ChiselFV: A Formal Verification Framework for Chisel

Mufan Xiang[†], Yongjian Li[‡*] , Yongxin Zhao[†‡*]
[†] Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai, China
[‡] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China

***Abstract*—Modern digital hardware is becoming ever more complex. And agile development, an efficient idea in software development, has been introduced into hardware. Furthermore, as a new hardware construction language, Chisel helps to raise the level of hardware design abstraction with the support of object-oriented and functional programming. Chisel plays a crucial role in future hardware design and open-source hardware development. However, the formal verification for Chisel is still limited. In this paper, we propose ChiselFV, a formal verification framework that has supported detailed formal hardware property descriptions and integrated mature formal hardware verification flows based on SymbiYosys. It builds on top of Chisel and uses Scala to drive the verification process. Thus the framework can be seen as an extension of Chisel. ChiselFV makes it easy to verify hardware designs formally when implementing them in Chisel.**

***Index Terms*—Hardware verification, Formal methods, Hardware description language, Chisel**

I. Introduction

Over the past several years, hardware design has grown to be ever more and more complex. The increased demand for high-performance computing systems has led to a larger need for domain-specific hardware accelerators [1]. The dominant traditional hardware description languages (HDLs), Verilog and VHDL, were initially developed as hardware simulation languages and were only later adopted as a basis for hardware synthesis. So these languages lack the powerful abstraction facilities common in modern software languages, leading to low productivity because it is difficult to reuse components. Chisel [2], a Scala-embedded hardware construction language, was introduced to lift digital circuit description to a more software-like high-level language [3]. Chisel attempts to solve these problems by providing functional and object-oriented programming support. With these features, Chisel supports advanced hardware design using highly parameterized generators and improves the abstraction level of hardware design. However, Chisel lacks the powerful and easy-to-use formal verification support that is mature in traditional HDLs.

In hardware development, because of the high cost of trial and error, verifying the correctness of the design is essential, especially for trusted systems dealing with human lives, or it may cause a massive loss if an unexpected event occurs. Therefore, we need to use many test cases to simulate the design to find bugs and use formal methods to ensure correctness. There are many mature verification methods in traditional hardware languages. However, in Chisel, formal verification work is still limited, and the previous workflow for low-level RTL cannot be migrated to Chisel easily.

Currently, there is some work on the Chisel level to verify the correctness of the design. ChiselVerify is a verification framework inspired by Universal Verification Method(UVM) and implemented on the Chisel level and supports both coverage-oriented and constrained random verification (CRV) flows [3]. However, ChiselVerify only focuses on applying testing techniques to hardware verification. Recently, Berkeley provided Chisel developers with an easy way to formally verify the designs [4]. They added a formal backend to the FIRRTL compiler, which converts a high-level intermediate representation (IR) into a normalized structural representation. The backend can translate FIRRTL expressions to the bit-vector expression language defined by the SMTLib format [5], and then use the Z3 solver [6] to execute the Bounded Model Checking (BMC) algorithm [7]. However, the current support for property description is minimal. It only supports assertions on expressions that return Boolean values, and the verification algorithm it supports is only BMC. Compared with the existing formal support for other HDLs, they are very primitive.

Traditional HDLs have mature hardware verification techniques but cannot be directly migrated to Chisel. For example, SystemVerilog Assertions (SVA) [8] can be used to describe properties. SVA is a language construct that provides a powerful way to write constraints, checkers, and cover points for hardware designs. It supports complex temporal property descriptions. It can be synthesized with SystemVerilog code by Yosys [9] and converted to a transition system, which can be solved by various model checking algorithms based on SAT/SMT solvers. Currently, Chisel is mainly used to construct hardware generators at the high level and then generate SystemVerilog code for synthesis. It is not feasible to define property descriptions in SystemVerilog because the generated SystemVerilog code is difficult to read, making it hard to find the corresponding signals in Chisel when the design is complex. In addition, Chisel's advantage is high parameterization, but in the generated SystemVerilog, the module instantiation has been completed, so the property needs to be redefined every time the

*Corresponding authors

This work is supported by National Key Research and Development Program (2020AAA0107800), National Natural Science Foundation of China (NSFC 62272165), the "Digital Silk Road" Shanghai International Joint Lab of Trustworthy Intelligent Software (Grant No. 22510750100), Shanghai Trusted Industry Internet Software Collaborative Innovation Center, and the Strategic Priority Research Program of the Chinese Academy of Sciences, Grant No. XDA0320000 and XDA0320300.

