CS110 Discussion

Introduction of Spinal HDL

Yuxuan Li

VSPLab@SIST

2026.04.24

liyx22025@shanghaitech.edu.cn

目录

1 Spinal HDL 是什么

Spinal HDL 是什么

- Spinal HDL 是一种高级硬件描述语言，相较 Verilog 更为优雅高效。
- Spinal HDL 本质上是 Scala 的一个库（Scala 是一个通用的面向对象语言，其执行程序需要运行在 Java 虚拟机上），而不是一门独立语言。故使用 Spinal HDL 本质上是在编写一段特殊的 Scala 程序，运行程序会输出对应的 Verilog 文件。
- Spinal HDL 的思想可以概括为用软件工程的方式构造硬件。所有的硬件信号都是某种 Scala 对象，当你使用诸如 `:=` 或 `+` 这样的“运算符”以及诸如 `when` 或 `switch` 的“关键字”对信号进行“赋值”时，本质上是在调用某些 Scala 函数在硬件信号间创造逻辑。运行这段程序，就是将这种逻辑以 Verilog 的形式输出。
- Spinal HDL 和主流 EDA 工具兼容，因为其最终仍然会产生 Verilog 文件。
- Spinal HDL 不是一种高层次综合（HLS），它只是以更好的方式描述硬件。

SpinalHDL 是什么

The goal with it is to stop playing with wires and gates, to start taking some distance with that low level stuff, to think reusable.

SpinalHDL 的文档

推荐直接阅读英文版文档

- 官方英文：https://spinalhdl.github.io/SpinalDoc-RTD/master
- 官方中文：https://spinalhdl.github.io/SpinalDoc-RTD/zh_CN/master
- 清华中文：https://thucgra.github.io/SpinalHDL_Chinese_Doc
- 模板：https://github.com/SpinalHDL/SpinalTemplateSbt

Verilog 存在什么问题

- Verilog 是一门非常糟糕的语言，从语法到架构充斥着大量蹩脚的设计。
- Verilog 并不会让我们距离硬件更近，相反，用 Spinal 描述硬件是更自然的。
- 硬件设计的复杂性很多时候并非来自硬件本身，而是来自 Verilog 这门语言！

组合逻辑与时序逻辑被迫切割

- 组合逻辑使用 `wire` 类型，在 `assign` 语句中使用 `=` 进行连接。
- 时序逻辑使用 `reg` 类型，在 `always` 块中使用 `<=` 进行连接。
- 事件敏感的范式对于描述硬件毫无意义！
- 既然已经通过类型区分了行为，为何又要通过两种截然不同的方式进行连接？
- 寄存器的复位和更新必须要置于同一个 `always` 块并用 `if(~rst_n)` 区分。
- 使用 `<=` 是非常糟糕的运算符选择！
- 阻塞赋值搭配 `always @(*)` 块会令 `reg` 类型会产生组合逻辑，加剧混乱。

组合逻辑与时序逻辑被迫切割

代码 1: 组合逻辑与时序逻辑被迫切割

```
1  reg[3:0] cnt;
2  wire[3:0] inv;
3  assign inv = ~cnt;
4  always @(posedge clk or negedge rst_n) begin
5      if(~rst_n)
6          cnt <= 4'b0;
7      else
8          cnt <= cnt + 4'b1;
9  end
10 assign out = inv[3];
```

模块必须在实例化时完成连接

- 模块必须在实例化时立即进行连接，导致需要定义大量中间信号。
- 模块对接口的封装是失败的，因为大部分接口信号都要在其外部重复定义。
- 无法直接连接两个模块的接口信号！
- 正确的方案：先实例化 `fa0` 和 `fa1`，随后直接连接 `fa0.cout` 和 `fa1.cin`。
- 实例化语法中的符号选择 `.a(a[0])` 非常糟糕，相较于 `a=a[0]` 很不清晰。

