Polynomial Formal Verification of a Processor: A RISC-V Case Study

Lennart Weingarten
Institute of Computer Science
University of Bremen
Bremen, Germany
len_wei@uni-bremen.de

Alireza Mahzoon
Institute of Computer Science
University of Bremen
Bremen, Germany
mahzoon@uni-bremen.de

Mehran Goli
Institute of Computer Science
University of Bremen
Bremen, Germany
mehran@uni-bremen.de

Rolf Drechsler
Institute of Computer Science
University of Bremen/DFKI
Bremen, Germany
drechsler@uni-bremen.de

Abstract—**Formal verification is an important task to ensure the correctness of a circuit. In the last 30 years, several formal methods have been proposed to verify various architectures. However, the space and time complexities of these methods are usually unknown, particularly, when it comes to complex designs, e.g., processors. As a result, there is always unpredictability in the performance of the verification tool. If we prove that a formal method has polynomial space and time complexities, we can successfully resolve the unpredictability problem and ensure the scalability of the method.**

In this paper, we propose a Polynomial Formal Verification (PFV) method based on Binary Decision Diagrams (BDDs) to fully verify a RISC-V processor. We take advantage of partial simulation to extract the hardware related to each instruction. Then, we create the reference BDD for each instruction with respect to its size and function. Finally, we run a symbolic simulation for each hardware instruction and compare it with the reference model. We prove that the whole verification task can be carried out in polynomial space and time. The experiments demonstrate that the PFV of a RISC-V RV32I processor can be performed in less than one second.

Index Terms—**Polynomial Formal Verification, PFV, BDD, BDD-based verification, Partial Simulation, RISC-V**

I. Introduction

Nowadays, *Central Processing Units* (CPUs) are working in very high frequencies and they usually contain millions of logic gates. The big size and high complexity of CPUs make them extremely error-prone. Thus, it is important to ensure the correctness of a CPU after the design. In the last decades, researchers have put a lot of effort to develop formal verification methods to attack the hard problem of verifying CPUs. Several formal techniques based on equivalence checking, theorem proving, and model checking have been introduced to prove the correctness of CPUs in different levels of abstraction [1] [2] [3]. Although formal verification methods have reported very good results in practice, the space and time complexities of many of them remained unknown. As a result, there is always unpredictability in the performance of formal methods.

The performance unpredictability causes serious problems in the production process of a CPU: It cannot be predicted before actually invoking the verification tool whether (a) it will successfully terminate or (b) run for an indefinite amount of time. As a result, the time schedule for the implementation and fabrication of a CPU can be affected. In addition, it becomes impossible to estimate the required resources, e.g., computational power and memory usage, to successfully carry out the whole verification process. The unpredictability in performance can be resolved by calculating the space and time complexities. If a verification method has polynomial space and time complexities (i.e., $\mathcal{O}(n^c)$, where n is a circuit parameter, and c is a small positive number) it is scalable and it can be used to verify the bigger instances of the circuit.

In the last few years, several *Polynomial Formal Verification* (PFV) methods have been proposed to verify different types of circuits [4] [5] [6] [7] [8]. The researchers have tried to either prove the polynomial complexity bounds for the existing verification techniques or extend the formal methods to achieve PFV. Several research works have particularly focused on arithmetic circuits. They have proven that the adder [9] and multiplier [10] architectures can be polynomially verified with some conditions on the availability of the design hierarchy. They take advantage of bit-level methods (e.g., *Binary Decision Diagrams* (BDDs)), word-level techniques (e.g., *Symbolic Computer Algebra* (SCA)), or a combination of them in their flow. Moreover, the researchers have also considered PFV of approximate circuits and tree-like circuits. Despite the progress in PFV of various digital circuits, PFV of a complete CPU has not been yet investigated.

In this paper, **we propose a PFV method based on BDDs to fully verify a single-cycle RISC-V processor**. We first illustrate the challenges of BDD-based verification in proving the correctness of a processor. Then, we introduce a divide and conquer strategy to overcome these challenges. We take advantage of partial simulation to extract the hardware related to each RISC-V instruction. Then, we create the reference BDD for each instruction with respect to its size and function. Subsequently, we perform the symbolic simulation for each instruction hardware and obtain the output BDDs. Finally, we compare the reference BDD and the instruction hardware BDD. A BDD is a canonical representation; thus, if the two BDDs are equivalent the correctness is ensured. We also calculate the time complexity of symbolic simulation for each instruction hardware to prove that our method is PFV. The experimental results demonstrate that our method can ensured the correctness of a RISC-V RV32I processor in less than one second. To the best of our knowledge, this is the first work aiming to

polynomially verify a RISC-V processor. The ideas in this work can be extended to more sophisticated RISC-V processors with pipelining.

