Hybrid Verification of a Hardware Modular Reduction Engine

Jun Sawada*, Peter Sandon†, Viresh Paruthi†, Jason Baumgartner†, Michael Case† Hari Mony†,
*IBM Austin Research Laboratory
†IBM Systems & Technology Group

***Abstract*—Wide-operand modular math functions pose an enormous challenge for verification. We present a novel method to verify a modular reduction engine implemented as a finite state machine (FSM), leveraging a combination of model checking and theorem proving. As a first step of the verification, pre-conditions and post-conditions for each state transition of the FSM are identified. Next the implications from the pre-conditions to the post-conditions are verified using a model checker. The last step entails combining all the implications in a theorem prover to derive the overall correctness proof. We carried out this verification using a hybrid formal verification platform comprising the *ACL2* theorem prover and IBM's model checker *SixthSense*, along with numerous techniques to cope with the complexities of this verification task. To our knowledge, this is the first published method for the exhaustive verification of an RTL-implementation of a wide-operand industrial modular reduction engine.**

I. Introduction

A. Modular Reduction

Cryptography is becoming a central feature of our networked world. Increasing performance demands on modern microprocessors have mandated native hardware support for encryption and decryption algorithms in the form of an on-board cryptographic accelerator co-processor.

Classes of cryptography asymmetric algorithms such as Rivest-Shamir-Adleman (RSA) and Elliptical Curve Cryptography (ECC) are realized using lower-level functions, such as Modular Reduction or Modular Exponentiation. These are implemented with finite state machines (FSM) which operate on wide-operands (e.g., on the order of 4096 bits), and may require a large number of clock cycles to complete the computation – even hundreds of thousands of clock cycles for large operand bit-widths.

Verification of such complex hardware is of critical importance, though poses formidable challenges. Traditional *informal* verification methods offer insufficient coverage given the wide operand widths, sequential depth of the computation and the inherently difficult nature of the logic. Even hardware-accelerated simulation and post-Silicon debug, offering dramatically greater explicit-state coverage, are rendered insufficient given the sheer size of the state-space. Additionally, the reference result needs to be computed in software which can prove to be the bottleneck. Traditional bit-level model checking approaches are unscalable even for small bit-widths of such arithmetic functions, and traditional higher-level techniques such as theorem proving become extremely tedious due to the need to reason about the intricate sequentially-deep state machines at the RTL level.

In this paper we present a method to verify modular reduction implemented as an FSM by leveraging a combination of model checking and theorem proving. Our approach decomposes the verification task into two parts: 1) verification of invariants associated with the FSM and 2) Combining the verified invariants to form a proof of correctness. The set of invariants describing the behavior of the FSM are verified using the model checker. These invariants are then combined by the theorem prover to form a proof that the FSM correctly implements the target algorithm. The presented technique allows us to overcome the limitations of traditional verification disciplines as outlined above. We can scale our technique to verify the correctness of modular reduction for a number of operand widths, leveraging the strengths of theorem proving to reason about parametrized computations, and leveraging the model checker to verify invariants which require precise characterization of temporally-deep RTL-implemented state machines.

B. ACL2SIX

There are two predominant formal verification techniques that have been successfully used to verify the correctness of bit-accurate sequential machines: model checking and theorem proving. Model checking is automated, though often fails to scale for designs containing complex arithmetic datapaths. On the other hand, interactive theorem proving techniques do scale, though often only with significant human effort – which may become formidable if requiring reachable-state characterization of complex bit-level state machines, or reasoning about bit-optimized arithmetic designs.

The combination of these two techniques is sometimes called *hybrid verification*, and may provide an ideal formal verification environment. The main motivation for the combination is to use the model checking for verifying the low-level details of bit-level hardware, and use theorem proving to focus the high-level mathematical and algorithmic correctness. A number of different hybrid tools have been developed [1], [2] and used for a variety of verification tasks [3], [4] in the past. However, it is our thesis that such hybrid tools have not been leveraged fully, due to either the weakness of the underlying model checker or limiting the theorem proving to a rather simplistic analysis.

Relating to verification of hardware encryption, Slobodová [5] verified microprocessor *instructions* to implement Advanced Encryption Standard (AES) algorithms against a reference model derived from its specification. Erkök et. al. [6] verified cryptographic hardware by checking equivalence between different stages of implementation. Smith et. al. [7] verified a Java block cipher implementations using a hybrid tool to symbolically simulate Java bytecode and equivalence-check the resulting expressions. This kind of equivalence checking relies on the fact that the operation takes a fixed delay and/or a fixed number of (micro-)instructions. A similar equivalence checking approach did not work for the FSM we will discuss in this paper, because the behavior of the machine significantly changes depending on input data, rendering typical equivalence algorithms such as BDD-sweeping ineffective. To our knowledge, encryption procedures implemented as intricate sequentially-deep state machines have not been fully verified.

From a perspective of practicality, it is also important to directly verify designs written in a Hardware Description Language (HDL) such as Verilog or VHDL. Our goal is to accept industrial designs written in the HDL without any modification. Some tools and past work have attempted to translate HDL into the theorem proving language [8], [9], [10]; however, full formalization of HDL is very tedious and difficult, or creates semantic gaps. Another problem in the translation of HDL to a formal language is that the theorem prover has to deal with low-level details of hardware. Industrial HDL often includes bit-level optimizations and peripheral circuit artifacts such as *scan* and *power-optimization* logic, overall hindering the effectiveness of the approach.

