Generating Architecture-Level Abstractions from RTL Designs for Processors and Accelerators Part I: Determining Architectural State Variables

Yu Zeng, Bo-Yuan Huang, Hongce Zhang, Aarti Gupta, Sharad Malik
Princeton University, Princeton, USA
{yuzeng, byhuang, hongcez}@princeton.edu, aartig@cs.princeton.edu, sharad@princeton.edu

***Abstract*—Today's Systems-on-Chips (SoCs) comprise general/special purpose programmable processors and specialized hardware modules referred to as accelerators. These accelerators serve as co-processors and are invoked through software or firmware. Thus, verifying SoCs requires co-verification of hardware with software/firmware. Co-verification using cycle-accurate hardware models is often not scalable, and requires hardware abstractions. Among various abstractions, architecture-level abstractions are very effective as they retain only the software visible state. An Instruction-Set Architecture (ISA) serves this role for processors and such ISA-like abstractions are also desirable for accelerators. Manually creating such abstractions for accelerators is tedious and error-prone, and there is a growing need for automation in deriving them from existing Register-Transfer Level (RTL) implementations. An important part of this automation is determining which state variables to retain in the abstract model. For processors and accelerators, this set of variables is naturally the Architectural State Variables (ASVs) – variables that are persistent across instructions. This paper presents the first work to automatically determine ASVs of processors *and* accelerators from their RTL implementations. We propose three novel algorithms based on different characteristics of ASVs. Each algorithm provides a sound abstraction, i.e., an over-approximate set of ASVs. The quality of the abstraction is measured by the size of the set of ASVs computed. Experiments on several processors and accelerators demonstrate that these algorithms perform best in different cases, and by combining them a high quality set of ASVs can be found in reasonable time.**

***Index Terms*—Hardware verification, accelerators, architectural abstraction, taint analysis, model checking**

I. Introduction

Modern systems-on-chips (SoCs) consist of not only general/special purpose programmable processors and peripheral devices, but also specialized hardware modules referred to as accelerators [1]. These accelerators often function as co-processors that are invoked through software or firmware. Verifying such SoCs is challenging especially for system-level properties involving both hardware (HW) and software/firmware (SW/FW). Co-verification of cycle-accurate hardware models with software using co-simulation is slow and their formal co-verification not scalable. Thus co-verification often uses high-level abstractions of hardware designs [2]–[5].

For programmable processors, an Instruction-Set Architecture (ISA) is a very effective abstraction for hardware. It hides lower-level implementation details and exposes only the architecture-level behavior that is visible to software. Thus, the ISA abstraction is widely used for verifying software/firmware [3], [6]–[9]. However, accelerators often lack such ISA specifications. More recently, a general architecture-level abstraction has been proposed for both processors and accelerators, called the Instruction-Level Abstraction (ILA) [10]. ILAs have been used for SoC functional and security verification with accelerators [11], [12]. However, the ILA-based abstractions are usually created manually, which is tedious and error-prone. There has been some work on using semi-automatic template-based synthesis for creating ILAs [11], but expertise is still needed for writing the template, and the template size is comparable to the final ILA size, limiting the value of this automation. Often the Register-Transfer Level (RTL) implementation of a hardware design is available, and it is desirable to automatically derive an architecture-level abstraction from this. These abstractions can be consistent with RTL by construction, and thus can be used with high confidence for system-level verification.

An important part of generating the architecture-level abstractions is to determine which state variables to retain in the abstract model and which state variables to abstract away. For processors, the set to retain is naturally the *Architectural State Variables (ASVs)*, the variables that are persistent across instructions. For example, ASVs for x86 include general-purpose registers, FLAGS register, program counter, etc. The recent ILA work has a similar notion of ASVs for accelerators as the state variables that are persistent across the commands presented at the accelerator interface (these commands are the accelerator instructions in the ILA). *Thus, extracting ASVs from RTL implementations is the first and critical step in determining architecture-level abstractions for processors* [1] *and accelerators – this step is the focus of our paper.* (The second part of deriving this abstraction: automatically generating the state update function for the ASVs for each instruction, is beyond the scope of this paper.)

Specifically, we propose three algorithms for extracting ASVs based on different ASV characteristics. Each algorithm provides an over-approximate set of ASVs, i.e., the inclusion of the actual ASVs is guaranteed. This ensures that the resulting abstraction is sound. The quality of the abstraction is measured by the size of the set of ASVs extracted – smaller is better. We have implemented these algorithms and present an experimental evaluation on several general-purpose processors and accelerators. Our experiments demonstrate that different algorithms perform best in different cases, i.e., there is no clear winner, and by combining them a high quality set of ASVs can be found in reasonable compute time.

Many previous works for system-level verification have also used hardware abstractions, such as Hardware Transactions [4], Transaction-Level Modeling (TLM) [13], [14], and Extended Finite State Machines (EFSM) [15]. However, only some have focused on architecture-level abstractions [8]–[10], [16], [17], which is essential for reasoning about software-visible behavior of hardware. A detailed comparison with these related efforts is described later (§V).

The contributions of this paper are as follows:

This work was supported in part by the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA; by the DARPA POSH program; and by NSF XPS Grant No. 1628926.

[1]In our experience we have seen the use of legacy processors/micro-controllers in SoCs for which the ISA is not available – these can also benefit from the proposed automation.

- To the best of our knowledge, this is the first work towards automatic generation of architecture-level abstractions for processors and accelerators from RTL designs. Although we focus only on the first step of automatically determining ASVs, this is a critical step and itself a challenging problem. We propose three novel algorithms for extracting ASVs which make use of different ASV characteristics, and discuss their trade-offs.
- We provide an experimental evaluation of these algorithms using six non-trivial designs of processors and accelerators from the OpenCores [18] website or published designs. We discuss in detail the wins and fails for each algorithm, and give guidelines for choosing among them, as well as combining them effectively.

II. Preliminaries

A. Target Hardware

Our proposed method can be applied to both general-purpose processors and accelerators. We focus on Loosely-Coupled-Accelerators (LCA) [19], [20], which are widely used in modern SoCs and often use a Memory-Mapped Input/Output (MMIO) interface. Two features of LCAs are especially important for our algorithms: 1) a clear definition of when an instruction is valid, and 2) a clear specification of the completion condition for each instruction, e.g., by reading the output or using a polling instruction.

B. Background Techniques

Architectural State Variables (ASVs) are the set of variables that are persistent across instructions, i.e., the values of the ASVs at the start of an instruction are sufficient to determine future observable state changes. In this section we provide a brief description of some standard techniques used in our algorithms for determining ASVs.

a) Self-composition: Self-composition is a popular technique for verifying security properties of programs [21], [22]. For example, to check for information leaks, two copies of a program are created with public inputs being the same, while private inputs are unconstrained. Then the two programs are checked to see if the public outputs are the same in all possible executions, i.e., there are no information leaks from private inputs to public outputs. In our algorithm, self-composition is used to check if a subset of registers can affect the outputs of the hardware design.

b) Taint analysis: Taint analysis is used in applications such as information-flow analysis [23]. In taint analysis, a taint is *introduced* on some input variables of interest, and each statement in the program *propagates* the taint to possibly other variables in the program (depending on the semantics of the statement), and finally the output variables of interest are *checked* to determine whether they are tainted or not. As with self-composition, we use taint-analysis to check if a subset of registers can affect the outputs of the hardware design.

c) Liveness analysis: Liveness analysis is widely used in compiler optimizations [24]. A variable is live at some point in a program if its value may be read before the next time the variable is written to. An ASV has the property that there exists some instruction where the ASV is live at the start of that instruction, and it is read by that instruction before any writes to it by that instruction. Therefore, we use liveness analysis in one of our algorithms.

d) Symbolic verification using model checking: For hardware models of processors and accelerators, "instructions" are stimuli at the input signals that are provided by the environment and can possibly take any allowed value in every clock cycle. Determining the ASVs requires considering all possible instruction sequences, which makes it very challenging. Symbolic model checking [25] allows exploring all possible instruction sequences and is used in our algorithms.

