CEGAR-Based Formal Hardware Verification: A Case Study

Zaher S. Andraus, Mark H. Liffiton, Karem A. Sakallah

Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109-2122

{zandrawi, liffiton, karem}@umich.edu

Abstract

We describe the application of the Reveal formal functional verification system to six representative hardware test cases. Reveal employs counterexample-guided abstraction refinement, or CEGAR, and is suitable for verifying the complex control logic of designs with wide datapaths. Reveal performs automatic datapath abstraction yielding an approximation of the original design with a much smaller state space. This approximation is subsequently used to verify the correctness of control logic interactions. If the approximation proves to be too coarse, it is automatically refined based on the spurious counterexample it generates. Such refinement can be viewed as a form of on-demand "learning" similar in spirit to conflict-based learning in modern Boolean satisfiability solvers. The abstraction/refinement process is iterated until the design is shown to be correct or an actual design error is reported. The Reveal system allows some user control over the abstraction and refinement steps. Using six representative benchmarks as case studies, we examine the effect on Reveal's performance of the various available options for abstraction and refinement. We also show examples of the types of learned "facts" that Reveal discovers in the refinement step. Based on our initial experience with this system, we believe that automating the verification for a useful class of hardware designs is now quite feasible.

1 Introduction

The paradigm of iterative abstraction and refinement has gained momentum in recent years as a particularly effective approach for the scalable verification of complex hardware and software systems. Dubbed *counterexample-guided abstraction refinement* (CEGAR), its power stems from the elimination (i.e., abstraction) of details that are irrelevant to the property being checked as well as from analyzing any spurious counterexamples to pinpoint and add just those details that are needed to refine the abstraction, i.e., to make it more precise. This is sometimes referred to as *lemma generation*, with the whole process viewed as an iterative, on-demand augmentation of an initial abstraction with lemmas derived from counterexamples that violate the actual design.

Whereas such a verification paradigm is appealing at a conceptual level, its success in practice hinges on effective automation of the abstraction and refinement steps, as well as various checking steps requiring sophisticated reasoning. Examples of recent CEGAR-based verification tools include [2, 3, 4, 5, 9, 10, 12, 13, 16, 17, 21]. Most of the literature on CEGAR, however, focuses on its algorithmic framework and reports summary data showcasing its ability to handle a sampling of large benchmarks. Very little is reported on the inner workings of the abstraction/refinement process and how it is affected by the choices available for both abstraction and refinement. In this paper, we address this issue by reporting the performance of Reveal, an automatic CEGAR-based system for the verification of safety properties of complex hardware designs, on six representative test cases. For each test case:

- we compare a number of methods to model and check the desired properties on the abstract design, including the use of a *Satisfiability Modulo Theories* (SMT) solver [11];
- we study trade-offs between various refinement options;
- we highlight the types of lemmas generated in the refinement stage and analyze the idiosyncrasies leading to them;
- and, finally, we compare the performance of Reveal against a number of existing automatic tools that perform formal verification for hardware, such as VCEGAR [16], BAT [19], UCLID [7], and VIS [22];

To our knowledge, this is the first work that performs such a comprehensive comparison for the formal verification of complex hardware designs including memory systems and pipelined microprocessors whose RTL descriptions have tens of thousands of HDL source lines, thousands of signals, and hundreds of thousands of state bits. Additionally, this paper provides experimental evidence that demonstrates the importance of datapath abstraction for the scalability of formal verification algorithms.

The rest of the paper is organized in five sections: Section 2 reviews Reveal's CEGAR framework, Sections 3 through 8 describe six hardware verification case studies, and Section 9 summarizes the paper's conclusions.

2 The Reveal Verification System

Figure 1 illustrates the architecture of the CEGAR-based Reveal automated verification system. Reveal performs checks of safety properties on hardware designs described in the Verilog hardware description language (HDL). A typical usage scenario involves providing two Verilog descriptions of the same hardware design, such as a high-level specification and a detailed implementation, and checking them for functional equivalence. Reveal adopts the CEGAR-based approach of Andraus *et al.* [2] which consists of the following steps:

- **Abstraction** to obtain a compact representation of the design for which formal property checking is more likely to terminate (i.e., to scale both in time and space) than if applied directly on the original design.
- **Property Checking** by formal reasoning to determine if the abstracted design satisfies the specified property.
- **Refinement** to determine if the abstraction was sufficient to establish whether the property holds or fails on the actual concrete design and, if otherwise, to provide one or

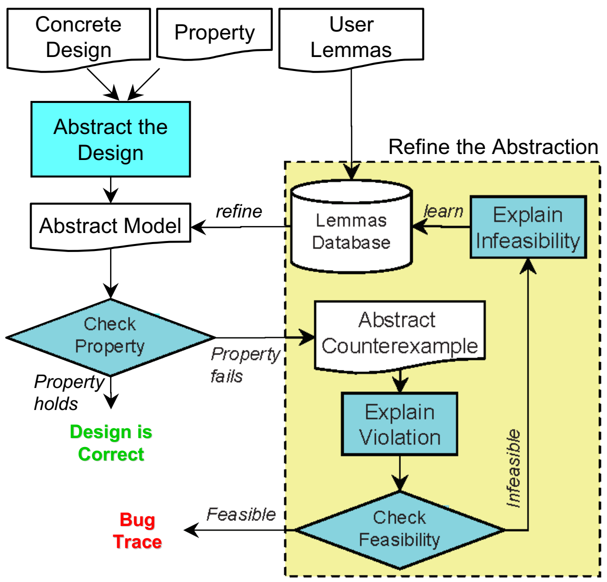

Figure 1: The Reveal Flow

more succinct explanations that are used to refine the abstraction for the next round of checking.

2.1 Abstraction

Reveal performs datapath abstraction [1, 7, 15], i.e., it replaces the design's datapath elements (registers and function units such as adders, shifters, etc.) with *terms*, *uninterpreted functions* (UFs), and *uninterpreted predicates* (UPs). This yields an abstract model of the design that maintains the consistency of the removed datapath elements without representing their detailed functionality and leads to a significant reduction in the size of the design's state space. The interactions among the control signals, however, are preserved, allowing meaningful verification of safety properties on the design's control logic.