978-3-9819263-7-8/DATE23/© 2023 EDAA

module is instantiated, which is not desirable.

Therefore, we choose to define properties on Chisel and strive to make Chisel have the same formal support as traditional HDLs. Our main contribution is to propose ChiselFV, which can be found in the GitHub repository [10]. We also provide the workflow of formal verification on the Chisel level and detailed application of ChiselFV on the actual case. The contribution of this work is as follows:

- **ChiselFV.** We propose a formal verification framework for Chisel, which can be used to describe hardware properties on the Chisel level. And we integrated the verification algorithm engine to perform verification in one click. At the same time, our ChiselFV can describe the temporal properties and free constant definition, which greatly enhances the hardware properties description capability.
- **Verification flow on Chisel level.** We propose a hardware verification workflow based on ChiselFV. With ChiselFV, we can synchronize the hardware properties description with the agile development process, thereby avoiding the design of complicated hardware properties in low-level hardware. The hardware verification flow can seamlessly integrate into the agile hardware development process. We use the formal verification process of a multi-ported memory and a textbook five-stage pipeline design to illustrate this point. And a design problem was found while verifying the processor.

This paper is organized into four sections. Section II details the design and supported verification ability of ChiselFV. Section III describes the formal verification process of a multi-ported memory and a five-stage pipeline processor based on ChiselFV. Related work is in Section IV. Section V concludes.

II. Formal Verification in Chisel

This section gives the structure of ChiselFV, details the property description and verification engine support in ChiselFV, and the methodology to verify with ChiselFV.

A. Framework Structure

ChiselFV is based on Chisel and can be seen as an extension of formal verification in Chisel. It is mainly implemented using Scala and provides the ability to describe Chisel module properties and automate formal verification. The figure Fig 1 shows the structure of ChiselFV.

As shown in Fig 1, ChiselFV is mainly an extension of Chisel's SystemVerilog generation and adds support for formal verification syntax. It also integrates the verification frontend and backend based on SymbiYosys, enabling one-key verification. The SystemVerilog generation is performed by calling the generation process in Chisel. For property descriptions, ChiselFV compiles the properties defined in Chisel into SVA code. Some auxiliary circuits will be generated to complete the property definition for complex property descriptions. And the configuration generator generates SymbiYosys-supported verification tasks configuration file .sby.

Hardware developers can construct hardware modules in Chisel, and Chisel's compiler can compile Chisel modules to SystemVerilog. The starting point of our work is to inject support for formal property descriptions in this stage. Besides the SystemVerilog generation, we also add support for generating SystemVerilog Assertions and verification configuration files.

