Complete and Efficient Verification for a RISC-V Processor using Formal Verification

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

Abhoy Kole
Cyber-Physical Systems
DFKI GmbH
Bremen, Germany
Abhoy.Kole@dfki.de

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

***Abstract*—Formal verification techniques are computationally complex and the exact time and space complexities are in general not known, which makes the performance of the process unpredictable. Some of the recent works have shown that it is possible to carry out formal verification with polynomial time and space complexities for specific designs like arithmetic circuits. However, the methodology used cannot be directly extended to complex designs like processors. A recent work has shown polynomial verification of a single-cycle RISC-V processor with limited functionality, which considers only the combinational parts of the ALU. In this paper we propose for the first time a complete verification approach that covers all the functional units of the processor, and at the same time considers its sequential behavior. Experimental results show that the verification can be carried out in polynomial time, and also demonstrate significant improvement over previous methods.**

***Index Terms*—Polynomial Formal Verification (PFV), RISC-V, Multi-Cycle, BDD**

I. Introduction

The high design complexity of modern-day processors makes the resulting circuits error-prone, which makes it very important to ensure design correctness before manufacturing. Previous efforts towards processor verification mainly rely on high-level fault models and simulation-based design verification. However, in recent times, formal methods for processor verification are being investigated, e.g. methods based on theorem proving, equivalence checking and model checking [1], [2]. Although these approaches guarantee completeness in verification, their time and space complexities remain uncertain.

The process of verification is typically evaluated in terms of space and time complexities. It is possible to verify larger circuit instances, if the complexity is a polynomial function of the circuit size n. In this regard the *Polynomial Formal Verification* (PFV) [3], [4] of circuits have drawn attention of researchers. The goal of PFV is to attain an upper bound for the time and space complexity for a given function for the verification process. Not all designs may result in achieving a polynomial upper bound.

To this end a number of works have been reported recently [4]–[8]. A recent work [9] proposed a PFV method to verify a single-cycle processor implementation for a limited set of RISC-V instructions. Here only combinational circuit blocks in the functional units are considered for verification. Although this method can be considered as an initial effort towards PFV for processors, it cannot be extended for multi-cycle implementations with sequential sub-circuits. Also this method incurs higher verification time due to complex implementation strategies. Needless to say, single-cycle implementation of instructions has limited utility in processor design, and hence in this paper we introduce a PFV method for a RISC-V processor (MicroRV32 [10]) that includes multi-cycle operations in the data path. A complete and efficient verification strategy is proposed, covering all the functional units of the processor (both combinational and sequential). We consider the base instruction set RV32I which includes all arithmetic instructions except multiplication and division, which are part of RISC-V extensions. The major contributions of the work is summarized below:

- We propose an improved data structure and code base for verifying a multi-cycle processor implementation that includes sequential sub-systems.
- We fully verify all the functional units of the MicroRV32 [10] processor like `Fetch`, `Decode`, `Extension`, `Execute` and `Control`.
- We incorporate improved implementation of partial/symbolic simulation and reference model generation for the verification methodology.

The paper is organized as follows. Section II provides the necessary background and related works in this domain. In Section III we present the complete verification methodology. In Section IV we show how the proposed method ensures PFV. In Section V we present the experimental results followed by concluding remarks in Section VI.

II. Background

A. Polynomial Formal Verification (PFV)

Knowing the time and space complexities of the design verification helps in planning the design process for a chip. This might also allow for a faster time-to-market, allowing delivery of a product that exactly conforms to the design specification. In this regard PFV techniques are of vital importance where the upper bounds of time and space complexities can be obtained. The objective of PFV is to investigate if polynomial upper bounds of time and space complexities can be determined for a given design. In classical formal verification techniques that use methods like *Binary Decision Diagram* (BDD), *Binary Moment*

Diagram (BMD), *Boolean Satisfiability* (SAT), or *Symbolic Computer Algebra* (SCA), the main objective is to prove correctness according to the specification and to ensure the absence of design errors. Apart from ensuring correctness, PFV also focuses on the resources needed to confirm it. The next sub-section discusses various approaches that exist in literature.

