, we proved the correctness of the floating-point result, including the exception flags, after rounding. This was also done using GL.

We were able to verify that all of the IA-32e multiply instructions, both integer and floating-point, were correctly implemented. The verification of multipliers fully stressed our BDD- and AIG-based methodology. The intrinsic complexity of the multipliers forced us to work on optimizing the performance of our system. All proofs showed great portability when we adopted them to a new project. Smaller proofs for instructions that include 8x8, 16x61 and 32x32 multiplication run as part of nightly regression proofs, while the regressions that involve 64x64 multiplication proofs require hours and run on weekly basis.

More recently, we turned our attention to the Nano MMX/SSE unit. This unit implements around 120 operations for working with packed integer data, e.g., it can add, subtract, shift, and compare packed data (signed and unsigned bytes, words, double-words, and quad-words); and shuffle and blend parts of vectors. Unlike our previous arithmetic verifications, this work began after GL had been connected to the proof-producing SAT solver, so we were able to easily switch between BDDs and SAT for particular instructions. Within a couple of weeks we were able to verify all but three instructions (which have a more complex interface) against simple ACL2 specification functions. The entire verification can be re-run in forty minutes (and we do this nightly), except for the MPSADB instruction which takes almost three hours for the SAT solver to finish.

All these proofs are very robust and porting them to a new design project required minimal effort. We regularly run regressions of the proofs to monitor changes in the design, and occasionally find newly introduced problems.

B. RTL-to-RTL equivalence checker

Whether it is a fix of a bug, or a change to improve timing, or addition of new functionality to the design, we need to assure that the RTL changes are done as intended. If those changes do not cross the latch boundaries, i.e. are purely combinational, logic designers can verify them using a web based tool implemented in our framework. The tool does not require any knowledge of formal methods. Designers are required to write in Verilog a wrapper around the two modules they want to compare, identifying matching inputs and outputs, and conditions under which they are supposed to match. In case of a mismatch, a counterexample is generated that can be directly animated by a Verilog simulator.

This is a perfect example of a lightweight FV tool that keeps designers in their comfort zone. So far there hasn't been any request for its extension to a more general equivalence checker, but we are leaving this option open.

C. Late changes in the design

The later a bug is found in the design process, the more costly it is to repair. Before any version of the Nano goes out as a product, it goes through intensive testing and debugging. Lab parts have to be able to boot tens of different operating systems, work under different temperatures and various voltages. Any bug that is found in this late phase of the process

causes delays that directly translate into financial loss. There are different ways to fix discovered problems. One way is to change the Nano microcode; this only requires metal-level mask changes. A more extreme update is to change transistors, thus requiring many more production masks to be changed. When making late-stage changes, it is important to keep a hardware fix as local as possible so the rest of the layout does not need to be changed; therefore, redundant gates/transistors are included with the Nano design. When there is a need for a minor fix, logic designers may use these extra gates, so only wiring needs to be changed. Finding a close-to-optimal solution for a late fix can be a non-trivial task and may require a lot of effort, and this occurs when time is most critical. In order to help Nano designers with this task, we have a tool that finds candidates for bug-fixes, and proves, after the change, that the circuit-level implementation with the suggested change is equivalent to the fixed logic design.

The task can be formulated as follows: given two Verilog files – one on the RTL level and one as a flat gate-level netlist, find a mapping between the signals in RTL and the netlist. We assume that the inputs and outputs of the design match and the state-elements in both designs match (modulo inversion). Our tool uses commonly-known, equivalence-checking techniques, including random simulation to compute candidate equivalence classes. We also use ABC to simplify AIGs and the SAT-solver to check final equivalences. Thus, a designer can be presented with potential solutions for a bug fix that is assured to be correct.

D. Clock-tree analysis

Another designer request was to analyze the Nano's clock tree. For a digital circuit design, the clock signal has to be distributed across the whole chip. The clock going into a particular unit is often gated (*and*ed with an enable signal) so it can be disabled. Disabling the clock is almost like turning off the unit: its state is frozen so the values on (most of) its wires stop changing, saving power.

Since it is challenging to effectively distribute the clock signal throughout the chip, units often have several clock inputs instead of just one. By convention, these clocks are supposed to be equivalent. That is, if there is any clock gating in that unit, then the enable signals to each clock should be equivalent.

