SpinalHDL Documentation

SpinalHDL contributors

2024 年 12 月 24 日

Contents

欢迎来到 SpinalHDL 的文档!

SpinalHDL 是一种开源高级硬件描述语言。它可以用作 VHDL 或 Verilog 的替代品，并且比它们有几个优点:

- 它专注于高效的硬件描述而不是事件驱动。
- 它被嵌入到通用编程语言中，从而具备强大的硬件逻辑生成能力。

关于该语言更详细的介绍请参见关于 *SpinalHDL*

本文档的 HTML 和 PDF 格式可在线获取:
> spinalhdl.github.io/SpinalDoc-RTD
(可从左下角访问 PDF 格式文档，点击 `v:master` ，然后点击 PDF)
中文版文档:
> github.com/thuCGRA/SpinalHDL_Chinese_Doc
您还可以在这里找到 API 文档:
> spinalhdl.github.io/SpinalHDL

CHAPTER 1

前言

初步说明：

- 以下所有陈述都是关于描述数字硬件电路的。验证是另一个有趣的话题。
- 为了简洁，我们假设 SystemVerilog 是 Verilog 的最新版本。
- When reading this, we should not underestimate how much our attachment for our favorite HDL will bias our judgement.

1.1 为什么要放弃传统的 HDL

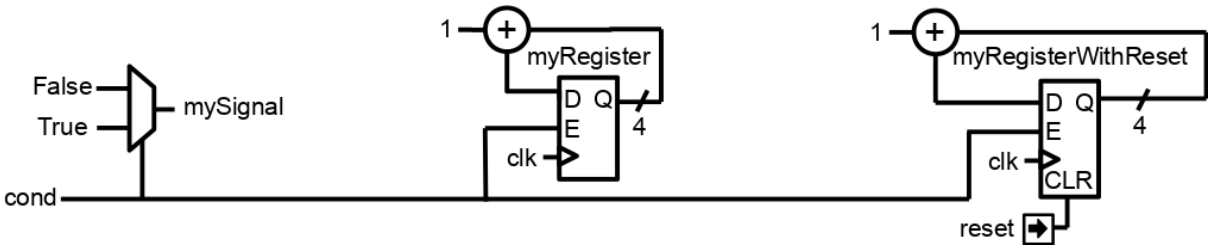
1.1.1 VHDL/Verilog 不是硬件描述语言

这些语言最初是为了模拟/文档目的而创建的事件驱动语言。只有在后来，它们才被用作综合工具的输入语言。这就解释了以下许多观点的根源。

1.1.2 事件驱动范式对于 RTL 没有任何意义

仔细想想，使用 process/always 块描述数字硬件 (RTL) 没有任何实际意义。为什么我们必须担心敏感列表？为什么我们必须在不同性质的进程 (process)/always 块之间分割我们的设计（组合逻辑/不带复位的寄存器/带异步复位的寄存器）？

例如，要实现这个：



使用 VHDL 流程，您可以编写以下内容：

```
signal mySignal : std_logic;
signal myRegister : unsigned(3 downto 0);
signal myRegisterWithReset : unsigned(3 downto 0);

process(cond)
begin
    mySignal <= '0';
    if cond = '1' then
        mySignal <= '1';
    end if;
end process;

process(clk)
begin
    if rising_edge(clk) then
        if cond = '1' then
            myRegister <= myRegister + 1;
        end if;
    end if;
end process;

process(clk,reset)
begin
    if reset = '1' then
        myRegisterWithReset <= 0;
    elsif rising_edge(clk) then
        if cond = '1' then
            myRegisterWithReset <= myRegisterWithReset + 1;
        end if;
    end if;
end process;
```

使用 SpinalHDL 你可以这样写：

```
val mySignal             = Bool()
val myRegister           = Reg(UInt(4 bits))
val myRegisterWithReset  = Reg(UInt(4 bits)) init(0)

mySignal := False
when(cond) {
    mySignal             := True
    myRegister           := myRegister + 1
    myRegisterWithReset := myRegisterWithReset + 1
}
```

所有事情都是这样，您可以习惯这种事件驱动的语义，直到您尝试更好的事物。

1.1.3 VHDL 和 Verilog 的最新版本不可用

EDA 行业在其工具中实现 VHDL 2008 和 SystemVerilog 综合功能的速度确实很慢。此外，当它完成时，似乎只实现了该语言的一个约束子集（不谈论仿真功能）。结果是使用这些语言修订版的任何有趣功能都不安全，因为：

- 它可能会使您的代码与许多 EDA 工具不兼容。
- 其他公司可能不会接受您的 IP，因为他们的流程尚未准备好。

无论如何，这些修订并没有改变 HDL 问题的核心：它们基于事件驱动范的范式，这对于描述数字硬件没有意义。

1.1.4 VHDL 结构记录（record），Verilog 结构（struct）已经破碎（SystemVerilog 在这方面很好，如果您可以使用它）

您不能使用它们来定义接口，因为您无法定义它们的内部信号方向。更糟糕的是，您无法向他们提供构造参数！因此，只能一次性定义好 RGB 记录/结构，但愿您永远不必将其与不同大小的颜色通道一起使用……

VHDL 的另一个奇特之处是，如果您想将某个数组添加到组件实体中，则必须将该数组的类型定义到包中…这就不能参数化了…

例如，下面是 SpinalHDL APB3 总线定义：

```
// Class which can be instantiated to represent a given APB3 configuration
case class Apb3Config(
  addressWidth  : Int,
  dataWidth     : Int,
  selWidth      : Int     = 1,
  useSlaveError : Boolean = true
)

// Class which can be instantiated to represent a given hardware APB3 bus
case class Apb3(config: Apb3Config) extends Bundle with IMasterSlave {
  val PADDR      = UInt(config.addressWidth bits)
  val PSEL       = Bits(config.selWidth bits)
  val PENABLE    = Bool()
  val PREADY     = Bool()
  val PWRITE     = Bool()
  val PWDATA     = Bits(config.dataWidth bits)
  val PRDATA     = Bits(config.dataWidth bits)
  val PSLVERROR  = if(config.useSlaveError) Bool() else null  // Optional signal

  // Can be used to setup a given APB3 bus into a master interface of the host␣
↪component
  // `asSlave` is automatically implemented by symmetry
  override def asMaster(): Unit = {
    out(PADDR, PSEL, PENABLE, PWRITE, PWDATA)
    in(PREADY, PRDATA)
    if(config.useSlaveError) in(PSLVERROR)
  }
}
```

然后 VHDL 2008 有部分的解决方案和 SystemVerilog 接口/modport 也能有所帮助，如果您的 EDA 工具/公司流程/公司政策允许您使用它们，那么您很幸运。

1.1.5 VHDL 和 Verilog 太冗长了

对于 VHDL 和 Verilog，当它开始涉及组件实例化互连时，必须使用 Ctrl-V/C 大法。

要更深入地理解它，下面是一个使用 SpinalHDL 实例化一些外设并添加用于访问它们所需的 APB3 解码器的示例。

```
// Instantiate an AXI4 to APB3 bridge
val apbBridge = Axi4ToApb3Bridge(
  addressWidth = 20,
  dataWidth    = 32,
  idWidth      = 4
)

// Instantiate some APB3 peripherals
val gpioACtrl = Apb3Gpio(gpioWidth = 32)
val gpioBCtrl = Apb3Gpio(gpioWidth = 32)
```

(续下页)

(接上页)

```
val timerCtrl = PinsecTimerCtrl()
val uartCtrl = Apb3UartCtrl(uartCtrlConfig)
val vgaCtrl = Axi4VgaCtrl(vgaCtrlConfig)

// Instantiate an APB3 decoder
// - Driven by the apbBridge
// - Map each peripheral in a memory region
val apbDecoder = Apb3Decoder(
  master = apbBridge.io.apb,
  slaves = List(
    gpioACtrl.io.apb -> (0x00000, 4 KiB),
    gpioBCtrl.io.apb -> (0x01000, 4 KiB),
    uartCtrl.io.apb  -> (0x10000, 4 KiB),
    timerCtrl.io.apb -> (0x20000, 4 KiB),
    vgaCtrl.io.apb   -> (0x30000, 4 KiB)
  )
)
```

完成。这就是所有内容。在实例化模块/组件时，你不必一个接一个地绑定信号，因为你可以以面向对象的方式访问它们的接口。

另外，关于 VHDL/Verilog 结构/记录，可以说它们确实是无用的，没有真正的参数化和可重用性功能，仅仅是试图掩盖这些语言设计不佳的事实。

1.1.6 元硬件描述能力

VHDL 和 Verilog 提供了一些实例细化工具，这些工具不会直接映射到硬件中，如循环/生成语句/宏/函数/过程/任务。但仅此而已。

即便如此，它们的作用也确实有限。例如，不能将进程/always 块/组件/模块块定义到任务/过程中。这确实是许多高级功能的瓶颈。

使用 SpinalHDL，您可以在总线上调用用户定义的任务/过程，如下所示：`myHandshakeBus.queue(depth=64)`。下面是一些包含定义的代码。

```
// Define the concept of handshake bus
class Stream[T <: Data](dataType:  T) extends Bundle {
  val valid   = Bool()
  val ready   = Bool()
  val payload = cloneOf(dataType)

  // Define an operator to connect the left operand (this) to the right operand␣
↪(that)
  def >>(that: Stream[T]): Unit = {
    that.valid := this.valid
    this.ready := that.ready
    that.payload := this.payload
  }

  // Return a Stream connected to this via a FIFO of depth elements
  def queue(depth: Int): Stream[T] = {
    val fifo = new StreamFifo(dataType, depth)
    this >> fifo.io.push
    return fifo.io.pop
  }
}
```

让我们进一步想象，假设你想定义一个有限状态机。使用 VHDL/Verilog，你需要编写大量原始代码，并使用一些 switch 语句来实现它。你不能定义”StateMachine”的概念，这将为你提供一个很好的语法来定义每个状态。否则，你可以使用第三方工具来绘制你的有限状态机，然后生成等价的 VHDL/Verilog 代码 …

SpinalHDL 的元硬件描述能力使您能够定义自己的工具，然后允许您以抽象方式定义事物，例如有限状态机。

下面是 SpinalHDL 上有限状态机的一个简单的用法示例：

```
// Define a new state machine
val fsm = new StateMachine {
  // Define all states
  val stateA, stateB, stateC = new State

  // Set the entry point
  setEntry(stateA)

  // Define a register used into the state machine
  val counter = Reg(UInt(8 bits)) init (0)

  // Define the state machine behavior for each state
  stateA.whenIsActive (goto(stateB))

  stateB.onEntry(counter := 0)
  stateB.onExit(io.result := True)
  stateB.whenIsActive {
    counter := counter + 1
    when(counter === 4) {
      goto(stateC)
    }
  }

  stateC.whenIsActive(goto(stateA))
}
```

假设你想生成 CPU 的指令解码逻辑。这可能需要一些复杂的实例细化时算法来生成尽可能少的逻辑。但是，在 VHDL/Verilog 中，你唯一的选择是用脚本生成你想要的 `.vhd` 和 `.v` 文件。

关于元硬件描述确实有很多话要说，但理解它并得其真味的唯一方法就是进行实验。它的目标是停止直接使用电线和门，与那些抽象层级较低的东西保持一定的距离，并思考可重用的方法。

CHAPTER 2

简介

本节介绍 SpinalHDL 项目：该语言以及与之相关的所有内容。

2.1 关于 SpinalHDL

2.1.1 什么是 SpinalHDL？

SpinalHDL 是一种开源高级硬件描述语言及其相关工具。它的开发始于 2014 年 12 月。

SpinalHDL 通过对数字硬件命名和建模使得高效地描述硬件成为可能；最明显的例子是 `Reg` 和 `Latch`。在 VHDL 和 Verilog 等事件驱动语言中，要使用这两个常见元素，用户必须用过程来模拟它们，以便综合工具可以推断出它是什么单元。使用 SpinalHDL，您只需声明一个 `Reg` 或 `Latch`。

SpinalHDL 是一种基于通用语言 Scala 的 *领域专用语言*。它带来了几个好处：

- 有免费的集成开发环境支持它，提供了许多简单文本编辑器所没有的功能：
  - 语法和类型错误在代码中高亮显示
  - 正确的重命名，甚至是跨文件的
  - 智能自动完成/建议
  - 导航工具（转到定义、显示所有引用等）
- 它允许实现简单到复杂的硬件生成器（元硬件描述），而无需处理多种语言（译者注：例如 Python+Verilog）。

备注： Scala 是一种使用 Java 虚拟机 (JVM) 的静态类型、函数式和面向对象的语言。

2.1.2 SpinalHDL 不是什么

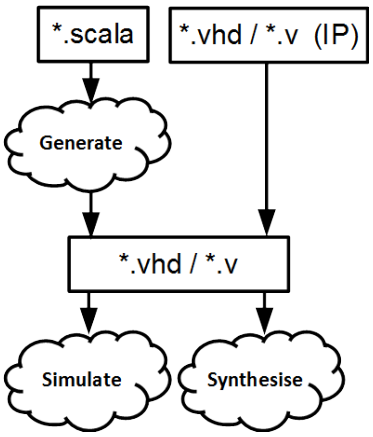
SpinalHDL 不是 HLS 工具：它的目标不是自动将抽象算法转换为数字电路。它的目的是借助命名事物来创建新的高层次抽象，以帮助设计人员重用他们的代码，而不是一遍又一遍地编写相同的事物。

SpinalHDL 不是模拟电路建模语言。VHDL 和 Verilog 使模拟电路设计人员能够向数字设计人员提供其 IP 模型。SpinalHDL 不解决这种情况，而是供数字设计师描述他们自己的数字设计。

2.1.3 Spinal 开发流程

一旦用 *SpinalHDL* 编写代码，该工具就可以：

- 生成 VHDL、Verilog 或 SystemVerilog，以这些语言之一实例化它或将其提供给任何仿真器或综合工具。没有逻辑开销，保留层次结构和名称，并且在生成期间运行设计检查。
- 使用 Verilator 或其他支持的仿真器进行仿真。



由于 SpinalHDL 可以与 VHDL 和（System）Verilog 互操作，因此您既可以用这些语言实例化 SpinalHDL IP（使用生成的代码），也可以用 SpinalHDL 实例化这些语言的 IP（使用 `BlackBox`）。

备注： SpinalHDL 与基于标准 VHDL/Verilog 的 EDA 工具（仿真器和综合器）* 完全可互操作 *，因为工具链生成的输出是 VHDL 或 Verilog。

2.1.4 SpinalHDL 相对于 VHDL / Verilog 的优势

由于 SpinalHDL 基于高级语言，因此它提供了多种优势来改进您的硬件编码：

1. **不再需要无休止的布线** - 一行代码完成复杂总线（例如 AXI）的创建和连接。
2. **不断发展的功能** - 创建您自己的总线定义和抽象层。
3. **减少代码大小** - 大幅减少，特别是对于接线而言。这使您能够更好地了解代码库、提高工作效率并减少麻烦。
4. **免费且用户友好的 IDE** - 得益于 Scala 工具的自动完成、错误突出显示、导航快捷方式等。
5. **功能强大且简单的类型转换** - 任何数据类型和位向量之间的双向转换。从 CPU 接口加载复杂的数据结构时非常有用。
6. **设计检查** - 早期阶段检查是否存在组合循环/锁存器。
7. **时钟域安全** - 早期 lint 通知是否存在无意识引入的违例跨时钟域信号。
8. **通用设计** - 使用 Scala 构造，因此对硬件描述的通用性没有任何限制。

2.2 一个简单的例子

以下是来自 入门 仓库的简单硬件描述。

```
case class MyTopLevel() extends Component {
  val io = new Bundle {
    val cond0 = in port Bool()
    val cond1 = in port Bool()
    val flag  = out port Bool()
    val state = out port UInt(8 bits)
  }

  val counter = Reg(UInt(8 bits)) init 0

  when(io.cond0) {
    counter := counter + 1
  }

  io.state := counter
  io.flag  := (counter === 0) | io.cond1
}
```

它在本节中被分成多个块进行解释。

2.2.1 Component

首先，这里有一个 SpinalHDL `Component` 的结构。

组件是一段逻辑，可以根据需要多次实例化（粘贴），外部仅可通过输入和输出信号（io）对其访问。

```
case class MyTopLevel() extends Component {
  val io = new Bundle {
    // port definitions go here
  }

  // component logic goes here
}
```

`MyTopLevel` 是组件的名称。

在 SpinalHDL 中，组件使用 `UpperCamelCase` （驼峰命名法）。

备注： 请参阅组件 了解更多信息。

2.2.2 端口

然后，定义端口。

```
val cond0 = in port Bool()
val cond1 = in port Bool()
val flag = out port Bool()
val state = out port UInt(8 bits)
```

方向：

- `cond0` 和 `cond1` 是输入端口
- `flag` 和 `state` 是输出端口

类型：

- cond0、cond1 和 flag 各为一个比特（3 条单独的线）
- state 是一个 8 位无符号整数（用一组 8 根线来表示一个无符号整数）

备注： 此语法仅自 SpinalHDL 1.8 起可用，请参阅输入/输出定义 了解旧语法和更多信息。

2.2.3 内部逻辑

最后，还有组件的逻辑：

```
val counter = Reg(UInt(8 bits)) init(0)

when(io.cond0) {
  counter := counter + 1
}

io.state := counter
io.flag := (counter === 0) | io.cond1
```

counter 是一个包含 8 位无符号整数的寄存器，初始值为 0。更改寄存器状态的赋值仅可在下一个时钟采样后回读。

备注： 由于寄存器的存在，时钟和复位两个隐式信号被添加组件中。有关更多信息，请参阅寄存器 和时钟域 。

然后描述一个条件规则：当输入 cond0（位于 “io” 线束中）被置 1 时，counter 加一，否则 counter 保持其值在最后一条规则中设置的值。但是，您可能会说，如果没有给出前置规则呢？对一个简单的 ** 信号 ** 来说，它将成为一个锁存器，并触发一个错误。但这里的 counter 是一个寄存器，所以它有一个默认值，没有其他规则，它就保持初始值。

这里创建了一个多路复用器：counter 寄存器的输入可以是其输出或其输出加一，这取决于 io.cond0 的值。

然后描述无条件规则（赋值）：

- 输出端口 state 连接到寄存器 counter 的输出。
- 输出端口 flag 是输入信号 cond1 与另一信号之间的 or 门的输出，该信号在 counter 信号等于 0 时为真，否则为假。

备注： 有关更多信息，请参阅语义 。

2.3 使用 SpinalHDL 的项目

Note that the following lists are very incomplete.

2.3.1 仓库

- J1Sc 堆栈 CPU
- VexRiscv CPU 和 SoC
- NaxRiscv CPU
- SaxonSoc
- open-rdma
- MicroRV32 SoC
- ...

2.3.2 公司

- 达坦科技，中国
  - RoCE v2 硬件实现
  - WaveBPF (wBPF)：一个“紧耦合多核”eBPF CPU，设计为用于处理内存数据的高吞吐量协处理器（例如网络数据包）。
- 易灵思（FPGA 供应商），中国

  “易灵思已在 FPGA 中使用 VexRISC-V 内核，并应用于全球客户的多种应用。”
- LeafLabs，美国马萨诸塞州

  SpinalHDL 加速神经科学（PDF 幻灯片）
- QsPin，比利时
- Tiempo Secure，法国

  适用于 ASIC 的 SpinalHDL（PDF 幻灯片）
- ...

2.3.3 大学

- 不来梅大学 - 数学与计算机科学学院，德国

  计算机体系结构研究和教育中的 SpinalHDL（PDF 幻灯片）
- 波茨坦大学 - 用于信号处理的嵌入式系统架构，德国

  用于 FPGA 集群的网络附加深度学习加速器（PDF 幻灯片）
- ...

2.4 联系方式

- 有关 SpinalHDL 语法和现场演讲的问题：
  - 英文 Matrix 频道
  - 中文 Matrix 频道
  - Google 群组
- 对于错误报告、功能请求和问题：
  - 开启一个工单

- 如果您对演示、研讨会或咨询感兴趣：
  - 通过电子邮件联系我们：spinalhdl@gmail.com

2.5 许可证

SpinalHDL 使用两种许可证，一种用于 `spinal.core`，一种用于 `spinal.lib` 以及仓库中的其他所有内容。

`spinal.core`（编译器）采用 LGPL 许可证，可概括如下：

- 您可以通过 SpinalHDL 描述及其生成的 RTL 赚钱。
- 您不必共享 SpinalHDL 描述及其生成的 RTL。
- 没有任何费用，也没有特许权使用费。
- 如果您对 SpinalHDL 核心进行改进，您必须分享您的修改以使该工具更好地为所有人服务。

`spinal.lib`（组件/工具/接口的通用库）处于宽松的 MIT 许可证下，因此您不必共享它，若您做出贡献也非常感谢。

2.6 贡献

- 主代码库
- 贡献者指南
- 本文档的存储库
- 捐赠渠道

2.7 常见问题

2.7.1 与人工编写的 VHDL/Verilog 相比，SpinalHDL 生成的 RTL 的开销是多少？

开销为零。SpinalHDL 生成与人类编写的 VHDL/Verilog 中相同的 HDL 构造，在最终实现中不存在额外的实例化工件，这也取决于你的使用方法。这使得在将生成的 HDL 与手写 HDL 进行比较时的开销为零。

由于 Scala/SpinalHDL 语言强大的表达能力，对于给定的复杂硬件设计来说，设计更加简洁，并且具有很强的类型安全性，强大的 HDL 安全范式导致更少的代码行，能够以更少的错误实现更多的功能。

SpinalHDL 不采用 HLS 方法，本身也不是 HLS 解决方案。其目标不是将任意代码转换为 RTL，而是通过提高抽象级别、增加代码重用能力来提供一种功能强大的语言来描述 RTL。

2.7.2 如果 SpinalHDL 将来没有支持了怎么办？

这个问题有两个方面：

1. SpinalHDL 生成 VHDL/Verilog 文件，这意味着 SpinalHDL 仍将在未来几十年内得到所有 EDA 工具的支持。
2. 如果 SpinalHDL 中存在错误并且不再支持修复它，这也不是致命的情况，因为 SpinalHDL 编译器是完全开源的。对于简单的问题，您也许可以在几个小时内自行解决问题。
3. 考虑一下商业 EDA 供应商需要多少时间来解决问题或在其封闭工具中添加新功能。还要考虑使用 SpinalHDL 时节省的成本和时间，以及您自己的实体回馈社区的潜力，其中包括工程时间、开源贡献时间或对项目的捐赠，以改善其未来。

2.7.3 SpinalHDL 是否在生成的 VHDL/Verilog 中保留注释？

事实并非如此。生成的文件应被视为网表。例如，当你编译 C 代码时，你关心生成的汇编代码中的注释吗？

2.7.4 SpinalHDL 可以扩展到大型项目吗？

是的，已经进行了一些实验，如生成数百个带缓存的 3KLUT CPU 大约需要 12 秒，与仿真或综合此类设计所需的时间相比，这是一个短得多的时间。

2.7.5 SpinalHDL 是如何诞生的

2014 年 12 月至 2016 年 4 月期间，这是一个个人爱好项目。但自 2016 年 4 月起，就有一个人全职从事这项工作。有些人也定期为该项目做出贡献。

2.7.6 既然有了 VHDL/Verilog/SystemVerilog，为什么还要开发新的语言呢？

前言 专门讨论这个主题。

2.7.7 如何使用 SpinalHDL 的未发布版本（但在 git 上提交）？

首先，如果您还没有克隆存储库，则需要获取存储库：

```
git clone --depth 1 -b dev https://github.com/SpinalHDL/SpinalHDL.git
cd SpinalHDL
```

在上面的命令中，可以用要签出的分支名称替换 `dev` 。`--depth 1` 阻止下载版本库历史。

然后将获取目录中的代码发布：

```
sbt clean '++ 2.12.13' publishLocal
```

这里 `2.12.13` 是使用的 Scala 版本。前两个数字必须与您的项目中使用的版本相匹配。您可以在 `build.sbt` 和/或 `build.sc` 中找到它：

```
ThisBuild / scalaVersion := "2.12.16" // in build.sbt
// or
def scalaVersion = "2.12.16" // in build.sc
```

然后在您的项目中，更新 `build.sbt` 或 `build.sc` 中指定的 SpinalHDL 版本：应将其设置为 `dev` 而不是版本号。

```
val spinalVersion = "1.7.3"
// becomes
val spinalVersion = "dev"
```

备注： 无论您之前签出哪个分支，这里总是 `dev` 。

2.8 其他学习资料

- 一个简短的展示案例（PDF 幻灯片）
- 语言演示（PDF 幻灯片）
- Jupyter 训练营
- 研讨会

网上还有一些更具体的视频：

- YouTube 视频
- f-si peertube 上的视频

这里有几场 SpinalHDL 网络研讨会的视频录像：

- Datenlord 的 Youtube 频道

备注： 其中一些教程没有使用最新版本的 SpinalHDL，因此它们可能会缺少一些最新的 SpinalHDL 功能。

CHAPTER 3

入门

让我们开始学习 SpinalHDL！在本章中，我们将安装和设置环境，体验该语言并学习如何生成 VHDL 和 Verilog，以及如何在编写代码的同时进行 lint 检查。

3.1 安装和设置

Spinal 是一个 Scala 库（使用 Java VM 的编程语言），因此需要设置 Scala 环境；有很多方法可以做到这一点。此外，它还生成 VHDL、Verilog 或 SystemVerilog，可供许多不同的工具使用。本节介绍支持的 *SpinalHDL 描述到仿真流程的安装方法*，但可能还有其他方法。

3.1.1 必需/推荐的工具

在下载 SpinalHDL 工具之前，您需要安装 Scala 环境：

- Java JDK，Java 环境
- Scala 2，编译器和库
- SBT ，Scala 程序构建工具

这些工具可以支持使用 Spinal 生成代码；但在没有任何其他工具的情况下，它仅限于 HDL 代码生成。

要启用更多功能，我们建议：

- 一个 IDE（例如当前推荐的带有 Scala 插件的 IntelliJ 或带有 Metals 扩展的 VSCodium ）可获得如下功能：
  - 代码建议/自动完成
  - 自动构建，在代码中显示存在语法错误
  - 通过单击生成代码
  - 单击即可运行仿真/测试（如果设置了支持的仿真器）
- 支持的仿真器，例如 Verilator ，可以直接从 SpinalHDL 测试设计。
- Gtkwave 查看 Verilator 在仿真过程中生成的波形。
- Git 版本控制系统

- 用 Verilator 进行仿真所需的 C++ 工具链
- 一个 linux shell，需要使用 Verilator 进行仿真

3.1.2 Linux 安装

在撰写本文时，推荐的 Scala 和 SBT 安装方法是通过 Coursier。除了 scala 工具之外，Coursier 还能够安装 Java JDK ，在下面的示例中，我们通过包管理器安装 Java。出于 Scala 版本兼容的考量，我们建议安装 JDK 17 (LTS)。

对于 Debian 或 Ubuntu，运行：

```
sudo apt-get update
sudo apt-get install openjdk-17-jdk-headless curl git
curl -fL "https://github.com/coursier/launchers/raw/master/cs-x86_64-pc-linux.gz"␣
↪| gzip -d > cs
chmod +x cs
# should find the just installed jdk, agree to cs' questions for adding to your␣
↪PATH
./cs setup
source ~/.profile
```

如果您想安装用于仿真和/或形式化证明的工具，我们推荐 oss-cad-suite。它包含波形查看器（gtkWave）、verilog 仿真器（verilator 和 iverilog）、VHDL 仿真器（GHDL）以及一些其他工具。如果您想自己构建工具，请查看旧版仿真工具安装说明。

首先，我们安装所需的 C++ 工具链并下载 oss-cad-suite。要使用它，我们必须为每个我们想要使用它的 shell 加载 oss-cad-suite 环境。请注意，oss-cad-suite 包含一个可能会干扰系统 Python 安装的 Python 3 解释器，如果永久加载（loaded permanently）它。

前往 oss-cad-suite 发布页面 获取最新版本的下载链接。您可以将 oss-cad-suite 下载/解压到您选择的文件夹中。(oss-cad-suite 的最后测试版本是 *2023-10-22*，但最新的版本可能也可以工作)

```
sudo apt-get install make gcc g++ zlib1g-dev
curl -fLO <download link>
tar xzf <file that you downloaded>
```

To use oss-cad-suite in a shell you need to load it's environment, e.g. via `source <path to oss-cad-suite>/environment`.

3.1.3 Mac OS X 安装

Mac OS X 上可以使用 homebrew 安装。默认情况下 homebrew 安装 Java 21，但 SpinalHDL 教程 SpinalTemplateSbt 使用 Scala 版本 2.12.16，Java 21 不支持该版本（17 仍然是推荐的 LTS 版本，https://whichjdk.com/）。因此，要安装 Java 1.7 版本，请执行以下操作：

```
brew install openjdk@17
```

然后，将其添加到您的路径（path）中。

```
export PATH="/opt/homebrew/opt/openjdk@17/bin:$PATH"
```

要管理 Java 的多个版本，还必须安装 jenv。

```
brew install jenv
```

Jenv 将这些行添加到.bash_profile 中

```
export PATH="$HOME/.jenv/bin:$PATH"
eval "$(jenv init -)"
```

接下来你必须安装 scala 的交互式构建工具 sbt。

```
brew install sbt
```

如果这对您有用，请告诉我们。如果这对您不起作用，您可以在此处阅读有关 Mac o SX 安装的 github 问题。https://github.com/SpinalHDL/SpinalHDL/issues/1216

如果您想安装用于仿真和/或形式化证明的工具，我们推荐 oss-cad-suite。

3.1.4 Windows 安装

备注： 虽然可以进行本机安装，但更简单且目前推荐的方法是在 Windows 上使用 WSL。如果您想使用 WSL，请安装您选择的 发行版 ，并按照 Linux 安装说明进行操作。WSL 实例中的数据可以从 Windows 访问，在 `\\wsl$` 路径下。如果您想使用 IntelliJ，则必须将 Linux 版本下载到 WSL，如果您想使用 VSCode，则可以使用 Windows 版本远程编辑 WSL 中的数据。

在撰写本文时，推荐的安装 Scala 和 SBT 方法是通过 Coursier。除了 scala 工具之外，Coursier 还能够安装 Java JDK 来使用，在下面的示例中我们手动安装 Java。由于 Scala 兼容性原因，我们建议安装 JDK 17 (LTS)。

首先，下载并安装 Adoptium JDK 17。下载、解压并运行 Coursier 安装程序，当询问是否更新您的 `PATH` 变量时，选择同意。重新启动以强制更新 `PATH`。

这足以生成硬件。对于仿真，请继续选择以下任一选项。如果您想自己构建仿真工具，请查看旧版仿真工具安装说明。

备注： SpinalHDL 维护者 *Readon <https://github.com/Readon>* 提供的一体化解决方案可用于安装和运行 SpinalHDL，并通过 Verilator 仿真、通过 SymbiYosys 进行形式化验证。下载 安装程序 并将该环境安装在磁盘上的任何位置。单击“开始”菜单中的 MSYS2-MINGW64 图标启动构建环境，并使用 MSYS2 默认控制台。另一种方法是使用 “Windows Terminal” 或类似 Tabby 的应用程序，并使用启动命令 `%MSYS2_ROOT%\msys2_shell.cmd -defterm -here -no-start -mingw64`，其中 `%MSYS2_ROOT%` 是 msys2 安装的位置。值得注意的是，如果要离线使用，要仔细选择项目所依赖的库，否则需要手动下载安装包。有关更多详细信息，请参阅对应仓库的自述文件。

用于仿真的 MSYS2 verilator 工具

我们建议通过 *MSYS2 <https://www.msys2.org>* 安装编译器/仿真器。其他安装 gcc/make/shell 的方法（例如 Chocolatey、scoop 等）也可能有效，但未经测试。

SpinalHDL 维护者 *Readon <https://github.com/Readon>* 正在维护一个 MSYS2 分支，默认安装所有需要的官方可用和自定义构建的软件包（也由 Readon 在 此处 *<https://github.com/Readon/ MINGW-SpinalHDL>* 维护）用于仿真和形式化验证。可以在 此处 *<https://github.com/Readon/msys2-installer>* 找到。如果使用该安装包，则已经安装了下面通过 pacman 安装的软件包，并且可以跳过这些安装步骤。

目前 verilator 4.228 是已知可以工作的最新版本。

下载最新的安装程序并使用默认设置安装 MSYS2。安装结束后你应该会得到一个 MSYS2 终端，运行：

```
pacman -Syuu
# will (request) close down terminal
# open 'MSYS2 MINGW64' from start menu
pacman -Syuu
pacman -S --needed base-devel mingw-w64-x86_64-toolchain mingw-w64-x86_64-iverilog␣
↪mingw-w64-x86_64-ghdl-llvm git
curl -O https://repo.msys2.org/mingw/mingw64/mingw-w64-x86_64-verilator-4.228-1-
↪any.pkg.tar.zst
pacman -U mingw-w64-x86_64-verilator-4.228-1-any.pkg.tar.zst
```

在 MSYS2 MINGW64 终端中，我们需要设置一些环境变量以使 Java/sbt 可用（您可以通过将它们添加到 MSYS2 中的 `~/.bashrc` 来长期保存这些设置）：

```
export VERILATOR_ROOT=/mingw64/share/verilator/
export PATH=/c/Program\ Files/Eclipse\ Adoptium/jdk-17.0.8.101-hotspot/bin:$PATH
export PATH=/c/Users/User/AppData/Local/Coursier/data/bin:$PATH
```

有了这个，您应该能够从 MSYS2 终端运行 sbt/verilator 仿真（通过调用 `sbt.bat` 使用）。

用 MSYS2 实施形式化验证

除了上面的步骤之外，我们还需要安装 yosys、sby、z3 和 yices。yosys(可用的 yosys-smtbmc) 和 sby 并不使用 MSYS2 官方提供的包（无法使用），而是由 *Readon <https://github.com/Readon>* 提供。如果您使用他们的安装程序，则不需要后面这些步骤（您应该检查是否有更新的软件包可用）。

```
pacman -S mingw-w64-x86_64-z3 mingw-w64-x86_64-yices mingw-w64-x86_64-autotools␣
↪mingw-w64-x86_64-python3-pip
python3 -m pip install click
curl -OL https://github.com/Readon/MINGW-SpinalHDL/releases/download/v0.4.9/mingw-
↪w64-x86_64-yosys-0.31-1-any.pkg.tar.zst
curl -OL https://github.com/Readon/MINGW-SpinalHDL/releases/download/v0.4.9/mingw-
↪w64-x86_64-python-sby-0.31-1-any.pkg.tar.zst
pacman -U *-yosys-*.pkg.tar.*
pacman -U *-python-sby-*.pkg.tar*
```

3.1.5 OCI 容器

也可以使用 SpinalHDL 开发容器。该容器托管在 `ghcr.io/spinalhdl/docker:master` ，可以与 Docker/Podman/Github Codespaces 一起使用。它用于 SpinalHDL CI 测试，因此，也可以成为本地运行的简单方法。

要使用容器，请运行例如 SpinalHDL 项目根目录中的 `podman run -v .:/workspace -it ghcr.io/spinalhdl/docker:master` ，使项目目录在 `/workspace` 中可用。

请查阅您的发行版（Linux、WSL）或 Docker (Windows) 的文档，了解如何安装您想要使用容器的运行时依赖。多个编辑器/IDE（例如 VSCode、IntelliJ、Neovide）允许对容器进行远程开发。如何进行远程开发请查阅相应编辑器的文档。

3.1.6 在无网络的 Linux 环境中安装 SBT

备注： 如果您不使用无网环境，我们建议您使用正常的 Linux 安装。（这里是无网环境安装的方法）

通常，SBT 使用在线仓库来下载和缓存项目的依赖项。该缓存位于几个文件夹中：

- `~/.sbt`
- `~/.cache/JNA`
- `~/.cache/coursier`

要设置无互联网环境，您可以：

1. 在有互联网的场景下配置环境（见上文）
2. 启动 Spinal 命令（请参阅*在 CLI（命令行）中结合 SBT 使用 Spinal*）来获取依赖项（例如使用 *入门* 仓库）
3. 将缓存复制到无互联网环境。

3.2 创建第一个 SpinalHDL 项目

我们为您准备了一个现成的项目：入门 仓库。

您可以 下载 它，或克隆它。

以下命令将项目克隆到名为 MySpinalProject 的新目录中，并初始化新的 git 历史记录：

```
git clone --depth 1 https://github.com/SpinalHDL/SpinalTemplateSbt.git␣
↪MySpinalProject
cd MySpinalProject
rm -rf .git
git init
git add .
git commit -m "Initial commit from template"
```

3.2.1 项目的目录结构

备注： 这里描述的结构是默认结构，但可以轻松修改。

项目的根目录中有以下文件：

文件	描述
build.sbt	sbt 的 Scala 配置
build.sc	mill 是一个 sbt 的替代品，它的配置方法是
hw/	包含硬件描述的文件夹
project/	更多 Scala 配置
README.md	描述您的项目的 text/markdown 文件
.gitignore	版本控制中要忽略的文件列表
.mill-version	mill 的更多配置
.scalafmt.conf	配置自动格式化代码的规则

正如您可能猜到的，这里有趣的是 hw/。它包含四个文件夹：spinal/、verilog/ 和 vhdl/ 保存您的 IP，gen/ 保存 Spinal 生成的 IP。

hw/spinal/ 包含一个以您的项目名称命名的文件夹。该名称以及公司名称必须在 build.sbt 和 build.sc 中设置；并且这个名称必须在 .scala 文件开头以 package yourprojectname 的形式给出。

在 hw/spinal/yourprojectname/ 中，是你的 IP 描述，仿真测试，形式化验证测试；还有 Config.scala，其中包含“Spinal“的配置。

备注： sbt 必须且 ** 仅 ** 在项目的根目录中使用，该目录包含 build.sbt 文件。

3.2.2 在 SpinalHDL 代码中使用 Spinal

现在，教程展示了如何根据您的开发环境在 SpinalHDL 代码中使用 Spinal：

- *在 CLI （命令行）中结合 SBT 使用 Spinal*
- *在 VSCodium 中使用 Spinal*
- *从 IntelliJ IDEA 使用 Spinal*

3.3 在 CLI （命令行）中结合 SBT 使用 Spinal

首先，在 template （提前下载）的根目录中打开一个终端。

可以直接从终端执行命令：

```
sbt "firstCommand with arguments" "secondCommand with more arguments"
```

但是 sbt 有一个相当长的启动时间，所以我们建议使用它的交互模式：

```
sbt
```

现在 sbt 会显示提示。让我们从 Scala 编译开始。它将获取依赖项，因此第一次运行可能需要一些时间：

```
compile
```

实际上，您永远不会仅使用 compile 命令，因为它会在需要时自动完成。与后续的构建相比，第一次构建时间将花费较长时间，因为 sbt 工具需要从冷启动构建整个项目，后续构建则尽可能使用增量构建。sbt 支持交互式 shell 内的自动补全，以帮助发现和使用可用命令。您可以使用 sbt shell 或者 sbt （不带参数）命令启动交互式 shell。

要运行特定的 HDL 代码生成或仿真，命令是 runMain。因此，如果您输入 runMain、空格和 tab，您应该得到以下结果：

```
sbt:SpinalTemplateSbt> runMain
   ;                                projectname.MyTopLevelVerilog
   projectname.MyTopLevelFormal     projectname.MyTopLevelVhdl
   projectname.MyTopLevelSim
```

自动补全功能会建议所有可以运行的内容。这是一个生成 Verilog 代码的例子：

```
runMain projectname.MyTopLevelVerilog
```

查看目录./hw/gen/：有一个新的 MyTopLevel.v 文件！

现在在命令的开头添加一个 ~ ：

```
~ runMain projectname.MyTopLevelVerilog
```

它打印出这个：

```
 sbt:SpinalTemplateSbt> ~ runMain mylib.MyTopLevelVerilog
 [info] running (fork) mylib.MyTopLevelVerilog
 [info] [Runtime] SpinalHDL v1.7.3    git head :␣
↪aeaeece704fe43c766e0d36a93f2ecbb8a9f2003
 [info] [Runtime] JVM max memory : 3968,0MiB
 [info] [Runtime] Current date : 2022.11.17 21:35:10
 [info] running (fork) projectname.MyTopLevelVerilog
 [info] [Runtime] SpinalHDL v1.9.3    git head :␣
↪029104c77a54c53f1edda327a3bea333f7d65fd9
 [info] [Runtime] JVM max memory : 4096.0MiB
```

(续下页)

(接上页)

```
 [info] [Runtime] Current date : 2023.10.05 19:30:19
 [info] [Progress] at 0.000 : Elaborate components
 [info] [Progress] at 0.508 : Checks and transforms
 [info] [Progress] at 0.560 : Generate Verilog
[info] [Done] at 0.603
[success] Total time: 1 s, completed Oct 5, 2023, 7:30:19 PM
[info] 1. Monitoring source files for projectname/runMain projectname.
↪MyTopLevelVerilog...
[info]    Press <enter> to interrupt or '?' for more options.
```

所以现在，每次保存源文件时，它都会重新生成 `MyTopLevel.v`。为此，它会自动编译源文件并执行 lint 检查。这样，当您编辑源文件时，您几乎可以实时在终端上打印错误。

您可以按 Enter 停止自动生成，然后按 Ctrl-D 退出 `sbt`。

也可以直接从终端启动它，而不使用 `sbt` 的交互式提示：

```
sbt "~ runMain mylib.MyTopLevelVerilog"
```

现在您可以使用您的环境了，让我们探索一下代码：一个简单的例子。

3.4 在 VSCodium 中使用 Spinal

备注： VSCodium 是 Visual Studio Code 的开源版本，但不包含 Microsoft 可下载版本的数据报告功能。

作为一次性安装任务，请转到“查看”->“扩展”菜单栏搜索“Scala”并安装“Scala（Metals）”扩展。

打开工作区：`File > Open Folder...` 并打开您之前在*创建第一个 SpinalHDL 项目* 中下载的文件夹。

另一种启动方法是 cd 进入适当的目录并输入 `codium .`

稍等一下，右下角应该会出现一个弹出通知：“找到多个构建定义。您想使用哪个?”。单击 `sbt` ，然后出现另一个弹出窗口，单击 `Import build`。

运行 `sbt bloopInstall` 时稍等片刻。然后会出现一个警告弹出窗口，您可以忽略它（不再显示）。

找到并打开 `hw/spinal/projectname/MyTopLevel.scala`。稍等一下，然后看到 Metals 在每个 `App` 之前显示的 `run | debug` 行。例如，单击 `object MyTopLevelVerilog` 上方的 `run` 。或者，您可以选择菜单栏 -> 运行 -> 运行而不调试。两种方法都会执行设计检查，并在检查通过后生成 Verilog 文件 `./hw/gen/MyTopLevel.v`

这就是使用 VSCodium 的 SpinalHDL 所需要做的全部工作。您现在拥有经过设计规则检查的 Verilog 和/或 VHDL，您可以将其用作您最喜欢的综合工具的输入。

现在您已经知道如何使用 VSCodium 开发环境了，让我们来探索一下代码：一个简单的例子 。

3.5 从 IntelliJ IDEA 使用 Spinal

除了上述要求，你还需要下载 IntelliJ IDEA（免费的 * 社区版 * 即可）。安装 IntelliJ 后，还需检查是否启用了 Scala 插件（可在此处查看 安装信息 ）。

并进行以下操作：

- 在 *Intellij IDEA* 中，在该仓库的根目录下“导入项目”，选择 *从外部模型导入 SBT 工程*并确保勾选所有复选框。
- 此外，您可能需要在 *IntelliJ* 中指定一些路径，例如 JDK 的安装位置。

- 在项目（Intellij 项目 GUI）中，右键单击 `src/main/scala/mylib/MyTopLevel.scala` 并选择 “Run MyTopLevel”。

这将在项目目录中生成输出文件 `MyTopLevel.vhd`，该文件实现了一个简单的 8 位计数器。

现在您可以使用您的环境了，让我们探索一下代码：一个简单的例子。

3.6 Scala 使用指南

重要： 变量和函数应该定义为 “object“，`class`，`function`。您不能在根目录下的 Scala 文件中定义它们。

3.6.1 基础内容

类型

在 Scala 中，主要有 5 种类型：

类型	形式	描述
Boolean	true, false	
Int	3, 0x32	32 位整数
Float	3.14f	32 位浮点数
Double	3.14	64 位浮点
String	“Hello world”	UTF-16 字符串

变量

在 Scala 中，您可以使用 var 关键字定义变量：

```
var number : Int = 0
number = 6
number += 4
println(number) // 10
```

Scala 能够自动推断类型。如果变量是在声明时赋值的，则不需要指定类型：

```
var number = 0   // The type of 'number' is inferred as an Int during compilation.
```

然而，在 Scala 中使用 var 并不常见。相反，经常使用由 `val` 定义的常量值：

```
val two   = 2
val three = 3
val six   = two * three
```

函数

例如，如果你想定义一个函数，当两个参数之和大于零时返回 `true` ，你可以这样做：

```
def sumBiggerThanZero(a: Float, b: Float): Boolean = {
  return (a + b) > 0
}
```

然后，要调用该函数，您可以编写：

```
sumBiggerThanZero(2.3f, 5.4f)
```

您还可以按名称指定参数，如果您有很多参数，这会很有用：

```
sumBiggerThanZero(
  a = 2.3f,
  b = 5.4f
)
```

返回类型

`return` 关键字不是必需的。如果没有它，Scala 会将函数的最后一条语句作为返回值。

```
def sumBiggerThanZero(a: Float, b: Float): Boolean = {
  (a + b) > 0
}
```

返回类型推断

Scala 能够自动推断返回类型。您不需要指定它：

```
def sumBiggerThanZero(a: Float, b: Float) = {
  (a + b) > 0
}
```

大括号

如果您的函数仅包含一条语句，则 Scala 函数不需要大括号：

```
def sumBiggerThanZero(a: Float, b: Float) = (a + b) > 0
```

不返回任何内容的函数

如果您希望函数不返回任何内容，则返回类型应设置为 `Unit`。它相当于 C/C++ `void` 类型。

```
def printer(): Unit = {
  println("1234")
  println("5678")
}
```

参数默认值

您可以为函数的每个参数指定默认值：

```
def sumBiggerThanZero(a: Float, b: Float = 0.0f) = {
  (a + b) > 0
}
```

Apply 函数

名为 apply 的函数很特殊，因为您无需输入名称即可调用它们：

```
class Array() {
  def apply(index: Int): Int = index + 3
}

val array = new Array()
val value = array(4)   // array(4) is interpreted as array.apply(4) and will␣
↪return 7
```

这个概念也适用于 Scala "object"（静态）

```
object MajorityVote {
  def apply(value: Int): Int = ...
}

val value = MajorityVote(4) // Will call MajorityVote.apply(4)
```

对象（Object）

在 Scala 中，没有 static 关键字。取而代之的是 object。在 object 定义中所有内容都是静态的。以下示例定义了一个名为 pow2 的静态函数，它接受浮点值作为参数并返回浮点值。

```
object MathUtils {
  def pow2(value: Float): Float = value * value
}
```

然后你可以这样来调用它：

```
MathUtils.pow2(42.0f)
```

入口点（main）

Scala 程序的入口点（主函数）应在对象内部定义为名为 main 的函数。

```
object MyTopLevelMain {
  def main(args: Array[String]) {
    println("Hello world")
  }
}
```

类

类的语法与 Java 非常相似。想象一下，您想要定义一个 Color 类，它将三个 Float 值 (r,g,b) 作为构造函数的参数：

```
class Color(r: Float, g: Float, b: Float) {
  def getGrayLevel(): Float = r * 0.3f + g * 0.4f + b * 0.4f
}
```

然后，实例化上一个示例中的类并使用其 getGrayLevel 函数：

```
val blue = new Color(0, 0, 1)
val grayLevelOfBlue = blue.getGrayLevel()
```

注意，如果你想从外部访问类的构造参数，那么这个参数应该定义为 val：

```
class Color(val r: Float, val g: Float, val b: Float) { ... }
...
val blue = new Color(0, 0, 1)
val redLevelOfBlue = blue.r
```

继承

举个例子，假设您要定义两个类，Rectangle 和 Square，它们扩展了 Shape 类：

```
class Shape {
  def getArea(): Float
}

class Square(sideLength: Float) extends Shape {
  override def getArea() = sideLength * sideLength
}

class Rectangle(width: Float, height: Float) extends Shape {
  override def getArea() = width * height
}
```

样例类

样例类是声明类的另一种方式。

```
case class Rectangle(width: Float, height: Float) extends Shape {
  override def getArea() = width * height
}
```

case class 和 class 之间有下面区别：

- 样例类不需要 new 关键字来实例化。
- 构造参数可从外部获取；你不需要将它们定义为 val。

在 SpinalHDL 中，这解释了编码约定背后的原因：通常建议使用 “case class“ 而不是 class，以减少代码量并提高一致性。

模板/类型参数化

想象一下，您想要设计一个类，它是某数据类型的队列，在这种情况下，您需要向该类提供类型参数：

```
class  Queue[T]() {
  def push(that: T) : Unit = ...
  def pop(): T = ...
}
```

如果你想将 T 类型限制为给定类型的子类（例如 Shape），你可以使用 <: Shape 语法：

```
class Shape() {
    def getArea(): Float
}
class Rectangle() extends Shape { ... }

class  Queue[T <: Shape]() {
  def push(that: T): Unit = ...
  def pop(): T = ...
}
```

对于函数来说也是如此：

```
def doSomething[T <: Shape](shape: T): Something = { shape.getArea() }
```

3.6.2 编码规范

简介

SpinalHDL 中使用的编码规范与 Scala 风格指南 中的规定相同。

一些额外的细节和案例将在下几页中解释。

类与样例类

当您定义一个 Bundle 或 Component 时，最好将其声明为案例类。

原因是：

- 它避免使用 new 关键字。在某些情况下，永远不必使用它比有时使用它要好。
- case class 提供了 clone 函数。当需要克隆 Bundle 时，该函数非常有用。例如，当您定义新的 Reg 或某种新的 Stream 时。
- 构造参数从外部直接可见。

样例类/类

所有类名称都应以大写字母开头

```
class Fifo extends Component {

}

class Counter extends Area {

}

case class Color extends Bundle {
```

(续下页)

(接上页)

```
}
```

伴生对象

伴随对象 应该以大写字母开头。

```
object Fifo {
  def apply(that: Stream[Bits]): Stream[Bits] = {...}
}

object MajorityVote {
  def apply(that: Bits): UInt = {...}
}
```

此规则的一个例外是当伴生对象用作函数时（其中包含了 apply 函数），并且这些 apply 函数不生成硬件：

```
object log2 {
  def apply(value: Int): Int = {...}
}
```

函数

函数应始终以小写字母开头：

```
def sinTable = (0 until sampleCount).map(sampleIndex => {
  val sinValue = Math.sin(2 * Math.PI * sampleIndex / sampleCount)
  S((sinValue * ((1 << resolutionWidth) / 2 - 1)).toInt, resolutionWidth bits)
})

val rom =  Mem(SInt(resolutionWidth bits), initialContent = sinTable)
```

实例

类的实例应始终以小写字母开头：

```
val fifo   = new Fifo()
val buffer = Reg(Bits(8 bits))
```

if / when

Scala 中的 if 和 SpinalHDL 中的 when 通常应按以下方式编写：

```
if(cond) {
  ...
} else if(cond) {
  ...
} else {
  ...
}

when(cond) {
```

(续下页)

(接上页)

```
  ...
} elsewhen(cond) {
  ...
} otherwise {
  ...
}
```

例外情况可能是：

- 可以在关键字前添加一个点，例如方法 `.elsewhen` 和 `.otherwise` 。
- 如果可以提高代码可读性，可以将 `if`/`when` 语句压缩到一行中。

switch

SpinalHDL 中的 `switch` 通常应按以下方式编写：

```
switch(value) {
  is(key) {

  }
  is(key) {

  }
  default {

  }
}
```

如果可以提高代码可读性，可以将 `is`/`default` 语句压缩到一行中。

参数

在样例类中对 `Component`/`Bundle` 的参数进行分组通常是受欢迎的，因为：

- 更容易使用/操作设计的配置
- 更好的可维护性

```
case class RgbConfig(rWidth: Int, gWidth: Int, bWidth: Int) {
  def getWidth = rWidth + gWidth + bWidth
}

case class Rgb(c: RgbConfig) extends Bundle {
  val r = UInt(c.rWidth bits)
  val g = UInt(c.gWidth bits)
  val b = UInt(c.bWidth bits)
}
```

但这不应该适用于所有情况。例如：在 FIFO 中，将 `dataType` 参数与 fifo 的 `depth` 参数分组是没有意义的。因为一般情况下，`dataType` 与设计相关，而 `depth` 则与设计的配置有关。

```
class Fifo[T <: Data](dataType: T, depth: Int) extends Component {

}
```

3.6.3 交互

简介

事实上，SpinalHDL 不是一种语言：它是一个常规的 Scala 库。乍一看这似乎很奇怪，但这是一个非常强大的组合。

您可以使用整个 Scala 世界的工具，通过 SpinalHDL 库来帮助您描述硬件，但要正确地做到这一点，了解 SpinalHDL 如何与 Scala 交互非常重要。

SpinalHDL 在 API 隐藏后的工作原理

当您执行 SpinalHDL 硬件描述时，每次使用 SpinalHDL 函数、运算符或类时，它都会构建一个内存中图来表示您设计的网表。

然后，当实例细化完成后（顶层 `Component` 类的实例化），SpinalHDL 将对构建的图进行一些处理，如果一切正常，它将把该图生成为 VHDL 或 Verilog 文件。

一切都是引用

例如，如果您定义一个 Scala 函数，它接受类型为 `Bits` 的参数，那么当您调用它时，它将作为引用传递。因此，如果您在函数内部赋值（通过 ":="）该参数，它对底层 `Bits` 对象具有相同的效果，就像您在函数外部给它赋值一样。

硬件类型

SpinalHDL 中的硬件数据类型是两个概念的组合：

- 给定 Scala 类型的实例
- 该实例的配置信息

例如，`Bits(8 bits)` 是 Scala 类型 `Bits` 及其 `8 bits` 这一配置信息（作为构造参数）的组合。

RGB 示例

我们以 Rgb 线束类为例：

```
case class Rgb(rWidth: Int, gWidth: Int, bWidth: Int) extends Bundle {
  val r = UInt(rWidth bits)
  val g = UInt(gWidth bits)
  val b = UInt(bWidth bits)
}
```

这里的硬件数据类型是 Scala `Rgb` 类及其 `rWidth`, `gWidth`, 和 `bWidth` 参数的组合。

这是一个用法示例：

```
// Define an Rgb signal
val myRgbSignal = Rgb(5, 6, 5)

// Define another Rgb signal of the same data type as the preceding one
val myRgbCloned = cloneOf(myRgbSignal)
```

您还可以使用函数来定义各种类型工厂（typedef）：

```
// Define a type factory function
def myRgbTypeDef = Rgb(5, 6, 5)

// Use that type factory to create an Rgb signal
val myRgbFromTypeDef = myRgbTypeDef
```

生成的 RTL 中的信号名称

为了命名生成的 RTL 中的信号，SpinalHDL 使用 Java 反射遍历整个组件层次结构，收集存储在类属性内的所有引用，并使用属性名称命名它们。

这就是函数内定义的每个信号的名称都会丢失的原因：

```
def myFunction(arg: UInt) {
  val temp = arg + 1  // You will not retrieve the `temp` signal in the generated␣
↪RTL
  return temp
}

val value = myFunction(U"000001") + 42
```

如果您想在生成的 RTL 中保留内部变量的名称，一种解决方案是使用逻辑区（Area）：

```
def myFunction(arg: UInt) new Area {
  val temp = arg + 1  // You will not retrieve the temp signal in the generated RTL
}

val myFunctionCall = myFunction(U"000001")  // Will generate `temp` with␣
↪`myFunctionCall_temp` as the name
val value = myFunctionCall.temp  + 42
```

Scala 用于实例细化，SpinalHDL 用于硬件描述

例如，如果您编写 Scala for 循环来生成某些硬件，它将生成展开的 VHDL/Verilog 代码。

另外，如果你想要一个常量，你不应该使用 SpinalHDL 硬件代码，而应该使用 Scala 代码。例如：

```
// This is wrong, because you can't use a hardware Bool as construction parameter.␣
↪(It will cause hierarchy violations.)
class SubComponent(activeHigh: Bool) extends Component {
  // ...
}

// This is right, you can use all the Scala world to parameterize your hardware.
class SubComponent(activeHigh: Boolean) extends Component {
  // ...
}
```

Scala 实例细化能力（if、for、函数式编程）

在 Scala 中，所有的语法都可以用来细化硬件设计。例如，Scala 的 `if` 语句可以用于启用或禁用部分硬件的生成。

```
val counter = Reg(UInt(8 bits))
counter := counter + 1
if(generateAClearWhenHit42) {  // Elaboration test, like an if generate in vhdl
  when(counter === 42) {       // Hardware test
    counter := 0
  }
}
```

Scala 的 `for` 循环也是如此：

```
val value = Reg(Bits(8 bits))
when(something) {
  // Set all bits of value by using a Scala for loop (evaluated during hardware elaboration)
  for(idx <- 0 to 7) {
    value(idx) := True
  }
}
```

此外，函数式编程技术可用于许多 SpinalHDL 类型：

```
val values = Vec(Bits(8 bits), 4)

val valuesAre42    = values.map(_ === 42)
val valuesAreAll42 = valuesAre42.reduce(_ && _)

val valuesAreEqualToTheirIndex = values.zipWithIndex.map{ case (value, i) => value === i }
```

3.6.4 Scala 使用指南

简介

Scala 是一种非常强大的编程语言，它受到一组语言的影响。但通常，这组语言与大多数程序员使用的语言并不交叉。这可能会阻碍新手理解 Scala 背后的概念和设计选择缘由。

接下来的几页将介绍 Scala，并尝试提供足够的信息，以便新手熟悉 SpinalHDL。

3.7 VHDL 用户入门

3.7.1 与 VHDL 对比

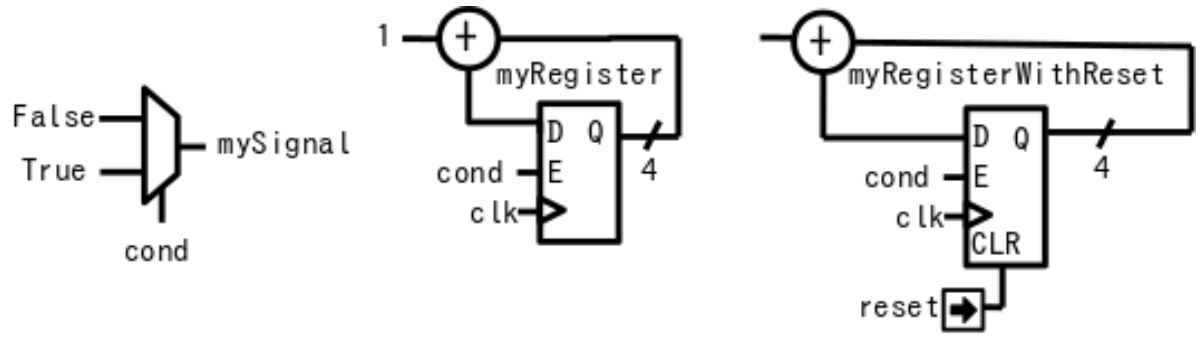
简介

本页将讨论 VHDL 和 SpinalHDL 之间的主要区别。但不会深入解释。

过程（Process）

通常编写 RTL 时需要用到过程，但是它们的语义使用起来可能很笨拙。由于它们在 VHDL 中的工作方式，可能会迫使您拆分代码并重复编写。

要生成以下 RTL：



您必须编写以下 VHDL：

```
  signal mySignal : std_logic;
  signal myRegister : std_logic_vector(3 downto 0);
  signal myRegisterWithReset : std_logic_vector(3 downto 0);
begin
  process(cond)
  begin
    mySignal <= '0';
    if cond = '1' then
      mySignal <= '1';
    end if;
  end process;

  process(clk)
  begin
    if rising_edge(clk) then
      if cond = '1' then
        myRegister <= myRegister + 1;
      end if;
    end if;
  end process;

  process(clk,reset)
  begin
    if reset = '1' then
      myRegisterWithReset <= (others => '0');
    elsif rising_edge(clk) then
      if cond = '1' then
        myRegisterWithReset <= myRegisterWithReset + 1;
      end if;
    end if;
  end process;
```

在 SpinalHDL 中，它是：

```
val mySignal = Bool()
val myRegister = Reg(UInt(4 bits))
val myRegisterWithReset = Reg(UInt(4 bits)) init(0)

mySignal := False
when(cond) {
  mySignal := True
  myRegister := myRegister + 1
```

(续下页)

(接上页)

```
  myRegisterWithReset := myRegisterWithReset + 1
}
```

隐式与显式定义对比

在 VHDL 中，当声明一个信号时，您无需指定它是组合信号还是寄存器。给它赋值的位置和方式决定了它是组合电路还是寄存器。

在 SpinalHDL 中，这些事情是明确的。寄存器直接在其声明中就定义为寄存器。

时钟域

在 VHDL 中，每次想要定义一堆寄存器时，都需要将时钟和复位信号传递给它们。此外，您必须在各处硬编码如何使用这些时钟和复位信号（包括它们的属性时钟沿、复位极性、复位性质（异步、同步））。

在 SpinalHDL 中，您可以定义 `ClockDomain`，然后定义使用它的硬件区域。

例如：

```
val coreClockDomain = ClockDomain(
  clock = io.coreClk,
  reset = io.coreReset,
  config = ClockDomainConfig(
    clockEdge = RISING,
    resetKind = ASYNC,
    resetActiveLevel = HIGH
  )
)
val coreArea = new ClockingArea(coreClockDomain) {
  val myCoreClockedRegister = Reg(UInt(4 bits))
  // ...
  // coreClockDomain will also be applied to all sub components instantiated in␣
↪the Area
  // ...
}
```

组件的内部组织方式

在 VHDL 中，有一个 `block` 功能，允许您在组件内定义逻辑子区域。然而，几乎没有人使用这一功能，因为大多数人不了解它们，也因为这些区域内定义的所有信号都无法从外部读取、使用。

在 SpinalHDL 中，你有一个 `Area` 功能，可以更好地实现这个概念：

```
val timeout = new Area {
  val counter = Reg(UInt(8 bits)) init(0)
  val overflow = False
  when(counter =/= 100) {
    counter := counter + 1
  } otherwise {
    overflow := True
  }
}

val core = new Area {
  when(timeout.overflow) {
    timeout.counter := 0
  }
}
```

在 Area 内部定义的变量和信号可以在组件的其他地方访问，包括其他 Area 区域内。

安全性

在 VHDL 中，就像在 SpinalHDL 中一样，很容易编写出组合逻辑环，或者因为忘记给路径中的信号驱动而得到一个锁存器（latch）。

然后，为了检测这些问题，您可以使用一些 lint 工具来分析您的 VHDL，但这些工具不是免费的。在 SpinalHDL 中，lint 过程集成在编译器内部，并且在一切正常之前它不会生成 RTL 代码。此外，它还会检查跨时钟域信号。

功能与流程

函数和过程在 VHDL 中不经常使用，可能是因为它们的功能非常有限：

- 您只能定义一块组合逻辑硬件，或者只能定义一块寄存器（如果您在时钟进程内调用函数/过程）。
- 您无法在其中定义流程。
- 您无法在其中实例化组件。
- 在这里面，您可以读/写的范围是有限的。

在 SpinalHDL 中，所有这些限制都被消除了。

在单个函数中混合使用组合逻辑和寄存器的示例：

```
def simpleAluPipeline(op: Bits, a: UInt, b: UInt): UInt = {
  val result = UInt(8 bits)

  switch(op) {
    is(0){ result := a + b }
    is(1){ result := a - b }
    is(2){ result := a * b }
  }

  return RegNext(result)
}
```

Stream 线束内的队列函数示例（带握手）。该函数实例化一个 FIFO 组件：

```
class Stream[T <: Data](dataType:  T) extends Bundle with IMasterSlave with␣
↪DataCarrier[T] {
  val valid = Bool()
  val ready = Bool()
  val payload = cloneOf(dataType)

  def queue(size: Int): Stream[T] = {
    val fifo = new StreamFifo(dataType, size)
    fifo.io.push <> this
    fifo.io.pop
  }
}
```

为在外部定义的信号赋值的示例函数：

```
val counter = Reg(UInt(8 bits)) init(0)
counter := counter + 1

def clear() : Unit = {
  counter := 0
}
```

(续下页)

(接上页)

```
when(counter > 42) {
  clear()
}
```

总线和接口

当谈到总线和接口时，VHDL 非常无聊。您有两个选择：

1) 随时随地逐线定义总线和接口：

```
PADDR   : in unsigned(addressWidth-1 downto 0);
PSEL    : in std_logic
PENABLE : in std_logic;
PREADY  : out std_logic;
PWRITE  : in std_logic;
PWDATA  : in std_logic_vector(dataWidth-1 downto 0);
PRDATA  : out std_logic_vector(dataWidth-1 downto 0);
```

2) 使用记录但无法参数化（静态固定在包中），并且您必须为每个信号定义方向：

```
P_m : in APB_M;
P_s : out APB_S;
```

SpinalHDL 对参数化总线和接口的声明提供非常强大的支持：

```
val P = slave(Apb3(addressWidth, dataWidth))
```

您还可以使用面向对象编程来定义专门的配置对象：

```
val coreConfig = CoreConfig(
  pcWidth = 32,
  addrWidth = 32,
  startAddress = 0x00000000,
  regFileReadyKind = sync,
  branchPrediction = dynamic,
  bypassExecute0 = true,
  bypassExecute1 = true,
  bypassWriteBack = true,
  bypassWriteBackBuffer = true,
  collapseBubble = false,
  fastFetchCmdPcCalculation = true,
  dynamicBranchPredictorCacheSizeLog2 = 7
)

// The CPU has a system of plugins which allows adding new features into the core.
// Those extensions are not directly implemented in the core, but are kind of an␣
↪additive logic patch defined in a separate area.
coreConfig.add(new MulExtension)
coreConfig.add(new DivExtension)
coreConfig.add(new BarrelShifterFullExtension)

val iCacheConfig = InstructionCacheConfig(
  cacheSize = 4096,
  bytePerLine = 32,
  wayCount = 1,  // Can only be one for the moment
  wrappedMemAccess = true,
  addressWidth = 32,
  cpuDataWidth = 32,
  memDataWidth = 32
```

(续下页)

(接上页)

```
)

new RiscvCoreAxi4(
  coreConfig = coreConfig,
  iCacheConfig = iCacheConfig,
  dCacheConfig = null,
  debug = debug,
  interruptCount = interruptCount
)
```

信号声明

VHDL 强制您在架构描述的顶部定义所有信号，这很烦人。

```
  ..
  .. (many signal declarations)
  ..
  signal a : std_logic;
  ..
  .. (many signal declarations)
  ..
begin
  ..
  .. (many logic definitions)
  ..
  a <= x & y
  ..
  .. (many logic definitions)
  ..
```

SpinalHDL 在信号声明方面非常灵活。

```
val a = Bool()
a := x & y
```

它还允许您在一行中定义和赋值信号。

```
val a = x & y
```

组件实例化

VHDL 对此非常冗长，因为您必须重新定义子组件实体的所有信号，然后在实例化组件时将它们一一绑定。

```
divider_cmd_valid : in std_logic;
divider_cmd_ready : out std_logic;
divider_cmd_numerator : in unsigned(31 downto 0);
divider_cmd_denominator : in unsigned(31 downto 0);
divider_rsp_valid : out std_logic;
divider_rsp_ready : in std_logic;
divider_rsp_quotient : out unsigned(31 downto 0);
divider_rsp_remainder : out unsigned(31 downto 0);

divider : entity work.UnsignedDivider
  port map (
    clk             => clk,
    reset           => reset,
    cmd_valid       => divider_cmd_valid,
```

(续下页)

(接上页)

```
    cmd_ready       => divider_cmd_ready,
    cmd_numerator   => divider_cmd_numerator,
    cmd_denominator => divider_cmd_denominator,
    rsp_valid       => divider_rsp_valid,
    rsp_ready       => divider_rsp_ready,
    rsp_quotient    => divider_rsp_quotient,
    rsp_remainder   => divider_rsp_remainder
  );
```

SpinalHDL 在这方面做出了很大提升，并允许您以面向对象的方式访问子组件的 IO。

```
val divider = new UnsignedDivider()

// And then if you want to access IO signals of that divider:
divider.io.cmd.valid := True
divider.io.cmd.numerator := 42
```

类型转换

VHDL 中有两种烦人的转换方法：

- boolean <> std_logic （例如：使用 `mySignal <= myValue < 10` 等条件赋值信号是不合法的）
- unsigned <> integer（例如：访问数组）

SpinalHDL 通过统一化对象来转换这些类型。

boolean/std_logic:

```
val value = UInt(8 bits)
val valueBiggerThanTwo = Bool()
valueBiggerThanTwo := value > 2  // value > 2 return a Bool
```

unsigned/integer:

```
val array = Vec(UInt(4 bits),8)
val sel = UInt(3 bits)
val arraySel = array(sel) // Arrays are indexed directly by using UInt
```

调整位宽

VHDL 对位宽限制严格可能是一件好事。

```
my8BitsSignal <= resize(my4BitsSignal, 8);
```

在 SpinalHDL 中，您有两种方法可以实现相同的目的：

```
// The traditional way
my8BitsSignal := my4BitsSignal.resize(8)

// The smart way
my8BitsSignal := my4BitsSignal.resized
```

参数化

2008 年修订版之前的 VHDL 在泛型方面存在许多问题。例如，您不能参数化记录，不能参数化实体中的数组，并且不能具有类型参数。
然后 VHDL 2008 出现并解决了这些问题。但根据供应商的不同，RTL 工具对 VHDL 2008 的支持确实很弱。

SpinalHDL 完全支持在其编译器中自然的集成泛型，并且它不依赖于 VHDL 泛型。

这是参数化数据结构的示例：

```
val colorStream = Stream(Color(5, 6, 5))
val colorFifo   = StreamFifo(Color(5, 6, 5), depth = 128)
colorFifo.io.push <> colorStream
```

以下是参数化组件的示例：

```
class Arbiter[T <: Data](payloadType: T, portCount: Int) extends Component {
  val io = new Bundle {
    val sources = Vec(slave(Stream(payloadType)), portCount)
    val sink = master(Stream(payloadType))
  }
  // ...
}
```

元硬件描述

VHDL 具有某种封闭的语法。您无法在其上添加抽象层。

而 SpinalHDL，由于它构建在 Scala 之上，所以非常灵活，并且允许您非常轻松地定义新的抽象层。

这种灵活性在后面的库中表现突出：*FSM* 库、*BusSlaveFactory* 库以及*JTAG* 库。

3.7.2 VHDL 等效语法

实体和架构

在 SpinalHDL 中，VHDL 实体和架构都在 `Component` 内定义。

这是一个具有 3 个输入（`a`、`b`、`c`）和一个输出（`result`）的组件。该组件还有一个 `offset` 构造参数（类似于 VHDL generic）。

```
case class MyComponent(offset: Int) extends Component {
  val io = new Bundle {
    val a, b, c = in UInt(8 bits)
    val result  = out UInt(8 bits)
  }
  io.result := a + b + c + offset
}
```

然后实例化该组件，您不需要绑定它：

```
case class TopLevel extends Component {
  ...
  val mySubComponent = MyComponent(offset = 5)

  ...
```

(续下页)

(接上页)

```
  mySubComponent.io.a := 1
  mySubComponent.io.b := 2
  mySubComponent.io.c := 3
  ??? := mySubComponent.io.result

  ...
}
```

数据类型

SpinalHDL 与 VHDL 相似的数据类型：

VHDL	SpinalHDL
std_logic	Bool
std_logic_vector	位
unsigned	UInt
signed	SInt

在 VHDL 中，要定义 8 位 `unsigned`，您必须给出位范围 `unsigned(7 downto 0)`，
而在 SpinalHDL 中，您只需提供位数 `UInt(8 bits)`。

VHDL	SpinalHDL
records	Bundle
array	Vec
enum	SpinalEnum

这是 SpinalHDL `Bundle` 定义的示例。`channelWidth` 是一个构造参数，类似于 VHDL 泛型，但用于数据结构：

```
case class RGB(channelWidth: Int) extends Bundle {
  val r, g, b = UInt(channelWidth bits)
}
```

例如，要实例化一个 `Bundle`，您需要这样写 `val myColor = RGB(channelWidth=8)`。

信号

这是一个关于信号实例化的示例：

```
case class MyComponent(offset: Int) extends Component {
  val io = new Bundle {
    val a, b, c = UInt(8 bits)
    val result  = UInt(8 bits)
  }
  val ab = UInt(8 bits)
  ab := a + b

  val abc = ab + c            // You can define a signal directly with its value
  io.result := abc + offset
}
```

赋值

在 SpinalHDL 中，:= 赋值运算符相当于 VHDL 信号赋值 (<=)：

```
val myUInt = UInt(8 bits)
myUInt := 6
```

在 VHDL 中，条件赋值通过使用 if/case 语句来完成：

```
val clear   = Bool()
val counter = Reg(UInt(8 bits))

when(clear) {
  counter := 0
}.elsewhen(counter === 76) {
  counter := 79
}.otherwise {
  counter(7) := ! counter(7)
}

switch(counter) {
  is(42) {
    counter := 65
  }
  default {
    counter := counter + 1
  }
}
```

字面量（Literals）

与 VHDL 中的字面量略有不同：

```
val myBool = Bool()
myBool := False
myBool := True
myBool := Bool(4 > 7)

val myUInt = UInt(8 bits)
myUInt := "0001_1100"
myUInt := "xEE"
myUInt := 42
myUInt := U(54,8 bits)
myUInt := ((3 downto 0) -> myBool, default -> true)
when(myUInt === U(myUInt.range -> true)) {
  myUInt(3) := False
}
```

寄存器

在 SpinalHDL 中，寄存器是显式指定的，而在 VHDL 中寄存器是推断的。以下是 SpinalHDL 寄存器的示例：

```
val counter = Reg(UInt(8 bits))  init(0)
counter := counter + 1   // Count up each cycle

// init(0) means that the register should be initialized to zero when a reset␣
↪occurs
```

过程块

过程块是一种仿真功能，对于设计 RTL 来说是不必要的。这就是为什么 SpinalHDL 不包含任何类似于过程块的功能，并且您可以在您想要的位置分配您想要的内容。

```
val cond = Bool()
val myCombinatorial = Bool()
val myRegister = UInt(8 bits)

myCombinatorial := False
when(cond) {
  myCombinatorial := True
  myRegister = myRegister + 1
}
```

3.8 快速参考

3.8.1 Core

重定向到 https://github.com/SpinalHDL/SpinalDoc/blob/master/cheatsheet/cheatSheet_core_oo.pdf

3.8.2 Lib

重定向到 https://github.com/SpinalHDL/SpinalDoc/blob/master/cheatsheet/cheatSheet_lib_oo.pdf

3.8.3 Symbolic

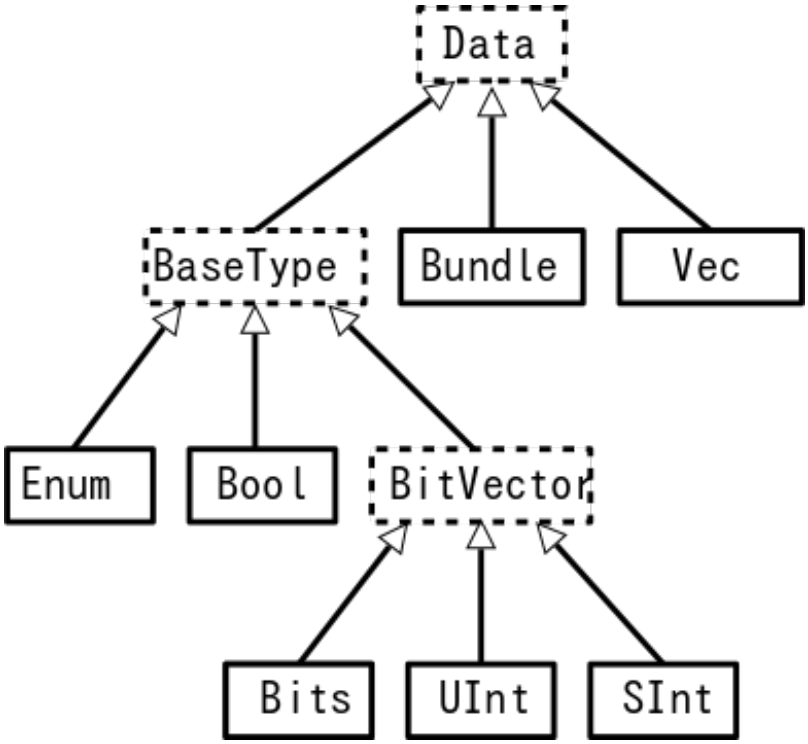
重定向到 https://github.com/SpinalHDL/SpinalDoc/blob/master/cheatsheet/cheatSheet_symbolic.pdf

CHAPTER 4

数据类型

该语言提供了 5 种基本类型和 2 种可以使用的复合类型。

- 基本类型有：*Bool* , *Bits* , 无符号整数*UInt* 对于，有符号整数*SInt* ， 以及*Enum*。
- 复合类型有：*Bundle* 有*Vec*。



除了基本类型之外，Spinal 还正在开发支持以下类型：

- *Fixed-point* 定点小数（部分支持）
- *Auto-range Fixed-point* 自动范围定点小数（支持加法、减法、乘法）
- *Floating-point* 浮点小数（实验性支持）

Additionally, if you want to assign a don’t care value to some hardware, for instance, to provide a default value, you can use the assignDontCare API to do so.

```
val myBits  = Bits(8 bits)
myBits.assignDontCare() // Will assign all the bits to 'x'
```

最后，还有一种特殊类型可用于检查 BitVector 是否符合位常量的指定模式，该模式包含像位掩码一样定义的空洞（位的位置不通过等式进行比较）。

下面是一个示例，展示如何实现此目的（注意使用“M”前缀）：

```
val myBits  = Bits(8 bits)
val itMatch = myBits === M"00--10--" // - for don't care value
```

4.1 Bool

4.1.1 描述

`Bool` 类型对应于硬件设计中使用的布尔值（True 或 False）或单个位/线。虽然名称类似，但不应与 Scala *Boolean* 类型混淆，后者不描述硬件，而是描述 Scala 生成器代码中的真值。

An important concept and rule-of-thumb to understand is that the Scala *Boolean* type is used in places where elaboration-time HDL code-generation decision making is occurring in Scala code. Like any regular program it affects execution of the Scala program that is SpinalHDL at the time the program is being run to perform HDL code generation.