模块必须在实例化时完成连接

代码 2: 模块必须在实例化时完成连接

```
1 wire fa0_cout;
2 wire fa1_cin;
3 full_adder fa0 (
4     .a(a[0]), .b(b[0]), .cin(1'b0),
5     .sum(sum[0]), .cout(fa0_cout));
6 full_adder fa1 (
7     .a(a[1]), .b(b[1]), .cin(fa1_cin),
8     .sum(sum[1]), .cout(cout));
9 assign fa1_cin = fa0_cout;
```

功能极为有限的硬件生成语句

当我们在 Verilog 中使用循环，这本质上是在程序化的生成硬件

- 有些代码是在描述硬件（硬件代码，综合后会保留）。
- 有些代码是生成硬件的程序（软件代码，综合过程中执行）。

功能极为有限的硬件生成语句

- 生成语句需要置于 `generate` 环境中，使用的变量是特殊的 `genvar` 类型，
- 生成语句只能使用 `for` 循环和 `if` 分支等有限的几个语句。
- 软件代码和硬件代码之间的语法相互纠缠，界限不清晰，容易混淆。
- 例如 `if` 可以同时用于生成语句中的分支和描述一个硬件 MUX。
- 例如 `genvar` 与 `wire` 和 `reg` 的使用方式一致，但意义完全不同。
- 涉及到模块时，需要提前定义中间信号会进一步限制代码编写。

功能极为有限的硬件生成语句

代码 3: 功能极为有限的硬件生成语句

```
genvar i;
generate
    for (i = 0; i < 4; i = i + 1) begin : gen_struct
        if (i % 2 == 0)
            assign c[i] = a[i] & b[i];
        else
            assign c[i] = a[i] | b[i];
    end
endgenerate
```

缺乏对信号结构体的抽象能力

- 没有结构体的概念（模块对应函数），无法将一系列特定位宽信号封装在一起。
- 总线信号若需要沿着模块接口传播，则所有接口都要完整列出其中的每个信号。
- 总线信号中，部分为输入，部分为输出，且会随着 `master/slave` 对应切换。

缺乏对信号结构体的抽象能力

代码 4: 缺乏对信号结构体的抽象能力

```
module ahb_lite_master (
    input wire[31:0]  HRDATA,
    output wire[31:0] HWDATA,
    output wire[31:0] HADDR,
    //...
);
//...
endmodule
```

2 Spinal HDL 的安装

Spinal HDL 的安装

Spinal HDL 的安装可以大致分为以下两个部分

- 软件环境：安装 Java 环境和 Scala 环境（`openjdk` 和 `coursier`）。
- 硬件环境：安装 Verilator 仿真器和 GtkWave 波形查看器（`oss-cad-suite`）。

Spinal HDL 库会在第一次编译相关 Scala 程序时自动被下载。

Spinal HDL 的开发需要在 Linux 系统（发行版推荐 Ubuntu 24.04）上进行。

安装 Java 和 Scala 环境

安装一些基础工具（如有可以忽略）

```
sudo apt install curl git
```

安装 Java 环境

```
sudo apt install openjdk-17-jdk-headless
sudo apt install openjdk-21-jdk-headless
```

文档推荐的是 `openjdk-17`。不过，由于 Ubuntu 24.04 自身带有一个不完整但被作为默认的 `openjdk-21`，会导致 Spinal HDL 仿真出现和 `<jni.h>` 相关的错误。比较简单的解决方案是完整安装 `openjdk-21`，测试表明高版本不会带来兼容性问题。

安装 Java 和 Scala 环境

安装 Scala 环境，这需要下载 coursier，这是一个用户级安装脚本

```
curl -fL https://github.com/coursier/launchers/raw/master/cs-x86_64-pc-
    linux.gz | gzip -d > cs
chmod +x cs
./cs setup
```

