A Proof-Carrying Based Framework for Trusted Microprocessor IP

(Invited Special Session Paper)

Yier Jin* and Yiorgos Makris†
*Department of Electrical Engineering and Computer Science, University of Central Florida
†Department of Electrical Engineering, University of Texas at Dallas
{yier.jin@eecs.ucf.edu, yiorgos.makris@utdallas.edu}

***Abstract*—We introduce a proof-carrying based framework for assessing the trustworthiness of third-party hardware Intellectual Property (IP), particularly geared toward microprocessor cores. This framework enables definition of and formal reasoning on security properties, which, in turn, are used to certify the genuineness and trustworthiness of the instruction set and, by extension, are used to prevent insertion of malicious functionality in the Hardware Description Language (HDL) code of an acquired microprocessor core. Security properties and trustworthiness proofs are derived based on a new formal hardware description language (formal-HDL), which is developed as part of the framework along with conversion rules to/from other HDLs to enable general applicability to IP cores independent of coding language. The proposed framework, along with the ability of a sample set of pertinent security properties to detect malicious IP modifications, is demonstrated on an 8051 microprocessor core.**

I. Introduction

Integrated circuits (ICs) are used in a wide range of applications ranging from military operations to consumer electronics. Due to economic reasons and time-to-market pressure, the circuit design process has changed drastically over the last few decades. One major recent trend is the sharp increase of application-specific integrated circuits (ASICs) and FPGAs, which is supported by the widely available third-party intellectual property (IP) cores. Indeed, more and more design houses are dedicated to IP cores development nowadays. The use of IP cores helps to lower the workload of designing large-scale circuits, such as system-on-chips (SoCs), because IP cores of microprocessors and their peripheral modules supporting sophisticated instruction sets are now available. As a result, lack of extensive design experience is no longer an obstacle for circuit designers to rapidly develop systems.

However, extensive use of IP cores also brings about security problems, making circuit designs vulnerable to malicious modifications, for two reasons: (1) Because of the increasing complexity of hardware IP cores and the pressure to lower design cost, system designers are prone to treating IP cores as black-boxes which are only verified through functional testing, and (2) Even if IP cores are fully tested, module interfaces connecting IP cores and higher level logics are weak points of the entire design. Attackers can easily contaminate circuit designs by providing malicious IP cores or modifying genuine IP cores through in-the-middle attacks to steal sensitive information or even control a mission-critical device. These challenges create an urgent need for trusted third-party IP cores. However, unlike the software domain, where a large community has been long established and numerous software protection methods have been proposed, the work toward hardware IP trustworthiness is far from complete, leaving the current IC supply chain vulnerable to RT-level hardware Trojan attacks. Accordingly, military and leading IC companies compromise by only using IP cores from trusted contractors. Besides increasing design costs, this practice is also becoming less secure because of the difficulty of tracking the source of IP cores.

Towards assessing the trustworthiness of hardware IP cores obtained from untrusted third parties and protecting manufactured ICs against hardware Trojan attacks, we leverage the body of work on proof-carrying code (PCC) in the software domain and develop a proof-carrying based framework for trusted hardware IP cores. Similar to PCC, the proposed framework is constructed inside a formal environment, with both security properties and hardware circuits being either written in a formal language or translated into formal logic. We use the Coq platform as the formal environment in this paper, although other formal platforms could be used as well [1]. Our contributions in this paper are as follows:

- The proposed proof-carrying based framework, different from previously proposed proof-carrying hardware (PCH) schemes [2], [3], can assess the trustworthiness of complex IP cores and is particularly geared toward microprocessor cores. We demonstrate the use of this method in preventing and/or detecting malicious modifications on an 8051 microprocessor core.
- A new hardware description language is developed as part of the proof-carrying based framework, based on the Coq functional language. Named formal-HDL, the new HDL can represent hardware IPs in formal logic so that security properties can be directly proven for these IPs. Conversion rules are also developed between formal-HDL and other HDLs to make the framework applicable to IP cores, independent of coding languages.
- A new set of circuit security properties are also developed to enrich the circuit security property library.[1] They are used to certify the genuineness and trustworthiness of instruction sets and, by extension, to prove whether or not an acquired microprocessor is trusted. The set of security properties, if presented in English text, mean that, *for all registers and memory defined by the instruction set, (1) instructions can only correctly modify values of registers and/or memory to which they have access, according to the specification; and (2) instructions are not allowed*

[1]A property library is a collection of security properties. Specifications for each property are provided, including appropriate target circuits, security level, etc., so that both IP providers and IP users can select properties from the library rather than construct security properties themselves. Construction of a security property library is beyond the scope this paper and will be discussed in future work.

