Scalability Matters: Future-Proof Formal Verification of a RISC-V Arithmetic Logic Unit

Luca Müller
DFKI GmbH
Bremen, Germany
luca.mueller@dfki.de

Mohamed Nadeem
University of Bremen
Bremen, Germany
mnadeem@uni-bremen.de

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

***Abstract*—The open and royalty-free RISC-V *Instruction Set Architecture (ISA)* defines base instruction sets with different integer register widths, accounting for instructions with up to 128-bit operands. This poses challenges for verification as a central part of *Electronic Design Automation (EDA)*, since formal methods that provide complete design coverage inherently lack in scalability. To avoid pitfalls when it comes to building future-proof systems, *Polynomial Formal Verification (PFV)* aims to combine these two factors, ensuring scalability for a growing class of circuits, while preserving full coverage.**

This work builds on these foundations, introducing a novel approach to verify RISC-V *Arithmetic Logic Units (ALUs)*. Scalability is achieved by decomposition techniques, ensuring provable complexity bounds. Different ALU implementations can be integrated with minimal effort and individual instructions are extracted independently of architecture, based solely on the RISC-V specification. Theoretical examination shows that polynomial complexity bounds can be established for all base instructions of a RISC-V ALU. Experimental evaluation shows that these theoretical polynomial complexity bounds lead to practically scalable formal verification for an open-source RISC-V ALU, enabling efficient verification for instructions with 32, 64 and 128-bit operands.

I. Introduction

Looking into the future is not a trivial task. A promising strategy in this regard is to learn from the past and deduce which errors to avoid. This strategy is currently being employed in the context of virtual address spaces, where initial considerations for a future of 128-bit *Central Processing Units (CPUs)* are being made, projecting a concrete need by 2040 [1]. The starting point for CPU design is the *Instruction Set Architecture (ISA)*, which acts as an interface specification. 128-bit address spaces are discussed in the specification document of the open RISC-V ISA [2], demonstrating the relevance of this emerging topic.

But new ideas reveal new challenges. One such challenge is ensuring that these larger hardware designs are correct, creating a need for proper verification in the *Electronic Design Automation (EDA)* process. Modern simulation-based techniques promise high efficiency, but increasing the number of bits makes the problem space explode, diminishing coverage [3]. Formal methods, on the other hand, can ensure complete coverage, but also face scalability issues themselves [4].

Recently, these scalability issues have started to be addressed with *Polynomial Formal Verification (PFV)*, which aims to provide formal verification techniques with polynomial time and space complexity, combining full coverage with efficient algorithms [5]. In this scope, different types of circuits, including adders [6], counters [7] and simple *Arithmetic Logic Units (ALUs)* [8] were considered, utilizing verification methods encompassing *Binary Decision Diagrams (BDDs)* [9], *Boolean Satisfiability (SAT)* [10] and *Answer Set Programming (ASP)* [11]. PFV of RISC-V CPUs has also been studied in previous work [12], [13], but evaluation is limited to only 32-bit variants, which impedes insight into scalability.

To address these limitations, we introduce a scalable PFV approach applicable to arbitrary RISC-V ALU designs of different sizes, extracting individual instructions in a ternary simulation process. ALUs are central components of CPUs and have to deal with operand sizes defined by the architecture, making them especially sensitive to larger address spaces. For all RISC-V base instructions, we establish that polynomial complexity bounds can be proven. We experimentally show that these polynomial bounds translate to scalable formal verification on an example open-source RISC-V ALU design with 32, 64, and 128-bit operands, demonstrating that our approach is future-proof. In summary, we make the following contributions.

- Introduce a novel, scalable PFV approach applicable to arbitrary RISC-V ALU designs, making use of instruction extraction and decomposition techniques.
- Show that theoretical polynomial complexity bounds can be established for all RISC-V base instructions.
- Experimentally confirm that theoretical polynomial bounds lead to practically scalable, future-proof formal verification for an open-source RISC-V ALU with 32, 64, and 128-bit operand instructions.

The remainder of this work is structured as follows. Section II provides preliminary information and discusses related work to keep this work self-contained. Section III introduces decomposition techniques which are key to ensure scalability. Section IV gives an overview of our proposed verification approach and goes into detail on relevant parts. Section V applies our approach to an open-source RISC-V ALU implementation, showing both theoretical and practical scalability. Section VI finally concludes this work, giving a brief outlook on future work.

II. Preliminaries

A. RISC-V

RISC-V [2] is an open and royalty-free ISA built on *Reduced Instruction Set Computer (RISC)* principles, gaining popularity in academia and industry alike [14]. It is designed for extensibility, offering different base instruction sets, e.g., *RV32I*, which provides sufficient instructions to support modern operating system environments. Further instructions are added through extensions, e.g., the *M* extension for integer multiplication and division, or the *F* extension for single-precision floating-point.

RISC-V offers support for different integer register widths, referred to as *XLEN* in the specification. Current base integer variants are *RV32I* and *RV64I*, providing 32-bit and 64-bit address spaces, respectively. Previous versions also included a draft for a 128-bit variant *RV128I* [15] and it is still mentioned as a potential future base ISA when the need for 128-bit address spaces arises, emphasizing on the relevance of planning ahead. Regardless of *XLEN*, RISC-V *I* base sets offer a total of 32 general-purpose registers, with one hard-wired zero value and 31 registers for free use, indicating that instruction lengths remain unchanged. Thus, the ALU is most affected by an increase of XLEN, as it has to deal with larger operands. This makes a RISC-V ALU a suitable target to study the effect of scalability on formal verification for circuits used in practice.

B. SAT-based Formal Verification

SAT is a decision problem, answering the question whether a propositional formula of a Boolean function is satisfiable, i.e., evaluates to 1 under any assignment of variables [16]. One such propositional formula in *Conjunctive Normal Form (CNF)* [17] is referred to as a SAT instance, which is typically passed to a SAT solver to find an answer. SAT has a variety of practical applications [18], including its use in *Computer-Aided Design (CAD)* [19]. Especially for verification, formal methods have gained traction not only in academia, but also in industry [20]. Apart from *Formal Property Verification (FPV)* [21], which proves that the *Design Under Verification (DUV)* fullfils a given set of properties, a standard way to employ SAT for formal hardware verification is *Combinational Equivalence Checking (CEC)*. Here, the DUV is compared against a *Reference Circuit (RC)* implementing the specified behavior for equivalence [22], formally proving complete correctness under all possible inputs. To achieve this, a miter circuit is constructed, unifying the inputs of both circuits and connecting their corresponding outputs via *XOR* operations. A mismatch, i.e., a bug in the DUV, is found if the value 1 can be observed at any output. This corresponds directly to satisfiability, hence converting the miter to a SAT instance via Tseitin transformation [23] and giving it to a SAT solver directly solves the CEC problem.