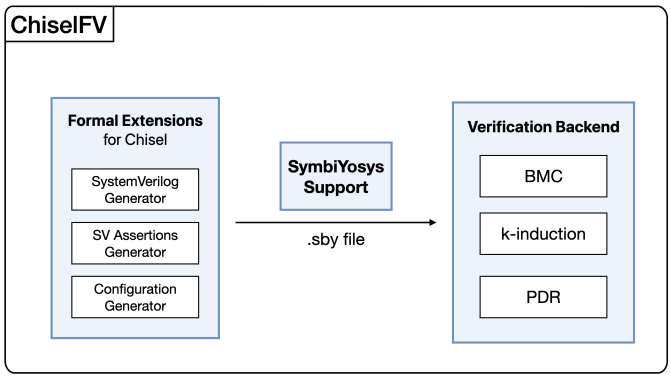

Figure 1: ChiselFV Structure

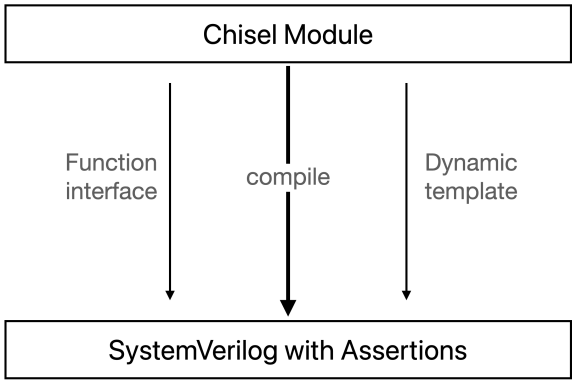

Figure 2: Technique in Chisel Compilation Extensions

As shown in Fig 2, we mainly use two techniques to extend the Chisel compilation process. The first is to use the Chisel hardware description abilities to implement the hardware code fragments equivalent to the property descriptions and encapsulate them into functions, providing an interface to call. This way, we implement a part of the formal property description syntax support. Such as to achieve the support for the temporal property description introduced in the following section, we mainly try to encapsulate the `ShiftRegister` instance provided by Chisel into a function and provide an interface to call.

Using function encapsulation and providing interfaces, there is a specific limitation, which needs to ensure that Chisel's hardware description abilities can implement the property description, and the second technique, dynamic template, is more flexible. Dynamic template mainly depends on the `BlackBox` class provided by Chisel. We write SystemVerilog fragments and encapsulate them into a new Chisel module by inheriting `BlackBox`, which will be injected into the Chisel module during compilation. At the same time, we can pass external parameters

to the code fragment, and the template will automatically apply them to the instance dynamically during compilation.

Finally, we provide the automatic generation of verification configuration files. ChiselFV can generate the corresponding verification configuration file according to the verification algorithm and parameters specified by the user. Here we use the .sby format, which is supported by SymbiYosys [11]. ChiselFV can also automatically call the backend to verify according to the specific verification algorithm and parameters.

B. Formal Description Syntax

We define the property description syntax in Fig 3. The primary verification task in ChiselFV is: when the assumption defined in `assume` block is true and whether the circuit assertion defined in `assert` block is true or not.

$$
\begin{aligned}
\langle \text{ChiselData} \rangle &::= \text{circuit node in Chisel} \\
\langle \text{num} \rangle &::= \text{nonnegative integer} \\
\langle \text{anyConst} \rangle &::= \text{circuit node with any constant value} \\
&\quad \text{defined by Chisel extension support} \\
\langle \text{valueExpr} \rangle &::= \text{past}(\langle \text{valueExpr} \rangle, \langle \text{num} \rangle) | \langle \text{ChiselData} \rangle \\
&\quad | \langle \text{anyConst} \rangle \\
\langle \text{comparisonOp} \rangle &::= \text{`}\neq\text{'} | \text{`}=\text{'} | \text{`}<\text{'} | \text{`}\leq\text{'} | \text{`}>\text{'} | \text{`}\geq\text{'} \\
\langle \text{logicalOp} \rangle &::= \text{`\&\&'} | \text{`||'} \\
\langle \text{booleanExpr} \rangle &::= \langle \text{valueExpr} \rangle \langle \text{comparisonOp} \rangle \langle \text{valueExpr} \rangle \\
&\quad | \langle \text{booleanExpr} \rangle \langle \text{logicalOp} \rangle \langle \text{booleanExpr} \rangle \\
&\quad | \text{`!'} \langle \text{booleanExpr} \rangle \\
&\quad | \text{`true.B'} | \text{`false.B'} \\
\langle \text{assumption} \rangle &::= \text{assume}(\langle \text{booleanExpr} \rangle) \\
\langle \text{assertion} \rangle &::= \text{assert}(\langle \text{booleanExpr} \rangle) \\
&\quad | \text{assertNextStepWhen}(\\
&\quad \langle \text{booleanExpr} \rangle, \langle \text{booleanExpr} \rangle) \\
&\quad | \text{assertAfterNStepWhen}(\\
&\quad \langle \text{booleanExpr} \rangle, \langle \text{num} \rangle, \langle \text{booleanExpr} \rangle) \\
&\quad | \text{assertAlwaysAfterNStep}(\\
&\quad \langle \text{booleanExpr} \rangle, \langle \text{num} \rangle, \langle \text{booleanExpr} \rangle)
\end{aligned}
$$

