# Proof-Carrying Hardware Intellectual Property: A Pathway to Trusted Module Acquisition

Eric Love, *Member, IEEE*, Yier Jin, *Student Member, IEEE*, and Yiorgos Makris, *Senior Member, IEEE*

***Abstract*—We present a novel framework for facilitating the acquisition of provably trustworthy hardware intellectual property (IP). The proposed framework draws upon research in the field of proof-carrying code (PCC) to allow for formal yet computationally straightforward validation of security-related properties by the IP consumer. These security-related properties, agreed upon *a priori* by the IP vendor and consumer and codified in a temporal logic, outline the boundaries of trusted operation, without necessarily specifying the exact IP functionality. A formal proof of these properties is then crafted by the vendor and presented to the consumer alongside the hardware IP. The consumer, in turn, can easily and automatically check the correctness of the proof and, thereby, validate compliance of the hardware IP with the agreed-upon properties. We implement the proposed framework using a synthesizable subset of Verilog and a series of pertinent definitions in the Coq theorem-proving language. Finally, we demonstrate the application of this framework on a simple IP acquisition scenario, including specification of security-related properties, Verilog code for two alternative circuit implementations, as well as proofs of their security compliance.**

***Index Terms*—Hardware intellectual property (IP), hardware security, proof-carrying code (PCC), proof-carrying hardware (PCH), trusted integrated circuit.**

## I. Introduction

THE problem of hardware security has grown more important and more difficult with the emergence of an increasingly globalized design process. The tight control manufacturers once exerted over their devices is no longer possible when more complicated systems now employ hardware components from a variety of different suppliers whose trustworthiness is unknown [1], [2]. Researchers have, accordingly, devised techniques to ensure trusted designs by diffusing the threat of malicious circuitry (a.k.a. hardware Trojans) being inserted into the supply chain, relying variously on physical, behavioral, and formal methods both in postsilicon designs [3]–[10] and presilicon designs [11], [12]. The key assumption behind these methods is that any trusted designs should be exactly the same as golden models and the mismatches between designs-under-test (DUTs) and golden models will be used to differentiate Trojan-infested chips from genuine chips.

However, our scheme is different from all previous approaches to the domain of trusted designs in two respects. First, we focus on the security of third-party Intellectual Property (3PIP) modules which are commonly used in contemporary designs. Second, no golden models are required in our methodology while security properties, which we will discuss later, are developed to ensure the trustworthiness of IP modules. We imagine an attacker who makes malicious modifications to a module's HDL code in order to introduce the potential for undesired behavior. This module may then be sold for use in a larger system which, with the inclusion of tampered IP, becomes itself vulnerable to attack.

If, however, we can guarantee that certain carefully specified properties hold across the outputs of components from untrusted IP vendors, then we may be able to guard against certain types of undesirable or insecure behavior. These can include disruption of operation, manipulation of signals, or misuse of sensitive data. Each case requires different kinds of properties, but a strong specification can render many modes of attack significantly more difficult to implement. If these safeguards become integrated into the design process, then when an IP consumer asks for some module to be constructed, he will provide the vendor with not only a functional specification, but also a list of specific security-related properties that the desired module must obey. It is then the vendor's task to construct a formal proof demonstrating adherence to these properties.

The goal of this paper is thus to propose a new IP acquisition and delivery protocol which can help IP consumers to quickly validate the trustworthiness of 3PIP they purchased from IP vendors. Fig. 1 outlines our proposed interaction between IP vendors and consumers.

A similar idea has been proposed for software as proof-carrying code (PCC) [13]. In its original form, PCC required the acceptance of a large, unverified code base at its core. As a solution to this problem, researchers have developed foundational PCC (FPCC) which uses a universal logic framework to model the semantics of all possible assembly language instructions and is written in the same logical inference language used to write correctness proofs, thereby subjecting the entire system to validation by the proof checker [14]–[16]. Further work has led to the creation of a certified assembly programming language

Manuscript received January 20, 2011; revised May 10, 2011; accepted June 13, 2011. Date of publication June 27, 2011; date of current version January 13, 2012. The work of E. Love was supported by the National Science Foundation under a Research Experience for Undergraduates (REU) Grant, NSF 1132205. The work of Y. Jin and Y. Makris was supported by the National Science Foundation under Grant NSF 1017719. The associate editor coordinating the review of this manuscript and approving it for publication was Dr. Kouichi Itoh.

E. Love is with the Department of Electrical Engineering and Computer Science, University of California at Berkeley, Berkeley, CA 94720 USA (e-mail: ericlove@eecs.berkeley.edu).

Y. Jin is with the Department of Electrical Engineering, Yale University, New Haven, CT 06520 USA (e-mail: yier.jin@yale.edu).

Y. Makris is with the Department of Electrical Engineering, The University of Texas at Dallas, Richardson, TX 75080 USA (e-mail: yiorgos.makris@utdallas.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TIFS.2011.2160627

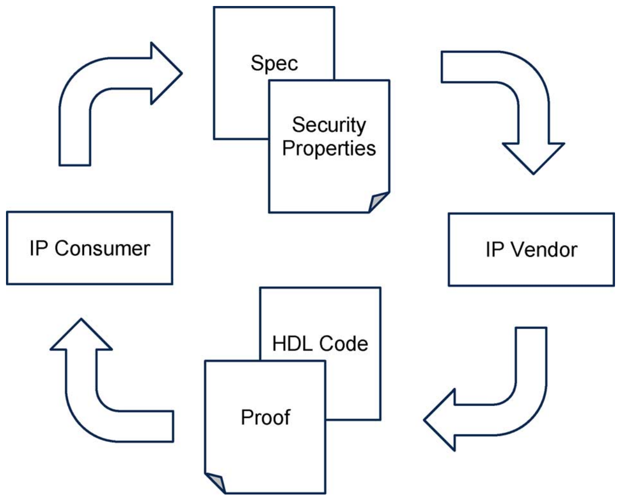

Fig. 1. IP acquisition and delivery protocol.

(CAP) [17], [18], upon whose construction and application we model our reformulation of PCC for use with hardware IP.

A parallel concept of proof-carrying hardware (PCH) was first proposed in [19], but the authors showed only that correctness proofs could be generated for FPGA bit streams in order to provide assurance that the given gate configuration implements a specific boolean logic function, and therefore they did not allow for true functional variation. Furthermore, their method relied on an SAT solver rather than a formal high-level proof assistant tool, and thus more closely resembles formal verification than PCC. We, however, shift our analysis from post-synthesis FPGA bit streams up to presynthesis register-transfer level (RTL) IP cores, expanding the domain of provable specifications to include more complicated behavioral properties given in a temporal logic, achieving for hardware the same level of flexibility offered for software by PCC.

While there is no clear analogue to the machine assembly language in the domain of hardware circuit representations, we feel that our choice of RTL IP as the semantic model for proof construction is nevertheless quite sensible. Much hardware is now designed and shipped in this form, and the widespread use of HDL-coded circuits in FPGA applications means that demand is high for methods of managing the rapid deployment of such IP. When designing for reconfigurability, a means of instantaneously verifying the trustworthiness of newly acquired hardware is certainly advantageous.

In our system, proofs are written in the Coq proof assistant language and are, therefore, easy to validate automatically, allowing the consumer to know very quickly whether or not the HDL code conforms to a given set of security-related properties.[1] Just as with PCC and PCH, the computational burden of verification falls on the IP vendor, not the consumer. The vendor must make a significant investment of time in the construction of a proof, but the consumer's task of verifying it is trivial in comparison. Of course, this is not to say that the task of constructing security compliance proofs is necessarily onerous. Many software PCC systems are now beginning to support automated proof writing, and there is no reason to believe the same could not be done for hardware [20].

One novel contribution is to create a set of definitions in the Coq [21] language (Section IV) that models the behavior of all possible statements in a domain-specific Verilog we specify in Section III. We also describe, in Sections IV and V, a set of rules to automatically generate the Coq representation of any given Verilog module for use in security compliance proofs. We then illustrate the usefulness of our framework by way of a contrived design scenario in Section VI. We present a model consumer with need for a specific component and imagine what sort of security requirements this consumer would have. The example covers formulation of security properties, translation into the temporal logic model we have implemented in Coq, sample HDL implementations, and the construction of proofs.

## II. Design Process and Utility of PCH IP

If the consumer wishes to order a component from the IP vendor, our design framework requires that he decide upon a set of security properties in addition to the standard functional specification. Both parties must then agree upon a fixed translation of these properties into a formal mathematical codification in the theorem-proving language. As the vendor writes HDL code for the final product he also produces a formal proof as shown in Fig. 1. This is not a type of testing procedure, but rather a new stage of the hardware design process to be carried out in addition to standard verification and debugging. Although we will see that the temporal logic used to specify security-related properties does resemble the syntax of many hardware assertion languages, the verification of these properties is not an assertion-based process. It is not at all necessary to test the module in simulation or emulate it on an FPGA to see that the properties are obeyed. Instead, the vendor need only construct a valid formal proof to show that these properties hold under all operational conditions.

This proof, once constructed, becomes a part of the finished package delivered to the IP consumer who, in turn, may then easily check the proof by running it through the Coq language interpreter. If the proof is valid, then he can accept the design, knowing that its operation stays within the functional boundaries set by the security property list. If the consumer is, say, a government or military organization, then he will have a strong reason to negotiate the production of such assurances. But it is also true that the vendor, too, will be able to assure himself that no inhouse manipulations of the design have introduced functionality in violation of these safety rules.

