# Formally Verifying Arithmetic Chisel Designs for All Bit Widths at Once

Weizhi Feng[1,2], Yicheng Liu[1,2], Jiaxiang Liu[3], David N. Jansen[1], Lijun Zhang[1,2], Zhilin Wu[1,2(✉)]
[1]Key Laboratory of System Software (Chinese Academy of Sciences) and State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
[2]University of Chinese Academy of Sciences, Beijing, China [3]Shenzhen University, Shenzhen, China
{fengwz,liuyc,dnjansen,zhanglj,wuzl}@ios.ac.cn,jiaxiang0924@gmail.com

## ABSTRACT

Chisel is an open-source hardware description language embedded in Scala to facilitate parameterized and reusable digital circuit design. Chisel is becoming increasingly popular and has been used to design RISC-V CPUs, e.g. RocketChip and XiangShan. While Chisel features high-level hardware designs, its verification is still low-level: Low-level (e.g. Verilog) programs are first generated from Chisel programs, then the verification tools are applied to these low-level programs. In this work, we focus on formal verification of arithmetic units. Efficient low-level formal verification of arithmetic units has always been a challenge and remains an active research area, attributed to the state explosion problem brought on by bit widths. To circumvent this problem for arithmetic Chisel designs, we propose an approach to their high-level formal verification so that their correctness is verified for all bit widths at once, instead of for each bit width separately. The key idea is to transform arithmetic Chisel designs into Scala software programs that simulate their behaviors, where the high-level features are preserved, then resort to Stainless, a deductive formal verification tool for Scala. We validate the effectiveness of this approach by formally verifying the correctness of dividers and multipliers in two representative open source RISC-V processors, namely, RocketChip and XiangShan. Compared to the existing proof-assistant-based parameterized verification approaches for arithmetic designs (e.g. Kami), the verification cost in our approach is much lower on average.

## KEYWORDS

Chisel, Multipliers and dividers, Scala, Formal verification, Proof refinement.

**ACM Reference Format:**
Weizhi Feng, Yicheng Liu, Jiaxiang Liu, David N. Jansen, Lijun Zhang, Zhilin Wu. 2024. Formally Verifying Arithmetic Chisel Designs for All Bit Widths at Once. In *61st ACM/IEEE Design Automation Conference (DAC '24), June 23–27, 2024, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3649329.3657311

## 1 INTRODUCTION

Chisel [3] is a hardware description language (HDL) embedded in Scala that comprises a library of special class definitions, predefined objects, and usage conventions for writing highly parameterized hardware design generators. High-level hardware designs in Chisel are generated by manipulating circuit components using Scala functions, and their interfaces are encoded by Scala types. Hardware can also be constructed by leveraging Scala's functional and object-oriented programming features, thus enabling parameterized, modular, and reusable hardware generators that improve the productivity and robustness of hardware designs. Chisel has been successfully used to build RISC-V processors such as RocketChip [1], RISC-V BOOM [35], NutShell [24], and XiangShan [30, 33].

The arithmetic unit is one of the most crucial parts in a processor. The demands for speed-, power-, and area-efficiency make arithmetic units, especially the multipliers and dividers therein, sophisticated and hence error-prone. Formal verification of arithmetic units is necessary. Currently, once a Chisel design is ready, it is compiled down and Verilog/SystemVerilog programs are generated. Off-the-shelf verification tools for Verilog/SystemVerilog can then be applied. Compared to Chisel programs, Verilog/SystemVerilog programs are low-level: When Verilog/SystemVerilog programs are generated from Chisel programs, parameters (e.g. bit widths) are instantiated and high-level data structures (e.g. bundles and vectors) are all flattened. Therefore, although Chisel hardware designs are high-level, their verification is nonetheless still low-level. Specifically for arithmetic units, the low-level verification through Verilog/SystemVerilog means that even though a parameterized Chisel design is a single design for all bit widths, it has to be compiled to multiple separate Verilog/SystemVerilog programs, one for each bit width, and each of them has to be verified separately.

In this work, we advocate to unleash the power of the high-level language Chisel: arithmetic designs in Chisel are better to be paired with high-level verification. Our goal is to *formally* verify the correctness of parameterized arithmetic Chisel designs for all bit widths at once, instead of once for each bit width separately. To achieve this goal, we face the following two challenges.

- **C1.** It is nontrivial to formalize the semantics of Chisel, since the actual semantics of Chisel is only defined by its compilation process.
- **C2.** It is not an easy task to formally verify the correctness of parameterized arithmetic designs, which typically requires to reason about non-linear integer operations as well as infinitely many possibilities of bit widths.

*Contribution.* We tackle the aforementioned challenges by proposing an approach for the high-level formal verification of parameterized arithmetic Chisel designs. Specifically, we make the following contributions.