In our hybrid verification tool called ACL2SIX, we use a small subset of the theorem prover's language to specify properties of target hardware. The tool reads the unmodified hardware design written in an HDL and directly verifies properties on it. We use a powerful bit-level model checking tool in order to automatically prove sizable verification subproblems, thus reducing the burden on interactive theorem proving while making the proof script robust and reusable. We use a fully featured, general-purpose theorem prover to allow verification of sometimes-difficult higher level mathematical problems.

The rest of the paper is organized as follows. We start by outlining a typical modular reduction engine implemented in hardware as an FSM. We next describe the ACL2SIX hybrid formal verification platform which combines the ACL2 theorem prover with IBM's formal toolset SixthSense. We then describe our verification approach including the pre-conditions and post-conditions used in the context of modular reduction, as well as enhancements to the underlying model checker SixthSense to enable application of the hybrid platform to a large design. Finally we provide some results and conclusions from the novel application.

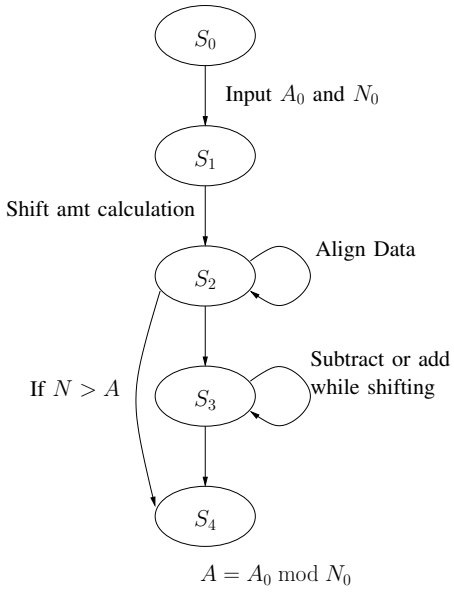

Fig. 1. Modular Reduction State Diagram

II. A Modular Reduction Engine

A modular reduction engine computes the remainder of one integer divided by another. That is, the engine computes $R = A_0 \bmod N_0$ for positive integers A_0 and N_0, such that $A_0 = N_0X + R$, and $0 \leq R < N_0$ for some integer X. For cryptographic applications, the size of these integers varies according to the strength of the cryptographic algorithm, with current applications requiring several thousand bits of precision.

The following is the modular reduction algorithm we wish to verify:

1. Set $A := A_0$ and $N := N_0$. Ensure that $A \geq 0, N > 0$.
2. If $A < N$, set $R := A$ and exit.
3. Left shift N to align its most significant '1' with that of A.
4. Divide Loop:
5. If $A \geq 0$, $A := A - N$, otherwise $A := A + N$.
6. If $N = N_0$, exit loop.
7. Right shift N by one bit.
8. Go to Divide Loop.
9. If $A \geq 0$, set $R := A$, otherwise $R := A + N$.

To understand how the algorithm works, note that $(A \bmod N_0) = (A_0 \bmod N_0)$ is an invariant, and $0 \leq R < N_0$ at the end of the algorithm.

Our goal is to verify a hardware which imlements this algorithm as an FSM. Figure 1 presents a state transition diagram from the design document of the hardware, and Table I provides an action table for the FSM.

S_0 **:** The FSM reads two input operands, A_0 and N_0, and stores them in the registers A and N.

S_1 **:** The FSM counts leading zero bits of N and A and stores their difference $lz(N) - lz(A)$ in the registers D and C. This corresponds to the number of bits to left-shift N in order to align the most significant bit of one in A and N.

TABLE I
ACTIONS OF MODULAR REDUCTION FINITE STATE MACHINE

State	Actions
$S_0(S=0)$	$S := 1; A := A_0; N := N_0$
$S_1(S=1)$	$S := 2; C := lz(N) - lz(A); D := C$
$S_2(S=2)$	if $(D < 0)$ $\{S := 4\}$ if $(D > 0)$ $\{N := N \ll 1; D := D - 1\}$ if $(D = 0)$ $\{S := 3\}$
$S_3(S=3)$	if $(C \geq 0 \wedge A \geq 0)$ $\{A := A - N; N := N \gg 1; C := C - 1\}$ if $(C \geq 0 \wedge A < 0)$ $\{A := A + N; N := N \gg 1; C := C - 1\}$ if $(C = 0 \wedge A \geq 0)$ $\{A := A - N; C := C - 1\}$ if $(C = 0 \wedge A < 0)$ $\{A := A + N; C := C - 1\}$ if $(C < 0 \wedge A \geq 0)$ $\{S := 4\}$ if $(C < 0 \wedge A < 0)$ $\{S := 4; A := A + N\}$

S_2 **:** If $D < 0$, N is larger than A and the FSM directly goes to the final state S_4. Otherwise, the FSM left-shifts N by one bit and remains in the same state. The FSM makes self-loop transitions $D = lz(N) - lz(A)$ times, and then it goes to the state S_3.

S_3 **:** The FSM remains in this state for $C + 1 = lz(N) - lz(A)+1$ iterations. It subtracts or adds N to A depending on the sign of A. It also shifts N to the right by one bit, except for the last iteration. Finally, it adds N if A is negative, and moves to the final state S_4.