因此，Scala *Boolean* 的值无法从硬件中观察到，因为它仅在 HDL 代码生成时存在于 SpinalHDL 程序中，这在硬件仿真/运行之前。

在您的设计中可能需要这样做，例如从 Scala 向硬件设计中传递一个值（该值可能作为参数化常量输入），您可以使用构造函数 *Bool(value: Boolean)* 将其类型转换为 *Bool* 。

同样，SpinalHDL *Bool* 的值在代码生成时无法看到，所有可以看到和操作的都是有关 *wire* 的 HDL 构造以及它如何路由（通过模块/组件）、驱动（源）并连接（汇据点）。

赋值运算符 := 的信号方向由 SpinalHDL 管理。在赋值运算符 := 左侧或右侧使用 *Bool* 实例指示它是给定赋值的源（提供状态）还是接收器（捕获状态）。

允许多次使用赋值运算符，因此信号线作为源（提供值以驱动 HDL 状态）连接并驱动其他 HDL 结构的多个输入是很正常的。当 Bool 实例用作源时，赋值语句在 Scala 中出现或执行的顺序并不重要，而当它用作汇（捕获状态）时则不同。

当多个赋值操作符驱动 Bool（Bool 位于赋值表达式的左侧）时，最后一个赋值语句将赢得胜利；并开始生效。最后一个将在 Scala 代码中最后执行。此事会影响 SpinalHDL Scala 代码的布局和排序，以确保在硬件设计中获得正确的优先顺序，以便在硬件中为 Bool 赋值新状态。

将 Scala/SpinalHDL *Bool* 实例作为对 net-list 中 HDL *net* 的引用，可能有助于理解这一概念。其中，赋值 := 操作符将 HDL 结构加入到同一个网中。

4.1.2 声明

声明布尔值的语法如下：（[] 之间的所有内容都是可选的）

语法	描述	返回类型
Bool()	创建 Bool 值	Bool
True	创建一个分配有 `true` 值的 Bool 对象	Bool
False	创建一个 Bool 值并赋值为 `false`	Bool
Bool(value: Boolean)	创建一个 Bool，并分配一个 Scala 布尔类型（true、false）的值。这显式地转换为 `True` 或 `False` 。	Bool

```
val myBool_1 = Bool()          // Create a Bool
myBool_1 := False              // := is the assignment operator (like verilog <=)

val myBool_2 = False           // Equivalent to the code above

val myBool_3 = Bool(5 > 12)    // Use a Scala Boolean to create a Bool
```

4.1.3 运算符

以下运算符可用于 Bool 类型:

逻辑运算

运算符	描述	返回类型
!x	逻辑非	Bool
x && y x & y	逻辑与	Bool
x || y x | y	逻辑或	Bool
x ^ y	逻辑异或	Bool
~x	逻辑非	Bool
x.set[()]	将 x 设置为 True	Unit (none)
x.clear[()]	将 x 设置为 False	Unit (none)
x.setWhen(cond)	当 cond 为 True 时设置 x 为 True	Bool
x.clearWhen(cond)	当 cond 为 True 时设置 x 为 False	Bool
x.riseWhen(cond)	当 x 为 False 并且 cond 为 True 时设置 x 为 True	Bool
x.fallWhen(cond)	当 x 为 True 并且 cond 为 True 时设置 x 为 False	Bool

```
val a, b, c = Bool()
val res = (!a & b) ^ c    // ((NOT a) AND b) XOR c

val d = False
when(cond) {
  d.set()                 // equivalent to d := True
}

val e = False
e.setWhen(cond)           // equivalent to when(cond) { d := True }

val f = RegInit(False) fallWhen(ack) setWhen(req)
/** equivalent to
 * when(f && ack) { f := False }
 * when(req) { f := True }
 * or
 * f := req || (f && !ack)
 */
```

(续下页)

(接上页)

```
// mind the order of assignments!  last one wins
val g = RegInit(False) setWhen(req) fallWhen(ack)
// equivalent to g := ((!g) && req) || (g && !ack)
```

边缘检测

所有边缘检测函数都将通过*RegNext* 实例化一个附加寄存器，以获取相关 `Bool` 的延迟值（一拍）。

此功能不会重新配置 D 型触发器以使用其他 CLK 时钟，它使用串联链中的两个 D 型触发器（两个 CLK 引脚都继承默认的 ClockDomain）。它具有组合逻辑，可根据输出 Q 状态进行边缘检测。

运算符	描述	返回类型
x.edge[()]	当 x 状态改变时返回 True	Bool
x.edge(initAt: Bool)	与 x.edge 相同但具有重置后的初始值	Bool
x.rise[()]	当 x 在上一个周期为低电平且现在为高电平时返回 True	Bool
x.rise(initAt: Bool)	与 x.rise 相同但具有重置后的初始值	Bool
x.fall[()]	当 x 在上一个周期为高且现在为低时返回 True	Bool
x.fall(initAt: Bool)	与 x.fall 相同但具有重置后的初始值	Bool
x.edges[()]	返回捆绑包（上升、下降、切换）	BoolEdges
x.edges(initAt: Bool)	与 x.edges 相同但具有重置后的初始值	BoolEdges
x.toggle[()]	在每个边缘返回 True	Bool

```
when(myBool_1.rise(False)) {
    // do something when a rising edge is detected
}


val edgeBundle = myBool_2.edges(False)
when(edgeBundle.rise) {
    // do something when a rising edge is detected
}
when(edgeBundle.fall) {
    // do something when a falling edge is detected
}
when(edgeBundle.toggle) {
    // do something at each edge
}
```

比较运算

运算符	描述	返回类型
x === y	等价性判断	Bool
x =/= y	不等价判断运算	Bool

```
when(myBool) { // Equivalent to when(myBool === True)
    // do something when myBool is True
}

when(!myBool) { // Equivalent to when(myBool === False)
    // do something when myBool is False
}
```

类型转换

运算符	描述	返回类型
x.asBits	二进制转换为 Bits	Bits(1 bit)
x.asUInt	二进制转换为 UInt	UInt(1 bit)
x.asSInt	二进制转换为 SInt	SInt(1 bit)
x.asUInt(bitCount)	二进制转换为 UInt 并调整大小，将 Bool 值放入 LSB 并用零填充。	UInt(bitCount bits)
x.asBits(bitCount)	二进制转换为位并调整大小，将布尔值放入 LSB 并用零填充。	Bits(bitCount bits)

```
// Add the carry to an SInt value
val carry = Bool()
val res = mySInt + carry.asSInt
```

杂项

运算符	描述	返回类型
x ## y	连接 Bits，x-> 高位，y-> 低位	Bits(w(x) + w(y) bits)
x #* n	n 次重复 x 并合并	Bits(n bits)

```
val a, b, c = Bool()

// Concatenation of three Bool into a single Bits(3 bits) type
val myBits = a ## b ## c
```

掩码布尔值

具有掩码的布尔型允许任意值（don’t care）。它们通常不单独使用，而是通过*MaskedLiteral* 使用。

```
// first argument: Scala Boolean value
// second argument: do we care ? expressed as a Scala Boolean
val masked = new MaskedBoolean(true, false)
```

4.2 位

`Bits` 类型表示多位向量，不传达任何算术含义。

4.2.1 声明

声明位向量的语法如下（[] 之间的所有内容都是可选的）：

语法	描述
Bits [()]	创建 Bits，其位数是从构造后最宽的赋值语句推断出来的
Bits(x bits)	创建具有 x 位的 Bits
B(value: Int[, x bits]) B(value: BigInt[, x bits])	创建 x 位 Bits，且赋值为’value’
B”[[size’]base]value”	创建 Bits 并赋值为’value’（基数：“h”、“d”、“o”、“b”）
B([x bits,] elements: Element*)	创建并赋值由*elements* 指定值的 Bits

```
val myBits1 = Bits(32 bits)
val myBits2 = B(25, 8 bits)
val myBits3 = B"8'xFF"    // Base could be x,h (base 16)
                          //               d   (base 10)
                          //               o   (base 8)
                          //               b   (base 2)
val myBits4 = B"1001_0011"  // _ can be used for readability

// Bits with all ones ("11111111")
val myBits5 = B(8 bits, default -> True)

// initialize with "10111000" through a few elements
val myBits6 = B(8 bits, (7 downto 5) -> B"101", 4 -> true, 3 -> True, default ->␣
↪false)

// "10000000" (For assignment purposes, you can omit the B)
val myBits7 = Bits(8 bits)
myBits7 := (7 -> true, default -> false)
```

当推断 Bits 的宽度时，赋予值的宽度仍然必须与信号的最终宽度相匹配：

```
// Declaration
val myBits = Bits()      // the size is inferred from the widest assignment
// ....
// .resized needed to prevent WIDTH MISMATCH error as the constants
// width does not match size that is inferred from assignment below
myBits := B("1010").resized  // auto-widen Bits(4 bits) to Bits(6 bits)
when(condxMaybe) {
  // Bits(6 bits) is inferred for myBits, this is the widest assignment
  myBits := B("110000")
}
```

4.2.2 运算符

以下运算符可用于 Bits 类型：

逻辑运算

运算符	描述	返回类型
~x	按位非	Bits(w(x) bits)
x & y	按位与	Bits(w(xy) bits)
x | y	按位或	Bits(w(xy) bits)
x ^ y	按位异或	Bits(w(xy) bits)
x.xorR	对 x 的所有位进行异或	Bool
x.orR	对 x 的所有位做或运算	Bool
x.andR	对 x 的所有位做与运算	Bool

y = 1 // 整数
x » y
	逻辑右移，y: Int
	结果的宽度可能会变少
		Bits(w(x) - y bits)

y = U(1) // UInt
x » y
	逻辑右移，y: UInt
	结果宽度相同
		Bits(w(x) bits)

y = 1 // 整数
x « y
	逻辑左移，y: Int
	结果的宽度可能会增加
		Bits(w(x) + y bits)

y = U(1) // UInt
x « y
	逻辑左移，y: UInt
	结果的宽度可能会增加
		Bits(w(x) + max(y) bits)

x |» y	逻辑右移，y: Int/UInt 结果宽度相同	Bits(w(x) bits)
x |« y	逻辑左移，y: Int/UInt 结果宽度相同	Bits(w(x) bits)
x.rotateLeft(y)	逻辑循环左移，y: UInt/Int 结果宽度相同	Bits(w(x) bits)
x.rotateRight(y)	逻辑循环右移，y：UInt/Int 结果宽度相同	Bits(w(x) bits)
x.clearAll[()]	清零所有位	修改 *x*
x.setAll[()]	将所有的位设置为 1	修改 *x*
x.setAllTo(value: Boolean)	将所有位设置为给定的布尔值（Scala Boolean）	修改 *x*
x.setAllTo(value: Bool)	将所有位设置为给定的布尔值（Spinal Bool）	修改 *x*

```
// Bitwise operator
val a, b, c = Bits(32 bits)
c := ~(a & b) // Inverse(a AND b)

val all_1 = a.andR // Check that all bits are equal to 1

// Logical shift
val bits_10bits = bits_8bits << 2  // shift left (results in 10 bits)
val shift_8bits = bits_8bits |<< 2 // shift left (results in 8 bits)

// Logical rotation
val myBits = bits_8bits.rotateLeft(3) // left bit rotation

// Set/clear
val a = B"8'x42"
when(cond) {
  a.setAll() // set all bits to True when cond is True
}
```

比较运算

运算符	描述	返回类型
x === y	等价性判断	Bool
x =/= y	不等价判断运算	Bool

```
when(myBits === 3) {
  // ...
}

val notMySpecialValue = myBits_32 =/= B"32'x44332211"
```

类型转换

运算符	描述	返回类型
x.asBits	二进制转换为 Bits	Bits(w(x) bits)
x.asUInt	二进制转换为 UInt	UInt(w(x) bits)
x.asSInt	二进制转换为 SInt	SInt(w(x) bits)
x.asBools	转换为一个布尔数组	Vec(Bool(), w(x))
x.asBool	提取 x 的 LSB	Bool(x.lsb)
B(x: T)	将数据转换为 Bits	Bits(w(x) bits)

要将 Bool、UInt 或 SInt 转换为 Bits，您可以使用 B(something) 或 B(something[, x bits])：

```
// cast a Bits to SInt
val mySInt = myBits.asSInt

// create a Vector of bool
val myVec = myBits.asBools

// Cast a SInt to Bits
val myBits = B(mySInt)

// Cast the same SInt to Bits but resize to 3 bits
//  (will expand/truncate as necessary, retaining LSB)
val myBits = B(mySInt, 3 bits)
```

位提取

所有位提取操作均可用于读取一个位/一组位。与其他 HDL 一样，提取运算符也可用于为 Bits 的一部分赋值。

所有位提取操作均可用于读取一个位/一组位。与其他 HDL 一样，它们也可用于选择要写入位的范围。

运算符	描述	返回类型
x(y: Int)	静态访问第 y 位	Bool
x(y: UInt)	访问第 y 位，这里 y 为可变的信号	Bool
x(offset: Int, width bits)	固定地选择偏移量和宽度，offset 为选择信号中 LSB 的索引	Bits(width bits)
x(offset: UInt, width bits)	选择偏移量可变和宽度固定的信号，offset 为选择信号中 LSB 的索引（可为另一信号）	Bits(width bits)
x(range: Range)	访问 Bits 的范围 。例如：myBits(4 downto 2)	Bits(range.size bits)
x.subdivideIn(y slices, [strict: Boolean])	将 x 分割为 y 片，y: Int	Vec(Bits(⋯), y)
x.subdivideIn(y bits, [strict: Boolean])	将 x 分割为 y 位的多个切片，y: Int	Vec(Bits(y bit), ⋯)
x.msb	访问 x 的最高有效位（最高索引）	Bool
x.lsb	访问 x 的最低有效位（索引 0）	Bool

一些基本示例：

```
// get the element at the index 4
val myBool = myBits(4)
// assign element 1
myBits(1) := True

// index dynamically
val index = UInt(2 bit)
val indexed = myBits(index, 2 bit)

// range index
val myBits_8bit = myBits_16bit(7 downto 0)
val myBits_7bit = myBits_16bit(0 to 6)
val myBits_6bit = myBits_16bit(0 until 6)
// assign to myBits_16bit(3 downto 0)
myBits_8bit(3 downto 0) := myBits_4bit

// equivalent slices, no reversing occurs
val a = myBits_16bit(8 downto 4)
val b = myBits_16bit(4 to 8)

// read / assign the msb / leftmost bit / x.high bit
val isNegative = myBits_16bit.msb
myBits_16bit.msb := False
```

分割细节

两个 subdivideIn 函数的所有参数都有一个可选参数 strict 参数（即 subdivideIn(slices: SlicesCount, strict: Boolean = true) ）。如果 strict 为 true，则如果输入无法等分，将引发错误。如果设置为 false，最后一个元素可能比其他（大小相等）元素小。

```
// Subdivide
val sel = UInt(2 bits)
val myBitsWord = myBits_128bits.subdivideIn(32 bits)(sel)
    // sel = 3 => myBitsWord = myBits_128bits(127 downto 96)
```

(续下页)

(接上页)

```
    // sel = 2 => myBitsWord = myBits_128bits( 95 downto 64)
    // sel = 1 => myBitsWord = myBits_128bits( 63 downto 32)
    // sel = 0 => myBitsWord = myBits_128bits( 31 downto  0)

// If you want to access in reverse order you can do:
val myVector    = myBits_128bits.subdivideIn(32 bits).reverse
val myRevBitsWord = myVector(sel)

// We can also assign through subdivides
val output8 = Bits(8 bit)
val pieces = output8.subdivideIn(2 slices)
// assign to output8
pieces(0) := 0xf
pieces(1) := 0x5
```

杂项

与上面列出的位提取操作相反，上述函数不能使用其返回值给原始信号赋值。

运算符	描述	返回类型
x.getWidth	返回位数	Int
x.bitsRange	返回范围（0 到 x.high）	范围
x.valueRange	返回最小到最大 x 值的范围，理解为无符号整数（0 到 2 ** width - 1）。	范围
x.high	返回 MSB（最高有效位）的索引（x 的最高索引，该索引从 0 开始计数）	Int
x.reversed	返回 x 的副本，其位顺序相反，MSB<>LSB 是镜像的。	Bits(w(x) bits)
x ## y	连接 Bits，x-> 高位，y-> 低位	Bits(w(x) + w(y) bits)
x #* n	n 次重复 x 并合并	Bits(w(x) * n bits)
x.resize(y)	返回一个新的信号与 x 信号直接连接但位宽变成了 y 位。如果位宽变大了，则根据需要在 MSB 处用零填充进行扩展，y: Int	Bits(y bits)
x.resized	返回一个允许自动调整位宽的 x 的副本信号。调整位宽操作被推迟到稍后的赋值操作。调整位宽可能会加宽或截断原信号，但保留 LSB。	Bits(w(x) bits)
x.resizeLeft(x)	调整位宽时保持 MSB 位置不变，x:Int 调整位宽可能会加宽或截断信号，同时保留 MSB。	Bits(x bits)
x.getZero	返回新的 Bits 的实例，该实例被分配了与 x 宽度相同的 0 值（常量）。	Bits(0, w(x) bits)
x.getAllTrue	返回 Bits 的新实例，该实例被赋予了与 x 宽度相同的 1 值（常量）。	Bits(w(x) bits).setAll()

备注： *validRange* 只能用于最小值和最大值能够保存在 32 位有符号整数的情况下。（这是由于 Scala `scala.collection.immutable.Range` 类型使用 *Int* 作为范围描述）

```
println(myBits_32bits.getWidth) // 32

// Concatenation
myBits_24bits := bits_8bits_1 ## bits_8bits_2 ## bits_8bits_3
// or
myBits_24bits := Cat(bits_8bits_1, bits_8bits_2, bits_8bits_3)

// Resize
```

(续下页)

(接上页)

```
myBits_32bits := B"32'x112233344"
myBits_8bits  := myBits_32bits.resized       // automatic resize (myBits_8bits =␣
↪0x44)
myBits_8bits  := myBits_32bits.resize(8)     // resize to 8 bits (myBits_8bits =␣
↪0x44)
myBits_8bits  := myBits_32bits.resizeLeft(8) // resize to 8 bits (myBits_8bits =␣
↪0x11)
```

4.2.3 掩码字面量

MaskedLiteral 值带有”不关心”值的位向量，其中”不关心”值用 - 表示。它们可用于直接比较或用于 switch 和 mux 等语句。

```
val myBits = B"1101"

val test1 = myBits === M"1-01" // True
val test2 = myBits === M"0---" // False
val test3 = myBits === M"1--1" // True
```

4.3 UInt/SInt

UInt/SInt 类型用于表达二进制补码无符号/有符号整数的位向量。他们可以做 Bits 相同的事情，但具有无符号/有符号整数算术和比较。

4.3.1 声明

以下是声明一个整数的语法：（[] 中的内容是可选的）

语法	描述
UInt[()] SInt[()]	创建一个无符号/有符号整数，自动推断位数
UInt(x bits) SInt(x bits)	创建一个 x 位的无符号/有符号整数
U(value: Int[,x bits]) U(value: BigInt[,x bits]) S(value: Int[,x bits]) S(value: BigInt[,x bits])	创建一个无符号/有符号整数，并将其分配给‘value’
U”[[size’]base]value” S”[[size’]base]value”	创建一个无符号/有符号整数，并将其分配给‘value’ (base: ‘h’，‘d’，‘o’，‘b’)
U([x bits,] elements: Element*) S([x bits,] elements: Element*)	创建一个无符号整数，并为其赋值一个由*elements* 指定的值

```
val myUInt = UInt(8 bit)
myUInt := U(2, 8 bit)
myUInt := U(2)
myUInt := U"0000_0101"  // Base per default is binary => 5
myUInt := U"h1A"        // Base could be x (base 16)
                        //               h (base 16)
                        //               d (base 10)
                        //               o (base 8)
                        //               b (base 2)
myUInt := U"8'h1A"
myUInt := 2             // You can use a Scala Int as a literal value

val myBool = Bool()
myBool := myUInt === U(7 -> true, (6 downto 0) -> false)
myBool := myUInt === U(8 bit, 7 -> true, default -> false)
myBool := myUInt === U(myUInt.range -> true)

// For assignment purposes, you can omit the U/S
// which also allows the use of "default -> ???"
myUInt := (default -> true)                          // Assign myUInt with "11111111"
myUInt := (myUInt.range -> true)                     // Assign myUInt with "11111111"
myUInt := (7 -> true, default -> false)              // Assign myUInt with "10000000"
myUInt := ((4 downto 1) -> true, default -> false) // Assign myUInt with "00011110"
```

4.3.2 运算符

以下运算符可用于 UInt 和 SInt 类型：

逻辑运算

运算符	描述	返回类型
~x	按位非	T(w(x) bits)
x & y	按位与	T(max(w(x), w(y)) bits)
x | y	按位或	T(max(w(x), w(y)) bits)
x ^ y	按位异或	T(max(w(x), w(y)) bits)
x.xorR	对 x 的所有位进行异或（缩减运算符）	Bool
x.orR	对 x 的所有位进行或操作（缩减运算符）	Bool
x.andR	对 x 的所有位进行与操作（缩减运算符）	Bool
x » y	算术右移，y : Int	T(w(x) - y bits)
x » y	算术右移，y : UInt	T(w(x) bits)
x « y	算术左移，y : Int	T(w(x) + y bits)
x « y	算术左移，y : UInt	T(w(x) + max(y) bits)
x |» y	逻辑右移，y : Int/UInt	T(w(x) bits)
x |« y	逻辑左移，y : Int/UInt	T(w(x) bits)
x.rotateLeft(y)	逻辑循环左移，y : UInt/Int y 的宽度被限制为 log2Up(x) 的宽度或更小	T(w(x) bits)
x.rotateRight(y)	逻辑循环右移，y : UInt/Int y 的宽度被限制为 log2Up(x) 的宽度或更小	T(w(x) bits)
x.clearAll[()]	清零所有位	修改 *x*
x.setAll[()]	将所有的位设置为 1	修改 *x*
x.setAllTo(value : Boolean)	将所有位设置为给定的布尔值（Scala Boolean）	修改 *x*
x.setAllTo(value : Bool)	将所有位设置为给定的布尔值（Spinal Bool）	修改 *x*

备注： Notice the difference in behavior between `x >> 2` (result 2 bit narrower than x) and `x >> U(2)` (keeping width) due to the Scala type of `y`.

在第一种情况下，“2” 是一个 Int 的值（可以看作是 “实例细化整数常量”），在第二种情况下，它是一个硬件信号（UInt 类型）这也可能不是一个常数。

```
val a, b, c = SInt(32 bits)
a := S(5)
b := S(10)

// Bitwise operators
c := ~(a & b)      // Inverse(a AND b)
assert(c.getWidth == 32)

// Shift
val arithShift = UInt(8 bits) << 2      // shift left (resulting in 10 bits)
val logicShift = UInt(8 bits) |<< 2     // shift left (resulting in 8 bits)
assert(arithShift.getWidth == 10)
assert(logicShift.getWidth == 8)
```

(续下页)

(接上页)

```
// Rotation
val rotated = UInt(8 bits) rotateLeft 3 // left bit rotation
assert(rotated.getWidth == 8)

// Set all bits of b to True when all bits of a are True
when(a.andR) { b.setAll() }
```

算术运算

运算符	描述	返回类型
x + y	加法	T(max(w(x), w(y)) bits)
x +^ y	带进位的加法	T(max(w(x), w(y)) + 1 bits)
x +| y	添加带有饱和（saturation）的加数（另请参见 *T.maxValue* 和 *T.minValue*）	T(max(w(x), w(y)) bits)
x - y	减法	T(max(w(x), w(y)) bits)
x -^ y	带进位的减法	T(max(w(x), w(y)) + 1 bits)
x -| y	带饱和（saturation）的减法（另请参见 *T.minValue* 和 *T.maxValue*）	T(max(w(x), w(y)) bits)
x * y	乘法	T(w(x) + w(y)) bits)
x / y	除法	T(w(x) bits)
x % y	求模运算	T(min(w(x), w(y)) bits)
~x	一元补码运算，按位非（NOT）	T(w(x) bits)
-x	SInt 类型的一元二进制补码。不适用于 UInt。	SInt(w(x) bits)

```
val a, b, c = UInt(8 bits)
a := U"xf0"
b := U"x0f"

c := a + b
assert(c === U"8'xff")

val d = a +^ b
assert(d === U"9'x0ff")

// 0xf0 + 0x20 would overflow, the result therefore saturates
val e = a +| U"8'x20"
assert(e === U"8'xff")
```

备注： 请注意此处如何进行仿真时判断（使用 ===），而不是前面示例中的细化时判断（使用 ==）。

比较运算

运算符	描述	返回类型
x === y	等价性判断	Bool
x =/= y	不等价判断运算	Bool
x > y	大于	Bool
x >= y	大于或等于	Bool
x < y	小于	Bool
x <= y	小于或等于	Bool

```
val a = U(5, 8 bits)
val b = U(10, 8 bits)
val c = UInt(2 bits)

when (a > b) {
  c := U"10"
} elsewhen (a =/= b) {
  c := U"01"
} elsewhen (a === U(0)) {
  c.setAll()
} otherwise {
  c.clearAll()
}
```

备注： 当比较 UInt 值时，允许“环绕”行为，即当值超过最大值时，它们将”环绕”到最小值。在这种情况下，可以使用 UInt 的 wrap 方法。对于 UInt 变量 x、y，如果 x.wrap < y，则如果 x 在环绕意义上小于 y，结果为真。

类型转换

运算符	描述	返回类型
x.asBits	二进制转换为 Bits	Bits(w(x) bits)
x.asUInt	二进制转换为 UInt	UInt(w(x) bits)
x.asSInt	二进制转换为 SInt	SInt(w(x) bits)
x.asBools	转换为 Bool 数组	Vec(Bool(), w(x))
x.asBool	提取 x 的 LSB	Bool(x.lsb)
S(x: T)	将数据转换为 SInt	SInt(w(x) bits)
U(x: T)	将数据转换为 UInt	UInt(w(x) bits)
x.intoSInt	转换为 SInt，扩展符号位	SInt(w(x) + 1 bits)
myUInt.twoComplement(en: Bool)	如果 en 为真，则生成二进制补码的数值，否则不变。(en 使结果为负数)	SInt(w(myUInt) + 1, bits)
mySInt.abs	以 UInt 值形式返回绝对值	UInt(w(mySInt) bits)
mySInt.abs(en: Bool)	当 en 为真时，返回 UInt 类型的绝对值，否则，只需将位解释为无符号数。	UInt(w(mySInt) bits)
mySInt.absWithSym	返回对称的 UInt 值的绝对值，缩小 1 位	UInt(w(mySInt) - 1 bits)

要将一个 Bool、一个 Bits 或一个 SInt 转换为一个 UInt，可以使用 U(something)。要将东西转换为一个 SInt，可以使用 S(something)。

```
// Cast an SInt to Bits
val myBits = mySInt.asBits
```

(续下页)

(接上页)

```
// Create a Vector of Bool
val myVec = myUInt.asBools

// Cast a Bits to SInt
val mySInt = S(myBits)

// UInt to SInt conversion
val UInt_30 = U(30, 8 bit)

val SInt_30 = UInt_30.intoSInt
assert(SInt_30 === S(30, 9 bit))

mySInt := UInt_30.twoComplement(booleanDoInvert)
    // if booleanDoInvert is True then we get S(-30, 9 bit)
    // otherwise we get S(30, 9 bit)

// absolute values
val SInt_n_4 = S(-3, 3 bit)
val abs_en = SInt_n_3.abs(booleanDoAbs)
    // if booleanDoAbs is True we get U(3, 3 bit)
    // otherwise we get U"3'b101" or U(5, 3 bit) (raw bit pattern of -3)

val SInt_n_128 = S(-128, 8 bit)
val abs = SInt_n_128.abs
assert(abs === U(128, 8 bit))
val sym_abs = SInt_n_128.absWithSym
assert(sym_abs === U(127, 7 bit))
```

位提取

所有位提取操作均可用于读取一个位/一组位。与其他 HDL 一样，提取运算符也对 `UInt` / `SInt` 的一部分赋值。

运算符	描述	返回类型
x(y: Int)	静态访问第 y 位	Bool
x(y: UInt)	访问第 y 位，这里 y 为可变的信号	Bool
x(offset: Int, width bits)	固定地选择偏移量和宽度，`offset` 为选择信号中 LSB 的索引	Bits(width bits)
x(offset: UInt, width bits)	选择偏移量可变和宽度固定的信号，`offset` 为选择信号中 LSB 的索引（可为另一信号）	Bits(width bits)
x(range: Range)	访问 Bits 的范围 。例如：myBits(4 downto 2)	Bits(range.size bits)
x.subdivideIn(y slices, [strict: Boolean])	将 x 分割为 y 片，y: Int	Vec(Bits(…), y)
x.subdivideIn(y bits, [strict: Boolean])	将 x 分割为 y 位的多个切片，y: Int	Vec(Bits(y bit), …)
x.msb	访问 x 的最高有效位（最高索引，SInt 的符号位）	Bool
x.lsb	访问 x 的最低有效位（索引 0）	Bool
mySInt.sign	访问最高符号位，仅适用于 SInt。	Bool

一些基本示例：

```
// get the element at the index 4
val myBool = myUInt(4)
// assign element 1
myUInt(1) := True
```

(续下页)

(接上页)

```
// index dynamically
val index = UInt(2 bit)
val indexed = myUInt(index, 2 bit)

// range index
val myUInt_8bit = myUInt_16bit(7 downto 0)
val myUInt_7bit = myUInt_16bit(0 to 6)
val myUInt_6bit = myUInt_16bit(0 until 6)
// assign to myUInt_16bit(3 downto 0)
myUInt_8bit(3 downto 0) := myUInt_4bit

// equivalent slices, no reversing occurs
val a = myUInt_16bit(8 downto 4)
val b = myUInt_16bit(4 to 8)

// read / assign the msb / leftmost bit / x.high bit
val isNegative = mySInt_16bit.sign
myUInt_16bit.msb := False
```

分割细节

两个 subdivideIn 函数的所有参数都有一个可选参数 strict 参数（即 subdivideIn(slices: SlicesCount, strict: Boolean = true)）。如果 strict 为 true，则如果输入无法等分，将引发错误。如果设置为 false，最后一个元素可能比其他（大小相等）元素小。

```
// Subdivide
val sel = UInt(2 bits)
val myUIntWord = myUInt_128bits.subdivideIn(32 bits)(sel)
    // sel = 3 => myUIntWord = myUInt_128bits(127 downto 96)
    // sel = 2 => myUIntWord = myUInt_128bits( 95 downto 64)
    // sel = 1 => myUIntWord = myUInt_128bits( 63 downto 32)
    // sel = 0 => myUIntWord = myUInt_128bits( 31 downto  0)

 // If you want to access in reverse order you can do:
 val myVector    = myUInt_128bits.subdivideIn(32 bits).reverse
 val myRevUIntWord = myVector(sel)

 // We can also assign through subdivides
 val output8 = UInt(8 bit)
 val pieces = output8.subdivideIn(2 slices)
 // assign to output8
 pieces(0) := 0xf
 pieces(1) := 0x5
```

杂项

与上面列出的位提取操作相反，上述函数不能使用其返回值给原始信号赋值。

运算符	描述	返回类型
x.getWidth	返回位数	Int
x.high	返回 MSB 的索引（对 Int 来说是允许的最高索引）	Int
x.bitsRange	返回范围（0 到 x.high）	范围
x.minValue	x 的最低可能值（例如 UInt 为 0）	BigInt
x.maxValue	x 的最大可能值	BigInt
x.valueRange	返回 x 的最小到最大可能值的范围（x.minValue 到 x.maxValue）。	范围
x ## y	连接 Bits，x-> 高位，y-> 低位	Bits(w(x) + w(y) bits)
x #* n	n 次重复 x 并合并	Bits(w(x) * n bits)
x @@ y	将 x:T 与 y:Bool/SInt/UInt 连接	T(w(x) + w(y) bits)
x.resize(y)	返回 x 调整大小后的副本，如果位宽变大，则用零填充其他位 对于 UInt 或 SInt（用符号填充）操作, y: Int	T(y bits)
x.resized	返回自动位宽调整后的 x 根据需要调整大小	T(w(x) bits)
x.expand	返回 x 并进行 1 位扩展	T(w(x)+1 bits)
x.getZero	返回类型 T 的新实例，该实例被分配与 x 相同宽度的零值（常量）。	T(0, w(x) bits).clearAll()
x.getAllTrue	返回类型 T 的新实例，该实例被分配了与 x 宽度相同的常量值。	T(w(x) bits).setAll()

备注： *validRange* 只能用于最小值和最大值能够保存在 32 位有符号整数的情况下。（这是由于 Scala `scala.collection.immutable.Range` 类型使用 *Int* 作为范围描述）

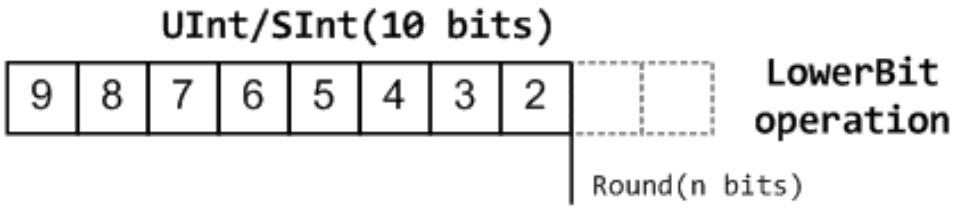
```
myBool := mySInt.lsb  // equivalent to mySInt(0)

// Concatenation
val mySInt = mySInt_1 @@ mySInt_1 @@ myBool
val myBits = mySInt_1 ## mySInt_1 ## myBool

// Resize
myUInt_32bits := U"32'x112233344"
myUInt_8bits  := myUInt_32bits.resized      // automatic resize (myUInt_8bits =␣
↪0x44)
val lowest_8bits = myUInt_32bits.resize(8)  // resize to 8 bits (myUInt_8bits =␣
↪0x44)
```

4.3.3 定点小数操作

对于定点小数，我们可以将其分为两部分：

- 低位运算（舍入方法）
- 高位运算（饱和运算）

低位运算



关于舍入运算: https://en.wikipedia.org/wiki/Rounding

SpinalHDL 中的名称	维基百科中的名称	API	数学算法描述	返回类型 (align=false)	支持情况
FLOOR	RoundDown	floor	floor(x)	w(x)-n bits	是
FLOORTOZERO	RoundToZero	floor-ToZero	sign*floor(abs(x))	w(x)-n bits	是
CEIL	RoundUp	ceil	ceil(x)	w(x)-n+1 bits	是
CEILTOINF	RoundToInf	ceilToInf	sign*ceil(abs(x))	w(x)-n+1 bits	是
ROUNDUP	RoundHalfUp	roundUp	floor(x+0.5)	w(x)-n+1 bits	是
ROUNDDOWN	RoundHalf-Down	round-Down	ceil(x-0.5)	w(x)-n+1 bits	是
ROUNDTOZERO	Round-HalfToZero	round-ToZero	sign*ceil(abs(x)-0.5)	w(x)-n+1 bits	是
ROUNDTOINF	RoundHalfToInf	round-ToInf	sign*floor(abs(x)+0.5)	w(x)-n+1 bits	是
ROUNDTOEVEN	RoundHalfTo-Even	roundTo-Even			不支持
ROUNDTOODD	Round-HalfToOdd	round-ToOdd			不支持

备注： **RoundToEven** 和 **RoundToOdd** 模式非常特殊，用于一些精度要求较高的大数据统计领域，SpinalHDL 尚不支持。

你会发现 *ROUNDUP*、*ROUNDDOWN*、*ROUNDTOZERO*、*ROUNDTOINF*、*ROUNDTOEVEN*、*ROUNTOODD* 在行为上非常接近，*ROUNDTOINF* 是最常见的。不同编程语言中的舍入行为可能不同。

编程语言	默认舍入类型	示例	评论
Matlab	ROUNDTOINF	round(1.5)=2,round(2.5)=3;round(-1.5)=-2,round(-2.5)=-3	四舍五入至 ± 无穷大
python2	ROUNDTOINF	round(1.5)=2,round(2.5)=3;round(-1.5)=-2,round(-2.5)=-3	四舍五入至 ± 无穷大
蟒蛇 3	ROUNDTO-EVEN	round(1.5)=round(2.5)=2; round(-1.5)=round(-2.5)=-2	向偶数舍入
Scala.math	ROUNDTOUP	round(1.5)=2,round(2.5)=3;round(-1.5)=-1,round(-2.5)=-2	永远向正无穷舍入
Spinal-HDL	ROUNDTOINF	round(1.5)=2,round(2.5)=3;round(-1.5)=-2,round(-2.5)=-3	四舍五入至 ± 无穷大

备注： 在 SpinalHDL 中，*ROUNDTOINF* 是默认的舍入类型 (`round = roundToInf`)

```
val A  = SInt(16 bits)
val B  = A.roundToInf(6 bits)          // default 'align = false' with carry, got 11 bit
val B  = A.roundToInf(6 bits, align = true) // sat 1 carry bit, got 10 bit
val B  = A.floor(6 bits)               // return 10 bit
val B  = A.floorToZero(6 bits)         // return 10 bit
val B  = A.ceil(6 bits)                // ceil with carry so return 11 bit
val B  = A.ceil(6 bits, align = true) // ceil with carry then sat 1 bit return 10 bit
val B  = A.ceilToInf(6 bits)
val B  = A.roundUp(6 bits)
val B  = A.roundDown(6 bits)
val B  = A.roundToInf(6 bits)
val B  = A.roundToZero(6 bits)
val B  = A.round(6 bits)               // SpinalHDL uses roundToInf as the default rounding mode

val B0 = A.roundToInf(6 bits, align = true)          //  ---+
                                                     //     |--> equal
val B1 = A.roundToInf(6 bits, align = false).sat(1) //  ---+
```

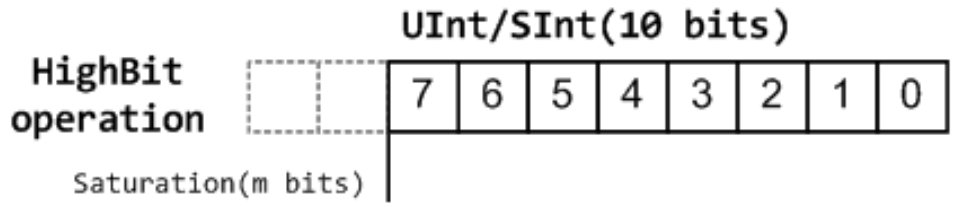
备注： 只有 `floor` 和 `floorToZero` 可以在没有 `align` 选项的情况下工作；他们不需要进位位。其他舍入操作默认使用进位位。

round Api

API	UInt/SInt	描述	返回类型 (align=false)	返回类型 (align=true)
floor	均支持		w(x)-n bits	w(x)-n bits
floorToZero	SInt	等于 UInt 类型的下限	w(x)-n bits	w(x)-n bits
ceil	均支持		w(x)-n+1 bits	w(x)-n bits
ceilToInf	SInt	等于 UInt 类型的 ceil 值	w(x)-n+1 bits	w(x)-n bits
roundUp	均支持	硬件实现简单	w(x)-n+1 bits	w(x)-n bits
roundDown	均支持		w(x)-n+1 bits	w(x)-n bits
roundToInf	SInt	最常使用	w(x)-n+1 bits	w(x)-n bits
round-ToZero	SInt	等于 UInt 类型的 roundDown	w(x)-n+1 bits	w(x)-n bits
round	均支持	SpinalHDL 中等效于 round-ToInf	w(x)-n+1 bits	w(x)-n bits

备注： 虽然 `roundToInf` 很常见，但 `roundUp` 的成本最低，时序也好，几乎没有性能损失。因此，强烈建议在生产环境中使用 `roundUp` 。

高位操作



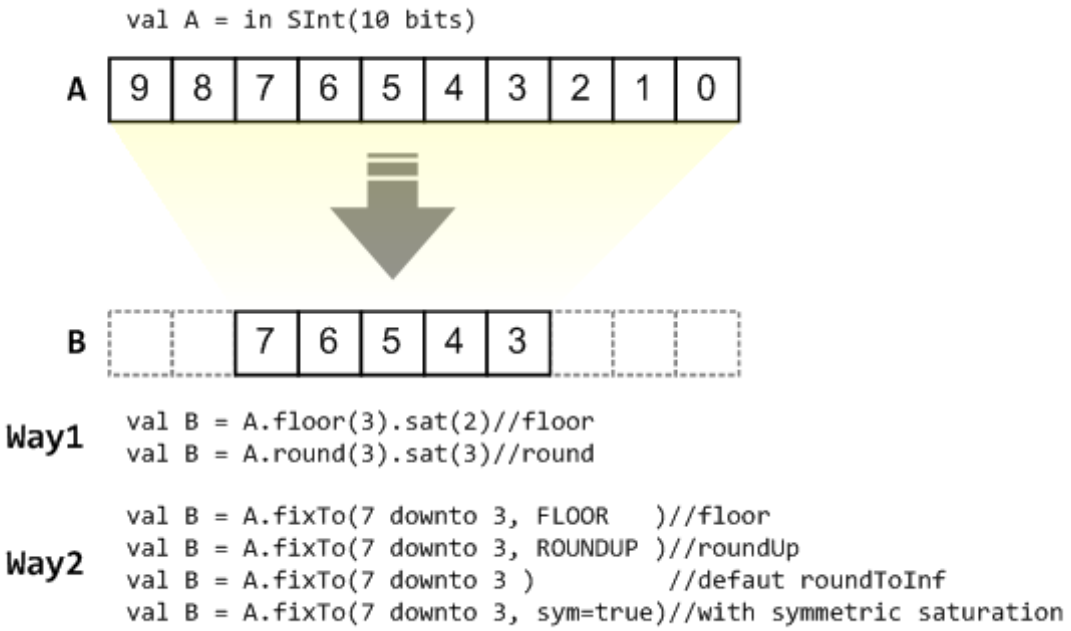
函数	操作	正向操作	负向操作
sat	饱和化	当 (Top[w-1, w-n].orR) 为真时设置为 maxValue	当 (Top[w-1, w-n].andR) 为真时设置为 minValue
trim	丢弃	不适用	不适用
symmetry	获取对称值	不适用	最小值 = -最大值

对称仅对 `SInt` 有效。

```
val A   = SInt(8 bits)
val B   = A.sat(3 bits)       // return 5 bits with saturated highest 3 bits
val B   = A.sat(3)            // equal to sat(3 bits)
val B   = A.trim(3 bits)      // return 5 bits with the highest 3 bits discarded
val B   = A.trim(3 bits)      // return 5 bits with the highest 3 bits discarded
val C   = A.symmetry          // return 8 bits and symmetry as (-128~127 to -127~127)
val C   = A.sat(3).symmetry   // return 5 bits and symmetry as (-16~15 to -15~15)
```

fixTo 函数

`UInt/SInt` 中提供了两种方法来实现定点小数位宽变化：



在 RTL 工作中强烈建议使用 `fixTo` 函数，您不需要像上图中的 **Way1** 那样手动处理进位对齐和位宽计算。

带自动饱和功能的定点数生成函数：

函数	描述	返回类型
fixTo(section, roundType, symmetric)	定点数生成	section.size bits

```
val A   = SInt(16 bits)
val B   = A.fixTo(10 downto 3) // default RoundType.ROUNDTOINF, sym = false
val B   = A.fixTo( 8 downto 0, RoundType.ROUNDUP)
val B   = A.fixTo( 9 downto 3, RoundType.CEIL,        sym = false)
val B   = A.fixTo(16 downto 1, RoundType.ROUNDTOINF, sym = true )
val B   = A.fixTo(10 downto 3, RoundType.FLOOR) // floor 3 bit, sat 5 bit @ highest
val B   = A.fixTo(20 downto 3, RoundType.FLOOR) // floor 3 bit, expand 2 bit @␣
↪highest
```

4.4 SpinalEnum

4.4.1 描述

`Enumeration` （枚举）类型对应于命名值的列表。

4.4.2 声明

枚举数据类型的声明如下：

```
object Enumeration extends SpinalEnum {
  val element0, element1, ..., elementN = newElement()
}
```

在上面的示例中，使用的是默认编码。VHDL （默认）使用本地枚举类型，Verilog （默认）使用二进制编码。

可以通过如下定义来强制设置指定枚举的编码：

```
object Enumeration extends SpinalEnum(defaultEncoding=encodingOfYourChoice) {
  val element0, element1, ..., elementN = newElement()
}
```

备注： 如果要将枚举定义为给定组件的 in/out，则必须执行以下操作：`in(MyEnum())` 或 `out(MyEnum())`

编码

支持以下枚举编码：

编码	位宽	描述
native		使用 VHDL 枚举系统，这是默认编码
binarySequential	log2Up(stateCount)	使用 Bits 按声明顺序存储状态（值从 0 到 n-1）
binaryOneHot	stateCount	使用位来存储状态。每一位对应一个状态，在硬件编码状态表示中一次仅设置一位。
graySequential	log2Up(stateCount)	将索引（像 binarySequential 中使用的数）编码为二进制格雷码。

自定义编码可以通过两种不同的方式执行：静态或动态。

```
/*
 * Static encoding
 */
object MyEnumStatic extends SpinalEnum {
  val e0, e1, e2, e3 = newElement()
  defaultEncoding = SpinalEnumEncoding("staticEncoding")(
    e0 -> 0,
    e1 -> 2,
    e2 -> 3,
    e3 -> 7)
}

/*
 * Dynamic encoding with the function :  _ * 2 + 1
 *   e.g. : e0 => 0 * 2 + 1 = 1
 *          e1 => 1 * 2 + 1 = 3
 *          e2 => 2 * 2 + 1 = 5
 *          e3 => 3 * 2 + 1 = 7
 */
val encoding = SpinalEnumEncoding("dynamicEncoding", _ * 2 + 1)

object MyEnumDynamic extends SpinalEnum(encoding) {
  val e0, e1, e2, e3 = newElement()
}
```

示例

实例化一个枚举信号并为其赋值：

```
object UartCtrlTxState extends SpinalEnum {
  val sIdle, sStart, sData, sParity, sStop = newElement()
}

val stateNext = UartCtrlTxState()
stateNext := UartCtrlTxState.sIdle

// You can also import the enumeration to have visibility of its elements
import UartCtrlTxState._
stateNext := sIdle
```

4.4.3 运算符

以下运算符可用于 Enumeration 类型：

比较运算

运算符	描述	返回类型
x === y	等价性判断	Bool
x =/= y	不等价判断运算	Bool

```
import UartCtrlTxState._

val stateNext = UartCtrlTxState()
stateNext := sIdle

when(stateNext === sStart) {
  ...
}

switch(stateNext) {
  is(sIdle) {
    ...
  }
  is(sStart) {
    ...
  }
  ...
}
```

类型

为了使用枚举（例如在函数中），您可能需要其类型。
值的类型（例如 sIdle 的类型）是

```
spinal.core.SpinalEnumElement[UartCtrlTxState.type]
```

或等效的

```
UartCtrlTxState.E
```

线束类型（例如 stateNext 的类型）是

```
spinal.core.SpinalEnumCraft[UartCtrlTxState.type]
```

或等效的

```
UartCtrlTxState.C
```

类型转换

运算符	描述	返回类型
x.asBits	二进制转换为 Bits	Bits(w(x) bits)
x.asBits.asUInt	二进制转换为 UInt	UInt(w(x) bits)
x.asBits.asSInt	二进制转换为 SInt	SInt(w(x) bits)
e.assignFromBits(bits)	Bits 转换为枚举	MyEnum()

```
import UartCtrlTxState._

val stateNext = UartCtrlTxState()
myBits := sIdle.asBits

stateNext.assignFromBits(myBits)
```

4.5 Bundle

4.5.1 描述

Bundle 是一种复合类型，它在单个名称下定义一组具有命名的信号（任何 SpinalHDL 基本类型）。

Bundle 可用于对数据结构、总线和接口进行建模。

4.5.2 声明

声明线束的语法如下：

```
case class myBundle extends Bundle {
  val bundleItem0 = AnyType
  val bundleItem1 = AnyType
  val bundleItemN = AnyType
}
```

例如，包含颜色的线束可以定义为：

```
case class Color(channelWidth: Int) extends Bundle {
  val r, g, b = UInt(channelWidth bits)
}
```

您可以在*Spinal HDL examples* 中找到*APB3 definition* 。

条件信号

Bundle 中的信号可以有条件地定义。除非 dataWidth 大于 0，否则在实力细化后的 myBundle`` 中将不会有 ``data 信号，如下例所示。

```
case class myBundle(dataWidth: Int) extends Bundle {
  val data = (dataWidth > 0) generate (UInt(dataWidth bits))
}
```

备注： 另请参阅*generate* 了解有关此 SpinalHDL 方法的信息。

4.5.3 运算符

以下运算符可用于 Bundle 类型：

比较运算

运算符	描述	返回类型
x === y	等价性判断	Bool
x =/= y	不等价判断运算	Bool

```
val color1 = Color(8)
color1.r := 0
color1.g := 0
color1.b := 0

val color2 = Color(8)
color2.r := 0
color2.g := 0
color2.b := 0

myBool := color1 === color2  // Compare all elements of the bundle
// is equivalent to:
// myBool := color1.r === color2.r && color1.g === color2.g && color1.b === color2.
↪b
```

类型转换

运算符	描述	返回类型
x.asBits	二进制转换为 Bits	Bits(w(x) bits)

```
val color1 = Color(8)
val myBits := color1.asBits
```

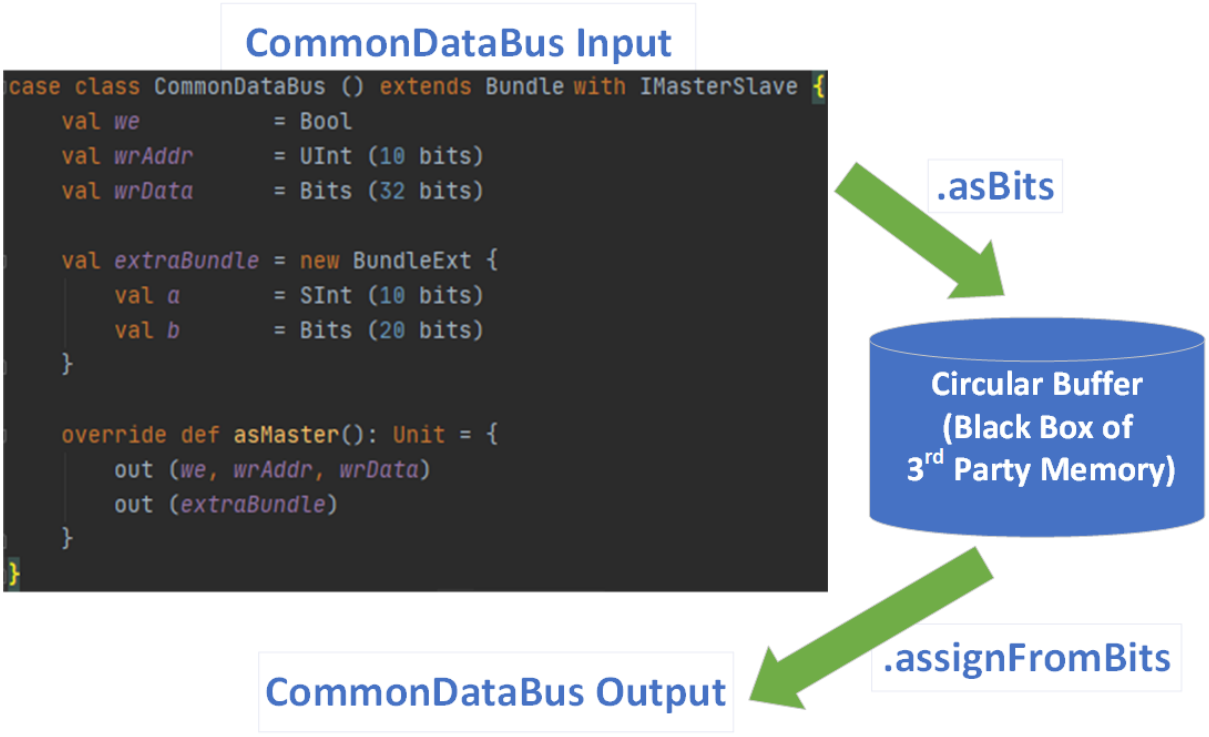
线束中的元素将按其定义的顺序映射到位，按 LSB 先放置的顺序。因此，color1 中的 r 将占据 myBits 的第 0 至 8 位（LSB），然后依次是 g 和 b ， b.msb 也是最终 Bits 类型的 MSB。

将位转换回线束

.assignFromBits 运算符可以被视为 .asBits 的逆操作。

运算符	描述	返回类型
x.assignFromBits(y)	将 Bits (y) 转换为 Bundle(x)	Unit
x.assignFromBits(y, hi, lo)	将 Bits (y) 转换为具有高/低边界的 Bundle(x)	Unit

下面的示例将名为 CommonDataBus 的线束保存到循环缓冲区（第三方内存）中，随后读出比特，并将其转换回 CommonDataBus 格式。




```
case class TestBundle () extends Component {
  val io = new Bundle {
    val we       = in     Bool()
    val addrWr   = in     UInt (7 bits)
    val dataIn   = slave  (CommonDataBus())

    val addrRd   = in     UInt (7 bits)
    val dataOut  = master (CommonDataBus())
  }

  val mm = Ram3rdParty_1w_1rs (G_DATA_WIDTH = io.dataIn.getBitsWidth,
                               G_ADDR_WIDTH = io.addrWr.getBitsWidth,
                               G_VENDOR     = "Intel_Arria10_M20K")

  mm.io.clk_in     := clockDomain.readClockWire
  mm.io.clk_out    := clockDomain.readClockWire

  mm.io.we         := io.we
  mm.io.addr_wr    := io.addrWr.asBits
  mm.io.d          := io.dataIn.asBits

  mm.io.addr_rd    := io.addrRd.asBits
  io.dataOut.assignFromBits(mm.io.q)
}
```

4.5.4 IO 元件方向

当您在组件的 IO 定义中实现 `Bundle` 时需要指定其方向。

in/out

如果线束的所有元素中的信号都朝同一方向传播，则可以使用 `in(MyBundle())` or `out(MyBundle())`。

例如：

```
val io = new Bundle {
  val input  = in (Color(8))
  val output = out(Color(8))
}
```

master/slave

如果您的接口遵循主/从拓扑结构，您可以使用 `IMasterSlave` 特征。然后你必须实现函数 `def asMaster(): Unit` 从 master 的角度设置每个元素的方向。然后你可以在 IO 定义中使用 `master(MyBundle())` 和 `slave(MyBundle())` 语法来使用。

有些函数定义为 toXXX，例如 `Flow` 类的 `toStream` 方法。这些函数通常可以由 master 端调用。另外，fromXXX 函数是为 slave 侧设计的。通常 master 端可用的功能多于 slave 端。

例如：

```
case class HandShake(payloadWidth: Int) extends Bundle with IMasterSlave {
  val valid   = Bool()
  val ready   = Bool()
  val payload = Bits(payloadWidth bits)

  // You have to implement this asMaster function.
  // This function should set the direction of each signals from an master point␣
↪of view
  override def asMaster(): Unit = {
    out(valid, payload)
    in(ready)
  }
}

val io = new Bundle {
  val input  = slave(HandShake(8))
  val output = master(HandShake(8))
}
```

4.6 Vec

4.6.1 描述

`Vec` 是一种复合类型，它在一个变量中定义一组可索引的信号（任何 SpinalHDL 基本类型）。

4.6.2 声明

声明向量的语法如下：

声明	描述
Vec.fill(size: Int)(type: Data)	创建一个包含 size 个元素的 Data 类型向量
Vec(x, y, ···)	创建一个向量，其中索引指向提供的元素。 不会创建新的硬件信号。 此构造函数支持混合宽度的元素。

示例

```
// Create a vector of 2 signed integers
val myVecOfSInt = Vec.fill(2)(SInt(8 bits))
myVecOfSInt(0) := 2                    // assignment to populate index 0
myVecOfSInt(1) := myVecOfSInt(0) + 3   // assignment to populate index 1

// Create a vector of 3 different type elements
val myVecOfMixedUInt = Vec(UInt(3 bits), UInt(5 bits), UInt(8 bits))

val x, y, z = UInt(8 bits)
val myVecOf_xyz_ref = Vec(x, y, z)

// Iterate on a vector
for(element <- myVecOf_xyz_ref) {
  element := 0   // Assign x, y, z with the value 0
}

// Map on vector
myVecOfMixedUInt.map(_ := 0) // Assign all elements with value 0

// Assign 3 to the first element of the vector
myVecOf_xyz_ref(1) := 3
```

4.6.3 运算符

以下运算符可用于 Vec 类型：

比较运算

运算符	描述	返回类型
x === y	等价性判断	Bool
x =/= y	不等价判断运算	Bool

```
// Create a vector of 2 signed integers
val vec2 = Vec.fill(2)(SInt(8 bits))
val vec1 = Vec.fill(2)(SInt(8 bits))

myBool := vec2 === vec1  // Compare all elements
// is equivalent to:
// myBool := vec2(0) === vec1(0) && vec2(1) === vec1(1)
```

类型转换

运算符	描述	返回类型
x.asBits	二进制转换为 Bits	Bits(w(x) bits)

```
// Create a vector of 2 signed integers
val vec1 = Vec.fill(2)(SInt(8 bits))

myBits_16bits := vec1.asBits
```

杂项

运算符	描述	返回类型
x.getBitsWidth	返回 Vec 的完整大小	Int

```
// Create a vector of 2 signed integers
val vec1 = Vec.fill(2)(SInt(8 bits))

println(widthOf(vec1)) // 16
```

库辅助函数

备注： 您需要以 `import spinal.lib._` 导入库，以将这些函数置于作用域中。

运算符	描述	返回类型
x.sCount(condition: T => Bool)	Count the number of occurrence matching a given condition in the Vec.	UInt
x.sCount(value: T)	Count the number of occurrence of a value in the Vec.	UInt
x.sExists(condition: T => Bool)	检查 Vec 中是否存在匹配条件的元素。	Bool
x.sContains(value: T)	检查 Vec 中是否存在具有给定值的元素。	Bool
x.sFindFirst(condition: T => Bool)	查找 Vec 中符合给定条件的第一个元素，如果成功找到，则返回该元素的索引。	(Bool, UInt)
x.reduceBalancedTree(op: (T, T) => T)	具有自动平衡功能的 reduce 函数，尽量减少生成电路的深度。op 应该是具有可交换性和可结合性的。	T
x.shuffle(indexMapping: Int => Int)	使用将旧索引映射到新索引的函数对 Vec 进行混洗（shuffle）。	Vec[T]

```
import spinal.lib._

// Create a vector with 4 unsigned integers
val vec1 = Vec.fill(4)(UInt(8 bits))

// ... the vector is actually assigned somewhere

val c1: UInt = vec1.sCount(_ < 128) // how many values are lower than 128 in vec
val c2: UInt = vec1.sCount(0) // how many values are equal to zero in vec

val b1: Bool = vec1.sExists(_ > 250) // is there a element bigger than 250
val b2: Bool = vec1.sContains(0) // is there a zero in vec
```

(续下页)

(接上页)

```
val (u1Found, u1): (Bool, UInt) = vec1.sFindFirst(_ < 10) // get the index of the␣
↪first element lower than 10
val u2: UInt = vec1.reduceBalancedTree(_ + _) // sum all elements together
```

备注： sXXX 前缀用于消除使用 lambda 函数作为参数的同名 Scala 函数带来的歧义。

警告： SpinalHDL 定点支持仅部分使用/测试，如果您发现任何错误，或者您认为缺少某些功能，请创建 Github issue。另外，请不要在代码中使用未归档的功能。

4.7 UFix/SFix

4.7.1 描述

`UFix` 和 `SFix` 类型对应于可用于定点算术的位向量。

4.7.2 声明

声明定点数的语法如下：

无符号定点小数

语法	位宽	分辨率	最大值	最小值
UFix(peak: ExpNumber, resolution: ExpNumber)	peak-resolution	2^resolution	2^peak-2^resolution	0
UFix(peak: ExpNumber, width: BitCount)	width	2^(peak-width)	2^peak-2^(peak-width)	0

有符号定点小数

语法	位宽	分辨率	最大值	最小值
SFix(peak: ExpNumber, resolution: ExpNumber)	peak-resolution+1	2^resolution	2^peak-2^resolution	-(2^peak)
SFix(peak: ExpNumber, width: BitCount)	width	2^(peak-width-1)	2^peak-2^(peak-width-1)	-(2^peak)

格式

所选格式遵循使用 Q 表示法定义定点小数格式的常用方法。更多信息可以在 Q 数据格式的维基百科页面上找到。

例如，Q8.2 表示 8+2 位的定点数，其中 8 位用于自然部分，2 位用于小数部分。如果定点数有符号，则多一位用于符号。

分辨率被定义为可以用该数字表示的最小的二的幂。

备注： 为了使表示二次幂的数字不易出错，在 spinal.core`` 中有一个称为 ``ExpNumber 的数字类型，它用于定点类型构造函数。这种类型有一个方便的封装，采用 exp 函数的形式（在本页的代码示例中使用）。

示例

```
// Unsigned Fixed-Point
val UQ_8_2 = UFix(peak = 8 exp, resolution = -2 exp) // bit width = 8 - (-2) = 10
↪bits
val UQ_8_2 = UFix(8 exp, -2 exp)

val UQ_8_2 = UFix(peak = 8 exp, width = 10 bits)
val UQ_8_2 = UFix(8 exp, 10 bits)

// Signed Fixed-Point
val Q_8_2 = SFix(peak = 8 exp, resolution = -2 exp) // bit width = 8 - (-2) + 1 =
↪11 bits
val Q_8_2 = SFix(8 exp, -2 exp)

val Q_8_2 = SFix(peak = 8 exp, width = 11 bits)
val Q_8_2 = SFix(8 exp, 11 bits)
```

4.7.3 赋值

有效赋值

当没有位丢失时，对定点小数值的分配是有效的。任何位丢失都会导致错误。

如果源定点小数值太大，truncated 函数将允许您调整源数字的大小以匹配目标大小。

示例

```
val i16_m2 = SFix(16 exp, -2 exp)
val i16_0  = SFix(16 exp,  0 exp)
val i8_m2  = SFix( 8 exp, -2 exp)
val o16_m2 = SFix(16 exp, -2 exp)
val o16_m0 = SFix(16 exp,  0 exp)
val o14_m2 = SFix(14 exp, -2 exp)

o16_m2 := i16_m2            // OK
o16_m0 := i16_m2            // Not OK, Bit loss
o14_m2 := i16_m2            // Not OK, Bit loss
o16_m0 := i16_m2.truncated  // OK, as it is resized to match assignment target
o14_m2 := i16_m2.truncated  // OK, as it is resized to match assignment target
val o18_m2 = i16_m2.truncated(18 exp, -2 exp)
val o18_22b = i16_m2.truncated(18 exp, 22 bit)
```

来自 Scala 常量

当给 UFix 或 SFix 信号赋值时，Scala 的 BigInt 或 Double 类型可以用作常量。

示例

```
val i4_m2 = SFix(4 exp, -2 exp)
i4_m2 := 1.25    // Will load 5 in i4_m2.raw
i4_m2 := 4       // Will load 16 in i4_m2.raw
```

4.7.4 原始值

可以使用 raw 属性读取或写入定点小数的整数表示形式。

示例

```
val UQ_8_2 = UFix(8 exp, 10 bits)
UQ_8_2.raw := 4         // Assign the value corresponding to 1.0
UQ_8_2.raw := U(17)     // Assign the value corresponding to 4.25
```

4.7.5 运算符

以下运算符可用于 UFix 类型：

算术运算

运算符	描述	返回值的分辨率	返回值的幅度
x + y	加法	Min(x.resolution, y.resolution)	Max(x.amplitude, y.amplitude)
x - y	减法	Min(x.resolution, y.resolution)	Max(x.amplitude, y.amplitude)
x * y	乘法	x.resolution * y.resolution)	x.amplitude * y.amplitude
x » y	算术右移，y : Int	x.amplitude » y	x.resolution » y
x « y	算术左移，y : Int	x.amplitude « y	x.resolution « y
x »| y	算术右移，y : Int	x.amplitude » y	x.resolution
x «| y	算术左移，y : Int	x.amplitude « y	x.resolution

比较运算

运算符	描述	返回类型
x === y	等价性判断	Bool
x =/= y	不等价判断运算	Bool
x > y	大于	Bool
x >= y	大于或等于	Bool
x < y	小于	Bool
x >= y	小于或等于	Bool

类型转换

运算符	描述	返回类型
x.asBits	二进制转换为 Bits	Bits(w(x) bits)
x.asUInt	二进制转换为 UInt	UInt(w(x) bits)
x.asSInt	二进制转换为 SInt	SInt(w(x) bits)
x.asBools	转换为 Bool 数组	Vec(Bool(),width(x))
x.toUInt	返回对应的 UInt（带截断）	UInt
x.toSInt	返回对应的 SInt（带截断）	SInt
x.toUFix	返回对应的 UFix	UFix
x.toSFix	返回对应的 SFix	SFix

杂项

名称	返回类型	描述
x.maxValue	返回可存储的最大值	Double
x.minValue	返回可存储的最小值	Double
x.resolution	x.amplitude * y.amplitude	Double

警告： SpinalHDL 浮点小数支持正在开发中，仅部分使用/测试，如果您有任何错误，或者您认为缺少某些功能，请创建 Github issue。另外，请不要在代码中使用未归档的功能。

4.8 浮点小数

4.8.1 描述

`Floating` 类型对应于 IEEE-754 编码数字。第二种类型称为 `RecFloating`，通过重新编码浮点值来帮助简化设计，简化 IEEE-754 浮点中的一些边缘情况。

它由一个符号位、一个指数字段和一个尾数字段组成。不同字段的宽度在 IEEE-754 或事实上的标准中定义。

此类型可通过以下方法导入使用：

```
import spinal.lib.experimental.math._
```

IEEE-754 浮点小数格式

这些数字被编码为 IEEE-754 浮点格式。

重新编码的浮动小数格式

由于 IEEE-754 对非规范化数字和特殊值有一些怪癖，加州大学伯克利分校提出了另一种重新编码浮点小数值的方法。

通过修改尾数，以便非标准化值可以与标准化值相同地对待。

指数字段比 IEEE-754 定义的大一位。

两种编码之间的符号位保持不变。

示例可以在 这里 找到

零

零的编码是，指数字段的三个前导零被设置为零。

非规范化值

在浮点数表示中，非规范化数值的编码方式与规范化数值相同。尾数部分会进行位移，使得最左边的 1 位成为隐含的，而指数部分则由 107（十进制）加上最左边 1 位的位置索引来计算。这样的编码确保了非规范化数值能够正确表示。

标准化值

标准化值的重新编码尾数与原始 IEEE-754 尾数完全相同。重新编码的指数被编码为 130（十进制）加上原始指数值。

无穷大

此时，重新编码的尾数值被视为无关紧要。重新编码的指数三位最高位是 6（十进制），其余指数可以视为不关心。

无效数（NaN）

此时，重新编码中尾数与原始 IEEE-754 尾数完全相同。重新编码的指数三位最高位是 7（十进制），其余指数可以视为不关心。

4.8.2 声明

声明浮点小数的语法如下：

IEEE-754 编码数

语法	描述
Floating(exponentSize: Int, mantissaSize: Int)	具有自定义指数和尾数大小的 IEEE-754 浮点值
Floating16()	IEEE-754 半精度浮点数
Floating32()	IEEE-754 单精度浮点数
Floating64()	IEEE-754 双精度浮点数
Floating128()	IEEE-754 四精度浮点数

重新编码的浮点数

语法	描述
RecFloating(exponentSize: Int, mantissaSize: Int)	使用自定义指数和尾数大小重新编码的浮点小数值
RecFloating16()	重新编码的半精度浮点数
RecFloating32()	重新编码的单精度浮点数
RecFloating64()	重新编码的双精度浮点数
RecFloating128()	重新编码的四精度浮点数

4.8.3 运算符

以下运算符可用于 `Floating` 和 `RecFloating` 类型：

类型转换

运算符	描述	返回类型
x.asBits	二进制转换为 Bits	Bits(w(x) bits)
x.asBools	转换为 Bool 数组	Vec(Bool(),width(x))
x.toUInt(size: Int)	返回对应的 UInt（带截断）	UInt
x.toSInt(size: Int)	返回对应的 SInt（带截断）	SInt
x.fromUInt	返回对应的 Floating（带截断）	UInt
x.fromSInt	返回对应的 Floating（带截断）	SInt

4.9 AFix

4.9.1 描述

`AFix` 是一个支持自动范围的定点类型，它在执行定点运算时跟踪可表示值的范围。

警告：此代码的大部分仍在开发中。API 和函数原型可能会发生变化。

感谢用户反馈！

4.9.2 声明

AFix 可以在创建时指定总位宽或指数部分位宽：

```
AFix.U(12 bits)              // U12.0
AFix(QFormat(12, 0, false)) // U12.0
AFix.UQ(8 bits, 4 bits)      // U8.4
AFix.U(8 exp, 12 bits)       // U8.4
AFix.U(8 exp, -4 exp)        // U8.4
AFix.U(8 exp, 4 exp)         // U8.-4
AFix(QFormat(12, 4, false)) // U8.4

AFix.S(12 bits)              // S11.0 + sign
AFix(QFormat(12, 0, true))  // S11.0 + sign
AFix.SQ(8 bits, 4 bits)      // S8.4  + sign
AFix.S(8 exp, 12 bits)       // S8.3  + sign
AFix.S(8 exp, -4 exp)        // S8.4  + sign
AFix(QFormat(12, 4, true))  // S7.4  + sign
```

这些将占据所有位的可表示范围。

例如：

`AFix.U(12 bits)` 可表示的范围是 0 to 4095。

`AFix.SQ(8 bits, 4 bits)` 的范围为 -4096 (-256) 到 4095 (255.9375)

`AFix.U(8 exp, 4 exp)` 的范围为 0 到 256

可以通过直接实例化类来创建自定义范围 `AFix` 值。

```
class AFix(val maxValue: BigInt, val minValue: BigInt, val exp: ExpNumber)

new AFix(4096, 0, 0 exp)      // [0 to 4096, 2^0]
new AFix(256, -256, -2 exp)  // [-256 to 256, 2^-2]
new AFix(16, 8, 2 exp)        // [8 to 16, 2^2]
```

在定点数表示法中，`maxValue` 和 `minValue` 变量用于存储可表示的最大和最小整数值。这些数值是在进行“2^exp“（即 2 的 exp 次方）的乘法运算后得到的定点数的实际值。

`AFix.U(2 exp, -1 exp)` 可以表示：`0, 0.5, 1.0, 1.5, 2, 2.5, 3, 3.5`

`AFix.S(2 exp, -2 exp)` 可以表示：`-2.0, -1.75, -1.5, -1.25, -1, -0.75, -0.5, -0.25, 0, 0.25, 0.5, 0.75, 1, 1.25, 1.5, 1.75`

指数值大于 0 是允许的，用来表示该数值大于 1 。

`AFix.S(2 exp, 1 exp)` 可以表示：`-4, 2, 0, 2`

`AFix(8, 16, 2 exp)` 可以表示：`32, 36, 40, 44, 48, 52, 56, 60, 64`

注意：`AFix` 将使用 5 位来保存此类型，因此它可以存储 `16`，即它的 `maxValue`。

4.9.3 数学运算

`AFix` 在硬件级别支持加法 (+)、减法 (-) 和乘法 (*)。也提供了除法（\）和模（%）运算符，但这些不建议用硬件实现。

对 `AFix` 执行运算就像常规 `Int` 整型数一样。有符号数和无符号数是可以互操作的。有符号值和无符号值之间没有类型差异。

```
// Integer and fractional expansion
val a = AFix.U(4 bits)          // [   0 (  0.)     to  15 (15.  )]  4 bits, 2^0
val b = AFix.UQ(2 bits, 2 bits) // [   0 (  0.)     to  15 ( 3.75)]  4 bits, 2^-2
val c = a + b                   // [   0 (  0.)     to  77 (19.25)]  7 bits, 2^-2
```

(续下页)

(接上页)

```
val d = new AFix(-4, 8, -2 exp) // [-  4 (- 1.25)   to   8 ( 2.00)]  5 bits, 2^-2
val e = c * d                   // [-308 (-19.3125) to 616 (38.50)] 11 bits, 2^-4

// Integer without expansion
val aa = new AFix(8, 16, -4 exp) // [8 to 16] 5 bits, 2^-4
val bb = new AFix(1, 15, -4 exp) // [1 to 15] 4 bits, 2^-4
val cc = aa + bb                 // [9 to 31] 5 bits, 2^-4
```

AFix 支持无范围扩展的操作。它通过从每个输入中选择对齐的最大和最小范围来实现此目的。

+| 和 -| 分别代表加法和减法操作，这两种操作在执行时不会对数值进行扩展。

4.9.4 不等式运算

AFix 支持标准的不等式运算。

```
A === B
A =\= B
A < B
A <= B
A > B
A >= B
```

警告：编译时超出范围的操作将被优化掉！

4.9.5 位移操作

AFix 支持十进制和位移操作

<< 将十进制小数点向左移动，即增加指数值。>> 将十进制小数点向右移动，即减小指数值。<<| 将位左移。给小数位追加零。>>| 将位向右移动。删除小数位。

4.9.6 饱和与舍入

AFix 实现饱和和所有常见的舍入方法。

饱和的工作原理是使 AFix 值的支持值范围饱和。有多个考虑到了指数特性的辅助函数。

```
val a = new AFix(63, 0, -2 exp) // [0 to 63, 2^-2]
a.sat(63, 0)                    // [0 to 63, 2^-2]
a.sat(63, 0, -3 exp)            // [0 to 31, 2^-2]
a.sat(new AFix(31, 0, -1 exp))  // [0 to 31, 2^-2]
```

AFix 舍入模式:

```
// The following require exp < 0
.floor() or .truncate()
.ceil()
.floorToZero()
.ceilToInf()
// The following require exp < -1
.roundHalfUp()
.roundHalfDown()
.roundHalfToZero()
.roundHalfToInf()
.roundHalfToEven()
.roundHalfToOdd()
```

这些舍入模式的数学示例在这里得到了更好的解释：Rounding - Wikipedia

所有这些模式都会产生指数为 0 的 AFix 值。如果需要舍入到不同的指数，请考虑移位或使用带有 truncated 标签的赋值。

4.9.7 赋值

AFix 会在赋值时自动检查并扩大范围和精度。默认情况下，将一个 AFix 值赋值给另一个范围或精度更小的 AFix 值是错误的。

.truncated 函数用于控制如何赋值给较小的类型。

```
def truncated(saturation: Boolean = false,
              overflow  : Boolean = true,
              rounding  : RoundType = RoundType.FLOOR)

def saturated(): AFix = this.truncated(saturation = true, overflow = false)
```

RoundType：

```
RoundType.FLOOR
RoundType.CEIL
RoundType.FLOORTOZERO
RoundType.CEILTOINF
RoundType.ROUNDUP
RoundType.ROUNDDOWN
RoundType.ROUNDTOZERO
RoundType.ROUNDTOINF
RoundType.ROUNDTOEVEN
RoundType.ROUNDTOODD
```

当设置 saturation 标志时，系统会在数值超出指定数据类型的界限时，自动将其调整至该类型的有效范围，以防止溢出。

overflow 标志将允许在舍入后直接赋值，而不进行范围检查。

将精度较高的值分配给精度较低的值时，始终需要进行舍入。

CHAPTER 5

结构设计

各章内容：

- 如何构建可重用的组件
- 除组件外的其他硬件组合方法
- 时钟/复位域的处理
- instantiation of existing VHDL and Verilog IP
- SpinalHDL 中如何为信号等分配名称，以及如何影响命名

5.1 组件和层次结构

像在 VHDL 和 Verilog 中一样，可以使用组件构建设计层次结构。然而，在 SpinalHDL 中，不需要在实例化时绑定它们的端口：

```
class AdderCell() extends Component {
  // Declaring external ports in a Bundle called `io` is recommended
  val io = new Bundle {
    val a, b, cin = in port Bool()
    val sum, cout = out port Bool()
  }
  // Do some logic
  io.sum := io.a ^ io.b ^ io.cin
  io.cout := (io.a & io.b) | (io.a & io.cin) | (io.b & io.cin)
}

class Adder(width: Int) extends Component {
  ...
  // Create 2 AdderCell instances
  val cell0 = new AdderCell()
  val cell1 = new AdderCell()
  cell1.io.cin := cell0.io.cout   // Connect cout of cell0 to cin of cell1

  // Another example which creates an array of ArrayCell instances
  val cellArray = Array.fill(width)(new AdderCell())
  cellArray(1).io.cin := cellArray(0).io.cout   // Connect cout of cell(0) to cin
```

(续下页)

(接上页)

```
↪of cell(1)
  ...
}
```

小技巧:

`val io = new Bundle { ... }`
建议在名为 `io` 的 `Bundle` 中声明外部端口。如果您将线束命名为 `io`，SpinalHDL 将检查其所有元素是否定义为输入或输出。

小技巧: 如果更符合您的风格，您也可以使用 `Module` 语法而不是 `Component` （它们是相同的东西）

5.1.1 输入/输出定义

定义输入和输出的语法如下：

语法	描述	返回类型
`in port Bool()` `out port Bool()`	创建输入 Bool/输出 Bool	Bool

`in Bits/UInt/SInt[(x bits)]`
`out Bits/UInt/SInt[(x bits)]`
`in Bits(3 bits)`
	创建相应类型的输入/输出端口	Bits/UInt/SInt

`in(T)` `out(T)`	对于所有其他数据类型，您可能需要在其周围添加一些括号。这是 Scala 的限制。	T

`master(T)`
`slave(T)`
`master(Bool())`
	此语法由 `spinal.lib` 库提供（如果您使用 `slave` 语法标注对象，则应导入 `spinal.lib.slave`）。T 必须继承自 `IMasterSlave`。一些参考文档在这里 。您实际上可能不需要括号，因此写成 `master T` 也可以。	T

组件之间的互连需要遵循一些规则：

- 组件只能 **读取**子组件的输出和输入信号。
- 组件可以读取自己的输出端口值（与 VHDL 不同）。

小技巧: 如果由于某种原因您需要从层次结构中较深的位置读取信号（例如用于调试或临时补丁），您可以使用 `some.where.else.theSignal.pull()` 函数返回的信号来完成此操作

5.1.2 裁剪信号

SpinalHDL will generate all the named signals and their dependencies, while all the useless anonymous / zero width ones are removed from the RTL generation.

您可以通过生成的 `SpinalReport` 对象上的 `printPruned` 和 `printPrunedIo` 函数收集所有已删除的无用信号列表：

```
class TopLevel extends Component {
  val io = new Bundle {
    val a,b = in port UInt(8 bits)
    val result = out port UInt(8 bits)
  }

  io.result := io.a + io.b

  val unusedSignal = UInt(8 bits)
  val unusedSignal2 = UInt(8 bits)

  unusedSignal2 := unusedSignal
}

object Main {
  def main(args: Array[String]) {
    SpinalVhdl(new TopLevel).printPruned()
    // This will report :
    //  [Warning] Unused wire detected : toplevel/unusedSignal : UInt[8 bits]
    //  [Warning] Unused wire detected : toplevel/unusedSignal2 : UInt[8 bits]
  }
}
```

5.1.3 参数化硬件（VHDL 中的“Generic”，Verilog 中的“Parameter”）

如果你想参数化你的组件，你可以将参数传递给组件的构造函数，如下所示：

```
class MyAdder(width: BitCount) extends Component {
  val io = new Bundle {
    val a, b   = in port UInt(width)
    val result = out port UInt(width)
  }
  io.result := io.a + io.b
}

object Main {
  def main(args: Array[String]) {
    SpinalVhdl(new MyAdder(32 bits))
  }
}
```

如果您有多个参数，最好给出一个专用的配置类，如下所示：

```
case class MySocConfig(axiFrequency  : HertzNumber,
                       onChipRamSize : BigInt,
                       cpu           : RiscCoreConfig,
                       iCache        : InstructionCacheConfig)

class MySoc(config: MySocConfig) extends Component {
  ...
}
```

您可以在配置中添加功能以及对配置属性的要求：

```
case class MyBusConfig(addressWidth: Int, dataWidth: Int) {
  def bytePerWord = dataWidth / 8
  def addressType = UInt(addressWidth bits)
  def dataType = Bits(dataWidth bits)

  require(dataWidth == 32 || dataWidth == 64, "Data width must be 32 or 64")
}
```

备注： 这种参数化完全发生在 SpinalHDL 代码生成的实例细化过程中。生成的 HDL 代码不包含使用 HDL 语言泛化特性的内容。此处描述的方法不会使用 VHDL 泛型或 Verilog 参数。

另请参阅*Blackbox* 了解有关该机制支持的更多信息。

5.1.4 综合后组件名称

在模块内，每个组件都有一个名称，称为“部分名称”。“完整”名称是通过将每个组件的父名称与“_”连接起来构建的，例如：`io_clockDomain_reset`。您可以使用 `setName` 将按此约定生成的名称替换为自定义的。这在与外部组件连接时特别有用。其他方法分别称为 `getName`、`setPartialName` 和 `getPartialName`。

综合时，每个模块都会获得定义它的 Scala 类的名称。您也可以调用 `setDefinitionName` 函数来覆盖它。

5.2 Area

有时，创建一个 `Component` 组件来定义某些逻辑是多余的，因为：

- 需要定义所有构造参数和 IO（冗长、重复）
- 拆分您的代码（超出需求）

对于这种情况，您可以使用 `Area` 来定义一组信号/逻辑：

```
class UartCtrl extends Component {
  ...
  val timer = new Area {
    val counter = Reg(UInt(8 bits))
    val tick = counter === 0
    counter := counter - 1
    when(tick) {
      counter := 100
    }
  }

  val tickCounter = new Area {
    val value = Reg(UInt(3 bits))
    val reset = False
    when(timer.tick) {          // Refer to the tick from timer area
      value := value + 1
    }
    when(reset) {
      value := 0
    }
  }

  val stateMachine = new Area {
    ...
```

(续下页)

(接上页)

```
  }
}
```

小技巧:

在 VHDL 和 Verilog 中，有时使用前缀将变量分隔成逻辑部分。建议您在 SpinalHDL 中使用 `Area` 代替它。

备注: *ClockingArea* 是一种特殊的 `Area`，它允许您在该硬件块中缺省使用指定的 `ClockDomain` 时钟域

5.3 函数

使用 Scala 函数生成硬件的方式与 VHDL/Verilog 完全不同，原因有很多：

- 您可以实例化寄存器、组合逻辑以及其中的组件。
- 您不必使用限制信号赋值范围的 `process`/`@always` 块。
- 一切都通过引用传递，这允许简化操作。
  例如，您可以将总线作为参数提供给函数，然后该函数可以在内部对其进行读/写。您还可以从 Scala 世界（函数、类型等）返回组件、总线或任何其他内容。

5.3.1 RGB 信号转灰度信号

例如，如果您想使用系数将红/绿/蓝颜色转换为灰度，您可以使用函数来完成：

```
// Input RGB color
val r, g, b = UInt(8 bits)

// Define a function to multiply a UInt by a Scala Float value.
def coef(value: UInt, by: Float): UInt = (value * U((255 * by).toInt, 8 bits) >> 8)

// Calculate the gray level
val gray = coef(r, 0.3f) + coef(g, 0.4f) + coef(b, 0.3f)
```

5.3.2 Valid Ready Payload 总线

例如，如果您定义一个带有 `valid`, `ready` 和 `payload` 信号的简单总线，则可以在其中定义一些有用的函数。

```
case class MyBus(payloadWidth: Int) extends Bundle with IMasterSlave {
  val valid   = Bool()
  val ready   = Bool()
  val payload = Bits(payloadWidth bits)

  // Define the direction of the data in a master mode
  override def asMaster(): Unit = {
    out(valid, payload)
    in(ready)
  }

  // Connect that to this
```

(续下页)

(接上页)

```
  def <<(that: MyBus): Unit = {
    this.valid   := that.valid
    that.ready   := this.ready
    this.payload := that.payload
  }

  // Connect this to the FIFO input, return the fifo output
  def queue(size: Int): MyBus = {
    val fifo = new MyBusFifo(payloadWidth, size)
    fifo.io.push << this
    return fifo.io.pop
  }
}

class MyBusFifo(payloadWidth: Int, depth: Int) extends Component {

  val io = new Bundle {
    val push = slave(MyBus(payloadWidth))
    val pop  = master(MyBus(payloadWidth))
  }

  val mem = Mem(Bits(payloadWidth bits), depth)

  // ...

}
```

5.4 时钟域

5.4.1 简介

在 SpinalHDL 中，时钟和复位信号可以组合起来创建 ** 时钟域 **。时钟域可以应用于设计的某些区域，然后实例化到这些区域中的所有同步元件将 ** 隐式 ** 使用该时钟域。

时钟域的应用方式类似于堆栈，这意味着当您的设计位于给定时钟域中，您仍然可以将该设计应用到另一个时钟域。

请注意，寄存器在创建时捕获其时钟域，而不是在赋值时捕获。因此，请确保在所需的 `ClockingArea` 内创建它们。

5.4.2 实例化

定义时钟域的语法如下（使用 EBNF 语法）：

```
ClockDomain(
  clock: Bool
  [,reset: Bool]
  [,softReset: Bool]
  [,clockEnable: Bool]
  [,frequency: IClockDomainFrequency]
  [,config: ClockDomainConfig]
)
```

这个定义有五个参数：

参数	描述	默认值
clock	定义时钟域中的时钟信号	
reset	复位信号。如果存在需要复位的寄存器，而时钟域没有提供复位，则会显示错误消息	null
softReset	复位意味着额外的同步复位	null
clockEnable	该信号的目标是禁用整个时钟域上的时钟，而无需在每个同步元件上手动实现	null
frequency	允许您指定给定时钟域的频率，然后在您的设计中读取它。该参数不生成 PLL 或其他硬件来控制频率	UnknownFrequency
config	指定信号的极性和复位的性质	当前配置

在设计中定义具有指定属性时钟域的示例如下：

```
val coreClock = Bool()
val coreReset = Bool()

// Define a new clock domain
val coreClockDomain = ClockDomain(coreClock, coreReset)

// Use this domain in an area of the design
val coreArea = new ClockingArea(coreClockDomain) {
  val coreClockedRegister = Reg(UInt(4 bits))
}
```

当不需要 *Area* 时，也可以直接应用时钟域。存在两种语法：

```
class Counters extends Component {
  val io = new Bundle {
    val enable = in Bool ()
    val freeCount, gatedCount, gatedCount2 = out UInt (4 bits)
  }
  val freeCounter = CounterFreeRun(16)
  io.freeCount := freeCounter.value

  // In a real design consider using a glitch free single purpose CLKGATE␣
↪primitive instead
  val gatedClk = ClockDomain.current.readClockWire && io.enable
  val gated = ClockDomain(gatedClk, ClockDomain.current.readResetWire)

  // Here the "gated" clock domain is applied on "gatedCounter" and "gatedCounter2"
  val gatedCounter = gated(CounterFreeRun(16))
  io.gatedCount := gatedCounter.value
  val gatedCounter2 = gated on CounterFreeRun(16)
  io.gatedCount2 := gatedCounter2.value

  assert(gatedCounter.value === gatedCounter2.value, "gated count mismatch")
}
```

配置

除了构造函数参数之外，每个时钟域的以下元素都可以通过 ClockDomainConfig 类进行配置：

属性	有效值
clockEdge	RISING, FALLING
resetKind	某些 FPGA 支持的 ASYNC、SYNC 和 BOOT （其中 FF 值由比特流加载）
resetActiveLevel	HIGH, LOW
softResetActiveLevel	HIGH, LOW
clockEnableActiveLevel	HIGH, LOW

```
class CustomClockExample extends Component {
  val io = new Bundle {
    val clk    = in Bool()
    val resetn = in Bool()
    val result = out UInt (4 bits)
  }

  // Configure the clock domain
  val myClockDomain = ClockDomain(
    clock  = io.clk,
    reset  = io.resetn,
    config = ClockDomainConfig(
      clockEdge        = RISING,
      resetKind        = ASYNC,
      resetActiveLevel = LOW
    )
  )

  // Define an Area which use myClockDomain
  val myArea = new ClockingArea(myClockDomain) {
    val myReg = Reg(UInt(4 bits)) init(7)

    myReg := myReg + 1

    io.result := myReg
  }
}
```

默认情况下，ClockDomain 应用于整个设计。该默认域的配置为：

- Clock：上升沿
- Reset：异步，高电平有效
- 无时钟使能

这对应于以下 ClockDomainConfig：

```
val defaultCC = ClockDomainConfig(
  clockEdge        = RISING,
  resetKind        = ASYNC,
  resetActiveLevel = HIGH
)
```

内部时钟

另一种创建时钟域的语法如下：

```
ClockDomain.internal(
  name: String,
  [config: ClockDomainConfig,]
  [withReset: Boolean,]
  [withSoftReset: Boolean,]
  [withClockEnable: Boolean,]
  [frequency: IClockDomainFrequency]
)
```

该定义有六个参数：

参数	描述	默认值
name	*clk* 和 *reset* 信号的名称	
config	指定信号的极性和复位的性质	当前配置
withReset	添加复位信号	true
withSoftReset	添加软复位信号	false
withClockEnable	添加时钟使能	false
frequency	时钟域频率	UnknownFrequency

这种方法的优点是使用已知/指定的名称而不是继承的名称来创建时钟和复位信号。

创建后，您必须分配 ClockDomain 的信号，如下例所示：

```
class InternalClockWithPllExample extends Component {
  val io = new Bundle {
    val clk100M = in Bool()
    val aReset  = in Bool()
    val result  = out UInt (4 bits)
  }
  // myClockDomain.clock will be named myClockName_clk
  // myClockDomain.reset will be named myClockName_reset
  val myClockDomain = ClockDomain.internal("myClockName")

  // Instantiate a PLL (probably a BlackBox)
  val pll = new Pll()
  pll.io.clkIn := io.clk100M

  // Assign myClockDomain signals with something
  myClockDomain.clock := pll.io.clockOut
  myClockDomain.reset := io.aReset || !pll.io.

  // Do whatever you want with myClockDomain
  val myArea = new ClockingArea(myClockDomain) {
    val myReg = Reg(UInt(4 bits)) init(7)
    myReg := myReg + 1

    io.result := myReg
  }
}
```

警告： 在您创建时钟域的其他组件中，您不得使用 .clock 和 .reset，而应使用 .readClockWire 和 .readResetWire，如下所示。对于全局时钟域，您必须始终使用这些 .readXXX 函数。

外部时钟

您可以在源中的任何位置定义由外部驱动的时钟域。然后，它会自动将时钟和复位线从顶层输入添加到所有同步元件。

```
ClockDomain.external(
  name: String,
  [config: ClockDomainConfig,]
  [withReset: Boolean,]
  [withSoftReset: Boolean,]
  [withClockEnable: Boolean,]
  [frequency: IClockDomainFrequency]
)
```

`ClockDomain.external` 函数的参数与 `ClockDomain.internal` 函数中的参数完全相同。下面是使用 `ClockDomain.external` 的设计示例：

```
class ExternalClockExample extends Component {
  val io = new Bundle {
    val result = out UInt (4 bits)
  }

  // On the top level you have two signals  :
  //     myClockName_clk and myClockName_reset
  val myClockDomain = ClockDomain.external("myClockName")

  val myArea = new ClockingArea(myClockDomain) {
    val myReg = Reg(UInt(4 bits)) init(7)
    myReg := myReg + 1

    io.result := myReg
  }
}
```

生成 HDL 时的信号优先级

在当前版本中，复位和时钟使能信号具有不同的优先级。它们的顺序是：`asyncReset`, `clockEnable`, `syncReset` 和 `softReset`。

Please be careful that clockEnable has a higher priority than syncReset. If you do a sync reset when the clockEnable is disabled (especially at the beginning of a simulation), the gated registers will not be reset.

这是一个例子：

```
val clockedArea = new ClockEnableArea(clockEnable) {
  val reg = RegNext(io.input) init(False)
}
```

它将生成 Verilog HDL 代码，例如：

```
always @(posedge clk) begin
  if(clockedArea_newClockEnable) begin
    if(!resetn) begin
      clockedArea_reg <= 1'b0;
    end else begin
      clockedArea_reg <= io_input;
    end
  end
end
```

If that behavior is problematic, one workaround is to use a when statement as a clock enable instead of using the ClockDomain.enable feature. This is open for future improvements.