该安装脚本会将 coursier 及相关工具安装至 ~/.local/share/coursier 中。

该安装脚本会自动将其路径添加到 $PATH 中，不需要手动处理。

安装 Verilator 仿真器和 GtkWave 波形查看器

安装一些基础工具（如有可以忽略）

```
sudo apt install make gcc g++ zlib1g-dev
```

安装 Verilator 和 GtkWave，这些工具都已集成在 oss-cad-suite

```
curl -fLO https://github.com/YosysHQ/oss-cad-suite-build/releases/
    download/2025-03-03/oss-cad-suite-linux-x64-20250303.tgz
tar -xzf oss-cad-suite-linux-x64-20250303.tgz
sudo mv oss-cad-suite /opt
```

安装 Verilator 仿真器和 GtkWave 波形查看器

编辑配置文件（分别是适用于 Bash 和 Zsh 的配置文件）

```
vim ~/.profile
vim ~/.zprofile
```

添加以下内容，使 `/opt/oss-cad-suite/bin` 位于搜索路径上

```
export PATH="$PATH:/opt/oss-cad-suite/bin"
```

重启或重新登录后生效。

VSCode 插件

SpinalHDL 开发可以直接使用 VSCode，需要安装以下两个插件

- Scala (Metals)
- Scala Syntax (official)

安装前一个会自动安装后一个插件。

随后可以直接在 VSCode 的图形界面中完成 Scala 程序的编译和运行。

3 Spinal HDL 的语法

Scala 的快速入门

- Scala 用 `val` 代表不可变类型，用 `var` 代表可变类型，Spinal 总是用 `val`。
- Scala 有 `class`、`case class`、`object` 三个和类有关的关键词。
- Scala 的 `class` 的成员默认都是公有的，若需保护或私有要特别声明。
- Scala 的 `class` 没有静态的概念，取而代之的是 `object`，成员全是静态的。
- Scala 的 `class` 在实例化的时候要使用 `new`，不过这和内存分配无关。
- Scala 的 `case class` 和 `class` 基本相同，但主要有两个区别：实例化时不需要 `new`，构造函数的参数自动作为公有成员。Spinal 中总是用 `case class`。
- Scala 的构造参数直接写在类声明上，类的大括号内就是构造函数的函数体。
- Scala 使用 `()` 而不是 `[]` 进行列表的索引。

Spinal HDL 的用法

- Spinal 模块总是以 `case class Foo() extends Component` 开头。
- Spinal 模块的类定义中，一般来说只有构造函数，没有更多成员函数。
- Spinal 模块的 IO 以 `val io = new Bundle` 开头，用 `in()` 和 `out()` 指定方向。
- 信号类型：`Bool()`，`Bits(x bits)`，`UInt(x bits)`，`SInt(x bits)`。
- 信号连接使用 `:=`，这是一个巧妙的运算符重载，本质是在调用信号对象的方法。
- 使用 `Reg()` 修饰信号代表其是一个寄存器，后接 `init()` 设定复位值。
- Spinal 中，时钟信号和复位信号会在需要的情况下自动添加。
- 使用 `when {} otherwise {}` 会产生硬件 MUX，这并不是 Scala 关键字！
- 使用 Scala 的 `for ()` 和 `if {} else {}` 可以很轻松的编写生成语句。

代码 5: 计数器

```
1 case class Counter() extends Component {
2   val io = new Bundle {
3     val output = out(Bool())
4   }
5   val cnt = Reg(UInt(4 bits)).init(0)
6   val inv = ~cnt
7   cnt := cnt + 1
8   io.output := inv(3)
9 }
```

代码 6: 多路选择器

```
1 case class Mux21(width: Int) extends Component {
2   val io = new Bundle {
3     val a = in(Bits(width bits))
4     val b = in(Bits(width bits))
5     val sel = in(Bool())
6     val v = out(Bits(width bits))
7   }
8   when(io.sel) { io.v := io.a } otherwise { io.v := io.b }
9 }
```