II. Related Works

Since the 90s, formal verification of processors has been an active field of research [11], [12]. Several methods have been developed in this regards ranging from the verification of pipelined control [13], ensuring the correctness of advanced data-path operations [14], improving scalability [15], HW/SW co-verification [16] to complete formal verification [17], [18].

The methods presented in [19], [20] focus on verification of a processor at the microcode level (i.e., very low-level software). They proposed a tool called MicroFormal that takes advantage of SAT/SMT techniques for backward compatibility checks and assertion-based verification. A technique for the automatic generation of a complete property suite from an architecture description, that can be used to formally verify a *register transfer level* (RTL) implementation of a processor is presented in [21]. It starts with an architecture description of the processor. By defining a number of mapping functions designers capture how the abstract concepts are mapped to the RTL implementation. These mapping functions refer to pipeline stages, stall and cancel signals, and similar objects. The method proposed in [22] enables designers to interconnect RTL and microcode verification by including theorem proving and SAT technique. An improved version of this work has been presented in [23]. Although the aforementioned methods are complementary to our approach and share some conceptual similarities, none of them targeted the PFV of a processor.

The authors of [24] proposed a PFV technique to verify a simple *Arithmetic Logic Unit* (ALU) using BDDs. However, the ALU supports only a limited number of logic and arithmetic operations. Moreover, it does not consider the other components in a CPU.

In this paper, we propose a PFV method to prove the correctness of single-cycle RISC-V processors. We ensure the polynomial upper-bound complexities in theory and demonstrate good performance by experimental evaluations.

III. Preliminaries

In this section, we first introduce RISC-V processors. Then, we review the formal verification method using BDDs.

A. RISC-V

RISC-V is a free and open *Instruction Set Architecture* (ISA) originating from the University of Berkeley [25], [26] that recently gained large momentum in both academia and industry.

A part from the open ISA feature, modularity and extensibility of RISC-V are the other main reasons for its popularity. Base model specifications are provided for 32, 64, and 128-bit integer instructions. The specification provides a long list of standard extensions, for example, integer multiplication/division, atomic instructions, and single/double floating-point precision among many others. The open ISA allows designers to create custom-built extensions, hence provide a great flexibility when designing a system using the RISC-V ISA. In this work, we focus on PFV of the base specification of n-bit RISC-V processors. In the experimental evaluations, we consider the verification of an RV32I base integer instruction set.

B. Formal Verification using BDDs

We briefly present some definitions:

- **Binary Decision Diagram (BDD):** a directed, acyclic graph whose nodes have two edges associated with the values of the variables 0 and 1. A BDD contains two terminal nodes (leaves) that are associated with the values of the function 0 or 1.
- **Ordered BDD (OBDD):** a BDD, where the variables occur in the same order in each path from the root to a leaf.
- **Reduced OBDD (ROBDD):** an OBDD that contains a minimum number of nodes for a given variable order.

We refer to ROBDD as BDD in the rest of the paper since it is the canonical representation that is used in the verification of arithmetic circuits.

The ITE operator (If-Then-Else) is used to calculate the results of the logic operations in BDDs:

$$ITE(f, g, h) = (f \wedge g) \vee (\bar{f} \wedge h). \quad (1)$$

The basic binary operations can be presented using the ITE operator:

$$\begin{aligned} f \wedge g &= ITE(f, g, 0), & f \vee g &= ITE(f, 1, g), \\ f \oplus g &= ITE(f, \overline{g}, g), & \bar{f} &= ITE(f, 0, 1). \end{aligned} \quad (2)$$

ITE can be also used recursively in order to compute the results:

$$ITE(f, g, h) = ITE(x_i, ITE(f_{x_i}, g_{x_i}, h_{x_i}), ITE(f_{\overline{x}_i}, g_{\overline{x}_i}, h_{\overline{x}_i})), \quad (3)$$

where f_{x_i} ($f_{\overline{x}_i}$) is the positive (negative) cofactor of f with respect to x_i, i.e., the result of replacing x_i by the value 1 (0).