*DAC '24, June 23–27, 2024, San Francisco, CA, USA*

ACM ISBN 979-8-4007-0601-1/24/06.
https://doi.org/10.1145/3649329.3657311

- To address challenge **C1,** we transform Chisel hardware generators into Scala software simulators, where the parallel-execution hardware semantics of Chisel is simulated by the sequential-execution software semantics of Scala. As a result, the formal semantics of a Chisel program is given by that of the resulting Scala program.
- After a Scala program is generated from the given Chisel program, to resolve the challenge **C2,** Stainless [18, 28], a deductive verification tool for Scala, is leveraged to prove the functional correctness of the Scala program. Furthermore, although Stainless is capable of automating a large part of the proof process, it may still get stuck. We propose various proof-refinement strategies and proof techniques to help.
- We validate the effectiveness of our approach by verifying the multipliers and dividers in representative RISC-V processor Chisel designs, namely, RocketChip and XiangShan.

To the best of our knowledge, this is the first time that high-level parameterized formal verification has been achieved for Chisel hardware designs.

*Related work.* Many efforts have been made to verify arithmetic circuit designs formally, especially for multipliers and dividers, e.g. by BDDs [5, 6, 10, 17], computer algebra [15, 16, 20, 22, 23], and theorem provers [25, 26]. These works focus on low-level designs with fixed bit widths and have to verify each bit width separately, where the verification complexity grows exponentially in the bit width. In contrast, our approach targets the high-level parameterized arithmetic Chisel designs, and allows formal verification for all bit widths at once.

Parameterized verification was also considered in [4, 9, 13]. These works propose dedicated HDLs based on proof assistants. Parameterized circuit designs can then be specified with these HDLs, formally verified in the proof assistants, and even synthesized into low-level circuits automatically. Nevertheless, our motivation is to verify existing arithmetic designs written in Chisel instead of reimplementing them. Therefore, these approaches are not directly applicable to our setting. Additionally, formal verification in these proof-assistant-based approaches is largely manual, while ours is mostly automated. As a result, the verification effort of our approach is much lower (see Section 4 for more details about the comparison). Automated parameterized verification was also investigated with the theorem prover Rewrite Rule Laboratory (RRL) [14]. The major difference between our approach and RRL lies in the fact that our approach verifies the actual arithmetic hardware designs, while RRL verifies only the underlying algorithms.

As for Chisel designs, the current official verification tool for Chisel is ChiselTest [19]. It includes features for formal verification and simulation [11, 12], as well as support for more expressive assertions [27, 29, 34]. However, all these works compile Chisel designs into low-level designs and instantiate parameters before applying verification techniques. Our work instead proposes to stay at the high level, keep the bit-width parameters of arithmetic Chisel designs and verify them for all bit widths at once. Our transformation from Chisel to Scala can be seen as an extension of the transformation of Verilog to C in [21]. Compared to the Verilog to C transformation, high-level features of Chisel, e.g. bundles and recursive functions, make the transformation much more involved.

*Organization.* Section 2 is devoted to the transformation of Chisel hardware programs into Scala software programs. Section 3 shows how Stainless can be used to prove the correctness of the Scala software programs generated from arithmetic Chisel designs. Section 4 describes the case studies. Section 5 concludes this paper.

## 2 TRANSFORMING CHISEL TO SCALA

The transformation algorithm addresses the semantic differences between a Chisel hardware generator and a Scala circuit simulator. We use the Chisel program in Listing 1 to illustrate the transformation. The program is parameterized by `len` as the bit widths of its input `in` (Line 3) and output `out` (Line 4). The core register `R` gets the value of `io.in` (Line 10) at the end of the first clock cycle. Then in each clock cycle afterwards, it is rotated to the right by 1 bit (Line 13). Hence, after `len` additional clock cycles, `state` becomes `true` again, `cnt` is equal to `len`, and `R` regains the value of `io.in`. For instance, if `len` is 4 and `io.in` is 1001, then at the end of the first clock cycle, `R` takes the value 1001 (where `cnt` is 0), and in the second clock cycle, the 0th bit of $R$, i.e. 1, is concatenated to the left of 100, thus at the end of this clock cycle, $R$ takes the value 1100 (where `cnt` is 1). Afterwards, it takes the values 0110, 0011, and 1001, at the end of the 3rd, 4th, and 5th clock cycle respectively.

```
1  class Example(len: Int) extends Module {
2    val io = IO(new Bundle {
3      val in  = Input(UInt(len.W))
4      val out = Output(UInt(len.W))
5      val ready = Output(Bool()) })
6    val state = RegInit(true.B)
7    val cnt = RegInit(0.U(len.W))
8    val R = Reg(UInt(len.W))
9    when (io.ready) {
10     R := io.in
11     state := false.B
12   }.otherwise {
13     R := Cat(R(0), R(len - 1, 1))
14     cnt := cnt + 1.U
15     when (cnt === (len - 1).U) { state := true.B } }
16   io.ready := state
17   io.out := R }
```

**Listing 1: A Chisel program: The running example**

### 2.1 Chisel Types and Operations

Chisel is embedded in Scala. A Chisel program often contains both Chisel and Scala types. To transform a Chisel program to Scala, Chisel types need to be modeled by Scala types.

