Formally Verifying Arithmetic Chisel Designs for All Bit Widths at Once

冯维直，刘易铖，刘嘉祥，
David N. Jansen，张立军，吴志林

61st ACM/IEEE Design Automation Conference (DAC'24)

联系人：冯维直
手机号：18110026007 邮箱：fengwz@ios.ac.cn

Background

Chisel是一种新兴的硬件描述语言(HDL)，提供参数化、模块化的硬件开发能力

- 成功用于RISC-V开源处理器开发，如XiangShan，RocketChip。
- 现有Chisel形式验证方法将高层代码转换为底层迁移系统/Verilog程序，再调用后端工具：参数被实例化，高层结构被展平，电路规模增大造成状态空间爆炸；对同一电路设计的不同参数实例需要分别进行验证；针对特定电路设计的专用算法，适用性差。

Challenges

我们希望从Chisel代码层直接进行验证，保留参数化信息，摆脱“位宽竞赛”

- 直接对Chisel程序硬件和软件的混合语义进行形式化分析较为困难。
- 保留参数信息的高层验证依赖对不定位宽位向量类型和不定长时钟周期问题进行建模和推导。

Approach

使用软件模拟硬件语义，并使用辅助技术帮助现有的软件形式验证工具完成验证

- 生成Scala软件程序模拟Chisel硬件设计语义；
- 使用Stainless（Scala的形式验证工具）对生成的Scala软件程序进行形式验证；
- 人工提供前置-后置-条件，不变式和证明引理辅助Stainless工具完成“半自动”证明。

- 例：赋值语句重排序

```
when (io.ready) {
  R := io.in
  state := false.B
}.otherwise {
  R := Cat(R(0), R(len - 1, 1))
  cnt := cnt + 1.U
  when (cnt === (len - 1).U) { state := true.B } }
io.ready := state
io.out := R }
```

Listing 1: A Chisel program: The running example

```
io_ready := regs.state  // main body
if (io_ready) {
  R_next := ins.io_in
  state_next := Lit(false).B }
else {
  R_next := Cat(regs.R(0), regs.R((len - 1), 1))
  cnt_next := (regs.cnt + Lit(1).U)
  if (regs.cnt === Lit(len - 1).U)
    state_next := Lit(true).B }
io_out := regs.R
```

Listing 2: Scala program generated from the running example

- 例：前后置条件和引理

```
def Run(ins: Inputs, regInit: Regs): (Outputs, Regs)={
  val (outs, newRegs) = Trans(ins, regInit)
  val timeout = setTimeout(newRegs.cnt.value == ins.
    io_in.value)
  if (!timeout) Run(ins, newRegs) else (outs, newRegs)}
def Init(ins: Inputs, rdInit: Regs): (Outputs, Regs)={
  require(len > 1)
  val rgInit = Regs(Lit(true).B,Lit(0,len).U,rdInit.R)
  Run(ins, rgInit) } }
ensuring(Regs.R.value == ins.io_in.value)
```

```
( R_n / Pow2(w-c_n)                          ==:|trivial|:
((R%2)*Pow2(w-1) + R/2) / Pow2(w-c-1)  ==:|Pow2Mul|:
((R%2)*Pow2(w-c-1)*Pow2(c) + R/2) / Pow2(w-c-1)
...  // other proof steps
i % Pow2(c_n) ).qed

def Pow2Mul(x: BigInt, y: BigInt): Unit = {
  require(x >= 0 && y >= 0)
  decreases(x)
  x match {
    case 0 => ()
    case _ => {
      Pow2(x+y)              ==:| trivial |:
      2 * Pow2(x-1+y)        ==:| Pow2Mul(x-1, y) |:
      2 * Pow2(x-1) + Pow2(y) ==:| trivial |:
      Pow2(x) + Pow2(y)
    }.qed
  }
}.ensuring(_ => Pow2(x+y) == Pow2(x) * Pow2(y))
```

Case Study

我们将转换过程实现为一个Scala编译插件，验证RISC-V处理器的整数乘除法器

Chisel 设计算法

- X-divider：移位相减除法
- R-divider：移位相减除法
- X-multiplier：booth编码Wallace-tree乘法
- R-multiplier：移位相加乘法

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×)

In kami (a similar verification), the increase is > 11 everywhere. Then we have less verification efforts