In the general case, solving the CEC problem directly as described above induces exponential time complexity in the number of inputs of the miter, as all input combinations have to be tested to confirm that none of them result in a high edge on any output. To alleviate this exponential behavior, previous work has focused on splitting up the problem with the help of decomposition, achieving linear verification times for some adder circuits [10]. This demonstrates the potential of SAT to achieve true scalability of formal verification by providing polynomial complexity bounds in the number of inputs.

C. Related Work

There are several works which focus on the verification of RISC-V CPU and ALU designs [24]. For instance, the authors of [25] propose a framework for the verification of open-source RISC-V cores, but neither achieve full coverage nor report on any scalability aspects. The *RISC-V Formal Verification Framework* [26] also offers a set of formal testbenches for processor designs, along with a formal specification. Using the framework for a custom core requires some manual work of connecting it to a specified interface, while we aim to require only minimal modifications if any. Once again, runtime and scalability guarantees are not given.

The authors of [12] and [13] specifically target PFV of a RISC-V processor using BDDs, with a focus on the ALU. Although they show theoretical polynomial complexity bounds on all operations of RV32I, they do not focus on other XLEN values beyond 32 in their experimental evaluation, limiting the significance on scalability.

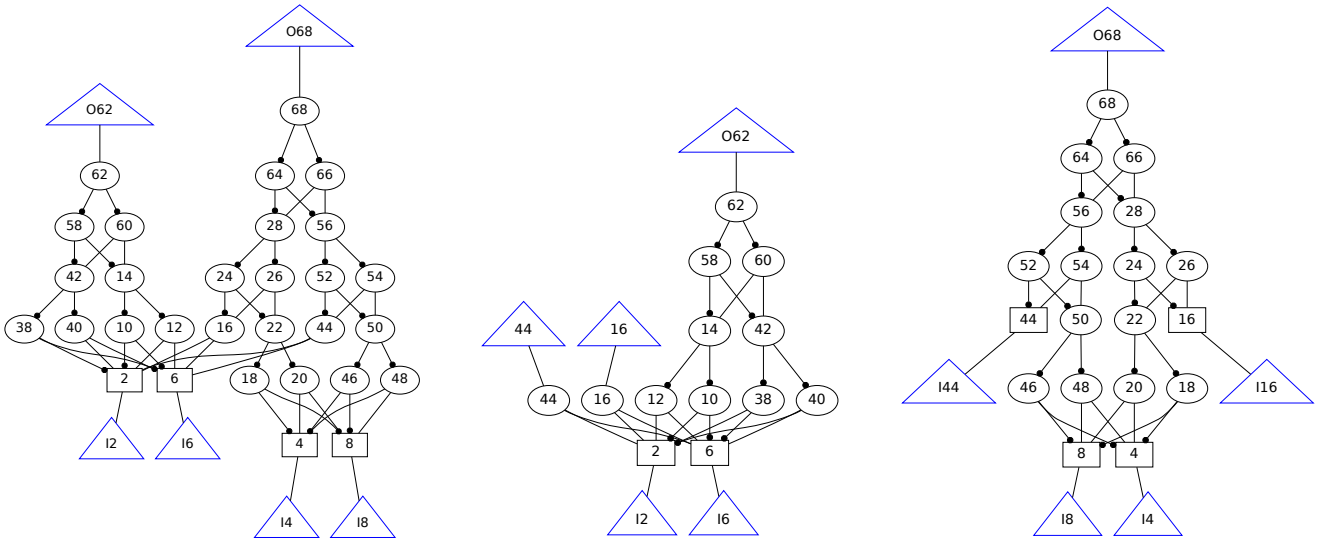
III. Decomposition Techniques

A. And-Inverter Graphs

The divide-and-conquer principle of splitting up a difficult problem into multiple simpler sub-problems is widely established. Decomposition has turned out to be an essential technique in this regard, allowing to prove reliable complexity bounds on formal verification methods [10]. The starting point for all decomposition techniques used in the following are *And-Inverter Graphs (AIGs)* [27], which are graph representations of combinational circuits, where each inner node represents an AND gate with potentially inverted inputs.

Figure 1(a) depicts an example of such an AIG, in particular a representation of a miter circuit between two *Ripple Carry Adders (RCAs)* with input-width $n = 2$ as a simple showcase. For consistency with the ALU model we employ later, the outgoing carry of both adders is disregarded. The $2 * n = 4$ input nodes, $I2, I4, I6$ and $I8$, as well as the 2 output nodes, $O62$ and $O68$ are indicated by triangle shapes. Inverters are indicated by dots at the end of connecting edges between an AND gate and its inputs.

Naively, this AIG representation could be translated into a SAT instance via Tseitin transformation and then solved directly. For our example, this leads to 16 input vectors which have to be tested to ensure complete coverage. Although this is still feasible, this results in 2^{2*n} input vectors in the general case, which has exponential complexity and scales poorly for high bit-widths. Thus, the authors of [28] introduce several decomposition techniques on AIGs to deal with this complexity. For our proposed verification approach, we will



(a) AIG of a miter circuit (b) Subgraph 1 (c) Subgraph 2

Fig. 1: An AIG and its cutwidth decomposition

use two of these decomposition techniques, cutwidth and treewidth decomposition, to address scalability, which are presented in the following.

B. Cutwidth Decomposition

The goal of cutwidth decomposition [29] is to divide the AIG of a circuit into different subgraphs, such that each such subgraph only contains gates relevant to one specific primary output of the circuit. For this, the circuit is ”cut” on intersections in the fan-in cone of primary outputs, hence the name. To achieve this on the AIG level, it is traversed from each primary output, starting with the *Least Significant Bit (LSB)*, toward the primary inputs, adding all nodes on its path to the relevant subgraph. Nodes which were already traversed for previous subgraphs are added as so-called cone nodes, signifying a connection between different subgraphs.