语境

您可以通过在任何地方调用 `ClockDomain.current` 来检索您所在的时钟域。

返回的 `ClockDomain` 实例具有以下可以调用的函数：

名称	描述	返回类型
frequency.getValue	返回时钟域的频率。 这是您配置域的任意值。	Double
hasReset	如果时钟域有复位信号则返回	Boolean
hasSoftReset	返回时钟域是否有软复位信号	Boolean
hasClockEnable	返回时钟域是否有时钟使能信号	Boolean
readClockWire	返回从时钟信号派生的信号	Bool
readResetWire	返回一个从复位信号派生的信号	Bool
readSoftResetWire	返回从软复位信号派生的信号	Bool
readClockEnableWire	返回从时钟使能信号派生的信号	Bool
isResetActive	当复位有效时返回 True	Bool
isSoftResetActive	当软复位有效时返回 True	Bool
isClockEnableActive	当时钟使能有效时返回 True	Bool

下面包含一个示例，其中通过 UART 控制器使用频率来设置其时钟分频器：

```
val coreClockDomain = ClockDomain(coreClock, coreReset,␣
↪frequency=FixedFrequency(100e6))

val coreArea = new ClockingArea(coreClockDomain) {
  val ctrl = new UartCtrl()
  ctrl.io.config.clockDivider := (coreClk.frequency.getValue / 57.6e3 / 8).toInt
}
```

5.4.3 跨时钟域设计

SpinalHDL 在编译时检查是否存在不需要的/未指定的跨时钟域信号读取。如果您想读取另一个 `ClockDomain` 逻辑区发出的信号，则应给目标信号增加 `crossClockDomain` 标记，如下例所示：

```
//              _____                          _____              _____
//             |     |  (crossClockDomain)    |     |            |     |
//  dataIn -->|     |----------------------->|     |----------->|     |--> dataOut
//            | FF  |                        | FF  |            | FF  |
//  clkA   -->|     |              clkB  -->|     |    clkB -->|     |
//  rstA   -->|_____|              rstB  -->|_____|    rstB -->|_____|



// Implementation where clock and reset pins are given by components' IO
class CrossingExample extends Component {
```

(续下页)

(接上页)

```
  val io = new Bundle {
    val clkA = in Bool()
    val rstA = in Bool()

    val clkB = in Bool()
    val rstB = in Bool()

    val dataIn  = in Bool()
    val dataOut = out Bool()
  }

  // sample dataIn with clkA
  val area_clkA = new ClockingArea(ClockDomain(io.clkA,io.rstA)) {
    val reg = RegNext(io.dataIn) init(False)
  }

  // 2 register stages to avoid metastability issues
  val area_clkB = new ClockingArea(ClockDomain(io.clkB,io.rstB)) {
    val buf0  = RegNext(area_clkA.reg) init(False) addTag(crossClockDomain)
    val buf1  = RegNext(buf0)          init(False)
  }

  io.dataOut := area_clkB.buf1
}


// Alternative implementation where clock domains are given as parameters
class CrossingExample(clkA : ClockDomain,clkB : ClockDomain) extends Component {
  val io = new Bundle {
    val dataIn  = in Bool()
    val dataOut = out Bool()
  }

  // sample dataIn with clkA
  val area_clkA = new ClockingArea(clkA) {
    val reg = RegNext(io.dataIn) init(False)
  }

  // 2 register stages to avoid metastability issues
  val area_clkB = new ClockingArea(clkB) {
    val buf0  = RegNext(area_clkA.reg) init(False) addTag(crossClockDomain)
    val buf1  = RegNext(buf0)          init(False)
  }

  io.dataOut := area_clkB.buf1
}
```

一般来说，可以使用 2 个或更多由目标时钟域驱动的触发器来防止亚稳态。`spinal.lib._` 中提供的 `BufferCC(input: T, init: T = null, bufferDepth: Int = 2)` 函数将实例化必要的触发器（触发器的数量将取决于 `bufferDepth` 参数）来减轻这种现象。

```
class CrossingExample(clkA : ClockDomain,clkB : ClockDomain) extends Component {
  val io = new Bundle {
    val dataIn  = in Bool()
    val dataOut = out Bool()
  }

  // sample dataIn with clkA
  val area_clkA = new ClockingArea(clkA) {
    val reg = RegNext(io.dataIn) init(False)
  }
```

(续下页)

(接上页)

```
  // BufferCC to avoid metastability issues
  val area_clkB = new ClockingArea(clkB) {
    val buf1    = BufferCC(area_clkA.reg, False)
  }

  io.dataOut := area_clkB.buf1
}
```

警告： BufferCC 函数仅适用于 Bit 类型的信号，或作为格雷编码计数器运行的 Bits 信号（每个时钟周期仅翻转 1 位），并且不能用于多位跨时钟域信号。对于多位情况，建议使用 StreamFifoCC 来满足高带宽要求，或者在带宽要求不高的情况下使用 StreamCCByToggle 来减少资源使用。

5.4.4 特殊计时逻辑区

慢时钟逻辑区

SlowArea 用于创建一个逻辑区，使用比当前时钟域慢的新时钟域：

```
class TopLevel extends Component {

  // Use the current clock domain : 100MHz
  val areaStd = new Area {
    val counter = out(CounterFreeRun(16).value)
  }

  // Slow the current clockDomain by 4 : 25 MHz
  val areaDiv4 = new SlowArea(4) {
    val counter = out(CounterFreeRun(16).value)
  }

  // Slow the current clockDomain to 50MHz
  val area50Mhz = new SlowArea(50 MHz) {
    val counter = out(CounterFreeRun(16).value)
  }
}

def main(args: Array[String]) {
  new SpinalConfig(
    defaultClockDomainFrequency = FixedFrequency(100 MHz)
  ).generateVhdl(new TopLevel)
}
```

警告： SlowArea 中使用的时钟信号与父区域相同。而 SlowArea 会添加一个时钟启用信号，以减慢其内部的采样率。换句话说，ClockDomain.current.readClockWire 将返回快速（父域）时钟。要获取时钟使能信号，请使用 ClockDomain.current.readClockEnableWire

启动复位

clockDomain.withBootReset() 可以指定寄存器的 resetKind 为 BOOT。*clockDomain.withSyncReset()* 可以指定寄存器的 resetKind 为 SYNC（同步复位）。

```
class  Top extends Component {
    val io = new Bundle {
      val data = in Bits(8 bit)
      val a, b, c, d = out Bits(8 bit)
    }
    io.a  :=  RegNext(io.data) init 0
    io.b  :=  clockDomain.withBootReset()  on RegNext(io.data) init 0
    io.c  :=  clockDomain.withSyncReset()  on RegNext(io.data) init 0
    io.d  :=  clockDomain.withAsyncReset() on RegNext(io.data) init 0
}
SpinalVerilog(new Top)
```

复位时钟域

ResetArea 用于创建一个新的时钟域区域，其使用指定的复位信号与当前时钟域复位相结合进行复位：

```
class TopLevel extends Component {

  val specialReset = Bool()

  // The reset of this area is done with the specialReset signal
  val areaRst_1 = new ResetArea(specialReset, false) {
    val counter = out(CounterFreeRun(16).value)
  }

  // The reset of this area is a combination between the current reset and the␣
↪specialReset
  val areaRst_2 = new ResetArea(specialReset, true) {
    val counter = out(CounterFreeRun(16).value)
  }
}
```

时钟使能逻辑区

ClockEnableArea 用于在当前时钟域中添加额外的时钟使能信号：

```
class TopLevel extends Component {

  val clockEnable = Bool()

  // Add a clock enable for this area
  val area_1 = new ClockEnableArea(clockEnable) {
    val counter = out(CounterFreeRun(16).value)
  }
}
```

5.5 实例化 VHDL 和 Verilog IP

5.5.1 描述

黑盒允许用户通过指定其接口将现有的 VHDL/Verilog 组件集成到设计中。正确地进行实力细化取决于仿真器或综合器。

5.5.2 定义一个黑盒

下面示例显示了定义黑盒的方法：

```
// Define a Ram as a BlackBox
class Ram_1w_1r(wordWidth: Int, wordCount: Int) extends BlackBox {
  // Add VHDL Generics / Verilog parameters to the blackbox
  // You can use String, Int, Double, Boolean, and all SpinalHDL base
  // types as generic values
  addGeneric("wordCount", wordCount)
  addGeneric("wordWidth", wordWidth)

  // Define IO of the VHDL entity / Verilog module
  val io = new Bundle {
    val clk = in Bool()
    val wr = new Bundle {
      val en   = in Bool()
      val addr = in UInt (log2Up(wordCount) bits)
      val data = in Bits (wordWidth bits)
    }
    val rd = new Bundle {
      val en   = in Bool()
      val addr = in UInt (log2Up(wordCount) bits)
      val data = out Bits (wordWidth bits)
    }
  }

  // Map the current clock domain to the io.clk pin
  mapClockDomain(clock=io.clk)
}
```

在 VHDL 中，`Bool` 类型的信号将被转换为 `std_logic`，`Bits` 将被转换为 `std_logic_vector`。如果你想获得 `std_ulogic`，你必须使用 `BlackBoxULogic` 而不是 `BlackBox`。
在 Verilog 中，`BlackBoxUlogic` 不会更改生成的 Verilog。

```
class Ram_1w_1r(wordWidth: Int, wordCount: Int) extends BlackBoxULogic {
  ...
}
```

5.5.3 泛型

有两种不同的方式来声明泛型：

```
class Ram(wordWidth: Int, wordCount: Int) extends BlackBox {
    addGeneric("wordCount", wordCount)
    addGeneric("wordWidth", wordWidth)

    // OR

    val generic = new Generic {
      val wordCount = Ram.this.wordCount
      val wordWidth = Ram.this.wordWidth
    }
}
```

5.5.4 实例化黑盒

实例化一个 BlackBox 就像实例化一个 Component 一样：

```
// Create the top level and instantiate the Ram
class TopLevel extends Component {
  val io = new Bundle {
    val wr = new Bundle {
      val en   = in Bool()
      val addr = in UInt (log2Up(16) bits)
      val data = in Bits (8 bits)
    }
    val rd = new Bundle {
      val en   = in Bool()
      val addr = in UInt (log2Up(16) bits)
      val data = out Bits (8 bits)
    }
  }

  // Instantiate the blackbox
  val ram = new Ram_1w_1r(8,16)

  // Connect all the signals
  io.wr.en   <> ram.io.wr.en
  io.wr.addr <> ram.io.wr.addr
  io.wr.data <> ram.io.wr.data
  io.rd.en   <> ram.io.rd.en
  io.rd.addr <> ram.io.rd.addr
  io.rd.data <> ram.io.rd.data
}

object Main {
  def main(args: Array[String]): Unit = {
    SpinalVhdl(new TopLevel)
  }
}
```

5.5.5 时钟和复位信号的映射

在黑盒定义中，您必须明确定义时钟和复位线。要将 ClockDomain 的信号映射到黑盒的相应输入，您可以使用 mapClockDomain 或 mapCurrentClockDomain 函数。mapClockDomain 具有以下参数：

名称	类型	缺省值	描述
clockDomain	ClockDomain	ClockDomain.current	指定提供信号的 clockDomain
时钟	Bool	Nothing	应连接到 clockDomain 时钟的黑盒输入
reset	Bool	Nothing	黑盒输入应连接到时钟域的复位信号
enable	Bool	Nothing	黑盒输入应连接到时钟域的使能信号

mapCurrentClockDomain 具有与 mapClockDomain 几乎相同的参数，但没有时钟域。

例如：

```
class MyRam(clkDomain: ClockDomain) extends BlackBox {

  val io = new Bundle {
    val clkA = in Bool()
    ...
    val clkB = in Bool()
    ...
  }

  // Clock A is map on a specific clock Domain
  mapClockDomain(clkDomain, io.clkA)
  // Clock B is map on the current clock domain
  mapCurrentClockDomain(io.clkB)
}
```

默认情况下，黑盒模块的端口是不绑定时钟域的，这意味着在使用这些端口时不会进行时钟交叉检查。您可以使用 ClockDomainTag 指定端口的时钟域：

```
class DemoBlackbox extends BlackBox {
  val io = new Bundle {
    val clk, rst = in Bool()
    val a = in Bool()
    val b = out Bool()
  }
  mapCurrentClockDomain(io.clk, io.rst)
  ClockDomainTag(this.clockDomain)(
    io.a,
    io.b
  )
}
```

您也可以将标记应用于整个线束：

```
val io = new Bundle {
  val clk, rst = in Bool()
  val a = in Bool()
  val b = out Bool()
}
ClockDomainTag(this.clockDomain)(io)
```

从 SpinalHDL 1.10.2 开始，您还可以将当前时钟域应用到所有端口：

```
val io = new Bundle {
  val clk, rst = in Bool()
  val a = in Bool()
```

(续下页)

(接上页)

```
  val b = out Bool()
}
setIoCd()
```

5.5.6 io 前缀

为了避免黑盒的每个 IO 上都有前缀 “io_”，可以使用函数 `noIoPrefix()` ，如下所示：

```
// Define the Ram as a BlackBox
class Ram_1w_1r(wordWidth: Int, wordCount: Int) extends BlackBox {

  val generic = new Generic {
    val wordCount = Ram_1w_1r.this.wordCount
    val wordWidth = Ram_1w_1r.this.wordWidth
  }

  val io = new Bundle {
    val clk = in Bool()

    val wr = new Bundle {
      val en   = in Bool()
      val addr = in UInt (log2Up(_wordCount) bits)
      val data = in Bits (_wordWidth bits)
    }
    val rd = new Bundle {
      val en   = in Bool()
      val addr = in UInt (log2Up(_wordCount) bits)
      val data = out Bits (_wordWidth bits)
    }
  }

  noIoPrefix()

  mapCurrentClockDomain(clock=io.clk)
}
```

5.5.7 重命名黑盒中的所有 io

`BlackBox` 或 `Component` 的 IO 可以在编译时使用 `addPrePopTask` 函数重命名。此函数在编译期间调用一个无参数函数，对于添加重命名通道非常有用，如下所示：

```
class MyRam() extends Blackbox {

  val io = new Bundle {
    val clk = in Bool()
    val portA = new Bundle {
      val cs   = in Bool()
      val rwn  = in Bool()
      val dIn  = in Bits(32 bits)
      val dOut = out Bits(32 bits)
    }
    val portB = new Bundle {
      val cs   = in Bool()
      val rwn  = in Bool()
      val dIn  = in Bits(32 bits)
      val dOut = out Bits(32 bits)
    }
```

(续下页)

(接上页)

```
  }

  // Map the clk
  mapCurrentClockDomain(io.clk)

  // Remove io_ prefix
  noIoPrefix()

  // Function used to rename all signals of the blackbox
  private def renameIO(): Unit = {
    io.flatten.foreach(bt => {
      if(bt.getName().contains("portA")) bt.setName(bt.getName().replace("portA_",
↪"") + "_A")
      if(bt.getName().contains("portB")) bt.setName(bt.getName().replace("portB_",
↪"") + "_B")
    })
  }

  // Execute the function renameIO after the creation of the component
  addPrePopTask(() => renameIO())
}

// This code generate these names:
//     clk
//     cs_A, rwn_A, dIn_A, dOut_A
//     cs_B, rwn_B, dIn_B, dOut_B
```

5.5.8 添加 RTL 源

使用函数 `addRTLPath()`，您可以将 RTL 源与黑盒关联起来。生成 SpinalHDL 代码后，您可以调用函数 `mergeRTLSource` 将所有源合并在一起。

```
class MyBlackBox() extends Blackbox {

  val io = new Bundle {
    val clk   = in  Bool()
    val start = in Bool()
    val dIn   = in  Bits(32 bits)
    val dOut  = out Bits(32 bits)
    val ready = out Bool()
  }

  // Map the clk
  mapCurrentClockDomain(io.clk)

  // Remove io_ prefix
  noIoPrefix()

  // Add all rtl dependencies
  addRTLPath("./rtl/RegisterBank.v")                        // Add a verilog file
  addRTLPath(s"./rtl/myDesign.vhd")                         // Add a vhdl file
  addRTLPath(s"${sys.env("MY_PROJECT")}/myTopLevel.vhd")    // Use an environment␣
↪variable MY_PROJECT (System.getenv("MY_PROJECT"))
}

...

class TopLevel() extends Component {
  // ...
```

(续下页)

(接上页)

```
  val bb = new MyBlackBox()
  // ...
}

val report = SpinalVhdl(new TopLevel)
report.mergeRTLSource("mergeRTL") // Merge all rtl sources into mergeRTL.vhd and␣
↪mergeRTL.v files
```

5.5.9 VHDL - 无数值类型

如果您只想在黑盒组件中使用 `std_logic_vector`，则可以将标签 `noNumericType` 添加到黑盒中。

```
class MyBlackBox() extends BlackBox {
  val io = new Bundle {
    val clk       = in  Bool()
    val increment = in  Bool()
    val initValue = in  UInt(8 bits)
    val counter   = out UInt(8 bits)
  }

  mapCurrentClockDomain(io.clk)

  noIoPrefix()

  addTag(noNumericType)  // Only std_logic_vector
}
```

上面的代码将生成以下 VHDL：

```
component MyBlackBox is
  port(
    clk       : in  std_logic;
    increment : in  std_logic;
    initValue : in  std_logic_vector(7 downto 0);
    counter   : out std_logic_vector(7 downto 0)
  );
end component;
```

5.6 保留名称的方法

本页将描述 SpinalHDL 如何将名称从 scala 代码传播到生成的硬件 RTL。您应该了解它们，从而尽可能保留这些名称，以生成可理解的网表。

5.6.1 Nameable 基类

所有可以在 SpinalHDL 中命名的事物都扩展了 Nameable 基类。

因此在实践中，以下类扩展了 Nameable 类：

- Component
- Area
- Data (UInt, SInt, Bundle, ···)

有一些 Nameable 类型 API 的示例

```
class MyComponent extends Component {
  val a, b, c, d = Bool()
  b.setName("rawrr") // Force name
  c.setName("rawrr", weak = true) // Propose a name, will not be applied if a␣
↪stronger name is already applied
  d.setCompositeName(b, postfix = "wuff") // Force toto to be named as b.getName()␣
↪+ _wuff"
}
```

会生成：

```
module MyComponent (
);
  wire                a;
  wire                rawrr;
  wire                c;
  wire                rawrr_wuff;
endmodule
```

In general, you don’t really need to access that API, unless you want to do tricky stuff for debug reasons or for elaboration purposes.

5.6.2 从 Scala 中提取名称

首先，从 1.4.0 版本开始，SpinalHDL 使用 scala 编译器插件，该插件可以在类构造期间在每次定义新 val 时，实现函数回调。

这个示例或多或少地展示了 SpinalHDL 本身是如何实现的：

```
// spinal.idslplugin.ValCallback is the Scala compiler plugin feature which will␣
↪provide the callbacks
class Component extends spinal.idslplugin.ValCallback {
  override def valCallback[T](ref: T, name: String) : T = {
    println(s"Got $ref named $name") // Here we just print what we got as a demo.
    ref
  }
}

class UInt
class Bits
class MyComponent extends Component {
  val two = 2
  val wuff = "miaou"
  val toto = new UInt
  val rawrr = new Bits
}

object Debug3 extends App {
  new MyComponent()
  // ^ This will print :
  // Got 2 named two
  // Got miaou named wuff
  // Got spinal.tester.code.sandbox.UInt@691a7f8f named toto
  // Got spinal.tester.code.sandbox.Bits@161b062a named rawrr
}
```

使用 ValCallback “自省” 功能，SpinalHDL 的组件类能够了解其内容和内容的名称。

但这也意味着，如果您希望某些东西获得名称，并且仅依赖于此自动命名功能，则对 Data (UInt、SInt、…) 实例的引用应存储在组件的某个 val 对象定义中。

例如：

```
class MyComponent extends Component {
  val a,b = in UInt(8 bits) // Will be properly named
  val toto = out UInt(8 bits)   // same

  def doStuff(): Unit = {
    val tmp = UInt(8 bits) // This will not be named, as it isn't stored anywhere
↪in a
                           // component val (but there is a solution explained
↪later)
    tmp := 0x20
    toto := tmp
  }
  doStuff()
}
```

将生成：

```
module MyComponent (
  input       [7:0]    a,
  input       [7:0]    b,
  output      [7:0]    toto
);
  // Note that the tmp signal defined in scala was "shortcuted" by SpinalHDL,
  //  as it was unnamed and technically "shortcutable"
  assign toto = 8'h20;
endmodule
```

5.6.3 组件中的区域

命名系统的一个重要方面是您可以在组件内定义新的名称空间并进行操作

例如通过 Area：

```
class MyComponent extends Component {
  val logicA = new Area {     // This define a new namespace named "logicA
    val toggle = Reg(Bool()) // This register will be named "logicA_toggle"
    toggle := !toggle
  }
}
```

会生成

```
module MyComponent (
  input               clk,
  input               reset
);
  reg                 logicA_toggle;
  always @ (posedge clk) begin
    logicA_toggle <= (! logicA_toggle);
  end
endmodule
```

5.6.4 函数中的逻辑区

您还可以定义将创建新逻辑区的函数，该逻辑区将为其所有内容提供命名空间：

```
class MyComponent extends Component {
  def isZero(value: UInt) = new Area {
    val comparator = value === 0
  }

  val value = in UInt (8 bits)
  val someLogic = isZero(value)

  val result = out Bool()
  result := someLogic.comparator
}
```

这将生成：

```
module MyComponent (
  input       [7:0]     value,
  output                result
);
  wire                  someLogic_comparator;

  assign someLogic_comparator = (value == 8'h0);
  assign result = someLogic_comparator;

endmodule
```

5.6.5 函数中的复合区（Composite）

SpinalHDL 1.5.0 中添加了复合区，它允许您创建一个范围，该范围将用作另一个 Nameable 的前缀：

```
class MyComponent extends Component {
  // Basically, a Composite is an Area that use its construction parameter as␣
↪namespace prefix
  def isZero(value: UInt) = new Composite(value) {
    val comparator = value === 0
  }.comparator  // Note we don't return the Composite,
                //  but the element of the composite that we are interested in

  val value = in UInt (8 bits)
  val result = out Bool()
  result := isZero(value)
}
```

将生成：

```
module MyComponent (
  input       [7:0]     value,
  output                result
);
  wire                  value_comparator;

  assign value_comparator = (value == 8'h0);
  assign result = value_comparator;

endmodule
```

5.6.6 复合区级联链

您还可以级联复合区：

```
class MyComponent extends Component {
  def isZero(value: UInt) = new Composite(value) {
    val comparator = value === 0
  }.comparator


  def inverted(value: Bool) = new Composite(value) {
    val inverter = !value
  }.inverter

  val value = in UInt(8 bits)
  val result = out Bool()
  result := inverted(isZero(value))
}
```

将生成：

```
module MyComponent (
  input      [7:0]    value,
  output              result
);
  wire                value_comparator;
  wire                value_comparator_inverter;

  assign value_comparator = (value == 8'h0);
  assign value_comparator_inverter = (! value_comparator);
  assign result = value_comparator_inverter;

endmodule
```

5.6.7 在一个线束（Bundle）的函数中的复合区

This behavior can be very useful when implementing Bundle utilities. For instance in the spinal.lib.Stream class is defined the following :

```
class Stream[T <: Data](val payloadType :  HardType[T]) extends Bundle {
  val valid   = Bool()
  val ready   = Bool()
  val payload = payloadType()

  def queue(size: Int): Stream[T] = new Composite(this) {
    val fifo = new StreamFifo(payloadType, size)
    fifo.io.push << self    // 'self' refers to the Composite construction␣
↪argument ('this' in
                            //  the example). It avoids having to do a boring
↪'Stream.this'
  }.fifo.io.pop

  def m2sPipe(): Stream[T] = new Composite(this) {
    val m2sPipe = Stream(payloadType)

    val rValid = RegInit(False)
    val rData = Reg(payloadType)

    self.ready := (!m2sPipe.valid) || m2sPipe.ready
```

(续下页)

(接上页)

```
    when(self.ready) {
      rValid := self.valid
      rData := self.payload
    }

    m2sPipe.valid := rValid
    m2sPipe.payload := rData
  }.m2sPipe
}
```

这将允许嵌套调用，同时保留名称：

```
class MyComponent extends Component {
  val source = slave(Stream(UInt(8 bits)))
  val sink = master(Stream(UInt(8 bits)))
  sink << source.queue(size = 16).m2sPipe()
}
```

会生成

```
module MyComponent (
  input               source_valid,
  output              source_ready,
  input       [7:0]   source_payload,
  output              sink_valid,
  input               sink_ready,
  output      [7:0]   sink_payload,
  input               clk,
  input               reset
);
  wire                source_fifo_io_pop_ready;
  wire                source_fifo_io_push_ready;
  wire                source_fifo_io_pop_valid;
  wire        [7:0]   source_fifo_io_pop_payload;
  wire        [4:0]   source_fifo_io_occupancy;
  wire        [4:0]   source_fifo_io_availability;
  wire                source_fifo_io_pop_m2sPipe_valid;
  wire                source_fifo_io_pop_m2sPipe_ready;
  wire        [7:0]   source_fifo_io_pop_m2sPipe_payload;
  reg                 source_fifo_io_pop_rValid;
  reg         [7:0]   source_fifo_io_pop_rData;

  StreamFifo source_fifo (
    .io_push_valid      (source_valid                 ), //i
    .io_push_ready      (source_fifo_io_push_ready    ), //o
    .io_push_payload    (source_payload               ), //i
    .io_pop_valid       (source_fifo_io_pop_valid     ), //o
    .io_pop_ready       (source_fifo_io_pop_ready     ), //i
    .io_pop_payload     (source_fifo_io_pop_payload   ), //o
    .io_flush           (1'b0                         ), //i
    .io_occupancy       (source_fifo_io_occupancy     ), //o
    .io_availability    (source_fifo_io_availability  ), //o
    .clk                (clk                          ), //i
    .reset              (reset                        )  //i
  );
  assign source_ready = source_fifo_io_push_ready;
  assign source_fifo_io_pop_ready = ((1'b1 && (! source_fifo_io_pop_m2sPipe_
↪valid)) || source_fifo_io_pop_m2sPipe_ready);
  assign source_fifo_io_pop_m2sPipe_valid = source_fifo_io_pop_rValid;
  assign source_fifo_io_pop_m2sPipe_payload = source_fifo_io_pop_rData;
  assign sink_valid = source_fifo_io_pop_m2sPipe_valid;
```

(续下页)

(接上页)

```
  assign source_fifo_io_pop_m2sPipe_ready = sink_ready;
  assign sink_payload = source_fifo_io_pop_m2sPipe_payload;
  always @ (posedge clk or posedge reset) begin
    if (reset) begin
      source_fifo_io_pop_rValid <= 1'b0;
    end else begin
      if(source_fifo_io_pop_ready)begin
        source_fifo_io_pop_rValid <= source_fifo_io_pop_valid;
      end
    end
  end

  always @ (posedge clk) begin
    if(source_fifo_io_pop_ready)begin
      source_fifo_io_pop_rData <= source_fifo_io_pop_payload;
    end
  end
endmodule
```

5.6.8 Unnamed signal handling

Since 1.5.0, for signal which end up without name, SpinalHDL will find a signal which is driven by that unnamed signal and propagate its name. This can produce useful results as long you don't have too large island of unnamed stuff.

The name attributed to such unnamed signal is : _zz_ + drivenSignal.getName()

请注意，当生成后端需要将某些特定表达式或表达式链分解为多个信号时，也会使用此命名模式。

Verilog 表达式分割

There is an instance of expressions (ex : the + operator) that SpinalHDL need to express in dedicated signals to match the behavior with the Scala API :

```
class MyComponent extends Component {
  val a,b,c,d = in UInt(8 bits)
  val result = a + b + c + d
}
```

会生成

```
module MyComponent (
  input      [7:0]    a,
  input      [7:0]    b,
  input      [7:0]    c,
  input      [7:0]    d
);
  wire       [7:0]    _zz_result;
  wire       [7:0]    _zz_result_1;
  wire       [7:0]    result;

  assign _zz_result = (_zz_result_1 + c);
  assign _zz_result_1 = (a + b);
  assign result = (_zz_result + d);

endmodule
```

Verilog 长表达式分割

There is a instance of how a very long expression chain will be split up by SpinalHDL :

```
class MyComponent extends Component {
  val conditions = in Vec(Bool(), 64)
  // Perform a logical OR between all the condition elements
  val result = conditions.reduce(_ || _)

  // For Bits/UInt/SInt signals the 'orR' methods implements this reduction␣
↪operation
}
```

会生成

```
module MyComponent (
  input               conditions_0,
  input               conditions_1,
  input               conditions_2,
  input               conditions_3,
  ...
  input               conditions_58,
  input               conditions_59,
  input               conditions_60,
  input               conditions_61,
  input               conditions_62,
  input               conditions_63
);
  wire                _zz_result;
  wire                _zz_result_1;
  wire                _zz_result_2;
  wire                result;

  assign _zz_result = ((((((((((((((((_zz_result_1 || conditions_32) || conditions_
↪33) || conditions_34) || conditions_35) || conditions_36) || conditions_37) ||␣
↪conditions_38) || conditions_39) || conditions_40) || conditions_41) ||␣
↪conditions_42) || conditions_43) || conditions_44) || conditions_45) ||␣
↪conditions_46) || conditions_47);
  assign _zz_result_1 = ((((((((((((((_zz_result_2 || conditions_16) ||␣
↪conditions_17) || conditions_18) || conditions_19) || conditions_20) ||␣
↪conditions_21) || conditions_22) || conditions_23) || conditions_24) ||␣
↪conditions_25) || conditions_26) || conditions_27) || conditions_28) ||␣
↪conditions_29) || conditions_30) || conditions_31);
  assign _zz_result_2 = ((((((((((((((conditions_0 || conditions_1) || conditions_
↪2) || conditions_3) || conditions_4) || conditions_5) || conditions_6) ||␣
↪conditions_7) || conditions_8) || conditions_9) || conditions_10) || conditions_
↪11) || conditions_12) || conditions_13) || conditions_14) || conditions_15);
  assign result = ((((((((((((((_zz_result || conditions_48) || conditions_49)␣
↪|| conditions_50) || conditions_51) || conditions_52) || conditions_53) ||␣
↪conditions_54) || conditions_55) || conditions_56) || conditions_57) ||␣
↪conditions_58) || conditions_59) || conditions_60) || conditions_61) ||␣
↪conditions_62) || conditions_63);

endmodule
```

When 语句条件

when(cond) { } 语句条件生成为名为 *when_* + fileName + line 的单独信号。对 switch 语句也会做类似的事情。

```
// In file Test.scala
class MyComponent extends Component {
  val value = in UInt(8 bits)
  val isZero = out(Bool())
  val counter = out(Reg(UInt(8 bits)))

  isZero := False
  when(value === 0) { // At line 117
    isZero := True
    counter := counter + 1
  }
}
```

会生成

```
module MyComponent (
  input       [7:0]     value,
  output reg            isZero,
  output reg [7:0]      counter,
  input                 clk,
  input                 reset
);
  wire                  when_Test_l117;

  always @ (*) begin
    isZero = 1'b0;
    if(when_Test_l117)begin
      isZero = 1'b1;
    end
  end

  assign when_Test_l117 = (value == 8'h0);
  always @ (posedge clk) begin
    if(when_Test_l117)begin
      counter <= (counter + 8'h01);
    end
  end
endmodule
```

最后一招

最后，如果信号没有名称（匿名信号），SpinalHDL 将寻找由匿名信号驱动的命名信号，并将其用作名称后缀：

```
class MyComponent extends Component {
  val enable = in Bool()
  val value = out UInt(8 bits)

  def count(cond : Bool): UInt = {
    val ret = Reg(UInt(8 bits)) // This register is not named (on purpose for the␣
↪example)
    when(cond) {
      ret := ret + 1
    }
    return ret
  }
```

(续下页)

(接上页)

```
  value := count(enable)
}
```

会生成

```
module MyComponent (
  input               enable,
  output      [7:0]   value,
  input               clk,
  input               reset
);
  // Name given to the register in last resort by looking what was driven by it
  reg         [7:0]   _zz_value;

  assign value = _zz_value;
  always @ (posedge clk) begin
    if(enable)begin
      _zz_value <= (_zz_value + 8'h01);
    end
  end
endmodule
```

最后的命名方法并不适合所有情况，但可以提供帮助。

请注意，以下划线开头的信号不会存储在 Verilator 波形中（这是故意的）

5.7 参数化

参数化有多个方面的含义：

- 在设计实例细化（elaboration）过程中向 SpinalHDL 提供细化过程的参数并对其进行管理
- 通过使用参数，设计人员可以实现任何类型硬件的构造、配置和互连任务。例如硬件设计中的可选组件生成。

与 HDL 泛化功能（例如 Verilog 模块参数和 VHDL 泛型）的目标类似。SpinalHDL 通过 Scala 类型安全和内置 HDL 设计规则检查带来额外保护，提供了更丰富、更强大的功能集。

用于组件参数化的 SpinalHDL 机制不是构建在任何 HDL 机制之上，因此，不会受到 HDL 语言级别/版本支持或手写 HDL 带来限制的影响。

对于希望使用 SpinalHDL 与参数化 Verilog 或通用 VHDL 进行互操作的读者，请参阅*BlackBox* IP 中有关您的项目所需场景的部分。

5.7.1 实例细化时参数

您可以使用所有 Scala 语法来提供实例细化时参数。

所有的语法意味着您可以使用 Scala 语言的全部功能，以按照您选择的复杂程度解决项目的参数化要求。

SpinalHDL 不会对如何实现参数化目标施加任何特殊限制。因此，有许多 Scala 设计模式和一些 SpinalHDL 帮助程序可用于管理参数，以适合不同场景。

以下是一些示例和可能的想法：

- Hardwired code and constants (not strictly parameter management at all but serves to highlight the most basic mechanism, a code change, not a parameter data change)
- 由伴随对象提供的常量值在 Scala 中是静态常量。
- 提供给 Scala 类构造函数的值，通常一个 `case class` 会导致 Scala 将这些构造函数的参数值捕获为常量。

- 常规 Scala 流程控制语法，包括但不限于条件、循环、lambda/monad 等。
- 配置类的模式（示例存在于库中，包括 UartCtrlConfig, SpiMasterCtrlConfig 等）
- 项目定义的 “插件” 模式（在 VexRiscV 项目中有示例，用于配置生成的 CPU IP 核所使用的功能集）
- 使用标准 Scala/JVM 库和 API 从文件或网络源中加载值和信息。
- “任何你可以创建的机制”

所有机制都会导致实力细化生成的 HDL 输出发生变化。

这可以仅仅使用 Scala 编程就完成设计，支持单个常数值到整个 SoC 的所有总线和互连架构描述的参数化。

这是一个类参数的示例

```
case class MyBus(width : Int) extends Bundle {
  val mySignal = UInt(width bits)
}
```

```
case class MyComponent(width : Int) extends Component {
  val bus = MyBus(width)
}
```

您还可以使用 Scala 对象（伴随对象模式）中定义的全局变量。

ScopeProperty 也可用于配置。

5.7.2 可选硬件

所以这里还有更多的可能性。

对于可选信号：

```
case class MyComponent(flag : Boolean) extends Component {
  val mySignal = flag generate (Bool())
  // equivalent to "val mySignal = if (flag) Bool() else null"
}
```

`generate` 函数是一种实现具有可选值的表达式机制。如果谓词为 true，generate 将计算给定表达式并返回结果，否则返回 null。

This may be used in cases to help parameterize the SpinalHDL hardware description using an elaboration-time conditional expression. Causing HDL constructs to be emitted or not-emitted in the resulting HDL. The generate method can be seen as SpinalHDL syntactic sugar reducing language clutter.

Project SpinalHDL code referencing `mySignal` would need to ensure it handles the possibility of null gracefully. This is usually not a problem as those parts of the design can also be omitted dependant on the `flag` value. Thus the feature of parameterizing this component is demonstrated.

您也可以在线束（Bundle）中做相同的事。

请注意，您还可以使用 scala 选项类（Option）。

如果你想禁用某个硬件块的生成：

```
case class MyComponent(flag : Boolean) extends Component {
  val myHardware = flag generate new Area {
    // optional hardware here
  }
}
```

您还可以使用 scala 中的 for 循环：

```
case class MyComponent(amount : Int) extends Component {
  val myHardware = for(i <- 0 until amount) yield new Area {
    // hardware here
  }
}
```

因此，您可以在实例细化时根据需要扩展这些 scala 用法，包括使用整个 scala 集合（List、Set、Map…）来构建一些数据模型，然后以程序方式将它们转换为硬件（例如，迭代这些列表中的元素）。

CHAPTER 6

语义

6.1 赋值

有多个赋值运算符：

符号	描述
:=	标准赋值，相当于 VHDL/Verilog 中的 “<=“。
\=	相当于 VHDL 中的 := 和 Verilog 中的 =。该值会立即就地更新。仅适用于组合信号，不适用于寄存器。
<>	2 个信号或相同类型的两个信号线束之间的自动连接。通过使用信号定义（输入/输出）来推断方向。（与 := 类似的行为）

When muxing (for instance using `when`, see *When/Switch/Mux*.), the last valid standard assignment := wins. Else, assigning twice to the same assignee from the same scope results in an assignment overlap. SpinalHDL will assume this is a unintentional design error by default and halt elaboration with error. For special use-cases assignment overlap can be programmatically permitted on a case by case basis. (see 赋值覆盖 *(Assignment overlap)*).

```
val a, b, c = UInt(4 bits)
a := 0
b := a
// a := 1 // this would cause an `assignment overlap` error,
          // if manually overridden the assignment would take assignment priority
c := a

var x = UInt(4 bits)
val y, z = UInt(4 bits)
x := 0
y := x      // y read x with the value 0
x \= x + 1
z := x      // z read x with the value 1

// Automatic connection between two UART interfaces.
uartCtrl.io.uart <> io.uart
```

它还支持线束赋值（将所有位信号转换为适当位宽的 *Bits* 类型的单位总线，然后在赋值表达式中使用更宽的形式）。在赋值表达式的左侧和右侧使用 () （Scala 元组语法）将多个信号捆绑在一起。

```
val a, b, c = UInt(4 bits)
val d       = UInt(12 bits)
val e       = Bits(10 bits)
val f       = SInt(2  bits)
val g       = Bits()

(a, b, c) := B(0, 12 bits)
(a, b, c) := d.asBits
(a, b, c) := (e, f).asBits           // both sides
g         := (a, b, c, e, f).asBits  // and on the right hand side
```

重要的是要理解，在 SpinalHDL 中，信号的性质（组合/时序）是在其声明中定义的，而不是通过赋值的方式定义的。所有数据类型实例都将定义一个组合信号，而用 `Reg(...)` 包装的实例将定义为一个时序信号（寄存器）。

```
val a = UInt(4 bits)               // Define a combinational signal
val b = Reg(UInt(4 bits))          // Define a registered signal
val c = Reg(UInt(4 bits)) init(0)  // Define a registered signal which is
                                   //  set to 0 when a reset occurs
```

6.1.1 位宽检查

SpinalHDL 检查赋值左侧和右侧的位数是否匹配。有多种方法可以改变给定 BitVector （`Bits`, `UInt`, `SInt`）的位宽：

调整位宽的技术	描述
x := y.resized	将 y 改变位宽后的副本分配给 x，其位宽是从 x 推断出来的。
x := y.resize(newWidth)	为 x 赋值一个 y 变为 newWidth 位宽后的副本。
x := y.resizeLeft(newWidth)	对 x 赋值 y 变为 newWidth 位宽后的副本。如果需要，可在 LSB 处进行填充。

所有改变位宽方法都可能导致生成的位宽比 y 的原始位宽更宽或更窄。当发生加宽时，额外的位将用零填充。

`x.resized` 根据赋值表达式左侧的目标位宽推断转换方法，并遵循与 `y.resize(someWidth)` 相同的语义。表达式 `x := y.resized` 相当于 `x := y.resize(x.getBitsWidth bits)`。

虽然示例代码片段显示了赋值语句的使用方法，但 resize 系列方法可以像任何普通 Scala 方法一样进行级联。

在一种情况下，Spinal 会自动调整位宽的大小：

```
// U(3) creates an UInt of 2 bits, which doesn't match the left side (8 bits)
myUIntOf_8bits := U(3)
```

因为 `U(3)` 是一个 “弱” 位计数推断信号，SpinalHDL 会自动加宽它。这可以被认为在功能上等同于 `U(3, 2 bits).resized`，但是请放心，如果场景需要缩小范围，SpinalHDL 将做正确的事情并报告错误。当尝试赋值 `myUIntOf_8bits`` 时，如果字面量需要 9 位（例如 ``U(0x100)`），则会报告错误。

6.1.2 组合逻辑环 (Combinatorial loops)

SpinalHDL 检查您的设计中是否存在组合逻辑环（锁存器）。如果检测到，会引发错误，并且 SpinalHDL 将打印造成循环的路径。

6.1.3 CombInit

`CombInit` 可用于复制信号及其当前的组合逻辑赋值。主要用例是能够稍后覆盖复制后信号，而不影响原始信号。

```
val a = UInt(8 bits)
a := 1

val b = a
when(sel) {
    b := 2
    // At this point, a and b are evaluated to 2: they reference the same signal
}

val c = UInt(8 bits)
c := 1

val d = CombInit(c)
// Here c and d are evaluated to 1
when(sel) {
    d := 2
    // At this point c === 1 and d === 2.
}
```

如果我们查看生成的 Verilog，会发现 “b“ 不存在。由于它是引用的 `a` 的副本，因此这些变量指代相同的 Verilog 信号。

```
always @(*) begin
  a = 8'h01;
  if(sel) begin
    a = 8'h02;
  end
end

assign c = 8'h01;
always @(*) begin
  d = c;
  if(sel) begin
    d = 8'h02;
  end
end
```

`CombInit` 在辅助函数中特别有用，可确保返回值不引用输入。

```
// note that condition is an elaboration time constant
def invertedIf(b: Bits, condition: Boolean): Bits = if(condition) { ~b } else {␣
↪CombInit(b) }

val a2 = invertedIf(a1, c)

when(sel) {
   a2 := 0
}
```

Without `CombInit`, if `c` == false (but not if `c` == true), `a1` and `a2` reference the same signal and the zero assignment is also applied to `a1`. With `CombInit` we have a coherent behavior whatever the `c` value.

6.2 When/Switch/Mux

6.2.1 When

与 VHDL 和 Verilog 中一样，当满足指定条件时可以有条件地赋值信号：

```
when(cond1) {
  // Execute when cond1 is true
} elsewhen(cond2) {
  // Execute when (not cond1) and cond2
} otherwise {
  // Execute when (not cond1) and (not cond2)
}
```

警告： 如果关键字 otherwise 与 when 条件的右括号 } 在同一行，则不需要点。

```
when(cond1) {
    // Execute when cond1 is true
} otherwise {
    // Execute when (not cond1) and (not cond2)
}
```

但如果 .otherwise 在另一行，则 ** 需要 ** 一个点：

```
when(cond1) {
    // Execute when cond1 is true
}
.otherwise {
    // Execute when (not cond1) and (not cond2)
}
```

6.2.2 WhenBuilder

有时需要为 when 条件生成一些参数，而 when else 结构并不太合适。因此，我们提供了一个’ whenBuilder’ 方法来实现这个目标

```
import spinal.lib._

val conds = Bits(8 bits)
val result = UInt(8 bits)

val ctx = WhenBuilder()
ctx.when(conds(0)) {
  result := 0
}
ctx.when(conds(1)) {
  result := 1
}
if(true) {
  ctx.when(conds(2)) {
    result := 2
  }
}
ctx.when(conds(3)) {
  result := 3
}
```

与 when/elsewhen/otherwise 方法相比，它可能更方便于参数化。我们也可以这样使用

```
for(i <- 5 to 7) ctx.when(conds(i)) {
  result := i
}

ctx.otherwise {
  result := 255
}

switch(addr) {
  for (i <- addressElements ) {
    is(i) {
      rdata :=  buffer(i)
    }
  }
}
```

这样，我们可以像在 *switch()* 中使用 *foreach* 一样，对优先级电路进行参数化，并以更直观的 if-else 格式生成代码。

6.2.3 Switch

与 VHDL 和 Verilog 中一样，当信号具有定义的值时，可以有条件地对信号赋值：

```
switch(x) {
  is(value1) {
    // Execute when x === value1
  }
  is(value2) {
    // Execute when x === value2
  }
  default {
    // Execute if none of precedent conditions met
  }
}
```

`is` 子句可以通过用逗号 `is(value1, value2)` 分隔来进行分解（逻辑 OR）。

示例

```
switch(aluop) {
  is(ALUOp.add) {
    immediate := instruction.immI.signExtend
  }
  is(ALUOp.slt) {
    immediate := instruction.immI.signExtend
  }
  is(ALUOp.sltu) {
    immediate := instruction.immI.signExtend
  }
  is(ALUOp.sll) {
    immediate := instruction.shamt
  }
  is(ALUOp.sra) {
    immediate := instruction.shamt
  }
}
```

相当于

```
switch(aluop) {
  is(ALUOp.add, ALUOp.slt, ALUOp.sltu) {
      immediate := instruction.immI.signExtend
  }
  is(ALUOp.sll, ALUOp.sra) {
      immediate := instruction.shamt
  }
}
```

其他选项

默认情况下，如果 switch 包含 default 语句，而 switch 的所有可能的逻辑值都已被是 is 语句“覆盖”，SpinalHDL 将生成“UNREACHABLE DEFAULT STATEMENT”错误。您可以通过指定“switch(myValue, coverUnreachable = true) { ···}“来删除此错误报告。

```
switch(my2Bits, coverUnreachable = true) {
    is(0) { ... }
    is(1) { ... }
    is(2) { ... }
    is(3) { ... }
    default { ... } // This will parse and validate without error now
}
```

备注： 此检查是针对逻辑值而不是物理值进行的。例如，如果您有一个以独热编码的 SpinalEnum(A,B,C)，SpinalHDL 将只关心 A,B,C 值（“001”“010”“100”）。物理值“000”“011”“101”“110”“111”将不被考虑。

默认情况下，如果给定的 is 语句多次提供相同的值，SpinalHDL 将生成“DUPLICATED ELEMENTS IN SWITCH IS(···) STATEMENT”错误。例如 is(42,42) { ... } 您可以通过指定 switch(myValue, strict = true){ ... } 来避免报告此错误。SpinalHDL 然后将负责删除重复的值。

```
switch(value, strict = false) {
    is(0) { ... }
    is(1,1,1,1,1) { ... } // This will be okay
    is(2) { ... }
}
```

6.2.4 本地声明

可以在 when/switch 语句中定义新信号：

```
val x, y = UInt(4 bits)
val a, b = UInt(4 bits)

when(cond) {
  val tmp = a + b
  x := tmp
  y := tmp + 1
} otherwise {
  x := 0
  y := 0
}
```

备注： SpinalHDL 会检查范围内定义的信号是否仅在该范围内使用/赋值。

6.2.5 Mux

如果您只需要一个带有 Bool 选择信号的 Mux ， 则有两种等效的语法：

语法	返 回 类 型	描述
Mux(cond, whenTrue, whenFalse)	T	当 cond 为 True 时返回 whenTrue ， 否则返回 whenFalse
cond ? whenTrue | whenFalse	T	当 cond 为 True 时返回 whenTrue ， 否则返回 whenFalse

```
val cond = Bool()
val whenTrue, whenFalse = UInt(8 bits)
val muxOutput  = Mux(cond, whenTrue, whenFalse)
val muxOutput2 = cond ? whenTrue | whenFalse
```

6.2.6 按位选择

按位选择看起来像 VHDL when 语法。

示例

```
val bitwiseSelect = UInt(2 bits)
val bitwiseResult = bitwiseSelect.mux(
  0 -> (io.src0 & io.src1),
  1 -> (io.src0 | io.src1),
  2 -> (io.src0 ^ io.src1),
  default -> (io.src0)
)
```

mux 检查所有可能的值是否都被覆盖以防止锁存器的生成。如果覆盖了所有可能的值，则不允许添加 default 语句：

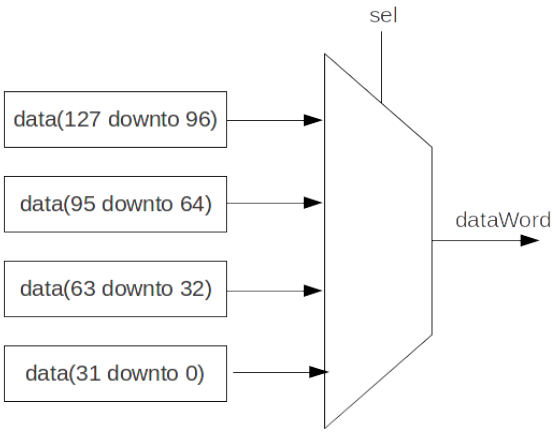
```
val bitwiseSelect = UInt(2 bits)
val bitwiseResult = bitwiseSelect.mux(
  0 -> (io.src0 & io.src1),
  1 -> (io.src0 | io.src1),
  2 -> (io.src0 ^ io.src1),
  3 -> (io.src0)
)
```

muxList(...) 和 muxListDc(...) 是另一种按位选择器，它们采用元组或映射作为输入。

muxList 可以用作 mux 的直接替代品，在生成案例的代码中提供更易于使用的接口。它具有与 mux 相同的检查行为，它要求完全覆盖并禁止在不需要时列出默认值。

如果未覆盖的值不重要，则可以使用 muxtListDc ，可以使用 muxtListDc 将它们保留为未分配状态。如果需要，这将添加默认情况。如果遇到这种默认情况，将在仿真过程中生成 *X*。muxListDc(...) 通常是一个很好的通用代码的替代方法。

下面是将 128 位的 Bits 划分为 32 位的示例：




```
val sel  = UInt(2 bits)
val data = Bits(128 bits)

// Dividing a wide Bits type into smaller chunks, using a mux:
val dataWord = sel.muxList(for (index <- 0 until 4)
                           yield (index, data(index*32+32-1 downto index*32)))

// A shorter way to do the same thing:
val dataWord = data.subdivideIn(32 bits)(sel)
```

下面是 muxListDc 的案例，从可配置位宽的向量中选择多个位：

```
case class Example(width: Int = 3) extends Component {
  // 2 bit wide for default width
  val sel = UInt(log2Up(count) bit)
  val data = Bits(width*8 bit)
  // no need to cover missing case 3 for default width
  val dataByte = sel.muxListDc(for(i <- 0 until count) yield (i, data(index*8, 8␣
↪bit)))
}
```

6.3 规则

SpinalHDL 背后的语义很重要，这样您就可以了解幕后真正发生的事情以及如何控制它。

这些语义由多个规则定义：

- 信号和寄存器彼此同时运行（并行行为，如 VHDL 和 Verilog）
- 对组合信号的赋值就像表达一条始终为真的规则
- 对寄存器的赋值就像表达一条应用于其时钟域的每个周期的规则
- 对于每个信号，最后一个赋值生效
- 每个信号和寄存器都可以作为对象在硬件实例细化期间进行操作，用 OOP 的思想

6.3.1 并发

您对每个组合或时序信号的赋值顺序不会对行为产生影响。

例如，以下两段代码是等效的：

```
val a, b, c = UInt(8 bits) // Define 3 combinational signals
c := a + b  // c will be set to 7
b := 2      // b will be set to 2
a := b + 3  // a will be set to 5
```

这相当于：

```
val a, b, c = UInt(8 bits) // Define 3 combinational signals
b := 2      // b will be set to 2
a := b + 3  // a will be set to 5
c := a + b  // c will be set to 7
```

更一般地说，当您使用 := 赋值运算符时，就像为左侧信号/寄存器指定一个附加的新规则。

6.3.2 最后有效赋值生效

如果通过使用 SpinalHDL := 运算符对组合信号或寄存器进行多次分配，则可能执行的最后一次赋值生效（因此可以将值设置为该状态的结果）。

It could be said that top to bottom evaluation occurs based on the state that exists at that time. If your upstream signal inputs are driven from registers and so have synchronous behavior, then it could be said that at each clock cycle the assignments are re-evaluated based on the new state at the time.

在硬件中，赋值语句可能无法在本时钟周期执行的一些原因，可能是由于它被包装在 `when(cond)` 子句中。

Another reason maybe that the SpinalHDL code never made it through elaboration because the feature was parameterized and disabled during HDL code-generation, see `paramIsFalse` use below.

举个例子：

```
// Every clock cycle  evaluation starts here
val paramIsFalse = false
val x, y = Bool()           // Define two combinational signals
val result = UInt(8 bits)   // Define a combinational signal

result := 1
when(x) {
  result := 2
  when(y) {
    result := 3
  }
}
if(paramIsFalse) {          // This assignment should win as it is last, but it␣
↪was never elaborated
  result := 4               //  into hardware due to the use of if() and it␣
↪evaluating to false at the time
}                           //  of elaboration.  The three := assignments above␣
↪are elaborated into hardware.
```

这将产生以下真值表：

x	y	=>	结果
False	False		1
False	True		1
True	False		2
True	True		3

6.3.3 信号和寄存器与 Scala 语言的协作（OOP 引用 + 函数）

在 SpinalHDL 中，每个硬件元素都由一个类实例建模。这意味着您可以通过使用实例的引用来操作实例，例如将它们作为参数传递给函数。

作为示例，以下代码实现了一个寄存器，当 inc 为 True 时递增，当 clear 为 True 时清零（clear 优先于 inc）：

```
val inc, clear = Bool()           // Define two combinational signals/wires
val counter = Reg(UInt(8 bits))   // Define an 8 bit register

when(inc) {
  counter := counter + 1
}
when(clear) {
  counter := 0     // If inc and clear are True, then this  assignment wins
}                  //  (last value assignment wins rule)
```

您可以通过将前面的示例与赋值给“counter“的函数混合来实现完全相同的功能：

```
val inc, clear = Bool()
val counter = Reg(UInt(8 bits))

def setCounter(value : UInt): Unit = {
  counter := value
}

when(inc) {
  setCounter(counter + 1)  // Set counter with counter + 1
}
when(clear) {
  counter := 0
}
```

您还可以将条件检查集成到函数内：

```
val inc, clear = Bool()
val counter = Reg(UInt(8 bits))

def setCounterWhen(cond : Bool,value : UInt): Unit = {
  when(cond) {
    counter := value
  }
}

setCounterWhen(cond = inc,   value = counter + 1)
setCounterWhen(cond = clear, value = 0)
```

并指定函数应实现的赋值：

```
val inc, clear = Bool()
val counter = Reg(UInt(8 bits))
```

(续下页)

(接上页)

```
def setSomethingWhen(something : UInt, cond : Bool, value : UInt): Unit = {
  when(cond) {
    something := value
  }
}

setSomethingWhen(something = counter, cond = inc,   value = counter + 1)
setSomethingWhen(something = counter, cond = clear, value = 0)
```

前面的所有示例在生成的 RTL 中，从 SpinalHDL 编译器的角度来看都是严格等效的。这是因为 SpinalHDL 只关心 Scala 运行时实例化的对象，它不关心 Scala 语法本身。

换句话说，从生成的 RTL 生成/SpinalHDL 的角度来看，当您调用 Scala 中生成硬件的函数时，就像该函数被内联了一样。Scala 循环也是如此，因为它们将以展开的形式出现在生成的 RTL 中。

CHAPTER 7

时序逻辑

7.1 寄存器

在 SpinalHDL 中创建寄存器与在 VHDL 或 Verilog 中创建寄存器有很大不同。

在 Spinal 中，没有 process/always 块。寄存器在声明时明确定义。这种与传统的事件驱动 HDL 的区别具有很大的影响：

- 您可以在同一范围内赋值寄存器和连线，这意味着代码不需要在 process/always 块之间拆分
- 它使事情变得更加灵活（参见*Functions*）

时钟和复位是分开处理的，有关详细信息，请参阅 时钟域 <*clock_domain*> 章节。

7.1.1 实例化

实例化寄存器有 4 种方法：

语法	描述
`Reg(type : Data)`	创建给定类型的寄存器
`RegInit(resetValue : Data)`	当发生复位时，为寄存器加载给定的 `resetValue`
`RegNext(nextValue : Data)`	创建寄存器，且每个周期对给定的 `nextValue` 进行采样
`RegNextWhen(nextValue : Data, cond : Bool)`	创建寄存器，当条件发生时对 `nextValue` 进行采样

这是声明一些寄存器的示例：

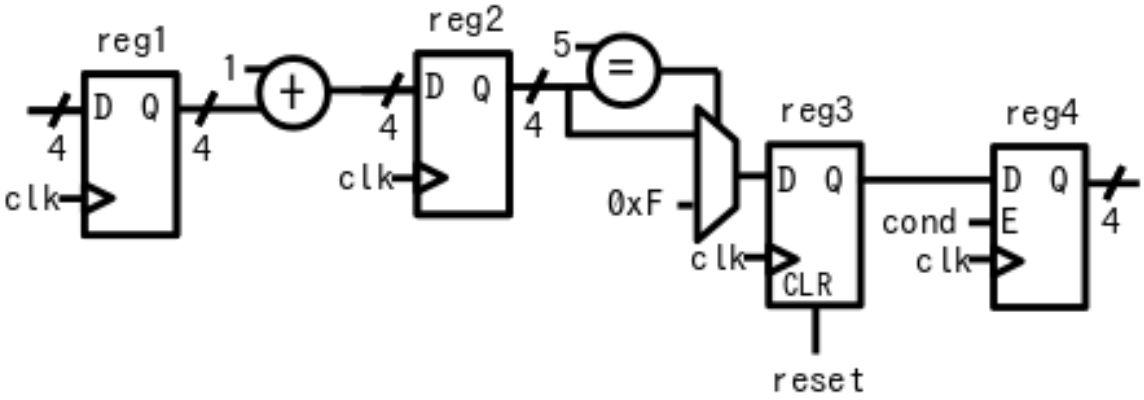
```
// UInt register of 4 bits
val reg1 = Reg(UInt(4 bits))

// Register that updates itself every cycle with a sample of reg1 incremented by 1
val reg2 = RegNext(reg1 + 1)

// UInt register of 4 bits initialized with 0 when the reset occurs
val reg3 = RegInit(U"0000")
reg3 := reg2
```

(续下页)

(接上页)

```
when(reg2 === 5) {
  reg3 := 0xF
}

// Register that samples reg3 when cond is True
val reg4 = RegNextWhen(reg3, cond)
```

上面的代码将推断出以下逻辑：



备注： 上面 reg3 示例显示了如何为 RegInit 创建寄存器赋值。也可以使用相同的语法赋值其他寄存器类型（“Reg”、“RegNext”、“RegNextWhen”）。就像组合赋值一样，规则是 “最后一个赋值生效”，但如果没有完成赋值，寄存器将保留其值。如果在设计中声明 Reg 并且没有适当地赋值和使用，EDA 流程中的工具会在它认为该寄存器不必要时裁剪寄存器（从设计中删除）。

另外，RegNext 是一个基于 Reg 语法构建的抽象。下面两个代码序列严格等效：

```
// Standard way
val something = Bool()
val value = Reg(Bool())
value := something

// Short way
val something = Bool()
val value = RegNext(something)
```

可以通过其他方式同时拥有多个选项，因此可在上述基本理解的基础上构建稍微更高级的组合：

```
// UInt register of 6 bits (initialized with 42 when the reset occurs)
val reg1 = Reg(UInt(6 bits)) init(42)

// Register that samples reg1 each cycle (initialized with 0 when the reset occurs)
// using Scala named parameter argument format
val reg2 = RegNext(reg1, init=0)

// Register that has multiple features combined

// My register enable signal
val reg3Enable = Bool()
// UInt register of 6 bits (inferred from reg1 type)
//   assignment preconfigured to update from reg1
//   only updated when reg3Enable is set
//   initialized with 99 when the reset occurs
val reg3 = RegNextWhen(reg1, reg3Enable, U(99))
// when(reg3Enable) {
```

(续下页)

(接上页)

```
//   reg3 := reg1; // this expression is implied in the constructor use case
// }

when(cond2) {       // this is a valid assignment, will take priority when executed
   reg3 := U(0)     //  (due to last assignment wins rule), assignment does not require
}                   //  reg3Enable condition, you would use `when(cond2 & reg3Enable)` for that

// UInt register of 8 bits, initialized with 99 when the reset occurs
val reg4 = Reg(UInt(8 bits), U(99))
// My register enable signal
val reg4Enable = Bool()
// no implied assignments exist, you must use enable explicitly as necessary
when(reg4Enable) {
   reg4 := newValue
}
```

7.1.2 复位值

除了直接创建具有复位值的寄存器的 `RegInit(value : Data)` 语法之外，您还可以通过在寄存器上调用 `init(value : Data)` 函数来设置复位值。

```
// UInt register of 4 bits initialized with 0 when the reset occurs
val reg1 = Reg(UInt(4 bits)) init(0)
```

如果您有一个包含线束（Bundle）的寄存器，则可以对线束的每个元素使用 `init` 函数。

```
case class ValidRGB() extends Bundle {
  val valid   = Bool()
  val r, g, b = UInt(8 bits)
}

val reg = Reg(ValidRGB())
reg.valid init(False)  // Only the valid if that register bundle will have a reset value.
```

7.1.3 用于仿真目的的初始化值

对于在 RTL 中不需要复位值，但需要仿真初始化值（以避免未知状态 X 传播）的寄存器，您可以通过调用 `randBoot()` 函数来请求随机初始化值。

```
// UInt register of 4 bits initialized with a random value
val reg1 = Reg(UInt(4 bits)) randBoot()
```

7.1.4 寄存器组

至于连线，可以使用 `Vec` 定义寄存器组。

```
val vecReg1 = Vec(Reg(UInt(8 bits)), 4)
val vecReg2 = Vec.fill(8)(Reg(Bool()))
```

初始化可以像往常一样使用 `init` 方法完成，它可以与寄存器上的 `foreach` 迭代相结合。

```
val vecReg1 = Vec(Reg(UInt(8 bits)) init(0), 4)
val vecReg2 = Vec.fill(8)(Reg(Bool()))
vecReg2.foreach(_ init(False))
```

如果由于初始化值未知而必须推迟初始化，请使用如下例所示的函数。

```
case class ShiftRegister[T <: Data](dataType: HardType[T], depth: Int, initFunc: T␣
↪=> Unit) extends Component {
   val io = new Bundle {
      val input  = in (dataType())
      val output = out(dataType())
   }

   val regs = Vec.fill(depth)(Reg(dataType()))
   regs.foreach(initFunc)

   for (i <- 1 to (depth-1)) {
         regs(i) := regs(i-1)
   }

   regs(0) := io.input
   io.output := regs(depth-1)
}

object SRConsumer {
   def initIdleFlow[T <: Data](flow: Flow[T]): Unit = {
      flow.valid init(False)
   }
}

class SRConsumer() extends Component {
   // ...
   val sr = ShiftRegister(Flow(UInt(8 bits)), 4, SRConsumer.initIdleFlow[UInt])
}
```

7.1.5 将线缆/信号转换为寄存器

有时将现有的连线转换为寄存器很有用。例如，当您使用线束（Bundle）时，如果您希望线束的某些输出成为寄存器，您可能更愿意编写 `io.myBundle.PORT := newValue` 而不用 `val PORT = Reg(...)` 并将其输出连接到带有 `io.myBundle.PORT := PORT` 的端口。为此，您只需在要实例化为寄存器的端口上使用 `.setAsReg()`：

```
val io = new Bundle {
   val apb = master(Apb3(apb3Config))
}

io.apb.PADDR.setAsReg()
io.apb.PWRITE.setAsReg() init(False)

when(someCondition) {
   io.apb.PWRITE := True
}
```

请注意，在上面的代码中，您还可以指定初始化值。

备注： 该寄存器是在线路/信号的时钟域中创建的，并且不依赖于使用 `.setAsReg()` 的位置。

在上面的示例中，线路在 `io` 线束中定义，与组件位于同一时钟域中。即使 `io.apb.PADDR.setAsReg()` 这条代码写在具有不同时钟域的 `ClockingArea` 中，寄存器也将使用组件的时钟域，而不是

ClockingArea 的时钟域。

7.2 RAM/ROM 存储器

要在 SpinalHDL 中创建内存，应使用 Mem 类。它允许您定义内存并向其添加读写端口。

下表显示了如何实例化存储器：

语法	描述
Mem(type : Data, size : Int)	创建随机访问存储器
Mem(type : Data, initialContent : Array[Data])	创建一个 ROM。如果您的目标是 FPGA，因为存储器可以推断为块 RAM，您仍然可以在其上创建写入端口。

备注： 如果你想定义一个 ROM，initialContent 数组的元素应该只是字面量（无法做运算，无法改变位宽）。这里有一个例子*here*。

备注： 要给 RAM 初始值，您还可以使用 init 函数。

备注： 掩码位宽是可以灵活设定的，您可以根据掩码的宽度将存储器分成位宽相同的多个片段。例如，如果您有一个 32 位内存字，并提供一个 4 位掩码，那么它将是一个字节掩码。如果您提供的掩码位数与存储器一个字的位数相同，那么它将是一个位掩码。

备注： 在仿真时可以对 Mem 进行操作，请参阅仿真中加载和存储存储器 部分。

下表显示了如何在存储器上添加访问端口：

语法	描述	返回类型
mem(address) := data	同步写入	
mem(address)	异步读取	T
mem.write(address data [enable] [mask])	使用可选掩码进行同步写入。 如果未指定使能（enable）条件，则会自动从调用此函数的条件范围（如 when 语句等）中推断出条件。	
mem.readAsync(address [readUnderWrite])	异步读取，具有可选的写入时读取（read-under-write）策略	T
mem.readSync(address [enable] [readUnderWrite] [clockCrossing])	同步读取，具有可选的使能信号、写入间读取策略、跨时钟域（`clockCrossing`）模式。	T
mem.readWriteSync(address data enable write [mask] [readUnderWrite]	推断读/写端口。 当 `enable && write` 满足时写入 `data`。 返回读取的数据，当 `enable` 为 true 时读取	T

备注： 如果由于某种原因您需要一个未在 Spinal 中实现的特定存储器端口，您始终可以通过为其指定 BlackBox 来抽象您的存储器。

重要： SpinalHDL 中的存储器端口不是推断的，而是明确定义的。您不应使用 VHDL/Verilog 等编码模板来帮助综合工具推断存储器。

下面是一个推断简单双端口 RAM（32 位 * 256）的示例：

```
val mem = Mem(Bits(32 bits), wordCount = 256)
mem.write(
  enable  = io.writeValid,
  address = io.writeAddress,
  data    = io.writeData
)

io.readData := mem.readSync(
  enable  = io.readValid,
  address = io.readAddress
)
```

7.2.1 同步使能注意事项

当使能信号用于由 *when* 等条件块保护的块中时，只会生成用使能信号作为访问条件的电路，也就是说 *when* 条件将被忽略。

```
val rom = Mem(Bits(10 bits), 32)
when(cond) {
  io.rdata := rom.readSync(io.addr, io.rdEna)
}
```

上面的例子中条件 *cond* 就不详细说明了。最好直接在使能信号中包含条件 *cond*，如下所示。

```
io.rdata := rom.readSync(io.addr, io.rdEna & cond)
```

7.2.2 写入时读取策略

此策略指定在同一周期内对同一地址发生写入时，读取的值将受到怎样的影响。

种类	描述
`dontCare`	发生这种情况时不用关心读取的值
`readFirst`	读取操作将得到写入之前的值
`writeFirst`	读取操作将得到由写入提供的值

重要： 生成的 VHDL/Verilog 始终处于 `readFirst` 模式，该模式与 `dontCare` 兼容，但与 `writeFirst` 不兼容。要生成包含此类功能的设计，您需要使能自动存储器黑盒。

7.2.3 混合位宽存储器

您可以使用以下函数指定访问存储器的端口，其位宽为二的幂次：

语法	描述
mem.writeMixedWidth(address data [readUnderWrite])	类似于 `mem.write`
mem.readAsyncMixedWidth(address data [readUnderWrite])	类似于 `mem.readAsync`, 会立即返回值，它驱动以 `data` 参数的形式传入的信号/对象
mem.readSyncMixedWidth(address data [enable] [readUnderWrite] [clockCrossing])	与 `mem.readSync` 类似，但它不是返回读取值，而是驱动 `data` 参数给出的信号/对象
mem.readWriteSyncMixedWidth(address data enable write [mask] [readUnderWrite] [clockCrossing])	相当于 `mem.readWriteSync`

重要： 至于写入时读取策略，要使用此功能，您需要启用自动内存黑盒，因为没有通用的 VHDL/Verilog 语言模板来推断混合位宽存储器。

7.2.4 自动黑盒化

由于使用常规 VHDL/Verilog 不可能推断所有 ram 类型，因此 SpinalHDL 集成了可选的自动黑盒系统。该系统会查看 RTL 网表中存在的所有存储器，并用一个黑盒替换它们。然后生成的代码将依赖第三方 IP 来提供内存功能，例如写入时读取策略和混合位宽端口。

这是一个如何缺省使能黑盒化存储器的例子：

```
def main(args: Array[String]) {
  SpinalConfig()
    .addStandardMemBlackboxing(blackboxAll)
    .generateVhdl(new TopLevel)
}
```

如果标准黑盒工具不足以满足您的设计需求，请毫不犹豫地创建 Github 工单。还有一种方法，可以创建您自己的黑盒工具。

黑盒策略

您可以使用多种策略来选择要黑盒的内存以及黑盒不可行时要执行的操作：

种类	描述
blackboxAll	黑盒化所有存储器。 对不可黑盒存储器抛出错误
blackboxAllWhatsYouCan	黑盒所有可黑盒的存储器
blackboxRequestedAndUninferable	用户指定的黑盒存储器和已知不可推断的存储器（混合位宽，…）。 对不可黑盒存储器抛出错误
blackboxOnlyIfRequested	用户指定的黑盒存储器 对不可黑盒存储器抛出错误

要显式地将存储器设置为黑盒，您可以使用其 generateAsBlackBox 函数。

```
val mem = Mem(Rgb(rgbConfig), 1 << 16)
mem.generateAsBlackBox()
```

你可以通过继承 MemBlackboxingPolicy 类定义你自己的黑盒化策略。

标准存储器黑盒

下面显示的是 SpinalHDL 中使用的标准黑盒的 VHDL 定义：

```
-- Simple asynchronous dual port (1 write port, 1 read port)
component Ram_1w_1ra is
  generic(
    wordCount : integer;
    wordWidth : integer;
    technology : string;
    readUnderWrite : string;
    wrAddressWidth : integer;
```

(续下页)

(接上页)

```
    wrDataWidth : integer;
    wrMaskWidth : integer;
    wrMaskEnable : boolean;
    rdAddressWidth : integer;
    rdDataWidth : integer
  );
  port(
    clk : in std_logic;
    wr_en : in std_logic;
    wr_mask : in std_logic_vector;
    wr_addr : in unsigned;
    wr_data : in std_logic_vector;
    rd_addr : in unsigned;
    rd_data : out std_logic_vector
  );
end component;

-- Simple synchronous dual port (1 write port, 1 read port)
component Ram_1w_1rs is
  generic(
    wordCount : integer;
    wordWidth : integer;
    clockCrossing : boolean;
    technology : string;
    readUnderWrite : string;
    wrAddressWidth : integer;
    wrDataWidth : integer;
    wrMaskWidth : integer;
    wrMaskEnable : boolean;
    rdAddressWidth : integer;
    rdDataWidth : integer;
    rdEnEnable : boolean
  );
  port(
    wr_clk : in std_logic;
    wr_en : in std_logic;
    wr_mask : in std_logic_vector;
    wr_addr : in unsigned;
    wr_data : in std_logic_vector;
    rd_clk : in std_logic;
    rd_en : in std_logic;
    rd_addr : in unsigned;
    rd_data : out std_logic_vector
  );
end component;

-- Single port (1 readWrite port)
component Ram_1wrs is
  generic(
    wordCount : integer;
    wordWidth : integer;
    readUnderWrite : string;
    technology : string
  );
  port(
    clk : in std_logic;
    en : in std_logic;
    wr : in std_logic;
    addr : in unsigned;
    wrData : in std_logic_vector;
    rdData : out std_logic_vector
```

(续下页)

(接上页)

```
  );
end component;

--True dual port (2 readWrite port)
component Ram_2wrs is
  generic(
    wordCount : integer;
    wordWidth : integer;
    clockCrossing : boolean;
    technology : string;
    portA_readUnderWrite : string;
    portA_addressWidth : integer;
    portA_dataWidth : integer;
    portA_maskWidth : integer;
    portA_maskEnable : boolean;
    portB_readUnderWrite : string;
    portB_addressWidth : integer;
    portB_dataWidth : integer;
    portB_maskWidth : integer;
    portB_maskEnable : boolean
  );
  port(
    portA_clk : in std_logic;
    portA_en : in std_logic;
    portA_wr : in std_logic;
    portA_mask : in std_logic_vector;
    portA_addr : in unsigned;
    portA_wrData : in std_logic_vector;
    portA_rdData : out std_logic_vector;
    portB_clk : in std_logic;
    portB_en : in std_logic;
    portB_wr : in std_logic;
    portB_mask : in std_logic_vector;
    portB_addr : in unsigned;
    portB_wrData : in std_logic_vector;
    portB_rdData : out std_logic_vector
  );
end component;
```

正如你所看到的，黑盒有一个技术参数。要设置它，您可以在相应的内存上使用 `setTechnology` 函数。目前有 4 种可能的技术：

- `auto`
- `ramBlock`
- `distributedLut`
- `registerFile`

如果已为您的设备供应商配置了 `SpinalConfig#setDevice(Device)` ，则黑盒化可以插入 HDL 属性。

生成的 HDL 属性可能如下所示：

```
(* ram_style = "distributed" *)
(* ramsyle = "no_rw_check" *)
```

SpinalHDL 尝试支持知名供应商和设备提供的许多常见存储器类型，但是这是一个不断变化的领域，并且该领域的项目要求可能非常具体。

如果这对您的设计流程很重要，请检查输出 HDL 是否有预期的属性/代码，同时查阅供应商的平台文档。

HDL 属性也可以使用 *addAttribute()* *addAttribute* 机制手动添加。

CHAPTER 8

设计错误

SpinalHDL 编译器将对您的设计执行许多检查，以确保生成的 VHDL/Verilog 对于仿真和综合来说是安全的。总体来说应该不可能生成有错误的 VHDL/Verilog 设计。以下是 SpinalHDL 检查的非详尽列表：

- 赋值覆盖 (Assignment overlapping)
- 跨时钟域 (Clock crossing)
- 层次违例 (Hierarchy violation)
- 组合逻辑环 (Combinatorial loops)
- 锁存器 (Latches)
- 无驱动信号 (Undriven signals)
- 位宽不匹配 (Width mismatch)
- 无法访问的 switch 语句 (Unreachable switch statements)

在每个 SpinalHDL 错误报告中，您都会找到堆栈跟踪，这对于准确找出设计错误在哪里非常有用。乍一看，这些设计检查可能有点矫枉过正，但一旦您开始远离传统的硬件描述方式，它们就变得非常宝贵。

8.1 赋值覆盖 (Assignment overlap)

8.1.1 简介

SpinalHDL 将检查，没有任何信号赋值会完全擦除前面的信号赋值。

8.1.2 示例

下面的代码

```
class TopLevel extends Component {
  val a = UInt(8 bits)
  a := 42
  a := 66 // Erase the a := 42 assignment
}
```

会出现以下错误：

```
ASSIGNMENT OVERLAP completely the previous one of (toplevel/a :  UInt[8 bits])
  ***
  Source file location of the a := 66 assignment via the stack trace
  ***
```

一个可能的修复方法是：

```
class TopLevel extends Component {
  val a = UInt(8 bits)
  a := 42
  when(something) {
    a := 66
  }
}
```

但是，如果您确实想要覆盖先前的赋值（因为有时覆盖是有意义的），您可以执行以下操作：

```
class TopLevel extends Component {
  val a = UInt(8 bits)
  a := 42
  a.allowOverride
  a := 66
}
```

8.2 跨时钟域违例 (Clock crossing violation)

8.2.1 简介

SpinalHDL 将检查您的设计中的每个寄存器是否仅（通过组合逻辑路径）与相同或同步时钟域的寄存器连接。

8.2.2 示例

下面的代码：

```
class TopLevel extends Component {
  val clkA = ClockDomain.external("clkA")
  val clkB = ClockDomain.external("clkB")

  val regA = clkA(Reg(UInt(8 bits)))    // PlayDev.scala:834
  val regB = clkB(Reg(UInt(8 bits)))    // PlayDev.scala:835

  val tmp = regA + regA                 // PlayDev.scala:838
  regB := tmp
}
```

会报错：

```
CLOCK CROSSING VIOLATION from (toplevel/regA :  UInt[8 bits]) to (toplevel/regB : ␣
↪UInt[8 bits]).
- Register declaration at
  ***
  Source file location of the toplevel/regA definition via the stack trace
  ***
- through
      >>> (toplevel/regA :  UInt[8 bits]) at ***(PlayDev.scala:834) >>>
      >>> (toplevel/tmp :  UInt[8 bits]) at ***(PlayDev.scala:838) >>>
      >>> (toplevel/regB :  UInt[8 bits]) at ***(PlayDev.scala:835) >>>
```

有多种可能的修复方法，如下所示：

- *crossClockDomain 标签*
- *setSynchronousWith 方法*
- *BufferCC 类型*

crossClockDomain 标签

标签 `crossClockDomain` 可用于向 SpinalHDL 编译器传达“没关系，不要对这个特定的跨时钟域操作感到恐慌”的信息。

```
class TopLevel extends Component {
  val clkA = ClockDomain.external("clkA")
  val clkB = ClockDomain.external("clkB")

  val regA = clkA(Reg(UInt(8 bits)))
  val regB = clkB(Reg(UInt(8 bits))).addTag(crossClockDomain)


  val tmp = regA + regA
  regB := tmp
}
```

setSynchronousWith

您还可以使用 `ClockDomain` 对象的 `setSynchronousWith` 方法指定两个时钟域同步。

```
class TopLevel extends Component {
  val clkA = ClockDomain.external("clkA")
  val clkB = ClockDomain.external("clkB")
  clkB.setSynchronousWith(clkA)

  val regA = clkA(Reg(UInt(8 bits)))
  val regB = clkB(Reg(UInt(8 bits)))


  val tmp = regA + regA
  regB := tmp
}
```

BufferCC

当交换单比特信号（如 Bool 类型）或格雷码时，您可以使用 BufferCC 安全地跨不同的 ClockDomain 时钟域。

警告： 不要将 BufferCC 用于多比特信号，因为如果时钟异步，那么接收端会存在读取损坏的风险。有关更多详细信息，请参阅时钟域 页面。

```
class AsyncFifo extends Component {
   val popToPushGray = Bits(ptrWidth bits)
   val pushToPopGray = Bits(ptrWidth bits)

   val pushCC = new ClockingArea(pushClock) {
     val pushPtr     = Counter(depth << 1)
     val pushPtrGray = RegNext(toGray(pushPtr.valueNext)) init(0)
     val popPtrGray  = BufferCC(popToPushGray, B(0, ptrWidth bits))
     val full        = isFull(pushPtrGray, popPtrGray)
     ...
   }

   val popCC = new ClockingArea(popClock) {
     val popPtr      = Counter(depth << 1)
     val popPtrGray  = RegNext(toGray(popPtr.valueNext)) init(0)
     val pushPtrGray = BufferCC(pushToPopGray, B(0, ptrWidth bits))
     val empty       = isEmpty(popPtrGray, pushPtrGray)
     ...
   }
}
```

8.3 组合逻辑环 (Combinatorial loop)

8.3.1 简介

SpinalHDL 将检查设计中是否存在组合逻辑环。

8.3.2 示例

下面的代码：

```
class TopLevel extends Component {
  val a = UInt(8 bits) // PlayDev.scala line 831
  val b = UInt(8 bits) // PlayDev.scala line 832
  val c = UInt(8 bits)
  val d = UInt(8 bits)

  a := b
  b := c | d
  d := a
  c := 0
}
```

会出现：

```
COMBINATORIAL LOOP :
  Partial chain :
```

(续下页)

(接上页)

```
    >>> (toplevel/a :  UInt[8 bits]) at ***(PlayDev.scala:831) >>>
    >>> (toplevel/d :  UInt[8 bits]) at ***(PlayDev.scala:834) >>>
    >>> (toplevel/b :  UInt[8 bits]) at ***(PlayDev.scala:832) >>>
    >>> (toplevel/a :  UInt[8 bits]) at ***(PlayDev.scala:831) >>>

  Full chain :
    (toplevel/a :  UInt[8 bits])
    (toplevel/d :  UInt[8 bits])
    (UInt | UInt)[8 bits]
    (toplevel/b :  UInt[8 bits])
    (toplevel/a :  UInt[8 bits])
```

一个可能的修复方式是：

```
class TopLevel extends Component {
  val a = UInt(8 bits) // PlayDev.scala line 831
  val b = UInt(8 bits) // PlayDev.scala line 832
  val c = UInt(8 bits)
  val d = UInt(8 bits)

  a := b
  b := c | d
  d := 42
  c := 0
}
```

8.3.3 误报

SpinalHDL 检测组合逻辑环的算法可能是悲观的，并且可能会给出误报。如果出现误报，您可以手动禁用对某一信号的逻辑环的检查，如下所示：

```
class TopLevel extends Component {
  val a = UInt(8 bits)
  a := 0
  a(1) := a(0) // False positive because of this line
}
```

可以通过以下方式修复：

```
class TopLevel extends Component {
  val a = UInt(8 bits).noCombLoopCheck
  a := 0
  a(1) := a(0)
}
```

还应该指出，诸如 `(a(1) := a(0))` 之类的赋值可能会使一些工具如 Verilator 无法适配。在这种情况下，使用 `Vec(Bool(), 8)` 可能更好。

8.4 层次违例 (Hierarchy violation)

8.4.1 简介

SpinalHDL 将会检查当前层次设计的信号不会访问到该组件的外部区域。

以下信号可以在组件内部读取：

- 当前组件中定义的所有无方向信号
- 当前组件的所有 in/out/inout 信号
- 子组件的所有 in/out/inout 信号

同时，以下信号可以在组件内部赋值：

- 当前组件中定义的所有无方向信号
- 当前组件的所有 out/inout 信号
- 子组件的所有 in/inout 信号

如果出现 `HIERARCHY VIOLATION` 错误，则意味着违反了上述规则之一。

8.4.2 示例

下面的代码：

```
class TopLevel extends Component {
  val io = new Bundle {
    // This is an 'in' signal of the current component 'Toplevel'
    val a = in UInt(8 bits)
  }
  val tmp = U"x42"
  io.a := tmp  // ERROR: attempting to assign to an input of current component
}
```

会报错：

```
HIERARCHY VIOLATION : (toplevel/io_a : in UInt[8 bits]) is driven by (toplevel/tmp
↪:  UInt[8 bits]), but isn't accessible in the toplevel component.
  ***
  Source file location of the `io.a := tmp` via the stack trace
  ***
```

一个可能的修复方法是：

```
class TopLevel extends Component {
  val io = new Bundle {
    val a = out UInt(8 bits) // changed from in to out
  }
  val tmp = U"x42"
  io.a := tmp  // now we are assigning to an output
}
```

8.5 IO 线束

8.5.1 简介

SpinalHDL 将检查每个 io 线束是否仅包含输入/输出/双向信号。其他类型的信号称为无方向信号。

8.5.2 示例

下面的代码：

```
class TopLevel extends Component {
  val io = new Bundle {
    val a = UInt(8 bits)
  }
}
```

会报错：

```
IO BUNDLE ERROR : A direction less (toplevel/io_a :  UInt[8 bits]) signal was␣
↪defined into toplevel component's io bundle
  ***
  Source file location of the toplevel/io_a definition via the stack trace
  ***
```

一个可能的修复方法是：

```
class TopLevel extends Component {
  val io = new Bundle {
    val a = in UInt(8 bits)  // provide 'in' direction declaration
  }
}
```

但是，如果出于元硬件描述的原因，您确实希望 io.a 没有方向，您可以这样做：

```
class TopLevel extends Component {
  val io = new Bundle {
    val a = UInt(8 bits)
  }
  a.allowDirectionLessIo
}
```

8.6 锁存器检测 (Latch detected)

8.6.1 简介

SpinalHDL 将检查在综合期间没有组合信号会引入锁存器。换句话说，这是检查没有组合信号被部分赋值。

8.6.2 示例

下面的代码：

```
class TopLevel extends Component {
  val cond = in(Bool())
  val a = UInt(8 bits)

  when(cond) {
    a := 42
  }
}
```

会报错：

```
LATCH DETECTED from the combinatorial signal (toplevel/a :  UInt[8 bits]), defined
↪at
  ***
  Source file location of the toplevel/io_a definition via the stack trace
  ***
```

一个可能的修复方法是：

```
class TopLevel extends Component {
  val cond = in(Bool())
  val a = UInt(8 bits)

  a := 0
  when(cond) {
    a := 42
  }
}
```

8.6.3 因多路复用器产生的错误

检测到锁存器的另一个原因通常是不详尽、缺少默认值的 mux/muxList 语句：

```
val u1 = UInt(1 bit)
u1.mux(
  0 -> False,
  // case for 1 is missing
)
```

可以通过添加缺失的条件（或默认条件）来修复：

```
val u1 = UInt(1 bit)
u1.mux(
  0 -> False,
  default -> True
)
```

例如对于通用位宽代码，使用 muxListDc 通常是更好的解决方案，因为对于不需要默认值的情况，这不会生成错误：

```
val u1 = UInt(1 bit)
// automatically adds default if needed
u1.muxListDc(Seq(0 -> True))
```

8.7 无驱动检测 (No driver on)

8.7.1 简介

SpinalHDL 将检查所有对设计有影响的组合信号是否被赋值。

8.7.2 示例

下面的代码：

```
class TopLevel extends Component {
  val result = out(UInt(8 bits))
  val a = UInt(8 bits)
  result := a
}
```

会报错：

```
NO DRIVER ON (toplevel/a :  UInt[8 bits]), defined at
  ***
  Source file location of the toplevel/a definition via the stack trace
  ***
```

一个可能的修复方法是：

```
class TopLevel extends Component {
  val result = out(UInt(8 bits))
  val a = UInt(8 bits)
  a := 42
  result := a
}
```

8.8 空指针异常 (NullPointerException)

8.8.1 简介

`NullPointerException` 是 Scala 运行时报告的错误，当变量在初始化之前被访问时可能会发生这种错误。

8.8.2 示例

下面的代码：

```
class TopLevel extends Component {
  a := 42
  val a = UInt(8 bits)
}
```

会报错：

```
Exception in thread "main" java.lang.NullPointerException
  ***
  Source file location of the a := 42 assignment via the stack trace
  ***
```

一个可能的修复方法是：

```
class TopLevel extends Component {
  val a = UInt(8 bits)
  a := 42
}
```

问题说明

SpinalHDL 不是一种语言，它是一个 Scala 库，这意味着它遵循与 Scala 通用编程语言相同的规则。

当运行上面的 SpinalHDL 硬件描述生成相应的 VHDL/Verilog RTL 时，SpinalHDL 硬件描述将作为 Scala 程序执行，并且 a 将是一个空引用，直到程序执行 `val a = UInt(8 bits)`，因此试图在此之前赋值给它将导致 `NullPointerException` 。

8.9 超出范围的常数 (Out of Range Constant)

8.9.1 简介

SpinalHDL 会检查，当一个值和一个常量对比时，该值是否具有更宽的位数。

8.9.2 示例

例如下面的代码：

```
val value = in UInt(2 bits)
val result = out(value < 42)
```

会导致如下错误：

```
OUT OF RANGE CONSTANT. Operator UInt < UInt
- Left  operand : (toplevel/value : in UInt[2 bits])
- Right operand : (U"101010" 6 bits)
 is checking a value against a out of range constant
```

8.9.3 特殊情况

在某些情况下，由于设计参数化，将值与更大的常量进行比较并获得静态已知的 `True/False` 结果是有意义的。

您可以选择将与超出范围的常量进行比较的一个实例列入专门的白名单。

```
val value = in UInt(2 bits)
val result = out((value < 42).allowOutOfRangeLiterals)
```

或者，您可以允许整个设计对超出范围的常量进行比较。

```
SpinalConfig(allowOutOfRangeLiterals = true)
```

8.10 定义为组件输入的寄存器 (Register defined as component input)

8.10.1 简介

在 SpinalHDL 中，用户不被允许定义一个将寄存器作为输入的组件。原因是为了防止用户试图用寄存器信号作为子组件的输入驱动时出现意外。如果确实需要一个寄存器输入，用户可以在 io 线束中先定义一个非寄存器输入，随后在组件内部对其添加寄存器。

8.10.2 示例

以下代码：

```
class TopLevel extends Component {
  val io = new Bundle {
    val a = in(Reg(UInt(8 bits)))
  }
}
```

会报错：

```
REGISTER DEFINED AS COMPONENT INPUT : (toplevel/io_a : in UInt[8 bits]) is defined
↪as a registered input of the toplevel component, but isn't allowed.
  ***
  Source file location of the toplevel/io_a definition via the stack trace
  ***
```

一个可能的修复方法是：

```
class TopLevel extends Component {
  val io = new Bundle {
    val a = in UInt(8 bits)
  }
}
```

如果需要一个寄存器信号 a ，可以这样做：

```
class TopLevel extends Component {
  val io = new Bundle {
    val a = in UInt(8 bits)
  }
  val a = RegNext(io.a)
}
```

8.11 作用域违例 (Scope violation)

8.11.1 简介

SpinalHDL 将会检查没有信号会在超出其定义的作用域之外被赋值使用。这个错误不容易触发，因为它需要一些特定的元硬件描述技巧。

8.11.2 示例

下面的代码：

```
class TopLevel extends Component {
  val cond = Bool()

  var tmp : UInt = null
  when(cond) {
    tmp = UInt(8 bits)
  }
  tmp := U"x42"
}
```

会报错：

```
SCOPE VIOLATION : (toplevel/tmp :  UInt[8 bits]) is assigned outside its␣
↪declaration scope at
  ***
  Source file location of the tmp := U"x42" via the stack trace
  ***
```

一个可能的修复方法是：

```
class TopLevel extends Component {
  val cond = Bool()

  var tmp : UInt = UInt(8 bits)
  when(cond) {

  }
  tmp := U"x42"
}
```