Figure 3: Syntax of Property Descriptions

For the property definition, the basic syntax is inherited from Chisel. It means that the data nodes with return type `Bool` in Chisel can be defined as property descriptions (defined as ⟨booleanExpr⟩ here).

In addition, we mainly enhance the description support for temporal properties and free constants. The following gives a detailed introduction.

1) Immediate Assertions: ChiselFV supports the basic immediate assertion types. In ChiselFV, we can use the syntax `assume(expr)` to define an assumption that must always be true, and use the syntax `assert(expr)` to provide an assertion. The model checker will try to find a counter-example to the assertion or prove its correctness. Note that the search space is the circuit state after the first reset, i.e., without considering the Chaos before the first reset.

2) Temporal Assertions: ChiselFV provides rich support for temporal property descriptions.

a. `past(expr, num)`:
   It can be used to get the value of a signal at time `<num>` before the current time.
b. `assertAfterNStepWhen(cond, num, expr)`:
   It can be used to assume that when the `cond` is true, the `expr` signal is true in `num` steps. In SVA syntax, it is similar to the statement `cond -> ##num expr`. Additionally, `assertNextStepWhen(cond, expr)` is equivalent to `assertAfterNStepWhen(cond, 1, expr)`.
c. `assertAlwaysAfterNStep(cond, num, expr)`:
   It can be used to assume that when the `cond` is true, the `expr` is always true after `num` steps. In SVA syntax, it is similar to the statement `cond -> ##[num:] expr`.

3) Universal Quantification: In ChiselFV, we provide support for universal quantification. Although we describe a fixed-width number in hardware circuits, this still greatly improves the expressiveness of property description. In the implementation, we treat it as a node equivalent to an input node and constrain the same value at every clock cycle. In ChiselFV, we can call `anyconst(w)` to get any value of fixed-width `w`.

C. Verification Engines

In ChiselFV, we select three mainstream algorithms as backend engines: BMC [12], k-induction [13] and PDR [14] algorithm. They are all typical model checking algorithms that rely on SAT/SMT solvers. After the front-end of ChiselFV, the hardware design and property definition are transformed into a transition system, and the property is to be verified. The BMC algorithm mainly verifies whether the model is safe within k steps. It can only try to find whether the model is wrong within some steps but cannot prove the correctness of the model. The k-induction algorithm tries to confirm that the property to be verified is an invariant of k steps. The PDR algorithm is to enhance the property step by step, trying to get an invariant of one step. The latter two may provide evidence of system correctness and can find errors. These three algorithms have their advantages and disadvantages. The BMC algorithm and k-induction are more intuitive and straightforward, but they cannot solve all problems. The PDR algorithm is more complex and can solve more problems as a supplement.

```
1  Check.kInduction(() => new Memory, 20)
```

Listing 1: A Code Clip to Call Formal Verification

In ChiselFV, we can call the verification engines with a simple line of code. As shown in Listing 1, this code can verify the property defined in the Memory module using the k-induction algorithm for 20 steps.

At the same time, the front-end and back-end engines in ChiselFV are separated, which makes the back-end have good extensibility. ChiselFV supports replacing different engines and replacing the SAT/SMT solvers used.

D. Methodology

Next, we discuss the workflow of designing and formal verifying a circuit in Chisel with the help of ChiselFV.

The first step of this workflow is to clarify the design requirements of the circuit, that is, to analyze what functions the circuit needs to implement, and then it can be used as our verification requirements. At the same time, we need to design the structure of the circuit.