The cutwidth decomposition for the miter AIG of Figure 1(a) is shown in Figure 1(b) and Figure 1(c) respectively. For the two primary outputs, two subgraphs are created, with the primary inputs now split between them. During decomposition, two cone nodes are created, 16 and 44, which are also indicated by triangle shapes in both subgraphs. To solve the CEC problem on the cutwidth decomposition, subgraphs are processed in sequence by evaluating them for each possible input vector. To this end, each subgraph is translated to a SAT instance, adding clauses to constraint variables according to the input vector under consideration. If the value 1 is observed at the primary output for any such input vector, the verification fails and terminates, as the miter condition is violated. Values observed at the cone outputs of a subgraph are stored in a database for efficient manipulation [30], such that they can be used to populate cone inputs of subsequent subgraphs. For our example, evaluation starts with the first subgraph of Figure 1(b), which is evaluated for all $2^2 = 4$ input vectors of primary inputs. The only cone output valuations that can be observed for these input vectors are $\{16 \mapsto 0, 44 \mapsto 0\}$ and $\{16 \mapsto 1, 44 \mapsto 1\}$, which are stored in the database. For the evaluation of the second subgraph of Fiugre 1(c), a total of $4 * 2 = 8$ input vectors need to be evaluated, i.e. the $2^2 = 4$ valuations of primary inputs, combined with the two valuations of cone inputs coming from the database. The verification task passes, requiring a total of $4 + 8 = 12$ input vectors to be evaluated, a decrease of four compared to the naive approach.

This small example clearly shows the potential of cutwidth decomposition to limit the complexity and improve scalability of the verification task. It can be observed that there are two main factors which influence the complexity of this approach: the number of subgraphs and the number of input vectors which need to be evaluated per subgraph. The former simply corresponds to the number of outputs of the miter circuit, which for our ALU model corresponds to the input width n and will be used in the following. The number of input vectors is directly related to the number of cone nodes induced by cutwidth decomposition. In particular, we are interested in the ingoing cutwidth, also referred to as CW in the following, which corresponds to the maximum number of inputs (primary inputs plus cone inputs) in a single subgraph. As in the worst case, all combinations have to be tested, the general complexity of the verification of the cutwidth decomposition is bounded by $\mathcal{O}(n * 2^{CW})$. For a proof of this bound and further information on cutwidth decomposition, the reader is referred to Section 3.1 of [28].

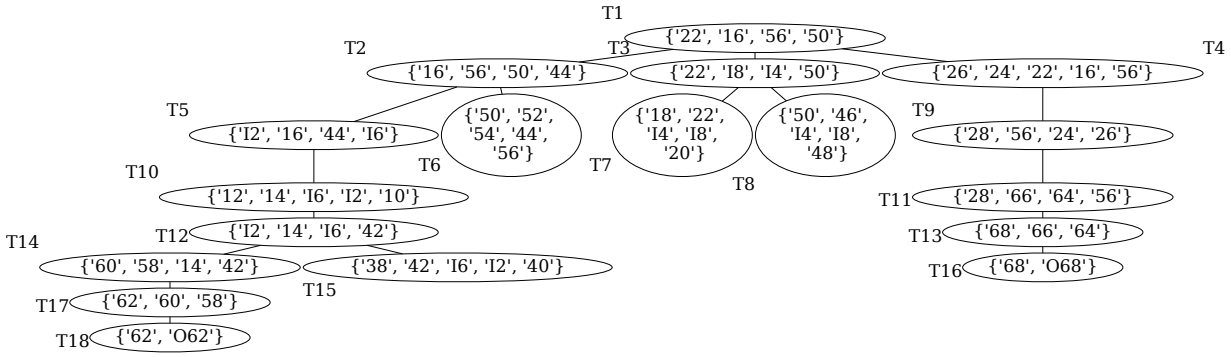


Fig. 2: Treewidth decomposition of the miter

C. Treewidth Decomposition

Treewidth decomposition [31] follows a different approach compared to cutwidth decomposition, transforming the AIG of a circuit to a tree. Intuitively, this is achieved by combining all fully-connected subsets of nodes, i.e., cliques [32], into bags, which constitute nodes in the resulting tree. Edges of the tree are added such that all bags sharing a gate of the original AIG are connected by some path. Unlike cutwidth decomposition, which is computed deterministically, the tree decomposition of an AIG is computed heuristically for lack of an efficient exact algorithm as an NP-complete problem [33].

A treewidth decomposition for the miter AIG of Figure 1(a) is shown in Figure 2. To solve the CEC problem on the treewidth decomposition, bags are first evaluated starting from the leaves ($T18, T15, T6, T7, T8$ and $T16$ in our example) towards the root ($T1$ in our example) of the tree. For each bag, all relevant clauses, i.e., containing only variables present in the given bag, are extracted from the CNF of the original AIG and the resulting instance is solved for all input vectors. Resulting values are once again stored in a database and passed to parent bags for further evaluation. Once the root node is reached, the tree is once again traversed from the root toward all bags containing a primary output ($T18$ and $T16$ in our example). This is to ensure that no child bag contains any database entries which could not be reproduced in the parent bag, which are in turn deleted. Once a bag containing a primary output node is reached, the verification task is reduced to ensuring that the database contains no entry for which the primary output evaluates to 1. If this is the case, the verification passes, else it fails, since the miter condition is violated.

As can be retrieved from the description above, the complexity of the verification of the treewidth decomposition once again depends on two factors: the number of bags and the number of nodes per bag. Each bag is visited at most twice, so their number has a linear impact on the complexity. Generally, the number of bags cannot be related directly to the number of inputs (however, it is bounded by the number of nodes in the AIG) due to the heuristic nature of the decomposition and will thus be referred to by B in the following. The number of nodes per bag once again exponentially influences the number of input vectors to be tested for each bag. The maximum number of nodes in a bag is called the treewidth, also referred to as TW in the following. In total, the complexity of the verification of the treewidth decomposition is bounded by $\mathcal{O}(B * 2^{TW})$. For a proof of this bound and further information on treewidth decomposition, the reader is referred to Section 3.2 of [28].