8.12 Spinal 无法克隆类 (Spinal can’t clone class)

8.12.1 简介

当 SpinalHDL 想要通过 `cloneOf` 函数创建一个新的数据类型实例，但做不到这一点时，就会出现此错误。出现这种情况的原因几乎都是因为它无法检索 `Bundle` 的构造参数。

8.12.2 例子 1

下面的代码：

```
// cloneOf(this) isn't able to retrieve the width value that was used to construct␣
↪itself
class RGB(width : Int) extends Bundle {
  val r, g, b = UInt(width bits)
}

class TopLevel extends Component {
  val tmp = Stream(new RGB(8)) // Stream requires the capability to cloneOf(new␣
↪RGB(8))
}
```

会报错：

```
*** Spinal can't clone class spinal.tester.PlayDevMessages$RGB datatype
*** You have two way to solve that :
*** In place to declare a "class Bundle(args){}", create a "case class Bundle(args)
↪{}"
*** Or override by your self the bundle clone function
  ***
  Source file location of the RGB class definition via the stack trace
  ***
```

一个可能的修复方法是：

```
case class RGB(width : Int) extends Bundle {
  val r, g, b = UInt(width bits)
}

class TopLevel extends Component {
  val tmp = Stream(RGB(8))
}
```

8.12.3 例子 2

下面的代码：

```
case class Xlen(val xlen: Int) {}

case class MemoryAddress()(implicit xlenConfig: Xlen) extends Bundle {
    val address = UInt(xlenConfig.xlen bits)
}

class DebugMemory(implicit config: Xlen) extends Component {
    val io = new Bundle {
        val inputAddress = in(MemoryAddress())
    }

    val someAddress = RegNext(io.inputAddress) // -> ERROR␣
↪*****************************
}
```

报错：

```
[error] *** Spinal can't clone class debug.MemoryAddress datatype
```

在这种情况下，一种解决方案是覆盖克隆函数以传递隐式参数。

```
case class MemoryAddress()(implicit xlenConfig: Xlen) extends Bundle {
  val address = UInt(xlenConfig.xlen bits)

  override def clone = MemoryAddress()
}
```

备注： 我们需要克隆的是硬件的单元，而不是最终在其中赋值的值。

备注： 另一种方法是使用*ScopeProperty*。

8.13 未赋值的寄存器 (Unassigned register)

8.13.1 简介

SpinalHDL 将检查所有影响设计的寄存器是否已在某处被赋值。

8.13.2 示例

下面的代码：

```
class TopLevel extends Component {
  val result = out(UInt(8 bits))
  val a = Reg(UInt(8 bits))
  result := a
}
```

会报错：

```
UNASSIGNED REGISTER (toplevel/a :  UInt[8 bits]), defined at
  ***
  Source file location of the toplevel/a definition via the stack trace
  ***
```

一个可能的修复方法是：

```
class TopLevel extends Component {
  val result = out(UInt(8 bits))
  val a = Reg(UInt(8 bits))
  a := 42
  result := a
}
```

8.13.3 只有初始化 (init) 的寄存器

在某些情况下，由于设计参数化，生成一个没有赋值而只有 `init` 语句的寄存器可能是有意义的。

```
class TopLevel extends Component {
  val result = out(UInt(8 bits))
  val a = Reg(UInt(8 bits)) init(42)

  if(something)
    a := somethingElse
  result := a
}
```

会报错：

```
UNASSIGNED REGISTER (toplevel/a :  UInt[8 bits]), defined at
  ***
  Source file location of the toplevel/a definition via the stack trace
  ***
```

要修复这个问题，如果寄存器有一个 `init` 语句但没有赋值，你可以让 SpinalHDL 将该未赋值的寄存器转换为组合逻辑：

```
class TopLevel extends Component {
  val result = out(UInt(8 bits))
```

(续下页)

(接上页)

```
  val a = Reg(UInt(8 bits)).init(42).allowUnsetRegToAvoidLatch

  if(something)
    a := somethingElse
  result := a
}
```

8.14 无法访问的 is 语句 (Unreachable is statement)

8.14.1 简介

SpinalHDL 将确保 `switch` 中的所有 `is` 语句均可访问。

8.14.2 示例

下面的代码：

```
class TopLevel extends Component {
  val sel = UInt(2 bits)
  val result = UInt(4 bits)
  switch(sel) {
    is(0){ result := 4 }
    is(1){ result := 6 }
    is(2){ result := 8 }
    is(3){ result := 9 }
    is(0){ result := 2 } // Duplicated is statement!
  }
}
```

会报错：

```
UNREACHABLE IS STATEMENT in the switch statement at
  ***
  Source file location of the is statement definition via the stack trace
  ***
```

一个可能的修复方法是：

```
class TopLevel extends Component {
  val sel = UInt(2 bits)
  val result = UInt(4 bits)
  switch(sel) {
    is(0){ result := 4 }
    is(1){ result := 6 }
    is(2){ result := 8 }
    is(3){ result := 9 }
  }
}
```

8.15 位宽不匹配 (Width mismatch)

8.15.1 简介

SpinalHDL 将检查赋值左侧和右侧的运算操作和信号具有相同的位宽。

8.15.2 赋值示例

下面的代码：

```
class TopLevel extends Component {
  val a = UInt(8 bits)
  val b = UInt(4 bits)
  b := a
}
```

会报错：

```
WIDTH MISMATCH on (toplevel/b :  UInt[4 bits]) := (toplevel/a :  UInt[8 bits]) at
  ***
  Source file location of the OR operator via the stack trace
  ***
```

一个可能的修复方法是：

```
class TopLevel extends Component {
  val a = UInt(8 bits)
  val b = UInt(4 bits)
  b := a.resized
}
```

8.15.3 运算操作示例

下面的代码：

```
class TopLevel extends Component {
  val a = UInt(8 bits)
  val b = UInt(4 bits)
  val result = a | b
}
```

会报错：

```
WIDTH MISMATCH on (UInt | UInt)[8 bits]
- Left  operand : (toplevel/a :  UInt[8 bits])
- Right operand : (toplevel/b :  UInt[4 bits])
  at
  ***
  Source file location of the OR operator via the stack trace
  ***
```

一个可能的修复方法是：

```
class TopLevel extends Component {
  val a = UInt(8 bits)
  val b = UInt(4 bits)
  val result = a | (b.resized)
}
```

CHAPTER 9

其他语言功能

该语言的核心定义了许多功能性语法：

- 类型/字面量
- 寄存器/时钟域
- 组件/逻辑区
- 随机访问/只读存储器
- When / Switch / Mux
- BlackBox（在 Spinal 内部集成 VHDL 或 Verilog IP）
- SpinalHDL 到 VHDL 的转换器

然后，通过使用这些功能，您可以定义数字硬件，并构建强大的库和抽象。这也是 SpinalHDL 相对于其他常用 HDL 的主要优势之一，因为您无需了解编译器内部原理即可扩展该语言。

一个很好的例子是*SpinalHDL lib*，它添加了许多实用程序、工具、总线和方法。

要使用下一章中介绍的功能，您需要在源代码中加入 `import spinal.core._` 。

9.1 实用工具

9.1.1 介绍

许多工具和实用程序都存在于*spinal.lib* 中，但有些工具和实用程序已经存在于 SpinalHDL Core 中。

语法	返回类型	描述
`widthOf(x : BitVector)`	Int	返回 Bits/UInt/SInt 信号的位宽
`log2Up(x : BigInt)`	Int	返回表示 x 状态所需的位数
`isPow2(x: BigInt)`	Boolean	如果 x 是 2 的幂，则返回 true
`roundUp(that : BigInt, by : BigInt)`	BigInt	返回第一个 by 乘以“that“（包含）的值
`Cat(x: Data*)`	位	连接所有参数，从 MSB 到 LSB，请参阅*Cat*
`Cat(x: Iterable[Data])`	位	连接参数, 从 LSB 到 MSB, 参见*Cat*

Cat

如上所述，Cat 有两个版本。两个版本都连接了它们包含的信号，但有细微的差别：

- Cat(x: Data*) 使用任意数量的硬件信号作为参数。它模拟了其他 HDL 且 MSB 变成了结果 Bits 最左端的参数，最右端是 LSB . 换种说法：输入按照参数顺序拼接.
- Cat(x: Iterable[Data]) 接受包含硬件信号的单个 Scala 可迭代集合（Seq / Set / List / ···）。此版本将列表的第一个元素放入 LSB，最后一个元素放入 MSB。

差异主要在于这样的约定：Bits 是从最高索引到最低索引写入的，而列表是从索引 0 开始写入到最高索引的。所有约定中，Cat 将索引 0 放置在 LSB 处。

```
val bit0, bit1, bit2 = Bool()

val first = Cat(bit2, bit1, bit0)

// is equivalent to

val signals = List(bit0, bit1, bit2)
val second = Cat(signals)
```

9.1.2 克隆硬件数据类型

您可以使用 cloneOf(x) 函数克隆给定的硬件数据类型。它将返回相同 Scala 类型和参数的新实例。

例如：

```
def plusOne(value : UInt) : UInt = {
  // Will provide new instance of a UInt with the same width as ``value``
  val temp = cloneOf(value)
  temp := value + 1
  return temp
}

// treePlusOne will become a 8 bits value
val treePlusOne = plusOne(U(3, 8 bits))
```

您可以在硬件类型页面 上获取有关如何管理硬件数据类型的更多信息。

备注： 如果你在 Bundle 上使用 cloneOf 函数，这个 Bundle 应该是一个 case class ，否则应该在内部重写 clone 函数。

```
// An example of a regular 'class' with 'override def clone()' function
class MyBundle(ppp : Int) extends Bundle {
   val a = UInt(ppp bits)
   override def clone = new MyBundle(ppp)
 }
 val x = new MyBundle(3)
 val typeDef = HardType(new MyBundle(3))
 val y = typeDef()

 cloneOf(x) // Need clone method, else it errors
 cloneOf(y) // Is ok
```

9.1.3 将数据类型作为构造函数参数传递

许多可重用硬件需要通过数据类型进行参数化。例如，如果您想定义 FIFO 或移位寄存器，则需要一个参数来指定组件所需的有效负载类型。

有两种类似的方法可以做到这一点。

老办法

老方法的一个很好的例子是 ShiftRegister 组件的定义：

```
case class ShiftRegister[T <: Data](dataType: T, depth: Int) extends Component {
  val io = new Bundle {
    val input  = in (cloneOf(dataType))
    val output = out(cloneOf(dataType))
  }
  // ...
}
```

以下是实例化该组件的方法：

```
val shiftReg = ShiftRegister(Bits(32 bits), depth = 8)
```

如您所见，原始硬件类型直接作为构造参数传递。每次你想创建这种硬件数据类型的新实例时，你需要使用 cloneOf(...) 函数。以这种方式做事并不是超级安全，因为很容易忘记使用 cloneOf。

安全的方法

安全的传递数据类型参数方法，示例如下：

```
case class ShiftRegister[T <: Data](dataType: HardType[T], depth: Int) extends␣
↪Component {
  val io = new Bundle {
    val input  = in (dataType())
    val output = out(dataType())
  }
  // ...
}
```

以下是实例化组件的方法（与之前完全相同）：

```
val shiftReg = ShiftRegister(Bits(32 bits), depth = 8)
```

请注意，上述示例中使用了一个 HardType 包装器，它包装了原始数据类型 T，这种做法比“旧方法”更容易使用。因为要创建硬件数据类型的新实例，只需调用 HardType 的 apply 函数（或者换句话说，在类型名后添加括号）。

此外，从用户的角度来看，这种机制是完全透明的，因为硬件数据类型可以隐式转换为 HardType。

9.1.4 频率和时间

SpinalHDL 有专用语法来定义频率和时间值：

```
val frequency = 100 MHz       // infers type TimeNumber
val timeoutLimit = 3 ms       // infers type HertzNumber
val period = 100 us           // infers type TimeNumber

val periodCycles = frequency * period             // infers type BigDecimal
val timeoutCycles = frequency * timeoutLimit      // infers type BigDecimal
```

对于时间定义，您可以使用以下后缀来获取 `TimeNumber`：
`fs、ps、ns、us、ms、sec、mn、hr`

对于时间定义，您可以使用以下后缀来获取 `HertzNumber`：
`Hz, KHz, MHz, GHz, THz`

`TimeNumber` 和 `HertzNumber` 是基于 `PhysicalNumber` 类，它使用 scala `BigDecimal` 来存储数字。

9.1.5 二进制前缀

SpinalHDL 允许根据 IEC 使用二进制前缀表示法定义整数。

```
val memSize = 512 MiB       // infers type BigInt
val dpRamSize = 4 KiB       // infers type BigInt
```

可以使用以下二进制前缀表示法：

二进制前缀	值
Byte, Bytes	1
KiB	1024 == 1 « 10
MiB	1024^2 == 1 « 20
GiB	1024^3 == 1 « 30
TiB	1024^4 == 1 « 40
PiB	1024^5 == 1 « 50
EiB	1024^6 == 1 « 60
ZiB	1024^7 == 1 « 70
YiB	1024^8 == 1 « 80

当然，BigInt 可以以字节为单位进行打印。例如，`BigInt(1024).byteUnit`.

```
val memSize = 512 MiB

println(memSize)
>> 536870912

println(memSize.byteUnit)
>> 512MiB

val dpRamSize = BigInt("123456789", 16)

println(dpRamSize.byteUnit())
>> 4GiB+564MiB+345KiB+905Byte
```

(续下页)

(接上页)

```
println((32.MiB + 12.KiB + 223).byteUnit())
>> 32MiB+12KiB+223Byte

println((32.MiB + 12.KiB + 223).byteUnit(ceil = true))
>> 33~MiB
```

9.2 存根（Stub）

您可以将组件层次结构清空作为一个存根（stub）：

```
class SubSysModule extends Component {
   val io = new Bundle {
     val dx = slave(Stream(Bits(32 bits)))
     val dy = master(Stream(Bits(32 bits)))
   }
   io.dy <-< io.dx
}
class TopLevel extends Component {
   val dut = new SubSysModule().stub   // instance an SubSysModule as empty stub
}
```

例如，它将生成以下 Verilog 代码：

```
module SubSysModule (
  input               io_dx_valid,
  output              io_dx_ready,
  input       [31:0]  io_dx_payload,
  output              io_dy_valid,
  input               io_dy_ready,
  output      [31:0]  io_dy_payload,
  input               clk,
  input               reset
);


  assign io_dx_ready = 1'b0;
  assign io_dy_valid = 1'b0;
  assign io_dy_payload = 32'h0;

endmodule
```

您还可以清空顶部组件

```
SpinalVerilog(new Pinsec(500 MHz).stub)
```

stub 有什么作用?

- 首先遍历所有组件并找出时钟，然后保留时钟
- 然后删除所有子组件
- then remove all assignment and logic we don’t want
- 给输出端口赋值 0

9.3 Assertions

除了 Scala 运行时断言 *assert(condition : Boolean, message : String)*，还可以使用以下语法添加硬件断言：

```
assert(assertion : Bool, message : String = null, severity:
AssertNodeSeverity = Error)
```

严重性等级是：

名称	描述
NOTE	用于报告提示性消息
WARNING	用于报告异常情况
ERROR	用于报告不应该发生的情况
FAILURE	用于报告致命情况并关闭仿真

一个实际的例子是检查当 `ready` 为低电平时，握手协议的 `valid` 信号不应该由高变低：

```
class TopLevel extends Component {
  val valid = RegInit(False)
  val ready = in Bool()

  when(ready) {
    valid := False
  }
  // some logic

  assert(
    assertion = !(valid.fall && !ready),
    message   = "Valid dropped when ready was low",
    severity  = ERROR
  )
}
```

备注： Scala 运行时断言 *assert(condition : Boolean, message : String)* 不支持严重性级别，一旦触发，将始终停止当前例化/仿真。

9.4 Report

您可以使用以下语法在 RTL 中添加调试以进行仿真：

```
object Enum extends SpinalEnum {
    val MIAOU, RAWRR = newElement()
}

class TopLevel extends Component {
    val a = Enum.RAWRR()
    val b = U(0x42)
    val c = out(Enum.RAWRR())
    val d = out (U(0x42))
    report(Seq("miaou ", a, b, c, d))
}
```

例如，它将生成以下 Verilog 代码：

```
$display("NOTE miaou %s%x%s%x", a_string, b, c_string, d);
```

从 SpinalHDL 1.4.4 开始，还支持以下语法：

```
report(L"miaou $a $b $c $d")
```

可以使用 REPORT_TIME 对象显示当前仿真时间

```
report(L"miaou $REPORT_TIME")
```

会导致：

```
$display("NOTE miaou %t", $time);
```

9.5 ScopeProperty

范围属性是可以在当前线程本地存储值的东西。它的 API 可用于设置/获取该值，还可以以堆栈方式对部分值进行修改。

换句话说，它是全局变量、scala 隐式变量、线程本地变量（ThreadLocal）的替代品。

- To compare with global variable, It allow to run multiple thread running the same code independently
- 与 scala 隐式变量相比，它与代码库的耦合较小
- 与线程本地变量（ThreadLocal）相比，它有一些 API 可以收集所有范围属性并稍后将它们恢复到相同状态

```
object Xlen extends ScopeProperty[Int]

object ScopePropertyMiaou extends App {
  Xlen.set(1)
  println(Xlen.get) // 1
  Xlen(2) {
    println(Xlen.get) // 2
    Xlen(3) {
      println(Xlen.get) // 3
      Xlen.set(4)
      println(Xlen.get) // 4
    }
    println(Xlen.get) // 2
  }
}
```

9.6 模拟信号和输入输出

9.6.1 简介

您可以使用 `Analog/inout` 功能定义三态信号。添加这些功能的原因有：

- 能够将三态信号添加到顶层（它避免了必须用一些手写的 VHDL/Verilog 包装它们）。
- 允许定义包含 `inout` 引脚的黑盒。
- 能够通过层次结构将黑盒的 `inout` 引脚连接到顶级 `inout` 引脚。

由于这些功能只是为了方便而添加的，因此请不要尝试使用三态逻辑的其他花哨的东西。

如果您想对内存映射 GPIO 外设等组件进行建模，请使用 Spinal 标准库中的*TriState/TriStateArray* 线束，它抽象了三态驱动程序的本质。

9.6.2 模拟信号

Analog 是一个关键字，它允许将信号定义为模拟信号，在数字世界中可能意味着 0, 1, 或 Z （断开、高阻状态）。

例如：

```
case class SdramInterface(g : SdramLayout) extends Bundle {
  val DQ    = Analog(Bits(g.dataWidth bits)) // Bidirectional data bus
  val DQM   = Bits(g.bytePerWord bits)
  val ADDR  = Bits(g.chipAddressWidth bits)
  val BA    = Bits(g.bankWidth bits)
  val CKE, CSn, CASn, RASn, WEn  = Bool()
}
```

9.6.3 输入/出

inout 是允许您将 Analog 信号设置为双向（“in” 和 “out”）信号的关键字。

例如：

```
case class SdramInterface(g : SdramLayout) extends Bundle with IMasterSlave {
  val DQ    = Analog(Bits(g.dataWidth bits)) // Bidirectional data bus
  val DQM   = Bits(g.bytePerWord bits)
  val ADDR  = Bits(g.chipAddressWidth bits)
  val BA    = Bits(g.bankWidth bits)
  val CKE, CSn, CASn, RASn, WEn  = Bool()

  override def asMaster() : Unit = {
    out(ADDR, BA, CASn, CKE, CSn, DQM, RASn, WEn)
    inout(DQ) // Set the Analog DQ as an inout signal of the component
  }
}
```

9.6.4 输入/出包装器

InOutWrapper 是一个工具，允许您将组件的所有 master TriState/TriStateArray/ReadableOpenDrain 线束转换为 inout(Analog(.. .)) 信号。它允许您保持硬件描述不受任何 Analog/inout 事物的影响，然后转换顶层以备综合。

例如：

```
case class Apb3Gpio(gpioWidth : Int) extends Component {
  val io = new Bundle {
    val gpio = master(TriStateArray(gpioWidth bits))
    val apb  = slave(Apb3(Apb3Gpio.getApb3Config()))
  }
  ...
}

SpinalVhdl(InOutWrapper(Apb3Gpio(32)))
```

这将生成：

```
entity Apb3Gpio is
  port(
    io_gpio : inout std_logic_vector(31 downto 0); -- This io_gpio was originally␣
↪a TriStateArray Bundle
    io_apb_PADDR : in unsigned(3 downto 0);
```

(续下页)

(接上页)

```
    io_apb_PSEL : in std_logic_vector(0 downto 0);
    io_apb_PENABLE : in std_logic;
    io_apb_PREADY : out std_logic;
    io_apb_PWRITE : in std_logic;
    io_apb_PWDATA : in std_logic_vector(31 downto 0);
    io_apb_PRDATA : out std_logic_vector(31 downto 0);
    io_apb_PSLVERROR : out std_logic;
    clk : in std_logic;
    reset : in std_logic
  );
end Apb3Gpio;
```

而不是：

```
entity Apb3Gpio is
  port(
    io_gpio_read : in std_logic_vector(31 downto 0);
    io_gpio_write : out std_logic_vector(31 downto 0);
    io_gpio_writeEnable : out std_logic_vector(31 downto 0);
    io_apb_PADDR : in unsigned(3 downto 0);
    io_apb_PSEL : in std_logic_vector(0 downto 0);
    io_apb_PENABLE : in std_logic;
    io_apb_PREADY : out std_logic;
    io_apb_PWRITE : in std_logic;
    io_apb_PWDATA : in std_logic_vector(31 downto 0);
    io_apb_PRDATA : out std_logic_vector(31 downto 0);
    io_apb_PSLVERROR : out std_logic;
    clk : in std_logic;
    reset : in std_logic
  );
end Apb3Gpio;
```

9.6.5 手动驱动模拟线束

如果 Analog 线束没有被驱动，它将默认为高阻态。因此，要手动实现三态驱动程序（以防因某种原因无法使用 InOutWrapper 类型），您必须有条件地驱动信号。

手动将 TriState 信号连接到 Analog 线束：

```
case class Example extends Component {
  val io = new Bundle {
    val tri = slave(TriState(Bits(16 bits)))
    val analog = inout(Analog(Bits(16 bits)))
  }
  io.tri.read := io.analog
  when(io.tri.writeEnable) { io.analog := io.tri.write }
}
```

9.7 VHDL 和 Verilog 生成

9.7.1 从 SpinalHDL 组件生成 VHDL 和 Verilog

要从 SpinalHDL 组件生成 VHDL，您只需在 Scala `main` 函数中调用 `SpinalVhdl(new YourComponent)` 即可。

生成 Verilog 完全相同，但用 `SpinalVerilog` 代替 `SpinalVHDL`

```
import spinal.core._

// A simple component definition.
class MyTopLevel extends Component {
  // Define some input/output signals. Bundle like a VHDL record or a Verilog␣
↪struct.
  val io = new Bundle {
    val a = in  Bool()
    val b = in  Bool()
    val c = out Bool()
  }

  // Define some asynchronous logic.
  io.c := io.a & io.b
}

// This is the main function that generates the VHDL and the Verilog corresponding␣
↪to MyTopLevel.
object MyMain {
  def main(args: Array[String]) {
    SpinalVhdl(new MyTopLevel)
    SpinalVerilog(new MyTopLevel)
  }
}
```

重要： `SpinalVhdl` 和 `SpinalVerilog` 可能需要创建组件类的多个实例，因此第一个参数不是 `Component` 引用，而是返回新组件的函数。

重要： `SpinalVerilog` 实施于 2016 年 6 月 5 日开始。该后端成功通过了与 VHDL 相同的回归测试（RISCV CPU、多核和流水线 Mandelbrot、UART RX/TX、单时钟域 fifo、双时钟域 fifo、格雷码计数器，…）。

如果您对这个新后端有任何问题，请创建 Github 工单 描述问题。

Scala 的参数化

参数名称	类型	默认值	描述
mode	SpinalMode	null	设置 SpinalHDL 生成 HDL 的模式。 可以设置为 VHDL 或 Verilog
defaultConfigForClockDomains	ClockDomainConfig	RisingEdgeClock AsynchronousReset ResetActiveHigh ClockEnableActiveHigh	设置将用作所有新“ClockDomain“时钟域默认值的配置。
onlyStdLogicVectorAtTopLevelIO	Boolean	false	将所有无符号/有符号顶级 io 更改为 std_logic_vector 类型。
defaultClockDomainFrequency	IClockDomainFrequency	UnknownFrequency	默认时钟频率。
targetDirectory	String	当前目录	生成文件的目录。

这是设置它们的语法：

```
SpinalConfig(mode=VHDL, targetDirectory="temp/myDesign").generate(new UartCtrl)

// Or for Verilog in a more scalable formatting:
SpinalConfig(
  mode=Verilog,
  targetDirectory="temp/myDesign"
).generate(new UartCtrl)
```

来自 shell 的参数化

您还可以使用命令行参数指定生成参数。

```
def main(args: Array[String]): Unit = {
  SpinalConfig.shell(args)(new UartCtrl)
}
```

命令行参数的语法是：

```
Usage: SpinalCore [options]

  --vhdl
        Select the VHDL mode
  --verilog
        Select the Verilog mode
  -d | --debug
        Enter in debug mode directly
  -o <value> | --targetDirectory <value>
        Set the target directory
```

9.7.2 生成的 VHDL 和 Verilog

如何将 SpinalHDL RTL 描述转换为 VHDL 和 Verilog 非常重要：

- Scala 中变量的名称将保留在 VHDL 和 Verilog 中。
- Scala 中的 `Component` 组件层次结构会保留在 VHDL 和 Verilog 中。
- Scala 中的 `when` 语句会生成为 VHDL 和 Verilog 中的 if 语句。
- Scala 中的 `switch` 语句在所有标准情况下都生成为 VHDL 和 Verilog 中的 case 语句。

组织

当您使用 VHDL 生成器时，所有模块都会生成到一个文件中，其中包含三个部分：

1. 包含所有 Enum 定义的包
2. 包含架构中所有元素使用函数的包
3. 您的设计所需的所有组件

当您使用 Verilog 生成时，所有模块都会生成到一个文件中，其中包含两个部分：

1. 使用的所有枚举定义
2. 您的设计需要的所有模块

组合逻辑

Scala:

```
class TopLevel extends Component {
  val io = new Bundle {
    val cond           = in  Bool()
    val value          = in  UInt(4 bits)
    val withoutProcess = out UInt(4 bits)
    val withProcess    = out UInt(4 bits)
  }
  io.withoutProcess := io.value
  io.withProcess := 0
  when(io.cond) {
    switch(io.value) {
      is(U"0000") {
        io.withProcess := 8
      }
      is(U"0001") {
        io.withProcess := 9
      }
      default {
        io.withProcess := io.value+1
      }
    }
  }
}
```

VHDL:

```
entity TopLevel is
  port(
    io_cond : in std_logic;
    io_value : in unsigned(3 downto 0);
    io_withoutProcess : out unsigned(3 downto 0);
```

(续下页)

(接上页)

```
    io_withProcess : out unsigned(3 downto 0)
  );
end TopLevel;

architecture arch of TopLevel is
begin
  io_withoutProcess <= io_value;
  process(io_cond,io_value)
  begin
    io_withProcess <= pkg_unsigned("0000");
    if io_cond = '1' then
      case io_value is
        when pkg_unsigned("0000") =>
          io_withProcess <= pkg_unsigned("1000");
        when pkg_unsigned("0001") =>
          io_withProcess <= pkg_unsigned("1001");
        when others =>
          io_withProcess <= (io_value + pkg_unsigned("0001"));
      end case;
    end if;
  end process;
end arch;
```

时序逻辑

Scala:

```
class TopLevel extends Component {
  val io = new Bundle {
    val cond    = in Bool()
    val value   = in UInt (4 bits)
    val resultA = out UInt(4 bits)
    val resultB = out UInt(4 bits)
  }

  val regWithReset = Reg(UInt(4 bits)) init(0)
  val regWithoutReset = Reg(UInt(4 bits))

  regWithReset := io.value
  regWithoutReset := 0
  when(io.cond) {
    regWithoutReset := io.value
  }

  io.resultA := regWithReset
  io.resultB := regWithoutReset
}
```

VHDL:

```
entity TopLevel is
  port(
    io_cond : in std_logic;
    io_value : in unsigned(3 downto 0);
    io_resultA : out unsigned(3 downto 0);
    io_resultB : out unsigned(3 downto 0);
    clk : in std_logic;
    reset : in std_logic
  );
```

(续下页)

(接上页)

```
end TopLevel;

architecture arch of TopLevel is

  signal regWithReset : unsigned(3 downto 0);
  signal regWithoutReset : unsigned(3 downto 0);
begin
  io_resultA <= regWithReset;
  io_resultB <= regWithoutReset;
  process(clk,reset)
  begin
    if reset = '1' then
      regWithReset <= pkg_unsigned("0000");
    elsif rising_edge(clk) then
      regWithReset <= io_value;
    end if;
  end process;

  process(clk)
  begin
    if rising_edge(clk) then
      regWithoutReset <= pkg_unsigned("0000");
      if io_cond = '1' then
        regWithoutReset <= io_value;
      end if;
    end if;
  end process;
end arch;
```

9.7.3 VHDL 和 Verilog 属性

在某些情况下，为设计中的某些信号提供属性以修改它们的综合方式很有用。

为此，您可以对设计中的任何信号或存储器调用以下函数：

语法	描述
`addAttribute(name)`	添加一个名为 `name` 的布尔属性，并将给定值设置为 true
`addAttribute(name, value)`	添加一个字符串属性，并将给定的 `name` 设置为 `value`

示例：

```
val pcPlus4 = pc + 4
pcPlus4.addAttribute("keep")
```

用 VHDL 生成声明：

```
attribute keep : boolean;
signal pcPlus4 : unsigned(31 downto 0);
attribute keep of pcPlus4: signal is true;
```

用 Verilog 生成声明：

```
(* keep *) wire [31:0] pcPlus4;
```

CHAPTER 10

模块库

spinal.lib 包的目的是：

- 提供硬件设计中常用的模块（FIFO、跨时钟域桥、一些有用的函数）
- 提供简单的外设 (UART, JTAG, VGA, ..)
- 提供一些总线定义 (Avalon, AMBA, ..)
- 提供一些方法 (Stream, Flow, Fragment)
- 提供一些例子以理解 spinal 的精髓
- Provide some tools and facilities (latency analyzer, QSys converter, ···)

要使用以下章节中介绍的特性，在大多数情况下，您需要使用 `import spinal.lib._` 在您的代码中。

重要:

该包目前正在建设中。文档中的特性可以被认为是稳定的。
请不要犹豫，使用 GitHub 提供建议/错误/修复/增强等意见

10.1 实用工具

一些实用工具也在*spinal.core* 中

10.1.1 免状态工具

语法	返回类型	描述
toGray(x : UInt)	位	返回从 x (UInt) 转换而来的灰度值
fromGray(x : Bits)	UInt	返回从 x (灰度) 转换而来的 UInt 值
Reverse(x : T)	T	翻转所有位 (lsb + n -> msb - n)
OHToUInt(x : Seq[Bool]) OHToUInt(x : BitVector)	UInt	返回 x 中唯一被设置（为 1）的比特位的索引
CountOne(x : Seq[Bool]) CountOne(x : BitVector)	UInt	返回 x 中被设为 1 的位数
CountLeadingZeroes(x : Bits)	UInt	返回从最显著位 (MSB) 开始连续零位的个数
MajorityVote(x : Seq[Bool]) MajorityVote(x : BitVector)	Bool	如果设置为 1 的位数 > x.size / 2，则返回 True
EndiannessSwap(that: T[, base:BitCount])	T	大端 <-> 小端
OHMasking.first(x : Bits)	位	对 x 应用掩码以仅保留第一个设为 1 的位
OHMasking.last(x : Bits)	位	对 x 应用掩码以仅保留最后一个设为 1 的位
OHMasking.roundRobin(requests : Bits, ohPriority : Bits)	位	对 x 应用掩码以仅保留 `requests` 中设置 1 的位。 它从 `ohPriority` 位置开始访问 `requests` 。 例如，如果 `requests` 为“1001”且 `ohPriority` 为“0010”，则 `roundRobin` 函数将从第二位开始访问 *requests* 并返回“1000”。
MuxOH (oneHot : IndexedSeq[Bool], inputs : Iterable[T])	T	根据 `oneHot` 向量从 `inputs` 中返回多路复用的 `T` 。
PriorityMux (sel: Seq[Bool], in: Seq[T])	T	返回第一个其 `sel` 为 `True` 的 `in` 元素。
PriorityMux (in: Seq[(Bool, T)]	T	返回第一个其 `sel` 为 `True` 的 `in` 元素。

10.1.2 全状态工具

语法	返回类型	描述
Delay(that: T, cycleCount: Int)	T	返回延迟了 cycleCount 周期的 that
History (that: T, length: Int [, when : Bool][, init : T])	Vec[T]	返回长度为 length 的 Vec 其第一个元素是 that ，最后一个元素是延迟了 length -1 的 that 内部移位寄存器会在 when 有效时采样
History (that: T, range: Range [, when : Bool][, init : T])	Vec[T]	和 History(that, length) 相同 但返回长度为 range.length 的 Vec 其中第一个元素延迟了 range.low 个周期 且最后一个是延迟了 range.high 个周期的元素
BufferCC(input : T)	T	返回利用两个触发器同步到当前时钟域的同步输入信号

计数器

计数器工具可用于轻松实例化硬件计数器。

实例化语法	备注
Counter(start: BigInt, end: BigInt[, inc : Bool])	
Counter(range : Range[, inc : Bool])	与 x to y x until y 语法兼容
Counter(stateCount: BigInt[, inc : Bool])	从 0 开始，到 stateCount - 1 结束
Counter(bitCount: BitCount[, inc : Bool])	从 0 开始到 (1 << bitCount) - 1 结束

计数器可以通过方法控制，并且可以连线可以被读取：

```
val counter = Counter(2 to 9) // Creates a counter of 8 states (2 to 9)
// Methods
counter.clear()               // Resets the counter
counter.increment()           // Increments the counter
// Wires
counter.value                 // Current value
counter.valueNext             // Next value
counter.willOverflow          // True if the counter overflows this cycle
counter.willOverflowIfInc     // True if the counter would overflow this cycle if␣
↪an increment was done
// Cast
when(counter === 5){ ... }    // counter is implicitly casted to its current value
```

当 Counter 溢出（达到最终值）时，它会重新启动下一个周期并设置为起始值。

备注： 目前仅支持向上计数器。

CounterFreeRun 构建一个始终运行的计数器：CounterFreeRun(stateCount: BigInt) 。

超时

超时工具可用于方便地实例化一个硬件超时。

实例化语法	备注
Timeout(cycles : BigInt)	在 cycles 个时钟周期后触发超时（tick）
Timeout(time : TimeNumber)	在持续 time 时间后触发超时（tick）
Timeout(frequency : HertzNumber)	以 frequency 频率触发超时信号（tick）

下方是可以与 Counter 工具一起使用的不同语法句式的示例

```
val timeout = Timeout(10 ms)   // Timeout who tick after 10 ms
when(timeout) {                // Check if the timeout has tick
    timeout.clear()            // Ask the timeout to clear its flag
}
```

备注： 如果您使用时间或频率设置实例化 Timeout ，则隐含地 ClockDomain 应该具有频率设置。

复位控制

复位控制 (ResetCtrl) 提供了一些实用工具来管理复位。

asyncAssertSyncDeassert

You can filter an asynchronous reset by using an asynchronously asserted synchronously deasserted logic. To do it you can use the ResetCtrl.asyncAssertSyncDeassert function which will return you the filtered value.

参数名称	类型	描述
input	Bool	输入的异步信号，用于触发复位
clockDomain	ClockDomain	返回的新复位信号对齐的时钟域 (ClockDomain)
inputPolarity	Polarity	HIGH/LOW (default=HIGH)
outputPolarity	Polarity	HIGH/LOW (default=clockDomain.config.resetActiveLevel)
bufferDepth	Int	防止亚稳态所需的寄存器级数（默认为 2）

另外还有一个 ResetCtrl.asyncAssertSyncDeassertDrive 版本的工具，它直接使用新构造的复位信号作为 clockDomain 这个时钟域的复位。

10.1.3 特殊工具

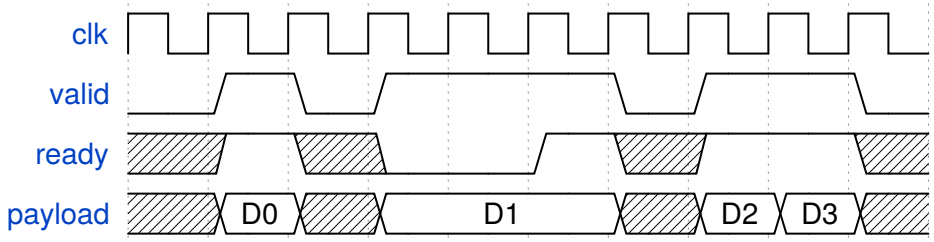
语法	返回类型	描述
LatencyAnalysis(paths : Node*)	Int	以周期为单位返回经过所有节点的最短路径，从第一个节点到最后一个节点

10.2 Stream

10.2.1 规范

反压流 (Stream) 是一个用于承载有效负载的简单握手协议。
例如，它可用于对 FIFO 推入和弹出数据、向 UART 控制器发送请求等。

信号	类型	驱动	描述	何时忽略
valid	Bool	Master	当为高时 => 接口上存在有效负载 (payload)	
ready	Bool	Slave	当为低时 => 从端不接收传输	valid 为低
payload	T	Master	传输任务内容	valid 为低



这里有一些在 SpinalHDL 中的用法示例：

```
class StreamFifo[T <: Data](dataType: T, depth: Int) extends Component {
  val io = new Bundle {
    val push = slave Stream (dataType)
    val pop = master Stream (dataType)
  }
  ...
}

class StreamArbiter[T <: Data](dataType: T,portCount: Int) extends Component {
  val io = new Bundle {
    val inputs = Vec(slave Stream (dataType),portCount)
    val output = master Stream (dataType)
  }
  ...
}
```

备注： 当 valid 为高且 ready 为低时，每个从端都可以控制是否允许有效负载变化。例如：

- 没有锁逻辑的优先级仲裁器可以从一个输入切换到另一个输入（这将改变有效负载）。
- UART 控制器可以直接使用写端口驱动 UART 引脚，并且只在传输结束时完成数据交换。对此需要注意。

10.2.2 语义

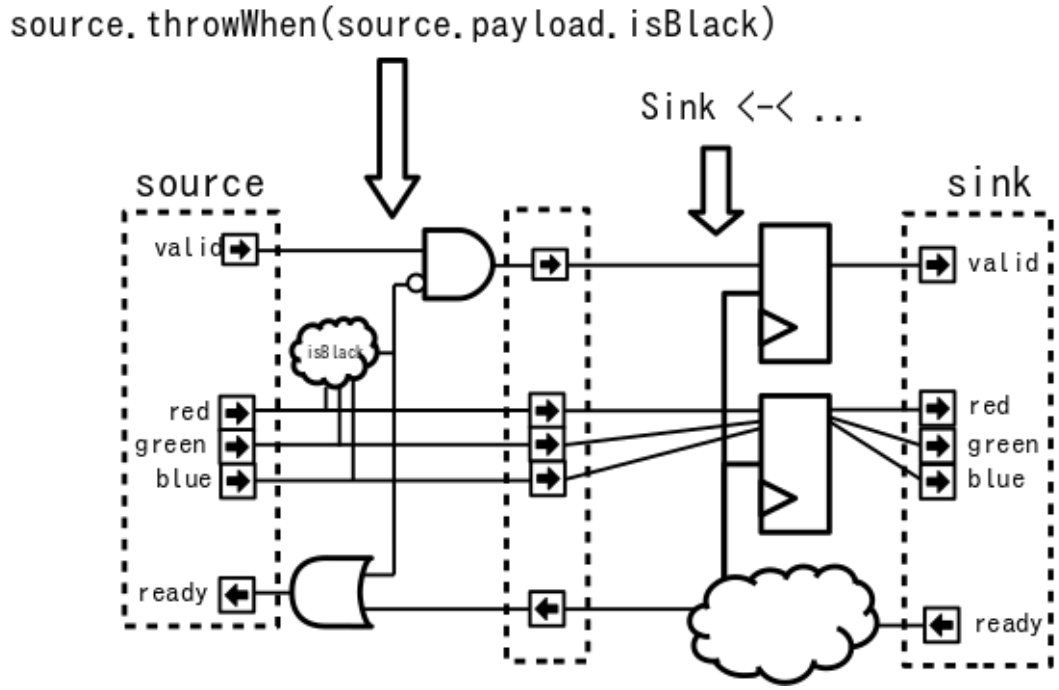
当手动读取/驱动反压流的信号时，请记住：

- After being asserted, `valid` may only be deasserted once the current payload was acknowledged. This means `valid` can only toggle to 0 the cycle after a the slave did a read by asserting `ready`.
- 相反，`ready` 可以随时改变。
- 传输仅在 `valid` 和 `ready` 均已置高的周期内进行。
- 一个反压流的 `valid` 不能以组合逻辑方式和 `ready` 连接，并且两者之间的任何路径都必须经过寄存器。
- 推荐 `valid` 和 `ready` 之间没有任何依赖。

10.2.3 函数

语法	描述	返回类型	延迟
Stream(type : Data)	创建一个给定类型的 Stream 反压流	Stream[T]	
master/slave Stream(type : Data)	创建一个给定类型的 Stream 反压流 通过相应的输入/输出设置进行初始化	Stream[T]	
x.fire	当总线上的传输完成时返回 True(valid && ready)	Bool	
x.isStall	当总线上的传输停滞时返回 True(valid && ! ready)	Bool	
x.queue(size:Int)	返回一个通过 FIFO 连接到 x 的 Stream	Stream[T]	2
x.m2sPipe() x.stage()	Return a Stream driven by x 通过寄存器，切断 valid/payload 路径 Cost = (payload width + 1) 触发器	Stream[T]	1
x.s2mPipe()	Return a Stream driven by x 通过寄存器级断开 ready 路径 Cost = payload width * (mux2 + 1 flip flop)	Stream[T]	0
x.halfPipe()	Return a Stream driven by x valid/ready/payload 路径通过一些寄存器分割 成本 = (payload 位宽 + 2) 个触发器，带宽除以二	Stream[T]	1
x « y y » x	将 y 连接到 x		0
x <-< y y >-> x	通过 m2sPipe 将 y 连接到 x		1
x </< y y >/> x	通过 s2mPipe 将 y 连接到 x		0
x <-/< y y >/-> x	通过 s2mPipe().m2sPipe() 将 y 连接到 x 这意味着 x 和 y 之间没有组合逻辑路径		1
x.haltWhen(cond : Bool)	返回连接到 x 的反压流 cond 为 true 时暂停	Stream[T]	0
x.throwWhen(cond : Bool)	返回连接到 x 的反压流 当 cond 为 true 时，传输数据将被抛弃	Stream[T]	0
x.translateWith(that : T2)		Stream[T2]	0

以下代码将创建此逻辑：




```
case class RGB(channelWidth : Int) extends Bundle {
  val red   = UInt(channelWidth bits)
  val green = UInt(channelWidth bits)
  val blue  = UInt(channelWidth bits)

  def isBlack : Bool = red === 0 && green === 0 && blue === 0
}

val source = Stream(RGB(8))
val sink   = Stream(RGB(8))
sink <-< source.throwWhen(source.payload.isBlack)
```

10.2.4 实用工具

有许多实用工具可以在设计中与反压流总线结合使用，本章将介绍它们。

StreamFifo

您可以在每个反压流上调用.queue(size) 来获取一个缓冲反压流。但您也可以实例化 FIFO 组件本身：

```
val streamA,streamB = Stream(Bits(8 bits))
// ...
val myFifo = StreamFifo(
  dataType = Bits(8 bits),
  depth    = 128
)
myFifo.io.push << streamA
myFifo.io.pop  >> streamB
```

参数名称	类型	描述
数据类型	T	有效负载 (payload) 数据类型
depth	Int	用于存储数据的存储器的大小

io 名称	类型	描述
push	Stream[T]	用于压入数据
pop	Stream[T]	用于弹出数据
flush	Bool	用于清除 FIFO 内的所有数据
occupancy	log2Up(depth + 1) bits 的 UInt	反映内部存储占用情况

StreamFifoCC

您可以通过以下方式实例化双时钟域版本的 fifo：

```
val clockA = ClockDomain(???)
val clockB = ClockDomain(???)
val streamA,streamB = Stream(Bits(8 bits))
// ...
val myFifo = StreamFifoCC(
  dataType  = Bits(8 bits),
  depth     = 128,
  pushClock = clockA,
  popClock  = clockB
)
myFifo.io.push << streamA
myFifo.io.pop  >> streamB
```

参数名称	类型	描述
数据类型	T	有效负载 (payload) 数据类型
depth	Int	用于存储数据的存储器的大小
pushClock	ClockDomain	压入数据端使用的时钟域
popClock	ClockDomain	弹出数据端使用的时钟域

io 名称	类型	描述
push	Stream[T]	用于压入数据
pop	Stream[T]	用于弹出数据
pushOccupancy	log2Up(depth + 1) bits 的 UInt	反映内部存储器占用情况（从压入数据端的角度）
popOccupancy	log2Up(depth + 1) bits 的 UInt	反映内部存储器占用情况（从弹出数据端的角度）

StreamCCByToggle

基于信号切换来连接跨时钟域的反压流组件。
这种实现跨时钟域桥的方式的特点是占用逻辑区小，但带宽较低。

```
val clockA = ClockDomain(???)
val clockB = ClockDomain(???)
val streamA,streamB = Stream(Bits(8 bits))
// ...
val bridge = StreamCCByToggle(
  dataType    = Bits(8 bits),
  inputClock  = clockA,
  outputClock = clockB
)
```

(续下页)

(接上页)

```
bridge.io.input  << streamA
bridge.io.output >> streamB
```

参数名称	类型	描述
数据类型	T	有效负载 (payload) 数据类型
inputClock	ClockDomain	压入数据端使用的时钟域
outputClock	ClockDomain	弹出数据端使用的时钟域

io 名称	类型	描述
input	Stream[T]	用于压入数据
output	Stream[T]	用于弹出数据

或者您也可以使用更简短的语句，直接返回跨时钟域的反压流：

```
val clockA = ClockDomain(???)
val clockB = ClockDomain(???)
val streamA = Stream(Bits(8 bits))
val streamB = StreamCCByToggle(
  input       = streamA,
  inputClock  = clockA,
  outputClock = clockB
)
```

StreamWidthAdapter（反压流位宽适应器）

该组件使输入反压流的位宽和输出反压流匹配。当 outStream 的负载的位宽大于 inStream 时，通过将多个输入传输任务的负载合并为一个；相反，如果 outStream 的负载位宽小于 inStream ，则一个输入传输任务将被拆分为多个输出传输任务。

在最好的情况下，inStream 的负载位宽应该是 outStream 的整数倍，如下所示。

```
val inStream = Stream(Bits(8 bits))
val outStream = Stream(Bits(16 bits))
val adapter = StreamWidthAdapter(inStream, outStream)
```

如上例所示，两个 inStream 传输任务将合并为一个 outStream 传输任务，并且第一个输入传输任务的负载将默认置于输出负载的低位上。

如果输入传输任务负载放置的期望顺序与默认设置不同，请参阅以下示例。

```
val inStream = Stream(Bits(8 bits))
val outStream = Stream(Bits(16 bits))
val adapter = StreamWidthAdapter(inStream, outStream, order = SlicesOrder.HIGHER_
↪FIRST)
```

还有一个称为 endianness 的传统参数，它与 ORDER 具有相同的效果。当 endianness 的值为 LITTLE 时，它与 order 的 LOWER_FIRST 相同；当为 BIG 时，它与 HIGHER_FIRST 相同。padding 参数是一个可选的布尔逻辑值，它用于确定适应器是否接受输入和输出负载位宽为非整数倍。

StreamArbiter（反压流仲裁器）

当您有多个 Stream 反压流并且您想要仲裁它们以驱动单个反压流时，您可以使用 StreamArbiterFactory。

```
val streamA, streamB, streamC = Stream(Bits(8 bits))
val arbitredABC = StreamArbiterFactory.roundRobin.onArgs(streamA, streamB, streamC)

val streamD, streamE, streamF = Stream(Bits(8 bits))
val arbitredDEF = StreamArbiterFactory.lowerFirst.noLock.onArgs(streamD, streamE,␣
↪streamF)
```

仲裁函数	描述
lowerFirst	较低端口优先级高于较高端口
roundRobin	公平轮询仲裁
sequentialOrder	Could be used to retrieve transaction in a sequential order 第一个传输应该来自端口 0，然后来自端口 1，…

锁函数	描述
noLock	端口选择可以在每个周期改变，即使被选择的端口的传输没有执行。
transaction-Lock	端口选择被锁定，直到所选端口上的数据传输完成。
fragmentLock	可用于仲裁 Stream[Flow[T]]。 在此模式下，端口选择被锁定，直到所选端口完成突发（last=True）。

生成函数	返回类型
on(inputs : Seq[Stream[T]])	Stream[T]
onArgs(inputs : Stream[T]*)	Stream[T]

StreamJoin

该实用工具接收多个输入反压流并等待所有输入反压流触发 *valid* ，然后再通过提供 *ready* 信号让所有输入流通过。

```
val cmdJoin = Stream(Cmd())
cmdJoin.arbitrationFrom(StreamJoin.arg(cmdABuffer, cmdBBuffer))
```

StreamFork

StreamFork 会将每个传入数据克隆到其所有输出流。如果 synchronous 为 true，则所有输出流将始终一起触发，这意味着直到所有输出流准备就绪前该流将都将暂停。如果 synchronous 为 false，那么一次可能只有一个输出流准备就绪，但需要一个额外的触发器（每个输出 1 位）。直到所有输出流都处理完每个项目前，输入流将阻塞。

```
val inputStream = Stream(Bits(8 bits))
val (outputstream1, outputstream2) = StreamFork2(inputStream, synchronous=false)
```

或者

```
val inputStream = Stream(Bits(8 bits))
val outputStreams = StreamFork(inputStream,portCount=2, synchronous=true)
```

StreamMux

Stream 的多路复用器实现。它接受一个 select 信号和 inputs 中的反压流，并返回一个 Stream，该 Stream 连接到 select 指定的其中一个输入流。StreamArbiter 是一个与此类似的工具，但功能更强大。

```
val inputStreams = Vec(Stream(Bits(8 bits)), portCount)
val select = UInt(log2Up(inputStreams.length) bits)
val outputStream = StreamMux(select, inputStreams)
```

备注： 当输出流暂停时，select 信号的 UInt 类型不能更改，否则可能会中断执行中的传输任务。使用 Stream 类型的 select 可以生成一个流接口，该接口仅在安全时触发并更改路径。

StreamDemux

Stream 的解复用实现。它需要一个 input 、一个 select 和一个 portCount 并返回一个 Vec(Stream)，其中输出流由 select 指定并连接到 input，其他输出流处于非活动状态。为了安全传输，请参阅上面的注释。

```
val inputStream = Stream(Bits(8 bits))
val select = UInt(log2Up(portCount) bits)
val outputStreams = StreamDemux(inputStream, select, portCount)
```

StreamDispatcherSequencial

该工具获取其输入流并将其按顺序连接到 outputCount 反压流。

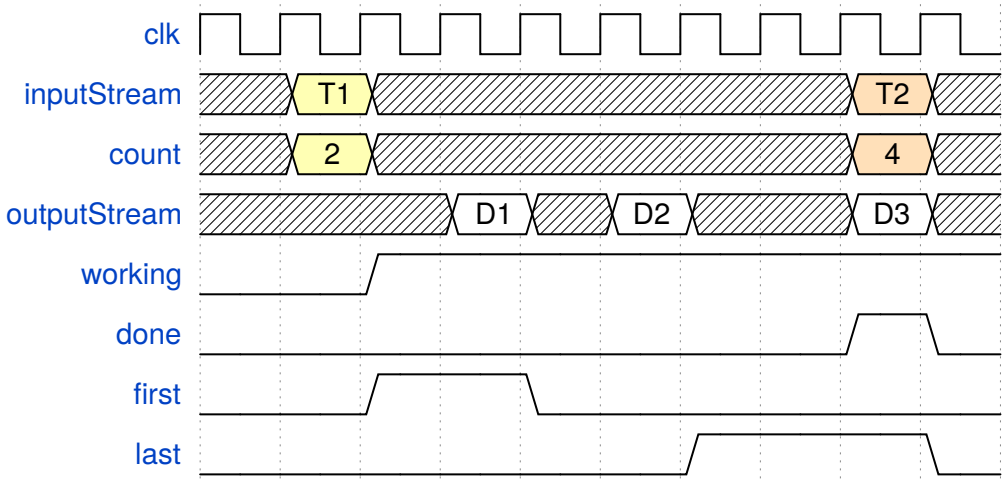
```
val inputStream = Stream(Bits(8 bits))
val dispatchedStreams = StreamDispatcherSequencial(
  input = inputStream,
  outputCount = 3
)
```

StreamTransactionExtender

该工具将使用一个输入传输并生成多个输出传输，它提供了将负载值重复 count+1 次到输出传输的功能。每当为单个负载而触发 inputStream 时，都会捕获并寄存 count 。

```
val inputStream = Stream(Bits(8 bits))
val outputStream = Stream(Bits(8 bits))
val count = UInt(3 bits)
val extender = StreamTransactionExtender(inputStream, outputStream, count) {
   // id, is the 0-based index of total output transfers so far in the current␣
↪input transaction.
   // last, is the last transfer indication, same as the last signal for extender.
   // the returned payload is allowed to be modified only based on id and last␣
↪signals, other translation should be done outside of this.
    (id, payload, last) => payload
}
```

该 extender 提供了多种状态信号，例如 working 、last 、done ，其中 working 表示有一个输入传输已接受并正在进行中，last 表示最后一个输出传输已准备好并等待完成，done 变为有效表示最后一个输出传输正在触发，并使当前输入传输任务处理完成且准备好启动另一个传输。



备注： 如果仅需要对输出流计数，那么可以使用 StreamTransactionCounter 。

10.2.5 仿真支持

对于仿真，有以下可用的主端和从端实现：

类	用法
StreamMonitor	用于主端和从端，如果 Stream 触发，则调用带有负载的函数。
StreamDriver	Testbench 中主端通过调用函数来应用值（如果可用）以驱动值。如果值可用，则函数必须返回。支持随机的延迟。
StreamReadyRandomizer	随机产生 ready 以接收数据，testbench 为从端。
ScoreboardInOrder	通常用于比较参考/dut 数据

```
import spinal.core._
import spinal.core.sim._
import spinal.lib._
import spinal.lib.sim.{StreamMonitor, StreamDriver, StreamReadyRandomizer,␣
↪ScoreboardInOrder}

object Example extends App {
  val dut = SimConfig.withWave.compile(StreamFifo(Bits(8 bits), 2))

  dut.doSim("simple test") { dut =>
    SimTimeout(10000)

    val scoreboard = ScoreboardInOrder[Int]()

    dut.io.flush #= false

    // drive random data and add pushed data to scoreboard
    StreamDriver(dut.io.push, dut.clockDomain) { payload =>
      payload.randomize()
      true
```

(续下页)

(接上页)

```
    }
    StreamMonitor(dut.io.push, dut.clockDomain) { payload =>
      scoreboard.pushRef(payload.toInt)
    }

    // randmize ready on the output and add popped data to scoreboard
    StreamReadyRandomizer(dut.io.pop, dut.clockDomain)
    StreamMonitor(dut.io.pop, dut.clockDomain) { payload =>
      scoreboard.pushDut(payload.toInt)
    }

    dut.clockDomain.forkStimulus(10)

    dut.clockDomain.waitActiveEdgeWhere(scoreboard.matches == 100)
  }
}
```

10.3 Flow

10.3.1 规范

数据流接口是一个简单的有效/负载协议，这意味着从端无法终止总线。它可用于表示来自 UART 控制器的数据、写入片上存储器的请求等。

信号	类型	驱动	描述	何时忽略
valid	Bool	Master	当为高时 => 接口上存在有效负载 (payload)	
payload	T	Master	传输任务内容	valid 为低

10.3.2 函数

语法	描述	返回类型	延迟
Flow(type: Data)	创建给定类型的流	Flow[T]	
master/slave Flow(type: Data)	创建给定类型的流 通过相应的输入/输出设置进行初始化	Flow[T]	
x.m2sPipe()	Return a Flow driven by x 通过寄存器，切断 valid/payload 路径	Flow[T]	1
x.stage()	等价于 x.m2sPipe()	Flow[T]	1
x « y y » x	将 y 连接到 x		0
x <-< y y >-> x	通过 m2sPipe 将 y 连接到 x		1
x.throwWhen(cond: Bool)	返回连接到 x 的流 当 cond 为高时，放弃传输的事务	Flow[T]	0
x.toReg()	返回一个寄存器，当 valid 为高时，该寄存器将加载 payload	T	
x.setIdle()	将流量设置为闲置状态：valid 设为 False ，且不关心 payload。		
x.push(newPayload: T)	为 payload 设置新的有效 payload。valid 被设为 True。		

10.3.3 代码示例

```
case class FlowExample() extends Component {
  val io = new Bundle {
    val request = slave(Flow(Bits(8 bit)))
    val answer = master(Flow(Bits(8 bit)))
  }
  val storage = Reg(Bits(8 bit))

  val fsm = new StateMachine {
    io.answer.setIdle()

    val idle: State = new State with EntryPoint {
      whenIsActive {
        when(io.request.valid) {
          storage := io.request.payload
          goto(sendEcho)
```

(续下页)

(接上页)

```
        }
      }
    }

    val sendEcho: State = new State {
      whenIsActive {
        io.answer.push(storage)
        goto(idle)
      }
    }
  }

  // This StateMachine behaves equivalently to
  // io.answer <-< io.request
}
```

10.3.4 仿真支持

类	用法
FlowMonitor	用于主端和从端，如果数据流传输数据，则调用带有负载的函数。
FlowDriver	Testbench 中主端通过调用函数来应用值（如果可用）以驱动值。如果值可用，则函数必须返回。支持随机的延迟。
ScoreboardInOrder	通常用于比较参考/dut 数据

```
package spinaldoc.libraries.flow

import spinal.core._
import spinal.core.sim._
import spinal.lib._
import spinal.lib.sim.{FlowDriver, FlowMonitor, ScoreboardInOrder}

import scala.language.postfixOps

case class SomeDUT() extends Component {
  val io = new Bundle {
    val input = slave(Flow(UInt(8 bit)))
    val output = master(Flow(UInt(8 bit)))
  }
  io.output <-< io.input
}

object Example extends App {
  val dut = SimConfig.withWave.compile(SomeDUT())

  dut.doSim("simple test") { dut =>
    SimTimeout(10000)

    val scoreboard = ScoreboardInOrder[Int]()

    // drive random data at random intervals, and add inputted data to scoreboard
    FlowDriver(dut.io.input, dut.clockDomain) { payload =>
      payload.randomize()
      true
    }
    FlowMonitor(dut.io.input, dut.clockDomain) { payload =>
      scoreboard.pushRef(payload.toInt)
```

(续下页)

(接上页)

```
    }

    // add all data coming out of DUT to scoreboard
    FlowMonitor(dut.io.output, dut.clockDomain) { payload =>
      scoreboard.pushDut(payload.toInt)
    }

    dut.clockDomain.forkStimulus(10)
    dut.clockDomain.waitActiveEdgeWhere(scoreboard.matches == 100)
  }
}
```

10.4 Fragment

10.4.1 规范

`Fragment` 包是通过使用多个“小”片段来传输“大”东西的概念。比如：

- 一张通过 `Stream[Fragment[Pixel]]` 传输尺寸为宽 * 高的图片
- 一个可以在 `Flow[Fragment[Bits]]` 上传输的、从没有流量控制的控制器里接收到的 UART 数据包
- 一个可以由 `Stream[Fragment[AxiReadResponse]]` 承载的 AXI 突发读

`Fragment` 包定义的信号是：

信号	类型	驱动	描述
fragment	T	Master	当前传输任务的“负载”
last	Bool	Master	当该片段是当前数据包的最后一个片段时为高

正如您在本说明和先例中看到的，`Fragment` 概念并未指定传输任务如何传输（您可以使用 Stream、Flow 或任何其他通信协议）。它仅添加足够的信息 (`last`) 来了解当前传输是给定数据包的第一个、最后一个还是中间的一个。

备注： 该协议没有携带’first’ 位，因为它可以通过执行’RegNextWhen(bus.last, bus.fire) init(True)’ 被生成在任何位置

10.4.2 函数

对于 `Stream[Fragment[T]]` 和 `Flow[Fragment[T]]`，给出以下函数：

语法	返回类型	描述
x.first	Bool	当当前或下一个传输是/将是数据包的第一个片段时返回 True
x.tail	Bool	当当前或下一个传输不是数据包的第一个片段时返回 True
x.isFirst	Bool	当当前传输任务被提交并且为数据包的第一个片段时返回 True
x.isTail	Bool	当当前传输任务被提交并且不是数据包的第一个/最后一个片段时返回 True
x.isLast	Bool	当当前传输任务被提交并且为数据包的最后一个片段时返回 True

对于 `Stream[Fragment[T]]`，以下函数也是可用的：

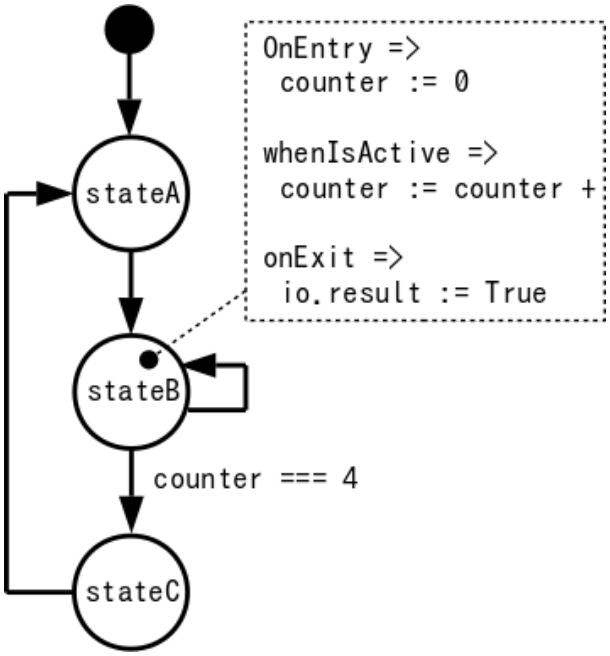
语法	返回类型	描述
x.insertHeader(header : T)	Stream[Fragment[T]]	对每个数据包 `x` 添加 `header` 并返回生成的总线

10.5 状态机

10.5.1 简介

在 SpinalHDL 中，您可以像在 VHDL/Verilog 中一样，通过使用枚举和 switch/case 语句来定义状态机。但在 SpinalHDL 中，您还可以使用专门的语句。

下面的状态机在随后的示例中实现：



样式 A：

```
import spinal.lib.fsm._

class TopLevel extends Component {
  val io = new Bundle {
    val result = out Bool()
  }

  val fsm = new StateMachine {
    val counter = Reg(UInt(8 bits)) init(0)
    io.result := False

    val stateA : State = new State with EntryPoint {
      whenIsActive(goto(stateB))
    }
    val stateB : State = new State {
      onEntry(counter := 0)
      whenIsActive {
        counter := counter + 1
        when(counter === 4) {
          goto(stateC)
        }
```

(续下页)

(接上页)

```
      }
      onExit(io.result := True)
    }
    val stateC : State = new State {
      whenIsActive(goto(stateA))
    }
  }
}
```

样式 B:

```
import spinal.lib.fsm._

class TopLevel extends Component {
  val io = new Bundle {
    val result = out Bool()
  }

  val fsm = new StateMachine {
    val stateA = new State with EntryPoint
    val stateB = new State
    val stateC = new State

    val counter = Reg(UInt(8 bits)) init(0)
    io.result := False

    stateA
      .whenIsActive(goto(stateB))

    stateB
      .onEntry(counter := 0)
      .whenIsActive {
        counter := counter + 1
        when(counter === 4) {
          goto(stateC)
        }
      }
      .onExit(io.result := True)

    stateC
      .whenIsActive(goto(stateA))
  }
}
```

10.5.2 StateMachine

StateMachine 是基类，它管理 FSM 的逻辑。

```
val myFsm = new StateMachine {
  // Definition of states
}
```

StateMachine 还提供了一些访问器:

名称	返回类型	描述
isActive(state)	Bool	当状态机处于给定状态时返回 True
isEntering(state)	Bool	当状态机进入给定状态时返回 True

入口点

通过扩展 EntryPoint 特征，可以将状态定义为状态机的入口点：

```
val stateA = new State with EntryPoint
```

或者使用 `setEntry(state)`：

```
val stateA = new State
setEntry(stateA)
```

转换

- 转换由 `goto(nextState)` 表示，它使状态机的状态在下一个周期转换到 `nextState` 。
- `exit()` 使状态机在下一个周期处于启动 (boot) 状态（或者，在 `StateFsm` 中，退出当前的嵌套状态机）。

这两个函数可以在状态定义中使用（见下文），或使用 `always { yourStatements }`，这将始终应用 `yourStatements`，并且优先级高于状态。

状态编码

默认情况下，FSM 状态向量将使用 RTL 生成语言/工具（Verilog 或 VHDL）的本地编码进行编码。可以使用 `setEncoding(...)` 方法重写默认编码，该方法可以使用 `SpinalEnumEncoding` 或 `(State, BigInt)` 类型的 varargs 来获取自定义编码。

列表 1: 使用 `SpinalEnumEncoding`

```
val fsm = new StateMachine {
  setEncoding(binaryOneHot)

  ...
}
```

列表 2: 使用自定义编码

```
val fsm = new StateMachine {
  val stateA = new State with EntryPoint
  val stateB = new State
  ...
  setEncoding((stateA -> 0x23), (stateB -> 0x22))
}
```

警告: 当使用 graySequential 枚举编码时，不会进行任何检查以验证 FSM 转换是否只在状态向量中产生单比特的变化。编码是根据状态定义的顺序完成的，设计者必须确保仅在需要时进行有效的转换。

10.5.3 状态

可以使用多种类型的状态:

- State （基础状态）
- StateDelay
- StateFsm
- StateParallelFsm

它们每个都提供了以下函数来定义与之相关的逻辑:

名称	描述

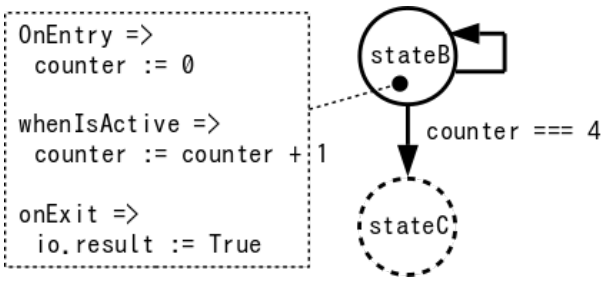
state.onEntry{
 	
yourStatements
}
	当状态机不在 state 状态，并且在下一个周期将处于 state 状态时，执行 yourStatements

state.onExit{
 	
yourStatements
}
	当状态机在 state 状态时执行 yourStatements，并且在下一个周期将处于另一个状态

state.whenIsActive{
 	
yourStatements
}
	当状态机在 state 状态时执行 yourStatements

state.whenIsNext{
 	
yourStatements
}
	当状态机在下一个周期处于 state 状态时，yourStatements 被执行（即使它已经处于该状态）

state. 隐含在 new State 块中:




```
val stateB : State = new State {
  onEntry(counter := 0)
  whenIsActive {
    counter := counter + 1
    when(counter === 4) {
      goto(stateC)
    }
  }
  onExit(io.result := True)
}
```

StateDelay（状态延迟）

StateDelay 允许您创建一个状态，该状态在执行 whenCompleted {...} 中的语句之前等待固定数量的周期。首选的使用方式是:

```
val stateG : State = new StateDelay(cyclesCount=40) {
  whenCompleted {
    goto(stateH)
  }
}
```

也可以写成一行:

```
val stateG : State = new StateDelay(40) { whenCompleted(goto(stateH)) }
```

StateFsm

StateFsm 允许您描述一个包含嵌套状态机的状态。当嵌套状态机完成(退出)时，执行 whenCompleted { ... } 中的语句。

这是一个 StateFsm 定义的示例:

```
// internalFsm is a function defined below
val stateC = new StateFsm(fsm=internalFsm()) {
  whenCompleted {
    goto(stateD)
  }
}

def internalFsm() = new StateMachine {
  val counter = Reg(UInt(8 bits)) init(0)

  val stateA : State = new State with EntryPoint {
    whenIsActive {
      goto(stateB)
    }
```

(续下页)

(接上页)

```
  }

  val stateB : State = new State {
    onEntry (counter := 0)
    whenIsActive {
      when(counter === 4) {
        exit()
      }
      counter := counter + 1
    }
  }
}
```

在上面的示例中，exit() 使状态机跳转到启动状态（内部隐藏状态）。这将通知 StateFsm 其内部状态机已经完成。

StateParallelFsm

StateParallelFsm 允许您处理多个嵌套状态机。当所有嵌套状态机完成时，执行 whenCompleted { ... } 中的语句。

示例：

```
val stateD = new StateParallelFsm (internalFsmA(), internalFsmB()) {
  whenCompleted {
    goto(stateE)
  }
}
```

关于入口状态的注释

上面定义入口状态的方式使得在复位和第一次时钟采样之间，状态机处于启动状态。只有在第一次时钟采样之后，定义的入口状态才会变为活动状态。这保证了能正确进入入口状态（在 onEntry 中应用语句），并支持嵌套状态机。

虽然它很有用，但也可以绕过该功能，直接让状态机启动到用户状态。

为此，请使用 *makeInstantEntry()* 而不是定义 new State 。该函数返回启动状态，复位后直接激活。

备注： 该状态的 onEntry 仅在从另一个状态转换到该状态时调用，而不是在启动期间。

备注： 在仿真过程中，启动状态始终命名为 BOOT 。

示例：

```
// State sequance: IDLE, STATE_A, STATE_B, ...
val fsm = new StateMachine {
  // IDLE is named BOOT in simulation
  val IDLE = makeInstantEntry()
  val STATE_A, STATE_B, STATE_C = new State

  IDLE.whenIsActive(goto(STATE_A))
  STATE_A.whenIsActive(goto(STATE_B))
  STATE_B.whenIsActive(goto(STATE_C))
  STATE_C.whenIsActive(goto(STATE_B))
}
```

```
//  State sequence : BOOT, IDLE, STATE_A, STATE_B, ...
val fsm = new StateMachine {
  val IDLE, STATE_A, STATE_B, STATE_C = new State
  setEntry(IDLE)

  IDLE.whenIsActive(goto(STATE_A))
  STATE_A.whenIsActive(goto(STATE_B))
  STATE_B.whenIsActive(goto(STATE_C))
  STATE_C.whenIsActive(goto(STATE_B))
}
```

使用状态值的注意事项

在某些情况下，用户想要检索状态机的状态值，可以通过 *stateReg* 访问。然而，状态机在展开过程中实例化，此时 *stateReg* 尚未初始化，因此直接访问 *stateReg* 会导致错误。使用如下的 *postBuild* 方法可以解决这个问题。

```
//  After or inside the fsm's definition.
fsm.postBuild{
  io.status := fsm.stateReg.asBits //io.status is the signal user want to assigned␣
↪to.
}
```

10.6 VexRiscv（RV32IM CPU）

VexRiscv 是一款 fpga 友好的 RISC-V 指令集架构（ISA）的 CPU 实现，具有以下功能：

- RV32IM 指令集
- 分 5 级流水线处理（取指令、解码、执行、内存操作、回写）
- 启用所有功能时其性能为 1.44 DMIPS/Mhz
- 针对 FPGA 进行了优化
- 可选的 MUL/DIV 扩展
- 指令和数据缓存可选
- MMU 可选
- 具有可选的调试扩展，它允许通过 GDB » openOCD » JTAG 连接并进行 Eclipse 调试
- 支持可选中断和异常处理，用于处理机器和用户模式下的中断、异常。（riscv-privileged-v1.9.1 规范规定）。
- 移位指令的两种实现方式，单周期/shiftNumber 周期
- 每个流水线级中都可以有旁路或互锁冒险逻辑
- FreeRTOS 移植版本在这里 https://github.com/Dolu1990/FreeRTOS-RISCV

更多信息在这里：https://github.com/SpinalHDL/VexRiscv

10.7 总线从端生成器

10.7.1 简介

在许多情况下，需要实现总线寄存器组，`BusSlaveFactory` 是一个提供了一种抽象且流畅的方式来定义它们的工具。

要了解该工具的功能，可以通过一个简单的使用 Apb3SlaveFactory 变体来实现内存映射 *UART* 的示例。还有另一个计时器 的示例，其中包含内存映射函数。

您可以在这里 找到有关 `BusSlaveFactory` 工具内部实现的更多文档

10.7.2 功能

有许多 `BusSlaveFactory` 工具的实现，包括后续总线：AHB3-lite、APB3、APB4、AvalonMM、AXI-lite 3、AXI4、BMB、Wishbone 和 PipelinedMemoryBus。
该工具的每个实现都接受相应总线的一个实例作为参数，然后提供以下函数，用于将您的硬件映射到内存映射：

名称	返回类型	描述
busDataWidth	Int	返回总线的数据宽度
read(that,address,bitOffset)		当通过总线读取地址 address 时，用 that 中 bitOffset 位置的数据填充响应
write(that,address,bitOffset)		当通过总线写入地址 address 时，将总线上 bitOffset 位置的数据赋值 that
onWrite(address)(doThat)		当 address 地址上发生写操作（出现写事务）时调用 doThat
onRead(address)(doThat)		当 address 上发生读操作（出现读事务）时调用 doThat
nonStopWrite(that,bitOffset)		将通过总线写入的 bitOffset 的数据赋值到 that
readAndWrite(that,address,bitOffset)		使 that 信号可通过 address 地址读写，并且该信号放置在数据的 bitOffset 位置
readMultiWord(that,address)		创建内存映射以从‘address’读取 that 。 如果 that 的位宽大于一个字（32 位），它将在以下地址上扩展寄存器
writeMultiWord(that,address)		创建内存映射以在‘address’处写入 that 。 如果 that 的位宽大于一个字（32 位），它将在以下地址上扩展寄存器
createWriteOnly(dataType,address,bitOffset)	T	在 address 地址处创建一个 dataType 类型的只写寄存器，并将其放置在字中的 bitOffset 位置
createReadWrite(dataType,address,bitOffset)	T	在 address 处创建一个 dataType 类型的读写寄存器，并将其放置在字中的 bitOffset 位置
createAndDriveFlow(dataType,address,bitOffset)	Flow[T]	在 address 地址处创建一个 dataType 类型的可写流（Flow）寄存器，并将其放置在字中的 bitOffset 位置
drive(that,address,bitOffset)		使用位于 address 地址的可写寄存器中 bitOffset 位置的信号驱动 that
driveAndRead(that,address,bitOffset)		使用位于 address 地址的可读写寄存器中 bitOffset 位置的信号驱动 that
driveFlow(that,address,bitOffset)		当对 address 地址写入时，通过使用位于 bitOffset 位的数据，在 that 流（Flow）上发出事务
readStreamNonBlocking(that, address, validBitOffset, payloadBitOffset)		读取 that 信号并在读取 address 地址时消耗事务。 valid <= validBitOffset bit payload <= payloadBitOffset+widthOf(payload) downto payloadBitOffset
doBitsAccumulationAndClearOnRead(that, address, bitOffset)		实例化一个内部寄存器，该寄存器在每个周期执行以下操作： reg := reg | that 然后，当发生读取时，寄存器被清除。该寄存器可通过 address 地址读取，并放置在字中的 bitOffset 位置

10.8 纤程框架

警告： 该框架设计目标不是用于一般 RTL 生成，而是针对大型系统设计管理和代码生成。它目前在 SaxonSoC 中用作顶级集成工具。

Currently in development.