III. Algorithms for Determining ASVs

We propose the following three algorithms for determining ASVs.

1) **AsvAOD Algorithm:** Our first algorithm checks whether a subset of state variables is sufficient to determine future output behavior using the notion of architectural observational determinism (defined in §III-A). This is expressed as a 2-hyperproperty, i.e., a property on two different executions of the same program. This 2-hyperproperty can be checked using model checking on a self-composition of the design using known techniques [21].
2) **AsvTA Algorithm:** The second algorithm uses taint analysis to determine if a specific register can affect any output in the future. We use unbounded model checking to perform symbolic taint analysis. This approach avoids two copies of the design needed for self-composition in the AsvAOD algorithm. However, each register is checked individually, and the proof effort is not reused across registers.
3) **AsvLA Algorithm:** The third algorithm combines liveness analysis with unbounded model checking. In this algorithm all registers can be checked simultaneously.

We note that there may exist *different* minimal sets of ASVs. As a simple example, consider a design with two registers with a fixed relationship between their values, e.g., they always store some value n and $n+1$, respectively. Thus, either one would suffice as an ASV, with the other being a dependent register – both are not needed.

We designed all three algorithms to identify an over-approximation of the set of ASVs, i.e., the computed ASV set is guaranteed to contain all ASVs, and it may sometimes contain non-ASVs also. This is acceptable for the purpose of deriving a *sound* high-level abstraction. The quality of the abstraction is determined by the size of the set of computed ASVs – smaller is better.

The following sections describe the details of these three algorithms. All three algorithms require users to specify the set of valid input commands, i.e., instructions. The last two algorithms further require the completion condition for each instruction.

A. AsvAOD: Checking Architectural Observational Determinism

An RTL design is an FSM $M = (S, I, O, R_0, N)$, where S is the set of state variables (e.g., register or memory), I is the set of input signals, O is the set of output signals, R_0 is the initial reset state, and $N : (S \times I) \to (S \times O)$ is the transition function. For a subset of state variables $A \subseteq S$, we say it preserves *architectural observational determinism* (AOD) if the following hyperproperty holds:

$$\begin{aligned}\forall t, t' \in T, n, n' \in \mathbb{N} : (t[n] =_A t'[n']) \wedge (t[n..] \approx_I t'[n'..]) \\ \implies (t[n..] \approx_O t'[n'..]), \qquad (1)\end{aligned}$$

where T is the set of infinite traces of M. State equivalence relation $s =_x s'$ holds whenever states s and s' are indistinguishable with respect to variable set x; trace equivalence relation $t \approx_x t'$ holds whenever traces t and t' are indistinguishable with respect to variable set x [26]. In other words, a state variable subset preserves AOD if it is sufficient to determine future architectural behavior (e.g., output values) for all possible inputs. Thus, it is necessary and sufficient for a set of ASVs to preserve AOD. We formulate the problem of finding the ASVs as an optimization problem of minimizing the state variable subset that preserves AOD, i.e., satisfies hyperproperty 1 above.

Algorithm 1 describes the optimization procedure. Given the design M and environment E that constrains inputs to valid instructions, we first characterize the design and generate groups of state variables (*pool*) using cluster analysis in line 1. Each group in *pool* contains

Algorithm 1: AsvAOD: Minimize AOD-preserving var subset

```
   Input: Design $M$ and environment $E$
   Output: AOD-preserving variable subset $A$
1  $pool, db \leftarrow$ Design_Analysis$(M, E)$
2  $A \leftarrow S$
3  while $pool \neq \emptyset$ do
4      $candid \leftarrow$ Search_Heuristic$(A, pool, db)$
5      $res, db \leftarrow$ AOD_Check$(M, E, A \setminus candid, db)$
6      if $res = true$ then
7          $A \leftarrow A \setminus candid$
8          $pool \leftarrow \{p \setminus candid \mid \forall p \in pool\}$
9      else cex or undetermined
10         $pool \leftarrow pool \setminus \{candid\}$
```

variables with certain similarity, and *db* is the database holding the information learned throughout the optimization. In line 2, we start with a trivial solution, the complete set S, which satisfies hyperproperty 1 for all deterministic designs. To further minimize the subset, we iteratively check the candidate group selected by the search heuristic, which ranks the groups based on the current progress as well as the previous results (line 4). The main AOD checking (line 5) is done through model checking and self-composition, a prominent way of checking 2-safety properties [21], [22]. Based on the result of the checking, the candidate groups and the AOD-preserving subset are updated accordingly. The result *res* is undetermined if the AOD check using model checking times out. In this case the final result can be an over-approximation of the set of ASVs.

We now discuss the salient aspects of this algorithm.

1) Design analysis: In preparation for the optimization, we analyze the design both syntactically and semantically. This is to group state variables that are *likely* to have similar behavior (i.e., AOD preserving or not). For example, we collect the relations between the state variables based on the design hierarchy from the RTL description; we distinguish state variables based on their shortest distance-to-output via model checking; we also synthesize invariants for state variables using their value ranges and possible encodings. Based on these characteristics, we generate a number of state variable groups through cluster analysis that we will check for AOD.

2) Search heuristic: We check AOD for the generated groups in rank order. The group rank depends on various aspects, such as group size and the correlation between the group and other previously checked variables. Group size has the highest importance, as with larger groups the AOD checks are more aggressive in minimizing the subset – failed AOD checks (getting counterexamples) are usually low cost, while the benefit is saving more potential iterations.

3) AOD check: To check if a variable subset is AOD-preserving, we use self-composition, by making two copies of the design. However, model checking this property, starting from the reset state R_0, requires aligning the two copies with uneven trace prefixes. To avoid that, we over-approximate the trace prefixes by having an arbitrary initial state. We check only the trace suffix equivalence, and no alignment is required. In response to the false positives caused by unreachable starting states, we use a CEGAR-style loop to refine the initial state [27]. To get a coarser refinement that blocks more unreachable states, we generalize the unreachable states in the spurious counterexamples based on our design analysis and the AOD-preserving variables.