IV. Verification Approach

A. ALU Model

Listing 1: Example RV32I ALU module definition

```
module ArithmeticLogicUnit (
  input       [31:0]   opA,
  input       [31:0]   opB,
  input       [2:0]    f3,
  input       [6:0]    f7,
  input       [4:0]    shamt,
  input       [6:0]    opcode,
  output reg  [31:0]   out
);
```

Before introducing the verification approach itself, we establish our underlying ALU model to account for any assumptions made in the following. We assume a standard fetch-decode-execute cycle where the ALU is responsible for making calculations whose results are potentially used at a later stage of the CPU to carry out the actual instruction (e.g., address computation for load/store instructions). All instructions in our model are single-cycle, so the ALU abstracts away from a clock signal and other sequential elements. We also focus on the *I* base instruction sets, e.g., RV32I, although the approach can in principle be applied to extensions as well.

Listing 1 shows the module definition of an example ALU implementing the RV32I base instruction set. As can be seen, operands have already been retrieved in the decode step and get passed to the ALU as two operands A and B with XLEN= 32. The output is of equal length, meaning potential overflows of instructions are not considered. The shift amount is logarithmic in XLEN, covering all possible shift values. Finally, the opcode, f3 and f7 fields are used to indicate the instruction result to output. The current RISC-V manual [2] specifies a total of 42 instructions for the RV32I base set, of which 37 are relevant to the ALU (FENCE, FENCE.TSO, PAUSE, ECALL and EBREAK do not require any ALU computation).

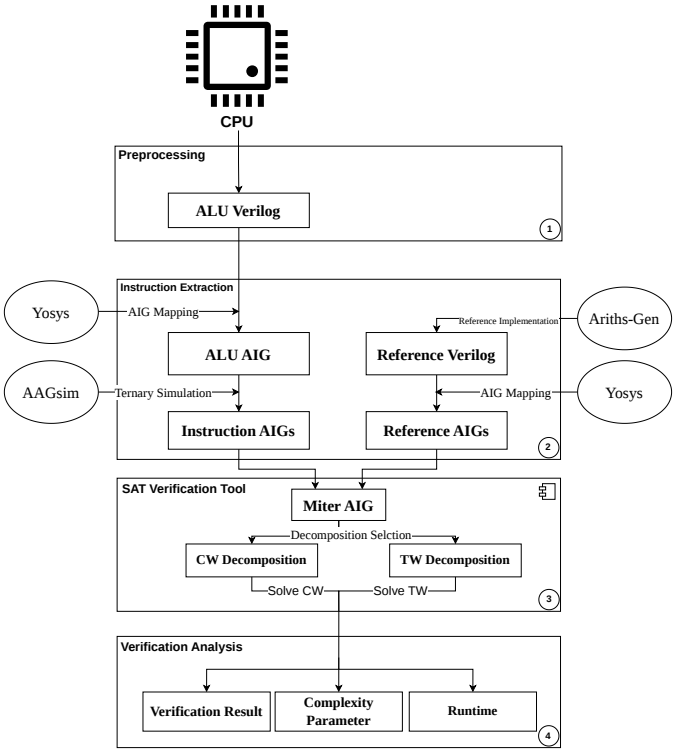


Fig. 3: Overview of the proposed verification approach

B. Overview of our Proposed Approach

An overview of our proposed verification approach is shown in Figure 3. The starting point is a RISC-V CPU implementation containing the ALU to verify, which can be written in any *Hardware Description Language (HDL)* that can be converted into Verilog (e.g., Chisel [34] and SpinalHDL [35] support this). In stage ①, the ALU has to be extracted from the CPU, which is custom to the implementation at hand. Assuming the ALU is defined as its own module, it may be automatically converted as stand-alone or manually extracted. Once the Verilog of the ALU has been retrieved, it is converted to an AIG representation with the help of Yosys [36] in stage ②. The resulting AIG has input nodes for all input ports of the ALU module, the same goes for the outputs. This fact is made use of during ternary simulation, where opcode, f3 and f7 are fixed to the respective values defined in the RISC-V specification to extract all individual instructions. This process is explained in further detail below. To obtain an RC for our extracted instruction AIGs serving as the DUV, we implement each instruction independently in Ariths-Gen [37], allowing for a convenient way to export RCs for different XLEN values. The DUV AIG and the RC AIG for a given instruction are passed to our SAT verification tool in stage ③, where the first step is the creation of a miter AIG. For each instruction, the decomposition type, either cutwidth or treewidth decomposition, is chosen and the respective decomposition and verification technique is carried out. In stage ④, the verification tool provides a variety of information on the verification run, including the result (passed or failed), complexity parameters such as the CW/TW value and the runtime.

C. Instruction Extraction

Listing 2: Stimuli for extraction of the ADD instruction

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx # opA[31:0]
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx # opB[31:0]
000                              # f3[2:0]
0000000                          # f7[6:0]
00000                            # shamt[4:0]
0110011                          # opcode[6:0]
```

The extraction of individual instructions in stage ② is conducted with the help of an internal tool called AAGsim. Its ternary simulation feature allows to simulate an input AIG, which in our case is the AIG of the complete ALU, by setting each of its inputs to one of three values, '0', '1' or 'X'(don't care). The output of this simulation is another AIG, where constants are cleaned. Input stimuli for the simulation are provided as a stimuli file, an example for the ADD instruction can be seen in Listing 2. The A and B operands are set to 'X' to preserve them in the output AIG. According to the RISC-V specification, f3 and f7 fields should be set to all zeroes, while the opcode is set to 011011 for the ADD instruction. The shift amount does not apply to this instruction, so it can be set to an arbitrary value. Ternary simulation of the AIG is carried out by our AAGsim tool in a two-step process. First, the AIG is simulated for the given input stimuli. To this end, the ternary values given in the stimuli file are applied to the inputs and the AIG is traversed toward the outputs, applying one of the three ternary values to each AIG node. In the second step, the AIG is reduced according to the simulation results. All AND nodes of the AIG are visited once and a reduction is attempted, redirecting all predecessors and successors of nodes which collapsed to constant values during simulation or nodes which map to the same successor node as a result of reduction. After that, all input and output nodes that collapsed to constants after simulation are removed from the graph, potentially reducing the number of inputs and outputs of the AIG. Finally, dead branches no longer connected to any output and nodes without any remaining incoming or outgoing edges are removed from the graph, finishing the reduction. Running this ternary simulation for the ALU AIG with the stimuli shown in Listing 2 results in an AIG for the ADD instruction, with exactly $2 * XLEN = 64$ inputs and 32 outputs, which are all driven by operands A and B. This output AIG is then passed to the verification tool alongside the reference AIG for the ADD instruction, to carry out the verification.

D. Verification

Our SAT verification tool in stage ④ realizes all functionality shown in Figure 3 and is implemented in Python [38]. The miter construction, as well as the cutwidth decomposition, are custom implementations, while htd [39] is used for treewidth decomposition. The cutwidth and treewidth verification techniques are implemented in accordance with the descriptions in Sections III-B and III-C respectively. Our tool offers support

Listing 3: Example output of the SAT verification tool