S_4 **:** The register A stores the final answer of $A_0 \bmod N_0$.

Although this description is considerably simpler than the optimized hardware implementation, it is sufficient to explain our verification approach in later sections. The modular reduction engine is implemented to accept input operands of different data widths. All the arithmetic operations on A and N, such as bit vector addition, subtraction, shifting and leading zero counting, are performed as per the size of input operands. The FSM implements the variable-size operations by iterating fixed-size arithmetic operations. For example, 65-bit adders are used to implement variable-size bit-vector addition up to 4096-bits. As a result, what appears to be a simple state transition is in fact iterative operations on fixed data width over many clock cycles.

Current guidelines for the use of the RSA algorithm for public key encryption in commercial applications call for the use of 1024 bit or 2048 bit keys [11]. The modular reduction operation used in this algorithm takes a number of clock cycles proportional to the square of the input data width divided by the size of the fixed-width processing and storage elements in the implementation. Even a single 512-bit computation will, therefore, take several thousand clock cycles to execute, while the number of possible input operand pairs is 2^{1024}. This makes it very difficult to verify the entire range of interesting cases using simulation.

III. ACL2SIX HYBRID VERIFICATION SYSTEM

The ACL2SIX verification system is a combination of the open-source theorem prover ACL2 [12], [13] and the IBM verification tool SixthSense [14]. An early version of the system has been reported in [15], and its application in [16]. As this system has been significantly modified since it was first reported, we outline the salient features of the enhanced hybrid environment.

The main philosophy of this hybrid tool is a divide-and-conquer approach for the verification problem. When we want to verify a property which cannot be verified by an automated model checker, we decompose it into a number of easier sub-problems, solve them one-by-one, and combine the results together. Each sub-problem is thus solved by a model checker, while the results are combined by a theorem prover. However, when the verification problem is decomposed into too many small problems, the burden of recombination via the theorem proving becomes rather high, and the proof may become labor intensive. Thus, it is critical to contain the degree of decomposition using a powerful model checker to scale to as large of sub-problems as possible.

An overview of the ACL2SIX system is shown in Figure 2. Suppose a user attempts to verify certain properties on a *design under test* (*DUT*). A DUT is usually a complex RTL hardware design written in VHDL or Verilog. A verification driver defines the environment in which the DUT operates, e.g. clocking conditions and other input constraints. In a typical setting, the verification driver may assert the reset signal at the beginning of the test, and then initiate the operation of the machine with non-deterministic data inputs. A verification driver is usually written in VHDL or some *synthesizable* language. As we discuss later, the verification driver is also used to help writing invariant conditions succinctly in the ACL2 language.

When a user attempts to check if a certain property holds using the ACL2SIX system, he/she writes the property in a small subset of the ACL2 theorem prover language. When invoked, ACL2 first compiles the property to a *property checker*. A property checker is a synthesized automata for the desired property, effectively a small state machine which asserts a particular gate to a logical '1' when the property holds. SixthSense then composes the DUT, the verification driver, and the property checker, and checks whether the property checker always evaluates to '1' for all input sequences. When the verification is successful, the property is saved in ACL2 as a theorem and may be used for future proofs. If the check fails, SixthSense produces an counterexample trace to assist the user in determining why the property does not hold.

Since ACL2 is a general-purpose theorem prover, its language is too expressive to be translated into HDL. Instead, the ACL2SIX system allows only a subset of the ACL2 language for specifying properties to be verified. The subset is rich enough to write various properties to prove the correctness of the DUT, and the translation of the properties does not cause any semantic inconsistency between this ACL2 language

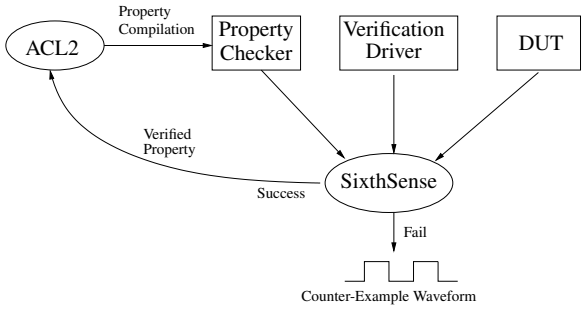

Fig. 2. Overall Data Flow for Property Check in ACL2SIX

TABLE II
EXAMPLE ACL2SIX PRE-DEFINED FUNCTIONS

Function Name	Brief Description
`(bv i j)`	Bit vector of value i and length j
`(b1p b)`	True if $b = 1$, false if $b = 0$
`(bv+ v1 v2)`	Sum of two bit vectors
`(bv- v1 v2)`	Difference of two bit vectors
`(bv-sll v n)`	Logical left shift of vector v by n bits
`(bv-srl v n)`	Logical right shift of vector v by n bits
`(bv-lz v i)`	Vector of length i counting leading zeros of v

subset and VHDL.

The language for ACL2SIX has four data types: bits, bit vectors, Boolean values and natural numbers. All properties must be written in terms of ACL2SIX pre-defined functions, under which those types are closed. The user may also specify user-defined non-recursive functions. However, these functions must also be defined in terms of those pre-defined functions. Additionally, the user-defined functions must carry *type* information using the ACL2 guard mechanism [17], so that the translation process can infer types of expressions. Table II lists some of the ACL2SIX pre-defined functions.