Chisel provides basic types like `UInt`, `SInt` and `Bool`, on which more complex types (e.g. `Bundle` and `Vec`) can be built. Hardware signals (e.g. `Reg`, `Wire` and `IO`) are constructed with these types. We define Scala types `UInt`, `SInt` and `Bool` to model the corresponding Chisel basic types, and represent all the hardware signals in Chisel with Scala variables. For instance, the hardware signal `in` of Chisel type `UInt` (Line 3 in Listing 1) is modeled by the Scala variable `io_in` of Scala type `UInt` (Line 1) in the generated Scala program depicted in Listing 2. Specifically, the bit-vector Chisel types are modeled by bounded mathematical integers. Take `UInt` as an example. The Scala `UInt` contains two fields: a mathematical integer `value` of type `BigInt` and its bit width `width`. The former is the unsigned interpretation of the bit-vector, and the latter restricts the range of `value`. The reasons to model Chisel `UInt` in this

way instead of bit-vectors directly are twofold. Firstly, as we target parameterized Chisel designs, most bit-vectors therein are of parameterized bit widths. Nevertheless, our verification back-end Stainless does not support such bit-vectors of parameterized widths. Secondly, as we focus on arithmetic circuits, mathematical integers are more suitable to prove arithmetic properties than bit-vectors. Note that modeling bit-vectors as bounded mathematical integers is still accurate.

The operations associated with the Chisel basic types, such as Boolean operations, arithmetic operations, and bitwise operations, are also modeled by our defined operations for the Scala types (e.g. the Chisel concatenation operation `Cat` at Line 13 in Listing 1 becoming the Scala `Cat` at Line 16 in Listing 2). Moreover, the connection (`:=`) in Chisel is transformed to our defined connection (`:=`) in Scala, which behaves like assignment with additional computation of the bit width w.r.t. the Chisel semantics. Finally, the Chisel control-flow `when-otherwise` is modeled by `if-else` in Scala.

```
1  case class Inputs(io_in: UInt)
2  case class Outputs(io_out: UInt, io_ready: Bool)
3  case class Regs(state: Bool, cnt: UInt, R: UInt)
4  case class Example(len: BigInt) {
5    def Trans(ins: Inputs, regs: Regs): (Outputs, Regs)={
6      var io_out = UInt.empty(len) // initialization
7      var io_ready = Bool.empty()
8      var state_next = regs.state
9      var cnt_next = regs.cnt
10     var R_next = regs.R
11     io_ready := regs.state  // main body
12     if (io_ready) {
13       R_next := ins.io_in
14       state_next := Lit(false).B }
15     else {
16       R_next := Cat(regs.R(0), regs.R((len - 1), 1))
17       cnt_next := (regs.cnt + Lit(1).U)
18       if (regs.cnt === Lit(len - 1).U)
19         state_next := Lit(true).B }
20     io_out := regs.R
21     (Outputs(io_out, io_ready),            // outputs
22      Regs(state_next, cnt_next, R_next)) } // next regs
23   def Run(ins: Inputs, regInit: Regs): (Outputs, Regs)={
24     val (outs, newRegs) = Trans(ins, regInit)
25     val timeout = setTimeout()
26     if (!timeout) Run(ins, newRegs) else (outs, newRegs)}
27   def Init(ins: Inputs, rdInit: Regs): (Outputs, Regs)={
28     val rgInit = Regs(Lit(true).B,Lit(0,len).U,rdInit.R)
29     Run(ins, rgInit) } }
```

**Listing 2: Scala program generated from the running example**

## 2.2 Modeling Sequential Circuits

Chisel provides the type `Reg` to define registers and construct sequential circuits. In Chisel programs, the global clock signal is implicitly connected to registers. A (sequential) Chisel design is therefore transformed into a Scala function named `Trans` (Line 5 in Listing 2), which describes the combinational behavior within a single clock cycle, and a loop that mimics clock ticking. The `Trans` function accepts the states of inputs and registers and generates new states of outputs and registers. These new states of registers will become the current register states in the next clock cycle. To simulate the correct behavior of register updates within a clock cycle, we include two Scala variables for each register: one representing the current state (e.g. `R` at Line 10 in Listing 2), and the other for the next state (e.g. `R_next` at Line 10 in Listing 2). The Scala variables for the next state are initialized with the variables for the current state (Lines 8–10 in Listing 2). They are updated if the corresponding registers are connected to new objects. For instance, `R` is connected at Line 13 in Listing 1, thus `R_next` is updated at Line 16 in Listing 2.