```
[INFO] Number of Sub-graphs : 2
[INFO] Maximum Number of Inputs : 4
[INFO] Maximum Number of Outputs : 3
[INFO] Maximum Number of Gates : 20
[INFO] Outgoing CutWidth : 2
[INFO] Ingoing CutWidth : 4
[INFO] Overall CW Decomposition Time[s] :
    0.001
[INFO] Total Solving Time[s] : 0.051
[INFO] Verification Pass!
[INFO] Total Time[s] : 0.120
```

for multiple SAT solvers, where CaDiCaL [40] is used as the default and MiniSAT [41], as well as CryptoMiniSat [42] are other options. For databases, PostgreSQL [43] is used.

Listing 3 shows an example output for the verification of the miter AIG of Figure 1(a) with cutwidth decomposition. The tool reports several statistics on the decomposition, such as the number of sub-graphs and the CW, allowing for an analysis on the verification complexity. The tool indicates that the verification passes and outputs the total runtime, as well as decomposition time and the actual solving time, giving insight into how the theoretical complexity translate into practical runtimes of our implementation.

V. Experimental Evaluation

A. Experimental Setup

To validate our proposed approach, we apply it to verify the open-source RV32I ALU MicroRV32 [44], implemented in SpinalHDL. While it natively only supports an XLEN of 32, we export 64 and 128 XLEN versions by extending the operands, output, and internal registers. Instructions added by RV64I and RV128I do not affect the ALU itself, e.g., LD requires the same address computation as LW and ADDW computes an addition only on the lower 32 bits, and are thus not considered in the following. Instructions are organized into four groups: 1) Arithmetic instructions like SUB and LW 2) Logic instructions like OR and XOR 3) Comparison instructions like SLT and BGE 4) Shift instructions like SLL and SRA. For each combination of instruction and XLEN, we create a DUV AIG from the ALU and verify it against an RC AIG created with Ariths-Gen. For all these combinations, we consider the correct case where DUV and RC match, as this is generally the more complex case. In the following, we refer to an instruction with two operands of size XLEN as INSTn, e.g., ADD32. For each instruction, we present theoretical complexity bounds and practical runtimes achieved with our tool, as well as peak memory consumption per group. To demonstrate that our verification approach is also suitable for finding errors, we additionally evaluate the incorrect case, i.e., we introduce a bug at a random location in the DUV by inverting one wire value for one representative of each instruction group. The extendability to arbitrary ALU designs is shown preliminarily on the arithmetic instructions of the ALU used in the Ibex [45] RISC-V core, using the same workflow to extract and verify individual instructions. All experiments are run on an AMD Ryzen 7 PRO 5850U processor with 42 GB of main memory.

B. Experimental Results

TABLE I: Experimental Results for Arithmetic Instructions

Instruction	n = 32		n = 64		n = 128	
	CW	RT [s]	CW	RT [s]	CW	RT [s]
ADD	12	1.92	14	5.30	16	28.81
ADDI	12	**1.99**	14	5.30	16	30.36
AUIPC	12	1.91	14	5.52	16	31.52
JAL	12	1.85	14	5.31	16	27.74
JALR	12	1.90	14	5.37	16	33.37
LB	12	1.85	14	5.48	16	29.59
LBU	12	1.89	14	5.29	16	31.62
LH	12	1.89	14	5.44	16	32.04
LHU	12	1.87	14	5.69	16	30.79
LUI	12	1.90	14	5.55	16	29.59
LW	12	1.88	14	5.44	16	29.17
SB	12	1.86	14	5.38	16	28.36
SH	12	1.87	14	**5.71**	16	30.19
SUB	12	1.98	14	5.59	16	**34.27**
SW	12	1.65	14	4.82	16	26.45
Σ		**28.21**		**81.17**		**453.86**

1) Arithmetic Instructions: Table I shows the experimental results for all arithmetic instructions. For all instructions, cutwidth decomposition was chosen, as it was demonstrated to work well on addition and similar operations. The cutwidth behaves logarithmically in n, increasing by 2 for each power of 2. Thus, the overall complexity for the arithmetic instructions can be expressed by $Complexity(Arithmetic) = \mathcal{O}(n * 2^{log(n)}) = \mathcal{O}(n^2)$, equating to a quadratic upper bound. Runtimes are very similar and scale quadratically for all instructions, with a maxima at $1.99s$ for ADDI32, $5.71s$ for SH64 and $34.27s$ for SUB128. Note that for load and store instructions, the target size of the value to load or store has no impact on the ALU operation, as the address to be calculated solely depends on XLEN. Hence, instructions like SB, SH and SW behave similarly for the same XLEN value. Regarding memory consumption, the peak for arithmetic instructions is 4,328 MB (JAL128).

TABLE II: Experimental Results for Logic Instructions

Instruction	n = 32		n = 64		n = 128	
	CW	RT [s]	CW	RT [s]	CW	RT [s]
AND	2	0.22	2	0.37	2	0.68
ANDI	2	0.22	2	0.38	2	0.68
OR	2	0.23	2	0.36	2	0.68
ORI	2	0.23	2	0.37	2	0.68
XOR	2	0.19	2	0.34	2	0.61
XORI	2	0.19	2	0.33	2	0.61
Σ		**1.28**		**2.15**		**3.93**

2) Logic Instructions: Table II shows the experimental results for all logic instructions. Cutwidth decomposition is

chosen once again for all instructions, resulting in completely independent subgraphs, as logic instructions are performed bit-wise. Since the value of each output is only influenced by the corresponding input bits of the two operands, CW is always 2. The complexity of verifying logic instructions can be expressed as $Complexity(Logic) = \mathcal{O}(n * 2^2) = \mathcal{O}(n)$, equating to a linear complexity. The very low cutwidth and linear behavior can also be observed in practical runtimes, where even for $n = 128$, all instructions can be verified in less than $0.7s$, scaling linearly. For logic instructions, the peak memory consumption is 935 MB (XORI128).

TABLE III: Experimental Results for Comparison Instructions