代码 7: 生成语句

```
case class BitAndOr(width: Int) extends Component {
  val io = new Bundle {
    val a = in(Bits(width bits))
    val b = in(Bits(width bits))
    val c = out(Bits(width bits))
  }
  for (i <- 0 until width) {
    if (i % 2 == 0) io.c(i) := io.a(i) & io.b(i)
    else io.c(i) := io.a(i) | io.b(i)
  }
}
```

数据类型：Bool

Spinal HDL 使用 `Bool` 表示一位的硬件信号，其字面值是 `True` 和 `False`

```
1  val b0 = Bool()
2  val b1 = Bool()
3  b0 := True
4  b1 := False
```

其中，`val` 是 Scala 中用于创建常量的关键字。所有 Spinal HDL 的信号都应该使用常量 `val` 而不是变量 `var` 来创建。`=` 是 Scala 的赋值运算符，`:=` 是 Spinal HDL 重载的用于硬件信号连接的运算符，`b0 := True` 和 `b1 := False` 并不是对 `b0` 和 `b1` 赋值，而是在调用硬件信号的成员函数，设定其连接关系。请务必区分这两者！

数据类型：Bool

Spinal HDL 中，有时可以在创建硬件信号时直接给出其连接（以下两种写法等价）

```
1  val b2 = b0 && b1
```

```
1  val b3 = Bool()
2  b3 := b0 && b1
```

但本质上来说，这仍然是先创建了 `Bool()` 对象再用 `:=` 与 `b0 & b1` 连接。

数据类型：Bool

Spinal HDL 在多数情况下，可以灵活转换 Scala 的数据类型

```
1  val b4 = true
```

- `Boolean` 是 Scala 的软件数据类型，字面值用 `true` 和 `false` 表示。
- `Bool` 是 Spinal HDL 的硬件信号类型，字面值用 `True` 和 `False` 表示。

请正确区分这两者：前者是生成硬件的程序中的变量，后者是硬件信号对象。

数据类型：Bits

Spinal HDL 使用 Bits 表示多位的硬件信号，其位宽设置和字面值表示都很方便

```
1  val bits0 = Bits(8 bits)
2  val bits1 = B(42, 8 bits)
3  val bits2 = B("01011010")
```

Bits 类型可以做逻辑运算，但不能做算术运算

```
1  bits0 := bits1 & bits2
```

数据类型：Bits

Spinal HDL 中 Bits 的索引和片选将分别返回 Bool 和 Bits

```
1  val xa = bits0(0)
2  val xb = bits0(0 to 3)
3  val xc = bits0(0 until 4)
4  val xd = bits0(3 downto 0)
```

其中，0 to 3 和 0 until 4 的差异在于右侧是闭区间还是开区间，3 downto 0 则是比较符合 Verilog 习惯的 3:0 的写法，它和 0 to 3 的效果其实是完全相同的。

Scala 和许多语言不同，使用 () 而不是 [] 进行数组索引。

数据类型：UInt SInt

Spinal HDL 中 UInt 和 SInt 是带有无符号和有符号假设的 Bits

```
1  val uint = UInt(8 bits)
2  val sint = SInt(8 bits)
```

- 创建对象用 Bits()，表示字面值用 B()
- 创建对象用 UInt()，表示字面值用 U()
- 创建对象用 SInt()，表示字面值用 S()

在 Verilog 层面，UInt 与 SInt 和 Bits 没有本质区别。

数据类型：UInt SInt

Spinal HDL 中只有使用 UInt 或 SInt 才能做算术运算（对于加法，两者是一致的）

```
1  val a = U(5, 4 bits)
2  val b = U(7, 4 bits)
3  val sum = UInt(4 bits)
4  sum := a + b
```

两者的区别主要表现在：右移、比较、乘法、除法

```
1  val u = U("11110000") |>> 2 // u = 00111100
2  val s = S("11110000") |>> 2 // s = 11111100
```