Signals in the Verilog description are classified as datapath or control based on their bit width [1]. This heuristic is premised on the assumption that datapath elements typically operate on multi-bit words and can, thus, be easily identified in a Verilog model. However, misclassification of a control signal as a datapath signal or vice versa does not compromise the correctness of the approach. Specifically, a control signal that is abstracted as part of the datapath will likely yield a spurious counterexample and may cause an increase in the number of refinement iterations. The less probable scenario of misclassifying a datapath signal as control causes the abstract model to be unnecessarily detailed and possibly makes the property checking step intractable. That said, adopting this heuristic ought to be backed up by empirical experiments, as we will show in this paper.

Formally, abstraction, including datapath abstraction, can be viewed as a relaxation of the system of constraints that characterize the concrete design. Specifically, if we let $C_i(X)$ denote the constraint imposed on the set of design signals X by component i, then the formula $\text{conc}(X) \doteq \bigwedge_{1 \le i \le n} C_i(X)$ characterizes the behavior of a design consisting of n components. For instance, a Verilog assignment statement such as $R[3] <= R[1] + R[2]$, where the variables correspond to 32-bit registers in a register file, is modeled by the constraint

$$C(R[1], R[2], R[3]) \doteq (R[3] \equiv R[1] + R[2]) \qquad (1)$$

This constraint can be viewed as the *consistency* function for the 32-bit addition operator: it evaluates to true for consistent assignments to its arguments, and false otherwise. Replacing the register variables with terms and the addition operator with the UF *add*, yields the constraint

$$A(T1, T2, T3) \doteq (T3 \equiv add(T1, T2)) \qquad (2)$$

where $T1$, $T2$, and $T3$ correspond, respectively, to $R[1]$, $R[2]$, and $R[3]$. Informally, the register variables and their corresponding terms encode unsigned integers except that the integer ranges of the register variables are bounded because of their finite bit width. Thus, each term can be regarded as an abstraction of the corresponding register variable, and the consistency condition in (2) is similarly an abstraction of the concrete consistency condition in (1).

Assuming that each concrete consistency constraint $C_i(X)$ is relaxed to a corresponding abstract consistency constraint $A_i(T)$, where X and T denote the concrete design signals and their corresponding term abstractions, we can now model the abstraction of the design by the formula $\text{abst}(T) \doteq \bigwedge_{1 \le i \le n} A_i(T)$ and note that

$$\text{conc}(X) \rightarrow \text{abst}(T) \qquad (3)$$

In addition to abstracting datapaths, tractable verification may require the abstraction of memory arrays. Applying only datapath abstraction to an n-word by m-bit memory yields an n-term abstraction. For memories of typical sizes in current designs, n is on the order of thousands to millions of words. *Memory abstraction* allows us to model an n-word memory by a formula whose size is proportional to the number of write operations, K, rather than to n. Note that memory abstraction is distinct from datapath abstraction. A useful mnemonic device is to think of datapath and memory abstraction as being, respectively, "horizontal" and "vertical;" they can be applied separately as well as jointly.

Reveal implements memory abstraction using lambda expressions as described in [7]. In particular, the expression

$$M'(X) = \lambda_X.\text{ITE}(X = A, D, M(X)) \qquad (4)$$

describes the next state of a memory array M after a write operation with address A and data D. Memory abstraction can also be realized using a theory of arrays [20]; we compare these two approaches experimentally in Sections 4 and 5.

2.2 Property Checking

Checking that the specified safety property holds on the abstract design amounts to proving the validity of

$$\text{abst}(T) \rightarrow \text{prop}(T) \qquad (5)$$

where $\text{prop}(T)$ denotes the desired property. In what follows, the formula in (5) will be referred to as the *abstract verification condition*, to distinguish it from the *concrete verification condition*, i.e.

$$\text{conc}(X) \rightarrow \text{prop}(X). \qquad (6)$$

In general, (5) is an instance of quantifier-free first order logic for the theory of equality with uninterpreted functions (EUF) [8]. Adding the limited form of integer arithmetic referred to as *counter arithmetic* makes such formulas instances of the so-called CLU logic [7] which is particularly useful in datapath abstraction.

The Reveal system is essentially a bounded model checking [6] verifier for safety properties with known sequential bounds. While this may seem to limit its utility, empirical observation suggests that it has application in many situations where such bounds are known a priori or can be easily derived from the particular structure of the designs being verified. Furthermore, such bounds tend to be rather small, often less than ten cycles. Examples include verification of pipelined microprocessors, packet routers, network processors, and dataflow architectures common in filters, etc.

Given a cycle bound k, (5) is generated by unrolling the abstract design's transition relation k times. The formula is then submitted to a checker to determine its validity. In the Reveal system, the validity of the formula is checked using the YICES Satisfiability Modulo Theories (SMT) solver [11]. Unlike earlier approaches [1, 2, 7], which convert such formulas to equi-satisfiable propositional formulas using suitable encodings, SMT solvers operate on these formulas directly by integrating specialized "theory" solvers within a backtrack propositional solver. SMT solvers take advantage of the high-level semantics of non-propositional constraints (e.g., integer arithmetic, equality of uninterpreted functions) while at the same time benefiting from the powerful reasoning capabilities of modern propositional SAT solvers.

If the checker determines that the formula is valid, Reveal exits indicating that the property holds on the design. This is guaranteed by the soundness of the abstraction. If, alternatively, the formula is found to be invalid, the checker produces an abstract counterexample T^* that indicates how the abstract model fails to satisfy the property.

