PolyMiR: Polynomial Formal Verification of the MicroRV32 Processor

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

Kamalika Datta
Institute of Computer Science
University of Bremen/DFKI
Bremen, Germany
kdatta@uni-bremen.de

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

***Abstract*—Formal verification techniques ensure completeness as opposed to simulation-based techniques. In general, the process of formal verification is computationally complex, and it is difficult to quantify the exact time and space complexities. Some of the recent works have shown that it is possible to achieve polynomial space and time complexities for verifying specific designs, like arithmetic circuits. However, this cannot be directly extended to complex circuits, like processors. A recent work has reported a formal verification method for the RISC-V processor with polynomial complexity, where only single-cycle instruction execution was considered and it was computation intensive. This method cannot be directly extended to multi-cycle operations, which are typical for most real processors. This paper introduces an improved data structure leading to *Binary Decision Diagram* (BDD) based *Polynomial Formal Verification* (PFV) with support for both single-cycle and multi-cycle operations. We use the MicroRV32 processor as a case study. Our method leads to significant improvement in runtime over the previous method. The entire process of verification can be carried out in polynomial space and time complexities for multi-cycle operations.**

***Index Terms*—Polynomial Formal Verification, RISC-V, BDD**

I. Introduction

With the increasing complexity of present-day designs, it is imperative to verify the designs before they are manufactured. Formal verification techniques are widely practiced by the industry towards achieving this objective. However, such methods are computationally complex and time consuming. Specifically, for complex designs like processors, it is very important to ensure design correctness before manufacturing them. Earlier efforts towards processor verification mostly relied on simulation-based techniques. However, over the last decades, there has been a distinct paradigm shift towards using formal methods for this purpose, and approaches based on equivalence checking, theorem proving, and model checking have been proposed (e.g. [1], [2]). Such methods guarantee completeness in the verification process; however, the space and time complexities of these methods remain unpredictable. This causes major problems during a typical processor design cycle, as it is not possible to predict beforehand the time estimate to complete the verification process. This often results in delays in the fabrication process and the time-to-market for newly manufactured products.

The difficulty of the verification process is typically evaluated in terms of time and space complexities. If the complexity is a polynomial function of n, then the approach can be used to verify bigger circuit instances. In this context, the problem of *Polynomial Formal Verification* (PFV) of circuits has become very important and several works in this area have been reported very recently [3], [4], [5]. These can be broadly classified into two categories – (a) try to establish polynomial complexity bounds for existing verification approaches, or (b) try to extend existing methods to ensure PFV. Some of the works specifically focus on arithmetic circuits, where it has been shown that adder [3] and multiplier [6] circuits can be verified with polynomial time and space complexities. In these works, bit-level methods based on *Binary Decision Diagrams* (BDDs) [7], word-level methods based on *Symbolic Computer Algebra* (SCA), or a combination of both, have been used. However, in spite of good progress in this area, PFV of a complete processor design has been rarely investigated.

In a recent work [8], a BDD-based PFV method has been proposed to verify the single-cycle implementation for a subset of the RISC-V instruction set. This is the first attempt towards PFV, but has limited utility as most practical processors have complex instructions with multi-cycle implementations. Moreover, the method uses complex data structures with implementations that result in higher verification time. Also the method cannot be directly extended to tackle multi-cycle operations due to its limited capabilities.

In this paper, we propose a PFV approach for a RISC-V processor (MicroRV32 [9]) that supports multi-cycle operations in the data path, which is the first such attempt to the best of the authors knowledge.

The main contributions of the present work can be summarised as follows:

- We present our verification results for a more comprehensive set of instructions as compared to earlier work.
- We incorporate improved data structure and code base that results in faster verification time as compared to the previous method, while verifying a more complex processor.
- We implement improved modeling of shift operations that was one of the bottlenecks of the previous approach.

The paper is organized as follows. Section 2 provides the necessary background and related works in this domain. In Section 3 we present the proposed verification methodology. In Section 4 we show how the proposed method ensures PFV. In Section 5 we present the experimental results followed by concluding remarks in Section 6.

II. Background and Related Work

In this section we first provide the necessary background information about the RISC-V processor architecture, and then we briefly discuss BDD-based formal verification techniques. Finally we summarize the related works in the broad area of processor verification.