纤程 (Fiber) 以乱序的方式运行硬件生成，有点类似于 Makefile，您可以在其中定义规则和依赖关系，然后在运行 make 命令时解决这些依赖关系。这与 Scala Future 功能非常相似。

使用这个框架可能会使简单的事情复杂化，但为复杂的情况提供了一些强大的功能：

- 您甚至可以在知道所有要求之前就定义事物，例如：在知道需要多少中断信号线之前实例化中断控制器
- Abstract/lazy/partial SoC architecture definition allowing the creation of SoC template for further specializations
- 以分散方式在多个代理之间自动进行需求协商，例如：内存总线的主设备和从设备之间

该框架主要由以下部分组成：

- `Handle[T]`，稍后可用于存储 `T` 类型的值。
- `handle.load` 允许设置句柄的值（将启动等待它的所有任务）
- `handle.get`，返回给定句柄的值。如果尚未加载该句柄，将阻止任务执行进入等待
- `Handle{ /*code*/ }`，它派生一个新任务来执行给定的代码。该代码的结果将被加载到句柄中
- `soon(handle)`，允许当前任务宣称它将加载一个 `handle` 句柄（用于调度）

10.8.1 简单的示例

这是一个简单的例子：

```
import spinal.core.fiber._

// Create two empty Handles
val a, b = Handle[Int]

// Create a Handle which will be loaded asynchronously by the given body result
val calculator = Handle {
    a.get + b.get // .get will block until they are loaded
}

// Same as above
val printer = Handle {
    println(s"a + b = ${calculator.get}") // .get is blocking until the calculator␣
↪body is done
}

// Synchronously load a and b, this will unblock a.get and b.get
a.load(3)
b.load(4)
```

它的运行步骤会是：

- 创建 a 和 b
- 创建计算器任务分支，但在执行 a.get 时被阻塞
- 创建打印任务分支，但在执行 calculator.get 时被阻塞

- 加载 a 和 b，这会重新调度计算器任务（因为它正在等待 a）
- 计算器执行 a+b 求和操作，并将结果加载到其句柄，这将重新调度打印任务
- 打印任务打印其结果
- 完成所有任务

因此，该示例的要点是表明我们在某种程度上克服了顺序执行，因为 a 和 b 可以在计算器定义之后被加载。

10.8.2 Handle[T]

Handle[T] 有点像 scala 的 Future[T]，它们允许在某个对象存在之前就谈及它，并等待它。

```
val x,y = Handle[Int]
val xPlus2 : Handle[Int] = x.produce(x.get + 2) // x.produce can be used to␣
↪generate a new Handle when x is loaded
val xPlus3 : Handle[Int] = x.derivate(_ + 3)    // x.derivate is as x.produce, but␣
↪also provide the x.get as argument of the lambda function
x.load(3) // x will now contain the value 3
```

soon(handle)

为了维护任务和句柄之间正确的依赖关系图，任务可以预先指明它将加载给定的句柄。在生成饥饿(starvation)/死锁的情况下非常有用，以便 SpinalHDL 准确报告问题所在。

10.9 二进制系统

10.9.1 规范

这里的对象与 HDL 无关，但是它们在数字系统中很常见，尤其是算法参考模型被广泛使用。此外，它们还用于 testbench 的编写。

语法	描述	返回类型
String.asHex	十六进制字符串转为 BigInt == BigInt(string, 16)	BigInt
String.asDec	十进制字符串转为 BigInt == BigInt(string, 10)	BigInt
String.asOct	八进制字符串转为 BigInt == BigInt(string, 8)	BigInt
String.asBin	二进制字符串转为 BigInt == BigInt(string, 2)	BigInt
Byte|Int|Long|BigInt.hexString()	转为十六进制字符串	String
Byte|Int|Long|BigInt.octString()	转为十进制字符串	String
Byte|Int|Long|BigInt.binString()	转为二进制字符串	String
Byte|Int|Long|BigInt.hexString(bitSize)	符号对齐位大小，然后转为十六进制字符串	String
Byte|Int|Long|BigInt.octString(bitSize)	符号对齐位大小，然后转为八进制字符串	String
Byte|Int|Long|BigInt.binString(bitSize)	符号对齐位大小，然后转为二进制字符串	String
Byte|Int|Long|BigInt.toBinInts()	转为二进制列表 (BinaryList)	List[Int]

续下页

表 1 – 接上页

语法	描述	返回类型
Byte|Int|Long|BigInt.toDecInts()	转为十进制列表 (DecimalList)	List[Int]
Byte|Int|Long|BigInt.toOctInts()	转为八进制列表 (OctalList)	List[Int]
Byte|Int|Long|BigInt.toBinInts(num)	转为二进制列表 (BinaryList)，对齐到 num 参数大小并填充 0	List[Int]
Byte|Int|Long|BigInt.toDecInts(num)	转为十进制列表 (DecimalList)，对齐到 num 参数大小并填充 0	List[Int]
Byte|Int|Long|BigInt.toOctInts(num)	转为八进制列表 (OctalList)，对齐到 num 参数大小并填充 0	List[Int]
"3F2A" .hexToBinInts	十六进制字符串转为二进制列表	List[Int]
"3F2A" .hexToBinIntsAlign	十六进制字符串转为二进制列表，并对齐到 4 的倍数大小	List[Int]
List(1,0,1,0,…).binIntsToHex	二进制列表转为十六进制字符串	String
List(1,0,1,0,…).binIntsToOct	二进制列表转为八进制字符串	String
List(1,0,1,0,…).binIntsToHexAlignHigh	二进制列表大小对齐到 4 的倍数（填充 0），然后转为十六进制字符串	String
List(1,0,1,0,…).binIntsToOctAlignHigh	二进制列表大小对齐到 3 的倍数（填充 0），然后转为十六进制字符串	String
List(1,0,1,0,…).binIntsToInt	二进制列表 (最大数目为 32) 转为 Int	Int
List(1,0,1,0,…).binIntsToLong	二进制列表 (最大数目为 64) 转为 Long	Long
List(1,0,1,0,…).binIntsToBigInt	二进制列表（数目无限制）转为 BigInt	BigInt
Int.toBigInt	32.toBigInt == BigInt(32)	BigInt
Long.toBigInt	3233113232L.toBigInt == BigInt(3233113232L)	BigInt
Byte.toBigInt	8.toByte.toBigInt == BigInt(8.toByte)	BigInt

10.9.2 String 转为 Int/Long/BigInt

```
import spinal.core.lib._

$: "32FF190".asHex

$: "12384798999999".asDec

$: "123456777777700".asOct

$: "10100011100111111".asBin
```

10.9.3 Int/Long/BigInt 转为 String

```
import spinal.core.lib._

$: "32FF190".asHex.hexString()
"32FF190"
$: "123456777777700".asOct.octString()
"123456777777700"
$: "10100011100111111".asBin.binString()
"10100011100111111"
$: 32323239988L.hexString()
7869d8034
$: 3239988L.octString()
14270064
```

(续下页)

(接上页)

```
$: 34.binString()
100010
```

10.9.4 Int/Long/BigInt 转为二进制列表

```
import spinal.core.lib._

$: 32.toBinInts
List(0, 0, 0, 0, 0, 1)
$: 1302309988L.toBinInts
List(0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0,␣
↪1, 1, 0, 0, 1)
$: BigInt("100101110", 2).toBinInts
List(0, 1, 1, 1, 0, 1, 0, 0, 1)
$: BigInt("123456789abcdef0", 16).toBinInts
List(0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1,␣
↪0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1,␣
↪1, 0, 0, 0, 1, 0, 0, 1)
$: BigInt("1234567", 8).toBinInts
List(1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1)
$: BigInt("123451118", 10).toBinInts
List(0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1,␣
↪1)
```

对齐到固定位宽

```
import spinal.core.lib._

$: 39.toBinInts()
List(1, 1, 1, 0, 0, 1)
$: 39.toBinInts(8)    // align to 8 bit zero filled at MSB
List(1, 1, 1, 0, 0, 1, 0, 0)
```

10.9.5 二进制列表转为 Int/Long/BigInt

```
import spinal.core.lib._

$: List(1, 1, 1, 0, 0, 1).binIntsToInt
39
$: List(1, 1, 1, 0:, 0, 1).binIntsToLong
39
$: List(0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1,␣
↪0, 1, 1, 0, 0, 1).binIntsToBigInt
1302309988
```

```
$: List(1, 1, 1, 0, 0, 1).binIntsToHex
27
$: List(1, 1, 1, 0, 0, 1).binIntsToHexAlignHigh
9c
$: List(1, 1, 1, 0, 0, 1).binIntsToOct
47
$: List(1, 1, 1, 0, 0, 1).binIntsToHexAlignHigh
47
```

10.9.6 BigInt 放大器

```
$: 32.toBigInt
32
$: 3211323244L.toBigInt
3211323244
$: 8.toByte.toBigInt
8
```

10.10 RegIf

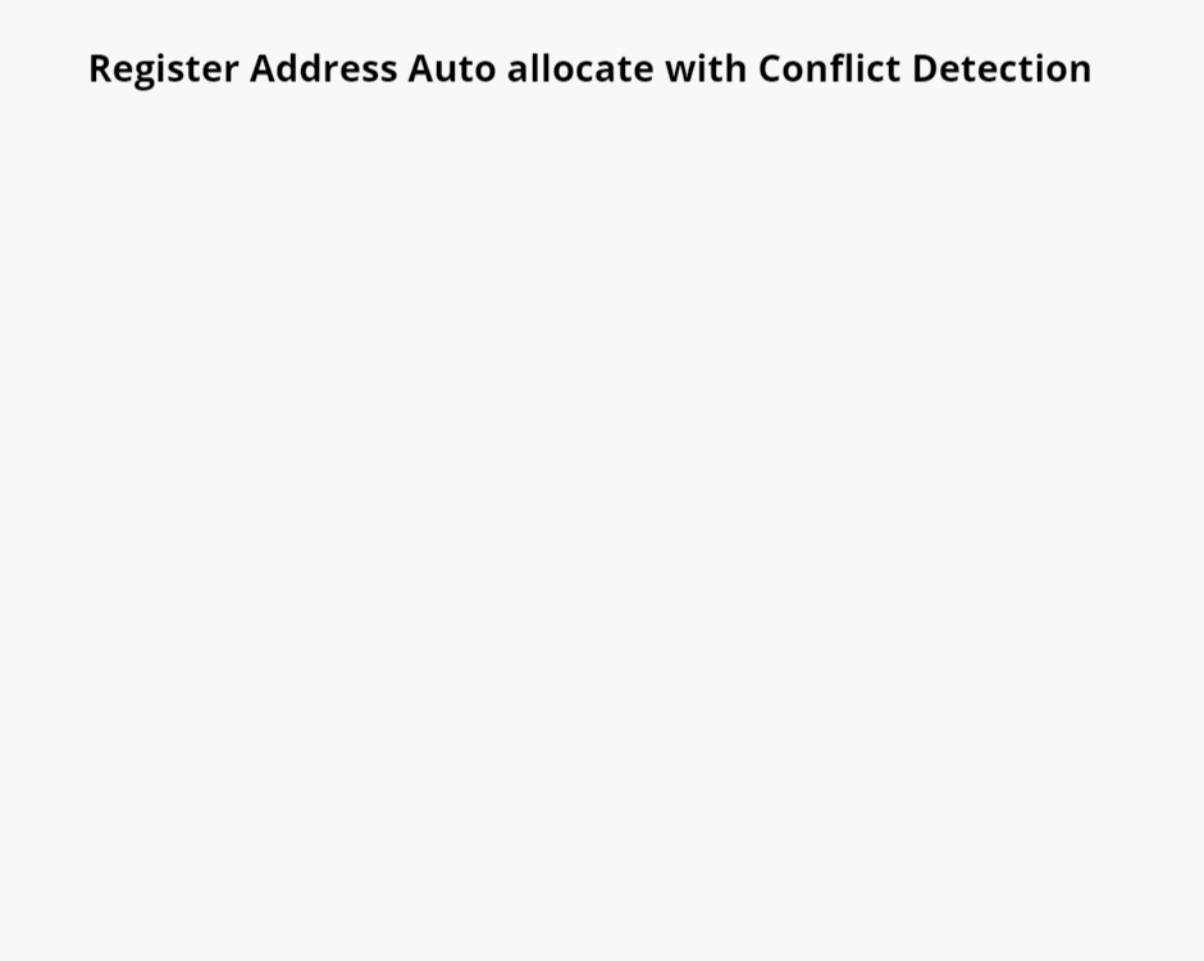
寄存器接口搭建器

- 自动寻址、字段分配和冲突检测
- 28 种寄存器访问类型（涵盖 UVM 标准定义的 25 种类型）
- 自动生成文档

10.10.1 自动分配

自动地址分配

```
class RegBankExample extends Component {
  val io = new Bundle {
    apb = slave(Apb3(Apb3Config(16,32)))
  }
  val busif = Apb3BusInterface(io.apb,(0x0000, 100 Byte))
  val M_REG0  = busif.newReg(doc="REG0")
  val M_REG1  = busif.newReg(doc="REG1")
  val M_REG2  = busif.newReg(doc="REG2")

  val M_REGn  = busif.newRegAt(address=0x40, doc="REGn")
  val M_REGn1 = busif.newReg(doc="REGn1")

  busif.accept(HtmlGenerator("regif", "AP"))
  // busif.accept(CHeaderGenerator("header", "AP"))
  // busif.accept(JsonGenerator("regif"))
  // busif.accept(RalfGenerator("regbank"))
  // busif.accept(SystemRdlGenerator("regif", "AP"))
}
```



Automatic filed allocation

```
val M_REG0  = busif.newReg(doc="REG1")
val fd0 = M_REG0.field(Bits(2 bit), RW, doc= "fields 0")
M_REG0.reserved(5 bits)
val fd1 = M_REG0.field(Bits(3 bit), RW, doc= "fields 0")
val fd2 = M_REG0.field(Bits(3 bit), RW, doc= "fields 0")
// auto reserved 2 bits
val fd3 = M_REG0.fieldAt(pos=16, Bits(4 bit), doc= "fields 3")
// auto reserved 12 bits
```

冲突检测

```
val M_REG1  = busif.newReg(doc="REG1")
val r1fd0 = M_REG1.field(Bits(16 bits), RW, doc="fields 1")
val r1fd2 = M_REG1.field(Bits(18 bits), RW, doc="fields 1")
  ...
cause Exception
val M_REG1  = busif.newReg(doc="REG1")
val r1fd0 = M_REG1.field(Bits(16 bits), RW, doc="fields 1")
val r1fd2 = M_REG1.fieldAt(pos=10, Bits(2 bits), RW, doc="fields 1")
  ...
cause Exception
```

10.10.2 28 种访问类型

其中大部分来自 UVM 规范

访问类型	描述	来源
RO	w：无影响，r：无影响	UVM
RW	w：保持原样，r：无影响	UVM
RC	w：无影响，r：清除所有比特	UVM
RS	w：无影响，r：置位所有比特	UVM
WRC	w：保持原样，r：清除所有比特	UVM
WRS	w：保持原样，r：置位所有比特	UVM
WC	w：清除所有比特，r：无影响	UVM

续下页

表 2 – 接上页

访问类型	描述	来源
WS	w：置位所有比特，r：无影响	UVM
WSRC	w：置位所有比特，r：清除所有比特	UVM
WCRS	w：清除所有比特，r：置位所有比特	UVM
W1C	w：1/0 对匹配位清除/无影响，r：无影响	UVM
W1S	w：1/0 对匹配位置位/无影响，r：无影响	UVM
W1T	w：1/0 对匹配位翻转/无影响，r：无影响	UVM
W0C	w：1/0 对匹配位清除/无影响，r：无影响	UVM
W0S	w：1/0 对匹配位置位/无影响，r：无影响	UVM
W0T	w：1/0 对匹配位翻转/无影响，r：无影响	UVM
W1SRC	w：1/0 对匹配位置位/无影响，r：清除所有比特	UVM
W1CRS	w：1/0 对匹配位清除/无影响，r：置位所有比特	UVM
W0SRC	w：1/0 对匹配位置位/无影响，r：清除所有比特	UVM
W0CRS	w：1/0 对匹配位清除/无影响，r：置位所有比特	UVM
WO	w：保持原样，r：错误	UVM
WOC	w：清除所有比特，r：错误	UVM
WOS	w：置位所有比特，r：错误	UVM
W1	w：硬复位后第一个 w 保持原样，其他 w 无影响，r：无影响	UVM
WO1	w：硬复位后第一个 w 保持原样，其他 w 无影响，r：错误	UVM
NA	w：保留，r：保留	新的
W1P	w：1/0 对匹配位脉冲 (pulse)/无影响，r：无影响	新的
W0P	w：0/1 对匹配位脉冲 (pulse)/无影响，r：无影响	新的
HSRW	w：硬件置位，软件 RW	新的
RWHS	w：软件 RW、硬件置位	新的
ROV	w：只读值，用于硬件版本	新的
CSTM	w：用户自定义类型，用于文档	新的

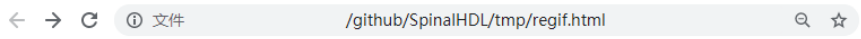
10.10.3 自动生成文档

文档类型

文档类型	用法	状态
HTML	`busif.accept(HtmlGenerator("regif", title = "XXX register file"))`	Y
CHeader	`busif.accept(CHeaderGenerator("header", "AP"))`	Y
JSON	`busif.accept(JsonGenerator("regif"))`	Y
RALF(UVM)	`busif.accept(RalfGenerator("header"))`	Y
System-RDL	`busif.accept(SystemRdlGenerator("regif", "addrmap_name", Some("name"), Some("desc")))`	Y
La-tex(pdf)		N
docx		N

HTM 自动文档现已完成，源代码示例：

生成的 HTML 文档：



TurboRegBank Interface Document

AddressOffset	RegName	Description	Width	Section	FieldName	R/W	Reset value	Field-Description
0x0	M_TURBO_EARLY_QUIT	Turbo Hardware-mode register 1	64	[63:3]	--	NA	0x0	Reserved
				[2:1]	early_count	RW	0x2	CRC validate early quit enable
				[0]	early_quit	RW	0x0	CRC validate early quit enable
0x8	M_TURBO_THETA	Turbo Hardware-mode register 2	64	[63:5]	--	NA	0x0	Reserved
				[4]	bib_order	RW	0x1	bib in Byte, defalut. 0:[76543210] 1:[01234567]
				[3:0]	theta	RW	0x6	Back-Weight, UQ(4,3), default 0.75
0x10	M_TURBO_START	Turbo Start register	64	[63:1]	--	NA	0x0	Reserved
				[0]	turbo_start	W1P	0x0	turbo start pulse
0x18	M_TURBO_MOD	Turbo Mode	64	[63:1]	--	NA	0x0	Reserved
				[0]	umts_on	RW	0x1	1: umts mode 0: emtc mode
0x20	M_TURBO_CRC_POLY	Turbo CRC Poly	64	[63:25]	--	NA	0x0	Reserved
				[24:1]	crc_poly	RW	0x864cfb	(D24+D23+D18+D17+D14+D11+D10+D7+D6+D5+D4+D3+D+1)
				[0]	crc_mode	RW	0x1	0: CRC24; 1: CRC16
0x28	M_TURBO_K	Turbo block size	64	[63:12]	--	NA	0x0	Reserved
				[11:0]	K	RW	0x20	decode block size max: 4032
0x30	M_TURBO_F1F2	Turbo Interleave Parameter	64	[63:25]	--	NA	0x0	Reserved
				[24:16]	f2	RW	0x2f	turbo interleave parameter f2
				[15:9]	--	NA	0x0	Reserved
				[8:0]	f1	RW	0x2f	turbo interleave parameter f1
0x38	M_TURBO_MAX_ITER	Turbo Max Iter Times	64	[63:6]	--	NA	0x0	Reserved
				[5:0]	max_iter	RW	0x0	Max iter times 1~63 avaliable
0x40	M_TURBO_FILL_NUM	Turbo block-head fill number	64	[63:6]	--	NA	0x0	Reserved
				[5:0]	fill_num	RW	0x0	0~63 avaliable, Head fill Number
0x48	M_TURBO_3G_INTER_PV	Turbo UMTS Interleave Parameter P,V	64	[63:21]	--	NA	0x0	Reserved
				[20:16]	v	RW	0x0	Primitive root v
				[15:9]	--	NA	0x0	Reserved
				[8:0]	p	RW	0x0	parameter of prime
0x50	M_TURBO_3G_INTER_CRP	Turbo UMTS Interleave Parameter C,R,p	64	[63:17]	--	NA	0x0	Reserved
				[16:8]	C	RW	0x7	interlave Max Column Number
				[7:5]	--	NA	0x0	Reserved
				[4]	KeqRxC	RW	0x0	1:K=R*C else 0
				[3:2]	CPtype	RW	0x0	CP relation 0: C=P-1 1: C=p 2: C=p+1
				[1:0]	Rtype	RW	0x0	inter Row number 0:5,1:10,2:20,3:20other
0x58	M_TURBO_3G_INTER_FILL	Turbo UMTS Interleave Fill number	64	[63:18]	--	NA	0x0	Reserved
				[17:16]	fillRow	RW	0x0	interlave fill Row number, 0~2 avaliable n+1 row
				[15:9]	--	NA	0x0	Reserved
				[8:0]	fillPos	RW	0x0	interlave start Column of fill Number

Powered By SpinalHDL

10.10.4 特殊访问用途

案例 1： RO 用法

RO 与其他类型不同。它不创建寄存器，需要外部信号来驱动它，注意，请不要忘记驱动它。

```
val io = new Bundle {
  val cnt = in UInt(8 bit)
}

val counter = M_REG0.field(UInt(8 bit), RO, 0, "counter")
counter :=  io.cnt
```

```
val xxstate = M_REG0.field(UInt(8 bit), RO, 0, "xx-ctrl state").asInput
```

```
val overflow = M_REG0.field(Bits(32 bit), RO, 0, "xx-ip parameter")
val ovfreg = Reg(32 bit)
overflow := ovfreg
```

```
val inc     = in Bool()
val counter = M_REG0.field(UInt(8 bit), RO, 0, "counter")
val cnt = Counter(100,  inc)
counter := cnt
```

案例 2： ROV 用法

ASIC 设计常常需要一些固化的版本信息。与 RO 不同，它不会产生有线信号

旧方法：

```
val version = M_REG0.field(Bits(32 bit), RO, 0, "xx-device version")
version := BigInt("F000A801", 16)
```

新方法：

```
M_REG0.field(Bits(32 bit), ROV, BigInt("F000A801", 16), "xx-device version
↪")(Symbol("Version"))
```

案例 3: HSRW/RWHS 在某些情况下的硬件设置类型，此类寄存器不仅可以由软件配置，还可以由硬件信号设置

```
val io = new Bundle {
  val xxx_set = in Bool()
  val xxx_set_val = in Bits(32 bit)
}

val reg0 = M_REG0.fieldHSRW(io.xxx_set, io.xxx_set_val, 0, "xx-device version")  //
↪ 0x0000
val reg1 = M_REG1.fieldRWHS(io.xxx_set, io.xxx_set_val, 0, "xx-device version")  //
↪ 0x0004
```

```
always @(posedge clk or negedge rstn)
  if(!rstn) begin
     reg0  <= '0;
     reg0  <= '0;
  end else begin
     if(hit_0x0000) begin
        reg0 <= wdata ;
     end
     if(io.xxx_set) begin      // HW have High priority than SW
        reg0 <= io.xxx_set_val ;
```

(续下页)

(接上页)

```
    end

    if(io.xxx_set) begin
       reg1 <= io.xxx_set_val ;
    end
    if(hit_0x0004) begin      // SW have High priority than HW
       reg1 <= wdata ;
    end
  end
```

案例 4: CSTM 虽然 SpinalHDL 包含 25 种寄存器类型和 6 种扩展类型，但在实际应用中仍然对私有寄存器类型有各种需求。因此，我们保留 CSTM 类型以实现可扩展性。CSTM 仅用于生成软件接口，不生成实际电路

```
val reg = Reg(Bits(16 bit)) init 0
REG.registerAtOnlyReadLogic(0, reg, CSTM("BMRW"), resetValue = 0, "custom field")

when(busif.dowrite) {
   reg :=  reg & ~busif.writeData(31 downto 16) |  busif.writeData(15 downto 0) &␣
↪busif.writeData(31 downto 16)
}
```

案例 5: parasiteField

这用于软件在多个地址上共享同一寄存器，而不是生成多个寄存器实体

示例 1：时钟门软件使能

```
val M_CG_ENS_SET = busif.newReg(doc="Clock Gate Enables")  // x00000
val M_CG_ENS_CLR = busif.newReg(doc="Clock Gate Enables")  // 0x0004
val M_CG_ENS_RO  = busif.newReg(doc="Clock Gate Enables")  // 0x0008

val xx_sys_cg_en = M_CG_ENS_SET.field(Bits(4 bit), W1S, 0, "clock gate enables,␣
↪write 1 set" )
                   M_CG_ENS_CLR.parasiteField(xx_sys_cg_en, W1C, 0, "clock gate␣
↪enables, write 1 clear" )
                   M_CG_ENS_RO.parasiteField(xx_sys_cg_en, RO, 0, "clock gate␣
↪enables, read only"
```

example2: interrupt raw reg with force interface for software

```
val RAW    = this.newRegAt(offset,"Interrupt Raw status Register\n set when event \
↪n clear raw when write 1")
val FORCE  = this.newReg("Interrupt Force  Register\n for SW debug use \n write 1␣
↪set raw")

val raw    = RAW.field(Bool(), AccessType.W1C,     resetValue = 0, doc = s"raw,␣
↪default 0" )
             FORCE.parasiteField(raw, AccessType.W1S,    resetValue = 0, doc = s
↪"force, write 1 set, debug use" )
```

案例 6: SpinalEnum

当字段类型为 SpinalEnum 时，resetValue 指定枚举元素的索引。

```
object UartCtrlTxState extends SpinalEnum(defaultEncoding = binaryOneHot) {
   val sIdle, sStart, sData, sParity, sStop = newElement()
}

val raw = M_REG2.field(UartCtrlTxState(), AccessType.RW, resetValue = 2, doc="state
↪")
// raw will be init to sData
```

10.10.5 字节掩码

withStrb

10.10.6 典型例子

批量创建 REG-Address 和字段寄存器

```
import spinal.lib.bus.regif._

class RegBank extends Component {
  val io = new Bundle {
    val apb = slave(Apb3(Apb3Config(16, 32)))
    val stats = in Vec(Bits(16 bit), 10)
    val IQ  = out Vec(Bits(16 bit), 10)
  }
  val busif = Apb3BusInterface(io.apb, (0x000, 100 Byte), regPre = "AP")

  (0 to 9).map { i =>
    // here use setName give REG uniq name for Docs usage
    val REG = busif.newReg(doc = s"Register${i}").setName(s"REG${i}")
    val real = REG.field(SInt(8 bit), AccessType.RW, 0, "Complex real")
    val imag = REG.field(SInt(8 bit), AccessType.RW, 0, "Complex imag")
    val stat = REG.field(Bits(16 bit), AccessType.RO, 0, "Accelerator status")
    io.IQ(i)( 7 downto 0) := real.asBits
    io.IQ(i)(15 downto 8) := imag.asBits
    stat := io.stats(i)
  }

  def genDocs() = {
    busif.accept(CHeaderGenerator("regbank", "AP"))
    busif.accept(HtmlGenerator("regbank", "Interupt Example"))
    busif.accept(JsonGenerator("regbank"))
    busif.accept(RalfGenerator("regbank"))
    busif.accept(SystemRdlGenerator("regbank", "AP"))
  }

  this.genDocs()
}

SpinalVerilog(new RegBank())
```

10.10.7 中断生成器

手动写中断

```
class cpInterruptExample extends Component {
   val io = new Bundle {
     val tx_done, rx_done, frame_end = in Bool()
     val interrupt = out Bool()
     val apb = slave(Apb3(Apb3Config(16, 32)))
   }
   val busif = Apb3BusInterface(io.apb, (0x000, 100 Byte), regPre = "AP")
   val M_CP_INT_RAW    = busif.newReg(doc="cp int raw register")
   val tx_int_raw      = M_CP_INT_RAW.field(Bool(), W1C, doc="tx interrupt enable␣
↪register")
   val rx_int_raw      = M_CP_INT_RAW.field(Bool(), W1C, doc="rx interrupt enable␣
↪register")
   val frame_int_raw   = M_CP_INT_RAW.field(Bool(), W1C, doc="frame interrupt␣
```

(续下页)

(接上页)

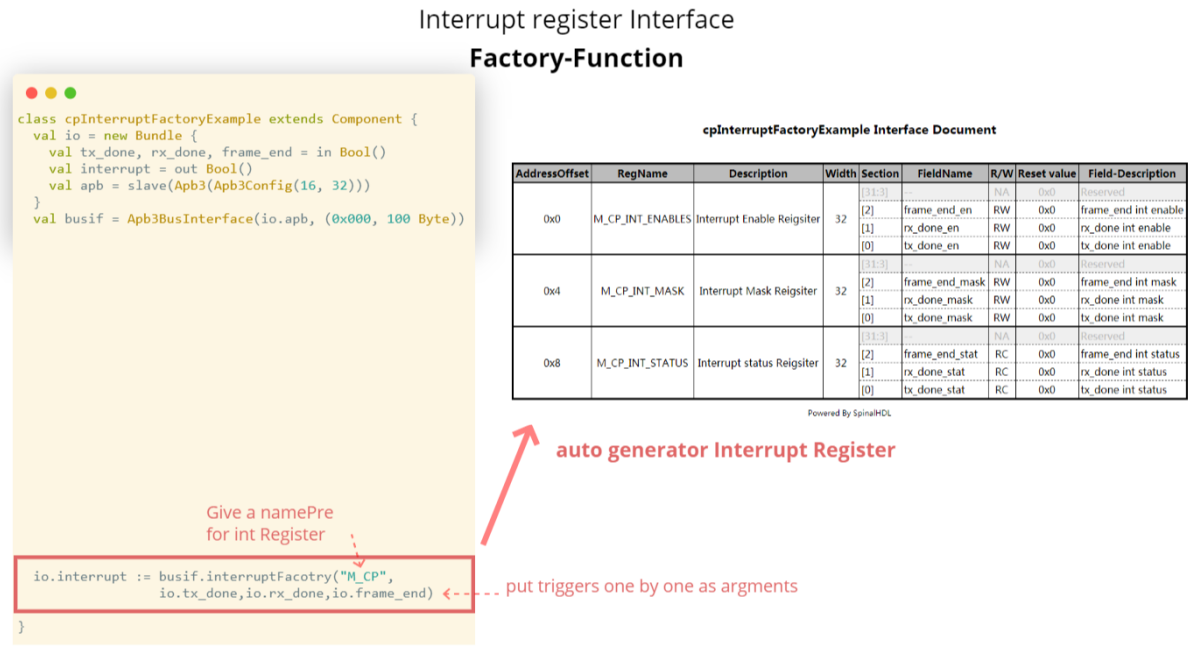
```
↪enable register")

  val M_CP_INT_FORCE = busif.newReg(doc="cp int force register\n for debug use")
  val tx_int_force     = M_CP_INT_FORCE.field(Bool(), RW, doc="tx interrupt␣
↪enable register")
  val rx_int_force     = M_CP_INT_FORCE.field(Bool(), RW, doc="rx interrupt␣
↪enable register")
  val frame_int_force  = M_CP_INT_FORCE.field(Bool(), RW, doc="frame interrupt␣
↪enable register")

  val M_CP_INT_MASK    = busif.newReg(doc="cp int mask register")
  val tx_int_mask      = M_CP_INT_MASK.field(Bool(), RW, doc="tx interrupt mask␣
↪register")
  val rx_int_mask      = M_CP_INT_MASK.field(Bool(), RW, doc="rx interrupt mask␣
↪register")
  val frame_int_mask   = M_CP_INT_MASK.field(Bool(), RW, doc="frame interrupt␣
↪mask register")

  val M_CP_INT_STATUS   = busif.newReg(doc="cp int state register")
  val tx_int_status      = M_CP_INT_STATUS.field(Bool(), RO, doc="tx interrupt␣
↪state register")
  val rx_int_status      = M_CP_INT_STATUS.field(Bool(), RO, doc="rx interrupt␣
↪state register")
  val frame_int_status   = M_CP_INT_STATUS.field(Bool(), RO, doc="frame interrupt␣
↪state register")

  rx_int_raw.setWhen(io.rx_done)
  tx_int_raw.setWhen(io.tx_done)
  frame_int_raw.setWhen(io.frame_end)

  rx_int_status := (rx_int_raw || rx_int_force) && (!rx_int_mask)
  tx_int_status := (tx_int_raw || rx_int_force) && (!rx_int_mask)
  frame_int_status := (frame_int_raw || frame_int_force) && (!frame_int_mask)

  io.interrupt := rx_int_status || tx_int_status || frame_int_status

}
```

这是一项非常繁琐且重复的工作，更好的方法是使用“生成器 (factory)”范例来自动生成每个信号的文档。

现在 InterruptFactory 可以做到这一点。

创建中断的简单方法：

```
class EasyInterrupt extends Component {
  val io = new Bundle {
    val apb = slave(Apb3(Apb3Config(16,32)))
    val a, b, c, d, e = in Bool()
  }

  val busif = BusInterface(io.apb,(0x000,1 KiB), 0, regPre = "AP")

  busif.interruptFactory("T", io.a, io.b, io.c, io.d, io.e)

  busif.accept(CHeaderGenerator("intrreg","AP"))
  busif.accept(HtmlGenerator("intrreg", "Interupt Example"))
  busif.accept(JsonGenerator("intrreg"))
  busif.accept(RalfGenerator("intrreg"))
  busif.accept(SystemRdlGenerator("intrreg", "AP"))
}
```

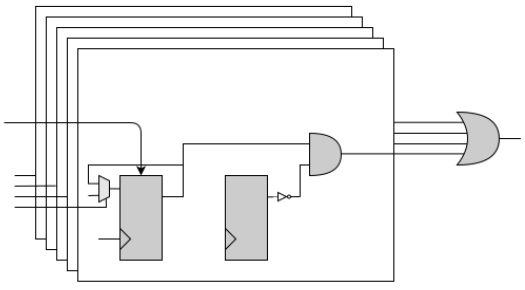


AddressOffset	RegName	Description	Width	Section	FieldName	R/W	Reset value	Field-Description
0x0	M_CP_INT_ENABLES	Interrupt Enable Reigsiter	32	[31:3]	-	NA	0x0	Reserved
				[2]	frame_end_en	RW	0x0	frame_end int enable
				[1]	rx_done_en	RW	0x0	rx_done int enable
				[0]	tx_done_en	RW	0x0	tx_done int enable
0x4	M_CP_INT_MASK	Interrupt Mask Reigsiter	32	[31:3]	-	NA	0x0	Reserved
				[2]	frame_end_mask	RW	0x0	frame_end int mask
				[1]	rx_done_mask	RW	0x0	rx_done int mask
				[0]	tx_done_mask	RW	0x0	tx_done int mask
0x8	M_CP_INT_STATUS	Interrupt status Reigsiter	32	[31:3]	-	NA	0x0	Reserved
				[2]	frame_end_stat	RC	0x0	frame_end int status
				[1]	rx_done_stat	RC	0x0	rx_done int status
				[0]	tx_done_stat	RC	0x0	tx_done int status

Powered By SpinalHDL

IP 级中断生成器

寄存器	访问类型	描述
RAW	W1C	中断原始状态 (int raw) 寄存器，由 int 事件设置，总线写 1 时清零
FORCE	RW	中断强制寄存器，用于软件调试
MASK	RW	int mask register, 1: off; 0: open; default 1 int off
STATUS	RO	中断状态，只读，`status = raw && ! mask`

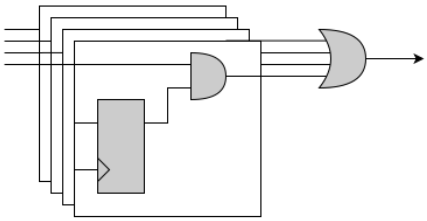


Spinal 用法：

```
busif.interruptFactory("T", io.a, io.b, io.c, io.d, io.e)
```

SYS 级中断合并

寄存器	访问类型	描述
MASK	RW	int mask register, 1: off; 0: open; default 1 int off
STATUS	RO	中断状态，RO，`status = int_level && ! mask`



Spinal 用法：

```
busif.interruptLevelFactory("T", sys_int0, sys_int1)
```

Spinal 的生成器

总线接口方法	描述
`InterruptFactory(regNamePre: String, triggers: Bool*)`	为脉冲事件创建 RAW/FORCE/MASK/STATUS
`InterruptFactoryNoForce(regNamePre: String, triggers: Bool*)`	为脉冲事件创建 RAW/MASK/STATUS
`InterruptLevelFactory(regNamePre: String, triggers: Bool*)`	为 level_int 合并创建 MASK/STATUS
`InterruptFactoryAt(addrOffset: Int, regNamePre: String, triggers: Bool*)`	在 addrOffset 处为脉冲事件创建 RAW/FORCE/MASK/STATUS
`InterruptFactoryNoForceAt(addrOffset: Int, regNamePre: String, triggers: Bool*)`	在 addrOffset 处为脉冲事件创建 RAW/MASK/STATUS
`InterruptFactoryAt(addrOffset: Int, regNamePre: String, triggers: Bool*)`	在 addrOffset 处为 level_int 合并创建 MASK/STATUS
`interrupt_W1SCmask_FactoryAt(addrOffset: BigInt, regNamePre: String, triggers: Bool*)`	create RAW/FORCE/MASK(SET/CLR)/STATUS for pulse event at addrOffset
`interruptLevel_W1SCmask_FactoryAt(addrOffset: BigInt, regNamePre: String, levels: Bool*)`	create RAW/FORCE/MASK(SET/CLR)/STATUS for level event at addrOffset

示例

```
class RegFileIntrExample extends Component {
   val io = new Bundle {
     val apb = slave(Apb3(Apb3Config(16,32)))
     val int_pulse0, int_pulse1, int_pulse2, int_pulse3 = in Bool()
     val int_level0, int_level1, int_level2 = in Bool()
     val sys_int = out Bool()
     val gpio_int = out Bool()
   }

   val busif = BusInterface(io.apb,  (0x000,1 KiB), 0, regPre = "AP")
```

(续下页)

(接上页)

```
  io.sys_int  := busif.interruptFactory("SYS",io.int_pulse0, io.int_pulse1, io.
↪int_pulse2, io.int_pulse3)
  io.gpio_int := busif.interruptLevelFactory("GPIO",io.int_level0, io.int_level1,␣
↪io.int_level2, io.sys_int)

  def genDoc() = {
    busif.accept(CHeaderGenerator("intrreg","Intr"))
    busif.accept(HtmlGenerator("intrreg", "Interrupt Example"))
    busif.accept(JsonGenerator("intrreg"))
    busif.accept(RalfGenerator("intrreg"))
    busif.accept(SystemRdlGenerator("intrreg", "Intr"))
    this
  }

  this.genDoc()
}
```

Interupt Example register interface

AddressOffset	RegName	Description	Width	Section	FieldName	R/W	Reset value	Field-Description
0x0	SYS_INT_RAW	Interrupt Raw status Register set when event clear when write 1	32	[31:4]	--	NA	28'b0	Reserved
				[3]	int_pulse3_raw	W1C	1'b0	raw, default 0
				[2]	int_pulse2_raw	W1C	1'b0	raw, default 0
				[1]	int_pulse1_raw	W1C	1'b0	raw, default 0
				[0]	int_pulse0_raw	W1C	1'b0	raw, default 0
0x4	SYS_INT_FORCE	Interrupt Force Register for SW debug use	32	[31:4]	--	NA	28'b0	Reserved
				[3]	int_pulse3_force	RW	1'b0	force, default 0
				[2]	int_pulse2_force	RW	1'b0	force, default 0
				[1]	int_pulse1_force	RW	1'b0	force, default 0
				[0]	int_pulse0_force	RW	1'b0	force, default 0
0x8	SYS_INT_MASK	Interrupt Mask Register 1: int off 0: int open default 1, int off	32	[31:4]	--	NA	28'b0	Reserved
				[3]	int_pulse3_mask	RW	1'h1	mask, default 1, int off
				[2]	int_pulse2_mask	RW	1'h1	mask, default 1, int off
				[1]	int_pulse1_mask	RW	1'h1	mask, default 1, int off
				[0]	int_pulse0_mask	RW	1'h1	mask, default 1, int off
0xc	SYS_INT_STATUS	Interrupt status Register status = (raw || force) && (!mask)	32	[31:4]	--	NA	28'b0	Reserved
				[3]	int_pulse3_status	RO	1'b0	stauts default 0
				[2]	int_pulse2_status	RO	1'b0	stauts default 0
				[1]	int_pulse1_status	RO	1'b0	stauts default 0
				[0]	int_pulse0_status	RO	1'b0	stauts default 0
0x10	GPIO_INT_MASK	Interrupt Mask Register 1: int off 0: int open default 1, int off	32	[31:4]	--	NA	28'b0	Reserved
				[3]	sys_int_mask	RW	1'h1	mask
				[2]	int_level2_mask	RW	1'h1	mask
				[1]	int_level1_mask	RW	1'h1	mask
				[0]	int_level0_mask	RW	1'h1	mask
0x14	GPIO_INT_STATUS	Interrupt status Register status = int_level && (!mask)	32	[31:4]	--	NA	28'b0	Reserved
				[3]	sys_int_status	RO	1'b0	stauts
				[2]	int_level2_status	RO	1'b0	stauts
				[1]	int_level1_status	RO	1'b0	stauts
				[0]	int_level0_status	RO	1'b0	stauts

Powered by SpinalHDL

Wed Apr 13 01:16:53 CST 2022

10.10.8 默认读取值

当软件读取保留地址时，当前的策略是正常返回，readerror=0。为了方便软件调试，可以配置回读值，默认为 0

```
busif.setReservedAddressReadValue(0x0000EF00)
```

```
default: begin
   busif_rdata  <= 32'h0000EF00 ;
   busif_rderr  <= 1'b0         ;
end
```

10.10.9 开发者区域

You can add your document Type by extending the *BusIfVisitor* Trait

```
case class Latex(fileName : String) extends BusIfVisitor{ ... }
```

BusIfVisitor give access BusIf.RegInsts to do what you want

```
// lib/src/main/scala/spinal/lib/bus/regif/BusIfBase.scala

trait BusIfVisitor {
  def begin(busDataWidth : Int) : Unit
  def visit(descr : FifoDescr)  : Unit
  def visit(descr : RegDescr)   : Unit
  def end()                     : Unit
}
```

10.11 总线

10.11.1 AHB-Lite3

Configuration and instantiation

首先，每当您想要创建 AHB-Lite3 总线时，您都需要一个配置对象。该配置对象是一个 `AhbLite3Config` 并具有以下参数：

参数名称	类型	默认值	描述
addressWidth	Int		HADDR 的位宽（字节粒度）
dataWidth	Int		HWDATA 和 HRDATA 的位宽

简而言之，AHB-Lite3 总线在 SpinalHDL 库中是如下定义的：

```
case class AhbLite3(config: AhbLite3Config) extends Bundle with IMasterSlave {
  //  Address and control
  val HADDR = UInt(config.addressWidth bits)
  val HSEL = Bool()
  val HREADY = Bool()
  val HWRITE = Bool()
  val HSIZE = Bits(3 bits)
  val HBURST = Bits(3 bits)
  val HPROT = Bits(4 bits)
  val HTRANS = Bits(2 bits)
  val HMASTLOCK = Bool()
```

(续下页)

(接上页)

```
  //  Data
  val HWDATA = Bits(config.dataWidth bits)
  val HRDATA = Bits(config.dataWidth bits)

  //  Transfer response
  val HREADYOUT = Bool()
  val HRESP = Bool()

  override def asMaster(): Unit = {
    out(HADDR,HWRITE,HSIZE,HBURST,HPROT,HTRANS,HMASTLOCK,HWDATA,HREADY,HSEL)
    in(HREADYOUT,HRESP,HRDATA)
  }
}
```

这是一个简单的使用示例：

```
val ahbConfig = AhbLite3Config(
  addressWidth = 12,
  dataWidth    = 32
)
val ahbX = AhbLite3(ahbConfig)
val ahbY = AhbLite3(ahbConfig)

when(ahbY.HSEL) {
  // ...
}
```

变体

有一个 AhbLite3Master 变体，唯一的区别是缺少 HREADYOUT 信号。当互连线和从端使用 AhbLite3 时，此变体只能由主端使用。

10.11.2 Apb3

AMBA3-APB 总线通常用于连接低带宽外设。

Configuration and instantiation

首先，每当您想要创建 APB3 总线时，您都需要一个配置对象。该配置对象是一个 Apb3Config 并具有以下参数：

参数名称	类型	默认值	描述
addressWidth	Int		PADDR 的位宽（字节粒度）
dataWidth	Int		PWDATA 和 PRDATA 的位宽
selWidth	Int	1	PSEL 的位宽
useSlaveError	Boolean	false	指定是否出现 PSLVERROR

简而言之，APB3 总线在 SpinalHDL 库中定义方式如下：

```
case class Apb3(config: Apb3Config) extends Bundle with IMasterSlave {
  val PADDR   = UInt(config.addressWidth bits)
  val PSEL    = Bits(config.selWidth bits)
  val PENABLE = Bool()
  val PREADY  = Bool()
  val PWRITE  = Bool()
```

(续下页)

(接上页)

```
  val PWDATA      = Bits(config.dataWidth bits)
  val PRDATA      = Bits(config.dataWidth bits)
  val PSLVERROR   = if(config.useSlaveError) Bool() else null
  // ...
}
```

这是一个简单的使用示例：

```
val apbConfig = Apb3Config(
  addressWidth = 12,
  dataWidth    = 32
)
val apbX = Apb3(apbConfig)
val apbY = Apb3(apbConfig)

when(apbY.PENABLE) {
  // ...
}
```

函数和运算符

名称	返回类型	描述
X » Y		将 X 连接到 Y。Y 的地址可以小于 X 的地址
X « Y		执行 » 运算符相反的操作

10.11.3 Axi4

AXI4 是 ARM 定义的高带宽总线。

Configuration and instantiation

首先，每当您想要创建 AXI4 总线时，您都需要一个配置对象。该配置对象是一个 `Axi4Config` 并具有以下参数：

注意：useXXX 用于指定总线是否存在 XXX 信号。

参数名称	类型	默认值
addressWidth	Int	
dataWidth	Int	
idWidth	Int	
userWidth	Int	
useId	Boolean	true
useRegion	Boolean	true
useBurst	Boolean	true
useLock	Boolean	true
useCache	Boolean	true
useSize	Boolean	true
useQos	Boolean	true
useLen	Boolean	true
useLast	Boolean	true
useResp	Boolean	true
useProt	Boolean	true
useStrb	Boolean	true
useUser	Boolean	false

简而言之，AXI4 总线在 SpinalHDL 库中定义方式如下：

```
case class Axi4(config: Axi4Config) extends Bundle with IMasterSlave {
  val aw = Stream(Axi4Aw(config))
  val w  = Stream(Axi4W(config))
  val b  = Stream(Axi4B(config))
  val ar = Stream(Axi4Ar(config))
  val r  = Stream(Axi4R(config))

  override def asMaster(): Unit = {
    master(ar,aw,w)
    slave(r,b)
  }
}
```

这是一个简单的使用示例：

```
val axiConfig = Axi4Config(
  addressWidth = 32,
  dataWidth    = 32,
  idWidth      = 4
)
val axiX = Axi4(axiConfig)
val axiY = Axi4(axiConfig)

when(axiY.aw.valid) {
  // ...
}
```

变体

Axi4 总线还有其他 3 种变体：

类型	描述
Axi4ReadOnly	只存在 AR 和 R 通道
Axi4WriteOnly	只存在 AW、W 和 B 通道
Axi4Shared	此变体是该库的首创。 它使用 4 个通道，W、B、R，还有一个新通道，称为 AWR。 AWR 通道可用于传输 AR 和 AW 事务。为了分离它们，需要一个 `write` 信号。 这种 Axi4Shared 变体的优点是使用更少的面积，特别是在互连方面。

函数和运算符

名称	返回类型	描述
X » Y		将 X 连接到 Y。能够像 AXI4 规范中指定的那样推断默认值，并以安全的方式调整一些位宽。
X « Y		执行 » 运算符相反的操作
X.toWriteOnly	Axi4WriteOnly	返回由 X 驱动的 Axi4WriteOnly 总线
X.toReadOnly	Axi4ReadOnly	返回由 X 驱动的 Axi4ReadOnly 总线

10.11.4 AvalonMM

AvalonMM 总线非常适合 FPGA。它非常灵活：

- 能够与 APB 一样简单
- 在许多需要带宽的应用中比 AHB 更好，因为 AvalonMM 有一种将读取响应与命令解耦的模式（减少延迟读延迟的影响）。
- 性能不如 AXI，但使用的逻辑面积少得多（读取和写入命令使用相同的握手通道。主端不需要存储挂起请求的地址，从而避免读取/写入冒险）

Configuration and instantiation

`AvalonMM` 包有一个构造参数 `AvalonMMConfig` 。由于 Avalon 总线的灵活性，`AvalonMMConfig` 有很多配置元素。有关 Avalon 规范的更多信息，请访问英特尔网站。

```
case class AvalonMMConfig( addressWidth : Int,
                           dataWidth : Int,
                           burstCountWidth : Int,
                           useByteEnable : Boolean,
                           useDebugAccess : Boolean,
                           useRead : Boolean,
                           useWrite : Boolean,
                           useResponse : Boolean,
                           useLock : Boolean,
                           useWaitRequestn : Boolean,
                           useReadDataValid : Boolean,
                           useBurstCount : Boolean,
                           // useEndOfPacket : Boolean,

                           addressUnits : AddressUnits = symbols,
                           burstCountUnits : AddressUnits = words,
                           burstOnBurstBoundariesOnly : Boolean = false,
                           constantBurstBehavior : Boolean = false,
                           holdTime : Int = 0,
                           linewrapBursts : Boolean = false,
                           maximumPendingReadTransactions : Int = 1,
                           maximumPendingWriteTransactions : Int = 0, // unlimited
                           readLatency : Int = 0,
                           readWaitTime : Int = 0,
                           setupTime : Int = 0,
                           writeWaitTime : Int = 0
                           )
```

这个配置类还有一些函数：

名称	返回类型	描述
getReadOnlyConfig	AvalonMMConfig	返回一个类似的配置，但禁用所有写入属性
getWriteOnlyConfig	AvalonMMConfig	返回一个类似的配置，但禁用所有读取属性

这个配置伴随对象还有一些函数来提供一些 `AvalonMMConfig` 模板：

名称	返回类型	描述
fixed(addressWidth,dataWidth,readLatency)	AvalonMMConfig	返回一个具有固定读取时间的简单配置
pipelined(addressWidth,dataWidth)	AvalonMMConfig	返回一个具有可变延迟读取的配置 (readDataValid)
bursted(addressWidth,dataWidth,burstCountWidth)	AvalonMMConfig	返回一个具有可变延迟读取和突发功能的配置

```
// Create a write only AvalonMM configuration with burst capabilities and byte␣
↪enable
val myAvalonConfig =  AvalonMMConfig.bursted(
                        addressWidth = addressWidth,
                        dataWidth = memDataWidth,
                        burstCountWidth = log2Up(burstSize + 1)
                      ).copy(
                        useByteEnable = true,
                        constantBurstBehavior = true,
                        burstOnBurstBoundariesOnly = true
                      ).getWriteOnlyConfig

// Create an instance of the AvalonMM bus by using this configuration
val bus = AvalonMM(myAvalonConfig)
```

10.11.5 Tilelink

Configuration and instantiation

这是一个简单的示例，它定义了两个不相干的 tilelink 总线实例并将它们连接起来：

```
import spinal.lib.bus.tilelink
val param = tilelink.BusParameter.simple(
  addressWidth = 32,
  dataWidth    = 64,
  sizeBytes    = 64,
  sourceWidth  = 4
)
val busA, busB = tilelink.Bus(param)
busA << busB
```

这里与上面相同，但是具有一致性通道

```
import spinal.lib.bus.tilelink
val param = tilelink.BusParameter(
  addressWidth = 32,
  dataWidth = 64,
  sizeBytes = 64,
  sourceWidth = 4,
  sinkWidth = 0,
  withBCE = false,
  withDataA = true,
  withDataB = false,
  withDataC = false,
  withDataD = true,
  node = null
)
val busA, busB = tilelink.Bus(param)
busA << busB
```

Those above where for the hardware instantiation, the thing is that it is the simple / easy part. When things goes into SoC / memory coherency, you kind of need an additional layer to negotiate / propagate parameters all around. That's what tilelink.fabric.Node is about.

10.11.6 tilelink.fabric.Node

tilelink.fabric.Node is an additional layer over the regular tilelink hardware instantiation which handle negotiation and parameters propagation at a SoC level.

It is mostly based on the Fiber API, which allows to create elaboration time fibers (user-space threads), allowing to schedule future parameter propagation / negotiation and hardware elaboration.

可以通过 3 种方式创建节点 (Node)：

- tilelink.fabric.Node.down()：创建一个可以向下连接（向从端）的节点，因此它将用于 CPU/DMA/桥的代理
- tilelink.fabric.Node()：创建中间节点
- tilelink.fabric.Node.up()：创建一个可以向上连接（向主端）的节点，因此它将用于外设/存储器/桥的代理

节点大多具有以下属性：

- bus : Handle[tilelink.Bus]；总线的硬件实例
- m2s.proposed : Handle[tilelink.M2sSupport]；由向上连接提出的功能集
- m2s.supported : Handle[tilelink.M2sSupport]: 向下连接支持的功能集
- m2s.parameter : Handle[tilelink.M2sParameter]: 最终的总线参数

您可以注意到它们都是句柄。Handle 是 SpinalHDL 中在纤程之间共享值的一种方式。如果一个纤程读取一个句柄，而这个句柄还没有值，它将阻止该纤程的执行，直到另一个纤程向该句柄提供一个值。

There is also a set of attributes like m2s, but reversed (named s2m) which specify the parameters for the transactions initiated by the slave side of the interconnect (ex memory coherency).

有两个演讲介绍了 tilelink.fabric.Node。这两个演讲可能并不完全遵循实际语法，它们仍然遵循以下概念：

- 介绍：https://youtu.be/hVi9xOGuuek
- 深入：https://peertube.f-si.org/videos/watch/bcf49c84-d21d-4571-a73e-96d7eb89e907

顶层示例

以下是一个简单的虚拟 SoC 顶层设计示例：

```
val cpu = new CpuFiber()

val ram = new RamFiber()
ram.up at(0x10000, 0x200) of cpu.down // map the ram at [0x10000-0x101FF], the ram␣
↪will infer its own size from it

val gpio = new GpioFiber()
gpio.up at 0x20000 of cpu.down // map the gpio at [0x20000-0x20FFF], its range of␣
↪4KB being fixed internally
```

您还可以定义互连中的中间节点，如下所示：

```
val cpu = new CpuFiber()

val ram = new RamFiber()
ram.up at(0x10000, 0x200) of cpu.down
```

(续下页)

(接上页)

```
// Create a peripherals namespace to keep things clean
val peripherals = new Area {
  // Create a intermediate node in the interconnect
  val access = tilelink.fabric.Node()
  access at 0x20000 of cpu.down

  val gpioA = new GpioFiber()
  gpioA.up at 0x0000 of access

  val gpioB = new GpioFiber()
  gpioB.up at 0x1000 of access
}
```

GPIOFiber 示例

GpioFiber 是一个简单的 tilelink 外设，可以读取/驱动 32 位三态阵列。

```
import spinal.lib._
import spinal.lib.bus.tilelink
import spinal.core.fiber.Fiber
class GpioFiber extends Area {
  // Define a node facing upward (toward masters only)
  val up = tilelink.fabric.Node.up()

  // Define a elaboration thread to specify the "up" parameters and generate the␣
↪hardware
  val fiber = Fiber build new Area {
    // Here we first define what our up node support. m2s mean master to slave␣
↪requests
    up.m2s.supported load tilelink.M2sSupport(
      addressWidth = 12,
      dataWidth = 32,
      // Transfers define which kind of memory transactions our up node will␣
↪support.
      // Here it only support 4 bytes get/putfull
      transfers = tilelink.M2sTransfers(
        get = tilelink.SizeRange(4),
        putFull = tilelink.SizeRange(4)
      )
    )
    // s2m mean slave to master requests, those are only use for memory coherency␣
↪purpose
    // So here we specify we do not need any
    up.s2m.none()

    // Then we can finally generate some hardware
    // Starting by defining a 32 bits TriStateArray (Array meaning that each pin␣
↪has its own writeEnable bit
    val pins = master(TriStateArray(32 bits))

    // tilelink.SlaveFactory is a utility allowing to easily generate the logic␣
↪required
    // to control some hardware from a tilelink bus.
    val factory = new tilelink.SlaveFactory(up.bus, allowBurst = false)

    // Use the SlaveFactory API to generate some hardware to read / drive the pins
    val writeEnableReg = factory.drive(pins.writeEnable, 0x0) init (0)
    val writeReg = factory.drive(pins.write, 0x4) init(0)
    factory.read(pins.read, 0x8)
```

(续下页)

(接上页)

```
  }
}
```

RamFiber 示例

RamFiber 是常规 tilelink Ram 组件的集成层。

```
import spinal.lib.bus.tilelink
import spinal.core.fiber.Fiber
class RamFiber() extends Area {
  val up = tilelink.fabric.Node.up()

  val thread = Fiber build new Area {
    // Here the supported parameters are function of what the master would like us␣
↪to ideally support.
    // The tilelink.Ram support all addressWidth / dataWidth / burst length / get /
↪ put accesses
    // but doesn't support atomic / coherency. So we take what is proposed to use␣
↪and restrict it to
    // all sorts of get / put request
    up.m2s.supported load up.m2s.proposed.intersect(M2sTransfers.allGetPut)
    up.s2m.none()

    // Here we infer how many bytes our ram need to be, by looking at the memory␣
↪mapping of the connected masters
    val bytes = up.ups.map(e => e.mapping.value.highestBound - e.mapping.value.
↪lowerBound + 1).max.toInt

    // Then we finally generate the regular hardware
    val logic = new tilelink.Ram(up.bus.p.node, bytes)
    logic.io.up << up.bus
  }
}
```

CpuFiber 示例

CpuFiber 是一个虚拟的主端集成的示例。

```
import spinal.lib.bus.tilelink
import spinal.core.fiber.Fiber

class CpuFiber extends Area {
  // Define a node facing downward (toward slaves only)
  val down = tilelink.fabric.Node.down()

  val fiber = Fiber build new Area {
    // Here we force the bus parameters to a specific configurations
    down.m2s forceParameters tilelink.M2sParameters(
      addressWidth = 32,
      dataWidth = 64,
      // We define the traffic of each master using this node. (one master => one␣
↪M2sAgent)
      // In our case, there is only the CpuFiber.
      masters = List(
        tilelink.M2sAgent(
          name = CpuFiber.this, // Reference to the original agent.
          // A agent can use multiple sets of source ID for different purposes
          // Here we define the usage of every sets of source ID
```

(续下页)

(接上页)

```
          // In our case, let's say we use ID [0-3] to emit get/putFull requests
          mapping = List(
            tilelink.M2sSource(
              id = SizeMapping(0, 4),
              emits = M2sTransfers(
                get = tilelink.SizeRange(1, 64), // Meaning the get access can be any power of 2 size in [1, 64]
                putFull = tilelink.SizeRange(1, 64)
              )
            )
          )
        )
      )
    )

    // Lets say the CPU doesn't support any slave initiated requests (memory coherency)
    down.s2m.supported load tilelink.S2mSupport.none()

    // Then we can generate some hardware (nothing useful in this example)
    down.bus.a.setIdle()
    down.bus.d.ready := True
  }
}
```

Tilelink 的一个特殊性是，它假设主端不会向未映射的内存空间发出请求。为了让主机识别允许访问哪些内存，您可以使用 spinal.lib.system.tag.MemoryConnection.getMemoryTransfers 工具，如下所示：

```
val mappings = spinal.lib.system.tag.MemoryConnection.getMemoryTransfers(down)
// Here we just print the values out in stdout, but instead you can generate some hardware from it.
for(mapping <- mappings) {
  println(s"- ${mapping.where} -> ${mapping.transfers}")
}
```

如果您在 CPU 的纤程中运行此命令，在下面的 soc 中：

```
val cpu = new CpuFiber()

val ram = new RamFiber()
ram.up at(0x10000, 0x200) of cpu.down

// Create a peripherals namespace to keep things clean
val peripherals = new Area {
  // Create a intermediate node in the interconnect
  val access = tilelink.fabric.Node()
  access at 0x20000 of cpu.down

  val gpioA = new GpioFiber()
  gpioA.up at 0x0000 of access

  val gpioB = new GpioFiber()
  gpioB.up at 0x1000 of access
}
```

你会得到：

```
- toplevel/ram_up mapped=SM(0x10000, 0x200) through=List(OT(0x10000))  -> GF
- toplevel/peripherals_gpioA_up mapped=SM(0x20000, 0x1000) through=List(OT(0x20000), OT(0x0))  -> GF
- toplevel/peripherals_gpioB_up mapped=SM(0x21000, 0x1000)
```

(续下页)

(接上页)

```
↪through=List(OT(0x20000), OT(0x1000))   -> GF
```

- “through=” 指定了到达目标所需的地址转换链。
- “SM” 表示 SizeMapping(address, size)
- “OT” 表示 OffsetTransformer(offset)

Note that you can also add PMA (Physical Memory Attributes) to nodes and retrieves them via this getMemoryTransfers utilities.

当前 PMA 的定义是：

```
object MAIN          extends PMA
object IO            extends PMA
object CACHABLE      extends PMA // an intermediate agent may have cached a copy␣
↪of the region for you
object TRACEABLE     extends PMA // the region may have been cached by another␣
↪master, but coherence is being provided
object UNCACHABLE    extends PMA // the region has not been cached yet, but should␣
↪be cached when possible
object IDEMPOTENT    extends PMA // reads return most recently put content, but␣
↪content should not be cached
object EXECUTABLE    extends PMA // Allows an agent to fetch code from this region
object VOLATILE      extends PMA // content may change without a write
object WRITE_EFFECTS extends PMA // writes produce side effects and so must not be␣
↪combined/delayed
object READ_EFFECTS  extends PMA // reads produce side effects and so must not be␣
↪issued speculatively
```

getMemoryTransfers 工具依赖于专用的 SpinalTag：

```
trait MemoryConnection extends SpinalTag {
  def up : Nameable with SpinalTagReady // Side toward the masters of the system
  def down : Nameable with SpinalTagReady // Side toward the slaves of the system
  def mapping : AddressMapping // Specify the memory mapping of the slave from the␣
↪master address (before transformers are applied)
  def transformers : List[AddressTransformer]  // List of alteration done to the␣
↪address on this connection (ex offset, interleaving, ...)
  def sToM(downs : MemoryTransfers, args : MappedNode) : MemoryTransfers = downs //
↪ Convert the slave MemoryTransfers capabilities into the master ones
}
```

该 SpinalTag 可以应用于给定内存总线连接的两端，以保持该连接在生成时可被发现，从而创建内存连接 (MemoryConnection) 图。它的一个优点是它与总线无关，这意味着它不是 tilelink 特有的。

位宽适配器 (WidthAdapter) 示例

位宽适配器是桥的一个简单例子。

```
class WidthAdapterFiber() extends Area {
  val up = Node.up()
  val down = Node.down()

  // Populate the MemoryConnection graph
  new MemoryConnection {
    override def up = up
    override def down = down
    override def transformers = Nil
    override def mapping = SizeMapping(0, BigInt(1) << WidthAdapterFiber.this.up.
↪m2s.parameters.addressWidth)
```

(续下页)

(接上页)

```
    populate()
  }

  // Fiber in which we will negotiate the data width parameters and generate the␣
↪hardware
  val logic = Fiber build new Area {
    // First, we propagate downward the parameter proposal, hopping that the␣
↪downward side will agree
    down.m2s.proposed.load(up.m2s.proposed)

    // Second, we will propagate upward what is actually supported, but will take␣
↪care of any dataWidth mismatch
    up.m2s.supported load down.m2s.supported.copy(
      dataWidth = up.m2s.proposed.dataWidth
    )

    // Third, we propagate downward the final bus parameter, but will take care of␣
↪any dataWidth mismatch
    down.m2s.parameters load up.m2s.parameters.copy(
      dataWidth = down.m2s.supported.dataWidth
    )

    // No alteration on s2m parameters
    up.s2m.from(down.s2m)

    // Finally, we generate the hardware
    val bridge = new tilelink.WidthAdapter(up.bus.p, down.bus.p)
    bridge.io.up << up.bus
    bridge.io.down >> down.bus
  }
}
```

10.12 通信接口

10.12.1 SPI XDR

这是一个 SPI 控制器，它支持：

- 半双工/全双工
- 单线/双线/四线 SPI
- SDR/DDR/.. 数据速率

您可以在此处找到其 APB3 实现：

https://github.com/SpinalHDL/SpinalHDL/blob/68b6158700fc2440ea7980406f927262c004faca/lib/src/main/scala/spinal/lib/com/spi/xdr/Apb3SpiXdrMasterCtrl.scala#L43

配置

这是一个示例。

```
Apb3SpiXdrMasterCtrl(
  SpiXdrMasterCtrl.MemoryMappingParameters(
    SpiXdrMasterCtrl.Parameters(
      dataWidth = 8, // Each transfer will be 8 bits
      timerWidth = 12, // The timer is used to slow down the transmission
      spi = SpiXdrParameter( // Specify the physical SPI interface
        dataWidth = 4, // Number of physical SPI data pins
        ioRate = 1, // Specify the number of transfer that each spi pin can do per␣
↪clock 1 => SDR, 2 => DDR
        ssWidth = 1 // Number of chip selects
      )
    )
    .addFullDuplex(id = 0) // Add support for regular SPI (MISO / MOSI) using the␣
↪mode id 0
    .addHalfDuplex( // Add another mode
      id = 1,  // mapped on mode id 1
      rate = 1, // When rate is 1, the clock will do up to one toggle per cycle,␣
↪divided by the (timer+1)
                // When rate bigger (ex 2), the controller will ignore the timer,␣
↪and use the SpiXdrParameter.ioRate
                // capabilities to emit up to "rate" transition per clock cycle.
      ddr = false, // sdr => 1 bit per SPI clock, DDR => 2 bits per SPI clock
      spiWidth = 4 // Number of physical SPI data pin used for serialization
    ),
    cmdFifoDepth = 32,
    rspFifoDepth = 32,
    xip = null
  )
)
```

软件驱动

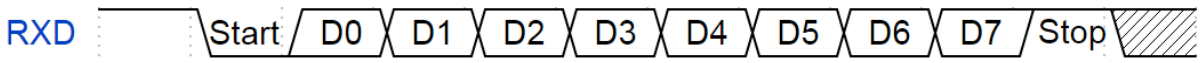
看：

https://github.com/SpinalHDL/SaxonSoc/blob/dev-0.3/software/standalone/driver/spi.h https://github.com/SpinalHDL/SaxonSoc/blob/dev-0.3/software/standalone/spiDemo/src/main.c

10.12.2 串口

例如，UART 协议可用于发送和接收 RS232/RS485 帧。

有一个 8 位帧的示例，无奇偶校验和一位停止位：



总线定义

```
case class Uart() extends Bundle with IMasterSlave {
  val txd = Bool() // Used to emit frames
  val rxd = Bool() // Used to receive frames

  override def asMaster(): Unit = {
    out(txd)
    in(rxd)
  }
}
```

UartCtrl

库中实现了 Uart 控制器。该控制器的特性是使用一个采样窗口读取 rxd 引脚，然后使用多数投票制来过滤其值。

IO 名称	方向	类型	描述
config	in	UartCtrlConfig	用于设置控制器的时钟分频器/奇偶校验/停止/数据长度
write	slave	Stream[Bits]	用于请求帧传输的反压流端口
read	master	Flow[Bits]	用于接收解码帧的流端口
uart	master	Uart	与实际实现的连接接口

控制器可以通过一个 UartCtrlGenerics 配置对象来实例化：

属性	类型	描述
dataWidthMax	Int	帧内最大位数
clockDividerWidth	Int	内部时钟分频器的位宽
preSamplingSize	Int	指定在一个 UART 波特开始时丢弃多少 samplingTick
samplingSize	Int	指定有多少 samplingTick 用于采样 UART 波特中段的 rxd 值
postSamplingSize	Int	指定在 UART 波特结束时丢弃多少个 samplingTick

10.12.3 USB 设备

SpinalHDL 库中存在一个 USB 设备控制器。

A few bullet points to summarize support:

- 实现了允许 CPU 配置和管理端点
- 存储端点状态和事务描述符的内部 RAM
- Up to 16 endpoints (for virtually no price)
- Support USB host full speed (12 Mbps)
- 在 Linux 上使用自己的驱动程序进行测试 (https://github.com/SpinalHDL/linux/blob/dev/drivers/usb/gadget/udc/spinal_udc.c)
- Bmb memory interface for the configuration
- 内部物理层需要一个时钟，该时钟需为 12 Mhz 的倍数，至少 48 Mhz
- 控制器频率不受限制
- 无需外部物理层

Linux 小工具经过测试且功能正常：

- 串行连接
- 以太网连接
- 大容量存储（ArtyA7 Linux 上约为 8 Mbps）

部署：

- https://github.com/SpinalHDL/SaxonSoc/tree/dev-0.3/bsp/digilent/ArtyA7SmpLinux
- https://github.com/SpinalHDL/SaxonSoc/tree/dev-0.3/bsp/radiona/ulx3s/smp

架构

控制器由以下部分组成：

- 一小部分控制寄存器
- 一个用于存储端点状态、传输描述符和端点 0 配置数据的内部 RAM。

每个端点的描述符链表时用于处理 USB 出入 (IN/OUT) 事务和数据。

端点 0 也像所有其他端点一样管理出入 USB 的传输事务，但也会有一些额外的硬件来管理设置 (SETUP) 事务：

- 它的链表在每次设置事务时都会被清除
- 来自设置 (SETUP) 事务的数据存储在固定位置 (SETUP_DATA)
- 它有一个用于设置 (SETUP) 事务的特定中断标志

寄存器

请注意，控制器的所有寄存器和存储器只能以 32 位字的访问方式进行访问，不支持字节访问。

帧 FRAME (0xFF00)

名称	类型	位	描述
usbFrameId	RO	31-0	当前 USB 帧的 ID

地址 ADDRESS (0xFF04)

名称	类型	位	描述
address	WO	6-0	The device will only listen at tokens with the specified address This field is automatically cleared on usb reset events
enable	WO	8	如果置 1，启用 USB 地址过滤
trigger	WO	9	Set the enable (see above) on the next EP0 IN token completion Cleared by the hardware after any EP0 completion

The idea here is to keep the whole register cleared until a USB SET_ADDRESS setup packet is received on EP0. At that moment, you can set the address and the trigger field, then provide the IN zero length descriptor to EP0 to finalize the SET_ADDRESS sequence. The controller will then automatically turn on the address filtering at the completion of that descriptor.

中断 INTERRUPT (0xFF08)

该寄存器的每个位都可以通过写入‘1’来清除。读取该寄存器将返回当前中断状态。

名称	类型	位	描述
endpoints	W1C	15-0	当端点产生中断时拉高
reset	W1C	16	当 USB 复位发生时拉高
ep0Setup	W1C	17	当端点 0 收到设置请求时拉高
suspend	W1C	18	当发生 USB 挂起时拉高
resume	W1C	19	当 USB 发生恢复时拉高
disconnect	W1C	20	当 USB 断开连接时拉高

暂停 HALT (0xFF0C)

该寄存器允许将单个端点置于休眠状态，以确保 CPU 操作的原子性，从而允许在端点寄存器和描述符上执行读/修改/写操作。如果 USB 主机在暂停启用且端点启用的情况下寻址给定端点，那么外设将返回 NAK。

名称	类型	位	描述
endpointId	WO	3-0	您想要进入休眠状态的目标端点
enable	WO	4	当设置暂停时为活动状态，当清除时端点解除暂停。
effective enable	RO	5	设置使能后，需要等待硬件本身设置该位，以保证原子性

配置 CONFIG (0xFF10)

名称	类型	位	描述
pullupSet	SO	0	写入‘1’以使能 dp 引脚上的 USB 设备上拉
pullupClear	SO	1	
interruptEnableSet	SO	2	写入‘1’让当前和未来的中断发生
interruptEnableClear	SO	3	

信息 INFO (0xFF20)

名称	类型	位	描述
ramSize	RO	3-0	内部 RAM 将有 (1 « this) 字节

端点 ENDPOINTS (0x0000 - 0x003F)

The endpoints status are stored at the beginning of the internal ram over one 32 bits word each.

名称	类型	位	描述
enable	RW	0	If not set, the endpoint will ignore all the traffic
stall	RW	1	如果设置，端点将始终返回 STALL 状态
nack	RW	2	如果设置，端点将始终返回 NACK 状态
dataPhase	RW	3	指定使用的出入数据 PID。‘0’=> DATA0。该字段也由控制器更新。
head	RW	15-4	指定当前描述符头（链表）。0 => 空列表，字节地址 = this « 4
isochronous	RW	16	
maxPacketSize	RW	31-22	

要获得端点响应，您需要：

- 将其使能标志设置为 1

那么有几种情况：-要么设置了 stall 或 nack 标志，所以控制器将始终响应相应的响应 -要么，对于 EP0 设置请求，控制器不会使用描述符，而是会将数据写入 SETUP_DATA 寄存器和 ACK -要么你有一个空链表 (head==0)，在这种情况下它将响应 NACK -要么你至少有一个由 head 指向的描述符，在这种情况下，它将执行该描述符，并在一切顺利时进行 ACK

设置数据 SETUP_DATA (0x0040 - 0x0047)

当端点 0 接收到 SETUP 事务时，该事务的数据将存储在该位置。

描述符

描述符允许指定一个端点需要如何处理出入 (IN/OUT) 事务的数据阶段。它们存储在内部 RAM 中，可以通过链表链接在一起，并且需要在 16 字节边界上对齐

名称	字	位	描述
offset	0	15-0	指定当前传输进度（以字节为单位）
code	0	19-16	0xF => 进行中，0x0 => 成功
next	1	15-4	指向下一个描述符 0 => 无，字节地址 = this « 4
length	1	31-16	分配给数据字段的字节数
方向	2	16	‘0’ => 输出，’1’ => 输入
interrupt	2	17	如果置位，描述符完成时将产生中断。
completionOnFull	2	18	通常，描述符补全只会在 USB 传输小于 maxPacketSize 时发生。但如果置位了该字段，那么当描述符被填满时也被视为事件已完成。(offset == length)
data1OnCompletion	2	19	描述符完成时强制端点 dataPhase 为 DATA1
data	…	…	

请注意，如果控制器接收到 IN/OUT 与描述符 IN/OUT 不匹配的帧，那么该帧将被忽略。

Also, to initialize a descriptor, the CPU should set the code field to 0xF

用法

```
import spinal.core._
import spinal.core.sim._
import spinal.lib.bus.bmb.BmbParameter
import spinal.lib.com.usb.phy.UsbDevicePhyNative
import spinal.lib.com.usb.sim.UsbLsFsPhyAbstractIoAgent
import spinal.lib.com.usb.udc.{UsbDeviceCtrl, UsbDeviceCtrlParameter}


case class UsbDeviceTop() extends Component {
  val ctrlCd = ClockDomain.external("ctrlCd", frequency = FixedFrequency(100 MHz))
  val phyCd = ClockDomain.external("phyCd", frequency = FixedFrequency(48 MHz))

  val ctrl = ctrlCd on new UsbDeviceCtrl(
    p = UsbDeviceCtrlParameter(
      addressWidth = 14
    ),
    bmbParameter = BmbParameter(
      addressWidth = UsbDeviceCtrl.ctrlAddressWidth,
      dataWidth = 32,
      sourceWidth = 0,
      contextWidth = 0,
      lengthWidth = 2
    )
  )

  val phy = phyCd on new UsbDevicePhyNative(sim = true)
  ctrl.io.phy.cc(ctrlCd, phyCd) <> phy.io.ctrl

  val bmb = ctrl.io.ctrl.toIo()
  val usb = phy.io.usb.toIo()
  val power = phy.io.power.toIo()
  val pullup = phy.io.pullup.toIo()
  val interrupts = ctrl.io.interrupt.toIo()
}
```

(续下页)

(接上页)

```
object UsbDeviceGen extends App {
  SpinalVerilog(new UsbDeviceTop())
}
```

10.12.4 USB OHCI

SpinalHDL 库中有 USB OHCi 控制器（主机）。

A few bullet points to summarize support:

- 它遵循 *OpenHCI USB 开放式主机控制接口规范* (OHCI)。
- 它已经与上游的 linux/uboot OHCI 驱动兼容。（tinyUSB 上也有 OHCI 驱动）
- This provides USB host full speed and low speed capabilities (12 Mbps and 1.5 Mbps)
- 在 linux 和 uboot 上测试过
- 一个可以承载多个端口（多至 16 个）的控制器
- 用于 DMA 访问的 Bmb 存储器接口
- Bmb memory interface for the configuration
- 内部物理层需要一个时钟，该时钟需要为 12 Mhz 的倍数，至少 48 Mhz
- 控制器频率不受限制
- 无需外部物理层

经过测试且功能正常的设备：

- 大容量存储（ArtyA7 Linux 上约为 8 Mbps）
- 键盘/鼠标
- 音频输出
- 集线器

限制：

- 某些 USB 集线器（目前已有一个）对将低速设备连接至全速主机的模式不友好。
- 某些现代设备无法在 USB 全速上运行（例如：Gbps 以太网适配器）
- 需要与 CPU 保持内存一致性（或者需要 CPU 能够刷新驱动中的数据缓存）

部署：

- https://github.com/SpinalHDL/SaxonSoc/tree/dev-0.3/bsp/digilent/ArtyA7SmpLinux
- https://github.com/SpinalHDL/SaxonSoc/tree/dev-0.3/bsp/radiona/ulx3s/smp

用法

```
import spinal.core._
import spinal.core.sim._
import spinal.lib.bus.bmb._
import spinal.lib.bus.bmb.sim._
import spinal.lib.bus.misc.SizeMapping
import spinal.lib.com.usb.ohci._
import spinal.lib.com.usb.phy.UsbHubLsFs.CtrlCc
import spinal.lib.com.usb.phy._
```

(续下页)

(接上页)

```
class UsbOhciTop(val p : UsbOhciParameter) extends Component {
  val ohci = UsbOhci(p, BmbParameter(
    addressWidth = 12,
    dataWidth = 32,
    sourceWidth = 0,
    contextWidth = 0,
    lengthWidth = 2
  ))

  val phyCd = ClockDomain.external("phyCd", frequency = FixedFrequency(48 MHz))
  val phy = phyCd(UsbLsFsPhy(p.portCount, sim=true))

  val phyCc = CtrlCc(p.portCount, ClockDomain.current, phyCd)
  phyCc.input <> ohci.io.phy
  phyCc.output <> phy.io.ctrl

  // propagate io signals
  val irq = ohci.io.interrupt.toIo
  val ctrl = ohci.io.ctrl.toIo
  val dma = ohci.io.dma.toIo
  val usb = phy.io.usb.toIo
  val management = phy.io.management.toIo
}

object UsbHostGen extends App {
  val p = UsbOhciParameter(
    noPowerSwitching = true,
    powerSwitchingMode = true,
    noOverCurrentProtection = true,
    powerOnToPowerGoodTime = 10,
    dataWidth = 64, // DMA data width, up to 128
    portsConfig = List.fill(4)(OhciPortParameter()) // 4 Ports
  )

  SpinalVerilog(new UsbOhciTop(p))
}
```

10.13 IO 口

10.13.1 可读开漏 IO(ReadableOpenDrain)

ReadableOpenDrain 线束定义如下：

```
case class ReadableOpenDrain[T<: Data](dataType : HardType[T]) extends Bundle with␣
↪IMasterSlave {
  val write,read : T = dataType()

  override def asMaster(): Unit = {
    out(write)
    in(read)
  }
}
```

然后，作为主端，您可以使用 `read` 信号读取外部值，并使用 `write` 设置您想要在输出上驱动的值。

这是一个用法示例：

```
val io = new Bundle {
  val dataBus = master(ReadableOpenDrain(Bits(32 bits)))
}

io.dataBus.write := 0x12345678
when(io.dataBus.read === 42) {

}
```

10.13.2 三态

在许多情况下，三态信号难以处理：

- 它们不是真正的数字性
- 除了 IO 之外，它们不用于数字设计
- 三态概念并不自然地适合 SpinalHDL 内部图。

SpinalHDL 为三态信号提供两种不同的抽象。`TriState` 线束和模拟信号和输入输出 信号。两者有不同的目的：

- TriState 应用于大多数目的，尤其是在设计中。该束包含一个附加信号来传递当前的方向。
- `Analog` 和 `inout` 应用于设备边界上的驱动以及其他一些特殊情况。有关更多详细信息，请参阅参考文档页面。

如上所述，推荐的方法是在设计中使用 `TriState` 。然后，在顶层，`TriState` 线束被赋值给模拟输入输出，以使综合工具推断出正确的 I/O 驱动。这可以通过输入/出包装器 自动完成，或者根据需要手动完成。

三态

TriState 线束定义如下：

```
case class TriState[T <: Data](dataType : HardType[T]) extends Bundle with␣
↪IMasterSlave {
  val read,write = T = dataType()
  val writeEnable = Bool()

  override def asMaster(): Unit = {
    out(write,writeEnable)
    in(read)
  }
}
```

主端可以使用 `read` 信号读取外部值，使用 `writeEnable` 启用输出，最后使用 `write` 设置输出驱动的值。

这是一个使用示例：

```
val io = new Bundle {
  val dataBus = master(TriState(Bits(32 bits)))
}

io.dataBus.writeEnable := True
io.dataBus.write := 0x12345678
when(io.dataBus.read === 42) {

}
```

三态阵列

在某些情况下，您需要控制每个单独引脚的输出使能（像 GPIO 一样）。在这种情况下，您可以使用 TriStateArray 线束。

它的定义如下：

```
case class TriStateArray(width : BitCount) extends Bundle with IMasterSlave {
  val read,write,writeEnable = Bits(width)

  override def asMaster(): Unit = {
    out(write,writeEnable)
    in(read)
  }
}
```

它与 TriState 线束相同，不同的是 `writeEnable` 是一个位 (Bits) 来控制每个输出缓冲区。

这是一个用法示例：

```
val io = new Bundle {
  val dataBus = master(TriStateArray(32 bits)
}

io.dataBus.writeEnable := 0x87654321
io.dataBus.write := 0x12345678
when(io.dataBus.read === 42) {

}
```

10.14 图形

10.14.1 颜色

RGB

您可以使用 Rgb 线束在硬件中对建模颜色。该 Rgb 线束采用 RgbConfig 类作为参数，该类指定每个通道的位数：

```
case class RgbConfig(rWidth : Int,gWidth : Int,bWidth : Int) {
  def getWidth = rWidth + gWidth + bWidth
}

case class Rgb(c: RgbConfig) extends Bundle {
  val r = UInt(c.rWidth bits)
  val g = UInt(c.gWidth bits)
  val b = UInt(c.bWidth bits)
}
```

这些类的使用如下：

```
val config = RgbConfig(5,6,5)
val color = Rgb(config)
color.r := 31
```

10.14.2 VGA

VGA 总线

VGA 总线通过 Vga 线束定义。

```
case class Vga (rgbConfig: RgbConfig) extends Bundle with IMasterSlave {
  val vSync = Bool()
  val hSync = Bool()

  val colorEn = Bool()  // High when the frame is inside the color area
  val color = Rgb(rgbConfig)

  override def asMaster() = this.asOutput()
}
```

VGA 时序

VGA 时序可以使用 VgaTimings 线束在硬件中建模：

```
case class VgaTimingsHV(timingsWidth: Int) extends Bundle {
  val colorStart = UInt(timingsWidth bits)
  val colorEnd = UInt(timingsWidth bits)
  val syncStart = UInt(timingsWidth bits)
  val syncEnd = UInt(timingsWidth bits)
}

case class VgaTimings(timingsWidth: Int) extends Bundle {
  val h = VgaTimingsHV(timingsWidth)
  val v = VgaTimingsHV(timingsWidth)

   def setAs_h640_v480_r60 = ...
   def driveFrom(busCtrl : BusSlaveFactory,baseAddress : Int) = ...
}
```

VGA 控制器

现有 VGA 控制器，其定义如下：

```
case class VgaCtrl(rgbConfig: RgbConfig, timingsWidth: Int = 12) extends Component
↪{
  val io = new Bundle {
    val softReset = in Bool()
    val timings   = in(VgaTimings(timingsWidth))

    val frameStart = out Bool()
    val pixels     = slave Stream (Rgb(rgbConfig))
    val vga        = master(Vga(rgbConfig))

    val error      = out Bool()
  }
  // ...
}
```

`frameStart` 是一个在每个新帧开始时脉冲一个周期的信号。
`pixels` 是一种颜色反压流，用于在需要时为 VGA 接口提供数据。
当需要传输像素但没有对象时，`error` 为高。

10.15 自动设计工具 (EDA)

10.15.1 QSysify

QSysify is a tool which is able to generate a QSys IP (tcl script) from a SpinalHDL component by analyzing its IO definition. It currently implement the following interfaces features :

- 主/从 AvalonMM
- 主/从 APB3
- 时钟域输入
- 复位输出
- 中断输入
- 导线（作为最后手段使用）

示例

以 UART 控制器为例：

```
case class AvalonMMUartCtrl(...) extends Component {
  val io = new Bundle {
    val bus =  slave(AvalonMM(AvalonMMUartCtrl.getAvalonMMConfig))
    val uart = master(Uart())
  }

  // ...
}
```

下面的 `main` 将生成 Verilog 和 QSys TCL 脚本，其中 io.bus 将作为 AvalonMM 总线，io.uart 作为导线：

```
object AvalonMMUartCtrl {
  def main(args: Array[String]) {
    // Generate the Verilog
    val toplevel = SpinalVerilog(AvalonMMUartCtrl(UartCtrlMemoryMappedConfig(...
↪))).toplevel

    // Add some tags to the avalon bus to specify it's clock domain (information␣
↪used by QSysify)
    toplevel.io.bus addTag(ClockDomainTag(toplevel.clockDomain))

    // Generate the QSys IP (tcl script)
    QSysify(toplevel)
  }
}
```

标签

由于 QSys 需要一些 SpinalHDL 硬件规范中未指定的信息，因此应在接口中添加一些标签：

AvalonMM / APB3

```
io.bus addTag(ClockDomainTag(busClockDomain))
```

中断输入

```
io.interrupt addTag(InterruptReceiverTag(relatedMemoryInterfacei,␣
↪interruptClockDomain))
```

复位输出

```
io.resetOutput addTag(ResetEmitterTag(resetOutputClockDomain))
```

添加新的接口支持

基本上，QSysify 工具可以使用接口 emitter 列表进行设置（如您在此处看到的）*<https://github.com/SpinalHDL/SpinalHDL/blob/764193013f84cfe4f82d7d1f1739c4561ef65860/lib/src/main/scala/spinal/lib/eda/alte*

您可以通过创建一个扩展 QSysifyInterfaceEmiter 的新类来创建自己的发射器 (emitter)

10.15.2 QuartusFlow

编译流是 Altera 定义的命令序列，这些命令使用命令行可执行文件的组合。完整的编译流会按顺序启动所有的编译器模块，进行综合、拟合、最终时序分析，并生成设备编程文件。

此文件 中的工具会帮助您消除冗余的 Quartus GUI。

对于单个 rtl 文件

对象 `spinal.lib.eda.altera.QuartusFlow` 可以自动报告单个 rtl 文件的使用面积和最大频率。

示例

```
val report = QuartusFlow(
   quartusPath="/eda/intelFPGA_lite/17.0/quartus/bin/",
   workspacePath="/home/spinalvm/tmp",
   toplevelPath="TopLevel.vhd",
   family="Cyclone V",
   device="5CSEMA5F31C6",
   frequencyTarget = 1 MHz
)
println(report)
```

上面的代码将使用 `TopLevel.vhd` 创建一个新的 Quartus 项目。

> **警告：** 此操作将删除文件夹 `workspacePath` ！

备注： `family` 和 `device` 值作为参数直接传递到 Quartus CLI。请检查 Quartus 文档以确定在您的项目中使用的正确值。

小贴士

为了测试具有太多引脚的组件，请将它们设置为 VIRTUAL_PIN 。

```
val miaou: Vec[Flow[Bool]] = Vec(master(Flow(Bool())), 666)
miaou.addAttribute("altera_attribute", "-name VIRTUAL_PIN ON")
```

对于一个现有项目

类 spinal.lib.eda.altera.QuartusProject 可以自动查找现有项目中的配置文件。它们用于对设备进行编译和编程。

示例

指定包含项目文件的路径，例如 .qpf 和 .cdf 。

```
val prj = new QuartusProject(
   quartusPath = "F:/intelFPGA_lite/20.1/quartus/bin64/",
   workspacePath = "G:/"
)
prj.compile()
prj.program()  // automatically find Chain Description File of the project
```

重要： 请记住在调用 prj.program() 之前保存项目的 .cdf 文件。

10.16 Pipeline

10.16.1 简介

spinal.lib.misc.pipeline 提供了一套流水线 API。相对于手动流水线它的主要优点是：

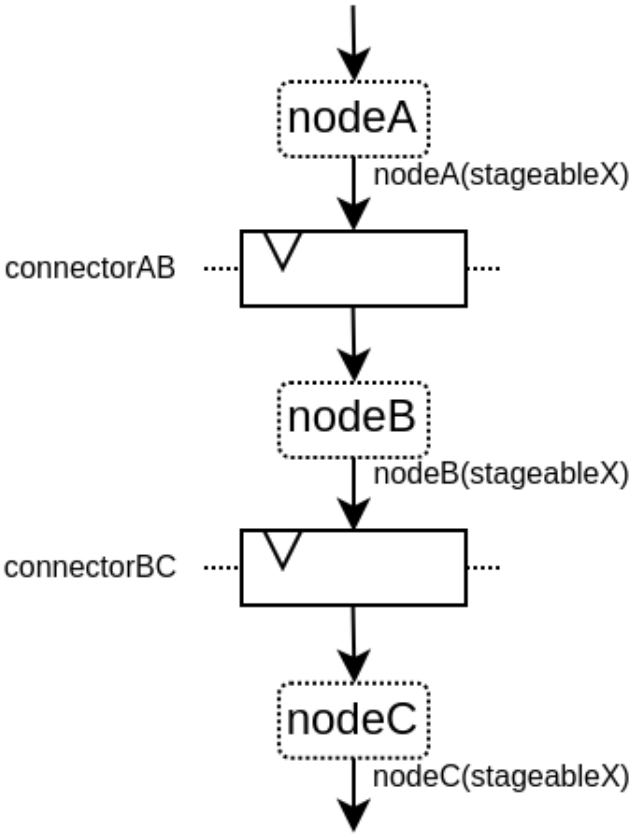
- 您不必预先准备好整个流水系统中所需的所有信号元素。您可以根据设计需要，以更特别的方式创建和使用可分级的信号，而无需重构所有中间阶段来处理信号
- 流水线的信号可以利用 SpinalHDL 的强大参数化能力，并且如果设计构建中不需要特定的参数化特征，则可以进行优化/移除，而不需要以显著的方式修改流水系统设计或项目代码库。
- 手动时序调整要容易得多，因为您不需要手动处理寄存器/仲裁器
- 它会自行管理仲裁器

API 由 4 个主要部分组成：

- Node：表示管道中的层
- Link：允许节点相互连接
- Builder：生成整个管道所需的硬件
- Payload：用于获取流水线的节点上的硬件信号

重要的是，Payload 不是硬件数据/信号实例，而是用于检索流水线在节点中数据/信号的关键，并且流水线构建器随后将在节点之间的每次给定 Payload 出现时自动互连/流水线。

以下是一个用于阐述的例子：



以下是关于此 API 的视频：

- https://www.youtube.com/watch?v=74h_-FMWWIM

简单示例

下面是一个简单的例子，它只使用了基本的 API：

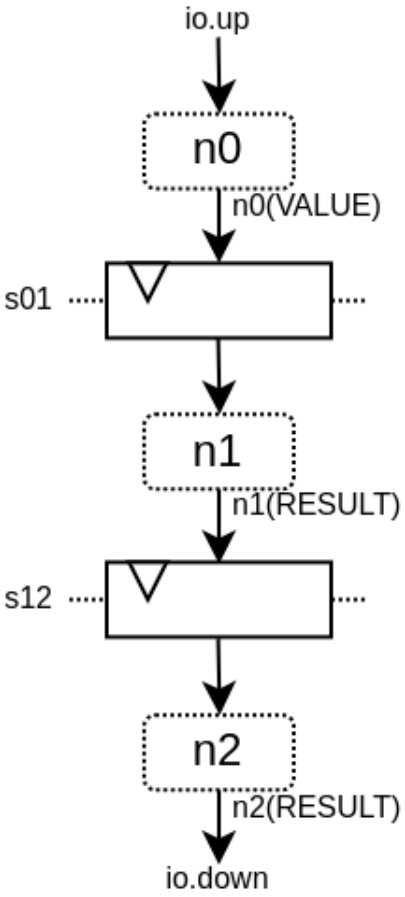
```
import spinal.core._
import spinal.core.sim._
import spinal.lib._
import spinal.lib.misc.pipeline._

class TopLevel extends Component {
  val io = new Bundle {
    val up = slave Stream (UInt(16 bits))
    val down = master Stream (UInt(16 bits))
  }

  // Let's define 3 Nodes for our pipeline
  val n0, n1, n2 = Node()

  // Let's connect those nodes by using simples registers
  val s01 = StageLink(n0, n1)
  val s12 = StageLink(n1, n2)

  // Let's define a few Payload things that can go through the pipeline
  val VALUE = Payload(UInt(16 bits))
  val RESULT = Payload(UInt(16 bits))

  // Let's bind io.up to n0
```

(续下页)

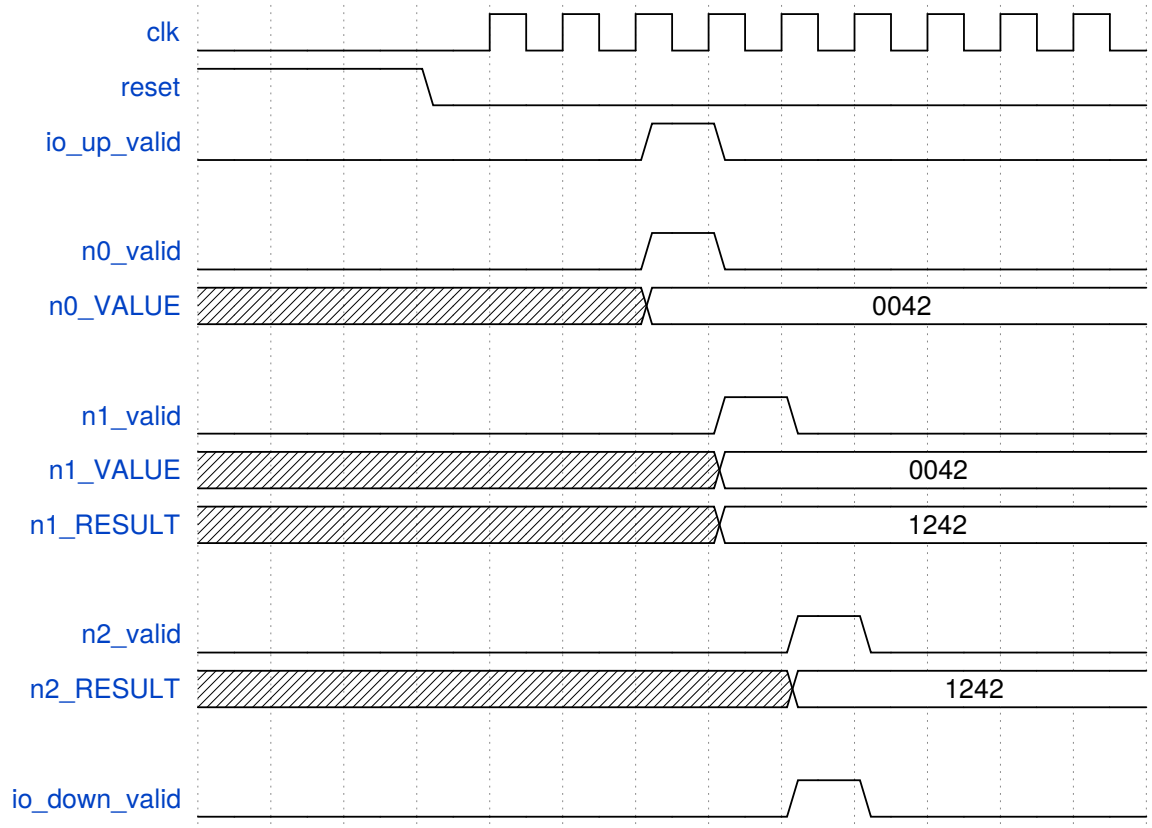
(接上页)

```
  io.up.ready := n0.ready
  n0.valid := io.up.valid
  n0(VALUE) := io.up.payload

  // Let's do some processing on n1
  n1(RESULT) := n1(VALUE) + 0x1200

  // Let's bind n2 to io.down
  n2.ready := io.down.ready
  io.down.valid := n2.valid
  io.down.payload := n2(RESULT)

  // Let's ask the builder to generate all the required hardware
  Builder(s01, s12)
}
```

这将产生以下硬件：



下面是一个仿真波形：



下面是相同的示例，但使用了更多的 API：

```
import spinal.core._
import spinal.core.sim._
import spinal.lib._
import spinal.lib.misc.pipeline._

class TopLevel extends Component {
  val VALUE = Payload(UInt(16 bits))

  val io = new Bundle {
    val up = slave Stream(VALUE)  // VALUE can also be used as a HardType
    val down = master Stream(VALUE)
  }

  // NodesBuilder will be used to register all the nodes created, connect them via␣
↪stages and generate the hardware
  val builder = new NodesBuilder()

  // Let's define a Node which connect from io.up
  val n0 = new builder.Node {
    arbitrateFrom(io.up)
    VALUE := io.up.payload
  }

  // Let's define a Node which do some processing
  val n1 = new builder.Node {
    val RESULT = insert(VALUE + 0x1200)
  }

  //  Let's define a Node which connect to io.down
  val n2 = new builder.Node {
    arbitrateTo(io.down)
    io.down.payload := n1.RESULT
```