The signal values in the DUT and the driver can be referenced by the following terms:

```
(vhdl-sigbit m sig n)
(vhdl-sigvec m sig (i j) n)
```

`(vhdl-sigbit` m sig n`)` is used to reference the value of bit sig in hardware model m at clock cycle n. Similarly, `(vhdl-sigvec` m sig `(`i j`)` n`)` refers to the bit range i to j of bit vector sig at cycle n. In these terms, sig is the name of a signal, and i, j and n are natural numbers. Model m is a list structure, from which we can infer the DUT, the verification driver, and other parameters needed to set up verification of the DUT. For example, the model for an adder may be simply defined as:

```
(defun adder ()
  '("adder"
    :driver "adder_dr.vhdl"))
```

where `"adder"` is the VHDL entity name of the adder and `"adder_dr.vhdl"` is a verification driver name. Other information such as the path to the VHDL file, or how the DUT is initialized may be added to the model definition.

The main idea behind the use of `vhdl-sigbit` and `vhdl-sigvec` is that they logically reference the signal values of the DUT, but they do not actually compute the values. In a system that fully embeds an HDL, a hardware model would be a translated HDL and signal values would be defined by its interpreter. In ACL2SIX, the model is just a stub to access the DUT written in HDL, and signal values are only defined using *constraint functions*. Specifically, both `vhdl-sigbit` and `vhdl-sigvec` are ACL2 macros defined in terms of ACL2 encapsulated functions `sigbit` and `sigvec`. An ACL2 encapsulated function is a mechanism to define an uninterpreted function with some constraints. We can infer types of the value returned by `sigbit` and `sigvec`, but its value is uninterpreted, and can be inferred only by calling SixthSense through the ACL2SIX system.

A typical ACL2 theorem definition to invoke SixthSense property checking has the following syntax:

```
(defthm name
  (implies type-info expr)
  :hints (("goal" :clause-processor
                  ( :function acl2six
                    :hints acl2six-args))))
```

In this definition, *name* is the name of the theorem, *type-info* is the type information for the free variables in *expr*, and *expr* is a property expression which is defined in terms of the ACL2SIX pre-defined functions. The ACL2 hint provided after keyword `:hints` usually tells the theorem prover how to prove a theorem, and in this case, it invokes a clause processor function `acl2six`. A clause processor is an ACL2 mechanism to implement an extension of the prover. It allows a user-defined function to simplify or even prove a logical expression. It may also work as an interface with other verification tools. When invoked through the clause processor mechanism, function `acl2six` translates *expr* to a property checker implemented in VHDL, runs SixthSense, and records successfully verified properties as theorems. A call to `acl2six` can be accompanied by additional arguments *acl2six-args*, with which the user can control SixthSense and specify types of algorithms to verify the property.

One important limitation of the ACL2SIX property compilation is that the verified property should be defined in terms of signals with fixed timing delays. For example, in order to check the output `"SUM"` of a two-stage 32-bit adder , we can evaluate the following ACL2 term:

```
(defthm adder-output
 (implies (natp n)
  (equal (vhdl-sigvec (addr) "SUM" (0 31) (+ n 2))
         (bv+ (vhdl-sigvec (addr) "A" (0 31) n)
              (vhdl-sigvec (addr) "B" (0 31) n))))
  :hints (("goal" :clause-processor
                  (:function acl2six
                   :hint '((:cycle-var n))))))
```

This property check compares the value of vector `"SUM"` at cycle $n+2$ with the summation of two vectors `"A"` and `"B"` at cycle n, where n is an arbitrary natural number. The cycle delay `2` in cycle expression `(+ n 2)` should be a constant,

and cannot be replaced with a variable or a complex expression. In terms of LTL, [18], we can only check a formula of the form $G(expr)$ where $expr$ is a formula written only with X operators. While we are generally interested in *unbounded* model checking of the underlying design where reasoning about specific clock cycles may seem to contradict this goal, we use this style of reasoning in an *inductive* framework where state machines are evaluated relative to arbitrary states which adhere to established invariants, vs. evaluating only relative to initial states.

IV. Verification of a Modular Reduction Engine

A. General Approach to Verifying a Finite State Machine

An FSM (such as that described in Section II) can be verified by a hybrid system such as ACL2SIX by first verifying every state transition using a model checker, and then combining the results using a theorem prover.

For example, let us consider an FSM that makes a state transition sequence of $S_0, S_1, S_2, \ldots, S_n$. Each state transition may take a number of clock cycles, and we assume that the transition from state S_i to state S_{i+1} takes Δ_i cycles. Each state S_i has a corresponding property P_i that must hold. For the state transition from S_i to S_{i+1}, P_i is the pre-condition and P_{i+1} is the post-condition. Let us write $P_i\{S_i\}$ to indicate that property P_i holds for state S_i. If we can verify $P_0\{S_0\}$ and $P_i\{S_i\} \Rightarrow P_{i+1}\{S_{i+1}\}$ for all $i < n$, it is straightforward to prove $P_n\{S_n\}$ using a theorem prover. In this way, we can verify the machine correctness specified by P_n. We may define P_n to specify, for example, that the final answer of the machine is correct.