A. RISC-V Processors

RISC-V is a popular open-source *Instruction Set Architecture* (ISA) developed at the University of Berkeley. It is based on the classical load-store architecture, and forms the foundation of the next-generation *Reduced Instruction Set Computer* (RISC) designs. Two distinguishing features of the RISC-V architecture are its modularity and extensibility. The open-source ISA specification for RISC-V provides base model support for 32-, 64-, and 128-bit integer instructions, and as such can be extended to include newer functionalities. It encourages the designers to customize the ISA to suit specific applications, by allowing them to create customized add-ons, thereby leveraging the enormous flexibility that it offers. This design flexibility, however, also adds to the complexity of processor verification and it becomes a challenge to complete the process within a reasonable amount of time.

B. BDD based Formal Verification

A *Binary Decision Diagram* (BDD) [7] is a popular data structure used for compact representation of Boolean functions. It is basically a finite *Directed Acyclic Graph* (DAG) with a designated *root node*. Each *non-leaf node* is labeled with a variable (say, x), and consists of two outgoing edges corresponding to $x = 0$ and $x = 1$ respectively. There are two terminal nodes labelled with constant values 0 and 1. An *Ordered BDD (OBDD)* is a BDD with a specified variable ordering that is consistent for every path from the root to the terminal nodes. By repeated application of a set of reduction rules, an OBDD can be converted to a *Reduced Ordered BDD* (ROBDD), which is a canonical representation containing minimum number of nodes. In the rest of the paper, we shall be using the terms BDD and ROBDD synonymously.

One of the most important applications of BDD is to verify the equivalence of two functions. The basic idea is that the ROBDD of two equivalent functions will be identical for the same variable ordering. We can use the *If-Then-Else* (ITE) operator [10] to carry out function transformations for achieving this.

The function ITE can be invoked recursively to decompose a given function progressively, as illustrated below for one step:

$$ITE(f, g, h) = ITE(x_i, ITE(f_{x_i}, g_{x_i}, h_{x_i}), ITE(f_{\overline{x}_i}, g_{\overline{x}_i}, h_{\overline{x}_i})) \quad (1)$$

In Eqn. (1), f_{x_i} and $f_{\overline{x}_i}$ are the positive and negative co-factors of f corresponding to the variable x_i. We get the result by replacing x_i with either 0 or 1. For the algorithm the result is calculated recursively using Eqn. (1). While generating the result for the BDDs (f, g and h), the BDD sub-graphs corresponding to f, g and h nodes are required. These sub-graphs are passed as arguments for subsequent calls to the ITE function. The number of nodes present in the BDD will be equal to the number of sub-graphs. The function is called at most once for each of the three arguments. Under the assumption that a search in the *Unique Table* can be performed in constant time, the worst-case computational complexity of the ITE algorithm will be $O(|f| \cdot |g| \cdot |h|)$, where $|f|$, $|g|$ and $|h|$ represent the sizes of the BDDs with respect to the number of nodes [10].

Now for the purpose of formally verifying a multi-output circuit, we need to build the BDDs for each of the outputs. We use symbolic simulation to generate the BDD for each output function. In a conventional simulation run, all the possible input values are applied to a circuit model, and the outputs are matched against the expected values. But using symbolic simulation we can use a single symbolic test that generally covers the whole input space. We begin the symbolic simulation by generating the BDDs for the corresponding input variables. Then using the ITE algorithm we get the BDD of the output(s) of the gate or building block that are directly fed from the primary inputs. The process is repeated till we reach one of the primary outputs. At the end the output BDDs are verified to check whether there is match against the original function specification.

C. Related Works

Formal verification has been an active area of research over the last several decades. Some of the techniques that have been explored include model checking, theorem proving, equivalence checking, and using formal specification languages. Some specific attempts in this regard range from verifying the operation of the control unit in pipelined processors [11], verifying correctness of complex data path operations [12], co-verification of hardware and software modules [13], etc. In [14], the authors have proposed techniques for verifying the functional operation of a processor at the microcode level, and developed an automated tool that uses SAT/SMT solvers for backward compatibility check coupled with assertion-based verification. In an alternate approach, authors in [15] proposed a technique for generating the complete property suite from a given architectural specification, which can be used for verifying the processor functionality at the *Register Transfer Level* (RTL). The verification approaches proposed in [16], [17] employ theorem proving techniques along with SAT solver to verify the processor microcode as well as RTL operations. It may be noted that none of the above approaches target *Polynomial Formal Verification* (PFV) of a given processor specification. In [18], the authors proposed a BDD-based PFV approach to verify the functional operation of a very simple *Arithmetic Logic Unit* (ALU) that support single-cycle operations; however, the other processor components are not considered. In a very recent work [8] the authors have proposed PFV method to verify a processor, but only single-cycle operation in the datapath is considered.