The proposed IP acquisition and delivery protocol overcomes the limitations of conventional IP security assurance methods. Previously proposed IP core checking methods are all based on the assumption that IP module providers are trusted. In order to prevent man-in-the-middle attacks, the IP vendors would also deliver test benches and simulation results which the IP consumers can use to check the genuineness of the delivered IP

[1]We note that the Coq proof assistant language, or any other proof assistant languages, do not ensure the security of IP modules nor can they create proofs automatically. The Coq proof assistant language is only chosen to represent IP modules, security properties, and proofs in a uniform way and has been proved effective in software PCC [15], [17], [18]. The Coq proof assistant platform can interpret the Coq language and check the proofs against security properties automatically. Any other platform which can provide similar functionality could be selected instead Coq in our scheme.

module. If, however, the IP vendors are untrusted or they lack methods to monitor inhouse manipulations, test benches themselves could be manipulated not to cover those malicious modifications. Furthermore, whether test benches can ensure the trustworthiness of IP cores is still controversial [12], [22]. Under the proposed framework, IP consumers switch their role from passive users to active IP protection proposers. That is, their involvement in defining the set of security properties constrains the room for inhouse manipulation as well as man-in-the-middle attacks. Since the security guarantees provided by this protocol are determined by the predefined properties, we should admit that an inappropriate set of security properties may leave the proof and delivered IP modules vulnerable. Security property libraries can be constructed beforehand to solve this problem, but this is out of the scope of this paper and will be elaborated in our future work.

The IP modules delivered by the vendor to the consumer will often be incorporated into a larger system. In such cases, it may be worthwhile to consider, at the beginning of the design cycle, whether having some additional properties proven about lower-level modules could simplify the construction of similar proofs for the larger design into which these subcomponents are integrated. As a continuation of that idea, we fully expect proofs to eventually be constructed modularly, much in the same fashion as IP cores themselves. As smaller components become embedded in larger systems, so too may the proofs of their respective security properties be used to demonstrate that the higher-level device is also subject to certain constraints in its operation. As some devices become standardized, and general consensus is reached on the sorts of relevant properties, a library of code-proof combinations will slowly be built. This will significantly simplify the task of proof construction while still maintaining the integrity of the framework. Additionally, some design teams may wish to engage a third-party proof-writer to construct a separate correctness proof in a strategy resembling $N$-version programming.

Moreover, we believe that the increasing use of FPGAs and other reprogrammable hardware means that an automatic system for establishing trust will take on growing importance in years to come. When HDL code is frequently recompiled for instant-update of deployed hardware systems, the ability to quickly establish the safety of that code is paramount.

## III. Provable Properties at the RT-Level: A Verilog Formulation for Safe Hardware

Because every statement in a module's HDL code must translate into a corresponding declaration in the theorem-proving language, it is necessary to specify this HDL and describe how such a translation might be carried out. We choose Verilog as our HDL for this paper, and give a precise definition of the syntax allowed under the current model.

This definition has three main components: combinational logic, sequential logic, and module declaration and instantiation. The combinational logic component consists of `assign` statements incorporating any of the standard bitwise logical and conditional operators, as shown in the complete syntactic specification below:

```
<sig-dec-block> ::= <sig-dec>
    | <sig-dec> <sig-dec-block>

<sig-dec> ::= <in-out-reg> <id-list> ";"
    | <in-out> <bus> <id-list> ";"

<in-out-reg> ::= <in-out> "reg"
    | <in-out> | "reg" | "wire"

<in-out> ::= "input" | "output"

<id-list> ::= <id> | <id> "," <id-list>

<bus> ::= "[" <num> ":" <num> "]"

<assign-block>

<assign-stmt> ::= <id> "=" <assign-right>

<assign-right> ::= <expression> |
  <expression> "?" <expression> ":"
<expression>

<expression> ::= <id> |
  <expression> "|" <expression> |
  <expression> "&" <expression> |
  "~" <expression> | "(" <expression> ")".
```

The grammar below specifies our handling of sequential behavior. In our current formulation, we support only synchronous sequential circuits, which should be sufficient for most applications. This is necessary because the notion of clock cycle as a discrete unit of time is used by our induction-based approach to proof construction; we rely on the countable set of synchronous clock cycles to assert predicates on signals and then prove inductively that these predicates hold over all points in time

```
<always-block> ::=
  "always @ (posedge clk)" <body>

<non-block-assign> ::= <id> "<=" <expr>

<body> ::= <stmt> |
  "begin" <block> "end"

<block> ::= <stmt> | <stmt> <block>

<stmt> ::= <non-block-assign> ";" |
  "if" <cond> <body> [<elseif>]+ [<else>]

<elseif> ::= "else if" <cond> <body>

<else> ::= "else" <body>

<expr> ::= <id>| "~" <expr>| "(" <expr> ")"|
  <expr> "&" <expr> | <expr> "|" <expr> |
  <expr> "+" <expr> |<expr> "-" <expr>

<cond> ::= <expr> "==" <expr> |
  <expr> "<" <expr> | <expr> "<=" <expr> |
  <expr> ">" <expr> | <expr> ">=" <expr> |
  <cond> "||" <cond> | <cond> "&&" <cond> |
  "!" <cond> | "(" <cond> ")".
```

Within the sequential logic, we have `if/else` statements (with the same logical and control operators as in combinational `assign` statements) and nonblocking assignment statements. As for the declaration of signals themselves, we have defined the `wire` and `reg` statements for both single-bit signals and bus lines. We also allow for module instantiations and definitions

```
<module-dec>::="module" <id> "(" <id-list> ")" ";"
                  <sig-dec-block>
                  <assign-block>
                  <always-block>
                  "endmodule"

<module> ::= <id> <id> "(" <id-list> ")" ";".
```

## IV. Proof Framework in Coq

Given the HDL specification presented in Section III, we derive a corresponding set of definitions in the Coq theorem language to model the functionality of circuits at the RT-level. This approach parallels [17]'s formulation of inference rules for the instruction set of CAP. An overview of the Coq proof assistant platform and the syntax of Coq language can be found in [21].

### *A. Combinational Logic*

We first define a `value` as an inductive set with two constructors, called `lo` and `hi`, and a `signal` as a mapping of time, specified in clock cycles and given as a natural number, onto a `value`. To handle the case of bus lines carrying multiple bits of information, we also define the type `bus` to represent an equivalent mapping of time onto a natural number:

```
  Inductive value := lo | hi.
  Definition signal := nat->value.
  Definition bus := nat->nat.
```

On top of these we build "expressions" consisting of combinational logic and control operations on sets of signals. Also defined as an inductive set, these expressions are essentially equivalent to the parse tree generated by a Verilog compiler, representing logical and arithmetic operations as a network of symbols:

```
(* expr represents syntax of possible      *)
(* operations on signals/bus values in Coq *)
Inductive expr :=
   | econs : signal->expr
   | and : expr->expr->expr
   | or: expr->expr->expr
   | not : expr->expr
   | cond : expr->expr->expr->expr
   | bus_eq : bus->bus->expr
   | bus_gt : bus->bus->expr
   | bus_lt : bus->bus->expr.
```

Each of these constructors defines an expression as some combination of other expressions or bus signals under a logical operator. Thus, the `and` constructor takes two expressions and returns one as a result. When evaluated by the semantic model we define shortly, this result will be interpreted as a logical AND of two signals. The `or` and `not` constructors behave in a similarly straightforward manner, and `cond` takes a control expression to select between two result expressions. A simple signal is converted to an expression with `econs`.

Expressions constructed from the recursive `expr` type are interpreted by the evaluate function `eval` which maps the expression tree onto the values of its signals at the specified time. We may thus say, for example, that the logical AND of two signals causes first one signal to be evaluated, followed by the second only if the first is `hi`. In this way, the `eval` function defines the operational semantics of expressions and is used to model the `assign` statement. Similarly, the `cmp_eq`, `cmp_gt`, and `cmp_lt` functions provide this definition recursively for bus values constructed with the inductive `nat` type in Coq

```
(*eval function defines semantics of expr's*)
Fixpoint eval (e:expr)(t:nat) {struct e} :=
  match e with
     | (econs sig) => (sig t)
     | (and ex1 ex2) => match (eval ex1 t) with
        lo => lo | hi => (eval ex2 t) end
     | (or ex1 ex2) => match (eval ex1 t) with
        hi => hi | lo => (eval ex2 t) end
     | (not ex) => match (eval ex t) with
        hi => lo | lo => hi end
     | (cond cex ex1 ex2) =>
          match (eval cex t) with
            hi => (eval ex1 t)
            | lo => (eval ex2 t) end
     | (bus_eq b1 b2) => (cmp_eq (b1 t) (b2 t))
     | (bus_gt b1 b2) => (cmp_gt (b1 t) (b2 t))
     | (bus_lt b1 b2) => (cmp_lt (b1 t) (b2 t))
  end.

Fixpoint cmp_eq (a b:nat) {struct b} :=
  match b with
  | O => match a with O => hi
           | _ => lo end
  | S n => match a with O => lo
           | S m => cmp_eq m n end
end.

Fixpoint cmp_gt (a b:nat) {struct b} :=
  match b with
  | O => match a with O => lo
           | _ => hi end
  | S n => match a with O => lo
           | S m => cmp_gt m n end
end.

Fixpoint cmp_lt (a b:nat) {struct b} :=
  match b with
  | O => lo
  | S n => match a with O => hi
           | S m => cmp_gt m n end
end.
```

The definition of `eval` provides the proof-writer (the IP vendor) with a sufficiently precise definition of combinational logic functionality to prove useful theorems about the behavior of signals. To prove, for example, that a signal assigned to the logical AND of two other signals is low at a given clock cycle, he need only show that at least one input signal is also low at this time and then "unfold" the definition in Coq to reveal the underlying structural relationship between inputs and outputs.