Thus the verification problem is reduced to the verification of $P_i\{S_i\} \Rightarrow P_{i+1}\{S_{i+1}\}$ for each i. Let us write $P_i(n)$ to indicate that P_i holds at clock cycle n. Since the transition from S_i to S_{i+1} takes Δ_i cycles, proving

$$P_i(n) \Rightarrow P_{i+1}(n + \Delta_i) \tag{1}$$

for all n will be sufficient. Let us further define $Q_i(n) = \big(P_i(n) \Rightarrow P_{i+1}(n + \Delta_i)\big)$. The ACL2SIX system and SixthSense use the following steps to verify $\forall n.Q_i(n)$

1) Convert $Q_i(n)$ to a circuit using logical gates and latches. Since $P_i(n)$ can be represented as a combinational circuit, we can latch the value of $P_i(n)$ for Δ_i-cycles, and then check that the latched value of P_i implies P_{i+1}.
2) Simplify the circuit representation of $Q(n)$ using a number of circuit reduction techniques, such as constant propagation, combinational and sequential simplifications [19], retiming [20], phase abstraction [21] and transient logic elimination [22]. This reduction itself may reduce $Q_i(n)$ to a tautology, in which case $Q_i(n)$ is proven and we stop. Otherwise, we go to the next step.
3) Prove $Q_i(n)$ by k-induction. This is done by proving the base cases $Q_i(0), Q_i(1), \ldots, Q_i(k-1)$ and the induction step $Q_i(n) \wedge Q_i(n+1) \wedge \cdots \wedge Q_i(n+k-1) \Rightarrow Q_i(n+k)$. This is attempted for ever-increasing values of k until either $Q_i(n)$ is proved or computational resources are exhausted.

TABLE III
Pre and post-conditions of State Transitions of the Modular Reduction Engine

Transition	Pre-condition	Post-Condition
S_0 to S_1	$S = 0$	$S' = 1 \wedge A' = A_0 \wedge N' = N_0$
S_1 to S_2	$S = 1$	$S' = 2 \wedge C' = lz(N) - lz(A) \wedge$ $D' = C' \wedge A' = A \wedge N' = N$
S_2 to S_2	$S = 2 \wedge D > 0$	$S' = 2 \wedge N' = N \ll 1 \wedge$ $D' = D - 1 \wedge C' = C \wedge$ $A' = A$
S_2 to S_3	$S = 2 \wedge D = 0$	$S' = 3 \wedge N' = N \wedge A' = A$
S_2 to S_4	$S = 2 \wedge D < 0$	$S' = 4 \wedge A' = A$
S_3 to S_3	$S = 3 \wedge C > 0 \wedge$ $A \geq 0$	$S' = 3 \wedge A' = A - N \wedge$ $N' = N \gg 1 \wedge C = C - 1$
S_3 to S_3	$S = 3 \wedge C > 0 \wedge$ $A < 0$	$S' = 3 \wedge A' = A + N \wedge$ $N' = N \gg 1 \wedge C = C - 1$
S_3 to S_3	$S = 3 \wedge C = 0 \wedge$ $A \geq 0$	$S' = 3 \wedge A' = A - N \wedge$ $N' = N \wedge C = C - 1$
S_3 to S_3	$S = 3 \wedge C = 0 \wedge$ $A < 0$	$S' = 3 \wedge A' = A + N$ $N' = N \wedge C = C - 1$
S_3 to S_4	$S = 3 \wedge C < 0 \wedge$ $A \geq 0$	$S' = 4 \wedge A' = A$
S_3 to S_4	$S = 3 \wedge C < 0 \wedge$ $A < 0$	$S' = 4 \wedge A' = A + N$

The ACL2SIX system and SixthSense are highly configurable, and so we could use any other model checking algorithms to verify $Q_i(n)$. However, we found that logic reductions followed by k-induction work well for the verification of many properties of our modular reduction engine.

B. Verification of a Modular Reduction Engine

Here we discuss the use of ACL2SIX to verify the modular reduction engine. Table III shows the list of pre-conditions and post-conditions for each state transition, as per Figure 1. For any symbol X, let X' represent its value after the state transition. The table therefore shows how symbol values change when state transitions occur. If we can verify that the pre-condition implies the post-condition for all possible state transitions, we can use a theorem prover to show that the value of register A is $A_0 \bmod N_0$ when state S_4 is reached. In other words, the FSM correctness is the logical consequence of this set of pre-condition and post-condition pairs.

We can represent the pre-condition and post-condition relation using the supported language of ACL2SIX. While the number of clock cycles between FSM state transitions is generally a function of the data width, for a given data width of input A_0 and N_0 each state transition requires a fixed number of clock cycles. Our approach is to verify the operational correctness for each input data width separately. Then, the

relation of the pre and post-conditions can be written using the ACL2SIX language, which requires that each delay be a fixed constant. We define the delay of the state transition parametrically, so that we can rerun the same proof script to re-verify the modular reduction engine for different input data widths by just changing parameters.