The result is computed recursively based on Eq. (3) in this algorithm. When calculating the results of ITE operations for the f, g, h BDDs, the arguments for subsequent calls to the ITE subroutine are the subdiagrams of f, g and h. The number of subdiagrams in a BDD is equivalent to the number of nodes. For each of the three arguments, the sub-routine is called at most once. Assuming that the search in the *Unique Table* is performed at a constant time, the computational complexity of the ITE algorithm, even in the worst-case, does not exceed $O(|f| \cdot |g| \cdot |h|)$, where $|f|$, $|g|$ and $|h|$ denote the size of the BDDs in terms of the number of nodes [27].

In order to formally verify an adder, we need to have the BDD representation of the outputs. Symbolic simulation helps us to obtain the BDD for each primary output. In a simulation, an input pattern is applied to a circuit, and the resulting output values are observed to see whether they match the expected values. On the other hand, symbolic simulation verifies a set of scalar tests (which usually covers the whole input space) with a single symbolic test. Symbolic simulation using BDDs is done by generating corresponding BDDs for the input signals. Then, starting from primary inputs, the BDD for the output of a gate

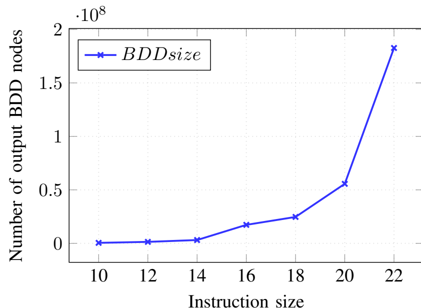

Fig. 1: Growth rate of the RV32I processor

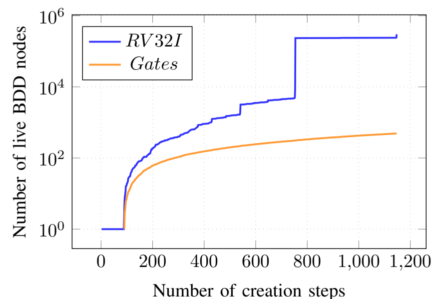

Fig. 2: RISC-V size with different input sizes

(or a building block) is obtained using the ITE algorithm. This process continues until we reach the primary outputs. Finally, the output BDDs are evaluated to see whether they match the BDDs of an adder.

IV. Challenges

In this section, we explain the challenges of verifying a RISC-V processor using BDDs.

A. BDD Size Explosion

While it has been shown that the size of BDD remains polynomial during the verification of some digital circuits (e.g., adders), there are many designs whose BDD sizes change exponentially with respect to the input bit-width (e.g. multipliers [28]). We now investigate the size of BDDs during the verification of single-cycle RISC-V processors.

Fig. 1 reports the overall size of output BDDs after the symbolic simulation of single-cycle RISC-V processors with different instruction sizes. It is evident that the BDD size grows exponentially with respect to the processor instruction size. The output BDD of a 22-bit single-cycle RISC-V processor contains more than 150 million nodes, and it takes more than 3 minutes to generate. We run out of memory during the symbolic simulation of RISC-V processors with a bigger instruction size on a running machine with 32 GByte of main memory.

Fig. 2 depicts the live overall size of BDDs during the symbolic simulation of a 32-bit RISC-V processor (RV32I). At each step of the symbolic simulation, the output BDD of a logic gate is calculated using the ITE operation. It can be seen in Fig. 2 that a huge growth occurs in the size of intermediate BDDs at step 754, where there is a 27× increase in the number of nodes. Eventually, the size of BDDs reach 3 million nodes at step 1147, and we cannot further continue the symbolic simulation as an explosion happens in the number of nodes. As a result, we run out of memory and the verification process remains incomplete.

B. Need for a Reference Model

In order to carry out a successful equivalence checking problem, a reference model has to be available. In the case of BDD-based verification, the reference model is a correct BDD that represents the function of a circuit with respect to its inputs. The reference model can be created by either symbolically simulating a golden circuit (i.e., a correct circuit) or deducting from the circuit function or algorithm. Unfortunately, none of these solutions are practical when it comes to a RISC-V processor.

On one hand, a golden circuit is usually not available in the case of a RISC-V processor. The processors support different sets of instructions; thus, we require a golden circuit for each variation. Moreover, the correctness of a golden model has to be ensured using a formal method (not equivalence checking), which is usually not possible. On the other hand, generating a BDD based on the function of a RISC-V processor is not trivial. Processors contain several instructions with different functionalities. Consequently, it is very difficult to represent the whole function of a CPU as a BDD.

V. Methodology

In this section, we present our methodology to verify a single-cycle RISC-V processor. First, we outline assumptions made for the RTL-Model then we give an overview of our methodology followed by a more detailed description of our partial simulator and reference model generator.