The loop that mimics the clock ticking is implemented as a recursive function `Run` (Line 23 in Listing 2). Since arithmetic circuits typically produce outputs after computing for several clock cycles, the number of invocations of the recursive function `Run` is controlled by the user-defined function `setTimeout` (Lines 25–26 in Listing 2), which is determined by the property of the program that we would like to verify. For instance, in Listing 2, the timeout condition is set to `newRegs.cnt.value == len`. The Scala function `Init` (Line 27 in Listing 2) combines the register initialization with the function `Run` and so models the original Chisel program faithfully.

## 2.3 Modeling Combinational Behaviors by Reordering Statements

Although a Chisel program is read sequentially when generating a circuit, to understand the *combinational behaviors of the circuit in one clock cycle,* the execution of the connect statements in the circuit, thus the connect statements in the original Chisel program as well, should be seen as parallel, except for the "last-connect-wins" principle. For instance, in Listing 1, the connect statements are executed in parallel to update the values of the signals, e.g. `state`, `cnt`, and `R`. Therefore, there is a mismatch between the parallel execution of the connect statements in a Chisel program and the sequential execution of the assignment statements in a Scala program. As a consequence, we have to reorder the statements during the transformation of a Chisel program, so that the generated Scala program simulates correctly the behavior of the original Chisel program.

We reorder the statements in a Chisel program according to the dependencies between them. For instance in Listing 1, `io.ready` is used as condition of `when` at Line 9, but it is connected at Line 16. Therefore, all statements in the `when` block depend on the statement at Line 16. Based on the dependencies, we reorder these statements accordingly. Thus, the statement `io_ready := regs.state` (Line 11 in Listing 2) is placed before the `if` statement (Line 12) generated from `when`.

Dependencies between statements are established w.r.t. the Chisel hardware signals and Scala variables therein. Reordering is then performed based on the topological order of the dependencies. Specifically, dependencies exist in the following four scenarios. (i) According to the Chisel "last-connect-wins" principle, if multiple connections to the same hardware signal coexist, the last connection takes effect. This is consistent with the sequential execution semantics of Scala programs. So, each connection to the same signal depends on the previous one, such that the last connection remains at the end. (ii) A hardware signal needs to be computed before being used. Thus a statement using some signal depends on the signal's last connection. (iii) All hardware signals and Scala variables should maintain the relative order between definition, assignment, and reference. (iv) The statements inside a `when-otherwise` or `if-else` block should follow the dependencies of the hardware signals or Scala variables involved in the branch condition.

For function invocations in Chisel programs, we conduct reordering within the body of these functions independently. For the instantiation of submodules, we deal with them similarly to function invocations, but we only allow submodules that are combinational circuits. Additionally, to obtain more flexible statement reordering, we break down a `when-otherwise` or `if-else` block into multiple blocks, such that each block contains only one statement. For instance, the `when-otherwise` block (Lines 9–15) containing 5 statements in Listing 1 is split into 5 blocks. The 1st one is `when(io.ready){R := io.in}` and the 4th one is `when(io.ready){}.otherwise{cnt := cnt + 1.U}`. In order to preserve the structure of the given Chisel program as much as possible during the transformation, we merge those Scala statements generated from a Chisel `when-otherwise` statement, provided that they are adjacent after the reordering. The `if-else` block (Lines 12–19) in Listing 2 is generated from the `when-otherwise` block by such a merging.

## 2.4 Chisel programs that can be transformed correctly

The combination of Scala and Chisel constructs makes the transformation of Chisel programs into Scala quite tricky in general. Our algorithm at present can only transform correctly a subclass of Chisel programs. Note that this subclass of Chisel programs is still general enough to cover the case studies in Section 4.

Here, we state the conditions that define the subclass of Chisel programs that can be correctly transformed. We first state the constraints on the macro level, that is, the conditions on one module as a whole. Then we state the constraints on the micro level, that is, the conditions for the constructs within modules.

*Macro level conditions.*

(1) `Module` classes (resp. `Bundle` classes) are forbidden to inherit from another `Module` class. For instance, if class A inherits from the class `Module` in a Chisel program, then no other class B can inherit from A.
(2) There is only one clock, namely the global clock.
(3) There are no circular dependencies between signals in the *global* sense. That is, if in one execution path, a signal $a$ depends on another signal $b$, then there should *not* be another branch where $b$ depends on $a$. (This includes cases where $a$ depends on $b$ indirectly through some other signal.)