Next, we use Chisel to implement the hardware module and the property description. This is the most challenging step in the process. This paper does not discuss using Chisel to implement hardware modules because this is not our primary research focus. However, formalizing the property description from natural language is still difficult. On the one hand, it depends on the experience of formal verification engineers, and on the other hand, there are also some fixed patterns and tricks. Our ChiselFV provides sufficient formal property description capabilities, which will be more intuitive to show in the next section, Case Studies.

After describing properties in ChiselFV, we can directly call the verification function to verify and get the result. The result may be a failure, which means that the property is violated in the circuit, and we get a counter-example; it may be a success, which means that the property will always be true; it may be unknown, which means that the current algorithm engine cannot solve the property. We can consider changing the algorithm engine or modifying the property expression.

III. Case Studies

This section uses the ChiselFV framework to make formal verification for multi-ported memory and a typical five-stage pipeline design in textbooks. Due to the length of the paper, more cases are placed in the GitHub repository [10].

A. Multi-ported Memory

Multi-ported Memories are essential for high-performance parallel computation systems. VLIW and vector processors, and other processing systems often rely upon multi-ported memories for parallel access, hence higher performance [15].

However, implementing multi-ported Memory is quite expensive, so FPGA manufacturers often only provide dual-ported block memory, and hardware engineers need to use the existing dual-ported block memory to design multi-ported Memory. Based on the previous work [16], we implement three high-parameterized multi-ported memories in Chisel, whose code can be found in GitHub repositories [17].

In this paper, our focus is not on the design itself, so we abstract the multi-ported Memory. We are only concerned about its input and output. The basic IO of a multi-ported memory with m write ports and n read ports is shown in Fig 4.

At the same time, we need to design the verification requirements, that is, what we need the multi-ported Memory to do. As shown in Fig 5, the property can be described as: at any time, any write port i can write $data_1$ to any address $addr$, and then after any time t, if there is no write operation to the same address during this period, $data_2$ read from the address should be the same as $data_1$.

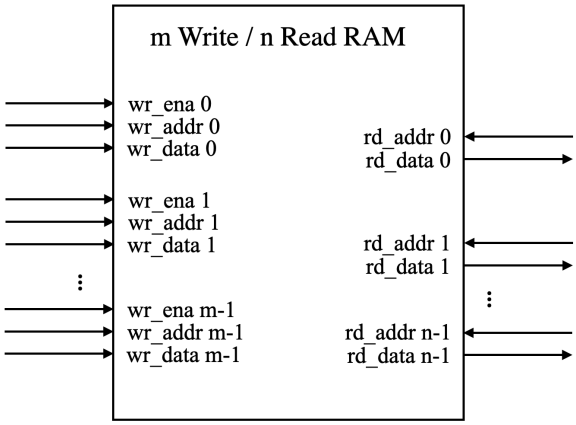

Figure 4: M write ports / N read ports Memory

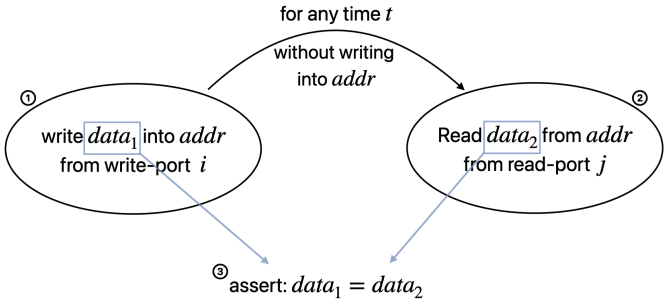

Figure 5: The property we need to verify in multi-ported memory

Next, we need to implement the module in Chisel and define the property. The Chisel implementation is in the repository [17], and the property definition can be found in the case folder of the ChiselFV project [10]. Here, we need to use the `anyconst` keyword to construct an arbitrary value of $addr$ and a register $data$ to store the value written to mem_{addr}. Whenever any write port writes to the $addr$ address, the written value is recorded in the register $data$, and whenever any read port reads from the $addr$ address, the read result is asserted to be the same as the value in $data$. The simplified code for the property definition is shown in Listing 2. It is necessary to add the assumption that different ports do not write to $addr$ simultaneously. The k-induction algorithm, within five steps, quickly proves the model and property, and the detailed code is in the repository.

```
1 val addr = anyconst(addrW)
2 val data = Reg(UInt(w.W))
3 for (i <- 0 until m) {
4     when(io.wrAddr(i) === addr && io.wrEna(i) =
          == true.B) {
5         data := io.wrData(i)
6     }
7 }
```

```
 8  for (i <- 0 until n) {
 9      when(io.rdAddr(i) === addr && hasWritten) {
10          assert(io.rdData(i) === data)
11      }
12  }
```