B. Related Works

Although PFV targets to ensure polynomial upper bounds and scalability, very few works have been carried out in this area so far. One of the very first works demonstrate PFV for Wallace-tree like multipliers using $*$BMD-verification [11] but was restricted to a theoretical analysis only. Very recently PFV of arithmetic circuits has gained some momentum and various works have been reported towards ensuring polynomial upper bounds [3], [4], [7], [8]. In [3] PFV of adders have been investigated. In [4] the authors show how the polynomial upper bounds can be leveraged under resource constraint scenarios. Here both bit-level and word-level complexities for various arithmetic circuits and *Arithmetic Logic Units (ALU)* are explored. In [8] verification of multiplier is performed using BDDs. In [7] PFV of floating-point adders are explored, which uses case splitting technique to ensure polynomial upper bound.

So far mostly PFV of arithmetic circuits and ALUs have been considered leaving the verification space of processors largely unexplored. A recent work [9] attempts PFV method to verify a single-cycle processor where the primary focus has been on the verification of the Execute unit. Another work [12] targets verification of multi-cycle processor where PFV of Decode and Extension units along with Execute unit has been considered. None of these works have performed PFV of a complete processor.

In this paper we consider a multi-cycle MicroRV32 processor [10], and perform the complete PFV of all the stages present therein. MicroRV32 does not support pipeline. It has multiple stages but only one instruction is executed at a time. This work focuses on the base instruction set RV32I and does not consider any extension.

III. Complete Verification Methodology

In this section we discuss the verification methodology used in this work. In particular we discuss about the data structures and the verification strategy that have been adopted for incorporating sequential logic. We discuss about the functional extraction and simulation required, and also the reference model generation.

A. Verification Methodology

We propose a PFV approach for multi-cycle processor architectures, which is motivated by the fact that most real-world processors fall in this category. We use the MicroRV32 processor architecture as a case study [10].

Fig. 1 shows the overall verification methodology. The specification of the MicroRV32 processor is provided in *SpinalHDL*, which is an abstract RTL specification. We use the *Scala Build Tool* (SBT) to convert SpinalHDL to Verilog. The Verilog

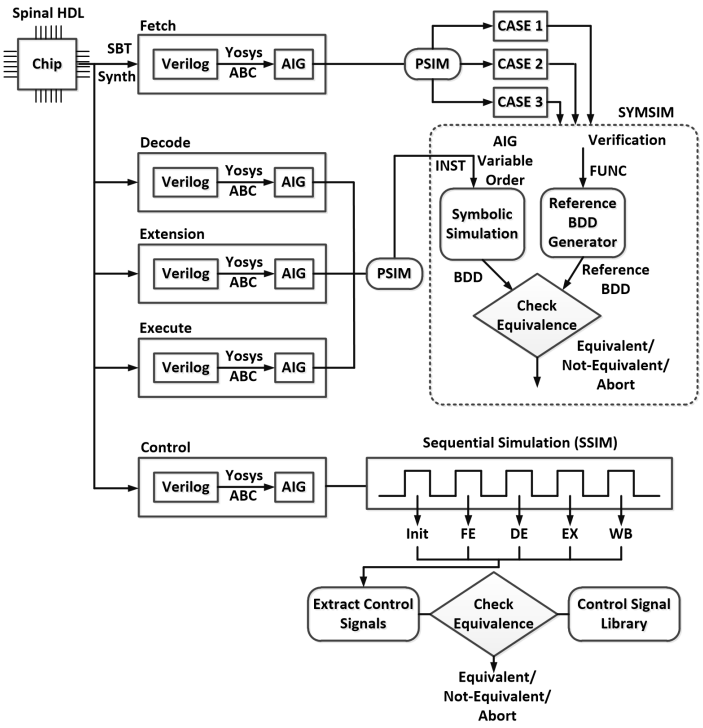

Fig. 1. The verification methodology

representation is then synthesised using *Yosys* and the *Berkeley-ABC tool* into the *And-Inverter Graph* (AIG) data structure. The following steps are performed for the verification.