2.3 Refinement

The refinement step is necessary to make the verification flow complete because the violation reported by the checker may be due to a poor abstraction rather than a real bug in the design. Refinement consists of three stages [2]:

- **Explaining the violation** by generalizing the abstract counterexample. This is accomplished by replacing the specific counterexample returned by the checker with a compact set of constraints $\mathrm{viol}(T)$ that include the counterexample as well as others that share its "profile" as far as the abstraction is concerned.
- **Checking the feasibility of the violation on the concrete model** by checking, with YICES, the satisfiability of

  $$\mathrm{viol}(X) \wedge \mathrm{conc}(X) \tag{7}$$

  If satisfiable, Reveal exits indicating that the property is violated and returns a bug trace to help locate the error.
- **Explaining why the violation is spurious.** If $\mathrm{viol} \wedge \mathrm{conc}$ is unsatisfiable, then the violation reported by the validity checker is an artifact of the abstraction and not a real bug. An explanation of its infeasibility on the concrete model is obtained by deriving the minimal unsatisfiable subsets (MUSes) of $\mathrm{viol} \wedge \mathrm{conc}$, using MUS extraction tools [18]. Negations of these MUSes are viewed as "lemmas" which are added to a growing database of such "facts" and used to refine the abstraction for the next iteration.

Note that the steps of explaining the violation and its infeasibility can be replaced, without compromising completeness, by simply negating the counterexample. However, this refinement is generally too weak, causing the number of refinement iterations to be too high for the approach to be practical.

2.4 Implementation and Experimental Setup

The Reveal system is written in C++ and employs four major modules: a formula generator, a solver, an MUS extractor, and a refinement module.

The Formula Generator creates equation (5) by flattening the Verilog model, calculating the transition relation for its state variables, and unrolling it to a user-defined depth.

The Solver Module is responsible for determining the validity of the FOL formula (5) and the satisfiability of formulas (6) and (7). It interfaces with the YICES SMT solver via a C++ API [24]. This module can determine, for example, whether (5) is valid in the EUF or CLU logics, or whether (7) is satisfiable in the bit vector (BV) logic.

The MUS Extractor is responsible for identifying MUSes from an unsatisfiable formula. Unlike [2], we use a modified version of the CAMUS MUS extraction algorithm [18] that works directly with the YICES solver. This eliminates the need to generate a propositional encoding of the abstract formula and leads to significant speedup in MUS generation. Given an unsatisfiable formula, CAMUS can be run in two modes: single- or multiple-MUS extraction.

The Refinement Module manages the lemma database and updates the abstraction for the next iteration. This module can accept user-supplied lemmas and can maintain a persistent lemma database that can be accessed across invocations.

In the following sections we will classify the various runs of Reveal by a one-, two-, or three-letter code that indicates the abstraction and refinement options used. Abstraction options will be labeled B (bit-level, i.e., no abstraction), C (CLU abstraction), and E (EUF abstraction). Refinement options will be labeled V (negating the violation) and L (refinement with lemmas). For lemma refinement, S will denote refinement with one lemma per iteration, while M will denote refinement with multiple lemmas. For example, the label CLM means CLU abstraction and refinement with multiple lemmas, whereas EV means EUF abstraction and refinement with the negation of the violation.

Our empirical case study compares the performance of Reveal against the following four verification systems:

- UCLID [7, 31] which allows modeling of the datapath with abstract terms, and memories with Lambda expressions. Since UCLID does not accept Verilog, we use VAPOR [1] to produce UCLID models.
- BAT [19, 30] which models memories with *set* and *get* functions for reads and writes, respectively, but models the datapath with finite-length bit-vectors. We are unaware of a convertor from Verilog to BAT's custom language.
- VCEGAR [16, 32] which performs word-level predicate abstraction on the Verilog input, but does not abstract memory arrays.
- VIS [22, 33] which, by default, uses bit-level reachability analysis to verify invariants. It can also be used in two special modes: one that performs bounded model checking of safety properties, and another that performs invariant checking with a CEGAR algorithm based on *hiding registers* [23]. We will denote the default mode by VIS, the BMC mode by VIS(BMC), and the last mode by VIS(AR).

In what follows, we present our case studies on the benchmarks summarized in Table 1. All experiments were conducted on a 2.2 GHz AMD Opteron processor with 8GB of

Table 1: Benchmark statistics

Name	Verilog Lines	Verilog Signals	State Bits
Sorter	79	30	35 to 1027
ICRAM	153	13	1.3×10^5
OMU	400 to 10K	40 to 260	1×10^6
DLX	2.4×10^3	399	1×10^{11}
Risc16F84	1.2×10^3	169	1×10^5
X86	1.3×10^4	1×10^3	5.8×10^3

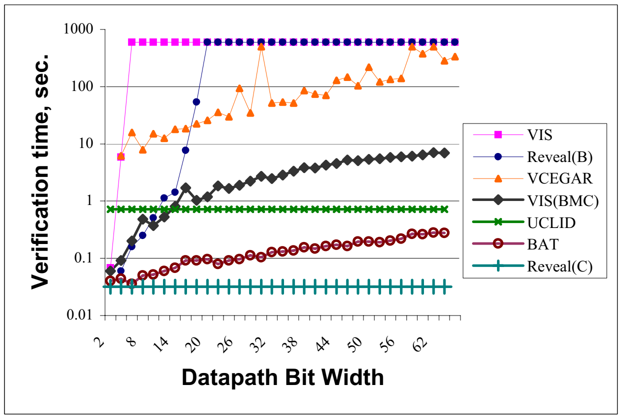

Figure 2: Runtime graphs for Sorter

RAM running Linux. VCEGAR, BAT, and UCLID use the zChaff SAT solver [34] and the SMV model checker [35].

3 Sorter Case Study

The Sorter design implements two versions of an algorithm that sorts four bit-vectors. It makes use of a Sort2 sub-unit that sorts two bit-vectors. In the first version, five Sort2 sub-units are instantiated and connected serially. The inputs are introduced to the first two sub-units, and the calculation propagates serially towards the outputs. The computation advances through 3 layers of registers, thus requiring three cycles to complete. The second version is based on just two Sort2 sub-units and a controller that uses them to carry out the sorting computation in three cycles as well.