Instruction	n = 32			n = 64			n = 128		
	B	TW	RT [s]	B	TW	RT [s]	B	TW	RT [s]
BEQ	192	**4**	2.54	384	**4**	5.73	768	**4**	14.36
BGE	413	**12**	27.28	810	**14**	108.17	1623	**16**	511.80
BGEU	308	8	7.68	622	9	18.75	1250	9	57.08
BLT	**415**	14	113.40	**819**	16	172.50	**1623**	16	430.95
BLTU	311	9	8.21	624	9	20.75	1248	10	58.98
BNE	192	4	2.68	384	4	6.17	768	4	14.99
SLT	416	12	21.57	813	14	120.15	1606	15	414.20
SLTI	414	15	125.77	812	14	126.51	1619	16	316.40
SLTIU	311	**9**	7.87	621	**8**	20.01	1256	**9**	53.99
SLTU	309	8	7.10	616	9	19.41	1250	9	57.94
Σ			**324.09**			**618.16**			**1930.69**

3) Comparison Instructions: Table III shows the experimental results for all comparison instructions. This time, treewidth decomposition is chosen, as all comparison instructions only have one output, making cutwidth decomposition inapplicable. Since the treewidth decomposition is conducted heuristically, TW values do not always scale consistently, e.g. SLTIU32 has a higher TW than SLTIU64. However, there are still trends and upper bounds which can be observed. For BEQ and BNE, TW values stay constant, while for the remaining instructions, TW behaves at most logarithmically in n (e.g., BGE). Looking at the number of bags in the treewidth decompositions, we can see a linear behavior in n for all instructions, e.g. for BLT the number of bags roughly doubles when doubling n. Thus, the complexity for verifying comparison instructions can be expressed as $Complexity(Comparison) = \mathcal{O}(n * 2^{log(n)}) = \mathcal{O}(n^2)$, which even decreases to $\mathcal{O}(n)$ for BEQ and BNE. Practical runtimes also scale as expected from these bounds and are directly influenced by values of TW and B (e.g., compare BEQ against BGE). Peak memory consumption for the comparison instructions is 4,112 MB (BLT64).

TABLE IV: Experimental Results for Shift Instructions

Instruction	n = 32			n = 64			n = 128		
	SG	CW	RT [s]	SG	CW	RT [s]	SG	CW	RT [s]
SLL	560	1	41.64	2,144	1	91.69	8,384	1	212.73
SLLI	560	1	41.87	2,144	1	91.85	8,384	1	217.90
SRA	560	1	41.87	2,144	1	91.82	8,384	1	215.21
SRAI	560	1	49.60	2,144	1	103.61	8,384	1	222.41
SRL	560	1	41.62	2,144	1	91.30	8,384	1	213.05
SRLI	560	1	41.76	2,144	1	91.36	8,384	1	213.39
Σ			**258.39**			**561.62**			**1294.68**

4) Shift Instructions: Table IV shows the experimental results for all shift instructions. In principle, cutwidth decomposition is suited for shift instructions, but it faces issues with highly connected AIGs, as all outputs are affected evenly by all inputs, depending on the shift amount. Thus, instead of trying to decompose the miter AIGs directly, another ternary simulation is conducted to extract one AIG for each possible shift amount value. For a shift instruction with input width n, there are n possible shift amount values, resulting in a quadratic number of subgraphs per instruction. As a trade-off, each subgraph is a simple connection from input to output, resulting in a constant CW of 1. In total, the complexity of verifying shift instructions can be expressed as $\mathcal{O}(n^2 * 2^1) = \mathcal{O}(n^2)$, equating to a quadratic complexity. Practical runtimes also scale less than quadratically, behaving very similarly for each shift instruction. Here, peak memory consumption scales up to 710 MB (SRA128).

TABLE V: Overall Experimental Results for the Correct Case

Instruction Group	n = 32	n = 64	n = 128
	RT [s]	RT [s]	RT [s]
Arithmetic	28.21	81.17	453.86
Logic	1.28	2.15	3.93
Comparison	324.09	618.16	1930.69
Shift	258.39	561.62	1294.68
Σ	**611.97**	**1263.10**	**3683.16**

5) Overall Results: The overall experimental results for the correct case are presented in Table V. Between instruction groups, there are quite stark differences. Logic instructions can be verified very fast, due to their bit-wise behavior. Arithmetic instructions also work well, agreeing with previous observations for cutwidth decomposition. Comparison and shift operations, which were not well-studied before, turned out to require some special treatment, but can still be handled by our proposed verification approach. For all instructions of the ALU, linear or quadratic time complexity bounds could be established, resulting overall in $Complexity(ALU) = \mathcal{O}(n^2)$. These theoretical bounds were confirmed to translate into practical scalability, as the verification time for all instructions remains below ten minutes, with runtimes not exploding even when testing for 128 bit-width operands.

TABLE VI: Experimental Results for the Incorrect Case

Instruction	n = 32			n = 64			n = 128		
	B/SG	CW/TW	RT [s]	B/SG	CW/TW	RT [s]	B/SG	CW/TW	RT [s]
ADD	-	12	0.46	-	14	0.20	-	16	2.44
AND	-	2	0.14	-	2	0.25	-	2	0.27
BEQ	192	4	2.41	384	4	4.83	771	4	13.74
SLL	560	1	37.66	2,144	1	55.91	8,384	1	89.20
Σ			**40.67**			**61.19**			**105.65**

6) Incorrect Case: Table VI shows the experimental results for the incorrect cases. We choose the first instruction of a given group as its representative. For each value of n, a bug is introduced in the DUV, and our verification approach is run until the error is detected. In the first column, we report the respective value of B for the BEQ instruction and the respective number of subgraphs for the SLL instruction. The second column represents the respective CW or TW value