For each Verilog `assign` we generate a corresponding proposition with the `assign` function we have written in Coq and express that proposition as a Hypothesis statement so that the code vendor may refer to it in his proof:

```
  Definition assign : signal->expr->Prop :=
    fun (a:signal)(e:expr) =>
      forall (t:nat), (a t) = (eval e t).
```

This yields a proposition that the value of the assigned signal is equal to the value returned by calling eval on the expression to the right of the assignment operator, for which it also provides a Coq definition according to the rules outlined above. Assignment statements for bus signals are modeled with separate but analogous functions:

```
Definition bus_assign :=
  fun (y x:bus) =>
    forall t:nat, y t = x t.

Definition bus_cond_assign :=
  fun (y a b:bus)(e:expr) =>
    forall t:nat, y t = match (eval e t) with
      | hi => (a t)
      | lo => (b t) end.

Definition add_assign :=
  fun (y a b: bus) =>
    forall t:nat, y t = plus (a t) (b t).

Definition sub_assign :=
  fun (y a b:bus) =>
    forall t:nat, y t = minus (a t) (b t).
```

Note that special functions are necessary for conditional assignment and addition/subtraction. Their use is clear, but this is one aspect of our current formulation that could be streamlined in future work.

## B. Sequential Logic

The fundamental inductive structure used to define sequential logic in Coq is what we have called the updateblock. Like the expression definition for combinational logic, updateblocks are constructed as trees of operations on signals and bus lines. In this case, the permitted operations are nonblocking assignment (to an expression), and conditional assignment for bus lines, as shown below:

```
Inductive updateblock :=
  | upd : signal->expr->nat->updateblock
  | upd_bus : bus->bus->nat->updateblock
  | upd_bus_cond : expr->bus->bus->
     bus->nat->updateblock
  | upd_bus_add : bus->bus->bus->
     nat->updateblock
  | upd_bus_sub : bus->bus->bus->
     nat->updateblock
  | updcons : updateblock->
     updateblock->updateblock.
```

Every block which appears within a Verilog always statement generates a corresponding hypothesis in our Coq model to capture the meaning of nonblocking assignment. As an example, suppose the following assignment is to take place, as a result of some condition, in clock cycle $n$:

```
  x <= x + 1;
```

This expression would be represented as an updateblock using the upd constructor which takes as parameters a signal (in this case, x), an expression (x+1), and a clock cycle (type nat). When processed by the semantic model for sequential logic (defined later in this section), this results in a proposition that the value of x in cycle $n+1$ is equal to $x+1$ (note that (x n) represents the value of $x$ at cycle $n$):

```
  (x (S n)) = (x n) + 1.
```

The use of the other constructors is similar to that of upd. For assignment to bus signals, we have defined upd_bus, which is equivalent to upd with the signal and expression parameters both replaced by the bus type. Conditional nonblocking assignment for bus signals is accomplished with the upd_bus_cond constructor whose syntax resembles that of cond for combinational expressions. Conditional nonblocking assignment for other signals is handled with a different construction, the ifblock, which we describe later in this section. The updateblock type also includes upd_bus_add and upd_bus_sub to support addition and subtraction, respectively, on bus signals. The last constructor, updcons, links update blocks together to form a list of successive assignments.

This semantic value of these blocks is generated by the recursive definition update. Just as eval captured the semantics of combinational assignment, the update function provides the semantics of updateblock:

```
Fixpoint update (u:updateblock) {struct u} :=
  match u with
    | (upd sig exp t) => (sig (S t)) = (eval
    exp t)
    | (upd_bus y x t) => (y (S t)) = (x t)
    | (upd_bus_cond cex y a b t) =>
      (y (S t)) = match (eval cex t) with
                    hi => a t
                    | lo => b t end
    | (upd_bus_add y a b t) =>
      (y (S t)) = (plus (a t) (b t))
    | (upd_bus_sub y a b t) =>
      (y (S t)) = (minus (a t) (b t))
    | (updcons block1 block2) =>
      (update block1) /\ (update block2)
  end.
```

Finally, we represent Verilog if statements in always blocks through the inductive ifblock definition. This is the highest level of the sequential logic representation tree, and it includes updateblocks within its structure. An ifblock may be either a simple assignment list (constructed with ifsimple from an updateblock), an if statement with no else (constructed with ifelse from a control expression and another ifblock so as to allow for nesting), or an ifelse statement (constructed with ifelse from a control expression and two ifblocks). As with update and eval, we have again provided a recursive definition of ifblock semantics using a function called doif

```
Inductive ifblock :=
    | noif : updateblock->ifblock
    | ifsimple : expr->ifblock->ifblock
    | ifelse : expr->ifblock->ifblock->ifblock.

Fixpoint doif (i : ifblock)(t : nat)
      {struct i} :=
if (e1) begin
    if(e2) begin
        a <= b;
        b <= c;
    end
    else a <= c;
end
else if (e3) c <= a;
```

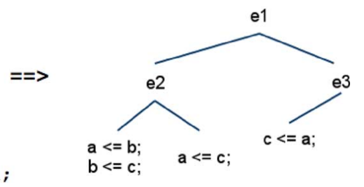

Fig. 2. If-else statement tree representation.

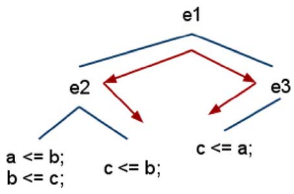

Fig. 3. If-block path traversal: For each variable that occurs in the left-hand side of a nonblocking assignment, traverse the if-tree and record all paths terminating in leaves where that variable is updated. Use the negation of the union of these paths to define condition for that variable retaining its previous value.

```
match i with
    | (noif up) => (update up)
    | (ifsimple exp ifb) =>
      match (eval exp t) with
          | hi => (doif ifb t)
          | lo => True
          end
    | (ifelse exp ifb1 ifb2) =>
      match (eval exp t) with
          | hi => (doif ifb1 t)
          | lo => (doif ifb2 t)
      end
end.
```

For the sake of simplicity, we have also assumed that no register may be updated more than once during a single clock cycle as this would lead to a potential reasoning error. In our example formulation, we state that an error would be raised by the Coq generator when this is the case, but a commercial implementation might choose to simply accept the last-declared assignment or follow some other reasonable scheme.

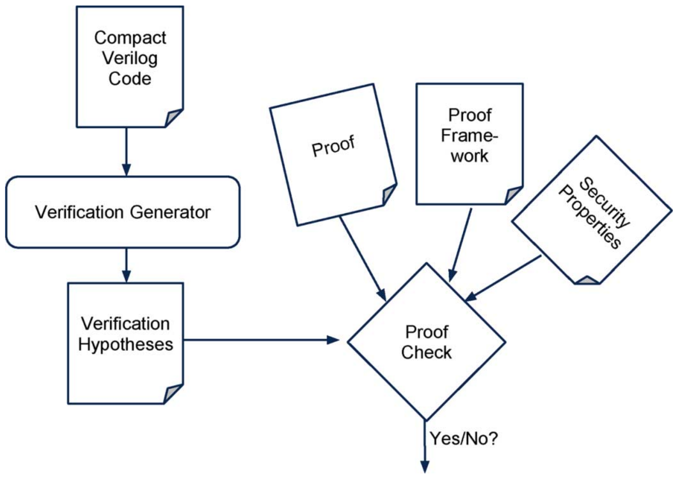

Fig. 4. Automated verification.

### C. Modeling the Behavior of Stored Values

This semantic model is sufficient to describe the assignment of new values to registers, but does not specify the retention of previously stored values in the case that no update occurs. It is, therefore, necessary to introduce a series of Coq propositions asserting that a register signal, when not assigned a new value during a given clock cycle, keeps the same value it had during the previous clock cycle. This behavior is implicit in Verilog, but we must make it explicit in our semantic representation so that it may be referenced in proofs.

Propositions about the maintenance of register values are generated by recursively examining the branches of each `ifblock` abstract syntax tree. Fig. 2 shows an example. For each register declared inside of a module, we look through that module's `ifblocks` and find all leaves at which the variable in question is updated. The path from the root to leaf represents one condition under which the variable receives a new value; by taking the union of all such paths we obtain a logical proposition that is true if an update occurs. By issuing an assertion that no update takes places if the negation of this proposition is true, we can specify that a register retains its value from one clock cycle to the next. Fig. 3 illustrates this approach for the variable `c` in the tree given in Fig. 2, resulting in the condition $\sim((e_1 \wedge \sim e_2) \vee (\sim e_1 \wedge e_3))$ for the retention of a stored value in `c`.

## V. Automatic Proof Validation

Fig. 4 shows the procedure for proof checking and module validation by the consumer upon receipt of the product from the vendor. The proof is first stripped of any circuit definitions declared with the `Hypothesis` statement in Coq to produce a "clean" version of the proof. The removed `Hypothesis` statements are the generated Verification Hypotheses (a.k.a. IP modules' Coq translation) used to model Verilog code based on the semantic representation described in Section IV. These hypotheses, once declared, admit a proposition as true so that it may be used as a precondition for a proof. They must, therefore, be deleted at the start of proof checking and then regenerated automatically from the provided HDL code. This is a necessary step because otherwise there is no guarantee that the circuit behavior defined in the IP vendor's proof actually matches that of the coded circuit. Although it is not necessary for IP vendors to provide Verification Hypotheses with proofs and IP cores, they are likely to do so, as often happens in software PCC domain, because IP vendors will need to check the proof anyway before delivering the whole package. The removal of Verification Hypotheses is required only when the proof is to be checked by a party who does not yet trust it. Note that this extra step does not increase the burden on IP consumers at all because the whole process is integrated into the automatic proof checking mechanism with very light computation workload. We also reject any lines containing the keyword `admit` which tells the Coq interpreter to accept a proposition as true without proof.