Listing 2: Verification Code of Multi-ported Memory Module

B. Five-stage Pipeline

In the classic architecture textbooks [18], a simple five-stage pipeline is given. This processor design has five pipeline stages: instruction fetch and decode, execution, memory access, and write back. It implements four typical instructions in the RISC-V instruction set, including `ld`, `sd`, `add`, and `beq`. It avoids data hazards by forwarding and stalling. On branch prediction, it adopts the way of assuming that the branch will not be taken to handle control hazards. If the prediction is wrong, a `nop` instruction will be inserted in the middle. The book uses the Verilog language to implement the processor. In our work, we first use Chisel to implement the processor and then use the ChiselFV framework to verify it formally. The implementation and verification code are in the GitHub repository [19].

Our verification solution is mainly inspired by the RISC-V Formal Verification Framework [20]. They provide a framework for verifying RISC-V processors at the SystemVerilog level, using SVA to define properties and then using verification tools to verify. However, for the reasons mentioned in Section I, that is, the poor readability of the SystemVerilog code generated by Chisel, we try to migrate the RISC-V Formal verification framework to the Chisel level by ChiselFV.

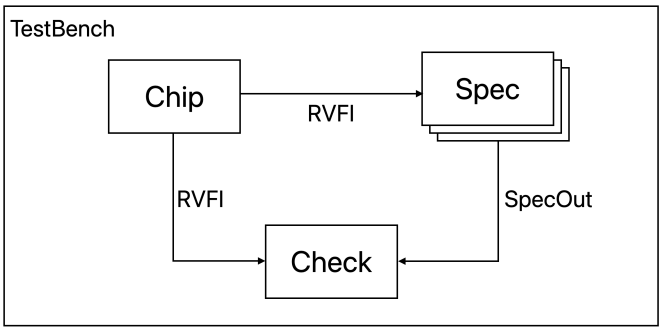

Figure 6: Chisel Version RISC-V Formal Verification Framework

The verification solution is shown in Figure 6. The pipeline processor is implemented in the Chip module, and the Chip needs to provide the RISC-V Formal Interface (RVFI) to output the critical information of each instruction execution process.

The RVFI interface definition is shown in Listing 3. For each instruction, we get its instruction content, decoding result, PC register update before and after execution, source operands, destination operands, and register state before instruction execution. Extracting RVFI from Chip is a crucial step in the verification implementation. We need to use the temporal ability provided by ChiselFV to get the information of each instruction execution process and output it simultaneously when the execution is completed.

```
 1  class RVFI_IO extends Bundle {
 2      val valid = Output(Bool())
 3      val insn = Output(UInt(32.W))
 4      val pc_rdata = Output(UInt(64.W))
 5      val pc_wdata = Output(UInt(64.W))
 6      val rs1_addr = Output(UInt(5.W))
 7      val rs2_addr = Output(UInt(5.W))
 8      val rs1_rdata = Output(UInt(64.W))
 9      val rs2_rdata = Output(UInt(64.W))
10      val rd_addr = Output(UInt(5.W))
11      val rd_wdata = Output(UInt(64.W))
12      val mem_addr = Output(UInt(32.W))
13      val mem_rdata = Output(UInt(64.W))
14      val mem_wdata = Output(UInt(64.W))
15      val regs = Vec(32, Output(UInt(64.W)))
16  }
```

Listing 3: RVFI Definition

Next, we need to define the Spec module and Check modules. For each instruction or small property that needs to be verified, a separate Check module is used to describe it. Its input is RVFI, and its output is SpecOut, which is the correct state information after the instruction execution. Define the Check module, the input is RVFI from the Chip and SpecOut from some Spec, and assert the consistency of the two signals to verify the correctness of the execution of each instruction of the processor. In the current version, we have verified the correctness of the execution of the `add`, `ld`, and `beq` instructions. The specific verification design can be found in the GitHub repository.