1) We generate the AIG file for each sub-system (stage) of the processor using the pre-processing step mentioned above.
2) Since a single unified approach cannot be used to verify all the processor units, we use partial/symbolic and sequential simulation for verifying the `Fetch` and `Control` units, and partial simulation for `Decode`, `Extension` and `Execute` units. *Partial Simulation* (PSIM) simulates a circuit with ternary values ('x', '0', '1') and extract functionalities and reduces the graph size. *Sequential Simulation* (SSIM) extends the ternary simulation of PSIM over multiple clock cycles. This has been compared with unrolling the circuit as it is done in the case of *Bounded Model Checking* (BMC). *Symbolic Simulation* (SYMSIM) uses variables instead of Boolean values in the simulation process. SYMSIM is used to generate BDDs. A detail explanation is provided in the later sub-sections.
   a) For the `Decode`, `Extension` and `Execute` units, we utilize PSIM to extract sub-circuits describing the underlying functionality for each instruction. For the verification SYMSIM is used.
   b) For `Fetch` unit we consider three different cases that are extracted using PSIM and verified using SYMSIM.
   c) The `Control` unit is the most critical component in verification, and needs a lot of careful consideration, as it models the entire processor behavior as a *Finite State Machine* (FSM). For the supported instruction types,

two pathways in the FSM are identified:

$$P_1 = IN \rightarrow FE \rightarrow DE \rightarrow EX \rightarrow FE$$
$$P_2 = IN \rightarrow FE \rightarrow DE \rightarrow EX \rightarrow WB \rightarrow FE$$

The S-type instructions uses P_2, while all other instruction types use P_1. To verify the pathways we utilize SSIM to extract each state of the `Control` unit as a sub-circuit.

3) For an extracted sub-circuits we use SYMSIM to first generate a BDD, and then use a reference model generator to create a reference BDD. Finally to verify the circuits we compare the corresponding BDD with the reference BDD.

B. Functional Extraction and Simulation

The extraction of functionality is carried out by partially simulating a circuit with given stimuli as inputs using the PSIM or SSIM tool. Each node is evaluated according to its input. Several reduction steps are incorporated to reduce the size of the graph. Nodes evaluating to a constant can be removed and fan-ins are removed if they do not affect the circuit outputs resulting in an reduced AIG. PSIM is only done for one clock cycle while SSIM allows for simulating the circuit for one or multiple clock cycles. After simulating, the reduction of AIG and the extraction of functionality is performed.

We separate out the processes of BDD generation and AIG representation, i.e. the simulation and reduction of the circuit are separated to allow simulation of multiple cycles. After completion of the simulation, the circuit is reduced and exported. We use the SYMSIM simulator for BDD generation.

We acquire the stimuli set from the description of the instructions as mentioned in the RISC-V ISA specification. A resultant configuration file is generated by merging all the stimuli files for all the instructions. In general, either '0', '1' or 'x' (unknown) values are present in the bit-vectors of the stimuli files. The stimuli file is configured accordingly to extract the hardware for the respective instructions. Initially the opcode bit vector is set depending on the instruction type. In the instruction encoding other bit vectors are set to define the type of operations (e.g., immediate or register operand). The remaining data bit vectors that are dependent on operand values are assigned to 'x'.

C. Sequential Logic Support

One of the major contributions of this paper over the previous works [9], [12] is that it is able to handle sequential elements therein (like latches and flip-flops). To support sequential elements like latches in the AIG format, our parser and simulation code base has been extended. We extended the PSIM to allow for sequential support, which we call SSIM. The AIG parser was extended to handle latches defined in the AIG header, which was not considered in the earlier work. Two different latch definitions were added to the parser to allow for handling a latch with and without reset/initialisation state. For the simulation code base a new node type has been added to the graph structure for specifying latches. This updated data structure can elegantly model the sequential elements in the design. The phases of graph reduction and multi-cycle simulation have been separated out for the purpose of simulating the complete instruction set. This allows multi-cycle simulation to be carried out before the phase of graph reduction.

Fig. 2 shows the example of a 2-bit left/right shift circuit. In this work with the extension of the parser and data structure we are able to handle sequential circuits. But it may be noted that this is like unrolling in BMC that works for sequential circuits with limited sequential depth.