978-1-4799-1071-7/13/$31.00 ©2013 IEEE

to modify values in registers and memory besides those that the specification allows, whether in a benign or a malicious way. Powered by the set of these security properties, our framework can detect most RTL Trojans in microprocessor cores and improve the security level of critical applications.

II. Proof-Carrying Hardware

Hardware IP cores, in the form of HDL code, shares similar syntax with software programs. This similarity between the process of IP core development and software program composition implies that concepts from the software domain, such as the proof-carrying code (PCC) paradigm, can be ported to trusted IP core design. The first such attempt in trusted hardware design appeared in [3], [4], where the authors introduced Proof-Carrying Hardware (PCH) in FPGAs and reconfigurable devices. Therein, a logic-level proof was generated to demonstrate that an agreed-upon specification function is combinationally equivalent to the FPGA implementation (aka FPGA bitstream file).

Subsequently, the authors in [2] expanded this method to the ASIC domain through a RT-level Proof-Carrying Hardware Intellectual Property (PCHIP) scheme which allows for formal but computationally straightforward validation of security properties prepared by IP vendors, such that IP consumers can easily verify the trustworthiness of the delivered IP cores through an automatic property-checking process. The authors in [5] enhanced the PCHIP framework by tracking information flow inside circuit logic to prove security properties related to sensitive data leakage from contaminated IP cores.

III. Challenges for Trusted Microprocessors

The above-mentioned proof-carrying hardware methodology faces limitations when assessing trustworthiness of microprocessor cores. The PCH scheme in [3] can only be applied in FPGA bitstreams and is also limited by the requirement to specify exact Boolean functionality. The IP transaction model in [2] and information flow tracking in [5] cannot be directly implemented in hierarchical circuit designs. Furthermore, previously proposed security properties, such as data secrecy and logic level functional correctness, are either too specific or too generic and cannot be used in designing trusted microprocessors. Since both PCH and PCHIP limit themselves from being extended into the area of trusted microprocessors, a new proof-carrying based framework, particular geared toward microprocessor cores, is required to assess trustworthiness of acquired microprocessor cores. The key parts of the proposed framework are security properties for trusted microprocessors and a new formal hardware description language for representing hierarchical designs in formal logic, which are introduced in this and the next section respectively.

A. Security Properties

Because the microprocessor architecture is largely decided by its instruction set, we seek to ensure the security of microprocessors by certifying the genuineness and trustworthiness of the instruction set. We propose a set of security properties

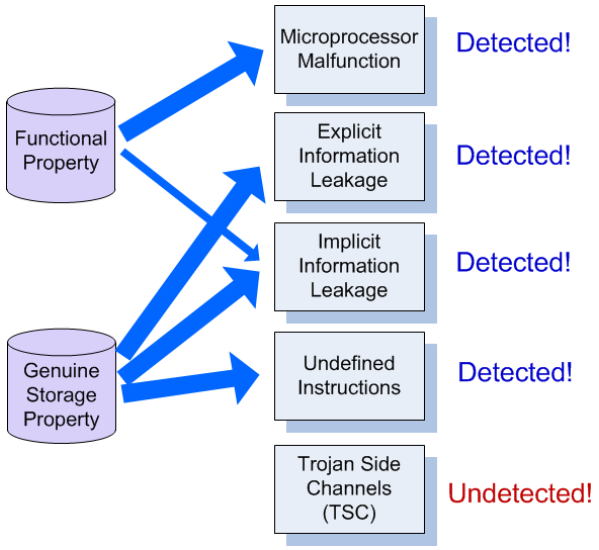

Fig. 1. Trojan Detection through Proposed Security Properties

that can be grouped into two categories, i.e., *functional properties* and *genuine storage properties*. These properties mostly focus on operations which write to registers and memory because the outcome of all instructions is in the form of register/memory value modifications.