(续下页)

(接上页)

```
  }

  // Let's connect those nodes by using registers stages and generate the related
↪hardware
  builder.genStagedPipeline()
}
```

10.16.2 Payload

Payload 对象用于引用可以通过流水线的数据。从技术上讲，Payload 是一个 HardType，它有一个名字，并被用作在流水线某个级中检索信号的“键”。

```
val PC = Payload(UInt(32 bits))
val PC_PLUS_4 = Payload(UInt(32 bits))

val n0, n1 = Node()
val s01 = StageLink(n0, n1)

n0(PC) := 0x42
n1(PC_PLUS_4) := n1(PC) + 4
```

请注意，我习惯于使用大写对 Payload 实例命名。这是为了让它非常明确，这不是一个硬件信号，更像是一个“键/类型”访问的东西。

10.16.3 Node

Node 主要托管有效/就绪仲裁信号，以及所有通过它的硬件信号所需的 Payload。

您可以通过以下方式访问其仲裁器：

API	访问	描述
node.valid	RW	指定节点上是否存在事务的信号。它是由上游逻辑驱动的。一旦置为 1，则它必须且仅能在 valid 和 ready 同时置位或 node.cancel 为高的周期后解除置位。valid 不依赖于 ready。
node.ready	RW	表示节点中的事务是否可以向下游传递的信号。它由下游驱动以创建反压。当没有事务（node.valid 被置 0）时，该信号无意义
node.cancel	RW	指定节点的事务是否正在从流水线中取消的信号。它由下游驱动。当没有事务时（node.valid 被置 0），该信号没有意义
node.isValid	RO	node.valid 的只读访问器
node.isReady	RO	node.ready 的只读访问器
node.isCancel	RO	node.cancel 的只读访问器
node.isFiring	RO	当节点事务成功继续传递时为 True（valid && ready && !cancel）。提交状态更改时非常有用。
node.isMoving	RO	当节点事务将不再存在于节点上时（从下一周期开始）为 True，要么是因为下游准备好接收事务，要么是因为事务已从流水线中取消。(valid && (ready || cancel)）用于“复位”(reset) 状态。
node.isCanceling	RO	当节点事务正在被取消时为 True。这意味着在将来的周期中它不会出现在流水线中的任何地方。

请注意，node.valid/node.ready 信号遵循与*Stream* 中相同的规范。

Node 的控制信号（valid/ready/cancel）和状态信号（isValid、isReady、isCancel、isFiring 等）是按需创建的。因此，当您想要读取某事物的状态时，使用状态信号很重要，只有当您想要驱动某事物时才使用控制信号。

以下是节点上可能出现的仲裁情况列表。valid/ready/cancel 定义了我们所处的状态，而 isFiring/isMoving 是这些状态的结果：

valid	ready	cancel	描述	isFiring	isMoving
0	X	X	无事务	0	0
1	1	0	正在进行	1	1
1	0	0	阻塞	0	0
1	X	1	取消	0	1

请注意，如果您想要建模诸如 CPU 级可能的阻塞和刷新的情况，可以查看 CtrlLink，因为它提供了执行此类操作的 API。

您可以通过以下方式访问由 Payload 引用的信号：

API	描述
node(Payload)	返回对应的硬件信号
node(Payload, Any)	与上述相同，但包括一个用作“次要键”的第二个参数。这有助于构建多通道硬件。例如，当您有一个多发射 CPU 流水线时，您可以使用通道 Int id 作为次要键
node.insert(Data)	返回一个新的 Payload 实例，该实例连接到给定的 Data 硬件信号

```
val n0, n1 = Node()

val PC = Payload(UInt(32 bits))
n0(PC) := 0x42
n0(PC, "true") := 0x42
n0(PC, 0x666) := 0xEE
val SOMETHING = n0.insert(myHardwareSignal) // This create a new Payload
when(n1(SOMETHING) === 0xFFAA){ ... }
```

您不仅可以手动方式来驱动/读取流水线的第一/最后一级的仲裁信号/数据，也有一些实用工具可以连接这些边界上的级。

API	描述
node.arbitrateFrom(Stream[T]])	由反压流驱动节点仲裁。
node.arbitrateFrom(Flow[T]])	由数据流驱动节点仲裁。
node.arbitrateTo(Stream[T]])	由节点驱动反压流仲裁。
node.arbitrateTo(Flow[T]])	由节点驱动数据流仲裁。
node.driveFrom(Stream[T]])((Node, T) => Unit)	由反压流驱动节点。提供的 lambda 函数可以用于连接数据
node.driveFrom(Flow[T]])((Node, T) => Unit)	与上述类似，但适用于 Flow
node.driveTo(Stream[T]])((T, Node) => Unit)	由节点驱动反压流。提供的 lambda 函数可以用于连接数据
node.driveTo(Flow[T]])((T, Node) => Unit)	与上述类似，但适用于 Flow

```
val n0, n1, n2 = Node()

val IN = Payload(UInt(16 bits))
val OUT = Payload(UInt(16 bits))

n1(OUT) := n1(IN) + 0x42

// Define the input / output stream that will be later connected to the pipeline
val up = slave Stream(UInt(16 bits))
val down = master Stream(UInt(16 bits)) // Note master Stream(OUT) is good as well

n0.driveFrom(up)((self, payload) => self(IN) := payload)
n2.driveTo(down)((payload, self) => payload := self(OUT))
```

为了减少冗长，在 Payload 与其数据表示之间有一组隐式转换，可在 Node 下使用：

```
val VALUE = Payload(UInt(16 bits))
val n1 = new Node {
    val PLUS_ONE = insert(VALUE + 1) // VALUE is implicitly converted into its␣
↪n1(VALUE) representation
}
```

您还可以通过导入它们来使用这些隐式转换：

```
val VALUE = Payload(UInt(16 bits))
val n1 = Node()

val n1Stuff = new Area {
    import n1._
    val PLUS_ONE = insert(VALUE) + 1 // Equivalent to n1.insert(n1(VALUE)) + 1
}
```

还有一个 API，它允许你创建新的 Area，这个 Area 无需导入就可提供给定节点实例的全部 API（包括隐式转换）：

```
val n1 = Node()
val VALUE = Payload(UInt(16 bits))

val n1Stuff = new n1.Area {
    val PLUS_ONE = insert(VALUE) + 1 // Equivalent to n1.insert(n1(VALUE)) + 1
}
```

当硬件具有可参数化的流水线位置时，这样的功能非常有用（请参阅重定时示例）。

10.16.4 Links

目前已经实现了一些不同的 Links（但您也可以创建自己的自定义 Links）。Links 的思想是以各种方式将两个节点连接在一起，它们通常有一个 *up* 节点和一个 *down* 节点。

DirectLink

非常简单，它只使用导线连接两个节点。以下是一个示例：

```
val c01 = DirectLink(n0, n1)
```

StageLink

这使用 data/valid 信号上的寄存器和 ready 信号上的一些仲裁连接了两个节点。

```
val c01 = StageLink(n0, n1)
```

S2mLink

这使用 ready 信号上的寄存器连接两个节点，这对于改进反压组合时序非常有用。

```
val c01 = S2mLink(n0, n1)
```

CtrlLink

这是一种特殊的 Link，用于连接两个节点，具有可选的流量控制/旁路逻辑。它的应用程序接口应该足够灵活，可以用它来实现 CPU 流水级。

以下是其流量控制 API（Bool 参数启用了相关功能）：

API	描述
haltWhen(Bool)	允许阻止当前传输事务（清除 up.ready down.valid）
throwWhen(Bool)	允许从流水线中取消当前事务（清除 down.valid，使事务驱动逻辑忘记其当前状态）
forgetOneWhen(Bool)	允许请求上游节点忘记其当前事务（但不会清除 down.valid）
ignoreReadyWhen(Bool)	允许忽略下游节点 ready（设置 up.ready 为 1）
duplicateWhen(Bool)	允许复制当前传输事务（清零 up.ready）
terminateWhen(Bool)	允许下游节点隐藏当前传输事务（清零 down.valid）

还要注意的是，如果要在条件作用域（例如在 when 语句中）进行通信流控制，可以调用以下函数：

- haltIt(), duplicateIt(), terminateIt(), forgetOneNow(), ignoreReadyNow(), throwIt()

```
val c01 = CtrlLink(n0, n1)

c01.haltWhen(something) // Explicit halt request

when(somethingElse) {
    c01.haltIt() // Conditional scope sensitive halt request, same as c01.
↪haltWhen(somethingElse)
}
```

您可以使用 node.up / node.down 查看哪些节点连接到了链接。

CtrlLink 还提供了访问 Payload 的 API：

API	描述
link(Payload)	与 Link.down(Payload) 相同
link(Payload, Any)	与 Link.down(Payload, Any) 相同
link.insert(Data)	与 Link.down.insert(Data) 相同
link.bypass(Payload)	允许在 link.up -> link.down 之间有条件地覆盖 Payload 值。例如，这可用于修复 CPU 流水线中的数据冲突。

```
val c01 = CtrlLink(n0, n1)

val PC = Payload(UInt(32 bits))
c01(PC) := 0x42
c01(PC, 0x666) := 0xEE

val DATA = Payload(UInt(32 bits))
// Let's say Data is inserted in the pipeline before c01
when(hazard) {
    c01.bypass(DATA) := fixedValue
}

// c01(DATA) and below will get the hazard patch
```

请注意，如果创建的 CtrlLink 不带节点参数，它将在内部创建自己的节点。

```
val decode = CtrlLink()
val execute = CtrlLink()

val d2e = StageLink(decode.down, execute.up)
```

其他链接

此外，还实现了 JoinLink / ForkLink。

您的自定义链接

您可以通过实现 Link 基类来实现自定义链接。

```
trait Link extends Area {
  def ups : Seq[Node]
  def downs : Seq[Node]

  def propagateDown(): Unit
  def propagateUp(): Unit
  def build() : Unit
}
```

不过，由于 API 还很新，后面可能会有一些变化。

10.16.5 Builders

要生成流水线硬件，您需要提供流水线中使用的所有链接列表。

```
// Let's define 3 Nodes for our pipeline
val n0, n1, n2 = Node()

// Let's connect those nodes by using simples registers
val s01 = StageLink(n0, n1)
val s12 = StageLink(n1, n2)

// Let's ask the builder to generate all the required hardware
Builder(s01, s12)
```

此外，还有一套“一体化“的构建工具，您可以利用它来帮助你自己。

StagePipeline

例如，StagePipeline 类有两个作用：- 它便于创建简单的流水线，这些流水线由后面部分组成：Node -> StageLink -> Node -> StageLink -> ⋯- 它可以动态地扩展流水线长度

以下是一个例子：

- 获取第 0 级输入
- 对第 1 级输入求和
- 对第 2 级输入求平方和
- 在第 3 级提供结果

```
// Let's define a few inputs/outputs
val a,b = in UInt(8 bits)
val result = out(UInt(16 bits))

// Let's create the pipelining tool.
val pip = new StagePipeline

// Let's insert a and b into the pipeline at stage 0
val A = pip(0).insert(a)
val B = pip(0).insert(b)
```

(续下页)

(接上页)

```
// Lets insert the sum of A and B into the stage 1 of our pipeline
val SUM = pip(1).insert(pip(1)(A) + pip(1)(B))

// Clearly, i don't want to say pip(x)(y) on every pipelined thing.
// So instead we can create a pip.Area(x) which will provide a scope which work in stage "x"
val onSquare = new pip.Area(2){
  val VALUE = insert(SUM * SUM)
}

// Lets assign our output result from stage 3
result := pip(3)(onSquare.VALUE)

// Now that everything is specified, we can build the pipeline
pip.build()
```

StageCtrlPipeline

与 StagePipeline 非常相似，但它用 StageLink 代替了 Nodes，允许在每个阶段上处理仲裁/旁路，这在 CPU 设计中非常有用。

以下是一个例子：

- 获取第 0 级输入
- 对第 1 级输入求和
- 检查总和值，最终在第 2 级放弃该次传输
- 在第 3 级提供结果

```
// Lets define a few inputs/outputs
val a,b = in UInt(8 bits)
val result = out(UInt(8 bits))

// Lets create the pipelining tool.
val pip = new StageCtrlPipeline

// Lets insert a and b into the pipeline at stage 0
val A = pip.ctrl(0).insert(a)
val B = pip.ctrl(0).insert(b)

// Lets sum A and B it stage 1
val onSum = new pip.Ctrl(1){
  val VALUE = insert(A + B)
}

// Lets check if the sum is bad (> 128) in stage 2 and if that is the case, we drop the transaction.
val onTest = new pip.Ctrl(2){
  val isBad = onSum.VALUE > 128
  throwWhen(isBad)
}

// Lets assign our output result from stage 3
result := pip.ctrl(3)(onSum.VALUE)

// Now that everything is specified, we can build the pipeline
pip.build()
```

10.16.6 组合能力（Composability）

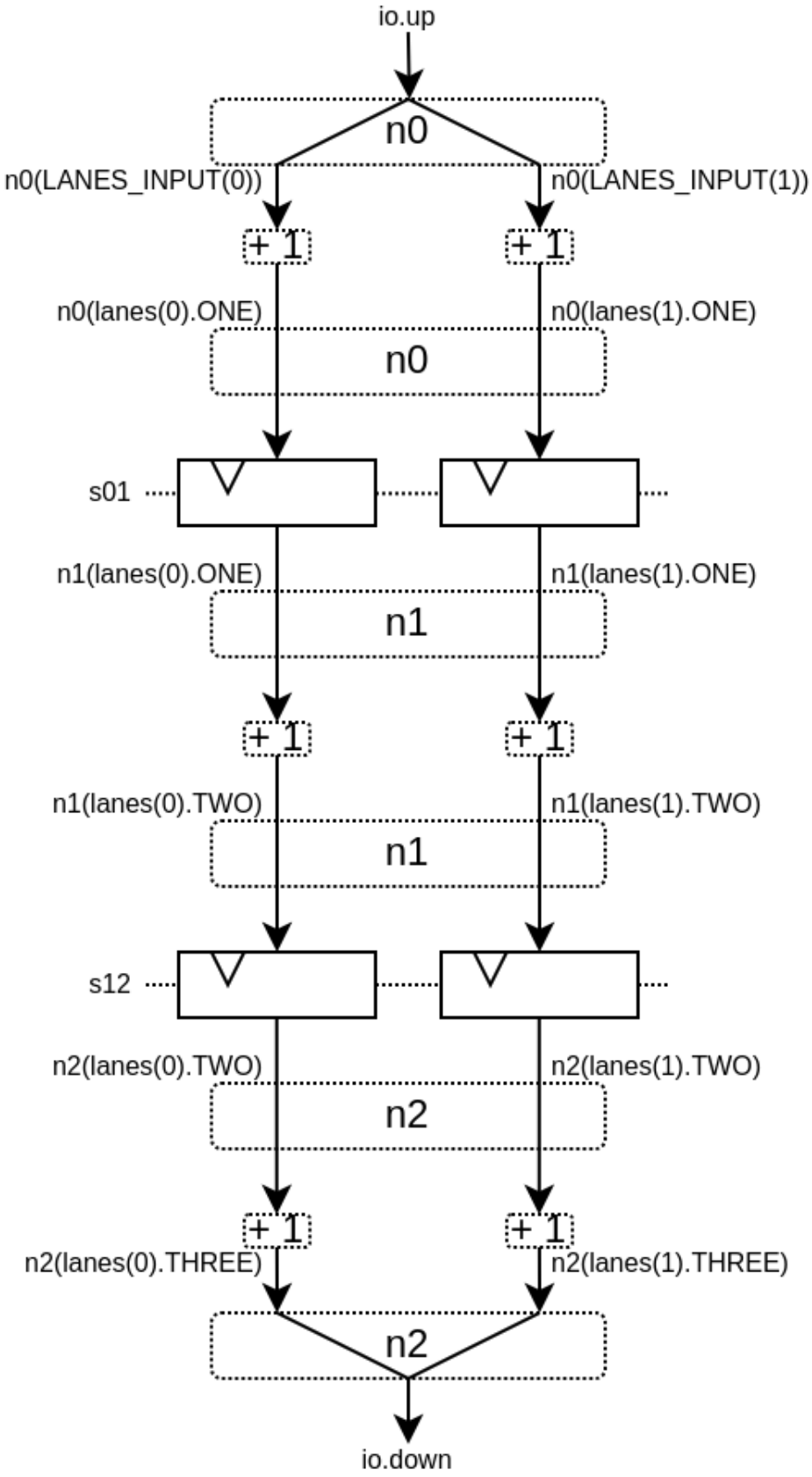
该 API 的一个优点是，它可以轻松地将多个并行事物组成一个流水线。这里的“组成“是指有时你设计的流水线需要进行并行处理。

试想一下，如果您需要对 4 对数字进行浮点乘法运算（稍后求和）。并且这 4 对数字是由一个数据流同时提供的，那么就不需要 4 条不同的流水线来进行乘法运算，而需要在同一条流水线上并行处理。

下面的示例展示了一种模式，它将多个通道组成一个流水线，来并行处理它们。

```
// This area allows to take a input value and do +1 +1 +1 over 3 stages.
// I know that's useless, but let's pretend that instead it does a multiplication␣
↪between two numbers over 3 stages (for FMax reasons)
class Plus3(INPUT: Payload[UInt], stage1: Node, stage2: Node, stage3: Node)␣
↪extends Area {
  val ONE = stage1.insert(stage1(INPUT) + 1)
  val TWO = stage2.insert(stage2(ONE) + 1)
  val THREE = stage3.insert(stage3(TWO) + 1)
}

// Let's define a component which takes a stream as input,
// which carries 'lanesCount' values that we want to process in parallel
// and put the result on an output stream
class TopLevel(lanesCount : Int) extends Component {
  val io = new Bundle {
    val up = slave Stream(Vec.fill(lanesCount)(UInt(16 bits)))
    val down = master Stream(Vec.fill(lanesCount)(UInt(16 bits)))
  }

  // Let's define 3 Nodes for our pipeline
  val n0, n1, n2 = Node()

  // Let's connect those nodes by using simples registers
  val s01 = StageLink(n0, n1)
  val s12 = StageLink(n1, n2)

  // Let's bind io.up to n0
  n0.arbitrateFrom(io.up)
  val LANES_INPUT = io.up.payload.map(n0.insert(_))

  // Let's use our "reusable" Plus3 area to generate each processing lane
  val lanes = for(i <- 0 until lanesCount) yield new Plus3(LANES_INPUT(i), n0, n1,␣
↪n2)

  // Let's bind n2 to io.down
  n2.arbitrateTo(io.down)
  for(i <- 0 until lanesCount) io.down.payload(i) := n2(lanes(i).THREE)

  // Let's ask the builder to generate all the required hardware
  Builder(s01, s12)
}
```

这将产生以下数据路径（假设 lanesCount = 2），这里没有给出仲裁：


io.up
n0
n0(LANES_INPUT(0))
n0(LANES_INPUT(1))
+ 1
+ 1
n0(lanes(0).ONE)
n0(lanes(1).ONE)
n0
s01
n1(lanes(0).ONE)
n1(lanes(1).ONE)
n1
+ 1
+ 1
n1(lanes(0).TWO)
n1(lanes(1).TWO)
n1
s12
n2(lanes(0).TWO)
n2(lanes(1).TWO)
n2
+ 1
+ 1
n2(lanes(0).THREE)
n2(lanes(1).THREE)
n2
io.down

10.16.7 重定时/可变长度

有时，你想设计一个流水线，但你并不真正知道关键路径在哪里，也不知道各阶段之间如何平衡。而且通常情况下，你无法依赖综合工具做好自动重定时工作。

因此，你需要一种简单的方法来构建流水线逻辑。

下面介绍如何使用此流水线 API：

```
// Define a component which will take a input stream of RGB value
// Process (~(R + G + B)) * 0xEE
// And provide that result into an output stream
class RgbToSomething(addAt : Int,
                     invAt : Int,
                     mulAt : Int,
                     resultAt : Int) extends Component {

  val io = new Bundle {
    val up = slave Stream(spinal.lib.graphic.Rgb(8, 8, 8))
    val down = master Stream (UInt(16 bits))
  }

  // Let's define the Nodes for our pipeline
  val nodes = Array.fill(resultAt+1)(Node())

  // Let's specify which node will be used for what part of the pipeline
  val insertNode = nodes(0)
  val addNode = nodes(addAt)
  val invNode = nodes(invAt)
  val mulNode = nodes(mulAt)
  val resultNode = nodes(resultAt)

  // Define the hardware which will feed the io.up stream into the pipeline
  val inserter = new insertNode.Area {
    arbitrateFrom(io.up)
    val RGB = insert(io.up.payload)
  }

  // sum the r g b values of the color
  val adder = new addNode.Area {
    val SUM = insert(inserter.RGB.r + inserter.RGB.g + inserter.RGB.b)
  }

  // flip all the bit of the RGB sum
  val inverter = new invNode.Area {
    val INV = insert(~adder.SUM)
  }

  // multiply the inverted bits with 0xEE
  val multiplier = new mulNode.Area {
    val MUL = insert(inverter.INV*0xEE)
  }

  // Connect the end of the pipeline to the io.down stream
  val resulter = new resultNode.Area {
    arbitrateTo(io.down)
    io.down.payload := multiplier.MUL
  }

  // Let's connect those nodes sequentially by using simples registers
  val links = for (i <- 0 to resultAt - 1) yield StageLink(nodes(i), nodes(i + 1))

  // Let's ask the builder to generate all the required hardware
```

(续下页)

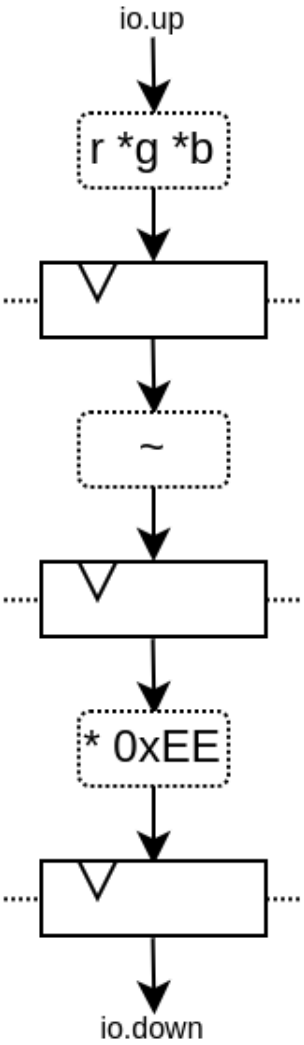
(接上页)

```
  Builder(links)
}
```

如果像这样生成该组件：

```
SpinalVerilog(
  new RgbToSomething(
    addAt    = 0,
    invAt    = 1,
    mulAt    = 2,
    resultAt = 3
  )
)
```

您将获得由 3 层寄存器（flip flop）分隔的 4 个处理阶段：



请注意，生成的硬件 verilog 还算干净（至少按我的标准来说是这样:P）：

```
// Generator : SpinalHDL dev    git head : 1259510dd72697a4f2c388ad22b269d4d2600df7
// Component : RgbToSomething
// Git hash  : 63da021a1cd082d22124888dd6c1e5017d4a37b2

`timescale 1ns/1ps

module RgbToSomething (
```

(续下页)

(接上页)

```
  input  wire          io_up_valid,
  output wire          io_up_ready,
  input  wire [7:0]    io_up_payload_r,
  input  wire [7:0]    io_up_payload_g,
  input  wire [7:0]    io_up_payload_b,
  output wire          io_down_valid,
  input  wire          io_down_ready,
  output wire [15:0]   io_down_payload,
  input  wire          clk,
  input  wire          reset
);

  wire       [7:0]     _zz_nodes_0_adder_SUM;
  reg        [15:0]    nodes_3_multiplier_MUL;
  wire       [15:0]    nodes_2_multiplier_MUL;
  reg        [7:0]     nodes_2_inverter_INV;
  wire       [7:0]     nodes_1_inverter_INV;
  reg        [7:0]     nodes_1_adder_SUM;
  wire       [7:0]     nodes_0_adder_SUM;
  wire       [7:0]     nodes_0_inserter_RGB_r;
  wire       [7:0]     nodes_0_inserter_RGB_g;
  wire       [7:0]     nodes_0_inserter_RGB_b;
  wire                 nodes_0_valid;
  reg                  nodes_0_ready;
  reg                  nodes_1_valid;
  reg                  nodes_1_ready;
  reg                  nodes_2_valid;
  reg                  nodes_2_ready;
  reg                  nodes_3_valid;
  wire                 nodes_3_ready;
  wire                 when_StageLink_l56;
  wire                 when_StageLink_l56_1;
  wire                 when_StageLink_l56_2;

  assign _zz_nodes_0_adder_SUM = (nodes_0_inserter_RGB_r + nodes_0_inserter_RGB_g);
  assign nodes_0_valid = io_up_valid;
  assign io_up_ready = nodes_0_ready;
  assign nodes_0_inserter_RGB_r = io_up_payload_r;
  assign nodes_0_inserter_RGB_g = io_up_payload_g;
  assign nodes_0_inserter_RGB_b = io_up_payload_b;
  assign nodes_0_adder_SUM = (_zz_nodes_0_adder_SUM + nodes_0_inserter_RGB_b);
  assign nodes_1_inverter_INV = (~ nodes_1_adder_SUM);
  assign nodes_2_multiplier_MUL = (nodes_2_inverter_INV * 8'hee);
  assign io_down_valid = nodes_3_valid;
  assign nodes_3_ready = io_down_ready;
  assign io_down_payload = nodes_3_multiplier_MUL;
  always @(*) begin
    nodes_0_ready = nodes_1_ready;
    if(when_StageLink_l56) begin
      nodes_0_ready = 1'b1;
    end
  end

  assign when_StageLink_l56 = (! nodes_1_valid);
  always @(*) begin
    nodes_1_ready = nodes_2_ready;
    if(when_StageLink_l56_1) begin
      nodes_1_ready = 1'b1;
    end
  end
```

(续下页)

(接上页)

```
  assign when_StageLink_l56_1 = (! nodes_2_valid);
  always @(*) begin
    nodes_2_ready = nodes_3_ready;
    if(when_StageLink_l56_2) begin
      nodes_2_ready = 1'b1;
    end
  end

  assign when_StageLink_l56_2 = (! nodes_3_valid);
  always @(posedge clk or posedge reset) begin
    if(reset) begin
      nodes_1_valid <= 1'b0;
      nodes_2_valid <= 1'b0;
      nodes_3_valid <= 1'b0;
    end else begin
      if(nodes_0_ready) begin
        nodes_1_valid <= nodes_0_valid;
      end
      if(nodes_1_ready) begin
        nodes_2_valid <= nodes_1_valid;
      end
      if(nodes_2_ready) begin
        nodes_3_valid <= nodes_2_valid;
      end
    end
  end

  always @(posedge clk) begin
    if(nodes_0_ready) begin
      nodes_1_adder_SUM <= nodes_0_adder_SUM;
    end
    if(nodes_1_ready) begin
      nodes_2_inverter_INV <= nodes_1_inverter_INV;
    end
    if(nodes_2_ready) begin
      nodes_3_multiplier_MUL <= nodes_2_multiplier_MUL;
    end
  end


endmodule
```

此外，您还可以轻松调整处理的级数和位置，例如，您可能希望将翻转的硬件逻辑移到与加法器相同级上。具体方法如下：

```
SpinalVerilog(
  new RgbToSomething(
    addAt    = 0,
    invAt    = 0,
    mulAt    = 1,
    resultAt = 2
  )
)
```

那么您可能需要移除输出寄存器级：

```
SpinalVerilog(
  new RgbToSomething(
    addAt    = 0,
    invAt    = 0,
    mulAt    = 1,
```

(续下页)

(接上页)

```
    resultAt = 1
  )
)
```

这个示例的一个特点是，中间值必须是 *addNode*。例如：

```
val addNode = nodes(addAt)
// sum the r g b values of the color
val adder = new addNode.Area {
  ...
}
```

遗憾的是，scala 不允许用 *new nodes(addAt).Area* 替换 *new addNode.Area*。一种变通方法是将其定义为一个类，比如：

```
class NodeArea(at : Int) extends NodeMirror(nodes(at))
val adder = new NodeArea(addAt) {
    ...
}
```

根据您的管道规模，它可以带来一些好处。

10.16.8 简单的 CPU 示例

下面是一个简单的 8 位 CPU 示例：

- 三级流水线（fetch, decode, execute）
- 嵌入的获取存储器
- add / jump / led /delay 指令

```
class Cpu extends Component {
  val fetch, decode, execute = CtrlLink()
  val f2d = StageLink(fetch.down, decode.up)
  val d2e = StageLink(decode.down, execute.up)

  val PC = Payload(UInt(8 bits))
  val INSTRUCTION = Payload(Bits(16 bits))

  val led = out(Reg(Bits(8 bits))) init(0)

  val fetcher = new fetch.Area {
    val pcReg = Reg(PC) init (0)
    up(PC) := pcReg
    up.valid := True
    when(up.isFiring) {
      pcReg := PC + 1
    }

    val mem = Mem.fill(256)(INSTRUCTION).simPublic
    INSTRUCTION := mem.readAsync(PC)
  }

  val decoder = new decode.Area {
    val opcode = INSTRUCTION(7 downto 0)
    val IS_ADD   = insert(opcode === 0x1)
    val IS_JUMP  = insert(opcode === 0x2)
    val IS_LED   = insert(opcode === 0x3)
    val IS_DELAY = insert(opcode === 0x4)
```

(续下页)

(接上页)

```
  }


  val alu = new execute.Area {
    val regfile = Reg(UInt(8 bits)) init(0)

    val flush = False
    for (stage <- List(fetch, decode)) {
      stage.throwWhen(flush, usingReady = true)
    }

    val delayCounter = Reg(UInt(8 bits)) init (0)

    when(isValid) {
      when(decoder.IS_ADD) {
        regfile := regfile + U(INSTRUCTION(15 downto 8))
      }
      when(decoder.IS_JUMP) {
        flush := True
        fetcher.pcReg := U(INSTRUCTION(15 downto 8))
      }
      when(decoder.IS_LED) {
        led := B(regfile)
      }
      when(decoder.IS_DELAY) {
        delayCounter := delayCounter + 1
        when(delayCounter === U(INSTRUCTION(15 downto 8))) {
          delayCounter := 0
        } otherwise {
          execute.haltIt()
        }
      }
    }
  }

  Builder(fetch, decode, execute, f2d, d2e)
}
```

下面是一个简单的测试平台，它实现了一个循环，使 led 计数值上升。

```
SimConfig.withFstWave.compile(new Cpu).doSim(seed = 2){ dut =>
  def nop() = BigInt(0)
  def add(value: Int) = BigInt(1 | (value << 8))
  def jump(target: Int) = BigInt(2 | (target << 8))
  def led() = BigInt(3)
  def delay(cycles: Int) = BigInt(4 | (cycles << 8))
  val mem = dut.fetcher.mem
  mem.setBigInt(0, nop())
  mem.setBigInt(1, nop())
  mem.setBigInt(2, add(0x1))
  mem.setBigInt(3, led())
  mem.setBigInt(4, delay(16))
  mem.setBigInt(5, jump(0x2))

  dut.clockDomain.forkStimulus(10)
  dut.clockDomain.waitSampling(100)
}
```

10.17 杂项

10.17.1 Plic 映射器

PLIC 映射器定义了 PLIC（平台级中断控制器）的寄存器生成和访问。

PlicMapper.apply

```
(bus: BusSlaveFactory, mapping: PlicMapping)(gateways : Seq[PlicGateway],
targets : Seq[PlicTarget])
```

PlicMapper 的参数：

- **bus**：连接此控制器的总线
- **mapping**：一个映射配置（见上文）
- **gateways**：用于生成总线访问控制的 PlicGateway（中断源）序列
- **targets**：生成总线访问控制的 PlicTarget 序列（如：多核）

它遵循 riscv 提供的接口：https://github.com/riscv/riscv-plic-spec/blob/master/riscv-plic.adoc

截至目前，有两种内存映射可用：

PlicMapping.sifive

遵循 SiFive 的 PLIC 映射（例如 E31 核心复合手册 ），基本上是一个成熟的 PLIC

PlicMapping.light

此映射生成更轻量级的 PLIC，但代价是缺少一些可选特性：

- not reading the interrupt’s priority
- not reading the interrupt’s pending bit (must use the claim/complete mechanism)
- not reading the target’s threshold

剩下的寄存器 & 逻辑会被生成.

10.17.2 插件

简介

对于某些设计，您可能希望通过使用某种插件来组合组件的硬件，而不是直接在组件中实现硬件。这可以提供一些关键特性：

- 您可以通过在组件的参数中添加新的插件来扩展组件的功能。例如，在 CPU 中添加浮点支持。
- 您可以通过使用另一组插件来轻松切换相同功能的各种实现。例如，某个 CPU 乘法器的实现可能在某些 FPGA 上表现良好，而其他实现可能在 ASIC 上表现良好。
- It avoid the very very very large hand written toplevel syndrome where everything has to be connected manually. Instead plugins can discover their neighborhood by looking/using the software interface of other plugins.

VexRiscv 和 NaxRiscv 项目就是这方面的例子。它们是具有大部分是空白的顶层的 CPU，其硬件部分通过插件注入。例如：

- PcPlugin
- FetchPlugin

- DecoderPlugin
- RegFilePlugin
- IntAluPlugin
- ...

And those plugins will then negotiate/propagate/interconnect to each others via their pool of services.

虽然 VexRiscv 使用严格的同步二阶段系统（设置 (setup)/构建 (build) 回调 (callback)），但 NaxRiscv 采用了一种更灵活的方法，使用 spinal.core.fiber API 来分叉实例化线程，这些线程可以联锁，以确保可行的实例化顺序。

插件 API(Plugin API) 提供了一个类似 NaxRiscv 的系统来定义使用插件的可组合组件。

执行顺序

主要思想是您有多个 2 执行环节：

- Setup phase, in which plugins can lock/retain each others. The idea is not to start negotiation / elaboration yet.
- Build phase, in which plugins can negotiation / elaboration hardware.

构建环节将不会在所有 FiberPlugin 完成其设置环节前启动。

```
class MyPlugin extends FiberPlugin {
  val logic = during setup new Area {
    // Here we are executing code in the setup phase
    awaitBuild()
    // Here we are executing code in the build phase
  }
}

class MyPlugin2 extends FiberPlugin {
  val logic = during build new Area {
    // Here we are executing code in the build phase
  }
}
```

简单示例

这是一个简单的虚设示例，其中包含一个将使用两个插件组合的 SubComponent：

```
import spinal.core._
import spinal.lib.misc.plugin._

// Let's define a Component with a PluginHost instance
class SubComponent extends Component {
  val host = new PluginHost()
}

// Let's define a plugin which create a register
class StatePlugin extends FiberPlugin {
  // during build new Area { body } will run the body of code in the Fiber build␣
↪phase, in the context of the PluginHost
  val logic = during build new Area {
    val signal = Reg(UInt(32 bits))
  }
}

// Let's define a plugin which will make the StatePlugin's register increment
class DriverPlugin extends FiberPlugin {
```

(续下页)

(接上页)

```
  // We define how to get the instance of StatePlugin.logic from the PluginHost.␣
↪It is a lazy val, because we can't evaluate it until the plugin is bound to its␣
↪host.
  lazy val sp = host[StatePlugin].logic.get

  val logic = during build new Area {
    // Generate the increment hardware
    sp.signal := sp.signal + 1
  }
}

class TopLevel extends Component {
  val sub = new SubComponent()

  // Here we create plugins and embed them in sub.host
  new DriverPlugin().setHost(sub.host)
  new StatePlugin().setHost(sub.host)
}
```

该 TopLevel 会生成以下 Verilog 代码：

```
module TopLevel (
  input  wire          clk,
  input  wire          reset
);


  SubComponent sub (
    .clk   (clk  ), // i
    .reset (reset)  // i
  );

endmodule

module SubComponent (
  input  wire          clk,
  input  wire          reset
);

  reg        [31:0]   StatePlugin_logic_signal; // Created by StatePlugin

  always @(posedge clk) begin
    StatePlugin_logic_signal <= (StatePlugin_logic_signal + 32'h00000001); //␣
↪incremented by DriverPlugin
  end
endmodule
```

请注意，每次在“构建期间”分叉一个实例化线程时，DriverPlugin.logic 线程的执行将在“sp”评估上被阻塞，直到 StatePlugin.logic 执行完成。

联锁/排序

插件可以通过 Retainer 实例相互联锁。每个插件实例都有一个内置锁，可以通过 retain/release 函数进行控制。

这是一个基于上面的 简单示例的例子，但这次，DriverPlugin 将通过由其他插件（在我们的例子中是 SetupPlugin）设置的数量对 StatePlugin.logic.signal 递增。为了确保 DriverPlugin 不会过早生成硬件，SetupPlugin 使用 DriverPlugin.retain/release 函数。

```
import spinal.core._
import spinal.lib.misc.plugin._
import spinal.core.fiber._

class SubComponent extends Component {
  val host = new PluginHost()
}

class StatePlugin extends FiberPlugin {
  val logic = during build new Area {
    val signal = Reg(UInt(32 bits))
  }
}

class DriverPlugin extends FiberPlugin {
  // incrementBy will be set by others plugin at elaboration time
  var incrementBy = 0
  // retainer allows other plugins to create locks, on which this plugin will wait␣
↪before using incrementBy
  val retainer = Retainer()

  val logic = during build new Area {
    val sp = host[StatePlugin].logic.get
    retainer.await()

    // Generate the incrementer hardware
    sp.signal := sp.signal + incrementBy
  }
}

// Let's define a plugin which will modify the DriverPlugin.incrementBy variable␣
↪because letting it elaborate its hardware
class SetupPlugin extends FiberPlugin {
  // during setup { body } will spawn the body of code in the Fiber setup phase␣
↪(it is before the Fiber build phase)
  val logic = during setup new Area {
    // *** Setup phase code ***
    val dp = host[DriverPlugin]

    // Prevent the DriverPlugin from executing its build's body (until release()␣
↪is called)
    val lock = dp.retainer()
    // Wait until the fiber phase reached build phase
    awaitBuild()

    // *** Build phase code ***
    // Let's mutate DriverPlugin.incrementBy
    dp.incrementBy += 1

    // Allows the DriverPlugin to execute its build's body
    lock.release()
  }
}
```

(续下页)

(接上页)

```
class TopLevel extends Component {
  val sub = new SubComponent()

  sub.host.asHostOf(
    new DriverPlugin(),
    new StatePlugin(),
    new SetupPlugin(),
    new SetupPlugin() // Let's add a second SetupPlugin, because we can
  )
}
```

这是生成的 verilog

```
module TopLevel (
  input  wire          clk,
  input  wire          reset
);


  SubComponent sub (
    .clk   (clk  ), // i
    .reset (reset)  // i
  );

endmodule

module SubComponent (
  input  wire          clk,
  input  wire          reset
);

  reg        [31:0]   StatePlugin_logic_signal;

  always @(posedge clk) begin
    StatePlugin_logic_signal <= (StatePlugin_logic_signal + 32'h00000002); // + 2␣
↪as we have two SetupPlugin
  end
endmodule
```

显然，这些示例对于它们的功能来说有些过度，总体上的思路更多地是：

- Negotiate / create interfaces between plugins (ex jump / flush ports)
- 安排实例化（例如解码/调度规范）
- 提供一个可扩展的分布式框架（最小硬编码）

CHAPTER 11

仿真

与往常一样，您可以使用标准仿真工具来仿真 SpinalHDL 生成的 VHDL/Verilog。然而，从 SpinalHDL 1.0.0 开始，该语言集成了一个 API 来编写测试平台并直接在 Scala 中测试您的硬件。该 API 提供了读取和写入 DUT 信号、分裂和合并仿真的进程、休眠和等待直到达到给定条件的功能。因此，使用 SpinalHDL 的仿真 API，可以轻松地将测试平台与最常见的 Scala 单元测试框架集成起来。

为了能够仿真用户定义的组件，SpinalHDL 使用外部 HDL 仿真器作为后台。目前支持四种仿真器：

- Verilator
- GHDL
- Icarus Verilog
- VCS **（实验性，自 SpinalHDL 1.7.0 起）**
- XSim **（实验性，自 SpinalHDL 1.7.0 起）**

使用外部 HDL 仿真器，可以直接测试生成的 HDL 源文件，而不会增加 SpinalHDL 代码库的复杂性。

11.1 用于仿真的 SBT 设置

要启用 SpinalSim，必须在 build.sbt 文件中添加以下行：

```
fork := true
```

此外，还必须在测试平台源文件中添加以下导入：

```
import spinal.core._
import spinal.core.sim._
```

另外，如果您要使用 gmake 而不是 make（例如 OpenBSD），您可以将 SPINAL_MAKE_CMD 环境变量设置为“gmake”

11.1.1 后台依赖的安装说明

GHDL 的设置和安装

备注： 如果您在 SpinalHDL 安装和设置 期间安装了推荐的 oss-cad-suite，您可以跳过下面的说明 - 但您需要激活 oss-cad-suite 环境。

尽管 GHDL 在 Linux 发行版软件包系统中普遍可用，但 SpinalHDL 依赖于 GHDL v0.37 发布后添加的 GHDL 代码库的错误修复。因此，建议从源代码安装 GHDL。还必须安装 C++ 库 boost-interprocess，它包含在类似 debian 发行版的 libboost-dev 包中。需要 boost-interprocess 来生成共享内存通信接口。

Linux

```
sudo apt-get install build-essential libboost-dev git
sudo apt-get install gnat # Ada compiler used to buid GHDL
git clone https://github.com/ghdl/ghdl.git
cd ghdl
mkdir build
cd build
../configure
make
sudo make install
```

还必须安装与您的 Java 版本相对应的 openjdk 软件包。

有关更多配置选项和 Windows 安装，请参阅 https://ghdl.github.io/ghdl/getting.html

Icarus Verilog 的设置和安装

备注： 如果您在 SpinalHDL 安装和设置 期间安装了推荐的 oss-cad-suite，您可以跳过下面的说明 - 但您需要激活 oss-cad-suite 环境。

在大多数最新的 Linux 发行版中，最新版本的 Icarus Verilog 通常可以通过软件包系统获得。还必须安装 C++ 库 boost-interprocess，它包含在类似 debian 发行版的 libboost-dev 包中。需要 boost-interprocess 来生成共享内存通信接口。

Linux

```
sudo apt-get install build-essential libboost-dev iverilog
```

Also the openjdk package that corresponds to your Java version has to be installed. Refer to https://iverilog.fandom.com/wiki/Installation_Guide for more information about Windows and installation from source.

VCS 仿真配置

环境变量

您应该确保提前定义了以下几个环境变量：

- VCS_HOME：VCS 安装的主路径。
- VERDI_HOME：Verdi 安装的主路径。
- 将 $VCS_HOME/bin 和 $VERDI_HOME/bin 添加到你的 PATH 中。

将以下路径添加到 LD_LIBRARY_PATH 变量的前部以启用 PLI 功能。

```
export LD_LIBRARY_PATH=$VERDI_HOME/share/PLI/VCS/LINUX64:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH=$VERDI_HOME/share/PLI/IUS/LINUX64:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH=$VERDI_HOME/share/PLI/lib/LINUX64:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH=$VERDI_HOME/share/PLI/Ius/LINUX64:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH=$VERDI_HOME/share/PLI/MODELSIM/LINUX64:$LD_LIBRARY_PATH
```

如果您遇到 Compilation of SharedMemIface.cpp failed 错误，请确保您已正确安装 C++ boost 库。头文件和库文件路径应分别添加到 CPLUS_INCLUDE_PATH、LIBRARY_PATH 和 LD_LIBRARY_PATH 中。

用户定义的环境设置

有时需要 VCS 环境设置文件 *synopsys_sim.setup* 来运行 VCS 仿真。此外，您可能希望在 VCS 开始编译之前运行一些脚本或代码来设置环境。您可以通过 *withVCSSimSetup* 来完成此操作。

```
val simConfig = SimConfig
  .withVCS
  .withVCSSimSetup(
    setupFile = "~/work/myproj/sim/synopsys_sim.setup",
    beforeAnalysis = () => { // this code block will be run before VCS analysis␣
↪step.
      "pwd".!
      println("Hello, VCS")
    }
  )
```

此方法将您自己的 *synopsys_sim.setup* 文件复制到 *workspacePath*（默认为 *simWorkspace*）目录下的 VCS 工作目录，并运行脚本。

VCS 标志

VCS 后台遵循三步编译流程：

1. 分析步骤：使用 vlogan 和“vhdlan“分析 HDL 模型。
2. 实例细化步骤：使用 vcs 细化模型并生成可执行的硬件模型。
3. 仿真步骤：运行仿真。

在每个步骤中，用户可以通过 VCSFlags 传递一些特定的标志，以启用一些功能，例如 SDF 反注释或多线程。

VCSFlags 采用三个参数，

名称	类型	描述
compileFlags	List[String]	传递标志到 vlogan 或 vhdlan。
elaborateFlags	List[String]	传递标志到 vcs。
runFlags	List[String]	传递标志给可执行硬件模型。

例如，您将 -kdb 标志传递给编译步骤和细化步骤，以进行 Verdi 调试，

```
val flags = VCSFlags(
    compileFlags = List("-kdb"),
    elaborateFlags = List("-kdb")
)

val config =
  SimConfig
    .withVCS(flags)
    .withFSDBWave
    .workspacePath("tb")
    .compile(UIntAdder(8))

...
```

波形生成

VCS 后端可以生成三种波形格式：VCD、VPD 和“FSDB“（需要 Verdi）。

您可以通过 SpinalSimConfig 的以下方法启用它们，

方法	描述
withWave	生成 VCD 波形。
withVPDWave	生成 VPD 波形。
withFSDBWave	生成 FSDB 波形。

此外，您可以使用 withWaveDepth(depth: Int) 来控制波形文件记录的深度。

Blackbox 仿真

Sometimes, IP vendors will provide you with some design entities in Verilog/VHDL format and you want to integrate them into your SpinalHDL design. The integration can done by following two ways:

1. 在 Blackbox 定义中，使用 addRTLPath(path: String) 将外部 Verilog/VHDL 文件分配给该黑盒。
2. 使用 SpinalReport 的 mergeRTLSource(fileName: String=null) 方法。

Verilator 的设置和安装

备注： 如果您在 SpinalHDL 安装和设置 期间安装了推荐的 oss-cad-suite，您可以跳过下面的说明 - 但您需要激活 oss-cad-suite 环境。

Linux 和 Windows 平台均支持 SpinalSim + Verilator。

建议使用的 Verilator 版本最老不低于 v4.218。虽然可以使用较旧的 verilator 版本，但 SpinalHDL 可以使用的一些可选的和 Scala 源文件相关功能（例如 Verilog $urandom 支持）可能不受旧版本 Verilator 支持，并且会在尝试仿真时会导致错误。

理想情况下，最新的 v4.xxx 和 v5.xxx 得到良好支持，并且应针对您遇到的任何问题打开错误报告工单。

Scala

不要忘记在 `build.sbt` 文件中添加以下内容：

```
fork := true
```

你总是需要在 Scala 测试平台中加入以下导入代码：

```
import spinal.core._
import spinal.core.sim._
```

Linux

您还需要安装最新版本的 Verilator：

```
sudo apt-get install git make autoconf g++ flex bison  # First time prerequisites
git clone http://git.veripool.org/git/verilator   # Only first time
unsetenv VERILATOR_ROOT  # For csh; ignore error if on bash
unset VERILATOR_ROOT  # For bash
cd verilator
git pull             # Make sure we're up-to-date
git checkout v4.218  # Can use newer v4.228 and v5.xxx
autoconf             # Create ./configure script
./configure
make -j$(nproc)
sudo make install
echo "DONE"
```

Windows

为了让 SpinalSim + Verilator 在 Windows 上工作，您必须执行以下操作：

- 安装 MSYS2
- 通过 MSYS2 获取 gcc/g++/verilator （对于 Verilator，您可以从源代码编译它）
- 将 MSYS2 的 `bin` 和 `usr\bin` 添加到 Windows `PATH` 中（即：`C:\msys64\usr\bin;C:\msys64\mingw64 \bin`）
- 检查 JAVA_HOME 环境变量是否指向 JDK 安装文件夹（即：`C:\Program Files\Java\jdk-13.0.2`）

然后您应该能够从 Scala 项目运行 SpinalSim + Verilator，而无需再使用 MSYS2。

从全新安装 MSYS2 MinGW 64 位开始，您必须在 MSYS2 MinGW 64 位 shell 中运行以下命令：

从 MinGW 包管理器安装

```
pacman -Syuu
# Close the MSYS2 shell once you're asked to
pacman -Syuu
pacman -S --needed base-devel mingw-w64-x86_64-toolchain \
                   git flex\
                   mingw-w64-x86_64-cmake

pacman -U http://repo.msys2.org/mingw/x86_64/mingw-w64-x86_64-verilator-4.032-1-
↪any.pkg.tar.xz

# Add C:\msys64\usr\bin;C:\msys64\mingw64\bin to your Windows PATH
```

从源码安装

```
pacman -Syuu
# Close the MSYS2 shell once you're asked to
pacman -Syuu
pacman -S --needed base-devel mingw-w64-x86_64-toolchain \
                   git flex\
                   mingw-w64-x86_64-cmake

git clone http://git.veripool.org/git/verilator
unset VERILATOR_ROOT
cd verilator
git pull
git checkout v4.218    # Can use newer v4.228 and v5.xxx
autoconf
./configure
export CPLUS_INCLUDE_PATH=/usr/include:$CPLUS_INCLUDE_PATH
export PATH=/usr/bin/core_perl:$PATH
cp /usr/include/FlexLexer.h ./src

make -j$(nproc)
make install
echo "DONE"
# Add C:\msys64\usr\bin;C:\msys64\mingw64\bin to your Windows PATH
```

重要： 确保您的 `PATH` 环境变量指向 JDK 1.8 并且不包含 JRE 安装。

重要： Adding the MSYS2 `bin` folders into your windows `PATH` could potentially have some side effects. This is why it is safer to add them as the last elements of the `PATH` to reduce their priority.

11.2 启动仿真器

11.2.1 简介

下面是一个硬件定义 + 测试平台的示例：

```
import spinal.core._

// Identity takes n bits in a and gives them back in z
class Identity(n: Int) extends Component {
  val io = new Bundle {
    val a = in Bits(n bits)
    val z = out Bits(n bits)
  }

  io.z := io.a
}
```

```
import spinal.core.sim._

object TestIdentity extends App {
  // Use the component with n = 3 bits as "dut" (device under test)
  SimConfig.withWave.compile(new Identity(3)).doSim{ dut =>
    // For each number from 3'b000 to 3'b111 included
```

(续下页)

(接上页)

```
    for (a <- 0 to 7) {
      // Apply input
      dut.io.a #= a
      // Wait for a simulation time unit
      sleep(1)
      // Read output
      val z = dut.io.z.toInt
      // Check result
      assert(z == a, s"Got $z, expected $a")
    }
  }
}
```

11.2.2 配置

SimConfig 将返回一个默认的仿真配置实例，您可以在该实例上调用多个函数来配置您的仿真过程：

语法	描述
withWave	打开仿真波形捕获与存储（默认格式）
withVcdWave	打开仿真波形捕获与存储（VCD 格式）
withFstWave	打开仿真波形捕获与存储（FST 二进制格式）
withConfig(SpinalConfig)	指定用于生成硬件的 SpinalConfig
allOptimisation	启用所有 RTL 编译优化以减少仿真时间（会增加编译时间）
workspacePath(path)	更改生成的仿真文件存放的文件夹
withVerilator	使用 Verilator 作为后台仿真器（默认）
withGhdl	使用 GHDL 作为后台仿真器
withIVerilog	使用 Icarus Verilog 作为后台仿真器
withVCS	使用 Synopsys VCS 作为后台仿真器

然后你可以调用 compile(rtl) 函数来编译硬件并预热仿真器。该函数将返回一个 SimCompiled 实例。

在此 SimCompiled 实例上，您可以使用以下函数进行仿真：

doSim[(simName[, seed])]{dut => /* main stimulus code */}
: 运行仿真，直到主线程运行完成并退出/返回。如果仿真完全卡住，它将检测并报告错误。只要例如时钟正在运行，仿真过程可以永远持续下去，因此建议使用 SimTimeout(cycles) 来限制可能的运行时间。

doSimUntilVoid[(simName[, seed])]{dut => ...}
: 运行仿真，直到通过调用 simSuccess() 或 simFailure() 结束。主激励线程可以继续或提前退出。只要有事件需要处理，仿真就会继续。如果完全卡住，仿真器将报告错误。

以下测试平台模板将使用以下顶层设计：

```
class TopLevel extends Component {
   val counter = out(Reg(UInt(8 bits)) init (0))
   counter := counter + 1
}
```

这是一个包含多个仿真配置的模板：

```
val spinalConfig = SpinalConfig(defaultClockDomainFrequency = FixedFrequency(10␣
↪MHz))

SimConfig
  .withConfig(spinalConfig)
  .withWave
```

(续下页)

(接上页)

```
  .allOptimisation
  .workspacePath("~/tmp")
  .compile(new TopLevel)
  .doSim { dut =>
    SimTimeout(1000)
    // Simulation code here
}
```

这是一个模板，其中仿真过程在主仿真线程执行完成后结束：

```
SimConfig.compile(new TopLevel).doSim { dut =>
  SimTimeout(1000)
  dut.clockDomain.forkStimulus(10)
  dut.clockDomain.waitSamplingWhere(dut.counter.toInt == 20)
  println("done")
}
```

这是一个模板，其中仿真过程通过显式调用 *simSuccess()* 结束：

```
SimConfig.compile(new TopLevel).doSimUntilVoid{ dut =>
  SimTimeout(1000)
  dut.clockDomain.forkStimulus(10)
  fork {
    dut.clockDomain.waitSamplingWhere(dut.counter.toInt == 20)
    println("done")
    simSuccess()
  }
}
```

注意它是否等同于：

```
SimConfig.compile(new TopLevel).doSim{ dut =>
  SimTimeout(1000)
  dut.clockDomain.forkStimulus(10)
  fork {
    dut.clockDomain.waitSamplingWhere(dut.counter.toInt == 20)
    println("done")
    simSuccess()
  }
  simThread.suspend() // Avoid the "doSim" completion
}
```

默认情况下，仿真文件应存放在 $WORKSPACE/$COMPILED 指定的目录中。

- $WORKSPACE 的默认值是 `simWorkspace`，但你可以利用 `SPINALSIM_WORKSPACE` 环境变量来指定一个不同的工作空间路径。
- $COMPILED 变量代表的是当前进行仿真的顶层模块的名称。
- 波形文件的存放路径会根据所选用的后端工具有所变化。在 Verilator 的情况下，它通常会被保存在默认的路径 `$WORKSPACE/$COMPILED/$TEST` 下。
- 对于 Verilator 后端，可以使用 `SimConfig.setTestPath(path)` 函数设置测试文件夹位置。
- You can retrieve the location of the test path during simulation by calling the currentTestPath() function.

11.2.3 在同一硬件上运行多个测试

```
val compiled = SimConfig.withWave.compile(new Dut)

compiled.doSim("testA") { dut =>
   // Simulation code here
}

compiled.doSim("testB") { dut =>
   // Simulation code here
}
```

11.2.4 从线程中抛出仿真成功或失败结果

在仿真过程中的任何时刻，您都可以调用 simSuccess 或 simFailure 来结束它。

当仿真时间太长时，可能会导致仿真失败，例如，因为测试平台正在等待从未发生的条件。为此，请调用 SimTimeout(maxDuration) 函数设置超时时间，其中 maxDuration 是仿真应被视为失败的时间（以仿真时间单位表示）。

例如，要使仿真在持续 1000 个时钟周期后失败：

```
val period = 10
dut.clockDomain.forkStimulus(period)
SimTimeout(1000 * period)
```

11.2.5 在失败之前捕获给定时间窗内的波形

如果您有一个很长时间的仿真，并且您不想捕获所有波形（太多错误，太慢），那么您主要有两种方法可以做到这一点。

要么您已经知道模拟在哪个 simTime 失败，在这种情况下您可以在测试平台中执行以下操作：

```
disableSimWave()
delayed(timeFromWhichIWantToCapture)(enableSimWave())
```

或者，您可以运行双步锁仿真，其中一个仿真的运行比另一个仿真的运行稍有延迟，一旦领先的模拟出现故障，它将开始记录波形。

为此，您可以使用 DualSimTracer 实用程序，其中包含已编译硬件的参数、故障前要捕获的时间窗大小以及随机种子。

这是一个例子：

```
package spinaldoc.libraries.sim

import spinal.core._
import spinal.core.sim._
import spinal.lib.misc.test.DualSimTracer

class Toplevel extends Component {
  val counter = out(Reg(UInt(16 bits))) init(0)
  counter := counter + 1
}

object Example extends App {
  val compiled = SimConfig.withFstWave.compile(new Toplevel())

  DualSimTracer(compiled, window = 10000, seed = 42){dut=>
```

(续下页)

(接上页)

```
    dut.clockDomain.forkStimulus(10)
    dut.clockDomain.onSamplings {
      val value = dut.counter.toInt

      if(value % 0x1000 == 0) {
        println(f"Value=0x$value%x at ${simTime()}")
      }

      // Throw a simulation failure after 64K cycles
      if(value == 0xFFFF) {
        simFailure()
      }
    }
  }
}
```

这将生成以下文件结构：

- simWorkspace/Toplevel/explorer/stdout.log : stdout of the simulation which is ahead
- simWorkspace/Toplevel/tracer/stdout.log : stdout of the simulation doing the wave tracing
- simWorkspace/Toplevel/tracer.fst : Waveform of the failure

scala 终端将显示仿真结果到标准输出。

11.3 仿真过程中访问信号

11.3.1 读写信号

顶层模块的每个接口信号都可以从 Scala 程序中读写：

语法	描述
`Bool.toBoolean`	将硬件 `Bool` 读取出来并转换为 Scala `Boolean` 值
`Bits/UInt/SInt.toInt`	将硬件 `BitVector` 读取出来并转换为 Scala 的 `Int` 值
`Bits/UInt/SInt.toLong`	将硬件 `BitVector` 读取出来并转换为 Scala 的 `Long` 值
`Bits/UInt/SInt.toBigInt`	将硬件中的 `BitVector` 值读取出来并转换为 Scala 中的 `BigInt` 值（无限位宽）
`SpinalEnumCraft.toEnum`	将硬件中的 `SpinalEnumCraft` 读取出来并转换为 Scala 的 `SpinalEnumElement` 值
`Bool #= Boolean`	将 Scala 的 `Boolean` 值赋值给硬件 `Bool`
`Bits/UInt/SInt #= Int`	将 Scala 的 `Int` 值赋值给硬件 `BitVector`
`Bits/UInt/SInt #= Long`	将 Scala 的 `Long` 值赋值给硬件 `BitVector`
`Bits/UInt/SInt #= BigInt`	将 Scala 的 `BigInt` 值赋值给硬件 `BitVector`
`SpinalEnumCraft #= SpinalEnumElement`	将 Scala 的 `SpinalEnumElement` 值赋值给硬件 `SpinalEnumCraft`
`Data.randomize()`	将随机值赋值给 SpinalHDL 硬件信号。

```
dut.io.a #= 42
dut.io.a #= 42l
dut.io.a #= BigInt("101010", 2)
dut.io.a #= BigInt("0123456789ABCDEF", 16)
println(dut.io.b.toInt)
```

11.3.2 访问组件层次结构内部的信号

要访问组件层次结构内部的信号，您必须首先将给定信号设置为 simPublic。

您可以直接在硬件描述中添加此 simPublic 标签：

```
object SimAccessSubSignal {
  import spinal.core.sim._

  class TopLevel extends Component {
    val counter = Reg(UInt(8 bits)) init(0) simPublic() // Here we add the␣
↪simPublic tag on the counter register to make it visible
    counter := counter + 1
  }

  def main(args: Array[String]) {
    SimConfig.compile(new TopLevel).doSim{dut =>
      dut.clockDomain.forkStimulus(10)

      for(i <- 0 to 3) {
        dut.clockDomain.waitSampling()
        println(dut.counter.toInt)
      }
    }
  }
}
```

或者您可以稍后在实例化仿真的顶层文件中添加它：

```
object SimAccessSubSignal {
  import spinal.core.sim._
  class TopLevel extends Component {
    val counter = Reg(UInt(8 bits)) init(0)
    counter := counter + 1
  }

  def main(args: Array[String]) {
    SimConfig.compile {
      val dut = new TopLevel
      dut.counter.simPublic()     // Call simPublic() here
      dut
    }.doSim{dut =>
      dut.clockDomain.forkStimulus(10)

      for(i <- 0 to 3) {
        dut.clockDomain.waitSampling()
        println(dut.counter.toInt)
      }
    }
  }
}
```

11.3.3 仿真中内存的加载和存储

可以在仿真中修改 Mem 硬件接口组件的内容。*data* 参数应该是一个字的位宽的值，*address* 是访问的字的地址。

没有 API 可以将地址和/或单个数据位转换为自然的字位宽以外的单位。

没有 API 可以用仿真中的'X'（未定义）状态来标记任何内存位置。

语法	描述
`Mem.getBigInt(address: Long): BigInt`	从仿真器的对应地址处读取一个字。
`Mem.setBigInt(address: Long, data: BigInt)`	在地址处向仿真器内的存储器写入一个字。

这是一个简单的使用内存的示例：

```
case class MemoryExample() extends Component {
  val wordCount = 64
  val io = new Bundle {
    val address = in port UInt(log2Up(wordCount) bit)
    val i = in port Bits(8 bit)
    val o = out port Bits(8 bit)
    val we = in port Bool()
  }

  val mem = Mem(Bits(8 bit), wordCount=wordCount)
  io.o := mem(io.address)
  when(io.we) {
    mem(io.address) := io.i
  }
}
```

设置仿真环境后，我们可以这样访问内存：

```
  SimConfig.withVcdWave.compile {
    val d = MemoryExample()
    // make memory accessible during simulation
    d.mem.simPublic()
    d
  }.doSim("example") { dut =>
```

我们可以在仿真期间读取数据，但必须注意确保数据可用（由于仿真器是事件驱动的，这可能会造成一个周期的延迟）：

```
    // do a write
    dut.io.we #= true
    dut.io.address #= 10
    dut.io.i #= 0xaf
    dut.clockDomain.waitSampling(2)
    // check written data is there
    assert(dut.mem.getBigInt(10) == 0xaf)
```

并且可以像这样写入内存：

```
    // set some data in memory
    dut.mem.setBigInt(15, 0xfe)
    // do a read to check if it's there
    dut.io.address #= 15
    dut.clockDomain.waitSampling(1)
    assert(dut.io.o.toBigInt == 0xfe)
```

必须注意的是，由于仿真器是事件驱动的，例如上面描述的读取操作必须延迟到该值在内存中实际可用后进行。

11.4 时钟域

11.4.1 激励函数 API

以下是 ClockDomain 激励函数的列表：

时钟域激励函数	描述
forkStimulus(period)	分裂一个仿真进程以生成时钟域激励（时钟、复位、softReset、clockEnable 信号）
forkSimSpeedPrinter(printPeriod)	分裂一个仿真进程，该进程将定期打印每秒实时千周期的仿真速度。printPeriod 是实时计数的秒数
clockToggle()	翻转时钟信号
fallingEdge()	清除时钟信号
risingEdge()	设置时钟信号
assertReset()	将复位信号设置为有效（电平）
deassertReset()	将复位信号设置为无效（电平）
assertClockEnable()	将时钟使能信号设置为有效（电平）
deassertClockEnable()	将时钟使能信号设置为有效（电平）
assertSoftReset()	将软复位信号设置为有效（电平）
deassertSoftReset()	将软复位信号设置为有效（电平）

11.4.2 等待相关 API

以下是 ClockDomain 实用工具函数的列表，您可以用它们来等待来自时钟域的给定事件：

时钟域等待函数	描述
waitSampling()	等待 ClockDomain 进行采样（有效时钟沿 && 无复位 && 时钟使能）
waitRisingEdge(cycleCount)	等待 cycleCount 个时钟的上升沿；如果没有另外指定，cycleCount 默认为 1 个周期。注意，cyclesCount = 0 是合法的，该功能对复位/softReset/clockEnable 不敏感
waitFallingEdge(cycleCount)	与 waitRisingEdge(cycleCount) 相同，但针对下降沿
waitActiveEdge(cycleCount)	与 waitRisingEdge(cycleCount) 相同，但针对 ClockDomainConfig 指定的边沿类型
waitInactiveEdge(cycleCount)	Same as waitActiveEdge(cycleCount) but for the edge level specified by the ClockDomainConfig
waitRisingEdgeWhere(condition)	与 waitRisingEdge() 功能相同，但要检查条件，上升沿发生时布尔 condition 必须为真
waitFallingEdgeWhere(condition)	与 waitFallingEdge() 相同，但针对的是下降沿
waitActiveEdgeWhere(condition)	与 waitRisingEdgeWhere 相同，但针对 ClockDomainConfig 指定的边沿类型
waitInactiveEdgeWhere(condition)	Same as waitActiveEdgeWhere, but for the edge level specified by the ClockDomainConfig
waitSamplingWhere(condition) : Boolean	等待直到时钟域采样并且给定条件为真
waitSamplingWhere(timeout)(condition) : Boolean	与上面定义的 waitSamplingWhere 相同，但阻塞不会超过 timeout 个周期。如果退出是因为超时，则返回 true

警告： 等待 API 的所有功能只能直接从线程内部调用，而不能从通过回调函数使用（通过回调 API 调用）。

11.4.3 回调函数 API

以下是 ClockDomain 实用工具函数的列表，您可以用它们来等待来自时钟域的给定事件：

时钟域回调函数	描述
onNextSampling { callback }	仅在下一个 ClockDomain 样本上执行一次回调代码（有效边沿 + 无复位 + 时钟使能）
onSamplings { callback }	每次 ClockDomain 采样时执行回调代码（有效边沿 + 无复位 + 时钟使能）
onActiveEdges { callback }	每次 ClockDomain 时钟符合其配置的边沿的条件时执行回调代码
onEdges { callback }	每次 ClockDomain 时钟出现上升沿或下降沿时执行回调代码
onRisingEdges { callback }	每次 ClockDomain 的时钟出现上升沿时执行回调代码
onFallingEdges { callback }	每次 ClockDomain 中时钟出现下降沿时执行回调代码
onSamplingWhile { callback : Boolean }	与 onSampling 相同，但您可以通过让回调返回 false 来停止它（永远）

11.4.4 默认时钟域

您可以访问顶层模块的默认 ClockDomain ，如下所示：

```
// Example of thread forking to generate a reset, and then toggling the clock each␣
↪5 time units.
// dut.clockDomain refers to the implicit clock domain created during component␣
↪instantiation.
fork {
  dut.clockDomain.assertReset()
  dut.clockDomain.fallingEdge()
  sleep(10)
  while(true) {
    dut.clockDomain.clockToggle()
    sleep(5)
  }
}
```

请注意，您还可以直接分裂标准复位/时钟产生进程：

```
dut.clockDomain.forkStimulus(period = 10)
```

如何等待时钟上升沿的示例：

```
dut.clockDomain.waitRisingEdge()
```

11.4.5 新时钟域

如果您的顶层模块中定义了一些未直接集成到其 ClockDomain 中的时钟和复位，您可以直接在测试平台中定义其相应的 ClockDomain：

```
// In the testbench
ClockDomain(dut.io.coreClk, dut.io.coreReset).forkStimulus(10)
```

11.5 全线程 API

在 SpinalSim 中，您可以使用多个线程来编写测试平台，方法与 SystemVerilog 类似，有点像 VHDL/Verilog 的 process/always 块。这允许您使用流畅的 API 编写并发任务并控制仿真时间。

11.5.1 分裂和合并仿真线程

```
// Create a new thread
val myNewThread = fork {
  // New simulation thread body
}

// Wait until `myNewThread` is execution is done.
myNewThread.join()
```

11.5.2 休眠和等待

```
// Sleep 1000 units of time
sleep(1000)

// waitUntil the dut.io.a value is bigger than 42 before continuing
waitUntil(dut.io.a > 42)
```

11.6 无线程 API

您可以使用一些函数来避免使用线程，但仍然允许您控制仿真时的流程。

无线程函数	描述
delayed(delay) { callback }	注册要在当前时间步 delay 步之后调用的回调代码。

与使用常规模拟线程 + 休眠相比，delayed 函数的优点是：

- 性能（无上下文切换）
- 内存使用情况（无本机 JVM 线程内存分配）

与 ClockDomain 对象相关的一些其他无线程函数被设计为*Callback API* 的一部分，而其他一些与仿真增量周期执行过程相关的函数被设计为*Sensitive API* 的一部分。

11.7 敏感 API

您可以注册要在仿真过程中每个仿真增量周期调用的回调函数：

敏感函数	描述
`forkSensitive { callback }`	注册要在仿真的每个仿真增量周期调用的回调代码
`forkSensitive { callback }`	注册要在仿真过程的每个仿真增量周期调用的回调代码，而回调返回值为 true（意味着应该为下一个仿真增量周期重新调度）

11.8 仿真器的具体细节

11.8.1 SpinalHDL 如何使用 Verilator 后端进行硬件仿真

1. SpinalHDL 在后台生成 DUT 的 Verilog 等效硬件模型，然后使用 Verilator 将其转换为 C++ 的周期精确模型。
2. C++ 模型被编译为共享对象 (.so)，该对象通过 JNI-FFI 绑定到 Scala。
3. 通过提供多线程仿真 API 来抽象原始的 Verilator API。

优点：

- 由于 Verilator 后端使用编译的 C++ 仿真模型，因此与大多数其他商业和免费模拟器相比，仿真速度很快。

限制：

- Verilator 仅接受可综合的 Verilog/System Verilog 代码。因此，在仿真中使用了包含不可综合语句的 Verilog 黑盒组件时必须特别小心。
- VHDL 黑盒无法模拟。
- 由于需要编译和链接生成的 C++ 模型，仿真的启动过程很慢。存在对增量编译和链接的一些支持，这可以在构建第一个仿真模型后为后续模型仿真提供加速。

11.8.2 SpinalHDL 如何使用 GHDL/Icarus Verilog 后端进行硬件仿真

1. 根据所选仿真器，SpinalHDL 生成 DUT 的 Verilog 或 VHDL 硬件模型。
2. HDL 模型加载到仿真器中。
3. 仿真器和 JVM 之间的通信是通过共享内存建立的。使用 VPI 向模拟器发出命令。

优点：

- GHDL 和 Icarus Verilog 都可以接受不可综合的 HDL 代码。
- 与 Verilator 相比，仿真启动过程要快得多。

限制：

- GHDL 仅接受 VHDL 代码。因此，该仿真器只能使用 VHDL 黑盒。
- Icarus Verilog 仅接受 Verilog 代码。因此，该仿真器只能使用 Verilog 黑盒。
- 与 Verilator 相比，仿真速度大约慢一个数量级。

最后，由于原生 Verilator API 相当粗糙，SpinalHDL 通过提供单线程和多线程仿真 API 对其进行抽象，以帮助用户构建测试平台实现。

11.8.3 SpinalHDL 如何使用 Synopsys VCS 后端进行硬件仿真

1. SpinalHDL 生成 DUT 的 Verilog/VHDL（取决于您的选择）硬件模型。
2. HDL 模型加载到仿真器中。
3. 仿真器和 JVM 之间的通信是通过共享内存建立的。使用 VPI 向模拟器发出命令。

优点：

- 支持 SystemVerilog/Verilog/VHDL 的所有语言特性。
- 支持加密 IP。
- 支持 FSDB 波形格式存储。
- 编译和仿真的性能好。

限制：

- Synopsys VCS 是一款 **商用**仿真工具。它是闭源的并且不是免费的。您必须拥有许可证才能 **合法**使用它。

在使用 VCS 作为仿真后台之前，请确保您已将系统环境检查为*VCS 环境*。

11.8.4 SpinalHDL 如何使用 Xilinx XSim 后端进行硬件仿真

1. SpinalHDL 生成 DUT 的 Verilog/VHDL（取决于您的选择）硬件模型。
2. HDL 模型加载到仿真器中。
3. 仿真器和 JVM 之间的通信是通过共享内存建立的。命令使用 XSI 发送到仿真器。

优点：

- 支持 Xilinx 内置原语和 IP 核。

限制：

- Xilinx XSim 是与 Vivado 一起安装的 **商用**工具。它是闭源的，并受许可条款的使用。您必须拥有许可证才能 **合法**使用它。
- 2019.1 之前的 Vivado 版本无法正常工作。

在使用 XSim 作为仿真后台之前，请确保您已完成以下步骤。1. 定义 VIVADO_HOME 环境变量以指定 vivado 所在的位置。例如，*export VIVADO_HOME=/d/Xilinx/Vivado/2022.1*（在 MSYS2 下）。2. 确保两个 vivado 路径在 PATH 内。对于 Windows MSYS2 用户，运行 shell 命令，如 *export PATH=$PATH:$VIVADO_HOME/bin:$VIVADO_HOME/lib/win64.o* 。对于 Linux 用户，只需用 *source* 执行位于 *VIVADO_HOME* 的 settings64.sh 文件。

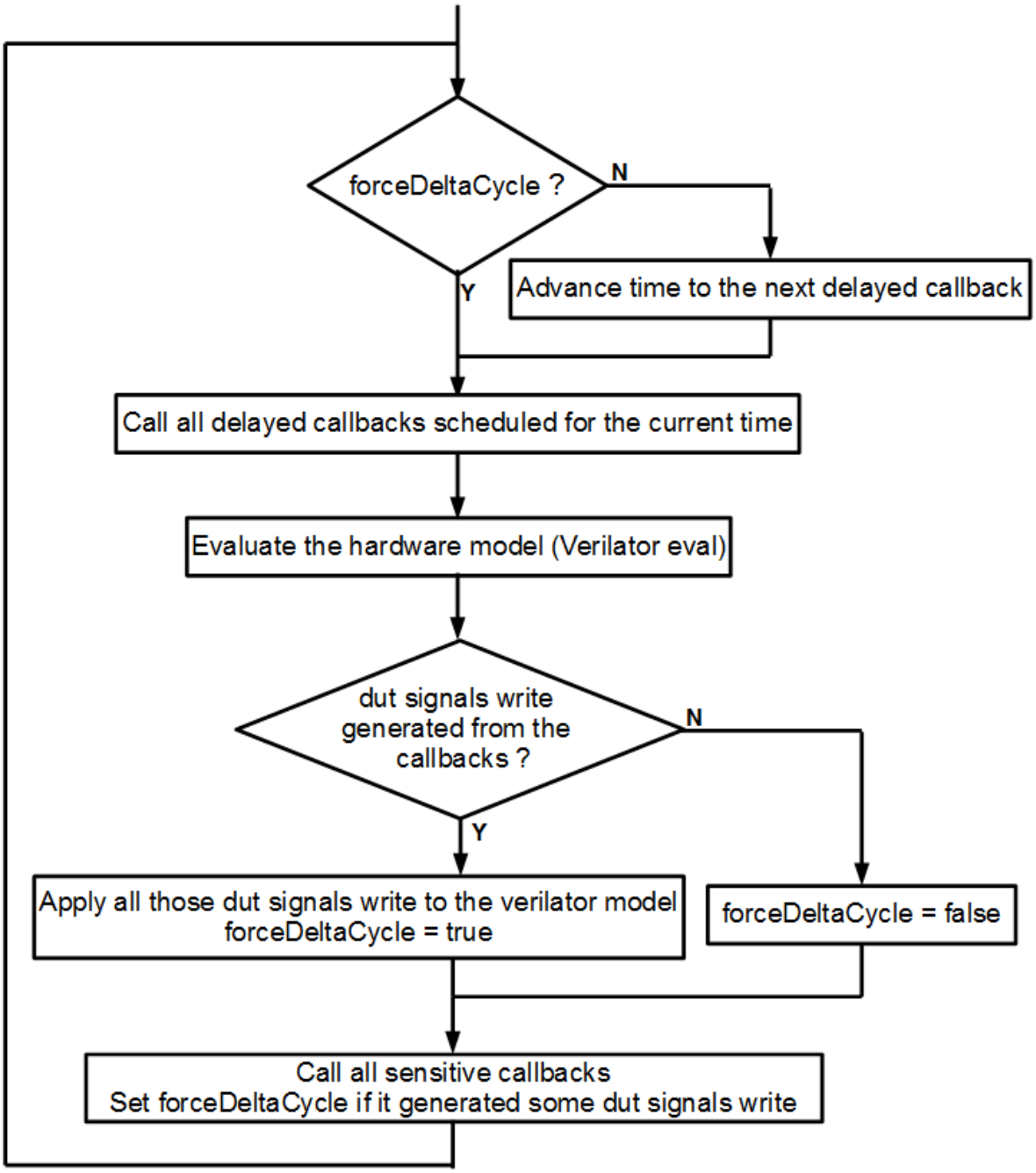
11.8.5 性能

当需要高性能仿真时，应使用 Verilator 作为后台。在像 Murax 这样的小型 SoC 上，英特尔 ® 酷睿 ™ i7-4720HQ 能够每秒模拟 120 万个时钟周期。然而，当 DUT 很简单并且需要仿真最多几千个时钟周期时，使用 GHDL 或 Icarus Verilog 可以产生更好的结果，因为它们的仿真负载开销较低。

11.9 仿真引擎

本页介绍了仿真引擎的内部结构。

仿真引擎通过在 Verilator C++ 仿真模型顶部应用以下循环来使用事件驱动仿真器（类似 VHDL/Verilog）：



在底层，仿真引擎管理以下原语：

- 敏感回调，允许用户在每个仿真增量周期中调用函数。
- 延迟回调，允许用户在未来的仿真时间调用函数。
- 仿真线程，允许用户描述并发的进程。
- 命令缓冲区，允许用户延迟对 DUT（被测设备）的写入访问，直到当前仿真增量周期结束。

这些原语有一些实际用途：

- 当给定条件发生时，例如时钟的上升沿，敏感回调可用于唤醒仿真线程。
- 延迟回调可用于安排激励，例如在给定时间后取消复位，或翻转时钟。
- 敏感回调和延迟回调都可用于恢复仿真线程。
- 例如，可以使用仿真线程来产生激励并检查 DUT 的输出值。
- 命令缓冲区的目的主要是避免 DUT 和测试平台之间的所有并发问题。

11.10 示例

11.10.1 异步加法器

此示例使用组合逻辑创建一个 `Component` ，对 3 个操作数执行一些简单的算术运算。

测试平台执行 100 次以下步骤：

- 将 `a`, `b`, 和 `c` 初始化为 0..255 范围内的随机整数。
- 激励 DUT (Device Under Test) 匹配 `a`, `b`, `c` 的输入。
- 等待 1 个仿真步长（以允许输入传播）。
- 检查输出是否正确。

```
import spinal.core._
import spinal.core.sim._

import scala.util.Random


object SimAsynchronousExample {
  class Dut extends Component {
    val io = new Bundle {
      val a, b, c = in UInt (8 bits)
      val result = out UInt (8 bits)
    }
    io.result := io.a + io.b - io.c
  }

  def main(args: Array[String]): Unit = {
    SimConfig.withWave.compile(new Dut).doSim{ dut =>
      var idx = 0
      while(idx < 100) {
        val a, b, c = Random.nextInt(256)
        dut.io.a #= a
        dut.io.b #= b
        dut.io.c #= c
        sleep(1) // Sleep 1 simulation timestep
        assert(dut.io.result.toInt == ((a + b - c) & 0xFF))
        idx += 1
      }
    }
  }
}
```

11.10.2 双时钟域 FIFO

此示例创建了一个专为跨时钟域而设计的 StreamFifoCC 以及 3 个仿真线程。

线程处理：

- 两个时钟域的管理
- 推入 FIFO
- 从 FIFO 弹出

FIFO 推送线程将输入随机化。

FIFO 弹出线程根据参考模型（普通的 scala.collection.mutable.Queue 实例）检查 DUT 的输出。

```
import spinal.core._
import spinal.core.sim._

import scala.collection.mutable.Queue


object SimStreamFifoCCExample {
  def main(args: Array[String]): Unit = {
    // Compile the Component for the simulator.
    val compiled = SimConfig.withWave.allOptimisation.compile(
      rtl = new StreamFifoCC(
        dataType = Bits(32 bits),
        depth = 32,
        pushClock = ClockDomain.external("clkA"),
        popClock = ClockDomain.external("clkB",withReset = false)
      )
    )

    // Run the simulation.
    compiled.doSimUntilVoid{dut =>
      val queueModel = mutable.Queue[Long]()

      // Fork a thread to manage the clock domains signals
      val clocksThread = fork {
        // Clear the clock domains' signals, to be sure the simulation captures␣
↪their first edges.
        dut.pushClock.fallingEdge()
        dut.popClock.fallingEdge()
        dut.pushClock.deassertReset()
        sleep(0)

        // Do the resets.
        dut.pushClock.assertReset()
        sleep(10)
        dut.pushClock.deassertReset()
        sleep(1)

        // Forever, randomly toggle one of the clocks.
        // This will create asynchronous clocks without fixed frequencies.
        while(true) {
          if(Random.nextBoolean()) {
            dut.pushClock.clockToggle()
          } else {
            dut.popClock.clockToggle()
          }
          sleep(1)
        }
      }
```

(续下页)

(接上页)

```
      // Push data randomly, and fill the queueModel with pushed transactions.
      val pushThread = fork {
        while(true) {
          dut.io.push.valid.randomize()
          dut.io.push.payload.randomize()
          dut.pushClock.waitSampling()
          if(dut.io.push.valid.toBoolean && dut.io.push.ready.toBoolean) {
            queueModel.enqueue(dut.io.push.payload.toLong)
          }
        }
      }

      // Pop data randomly, and check that it match with the queueModel.
      val popThread = fork {
        for(i <- 0 until 100000) {
          dut.io.pop.ready.randomize()
          dut.popClock.waitSampling()
          if(dut.io.pop.valid.toBoolean && dut.io.pop.ready.toBoolean) {
            assert(dut.io.pop.payload.toLong == queueModel.dequeue())
          }
        }
        simSuccess()
      }
    }
  }
}
```

11.10.3 单时钟域 FIFO

此示例创建一个 StreamFifo，并生成 3 个仿真线程。与*Dual clock fifo* 示例不同，此 FIFO 不需要复杂的时钟管理。

3 个仿真线程处理：

- 管理时钟/复位
- 推入 FIFO
- 从 FIFO 弹出

FIFO 推送线程将输入随机化。

FIFO 弹出线程根据参考模型（普通的 scala.collection.mutable.Queue 实例）检查 DUT 的输出。

```
import spinal.core._
import spinal.core.sim._

import scala.collection.mutable.Queue


object SimStreamFifoExample {
  def main(args: Array[String]): Unit = {
    // Compile the Component for the simulator.
    val compiled = SimConfig.withWave.allOptimisation.compile(
      rtl = new StreamFifo(
        dataType = Bits(32 bits),
        depth = 32
      )
    )
```

(续下页)

(接上页)

```
    // Run the simulation.
    compiled.doSimUntilVoid{dut =>
      val queueModel = mutable.Queue[Long]()

      dut.clockDomain.forkStimulus(period = 10)
      SimTimeout(1000000*10)

      // Push data randomly, and fill the queueModel with pushed transactions.
      val pushThread = fork {
        dut.io.push.valid #= false
        while(true) {
          dut.io.push.valid.randomize()
          dut.io.push.payload.randomize()
          dut.clockDomain.waitSampling()
          if(dut.io.push.valid.toBoolean && dut.io.push.ready.toBoolean) {
            queueModel.enqueue(dut.io.push.payload.toLong)
          }
        }
      }

      // Pop data randomly, and check that it match with the queueModel.
      val popThread = fork {
        dut.io.pop.ready #= true
        for(i <- 0 until 100000) {
          dut.io.pop.ready.randomize()
          dut.clockDomain.waitSampling()
          if(dut.io.pop.valid.toBoolean && dut.io.pop.ready.toBoolean) {
            assert(dut.io.pop.payload.toLong == queueModel.dequeue())
          }
        }
        simSuccess()
      }
    }
  }
}
```

11.10.4 同步加法器

这个例子创建了一个由时序逻辑组成的 `Component`，它对 3 个操作数进行了一些简单的算术运算。

测试平台执行 100 次以下步骤：

- 将 `a`, `b`, 和 `c` 初始化为 0..255 范围内的随机整数。
- 激励 DUT 匹配 `a`, `b`, `c` 的输入。
- 等待直到仿真再次对 DUT 的信号进行采样。
- 检查输出是否正确。

此示例与异步加法器*Asynchronous adder* 示例之间的主要区别在于，此 `Component` 必须使用 `forkStimulus` 来生成时钟信号，因为它在内部使用顺序逻辑。

```
import spinal.core._
import spinal.core.sim._

import scala.util.Random


object SimSynchronousExample {
  class Dut extends Component {
```

(续下页)

(接上页)

```
    val io = new Bundle {
      val a, b, c = in UInt (8 bits)
      val result = out UInt (8 bits)
    }
    io.result := RegNext(io.a + io.b - io.c) init(0)
  }

  def main(args: Array[String]): Unit = {
    SimConfig.withWave.compile(new Dut).doSim{ dut =>
      dut.clockDomain.forkStimulus(period = 10)

      var resultModel = 0
      for(idx <- 0 until 100) {
        dut.io.a #= Random.nextInt(256)
        dut.io.b #= Random.nextInt(256)
        dut.io.c #= Random.nextInt(256)
        dut.clockDomain.waitSampling()
        assert(dut.io.result.toInt == resultModel)
        resultModel = (dut.io.a.toInt + dut.io.b.toInt - dut.io.c.toInt) & 0xFF
      }
    }
  }
}
```

11.10.5 串口解码器

```
// Fork a simulation process which will analyze the uartPin and print transmitted␣
↪bytes into the simulation terminal.
fork {
  // Wait until the design sets the uartPin to true (wait for the reset effect).
  waitUntil(uartPin.toBoolean == true)

  while(true) {
    waitUntil(uartPin.toBoolean == false)
    sleep(baudPeriod/2)

    assert(uartPin.toBoolean == false)
    sleep(baudPeriod)

    var buffer = 0
    for(bitId <- 0 to 7) {
      if(uartPin.toBoolean)
        buffer |= 1 << bitId
      sleep(baudPeriod)
    }

    assert(uartPin.toBoolean == true)
    print(buffer.toChar)
  }
}
```

11.10.6 串口编码器

```
// Fork a simulation process which will get chars typed into the simulation␣
↪terminal and transmit them on the simulation uartPin.
fork {
  uartPin #= true
  while(true) {
    // System.in is the java equivalent of the C's stdin.
    if(System.in.available() != 0) {
      val buffer = System.in.read()
      uartPin #= false
      sleep(baudPeriod)

      for(bitId <- 0 to 7) {
        uartPin #= ((buffer >> bitId) & 1) != 0
        sleep(baudPeriod)
      }

      uartPin #= true
      sleep(baudPeriod)
    } else {
      sleep(baudPeriod * 10) // Sleep a little while to avoid polling System.in␣
↪too often.
    }
  }
}
```

CHAPTER 12

形式化验证

12.1 介绍

SpinalHDL 允许生成 SystemVerilog 断言 (SVA) 的子集。主要是断言（assert）、假设（assume）、覆盖（cover）和其他一些内容。

此外，它还提供了形式化验证后端，允许直接在开源 Symbi-Yosys 工具链中运行形式化验证。

12.2 形式化验证后端

您可以通过以下方式运行组件的形式化验证：

```
import spinal.core.formal._
FormalConfig.withBMC(15).doVerify(new Component {
    // Toplevel to verify
})
```

目前支持 3 种模式：

- withBMC(depth)
- withProve(depth)
- withCover(depth)

12.3 安装要求

要安装 Symbi-Yosys，您有几种选择。您可以在以下位置获取预编译包：

- https://github.com/YosysHQ/oss-cad-suite-build/releases
- https://github.com/YosysHQ/fpga-toolchain/releases（EOL - 由 oss-cad-suite 取代）

或者你可以从头开始编译：

- https://symbiyosys.readthedocs.io/en/latest/install.html

12.4 示例

12.4.1 外部断言

这是一个简单计数器和相应的形式化测试代码的示例。

```
import spinal.core._

// Here is our DUT
class LimitedCounter extends Component {
  // The value register will always be between [2:10]
  val value = Reg(UInt(4 bits)) init(2)
  when(value < 10) {
    value := value + 1
  }
}

object LimitedCounterFormal extends App {
  // import utilities to run the formal verification, but also some utilities to␣
↪describe formal stuff
  import spinal.core.formal._

  // Here we run a formal verification which will explore the state space up to 15␣
↪cycles to find an assertion failure
  FormalConfig.withBMC(15).doVerify(new Component {
    // Instantiate our LimitedCounter DUT as a FormalDut, which ensure that all␣
↪the outputs of the dut are:
    // - directly and indirectly driven (no latch / no floating wire)
    // - allows the current toplevel to read every signal across the hierarchy
    val dut = FormalDut(new LimitedCounter())

    // Ensure that the state space start with a proper reset
    assumeInitial(ClockDomain.current.isResetActive)

    // Check a few things
    assert(dut.value >= 2)
    assert(dut.value <= 10)
  })
}
```

12.4.2 内部断言

如果您愿意，可以将形式化验证语句直接嵌入到 DUT 中：

```
class LimitedCounterEmbedded extends Component {
  val value = Reg(UInt(4 bits)) init(2)
  when(value < 10) {
    value := value + 1
  }

  // That code block will not be in the SpinalVerilog netlist by default. (would␣
↪need to enable SpinalConfig().includeFormal. ...
  GenerationFlags.formal {
    assert(value >= 2)
    assert(value <= 10)
  }
}

object LimitedCounterEmbeddedFormal extends App {
```

(续下页)

(接上页)

```
  import spinal.core.formal._

  FormalConfig.withBMC(15).doVerify(new Component {
    val dut = FormalDut(new LimitedCounterEmbedded())
    assumeInitial(ClockDomain.current.isResetActive)
  })
}
```

12.4.3 外部激励

如果您的 DUT 有输入，您需要从测试代码中驱动它们。您可以使用所有常规硬件语句来执行此操作，但您也可以使用形式化验证中的 *anyseq*、*anyconst*、*allseq*、*allconst* 语句：

```
class LimitedCounterInc extends Component {
  // Only increment the value when the inc input is set
  val inc = in Bool()
  val value = Reg(UInt(4 bits)) init(2)
  when(inc && value < 10) {
    value := value + 1
  }
}

object LimitedCounterIncFormal extends App {
  import spinal.core.formal._

  FormalConfig.withBMC(15).doVerify(new Component {
    val dut = FormalDut(new LimitedCounterInc())
    assumeInitial(ClockDomain.current.isResetActive)
    assert(dut.value >= 2)
    assert(dut.value <= 10)

    // Drive dut.inc with random values
    anyseq(dut.inc)
  })
}
```

12.4.4 更多关于断言/past（以前某个时钟内的状态）的例子

例如，我们可以检查该值是否在正向计数（如果尚未达到 10）：

```
FormalConfig.withBMC(15).doVerify(new Component {
  val dut = FormalDut(new LimitedCounter())
  assumeInitial(ClockDomain.current.isResetActive)

  // Check that the value is incrementing.
  // hasPast is used to ensure that the past(dut.value) had at least one sampling␣
↪out of reset
  when(pastValid() && past(dut.value) =/= 10) {
    assert(dut.value === past(dut.value) + 1)
  }
})
```

12.4.5 假设内存中的内容

这是一个示例，我们希望防止值 1 出现在内存中：

```
class DutWithRam extends Component {
  val ram = Mem.fill(4)(UInt(8 bits))
  val write = slave(ram.writePort)
  val read = slave(ram.readAsyncPort)
}

object FormalRam extends App {
  import spinal.core.formal._

  FormalConfig.withBMC(15).doVerify(new Component {
    val dut = FormalDut(new DutWithRam())
    assumeInitial(ClockDomain.current.isResetActive)

    // assume that no word in the ram has the value 1
    for(i <- 0 until dut.ram.wordCount) {
      assumeInitial(dut.ram(i) =/= 1)
    }

    // Allow the write anything but value 1 in the ram
    anyseq(dut.write)
    clockDomain.withoutReset() { // As the memory write can occur during reset, we
↪need to ensure the assume apply there too
      assume(dut.write.data =/= 1)
    }

    // Check that no word in the ram is set to 1
    anyseq(dut.read.address)
    assert(dut.read.data =/= 1)
  })
}
```

12.5 实用工具和原语

12.5.1 断言/时钟/复位

断言（assert）是一直被时钟驱动的，但在复位期间被禁用。这也适用于假设（assume）和覆盖（cover）。如果您想在复位期间保持断言被检查，您可以执行以下操作：

```
ClockDomain.current.withoutReset() {
  assert(wuff === 0)
}
```

12.5.2 指定信号的初始值

For instance, for the reset signal of the current clockdomain (useful at the top)

```
ClockDomain.current.readResetWire initial(False)
```

12.5.3 指定初始假设

```
assumeInitial(clockDomain.isResetActive)
```

12.5.4 内存内容（Mem）检查

如果您的设计中有 Mem，并且想要检查其内容，可以通过以下方式进行：

```
// Manual access
for(i <- 0 until dut.ram.wordCount) {
  assumeInitial(dut.ram(i) =/= X) // No occurrence of the word X
}

assumeInitial(!dut.ram.formalContains(X)) // No occurrence of the word X

assumeInitial(dut.ram.formalCount(X) === 1) // only one occurrence of the word X
```

12.5.5 在复位的时候进行断言检查，可以这样做

```
ClockDomain.current.duringReset {
  assume(rawrrr === 0)
  assume(wuff === 3)
}
```

12.5.6 形式化验证的原语

语法	返回类型	描述
`assert(Bool)`		
`assume(Bool)`		
`cover(Bool)`		
`past(that : T, delay : Int)` `past(that : T)`	T	返回 `delay` 周期以前的“that“值。(默认 1 个周期)
`rose(that : Bool)`	Bool	当 `that` 从 False 变为 True 时返回 True
`fell(that : Bool)`	Bool	当 `that` 从 True 变为 False 时返回 True
`changed(that : Bool)`	Bool	当 `that` 当前值与上一个周期相比发生变化时返回 True
`stable(that : Bool)`	Bool	当 `that` 当前值与上一个周期相比没有改变时返回 True
`initstate()`	Bool	第一个周期时返回 True
`pastValid()`	Bool	当过去的值有效时返回 True（第一个周期为 False）。建议在 `past`, `rose`, `fell`, `changed` 和 `stable` 每次使用的地方均使用它。
`pastValidAfterReset()`	Bool	Similar to `pastValid`, where only difference is that this would take reset into account. Can be understood as `pastValid & past(!reset)`.