The property we verified is the equality between corresponding outputs in the two versions. All the bit-vectors in the two units, including the inputs and the outputs, are of bit-width W, which we vary from 2 to 64 to see the effect of the datapath width on the scalability of each tool. Figure 2 shows the runtime of each of the verification tools as a function of W, and Table 2 shows the number of bits in the concrete verification condition (i.e., formula (6)) and statistics about the number of the nodes in the abstract verification condition (i.e., formula (5)). The last column, labeled by R, shows the ratio between the number of bits and the number of nodes. The results demonstrate the following trends:

- The effect of datapath abstraction is evident from the performance of Reveal(C) and UCLID, which are oblivious to W. In both cases the abstract model is unaltered when changing the datapath bit width; thus the time needed to verify the abstract model is constant. Furthermore, the only interaction between the datapath and the control involves bit vector inequalities, allowing the CLU logic to prove the property without any refinement.
- BAT's performance degrades when increasing W, since the datapath is unabstracted. Nonetheless, BAT's reduction to CNF appears to play an important role in keeping the runtime low.
- VCEGAR takes 6.1 seconds to prove the property for W=2 as it incrementally discovers between 33 and 40 predicates within 58 to 130 iterations. Additionally, the runtime grows exponentially with the width of the datapath. We suspect that the reason behind this is the expense of simulating the abstract counterexample on the concrete design in each refinement iteration, as well as the repeated generation of the abstract model each time a new predicate is added.
- The runtimes of Reveal(B), VIS, and VIS(BMC) degrade rapidly as the bit width is increased. The runtimes of VIS(AR) are similar to VIS and were removed from the graph to avoid clutter.

Table 2: Verification conditions for Sorter, ICRAM, & OMU

Test	conc → prop	abst → prop					R
	Bits	**Terms**	**Bools**	**UFs**	**UPs**	**Total**	
Sorter, W=8	127	14	12	0	0	25	5.08
Sorter, W=16	249	14	12	0	0	25	9.96
Sorter, W=32	473	14	12	0	0	25	18.9
Sorter, W=64	921	14	12	0	0	25	36.8
ICRAM	287	31	48	9	2	90	3.12
OMU, K=16	1346	67	275	2	0	344	3.91
OMU, K=32	3154	131	1059	2	0	1192	2.65
OMU, K=64	8306	259	4163	2	0	4423	1.88
OMU, K=128	25K	515	17K	2	0	17K	1.47

4 Instruction Cache RAM Case Study

The Instruction Cache RAM (ICRAM) test case is obtained from the publicly available Verilog description of the Sun PicoJava II Microprocessor [26]. This unit includes a memory array of 16K 8-bit words, 32-bit input and output data ports, and single-bit control signals to trigger certain operations in the cache such as reading, writing, BIST testing, and switching to "power down" mode. The ICRAM interacts with the Instruction Cache Unit [25] which manages the instructions tags and buffers for the entire microprocessor.

The address space of the ICRAM is divided into two "banks," distinguished by a single bit in the address register. A write operation takes an address signal adr[13:3], a data signal di[31:0], and control signals selecting the destination bank $b \in \{0,1\}$. The memory update for write(adr, di, b) is:

```
mem[{adr,b,00}]<=di[31:24]
mem[{adr,b,01}]<=di[23:16]
mem[{adr,b,10}]<=di[15:8]
mem[{adr,b,11}]<=di[7:0]
```

The ICRAM has been formally verified by VCEGAR [16] and BAT [19]. The property verified is that given an arbitrary initial memory array, performing a write(adr,di,0), then performing a read from address {adr,001}, will yield a value that is equal to di[23:16].

We verified this example with Reveal(C), Reveal(B), BAT, and UCLID. The runtimes are 30ms, 38ms, 50ms, and 92ms, respectively. This result is counterintuitive given that the original design has 2^{17} state bits. The efficiency of these methods stems primarily from the reduction obtained by memory abstraction; as shown in Table 2, both the concrete and the abstract verification conditions are very small despite the large state space. Moreover, due to the simple

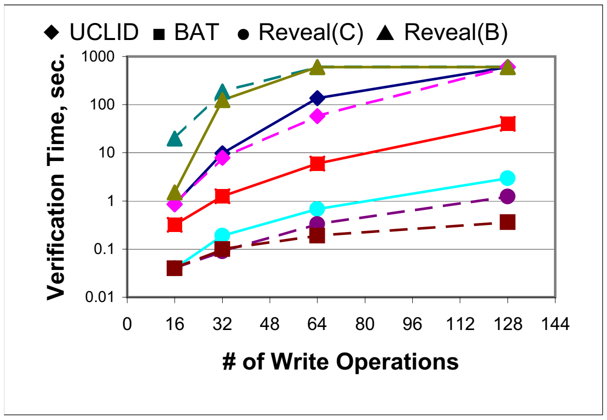

Figure 3: Runtime graphs for OMU. Dashed and solid lines correspond to the first and second properties, respectively. VIS and VCEGAR were omitted to avoid clutter.

interaction between the control and datapath, the abstraction in UCLID and Reveal(C) is sound and complete. Therefore, refinement is not triggered.

Left unabstracted, the memory array causes VCEGAR and VIS to encounter "vertical" state explosion. VCEGAR's runtime was shown in [16] and [19] to grow exponentially with the memory size. Likewise, VIS times out for this example. In particular, the verification in VIS begins with converting any n-word by m-bit memory into $n \cdot m$ single-bit registers regardless of the property being verified. "Flattening" the memory in this way also leads to loss of the structural correlation between the memory registers, which can otherwise be used by the model checker during verification.

5 Out-of-Order Memory Updates Case Study

The Out-of-Order Memory Updates example (OMU) has been previously introduced in [19] to demonstrate the effectiveness of memory abstraction for RTL verification. The design instantiates an array of 65K 16-bit words, which can be read from or written to via designated signals.

The design is verified by simulating two sequences of write operations on the memory array. The initial memory M is modified by a sequence of K writes to locations A, A+1,A+2,...,A+K-1, with the data words $D_1,D_2,...,D_K$, respectively, resulting in memory *M1*. Independently, a second sequence of writes is performed on M in locations A+K-1, A+K-2,...,A, with the data words, $D_K,D_{K-1},...,D_1$, respectively, resulting in memory *M2*. Since the addresses for the write operations are mutually distinct, the ordering of the writes does not affect the final state of the memory. In particular, the content of location A in both *M1* and *M2* is equal. A second, more generic, property is verified by simulating a similar sequence of writes to distinct locations $A_1,A_2,...,A_K$. In other words, we allow the addresses to be arbitrary, albeit mutually disequal.