The functional properties explicitly define the behavior of the processor when an instruction is issued and operated. It depicts the microprocessor behavior as defined by the instruction set, meaning that *instructions can only correctly modify values of registers and/or memory to which they have access, according to the specification*. At the same time, the genuine storage properties set the operation boundary of each instruction by limiting its writing privilege to registers and memory, meaning *instructions are not allowed to modify values in storage cells other than those allowed by the instruction set specification*. These two types of security properties, when combined, can provide a strong protection on microprocessors against various types of hardware Trojans.

B. Anti-Trojan Mechanism

Considering the fact that transistor level modifications can hardly be done in RTL code, hardware Trojans inserted in IP cores most often cause functional modifications. In the case of microprocessor cores, they are typically designed to perform additional functionality which attackers can rely on in order to change circuit status, steal internal information, or even take control of the microprocessor [6]. In the following, five Trojan types are covered, representing the majority of malicious functional modifications. We also discuss whether these five types will be detected by the proposed security properties. An illustrated version is provided in Figure 1.

C. Microprocessor Malfunction

The first type of inserted hardware Trojan is triggered by rare events, such as special datapath signal patterns, a sequence of control signals, or even internal registers manipulated by sequences of previous instructions, and will cause malfunction

to internal modules/datapath. In this case, whether or not instructions are involved in Trojan triggering and payload, the Trojan affects the instruction operation in an indirect way. For example, if the ALU module is modified such that the addition functionality will not correctly performed when the Trojan is triggered, any ADD instructions that need to use the ALU module will fail the proof for functional property. No matter how the Trojan is triggered or whether or not it is activated, it results in existence of a case where the ADD instructions cannot be correctly performed (or additional conditions are required ensuring that the Trojan is not activated, in order to have the ADD instruction operate correctly). This will be detected, since the proof process will explore all possible outcomes of the ADD instruction. More specifically, the functional property goal and the actual proving process cannot converge given the presence of hardware Trojans because the Trojan-infected design will only perform the addition correctly when the Trojan is not activated, an extra condition that is not available for genuine microprocessors. This condition is called a suspicious condition and will reveal the location and functionality of the inserted Trojan for further diagnosis.

D. Explicit Information Leakage

The second type of inserted hardware Trojan, in contrast to the first, does not interfere with the original functionality but tries to leak internal information. When triggered, the Trojan will store sensitive information in predefined registers or memory (we assume attackers are smart enough to identify sensitive information). Attackers can later use legitimate instructions to read out the stored data from contaminated registers or memory. Because the Trojan does not cause malfunction to any instructions and the final read-out process is performed through genuine instructions, the functional property is not violated by the Trojan-infected microprocessor. However, the Trojan's behavior violates the second type of security properties, i.e., genuine storage. Similar to the functional property, the genuine storage property also applies to every instruction to define writing privileges when each instruction is performed. Again, let us used the ADD instruction as an example, which places its calculation result in the Accumulator and also affects some flag bits in Program Status Word (PSW), including carry, overflow, and auxiliary carry flags. Note that the PC pointer is also modified in the way `PC <= PC + 2` for an ADD instruction. Other than these registers, all other registers and memory defined by the instruction set cannot be accessed if the genuine storage property holds. If the inserted Trojan, under certain circumstances, modifies values in these registers/memory, the writing operation will be easily detected when we try to prove genuine storage property for the ADD.

E. Implicit Information Leakage

The third type of Trojan is derived from the finding that our properties only cover registers/memory which are defined in the instruction set. This leaves internal registers in submodules and Trojan registers–the registers inserted by the malicious party–unprotected. The third type of Trojan, when triggered, stores internal data in those registers. We should admit that the writing operations to internal registers or Trojan registers cannot be detected through the proposed properties. However, the inserted hardware Trojan is useless for attackers because the sensitive information is leaked internally. To make the hardware Trojan practical, attackers need to move data from internal registers or Trojan registers to primary outputs or communication channels. Because all legitimate primary outputs and communication channels are defined by the instruction set, the malicious data-transferring behavior will be detected by the genuine storage property. It is also possible that the leaked data is moved to legitimate output registers, but the Trojan will then be detected when we try to prove the functional property for the instruction with which the malicious data-transferring behavior is associated.

F. Undefined Instructions

The undefined instruction is the fourth type of hardware Trojan targeting microprocessors because the instruction set does not always explicitly describe behaviors for such instructions. The mechanism to deal with undefined instructions varies among different microprocessor cores so undefined instructions may be simply ignored or cause interrupts. For the 8051 microprocessor, we employ the method that treats all undefined instructions as NOP such that for every instruction which is not defined by the instruction set, no register/memory writing operations are allowed, a behavior consistent with the genuine storage property.