Upon receiving untrusted HDL code, then, IP consumers need to regenerate the Coq circuit model (Verification Hypotheses). This process is quite straightforward and is based on the HDL-Coq conversion rules developed in Section III. Fig. 4 illustrates these steps. The regenerated verification hypotheses are combined with the "clean" version of the proof as well the framework of definitions in Coq as described previously. At this point, the entire assemblage of Coq code is given to the interpreter to be checked. The consumer simply executes the Coq interpreter program, and if execution passes through to the end of the proof, then the proof is valid and the code obeys the security-related properties. The whole proof checking process is performed in an automatic and fast way on the Coq proof assistant platform so that IP consumer can quickly check the trustworthiness of the delivered IP modules. After passing the proof checking, the IP modules can be synthesized for ASIC or FPGA but IP module synthesis is not part of our protocol.

## VI. Example Design Scenario

In order to demonstrate the capabilities of our proposed methodology, we describe a sample design scenario where an assurance of trustworthiness is desired. By way of this example, we will show how intelligently selected security-related properties can prevent certain kinds of malicious behavior, how these properties are translated into a formal logic, how the vendor of code that conforms to these properties can construct a correctness proof, and, finally, how the consumer can check this proof against the code.

Of particular noteworthiness in this example is the relative freedom granted to the HDL coder in deciding how to implement the desired circuit, following as a consequence of the higher level of sophistication allowed in our property specification model as compared to others. We develop an abstract notion of a "protocol" which delimits a range of acceptable behaviors. This is in contrast to [19]'s proposal for PCH which allowed only for proofs that an FPGA layout implements a specific boolean logic function, requiring a level of specificity that precludes any functional differences between implementations and does not really bring the full potential of software PCC into the hardware domain.

### *A. Register File Copy Controller*

Our example is the following: suppose that the client needs a circuit which controls access to two register-files. Moreover, suppose that this controller is required to have a special mode called "copy" which, when activated by a special flag signal, `CF`, causes the controller to transfer the contents of one register file into the other. The sequence of reads and writes is not important, and neither are the addresses at which any individual value is stored; it is only required that each value in the first register file be copied, unchanged, to some location in the second. The illustration in Fig. 5 shows how such a component appears in block form.

A possible application for this module could be in an automatic teller machine (ATM) where it will be used to create and maintain two lists of account numbers for transaction processing. Such a setting provides ample motivation for strengthening security, since a nefarious hardware coder could exploit his control of the circuitry for financial gain or to obtain access to otherwise confidential information.

### *B. Choosing Security-Related Properties*

One can easily imagine several types of behavior a malevolent supplier might introduce into his implementation of this circuit; he could scan incoming data for a specific trigger value to activate a special mode, selectively block certain registers from being copied, enter an infinite loop on yet another trigger, and so on. With carefully crafted security properties, however, the consumer can successfully safeguard against each of these.

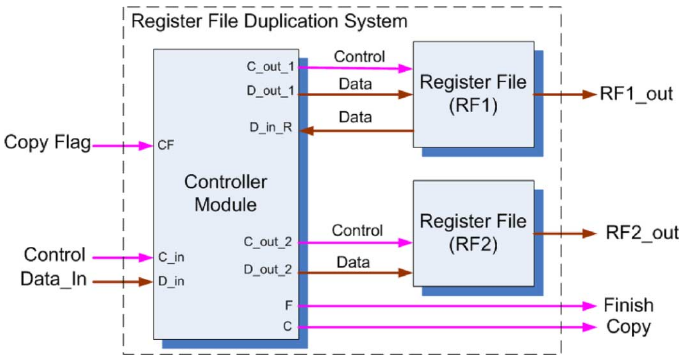

Fig. 5. Register file duplication system and the controller module.

Being aware of these possible modes of attack, the consumer will probably choose a set of properties such as the following: 1) **Stability**: do not enter copy mode unless the copy flag has been raised; 2) **Transparency**: when not in copy mode, simply pass control signals through to both RFs; and 3) **Termination-Transfer**: when the copy flag is raised, enter copy mode, transfer all values, unmodified from RF1 to RF2, and then exit copy mode within a certain predefined number of cycles.

These properties outline the limits of acceptable circuit behavior, and so a proof of compliance with them will guard against the kinds of attacks enumerated in the previous section; any circuit engaging in such behavior clearly breaks the rules.

We will now see how these properties may be translated into a formal mathematical logic. Below is our rendering of the specification into a set of Coq theorems (the bodies of each proof are blank initially—these are to be filled in by the vendor). Stability is easily expressed as a proposition that we must remain outside of copy-mode in all cycles for which the controller is not already in copy mode and for which the copy flag is low:

```
Theorem stable_c : forall t:nat, t > 0 ->
  c t = lo -> cf t = lo -> c (S t) = lo.
```

The definition of Transparency is similarly straightforward in that it simply asserts an equality of the input and output control signals:

```
Theorem transparency : forall t:nat,
  t > 0 -> c t = lo ->
  a1 t = a1_in t /\ d1 t = d1_in t /\
  we1 t = we1_in t /\
  a2 t = a2_in t /\ d2 t = d2_in t /\
  we2 t = we2_in t.
```

The last property exhibits significantly greater complexity, revealing where our framework can be most versatile. We define Termination-Transfer—which contains the bulk of our specification—as a hierarchy of subproperties. For example, to define the operation of reading from an address $a$, we create a property called `read` which asserts that the value sent on the address line to RF1 during the stated clock cycle is equal to $a$. We also pass a variable $X$ to capture the value returned from RF1, allowing us to refer to this value when we show that it is written to RF2:

```
Definition read := fun (a n t X : nat) =>
  (a1 (t+n)) = a /\ (q1 (t + n)) = X.
```

The write operation is defined in a similar fashion, stipulating that write-enable is high during cycle $t+n$ and that the value sent on the data line to RF2 is equal to some value $X$. In defining the complete transfer operation predicate, we link the read and write properties together, asserting that the read $X$ is also the $X$ to be written. But before we can describe the top-level transfer property, we complete the `write` definition by specifying write-uniqueness. Given as the `unique` property, this asserts that a value, once stored, will not be overwritten. That is, there exists no index $nm > n$ in the current copy at which write-enable is high and the same address is sent to RF2:

```
Definition unique := fun (n t nf : nat) =>
  forall nm:nat, nm > 0 -> nm < nf -> nm > n
  ->
  (we2 (t+nm)) = hi -> (a2 (t+nm)) <> (a2
  (t+n)).

Definition write := fun (X n t nf : nat) =>
   (we2 (t+n)) = hi /\ (d2 (t+n)) = X /\
   (unique n t nf).
```

Finally, `transfer` is defined to indicate a counter index $nf$ by which all possible addresses have been read from RF1 and written to RF2. We make this assertion with a universal quantifier over all addresses $a$, requiring the existence of some index $n$ such that at time $t+n$ we read the value $X$ from address $a$ and for which there also exists some other index $nw$ at which we write $X$ to RF2:

```
Definition transfer := fun (t nf : nat) =>
  forall a:nat, a <= regs -> exists n:nat,
    n > 0 /\ n < nf /\ exists X:nat,
    (read a n t X) /\ exists nw:nat, nw > 0
    /\ nw < nf /\ (write X nw t nf).
```

In specifying the security properties as we have done with a multilevel property tree, we have paralleled the work presented in [17], which also constructs an elaborate series of quantified predicates in order to define a "valid free list" for a memory allocator. We claim that this structural similarity provides evidence for our framework's unique success in porting the flexibility of PCC into the hardware domain.