In this paper, we propose a generalized PFV method to prove the correctness of RISC-V processors supporting both single-cycle and multi-cycle operations. We establish polynomial upper-bound complexities in both space and time, and

demonstrate good performance through rigorous experimental evaluation.

III. Proposed Verification Methodology

A. Motivation

There have been very few prior works that consider PFV of digital circuits in general and processor verification in particular. In [8], a PFV scheme for a single-cycle RISC-V processor was proposed, where an instruction is assumed to be executed in a single cycle. In other words, for every instructions a pure combinational circuit is extracted for execution, with the clock period greater than the maximum instruction latency. For verification, it is necessary to have a reference model against which the extracted circuit specifications can be matched. The authors in [8] used a BDD-based reference model that models the function of each instruction. Also the verification time is higher due to complex code base and implementation strategies. One of the main drawbacks of this work is that it can be used only for single-cycle data paths, and for a limited set of RISC-V instructions. However, this work can be regarded as the first step towards a unified PFV solution for processor architectures.

Fig. 1 shows the high-level operation of single-cycle and multi-cycle execution of instructions. Fig. 1(a) shows the basic execution control in a single-cycle implementation, where the inputs (i.e., the instruction to be executed) are fed to a combinational circuit that produces the outputs (i.e., results from the execution). Fig. 1(b) shows the operation schematic for a multi-cycle MicroRV32 processor [9], where the instruction cycle is divided into four stages, *Fetch* (IF), *Decode* (ID), *Execute* (EX), and *Write Back* (WB). The normal sequence of execution of an instruction is IF $\rightarrow$ ID $\rightarrow$ EX $\rightarrow$ WB, which can be disturbed in the presence of *interrupt* or *trap* during instruction execution. This paper presents an improved

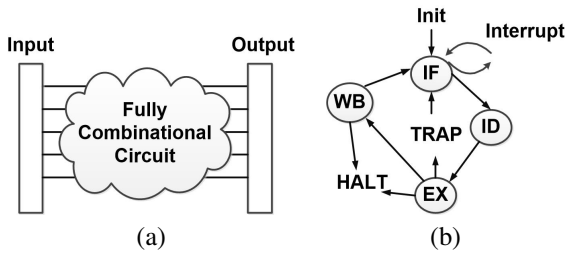

Fig. 1. Overall operation schematic for: (a) single cycle processor, and (b) multi-cycle processor.

PFV approach for multi-cycle implementations of processor architecture. As a case study, we consider the MicroRV32 processor [9], with the following assumptions:

a) The instruction cycle is split into four simpler stages, viz. (IF), (ID), (EX), and (WB). Each of these stages can be executed in a single clock cycle.
b) All instructions may not need all the four stages for execution. Thus, the number of clock cycles required may vary from one instruction type to another, with the maximum being 4.
c) We assume a non-pipelined processor implementation. This implies that the instructions are executed sequentially, and there is no overlap in the execution of two instructions.

Let k denote the number of stages with execution times $\{t_1, t_2, \ldots, t_k\}$, and N denote the number of instructions. For a single-cycle implementation, the clock period is lower bounded by $\Delta_{single} \geq \sum_{i=1}^{k} t_i$. However, in the corresponding multi-cycle implementation, the clock period will be lower bounded by $\Delta_{multi} \geq \max_{i=1}^{k}\{t_i\}$. The total execution time in the single-cycle and multi-cycle implementations can be estimated as:

$$T_{single} = N\Delta_{single} \quad (2)$$

$$T_{multi} = \sum_{i=1}^{N} \delta_i \Delta_{multi} \quad (3)$$

where δ_i denotes the number of cycles required to execute the i^{th} instruction. In general, $T_{single} > T_{multi}$.

B. Overall Methodology

In this subsection we present the model assumptions and the overall verification flow of the proposed approach.