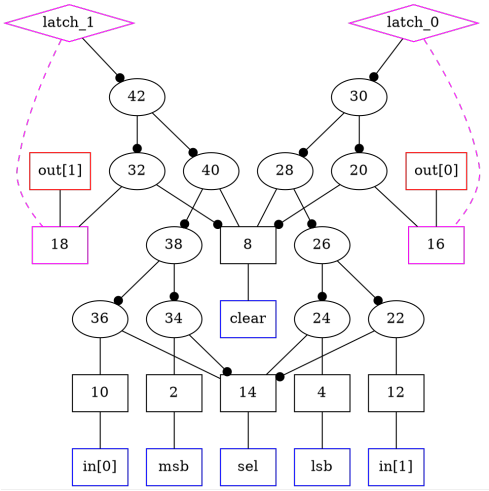

Fig. 2. 2-bit left/right shift example with memory elements

D. Reference Model Creation

The *Reference Model Generator* (RMG) plays an important role in the verification phase. For the considered MicroRV32 architecture, a reference BDD is generated for each of the instructions, and a library is created consisting of all the reference BDDs. To generate a reference model, a simplified but functionally equivalent architecture for executing the individual instructions is considered. The RMG must also generate all the reference BDDs for all the outputs of the instruction hardware. As the complexity of the generated BDDs greatly depends on variable ordering, we exploit this to generate efficient BDDs, which is also the same as the one generated by SYMSIM.

For the `Fetch` unit three cases are considered for extraction: (i) the reset state, (ii) reading from the buffer, and (iii) reading from buffer and writing data into the buffer of the `Fetch` unit. Reference models are generated for all the cases.

For `Decode`, `Extension` and `Execute` units, a reference model is generated for each of the supported instructions. We use bitwise logical operations for reference BDD generation for the logical instructions of the MicroRV32 instruction set like `OR`, `XOR`, and `AND` and their immediate analogue, e.g. `ORI`, `XORI`, and `ANDI`.

There exists a number of instructions where the addition operation is used. For `ADD` and `ADDI` instructions we need an adder to add the operands and generate the result. For the jump instructions like `JAL` and `JALR`, an offset is added to the program counter to generate the target address. Likewise for

LOAD or STORE instructions, we calculate the effective address of an operand in memory by adding the contents of a register with some offset value as specified in the instruction. The RMG generates the reference BDDs for all such instructions using the adder function. The SUB instruction subtracts one operand from another. Before a branch instruction (like BEQ, BNE, BLT and BGT), some comparison is done. The comparison is generally performed by subtracting two register values, viz. using the SET instructions (like SLT, SLTU, etc.). For all such instructions, we use a suitable implementation of the subtractor. This function in turn helps to generate the reference BDD for comparison. For the shift operations like SRL and SLL, generally no arithmetic computation is involved. A generic shift operation is implemented for logic and arithmetic shift and their immediate counterparts.

And for the Control unit, each state of the FSM is matched with the correct golden response for the particular state (i.e., the golden response) for each instruction type. From the specification of each instruction and each state, the required control signals to be generated are stored in a library. We then perform SSIM and extract the control signals generated for each state. Then for each instruction, we compare the control signals extracted from the SSIM with those stored in the library.

IV. Polynomial Analysis

In this section we discuss about the polynomial time complexities for all stages of the processor. Three cases are required to be verified for the Fetch unit. For an n-bit processor, the verification of reset operation has a run-time complexity of $\mathcal{O}(n)$, as it involves resetting the respective register associated with the Fetch stage, followed by assessing the logic states of all n bits from the register in an iterative fashion. Similarly, the read operation requires verifying the logic values stored in the buffer by a prior write operation and it encompasses simulation of a given input value to write it back in the buffer in constant time (i.e., $\mathcal{O}(1)$), and then read the n-bit buffer state to compare with the input bit-vector in $\mathcal{O}(n)$ time. Finally, the verification complexity of the read and write operations together requires repeating the simulation of assigning an input value to the buffer and then reading the buffer state to compare with the input value two times, i.e. $\mathcal{O}(2n)$. This results in a polynomial verification complexity of $\mathcal{O}(n)$ $\big(\approx \mathcal{O}(n) + \mathcal{O}(n) + \mathcal{O}(2n)\big)$ for the considered Fetch unit.