*Micro level conditions.*

(1) For each module, scopes of the signals and variables in it, except those defined in functions, should be *global*, i.e., their scopes should be the whole module, instead of only a `when` or `if` statement. Similar restrictions are enforced for functions.
(2) For each connect statement, signals declared with the Scala keyword `var` are forbidden to occur in its left-hand-side, as these signals are similar to modifiable references and the actual signals represented by them are hard to identify statically. Function parameters share similar restrictions.
(3) For each bundle, its construct parameters (used e.g. for bit widths) can only be atomic signals of `Int` types, but not bundle types. Moreover, it is *pure* in the sense that only signal declarations are allowed, statements of the other types, as well as class or function definitions, are all forbidden.
(4) For each `Module` or `Bundle` class, the definition of classes in it is forbidden. Moreover, for each module, functions defined outside it are forbidden to be used (that is, only functions defined in this module and library functions are allowed).
(5) For each function, its behavior is required to be combinational in the sense that it does not involve registers. Moreover, whenever this function is used, its input signals do not depend on its output signals.
(6) For each submodule or `for` loop, conditions similar to functions are enforced. Moreover, `while` loops are forbidden.

# 3 FORMAL VERIFICATION OF SCALA PROGRAMS

Stainless is a verification tool for an expressive subset of Scala. It provides the keywords `require` and `ensuring` to write pre- and post-conditions, generates *verification conditions* (VCs) and utilizes SMT solvers to prove them. The verification succeeds if all the generated VCs are proved. It also offers a domain specific language for users to guide the proofs with hints in case the VCs cannot be proved by SMT solvers.

Given a Scala program generated by our transformation (Section 2), we conduct the verification process as follows.

(1) Formalize the expected correctness properties with pre-, post-conditions and a terminating condition; specify the terminating condition in the function `setTimeout` (Line 25 in Listing 2) and the pre- and post-conditions using keywords `require` and `ensuring`, respectively, in the function `Init`.
(2) Apply Stainless to verify the annotated Scala program.

For example, we have deduced in Section 2 that the register `R` in the parameterized Chisel program in Listing 1 is equal to `io.in` when `cnt` equals to `len`. If we would like to verify this property, the terminating condition should be `cnt == len`. Therefore in the first step, we specify the terminating condition in `setTimeout` as `newRegs.cnt.value == len`. The post-condition of `Init` is specified as `ensuring(Regs.R.value == ins.io_in.value)`. As for the pre-condition, we require that the bit width `len > 1`, defined with the keyword `require`. Then in the second step, Stainless is invoked to verify the Scala program.

The application of Stainless in the second step is usually not as easy as pushing a button. Since Scala programs generated from arithmetic Chisel designs are parameterized and typically involve non-linear computations, the verification with Stainless often gets stuck. Human effort is needed to guide or refine the proofs. We empirically propose some proof-refinement strategies and proof techniques to help Stainless go through the proof process.

## 3.1 Proof-Refinement Strategies

Let us distinguish the situations where the proofs get stuck and formulate strategies to refine the proofs.

*Stuck with* `Init`. When the verification of `Init`'s post-condition is stuck, we first add pre- and post-conditions for `Run` and `Trans`. *Invariants* are essential for the verification of parameterized programs. The key is to identify the invariants of the recursive function `Run` that imply the post-condition of `Init`. The invariants are then provided as the pre- and post-conditions of `Run` and `Trans`.

For example, the Chisel program in Listing 1 updates the register R by rotating it to the right by 1 bit in each clock cycle, and meanwhile increases cnt by 1. Thus two invariants are established: the most significant cnt bits of R are identical to the least significant cnt bits of io.in, and the least significant len − cnt bits of R are identical to the most significant len − cnt bits of io.in. The invariants can be expressed as the pre- and post-conditions of Trans as in Listing 3, where variable names are simplified for brevity: R_n for R_next, w for len, c for cnt, c_n for cnt_next, and i for io_in. Note that we use the names of Scala variables to refer to their values (e.g. R for R.value) for clarity. Recall that we use mathematical integers to model bit-vectors, hence the bit-vector operations become integer operations in Listing 3, where Pow2($x$) is defined to compute $2^x$.

```
1 require( 0 <= c < w &&
2   R / Pow2(w-c) == i % Pow2(c) &&
3   R % Pow2(w-c) == i / Pow2(c) )
4 ...   // body of Trans
5 ensuring( 0 <= c_n <= w &&
6   R_n / Pow2(w-c_n) == i % Pow2(c_n) &&
7   R_n % Pow2(w-c_n) == i / Pow2(c_n) )
```

**Listing 3: Pre- and Post-Conditions of Trans**

*Stuck with verification conditions.* When the proof of a verification condition is stuck, we first check whether the condition involves any variable that was modified by a function or a loop. If yes, we add pre- and post-conditions for that function or loop, similar to the case "stuck with Init". Otherwise, we add a manual proof for that condition via the DSL provided by Stainless.