We use the SpinalHDL [19] model of the RISC-V processor, and consider certain assumptions regarding extraction of the instruction execution stages from the model. The interconnections between the stages are assumed to be free from any faults. The overall verification flow used is shown in Fig. 2. For the considered MicroRV32 processor, the RTL specification is first exported into its equivalent Verilog description. Subsequently, for verification the following steps are carried out in sequence.

a) The RISC-V processor core represented in SpinalHDL is first converted into an Verilog RTL representation, and then into an *AND-Inverter Graph* (AIG) [20] representation.
b) The functionalities of the different stages (viz., IF, ID, EX and WB) are then extracted from the AIG description, and converted into equivalent AIG representations using partial simulation (see Section III-C).
c) For every instruction type, the AIGs for the stages that are actually used in the instruction are combined together. The final AIG is converted into an equivalent BDD representation using symbolic simulation. We generate one output BDD for each instruction type.
d) In the next step, we generate the reference model for our design, i.e., the reference BDD. We have used a divide-and-conquer strategy to limit the exponential growth in the BDD size.
e) Finally, we perform equivalence checking to verify whether the two BDD representations are functionally equivalent or not.

In this work, both the functional specifications of individual instructions and the corresponding reference models are expressed in the form of BDDs. To handle multi-cycle instruction execution, the present work supports storage units like latches in the graph structure. We divide the larger problem of processor verification into smaller sub-problems each of which can be solved in polynomial time.

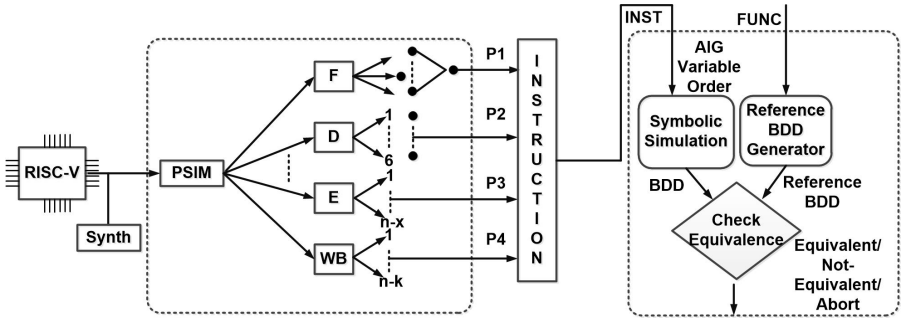

Fig. 2. Proposed verification methodology for PFV

C. Partial Simulation

Partial simulation is an important step in the verification flow. The *Partial Simulator* (PSIM) tool takes two inputs, an AIG and a set of stimuli. In the present work, the AIG contains the complete specification of the RISC-V processor. In the proposed framework as shown in Fig. 2, we have used an improved data structure for the intermediate representation in the form of AIG. In the previous work [8] CUDD [21] was directly incorporated in the AIG-data-structure itself. In contrast, the present work separates the BDD generation and the AIG representation, where a symbolic simulator is used for generating the BDDs. The stimuli set is obtained from the instruction descriptions as specified in the RISC-V manual. A final configuration file is created by consolidating the stimuli files for all the instructions. Stimuli consist of a set of bit vectors, with the individual bits being '0', '1' or 'X' (unknown). To extract the hardware corresponding to the instruction, we need to suitably configure the stimuli. Firstly, depending on the instruction type, the opcode bit vector is set accordingly. Other bit vectors in the instruction encoding are set to specify the type of the operation (e.g. register or immediate operand, etc.). The other bit-vectors that depend on the operand values (e.g. register or immediate) are set to 'X'. For more details about the RISC-V standard see [22], [23].

In the first step of partial simulation, we apply the values specified in the stimuli to the primary inputs of the circuit under consideration, and each node of the graph is evaluated. In an iterative simplification step, we observe whether an input to a node is a fixed value; if so, we remove the node. Further, the garbage fan-ins that do not impact the circuit outputs and also the associated hardware, are removed.

D. Generation of Reference Model

The *Reference Model Generator* (RMG) is a key component in the verification process. For the RISC-V architecture, we first generate a library consisting of one reference BDD for each of the instructions. To generate the reference BDD, we consider a simplified architecture for a functionally equivalent realization of an instruction. As an example, for the `ADD` instruction, we consider an implementation that uses an *Ripple Carry Adder* (RCA). The RMG must create all the reference BDDs for the outputs of the instruction processing hardware.