The verification of the Decode and Extension units together requires consideration of different scenarios depending on the type of operands; register-register type (e.g., ADD, SUB, etc.) or register-immediate type (e.g., ADDI, SLLI, etc.) as well as instruction types: R – register, I – immediate, S – load and store, B – branch, U – upper immediate, and J – jump. Since all the instructions except R-type are of register-immediate type, to isolate them from the rest, the SYMSIM infers a bit-wise MUX as a part of the Extension unit. It can be noted that translation of a MUX as an ITE operation of 3 variables, i.e. $ITE(s, A_i, B_i)$ results in a BDD of size 7 whereas for a single variable it is 3. In view of the ITE analogy of SYMSIM reported in [9], the overall complexity of verifying both the units together for an n-bit processor is bounded polynomially by $\mathcal{O}(n^2)$ ($\approx n \cdot |s| \cdot \sum_{i=0}^{n-1} |A_i| \cdot |B_i| = n^2 \cdot |s| \cdot 3^2$).

The verification of the Execute unit is heavily influenced by the operation types: *Logic*, *Shift*, *Addition* and *Subtraction*. The run-time complexity of SYMSIM for *Logic* (e.g., AND, ANDI, OR, ORI, XOR, and XORI) and *Shift* (e.g., SLL, and SRL) group of operations is bounded by $\mathcal{O}(n)$, while for the *Addition* (e.g., ADD, and ADDI) group it is $\mathcal{O}(n^2)$. Under the *Subtraction* group, besides the subtraction instruction itself (SUB) it also encompasses comparison (e.g., SLT, SLTU, etc.) and branch (e.g., BEQ, and BNE). The subtraction operation can be re-described as 2's complement addition, i.e. $A - B = A + B' + 1$ where B' denotes 1's complement of B. Assuming the use of a *Ripple Carry Adder* (RCA) for realization with an array of n XOR gates for producing 1's complement of the subtrahend (B) and an initial carry input 1, the complexity of SYMSIM for the *Subtraction* group attains similar bounds to that of *Addition* group, i.e. $\mathcal{O}(n^2)$. It has also been shown that not only RCA but more complex adders can also be polynomially verified [13]. Thus, the Execute unit verification also has the polynomial run-time complexity bounded by $\mathcal{O}(n^2)$.

Finally, the Control unit verification involves SSIM of its FSM states, i.e. $FE \rightarrow DE \rightarrow EX \rightarrow WB$ followed by evaluating the status of control signals associated with a particular state against the expected golden reference. Considering simulation of the individual FSM state and formal verification of an control signal status as atomic (i.e., $\mathcal{O}(1)$), the complexity of simulating a processor with m states defining the Control unit FSM together with verifying control signal status is $\mathcal{O}(m+m) \sim O(m)$. Thus, for k number of instructions the verification complexity has a polynomial bounds of $\mathcal{O}(km)$ which can be optimized further considering only a sample instruction of each type supported by the target processor, e.g. R-, S-, B-, U-, J-, and I-type instructions for the MicroRV32 processor under verification.

V. Experimental Results

All the experiments have been carried out on a system with an Intel i7-8565U CPU (1.80GHz) with 16GB of main memory (Thinkpad T490). All the tools PSIM, SSIM and SYMSIM are implemented using C++. A divide-and-conquer approach is utilized to simplify the entire verification process of the multi-cycle processor. To extract the underlying functionality, we use an in-house PSIM and SSIM tool. To generate the BDDs required in verification, we use SYMSIM using CUDD [14]. For each instance, multiple experiments are conducted and the average values reported as the final result.

We present the results for all the four stages of the MicroRV32 processor, viz. Fetch, Control, Execute (ALU), and Decode- and Extension unit (DEU). The main contribution of this work is the inclusion of the sequential subsystems (Fetch and Control units) in the verification of the entire processor. For completeness we include the results of DEU and Execute Unit from [12]. Because the extension unit is part of the decode, it was not verified separately.

For the verification of the Fetch unit, three cases are considered. We extract the sub-circuits for each of the cases and evaluate it individually. For the reset state, all primary outputs are checked to be zero. To verify the second case for reading from the buffer, and the third case for both reading and writing from/to buffer, we use SYMSIM. SYMSIM allows us to verify the sequential sub-circuits, by simulating variables through the sub-circuits and evaluating them to verify if they are present in specific memory/primary output addresses after given number of clock cycles.