B. *AsvTA: Taint Analysis based Algorithm*

Our second algorithm also determines ASVs by determining whether the value of a register at the start of an instruction can affect any output value in the future. However, unlike the AsvAOD algorithm, it employs taint analysis and avoids constructing a self-composition. We introduce a taint on an individual register at the beginning of an instruction, and use taint analysis to determine if it can affect any future output. This is a necessary condition for that register to be an ASV. Thus, if it does not affect any future output, this register is not an ASV. However this is not a sufficient condition, e.g., our taint analysis is done one register at a time, and will therefore mark all dependent registers (like the example with n and $n+1$ discussed earlier) as ASVs, even when only one of them would suffice. So if a register does affect some future output, it is labeled as a *potential* ASV.

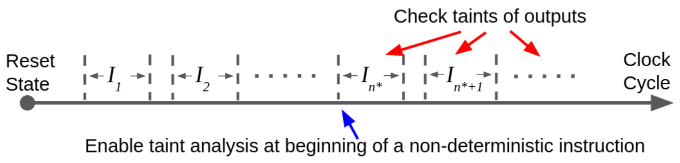


Fig. 1: Sequence of Instructions for Taint Analysis

Fig. 1 shows the verification setup in our taint-based algorithm. Note that the design starts from the reset state and executes a nondeterministic number of symbolic instructions, shown as $I_1, I_2, \ldots, I_{n^*}, I_{n^*+1}, \ldots$ in the figure. The taint analysis starts after an arbitrary instruction, say I_{n^*}, in the figure.

Algorithm 2: AsvTA: Taint Analysis based Algorithm

```
   Input: Design $M$ and environment $E$
   Output: Potential ASV set $A$
1  $A \leftarrow \{\}$
2  $assum \leftarrow$ Generate_Instruction_Assum$(E)$
3  $asserts \leftarrow$ Generate_Output_Asserts$(M)$
4  foreach register $r \in M$ do
5      $assum' \leftarrow assum \wedge$ Generate_Taint_Assum$(r)$
6      $res \leftarrow$ Output_Taint_Check$(M, E, assum', assert)$
7      if $res = tainted$ then
8          $A \leftarrow A \cup \{r\}$
9      end
10 end
```

The pseudo-code for our taint analysis algorithm, referred as AsvTA, is shown in Algorithm 2. It works as follows:

- **Line 2.** We add an assumption that all previous instructions have finished before the start of I_{n^*}. This is to make sure that we are checking the influence of a register r's value over outputs of current and later instructions.
- **Line 3.** We add assertions that taints of outputs are 0, i.e., they are not affected by the chosen register.
- **Line 4.** We iterate over each register r.
- **Line 5.** We set r's taint to be positive at the start of I_{n^*}. Similar to previous work [28], the added taint variables are of the same width as design variables, so that bit-level operations can be tracked. As a technical detail, its taint is set to positive only when its value is not same as its reset value, because an ASV should have reachable values other than its reset value.

- **Line 6.** We check all output assertions using a model checker, which performs unbounded analysis.
- **Lines 7 to 9.** If all assertions are proven to be true, then A is not an ASV. Otherwise, register r is added to the ASV set A.

Note that our algorithm uses a model checker to symbolically check if outputs are tainted, similar to a prior work that also uses taint analysis but for a different application [28]. The advantage with model checking is that taints can be propagated in a path-sensitive manner, unlike typical static analysis which is usually path-insensitive. This improves the precision of the taint analysis, thereby improving the accuracy of the set of ASVs that we identify.

C. AsvLA: Liveness Analysis based Algorithm

Since AsvTA performs a separate taint analysis per register, the proof effort is not reused across different registers. To improve upon this aspect, our third algorithm, called AsvLA, checks all registers together. It is based on liveness analysis, a standard data-flow analysis used often in compiler optimizations [24]. An ASV has the property that there exists some instruction where the ASV is *live* at the start of that instruction, and it is *read by that instruction before it is possibly written to* by that instruction. Note that such liveness is a necessary condition for a variable to be an ASV, but not a sufficient condition, e.g., the value of a register r may be used for a redundant update to another register, in which case it does not affect architectural behavior. (More examples can be found in §IV-C.) Thus, the set of live registers/variables at the start of instructions is an *over-approximation* of the set of ASVs.

1) Illustrative example: We first give an overview of our AsvLA algorithm on a running example.

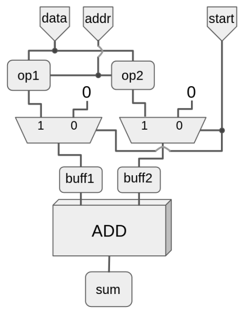


Fig. 2: A simple circuit for ADD operation

Figure 2 shows a simple circuit for an ADD operation with three input variables (*addr*, *data*, *start*) and five registers (*op1*, *op2*, *buff1*, *buff2* and *sum*). The two operands are stored in *op1* and *op2*, and the result in *sum*. (Other input/output variables, such as clock, reset, input valid signal and carry bit are not shown for simplicity.) The adder circuit has 3 instructions: (1) `UPDATE_OP1`: sets *addr* to 1 and assigns *data* to *op1*, (2) `UPDATE_OP2`: sets *addr* to 2 and assigns *data* to *op2*, (3) `START_COMPUTATION`: sets *start* to 1 and *addr* to 0, selects *op1* and *op2* through the two Muxes, and assigns their values to *buff1* and *buff2*. The ADD module reads values from *buff1* and *buff2* and performs the ADD operation.

We can easily determine that both *op1* and *op2* are ASVs. The value of *op1* at the beginning of `START_COMPUTATION` affects the computation result; similarly for *op2*. Also, they satisfy the definition of live variables at the start of the instruction: in the execution of `START_COMPUTATION`, they are read but not written. In contrast, *buff1* and *buff2* are not ASVs. During the execution of `START_COMPUTATION`, their values are first over-written by values of *op1* and *op2*, and these updated values are read by the ADD module. Thus, they do not satisfy the definition of live registers – they are written to before being read by ADD.

This example illustrates that a variable's liveness at the start of an instruction depends on the relative order in which the instruction reads from and writes to that variable. To determine this relative order, we use a *modified taint analysis method* to record the cycles in which each register is written to or read from by the instruction. In particular, we use three kinds of auxiliary variables – write taints, read flags, and signatures – to record the timing of reading and writing registers. We then construct properties on these auxiliary variables and check them using model checking to determine which registers are read before being possibly written. This set of registers is reported as ASVs.