In order to generate the smallest possible BDDs for a majority of the instructions, RMG exploits the benefit of interleaved variable ordering for some of the instructions like `ADD`, `SUB`, which also matches with that generated by symbolic simulation.

The RISC-V instruction set contains logical instructions like `AND`, `OR`, and `XOR`, and also their counterparts that use immediate addressing, like `ANDI`, `ORI` and `XORI`. For such instructions, bitwise logical operations are used to generate the reference BDDs.

In RISC-V, the addition function is used variously in the different instructions. For the `ADD` and `ADDI` instructions, an adder is needed to add the operands to produce the result. Similarly, for the `LOAD` and `STORE` instructions, we need to compute the effective address of an operand in memory by adding the contents of a register with an offset value specified as part of the instruction. And for instructions that modify the value of the *Program Counter* (PC), like `JAL` and `JALR`, we need to add an offset specified in the instruction with the value of the PC.

The subtraction instructions, `SUB` requires subtraction of one operand from the other. The branch instructions (like `BEQ`, `BNE`, `BLT` and `BGT`) require comparison of two register operands and decide based on the outcome. This is performed by subtracting the operands and checking the relevant flag. A very similar structure appears for the various SET instructions (like `SLT`, `SLTU`, etc.), where we need to subtract two register operands. In RMG, there is an implementation of the subtraction operation, which can be used to generate the reference BDDs for all these instructions.

The shift operations, `SLL` and `SRL`, do not involve any arithmetic or logical computation to generate the results. We implement a generic shift operation for various shifts and their immediate counterpart.

IV. Ensuring Polynomial Formal Verification

In this section we show how polynomial formal verification can be achieved for the RISC-V processor architecture, by providing upper bounds for space and time complexities for the corresponding multi-cycle processor design. The four stages in the instruction execution cycle for the RISC-V processor are considered individually for the verification process.

The (IF) stage loads one full instruction from the memory into the instruction register, and can be achieved in constant time. Hence, the complexity is $O(1)$. In the (ID) stage, we consider two different scenarios depending on the type of instruction. An instruction can be either register-to-register

type (e.g., ADD, SUB, etc.) or register-immediate type (e.g., ADDI, SLLI, etc.). Depending on the instruction type, the instruction execution is expressed in terms of its components: R – register, I – immediate, S – load and store, B – branch, U – upper immediate, and J – jump.

All instruction types, except R-type, use immediate values and therefore utilize the Extension Unit. Independent of those two scenarios the verification time should be $O(1)$. For the memory constraint using BDDs the upper bound is dependent on the number of input bits n and the number of instruction types k supported by the decoder $k \cdot O(n)$.

Depending on the instruction type, the (EX) stage will perform the operations specified by the instruction. The complexity of this stage heavily depends on the complexity of the instruction; therefore, it will be upper bounded by the most complex instructions. We group the instructions depending on the underlying functionality, as follows: *Logic*, *Shift*, *Addition* and *Subtraction*. Similar to the approach used in [8], we argue about the complexity of each operation, by utilizing symbolic simulation and the properties of the ITE-operator. For the *Logic* group (AND, ANDI, OR, ORI, XOR, XORI) the instruction time complexity is bounded by $O(n)$, as for each of the n input bits one of the logic gates is executed to calculate the result.

The *Shift* group of instruction (SLL, SRL, etc) can shift the contents of a register by maximum n bit positions, where n is the number of bits in a register. As there are n different possibilities, the complexity will be $O(n)$. Since the time complexity of verifying all the instructions as part of the RISC-V ISA has been shown to be polynomial, it is possible to achieve PFV of the multi-cycle RISC-V processor using the proposed approach. For the addition the upper bound has already been shown to be $O(n^2)$ [3][8]. Hence we consider the same time complexity in this work.

The *Subtractive* group of instructions consists of all the instructions that utilize the subtraction operation during execution. This includes the subtraction instruction itself SUB, comparison (e.g., SLT, SLTU, etc.) and branch (e.g., BEQ, BNE). We consider that the subtraction operation $A - B$ is carried out using an adder as $(A + B' + 1)$, where B' denotes the ones complement of B. We assume that a RCA is used for the addition operation, an array of n XOR gates produce the ones complement of the second operand, and the addition by 1 is achieved by setting the carry input to 1. The overall complexity will be similar to that of addition, with the time complexity of symbolic simulation bounded by $O(n^2)$.