For instance, when proving R_n / Pow2(w-c_n) == i % Pow2(c_n) (Line 6 in Listing 3) of Trans, Stainless is stuck. Since the condition contains no variable that was modified by functions or loops (cf. Listing 2), so we write a deductive proof with the DSL as shown in Listing 4. The proof generally states that each expression is equal to the expression at the next line. The *tactic* trivial indicates that the equality holds by trivial simplification.

```
1 { R_n / Pow2(w-c_n)                          ==:| trivial |:
2   ((R%2)*Pow2(w-1) + R/2) / Pow2(w-c-1) ==:| trivial |:
3   ((R%2)*Pow2(w-c-1)*Pow2(c) + R/2) / Pow2(w-c-1)
4   ...   // other proof steps
5   i % Pow2(c_n) }.qed
```

**Listing 4: A Deductive Proof in Trans**

*Stuck with tactics.* When the check of some tactic (e.g. trivial) is stuck, we need to add lemmas and/or refine the proof. For instance, the proof in Listing 4 gets stuck at Line 2, where we leverage the property $2^x \cdot 2^y = 2^{x+y}$ to equate Pow2(w-c-1) * Pow2(c) with Pow2(w-1). This step is not "trivially" proved because Stainless cannot deduce the property automatically. Hence we define the property as a lemma Pow2Mul and prove it by induction. We replace the problematic trivial with the instantiation of this lemma Pow2Mul, then the proof of this step goes through.

## 3.2 Proof Techniques

We summarize the proof techniques that are useful for proving the functional correctness of arithmetic designs in Stainless.

*Ghost variables.* In some scenarios, the variables in the program are insufficient or inconvenient to formulate pre- or post-conditions. We can define ghost variables to store needed values. For example in the XiangShan divider, the most significant $2 \cdot$ len $+ 1 -$ cnt bits and the least significant cnt bits of a register shiftReg are needed for representing the pre- and post-conditions. We hence introduce two ghost variables to store them.

*Bit-vector related operations and lemmas.* Chisel designs usually manipulate bit-vectors, but we model bit-vectors and their operations with mathematical integers and their operations (e.g. integer division and modulo, the Pow2 function). Some properties related to these integer operations are difficult for Stainless to prove automatically. To better facilitate our verification requirements, we develop a library of these operations and the corresponding lemmas. For instance, the lemma $x \geq y \implies (a \bmod 2^x) \bmod 2^y = a \bmod 2^y$ reflects that if $x \geq y$, taking the least significant $x$ bits of $a$ and then taking from the result the least significant $y$ bits, will have the same effect as taking the least significant $y$ bits from $a$ directly. In the end, we obtain a bit-vector library that includes 6 operations and 10 lemmas, which occupy 320 lines of Scala code in total.

*List-related operations and lemmas.* As mentioned, our transformation models UInt with mathematical integers. But some arithmetic designs (e.g. the XiangShan multiplier) split a UInt signal into bits and store them in an Array or Seq signal. Our algorithm will transform the Array or Seq signal into a Scala list. We define some operations regarding the different interpretations of the value(s) that a list may represent, e.g. Sum() that computes for a list $a_1, \cdots, a_n$ the sum $\sum_{1 \leq i \leq n} a_i$ and toZ() that computes the weighted sum $\sum_{1 \leq i \leq n} a_i 2^i$. We also prove some lemmas expressing basic properties of these operations. Furthermore, when a new list $l'$ is obtained from a list $l$ by updating the value in some position $i$ to $v$, we also write some lemmas to prove properties related to this update, i.e. properties relating $l$, $l'$, $i$, and $v$. Finally, we define some operations related to two-dimensional lists, that is, lists of lists, which are also used in some arithmetic designs (e.g. the XiangShan multiplier). In the end, we obtain a list library that includes 7 operations and 3 lemmas, which occupy 140 lines of Scala code in total.

# 4 IMPLEMENTATION AND CASE STUDIES

We implemented our transformation as a plugin of the Scala compiler[1]. The plugin converts the Scala abstract syntax tree (AST) into a special AST, then applies the passes including preprocessing, dependency analysis, and statement reordering, and finally generates Scala code as input to Stainless. The automatic transformation plugin comprises approximately 4500 lines of Scala code.