数据类型：SpinalEnum

Spinal HDL 中 SpinalEnum 用于定义信号枚举（类似于 C 的 enum）

```
1 object AluOp extends SpinalEnum {
2   val add, sub, xor, and, slt = newElement()
3 }
```

适合用于不关心具体编码的控制信号

```
1   val aluOpEnum = AluOp()
2   val aluOpBits = Bits(3 bits)
3   aluOpEnum := AluOp.slt
4   aluOpBits := B("100")
```

数据类型：Bundle

Spinal HDL 中 Bundle 用于定义信号簇（类似于 C 的 struct）

```
case class MyData(width: Int) extends Bundle {
  val data = Bits(width bits)
  val sel = Bool()
}
```

适合用于封装接口信号，使用 . 访问内部信号

```
  val a = MyData(32)
  a.data := B(255, 32 bits)
  a.sel := True
```

寄存器：Reg

Spinal HDL 中，寄存器需要用 Reg 修饰，使用 .init() 设置复位值

```
1  val pcReg = Reg(Bits(32 bits)).init(0)
2  val pcPlus4 = Bits(32 bits)
3  pcPlus4 := (pcReg.asUInt + 4).asBits
4  pcReg := pcPlus4
```

这一点和 Verilog 非常不同！首先，不需要手动引入时钟 clock 和复位 rst，若一个模块包含寄存器，这些信号会自动被接入。其次，寄存器的更新也会使用 :=，这里做的就是在寄存器的输入端连接了一个信号，寄存器的状态在 clock 上升沿更新是寄存器的性质。最后，寄存器的复位值用 .init() 设置，其在 rst 高电平时复位。

存储器：Mem

Spinal HDL 中，存储器有专门的 Mem 建模，而不是使用 Reg 数组

```
1  val dataMem = Mem(Bits(32 bits), 0x1000).init(Seq.fill(0x1000)(0))
2  val addr = Bits(32 bits)
3  val dataRd = Bits(32 bits)
4  val dataWr = Bits(32 bits)
5  dataRd := dataMem(addr(2 until 14).asUInt)
6  dataMem(addr(2 until 14).asUInt) := dataWr
```

这里需要注意，访问 Mem 时，地址一定要位宽正确且需转换为 UInt 类型。

逻辑语句：when ... elsewhen ... otherwise

Spinal HDL 中，硬件的条件语句可以使用 `when...elsewhen...otherwise`

```
1  when(rs1 === B(0, 5 bits)) {
2    rs1Data := 0
3  } otherwise {
4    rs1Data := regs(rs1.asUInt)
5  }
6
7  when(rd =/= B(0, 5 bits)) {
8    regs(rd.asUInt) := rd
9  }
```

逻辑语句：switch ... is ... default

Spinal HDL 中，硬件的条件语句可以使用 `switch...is...default`

```
1  switch(aluOp) {
2    is(AluOp.add) { result := (srca.asUInt + srcb.asUInt).asBits }
3    is(AluOp.and) { result := (srca & srcb) }
4    is(AluOp.xor) { result := (srca ^ srcb) }
5    is(AluOp.slt) { result := (srca.asSInt < srcb.asSInt).asBits }
6    default { result := 0 }
7  }
```

定义模块：Module

Spinal HDL 中，模块总是一个继承自 Component 的 case class，模块的接口信号的方向用 in() 和 out() 包裹，并一般放在一个称为 io 的 Bundle 中以便检查。

```
1  case class Alu() extends Component {
2    val io = new Bundle {
3      val srca = in(Bits(32 bits))
4      val srcb = in(Bits(32 bits))
5      val aluResult = out(Bits(32 bits))
6      val aluOp = in(AluOp())
7    }
8    // ...
9  }
```

定义模块：Module

Spinal HDL 中，有时需要接口上的 Bundle，部分信号为输入，部分信号为输出

```
1  case class Memory() extends Component {
2    val io = new Bundle {
3      val aluResult = in(Bits(32 bits))
4      val rs2Data = in(Bits(32 bits))
5      val memWrite = in(Bool())
6      val memResult = out(Bits(32 bits))
7      val dmem = master(PortDmem())
8    }
9    // ...
10 }
```