V. Experimental Evaluation

The experiments are conducted on a Thinkpad T490 with Intel i7-8565U CPU (1.80GHz) and 16GB of memory. The proposed PFV method, partial simulation, reference model creation and equivalence checking are all implemented using C++. For BDD generation, the CUDD package [21] is used. We have used the MicroRV32 multi-cycle processor specification [9] to evaluate our proposed verification method. The MicroRV32 core supports the base instruction set I and M, C extensions with CSR Registers for SW traps and timer IRQ. In this work we focus on the verification of the base 32I.

TABLE I
Results of ALU, Decode- and Extension Unit(DEU)

G.	Inst.	ALU				DEU	
		PSIM $[ms]$	PFV $[ms]$	Nodes	Peak	PSIM $[ms]$	PFV $[ms]$
Logic	AND	4.41	0.10	97	100	0.83	0.09
	ANDI	4.13	0.09	97	100	0.91	0.09
	OR	4.66	0.09	97	100	0.77	0.10
	ORI	4.13	0.09	97	100	0.83	0.10
	XOR	4.48	0.11	161	164	0.72	0.10
	XORI	4.66	0.12	161	164	0.84	0.11
Shifts	SLL	5.57	0.73	1632	1671	0.84	0.10
	SLLI	4.59	0.73	1632	1635	0.87	0.10
	SRL	4.91	0.62	1431	1434	0.78	0.10
	SRLI	4.83	0.69	1431	1434	0.88	0.10
	SRA	4.77	0.75	1396	1399	0.72	0.10
	SRAI	4.86	0.80	1396	1399	0.83	0.09
Addition	ADD	12.33	0.54	1327	1330	5.14	0.24
	ADDI	4.73	0.42	1327	1330	0.81	0.09
	JAL	4.87	0.42	1327	1330	0.75	0.06
	JALR	4.53	0.47	1327	1330	0.88	0.08
	AUIPC	4.62	0.41	1327	1330	0.83	0.07
	LUI	4.64	0.42	1327	1330	0.84	0.07
	LB	4.37	0.50	1327	1330	0.82	0.10
	SB	4.95	0.45	1327	1330	0.85	0.06
	LW	4.84	0.32	804	807	0.87	0.07
	SW	4.59	0.28	804	807	0.74	0.07
	LH	5.10	0.26	804	807	0.96	0.08
	SH	4.98	0.33	804	807	0.95	0.09
	LBU	4.35	0.29	804	807	0.89	0.08
	LHU	4.98	0.30	804	807	0.97	0.08
Subtraction	SUB	5.49	0.42	1415	1418	1.01	0.12
	BEQ	4.74	0.33	951	954	0.92	0.10
	BNE	4.42	0.26	951	954	0.77	0.10
	BGE	4.44	0.25	951	954	0.81	0.10
	BLT	4.53	0.26	951	954	0.87	0.10
	BLT	4.53	0.26	951	954	0.87	0.10
	BLTU	4.47	0.26	954	957	0.78	0.10
	BGEU	4.40	0.25	954	957	0.83	0.09
	SLT	4.98	0.69	992	995	0.79	0.07
	SLTU	4.63	0.65	809	812	0.82	0.11
	SLTIU	5.01	0.67	809	812	0.84	0.10
	$\sum$	182.23	14.99	35797	35944	35.39	3.51

The verification results are reported in Table I for the ALU, *Decode and Extension Unit* (DEU). The first column (G) represents the group of the various operations, and the second column (Inst) represents the instruction. The next two columns show the time for partial simulation (PSIM) and the polynomial formal verification (PFV) respectively (in ms), while the next two columns respectively denote the number of BDD nodes (Nodes) and the peak number of nodes required during the BDD creation (Peak). It may be noted that the third to sixth columns represent the results for the ALU unit.

The last two columns in the table reports the PSIM and PFV values for the DEU respectively. The number of nodes and peak values are not mentioned in the table, where the values are 65 and 68 respectively for all the instructions. The sum for number of (Nodes) and (Peaks) are 2405 and 2516 respectively. PFV includes the times for BDD generation, BDD reference model generation and comparison of both models.