## C. Sample Implementation

To see what might constitute an acceptable implementation of the Register File Copy Controller circuit, we have crafted two security-compliant examples. The first performs the copy operation by sequentially reading register values from RF1, saving them for one clock cycle, and then writing them to the register at the same address in RF2. The second completes this task in reverse order, counting down from the highest register address to the lowest. Between these two examples, we demonstrate that our model allows for security-related properties that do not necessarily specify all details of operation, but which instead merely provide the boundaries for acceptable operation:

```
(* Verilog code for first sample safety-compliant*)
(* RF-controller module                          *)
module controller1 (clk, reset, a1_in,
  d1_in, we1_in, a2_in, d2_in,
  we2_in, q1, cf, a1, d1, we1, a2, d2, we2,
  c);

  (* Inputs: clock and reset signals, as well      *)
  (* as input addresses, data and write-enable     *)
  (* signals for RF1 and RF2. q1 is the value      *)
  (* read back from RF1 during copy, and cf is     *)
  (* the copy flag, which must be asserted to      *)
  (* enter copy mode.                              *)
  input clk, reset;
  input [4:0] a1_in, a2_in;
  input [31:0] d1_in, d2_in, q1;
  input we1_in, we2_in, cf;

  output [4:0] a1, a2;
  output [31:0] d1, d2;
  output we1, we2;
  output reg c;

  (* Registers: cur_read and cur_write contain     *)
  (* addresses of the current read/write loop      *)
  (* counters respectively, and stored_value       *)
  (* is used to store read data during a copy.     *)
  (* cprev is the value of c in the previous       *)
  (* cycle.                                        *)
  reg [4:0] cur_read, cur_write;
  reg [31:0] stored_value;
  reg cprev;

  (* Combinational assignment: set the address     *)
  (* of RF1 to either cur_read or the current      *)
  (* input address depending on whether we are     *)
  (* in copy mode. For a2 we choose between        *)
  (* the input address and cur_write for the       *)
  (* same reason. The we1 signal is always         *)
  (* disabled in copy mode, and we2 is always      *)
  (* hi after the first cycle of copy mode.        *)
  assign a1 = (c) ? cur_read : a1_in;
  assign d1 = d1_in;
  assign we1 = (c) ? 1'b0 : we1_in;

  assign a2 = (c) ? cur_write : a2_in;
  assign d2 = (c) ? stored_value : d2_in;
  assign we2 = (cprev & c) ? 1'b1 :
                  (c ? 1'b0 : we2_in);

  (* Sequential logic section: we have a reset     *)
  (* signal that clears the registers at start.    *)
  (* When not in reset, check for end of copy      *)
  (* where cur_write is 31, and end copy mode      *)
  (* if it is. Otherwise, increment cur_read       *)
  (* and cur_write if copying, or reset to zero    *)
  (* if entering copy mode.                        *)
  always @ (posedge clk) begin
    if (~reset) begin
      cur_write <= 5'b00000;
      cur_read <= 5'b00000;
      c <= 1'b0;
      cprev <= 1'b0;
      stored_value <= 32'd0;
    end
      else begin
        if (cur_write == 5'b11111 &~cf) begin
          cur_write <= 5'b00000;
          cur_read <= 5'b00000;
          c <= 1'b0;
          cprev <= 1'b0;
          stored_value <= 32'd0;
        end
        else begin
          cur_read <= (cf &~c) ? 5'b00000 :
                     cur_read + 5'b00001;
        cur_write <= (cf & ~c) ? 5'b00000 :
                   cur_read;

        stored_value <= q1;
        c <= (cf | c);
        cprev <= c;
      end
    end
  end
endmodule


(* Example controller #2: we omit everything     *)
(* but the always block, since it is the same    *)
(* as in example #1. In this case, we count      *)
(* backwards to 0 in the address read/writes     *)

  (* ... *)
  always @ (posedge clk) begin
    if (~reset) begin
    cur_write <= 5'b00000;
    cur_read <= 5'b00000;
    c <= 1'b0;
      cprev <= 1'b0;
      stored_value <= 32'd0;
    end
      else begin
        if (cur_write == 5'b00000 & ~cf) begin
          cur_write <= 5'b00000;
          cur_read <= 5'b00000;
          c <= 1'b0;
          cprev <= 1'b0;
          stored_value <= 32'd0;
        end
        else begin
          cur_read <= (cf & ~c) ? 5'b11111 :
            cur_read - 5'b00001;
          cur_write <= (cf & ~c) ? 5'b11111 :
            cur_read;
          stored_value <= q1;
          c <= (cf | c);
          cprev <= c;
        end
      end
    end
    (* ··· *).
```

We do not present code for a negative example (i.e., one which fails to obey the rules), but it is easy to see how the code for either sample circuit could be modified to behave maliciously. One could, for instance, simply introduce a conditional operator into the read/write loop to exit when a specific trigger value is read. For such a circuit, it would not be possible to construct a proof, and the proofs later presented for our security-safe examples would certainly not be valid for this malicious variant.

## D. Proving Security Compliance

Once the circuit has been coded, the first step of any proof construction is the generation of Verification Hypotheses. These are the Coq propositions described in Section IV and used to represent HDL statements in a formal proof. They are generated automatically from Verilog code according to the previously outlined rules. The following Coq code, then, represents the example `controller1` from above:

```
(* what hypotheses & proof for controller1 *)
(* circuit would look like                 *)
Section controller1.

(* inputs and outputs *)
Variable reset : signal.
Variables a1_in d1_in a2_in d2_in a1 a2 d1 d2
q1: bus.
Variables we1_in we2_in we1 we2 : signal.
Variables cf c : signal.

(* internal signals *)
Variables cur_read cur_write stored_value : bus.
Variable cprev : signal.

(* Address of highest register—we use a variable  *)
(* rather than "31" in order to make keep the      *)
(* proof clean.
Variable regs : nat.
Hypothesis rmin : regs > 5.
Lemma rno : cmp_eq 0 regs = lo.
  assert (regs <> 0). omega.
  rewrite cmp_neq_neg.
  trivial. omega.
Qed.

(* hypotheses—these represent the semantics of  *)
(* each Verilog statement as written in the     *)
(* example code                                 *)

Hypothesis initial_c : (c O) = lo.

Hypothesis assign_a1 :
  (bus_cond_assign a1 cur_read a1_in (econs c)).

Hypothesis assign_d1 : (bus_assign d1 d1_in).

Hypothesis assign_we1 :
  (assign we1
    (cond (econs c) (econs Gnd) (econs we1_in))).

Hypothesis assign_a2 :
  (bus_cond_assign a2 cur_write a2_in (econs c)).

Hypothesis assign_d2 :
  (bus_cond_assign d2 stored_value
    d2_in (econs c)).

Hypothesis assign_we2 : (assign we2
  (cond (and (econs cprev) (econs c))
    (econs Vdd)
    (cond (econs c)
      (econs Gnd)
      (econs we2_in)))).

(* automatically defined for systems with a     *)
(* "reset" signal                               *)
Hypothesis initial_reset : reset 0 = lo.
Hypothesis reset_hi :
  forall t : nat, t > 0 –> reset t = hi.

(* whenever the
Definition sum_cur_read_1 :=
  fun t:nat => cur_read t + 1.

Hypothesis if_not_reset : forall t:nat,
  (doif (ifelse (not (econs reset))
    (noif (updcons
      (updcons
      (updcons
        (upd_bus cur_read (const 0) t)
        (upd_bus cur_write (const 0) t))
      (updcons
        (upd c (econs Gnd) t)
        (upd cprev (econs Gnd) t)))
      (upd_bus stored_value (const 0) t)))
  (ifelse (and (bus_eq cur_write
            (const regs)) (not (econs cf)))
    (noif (updcons
      (updcons
      (updcons
        (upd_bus cur_read (const 0) t)
        (upd_bus cur_write (const 0) t))
      (updcons
        (upd c (econs Gnd) t)
        (upd cprev (econs Gnd) t)))
      (upd_bus stored_value (const 0) t)))

  (noif (updcons
    (updcons
    (updcons
      (upd_bus_cond (and (econs cf)
                    (not (econs c)))
        cur_read (const 0) sum_cur_read_1 t)
      (upd_bus_cond (and (econs cf)
                    (not (econs c)))
        cur_write (const 0) cur_read t))
    (updcons
      (upd c (or (econs cf) (econs c)) t)
      (upd cprev (econs c) t)))
    (upd_bus stored_value q1 t)))))
  t).
```

With this generation having been completed, we may now begin construction of a proof. The first two required properties are trivial, so we will not describe their proofs here. For the more complex Termination-Transfer rule, however, we are forced to adopt a more elaborate plan of attack; just as the property itself was stated as a combination of smaller definitions, so too will the proof be constructed from a set of more primitive lemmas. The full proof for the first example circuit may be seen in Appendix A, but an outline of its construction is given below.

The method of induction on clock cycle informs our general technique. Most lemmas rely on a "transition cycle" $t$ which marks the transition into copy-mode, and an index $n$ which counts a certain number of cycles after this transition. Thus, if the transition occurs at time 15, then time 18 could be represented as $t = 15$ and $n = 3$.

At the center of these proofs is the `count_all` lemma, showing inductively that the circuit remains in copy mode until the current write address reaches its highest possible value, and that this write address is always two less than the count index $n$:

```
Lemma count_all : forall t:nat, t>0 ->
  cf t = hi -> c t = lo -> forall n:nat,
  n < S regs -> cur_read(t + S n) = n /\
  cur_write(t+S(S n)) = n /\
  c (t+S(S n)) = hi /\ cprev (t+S(S n)) = hi.
          ...
```

With this fact in place, the lemma `read_eq` easily follows, establishing that the current read address remains one less than the write address for the duration of copying. Other lemmas are then constructed on top of these, proving for example that the uniqueness subproperty holds on all writes and that the sequence of operations performed in copy mode is a complete transfer.

It is easy to imagine a parallel proof for the second circuit and, indeed, we have constructed one using the same structure of lemmas. To do this, we simply rewrote most of the lemmas in a manner consistent with the new direction of operation, changing definitions too where appropriate.

## VII. Conclusion

While traditional approaches to hardware security have focused on leveraging assertion-based testing and formal verification methods, we have shown that work done by computer science researchers on PCC can be successfully translated to the domain of hardware trustworthiness in order to provide a definitive guarantee that HDL code obeys a set of security-related properties.

By assigning vendors the task of constructing compliance proofs for their hardware IP, we allow consumers to know quickly and easily that the hardware they purchase operates within the parameters they have chosen as provable security properties. With a set of well-formulated and proven properties, the consumer will know that he cannot be the victim of certain varieties of attack, as it will be impossible to prove adherence to the rules for any module that engages in the undesired behavior. We should note that IP vendors may also recruit third parties to write proofs rather than do it themselves. Again, whether the third party is trusted or not does not really matter under the proposed IP acquisition and delivery protocol as long as the security properties are predefined. In fact, the proposed protocol does not include any constraints on the origin of the proof codes. Any party (but usually the IP vendors) can provide the proof with which IP consumers can quickly validate the trustworthiness of the delivered IP modules.