The designers wanted a way to check that this convention was met, since otherwise serious electrical problems could result. This sounds like a simple problem with a clear solution: all we need to do is symbolically simulate the processor, then gather the expressions for the clock signals being given to each unit and check whether they are equivalent. The logic in the clock tree is simple, so the equivalence check should be trivial. Unfortunately, symbolic simulation of the whole processor is still out of our reach. There are several difficulties. Most obvious is the scale of the problem. The more pragmatic one is that certain Verilog constructs are not yet supported by VL. We were able to clear these hurdles by using a cone-of-influence reduction to extract the clock tree from the rest of the processor, along with some manual overrides of problematic modules. The clock verification has been reduced to a very simple combinational equivalence problem.

E. Dependency analysis

The problem described in this section occurred in the context of the verification of transistor-level design before we had a transistor analyzer (Section II-B. One of the classic hardware design problems is to assure equivalence, or some other weaker notion of correctness. For example, is the RTL design correct with respect to its architectural model? Is the transistor-level implementation of the design correct with respect to the RTL? Both questions can be partially answered by simulation-based techniques or by application of formal methods. Tools providing the latter are generally known as equivalence checkers. The difficulty of the equivalence-checking problem depends on how closely the transistor-level implementation structurally matches its RTL counterpart. If the state elements of the two design levels match, the problem is reduced to combinational equivalence problem. Otherwise, we might be lucky to find a retiming transformation that brings the two designs to a state-mapped pair. In the worst case we end up with a general model-checking problem for a safety property.

All of these cases have, at least in theory, a solution. However, we have not mentioned the preamble of the verification of transistor-level design: we are dealing with two designs on very different level of abstraction – RTL as a specification and a transistor-level implementation. In order to formulate the equivalence checking problem, we need to extract a finite-state machine from both – the RTL design and the transistor-level design. At the time when the problem occurred, we had a solution for the former, but we did not have a complete solution for the latter (until we completed our transistor analyzer), at least not for the general case of a complex dynamic design.

Centaur designers were relying on a CAD-vendor-supplied tool. The tool takes a Spice file and a corresponding Verilog file, extracts finite-state machines for both, and performs concrete or symbolic simulation of both levels while comparing their respective outputs. The capacity of this commercial equivalence tool is limited and often it switches to random simulation after just a few symbolic steps. If the tool finds some mismatches, it reports them in a form of a counterexample. However, if the tool does not find any counterexample, designers do not really know if the two models are equivalent – especially when the tool switches to random simulation. How good is the coverage of the tests? Even when the tool finishes symbolic simulation successfully, designers are left with the question whether the number of steps simulated is actually sufficient to ensure the equivalence of the two models.

In order to help designers with the settings of the equivalence-checker, we provide a tool that checks dependency of RTL signals on other signals and their delays. Often a pipelined design transforms its inputs to its outputs in a fixed number of cycles. In most cases, a designer knows the depth

of such a pipeline. However, this may not be as simple as it looks. Under some conditions, inputs may be stored in internal latches before they propagate. In order to relieve the complexity of the symbolic simulation, some of control signals may be set to specific values. This can effect the time needed to propagate inputs to the outputs. Our tool can help designers discover a *reset* sequence that brings a machine model to a state where all state-elements have Boolean values. Once done, we then check whether, starting from Boolean values, the states maintain Boolean values. Next we compute for each output signal its dependency on input signals and state-holding elements, and delays. This is done by checking dependency of update functions on variables using SAT-solver.

As an example, consider an output o in a perfectly pipelined design. Assume that o at time 2 depends on inputs from set I_0 at time 0, I_1 at time 1 and I_2 at time two. If at time 3, the same output depends on I_0 at time 1, I_1 at time 2, and I_2 at time 3, we can safely say that 3 symbolic simulation cycles are enough to verify o. The situation might be more complex and we may need to consider more complicated scenarios. In any case, dependency analysis provides us a better estimate of the quality of the results obtained from our commercial equivalence checker.