G. Trojan Side Channels (TSC)

The discussion above demonstrates that most RT-level hardware Trojans can be detected when the functional property and the genuine storage property are applied, but it does not mean that the set of security properties is comprehensive in fully ensuring trustworthiness of any microprocessors. There exist hardware Trojans that can escape detection, such as Trojan side channels (TSC). TSC does not rely on primary outputs or any legitimate communication channels to leak/steal internal information and will not cause malfunction when triggered [7], so neither the functional property nor the genuine storage property can detect them. The threats from TSC indicate that more security properties are required for microprocessor cores, and we believe that a comprehensive property library introducing various security properties, as well as their application domains for both IP vendors and IP consumers, will ultimately be required.

IV. Introduction to Formal-HDL

Another challenge for assessing the trustworthiness of microprocessors is to represent large-scale hierarchical designs in a formal platform so that designs can be recognized by the formal environment but still keep their circuit level functionality and structure [8]. To solve this problem, we developed a new formal hardware description language (formal-HDL) with similar syntax and semantics to other HDLs to support hierarchical architecture. Different from previously proposed formal circuit representatives [2], [5], the formal-HDL directly supports hierarchical structures for RTL models. Conversion rules for hardware codes written in the proposed formal-HDL

and other HDLs are also developed as supplementary materials so that we can prove security properties written in other HDLs (mostly in Verilog and VHDL).[2] Lacking of compiling and synthesis EDA tools supporting formal-HDL codes is another reason for circuit code conversion between Verilog (or VHDL) and formal-HDL. Meanwhile, formal-HDL codes need to be first converted to Verilog (or VHDL) codes to be merged into the current IC manufacturing process.

The definition of formal-HDL follows similar rules to other HDLs with two major differences. First, formal-HDL is constructed in the Coq formal platform and is itself a subset of the Coq language, so any operations valid in Coq programs also apply to formal-HDL. Second, formal-HDL is defined in the way that updating of IP cores can be easily achieved based on previous version IP cores. Formal-HDL also allows users to freely develop special operations tailored to specific circuits and then reuse those operations in similar designs to make code composition flexible. The full definition of formal-HDL contains three components: basic circuit units, combinational and sequential logic, and module definition/instantiation, which are introduced below with sample codes. A combination of these components can easily represent circuit designs of various complexities.

A. Basic Circuit Units

Basic circuit units should be defined as a preliminary step, wherein signals and buses are the most important components. In formal-HDL, which is exclusively used in the digital domain, three levels of electrical values are defined, high voltage, low voltage, and unknown status to cover most synthesizable and behavioral logic. To support temporal logic, the `bus` type is defined as a function which takes one parameter, a timing variable, and returns a list of signal values. Under this definition, any attempt to acquire bus values should provide a timing variable `t` first. All circuit signals are of type `bus`, so their electrical types can only be concluded from their behavior of whether their values are modified during combinational operations or sequential operations. Inputs and outputs are also defined to be `bus` type.

```
Inductive value := lo|hi|x.
Definition bus_value := list value.
Definition bus := nat -> bus_value.
Definition input := bus.
Definition output := bus.
Definition wire := bus.
Definition reg := bus.
```