Our example design scenario shows how a set of complicated security properties can be constructed for a given module and proven for valid implementations. It also demonstrates how these properties, when proven, will not allow for the inclusion of anticipated malicious behaviors. In this case, these might include an if-statement that looks for a specific read address and deliberately blocks its corresponding memory cell from being transferred, or yet another which locks the controller up in an infinite loop when it detects a special trigger value.

Having anticipated these varieties of attack, the IP consumer formulated a set of security-properties which, if proven, would guarantee that no such malicious behavior could be implemented. Since it is required that the module exit copy-mode after a certain number of cycles, it would be impossible for any infinite loop denial-of-service attack to provably satisfy the `termination_transfer` property. It is likewise true that any implementation which maliciously fails to copy certain registers—under any condition—could not be proven to comply with the complex `transfer` property hierarchy.

It is not difficult to imagine an extension of our framework for use in other applications beyond the example design scenario presented above. We believe that the current needs of many hardware IP-consuming organizations could be better served with such a framework for provably trustworthy hardware acquisition as an established component of the design cycle. Future work will include the production of an automated verification generator, a more complete semantic representation of the Verilog language, a more thorough analysis of the soundness of the inference rules for generating Coq representations, and the development of a better behavioral circuit model in the theorem-proving language.

## APPENDIX A
## FULL PROOF OF TEST CIRCUIT 1

```
(* what hypotheses & proof for
  controller1 circuit would look like *)
Section controller1.

(* inputs and outputs *)
Variable reset : signal.
Variables a1_in d1_in a2_in d2_in : bus.
Variables a1 a2 d1 d2 q1 : bus.
Variables we1_in we2_in we1 we2 : signal.
Variables cf c : signal.

(* internal signals *)
Variables cur_read cur_write stored_value : bus.
Variable cprev : signal.

(* number of highest register *)
Variable regs : nat.
Hypothesis rmin :  regs > 5.
Lemma rno : cmp_eq 0 regs = lo.
  assert(regs <> 0). omega.
  rewrite cmp_neq_neg.
  trivial. omega.
Qed.

(* hypotheses *)

Hypothesis initial_c : (c O) = lo.

Hypothesis assign_a1 :
  (bus_cond_assign a1 cur_read a1_in (econs c)).

Hypothesis assign_d1 : (bus_assign d1 d1_in).

Hypothesis assign_we1 :
  (assign we1 (cond (econs c)
    (econs Gnd) (econs we1_in))).

Hypothesis assign_a2 :
  (bus_cond_assign a2 cur_write a2_in (econs c)).

Hypothesis assign_d2 :
  (bus_cond_assign
    d2 stored_value d2_in (econs c)).

Hypothesis assign_we2 : (assign we2
  (cond (and (econs cprev) (econs c))
    (econs Vdd)
    (cond (econs c)
      (econs Gnd)
      (econs we2_in)))).

Hypothesis initial_reset : reset 0 = lo.

Hypothesis reset_hi :
  forall t : nat, t > 0 -> reset t = hi.

Definition sum_cur_read_1 :=
  fun t:nat => cur_read t + 1.

(* property definitions for final theorem *)

Definition unique := fun (n t nf : nat) =>
  forall nm:nat, nm > 0 -> nm < nf -> nm > n ->
  (we2 (t+nm)) = hi -> (a2 (t+nm)) <> (a2 (t+n)).

Definition write := fun (X n t nf : nat) =>
  (we2 (t+n)) = hi /\ (d2 (t+n)) = X /\ (unique n t nf).

Definition read := fun (a n t X : nat) =>
  (a1 (t+n)) = a /\ (q1 (t + n)) = X.

Definition transfer := fun (t nf : nat) =>
  forall a:nat, a <= regs -> exists n:nat,
    n > 0 /\ n < nf /\ exists X:nat,
    (read a n t X) /\
    exists nw:nat,
      nw > 0 /\ nw < nf /\ (write X nw t nf).

Hypothesis if_not_reset : forall t:nat,
  (doif (ifelse (not (econs reset))
    (noif (updcons
      (updcons
      (updcons
        (upd_bus cur_read (const 0) t)
        (upd_bus cur_write (const 0) t))
      (updcons
        (upd c (econs Gnd) t)
        (upd cprev (econs Gnd) t)))
      (upd_bus stored_value (const 0) t)))

    (ifelse (and (bus_eq cur_write
            (const regs)) (not (econs cf)))
      (noif (updcons
        (updcons
        (updcons
          (upd_bus cur_read (const 0) t)
          (upd_bus cur_write (const 0) t))
        (updcons
          (upd c (econs Gnd) t)
          (upd cprev (econs Gnd) t)))
        (upd_bus stored_value (const 0) t)))

      (noif (updcons
        (updcons
        (updcons
          (upd_bus_cond (and (econs cf)
                  (not (econs c)))
            cur_read (const 0) sum_cur_read_1 t)
          (upd_bus_cond (and (econs cf)
                  (not (econs c)))
            cur_write (const 0) cur_read t))
        (updcons
          (upd c (or (econs cf) (econs c)) t)
          (upd cprev (econs c) t)))
        (upd_bus stored_value q1 t)))))
  t).
(* useful user-designed lemmas for use in proof *)

Lemma transitions : forall t:nat, t > 0 ->
  c t = lo -> cf t = hi -> c (S t) = hi /\
  cprev (S t) = lo /\ cur_read (S t) = 0
  /\ cur_write (S t) = 0.
intros.

generalize if_not_reset. intro inr.
specialize inr with (t:=t).
unfold doif in inr.
unfold update in inr.
unfold eval in inr.
rewrite H0 in inr.
rewrite H1 in inr.

apply reset_hi in H.  rewrite H in inr.
generalize inr.
unfold const.
unfold sum_cur_read_1.
case cmp_eq.
  tauto.
  tauto.
Qed.

Lemma count_helper : forall t:nat, t>0 ->
```

```
  cf t = hi -> c t = lo -> forall n:nat,
  n < S regs -> cur_read(t + S(S n)) = S n /\
  cur_write(t+S(S n)) = n /\
  c (t+S(S n)) = hi /\ cprev (t+S(S n)) = hi.
intros.

induction n.

  generalize transitions. intro tr.
  specialize tr with (t:=t).
  apply tr in H.
  generalize if_not_reset. intro inr.
  specialize inr with (t:=S t).
  unfold doif in inr.
  unfold update in inr.
  unfold eval in inr.
  assert (reset (S t) = hi).
    apply reset_hi. omega.
  rewrite H3 in inr.
  assert (cur_write (S t) = 0).
    apply H. rewrite H4 in inr.
  unfold const in inr. rewrite rno in inr.
  assert (c(S t) = hi).
    apply H. rewrite H5 in inr.
  assert (cur_read(S t) = 0). apply H.
  unfold sum_cur_read_1 in inr. rewrite H6 in inr.
  assert (S(S t) = t+2).
    omega. rewrite H7 in inr.
  generalize inr.
  case cf.
    intuition. intuition.
  assumption. assumption.

  assert (n < S regs). omega.
  apply IHn in H3.
  generalize if_not_reset. intro inr.
  specialize inr with (t:=t + S (S n)).
  unfold doif in inr.
  unfold update in inr.
  unfold eval in inr.
  assert (reset (t+S(S n)) = hi).
    apply reset_hi. omega.
  rewrite H4 in inr.
  assert (cur_write(t+S(S n)) = n).
    apply H3. rewrite H5 in inr.
  assert (cur_read(t+S(S n)) = S n). apply H3.
  unfold sum_cur_read_1 in inr. rewrite H6 in inr.
  assert (c(t + S(S n)) = hi).
    apply H3. rewrite H7 in inr.
  unfold const in inr.
  assert (n <> regs).
    omega. apply cmp_neq_neg in H8.
  rewrite H8 in inr.
  assert (S(t+S(S n)) = t+S(S(S n))). omega.
  rewrite H9 in inr.
  assert (S n + 1 = S(S n)).
    omega. rewrite H10 in inr.
  generalize inr.
  case cf.
    intuition. intuition.
Qed.

Lemma count_all : forall t:nat, t>0 ->
  cf t = hi -> c t = lo -> forall n:nat,
  n < S regs -> cur_read(t + S n) = n /\
  cur_write(t+S(S n)) = n /\
  c (t+S(S n)) = hi /\ cprev (t+S(S n)) = hi.
intros.
induction n.
  split.
    assert(t+1=S t). omega. rewrite H3.
    generalize transitions. intro tr.
    specialize tr with (t:=t).
    apply tr.
      assumption. assumption. assumption.
  generalize count_helper. intro ch.
  specialize ch with (t:=t).
  assert(cur_read(t+2) = 1 /\
    cur_write(t+2) = 0 /\ c (t+2) = hi /\
    cprev (t+2) = hi).
  apply ch.
  assumption. assumption. assumption. omega.
  apply H3.

  split.
    generalize count_helper. intro ch.
    specialize ch with (t:=t).
    assert (cur_read(t+S(S n)) = S n /\
      cur_write(t+S(S n)) = n /\
      c(t+S(S n)) = hi /\ cprev(t+S(S n)) = hi).
      apply ch.
        assumption. assumption. assumption.
        omega.
    apply H3.
    assert (cur_read(t+S(S(S n))) = S(S n) /\
        cur_write(t+S(S(S n))) = S n /\
        c(t+ S(S(S n))) = hi /\
        cprev (t + S(S(S n))) = hi).
        apply count_helper.
          assumption. assumption. assumption.
          omega.
      apply H3.
Qed.

Lemma c_lock_helper : forall t:nat, t>0 ->
  cf t = hi -> c t = lo -> forall n:nat,
  n < S(S regs) -> c (t+S n) = hi.
intros.
induction n.
  assert(t+1 = S t). omega. rewrite H3.
  generalize transitions.
  intro tr. specialize tr with (t:=t).
  apply tr.
  assumption. assumption. assumption.

  assert(cur_read(t + S n) = n /\
    cur_write(t+S(S n)) = n /\
    c (t+S(S n)) = hi /\
    cprev (t+S(S n)) = hi).
    apply count_all.
      assumption. assumption. assumption.
      omega.
  apply H3.
Qed.

Lemma c_lock : forall t:nat, t > 0 ->
  cf t = hi -> c t = lo -> forall n:nat,
  n > 0 -> n < S(S(S regs)) -> c(t+n) = hi.
intros.
induction n.
  assert(t+0=t). omega. rewrite H4.
  rewrite H1. rewrite lo_neq_hi.
  intuition.

  apply c_lock_helper.
    assumption. assumption. assumption.
    omega.
Qed.

Lemma cprev_lock : forall t:nat, t > 0 ->
  cf t = hi -> c t = lo -> forall n:nat,