A. RTL-Model Assumptions

For the RTL-Model of the RISC-V a few assumptions were made. For the single-cycle RV32I processor we assume that the inputs (outputs) of registers and memories are the primary outputs (inputs) of the circuit. Thus, the circuit under verification is fully combinational. However, we can still activate each instruction, apply desirable inputs, and observe the expected outputs. Moreover, the variable order is chosen interleaved, since it gives us the smallest BDD sizes.

B. Overview

An overview of our methodology is presented in Fig. 3. It consists of two pathways. The top path describes the process of constructing BDDs for each instruction hardware from the RTL description. The bottom path demonstrates the process of generating reference BDDs of each instruction. Finally, the BDDs from the two paths are checked for equivalency.

The top path in Fig. 3 helps us to overcome the challenge of exponential growth in the BDD sizes using a divide and conquer strategy. We extract each instruction hardware by partial simulation and construct the BDDs using symbolic simulation. Thus, we break the problem into smaller problems, which can be solved in a polynomial time. The main steps in the top path are as follows:

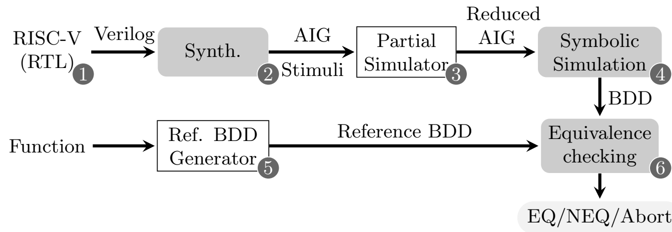

Fig. 3: The proposed verification methodology overview

1) The inputs and outputs of registers and memories are detected; then, they are considered as the new primary outputs and inputs, respectively.
2) The RTL description is converted into the *AND-Inverter Graph* (AIG) representation, which later facilitates the partial simulation.
3) First, the opcode of an instruction is set to specify the instruction type (e.g., R-Type). Then, based on the specified type the remaining parts of instruction (func3 and func7) are assigned. Subsequently, the general parts of instruction (e.g., register address or immediate value) are considered unknown. Finally, the circuit is partially simulated based on the set and unknown values to obtain the instruction hardware. This process is repeated for all instructions.
4) The symbolic simulation is performed for each instruction hardware and the output BDDs are obtained.

The bottom path in Fig. 3 helps us to overcome the problem of the reference model:

5) We generate the reference model individually for each instruction based on its function.
6) The output BDD for each instruction hardware generated by symbolic simulation and the reference BDD are compared to prove the correctness.

In the next sections, we explain the partial simulation and reference model generation in more details.

C. Partial Simulation

The *Partial Simulator* (PSIM) receives two input files: an AIG and a set of stimuli. In the case of a RISC-V processor, the AIG file is the full description of the processor. The stimuli set is selected by following the instruction definition determined by the RISC-V specification manual. For each instruction a stimuli file is used. The stimuli file consists of a list of bit-vectors, while each bit is either set to a value (0 or 1) or specified as unknown (X). In order to extract specific instruction hardware, the stimuli file has to be configured as follows:

- The opcode bit-vector is set to values in order to determine the instruction type (e.g., ADD).
- The func3 and func7 bit-vectors are set to values in order to determine the type of operation (e.g., immediate).
- The remaining bit-vectors which are related to the input of operations (e.g., from registers or immediate values) are set to unknown.

The partial simulation is performed in two levels: In the first step, the values given by the stimuli are applied to the primary inputs of the circuit. Then, each node is evaluated by the input values. The nodes are iteratively simplified and removed if at least one of their inputs is set to a value. Subsequently, the garbage fan-ins which do not have any influence on the circuit outputs are cleaned up and the hardware related to them is removed. Finally, we perform a high-level optimization to detect the group of nodes, which can be reduced to a fewer number of nodes.

D. Reference Model Generator

We introduce a *Reference Model Generator* (RMG) to create reference BDDs for each instruction of an n-bit RISC-V processor. To generate them a known architecture with the same functionality as the instruction is utilized (e.g. for addition the architecture of the *Ripple Carry Adder* (RCA)). RMG has to generate all reference BDDs for the outputs of instruction hardware. These outputs include data_out[$n : 0$] (the result of the operation), zero_flag (one if all bits of data_out is zero), sign_flag (one if data_out is negative), and carry_flag (one if there is a carry out). RMG takes advantage of interleaved variable ordering since it gives the smallest BDDs for most of the instructions, and it matches the variable ordering of symbolic simulation.