When actually writing an ACL2 theorem representing the conditions in Table III, additions $+$, subtractions $-$, shifting $\ll$, $\gg$, and leading zero counting lz are specified using the pre-defined functions given in Table II. As briefly discussed in Section II, the hardware implements the arithmetic operations of long bit vectors by repeatedly applying 65-bit arithmetic operations. For example, 512-bit addition is performed by repeating 65-bit additions 8 times over tens of clock cycles. However, such hardware implementation details should be automatically verified and hidden from the ACL2 proof level. In fact, our proof script simply specifies such an addition as the sum of two long and continuous bit vectors. In this way we simplify the to-be-proven theorems as much as possible. This requires the underlying model checker such as SixthSense to do the heavy lifting of verifying high-level specifications against intricate implementation artifacts.

The abstraction of bit-level details allows the pre-conditions and post-conditions (Table III) to be described concisely at a high-level. However, simply attempting to verify "pre-condition implies post-condition" frequently fails because the hardware often requires additional conditions to operate properly. For example, the hardware goes through an initialization phase that sets up the clock buffers, the hardware control logic and other components for proper operations. The hardware is designed to operates properly only *after* such initialization, relying upon post-initialization reachable state invariants. Let us define such a global invariant as $\text{inv}(n)$. Additionally, there might be other reachability invariants that holds when the machine is at state S_i but not captured in the the conditions described in Table III. Let such a state invariant be denoted as $\text{cond}_i(n)$. Then it is sufficient to verify:

$$(\text{inv}(n) \wedge \text{cond}_i(n) \wedge P_i(n)) \Rightarrow P_{i+1}(n + \Delta_i) \qquad (2)$$

for all the state transitions, instead of Equation 1. Separately, we need to verify that the global invariant condition is in fact an invariant by:

$$\text{inv}(n) \Rightarrow \text{inv}(n+1) \qquad (3)$$

and the state invariant condition is satisfied at each state by:

$$(\text{inv}(n) \wedge \text{cond}_i(n)) \Rightarrow \text{cond}_{i+1}(n + \Delta_i). \qquad (4)$$

In our approach, we define the global and state invariants in the verification driver in Figure 2. For example, the global invariant inv may be defined as a VHDL signal `"DRIVER.INV"` in the verification driver which represents the conjunction of numerous invariant conditions. In ACL2SIX, we can refer to this global invariant at any time n as `(vhdl-sigbit (modred) "DRIVER.INV" n)`. In this way, we keep the hardware-dependent and sometimes tedious definition of invariant conditions out of the proof script.

Finding the proper global invariant inv and state invariant cond_i is the most critical task for the entire verification methodology. This is usually done by repeated attempts to verify formula 2, 3 and 4, analyzing failed verification results by viewing generated counterexample waveforms, and iteratively tightening the invariants until the proof is successfully completed.

Some simple invariant conditions are automatically deduced during the 3-step verification algorithm discussed in Subsection IV-A. For example, circuit reduction algorithms in step 2) may simplify the design by merging redundant gates, or performing other property-preserving temporal abstractions. Such transformations are critical to simplify the manual effort of deriving invariants; in a sense, such transformations automate the derivation of a subset of design invariants. For example, if two latches are merged since they always evaluate to the same value, this rules out a possible induction counterexample where they exhibit differing values. Similarly, k-induction with a larger value of k tends to prove more properties without manually specifying some invariants. Thus, the more powerful the underlying bit-level model checker is, the less the verification engineer must manually specify the invariant conditions.

Once all the post-conditions are verified from pre-conditions, the theorem-prover is used to deduce the correctness proof of the hardware operation as a logical consequence of all the verified properties. During theorem proving, it is critical to analyze state loops. This is usually carried out by specifying loop invariants, verifying them by induction, and using them to deduce the termination condition. For example, in the i'th iteration of S_2 of our modular reduction finite state machine, the following loop invariant should hold.

$$\begin{aligned}&(A = A_0) \wedge (N = N_0 \ll i)\\ &\quad \wedge \big(C = lz(N_0) - lz(A_0)\big) \wedge (D = C - i)\end{aligned}$$

The state loop at S_3 satisfies a slightly more complicated loop invariant, with inequality $-2N \leq A < 2N$ being true except during the last iteration of S_3. This condition is critical for proving the correctness of the final answer.

At this high-level analysis of loop invariants, the theorem proving task is no different from a pure theorem proving verification approach. However, a pure theorem proving approach typically requires significant effort in verifying the low-level implementation of hardware. We can instead accelerate the process using the automated model checker to reason about intricate implementation details, and let the theorem prover focus on the algorithmic level. This leverages the orthogonal strengths of theorem proving and model checking: theorem proving becomes more robust as details of the hardware implementation are abstracted away, and model checking becomes more robust as it focuses on a specific small function of a large sequential machine.

C. Counter-Example Generation

ACL2SIX relies upon SixthSense to unboundedly verify a set of properties, inasmuch as those properties represent

temporally-bounded pre-condition to post-condition checks. SixthSense will produce one of the following three answers: 1) the property fails relative to specified initial states; 2) the property passes; 3) the property is unsolved given the specified set of algorithms.

When using induction as the core proof technique, properties may often be reported as unsolved even if they truly hold in all reachable states. This is a byproduct of the weakness of induction: an induction counterexample due to a transition from a passing to a failing state render the inductive check inconclusive, yet it is not known whether the inductive starting state is reachable or not. If relying upon induction as a proof technique, it is necessary for a verification engineer to analyze the induction counterexample to derive invariants which rule out that counterexample.