From Table I, we can infer a more general view of the results. As expected, the most simple instructions (viz., the *Logic* group) consist of the least number of nodes, as compared

to all other groups. The time taken for partial simulation is significantly higher than the verification process. From Table I, the number of Nodes for individual group of operations are similar (e.g. for Addition group it is 1327 for most and 804 for all the load and store instructions).

In the subtraction group, the `SUB` instruction has the most number of nodes. This is because for the SUB instruction, subtraction is defined for all the output bits; however, for all other instructions in the group only the last sign extended node is used for comparison. All other output nodes correspond to the zero BDD.

From the two columns Nodes and Peak, we can infer that the number of temporarily required nodes (peaks) is slightly above the number of exact nodes to represent the function itself.

The total time to partially simulate the ALU stage takes around 0.18 seconds ($182.23ms$), while the total verification time is 0.015 seconds ($14.99ms$) and for DEU the partial simulation takes 0.035 seconds ($35.39ms$) and the verification takes 0.003 seconds ($3.51ms$). This results in total partial simulation time to 0.22 seconds and the verification time 0.019 seconds.

A. Comparison with Previous Work

We incorporate more operations compared to the previous work in [8] – specifically, nine new operations have been added. Moreover, in the previous work the instructions were run using single-cycle whereas in this work we use a multi-cycle processor that consists of various stages wherein we report the results for ALU and DEU. For our method the total time for the verification process is less than half a second, as compared to 16 minutes reported previously. It can be seen from Table I that the runtime for verification is significantly less as compared to [8]. In the earlier work the shift operations were generated manually; for each shift operation, all possible shift combinations were generated which required several partial simulations. In our work we only need one general reference model; hence we implement a generic shift operation for all (`SLL, SRL, SRA`) and their immediate counterparts. Also, the use of improved data structures and code base in the present work helps in reducing the overall runtime.

VI. Conclusion

In this paper we present a *Polynomial Formal Verification* (PFV) method to verify a 32-bit multi-cycle RISC-V processor (MicroRV32). We perform partial simulation to extract information about the hardware associated with each instruction, and then carry out symbolic simulation to generate the corresponding output BDD. To ensure the equivalence, we generate a reference BDD that is compared with the BDD generated using symbolic simulation. The use of improved data structure, code base and generic shift implementation results in significantly less PFV time compared to state-of-the-art methods. We further show that the polynomial upper bound complexity is indeed achieved. As a future work we plan to verify a pipelined processor.

Acknowledgment

This work was supported in part by DFG within the Reinhart Koselleck Project PolyVer (DR 287/36-1) and partly by the German Federal Ministry of Education and Research (BMBF) within the ECXL project under grant no. 01IW22002. We are grateful to Sallar Ahmadi-Pour for his support in providing the MicroRV RISC-V code and helpful guidance.

References

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

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

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

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

[5] J. Kleinekathöfer, A. Mahzoon, and R. Drechsler, "Polynomial formal verification of floating point adders," in *DATE*, 2023, pp. 1–2.

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

[7] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *TC*, vol. 35, no. 8, pp. 677–691, 1986.

[8] L. Weingarten, A. Mahzoon, M. Goli, and R. Drechsler, "Polynomial formal verification of processor: A RISC-V case study," in *24th International Symposium on Quality Electronic Design (ISQED)*, 2023, pp. 1–7.

[9] S. Ahmadi-Pour, V. Herdt, and R. Drechsler, "The microrv32 framework: An accessible and configurable open source risc-v cross-level platform for education and research," *Journal of Systems Architecture*, vol. 133, p. 102757, 2022.

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

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

[12] 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.

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

[14] 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*, ser. Lecture Notes in Computer Science, vol. 3576, 2005, pp. 185–198.

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

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

[17] S. Goel, A. Slobodová, R. Sumners, and S. Swords, "Verifying x86 instruction implementations," in *International Conference on Certified Programs and Proofs (CPP)*, 2020, pp. 47–60.

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

[19] C. Papon, "SpinalHDL: Scala based HDL," https://github.com/SpinalHDL/SpinalHDL, 2021.

[20] A. Kuehlmann, M. K. Ganai, and V. Paruthi, "Circuit-based boolean reasoning," in *Proceedings of the 38th annual Design Automation Conference*, 2001, pp. 232–237.

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

[22] 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.

[23] ——, "The risc-v instruction set manual; volume ii: Privileged architecture," in *SiFive Inc. and CS Division, EECS Department, University of California, Berkeley*, 2019.