请注意，您可以对 past 的返回值使用 init 语句：

```
when(past(enable) init(False)) { ... }
```

12.6 局限性

不支持非时钟驱动的断言。但它们在第三方形式化验证示例中有这样使用，似乎主要与代码风格相关。

12.7 命名策略

所有与形式验证相关的函数都返回 Area 或 Composite（首选），并命名为 formalXXXX。`formalContext` 可用于创建形式相关逻辑，还有可能是 `formalAsserts`、`formalAssumes` 和 `formalCovers` 。

12.7.1 对于组件

证明模式中需要的，`Component` 内部所需的最少断言可以命名为 `formalAsserts`。

12.7.2 对于实现 IMasterSlave 的接口

There could be functions in name `formalAssertsMaster`, `formalAssertsSlave`, `formalAssumesMaster`, `formalAssumesSlave` or `formalCovers`. Master/Slave are target interface type, so that `formalAssertsMaster` can be understand as "formal verification assertions for master interface".

CHAPTER 13

示例

13.1 简单示例

13.1.1 APB3 定义

简介

此示例将展示定义一个 APB3 `Bundle` 的语句。

规范

ARM 关于 APB3 的端口规范如下：

信号名称	类型	驱动端	描述
PADDR	UInt(addressWidth bits)	Master	以字节为单位的地址
PSEL	Bits(selWidth)	Master	每个从端 1bit
PENABLE	Bool	Master	
PWRITE	Bool	Master	
PWDATA	Bits(dataWidth bits)	Master	
PREADY	Bool	Slave	
PRDATA	Bits(dataWidth bits)	Slave	
PSLVER-ROR	Bool	Slave	可选

实现

该规范表明 APB3 总线具有多种可能的配置。为了实现这一点，我们可以在 Scala 中定义一个配置类：

```
case class Apb3Config(
  addressWidth: Int,
  dataWidth: Int,
  selWidth: Int = 1,
  useSlaveError: Boolean = true
)
```

然后我们可以定义用于表示硬件总线的 APB3 `Bundle`：

```
case class Apb3(config: Apb3Config) extends Bundle with IMasterSlave {
  val PADDR = UInt(config.addressWidth bits)
  val PSEL = Bits(config.selWidth bits)
  val PENABLE = Bool()
  val PREADY = Bool()
  val PWRITE = Bool()
  val PWDATA = Bits(config.dataWidth bits)
  val PRDATA = Bits(config.dataWidth bits)
  val PSLVERROR  = if(config.useSlaveError) Bool() else null

  override def asMaster(): Unit = {
    out(PADDR,PSEL,PENABLE,PWRITE,PWDATA)
    in(PREADY,PRDATA)
    if(config.useSlaveError) in(PSLVERROR)
  }
}
```

用法

以下是该定义的用法示例：

```
case class Apb3User(apbConfig: Apb3Config) extends Component {
  val io = new Bundle {
    val apb = slave(Apb3(apbConfig))
  }

  io.apb.PREADY := True
  when(io.apb.PSEL(0) && io.apb.PENABLE) {
    // ...
  }

  io.apb.PRDATA := B(0)
}

object Apb3User extends App {
  val config = Apb3Config(
    addressWidth = 16,
    dataWidth = 32,
    selWidth = 1,
    useSlaveError = false
  )
  SpinalVerilog(Apb3User(config))
}
```

13.1.2 进位加法器

此示例定义了一个具有输入 a 和 b 以及输出 result 的组件。在任何时候，result 将是 a 和 b （组合逻辑）的和。该和是由进位加法器手动完成的。

```
case class CarryAdder(size : Int) extends Component {
  val io = new Bundle {
    val a = in UInt(size bits)
    val b = in UInt(size bits)
    val result = out UInt(size bits)  // result = a + b
  }

  var c = False   // Carry, like a VHDL variable
  for (i <- 0 until size) {
    // Create some intermediate value in the loop scope.
    val a = io.a(i)
    val b = io.b(i)

    // The carry adder's asynchronous logic
    io.result(i) := a ^ b ^ c
    c \= (a & b) | (a & c) | (b & c); // variable assignment
  }
}

object CarryAdderProject extends App {
  SpinalVhdl(CarryAdder(4))
}
```

13.1.3 颜色求和

首先，我们定义一个带加法运算符的 Color Bundle 。

```
case class Color(channelWidth: Int) extends Bundle {
  val r = UInt(channelWidth bits)
  val g = UInt(channelWidth bits)
  val b = UInt(channelWidth bits)

  def +(that: Color): Color = {
    val result = Color(channelWidth)
    result.r := this.r + that.r
    result.g := this.g + that.g
    result.b := this.b + that.b
    result
  }

  def clear(): Color = {
    this.r := 0
    this.g := 0
    this.b := 0
    this
  }
}
```

然后，我们定义一个带 sources 颜色向量输入、输出 sources 输入之和 result 的组件。

```
case class ColorSumming(sourceCount: Int, channelWidth: Int) extends Component {
  val io = new Bundle {
    val sources = in Vec(Color(channelWidth), sourceCount)
    val result = out(Color(channelWidth))
  }
```

(续下页)

(接上页)

```
  var sum = Color(channelWidth)
  sum.clear()
  for (i <- 0 until sourceCount) {
    sum \= sum + io.sources(i)
  }
  io.result := sum
}
```

13.1.4 带清零的计数器

此示例定义了一个具有 clear 输入和 value 输出的组件。每个时钟周期 value 输出都会递增，但是当 clear 为高电平时，value 被清零。

```
case class Counter(width: Int) extends Component {
  val io = new Bundle {
    val clear = in Bool()
    val value = out UInt(width bits)
  }

  val register = Reg(UInt(width bits)) init 0
  register := register + 1
  when(io.clear) {
    register := 0
  }
  io.value := register
}
```

13.1.5 锁相环黑盒和复位控制器

假设您想要定义一个 TopLevel 组件来实例化 PLL BlackBox，并利用它创建一个新的时钟域，该时钟域将由您的核心逻辑使用。我们还假设您希望将一个外部的异步复位适配到这个核心时钟域的同步复位源中。

本页的代码示例中将默认使用以下导入：

```
import spinal.core._
import spinal.lib._
```

PLL BlackBox 定义

这是定义 PLL BlackBox 的方法：

```
case class PLL() extends BlackBox {
  val io = new Bundle {
    val clkIn = in Bool()
    val clkOut = out Bool()
    val isLocked = out Bool()
  }
  noIoPrefix()
}
```

其对应下面的 VHDL 组件：

```
component PLL is
  port(
    clkIn     : in std_logic;
    clkOut    : out std_logic;
    isLocked : out std_logic
  );
end component;
```

TopLevel 定义

下面的例子展示了如何定义 TopLevel 来实例化锁相环，创建新的 ClockDomain 并将异步复位输入调整连接至同步复位端口：

```
case class TopLevel() extends Component {
  val io = new Bundle {
    val aReset = in Bool()
    val clk100Mhz = in Bool()
    val result = out UInt(4 bits)
  }

  // Create an Area to manage all clocks and reset things
  val clkCtrl = new Area {
    // Instantiate and drive the PLL
    val pll = new PLL
    pll.io.clkIn := io.clk100Mhz

    // Create a new clock domain named 'core'
    val coreClockDomain = ClockDomain.internal(
      name = "core",
      frequency = FixedFrequency(200 MHz)  // This frequency specification can be used
    )                                      // by coreClockDomain users to do some calculations

    // Drive clock and reset signals of the coreClockDomain previously created
    coreClockDomain.clock := pll.io.clkOut
    coreClockDomain.reset := ResetCtrl.asyncAssertSyncDeassert(
      input = io.aReset || ! pll.io.isLocked,
      clockDomain = coreClockDomain
    )
  }

  // Create a ClockingArea which will be under the effect of the clkCtrl.coreClockDomain
  val core = new ClockingArea(clkCtrl.coreClockDomain) {
    // Do your stuff which use coreClockDomain here
    val counter = Reg(UInt(4 bits)) init 0
    counter := counter + 1
    io.result := counter
  }
}
```

13.1.6 RGB 信号转灰度信号

让我们假设需要一个将 RGB 颜色转换为灰度并将其写入外部存储器的组件。

io 名称	方向	描述
clear	in	将所有内部寄存器清零
r,g,b	in	颜色输入
wr	out	写存储器
address	out	存储器地址，每个周期增加
data	out	存储器数据，灰度等级

```
case class RgbToGray() extends Component {
  val io = new Bundle {
    val clear = in Bool()
    val r,g,b = in UInt(8 bits)

    val wr = out Bool()
    val address = out UInt(16 bits)
    val data = out UInt(8 bits)
  }

  def scaled(value : UInt,by : Float): UInt = value * U((255*by).toInt,8 bits) >> 8

  val gray = RegNext(
    scaled(io.r,0.3f) +
      scaled(io.g,0.4f) +
      scaled(io.b,0.3f)
  )

  val address = CounterFreeRun(stateCount = 1 << 16)
  io.address := address
  io.wr := True
  io.data := gray

  when(io.clear) {
    gray := 0
    address.clear()
    io.wr := False
  }
}
```

13.1.7 正弦 ROM

让我们假设您想要生成一个正弦波，并且还有它的滤波版本（这在实际中完全没用，但让我们以此为例）。

参数名称	类型	描述
resolutionWidth	Int	用于表示数值的位宽
sampleCount	Int	一个正弦周期内的采样点数

IO 名称	方向	类型	描述
sin	out	SInt(resolutionWidth bits)	作为正弦波的输出
sinFiltered	out	SInt(resolutionWidth bits)	作为滤波后正弦波的输出

那么让我们定义一个 `Component`：

```
case class SineRom(resolutionWidth: Int, sampleCount: Int) extends Component {
  val io = new Bundle {
    val sin = out SInt(resolutionWidth bits)
    val sinFiltered = out SInt(resolutionWidth bits)
  }
...
```

为了在 sin 输出端口上输出正弦波，您可以定义一个 ROM，其包含正弦波一个周期内所有采样点（可能只是四分之一，但让我们以最简单的方式做事）。然后你可以用相位计数器读取该 ROM，这将生成你的正弦波。

```
  // Calculate values for the lookup table
  def sinTable = for(sampleIndex <- 0 until sampleCount) yield {
    val sinValue = Math.sin(2 * Math.PI * sampleIndex / sampleCount)
    S((sinValue * ((1<<resolutionWidth)/2-1)).toInt,resolutionWidth bits)
  }

  val rom =  Mem(SInt(resolutionWidth bits),initialContent = sinTable)
  val phase = Reg(UInt(log2Up(sampleCount) bits)) init 0
  phase := phase + 1

  io.sin := rom.readSync(phase)
```

随后生成 sinFiltered，例如您可以使用一个一阶低通滤波器：

```
  io.sinFiltered := RegNext(io.sinFiltered  - (io.sinFiltered  >> 5) + (io.sin >>␣
↪5)) init 0
```

这是完整的代码：

```
case class SineRom(resolutionWidth: Int, sampleCount: Int) extends Component {
  val io = new Bundle {
    val sin = out SInt(resolutionWidth bits)
    val sinFiltered = out SInt(resolutionWidth bits)
  }

  // Calculate values for the lookup table
  def sinTable = for(sampleIndex <- 0 until sampleCount) yield {
    val sinValue = Math.sin(2 * Math.PI * sampleIndex / sampleCount)
    S((sinValue * ((1<<resolutionWidth)/2-1)).toInt,resolutionWidth bits)
  }

  val rom =  Mem(SInt(resolutionWidth bits),initialContent = sinTable)
  val phase = Reg(UInt(log2Up(sampleCount) bits)) init 0
  phase := phase + 1

  io.sin := rom.readSync(phase)

  io.sinFiltered := RegNext(io.sinFiltered  - (io.sinFiltered  >> 5) + (io.sin >>␣
↪5)) init 0
}
```

13.2 中级示例

13.2.1 分形计算器

简介

此示例将展示使用数据流和定点计算实现一个未经优化的 Mandelbrot 分形计算器。

规范

该组件将接收像素任务的一个 `Stream`（其中包含 Mandelbrot 空间中的 XY 坐标），并将生成一个像素结果的一个 `Stream`（包含对应任务的迭代次数）。

定义组件的 IO 为：

IO 名称	方向	类型	描述
cmd	slave	Stream[PixelTask]	提供 XY 坐标来处理
rsp	master	Stream[PixelResult]	返回对应 cmd 交换所需的迭代次数

让我们设计 PixelTask `Bundle`：

元素名称	类型	描述
x	SFix	Mandelbrot 空间中的坐标
y	SFix	Mandelbrot 空间中的坐标

定义 PixelResult `Bundle`：

元素名称	类型	描述
iteration	UInt	求解 Mandelbrot 坐标所需的迭代次数

细化参数（泛型）

让我们定义为系统提供构造参数的类：

```
case class PixelSolverGenerics(fixAmplitude: Int,
                               fixResolution: Int,
                               iterationLimit: Int) {
  val iterationWidth = log2Up(iterationLimit+1)
  def iterationType = UInt(iterationWidth bits)
  def fixType = SFix(
    peak=fixAmplitude exp,
    resolution=fixResolution exp
  )
}
```

备注： iterationType 和 fixType 是可以调用来实例化新信号的函数，它就像 C 语言中的 typedef。

Bundle 定义

```
case class PixelTask(g: PixelSolverGenerics) extends Bundle {
  val x, y = g.fixType
}

case class PixelResult(g: PixelSolverGenerics) extends Bundle {
  val iteration = g.iterationType
}
```

组件实现

现在进行实现。下面是一个非常简单的实现，没有使用流水线处理/多线程技术。

```
case class PixelSolver(g: PixelSolverGenerics) extends Component {
  val io = new Bundle {
    val cmd = slave  Stream(PixelTask(g))
    val rsp = master Stream(PixelResult(g))
  }

  import g._

  // Define states
  val x, y = Reg(fixType) init(0)
  val iteration = Reg(iterationType) init(0)

  // Do some shared calculation
  val xx = x*x
  val yy = y*y
  val xy = x*y

  // Apply default assignment
  io.cmd.ready := False
  io.rsp.valid := False
  io.rsp.iteration := iteration

  when(io.cmd.valid) {
    // Is the mandelbrot iteration done ?
    when(xx + yy >= 4.0 || iteration === iterationLimit) {
      io.rsp.valid := True
      when(io.rsp.ready) {
        io.cmd.ready := True
        x := 0
        y := 0
        iteration := 0
      }
    } otherwise {
      x := (xx - yy + io.cmd.x).truncated
      y := (((xy) << 1) + io.cmd.y).truncated
      iteration := iteration + 1
    }
  }
}
```

13.2.2 串口

规范

本 UART 控制器教程基于 此文件 实现。

该实现的特征有：

- ClockDivider/Parity/StopBit/DataLength 的配置由组件输入设置。
- RXD 输入通过使用 N 个样本的采样窗口和多数投票法进行滤波。

该 UartCtrl 控制端口为：

名称	类型	描述
config	UartCtrlConfig	将所有配置发给控制器
write	Stream[Bits]	系统向控制器发送传输顺序所用的端口
read	Flow[Bits]	控制器发送已成功接收帧给系统所用的端口
uart	Uart	带 rxd/txd 的 Uart 接口

数据结构

在实现控制器本体之前，我们需要定义一些数据结构。

控制器构造参数

名称	类型	描述
dataWidthMax	Int	使用单个 UART 帧可以发送的最大数据位数
clockDividerWidth	Int	时钟分频器的位数
preSamplingSize	Int	采样窗口开始时要丢弃的样本数
samplingSize	Int	使用多个窗口中部的样本来获得过滤后的 RXD 值
postSamplingSize	Int	采样窗口结束时丢弃的样本数

为了使实现更容易，我们假设 `preSamplingSize + samplingSize + postSamplingSize` 始终是 2 的幂次方。这样做可以在一些地方跳过计数器清零操作。

同时，不需要将每个构造参数（泛型）一一添加到 `UartCtrl` 中，我们可以将它们分组到一个类中，该类将用作 `UartCtrl` 的单个参数。

```
case class UartCtrlGenerics(dataWidthMax: Int = 8,
                            clockDividerWidth: Int = 20, // baudrate = Fclk /␣
↪rxSamplePerBit / clockDividerWidth
                            preSamplingSize: Int = 1,
                            samplingSize: Int = 5,
                            postSamplingSize: Int = 2) {
  val rxSamplePerBit = preSamplingSize + samplingSize + postSamplingSize
  assert(isPow2(rxSamplePerBit))
  if((samplingSize % 2) == 0)
    SpinalWarning(s"It's not nice to have a odd samplingSize value (because of the␣
↪majority vote)")
}
```

UART 接口

让我们定义一个没有流量控制的 UART 接口线束。

```
case class Uart() extends Bundle with IMasterSlave {
  val txd = Bool()
  val rxd = Bool()

  override def asMaster(): Unit = {
    out(txd)
    in(rxd)
  }
}
```

UART 配置枚举

让我们定义奇偶校验和停止位枚举。

```
object UartParityType extends SpinalEnum(binarySequential) {
  val NONE, EVEN, ODD = newElement()
}

object UartStopType extends SpinalEnum(binarySequential) {
  val ONE, TWO = newElement()
  def toBitCount(that: C): UInt = (that === ONE) ? U"0" | U"1"
}
```

UartCtrl 配置线束

让我们定义一些线束，它们将被用于设置 UartCtrl 的 IO 单元。

```
case class UartCtrlFrameConfig(g: UartCtrlGenerics) extends Bundle {
  val dataLength = UInt(log2Up(g.dataWidthMax) bits) // Bit count = dataLength + 1
  val stop       = UartStopType()
  val parity     = UartParityType()
}

case class UartCtrlConfig(g: UartCtrlGenerics) extends Bundle {
  val frame         = UartCtrlFrameConfig(g)
  val clockDivider = UInt(g.clockDividerWidth bits) // see UartCtrlGenerics.
↪clockDividerWidth for calculation

  def setClockDivider(baudrate: Double, clkFrequency: HertzNumber = ClockDomain.
```

(续下页)

(接上页)

```
↪current.frequency.getValue): Unit = {
    clockDivider := (clkFrequency.toDouble / baudrate / g.rxSamplePerBit).toInt
  }
}
```

实现

在 UartCtrl 中会实例化 3 个东西：

- 一个时钟分频器，以 UART RX 采样率产生采样脉冲。
- 一个 UartCtrlTx 组件
- 一个 UartCtrlRx 组件

UARTCtrlTx

该 Component 的接口如下：

名称	类型	描述
configFrame	UartCtrlFrameConfig	包含数据位宽计数和奇偶校验位/停止位配置
samplingTick	Bool	以每 UART 波特 rxSamplePerBit 次脉冲为时间参考
write	Stream[Bits]	系统向控制器发出传输命令的端口
txd	Bool	UART txd 引脚

让我们定义用于存储 UartCtrlTx 状态的枚举：

```
object UartCtrlTxState extends SpinalEnum {
  val IDLE, START, DATA, PARITY, STOP = newElement()
}
```

让我们定义 UartCtrlTx 的框架：

```
class UartCtrlTx(g : UartCtrlGenerics) extends Component {
  import g._

  val io = new Bundle {
    val configFrame  = in(UartCtrlFrameConfig(g))
    val samplingTick = in Bool()
    val write        = slave Stream (Bits(dataWidthMax bits))
    val txd          = out Bool()
  }

  // Provide one clockDivider.tick each rxSamplePerBit pulses of io.samplingTick
  // Used by the stateMachine as a baud rate time reference
  val clockDivider = new Area {
    val counter = Reg(UInt(log2Up(rxSamplePerBit) bits)) init(0)
    val tick = False
    ..
  }

  // Count up each clockDivider.tick, used by the state machine to count up data␣
```

(续下页)

(接上页)

```
↪bits and stop bits
  val tickCounter = new Area {
    val value = Reg(UInt(Math.max(dataWidthMax, 2) bits))
    def reset() = value := 0
    ..
  }

  val stateMachine = new Area {
    import UartCtrlTxState._

    val state = RegInit(IDLE)
    val parity = Reg(Bool())
    val txd = True
    ..
    switch(state) {
      ..
    }
  }

  io.txd := RegNext(stateMachine.txd) init(True)
}
```

完整实现如下：

```
class UartCtrlTx(g : UartCtrlGenerics) extends Component {
  import g._

  val io = new Bundle {
    val configFrame  = in(UartCtrlFrameConfig(g))
    val samplingTick = in Bool()
    val write        = slave Stream Bits(dataWidthMax bits)
    val txd          = out Bool()
  }

  // Provide one clockDivider.tick each rxSamplePerBit pulse of io.samplingTick
  // Used by the stateMachine as a baudrate time reference
  val clockDivider = new Area {
    val counter = Reg(UInt(log2Up(rxSamplePerBit) bits)) init 0
    val tick = False
    when(io.samplingTick) {
      counter := counter - 1
      tick := counter === 0
    }
  }

  // Count up each clockDivider.tick, used by the state machine to count up data␣
↪bits and stop bits
  val tickCounter = new Area {
    val value = Reg(UInt(log2Up(Math.max(dataWidthMax, 2)) bits))
    def reset(): Unit = value := 0

    when(clockDivider.tick) {
      value := value + 1
    }
  }

  val stateMachine = new Area {
    import UartCtrlTxState._

    val state = RegInit(IDLE)
    val parity = Reg(Bool())
```

(续下页)

(接上页)

```
    val txd = True

    when(clockDivider.tick) {
      parity := parity ^ txd
    }

    io.write.ready := False
    switch(state) {
      is(IDLE) {
        when(io.write.valid && clockDivider.tick) {
          state := START
        }
      }
      is(START) {
        txd := False
        when(clockDivider.tick) {
          state := DATA
          parity := io.configFrame.parity === UartParityType.ODD
          tickCounter.reset()
        }
      }
      is(DATA) {
        txd := io.write.payload(tickCounter.value)
        when(clockDivider.tick) {
          when(tickCounter.value === io.configFrame.dataLength) {
            io.write.ready := True
            tickCounter.reset()
            when(io.configFrame.parity === UartParityType.NONE) {
              state := STOP
            } otherwise {
              state := PARITY
            }
          }
        }
      }
      is(PARITY) {
        txd := parity
        when(clockDivider.tick) {
          state := STOP
          tickCounter.reset()
        }
      }
      is(STOP) {
        when(clockDivider.tick) {
          when(tickCounter.value === UartStopType.toBitCount(io.configFrame.stop))
↪{
            state := io.write.valid ? START | IDLE
          }
        }
      }
    }
  }

  io.txd := RegNext(stateMachine.txd, True)
}
```

UartCtrlRx

该 Component 的接口如下：

名称	类型	描述
configFrame	UartCtrlFrameConfig	包含数据位宽和奇偶校验/停止位配置
samplingTick	Bool	以每 UART 波特 rxSamplePerBit 次脉冲为时间参考
read	Flow[Bits]	控制器发送已成功接收帧给系统所用的端口
rxd	Bool	UART rxd 引脚，与当前时钟域不同步

让我们定义用于存储 UartCtrlTx 状态的枚举：

```
object UartCtrlRxState extends SpinalEnum {
  val IDLE, START, DATA, PARITY, STOP = newElement()
}
```

让我们定义 UartCtrlRx 的框架：

```
class UartCtrlRx(g : UartCtrlGenerics) extends Component {
  import g._
  val io = new Bundle {
    val configFrame  = in(UartCtrlFrameConfig(g))
    val samplingTick = in Bool()
    val read         = master Flow (Bits(dataWidthMax bits))
    val rxd          = in Bool()
  }

  // Implement the rxd sampling with a majority vote over samplingSize bits
  // Provide a new sampler.value each time sampler.tick is high
  val sampler = new Area {
    val syncroniser = BufferCC(io.rxd)
    val samples     = History(that=syncroniser,when=io.samplingTick,
↪length=samplingSize)
    val value       = RegNext(MajorityVote(samples))
    val tick        = RegNext(io.samplingTick)
  }

  // Provide a bitTimer.tick each rxSamplePerBit
  // reset() can be called to recenter the counter over a start bit.
  val bitTimer = new Area {
    val counter = Reg(UInt(log2Up(rxSamplePerBit) bits))
    def reset() = counter := preSamplingSize + (samplingSize - 1) / 2 - 1)
    val tick = False
    ...
  }

  // Provide bitCounter.value that count up each bitTimer.tick, Used by the state␣
↪machine to count data bits and stop bits
  // reset() can be called to reset it to zero
  val bitCounter = new Area {
    val value = Reg(UInt(Math.max(dataWidthMax, 2) bits))
    def reset() = value := 0
    ...
```

(续下页)

(接上页)

```
  }

  val stateMachine = new Area {
    import UartCtrlRxState._

    val state    = RegInit(IDLE)
    val parity   = Reg(Bool())
    val shifter = Reg(io.read.payload)
    ...
    switch(state) {
      ...
    }
  }
}
```

完整实现如下：

```
class UartCtrlRx(g : UartCtrlGenerics) extends Component {
  import g._
  val io = new Bundle {
    val configFrame  = in(UartCtrlFrameConfig(g))
    val samplingTick = in Bool()
    val read         = master Flow Bits(dataWidthMax bits)
    val rxd          = in Bool()
  }

  // Implement the rxd sampling with a majority vote over samplingSize bits
  // Provide a new sampler.value each time sampler.tick is high
  val sampler = new Area {
    val synchronizer = BufferCC(io.rxd)
    val samples     = spinal.lib.History(that=synchronizer, when=io.samplingTick,␣
↪length=samplingSize)
    val value       = RegNext(MajorityVote(samples))
    val tick        = RegNext(io.samplingTick)
  }

  // Provide a bitTimer.tick each rxSamplePerBit
  // reset() can be called to recenter the counter over a start bit.
  val bitTimer = new Area {
    val counter = Reg(UInt(log2Up(rxSamplePerBit) bits))
    def reset(): Unit = counter := preSamplingSize + (samplingSize - 1) / 2 - 1
    val tick = False
    when(sampler.tick) {
      counter := counter - 1
      tick := counter === 0
    }
  }

  // Provide bitCounter.value that count up each bitTimer.tick, Used by the state␣
↪machine to count data bits and stop bits
  // reset() can be called to reset it to zero
  val bitCounter = new Area {
    val value = Reg(UInt(log2Up(Math.max(dataWidthMax, 2)) bits))
    def reset(): Unit = value := 0

    when(bitTimer.tick) {
      value := value + 1
    }
  }

  val stateMachine = new Area {
```

(续下页)

(接上页)

```
    import UartCtrlRxState._

    val state   = RegInit(IDLE)
    val parity  = Reg(Bool())
    val shifter = Reg(io.read.payload)

    // Parity calculation
    when(bitTimer.tick) {
      parity := parity ^ sampler.value
    }

    io.read.valid := False
    switch(state) {
      is(IDLE) {
        when(!sampler.value) {
          state := START
          bitTimer.reset()
          bitCounter.reset()
        }
      }
      is(START) {
        when(bitTimer.tick) {
          state := DATA
          bitCounter.reset()
          parity := io.configFrame.parity === UartParityType.ODD
          when(sampler.value) {
            state := IDLE
          }
        }
      }
      is(DATA) {
        when(bitTimer.tick) {
          shifter(bitCounter.value) := sampler.value
          when(bitCounter.value === io.configFrame.dataLength) {
            bitCounter.reset()
            when(io.configFrame.parity === UartParityType.NONE) {
              state := STOP
            } otherwise {
              state := PARITY
            }
          }
        }
      }
      is(PARITY) {
        when(bitTimer.tick) {
          state := STOP
          bitCounter.reset()
          when(parity =/= sampler.value) {
            state := IDLE
          }
        }
      }
      is(STOP) {
        when(bitTimer.tick) {
          when(!sampler.value) {
            state := IDLE
          }.elsewhen(bitCounter.value === UartStopType.toBitCount(io.configFrame.
↪stop)) {
            state := IDLE
            io.read.valid := True
          }
```

(续下页)

(接上页)

```
        }
      }
    }
  }
  io.read.payload := stateMachine.shifter
}
```

UartCtrl

让我们编写 UartCtrl 来实例化 UartCtrlRx 和 UartCtrlTx 部分，生成时钟分频器逻辑，并将它们相互连接。

```
class UartCtrl(g: UartCtrlGenerics=UartCtrlGenerics()) extends Component {
  val io = new Bundle {
    val config = in(UartCtrlConfig(g))
    val write  = slave(Stream(Bits(g.dataWidthMax bits)))
    val read   = master(Flow(Bits(g.dataWidthMax bits)))
    val uart   = master(Uart())
  }

  val tx = new UartCtrlTx(g)
  val rx = new UartCtrlRx(g)

  // Clock divider used by RX and TX
  val clockDivider = new Area {
    val counter = Reg(UInt(g.clockDividerWidth bits)) init 0
    val tick = counter === 0

    counter := counter - 1
    when(tick) {
      counter := io.config.clockDivider
    }
  }

  tx.io.samplingTick := clockDivider.tick
  rx.io.samplingTick := clockDivider.tick

  tx.io.configFrame := io.config.frame
  rx.io.configFrame := io.config.frame

  tx.io.write << io.write
  rx.io.read >> io.read

  io.uart.txd <> tx.io.txd
  io.uart.rxd <> rx.io.rxd
}
```

为了更简单地使用具有固定设置的 UART，我们引入了 UartCtrl 的伴生对象。这使我们能够以不同的参数集实例化 UartCtrl 组件，提供了额外的实例化方式。这里我们定义了 UartCtrlInitConfig，用它保存那些无法在运行时配置的组件的设置。请注意，如果需要一个能在运行时进行配置的 UART，您仍然可以像所有其他组件一样（通过 val uart = new UartCtrl() ）手动实例化 UartCtrl。

```
case class UartCtrlInitConfig(baudrate: Int = 0,
                              dataLength: Int = 1,
                              parity: UartParityType.E = null,
                              stop: UartStopType.E = null
                             ) {
  require(dataLength >= 1)
  def initReg(reg : UartCtrlConfig): Unit = {
```

(续下页)

(接上页)

```
    require(reg.isReg)
    if(baudrate != 0) reg.clockDivider init((ClockDomain.current.frequency.
↪getValue / baudrate / reg.g.rxSamplePerBit).toInt-1)
    if(dataLength != 1) reg.frame.dataLength init (dataLength - 1)
    if(parity != null) reg.frame.parity init parity
    if(stop != null) reg.frame.stop init stop
  }
}

object UartCtrl {
  def apply(config: UartCtrlInitConfig, readonly: Boolean = false): UartCtrl = {
    val uartCtrl = new UartCtrl()
    uartCtrl.io.config.setClockDivider(config.baudrate)
    uartCtrl.io.config.frame.dataLength := config.dataLength - 1
    uartCtrl.io.config.frame.parity := config.parity
    uartCtrl.io.config.frame.stop := config.stop
    if (readonly) {
      uartCtrl.io.write.valid := False
      uartCtrl.io.write.payload := B(0)
    }
    uartCtrl
  }
}
```

简单应用

用 115200-N-8-1 的参数综合 UartCtrl：

```
  val uartCtrl = UartCtrl(
    config=UartCtrlInitConfig(
      baudrate = 115200,
      dataLength = 7,
      parity = UartParityType.NONE,
      stop = UartStopType.ONE
    )
  )
```

如果您仅使用 txd 引脚，请添加：

```
  uartCtrl.io.uart.rxd := True
  io.tx := uartCtrl.io.uart.txd
```

相反，如果您仅使用 rxd 引脚：

```
  val uartCtrl = UartCtrl(
    config = UartCtrlInitConfig(
      baudrate = 115200,
      dataLength = 7,
      parity = UartParityType.NONE,
      stop = UartStopType.ONE
    ),
    readonly = true
  )
```

带 TestBench 的例子

下面是一个顶层的示例，它执行以下操作：

- 实例化 UartCtrl 并将其配置设置为 921600 baud/s，无奇偶校验，1 个停止位。
- 每次从 UART 接收到一个字节时，它都会将其写到 LED 输出上。
- 把 switches 输入值以每 2000 个周期发送到 UART。

```
case class UartCtrlUsageExample() extends Component {
  val io = new Bundle {
    val uart = master(Uart())
    val switches = in Bits(8 bits)
    val leds = out Bits(8 bits)
  }

  val uartCtrl = new UartCtrl()
  // set config manually to show that this is still OK
  uartCtrl.io.config.setClockDivider(921600)
  uartCtrl.io.config.frame.dataLength := 7  // 8 bits
  uartCtrl.io.config.frame.parity := UartParityType.NONE
  uartCtrl.io.config.frame.stop := UartStopType.ONE
  uartCtrl.io.uart <> io.uart

  // Assign io.led with a register loaded each time a byte is received
  io.leds := uartCtrl.io.read.toReg()

  // Write the value of switch on the uart each 2000 cycles
  val write = Stream(Bits(8 bits))
  write.valid := CounterFreeRun(2000).willOverflow
  write.payload := io.switches
  write >-> uartCtrl.io.write
}

object UartCtrlUsageExample extends App {
  SpinalConfig(
    defaultClockDomainFrequency = FixedFrequency(100 MHz)
  ).generateVhdl(UartCtrlUsageExample())
}
```

您可以在 这里 为这个小 UartCtrlUsageExample 获取一个简单的 VHDL 测试文件。

额外奖励：享受 Stream 带来的乐趣

如果您想将从 UART 接收到的数据入队：

```
  val queuedReads = uartCtrl.io.read.toStream.queue(16)
```

如果要在写接口上添加一个队列并做一些流控制：

```
  val writeCmd = Stream(Bits(8 bits))
  val stopIt = Bool()
  writeCmd.queue(16).haltWhen(stopIt) >> uartCtrl.io.write
```

如果您想在发送 switches 值之前发送 0x55 标头，可以将上例中的写生成器替换为：

```
  val write = Stream(Fragment(Bits(8 bits)))
  write.valid := CounterFreeRun(4000).willOverflow
  write.fragment := io.switches
  write.last := True
  write.stage().insertHeader(0x55).toStreamOfFragment >> uartCtrl.io.write
```

13.2.3 VGA

简介

VGA 接口正在成为一种“濒危”的技术，但实现 VGA 控制器仍然是一个很好的练习。

有关 VGA 协议的说明可以在 此处 找到。

本 VGA 控制器教程基于 此文件 实现。

数据结构

在实现控制器本体之前，我们需要定义一些数据结构。

RGB 颜色

首先，我们需要一个三通道的颜色结构（红、绿、蓝）。该数据结构将用于向控制器提供像素，也将由 VGA 总线使用。

```
case class RgbConfig(rWidth : Int, gWidth : Int, bWidth : Int) {
  def getWidth = rWidth + gWidth + bWidth
}

case class Rgb(c: RgbConfig) extends Bundle {
  val r = UInt(c.rWidth bits)
  val g = UInt(c.gWidth bits)
  val b = UInt(c.bWidth bits)
}
```

VGA 总线

io 名称	驱动	描述
vSync	master	垂直同步，表示新帧的开始
hSync	master	水平同步，表示新行的开始
colorEn	master	当位于界面可见部分时为高
color	master	携带颜色信息，当 colorEn 为低时无效

```
case class Vga(rgbConfig: RgbConfig) extends Bundle with IMasterSlave {
  val vSync = Bool()
  val hSync = Bool()

  val colorEn = Bool()
  val color   = Rgb(rgbConfig)

  override def asMaster() : Unit = this.asOutput()
}
```

此 Vga `Bundle` 使用 `IMasterSlave` 特质，它允许您使用以下命令创建主/从 VGA 接口：

```
master(Vga(...))
slave(Vga(...))
```

VGA 时序

VGA 接口使用 8 种不同的时序进行驱动。以下是一个能够携带它们的 Bundle 的简单示例。

```
case class VgaTimings(timingsWidth: Int) extends Bundle {
  val hSyncStart  = UInt(timingsWidth bits)
  val hSyncEnd    = UInt(timingsWidth bits)
  val hColorStart = UInt(timingsWidth bits)
  val hColorEnd   = UInt(timingsWidth bits)
  val vSyncStart  = UInt(timingsWidth bits)
  val vSyncEnd    = UInt(timingsWidth bits)
  val vColorStart = UInt(timingsWidth bits)
  val vColorEnd   = UInt(timingsWidth bits)
}
```

但这并不是一种很好的指定方式，因为在垂直和水平时序方面存在冗余。

让我们写得更清晰一些：

```
case class VgaTimingsHV(timingsWidth: Int) extends Bundle {
  val colorStart = UInt(timingsWidth bits)
  val colorEnd   = UInt(timingsWidth bits)
  val syncStart  = UInt(timingsWidth bits)
  val syncEnd    = UInt(timingsWidth bits)
}

case class VgaTimings(timingsWidth: Int) extends Bundle {
  val h = VgaTimingsHV(timingsWidth)
  val v = VgaTimingsHV(timingsWidth)
}
```

然后，我们可以添加一些函数来为特定分辨率和帧率设置这些时序参数：

```
case class VgaTimingsHV(timingsWidth: Int) extends Bundle {
  val colorStart = UInt(timingsWidth bits)
  val colorEnd   = UInt(timingsWidth bits)
  val syncStart  = UInt(timingsWidth bits)
  val syncEnd    = UInt(timingsWidth bits)
}

case class VgaTimings(timingsWidth: Int) extends Bundle {
  val h = VgaTimingsHV(timingsWidth)
  val v = VgaTimingsHV(timingsWidth)

  def setAs_h640_v480_r60(): Unit = {
    h.syncStart := 96 - 1
    h.syncEnd := 800 - 1
    h.colorStart := 96 + 16 - 1
    h.colorEnd := 800 - 48 - 1
    v.syncStart := 2 - 1
    v.syncEnd := 525 - 1
    v.colorStart := 2 + 10 - 1
    v.colorEnd := 525 - 33 - 1
  }

  def setAs_h64_v64_r60(): Unit = {
    h.syncStart := 96 - 1
    h.syncEnd := 800 - 1
    h.colorStart := 96 + 16 - 1 + 288
    h.colorEnd := 800 - 48 - 1 - 288
    v.syncStart := 2 - 1
    v.syncEnd := 525 - 1
```

(续下页)

(接上页)

```
    v.colorStart := 2 + 10 - 1 + 208
    v.colorEnd := 525 - 33 - 1 - 208
  }
}
```

VGA 控制器

规范

io 名称	方向	描述
softReset	in	复位内部计数器并保持 VGA 接口不激活
timings	in	指定 VGA 水平和垂直时序
pixels	slave	为 VGA 控制器提供 RGB 颜色流输入
error	out	当像素流太慢时为高
frameStart	out	新帧开始时为高电平
vga	master	VGA 接口

该控制器不集成任何像素缓冲。它直接从 `pixels Stream` 中获取像素，并在正确的时间将它们放在 `vga.color` 上。如果 `pixels` 无效，则 `error` 在一个周期内变高。

组件及 io 定义

让我们定义一个新的 VgaCtrl `Component` ，它将 `RgbConfig` 和 `timingsWidth` 作为参数。我们将位宽设置为默认值 12。

```
case class VgaCtrl(rgbConfig: RgbConfig, timingsWidth: Int = 12) extends Component
↪{
  val io = new Bundle {
    val softReset = in Bool()
    val timings = in(VgaTimings(timingsWidth))
    val pixels = slave Stream Rgb(rgbConfig)

    val error = out Bool()
    val frameStart = out Bool()
    val vga = master(Vga(rgbConfig))
  }
...
```

水平和垂直逻辑

The logic that generates horizontal and vertical synchronization signals is quite the same. It kind of resembles ~PWM~. The horizontal one counts up each cycle, while the vertical one use the horizontal synchronization signal as to increment.

Let's define `HVArea`, which represents one ~PWM~ and then instantiate it two times: one for both horizontal and vertical synchronization.

```
case class VgaCtrl(rgbConfig: RgbConfig, timingsWidth: Int = 12) extends Component
↪{
...
  case class HVArea(timingsHV: VgaTimingsHV, enable: Bool) extends Area {
    val counter = Reg(UInt(timingsWidth bits)) init 0

    val syncStart  = counter === timingsHV.syncStart
    val syncEnd    = counter === timingsHV.syncEnd
    val colorStart = counter === timingsHV.colorStart
    val colorEnd   = counter === timingsHV.colorEnd

    when(enable) {
      counter := counter + 1
      when(syncEnd) {
        counter := 0
      }
    }

    val sync    = RegInit(False) setWhen syncStart clearWhen syncEnd
    val colorEn = RegInit(False) setWhen colorStart clearWhen colorEnd

    when(io.softReset) {
      counter := 0
      sync    := False
      colorEn := False
    }
  }
  val h = HVArea(io.timings.h, True)
  val v = HVArea(io.timings.v, h.syncEnd)
...
```

正如你所看到的，它是通过使用 `Area` 来完成的。这是为了避免创建一个新的 `Component` ，否则会变得冗长得多。

互连

现在我们有了水平和垂直同步的时序生成器，我们需要驱动输出。

```
case class VgaCtrl(rgbConfig: RgbConfig, timingsWidth: Int = 12) extends Component
↪{
...
  val colorEn = h.colorEn && v.colorEn
  io.pixels.ready := colorEn
  io.error := colorEn && ! io.pixels.valid

  io.frameStart := v.syncEnd

  io.vga.hSync := h.sync
  io.vga.vSync := v.sync
  io.vga.colorEn := colorEn
  io.vga.color := io.pixels.payload
...
```

额外奖励

上面定义的 VgaCtrl 是通用的（不特定于某个应用）。我们可以假设这样一种情况，系统提供 RGB 的 `Fragment` 流 `Stream`，这意味着系统在图片开始/结束指示之间传输像素。

在这种情况下，我们可以通过在发生 `error` 时激活 `softReset` 输入以实现自动管理 `softReset`，然后等待当前 `pixels` 图片结束以取消激活 `error`。

让我们向 `VgaCtrl` 添加一个函数，可以从父组件调用该函数，以通过使用 RGB 的 `Fragment` 流 `Stream` 来提供数据给 `VgaCtrl`。

```
case class VgaCtrl(rgbConfig: RgbConfig, timingsWidth: Int = 12) extends Component
↪{
...
  def feedWith(that : Stream[Fragment[Rgb]]): Unit = {
    io.pixels << that.toStreamOfFragment

    val error = RegInit(False)
    when(io.error) {
      error := True
    }
    when(that.isLast) {
      error := False
    }

    io.softReset := error
    when(error) {
      that.ready := True
    }
  }
}
```

13.3 高级示例

13.3.1 JTAG TAP

简介

重要： 本页的目的是展示一个 JTAG TAP（从设备）的非常规方式实现方法。

重要：

这个实现并不简单，它混合了面向对象编程、抽象接口解耦、硬件生成和硬件描述。
当然，简单的 JTAG TAP 实现只需通过简单的硬件描述就可以完成，但这里的目标实际上是更进一步，创建一个可重用和可扩展的 JTAG TAP 生成器

重要： 本页不会解释 JTAG 的工作原理。可以在 这里 找到一个很好的教程。

常用的 HDL 与 Spinal 之间的一个重要区别在于，SpinalHDL 允许您定义硬件生成器/构建器。这与描述硬件的方式非常不同。让我们看看下面的例子，因为在生成/构建/描述之间的区别可能看起来像是在“玩文字游戏”，或者可以用不同的方式解释。

下面的示例是一个 JTAG TAP，它允许 JTAG 主设备读取 `switchs/keys` 的输入并写入 `leds` 的输出。主设备也可以通过使用 UID 0x87654321 来识别此 TAP。

```
class SimpleJtagTap extends Component {
  val io = new Bundle {
    val jtag    = slave(Jtag())
    val switchs = in  Bits(8 bits)
    val keys    = in  Bits(4 bits)
    val leds    = out Bits(8 bits)
  }

  val tap = new JtagTap(io.jtag, 8)
  val idcodeArea  = tap.idcode(B"x87654321") (instructionId=4)
  val switchsArea = tap.read(io.switchs)     (instructionId=5)
  val keysArea    = tap.read(io.keys)        (instructionId=6)
  val ledsArea    = tap.write(io.leds)       (instructionId=7)
}
```

正如您所看到的，上例创建了一个 JtagTap，不过又调用了一些生成 (Generator)/构建 (Builder) 函数（idcode、read、write）来创建每个 JTAG 指令。这就是我所说的“硬件生成器/构建器”，然后用户使用这些生成器/构建器来描述硬件。还有一点是，在通常的 HDL 中，你只能描述你的硬件，这意味着很多繁琐的工作。

本 JTAG TAP 教程基于 该库 实现。

JTAG 总线

首先我们需要定义一个 JTAG 总线束。

```
case class Jtag() extends Bundle with IMasterSlave {
  val tms = Bool()
  val tdi = Bool()
  val tdo = Bool()

  override def asMaster() : Unit = {
    out(tdi, tms)
    in(tdo)
  }
}
```

正如您所看到的，该总线不包含 TCK 引脚，因为它将由时钟域提供。

JTAG 状态机

让我们定义 JTAG 状态机，依据 此处 所述。

```
object JtagState extends SpinalEnum {
    val RESET, IDLE,
        IR_SELECT, IR_CAPTURE, IR_SHIFT, IR_EXIT1, IR_PAUSE, IR_EXIT2, IR_UPDATE,
        DR_SELECT, DR_CAPTURE, DR_SHIFT, DR_EXIT1, DR_PAUSE, DR_EXIT2, DR_UPDATE =␣
↪newElement()
}

class JtagFsm(jtag: Jtag) extends Area {
    import JtagState._
    val stateNext = JtagState()
    val state = RegNext(stateNext) randBoot()

    stateNext := state.mux(
    default    -> (jtag.tms ? RESET      | IDLE),           // RESET
    IDLE       -> (jtag.tms ? DR_SELECT  | IDLE),
    IR_SELECT  -> (jtag.tms ? RESET      | IR_CAPTURE),
```

(续下页)

(接上页)

```
    IR_CAPTURE -> (jtag.tms ? IR_EXIT1   | IR_SHIFT),
    IR_SHIFT   -> (jtag.tms ? IR_EXIT1   | IR_SHIFT),
    IR_EXIT1   -> (jtag.tms ? IR_UPDATE  | IR_PAUSE),
    IR_PAUSE   -> (jtag.tms ? IR_EXIT2   | IR_PAUSE),
    IR_EXIT2   -> (jtag.tms ? IR_UPDATE  | IR_SHIFT),
    IR_UPDATE  -> (jtag.tms ? DR_SELECT  | IDLE),
    DR_SELECT  -> (jtag.tms ? IR_SELECT  | DR_CAPTURE),
    DR_CAPTURE -> (jtag.tms ? DR_EXIT1   | DR_SHIFT),
    DR_SHIFT   -> (jtag.tms ? DR_EXIT1   | DR_SHIFT),
    DR_EXIT1   -> (jtag.tms ? DR_UPDATE  | DR_PAUSE),
    DR_PAUSE   -> (jtag.tms ? DR_EXIT2   | DR_PAUSE),
    DR_EXIT2   -> (jtag.tms ? DR_UPDATE  | DR_SHIFT),
    DR_UPDATE  -> (jtag.tms ? DR_SELECT  | IDLE)
    )
}
```

备注： `state` 中的 `randBoot()` 会使其以随机状态初始化。这仅用于仿真目的。

JTAG TAP

让我们不使用任何指令，只用最基本的指令寄存器 (IR) 控制和旁路控制实现 JTAG TAP 的核心部分。

```
class JtagTap(val jtag: Jtag, instructionWidth: Int) extends Area with␣
↪JtagTapAccess {
  val fsm = new JtagFsm(jtag)
  val instruction = Reg(Bits(instructionWidth bits))
  val instructionShift = Reg(Bits(instructionWidth bits))
  val bypass = Reg(Bool())

  jtag.tdo := bypass

  switch(fsm.state) {
    is(JtagState.IR_CAPTURE) {
      instructionShift := instruction
    }
    is(JtagState.IR_SHIFT) {
      instructionShift := (jtag.tdi ## instructionShift) >> 1
      jtag.tdo := instructionShift.lsb
    }
    is(JtagState.IR_UPDATE) {
      instruction := instructionShift
    }
    is(JtagState.DR_SHIFT) {
      bypass := jtag.tdi
    }
  }
}
```

备注： 暂时忽略 *with JTagTapAccess* 的引用，下面将会进一步解释。

Jtag 指令

现在 JTAG TAP 核心部分已经完成了，我们可以考虑如何通过可重用的方式来实现 JTAG 指令。

JTAG TAP 类接口

首先，我们需要定义指令如何与 JTAG TAP 内核交互。我们当然可以直接使用 JtagTap 区域（area）。但这不是很好，因为在某些情况下 JTAG TAP 内核是由另一个 IP（例如 Altera 的虚拟 JTAG）提供的。

因此，让我们在 JTAG TAP 内核和指令之间定义一个简单抽象接口：

```
trait JtagTapAccess {
  def getTdi: Bool
  def getTms: Bool
  def setTdo(value: Bool): Unit

  def getState: JtagState.C
  def getInstruction(): Bits
  def setInstruction(value: Bits) : Unit
}
```

然后让 JtagTap 实现这个抽象接口：

列表 1: 添加到 class JtagTap

```
class JtagTap(val jtag: Jtag, ...) extends Area with JtagTapAccess {
...
  override def getTdi: Bool = jtag.tdi
  override def setTdo(value: Bool): Unit = jtag.tdo := value
  override def getTms: Bool = jtag.tms

  override def getState: JtagState.C = fsm.state
  override def getInstruction(): Bits = instruction
  override def setInstruction(value: Bits): Unit = instruction := value
}
```

基类

让我们为 JTAG 指令定义一个有用的基类，它根据所选指令和 JTAG TAP 的状态提供一些回调（doCapture/doShift/doUpdate/doReset）：

```
class JtagInstruction(tap: JtagTapAccess,val instructionId: Bits) extends Area {
  def doCapture(): Unit = {}
  def doShift(): Unit = {}
  def doUpdate(): Unit = {}
  def doReset(): Unit = {}

  val instructionHit = tap.getInstruction === instructionId

  Component.current.addPrePopTask(() => {
    when(instructionHit) {
      when(tap.getState === JtagState.DR_CAPTURE) {
        doCapture()
      }
      when(tap.getState === JtagState.DR_SHIFT) {
        doShift()
      }
      when(tap.getState === JtagState.DR_UPDATE) {
        doUpdate()
```

(续下页)

(接上页)

```
      }
    }
    when(tap.getState === JtagState.RESET) {
      doReset()
    }
  })
}
```

备注:

关于 Component.current.addPrePopTask(…)：
这允许您在当前组件构造结束时调用给定的代码。由于 JtagInstruction 面向对象的性质，doCapture、doShift、doUpdate 和 doReset 不应在子类构造之前调用（因为子类将使用它作为回调来执行某些逻辑）。

读指令

让我们实现一条允许 JTAG 读取信号的指令。

```
class JtagInstructionRead[T <: Data](data: T)(tap: JtagTapAccess, instructionId:␣
↪Bits) extends JtagInstruction(tap, instructionId) {
  val shifter = Reg(Bits(data.getBitsWidth bits))

  override def doCapture(): Unit = {
    shifter := data.asBits
  }

  override def doShift(): Unit = {
    shifter := (tap.getTdi ## shifter) >> 1
    tap.setTdo(shifter.lsb)
  }
}
```

写指令

让我们实现一条允许 JTAG 写入寄存器（并读取其当前值）的指令。

```
class JtagInstructionWrite[T <: Data](data: T, cleanUpdate: Boolean, readable:␣
↪Boolean)(tap: JtagTapAccess, instructionId: Bits) extends JtagInstruction(tap,␣
↪instructionId) {
  val shifter, store = Reg(Bits(data.getBitsWidth bit))

  override def doCapture(): Unit = {
    shifter := store
  }
  override def doShift(): Unit = {
    shifter := (tap.getTdi ## shifter) >> 1
    tap.setTdo(shifter.lsb)
  }
  override def doUpdate(): Unit = {
    store := shifter
  }

  data.assignFromBits(store)
}
```

Idcode 指令

让我们实现向 JTAG 返回 idcode 的指令，并且当发生复位时，将指令寄存器 (IR) 设置为它自己的 instructionId。

```
class JtagInstructionIdcode[T <: Data](value: Bits)(tap: JtagTapAccess,␣
↪instructionId: Bits) extends JtagInstruction(tap, instructionId) {
  val shifter = Reg(Bits(32 bit))

  override def doShift(): Unit = {
    shifter := (tap.getTdi ## shifter) >> 1
    tap.setTdo(shifter.lsb)
  }

  override def doReset(): Unit = {
    shifter := value
    tap.setInstruction(instructionId)
  }
}
```

用户友好型包装

让我们向 JtagTapAccess 添加一些对用户友好的功能，使指令实例化更容易。

列表 2: 添加 `trait JtagTapAccess`

```
trait JtagTapAccess {
...
  def idcode(value: Bits)(instructionId: Bits) =
    new JtagInstructionIdcode(value)(this, instructionId)

  def read[T <: Data](data: T)(instructionId: Bits)    =
    new JtagInstructionRead(data)(this, instructionId)

  def write[T <: Data](data: T, cleanUpdate: Boolean = true, readable: Boolean =␣
↪true)(instructionId: Bits) =
    new JtagInstructionWrite[T](data, cleanUpdate, readable)(this, instructionId)
}
```

使用演示

现在，我们可以非常容易地创建应用特定的 JTAG TAP，而无需编写任何逻辑或任何互连。

```
class SimpleJtagTap extends Component {
  val io = new Bundle {
    val jtag    = slave(Jtag())
    val switchs = in  Bits(8 bits)
    val keys    = in  Bits(4 bits)
    val leds    = out Bits(8 bits)
  }

  val tap = new JtagTap(io.jtag, 8)
  val idcodeArea  = tap.idcode(B"x87654321") (instructionId=4)
  val switchsArea = tap.read(io.switchs)     (instructionId=5)
  val keysArea    = tap.read(io.keys)        (instructionId=6)
  val ledsArea    = tap.write(io.leds)       (instructionId=7)
}
// end SimpleJtagTap
```

这种处理方式（生成硬件）也可以应用于生成 APB/AHB/AXI 的从端总线等。

13.3.2 内存映射 UART

简介

此示例将采用先前示例 中实现的 UartCtrl 组件来创建内存映射 UART 控制器。

规范

该实现将基于带有 RX FIFO 的 APB3 总线。

这是寄存器映射表：

名称	类型	访问	地址	描述
clockDivider	UInt	RW	0	设置 UartCtrl 时钟分频器
frame	UartCtrlFrameConfig	RW	4	设置数据长度、奇偶校验和停止位配置
writeCmd	位	W	8	向 UartCtrl 发送写命令
writeBusy	Bool	R	8	当可以发送新的 writeCmd 时，位 0 => 0
read	Bool / Bits	R	12	位 7 到 0 => rx payload 位 31 => rx payload valid

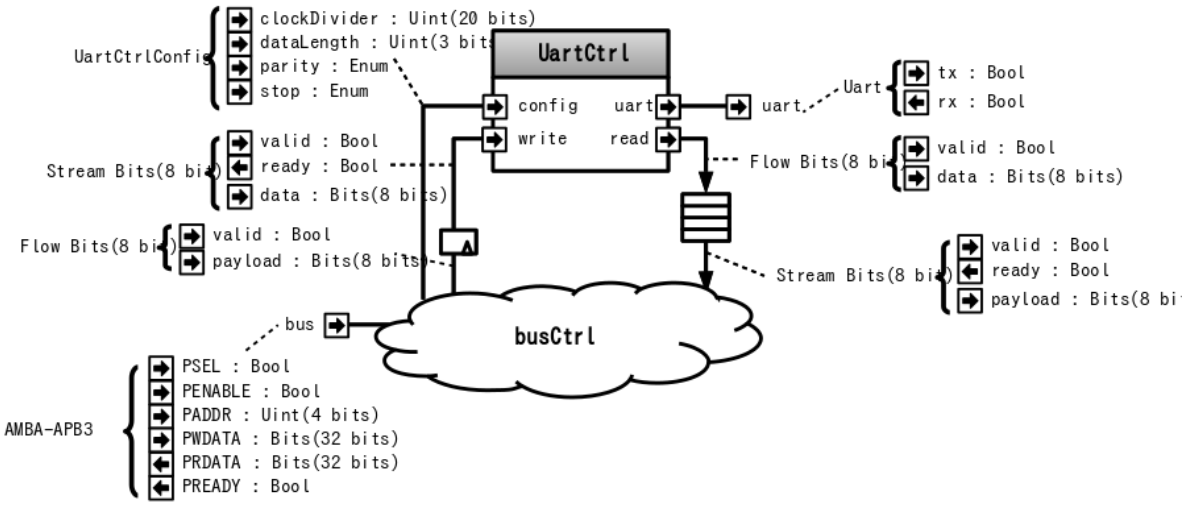
实现

此实现将使用 Apb3SlaveFactory 工具。它允许您使用良好的语法定义 APB3 从端。您可以在这里 找到该工具的文档。

首先，我们只需要定义控制器要使用的 Apb3Config。它在 Scala object 中被定义为一个函数，以便能够从任何地方获取它。

```
object Apb3UartCtrl {
  def getApb3Config = Apb3Config(
    addressWidth = 4,
    dataWidth    = 32
  )
}
```

然后我们可以定义一个 Apb3UartCtrl 组件，该组件实例化了一个 UartCtrl 并在它和 APB3 总线之间创建内存映射逻辑：




```
case class Apb3UartCtrl(uartCtrlConfig: UartCtrlGenerics, rxFifoDepth: Int)␣
↪extends Component {
  val io = new Bundle {
    val bus =  slave(Apb3(Apb3UartCtrl.getApb3Config))
    val uart = master(Uart())
  }

  // Instantiate an simple uart controller
  val uartCtrl = new UartCtrl(uartCtrlConfig)
  io.uart <> uartCtrl.io.uart

  // Create an instance of the Apb3SlaveFactory that will then be used as a slave␣
↪factory drived by io.bus
  val busCtrl = Apb3SlaveFactory(io.bus)

  // Ask the busCtrl to create a readable/writable register at the address 0
  // and drive uartCtrl.io.config.clockDivider with this register
  busCtrl.driveAndRead(uartCtrl.io.config.clockDivider,address = 0)

  // Do the same thing than above but for uartCtrl.io.config.frame at the address 4
  busCtrl.driveAndRead(uartCtrl.io.config.frame,address = 4)

  // Ask the busCtrl to create a writable Flow[Bits] (valid/payload) at the␣
↪address 8.
  // Then convert it into a stream and connect it to the uartCtrl.io.write by␣
↪using an register stage (>->)
  busCtrl.createAndDriveFlow(Bits(uartCtrlConfig.dataWidthMax bits),address = 8).
↪toStream >-> uartCtrl.io.write

  // To avoid losing writes commands between the Flow to Stream transformation␣
↪just above,
  // make the occupancy of the uartCtrl.io.write readable at address 8
  busCtrl.read(uartCtrl.io.write.valid,address = 8)

  // Take uartCtrl.io.read, convert it into a Stream, then connect it to the input␣
↪of a FIFO of 64 elements
  // Then make the output of the FIFO readable at the address 12 by using a non␣
↪blocking protocol
  // (Bit 7 downto 0 => read data <br> Bit 31 => read data valid )
  busCtrl.readStreamNonBlocking(uartCtrl.io.read.queue(rxFifoDepth),
                                address = 12, validBitOffset = 31,␣
↪payloadBitOffset = 0)
}
```

重要:

是的，仅此而已。它同样是可综合的。
Apb3SlaveFactory 工具不是硬编码到 SpinalHDL 编译器中的东西。它是使用 SpinalHDL 常规的硬件描述语法实现的。

13.3.3 Pinesec

记得添加内容！XD

13.3.4 插槽 (Slots)

简介

假设您有一些硬件必须跟踪多个相似的正在进行的活动，您可能需要实现一组“插槽”来执行此操作。此示例展示如何使用 Area、OHMasking.first、onMask 和 reader 来完成此操作。

实现

此实现避免使用 Vec。相反，它使用允许在每个插槽中混合信号、寄存器和逻辑定义的逻辑区。

Note that the *reader* API is for SpinalHDL version coming after 1.9.1

```
package spinaldoc.examples.advanced

import spinal.core._
import spinal.lib._
import scala.language.postfixOps

case class SlotsDemo(slotsCount : Int) extends Component {
  // ...


  // Create the hardware for each slot
  // Note each slot is an Area, not a Bundle
  val slots = for(i <- 0 until slotsCount) yield new Area {
    // Because the slot is an Area, we can define mix signal, registers, logic␣
↪definitions
    // Here are the registers for each slots
    val valid = RegInit(False)
    val address = Reg(UInt(8 bits))
    val age = Reg(UInt(16 bits)) // Will count since how many cycles the slot is␣
↪valid

    // Here is some hardware behavior for each slots
    // Implement the age logic
    when(valid) {
      age := age + 1
    }

    // removeIt will be used as a slot interface later on
    val removeIt = False
    when(removeIt) {
      valid := False
    }
  }
```

(续下页)

(接上页)

```
  // Logic to allocate a new slot
  val insert = new Area {
    val cmd = Stream(UInt(8 bits)) // interface to issue requests
    val free = slots.map(!_.valid)
    val freeOh = OHMasking.first(free) // Get the first free slot (on hot mask)
    cmd.ready := free.orR // Only allow cmd when there is a free slot
    when(cmd.fire) {
      // slots.onMask(freeOh)(code) will execute the code for each slot where the␣
↪corresponding freeOh bit is set
      slots.onMask(freeOh){slot =>
        slot.valid := True
        slot.address := cmd.payload
        slot.age := 0
      }
    }
  }

  // Logic to remove the slots which match a given address (assuming there is not␣
↪more than one match)
  val remove = new Area {
    val cmd = Flow(UInt(8 bits)) // interface to issue requests
    val oh = slots.map(s => s.valid && s.address === cmd.payload) // oh meaning
↪"one hot"
    when(cmd.fire) {
      slots.onMask(oh){ slot =>
        slot.removeIt := True
      }
    }

    val reader = slots.reader(oh) // Create a facility to read the slots using "oh
↪" as index
    val age = reader(_.age) // Age of the slot which is selected by "oh"
  }

  // ...
}

object SlotsDemo extends App {
  SpinalVerilog(SlotsDemo(4))
}
```

应用

例如，在 Tilelink 总线（具有一致性机制）hub 中，这种插槽模式用于跟踪所有正在进行的内存操作：

https://github.com/SpinalHDL/SpinalHDL/blob/008c73f1ce18e294f137efe7a1442bd3f8fa2ee0/lib/src/main/scala/spinal/lib/bus/tilelink/coherent/Hub.scala#L376

As well in the DRAM / SDR / DDR memory controller to implement the handling of multiple memory transactions at once (having multiple precharge / active / read / write running at the same time to improve performances) :

https://github.com/SpinalHDL/SpinalHDL/blob/1edba1890b5f629b28e5171b3c449155337d2548/lib/src/main/scala/spinal/lib/memory/sdram/xdr/Tasker.scala#L202

以及在 NaxRiscv（乱序 CPU）的加载存储单元中处理存储队列/加载队列的硬件（难度有些可怕，不宜在文档中展示 XD）

13.3.5 计时器

简介

计时器模块可能是最基本的硬件模块之一。但即使对于计时器，您也可以使用 SpinalHDL 做一些有趣的事情。这个示例将定义一个简单的计时器组件，其中集成了一个总线桥接实用工具。

计时器

那么让我们从 `Timer` 组件开始。

规范

`Timer` 组件将具有一个构造参数：

参数名称	类型	描述
width	Int	指定计时器计数器的位宽

还有一些输入/输出：

IO 名称	方向	类型	描述
tick	in	Bool	当 `tick` 为 True 时，计时器计数到 `limit` 。
clear	in	Bool	当 `tick` 为 True 时，计时器设为零。`clear` 优先于 `tick` 。
limit	in	UInt(width bits)	当计时器值等于 `limit` 时，禁止 `tick` 输入。
full	out	Bool	当计时器值等于 `limit` 并且 `tick` 为高时，`full` 为高。
value	out	UInt(width bits)	引出计时器的计数器值。

实现

```
case class Timer(width : Int) extends Component {
  val io = new Bundle {
    val tick  = in Bool()
    val clear = in Bool()
    val limit = in UInt(width bits)

    val full  = out Bool()
    val value = out UInt(width bits)
  }

  val counter = Reg(UInt(width bits))
  when(io.tick && !io.full) {
    counter := counter + 1
  }
  when(io.clear) {
    counter := 0
  }

  io.full := counter === io.limit && io.tick
  io.value := counter

}
```

桥接函数

现在我们可以从这个例子的主要目的开始：定义总线桥接功能。为此，我们将使用两种技术：

- 使用在文档此处 的 `BusSlaveFactory` 工具
- 在 `Timer` 组件内定义一个函数，这个函数可以从父组件调用该函数，并以抽象方式驱动 `Timer` 的 IO。

规范

该桥接函数将使用以下参数：

参数名称	类型	描述
busCtrl	BusSlaveFactory	函数将使用 `BusSlaveFactory` 实例来创建桥接逻辑。
baseAddress	BigInt	桥接逻辑应映射到的基地址。
ticks	Seq[Bool]	可用作 tick 信号的 Bool 源序列。
clears	Seq[Bool]	可用作 clear 信号的 Bool 源序列。

假设寄存器映射的总线系统位宽是 32 位：

名称	访问	位宽	地址偏移	位偏移	描述
ticksEnable	RW	len(ticks)	0	0	Each `ticks` bool can be activated if the corresponding `ticksEnable` bit is high.
clearsEnable	RW	len(clears)	0	16	Each `clears` bool can be activated if the corresponding `clearsEnable` bit is high.
limit	RW	width	4	0	访问计时器组件的 limit 值。 当写入该寄存器时，计时器清零。
value	R	width	8	0	访问计时器的值。
clear	W		8		当写入该寄存器时，计时器清零。

实现

让我们在 `Timer` 组件中添加这个桥接函数。

```
case class Timer(width : Int) extends Component {
...
  // The function prototype uses Scala currying funcName(arg1,arg2)(arg3,arg3)
  // which allow to call the function with a nice syntax later
  // This function also returns an area, which allows to keep names of inner␣
↪signals in the generated VHDL/Verilog.
  def driveFrom(busCtrl: BusSlaveFactory, baseAddress: BigInt)(ticks: Seq[Bool],␣
↪clears: Seq[Bool]) = new Area {
    // Offset 0 => clear/tick masks + bus
```

(续下页)

(接上页)

```
    val ticksEnable = busCtrl.createReadAndWrite(Bits(ticks.length bits),␣
↪baseAddress + 0,0) init(0)
    val clearsEnable = busCtrl.createReadAndWrite(Bits(clears.length bits),␣
↪baseAddress + 0,16) init(0)
    val busClearing = False

    io.clear := (clearsEnable & clears.asBits).orR | busClearing
    io.tick := (ticksEnable  & ticks.asBits ).orR

    // Offset 4 => read/write limit (+ auto clear)
    busCtrl.driveAndRead(io.limit, baseAddress + 4)
    busClearing.setWhen(busCtrl.isWriting(baseAddress + 4))

    // Offset 8 => read timer value / write => clear timer value
    busCtrl.read(io.value, baseAddress + 8)
    busClearing.setWhen(busCtrl.isWriting(baseAddress + 8))
  }
}
```

用法

下面是一些演示代码，它与 Pinsec SoC 计时器模块中使用的代码非常接近。基本上，它实例化了以下元素：

- 1 个 16 位预分频器
- 1 个 32 位计时器
- 3 个 16 位计时器

然后，通过使用 `Apb3SlaveFactory` 和 `Timer` 内定义的函数，它在 APB3 总线和所有实例化组件之间创建桥接逻辑。

```
  val io = new Bundle {
    val apb = slave(Apb3(Apb3Config(addressWidth=8, dataWidth=32)))
    val interrupt = out Bool()
    val external = new Bundle {
      val tick = in Bool()
      val clear = in Bool()
    }
  }

  // Prescaler is very similar to the timer, it mainly integrates a piece of auto␣
↪reload logic.
  val prescaler = Prescaler(width = 16)

  val timerA = Timer(width = 32)
  val timerB,timerC,timerD = Timer(width = 16)

  val busCtrl = Apb3SlaveFactory(io.apb)

  prescaler.driveFrom(busCtrl, 0x00)

  timerA.driveFrom(busCtrl, 0x40)(
    ticks=List(True, prescaler.io.overflow),
    clears=List(timerA.io.full)
  )
  timerB.driveFrom(busCtrl, 0x50)(
    ticks=List(True, prescaler.io.overflow, io.external.tick),
    clears=List(timerB.io.full, io.external.clear)
```

(续下页)

(接上页)

```
  )
  timerC.driveFrom(busCtrl, 0x60)(
    ticks=List(True, prescaler.io.overflow, io.external.tick),
    clears=List(timerC.io.full, io.external.clear)
  )
  timerD.driveFrom(busCtrl, 0x70)(
    ticks=List(True, prescaler.io.overflow, io.external.tick),
    clears=List(timerD.io.full, io.external.clear)
  )

  val interruptCtrl = InterruptCtrl(4)
  interruptCtrl.driveFrom(busCtrl, 0x10)
  interruptCtrl.io.inputs(0) := timerA.io.full
  interruptCtrl.io.inputs(1) := timerB.io.full
  interruptCtrl.io.inputs(2) := timerC.io.full
  interruptCtrl.io.inputs(3) := timerD.io.full
  io.interrupt := interruptCtrl.io.pendings.orR
}
```

示例分为三类：

- 可用于熟悉 SpinalHDL 基础知识的简单示例。
- 使用传统方法设计组件的中级示例。
- 通过使用面向对象编程、函数式编程和元硬件描述，比传统 HDL 更进一步的高级示例。

它们都可以在相应部分下的侧边栏中访问。

重要： SpinalHDL 实践教程包含许多实验及其解决方案。请参阅 此处。

备注： 您还可以在这里 找到使用 SpinalHDL 的仓库列表

13.4 入门

所有示例均假设您的 scala 文件顶部有以下导入：

```
import spinal.core._
import spinal.lib._
```

要为给定组件生成 VHDL，您可以将以下内容放在 scala 文件的底部：

```
object MyMainObject {
  def main(args: Array[String]) {
    SpinalVhdl(new TheComponentThatIWantToGenerate(constructionArguments))    // Or␣
↪SpinalVerilog
  }
}
```

CHAPTER 14

历史遗留

14.1 RiscV

警告： 本页仅记录在 SpinalHDL 中创造的第一代 RISC-V CPU。本页面未记录 VexRiscV CPU，它是该 CPU 的第二代，可在 此处 <*https://github.com/SpinalHDL/VexRiscv*> 获取，并提供更好的性能/面积/特性。

14.1.1 特性

RISC-V CPU

- 5 级流水线（获取解码执行 0 执行 1 回写）
- 多种分支预测模式：（禁用、静态或动态）
- 数据路径可在完全旁路和完全互锁之间进行参数化

扩展

- 一周期的乘法
- 34 个周期的除法
- 迭代移位器（N 次移位 -> N 个周期）
- 单周期移位器
- 中断控制器
- 调试模块（带有 JTAG 桥、openOCD 端口和 GDB）
- 具有封装的突发内存接口的指令缓存，单向
- 具有清除/刷新整个缓存或特定地址的指令的数据缓存，单向

性能/面积（在 cyclone II 上）

- 小核 -> 846 LE, 0.6 DMIPS/Mhz
- 调试模块（无 JTAG）-> 240 LE
- JTAG Avalon 主控 -> 238 LE

- 带 MUL/DIV/全移位器/I$/中断/调试的大核 -> 2200 LE，1.15 DMIPS/Mhz，至少 100Mhz（使用默认综合选项）

14.1.2 基础 FPGA 项目

您可以在这里找到一个 DE1-SOC 项目，它将两个 CPU 实例与 MUL/DIV/全移位器/I$/中断/调试集成在一起：

https://drive.google.com/drive/folders/0B-CqLXDTaMbKNkktb2k3T3lzcUk?usp=sharing

CPU/JTAG/VGA IP 是预先生成的。Quartus Prime：15.1。

14.1.3 如何生成 CPU VHDL

警告： 最近版本的 SpinalHDL 中不存在该 Avalon 版本的 CPU。请考虑使用 VexRiscv 。

14.1.4 如何调试

你可以在这里找到 openOCD 分支：

https://github.com/Dolu1990/openocd_riscv

可以在此处找到示例目标配置文件：

https://github.com/Dolu1990/openocd_riscv/blob/riscv_spinal/tcl/target/riscv_spinal.cfg

然后就可以使用 RISCV GDB 了。

14.1.5 Todo

- 文档
- Optimize instruction/data caches FMax by moving line hit condition forward into combinatorial paths.

联系 spinhdl@gmail.com 了解更多信息

14.2 pinsec

14.2.1 简介

备注： This page only documents the SoC implemented with the first generation of RISC-V CPU created in SpinalHDL. This page does not document the VexRiscV CPU, which is the second generation of this SoC (and CPU) is available here and offers better performance/area/features.