We have successfully applied our approach to verify the correctness of Chisel designs for integer division and multiplication in open-source RISC-V processors XiangShan and RocketChip, for all bit widths at once. Both divider designs are based on the classic, widely-used shift/subtract algorithm: one is from the Radix2Divider module [32] in the XiangShan project, the other is from the RocketChip project [2]. The two multiplier designs are also from XiangShan and RocketChip, respectively. The XiangShan multiplier [31]

[1] https://github.com/iscas-tis/chicala

**Table 1: Verification Effort**

| Design | #Chisel (#Verilog) | #Scala | #Scala-vrf |
|---|---|---|---|
| X-divider | 73 (388) | 166 (2.3×) | 484 (2.9×) |
| R-divider | 129 (177) | 251 (1.9×) | 451 (1.8×) |
| X-multiplier | 137 (27278) | 154 (1.1×) | 1675 (10.9×) |
| R-multiplier | 120 (153) | 229 (1.9×) | 507 (2.2×) |

implements the Booth encoding and the Wallace-Tree algorithm, which is one of the most efficient and widely-used algorithms for integer multiplication. The list library mentioned in Section 3.2 plays an important role in the verification of this multiplier. On the other hand, the RocketChip multiplier [2] is based on the textbook shift/add algorithm. In all case studies, ghost variables and the bit-vector library are utilized to ease the verification process.

We summarize the verification effort for Chisel designs in Table 1, which shows the line counts of the Chisel/Verilog (64-bits) code ("#Chisel (#Verilog)"), the generated Scala code ("#Scala"), and the Scala code with pre- and post-conditions, lemmas, and proofs ("#Scala-vrf"). The multipliers in parentheses are the ratios to the preceding column. We conclude that our transformation does not induce code-size explosion (2.3× at most). The manual proof effort in all but one cases is lightweight (1.8–2.9×). The much greater proof effort in the X-multiplier is attributed to the fact that its data structure and algorithm are much more complex than the other designs. (This is also reflected by the huge line-number blow-up in the Verilog code.) Moreover, we compare our proof effort with Kami [9]. In Kami, a Booth multiplier (without Wallace-tree) and a non-restoring divider are implemented and formally verified [7, 8]. The ratio between the number of lines of the proof code and that of the implementation code is more than 11. Therefore, the proof effort in our approach is much lower than in Kami[2].

## 5 CONCLUSION AND FUTURE WORK

We proposed a novel approach to verify the correctness of arithmetic Chisel designs, for all bit widths at once, instead of once for each bit width separately. Our approach is based on an automatic transformation of arithmetic Chisel designs into Scala programs, and Stainless, a deductive verification tool for Scala, as well as some proof-refinement strategies and proof techniques specialized to arithmetic designs. This work is a proof-of-concept for the high-level verification of Chisel hardware designs. We hope that we can inspire more research in this direction.

As future work, at first, we would like to validate this approach on more arithmetic Chisel designs. It is also interesting to see whether the automated transformation from Chisel programs to Scala programs can be improved and extended to facilitate high-level (not necessarily formal) verification of more complex Chisel hardware designs. Finally, we plan to validate the transformation from Chisel to Scala empirically by running both Chisel and Scala code and comparing their results.

[2]Although in Kami, for each multiplier/divider, the proofs for 32 and 64 bits are provided separately, the two proofs are actually almost identical. Thus they can be seen as parameterized.

## ACKNOWLEDGMENTS

This work is supported by the Strategic Priority Research Program of the Chinese Academy of Sciences, Grant No. XDA0320101.