B. Combinational and Sequential Logic

Combinational and sequential logic are higher level logic descriptions constructed on top of buses. The basic blocking and non-blocking assignments are listed below with key words `assign` and `update`, meaning whether the bus value will be updated during the current clock cycle or the next clock cycle. Because we define combinational and sequential logic separately, all complex operations, such as if-else statement, case statement, etc., have two symmetric versions that share the same format with the only difference at the final assignment stage (either an `assign` or an `update` is used). For example, two types of IF blocks are introduced below for combinational or sequential logic, `aifblock` and `ifblock`.

```
Fixpoint assign (a:assignblock)(t:nat) {struct a} :=
(* Blocking assignment *)
 match a with
  | expr_assign bus_one e => bus_one t = eval e t
  | assign_useless => True
  | assign_cons a1 a2 =>
             (assign a1 t) /\ (assign a2 t)
 end.

Fixpoint update (u:updateblock)(t:nat) {struct u} :=
(* Non-blocking assignment *)
 match u with
  | (upd_expr bus exp) => (bus (S t)) = (eval exp t)
  | (updcons block1 block2) =>
            (update block1 t) /\ (update block2 t)
  | upd_useless => True
end.

Inductive aifblock :=
 | anoif : assignblock->aifblock
 | aifsimple : expr->aifblock->aifblock
 | aifelse : expr->aifblock->aifblock->aifblock.

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

C. Module Definitions/Instantiations

Module definition/instantiation is critical when dealing with hierarchical circuit structures, but it is never a problem for Verilog (and VHDL), as long as interfacing signals and their timing are correctly defined. Concerning the task of security property verification, however, treating a sub-module as a functional unit by ignoring its internal structure may cause problems. Security properties that are proven for the top level module and all its sub-modules do not guarantee that the same properties will hold for the whole hierarchical design, whereas attackers can easily insert hardware Trojans to maliciously modify the interface without violating security properties proven for all modules separately. As a result, the operation of module definition/instantiation should be defined in a way that the details of sub-modules are visible from the view of the top level module so that any security properties, if proven, remain valid for the whole design. So, we adopt a design flattening methodology by putting the full logic description of sub-modules in their interface when instantiating these modules to make the whole hierarchical circuit design transparent from the view of the top module. `module` and `module-inst` are key words for module definitions and instantiations.

V. Proof-Carrying Based Framework

Supported by the formal-HDL and the set of security properties, a proof-carrying based framework for assessing the trustworthiness of IP cores, particularly microprocessor cores is then constructed. Figure 2 outlines the basic procedure for microprocessor vendors to prepare the trusted microprocessor bundle for end-users. The vendors will first design the circuit architecture and compose HDL codes according to the microprocessor instruction set. The formal-HDL codes

[2]We only demonstrate the conversion rules from Verilog to formal-HDL since the 8051 microprocessor core we use is written in Verilog.

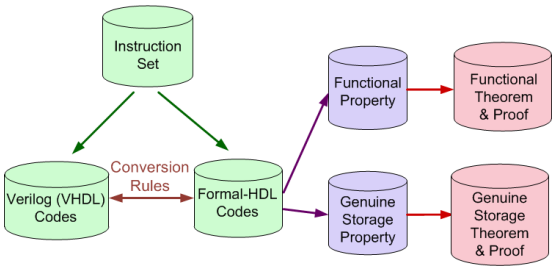

Fig. 2. Trusted Microprocessor Construction

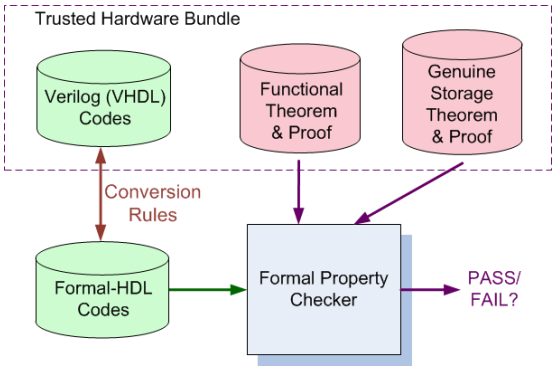

Fig. 3. Security Properties Verification

can be constructed directly or converted from codes in other HDLs according to conversion rules. Traditional functionality testing relying on testing programs will be performed for the microprocessor as the preliminary step before proving security properties. Then for each instruction, IP producers need to formalize security properties from English text into theorems in the formal platform (we use the Coq platform here). Proofs will then be constructed in the form of a list of tactics, showing that formal theorems hold for the microprocessor core. A trusted bundle containing HDL codes, formal theorems, and their proofs will then be delivered to end-users.

Upon receiving the trusted hardware bundle from IP producers, end-users will verify security properties after performing traditional functional/structural testing to assess trustworthiness of the acquired microprocessor core. The procedure for property verification is depicted in Figure 3. Because formalized security theorems are easy to read and understand, the checking process of the correctness and accuracy of the formalized theorems is not shown in Figure 3. The microprocessor core written in formal-HDL will then be loaded into the formal property checker, along with the delivered formal theorems and their proofs. A "PASS" signal from the property checker provides strong evidence that the acquired microprocessor is trusted because it is consistent with the defined security properties. However, a "FAIL" signal triggers an alarm that security properties cannot be proven; violation of security properties leads to detection of malicious modifications in the scope of trusted circuit designs.

VI. Demonstrations on an 8051 Microprocessor

To better illustrate the working procedure of the proposed proof-carrying based framework shown in Figures 2 and 3 and to demonstrate the effectiveness of the proposed security properties in assessing the trustworthiness of microprocessor cores, an open source pipelined 8051 core written in Verilog is used as the experimental vehicle [9]. The block diagram of the 8051 microprocessor is shown in Figure 4, which has 4KB on-chip program memory, 128B on-chip data RAM, and supports 64KB memory address space. Most instructions will be finished in 3 clock cycles. Details about the 8051 instruction set can be found in [10]. Although the 8051 is of relatively small size, this pipelined version of 8051 core contains the instruction decoding, execution, and memory access procedures, which make it a good starting point for our research in designing trusted microprocessors. Note that even though the proposed methodology is valid for more complicated processors, the proposed framework cannot be directly applied to modern processors. The updated framework will be addressed in our later work.

A. Functional Property

The functional property, meaning *instructions can only correctly modify values of registers and/or memory to which they have access, according to the specification*, ensures the functional correctness of all instructions. It serves as the basis for all other security properties because it is meaningless to prove security properties on a malfunctioning IP core. Because security properties apply to all instructions, including undefined instructions that are treated as NOP, each instruction has its own version of formalized theorem. Taking one ADD instruction, `ADD A, #data` for example, it adds an 8-bit number `#data` to the Accumulator and stores the calculated result back to the Accumulator in the way that `(Accumulator) <- (Accumulator) + #data`. The carry, auxiliary-carry, and overflow flags of the PSW are also set accordingly. The functional property of the instruction `ADD A, #data` is proved in two parts: (1) the correctness of the 8-bit addition result stored in the Accumulator and 1-bit carry flag; and (2) the correctness of auxiliary carry flag and overflow flag. The formalized functional theorem of the first part is shown below, where the instruction itself is converted to pre-conditions while a correctly performed addition operation is converted to the post-condition. The functional theorems for the second part follow a similar style.