We compared Reveal(C), Reveal(B), BAT, and UCLID on these two properties, while varying K over {16,32,64,128}. The runtimes are plotted in Figure 3 on a logarithmic scale. Similarly to the ICRAM case, the effect of modeling the memory is evident in this example. In particular,

- Reveal(C) scales well on both properties, taking less than 3 seconds for all the values of K. This is attributed to the memory abstraction via Lambda expressions [7]. Refinement was not triggered since the datapath/control interactions are exclusive to (dis-) equalities.
- BAT appears to be sensitive to the pattern of memory writes; proving the property for arbitrary addresses is two orders of magnitude slower than for consecutive addresses.
- UCLID is two orders of magnitude slower than BAT and Reveal(C) on both properties. Despite its memory and datapath abstractions, its reduction to CNF [7] is significantly slower in proving the property on the abstract model.
- Reveal(B) clearly demonstrates the state explosion problem, as the runtime rapidly worsens when increasing K.
- As with the ICRAM case, VCEGAR's runtime was shown in [19] to grow exponentially in the number of writes to memory. VIS times out on this example for any number of writes. The lack of memory abstraction hinders both.

6 DLX Case Study

DLX is a 32-bit RISC microprocessor [14]. Its salient features include a 32-bit address space with separate instruction and data memories, a 32-word register file with two read ports and one write port, and 38 op-codes for arithmetic, logical, and control operations.

Our case study involved comparing two versions of DLX, both written in Verilog 95 [28]. The first version, which we will refer to as *DLXSpec*, is a single-cycle implementation of the instruction set architecture (ISA) and serves as the architectural specification of the microprocessor. The second version, labeled *DLXImpl*, is a standard 5-stage pipelined design consisting of instruction fetch, instruction decode, instruction execute, memory access, and write-back stages.

Starting *DLXSpec* and *DLXImpl* from their reset states, the property we checked for was equivalence of corresponding state elements (register and memory locations) after a bounded number of execution cycles. Specifically, let E_i^S and E_j^I denote the values of two corresponding state elements from the specification and implementation after i and j cycles from reset, respectively. These two elements would, then, be considered equivalent if:

$$(E_1^S = E_1^I) \vee (E_1^S = E_2^I) \vee \ldots \vee (E_1^S = E_5^I)$$

To compare the various abstraction and refinement options in Reveal and to demonstrate its ability to (dis-)prove properties, we verified a number of (buggy and bug-free) variations of the design. We focus on $E \doteq PC$ here, but similar verification can be used for other state elements. The buggy versions were obtained by injecting errors in the RTL of *DLXImpl*. These variations are described in Table 3 in rows D1, D2, and D3. Following the description of each version, the columns labeled A.C. (C.C.) show the number of nodes (bits) in the abstract (concrete) verification condition. The column labeled R shows the ratio of bits to nodes. The remaining columns contain statistics for each mode of Reveal. Columns labeled T, I, and L, describe, respectively, runtime (seconds), number of iterations, and total number of refinement lemmas (when applicable). The columns labeled A show the ratio of the runtime of verifying the abstract model to the total runtime as a percentage. Finally, the smallest runtime is emphasized in each row; there can be multiple in each row when the difference is insignificant.

The performance of the various options in Reveal demonstrate the role of automatic refinement. Since the control and the datapath in this design are intermixed, refinement is needed to "recover" facts that were lost in the course of the

Table 3: Verification results for the X86, DLX, and Risc16F84 variations

#	Test Case/Version	A.C.	C.C.	R	Reveal(CV)		Reveal(ELS)		Reveal(CLS)		Reveal(ELM)				Reveal(CLM)				Reveal(B)
					T	**I**	**T**	**I**	**T**	**I**	**T**	**A**	**I**	**L**	**T**	**A**	**I**	**L**	**T**
D1	Bug-free DLXSpec and DLXImpl	3945	22K	5.58	>600	>1507	1.92	9	1.8	8	**0.6**	39	4	8	1.0	27	6	12	>600
D2	Pipeline "Stall" control stuck at 1	522	3552	6.8	**0.11**	1	0.15	1	**0.12**	1	**0.11**	3	1	0	**0.1**	7	1	0	0.21
D3	Address of jump instruction calculated incorrectly	3915	22K	5.62	3.16	45	2.22	11	1.16	5	**1.13**	23	3	5	**1.1**	25	4	8	6.7
R1	Bug-free OCSpec and OCImpl	2904	7286	2.54	>600	>1767	>600	>1204	>600	>1085	257	.7	93	185	**148**	.8	68	170	209
R2	Floating "carry-in" signal for addition	2928	7376	2.52	**0.79**	8	56	20	>600	>1881	72	1	44	13	40	1.1	33	39	15.2
R3	"aluout_zero_node" is stuck-at-1 in OCImpl	2849	7224	2.54	115	654	50	123	121	311	**2.6**	.6	5	15	27.3	0.2	40	73	11.6
X1	Bug-free X86 design and property	70K	153K	2.19	>600	>388	>600	>1158	>600	>945	**36.5**	31	40	104	60.4	59	19	96	>600
X2	The property swaps the signals $en_{Integer}$ and $en_{FloatingPoint}$	67K	153K	2.28	>600	>461	>600	>1062	>600	>1046	**30.5**	32	78	161	103	63	24	86	>600
X3	The FSM transits from state 000 to state 011 instead of 010	2646	2764	1.04	**1.98**	2	**1.95**	2	**1.96**	2	**2.0**	6	2	6	**2.1**	6	1	0	2.72
X4	The opcode for CMP activates the Floating Point unit	67K	153K	2.28	>600	>308	>600	>847	>600	>1252	**23**	48	12	41	58.7	74	7	43	>600

abstraction, yet are relevant to (dis-)proving the property. To shed some light on the types of lemmas discovered during this process, we traced the source of these lemmas back to the original Verilog code. Most of these lemmas were related to the pipeline registers and control logic in *DLXImpl*. For instance, the lemma $(\text{IR3} = 32\text{'d0}) \rightarrow (\text{IR3}[31:26] \neq 6\text{'d4})$, which states that it's not possible to extract a non-zero field from a zero bit vector, was traced to the following code segment involving IR3:

```
define BEQ 4
define op 31:26
initial IR3 = 32'd0;
case IR3[`op] `BEQ: ...
```

In this case, the initial abstraction lost the fact that IR3[31:26] can not be equal to 4, and it found a spurious counterexample that executed the BEQ instruction.