In the course of this verification effort, SixthSense was enhanced to produce *induction counterexample traces* for analysis by the verification engineer. SixthSense is based on the concept of transformation-based verification [20], where synergistic algorithms are applied to simplify large problems into smaller problems before applying a core proof technique. These simplifications include logic rewriting techniques [19], phase abstraction [21], redundancy removal [23], and transient logic elimination [22]. Such simplifications often considerably reduce verification resources for the core proof technique, and often considerably improve the effectiveness of induction since they rule out possible induction counterexamples where the reduced behavior does not hold. Without such reductions, the manual effort to derive such invariants often becomes infeasible given a significant amount of design artifacts.

When SixthSense generates a counterexample trace after such simplifications, that trace must be "lifted" to undo the effects of those transformations before it can be presented to the user. For traditional counterexamples, this process is straightforward as only *input* valuations need to be accounted for, allowing a top-level simulation to be used relative to this *test case* to derive values to all signals. When lifting an induction counterexample, the set of valuations to be accounted for include those of the state elements in the inductive starting state. It is further noteworthy that such counterexamples should be minimally-assigned, to improve the identification of the root cause of the induction failure.

SixthSense required several customizations to support induction trace generation. For transformation engines which may merge redundant gates, bookkeeping was added reflecting such transformations so that it may be back-annotated in a lifted trace, without which the induction trace may not truly reflect an induction counterexample. Additionally, some transformations performed by SixthSense are not themselves inductively provable, requiring more intricate unreachable-state invariants. When leveraging such reductions, we found it necessary to pass the automatically-derived unreachable-state invariants to the induction process along with the reduced design, to avoid it from rendering induction counterexamples which had no counterpart in the pre-reduced design.

TABLE IV
TIME AND MEMORY REQUIRED FOR VERIFYING MODULAR REDUCTION

Data Width	56-bit	256-bit	384-bit	512-bit
Total Time	10442s	20646s	37607s	98199s
Theorem Prover Time	257s	289s	474s	1690s
Property Check Time	10188s	20261s	37139s	97012s
Avg. Time per Prop.	118s	151s	223s	489s
Max Time per Prop.	138s	368s	1232s	3456s
Avg. Mem. per Prop.	1195MB	1459MB	1967MB	2719MB
Max Mem. per Prop.	1393MB	4201MB	5680MB	8571MB

D. Verification Results

With the approach discussed in the previous subsections, we have verified the mathematical correctness of the modular reduction engine for input data widths of 56-bits, 192-bits, 256-bit, 384-bits and 512-bits. In addition to verifying simple modular reduction, we also verified modular addition, modular subtraction and modular negation. Table IV shows the time and memory required to verify all four of these operations using a 2.27GHz Intel Xeon X7560 processor running Linux 2.6.18. The number of properties verified by invoking SixthSense varied from 86 for the 56-bit operation to 198 for the 512-bit operation. For the 1024-bit and larger input data widths, some properties could not be proven by SixthSense in 24 hours, and we did not complete the verification.

The verification process requires several iterations to attempt to inductively prove the properties. An initial property check almost always fails, causing SixthSense to produce induction counterexamples. The examination of the counterexample often reveals that the state invariants are not strong enough to constrain the hardware to behave correctly. This leads to manual strengthening of the invariants to help the verification process converge. The proof scripts are written parametrically, so that the verification for different bit widths goes through automatically, or with little human guidance.

The total labor time is difficult to measure scientifically, as it depends on numerous factors. Roughly speaking, one engineer finished the verification of 56-bit modular reduction in a few weeks. Then, two engineers spent several months to extend the results to various operations including modular add, subtract and negation operations of various data widths, while working on this part-time. Roughly equal amount of time was spent on invariant property checking and theorem proving. However, this could change significantly depending on how the verification problem is decomposed.

During the course of this effort, an engineer with a background in the VHDL and LISP languages was readily able to learn the ACL2SIX system to specify and debug invariants. However, the use of theorem proving beyond trivial proofs, such as case splitting, has a steeper learning curve.

A similar approach has been applied to the modular inverse operation implemented in the same modular reduction engine. Given operands A and N, it obtains a number X such that $(A \times X) \bmod N = 1$. The hardware uses a binary extended

Euclid algorithm [24] to calculate the number. We quickly identified that the operation may overflow out of fixed-size registers. Even the original algorithm description in [24] failed to warn that there is a danger of overflow. The DUT had a 4-bit head-room for 256-bit modular inverse calculation and 6-bits for 384-bit operation. In other words, the intermediate value can be 16 times or 64 times the maximal input values, respectively, and designers believed this to be sufficient to avoid an overflow. Using bounded model checking, we identified combinations of A and N which overflow the registers. This event is extremely rare, and neither a random simulation nor post-silicon testing could have identified such A and N. Designers added an hardware overflow check and software support to correct the problem.

V. Conclusion

In this paper, we have verified an industrial modular reduction engine implemented in the cryptographic function accelerator. We have successfully verified the mathematical correctness of the modular reduction engine upto 512-bit input data width. This is beyond what can be formally verified by either a stand-alone model checker or a theorem prover. We also applied this approach to the modular addition, modular subtraction, and modular negation operations, and verified their correctness.