```
Theorem ADD_A_DATA :
forall t:nat, forall dt:bus_value,(length dt)=8 ->
state_decoder t = lo::lo::nil ->
(op1 t) = OC8051_ADD_C ->
(op2 t) = dt ->
listvalue2nat (data_out (S (S t))) +
    (listvalue2nat (desCy (S t)))*exp2 8 =
listvalue2nat (acc (S t)) + listvalue2nat dt.
```

More specifically, `op1` and `op2` indicate the instruction type and the immediate data in the instruction, respectively. The `length dt = 8` requires the immediate data to be 8-bit long. Because we prove security properties for each instruction separately, NOPs are inserted before and after the target instruction. For example, one of the pre-condition, `state_decoder t = lo::lo::nil`, exclude the situation that a JUMP instruction is previously issued. The post-condition is derived ensuring the correctness of values in

the Accumulator and the carry flag. Proofs of this functional theorem are omitted here due to page limit.

B. Genuine Storage Property

The genuine storage property *–instructions are not allowed to modify values in registers and memory besides those the specification allows, whether in a benign or a malicious way–* provides a higher level protection of a microprocessor from hardware Trojan attacks. In the 8051 microprocessor, special function registers (SFR), internal memory, and external memory are the storage elements defined by the instruction set. Other storage cells, such as internal registers and Trojan registers, are not considered by the genuine storage property but this "omission" does not hamper the Trojan detection capability as we discussed in Section III.

We use the `ADD A, #data` instruction again to demonstrate how genuine storage property is formalized and then proven to prevent insertion of malicious modifications in the 8051 microprocessor core. Defined by the instruction set, values in the Accumulator and the PSW will be modified (note that we do not consider the PC pointer because it will be updated in every clock cycle).

```
Definition add_a_data_sfr :=
    OC8051_SFR_ACC::OC8051_SFR_PSW::(nil)::nil.