TABLE I
RESULTS OF FETCH UNIT

Cases	PSIM $[ms]$	PFV $[ms]$	#Var
Reset Buffer	3.42	0.22	65
Read Buffer	0.66	0.11	33
Read & Write Buffer	0.61	0.11	65
Total	4.69	0.44	163

The results for the Fetch unit are presented in Table I. The first column represents all the Cases of fetch, the second and third columns respectively represent the PSIM and the PFV times, both in milliseconds. The last column #Var shows the number of variables needed for the SYMSIM. The PSIM time for reset case is longer as compared to the other two cases. This is because for the reset case the extraction step of the PSIM reduces the entry graph to a vector of zeros. For the reset case the verification time involves generation of BDDs as a reference model. But for the other two cases, we only use SYMSIM and equivalence checking. The number of variables needed for reset and read & write cases consist of 32 bits each for input and output plus a terminal BDD node, for a total of 65. For the read case 32 variables and one terminal variable are used.

The Control unit models a FSM. As mentioned previously two different pathways P_1 and P_2 have to be considered for the MicroRV32 processor (Section III(C)). P_1 is used for all instruction types except the S-type (i.e. load and store instructions), which utilizes the second pathway P_2. To verify the Control unit, the circuit is split into its FSM states, which is carried out using SSIM. Each state is then verified by comparing it to a golden reference of control signals, which are extracted from the specification.

TABLE II
RESULTS OF CONTROL UNIT

Inst. Type	Path	SSIM $[ms]$	PFV $[ms]$	#CtrlS
R	P_1	9.07	0.36	39
I	P_1	8.75	0.40	56
S	P_2	12.46	0.49	74
B	P_1	7.64	0.28	37
U	P_1	9.08	0.39	54
J	P_1	9.03	0.68	48
Total		56.03	2.6	308

Results for the Control unit are presented in Table II. The first two columns respectively lists the instruction type, and the path used in the FSM. The time needed for PSIM and PFV (in milliseconds) are presented in the next two columns. The number of control signals #CtrlS needed is provided in the final column.

For verifying the Init, Fetch and Decode states, we compare the generated control signals from SSIM with the library of control signals present for each state. Then for the Execute state we consider 6 types of instructions as mentioned in Table II. For each of the instruction types, SSIM generates the control signals that are checked with the control signals from the library. Finally the Writeback state is verified in a similar way. In Table II, the SSIM time and PFV time include the times for Init, Fetch, Decode, Execute and Writeback. And #CtrlS is the total number of control signals required for each state.

The verification results for the Execute unit (ALU) and Decode- and Extension Unit (DEU) are presented in Table III [12]. For completeness we have added the result and discussion for Execute and DEU units from [12]. The results for the ALU unit are presented in columns 3-6, while those for the DEU unit are presented in columns 7-8. The first two columns in the table indicate the instruction category and the specific instruction respectively. The instruction category represents the nature of the operation being carried out during the execution of the instruction. The third column PSIM denotes the time for carrying out partial simulation, while the fourth column denotes the time for carrying out polynomial formal verification PFV, both in milliseconds. The PFV value shown is the total time taken for the creation of BDD, generation of the BDD reference model, and checking for equivalence of the two models. The fifth and sixth columns denote the number of nodes #Nodes in the BDD and the Peak number of nodes observed during the creation of the BDD. The last two columns in the table denote the values for PSIM and PFV respectively in milliseconds for the DEU unit. Since the number of BDD nodes and the corresponding peak values of all the instructions are the same for the DEU unit (viz., 65 and 68), they are not explicitly shown in the table. The total number of BDD nodes and peak values are 2405 and 2516 respectively.

From the Table III the number of BDD nodes increases with the complexity of the operation(s) involved. This can be verified from the fact that the Logic group of instructions require smallest number of BDD nodes, which is larger for the other groups. In terms of the total run time, PSIM dominates PFV e.g., for the ALU unit of the AND instruction, the times taken for PSIM and PFV are 4.41ms and 0.10ms respectively.