Algorithm 3: AsvLA: Liveness Analysis based Algorithm

```
Input: Design $M$ and environment $E$
Output: Potential ASV set $A$
1  $A \leftarrow \{\}$
2  $M' \leftarrow$ Yosys_Simplify($M$)
3  foreach statement $s \in M'$ do
4      $aux \leftarrow$ Generate_Auxiliary_Variables_Statements($s$)
5      $M' \leftarrow M' \cup \{aux\}$
6  end
7  $assum \leftarrow$ Generate_Input_Assum($M', E$)
8  $assert \leftarrow$ Generate_Register_Asserts($M'$)
9  $all\_results \leftarrow$ Liveness_Check($M', E, assum, assert$)
10 foreach $result\_of(r) \in all\_results$ do
11     if $result\_of(r) = live$ then
12         $A \leftarrow A \cup \{r\}$
13     end
14 end
```

2) Practical Implementation: The pseudo-code for a practical implementation of our AsvLA algorithm is shown in Algorithm 3. It works as follows:

- **Line 2.** *Source code simplification.* We use Yosys [29] to read in Verilog RTL designs. Yosys generates an equivalent, simplified Verilog design (with a smaller subset of logical/arithmetic operators and conditional logic).
- **Lines 3 to 6.** *In-line code instrumentation.* We instrument the RTL code for a taint-like analysis to track liveness of variables. The simplified RTL design makes it easier to instrument it with auxiliary variables and statements that track the reads/writes of the design variables. For each RTL statement, the instrumented code is added in-line. Details are described later in §III-C3.
- **Line 7.** *Assumptions generation.* Based on the set of valid instructions and finish conditions provided by the user, we create a set of assumptions for input variables and input taints for use in the model checking step. Assumptions on input variables symbolically specify the set of valid instructions. Assumptions about input taints specify that they can be positive only when input variables are constrained by instructions. We also specify the finish conditions for instructions as predicates on the RTL design variables.
- **Line 8.** *Assertions generation.* An assertion is generated for each register that it is never read before being written.
- **Line 9.** *Model checking.* We use a model checker to check the assertions. Since modern model checking tools can check

multiple assertions simultaneously, this allows sharing the proof effort across the check for multiple registers.

- **Line 10 to 14.** If a counterexample is found for the assertion of register r, then r is live and is added to the set of ASVs A.

In the assumptions generation step, we assume that there is only one instruction executing at a time even if the actual RTL design may have multiple instructions in flight due to pipelining etc. This is because, in general, instructions are specified as being atomic, i.e., other instructions see the ASVs only at the end of instruction execution and thus this assumption does not change the set of ASVs determined. (There is one exception to this in our case studies – the `start_encrypt` instruction for the AES accelerator sets a "busy" flag which can be read by a polling instruction that reads this flag before the encrypt instruction completes. Non-atomic instructions such as this one can be determined from the design documentation, and we permit other instructions to overlap with these non-atomic instructions.) Also, similar to the AsvTA algorithm, if the value in a register at the start of its instruction is the reset value, then it is not considered to be live at the start of the instruction.

The model checking step considers a sequence of instructions shown in Fig. 3. Here, I_c symbolically represents any valid instruction that is checked to see if any register at its start is read by it before being possibly written. The design starts from the reset state, and executes a nondeterministic number of instructions before I_c. This implicitly considers all possible *reachable* start states for I_c for the purpose of determining ASVs. Note that we leverage model checking for constraining the start state for an arbitrary instruction I_c – this avoids the need to provide reachability invariants.

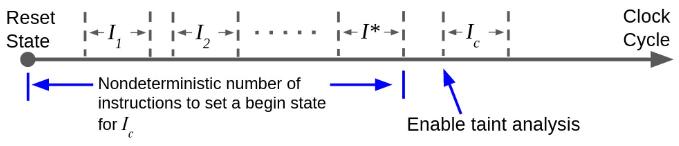


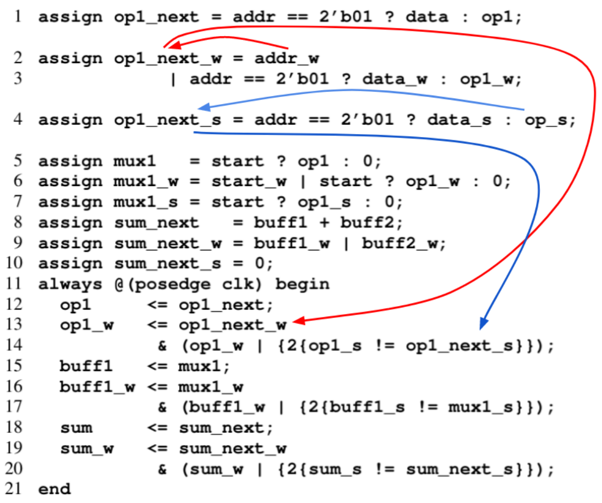
Fig. 3: Sequence of instructions for Liveness Analysis

3) Tracking Read/Write timing using Auxiliary Variables: We continue with the motivating example in Fig. 2 to show how auxiliary variables are added, along with code instrumentation, to track the timing of reading and writing registers in our liveness analysis. The code in Fig.4 shows the partial Verilog code for the circuit shown in Fig. 2, including the auxiliary variables and added code.

Write Taints: The variables ending with "_w" are write taints. Each variable has its own write taint which has the same bitwidth and same datatype (wire or reg). All write taints of register type are initialized to be `0`. In the cycle when the instruction being checked (I_c in Fig. 3) is issued, write taints of input variables become positive. Then write taints are propagated as illustrated with the code in Fig. 4.

Signatures: We use auxiliary variables called *signatures*, with the suffix "_s", to determine if a register is updated with its old value when written to (and thus not really changed). This is done by checking if a register's next state value has the same signature as itself. Each register is given a unique non-zero constant signature (e.g., `op1_s` is 1, `op2_s` is 2, etc.) Signatures can be selected to pass through conditional logic (e.g., `Ite` and `case` statements), as shown in line 4 and line 7 in Fig 4. For arithmetic or logical statements, signatures are simply blocked as shown in line 10.

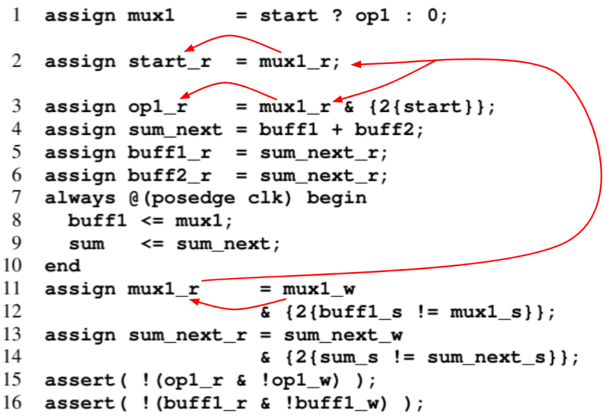
With only write taints and no signatures, registers that are simply assigned their old values can be falsely tainted by the taints from conditional variables. For example, when `UPDATE_OP2` (`addr = 2'b10`) is issued with positive `addr_w`, write taints are propagated



```
 1 assign op1_next = addr == 2'b01 ? data : op1;

 2 assign op1_next_w = addr_w
 3              | addr == 2'b01 ? data_w : op1_w;

 4 assign op1_next_s = addr == 2'b01 ? data_s : op_s;

 5 assign mux1   = start ? op1 : 0;
 6 assign mux1_w = start_w | start ? op1_w : 0;
 7 assign mux1_s = start ? op1_s : 0;
 8 assign sum_next   = buff1 + buff2;
 9 assign sum_next_w = buff1_w | buff2_w;
10 assign sum_next_s = 0;
11 always @(posedge clk) begin
12   op1     <= op1_next;
13   op1_w   <= op1_next_w
14              & (op1_w | {2{op1_s != op1_next_s}});
15   buff1   <= mux1;
16   buff1_w <= mux1_w
17              & (buff1_w | {2{buff1_s != mux1_s}});
18   sum     <= sum_next;
19   sum_w   <= sum_next_w
20              & (sum_w | {2{sum_s != sum_next_s}});
21 end
```