For the logic instructions, including AND, OR, and XOR, as well as the immediate variations, i.e., ANDI, ORI and XORI, the reference BDDs of data_out are created by the bit-wise logic operations. The zero_flag BDD is generated by NORing the data_out BDDs. The sign_flag BDD is related to the most significant bit of data_out, i.e., data_out[$n - 1$]. These two flags are generated similarly for the other instructions. Finally, the carry_flag BDD is a constant zero terminal.

There are the ADD and ADDI instructions in a RISC-V processor, that directly compute the addition of inputs. Moreover, there are the jump (e.g., JAL, JALR, LUI, and AUIPC), load, and store instructions that work based on the addition. The jump instructions add an offset to the current address to obtain the final jump address. Similarly, the load and store functions add an offset to the current address to find the location of target data for loading or storing. RMG takes advantage of an adder function to generate the BDDs for the addition-based instructions. Moreover, the BDD for the carry_flag is equal to the carry-out BDD of the addition.

The SUB instruction in a RISC-V processor is performed by subtracting the inputs. However, there are also some other instructions, including branch instructions (e.g., BEQ, BNE, BGE, and BLT) that use subtraction. They compare two register entries through subtraction and decide about the branching. Moreover, the comparison instructions (SLT, SLTI, SLTU, and SLTIU) are done based on subtraction. There is an implementation of the SUB function in RMG that helps with the reference BDD generation of the aforementioned instructions.

Lastly, the SLL and SRL shift operations are simply reconnecting primary input to primary outputs using a given number of shift bits. Thus, we do not require reference BDDs for them.

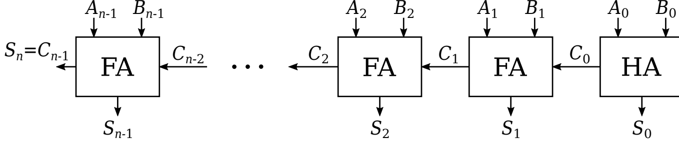

Fig. 4: Ripple Carry Adder

VI. Polynomial Formal Verification

In this section, we first group the instructions based on the underlying ALU operation. The groups are Logic, Additive, Subtractive, and Shifts. Then, we prove that the symbolic simulation of instructions in each group has polynomial time complexities.

A. Logic

The Logic group contains the logic operations, including AND, ANDI, OR, ORI, and XOR, XORI. We calculate the time complexity of symbolic simulation for the AND instruction when it is applied to two inputs $A_{n-1}A_{n-2}\dots A_0$ and $B_{n-1}B_{n-2}\dots B_0$. The bit-wise AND operation can be translated into ITE operation as follows:

$$A_i \wedge B_i = ITE(A_i, B_i, 0). \tag{4}$$

Since there are in total n AND operations, the complexity Cpx of symbolic simulation is calculated as follows:

$$Cpx_{[AND]} = n \cdot |A_i| \cdot |B_i| = 16n = \mathcal{O}(n). \tag{5}$$

The time complexity of symbolic simulation for other logic instructions can be obtained, similarly.

B. Additive

The Additive group contains the operations based on addition, including jump, load, and store operations. We calculate the time complexity of symbolic simulation for addition which can be easily extended to other addition-based operations. We assume that the adder architecture is implemented based on ripple carry algorithm and $A_{n-1}A_{n-2}\dots A_0$ and $B_{n-1}B_{n-2}\dots B_0$ are the inputs.

Fig. 4 presents the structure of an n-bit RCA. In order to obtain the computational complexity of symbolic simulation for an n-bit RCA, we first calculate the complexity of symbolic simulation for a single full-adder. The sum and carry bits of a full-adder can be expressed in terms of ITE operations:

$$\begin{aligned} S_i &= A_i \oplus B_i \oplus C_{i-1} = ITE(C_{i-1}, A_i \odot B_i, A_i \oplus B_i),\\ A_i \odot B_i &= ITE(A_i, B_i, \overline{B}_i),\\ A_i \oplus B_i &= ITE(A_i, \overline{B}_i, B_i). \end{aligned} \tag{6}$$

$$\begin{aligned} C_i &= (A_i \wedge B_i) \vee (A_i \wedge C_{i-1}) \vee (B_i \wedge C_{i-1})\\ &= ITE(C_{i-1}, A_i \vee B_i, A_i \wedge B_i),\\ A_i \vee B_i &= ITE(A_i, 1, B_i),\\ A_i \wedge B_i &= ITE(A_i, B_i, 0). \end{aligned} \tag{7}$$