Upon closer examination, we found that *DLXImpl* consists mainly of a datapath that is responsible for computing values for the PC and memory to be committed, and control logic that orchestrates the pipeline. Furthermore, the datapath in *DLXImpl* is very similar, and in most cases identical, to the datapath in *DLXSpec*. As a result, refinement only affects those portions of the design involving interactions between the datapath and control logic in *DLXImpl*.

Table 3 also shows that the use of lemmas for refinement (modes ELS, CLS, ELM, and CLM) is far superior to using the violation (mode CV). Also, using multiple lemmas in each refinement (modes CLM and ELM) outperforms refinement with a single lemma at a time (modes ELS and CLS).

Surprisingly, Reveal(B) is able to terminate on the buggy versions of the design. This is attributed to the ability of the BV solver in YICES to efficiently find a satisfying assignment to equation (6). The rest of the case studies in this paper confirm that proving the unsatisfiability of this equation is intractable with Reveal(B), while proving its satisfiability may be tractable in some cases, though not all.

In order to compare the performance of YICES and UCLID in solving the abstract formula, we generate the expression: $(\text{conc} \rightarrow \text{prop}) \vee \bigvee_i \text{lemma}_i$, which represents the final "refined" verification condition created in Reveal. This expression is dumped as a Verilog word-level combinational circuit, and VAPOR is then used to generate its corresponding UCLID model. UCLID spends two orders of magnitude more time than the time spent by Reveal in solving the abstract formula. We observed a similar trend in the rest of the test cases.

Finally, we ran VIS and VCEGAR on this design. VIS was unable to create a netlist due to what we believe is an internal error in the tool. Regardless, we do not think that VIS could verify this design due to its large memory arrays. VCEGAR processed the input but timed out at 600 seconds.

7 Risc16F84 Case Study

This design is an implementation of the Risc16F84 microcontroller [29]. It has a 2^{13}x14-bit instruction memory, a 2^9x8-bit data memory, 34 op-codes, and a 4-stage pipeline.

Similarly to the DLX case, we denote the implementation and specification by *OCImpl* and *OCSpec* respectively. *OCImpl* processes one instruction every four cycles, while *OCSpec* needs one cycle to process each instruction. The equivalence criterion in this case is:

$$\bigwedge_j (I_0^j = S_0^j) \rightarrow \bigwedge_j (I_4^j = S_1^j)$$

where I_i^j and S_i^j denote the state of the j^{th} state element in *OCImpl* and *OCSpec*, respectively, after i cycles of execution. In essence, this is an inductive criterion: given equal state elements in the current cycle, it requires equal state elements after processing a single instruction. Note that unlike the DLX case, where the verification starts from the rest state, here we start from an arbitrary "matching" state.

Reveal was able to discover a genuine bug in this design. The following Verilog code in *OCImpl* uses a *floating* signal `c_in` as the carry-in bit to a 8-bit addition operation.

```
// risc16f84_lite.v
reg c_in; // line 223
{add_node,temp} <= {1'b0,aluinp1_reg,1'b1}+
                   {1'b0,aluinp2_reg,c_in}; // line 519
```

OCSpec, on the other hand, performs addition without any carry-in bit. Reveal thus produces a counterexample showing the deviation, with `c_in` assigned to 1. The unit designer acknowledged this problem, and asserted that the simulation carried out for this design assumed `c_in=0`.

Table 3 contains results for three versions of this design. R1 is a bug-free version, R2 has the aforementioned bug, and R3 was obtained by injecting a stuck-at-1 bug in the signal `aluout_zero_node`. In these results we observe the following:

- Refinement with lemmas is superior to refinement with the violation. Furthermore, the use of multiple lemmas for refinement is crucial for verifying version R1.

- Unlike the DLX case (and the X86 to follow), the verification condition here is relatively small despite the huge memory embedded in the Risc16F84 design. This is attributed to memory abstraction discussed in previous sections.
- Unlike the DLX case, where the verification of the bug-free version (D1) with Reveal(B) times-out, the verification of the bug-free version (R1) with Reveal(B) terminates after 209 seconds. It also terminates rapidly on the 2 buggy versions. This makes its performance comparable with Reveal(C) and Reveal(E). As we saw in previous sections, the runtime of Reveal(B) grows exponentially with the number of bits in the concrete verification condition. On the other hand, the performance of Reveal(C) and Reveal(E) depends on the number of nodes in the verification condition as well as the control/datapath intermix. If we denote the number of bits in the concrete verification condition as b, we can extrapolate that the performance of Reveal(B) is comparable with Reveal(C) and Reveal(E) when
  - b is "small enough", i.e. $b<b_{\max}$ for some $b_{\max}$, and
  - $R=\dfrac{b}{N}$ is "small enough", i.e. $R<R_{\max}$ for some $R_{\max}$.

  If we choose $b_{\max}=4000$ and $R_{\max}=3$, then the above criterion predicts the performance of Reveal(B) for 15 out of the 18 variations of the test cases in this paper. For the RISC16F84, R is approximately 2.5, i.e. the bit vectors in the verification condition are on average 2 to 3 bits wide.
- The R2 case shows an interesting outlier, in which Reveal(CV) is significantly faster than any version that refines with lemmas. This is due to the heuristic nature of the satisfiability search for finding a bug. Any search, regardless of the refinement used, could "get lucky" and reach a bug early in this way, though only rarely.