based on the considered instruction. Results for the ADD and AND instructions conducted with cutwidth decomposition show that CW values remain the same as for the correct case, as decomposition remains unaffected by the inversion of a wire. Runtimes across all values of n are decreased compared to the correct case, since our verification can simply terminate without evaluating the remaining subgraphs once the error is detected. For example, instruction ADD64 takes 6.3s to verify in the correct case, whereas a bug is detected in only 0.2s in the incorrect case. As can be seen, depending on the location of the bug, runtimes for higher values of n may even be lower than for previous sizes. For the BEQ instruction, the TW values once again remain the same as for the correct case. However, runtimes remain very similar, since the evaluation of bags still has to be carried out completely, and verification can only terminate during the evaluation of outputs. For the SLL instruction, a similar trend as for the other instructions verified by cutwidth decomposition can be observed, with greatly reduced runtimes, especially for higher values of n. Peak memory consumptions for the incorrect cases amount to 3,313 MB for ADD128 (down from 3,421 MB), 933 MB for AND128, 4,039 MB for BEQ128, and 710 MB for SLL128 (all same as correct case). In addition to runtimes and memory consumption, it should be noted that our verification approach is also suited at locating the bug in the DUV, as for cutwidth decomposition, the input vector under which the error was encountered is reported by our tool, and for treewidth decomposition, the output for which verification fails is reported.

TABLE VII: Experimental Results for Arithmetic Instructions of Ibex RISC-V ALU

Instruction	**n = 32**		**n = 64**		**n = 128**	
	CW	**RT [s]**	**CW**	**RT [s]**	**CW**	**RT [s]**
ADD	5	0.51	5	1.01	5	1.99
ADDI	5	0.53	5	1.01	5	2.02
AUIPC	5	0.52	5	1.02	5	2.02
JAL	5	0.54	5	1.01	5	2.04
JALR	5	0.52	5	1.03	5	2.01
LB	5	0.54	5	1.01	5	2.02
LBU	5	0.53	5	1.04	5	2.03
LH	5	0.52	5	1.03	5	2.04
LHU	5	0.52	5	1.02	5	2.02
LUI	5	0.52	5	1.03	5	2.05
LW	5	0.55	5	1.02	5	2.04
SB	5	0.52	5	1.04	5	2.05
SH	5	0.52	5	1.01	5	2.03
SUB	5	0.52	5	1.02	5	2.03
SW	5	0.52	5	1.03	5	2.03
Σ		**7.88**		**15.33**		**30.42**

7) Application to other ALUs: The results for arithmetic instructions implemented in the ALU used in the Ibex RISC-V core are shown in Table VII. Interestingly, the cutwidth stays constant, at 5 across all instructions and XLEN values, resulting in a linear complexity in n. It can be seen that, just like for the arithmetic instructions of MicroRV32 displayed in Table I, the practical runtimes are very similar across instructions. The total runtime of all arithmetic instructions of the Ibex ALU doubles for each doubling in XLEN, mirroring the linear scalability we expect from the cutwidth. Peak memory consumption is 2,533 MB for ADD128. We note that, while the complexity bounds for the arithmetic instructions in the Ibex RISC-V core, just like for MicroRV32, scale polynomially, this does not imply that this behavior must hold for the remaining instructions, or general RISC-V ALU implementations, pointing at a potential limitation of our proposed approach that would need to be evaluated further.

VI. Conclusion

In this work, we introduced a novel approach to formally verify arbitrary RISC-V ALU designs, relying on ternary simulation to extract individual instructions and decomposition techniques to split up the problem, enabling us to determine provable complexity bounds. For an open-source ALU, we were able to establish quadratic complexity bounds on the whole verification process. We also showed that these theoretically proven bounds translate into practice, enabling scalable formal verification of future-proof systems, with efficient runtimes for RISC-V instructions with 32, 64 and 128-bit operands.

Future work may focus on comparing different open-source RISC-V ALU implementations comprehensively for their verifiability and complexity using our approach. As an extension to our approach, other RISC-V extensions may be targeted to consider a wider range of instructions.

Acknowledgements

This work was supported by the German Research Foundation (DFG) within the Reinhart Koselleck Project *PolyVer* (DR 287/36-1) and by the German Ministry for Research, Technology and Space (BMFTR) with project *ExaVerse* (grant number 01IW25003).

References

[1] C. S. Deshpande, A. Perais, and F. Pétrot, "Toward practical 128-bit general purpose microarchitectures," *IEEE Computer Architecture Letters*, vol. 22, no. 2, pp. 81–84, 2023.

[2] RISC-V International, *The RISC-V Instruction Set Manual Volume I: Unprivileged Architecture*, version 20250508 ed., 2025.

[3] W. K. Lam, *Hardware Design Verification: Simulation and Formal Method-Based Approaches*. USA: Prentice Hall PTR, 1st ed., 2008.

[4] R. Drechsler, *Advanced formal verification*. Boston: Kluwer Academic Publishers, 2004.

[5] R. Drechsler, "Fast and exact is doable: Polynomial algorithms in test and verification," in *2022 IEEE 23rd Latin American Test Symposium (LATS)*, (Montevideo, Uruguay), p. 1–2, IEEE, 2022.

[6] R. Drechsler, "PolyAdd: Polynomial formal verification of adder circuits," in *2021 24th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, (Vienna, Austria), p. 99–104, IEEE, Apr. 2021.

[7] C. Dominik and R. Drechsler, "Polynomial formal verification of sequential circuits," in *2024 Design, Automation & Test in Europe Conference (DATE)*, (Valencia, Spain), p. 1–6, IEEE, Mar. 2024.

[8] L. Müller, M. Nadeem, and R. Drechsler, "Cutwidth decomposition on Circuit-AIGs: Taming verification complexity of arithmetic circuits," in *2026 27th International Symposium on Quality Electronic Design (ISQED)*, IEEE, 2026.

[9] R. Drechsler, "Polynomial circuit verification using BDDs," in *2021 5th International Conference on Electrical, Electronics, Communication, Computer Technologies and Optimization Techniques (ICEECCOT)*, (Mysuru, India), p. 49–52, IEEE, Dec. 2021.