Eventually, we found that there was an error in the design of this processor. As shown in Table I ChiselFV gives an execution path that would fail. For the `beq` instruction, we need to compare the value of the two source operands, but in the original design, these two source operands were not considered data hazards. Instead, they were directly taken from the register. Later, we modified the design on the Chisel version and passed the relevant verification.

Table I: ChiselFV Output Counterexample

binary	**instruction**	**PC**	**related regs**
0x00000013	NOP	0xFFFFFFFC	x17: 0 x20: 0
0x0000F883	ld x17, 0(x1)	0x0	x17: 0x80000001 x20: 0
0x00508A83	ld x21, 5(x1)	0x4	x17: 0x80000001 x20: 0
0x00108C93	addi x25, x1, 1	0x8	x17: 0x80000001 x20: 0
0x03488063	beq x17, x20, 32	0xC	x17: 0x80000001 x20: 0 wrong branch taken
0x00000013	NOP	-	x17: 0x80000001 x20: 0
0xFC108AE3	-	0x2C	x17: 0x80000001 x20: 0

IV. Related Work

This section presents a brief overview of the formal verification in hardware.

Formal verification of hardware mainly uses model checking. Model Checking is a method that can be fully automated but has a state explosion problem. To counter this problem, new technologies are introduced to reduce the state space of the transition system.

At the turn of the last century, a new generation of Boolean satisfiability (SAT) solvers such as Chaff [21] and Satisfiability Modulo Theories (SMT) [22] brought about a leap in the performance and scalability. Therefore, many model checking algorithms based on SAT/SMT solvers have been widely applied in hardware verification, such as BMC [23], k-induction [24], IC3/PDR [25], etc.

Besides the algorithms, the hardware model checking tools are also an important research direction. Aina Niemetz, Clifford Wolf's work [26] proposed BtorMC, a model checker built on Boolector. BtorMC supports the format Btor2, a new word-level model checking and witness format. Pono, a flexible and extensible SMT-based model checker, is designed to be both a research platform for developing and improving model checking algorithms and a performance-competitive tool that can be used for academic and industry verification applications [27]. SymbiYosys is an open-source frontend driver program for Yosys-based formal hardware verification flow [11]. ChiselVerify [3] and Chisel's formal support [4], detailed in Chapter I, are both verification work at the Chisel level.

In addition to Chisel, there are other languages used to generate HDLs. SpinalHDL [28] is similar to Chisel, an HDL generation tool built at the Scala level, with support for basic formal property descriptions and tool calls. Amaranth HDL [29], on the other hand, chooses to use Python to build the hardware.

V. Conclusion and Future Work

In this paper, we introduce ChiselFV, a formal verification framework for Chisel. ChiselFV provides a workflow for formal verification of Chisel modules, which can be combined with Chisel-based hardware design development to significantly improve the efficiency and simplicity of hardware development and verification. ChiselFV provides strong property description support, which enables verification work previously done at the low-level RTL level, but can be done in Chisel, improving the level of hardware verification.

In the future, we will provide more advanced support for Chisel's formal verification. Currently, our verification backend is to convert it to SVAs and then perform the verification process. This means we cannot symbolize the module's parameters in Chisel and must instantiate it before verification. We will further develop a verification backend for Chisel, which will be able to symbolize the parameters of the modules for formal verification. At the same time, we will apply ChiselFV to more open-source Chisel projects to continuously improve the usefulness of ChiselFV.

References

[1] W. J. Dally, Y. Turakhia, and S. Han, "Domain-specific hardware accelerators," *Communications of the ACM*, vol. 63, no. 7, pp. 48–57, 2020.