  n < regs+1 -> cprev(t+S(S n)) = hi.
intros.
induction n.
  assert(c(t+1) = hi). apply c_lock.
    assumption. assumption. assumption.
    omega. omega.
  generalize if_not_reset. intro inr.
  specialize inr with (t:=t+1).
  unfold doif in inr.
  unfold update in inr.
  unfold eval in inr.

  assert(reset(t+1) = hi).
    apply reset_hi. omega.
  rewrite H4 in inr.
  rewrite H3 in inr.

  assert(c (S t) = hi /\ cprev (S t) = lo /\
    cur_read (S t) = 0 /\ cur_write (S t) = 0).
    apply transitions.
    assumption. assumption. assumption.

  assert(cur_write(S t) = 0). apply H5.
  assert(S t = t+1).
    omega. rewrite H7 in H6.
  rewrite H6 in inr.
  unfold const in inr.
  rewrite rno in inr.
  assert(S(t+1) = t+2).
    omega. rewrite H8 in inr.
  generalize inr.
  case cf.
    tauto. tauto.

  assert(cur_read(t + S (S n)) = S n /\
    cur_write(t+S(S(S n))) = S n /\
    c (t+S(S(S n))) = hi /\
    cprev (t+S(S(S n))) = hi).
    apply count_all.
    assumption. assumption. assumption.
    omega.
  apply H3.
Qed.

Lemma read_eq : forall t:nat, t > 0 ->
  cf t = hi -> c t = lo -> forall n:nat,
  n < regs+1 -> cur_read (t+S n) = n.
intros.
assert(cur_read(t + S n) = n /\
    cur_write(t+S(S n)) = n /\
    c (t+S(S n)) = hi /\
    cprev (t+S(S n)) = hi).
    apply count_all.
      assumption. assumption. assumption.
      omega.
apply H3.
Qed.

Lemma unchc : forall t:nat, t > 0 ->
  cf t = lo -> c t = lo -> c (S t) = c t.
intros.
generalize if_not_reset. intro inr.
specialize inr with (t:=t).
unfold doif in inr.
unfold update in inr.
unfold eval in inr.
assert (reset t = hi).
    apply reset_hi. assumption.
rewrite H2 in inr.
rewrite H0 in inr.
rewrite H1 in inr.
generalize inr.
unfold Gnd.
rewrite H1.
case cmp_eq.
  tauto. tauto.
Qed.

Lemma cftrans : forall t:nat, t>0->
  c t = lo -> c (S t) = hi -> cf t = hi.
intros.
assert(cf t = lo -> c (S t) = c t).
  intro.
  apply unchc. assumption.
  assumption. assumption.
assert(~(cf t = lo)).
  rewrite H0 in H2.
  rewrite H1 in H2.
  symmetry in H2.
  rewrite lo_neq_hi in H2.
  intuition.
apply lohi in H3.
assumption.
Qed.

Lemma unique_write : forall n t : nat, t > 0 ->
  c t = lo -> cf t = hi -> n > 1 ->
  n < S(S(S regs)) ->
     (unique n t (S(S(S regs)))).
intros.
unfold unique.
intros.
assert(c (t+n) = hi).
  apply c_lock.
    assumption. assumption. assumption.
  omega. assumption.

assert(cur_write (t + nm) = nm-2).
  assert(cur_read(t + S (nm-2)) = nm-2 /\
    cur_write(t+S(S (nm-2))) = nm-2 /\
    c (t+S(S (nm-2))) = hi /\
    cprev (t+S(S (nm-2))) = hi).
    apply count_all.
    assumption. assumption. assumption.
    omega.
  assert(t+S(S(nm-2)) = t+nm). omega.
  rewrite H10 in H9.
  apply H9.

assert(c (t+nm) = hi).
  apply c_lock.
  assumption. assumption.
  assumption. assumption.
  omega.

unfold bus_cond_assign in assign_a2.
unfold eval in assign_a2.
generalize assign_a2. intro a21.
specialize a21 with (t:=t+nm).
rewrite H10 in a21.
rewrite H9 in a21.
rewrite a21.

assert(cur_write(t+S(S (n-2))) = n-2).
assert(cur_read(t + S (n-2)) = n-2 /\
  cur_write(t+S(S (n-2))) = n-2 /\
  c (t+S(S (n-2))) = hi /\
  cprev (t+S(S (n-2))) = hi).
  apply count_all.
  assumption. assumption. assumption.
  omega.
apply H11.

assert(t+S(S(n-2)) = t+n). omega.
rewrite H12 in H11.

generalize assign_a2. intro a22.
specialize a22 with (t:=t+n).
rewrite H8 in a22.
rewrite H11 in a22.
rewrite a22.
clear H0. clear H1. clear H2. clear H3.
clear H4. clear H7. clear H8. clear H9.
clear H10. clear a21. clear H11.
omega.
Qed.

(* show the cycle-address correspondence *)
```

```
Lemma read_regs1 : forall t a: nat, t > 0 ->
  c t = lo -> cf t = hi -> a < regs+1 ->
  a1 (t+S a) = a.
intros.
generalize assign_a1. intro aa.
unfold bus_cond_assign in aa.
specialize aa with (t:=t+(S a)).
unfold eval in aa.

assert(c (t + S a) = hi).
  apply c_lock.
  assumption. assumption. assumption.
  omega. omega.

rewrite H3 in aa.
rewrite aa.
apply read_eq.
  assumption. assumption. assumption.
assumption.
Qed.

(* show that a full transfer can be completed
  in 34 cycles *)
Lemma transfer_regs3 : forall t: nat,
  t > 0 -> c t = lo -> cf t = hi ->
     (transfer t (S(S(S regs)))).
intros.
unfold transfer.
intros.
exists (S a).
split. omega.
split. omega.

exists (q1 (t + S a)).

split.
unfold read.

split.
apply read_regs1.
  assumption. assumption. assumption.
  omega. trivial.

exists (S(S a)).
split. omega. split. omega.

unfold write.
split.
generalize assign_we2. intro aw2.
unfold assign in aw2. unfold eval in aw2.
specialize aw2 with (t:=t+S(S a)).
assert(cprev(t+S(S(a))) = hi). apply
cprev_lock.
  assumption. assumption. assumption. omega.
rewrite H3 in aw2.
assert(c (t+S(S a)) = hi).
  apply c_lock.
  assumption. assumption. assumption.
  omega. omega.
rewrite H4 in aw2.
unfold Vdd in aw2.
assumption.

split.
generalize assign_d2. intro ad2.
unfold bus_cond_assign in ad2.
specialize ad2 with (t:=t+S(S a)).
unfold eval in ad2.
assert(c (t+S(S a)) = hi). apply c_lock.
  assumption. assumption. assumption.
  omega. omega.
rewrite H3 in ad2.
generalize if_not_reset. intro inr.
unfold doif in inr.
unfold update in inr.
unfold eval in inr.
specialize inr with (t:=t+S a).
assert(reset(t+S a) = hi).
  apply reset_hi. omega.
rewrite H4 in inr.
assert(cur_write(t+S a) <> regs).
  induction a.
  generalize transitions.
  intro tr. specialize tr with (t:=t).
  assert(S t = t+1). omega. rewrite H5 in tr.
  assert(cur_write(t+1) = 0). apply tr.
    assumption. assumption. assumption.
  omega.
  assert(cur_write(t+S(S a)) = a).
    assert(cur_read(t + S a) = a /\
      cur_write(t+S(S a)) = a /\
      c (t+S(S a)) = hi /\ cprev (t+S(S a)) = hi).
      apply count_all.
      assumption. assumption. assumption.
      omega.
    apply H5.
  rewrite H5.
  omega.
apply cmp_neq_neg in H5.
unfold const in inr. rewrite H5 in inr.
assert(c(t+S a) = hi).
  apply c_lock.
    assumption. assumption. assumption.
  omega. omega.
rewrite H6 in inr.
assert(stored_value(S(t+S a)) = q1(t+S a)).
  generalize inr.
  case cf. tauto. tauto.
assert(S (t + S a) = t+S(S a)).
  omega. rewrite H8 in H7.
rewrite H7 in ad2.
assumption.
apply unique_write.
  assumption. assumption. assumption.
  omega. omega.
Qed.


(* Theorems to be proven about circuit *)

(* stable_c: show that we never enter
  copy-mode unless the cf flag is raised *)
Theorem stable_c : forall t:nat, t > 0 ->
  c t = lo -> cf t = lo -> c (S t) = lo.
intros.
generalize if_not_reset. intro inr.
unfold doif in inr.
unfold update in inr.
unfold eval in inr.
specialize inr with (t:=t).
rewrite H0 in inr. rewrite H1 in inr.
assert(reset t = hi).
  apply reset_hi. omega. rewrite H2 in inr.
unfold const in inr.
generalize inr.
unfold Gnd.
case cmp_eq. tauto. tauto.
Qed.

(* when not in copy-mode, just pass
  along inputs to RFs *)
Theorem transparency : forall t:nat,
  t > 0 -> c t = lo ->
  a1 t = a1_in t /\ d1 t = d1_in t /\
  we1 t = we1_in t /\ a2 t = a2_in t /\
  d2 t = d2_in t /\ we2 t = we2_in t.
intros.

split.
  generalize assign_a1. intro aa1.
  unfold bus_cond_assign in aa1.
  specialize aa1 with (t:=t).
  unfold eval in aa1. rewrite H0 in aa1.
  assumption.

split.
  generalize assign_d1. intro ad1.
  unfold bus_assign in ad1.
  specialize ad1 with (t:=t).
  assumption.

split.
  generalize assign_we1. intro aw1.
  unfold assign in aw1.
  specialize aw1 with (t:=t).
  unfold eval in aw1. rewrite H0 in aw1.
  assumption.

split.
  generalize assign_a2. intro aa2.
  unfold bus_cond_assign in aa2.
  specialize aa2 with (t:=t).
  unfold eval in aa2. rewrite H0 in aa2.
  assumption.

split.
  generalize assign_d2. intro ad2.
  unfold bus_cond_assign in ad2.
  specialize ad2 with (t:=t).
  unfold eval in ad2. rewrite H0 in ad2.
  assumption.

  generalize assign_we2. intro aw2.
  unfold assign in aw2. specialize aw2 with (t:=t).
  unfold eval in aw2. rewrite H0 in aw2.
  generalize aw2.
  case cprev. tauto. tauto.
Qed.


(* prove that we terminate in a finite—<64—number of
  clock cycles, and that we have a complete transfer
  according to the extended definition—see above
  *)
Theorem termination_transfer : forall t:nat, t > 0
->
  c t = lo -> c (S t) = hi ->
  exists n:nat, cf (t+n) = lo -> n > 0 /\
    n < regs+regs /\ c (t+S n) = lo /\
  (transfer t (S n)).
intros.
exists (S(S regs)). intros.

split. omega. split. omega. split.
apply cftrans in H1.
assert (cur_read(t + S regs) = regs /\
  cur_write(t+S(S regs)) = regs /\
  c (t+S(S regs)) = hi /\
  cprev (t+S(S regs)) = hi).
  apply count_all.
    assumption. assumption. assumption. omega.
generalize if_not_reset.
intro inr.
specialize inr with (t:=t+S(S regs)).

unfold doif in inr.
unfold update in inr.
unfold eval in inr.

assert (cur_write(t+S(S regs)) = regs).
    apply H3. rewrite H4 in inr.
assert (reset (t+S(S regs)) = hi).
    apply reset_hi. omega.
rewrite H5 in inr.
unfold const in inr.
assert (regs = regs). trivial. rename H6 into rr.
apply cmp_eq_lem in rr.
rewrite rr in inr.
rewrite H2 in inr.
assert (S(t+S(S regs)) = t+S(S(S regs))). omega.
rewrite H6 in inr.
apply inr.
assumption.
assumption.

apply transfer_regs3. assumption. assumption.
  apply cftrans in H1.
    assumption. assumption. assumption.
Qed.
```