Theorem sfr_protection :
 forall t:nat, forall lv:bus_value,
 state_decoder t = lo::lo::nil ->
 op1 t = OC8051_ADD_C ->
 op1 (S t) = OC8051_NOP ->
 op1 (S (S t)) = OC8051_NOP ->
 In lv ((bv_depend (wr_addr_m (S t)) (wr_r (S t)))
   ::(bv_depend (wr_addr_m (S (S t)))
                (wr_r (S (S t))))::nil) ->
 In lv add_a_data_sfr.
```

The `add_a_data_sfr` defines the list of registers and memory addresses that will be updated when the ADD instruction is performed. As we mentioned earlier, NOP instructions are issued before and after the ADD instruction, as shown in pre-conditions of `op1 (S t)` and `op1 (S (S t))`. `In lv ( ... ) -> In lv add_a_data_sfr` means that if any registers/memory addresses are write-enabled during the operation of ADD instruction, these addresses must be included in the address list `add_a_data_sfr`. Under the proposed proof-carrying based framework, only if formalized functional theorems and genuine storage theorems for all instructions are proven can we trust the acquired 8051 microprocessor core.

VII. Conclusions

The increased use of IP cores in ASIC and FPGA applications brings about security problems because of the likelihood that RT-level hardware Trojans may be inserted into third-party IP cores and cause malfunction to the whole design through the IP core instantiation process. Furthermore, Trojan-infected IP cores also invalidate most of the previously proposed post-silicon Trojan detection methods because many of these methods are constructed upon the assumption that golden chips are available, which does not hold under the threats of RT-level hardware Trojans.

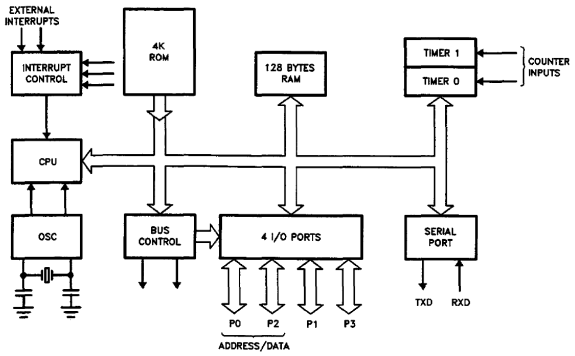

Fig. 4. Block Diagram of the 8051 Microprocessor [10]

To ensure the security of third-party IP cores, especially microprocessor cores, a new proof-carrying based framework which borrows concepts from software proof-carrying code (PCC) is proposed herein. The framework leverages two hardware security properties, the functional property and the genuine storage property, to prove functional correctness and legitimate register/memory access for each instruction. If both security properties are proven to hold, we can exclude most of RT-level hardware Trojans from the threat list of a microprocessor core. Further work will include the development of a property library so that both IP providers and IP users can pick properties from the library rather than construct security properties themselves, the expansion of the current framework to cover modern processor cores, and the development of automation tools for proof construction.

References

[1] INRIA, "The coq proof assistant," September 2010, `http://coq.inria.fr/`.

[2] E. Love, Y. Jin, and Y. Makris, "Proof-carrying hardware intellectual property: A pathway to trusted module acquisition," *IEEE Transactions on Information Forensics and Security*, vol. 7, no. 1, pp. 25–40, 2012.

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

[4] S. Drzevitzky and M. Platzner, "Achieving hardware security for reconfigurable systems on chip by a proof-carrying code approach," in *6th International Workshop on Reconfigurable Communication-centric Systems-on-Chip*, 2011, pp. 1–8.

[5] Y. Jin and Y. Makris, "Proof carrying-based information flow tracking for data secrecy protection and hardware trust," in *IEEE 30th VLSI Test Symposium (VTS)*, 2012, pp. 252–257.

[6] Y. Jin, M. Maniatakos, and Y. Makris, "Exposing vulnerabilities of untrusted computing platforms," in *Computer Design (ICCD), IEEE 30th International Conference on*, 2012, pp. 131–134.

[7] L. Lin, M. Kasper, T. Guneysu, C. Paar, and W. Burleson, "Trojan side-channels: Lightweight hardware Trojans through side-channel engineering," in *Cryptographic Hardware and Embedded Systems*, vol. 5747 of *LNCS*, pp. 382–395. Springer-Verlag Berlin, 2009.

[8] Thomas Braibant, "Coquet: A coq library for verifying hardware," in *Certified Programs and Proofs*, Jean-Pierre Jouannaud and Zhong Shao, Eds., vol. 7086 of *Lecture Notes in Computer Science*, pp. 330–345. Springer Berlin Heidelberg, 2011.

[9] *http://www.opencores.org/projects*.

[10] Intel Corporation, "Intel mcs51 family user manual," 1981.