Least amount of time and memory is used for the Logic verification. Shift instruction group needs the most number of nodes compared to all other instructions due to the modeling of a generic shift. Two outliers in terms of PSIM or #node can be identified. ADD uses the most PSIM time of the Addition group and all other groups for the ALU and DEU verification. SUB does not differ much in terms of PSIM time compared to other instructions in the Subtraction group, but the number of nodes needed is significantly more for the ALU verification. In total the verification time for the ALU takes about 200ms

TABLE III
RESULTS OF EXECUTE (ALU), DECODE- AND EXTENSION UNIT (DEU)

G.	Inst.	ALU PSIM $[ms]$	ALU PFV $[ms]$	ALU Nodes	ALU Peak	DEU PSIM $[ms]$	DEU 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

and 40ms for DEU. The total require nodes for ALU are about 37k and 2.5k for DEU.

A. Improvement over [9], [12]

It is pertinent to compare the contributions of the present work with the previous similar works [9], [12]. Additions and enhancements have been incorporated that have led to significant improvements in the performance of the verification tool.

The changes are summarized as follows:

a) The present work uses improved data structures and code bases that helps in improving the run times significantly. In particular, support for sequential circuit element has been included that is limited to low sequential depth.
b) For the simulation of sequential circuits SSIM has been created, and extension of PSIM is incorporated to allow for ternary simulation over multiple clock cycles.
c) Extension of our code base, parser and data structures, were extended to support the AIG format for latch description.
d) Results for Reset and Control units have been incorporated.

VI. CONCLUSION

In this paper we present a complete PFV approach for verifying the MicroRV32 architecture that implements a multi-cycle processor. We use an improved data structure and code base for sequential logic support for lower sequential depth. A previous approach considers only the PFV of a single-cycle processor with combinational support for Execute unit only. Whereas in this work we show how PFV of multi-cycle processor can be performed with both combinational and sequential components to fully verify all the functional units of a processor. We first perform a pre-processing step to generate the AIG of the processor and then use partial simulator to extract the functionalities. We use BDDs to verify the extracted functionalities of the processor. We incorporate sequential simulation to verify the complete processor. Experimental results confirm that our method can perform verification of the complete processor at very low cost.

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] J. Davis, A. Slobodová, and S. Swords, "Microcode Verification - Another Piece of the Microprocessor Verification Puzzle," in *ITP*, ser. Lecture Notes in Computer Science, vol. 8558, 2014, pp. 1–16.
[2] S. Goel, A. Slobodová, R. Sumners, and S. Swords, "Verifying x86 instruction implementations," in *CPP*, 2020, pp. 47–60.
[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. R. Burch, "Using BDDs to Verify Multipliers," in *DAC*, 1991, pp. 408–412.
[6] M. Barhoush, A. Mahzoon, and R. Drechsler, "Polynomial Word-Level Verification of Arithmetic Circuits," in *MEMOCODE*, 2021, pp. 1–9.
[7] J. Kleinekathöfer, A. Mahzoon, and R. Drechsler, "Polynomial Formal Verification of Floating Point Adders," in *DATE*, 2023, pp. 1–2.
[8] J. Kumar, Y. Miyasaka, A. Srivastava, and M. Fujita, "Formal Verification of Integer Multiplier Circuits Using Binary Decision Diagrams," *TCAD*, vol. 42, no. 4, pp. 1365–1378, 2023.
[9] L. Weingarten, A. Mahzoon, M. Goli, and R. Drechsler, "Polynomial Formal Verification of Processor: A RISC-V Case Study," in *ISQED*, 2023, pp. 1–7.
[10] 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," *JSA*, vol. 133, p. 102757, 2022.
[11] M. Keim, R. Drechsler, B. Becker, M. Martin, and P. Molitor, "Polynomial Formal Verification of Multipliers," *FMSD*, vol. 22, no. 1, pp. 39–58, 2003.
[12] L. Weingarten, K. Datta, and R. Drechsler, "PolyMir: Polynomial Formal Verification of the MicroRV32 Processor," in *NANOARCH*, 2023.
[13] A. Mahzoon and R. Drechsler, "Polynomial Formal Verification of Prefix Adders," in *ATS*, 2021, pp. 85–90.
[14] F. Somenzi, "CUDD: CU Decision Diagram Package Release 2.7.0," available at https://github.com/ivmai/cudd, 2018.