Note that the size of the BDD for a single variable ($|x_i|$), AND/OR of two variables ($|x_i \wedge y_i|$ and $|x_i \vee y_i|$), and XOR/XNOR of two variables ($|x_i \oplus y_i|$ and $|x_i \odot y_i|$) equals 3, 4, and 5, respectively. As a result, the complexities of computing S_i and C_i are as follows:

$$\begin{aligned} Cpx(S_i) &= Cpx(A_i \odot B_i) + Cpx(A_i \oplus B_i)\\ &\quad + |C_{i-1}| \cdot |A_i \odot B_i| \cdot |A_i \oplus B_i|\\ &= |A_i| \cdot |B_i| \cdot |\overline{B}_i| + |A_i| \cdot |\overline{B}_i| \cdot |B_i|\\ &\quad + |C_{i-1}| \cdot |A_i \odot B_i| \cdot |A_i \oplus B_i|\\ &= 3 \cdot 3 \cdot 3 + 3 \cdot 3 \cdot 3 + |C_{i-1}| \cdot 5 \cdot 5\\ &= 25 \cdot |C_{i-1}| + 54, \end{aligned} \tag{8}$$

$$\begin{aligned} Cpx(C_i) &= Cpx(A_i \vee B_i) + Cpx(A_i \wedge B_i) + |C_{i-1}| \cdot |A_i \vee B_i| \cdot |A_i \wedge B_i|\\ &= |A_i| \cdot |B_i| + |A_i| \cdot |B_i| + |C_{i-1}| \cdot |A_i \vee B_i| \cdot |A_i \wedge B_i|\\ &= 3 \cdot 3 + 3 \cdot 3 + |C_{i-1}| \cdot 4 \cdot 4\\ &= 16 \cdot |C_{i-1}| + 18,\\ Cpx(FA_i) &= Cpx(S_i) + Cpx(C_i)\\ &= 41 \cdot |C_{i-1}| + 72, \end{aligned} \tag{9}$$

where the computational complexities depends on the size of the incoming carry BDD to the full-adder.

It has been proven in [29] that the BDD size of the ith carry bit (C_i) is bounded above by $3i + 4$. Thus, the overall complexity of verifying a RCA can be obtained as follows:

$$\begin{aligned} Cpx_{[RCA]} &= \sum_{i=1}^{n-1} (41 \cdot |C_{i-1}| + 72) = \sum_{i=1}^{n-1} (41 \cdot (3(i-1) + 4) + 72)\\ &= \sum_{i=1}^{n-1} (123 \cdot i + 113) = \mathcal{O}(n^2). \end{aligned} \tag{10}$$

C. Subtractive

The Subtractive group contains the operations based on subtraction, including branch and comparison operations. Subtraction uses the ripple carry architecture, while it adds XOR gates to the primary inputs to allow the negation of one of the operands. The complexity calculation of a subtractor is very similar to an adder and the time complexity of symbolic simulation is bounded by $\mathcal{O}(n^2)$. We do not include the calculation due to page limitations.

D. Shifts

The shift operations are proven by showing the correct input-output correctness for all possible combinations. Since there are n possible variations, the verification complexity is $\mathcal{O}(n)$. We proved that the time complexity of verifying all hardware instructions is polynomial. Thus, we conclude that the PFV of an n-bit single-cycle RISC-V processor is possible using our proposed method.

VII. Experimental Results

We have implemented our PFV method in C++. We used CUDD package [30] to perform operations on BDDs. All experiments are carried out on an Intel(R) Core(TM) i7-8565U CPU with 1.80GHz and 16 GByte of main memory. In order to evaluate the efficiency of our verifier in practice, we consider a 32-bit single-cycle RISC-V processor (RV32I). It contains the four groups of instructions, i.e., Logic, Additive, Subtractive, and Shifts.