IV. Conclusion

We described the formal verification framework being used at Centaur. It is based on ACL2 – a theorem prover that is publicly available under GNU General Public License. So far we are very happy with its stability and support. It gives us valuable flexibility to implement methods and techniques and prove their soundness. The VL translator allows us to build a formal model from a Verilog design. The GL system, which is implemented and verified in ACL2, equips us with SAT and BDD based procedures that are suitable for problems arising in hardware verification. Various problem-oriented tools have been developed and fully or partially verified. Our main goal is to help validation engineers and logic and transistor designers to reach their goals without sacrificing quality of the design. Our system allows us to react quickly to new problems that arise in the design process.

Since the first use of the ACL2 in Centaur's development environment, many ACL2 *books* that boost the performance of our verification framework have been developed. These include efficient hashing and garbage collection, memoization of functions, support for pseudo-random functions, efficient set operations, just to name a few. A package that manipulates special types of Binary Decision Diagrams (UBDDs) has been re-implemented within the ACL2 logic and proved correct. GL system and VL translator have been released to ACL2 community.

A verified connection to a SAT-solver enables us to solve problems that were out of reach of automatic verification before. One of our immediate goals is to enhance the PDR model-checker with proof checking. Another goal is to finish the transistor analyzer and incorporating it into an equivalence checker that can completely replace the vendor tools used now.

Our goal is to continue improving our tools, both in depth – allowing us to solve problems on a bigger scale, and with higher assurance of their correctness; and in breadth – enriching the variety of point tools tailored for specific problems in our designs.

Acknowledgement

We would like to thank Matt Kaufmann for his continuous support of ACL2 and Gary Byers for his support of GCL. Niklas Een gave us an early access to Berkeley ZZ and provided the means to integrate his tool in our framework and enabled SAT result verification. Alan Mishchenko was kind to answer all our questions about ABC. Also, our thanks go to Robert Boyer who was very active especially in the early hard days of FV at Centaur.

References

[1] M. Kaufmann, J. S. Moore, and P. Manolios, *Computer-Aided Reasoning: An Approach*. Norwell, MA, USA: Kluwer Academic Publishers, 2000.

[2] M. Kaufmann, J. S. Moore, and R. S. Boyer, "ACL2 version 4.2," `http://www.cs.utexas.edu/~moore/acl2/`, 2011.

[3] W. Hunt, Jr., "Verifying VIA nano microprocessor components," in *Proceedings of the Formal Methods in Computer-Aided Design*, R. Bloem and N. Sharygina, Eds. ACM/IEEE, 2010, pp. 3–10.

[4] S. Swords, "A verified framework for symbolic execution in the ACL2 theorem prover," http://hdl.handle.net/2152/ETD-UT-2010-12-2210, 2010.

[5] D. Hardin, *Design and Verification of Microprocessor Systems for High-Assurance Applications*. Springer, 2010.

[6] N. Een, A. Mishchenko, and N. Amla, "A single-instance incremental sat formulation of proof- and counterexample-based abstraction," in *Proceedings of the Formal Methods in Computer-Aided Design*, R. Bloem and N. Sharygina, Eds. ACM/IEEE, 2010, pp. 181–186.

[7] "ABC: A system for sequential synthesis and verification," http://www.eecs.berkeley.edu/ alanmi/abc/.

[8] A. Biere, A. Cimatti, E. Clarke, M. Fujita, and Y. Zhu, "Symbolic model checking using sat procedures instead of bdds," in *ACM Design Automation Conference (DAC)*. ACM, 1999.

[9] K. McMillan, "Interpolation and sat-based model-checking," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, vol. 2725. Springer-Verlag, 2003, pp. 1–13.

[10] A. Bradley, "Sat-based model checking without unrolling," in *In Proceedings of VMCAI*, 2011.

[11] N. Een, A. Mishchenko, and R. Brayton, "Efficient implementation of property directed reachability," in *Proceedings of IWLS*. IEEE/ACM, 2011.

[12] "ACL2 books repository," http://acl2-books.googlecode.com, 2011.

[13] W. A. Hunt, Jr. and S. Swords, "Use of the E language," in *Hardware design and Functional Languages*, 2009.

[14] ——, "Centaur technology media unit verification," in *Proceedings of the 21st International Conference on Computer Aided Verification*, ser. Lecture Notes in Computer Science, vol. 5643. Springer, 2009, pp. 353–367.