[2] J. Bachrach *et al.*, "Chisel: constructing hardware in a scala embedded language," in *DAC Design Automation Conference 2012*. IEEE, 2012, pp. 1212–1221.

[3] A. Dobis, T. Petersen *et al.*, "Chiselverify: An open-source hardware verification library for chisel and scala," in *2021 IEEE Nordic Circuits and Systems Conference (NorCAS)*. IEEE, 2021, pp. 1–7.

[4] ——, "Open-source verification with chisel and scala," *arXiv preprint arXiv:2102.13460*, 2021.

[5] C. Barrett, A. Stump, C. Tinelli *et al.*, "The smt-lib standard: Version 2.0," in *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*, vol. 13, 2010, p. 14.

[6] L. d. Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[7] A. Biere, A. Cimatti *et al.*, "Bounded model checking." *Handbook of satisfiability*, vol. 185, no. 99, pp. 457–481, 2009.

[8] S. Vijayaraghavan and M. Ramanathan, *A practical guide for SystemVerilog assertions*. Springer Science & Business Media, 2005.

[9] C. Wolf, "Yosys open synthesis suite," 2016.

[10] Chiselfv. [Online]. Available: https://github.com/Moorvan/ChiselFV

[11] Wolf. Symbiyosys (sby) – front-end for yosys-based formal verification flows. [Online]. Available: https://github.com/YosysHQ/SymbiYosys

[12] A. Biere, A. Cimatti *et al.*, "Symbolic model checking without bdds," in *International conference on tools and algorithms for the construction and analysis of systems*. Springer, 1999, pp. 193–207.

[13] M. Sheeran, S. Singh, and G. Stålmarck, "Checking safety properties using induction and a sat-solver," in *International conference on formal methods in computer-aided design*. Springer, 2000, pp. 127–144.

[14] A. R. Bradley, "Sat-based model checking without unrolling," in *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 2011, pp. 70–87.

[15] A. M. Abdelhadi and G. G. Lemieux, "Modular multi-ported sram-based memories," in *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, 2014, pp. 35–44.

[16] M. Xiang, Y. Li *et al.*, "Parameterized design and formal verification of multi-ported memory," in *2022 26th International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, 2022, pp. 33–41.

[17] Design and verification of multi-ported memory. [Online]. Available: https://github.com/VerificaticationStudio/MultPortedRAM

[18] D. Patterson and J. Hennessy, "Computer organization and design risc-v edition," 2017.

[19] Risc-v formal framework in chisel. [Online]. Available: https://github.com/Moorvan/RISCV-Formal-Chisel

[20] Risc-v formal verification framework. [Online]. Available: https://github.com/SymbioticEDA/riscv-formal

[21] M. W. Moskewicz, C. F. Madigan *et al.*, "Chaff: Engineering an efficient SAT solver," in *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*. ACM, 2001, pp. 530–535.

[22] C. Barrett and C. Tinelli, "Satisfiability modulo theories," in *Handbook of model checking*. Springer, 2018, pp. 305–343.

[23] A. Biere, A. Cimatti *et al.*, "Symbolic model checking without bdds," in *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99*, vol. 1579. Springer, 1999, pp. 193–207.

[24] C. Tinelli, "Smt-based model checking." in *NASA Formal Methods*, 2012, p. 1.

[25] A. R. Bradley, "Sat-based model checking without unrolling," in *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, vol. 6538. Springer, 2011, pp. 70–87.

[26] A. Niemetz, M. Preiner *et al.*, "Btor2, btormc and boolector 3.0," in *International Conference on Computer Aided Verification*. Springer, 2018, pp. 587–595.

[27] M. Mann, A. Irfan *et al.*, "Pono: a flexible and extensible smt-based model checker," in *International Conference on Computer Aided Verification*. Springer, 2021, pp. 461–474.

[28] Spinalhdl. [Online]. Available: https://github.com/SpinalHDL/SpinalHDL

[29] Amaranth hdl. [Online]. Available: https://github.com/amaranth-lang/amaranth