Table I reports the verification results for the RV32I processor. The first column **G.** denotes the instruction group based on the underlying ALU operation. The second column

TABLE I: Results of verifying RV32I processor

G.	Inst.	PSIM $[s]$	PFV $[ms]$	#Steps	#Nodes	Peak
Logic	AND	5.97	0.20	161	211	245
	ANDI	6.22	0.22	141	203	218
	OR	6.24	0.22	225	160	194
	ORI	6.05	0.29	205	203	218
	XOR	7.28	0.54	353	349	382
	XORI	7.13	0.67	333	347	362
Additive	ADD	6.35	1.59	872	7692	7725
	ADDI	6.64	2.42	786	6911	6926
	JAL	6.29	1.30	806	7836	7869
	JALR	6.58	1.23	806	7836	7869
	AUIPC	5.55	0.07	113	121	144
	LUI	5.53	0.07	113	121	144
	LB	6.55	2.32	852	6744	6759
	SB	6.57	1.96	786	6911	6926
	LW	6.29	1.99	786	6911	6926
	SW	6.33	2.00	786	6911	6926
Subtractive	SUB	6.56	1.49	811	7773	7806
	BEQ	6.28	1.58	898	7721	7754
	BNQ	6.40	1.57	899	7721	7754
	BGE	6.64	1.45	811	7773	7806
	BLT	6.46	1.58	906	7807	7840
	SLT	5.93	0.35	496	903	936
	SLTI	6.17	0.54	476	740	755
	SLTU	5.97	0.35	489	729	762
	SLTIU	6.05	0.53	469	633	648
Shifts	SLL	193.69	0.16	97	112	131
	SLLI	206.08	0.13	97	112	131
	SRL	176.13	0.13	97	112	131
	SRLI	201.12	0.16	97	112	131
	$\sum$	969.21	27.14	14767	101715	102418

Inst. represents the instruction name. The run-time of partial simulation is reported in the third column **PSIM**. The fourth column **PFV** represents the run-time of symbolic simulation, reference model generation, and comparison. The number of symbolic simulation steps is reported in the fifth column **#Steps**. The sixth column **#Nodes** gives the number of output BDD nodes after the symbolic simulation. Finally, the last column **Peak** reports the BDD nodes' peak size during the symbolic simulation. Please note that the experiments were run 1000 times for each instruction, and the PFV run-times were averaged to avoid outliers.

The overall run-time for partial simulation and PFV is around 16 minutes. Most of this run-time is dedicated to the verification of shift instructions since they require several partial simulations. The overall size of output BDDs merely exceeds 100 K nodes, which is much smaller than the size of output BDDs generated for the entire processor in Fig. 1 and Fig. 2. As a result, our proposed technique can verify an RV32I processor in a short time while it keeps memory usage very low. Moreover, the difference between the size of output BDD and the peak BDD size is subtle. Thus, we never observe a huge increase in the number of nodes during the symbolic simulation.

VIII. Conclusion

In this paper, we presented a PFV method to fully verify a single-cycle RISC-V processor using BDDs. We overcame the challenges of BDD-based verification (i.e., BDD size explosion and reference model) by proposing a divide and conquer approach. The hardware related to each instruction is extracted using partial simulation and its output BDD is obtained by symbolic simulation. Then, it is compared to a generated reference BDD to ensure the correctness. We ensured the polynomial upper-bound complexities in theory and demonstrated good performance by experimental evaluations. Based on our results and the assumptions made about the RISC-V processor, we thereby demonstrated the capabilities of our methodology.

In our future research, we consider the PFV of multi-cycle and pipelined RISC-V CPUs. Moreover, we plan to consider the PFV of RISC-V models with F and M extensions, which include floating point operations and multiplication/division, respectively.

Acknowledgment

This work was supported in part by DFG within the Reinhart Koselleck Project PolyVer (DR 287/36-1).

References

[1] V. Patankar, A. Jain, and R. Bryant, "Formal verification of an ARM processor," in *VLSI Design*, pp. 282–287, 1999.

[2] P. Mishra and N. Dutt, "A methodology for validation of microprocessors using equivalence checking," in *MTV Workshop*, pp. 83–88, 2003.

[3] C. Kern and M. R. Greenstreet, "Formal verification in hardware design: A survey," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 4, pp. 123–193, apr 1999.

[4] R. Drechsler and A. Mahzoon, "Polynomial formal verification: Ensuring correctness under resource constraints," in *ICCAD*, pp. 70:1–70:9, 2022.

[5] R. Drechsler, A. Mahzoon, and M. Goli, "Towards polynomial formal verification of complex arithmetic circuits," in *DDECS*, pp. 1–6, 2022.

[6] J. Kleinekathöfer, A. Mahzoon, and R. Drechsler, "Polynomial formal verification of floating point adders," in *DATE*, 2023.

[7] M. Schnieber, S. Fröhlich, and R. Drechsler, "Polynomial formal verification of approximate adders," in *DSD*, pp. 761–768, 2022.