## REFERENCES

[1] K. Asanović, R. Avižienis, et al. 2016. *The rocket chip generator.* Technical Report. UCB/EECS-2016-17, EECS Department, University of California, Berkeley.
[2] K. Asanović, R. Avižienis, et al. 2023. Multiplier and divider in RocketChip. https://github.com/chipsalliance/rocket-chip/blob/master/src/main/scala/rocket/Multiplier.scala.
[3] J. Bachrach, H. Vo, B. C. Richards, et al. 2012. Chisel: constructing hardware in a Scala embedded language. In *DAC*. 1216–1225.
[4] F. Bornebusch, C. Lüth, and et al. 2020. Towards Automatic Hardware Synthesis from Formal Specification to Implementation. In *ASP-DAC*. 375–380.
[5] R. E. Bryant. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers* 35, 8 (1986), 677–691.
[6] R. E. Bryant. 1996. Bit-Level Analysis of an SRT Divider Circuit. In *DAC*. 661–665.
[7] J. Choi, M. Vijayaraghavan, B. Sherman, et al. 2023. Non-restoring divider in Kami. https://github.com/mit-plv/kami/blob/rv32i/Kami/Ex/Divider64.v.
[8] J. Choi, M. Vijayaraghavan, B. Sherman, et al. 2023. Radix-4 Booth Multiplier in Kami. https://github.com/mit-plv/kami/blob/rv32i/Kami/Ex/Multiplier64.v.
[9] J. Choi, M. Vijayaraghavan, B. Sherman, and et al. 2017. Kami: a platform for high-level parametric hardware specification and its modular verification. *Proc. ACM Program. Lang.* 1, ICFP (2017), 24:1–24:30.
[10] E. M. Clarke, M. Khaira, and X. Zhao. 1996. Word Level Model Checking - Avoiding the Pentium FDIV Error. In *DAC*. 645–648.
[11] A. Dobis, K. Laeufer, H. J. Damsgaard, et al. 2023. Verification of Chisel Hardware Designs with ChiselVerify. *Microprocess. Microsystems* 96 (2023), 104737.
[12] A. Dobis, T. Petersen, H. J. Damsgaard, et al. 2021. ChiselVerify: An Open-Source Hardware Verification Library for Chisel and Scala. In *NorCAS*. 1–7.
[13] J. P. P. Flor, W. Swierstra, and Y. Sijsling. 2015. Pi-Ware: Hardware Description and Verification in Agda. In *TYPES (LIPIcs, Vol. 69)*. 9:1–9:27.
[14] D. Kapur and M. Subramaniam. 2000. Using an induction prover for verifying arithmetic circuits. *Int. J. Softw. Tools Technol. Transf.* 3, 1 (2000), 32–65.
[15] D. Kaufmann and A. Biere. 2021. AMulet 2.0 for Verifying Multiplier Circuits. In *TACAS*. 357–364.
[16] D. Kaufmann and A. Biere. 2023. Improving AMulet2 for verifying multiplier circuits using SAT solving and computer algebra. *Int. J. Softw. Tools Technol. Transf.* 25, 2 (2023), 133–144.
[17] J. Kumar, Y. Miyasaka, A. Srivastava, and M. Fujita. 2023. Formal Verification of Integer Multiplier Circuits Using Binary Decision Diagrams. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 42, 4 (2023), 1365–1378.
[18] EPFL IC LARA. 2023. Stainless: Verification framework for a subset of the Scala programming language. https://github.com/epfl-lara/stainless.
[19] Richard Lin and Kevin Laeufer. 2023. ChiselTest. https://github.com/ucb-bar/chiseltest
[20] J. Lv, P. Kalla, and F. Enescu. 2013. Efficient Gröbner Basis Reductions for Formal Verification of Galois Field Arithmetic Circuits. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 32, 9 (2013), 1409–1420.
[21] R. Mukherjee, M. Tautschnig, and D. Kroening. 2016. v2c - A Verilog to C Translator. In *TACAS*. 580–586.
[22] C. Scholl and A. Konrad. 2020. Symbolic Computer Algebra and SAT Based Information Forwarding for Fully Automatic Divider Verification. In *DAC*. 1–6.
[23] C. Scholl, A. Konrad, and et al. 2021. Verifying Dividers Using Symbolic Computer Algebra and Don't Care Optimization. In *DATE*. 1110–1115.
[24] OSCPU team. 2023. NutShell RISC-V CPU. https://github.com/OSCPU/NutShell.
[25] M. Temel and W. A. Hunt. 2021. Sound and Automated Verification of Real-World RTL Multipliers. In *FMCAD*. 53–62.
[26] M. Temel, A. Slobodová, and W. A. Hunt. 2020. Automated and Scalable Verification of Integer Multipliers. In *CAV*. 485–507.
[27] YCA Tsai. 2021. *Dynamic verification library for Chisel.* Master's thesis. University of California, Berkeley.
[28] N. Voirol. 2019. *Verified Functional Programming.* Ph. D. Dissertation. EPFL, Switzerland. https://doi.org/10.5075/epfl-thesis-9479
[29] M. Xiang, Y. Li, and Y. Zhao. 2023. ChiselFV: A Formal Verification Framework for Chisel. In *DATE*. 1–6.
[30] Y. Xu, Z. Yu, D. Tang, et al. 2022. Towards Developing High Performance RISC-V Processors Using Agile Methodology. In *MICRO*. 1178–1199.
[31] Y. Xu, Z. Yu, D. Tang, et al. 2023. Multiplier in XiangShan. https://github.com/OpenXiangShan/XiangShan/blob/master/src/main/scala/xiangshan/backend/fu/Multiplier.scala.
[32] Y. Xu, Z. Yu, D. Tang, et al. 2023. Radix2Divider in XiangShan. https://github.com/OpenXiangShan/XiangShan/blob/master/src/main/scala/xiangshan/backend/fu/Radix2Divider.scala.
[33] Y. Xu, Z. Yu, D. Tang, et al. 2023. XiangShan: An open-source high-performance RISC-V processor. https://github.com/OpenXiangShan/XiangShan.
[34] S. Yu, Y. Dong, J. Liu, et al. 2022. CHA: Supporting SVA-Like Assertions in Formal Verification of Chisel Programs (Tool Paper). In *SEFM*. 324–331.
[35] J. Zhao, B Korpan, A. Gonzalez, and K. Asanovic. 2023. RISC-V BOOM: The Berkeley Out-of-Order RISC-V Processor. https://boom-core.org/.