Fig. 4: Example of Instrumented Code with Write Taints and Signatures (`addr`, `data` and `start` are input variables)

to line 13, as shown by the red arrow. `op1_w` is initially zero. Without line 14, `op1_w` would be positive in the next cycle even though it is not written. With signatures propagated as shown by the blue arrow, `op1_s == op1_next_s`, so `op1_w` is not falsely tainted in the next cycle. The use of `op1_w` in line 14 is to prevent loss of write taints after it is tainted.



```
 1 assign mux1      = start ? op1 : 0;

 2 assign start_r   = mux1_r;

 3 assign op1_r     = mux1_r & {2{start}};
 4 assign sum_next  = buff1 + buff2;
 5 assign buff1_r   = sum_next_r;
 6 assign buff2_r   = sum_next_r;
 7 always @(posedge clk) begin
 8   buff1 <= mux1;
 9   sum   <= sum_next;
10 end
11 assign mux1_r      = mux1_w
12                    & {2{buff1_s != mux1_s}};
13 assign sum_next_r = sum_next_w
14                    & {2{sum_s != sum_next_s}};
15 assert( !(op1_r & !op1_w) );
16 assert( !(buff1_r & !buff1_w) );
```

Fig. 5: Example Instrumented Code for Read Flags and Assertions

Read Flags: Example code with read flag variables for tracking reads is shown in Fig. 5, where variables ending with "_r" are used to track when variables are read to update registers. All read flags are of type wire. When a register is updated in one cycle, the variables used in its update function in the previous cycle are considered "read". Propagation via read flags originates in non-blocking assignments, as shown in lines 11 to 14 in Fig. 5. In line 11, positive `mux1_w` means register `buff1` is to be updated. Since value for `mux1` is read for the update, `mux1_r` is positive. This is propagated back to variables that `mux1` depends on, `start` and `op1`, as shown by the arrows. Lines 2 and 3 show the expressions of read flags for conditional statements. Lines 5 and 6 show the expressions for arithmetic/logical statements. Read flags are simply propagated by bit-select/concatenate statements (not shown in Fig. 5).

Assertions: Assertions to check for liveness of register `op1` and `buff1` are shown in lines 15 and 16, respectively, in Fig. 5. These assertions check that registers are not read before written. If an assertion is proven to be true for all executions, the corresponding register is not an ASV. Otherwise if a counterexample is found, the register is a live register and thus a potential ASV. Suppose `START_COMPUTATION` (`start` = 1, `addr` = 0) is issued with positive `start_w` in cycle $t0$. Then `mux1_w` is also positive in $t0$. According to the propagation of read flags, shown by the red arrow in Fig. 5, `op1_r` is positive. As `op1_w` is 0 in cycle $t0$, a counterexample is found for the assertion of `op1`, and thus it is a live register. On the other hand, for register `buff1`, by combining lines 5, 13, 14 in Fig. 5 and line 9 in Fig. 4, we have:

```
assign buff1_r = (buff1_w | buff2_w)
               & (2(sum_s != sum_next_s))
```

Note that `buff1` and `buff2` must be written simultaneously. Therefore `buff1_w` and `buff2_w` must be equal to each other. Then it is easy to see that `buff1_r` can never be positive without `buff1_w` being positive. As a result, the assertion for `buff1` is always true and it is not live and thus not an ASV. (Similarly for `buff2`.) These analyses demonstrate how our algorithm can distinguish ASVs and non-ASVs with the help of auxiliary variables.

IV. Experimental Results And Discussion

We applied our algorithms to four application-specific accelerators and two processors.

Designs	AES	SHA	GB	RBM	8051	Pico RISCV
# Registers	420	42	343	408	149	182
# Register Bits	7563	1612	4959	6049	792	1688
# RTL LoC	1111	1837	6915	12971	13464	2014
# Mem Bits	0	0	4016	3.7MB	128	0

TABLE I: Statistics of RTL designs used in experiments

Table I shows statistics of the RTL designs used in our experiments, and the results for our three algorithms are shown in Table II. The first row of Table I lists the name of the design: "AES" is a cryptographic engine from OpenCores [18], which implements the Advanced Encryption Standard. "SHA" is an RTL implementation of a cryptographic hash function [30]. "GB" is an image processing accelerator for Gaussian Blur algorithms [31] originally designed in the domain specific language Halide [32] and high-level synthesized to Verilog. "RBM" is an accelerator supporting both training and prediction phases of Restricted Boltzmann Machine, a stochastic neural network algorithm [33]. We choose these designs as representative accelerators for cryptography, image processing, and machine learning that are widely used in modern SoCs. Among processors, "8051" [34] is a micro-controller of the MCS-51 family. It supports all 140 instructions in its ISA as well as a UART, two timers, and interrupts. "PicoRISCV" [35] is a size-optimized RISC-V core (without pipeline) that implements the RV32I instruction set.

We use JasperGold [36] for all model checking queries. The experiments are conducted on machines with 32 cores (Intel Xeon Gold 6142) and 384GB of memory. In the AsvAOD algorithm, the budget (time and bound) of each model checking query is dynamically assigned based on the instance. For the AsvTA algorithm, each assertion is given a time-out of two hours. For the AsvLA algorithm, the times-out is 120 hours (since all assertions are checked together). For time-outs in any algorithm, the checked registers are classified as potential (as opposed to confirmed) ASVs – this contributes to over-approximation in the reported set of ASVs.

In Table II, the second column reports the number of total variables in the design including registers and memories/arrays. The third column (# Manually Determined ASV) lists the numbers of ASVs determined through manual inspection of the design, reported as #register/#memory ASVs. (We assume that these have been correctly determined for use as the reference.) For each algorithm, we report the runtime (Time) and memory usage (Memory) for all model checking queries using JasperGold. (The runtime for code instrumentation and assertion generation is negligible compared to the model checking time, and is thus not included here.) In the "ASV" column, we report the number of ASVs identified by the algorithm as #registers/#memory These numbers include the *confirmed ASVs* as well as *potential ASVs*. We also report in parentheses the number of ASVs for which the model checking timed out, due to which we regard them as potential ASVs. Note that this number is 0 for many designs in two algorithms, but high for GB. "N/A" in these columns means that most queries timed-out, so no useful result is available.

A. AsvAOD Results: Checking AOD using Self-Composition

In the AsvAOD algorithm, there are two critical factors that impact the run time: 1) the complexity of model checking queries, and 2) the quality of variable group candidates. Checking hyperproperty 1 using self-composition requires making two copies of the design and therefore increases the problem state space. However, by starting from an arbitrary initial state with inductive invariants, both the counterexample length and proof depth are effectively shortened – a strength of this algorithm. Overall, we found checking hyperproperties manageable in all the case studies. Another performance-impacting part is the CEGAR-style refinement loop. Besides limiting the number of iterations, we also avoid certain types of variables, e.g., internal memory, when generalizing the refinements. Multiple reachability checks are grouped together to utilize the multi-property engines in JasperGold for further speedup.

To evaluate our variable group selection effectiveness, we also compared the total number of iterations in the optimization procedure (not reported in the table). For most designs, the optimization converged quickly. One exception is PicoRISCV, a design without hierarchy. With less precise variable groups, the optimization ends up taking more (smaller) steps.