[8] M. Schnieber, S. Fröhlich, and R. Drechsler, "Polynomial formal verification of approximate functions," in *ISVLSI*, pp. 92–97, 2022.

[9] R. Drechsler, "PolyAdd: Polynomial formal verification of adder circuits," in *DDECS*, pp. 99–104, 2021.

[10] M. Barhoush, A. Mahzoon, and R. Drechsler, "Polynomial word-level verification of arithmetic circuits," in *MEMOCODE*, pp. 1–9, 2021.

[11] W. A. Hunt Jr, "Microprocessor design verification," *J. Autom. Reason.*, vol. 5, no. 4, pp. 429–460, 1989.

[12] K. Hamaguchi, H. Hiraishi, and S. Yajima, "Design verification of a microprocessor using branching time regular temporal logic," in *CAV* (G. von Bochmann and D. K. Probst, eds.), vol. 663 of *Lecture Notes in Computer Science*, pp. 206–219, 1992.

[13] J. R. Burch and D. L. Dill, "Automatic verification of pipelined microprocessor control," in *CAV*, vol. 818 of *Lecture Notes in Computer Science*, pp. 68–80, 1994.

[14] R. Kaivola and K. R. Kohatsu, "Proof engineering in the large: formal verification of Pentium 4 floating-point divider," *Int. J. Softw. Tools Technol. Transf.*, vol. 4, no. 3, pp. 323–334, 2003.

[15] R. Kaivola, "Formal verification of Pentium® 4 components with symbolic simulation and inductive invariants," in *CAV*, vol. 3576 of *Lecture Notes in Computer Science*, pp. 170–184, 2005.

[16] M. D. Nguyen, M. Wedler, D. Stoffel, and W. Kunz, "Formal hardware/software co-verification by interval property checking with abstraction," in *DAC*, pp. 510–515, 2011.

[17] N. Ayewah, N. Kikkeri, and P. Seidel, "Challenges in the formal verification of complete state-of-the-art processors," in *ICCD*, pp. 603–608, 2005.

[18] K. Devarajegowda, M. R. Fadiheh, E. Singh, C. W. Barrett, S. Mitra, W. Ecker, D. Stoffel, and W. Kunz, "Gap-free processor verification by s^{2}qed and property generation," in *DATE*, pp. 526–531, 2020.

[19] T. Arons, E. Elster, L. Fix, S. Mador-Haim, M. Mishaeli, J. Shalev, E. Singerman, A. Tiemeyer, M. Y. Vardi, and L. D. Zuck, "Formal verification of backward compatibility of microcode," in *CAV*, vol. 3576 of *Lecture Notes in Computer Science*, pp. 185–198, 2005.

[20] A. Franzén, A. Cimatti, A. Nadel, R. Sebastiani, and J. Shalev, "Applying SMT in symbolic execution of microcode," in *FMCAD*, pp. 121–128, 2010.

[21] U. Kühne, S. Beyer, J. Bormann, and J. Barstow, "Automated formal verification of processors based on architectural models," in *FMCAD*, pp. 129–136, 2010.

[22] J. Davis, A. Slobodová, and S. Swords, "Microcode verification - another piece of the microprocessor verification puzzle," in *ITP*, vol. 8558 of *Lecture Notes in Computer Science*, pp. 1–16, 2014.

[23] S. Goel, A. Slobodová, R. Sumners, and S. Swords, "Verifying x86 instruction implementations," in *CPP*, pp. 47–60, 2020.

[24] R. Drechsler, A. Mahzoon, and L. Weingarten, "Polynomial formal verification of arithmetic circuits," in *ICCIDE*, pp. 457–470, 2021.

[25] A. Waterman and K. Asanović, "The RISC-V instruction set manual; volume i: Unprivileged ISA," in *SiFive Inc. and CS Division, EECS Department, University of California, Berkeley*, 2019.

[26] A. Waterman and K. Asanović, "The RISC-V instruction set manual; volume ii: Privileged architecture," in *SiFive Inc. and CS Division, EECS Department, University of California, Berkeley*, 2019.

[27] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient implementation of a BDD package," in *DAC*, pp. 40–45, 1990.

[28] R. E. Bryant, "On the complexity of VLSI implementations and graph representations of boolean functions with application to integer multiplication," *TC*, vol. 40, no. 2, pp. 205–213, 1991.

[29] I. Wegener, *Branching Programs and Binary Decision Diagrams*. SIAM, 2000.

[30] F. Somenzi, "CUDD: CU decision diagram package release 2.7.0." available at https://github.com/ivmai/cudd, 2018.