We have found the hybrid verification technique using ACL2SIX an extremely powerful tool to analyze hardware accelerators implementing finite state machines. Our formal verification approach should not be viewed as an alternative to random simulation. Rather we provide an additional capability to verify systems that typical random simulation approaches fail to verify due to the sheer size of the input domain and the length of simulation cycles. We are currently working on Montgomery multiplication and exponentiation, which are yet another important subroutine of encryption accelerators.

An important trick to successful hybrid verification is to decompose the correctness problem into sub-problems of the right size. If the problem is decomposed into too many sub-problems, the theorem proving becomes time-consuming. If the problem is decomposed into larger sub-problems, the model checking fails to discharge them.

Our hybrid verification tool is already quite powerful, but it still has room for improvement. The verification of wider modular calculations, such as with data width of 4092-bits, have not been completed yet because certain state transition property checks are beyond the tool's capability. Also the theorem proving using ACL2 takes some expertise and time, even if the underlying automatic property check by SixthSense has significantly removed the burden. Further improvements are in progress to alleviate these issues.

Acknowledgment

We thank Bart Blaner and Ross Leavens at IBM for their constant feedback for the direction of our research, and on the design of the AMF engine.

References

[1] S. Rajan, N. Shankar, and M. Srivas, "An integration of model-checking with automated proof checking," in *CAV '95*, ser. Lecture Notes in Computer Science, P. Wolper, Ed., vol. 939. Liege, Belgium: Springer-Verlag, jun 1995, pp. 84–97.

[2] J. O'Leary, X. Zhao, R. Gerth, and C.-J. H. Seger, "Formally verifying IEEE compliance of floating-point hardware," *Intel Technology Journal*, vol. Q1, Feb. 1999.

[3] R. Kaivola and M. Aagaard, "Divider circuit verification with model checking and theorem proving," in *Proceedings of the 13th International Conference on Theorem Proving in Higher Order Logics*, ser. TPHOLs '00. London, UK: Springer-Verlag, 2000, pp. 338–355.

[4] A. Slobodová, *Challenges for Formal Verification in Industrial Setting*, ser. Lecture Notes in Computer Science. Springer, 2007, vol. 4346, pp. 1–22.

[5] ——, "Formal verification of hardware support for advanced encryption standard," in *FMCAD*. IEEE Press, 2008, pp. 1–4.

[6] L. Erkök, M. Carlsson, and A. Wick, "Hardware/software co-verification of cryptographic algorithms using cryptol," in *FMCAD*, 2009, pp. 188–191.

[7] E. W. Smith and D. L. Dill, "Automatic formal verification of block cipher implementations," in *FMCAD*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 6:1–6:7.

[8] D. Borrione and P. Georgelin, "Formal verification of vhdl using VHDL-like ACL2 models," in *In Forum on Design Languages (FDL)*, 1999.

[9] 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," *London Mathematical Society Journal of Computation and Mathematics*, vol. 1, pp. 148–200, December 1998.

[10] W. A. Hunt and E. Reeber, "Formalization of the DE2 language," in *Proceedings of the 13th Working Conference on Correct Hardware Design and Verification Methods (CHARME 2005), LNCS*. Springer-Verlag, 2005, pp. 20–34.

[11] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid, "Recommendation for key management part 1: General," in *NIST Special Publication 800-57, August 2005, National Institute of Standards and Technology. Available at http://csrc.nist.gov/publications/nistpubs/800-57/SP800-57-Part1.pdf*, 2005.

[12] M. Kaufmann and J. S. Moore, "An industrial strength theorem prover for a logic based on common lisp," *IEEE Transactions on Software Engineering*, vol. 23, no. 4, pp. 203–213, apr 1997.

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

[14] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann, "Scalable automated verification via expert-system guided transformations," in *FMCAD*, 2004, pp. 159–173.

[15] J. Sawada and E. Reeber, "ACL2SIX: A hint used to integrate a theorem prover and an automated verification tool," in *FMCAD*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 161–170.

[16] J. Sawada, "Automatic verification of estimate functions with polynomials of bounded functions," in *FMCAD*, 2010, pp. 151–158.

[17] M. Kaufmann and J. S. Moore, "ACL2 user's manual," See URL http://www.cs.utexas.edu/users/moore/acl2/acl2-doc.html#User's-Manual.

[18] A. Pnueli, "The temporal logic of programs," in *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*. Washington, DC, USA: IEEE Computer Society, 1977, pp. 46–57.

[19] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting: A fresh look at combinational logic synthesis," in *DAC*, 2006.

[20] A. Kuehlmann and J. Baumgartner, "Transformation-based verification using generalized retiming," in *CAV*, July 2001.

[21] P. Bjesse and J. Kukula, "Automatic generalized phase abstraction for formal verification," in *ICCAD*, Nov. 2005.

[22] M. Case, H. Mony, J. Baumgartner, and R. Kanzelman, "Enhanced verification through temporal decomposition," in *FMCAD*, Nov. 2009.

[23] C. A. J. van Eijk, "Sequential equivalence checking without state space traversal," in *DATE*, Feb. 1998.

[24] A. J. Menezes, S. A. Vanstone, and P. C. V. Oorschot, *Handbook of Applied Cryptography*, 1st ed. Boca Raton, FL, USA: CRC Press, Inc., 1996.