B. AsvTA Results: Taint Analysis based Algorithm

The AsvTA algorithm can find a set of ASVs with little over-approximation for several designs including AES, SHA, and PicoRISCV. For the other designs the over-approximation was higher. For 8051, it over-approximated ASVs only in the UART module, because the UART works asynchronously and we cannot statically specify when the outputs of the UART are valid using any valid signal. For the remaining designs, the source of over-approximation is due to time-outs. In particular, if an instruction execution takes a large number of cycles, the model checker needs to reason over this large number of cycles and may time-out, e.g., GB runs more than three thousand execution cycles before giving the first valid output.

C. AsvLA Results: Liveness Analysis based Algorithm

From the results in Table II, we see that the AsvLA algorithm can find a set of ASVs with little over-approximation in reasonable time for most designs. Here, checking properties for all registers simultaneously benefits from the multi-property verification engines in JasperGold. For instance, in 8051, the proofs for 55 properties are completed by a single run of the multi-property engine. These properties are highly related to each other. Further, the scope of the

Designs	# Total Variables	# Manually Determined ASV	Algorithm 1: AsvAOD			Algorithm 2: AsvTA			Algorithm 3: AsvLA		
			Time (hr:min:sec)	Memory	# ASV	Time (hr:min:sec)	Memory	# ASV	Time (hr:min:sec)	Memory	# ASV
AES	420	5/0	00:13:47	8.4G	8/0 (8)	137:10:03	100.2G	5/0 (0)	00:16:58	92.3G	10/0 (0)
SHA	42	3/0	00:02:57	1.2G	11/0 (11)	00:01:05	546M	5/0 (0)	00:00:17	2M	7/0 (0)
GB	343	8/16	01:48:39	7.3G	16/16 (13)	136:17:01	256G	132/16 (70)	120:00:00	280G	308/16 (253)
RBM	408	27/22	05:38:24	7.2G	104/22 (104)	N/A	N/A	N/A	19:48:38	351G	27/36 (0)
8051	149	49/1	00:54:14	5.7G	68/1 (68)	43:24:07	74.5G	77/1 (13)	11:40:32	118G	70/1 (0)
PicoRISCV	182	33/0	57:54:20	80.4G	50/0 (50)	41:11:45	22.4G	39/0 (0)	00:39:03	23.9G	43/0 (0)

TABLE II: Model checking results for all algorithms. # Total Variables: total number of register and memory variables, # Manually Determined ASV: number of ASVs based on human inspection: #registers/#memory, Time: model checking runtime, Memory: memory used, ASV: number of found ASVs reported as follows: #register ASVs/#memory ASVs (#ASVs with timed-out proof). "N/A" indicates that most model checking queries timed out, so no valid result is available.

property check is only within one instruction, and we do not check if the result of the register read eventually impacts any outputs. While this can lead to over-approximation, it results in faster verification run times. However, for the GB design, this algorithm also resulted in many timed-out properties, for the same reasons as for AsvTA.

By studying the over-approximated ASVs reported by AsvLA, we found three main causes for over-approximations:

1) Registers in the control logic may be read during instruction processing, even when they are not involved in retaining the state across instructions. For example, while the mem_state register in PicoRISCV is not an ASV, its value is read as a part of the instruction processing control state, and thus it is falsely recognized as an ASV. Such over-approximation is mostly seen in the processor case studies, as they have many control logic FSMs.

2) Even though some register is live at the start of an instruction, and its value is read to update some other register, these updates may not affect any outputs.

3) Some registers used for performance optimization may be falsely determined to be ASVs. For example, some accelerator designs use registers to store the last input value. If the next input value is the same as the last one, the accelerator returns cached results directly for better performance and energy efficiency (referred to as memo functions in programming). These registers are live, but may not be ASVs, since the outputs could be computed without them.

The three kinds of over-approximations occur because the weaker ”liveness” condition is being used to over-approximate the set of ASVs. The AsvLA algorithm is trading-off precision of results for faster performance through verifying multiple properties together.

D. Soundness of Results and Staged Algorithm

As discussed earlier, our algorithms are sound. As a sanity check for our implementations, we confirmed by manual inspection that no ASVs are missed in the results for any of the three algorithms.

Although all three algorithms are sound, Table II shows that there is no single winner across all cases, in terms of the quality of abstraction and execution time. This prompts our consideration of different combinations of these algorithms. The simplest combination is to use them all, and then determine the ASV set as the intersection of the ASV sets reported by the three algorithms (since each algorithm returns an over-approximation of the actual set of ASVs). Running them all can be done in parallel or sequentially, depending on the available computing resources. We refer to these as the *Run-All Parallel* and *Run-All Sequential* algorithms.

The algorithms can also be run in some sequence, with the later algorithms needing to only consider the *potential* ASVs of the previous algorithms. This sequence can be determined based on characteristics of the examples, as discussed below.

- The runtime and memory consumption of the AsvAOD algorithm are low for most cases. The exception is non-hierachical designs (e.g., PicoRISCV) because this algorithm crucially depends on design hierarchy for grouping variables. It also performs especially well for designs with instructions that take a large number ($>$ 1000) of cycles to finish (e.g., GB). (This information is available in the design documentation and does not need RTL analysis.) This is because it starts from an arbitrary state with inductive invariants, in contrast to the other two algorithms that start from the reset state.
- For designs whose instructions finish within 100 cycles as per the design documentation (e.g., AES, SHA, 8051 and PicoRISCV), the AsvTA algorithm usually finds the smallest set of ASVs at the cost of longer runtimes. Thus, for these cases, this algorithm is suitable for refining the results of other algorithms. For designs where it is hard to define when the outputs are valid (e.g., UART in 8051), the AsvTA algorithm results in a larger over-approximation of ASVs and should be avoided.
- For most designs, the AsvLA algorithm can find a set of ASVs with slightly more over-approximation in reasonable time. The exception is designs with instructions that take many cycles to finish, e.g., GB. Unlike the AsvAOD algorithm, this algorithm is not at a disadvantage in non-hierarchical designs, so it can be used as the first one for non-hierarchical designs with moderate instruction completion times.

Based on the above characteristics, we considered the following *Staged Algorithm* that combines the three algorithms:

- For hierarchical designs, first run the AsvAOD algorithm, then refine with the AsvLA algorithm, followed by the AsvTA algorithm.
- For non-hierarchical designs, first run the AsvLA algorithm, followed by the AsvTA algorithm, and skip the AsvAOD algorithm.

We ran experiments with this Staged Algorithm on our case studies, with a total time limit of 24 hours. The number of ASVs found and the total model checking runtime of the Staged Algorithm are shown in the "Auto" column and "Staged" columns in Table III. This number of ASVs reported is the same as the intersection of the results of the Run-All algorithms (parallel or sequential) and this is achieved in much less time than the Run-All algorithms.