[15] R. E. Bryant, "Boolean analysis of MOS circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 6, no. 4, pp. 634–649, 1987.

[16] A. Kuehlmann and F. Krohm, "Equivalence checking using cuts and heaps," in *Design Automation Conference*, 1997.

[17] M. Ganai and A. Kuehlmann, "On-the-fly compression of logical circuits," in *IWLS*. IEEE/ACM, 2000.

[18] M. Kaufmann, "A framework for verified use of an external aig+sat solver with acl2," Talk: http://www.cs.utexas.edu/users/moore/acl2/seminar/#04-13-11, 2011.

[19] R. Bryant, "Graph-based algorithms for boolean function manipulation," *Transactions on Computers*, vol. C-35, pp. 677–691, 1986.

[20] R. S. Boyer and W. A. Hunt, Jr., "Function memoization and unique object representation for ACL2 functions," in *Proceedings of the Sixth International Workshop on the ACL2 Theorem Prover and its Applications*. Seattle, Washington: ACM Digital Library, 2006.

[21] ——, "Symbolic simulation in ACL2," *Proceedings of the Eighth International Workshop on the ACL2 Theorem Prover and its Applications*, 2009.

[22] M. D. Aagaard, R. B. Jones, and C.-J. H. Seger, "Formal verification using parametric representations of boolean constraints," in *Proceedings of the 36th Design Automation Conference*, 1999, pp. 402–407.

[23] W. Hunt Jr., S. Swords, J. Davis, and A. Slobodova, *Design and Verification of Microprocessor Systems for High-Assurance Applications*. Springer, 2010, ch. Use of Formal Verification at Centaur Technology, pp. 65–88.

[24] M. Aagaard, R. Jones, T. Mehlham, J. O'Leary, and C.-J. Seger, "A methodology for large-scale hardware verification," in *Proceedings of Formal Methods in Computer-Aided Design*, ser. Lecture Notes in Computer Science, vol. 1954. Springer, 2000.

[25] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodová, C. Taylor, V. Frolov, E. Reeber, and A. Naik, "Replacing testing with formal verification in Intel® Core™ i7 processor execution engine validation," in *Proceedings of the 21st International Conference on Computer Aided Verification*, ser. Lecture Notes in Computer Science, vol. 5643. Springer, 2009, pp. 414–429.

[26] R. Kaivola and N. Narasimhan, "Multiplier verification with symbolic simulation and theorem proving," in *Proceedings of Design, Automation and Test in Europe*, 2002.

[27] D. 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, pp. 22–55. [Online]. Available: http://www.springerlink.com/content/v573551x2q33162v/

[28] D. Russinoff and A. Flatau, *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Press, 2000, ch. Mechanical Verification of Register-Transfer Logic: A Floating-Point Multiplier.

[29] D. Russinoff, "A mechanically checked proof of IEEE compliance of a register-transfer-level specification of the AMD-K7 floating-point multiplication, division, and square root instructions," *LMS Journal of Computation and Mathematics*, vol. 1, pp. 148–200, 1998.

[30] ——, *Design and Verification of Microprocessor Systems for High-Assurance Applications*. Springer, 2010, ch. A Mechanically Verified Commercial SRT Divider, pp. 23–64.

[31] C. Jacobi, K. Weber, V. Paruthi, and J. Baumgartner, "Automatic formal verification of fused-multiply-add FPUs," in *Design, Automation and Test in Europe, 2005. Proceedings*, 2005, pp. 1298–1303 Vol. 2.

[32] W. Hunt Jr. and B. Brock, "The verification of a bit-alice ALU," in *Workshop on Hardware Specification, Verification, and Synthesis: Mathematical Aspects*, ser. Lecture Notes in Computer Science, vol. 408. Springer, 1989.

[33] A. Slobodová, "Formal verification of hardware support for Advanced Encryption Standard." in *Proceedings of Formal Methods in Computer-Aided Design*. IEEE Press, 2008, pp. 1–4.

[34] ——, "Challenges for formal verification in industrial setting," in *Formal Methods: Applications and Technology*, 2006, pp. 1–22.

[35] A. Slobodová and K. Nagalla, "Formal verification of floating-point multiply-add on Itanium® processor," in *Proceedings of Designing Correct Circuits*, 2004.