An analysis of the lemmas discovered in all variations of this test case reveals that most of the spurious counterexamples are due to the *variable opcode width* feature, wherein the opcode field can be k-bits wide for any $k \in K = \{2, 3, 4, 5, 6, 7, 14\}$. For instance, the opcode of the *goto* instruction is IR[13:11]=3'b101, while the opcode for *addlw* is IR[13:9]=5'b11111. The encoding guarantees that only one opcode is active at any given time. This information is lost when abstracting the bit vector extraction operation. This results in the occurrence of lemmas of the form $(IR[13{:}k_1] = v_1) \rightarrow (IR[13{:}k_2] \neq v_2)$ for values $v_1 \neq v_2$ and distinct indices $k_1, k_2 \in K$.

On this example, UCLID timed-out after 600 seconds for R1, and is two orders of magnitude slower than YICES on R2 and R3. VCEGAR runs out of memory after 370 seconds, and VIS was not able to process this design since it does not support *blocking assignments*, which are used throughout the Verilog description. We believe that VIS would otherwise encounter an additional obstacle with the large memories.

8 X86 Case Study

The X86 design is an open source RTL Verilog model developed at IIT, Madras that implements Intel's IA-32 ISA [27]. The design contains four high-level modules. The *Decoder* module, which is the main focus of our verification effort, is responsible for fetching an instruction prefix from the memory, finding the total length of the instruction, fetching and decoding the rest of the instruction, and providing the result to the *Control* module. The top module of the Decoder instantiates the fetching unit, the instruction length find unit, and six decoding units, which correspond to six instruction types that exist in the x86 architecture and its extensions, namely Integer, Floating Point, MMX, SSE, SSE2, and SSE3. Each decoding unit has an enable signal that orchestrates its operation with the Decoder top module.

Upon reset, the Decoder fetches the PC and the corresponding instruction from memory. We verified the property that the Decoder activates the corresponding decode unit when the instruction is confined to a set of 6 Integer and Floating Point op-codes as follows:

$$(\text{opcode} \in \{\text{CMP}, \text{JMP}, \text{MOV}, \text{FADD}, \text{FCMOV}, \text{FINIT}\}) \rightarrow$$
$$((\text{opcode} \in \{\text{CMP}, \text{JMP}, \text{MOV}\}) \leftrightarrow \text{en}_{Integer}) \wedge$$
$$((\text{opcode} \in \{\text{FADD}, \text{FCMOV}, \text{FINIT}\}) \leftrightarrow \text{en}_{FloatingPoint})$$

When the verification was invoked in Reveal, the tool was able to discover a coding problem in the design. In particular, the RTL description includes the code

```
// sse3Decoder.v
op2 = 32'd0; // line 55
if (...) // line 185
  op2[16:0] = instrSeq[31:16]; // line 188
```

which uses a blocking assignment to initialize the signal op2, and then extracts a 16-bit displacement value from the instruction stream and assigns it to a 17-bit register. Most synthesis tools will zero-extend the RHS expression to make the sizes consistent, in which case the resulting model is still correct. Nonetheless, such an error may indicate additional problems in other units of the design. We have notified the unit designers about this problem, and we modified the Verilog to eliminate the problem for the later experiments.

Similarly to the previous two test cases, we compared the performance of Reveal on two buggy versions and one bug-free version and included the results in Table 3. These results reassert the importance of refinement with multiple lemmas.

A notable phenomenon in this case is that Reveal(C) converges significantly faster than Reveal(E) in terms of refinement iterations. This is attributed to the heavy use of counters in the FSM of the X86 decoder. Along these lines, note that the number of lemmas accumulated in Reveal(C) is much smaller than in Reveal(E). On the other hand, Reveal(C) spends more time verifying the abstract model, almost twice as much as Reveal(E), despite Reveal(C)'s smaller number of refinement iterations.

To further assess the effect of lemmas on the convergence of the algorithm, we ran Reveal(C) on a version that combines the three bugs present in X2, X3 and X4. This was an iterative session, in which Reveal was re-invoked after correcting each reported bug. We tested Reveal in two modes: a mode in which learned lemmas are discarded after each run and a mode in which learned lemmas are saved and used across runs. The total runtime for the first mode was 232 seconds, whereas the runtime in the second mode was 166 seconds, a 40% improvement in speed. This confirmed our conjecture that lemmas discovered in one verification run can be profitably used in subsequent runs. The verification of real-life designs involves tens to hundreds of invocations of the tool, thus a significantly larger speedup could be seen in practice.

UCLID exhausts available memory during its CNF encoding stage on most of the variations of this design after approximately 250s. VIS cannot process the input Verilog due to blocking assignments, and VCEGAR halted due to an internal error after parsing.

9 Conclusions

This paper reported on six case studies of automatic formal verification of safety properties, with emphasis on equivalence, using the CEGAR-based Reveal system. This system is particularly suited for the verification of designs that consist of wide datapaths whose operation is orchestrated by complex control logic. Abstraction of the datapath allows the verification system to focus on the control interactions and enables it to scale to much larger designs than is possible if it had to operate at the bit level. All aspects of the Reveal system are automated, making it quite easy to use. In particular, its demand-based lemma generation capability eliminates one of the obstacles that had complicated the deployment of formal verification tools in the past. Finally, its support of Verilog allows it to be directly used by designers.