E. Quality of Abstractions

The quality of the abstraction provided by a set of ASVs is reflected in the number of state bits in the set of ASVs, relative to the number

Designs	Time (hr:min:sec)			# ASVs	
	Staged	Run-All P	Run-All S	Manual	Auto
AES	01:57:32	137:10:03	137:40:48	5/0	5/0
SHA	00:04:01	00:02:57	00:04:19	3/0	5/0
GB	01:56:02	136:17:01	258:05:40	8/16	16/16
RBM	24:00:00	19:48:38	25:27:02	27/22	27/22
8051	00:57:31	43:24:07	55:58:53	49/1	64/1
PicoRISCV	02:17:27	57:54:20	99:06:05	33/0	37/0

TABLE III: Results of combining algorithms
"Staged" column shows the model checking time for the Staged Algorithm. "Run-All P" shows runtime for running the three algorithms in parallel. "Run-All S" shows runtime for running them sequentially. "Manual" column lists numbers of manually determined ASVs (same as the third column in Table II). "Auto" column lists the number of ASVs found by the Staged Algorithm.

Designs	AES	SHA	GB	RBM	8051	PicoRISCV
Automatically Determined (%)	3.83	3.54	6.25	5.22	52.5	62.4
Manually Determined (%)	3.83	2.98	4.82	5.22	46.8	60.7

TABLE IV: Quality of abstractions
Measured as ratio (%) of number of register bits in ASVs and in RTL. The 'Automatically Determined %" row corresponds to "Auto" column of Table. III. The "Manually Determined %" row corresponds to "# Manually Determined ASVs" column in Table II.

of state bits in the given RTL model. Since this is the first work to automatically determine ASVs from RTL designs, there is no other work to compare this against, because a comparison with the size of abstractions at other levels would not be fair. Instead, we compare it against the manually determined ASVs. (As before, we assume these are determined correctly for the purpose of this comparison.) We calculate the percentage of register bits in the respective ASVs in comparison to RTL, shown in Table IV. (We did not include the bits for memories because memories are usually treated differently than registers by model checking tools.)

The "Automatically Determined %" row in Table IV shows the percentage of bits retained in the ASVs in the "Auto" column of Table. III. The "Manually Determined %" serves as a lower bound of "Automatically Determined %." We can see that although the set of AVSs are over-approximated for some designs, the percentage of reported bits is very close to the lower bound for all designs.

Further, note that for the first four designs in Table IV, the reported ASV state bits are less than 7% of original RTL model. While compared with accelerators, the number of reported ASV state bits for processors (8051, PicoRISCV) is larger, the set of reported ASVs still provide significant abstraction, as the size of the state space for verification grows exponentially with the number of state bits.

V. Related Work

Our work is broadly related to other efforts in hardware abstractions, their automatic generation, and verification.

Architecture-level Abstractions. For processors, stratified program synthesis has been used to generate an x86 ISA specification [16]. It synthesizes semantically equivalent programs with a base set of instructions to model more complex instruction specifications. However, this method may not cover all instructions due to limitations of the synthesis engine, and is only applicable to processors. A more recent work uses template-based program synthesis to generate architecture-level abstractions for accelerators [11]. But this method requires users to identify the ASVs and provide the synthesis templates. Our goal is to automatically identify the ASVs from a given RTL implementation.

FSM/EFSM based Hardware Abstractions. There have been some efforts on automatic generation of hardware abstractions such as Finite-State Machines (FSM) and its variant Extended Finite-State Machine (EFSM) [37]–[40]. For example, A^2T [14] automatically merges states in an EFSM into macro states and eventually converts EFSM of the hardware to Transaction-level Model (TLM) abstractions. Although the number of states is reduced by this method, the merged macro state still contains implementation details, e.g., how the data are transmitted, buffered or processed in the specific hardware, and thus this is not a suitable architecture-level abstraction for specifying software-visible updates. Path Predicate Abstraction [41] groups FSM states into sets based on predicates. Only "important" state sets are retained in the FSM while other states are abstracted as intermediate operations. This approach reduces the state space significantly, but it needs manually-specified predicates, which is an iterative process and needs understanding the hardware design. Our approach does not need any information about the design implementation, other than the encoding and finish conditions for valid instructions at the interface. This information is generally available from user manuals as it is also needed for test generation. Further, Path Predicate Abstraction also retains an "operational view" of the implementation [41], i.e., how hardware does the computation, retaining more detail than is needed in the architecture-level abstractions.

Other Hardware Abstractions. BlueSpec [42], Esterel [43], and SystemC [44] have been used to manually construct high-level models to support hardware design and verification. Similarly Hardware Transactions [4], and Transaction-Level Modeling (TLM) [13] have been used for high-level modeling of components and their communication. Some of these also support synthesis to RTL implementations, but they do not target architecture-level abstractions as such.

ATLAS [45] targets term-level abstraction, which abstracts bit-level data representations into symbolic terms. UCLID [46] employs not only data abstraction, but also operator abstractions with uninterpreted functions. These abstractions help reduce complexity for formal verification and can also be applied to architecture-level abstractions.

Taint Analysis and Verification. Our proposed algorithms are related to prior work on taint analysis [23], information flow security verification [21], [22] and constant-time code verification [28]. However, our purpose is different – to find ASVs. Correspondingly, the details of taint tracking are different in our algorithms.

VI. Conclusions and Future Work

Architecture-level abstractions of hardware are needed for effective system-level co-verification in SoCs. We consider the broad problem of deriving such abstract models from legacy RTL. The key first step is to automaticly determine architectural state variables (ASVs) in RTL designs. We provide three different ways of characterizing over-approximated sets of ASVs – using sensitivity analysis, taint analysis, and liveness analysis. We provide an algorithm for each of these, as well as their experimental evaluation on a set of processors and accelerators. Our experiments show that all these algorithms give sound results with no ASVs missed. There is no clear winner among the three algorithms across all designs, and we explore using their combinations based on the design characteristics in our case studies. Further, while the ASVs found are over-approximated for some designs, the quality of abstractions with the combined algorithms is high with encouraging results. Future work will complete the task of deriving these architecture-level abstractions using these ASVs. That work will focus on extracting the state-update function from the RTL design for these ASVs for each processor/accelerator instruction.

References

[1] R. Saleh, S. Wilton, S. Mirabbasi, A. Hu, M. Greenstreet, G. Lemieux, P. P. Pande, C. Grecu, and A. Ivanov, “System-on-chip: Reuse and Integration,” *Proc. IEEE*, 2006.

[2] A. Horn, M. Tautschnig, C. Val, L. Liang, T. Melham, J. Grundy, and D. Kroening, “Formal co-validation of low-level hardware/software interfaces,” in *2013 Formal Methods in Computer-Aided Design*, pp. 121–128, 2013.

[3] M. D. Nguyen, M. Wedler, D. Stoffel, and W. Kunz, “Formal hardware/-software co-verification by interval property checking with abstraction,” in *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 510–515, 2011.

[4] R. Mukherjee, M. Purandare, R. Polig, and D. Kroening, “Formal techniques for effective co-verification of hardware/software co-designs,” in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2017.

[5] P. Subramanyan, B. Huang, Y. Vizel, A. Gupta, and S. Malik, “Template-based parameterized synthesis of uniform instruction-level abstractions for soc verification,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 8, pp. 1692–1705, 2018.

[6] L. Nelson, J. Bornholt, R. Gu, A. Baumann, E. Torlak, and X. Wang, “Scaling symbolic evaluation for automated verification of systems code with serval,” in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP ’19, (New York, NY, USA), p. 225–242, Association for Computing Machinery, 2019.