定义模块：Moudle

这种情况需要一个带有 IMasterSlave 的 Bundle，通过重载 asMaster() 来确定作为 master 的输入输出定义，随后就可以使用 master() 和 slave() 进行修饰了

```
1  case class PortDmem() extends Bundle with IMasterSlave {
2    val addr = Bits(32 bits)
3    val dataRd = Bits(32 bits)
4    val dataWr = Bits(32 bits)
5    val enableWr = Bool()
6    def asMaster(): Unit = {
7      in(dataRd)
8      out(addr, dataWr, enableWr)
9    }
10 }
```

4 完整的例子

完整的例子：Spinal HDL 的引入

以下代码应当放置在每个 Scala 文件的最前面，关键字 `package` 指定项目名称，这应当与项目的目录名称保持一致，关键字 `import` 将会引入 Spinal HDL 的相关库。

```
1 package SpinalSulfur
2
3 import spinal.core._
4 import spinal.core.sim._
5 import spinal.lib._
```

完整的例子：Spinal HDL 的配置

以下代码用于配置代码的输出目录（对于每个工程）

```
object Config {
  def spinal = SpinalConfig(
    targetDirectory = "build/hw",
    defaultConfigForClockDomains =
      ClockDomainConfig(resetActiveLevel = HIGH),
    onlyStdLogicVectorAtTopLevelIo = false
  )
  def sim = SimConfig.withConfig(spinal)
    .workspacePath("build/sim").withFstWave
}
```

完整的例子：模块

```
1  case class Divider() extends Component {
2    val io = new Bundle {
3      val sdiv = in(UInt(32 bits))
4      val sclk = out(Bool())
5    }
6    val scnt = Reg(UInt(32 bits)).init(0)
7    val sclk = Reg(Bool()).init(False)
8    val flip = (scnt === ((io.sdiv |>> 1) - 1))
9    when(flip) { scnt := 0 } otherwise { scnt := scnt + 1 }
10   when(flip) { sclk := ~sclk }
11   io.sclk := sclk
12 }
```

完整的例子：生成

通过 `Config.spinal.generateVerilog` 生成 Verilog 代码。

```scala
object DividerVerilog extends App {
  Config.spinal.generateVerilog(Divider())
}
```

```scala
object DividerVhdl extends App {
  Config.spinal.generateVhdl(Divider())
}
```

在 Scala 中，类似于 C 语言 `main()` 函数的程序入口可以有很多个，每个入口应当是一个继承自 `App` 的 `object`。在运行项目时通过 `sbt` 命令决定使用哪一入口。

完整的例子：仿真

通过 Config.sim.compile 运行仿真，你会得到一个模块的实例 dut。通过 #= 进行信号激励，通过 dut.clockDomain.waitRisingEdge() 跳至下一时钟上升沿。

```
1  object DividerSim extends App {
2    Config.sim.compile(Divider()).doSim { dut =>
3      dut.clockDomain.forkStimulus(period = 10, resetCycles = 9)
4      dut.clockDomain.waitRisingEdge()
5      dut.io.sdiv #= 20
6      for (i <- 0 until 100) { dut.clockDomain.waitRisingEdge() }
7      dut.io.sdiv #= 6
8      for (i <- 0 until 100) { dut.clockDomain.waitRisingEdge() }
9    }
10 }
```

完整的例子：查看结果

使用 sbt 运行程序（需要 build.sbt，模板有）

```
sbt "runMain DividerVerilog"
sbt "runMain DividerVhdl"
sbt "runMain DividerSim"
```

使用 gtkwave 查看波形

```
gtkwave build/sim/Divider/test/wave.fst
```

https://github.com/liyuxuan3003/SpinalSulfurTemplate

谢谢大家!