Based on the experiments in the previous sections we draw the following unified conclusions:

- The performance of many verification tools such as Reveal, BAT, and UCLID depends on the size of the verification condition. This size is determined by the design as well as the property, and can be immensely reduced by applying memory abstraction. Memory abstraction is, thus, essential for the practical verification of hardware models, and it gives these tools an edge over approaches that do not abstract memory such as VCEGAR and VIS.
- Datapath abstraction and counterexample-guided refinement is scalable for various types of designs, including those where datapath and control interleave intensively.
- The refinement loop converges much quicker when multiple refinement lemmas are used per iteration. Although most of the lemmas detected in Reveal are fairly simple, it is infeasible to "predict" relevant ones a priori. Succinct lemmas have two main advantages: they eliminate significant spurious behavior from the abstract model, and they are human-friendly 'empirical data' that gives the verification engineer insight into the abstraction/refinement process for the particular design at hand. The high-level nature of the lemmas also allows them to be re-used. In contrast, propositional clauses often stored (via conflict learning) and re-used in incremental SAT solving cannot be used across verification sessions since they become obsolete when the design is altered, even slightly, for the purpose of fixing bugs or modifying assumptions.
- The performance of methods that leave the datapath unabstracted, such as BAT, Reveal(B), and VIS, is comparable to CEGAR-based tools on relatively small verification conditions. This suggests that an optimal approach might incorporate techniques from both realms.
- The advantage of counting arithmetic does not always reflect positively on the total runtime, as there is a tradeoff between the number of refinement iterations and the time spent on solving the abstract formula.
- While predicate abstraction has been shown to be scalable for model checking in general [16], we conjecture that the need for a large number of predicates is a common occurrence when verifying designs with large memories and for equivalence properties that were the focus of this paper.

References

[1] Z. Andraus and K. Sakallah, "Automatic Abstraction and Verification of Verilog Models," Proc. DAC, pp. 218-223, 2004.

[2] Z. Andraus, M. Liffiton and K. Sakallah, "Refinement Strategies for Verification Methods Based on Datapath Abstraction," Proc. ASPDAC, pp. 19-24, Jan. 2006.

[3] F. Balarin, and A. Sangiovanni-Vincentelli. "An Iterative Approach to Language Containment," In Proc. CAV, LNCS vol. 697, pp. 29-40, 1993.

[4] T. Ball, and S. K. Rajamani. "The Slam Project: Debugging System Software via Static Analysis," In Proc. POPL, pp. 1-3, Jan. 2002.

[5] T. Ball, and S. K. Rajamani. "Boolean Programs: A Model and Process for Software Analysis," Technical Report 2000-14, Microsoft Research, 2000.

[6] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. "Symbolic Model Checking without BDDs," Proc. TACAS, LNCS, Springer-Verlag, pp. 193-207, 1999.

[7] R. E. Bryant, S. K. Lahiri, and S. A. Seshia. "Modeling and Verifying Systems Using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions," In Proc. CAV, LNCS vol. 2404, July 2002.

[8] J. R. Burch, and D. L. Dill. "Automatic Verification of Pipelined Microprocessor Control," In Proc. CAV, LNCS vol. 818, pp. 68-80, 1994.

[9] E. Clarke, O. Grumberg. S. Jha, Y. Lu and H. Veith. "Counterexample-Guided Abstraction Refinement," In Proc. CAV, LNCS vol. 1855, pp. 154-169, 2000.

[10] S. Das, and D. Dill. "Successive Approximation of Abstract Transition Relations," In 16th Annual IEEE Symposium on Logic in Computer Science, pp. 51, 2001.

[11] B. Dutertre and L. de Moura. "A Fast Linear Arithmetic Solver for DPLL(T)," In Proc. CAV, LNCS vol. 4144, pp. 81-94, 2006.

[12] S. Govindaraju, and D. Dill. "Counterexample-Guided Choice of Projections in Approximate Symbolic Model Checking," In Proc. ICCAD, November 2000.

[13] S. Graf, and H. Saidi. "Construction of Abstract State Graphs with PVS," In Proc. CAV, LNCS vol. 1254, pp. 72-83, 1997.

[14] J. L. Hensessy, and D. J. Patterson. *Computer Architecture: A Quantitative Approach.: 2nd edition.* Morgan Kaufman, 1996.

[15] R. Hojati, and R. K. Brayton. "Automatic Datapath Abstraction of Hardware Systems," In Proc. CAV, LNCS vol. 939, pp. 98-113, June 1995.

[16] H. Jain, D. Kroening, N. Sharygina, and E. Clarke. "Word Level Predicate Abstraction and Refinement for Verifying RTL Verilog," In Proc. DAC, pp. 445-450, June 2005.

[17] R. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoritic Approach.* Princeton University Press, 1999.

[18] M. H. Liffiton, M. D. Moffitt, M. E. Pollack, and K. A. Sakallah. "Identifying Conflicts in Overconstrained Temporal Problems," In Proc. IJCAI, August 2005.

[19] P. Manolios, S. K. Srinivasan, and D. Vroon, "Automatic Memory Reductions for RTL Model Verification," In Proc. of ICCAD, November 2006.

[20] G. Nelson and D. C. Oppen. "Simplification by cooperating decision procedures." ACM Transactions on Programming Languages and Systems (TOPLAS), 2(1):245-257, 1979.

[21] A. Pardo, and G. Hachtel. "Incremental CTL Model Checking using BDD Subsetting," In Proc. of DAC, pp. 457-462, 1998.

[22] "VIS: A System for Verfication and Synthesis", The VIS Group, In Proc. CAV, LNCS vol. 1102, pp. 428-432, July 1996.

[23] F. Wang, B. Li, H. Jin, G. D. Hachtel, and F. Somenzi, "Improving Ariadne's Bundle by Following Multiple Threads in Abstraction Refinement," ICCAD 2003, pp. 408-415.

[24] YICES v1.0.3, http://yices.csl.sri.com/

[25] "picoJava-II Microarchitecture Guide"

[26] http://www.sun.com/processors/technologies.html

[27] http://vlsi.cs.iitm.ernet.in/x86_proj/x86Homepage.html

[28] http://www.eecs.umich.edu/vips/stresstest.html
[29] http://www.opencores.org
[30] http://www-static.cc.gatech.edu/fac/Pete.Manolios/bat/
[31] UCLID v1.0, http://www.cs.cmu.edu/~uclid/
[32] VCEGAR v1.1, http://www.cs.cmu.edu/~modelcheck/vcegar/
[33] VIS v2.1, http://vlsi.colorado.edu/~vis/
[34] zChaff v2007.3.12_64bit, http://www.princeton.edu/~chaff/
[35] Cadence SMV v2.4.2-2, http://www.kenmcmil.com/smv.html