[7] L. Semeria and A. Ghosh, “Methodology for hardware/software co-verification in c/c++,” in *Proceedings 2000. Design Automation Conference. (IEEE Cat. No.00CH37106)*, pp. 405–408, 2000.

[8] S. Dasgupta, D. Park, T. Kasampalis, V. S. Adve, and G. Rosu, “A complete formal semantics of x86-64 user-level instruction set architecture,” in *Proc. PLDI*, 2019.

[9] A. Armstrong, T. Bauereiß, B. Campbell, A. D. Reid, K. E. Gray, R. M. Norton, P. Mundkur, M. Wassell, J. French, C. Pulte, S. Flur, I. Stark, N. R. Krishnaswami, and P. Sewell, “Isa semantics for armv8-a, risc-v, and cheri-mips,” *PACMPL*, vol. 3, pp. 71:1–71:31, 2019.

[10] B.-Y. Huang, H. Zhang, P. Subramanyan, Y. Vizel, A. Gupta, and S. Malik, “Instruction-Level Abstraction: A uniform specification for system-on-chip (SoC) verification,” *ACM Trans. Des. Autom. Electron. Syst.*, 2018.

[11] P. Subramanyan, Y. Vizel, S. Ray, and S. Malik, “Template-based synthesis of instruction-level abstractions for SoC verification,” in *Proc. FMCAD*, 2015.

[12] B.-Y. Huang, S. Ray, A. Gupta, J. M. Fung, and S. Malik, “Formal security verification of concurrent firmware in socs using instruction-level abstraction for hardware*,” in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2018.

[13] P. Herber and S. Glesner, “A hw/sw co-verification framework for systemc,” *ACM Trans. Embed. Comput. Syst.*, vol. 12, Mar. 2013.

[14] N. Bombieri, F. Fummi, and G. Pravadelli, “Automatic abstraction of rtl ips into equivalent tlm descriptions,” *IEEE Transactions on Computers*, vol. 60, no. 12, pp. 1730–1743, 2011.

[15] K.-T. Cheng and A. S. Krishnakumar, “Automatic generation of functional vectors using the extended finite state machine model,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 1, p. 57–79, Jan. 1996.

[16] S. Heule, E. Schkufza, R. Sharma, and A. Aiken, “Stratified synthesis: Automatically learning the x86-64 instruction set,” in *Proc. PLDI*, 2016.

[17] U. Degenbaev, *Formal specification of the x86 instruction set architecture*. PhD thesis, 2012.

[18] H. Hsing, “Opencores.org:tiny aes.” https://opencores.org/projects/tiny_aes, 2014. Accessed: 2017-11-17.

[19] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, and G. Reinman, “Architecture support for accelerator-rich CMPs,” in *The 49th Annual Design Automation Conference 2012, DAC*, pp. 843–849, 2012.

[20] E. Singh, F. Lonsing, S. Chattopadhyay, M. Strange, P. Wei, X. Zhang, Y. Zhou, D. Chen, J. Cong, P. Raina, Z. Zhang, C. Barrett, and S. Mitra, “A-QED verification of hardware accelerators,” in *Proc. DAC*, 2020.

[21] G. Barthe, P. R. D’Argenio, and T. Rezk, “Secure information flow by self-composition,” in *Proc. CSFW-17*, 2004.

[22] T. Terauchi and A. Aiken, “Secure information flow as a safety problem,” in *International Static Analysis Symposium*, Springer, 2005.

[23] E. J. Schwartz, T. Avgerinos, and D. Brumley, “All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask),” in *Proc. S&P*, 2010.

[24] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[25] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT Press, 1999.

[26] M. R. Clarkson and F. B. Schneider, “Hyperproperties,” *Journal of Computer Security*, 2010.

[27] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, “Counterexample-guided abstraction refinement,” in *Proc.CAV*, 2020.

[28] K. v. Gleissenthall, R. G. Kıcı, D. Stefan, and R. Jhala, “IODINE: Verifying constant-time execution of hardware,” in *Proc. USENIX Security*, 2019.

[29] C. Wolf, “Yosys open synthesis suite.” http://www.clifford.at/yosys/.

[30] J. Strömbergson, “secworks/sha1.” https://github.com/secworks/sha1, 2020. Accessed: 2017-11-17.

[31] J. Pu, S. Bell, X. Yang, J. Setter, S. Richardson, J. Ragan-Kelley, and M. Horowitz, “Programming heterogeneous systems from an image processing DSL,” *ACM Trans. Archit. Code Optim.*, 2017.

[32] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, “Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines,” *SIGPLAN Not.*, 2013.

[33] C. Pilato, Q. Xu, P. Mantovani, G. Di Guglielmo, and L. P. Carloni, “On the design of scalable and reusable accelerators for big data applications,” in *Proc. CF*, 2016.

[34] S. T. Jakas, “Opencores.org:8051 core.” https://opencores.org/projects/8051, 2016. Accessed: 2017-11-17.

[35] C. Wolf, “cliffordwolf/picorv32.” https://github.com/cliffordwolf/picorv32, 2020. Accessed: 2020-04-01.

[36] Cadence Design Systems, Inc., “JasperGold Formal Verification Platform.” http://www.jasper-da.com/products/jaspergold-apps/. Accessed: 2020-11-19.

[37] J. . Giomi, “Finite state machine extraction from hardware description languages,” in *Proceedings of Eighth International Application Specific Integrated Circuits Conference*, pp. 353–357, 1995.

[38] A. Kamkin, S. Smolov, and I. Melnichenko, “Static analysis of hdl descriptions: Extracting models for verification,” in *East-West Design Test Symposium (EWDTS 2013)*, pp. 1–4, 2013.

[39] Y. Shi, C. W. Ting, B. Gwee, and Y. Ren, “A highly efficient method for extracting fsms from flattened gate-level netlist,” in *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, pp. 2610–2613, 2010.

[40] I. Ghosh, A. Raghunathan, and N. K. Jha, “A design for testability technique for rtl circuits using control/data flow extraction,” in *Proceedings of International Conference on Computer Aided Design*, pp. 329–336, 1996.

[41] J. Urdahl, D. Stoffel, and W. Kunz, “Path predicate abstraction for sound system-level models of rt-level circuit designs,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 33, no. 2, pp. 291–304, 2014.

[42] R. Nikhil, “Bluespec System Verilog: efficient, correct RTL from high level specifications,” in *Proc. MEMOCODE*, 2004.

[43] G. Berry, M. Kishinevsky, and S. Singh, “System level design and verification using a synchronous language,” in *Proc. ICCAD*, 2003.

[44] P. R. Panda, “SystemC - a modeling platform supporting multiple design abstractions,” in *International Symposium on System Synthesis*, 2001.

[45] B. A. Brady, R. E. Bryant, S. A. Seshia, and J. W. O’Leary, “Atlas: Automatic term-level abstraction of rtl designs,” in *Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010)*, pp. 31–40, 2010.

[46] Z. S. Andraus and K. A. Sakallah, “Automatic abstraction and verification of verilog models,” in *Proceedings of the 41st Annual Design Automation Conference*, DAC ’04, (New York, NY, USA), p. 218–223, Association for Computing Machinery, 2004.