[10] L. Müller and R. Drechsler, "Polynomial formal verification parameterized by cutwidth properties of a circuit using Boolean satisfiability," *Microprocessors and Microsystems*, vol. 118, p. 105199, Nov. 2025.

[11] M. Nadeem, J. Kleinekathofer, and R. Drechsler, "Polynomial formal verification exploiting constant cutwidth," *34th International Workshop on Rapid System Prototyping (RSP)*, 2023.

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

[13] L. Weingarten, K. Datta, A. Kole, and R. Drechsler, "Complete and efficient verification for a RISC-V processor using formal verification," in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1–6, IEEE, 2024.

[14] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Commun. ACM*, vol. 62, p. 48–60, Jan. 2019.

[15] RISC-V International, *The RISC-V Instruction Set Manual Volume I: Unprivileged Architecture*, version 20240411 ed., 2024.

[16] A. Biere, M. Heule, H. van Maaren, and T. Walsh, eds., *Handbook of satisfiability*. Frontiers in Artificial Intelligence and Applications, Amsterdam; Washington, DC: IOS Press, second edition ed., 2021.

[17] S. D. Prestwich, "CNF encodings," in *Handbook of Satisfiability*, 2021.

[18] O. Beyersdorff, A. Biere, V. Ganesh, J. Nordström, and A. Oertel, "Theory and practice of SAT and combinatorial solving (dagstuhl seminar 22411)," *Dagstuhl Reports*, vol. 12, pp. 84–105, 2022.

[19] A. Kuehlmann, V. Paruthi, F. Krohm, and M. Ganai, "Robust Boolean reasoning for equivalence checking and functional property verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, no. 12, pp. 1377–1394, 2002.

[20] R. Brinkmann and D. Kelf, "Formal Verification—The Industrial Perspective," in *Formal System Verification: State-of the-Art and Future Trends* (R. Drechsler, ed.), p. 155–182, Cham: Springer International Publishing, 2018.

[21] P. DasGupta, *A Roadmap for Formal Property Verification*, p. 217–241. Dordrecht: Springer Netherlands, 2006.

[22] E. Goldberg, M. R. Prasad, and R. K. Brayton, "Using SAT for combinational equivalence checking," *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*, pp. 114–121, 2001.

[23] G. S. Tseitin, *On the Complexity of Derivation in Propositional Calculus*, p. 466–483. Berlin, Heidelberg: Springer Berlin Heidelberg, 1983.

[24] C. Tain, S. Patil, and H. Al-Asaad, "Survey of verification of RISC-V processors," *Journal of Electronic Testing*, vol. 41, p. 111–138, May 2025.

[25] P. D. Schiavone, E. Sánchez, A. Ruospo, F. Minervini, F. Zaruba, G. Haugou, and L. Benini, "An open-source verification framework for open-source cores: A RISC-V case study," in *2018 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, pp. 43–48, IEEE, 2018.

[26] C. Wolf, "RISC-V formal verification framework." https://github.com/SymbioticEDA/riscv-formal. accessed:25/10/29.

[27] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting: a fresh look at combinational logic synthesis," in *2006 43rd ACM/IEEE Design Automation Conference*, (San Francisco, CA, USA), pp. 532–535, 2006.

[28] M. Nadeem, L. Müller, C. K. Jha, and R. Drechsler, "Advanced and-inverter graph decomposition technique for reducing circuit complexity," *ACM Trans. Des. Autom. Electron. Syst.*, Oct. 2025.

[29] D. M. Thilikos, M. Serna, and H. L. Bodlaender, "Cutwidth I: A linear time fixed parameter algorithm," *Journal of Algorithms*, vol. 56, no. 1, pp. 1–24, 2005.

[30] P. Revesz, "Introduction to databases," *Texts in Computer Science*, 2010.

[31] N. Robertson and P. Seymour, "Graph minors. II. Algorithmic aspects of tree-width," *Journal of Algorithms*, vol. 7, no. 3, pp. 309–322, 1986.

[32] J. W. Moon and L. Moser, "On cliques in graphs," *Israel journal of Mathematics*, vol. 3, no. 1, pp. 23–28, 1965.

[33] H. L. Bodlaender and A. M. Koster, "Treewidth computations I. Upper bounds," *Information and Computation*, vol. 208, no. 3, pp. 259–275, 2010.

[34] L. LF Projects, "RISC-V formal verification framework." https://www.chisel-lang.org/. accessed:25/10/30.

[35] C. Papon, "Spinalhdl." https://spinalhdl.github.io/. accessed:25/10/30.

[36] C. Wolf, J. Glaser, and J. Kepler, "Yosys-a free Verilog synthesis suite," in *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, vol. 97, 2013.

[37] J. Klhufek and V. Mrazek, "ArithsGen: Arithmetic circuit generator for hardware accelerators," in *2022 25th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, pp. 44–47, 2022.

[38] G. Van Rossum and F. L. Drake, *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009.

[39] M. Abseher, N. Musliu, and S. Woltran, *htd – A Free, Open-Source Framework for (Customized) Tree Decompositions and Beyond*, vol. 10335 of *Lecture Notes in Computer Science*, p. 376–386. Springer International Publishing, 2017.

[40] A. Biere, T. Faller, K. Fazekas, M. Fleury, N. Froleyks, and F. Pollitt, *CaDiCaL 2.0*, vol. 14681 of *Lecture Notes in Computer Science*, p. 133–152. Cham: Springer Nature Switzerland, 2024.

[41] N. Eén and N. Sörensson, *An Extensible SAT-solver*, vol. 2919 of *Lecture Notes in Computer Science*, p. 502–518. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004.

[42] M. Soos, K. Nohl, and C. Castelluccia, "Extending SAT solvers to cryptographic problems," in *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference* (O. Kullmann, ed.), vol. 5584 of *Lecture Notes in Computer Science*, (Swansea, UK), pp. 244–257, Springer, 2009.

[43] T. P. G. D. Group, "PostgreSQL." https://www.postgresql.org/. accessed:25/10/31.

[44] 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.

[45] lowRISC, "Ibex RISC-V core." https://github.com/lowRISC/ibex. accessed:26/07/05.