## References

[1] Defense Science Board (DSB) Study on High Performance Microchip Supply 2005 [Online]. Available: http://www.cra.org/govaffairs/images/2005-02-hpms_report_final.pdf

[2] S. Adee, "The hunt for the kill switch," *IEEE Spectrum*, vol. 45, no. 5, pp. 34–39, May 2008.

[3] D. Agrawal, S. Baktir, D. Karakoyunlu, P. Rohatgi, and B. Sunar, "Trojan detection using IC fingerprinting," in *Proc. IEEE Symp. Security and Privacy*, 2007, pp. 296–310.

[4] Y. Jin and Y. Makris, "Hardware Trojans in wireless cryptographic ICs," *IEEE Des. Test Comput.*, vol. 27, no. 1, pp. 26–35, Jan./Feb. 2010.

[5] M. Tehranipoor and F. Koushanfar, "A survey of hardware Trojan taxonomy and detection," *IEEE Des. Test Comput.*, vol. 27, no. 1, pp. 10–25, Jan./Feb. 2010.

[6] Y. Jin and Y. Makris, "Hardware Trojan detection using path delay fingerprint," in *Proc. IEEE Int. Workshop on Hardware-Oriented Security and Trust*, 2008, pp. 51–57.

[7] M. Potkonjak, A. Nahapetian, M. Nelson, and T. Massey, "Hardware Trojan horse detection using gate-level characterization," in *Proc. 46th Annu. Design Automation Conf. (DAC'09)*, 2009, pp. 688–693.

[8] R. M. Rad, X. Wang, M. Tehranipoor, and J. Plusquellic, "Power supply signal calibration techniques for improving detection resolution to hardware Trojans," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design*, 2008, pp. 632–639.

[9] F. Wolff, C. Papachristou, S. Bhunia, and R. S. Chakraborty, "Towards Trojan-free trusted ICs: Problem analysis and detection scheme," in *Proc. IEEE Design Automation and Test in Europe*, 2008, pp. 1362–1365.

[10] H. Salmani, M. Tehranipoor, and J. Plusquellic, "New design strategy for improving hardware Trojan detection and reducing Trojan activation time," in *Proc. IEEE Int. Workshop on Hardware-Oriented Security and Trust*, 2009, pp. 66–73.

[11] Y. Jin, N. Kupp, and M. Makris, "DFTT: Design for Trojan test," in *Proc. IEEE Int. Conf. Electronics Circuits and Systems*, 2010, pp. 1175–1178.

[12] M. Banga and M. S. Hsiao, "Trusted RTL: Trojan detection methodology in pre-silicon designs," in *Proc. 2010 IEEE Int. Symp. Hardware-Oriented Security and Trust (HOST)*, 2010, pp. 56–59.

[13] G. C. Necula, "Proof-carrying code," in *Proc. 24th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL'97)*, 1997, pp. 106–119.

[14] A. W. Appel, "Foundational proof-carrying code," *Found. Intrusion Tolerant Syst.*, pp. 247–256, 2003.

[15] N. A. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni, "A syntactic approach to foundational proof-carrying code," *J. Automated Reasoning*, vol. 31, pp. 191–229, 2003.

[16] A. W. Appel and D. McAllester, "An indexed model of recursive types for foundational proof-carrying code," *ACM Trans. Program. Lang. Syst.*, vol. 23, no. 5, pp. 657–683, 2001.

[17] D. Yu, N. A. Hamid, and Z. Shao, "Building certified libraries for PCC: Dynamic storage allocation," *Sci. Comput. Program.*, vol. 50, pp. 101–127, 2004.

[18] X. Feng, Z. Shao, A. Vaynberg, S. Xiang, and Z. Ni, "Modular verification of assembly code with stack-based control abstractions," in *Proc. SIGPLAN Notes*, 2006, vol. 41, no. 6, pp. 401–414.

[19] S. Drzevitzky, U. Kastens, and M. Platzner, "Proof-carrying hardware: Towards runtime verification of reconfigurable modules," in *Proc. Int. Conf. Reconfigurable Computing and FPGAs*, 2009, pp. 189–194.

[20] G. Morrisett, K. Crary, N. Glew, and D. Walker, "Stack-based typed assembly language," *J. Functional Program.*, vol. 12, no. 1, pp. 43–88, 2002.

[21] INRIA, The Coq proof assistant Sept. 2010 [Online]. Available: http://coq.inria.fr/

[22] Y. Jin, N. Kupp, and Y. Makris, "Experiences in hardware Trojan design and implementation," in *Proc. IEEE Int. Workshop on Hardware-Oriented Security and Trust*, 2009, pp. 50–57.

**Eric Love** (S'11–M'11) received the B.S. degree in electrical engineering and computer science from Yale University, New Haven, CT, in 2011. He is currently working toward the Ph.D. degree in computer science at the University of California at Berkeley.

His research interests include secure hardware devices, reconfigurable architectures, and programming language based security.

**Yier Jin** (S'07) received the Bachelor's and M.S. degrees from Zhejiang University, China, in 2005 and 2007, respectively. He is currently working toward the Ph.D. degreee in electrical engineering at Yale University, New Haven, CT.

His research interests include reliable, secure, and trustworthy ICs, crypto-processor designs, as well as dynamically reconfigurable architectures.

**Yiorgos Makris** (S'99–A'01–M'03–SM'08) received the Diploma of computer engineering and informatics from the University of Patras, Greece, in 1995, and the M.S. and Ph.D. degrees in computer science and engineering from the University of California, San Diego, in 1997 and 2001, respectively.

After spending over ten years on the Faculty of Electrical Engineering at Yale University, New Haven, CT, he moved to The University of Texas at Dallas, Richardson, TX, where he is currently an Associate Professor of Electrical Engineering, leading the Trusted and Reliable Architectures (TRELA) Research Group. His current research interests include soft-error mitigation in digital circuits, machine learning-based testing of analog/RF circuits, mitigation of hardware Trojans, as well as test and reliability of asynchronous circuits. He serves on the organizing and program committees of many conferences in the areas of test, reliability, and trustworthiness and is the program chair for the 2011 Test Technology Education Program (TTEP) of the IEEE Test Technology Technical Council (TTTC).