简介

Pinsec 是完全用 SpinalHDL 编写的一个小型 FPGA SoC 的名称。该项目有多个目的：

- 证明 SpinalHDL 是重要项目中可行的 HDL 替代方案。
- 在具体项目中展示 SpinalHDL 元硬件描述功能的优势。
- 提供完全开源的 SoC。

Pinsec 具有以下硬件特性：

- 用于高速总线的 AXI4 互连
- 用于外设的 APB3 互连
- 具有指令缓存、MUL/DIV 扩展和中断控制器的 RISCV CPU
- 用于加载二进制文件和调试 CPU 的 JTAG 桥
- SDRAM SDR 控制器
- 片上内存
- 1 个 UART 控制器
- 1 个 VGA 控制器
- 一些定时器模块
- 一些 GPIO

顶层代码的解释可以在这里 找到

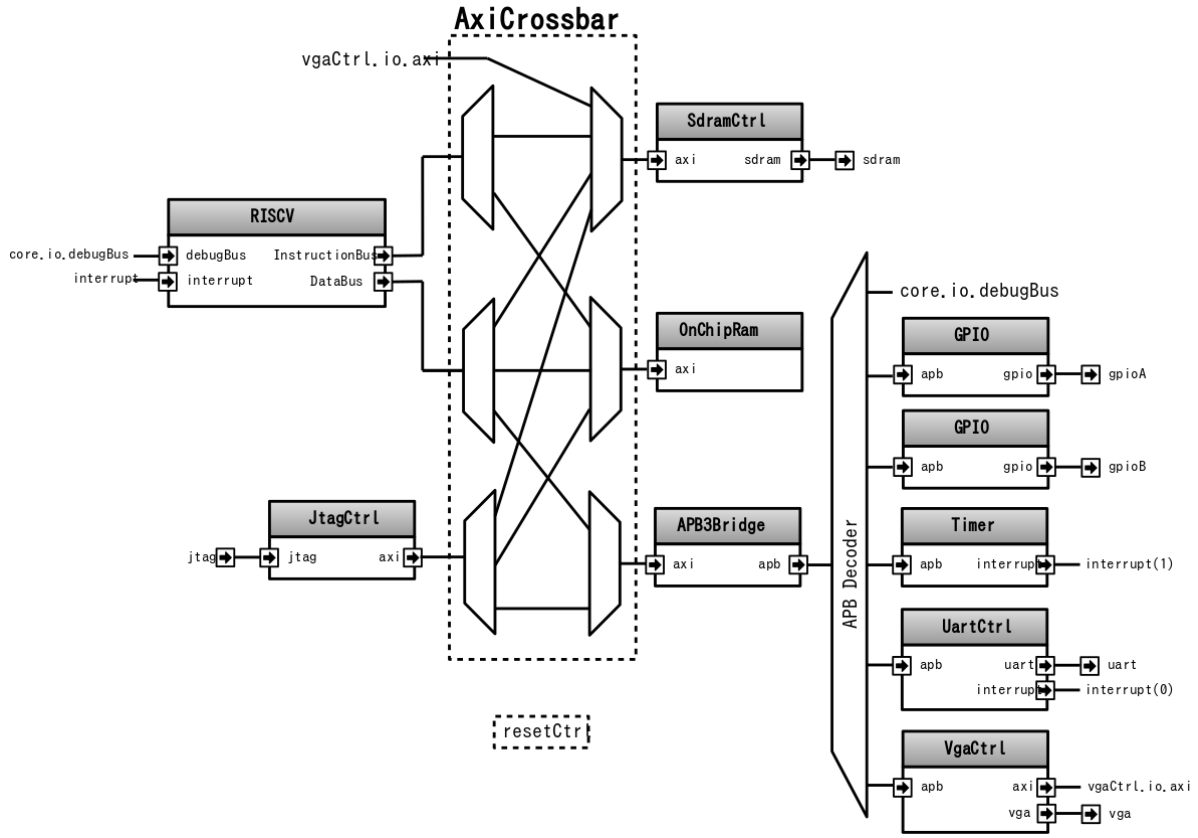
板级支持

DE1-SOC FPGA 项目可以在 这里 找到，其中包含一些二进制文件 demo。

14.2.2 硬件

简介

这是 Pinsec 顶层硬件图：



RISCV

RISCV 是一款 5 级流水线 CPU，具有以下特性：

- 指令缓存
- 单周期桶式移位器
- 单周期 MUL、34 周期 DIV
- 中断支持
- 动态分支预测
- 调试端口

AXI4

如前所述，Pinsec 集成了 AXI4 总线结构。AXI4 不是最容易使用的总线，但具有许多优点，例如：

- 灵活的拓扑结构
- 高带宽潜力
- 潜在的乱序请求完成
- 满足时钟时序的简单方法
- 被许多 IP 核使用的标准
- 适合 SpinalHDL 反压流 (Stream) 的握手方法。

从面积利用率的角度来看，AXI4 肯定不是最轻量的解决方案，但某些技术可以大大减少这种担忧：

- 在可能的情况下使用只读/只写 AXI4 变体
- 引入 Axi4-Shared 变体，其中引入新的 ARW 通道来代替和组合 AR 和 AW 通道。该解决方案将地址解码和地址仲裁的资源使用量减少了两倍。
- 根据互连实现的不同，如果所有主设备都不会使 R/B 通道停滞（RREADY 和 BREADY 被固定为 1），则可以进行时序松弛。在这种情况下，可以通过综合去除两个 xREADY 信号，从而放宽时序。
- 正如 AXI4 规范所建议的，互连可以通过聚合相应的输入端口 ID 来扩展事务 ID。这允许互连具有无限数量的待处理请求，并且能够以极小的面积成本（扩大事务 ID）支持乱序完成。

Pinsec 互连不会引入延迟周期。

APB3

在 Pinsec 中，所有外设均实现了一个 APB3 总线接口。选择 APB3 的原因如下：

- 非常简单的总线（无突发）
- 使用很少的资源
- 被许多 IP 核使用的标准

生成 RTL

要生成 RTL，您有多种方案：

您可以下载 SpinalHDL 源代码，然后运行：

```
sbt "project SpinalHDL-lib" "run-main spinal.lib.soc.pinsec.Pinsec"
```

或者您可以在自己的 SBT 项目中创建自己的 main，然后运行它：

```
import spinal.lib.soc.pinsec._

object PinsecMain {
  def main(args: Array[String]) {
    SpinalVhdl(new Pinsec(100 MHz))
    SpinalVerilog(new Pinsec(100 MHz))
  }
}
```

备注： 目前，由于最新版本的 GHDL 与 cocotb 不兼容，因此仅在仿真和 FPGA 中测试了 verilog 版本。

14.2.3 SoC 顶层 (Pinsec)

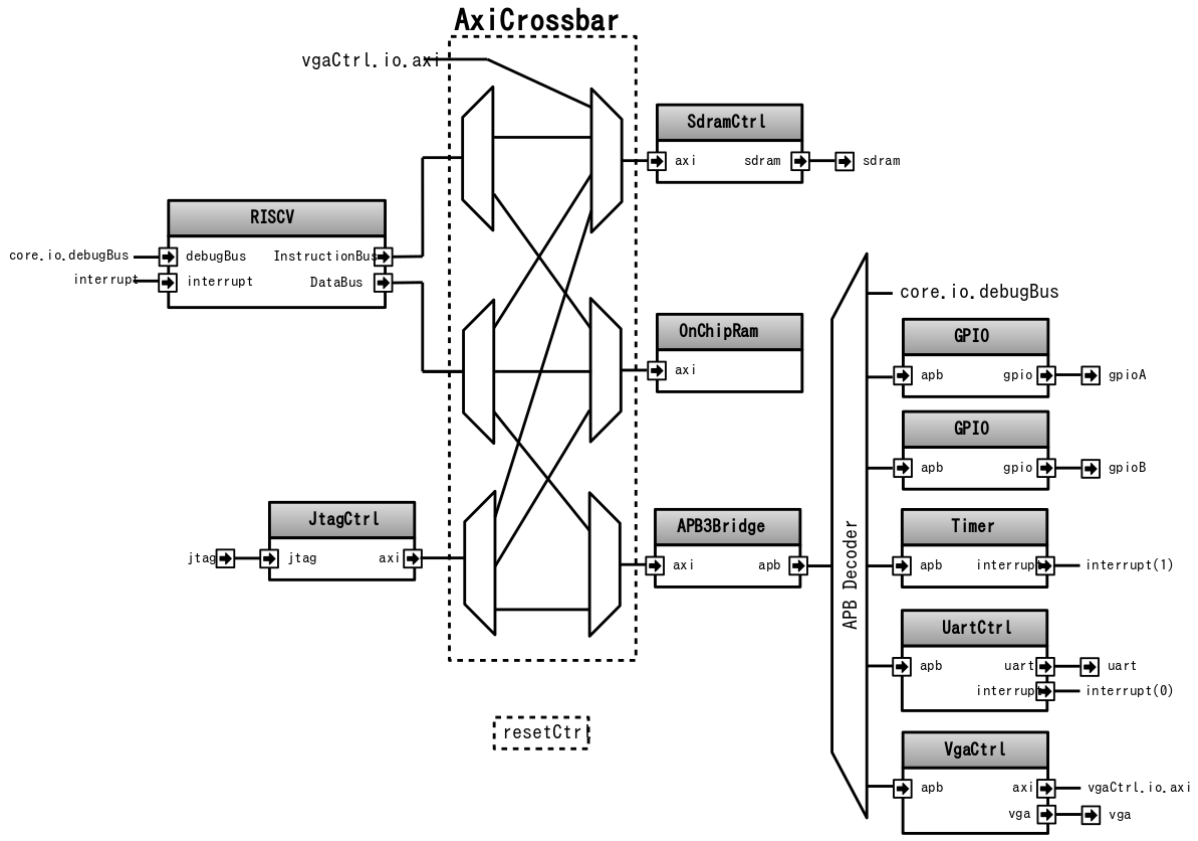
简介

Pinsec 是一个专为 FPGA 设计的小型 SoC。它可以在 SpinalHDL 库中找到，并且可以在这里 找到一些文档

它的顶层实现是一个有趣的例子，因为它混合了一些设计模式，使其非常容易修改。可以轻松实现向总线结构添加新的主设备或新的外设。

可以在以下链接中查阅顶层实现：https://github.com/SpinalHDL/SpinalHDL/blob/master/lib/src/main/scala/spinal/lib/soc/pinsec/Pinsec.scala

这是 Pinsec 顶层硬件图：



定义所有 IO

```
val io = new Bundle {
  // Clocks / reset
  val asyncReset = in Bool()
  val axiClk     = in Bool()
  val vgaClk     = in Bool()

  // Main components IO
  val jtag       = slave(Jtag())
  val sdram      = master(SdramInterface(IS42x320D.layout))

  // Peripherals IO
  val gpioA      = master(TriStateArray(32 bits))    // Each pin has an individual␣
↪output enable control
  val gpioB      = master(TriStateArray(32 bits))
  val uart       = master(Uart())
```

(续下页)

(接上页)

```
  val vga         = master(Vga(RgbConfig(5,6,5)))
}
```

时钟和复位

Pinsec 有三个时钟输入：

- axiClock
- vgaClock
- jtag.tck

以及一个复位输入：

- asyncReset

最终将给出 5 个时钟域 (ClockDomain)（时钟/复位对）：

名称	时钟	描述
resetCtrlClockDomain	axiClock	由复位控制器使用，该时钟域的触发器由 FPGA 比特流初始化
axiClockDomain	axiClock	由连接到 AXI 和 APB 互连的所有组件使用
coreClockDomain	axiClock	与 axiClockDomain 的唯一区别是，复位也可以通过调试模块控制
vgaClockDomain	vgaClock	被 VGA 控制器后端用作像素时钟
jtagClockDomain	jtag.tck	用于为 JTAG 控制器的前端提供时钟

复位控制器

首先我们需要定义复位控制器时钟域，它没有复位线，而是使用 FPGA 比特流加载来设置触发器。

```
val resetCtrlClockDomain = ClockDomain(
  clock = io.axiClk,
  config = ClockDomainConfig(
    resetKind = BOOT
  )
)
```

然后我们可以在这个时钟域下定义一个简单的复位控制器。

```
val resetCtrl = new ClockingArea(resetCtrlClockDomain) {
  val axiResetUnbuffered  = False
  val coreResetUnbuffered = False

  // Implement an counter to keep the reset axiResetOrder high 64 cycles
  // Also this counter will automaticly do a reset when the system boot.
  val axiResetCounter = Reg(UInt(6 bits)) init(0)
  when(axiResetCounter =/= U(axiResetCounter.range -> true)) {
    axiResetCounter := axiResetCounter + 1
    axiResetUnbuffered := True
  }
  when(BufferCC(io.asyncReset)) {
    axiResetCounter := 0
  }

  // When an axiResetOrder happen, the core reset will as well
  when(axiResetUnbuffered) {
    coreResetUnbuffered := True
  }
```

(续下页)

(接上页)

```
  // Create all reset used later in the design
  val axiReset   = RegNext(axiResetUnbuffered)
  val coreReset  = RegNext(coreResetUnbuffered)
  val vgaReset   = BufferCC(axiResetUnbuffered)
}
```

每个系统的时钟域设置

现在复位控制器已经实现，我们可以为 Pinsec 的所有子系统定义时钟域：

```
val axiClockDomain = ClockDomain(
  clock     = io.axiClk,
  reset     = resetCtrl.axiReset,
  frequency = FixedFrequency(50 MHz) // The frequency information is used by the␣
↪SDRAM controller
)

val coreClockDomain = ClockDomain(
  clock = io.axiClk,
  reset = resetCtrl.coreReset
)

val vgaClockDomain = ClockDomain(
  clock = io.vgaClk,
  reset = resetCtrl.vgaReset
)

val jtagClockDomain = ClockDomain(
  clock = io.jtag.tck
)
```

此外，Pinsec 的所有核心系统都将在一个 `axi` 时钟域里定义：

```
val axi = new ClockingArea(axiClockDomain) {
  // Here will come the rest of Pinsec
}
```

主要组件

Pinsec 主要由 4 个主要组件构成：

- 1 个 RISCV CPU
- 1 个 SDRAM 控制器
- 1 个片上存储器
- 1 个 JTAG 控制器

RISCV CPU

Pinsec 中使用的 RISCV CPU 具有多种参数化可能性：

```
val core = coreClockDomain {
  val coreConfig = CoreConfig(
    pcWidth = 32,
    addrWidth = 32,
    startAddress = 0x00000000,
    regFileReadyKind = sync,
    branchPrediction = dynamic,
    bypassExecute0 = true,
    bypassExecute1 = true,
    bypassWriteBack = true,
    bypassWriteBackBuffer = true,
    collapseBubble = false,
    fastFetchCmdPcCalculation = true,
    dynamicBranchPredictorCacheSizeLog2 = 7
  )

  // The CPU has a systems of plugin which allow to add new feature into the core.
  // Those extension are not directly implemented into the core, but are kind of␣
↪additive logic patch defined in a separated area.
  coreConfig.add(new MulExtension)
  coreConfig.add(new DivExtension)
  coreConfig.add(new BarrelShifterFullExtension)

  val iCacheConfig = InstructionCacheConfig(
    cacheSize =4096,
    bytePerLine =32,
    wayCount = 1,  // Can only be one for the moment
    wrappedMemAccess = true,
    addressWidth = 32,
    cpuDataWidth = 32,
    memDataWidth = 32
  )

  // There is the instantiation of the CPU by using all those construction␣
↪parameters
  new RiscvAxi4(
    coreConfig = coreConfig,
    iCacheConfig = iCacheConfig,
    dCacheConfig = null,
    debug = true,
    interruptCount = 2
  )
}
```

片上 RAM

AXI4 片上 RAM 的实例化非常简单。

事实上，它不是 AXI4，而是 Axi4Shared，这意味着 ARW 通道取代了 AR 和 AW 通道。该解决方案占用的面积更少，同时可与完整的 AXI4 实现完全互操作。

```
val ram = Axi4SharedOnChipRam(
  dataWidth = 32,
  byteCount = 4 KiB,
  idWidth = 4      // Specify the AXI4 ID width.
)
```

SDRAM 控制器

首先，您需要定义 SDRAM 设备的布局和时序。在 DE1-SOC 上，SDRAM 型号是 IS42x320D。

```
object IS42x320D {
  def layout = SdramLayout(
    bankWidth    = 2,
    columnWidth = 10,
    rowWidth     = 13,
    dataWidth    = 16
  )

  def timingGrade7 = SdramTimings(
    bootRefreshCount =   8,
    tPOW             = 100 us,
    tREF             =  64 ms,
    tRC              =  60 ns,
    tRFC             =  60 ns,
    tRAS             =  37 ns,
    tRP              =  15 ns,
    tRCD             =  15 ns,
    cMRD             =   2,
    tWR              =  10 ns,
    cWR              =   1
  )
}
```

然后您可以使用这些定义来参数化 SDRAM 控制器实例。

```
val sdramCtrl = Axi4SharedSdramCtrl(
  axiDataWidth = 32,
  axiIdWidth   = 4,
  layout       = IS42x320D.layout,
  timing       = IS42x320D.timingGrade7,
  CAS          = 3
)
```

JTAG 控制器

JTAG 控制器可用于在 PC 访问存储器并调试 CPU。

```
val jtagCtrl = JtagAxi4SharedDebugger(SystemDebuggerConfig(
  memAddressWidth = 32,
  memDataWidth    = 32,
  remoteCmdWidth  = 1,
  jtagClockDomain = jtagClockDomain
))
```

外设

Pinsec 有一些集成的外设：

- GPIO
- 计时器
- 串口
- VGA

GPIO

```
val gpioACtrl = Apb3Gpio(
  gpioWidth = 32
)

val gpioBCtrl = Apb3Gpio(
  gpioWidth = 32
)
```

计时器

Pinsec 定时器模块包括：

- 1 个预分频器
- 1 个 32 位定时器
- 三个 16 位定时器

所有这些都被打包到 PinsecTimerCtrl 组件中。

```
val timerCtrl = PinsecTimerCtrl()
```

UART 控制器

首先我们需要为 UART 控制器定义一个配置：

```
val uartCtrlConfig = UartCtrlMemoryMappedConfig(
  uartCtrlConfig = UartCtrlGenerics(
    dataWidthMax      = 8,
    clockDividerWidth = 20,
    preSamplingSize   = 1,
    samplingSize      = 5,
    postSamplingSize  = 2
  ),
  txFifoDepth = 16,
  rxFifoDepth = 16
)
```

然后我们可以用它来实例化 UART 控制器

```
val uartCtrl = Apb3UartCtrl(uartCtrlConfig)
```

VGA 控制器

首先我们需要定义 VGA 控制器的配置：

```
val vgaCtrlConfig = Axi4VgaCtrlGenerics(
  axiAddressWidth = 32,
  axiDataWidth    = 32,
  burstLength     = 8,           // In Axi words
  frameSizeMax    = 2048*1512*2, // In byte
  fifoSize        = 512,         // In axi words
  rgbConfig       = RgbConfig(5,6,5),
  vgaClock        = vgaClockDomain
)
```

然后我们可以用它来实例化 VGA 控制器

```
val vgaCtrl = Axi4VgaCtrl(vgaCtrlConfig)
```

总线互连

共有三个互连组件：

- AXI4 交叉开关 (crossbar)
- AXI4 桥接到 APB3
- APB3 解码器

AXI4 桥接到 APB3

该桥将用于将低带宽外设连接到 AXI 交叉开关。

```
val apbBridge = Axi4SharedToApb3Bridge(
  addressWidth = 20,
  dataWidth    = 32,
  idWidth      = 4
)
```

AXI4 交叉开关 (crossbar)

The AXI4 crossbar that interconnect AXI4 masters and slaves together is generated by using an factory. The concept of this factory is to create it, then call many function on it to configure it, and finally call the `build` function to ask the factory to generate the corresponding hardware :

```
val axiCrossbar = Axi4CrossbarFactory()
// Where you will have to call function the the axiCrossbar factory to populate␣
↪its configuration
axiCrossbar.build()
```

首先，您需要添加从端接口：

```
//           Slave  -> (base address,  size) ,

axiCrossbar.addSlaves(
  ram.io.axi       -> (0x00000000L,   4 KiB),
  sdramCtrl.io.axi -> (0x40000000L,  64 MiB),
  apbBridge.io.axi -> (0xF0000000L,   1 MiB)
)
```

然后，您需要添加从端和主端之间的互连矩阵（这展现可见性）：

```
//         Master -> List of slaves which are accessible

axiCrossbar.addConnections(
  core.io.i       -> List(ram.io.axi, sdramCtrl.io.axi),
  core.io.d       -> List(ram.io.axi, sdramCtrl.io.axi, apbBridge.io.axi),
  jtagCtrl.io.axi -> List(ram.io.axi, sdramCtrl.io.axi, apbBridge.io.axi),
  vgaCtrl.io.axi  -> List(            sdramCtrl.io.axi)
)
```

然后，为了减少组合路径长度并拥有良好的设计 FMax，您可以要求生成器在给定的主端或从端之间插入流水线级：

备注:

以下代码中的 `halfPipe` / » / « / >/-> 由反压流 (Stream) 总线库提供。
可以在这里 找到一些文档。简而言之，这只是一些流水线和互连的东西。

```
// Pipeline the connection between the crossbar and the apbBridge.io.axi
axiCrossbar.addPipelining(apbBridge.io.axi,(crossbar,bridge) => {
  crossbar.sharedCmd.halfPipe() >> bridge.sharedCmd
  crossbar.writeData.halfPipe() >> bridge.writeData
  crossbar.writeRsp             << bridge.writeRsp
  crossbar.readRsp              << bridge.readRsp
})

// Pipeline the connection between the crossbar and the sdramCtrl.io.axi
axiCrossbar.addPipelining(sdramCtrl.io.axi,(crossbar,ctrl) => {
  crossbar.sharedCmd.halfPipe()  >>  ctrl.sharedCmd
  crossbar.writeData            >/-> ctrl.writeData
  crossbar.writeRsp              <<  ctrl.writeRsp
  crossbar.readRsp               <<  ctrl.readRsp
})
```

APB3 解码器

APB3 桥和所有外设之间的互连是通过 APB3Decoder 完成的：

```
val apbDecoder = Apb3Decoder(
  master = apbBridge.io.apb,
  slaves = List(
    gpioACtrl.io.apb -> (0x00000, 4 KiB),
    gpioBCtrl.io.apb -> (0x01000, 4 KiB),
    uartCtrl.io.apb  -> (0x10000, 4 KiB),
    timerCtrl.io.apb -> (0x20000, 4 KiB),
    vgaCtrl.io.apb   -> (0x30000, 4 KiB),
    core.io.debugBus -> (0xF0000, 4 KiB)
  )
)
```

杂项

要将所有顶层 IO 连接到组件，需要以下代码：

```
io.gpioA <> axi.gpioACtrl.io.gpio
io.gpioB <> axi.gpioBCtrl.io.gpio
io.jtag  <> axi.jtagCtrl.io.jtag
io.uart  <> axi.uartCtrl.io.uart
io.sdram <> axi.sdramCtrl.io.sdram
io.vga   <> axi.vgaCtrl.io.vga
```

最后需要组件之间的一些连接，例如中断和核心调试模块复位

```
core.io.interrupt(0) := uartCtrl.io.interrupt
core.io.interrupt(1) := timerCtrl.io.interrupt

core.io.debugResetIn := resetCtrl.axiReset
when(core.io.debugResetOut) {
  resetCtrl.coreResetUnbuffered := True
}
```

14.2.4 软件

RISCV 工具链

CPU 执行的二进制文件可以用 ASM/C/C++ 定义，并由 GCC 的 RISCV 分支进行编译。此外，为了加载二进制文件并调试 CPU，可以使用 OpenOCD 分支和 RISCV 的 GDB。

RISCV 工具：https://github.com/riscv/riscv-wiki/wiki/RISC-V-Software-Status
OpenOCD 分支：https://github.com/Dolu1990/openocd_riscv
软件示例：https://github.com/Dolu1990/pinsecSoftware

OpenOCD/GDB/Eclipse 配置

关于 OpenOCD 分支，这里有一个可用于连接 Pinsec SoC 的配置文件：https://github.com/Dolu1990/openocd_riscv/blob/riscv_spinal/tcl/target/riscv_spinal.cfg

这是一个用于运行 OpenOCD 工具的参数示例：

```
openocd -f ../tcl/interface/ftdi/ft2232h_breakout.cfg -f ../tcl/target/riscv_
↪spinal.cfg -d 3
```

要使用 eclipse 进行调试，您将需要 Zylin 插件，然后创建一个“Zynlin embedded debug (native)”。

初始化命令：

```
target remote localhost:3333
monitor reset halt
load
```

运行命令：

```
continue
```

CHAPTER 15

杂项

本节包含的内容可能是：

- 过时的
- 可以更好地策划
- 可能包含重复信息（最好在此处或另一个存储库中找到）
- 文件草稿和正在进行的工作

因此，请谨慎考虑本节中的信息，并尽力让作者提供文档。

15.1 常见错误

本页将讨论人们在使用 SpinalHDL 时可能出现的错误。

15.2 “main”线程中异常 java.lang.NullPointerException

控制台输出：

```
Exception in thread "main" java.lang.NullPointerException
```

代码示例：

```
val a = b + 1          // b can't be read at that time, because b isn't␣
↪instantiated yet
val b = UInt(4 bits)
```

问题解释：

SpinalHDL 不是一种语言，它是一个 Scala 库，这意味着它遵循与 Scala 语言相同的通用规则。当您运行 SpinalHDL 硬件描述来生成相应的 VHDL/Verilog RTL 时，您的 SpinalHDL 硬件描述将作为 Scala 程序执行，并且执行程序到该行时，b 将是一个 `null` 空引用，这就是为什么你在这之前不能使用它的原因。

15.3 层次违例 (Hierarchy violation)

SpinalHDL 编译器从层次结构的角度检查所有赋值是否合法。后续章节将详细阐述多个案例

15.3.1 Signal X can't be assigned by Y

控制台输出:

```
Hierarchy violation : Signal X can't be assigned by Y
```

代码示例:

```
class ComponentX extends Component {
  ...
  val X = Bool()
  ...
}

class ComponentY extends Component {
  ...
  val componentX = new ComponentX
  val Y = Bool()
  componentX.X := Y // This assignment is not legal
  ...
}
```

```
class ComponentX extends Component {
  val io = new Bundle {
    val X = Bool() // Forgot to specify an in/out direction
  }
  ...
}

class ComponentY extends Component {
  ...
  val componentX = new ComponentX
  val Y = Bool()
  componentX.io.X := Y // This assignment will be detected as not legal
  ...
}
```

问题解释:

You can only assign input signals of subcomponents, else there is an hierarchy violation. If this issue happened, you probably forgot to specify the X signal's direction.

15.3.2 Input signal X can't be assigned by Y

控制台输出:

```
Hierarchy violation : Input signal X can't be assigned by Y
```

代码示例:

```
class ComponentXY extends Component {
  val io = new Bundle {
    val X = in Bool()
  }
```

(续下页)

(接上页)

```
  ...
  val Y = Bool()
  io.X := Y // This assignment is not legal
  ...
}
```

问题解释：

You can only assign an input signals from the parent component, else there is an hierarchy violation. If this issue happened, you probably mixed signals direction declaration.

15.3.3 Output signal X can’t be assigned by Y

控制台输出：

```
Hierarchy violation : Output signal X can't be assigned by Y
```

代码示例：

```
class ComponentX extends Component {
  val io = new Bundle {
    val X = out Bool()
  }
  ...
}

class ComponentY extends Component {
  ...
  val componentX = new ComponentX
  val Y = Bool()
  componentX.X := Y // This assignment is not legal
  ...
}
```

问题解释：

You can only assign output signals of a component from the inside of it, else there is an hierarchy violation. If this issue happened, you probably mixed signals direction declaration.

15.4 spinal.core 组件

本文档描述了该语言的核心组件。它涵盖了大部分情况

核心语言组件如下：

- *时钟域*，允许在设计中定义和操作多个时钟域
- *存储器实例化*，允许自动实例化 RAM 和 ROM 存储器。
- *IP 实例化*，使用现有的 VHDL 或 Verilog 组件实例化。
- 赋值
- When / Switch
- 组件层次结构
- Area
- 函数
- 实用函数

- VHDL 生成器

15.4.1 时钟域定义

在 *Spinal* 中，时钟和复位信号可以组合起来创建 ** 时钟域 **。时钟域可以应用于设计的某些区域，该区域的所有实例化的同步元件将 ** 隐式地 ** 使用该时钟域。

时钟域像堆栈一样工作，这意味着，如果您的逻辑位于给定时钟域中，您仍然可以在其上应用另一个时钟域。

时钟域语法

定义时钟域的语法如下（使用 EBNF 语法）：

```
ClockDomain(clock : Bool[,reset : Bool[,enable : Bool]])
```

这个定义需要三个参数：

1. 时钟域的时钟信号
2. 可选的 `reset` 复位信号。如果需要重置的寄存器并且其时钟域没有提供重置，则会出现错误提示
3. 可选的 `enable` 使能信号。该信号的目标是禁用整个时钟域上的时钟，而无需在每个同步元件上手动实现。

在设计中定义具有指定属性时钟域的示例如下：

```
val coreClock = Bool()
val coreReset = Bool()

// Define a new clock domain
val coreClockDomain = ClockDomain(coreClock,coreReset)

...

// Use this domain in an area of the design
val coreArea = new ClockingArea(coreClockDomain) {
  val coreClockedRegister = Reg(UInt(4 bits))
}
```

时钟配置

除了此处 给出的构造函数参数之外，每个时钟域的以下元素都可以通过 `ClockDomainConfig` 类进行配置：

属性	有效值
clockEdge	RISING, FALLING
ResetKind	ASYNC, SYNC
resetActiveHigh	true, false
clockEnableActiveHigh	true, false

```
class CustomClockExample extends Component {
  val io = new Bundle {
    val clk = in Bool()
    val resetn = in Bool()
    val result = out UInt (4 bits)
  }
  val myClockDomainConfig = ClockDomainConfig(
    clockEdge = RISING,
```

(续下页)

(接上页)

```
    resetKind = ASYNC,
    resetActiveLevel = LOW
  )
  val myClockDomain = ClockDomain(io.clk,io.resetn,config = myClockDomainConfig)
  val myArea = new ClockingArea(myClockDomain) {
    val myReg = Reg(UInt(4 bits)) init(7)
    myReg := myReg + 1

    io.result := myReg
  }
}
```

默认情况下，时钟域应用于整个设计。缺省的配置是：

- clock： 上升沿触发
- reset：异步，高电平有效
- 无使能信号

外部时钟

您可以在任何地方定义由外部驱动的时钟域。然后，它会自动将时钟和复位线从顶层输入添加到所有同步元件。

```
class ExternalClockExample extends Component {
  val io = new Bundle {
    val result = out UInt (4 bits)
  }
  val myClockDomain = ClockDomain.external("myClockName")
  val myArea = new ClockingArea(myClockDomain) {
    val myReg = Reg(UInt(4 bits)) init(7)
    myReg := myReg + 1

    io.result := myReg
  }
}
```

跨时钟域设计

SpinalHDL 在编译时检查是否存在不需要的/未指定的跨时钟域信号访问。如果您想读取另一个 ClockDomain （时钟域）发出的信号，则应给目标信号增加 crossClockDomain 标记，如下例所示：

```
val asynchronousSignal = UInt(8 bits)
...
val buffer0 = Reg(UInt(8 bits)).addTag(crossClockDomain)
val buffer1 = Reg(UInt(8 bits))
buffer0 := asynchronousSignal
buffer1 := buffer0   // Second register stage to be avoid metastability issues
```

```
// Or in less lines:
val buffer0 = RegNext(asynchronousSignal).addTag(crossClockDomain)
val buffer1 = RegNext(buffer0)
```

15.4.2 赋值

有多种赋值运算符：

Symbol	描述
:=	标准赋值，相当于 VHDL/Verilog 中的“<=” 最后一次分配获胜，值在下一个 delta 周期更新
/=	相当于 VHDL 中的:= 和 Verilog 中的 =（不常用） 值立即更新
<>	2 个信号之间的自动连接。通过输入/输出设置推断信号方向 Similar behavioral than :=

```
// Because of hardware concurrency is always read with the value '1' by b and c
val a,b,c = UInt(4 bits)
a := 0
b := a
a := 1  // a := 1 win
c := a

var x = UInt(4 bits)
val y,z = UInt(4 bits)
x := 0
y := x      // y read x with the value 0
x \= x + 1
z := x      // z read x with the value 1
```

SpinalHDL 检查左右分配端的位数是否匹配。有多种方法可以改变 BitVector （Bits、UInt、SInt）的位数：

改变位宽的方式	描述
x := y.resized	将 y 改变位宽后的副本赋值给 x，自动推断位宽以匹配 x
x := y.resize(newWidth)	将 y 改变位宽后的副本分配给 x，大小是手动计算的

There are 2 cases where spinal automatically resize things :

Assignment	问题	SpinalHDL 行为
myUIntOf_8bit := U(3)	U(3) 创建一个 2 位的 UInt，与左侧不匹配	由于 U(3) 是“弱”位推断信号，SpinalHDL 自动调整其位宽
myUIntOf_8bit := U(2 -> False default -> true)	右侧部分推断出 3 位 UInt，与左侧部分不匹配	SpinalHDL 将默认值重新应用于丢失的位

15.4.3 When / Switch

与 VHDL 和 Verilog 一样，信号线和寄存器可以通过使用 when 和 switch 语法进行条件赋值

```
when(cond1) {
  // execute when      cond1 is true
}.elsewhen(cond2) {
  // execute when (not cond1) and cond2
}.otherwise {
  // execute when (not cond1) and (not cond2)
}

switch(x) {
  is(value1) {
    // execute when x === value1
  }
  is(value2) {
    // execute when x === value2
  }
  default {
    // execute if none of precedent condition meet
  }
}
```

您还可以在 when/switch 语句中定义新信号。如果您想计算中间值时，它很有用。

```
val toto,titi = UInt(4 bits)
val a,b = UInt(4 bits)

when(cond) {
  val tmp = a + b
  toto := tmp
  titi := tmp + 1
} otherwise {
  toto := 0
  titi := 0
}
```

15.4.4 组件/层次结构

与 VHDL 和 Verilog 一样，您可以定义可用于构建设计层次结构的组件。但与它们不同的是，您不需要在实例化时绑定它们。

```
class AdderCell extends Component {
  // Declaring all in/out in an io Bundle is probably a good practice
  val io = new Bundle {
    val a, b, cin = in Bool()
    val sum, cout = out Bool()
  }
  // Do some logic
  io.sum := io.a ^ io.b ^ io.cin
  io.cout := (io.a & io.b) | (io.a & io.cin) | (io.b & io.cin)
}

class Adder(width: Int) extends Component {
  ...
  // Create 2 AdderCell
  val cell0 = new AdderCell
  val cell1 = new AdderCell
  cell1.io.cin := cell0.io.cout // Connect carrys
```

(续下页)

(接上页)

```
  ...
  val cellArray = Array.fill(width)(new AdderCell)
  ...
}
```

定义输入/输出的语法如下：

语法	描述	返回类型
in/out(x : Data)	设置 x 为输入/输出	x
in/out Bool()	创建输入/输出 Bool 值	Bool
in/out Bits/UInt/SInt(x bits)	创建相应类型的输入/输出端口	T

组件互连有一些规则：

- 组件只能读取子组件的输出/输入信号值
- 组件可以读取输出/输入端口值
- 如果由于某种原因，您需要从层次结构中的远处读取信号时（调试、临时补丁），您可以使用 some.where.else.theSignal.pull() 返回的值来实现。

15.4.5 Area

有时，创建一个组件来定义某些逻辑是多余的并且过于冗长。对于这种情况，您可以使用 Area 逻辑区：

```
class UartCtrl extends Component {
  ...
  val timer = new Area {
    val counter = Reg(UInt(8 bits))
    val tick = counter === 0
    counter := counter - 1
    when(tick) {
      counter := 100
    }
  }
  val tickCounter = new Area {
    val value = Reg(UInt(3 bits))
    val reset = False
    when(timer.tick) {          // Refer to the tick from timer area
      value := value + 1
    }
    when(reset) {
      value := 0
    }
  }
  val stateMachine = new Area {
    ...
  }
}
```

15.4.6 函数

使用 Scala 函数生成硬件的方式与 VHDL/Verilog 完全不同，原因有很多：

- 您可以在其中实例化寄存器、组合逻辑和组件。
- 您不必使用限制信号分配范围的 `process/@always`
- 一切都按参考工作，这允许许多操作。
  For example you can give to a function an bus as argument, then the function can internally read/write it.
  您还可以返回一个组件、总线以及 scala 世界中的任何其他内容。

RGB 信号转灰度信号

例如，如果您想使用系数将红/绿/蓝颜色转换为灰色，您可以使用函数来应用它们：

```
// Input RGB color
val r,g,b = UInt(8 bits)

// Define a function to multiply a UInt by a scala Float value.
def coef(value : UInt,by : Float) : UInt = (value * U((255*by).toInt,8 bits) >> 8)

// Calculate the gray level
val gray = coef(r,0.3f) +
           coef(g,0.4f) +
           coef(b,0.3f)
```

Valid Ready Payload 总线

例如，如果您定义一个简单的 Valid Ready Payload 总线，则可以在其中定义有用的函数。

```
class MyBus(payloadWidth:  Int) extends Bundle {
  val valid = Bool()
  val ready = Bool()
  val payload = Bits(payloadWidth bits)

  // connect that to this
  def <<(that: MyBus) : Unit = {
    this.valid := that.valid
    that.ready := this.ready
    this.payload := that.payload
  }

  // Connect this to the FIFO input, return the fifo output
  def queue(size: Int): MyBus = {
    val fifo = new Fifo(payloadWidth, size)
    fifo.io.push << this
    return fifo.io.pop
  }
}
```

15.4.7 VHDL 生成

有一个小组件和一个生成相应 VHDL 的 `main` 。

```
// spinal.core contain all basics (Bool, UInt, Bundle, Reg, Component, ..)
import spinal.core._

// A simple component definition
class MyTopLevel extends Component {
  // Define some input/output. Bundle like a VHDL record or a verilog struct.
  val io = new Bundle {
    val a = in Bool()
    val b = in Bool()
    val c = out Bool()
  }

  // Define some asynchronous logic
  io.c := io.a & io.b
}

// This is the main of the project. It create a instance of MyTopLevel and
// call the SpinalHDL library to flush it into a VHDL file.
object MyMain {
  def main(args: Array[String]) {
    SpinalVhdl(new MyTopLevel)
  }
}
```

15.4.8 实例化 VHDL 和 Verilog IP

在某些情况下，将 VHDL 或 Verilog 组件实例化到 SpinalHDL 设计中可能会很有用。为此，您需要定义 BlackBox，它就像一个组件，但其内部实现应由单独的 VHDL/Verilog 文件提供给仿真/综合工具。

```
class Ram_1w_1r(_wordWidth: Int, _wordCount: Int) extends BlackBox {
  val generic = new Generic {
    val wordCount = _wordCount
    val wordWidth = _wordWidth
  }

  val io = new Bundle {
    val clk = in Bool()

    val wr = new Bundle {
      val en = in Bool()
      val addr = in UInt (log2Up(_wordCount) bits)
      val data = in Bits (_wordWidth bits)
    }
    val rd = new Bundle {
      val en = in Bool()
      val addr = in UInt (log2Up(_wordCount) bits)
      val data = out Bits (_wordWidth bits)
    }
  }

  mapClockDomain(clock=io.clk)
}
```

15.4.9 实用工具

SpinalHDL 核心包含一些实用工具：

语法	描述	返回类型
log2Up(x : BigInt)	返回表示 x 状态所需的位数	Int
isPow2(x : BigInt)	如果 x 是 2 的幂，则返回 true	Boolean

Spindle.lib 中提供了更多工具和实用程序

15.5 Element

可以这样定义元素：

元素语法	描述
x : Int -> y : Boolean/Bool	用 y 设置位 x
x : Range -> y : Boolean/Bool	设置 x 范围内的每个位为 y
x : Range -> y : T	设置 x 范围内的位为 y
x : Range -> y : String	设置 x 范围内的位为 y 字符串格式遵循与 B”xyz”相同的规则
default -> y : Boolean/Bool	使用 y 值设置所有未连接的位。 此功能只允许在没有 B 前缀或 B 前缀与位规范相结合的情况下进行赋值

15.6 范围

您可以定义一个范围值

范围语法	描述	位宽
(x downto y)	[x:y], x >= y	x-y+1
(x to y)	[x:y], x <= y	y-x+1
(x until y)	[x:y[, x < y	y-x

CHAPTER 16

开发者专区

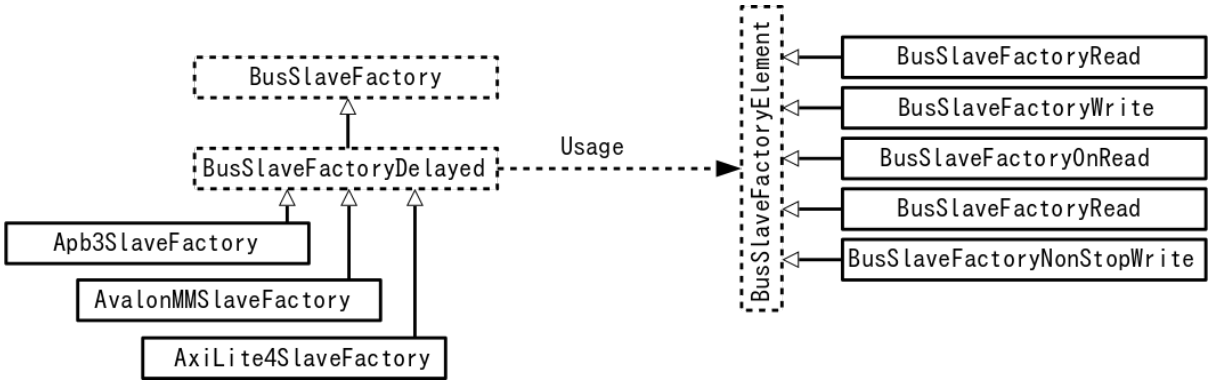
16.1 总线从端（Factory）实现

16.1.1 简介

本页将记录 BusSlaveFactory 工具及其变体之一的实现。您可以在此处获取有关该工具功能的更多信息*here*。

16.1.2 规范

类图如下：



`BusSlaveFactory` 抽象类定义了每个实现应提供的最低要求：

名称	描述
busDataWidth	返回总线的数据宽度
read(that,address,bitOffset)	当通过总线读取地址 address 时，用 that 中 bitOffset 位置的数据填充响应
write(that,address,bitOffset)	当通过总线写入地址 address 时，将总线上 bitOffset 位置的数据赋值 that
onWrite(address)(doThat)	当 address 地址上发生写操作（出现写事务）时调用 doThat
onRead(address)(doThat)	当 address 上发生读操作（出现读事务）时调用 doThat
nonStopWrite(that,bitOffset)	将通过总线写入的 bitOffset 的数据赋值到 that

通过使用它们，BusSlaveFactory 还能够提供许多实用工具：

名称	返回类型	描述
readAndWrite(that,address,bitOffset)		使 that 信号可通过 address 地址读写，并且该信号放置在数据的 bitOffset 位置
readMultiWord(that,address)		创建内存映射以从‘address’地址读取 that 信号。: 如果 that 的位宽大于一个字（32 位），它将在以下地址上扩展寄存器
writeMultiWord(that,address)		创建内存映射以通过总线‘address’地址写入 that 信号。: 如果 that 的位宽大于一个字（32 位），它将在以下地址上扩展寄存器
createWriteOnly(dataType,address,bitOffset)	T	在 address 地址处创建一个 dataType 类型的只写寄存器，并将其放置在字中的 bitOffset 位置
createReadWrite(dataType,address,bitOffset)	T	在 address 处创建一个 dataType 类型的读写寄存器，并将其放置在字中的 bitOffset 位置
createAndDriveFlow(dataType,address,bitOffset)	Flow[T]	在 address 地址处创建一个 dataType 类型的可写流（Flow）寄存器，并将其放置在字中的 bitOffset 位置
drive(that,address,bitOffset)		使用位于 address 地址的可写寄存器中 bitOffset 位置的信号驱动 that
driveAndRead(that,address,bitOffset)		使用位于 address 地址的可读写寄存器中 bitOffset 位置的信号驱动 that
driveFlow(that,address,bitOffset)		当对 address 地址写入时，通过使用位于 bitOffset 位的数据，在 that 流（Flow）上发出事务
readStreamNonBlocking(that,address,validBitOffset,payloadBitOffset)		读取 that 信号并在读取 address 地址时消耗事务。 valid <= validBitOffset bit payload <= payloadBitOffset+widthOf(payload) downto payloadBitOffset
doBitsAccumulationAndClearOnRead(that,address,bitOffset)		实例化一个内部寄存器，该寄存器在每个周期执行以下操作: reg := reg | that 然后，当发生读取时，寄存器被清除。该寄存器可通过 address 地址读取，并放置在字中的 bitOffset 位置

关于 BusSlaveFactoryDelayed，它仍然是一个抽象类，但它捕获每个原语（BusSlaveFactoryElement）对数据模型的调用。该数据模型是一个包含所有原语的列表，也是一个 HashMap，它将使用的每个地址链接到正在使用它的原语列表。然后，当它们全部被收集时（在当前组件的末尾），它会执行一个回调，该回调应该由扩展它的类实现。该回调函数中实现了这个原语列表中每个原语相对应的硬件。

16.1.3 实现

BusSlaveFactory

让我们来描述一下原语抽象函数：

```
trait BusSlaveFactory  extends Area {

  def busDataWidth : Int

  def read(that : Data,
           address : BigInt,
           bitOffset : Int = 0) : Unit

  def write(that : Data,
            address : BigInt,
            bitOffset : Int = 0) : Unit

  def onWrite(address : BigInt)(doThat : => Unit) : Unit
  def onRead (address : BigInt)(doThat : => Unit) : Unit

  def nonStopWrite( that : Data,
                    bitOffset : Int = 0) : Unit

  // ...
}
```

然后让我们利用这些来实现一些有用的工具：

```
trait BusSlaveFactory  extends Are {
  // ...
  def readAndWrite(that : Data,
                   address: BigInt,
                   bitOffset : Int = 0): Unit = {
    write(that,address,bitOffset)
    read(that,address,bitOffset)
  }

  def drive(that : Data,
            address : BigInt,
            bitOffset : Int = 0) : Unit = {
    val reg = Reg(that)
    write(reg,address,bitOffset)
    that := reg
  }

  def driveAndRead(that : Data,
                   address : BigInt,
                   bitOffset : Int = 0) : Unit = {
    val reg = Reg(that)
    write(reg,address,bitOffset)
    read(reg,address,bitOffset)
    that := reg
  }

  def driveFlow[T <: Data](that : Flow[T],
                           address: BigInt,
                           bitOffset : Int = 0) : Unit = {
    that.valid := False
    onWrite(address) {
      that.valid := True
    }
```

(续下页)

(接上页)

```
    nonStopWrite(that.payload,bitOffset)
  }

  def createReadWrite[T <: Data](dataType: T,
                                 address: BigInt,
                                 bitOffset : Int = 0): T = {
    val reg = Reg(dataType)
    write(reg,address,bitOffset)
    read(reg,address,bitOffset)
    reg
  }

  def createAndDriveFlow[T <: Data](dataType : T,
                                 address: BigInt,
                                 bitOffset : Int = 0) : Flow[T] = {
    val flow = Flow(dataType)
    driveFlow(flow,address,bitOffset)
    flow
  }

  def doBitsAccumulationAndClearOnRead(   that : Bits,
                                          address : BigInt,
                                          bitOffset : Int = 0): Unit = {
    assert(that.getWidth <= busDataWidth)
    val reg = Reg(that)
    reg := reg | that
    read(reg,address,bitOffset)
    onRead(address) {
      reg := that
    }
  }

  def readStreamNonBlocking[T <: Data] (that : Stream[T],
                                        address: BigInt,
                                        validBitOffset : Int,
                                        payloadBitOffset : Int) : Unit = {
    that.ready := False
    onRead(address) {
      that.ready := True
    }
    read(that.valid  ,address,validBitOffset)
    read(that.payload,address,payloadBitOffset)
  }

  def readMultiWord(that : Data,
                address : BigInt) : Unit  = {
    val wordCount = (widthOf(that) - 1) / busDataWidth + 1
    val valueBits = that.asBits.resize(wordCount*busDataWidth)
    val words = (0 until wordCount).map(id => valueBits(id * busDataWidth ,␣
↪busDataWidth bits))
    for (wordId <- (0 until wordCount)) {
      read(words(wordId), address + wordId*busDataWidth/8)
    }
  }

  def writeMultiWord(that : Data,
                 address : BigInt) : Unit  = {
    val wordCount = (widthOf(that) - 1) / busDataWidth + 1
    for (wordId <- (0 until wordCount)) {
      write(
        that = new DataWrapper {
```

(续下页)

(接上页)

```
          override def getBitsWidth: Int =
            Math.min(busDataWidth, widthOf(that) - wordId * busDataWidth)

          override def assignFromBits(value : Bits): Unit = {
            that.assignFromBits(
              bits     = value.resized,
              offset   = wordId * busDataWidth,
              bitCount = getBitsWidth bits)
          }
        },address = address + wordId * busDataWidth / 8,0
      )
    }
  }
}
```

BusSlaveFactoryDelayed

让我们实现用于存储原语的类：

```
trait BusSlaveFactoryElement

// Ask to make `that` readable when a access is done on `address`.
// bitOffset specify where `that` is placed on the answer
case class BusSlaveFactoryRead(that : Data,
                               address : BigInt,
                               bitOffset : Int) extends BusSlaveFactoryElement

// Ask to make `that` writable when a access is done on `address`.
// bitOffset specify where `that` get bits from the request
case class BusSlaveFactoryWrite(that : Data,
                                address : BigInt,
                                bitOffset : Int) extends BusSlaveFactoryElement

// Ask to execute `doThat` when a write access is done on `address`
case class BusSlaveFactoryOnWrite(address : BigInt,
                                  doThat : () => Unit) extends␣
↪BusSlaveFactoryElement

// Ask to execute `doThat` when a read access is done on `address`
case class BusSlaveFactoryOnRead( address : BigInt,
                                  doThat : () => Unit) extends␣
↪BusSlaveFactoryElement

// Ask to constantly drive `that` with the data bus
// bitOffset specify where `that` get bits from the request
case class BusSlaveFactoryNonStopWrite(that : Data,
                                       bitOffset : Int) extends␣
↪BusSlaveFactoryElement
```

然后让我们实现 BusSlaveFactoryDelayed 本身：

```
trait BusSlaveFactoryDelayed extends BusSlaveFactory {
  // elements is an array of all BusSlaveFactoryElement requested
  val elements = ArrayBuffer[BusSlaveFactoryElement]()


  // elementsPerAddress is more structured than elements, it group all␣
↪BusSlaveFactoryElement per requested addresses
  val elementsPerAddress = collection.mutable.HashMap[BigInt,
```

(续下页)

(接上页)

```
↪ArrayBuffer[BusSlaveFactoryElement]]()

  private def addAddressableElement(e : BusSlaveFactoryElement,address : BigInt) =
↪{
    elements += e
    elementsPerAddress.getOrElseUpdate(address,␣
↪ArrayBuffer[BusSlaveFactoryElement]()) += e
  }

  override def read(that : Data,
            address : BigInt,
            bitOffset : Int = 0) : Unit  = {
    assert(bitOffset + that.getBitsWidth <= busDataWidth)
    addAddressableElement(BusSlaveFactoryRead(that,address,bitOffset),address)
  }

  override def write(that : Data,
             address : BigInt,
             bitOffset : Int = 0) : Unit  = {
    assert(bitOffset + that.getBitsWidth <= busDataWidth)
    addAddressableElement(BusSlaveFactoryWrite(that,address,bitOffset),address)
  }

  def onWrite(address : BigInt)(doThat : => Unit) : Unit = {
    addAddressableElement(BusSlaveFactoryOnWrite(address,() => doThat),address)
  }
  def onRead (address : BigInt)(doThat : => Unit) : Unit = {
    addAddressableElement(BusSlaveFactoryOnRead(address,() => doThat),address)
  }

  def nonStopWrite( that : Data,
                    bitOffset : Int = 0) : Unit = {
    assert(bitOffset + that.getBitsWidth <= busDataWidth)
    elements += BusSlaveFactoryNonStopWrite(that,bitOffset)
  }

  // This is the only thing that should be implement by class that extends␣
↪BusSlaveFactoryDelayed
  def build() : Unit

  component.addPrePopTask(() => build())
}
```

AvalonMMSlaveFactory

首先，让我们实现提供兼容 AvalonMM 配置对象的伴随对象，对应于下表：

信号名称	类型	描述
read	Bool	保持一个周期高电平来产生一个读请求
write	Bool	保持一个周期高电平来产生一个写请求
address	UInt(addressWidth bits)	字节为粒度但是字对齐的
writeData	Bits(dataWidth bits)	
readDataValid	Bool	保持高电平来响应读命令
readData	Bits(dataWidth bits)	readDataValid 为高时有效

```
object AvalonMMSlaveFactory {
  def getAvalonConfig( addressWidth : Int,
```

(续下页)

(接上页)

```
                        dataWidth : Int) = {
    AvalonMMConfig.pipelined(    // Create a simple pipelined configuration of the␣
↪Avalon Bus
      addressWidth = addressWidth,
      dataWidth = dataWidth
    ).copy(                      // Change some parameters of the configuration
      useByteEnable = false,
      useWaitRequestn = false
    )
  }

  def apply(bus : AvalonMM) = new AvalonMMSlaveFactory(bus)
}
```

然后，让我们实现 AvalonMMSlaveFactory 本身。

```
class AvalonMMSlaveFactory(bus : AvalonMM) extends BusSlaveFactoryDelayed {
  assert(bus.c == AvalonMMSlaveFactory.getAvalonConfig(bus.c.addressWidth,bus.c.
↪dataWidth))

  val readAtCmd = Flow(Bits(bus.c.dataWidth bits))
  val readAtRsp = readAtCmd.stage()

  bus.readDataValid := readAtRsp.valid
  bus.readData := readAtRsp.payload

  readAtCmd.valid := bus.read
  readAtCmd.payload := 0

  override def build(): Unit = {
    for(element <- elements) element match {
      case element : BusSlaveFactoryNonStopWrite =>
        element.that.assignFromBits(bus.writeData(element.bitOffset, element.that.
↪getBitsWidth bits))
      case _ =>
    }

    for((address,jobs) <- elementsPerAddress) {
      when(bus.address === address) {
        when(bus.write) {
          for(element <- jobs) element match {
            case element : BusSlaveFactoryWrite => {
              element.that.assignFromBits(bus.writeData(element.bitOffset, element.
↪that.getBitsWidth bits))
            }
            case element : BusSlaveFactoryOnWrite => element.doThat()
            case _ =>
          }
        }
        when(bus.read) {
          for(element <- jobs) element match {
            case element : BusSlaveFactoryRead => {
              readAtCmd.payload(element.bitOffset, element.that.getBitsWidth bits)␣
↪:= element.that.asBits
            }
            case element : BusSlaveFactoryOnRead => element.doThat()
            case _ =>
          }
        }
      }
    }
```

(续下页)

(接上页)

```
  }

  override def busDataWidth: Int = bus.c.dataWidth
}
```

16.1.4 结论

就这些了，您可以在此处查看一个使用 Apb3SlaveFactory 来创建 Apb3UartCtrl 的示例。

如果您想添加对新内存总线的支持，非常简单，您只需继承、实现 BusSlaveFactoryDelayed 特征（trait）的另一个变体即可。Apb3SlaveFactory 可能是一个很好的参考：D

16.2 项目中如何使用本地的 SpinalHDL 克隆作为依赖

使用 SpinalHDL 的默认方式是通过 sbt 或 mill 自动下载其发布的版本。如果你想使用最新版本，你可能会要使用 git 直接从上游获取未发布的“dev”分支。这可能是因为你想要使用一个新功能，或者因为你想要在一个项目中测试你自己的 Spinal 扩展（请考虑通过打开一个 PR 来提交它）。

有关使用的示例，请参阅 VexiiRiscv 。

16.2.1 创建本地的 SpinalHDL git 克隆

```
cd /somewhere
git clone --depth 1 -b dev https://github.com/SpinalHDL/SpinalHDL.git
```

在上面的命令中，可以用要签出的分支名称替换 dev 。--depth 1 阻止下载版本库历史。

16.2.2 配置构建系统

根据您使用的工具，将本地 git 文件夹作为依赖项加载的 sbt 或 mill 指令：

配置 sbt (更新 build.sbt)

```
ThisBuild / version := "1.0"              // change as needed
ThisBuild / scalaVersion := "2.12.18"     // change as needed
ThisBuild / organization := "org.example" // change as needed

val spinalRoot = file("/somewhere/SpinalHDL")
lazy val spinalIdslPlugin = ProjectRef(spinalRoot, "idslplugin")
lazy val spinalSim = ProjectRef(spinalRoot, "sim")
lazy val spinalCore = ProjectRef(spinalRoot, "core")
lazy val spinalLib = ProjectRef(spinalRoot, "lib")

lazy val projectname = (project in file("."))
.settings(
   Compile / scalaSource := baseDirectory.value / "hw" / "spinal",
).dependsOn(spinalIdslPlugin, spinalSim, spinalCore, spinalLib)

scalacOptions += (spinalIdslPlugin / Compile / packageBin / artifactPath).map {␣
↪file =>
  s"-Xplugin:${file.getAbsolutePath}"
}.value
```

(续下页)

(接上页)

```
fork := true
```

配置 mill (更新 build.sc)

```
import mill._, scalalib._

import $file.^.SpinalHDL.build
import ^.SpinalHDL.build.{core => spinalCore}
import ^.SpinalHDL.build.{lib => spinalLib}
import ^.SpinalHDL.build.{idslplugin => spinalIdslplugin}

val spinalVers = "1.10.2a"
val scalaVers = "2.12.18"

object projectname extends RootModule with SbtModule {
   def scalaVersion = scalaVers
   def sources = T.sources(
      this.millSourcePath / "hw" / "spinal"
   )

   def idslplugin = spinalIdslplugin(scalaVers)
   def moduleDeps = Seq(
      spinalCore(scalaVers),
      spinalLib(scalaVers),
      idslplugin
   )
   def scalacOptions = super.scalacOptions() ++ idslplugin.pluginOptions()
}
```

注意 `import $file.^.SpinalHDL.build` 行。它使用了 ammonite REPL 的魔法 `$file` 来查找 SpinalHDL 的 `build.sc`。(^ 从当前目录向上移动一个目录。) 假设目录结构如下：

```
/somewhere
|-SpinalHDL    # <-- cloned spinal git
| |-build.sc
|-projectname
| |-build.sc  # <-- your project, mill is ran from here
```

16.2.3 完成

请注意添加到 `scalacOptions` 的内容。如果没有它，编译任何 Spinal 项目都可能产生无数的 `SCOPE VIOLATION` 或 `HIERARCHY VIOLATION` 错误，因为 spinal 的 `idslplugin` 实际上并没有被调用。

更改后，下一次编译项目将花费大量时间（约 2 分钟），但这只是第一次编译。在此之后，项目的编译时间应该和往常一样。

16.3 如何修改本文档

如果您希望将您的页面添加到此文档中，需要将源文件添加到适当的部分。我选择创建一个结构来重新整理文档的各个部分，这并不是严格必要的，但为了清晰性，强烈建议这样做。

本文档使用递归索引树：每个文件夹都有一个特殊的 `index.rst` 文件，告诉 sphinx 哪个文件存在、以什么顺序将其放入文档树中。

16.3.1 标题约定

Sphinx 非常智能，文档结构是根据您如何使用非字母数字字符（例如：`= - ` : ' " ~ ^ _ * + # < >`）推导出来的，您只需要保持一致即可。不过，为了保持一致性，我们使用这些约定：

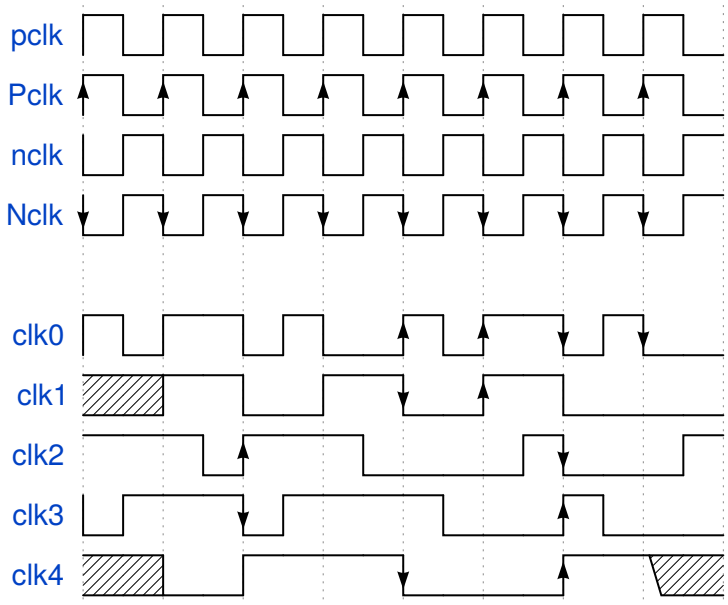
- = 用于章节标题的前后行
- = 标题的下划线
- - 段落的下划线
- ^ 表示子段落

16.3.2 Wavedrom 的集成

本文档使用了 `sphinxcontrib-wavedrom` 插件，因此您可以使用 WaveJSON 语法指定时序图或寄存器描述，如下所示：

```
.. wavedrom::

   { "signal": [
      { "name": "pclk", "wave": "p......." },
      { "name": "Pclk", "wave": "P......." },
      { "name": "nclk", "wave": "n......." },
      { "name": "Nclk", "wave": "N......." },
      {},
      { "name": "clk0", "wave": "phnlPHNL" },
      { "name": "clk1", "wave": "xhlhLHl." },
      { "name": "clk2", "wave": "hpHplnLn" },
      { "name": "clk3", "wave": "nhNhplPl" },
      { "name": "clk4", "wave": "xlh.L.Hx" }
   ]}
```

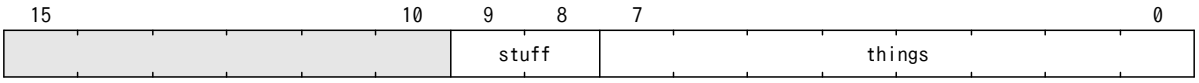
你可以得到：



备注： 如果您希望 Wavedrom 图出现在 pdf 导出中，则需要使用“non relaxed” JSON 方言。长话短说，就是没有使用 javascript 代码并使用 " 包围键值（例如 "name" ）。

您可以使用相同的语法描述寄存器映射：

```
{"reg":[
  {"bits": 8, "name": "things"},
  {"bits": 2, "name": "stuff" },
  {"bits": 6}
 ],
 "config": { "bits":16,"lanes":1 }
 }
```



16.3.3 新章节

如果你想添加一个新的节，你需要在顶部索引中指定新节的索引文件。我建议将文件夹命名为节名称，但这不是必需的；Sphinx 将从索引文件的标题中获取该部分的名称。

示例

我想记录 SpinalHDL 中的新功能，并且我想为其创建一个章节；我们称之为 Cheese

所以我需要创建一个名为 Cheese 的文件夹（名称并不重要），并在其中创建一个索引文件，例如：

```
======
Cheese
======

.. toctree::
:glob:

introduction
*
```

备注： `.. toctree::` 指令接受一些参数，在本例中 `:glob:` 使您可以使用 `*` 包含所有剩余文件。

备注： 文件路径是相对于索引文件的，如果要指定绝对路径，需要在前面加上 "/"

备注： `introduction.rst` 将始终是列表中的第一个，因为它是在索引文件中指定的。其他文件将按字母顺序排列。

现在我可以添加 `introduction.rst` 和其他文件，比如 `cheddar.rst`，`stilton.rst` 等。

剩下要做的唯一一件事就是将 `cheese` 添加到顶部索引文件中，如下所示：

```
Welcome to SpinalHDL's documentation!
=====================================

.. toctree::
   :maxdepth: 2
   :titlesonly:

   rst/About SpinalHDL/index
   rst/Getting Started/index
   rst/Data types/index
   rst/Structuring/index
   rst/Semantic/index
   rst/Sequential logic/index
   rst/Design errors/index
   rst/Other language features/index
   rst/Libraries/index
   rst/Simulation/index
   rst/Examples/index
   rst/Legacy/index
   rst/Developers area/index
   rst/Cheese/index
```

就是这样，现在您可以在 `cheese` 中添加您想要的所有内容，所有页面都将显示在文档中。

16.4 通过 Mill 构建（输出）

SpinalHDL 本身可以用 Mill 构建。这是一个 Sbt 构建工具的替代品，可以在 Introduction_to_Mill 中找到。它可以编译/测试/发布本地现有模块。通过 mill 构建可以比 Sbt 快得多，这在调试时很有用。

16.4.1 编译 SpinalHDL 库

```
mill __.compile
sbt compile # equivalent alternatives
```

16.4.2 运行所有测试套件

```
mill __.test
sbt test # equivalent alternatives
```

16.4.3 运行指定的测试套件

```
mill tester.test.testOnly spinal.xxxxx.xxxxx
sbt "tester/testOnly spinal.xxxxx.xxxxx" # equivalent alternatives
```

16.4.4 运行指定程序（App）

```
mill tester.runMain spinal.xxxxx.xxxxx
sbt "tester/runMain spinal.xxxxx.xxxxx" # equivalent alternatives
```

16.4.5 本地发布

Mill 还可以将库作为 dev 版本发布到本地 ivy2 存储库。

```
mill __.publishLocal
sbt publishLocal # equivalent alternatives
```

16.5 SpinalHDL 内部数据模型

16.5.1 简介

本页面提供有关 SpinalHDL 使用的内部数据结构的文档，用于存储和修改用户通过 SpinalHDL API 描述的网表。

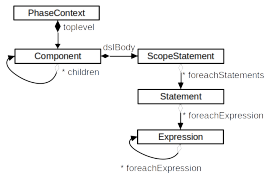
16.5.2 总体结构

下图遵循 UML 命名法：

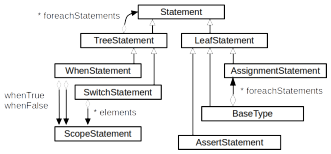
- 带有白色箭头的链接表示“源继承目标”
- 带有黑色菱形的链接表示“源包含目标”
- 带有白色菱形的链接意味着“源中有一个对目标的引用”
- “* “ 符号表示“多个”

大多数数据结构都是使用双向链表存储，这方便了元素的插入和删除。

全局数据结构图如下：



这里是关于 *Statement* 类的更多细节：



一般来说，当数据模型内的元素使用其他表达式或语句时，该元素通常包括迭代这些用法的函数。例如，每个表达式都配有一个 *foreachExpression* 函数。

使用这些迭代函数时，您也可以从树中删除当前元素。

此外，作为旁注，虽然 *foreachXXX* 函数仅迭代一层深度，但通常有相应的 *walkXXX* 函数执行递归迭代。例如，在 *((a+b)+c)+d* 表达式上使用 *myExpression.walkExpression* ，这将遍历整个加法运算树。

还有像 *myExpression.remapExpressions(Expression => Expression),* 这样的实用工具，它会迭代 *myExpression* 中使用的所有表达式，并将它们替换为您提供的表达式。

More generally, most of the graph checks and transformations done by SpinalHDL are located in <https://github.com/SpinalHDL/SpinalHDL/blob/dev/core/src/main/scala/spinal/core/internals/Phase.scala>

16.5.3 探索数据模型

在不使用快捷方式的情况下，识别网表中所有加法器的示例如下：

```
object FindAllAddersManually {
  class Toplevel extends Component {
    val a,b,c = in UInt(8 bits)
    val result = out(a + b + c)
  }

  import spinal.core.internals._

  class PrintBaseTypes(message : String) extends Phase {
    override def impl(pc: PhaseContext) = {
      println(message)

      recComponent(pc.topLevel)

      def recComponent(c: Component): Unit = {
        c.children.foreach(recComponent)
        c.dslBody.foreachStatements(recStatement)
      }

      def recStatement(s: Statement): Unit = {
        s.foreachExpression(recExpression)
        s match {
          case ts: TreeStatement => ts.foreachStatements(recStatement)
          case _ =>
        }
      }

      def recExpression(e: Expression): Unit = {
        e match {
          case op: Operator.BitVector.Add => println(s"Found ${op.left} + ${op.
↪right}")
          case _ =>
        }
        e.foreachExpression(recExpression)
      }

    }
    override def hasNetlistImpact = false
```

(续下页)

(接上页)

```
    override def toString = s"${super.toString} - $message"
  }

  def main(args: Array[String]): Unit = {
    val config = SpinalConfig()

    // Add a early phase
    config.addTransformationPhase(new PrintBaseTypes("Early"))

    // Add a late phase
    config.phasesInserters += {phases =>
      phases.insert(phases.indexWhere(_.isInstanceOf[PhaseVerilog]), new␣
↪PrintBaseTypes("Late"))
    }
    config.generateVerilog(new Toplevel())
  }
}
```

这将生成：

```
[Runtime] SpinalHDL v1.6.1    git head : 3100c81b37a04715d05d9b9873c3df07a0786a9b
[Runtime] JVM max memory : 8044.0MiB
[Runtime] Current date : 2021.10.16 20:31:33
[Progress] at 0.000 : Elaborate components
[Progress] at 0.163 : Checks and transforms
Early
Found (toplevel/a : in UInt[8 bits]) + (toplevel/b : in UInt[8 bits])
Found (toplevel/??? :  UInt[? bits]) + (toplevel/c : in UInt[8 bits])
[Progress] at 0.191 : Generate Verilog
Late
Found (UInt + UInt)[8 bits] + (toplevel/c : in UInt[8 bits])
Found (toplevel/a : in UInt[8 bits]) + (toplevel/b : in UInt[8 bits])
[Done] at 0.218
```

请注意，在许多情况下，都可以使用快捷方式。前面提到的所有递归过程，都可以用一个递归过程代替。：

```
override def impl(pc: PhaseContext) = {
  println(message)
  pc.walkExpression {
    case op: Operator.BitVector.Add => println(s"Found ${op.left} + ${op.right}")
    case _ =>
  }
}
```

16.5.4 编译环节

以下是按顺序排列的默认环节的完整列表，用于从顶级组件修改、检查和生成 Verilog 代码。：

<https://github.com/SpinalHDL/SpinalHDL/blob/ec8cd9f513566b43cbbdb08d0df4dee1f0fee655/core/src/main/scala/spinal/core/internals/Phase.scala#L2487>

如果您作为用户使用 *SpinalConfig.addTransformationPhase(new MyPhase())* 添加新的编译环节，则该环节将在用户组件实例细化后立即插入，这在编译序列中相对较早。在此环节，您仍然可以使用完整的 SpinalHDL 用户 API 将元素引入网表。

如果您选择使用 SpinalConfig.phasesInserters API，则必须谨慎行事并确保对网表所做的任何修改与已执行的阶段保持一致。例如，如果您在 *PhaseInferWidth* 之后插入环节，则必须指定引入的每个节点的位宽。

16.5.5 在不使用插件的情况下，以用户身份修改网表

有多个用户 API 使您能够在用户实例细化环节进行修改。:

- mySignal.removeAssignments： 将删除所有先前对给定信号的赋值 :=
- mySignal.removeStatement : Will void the existence of the signal
- mySignal.setAsDirectionLess： 将输入/输出信号转换为内部信号（无方向）
- mySignal.setName： 在信号上强制指定名称（还有许多其他变体）
- mySubComponent.mySignal.pull()： 将提供给定信号的可读副本，即使该信号位于层次结构中的其他位置
- myComponent.rework{ myCode }：在 *myComponent* 上下文中执行 *myCode*，允许使用用户 API 修改它

例如，以下代码可用于修改顶级组件，向组件的每个输入和输出添加三级移位寄存器。这对于综合器测试特别有用。

```
def ffIo[T <: Component](c : T): T = {
  def buf1[T <: Data](that : T) = KeepAttribute(RegNext(that)).addAttribute("DONT_
↪TOUCH")
  def buf[T <: Data](that : T) = buf1(buf1(buf1(that)))
  c.rework {
    val ios = c.getAllIo.toList
    ios.foreach{io =>
      if(io.getName() == "clk") {
        // Do nothing
      } else if(io.isInput) {
        io.setAsDirectionLess().allowDirectionLessIo  // allowDirectionLessIo is␣
↪to disable the io Bundle linting
        io := buf(in(cloneOf(io).setName(io.getName() + "_wrap")))
      } else if(io.isOutput) {
        io.setAsDirectionLess().allowDirectionLessIo
        out(cloneOf(io).setName(io.getName() + "_wrap")) := buf(io)
      } else ???
    }
  }
  c
}
```

您可以通过以下方式使用该代码：

```
SpinalVerilog(ffIo(new MyToplevel))
```

这是一个函数，使您能够执行主体代码，就好像当前组件的上下文不存在一样。这对于定义新信号特别有用，这样信号不受当前条件范围（例如 when 或 switch）的影响。

```
def atBeginingOfCurrentComponent[T](body : => T) : T = {
  val body = Component.current.dslBody  // Get the head of the current component␣
↪symbols tree (AST in other words)
  val ctx = body.push()                 // Now all access to the SpinalHDL API␣
↪will be append to it (instead of the current context)
  val swapContext = body.swap()         // Empty the symbol tree (but keep a␣
↪reference to the old content)
  val ret = that                        // Execute the block of code (will be␣
↪added to the recently empty body)
  ctx.restore()                         // Restore the original context in which␣
↪this function was called
  swapContext.appendBack()              // append the original symbols tree to the␣
↪modified body
  ret                                   // return the value returned by that
}
```

(续下页)

(接上页)

```
val database = mutable.HashMap[Any, Bool]()
def get(key : Any) : Bool = {
  database.getOrElseUpdate(key, atBeginingOfCurrentComponent(False)
}

object key

when(something) {
  if(somehow) {
    get(key) := True
  }
}
when(database(key)) {
   ...
}
```

例如，这种功能被用在 VexRiscv 管道中，以根据需要动态创建组件或元素。

16.5.6 用户空间网表分析

SpinalHDL 的数据模型也是可访问的，并且可以在用户实力细化时读取。下面的示例可以帮助找到遍历信号列表的最短逻辑路径（就时钟周期而言）。在本例中，它用于分析 VexRiscv FPU 设计的延迟。

```
println("cpuDecode to fpuDispatch " + LatencyAnalysis(vex.decode.arbitration.
↪isValid, logic.decode.input.valid))
println("fpuDispatch to cpuRsp    " + LatencyAnalysis(logic.decode.input.valid,␣
↪plugin.port.rsp.valid))

println("cpuWriteback to fpuAdd   " + LatencyAnalysis(vex.writeBack.input(plugin.
↪FPU_COMMIT), logic.commitLogic(0).add.counter))

println("add                      " + LatencyAnalysis(logic.decode.add.rs1.
↪mantissa, logic.get.merge.arbitrated.value.mantissa))
println("mul                      " + LatencyAnalysis(logic.decode.mul.rs1.
↪mantissa, logic.get.merge.arbitrated.value.mantissa))
println("fma                      " + LatencyAnalysis(logic.decode.mul.rs1.
↪mantissa, logic.get.decode.add.rs1.mantissa, logic.get.merge.arbitrated.value.
↪mantissa))
println("short                    " + LatencyAnalysis(logic.decode.shortPip.rs1.
↪mantissa, logic.get.merge.arbitrated.value.mantissa))
```

在这里您可以找到该 LatencyAnalysis 工具的实现：<https://github.com/SpinalHDL/SpinalHDL/blob/3b87c898cb94dc08456b4fe2b1e8b145e6c86f63/lib/src/main/scala/spinal/lib/Utils.scala#L620>

16.5.7 遍历、枚举正在使用的每个时钟域

在本例中，这是在实例细化完成之后利用 SpinalHDL 报告完成的。

```
object MyTopLevelVerilog extends App {
  class MyTopLevel extends Component {
    val cdA = ClockDomain.external("rawrr")
    val regA = cdA(RegNext(False))

    val sub = new Component {
      val cdB = ClockDomain.external("miaou")
      val regB = cdB(RegNext(False))
```

(续下页)

(接上页)

```
      val clkC = CombInit(regB)
      val cdC = ClockDomain(clkC)
      val regC = cdC(RegNext(False))
    }
  }

  val report = SpinalVerilog(new MyTopLevel)

  val clockDomains = mutable.LinkedHashSet[ClockDomain]()
  report.toplevel.walkComponents(c =>
    c.dslBody.walkStatements(s =>
      s.foreachClockDomain(cd =>
        clockDomains += cd
      )
    )
  )

  println("ClockDomains : " + clockDomains.mkString(", "))
  val externals = clockDomains.filter(_.clock.component == null)
  println("Externals : " + externals.mkString(", "))
}
```

打印信息是

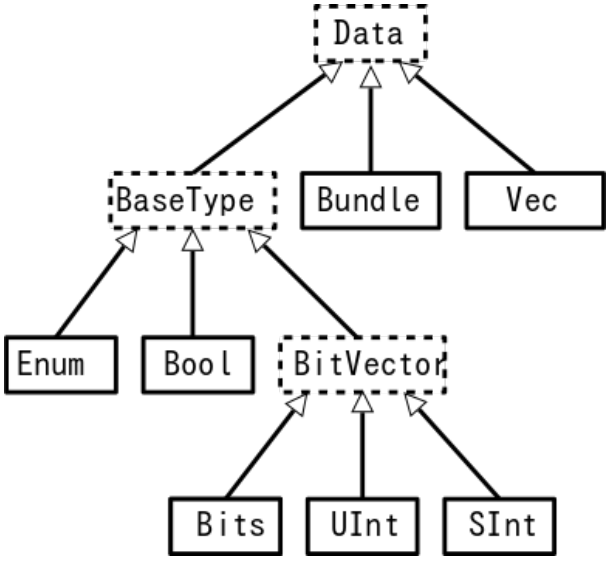
```
ClockDomains : rawrr_clk, miaou_clk, clkC
Externals : rawrr_clk, miaou_clk
```

16.6 类型

16.6.1 简介

该语言提供了 5 种基本类型和 2 种复合类型。

- 基本类型：Bool、Bits、UInt （无符号整数）、SInt`` （有符号整数）、``Enum。
- 复合类型：Bundle、Vec。



这些类型及其用法（包括示例）将在后面解释。

定点小数支持已归档*Fixed-Point*

16.6.2 Bool

这是对应于单个位的标准 *boolean* 类型。

声明

声明值/对象的语法如下：

语法	描述	返回类型
Bool()	创建 Bool 值	Bool
True	创建一个分配有 `true` 值的 Bool 对象	Bool
False	创建一个 Bool 值并赋值为 `false`	Bool
Bool(value : Boolean)	创建一个赋值有 Scala 布尔值的 Bool 信号	Bool

在 SpinalHDL 中使用这种类型会生成：

```
val myBool = Bool()
myBool := False          // := is the assignment operator
myBool := Bool(false)    // Use a Scala Boolean to create a literal
```

运算符

以下运算符可用于 `Bool` 类型

运算符	描述	返回类型
!x	逻辑非	Bool
x && y x & y	逻辑与	Bool
x || y x | y	逻辑或	Bool
x ^ y	逻辑异或	Bool
x.set[()]	将 x 设置为 True	
x.clear[()]	将 x 设置为 False	
x.rise[()]	当 x 在上一个周期为低电平且现在为高电平时返回 True	Bool
x.rise(initAt : Bool)	与 x.rise 相同但具有重置后的初始值	Bool
x.fall[()]	当 x 在上一个周期为高且现在为低时返回 True	Bool
x.fall(initAt : Bool)	与 x.fall 相同但具有重置后的初始值	Bool
x.setWhen(cond)	当 cond 为 True 时设置 x 为 True	Bool
x.clearWhen(cond)	当 cond 为 True 时设置 x 为 False	Bool

16.6.3 BitVector 系列 - (Bits, UInt, SInt)

BitVector 是一个系列的类型，用于在单个值中存储多位信息。该类型具有三个子类型，可用于描述不同的行为：
Bits 不传达任何符号信息，而 UInt （无符号整数）和 SInt （有符号整数）提供了计算正确结果所需的操作（如果使用有符号/无符号算术）。

声明语法

语法	描述	返回类型
Bits/UInt/SInt [()]	创建一个 BitVector，自动推断其位数	Bits/UInt/SInt
Bits/UInt/SInt(x bits)	创建一个具有 x 位的 BitVector 信号	Bits/UInt/SInt
B/U/S(value : Int[,width : BitCount])	创建一个赋值为“value”的 BitVector 信号	Bits/UInt/SInt
B/U/S”[[size’]base]value”	创建一个赋值为“value”的 BitVector 信号	Bits/UInt/SInt
B/U/S([x bits], element, ···)	创建一个 BitVector 信号，并为各元素赋值（见下表）	Bits/UInt/SInt

可以这样定义元素：

元素语法	描述
x : Int -> y : Boolean/Bool	用 y 设置位 x
x : Range -> y : Boolean/Bool	设置 x 范围内的每个位为 y
x : Range -> y : T	设置 x 范围内的位为 y
x : Range -> y : String	设置 x 范围内的位为 y 字符串格式遵循与 B/U/S”xyz”相同的规则
x : Range -> y : T	设置 x 范围内的位为 y
default -> y : Boolean/Bool	使用 y 值设置所有未连接的位。 此功能只能用于对没有 U/B/S 前缀信号的赋值

您可以定义一个范围值

范围语法	描述	位宽
(x downto y)	[x:y] x >= y	x-y+1
(x to y)	[x:y] x <= y	y-x+1
(x until y)	[x:y[x < y	y-x

```
val myUInt = UInt(8 bits)
myUInt := U(2,8 bits)
myUInt := U(2)
myUInt := U"0000_0101"  // Base per default is binary => 5
myUInt := U"h1A"        // Base could be x (base 16)
                        //               h (base 16)
                        //               d (base 10)
                        //               o (base 8)
                        //               b (base 2)
myUInt := U"8'h1A"
myUInt := 2             // You can use scala Int as literal value
```

(续下页)

(接上页)

```
val myBool := myUInt === U(7 -> true,(6 downto 0) -> false)
val myBool := myUInt === U(myUInt.range -> true)

// For assignment purposes, you can omit the B/U/S, which also alow the use of the␣
↪[default -> ???] feature
myUInt := (default -> true)                          // Assign myUInt with "11111111"
myUInt := (myUInt.range -> true)                     // Assign myUInt with "11111111"
myUInt := (7 -> true,default -> false)               // Assign myUInt with "10000000"
myUInt := ((4 downto 1) -> true,default -> false) // Assign myUInt with "00011110"
```

运算符

运算符	描述	返回类型
~x	按位非	T(w(x) bits)
x & y	按位与	T(max(w(x), w(y) bits)
x | y	按位或	T(max(w(x), w(y) bits)
x ^ y	按位异或	T(max(w(x), w(y) bits)
x(y)	读取位域，y : Int/UInt	Bool
x(hi,lo)	读取位域，hi : Int，lo : Int	T(hi-lo+1 bits)
x(offset,width)	读取位域，offset: UInt，width: Int	T(width bits)
x(y) := z	赋值位，y : Int/UInt	Bool
x(hi,lo) := z	赋值位域，hi : Int，lo : Int	T(hi-lo+1 bits)
x(offset,width) := z	赋值位域，offset: UInt，width: Int	T(width bits)
x.msb	返回最高有效位	Bool
x.lsb	返回最低有效位	Bool
x.range	返回范围（x.high downto 0）	范围
x.high	返回类型 x 的上限	Int
x.xorR	对 x 的所有位进行异或	Bool
x.orR	对 x 的所有位做或运算	Bool
x.andR	对 x 的所有位做与运算	Bool
x.clearAll[()]	清零所有位	T
x.setAll[()]	将所有的位设置为 1	T
x.setAllTo(value : Boolean)	将所有位设置为给定的布尔值（Scala Boolean）	
x.setAllTo(value : Bool)	将所有位设置为给定的布尔值（Spinal Bool）	
x.asBools	转换为 Bool 数组	Vec(Bool(),width(x))

掩码过滤结果比较

有时候，你需要检查一个 `BitVector` 和一个包含空位（不需要进行相等性表达式比较的位）的位掩码之间的相等性。

此操作的示例（注意使用“M”前缀）：

```
val myBits = Bits(8 bits)
val itMatch = myBits === M"00--10--"
```

16.6.4 位

运算符	描述	返回类型
x » y	逻辑右移，y : Int	T(w(x) - y bits)
x » y	逻辑右移，y : UInt	T(w(x) bits)
x « y	逻辑左移，y : Int	T(w(x) + y bits)
x « y	逻辑左移，y : UInt	T(w(x) + max(y) bits)
x.rotateLeft(y)	逻辑循环左移，y : UInt	T(w(x))
x.resize(y)	返回调整位宽后的 x，根据需要在 MSB 处填充零位以加大位宽，也可以截断保留在 LSB 侧的宽度，y : Int	T(y bits)
x.resizeLeft(y)	返回调整位宽的 x，根据需要在 LSB 处填充零位以加大位宽，也可以在 MSB 端截断宽度，y : Int	T(y bits)

16.6.5 UInt、SInt

运算符	描述	返回类型
x + y	加法	T(max(w(x), w(y) bits)
x - y	减法	T(max(w(x), w(y) bits)
x * y	乘法	T(w(x) + w(y) bits)
x > y	大于	Bool
x >= y	大于或等于	Bool
x < y	小于	Bool
x <= y	小于或等于	Bool
x » y	算术右移，y : Int	T(w(x) - y bits)
x » y	算术右移，y : UInt	T(w(x) bits)
x « y	算术左移，y : Int	T(w(x) + y bits)
x « y	算术左移，y : UInt	T(w(x) + max(y) bits)
x.resize(y)	返回算术位宽调整后的 x，y。x, y : Int	T(y bits)

16.6.6 Bool, Bits, UInt, SInt

运算符	描述	返回类型
x.asBits	二进制转换为 Bits	Bits(w(x) bits)
x.asUInt	二进制转换为 UInt	UInt(w(x) bits)
x.asSInt	二进制转换为 SInt	SInt(w(x) bits)
x.asBool	二进制转换为 Bool	Bool(x.lsb)

16.6.7 Vec

声明	描述
Vec(type : Data, size : Int)	创建一个指定类型和位宽的向量
Vec(x,y,..)	创建一个向量，其中索引指向给定元素。 这会构造支持混合宽度的元素

运算符	描述	返回类型
x(y)	读取元素 y, y : Int/UInt	T
x(y) := z	将元素 y 赋值给 z, y : Int/UInt	

```
val myVecOfSInt = Vec(SInt(8 bits),2)
myVecOfSInt(0) := 2
myVecOfSInt(1) := myVecOfSInt(0) + 3

val myVecOfMixedUInt = Vec(UInt(3 bits), UInt(5 bits), UInt(8 bits))

val x,y,z = UInt(8 bits)
val myVecOf_xyz_ref = Vec(x,y,z)
for(element <- myVecOf_xyz_ref) {
  element := 0   // Assign x,y,z with the value 0
}
myVecOf_xyz_ref(1) := 3    // Assign y with the value 3
```

16.6.8 Bundle

线束可用于对数据结构信号总线和接口进行建模。
线束内定义的所有数据属性（Bool、Bits、UInt…）都被视为线束的一部分。

简单示例（RGB/VGA）

以下示例显示了具有某些内部函数的 RGB 线束的定义。

```
case class RGB(channelWidth : Int) extends Bundle {
  val red   = UInt(channelWidth bits)
  val green = UInt(channelWidth bits)
  val blue  = UInt(channelWidth bits)

  def isBlack : Bool = red === 0 && green === 0 && blue === 0
  def isWhite : Bool = {
    val max = U((channelWidth-1 downto 0) -> true)
    return red === max && green === max && blue === max
  }
}
```

然后，您还可以根据需要将一个线束实例放置在另一个线束中（没有深度限制）：

```
case class VGA(channelWidth : Int) extends Bundle {
  val hsync = Bool()
  val vsync = Bool()
  val color = RGB(channelWidth)
}
```

And finally instantiate your Bundles inside the hardware :

```
val vgaIn  = VGA(8)         // Create a RGB instance
val vgaOut = VGA(8)
vgaOut := vgaIn             // Assign the whole bundle
vgaOut.color.green := 0     // Fix the green to zero
val vgaInRgbIsBlack = vgaIn.rgb.isBlack   // Get if the vgaIn rgb is black
```

如果想将你的线束指定为组件的输入或输出，你必须通过以下方式来完成：

```
class MyComponent extends Component {
  val io = Bundle {
    val cmd = in(RGB(8))     // Don't forget the bracket around the bundle.
    val rsp = out(RGB(8))
  }
}
```

接口示例（APB）

如果你想定义一个接口，比如一个 APB 接口，就可以使用线束：

```
class APB(addressWidth: Int,
          dataWidth: Int,
          selWidth : Int,
          useSlaveError : Boolean) extends Bundle {

  val PADDR       = UInt(addressWidth bits)
  val PSEL        = Bits(selWidth bits)
  val PENABLE     = Bool()
  val PREADY      = Bool()
  val PWRITE      = Bool()
  val PWDATA      = Bits(dataWidth bits)
  val PRDATA      = Bits(dataWidth bits)
  val PSLVERROR   = if(useSlaveError) Bool() else null    // This wire is created␣
↪only when useSlaveError is true
}

// Example of usage :
val bus = APB(addressWidth = 8,
              dataWidth = 32,
              selWidth = 4,
              useSlaveError = false)
```

一种好的做法是将所有构造参数分组到一个配置类中。这可以使组件中的参数化变得更加容易，特别是当您必须在多个位置重用相同的配置时。另外，如果您需要添加另一个构造参数，您只需将其添加到配置类中，并且在任何地方都可以实例化该参数：

```
case class APBConfig(addressWidth: Int,
                     dataWidth: Int,
                     selWidth : Int,
                     useSlaveError : Boolean)

class APB(val config: APBConfig) extends Bundle {    // [val] config, make the␣
↪configuration public
  val PADDR       = UInt(config.addressWidth bits)
  val PSEL        = Bits(config.selWidth bits)
  val PENABLE     = Bool()
  val PREADY      = Bool()
  val PWRITE      = Bool()
  val PWDATA      = Bits(config.dataWidth bits)
  val PRDATA      = Bits(config.dataWidth bits)
  val PSLVERROR   = if(config.useSlaveError) Bool() else null
}

// Example of usage
val apbConfig = APBConfig(addressWidth = 8,dataWidth = 32,selWidth = 4,
↪useSlaveError = false)
val busA = APB(apbConfig)
val busB = APB(apbConfig)
```

然后在某些时候，您可能需要使用 APB 总线作为某些组件的主端接口或从端接口。为此，您可以定义一些函数：

```
import spinal.core._

case class APBConfig(addressWidth: Int,
                     dataWidth: Int,
                     selWidth : Int,
                     useSlaveError : Boolean)

class APB(val config: APBConfig) extends Bundle {
  val PADDR      = UInt(config.addressWidth bits)
  val PSEL       = Bits(config.selWidth bits)
  val PENABLE    = Bool()
  val PREADY     = Bool()
  val PWRITE     = Bool()
  val PWDATA     = Bits(config.dataWidth bits)
  val PRDATA     = Bits(config.dataWidth bits)
  val PSLVERROR  = if(config.useSlaveError) Bool() else null

  def asMaster(): this.type = {
    out(PADDR,PSEL,PENABLE,PWRITE,PWDATA)
    in(PREADY,PRDATA)
    if(config.useSlaveError) in(PSLVERROR)
    this
  }

  def asSlave(): this.type = this.asMaster().flip() // Flip reverse all in out␣
↪configuration.
}

// Example of usage
val apbConfig = APBConfig(addressWidth = 8,dataWidth = 32,selWidth = 4,
↪useSlaveError = false)
val io = new Bundle {
  val masterBus = APB(apbConfig).asMaster()
  val slaveBus = APB(apbConfig).asSlave()
}
```

更进一步优化，spine.lib 中集成了一个名为 IMasterSlave 的小型主、从端口定义工具。当线束扩展 IMasterSlave 时，它应该实现/覆盖 asMaster 函数，以便设置主端接口或从端接口中的信号方向：

```
val apbConfig = APBConfig(addressWidth = 8,dataWidth = 32,selWidth = 4,
↪useSlaveError = false)
val io = new Bundle {
  val masterBus = master(apbConfig)
  val slaveBus  = slave(apbConfig)
}
```

实现此 IMasterSlave 的 APB 总线示例：

```
// You need to import spinal.lib._ to use IMasterSlave
import spinal.core._
import spinal.lib._

case class APBConfig(addressWidth: Int,
                     dataWidth: Int,
                     selWidth : Int,
                     useSlaveError : Boolean)

class APB(val config: APBConfig) extends Bundle with IMasterSlave {
  val PADDR      = UInt(addressWidth bits)
```

(续下页)

(接上页)

```
  val PSEL        = Bits(selWidth bits)
  val PENABLE     = Bool()
  val PREADY      = Bool()
  val PWRITE      = Bool()
  val PWDATA      = Bits(dataWidth bits)
  val PRDATA      = Bits(dataWidth bits)
  val PSLVERROR   = if(useSlaveError) Bool() else null   // This wire is created␣
↪only when useSlaveError is true

  override def asMaster() : Unit = {
    out(PADDR,PSEL,PENABLE,PWRITE,PWDATA)
    in(PREADY,PRDATA)
    if(useSlaveError) in(PSLVERROR)
  }
  // The asSlave is by default the flipped version of asMaster.
}
```

16.6.9 Enum

SpinalHDL 支持一些编码的枚举：

编码	位宽	描述
native		使用 VHDL 枚举系统，这是默认编码
二进制顺序	log2Up(stateCount) bit	使用 UInt 按声明顺序存储状态（值从 0 到 n-1）
binary-One-Hot	stateCount	使用位来存储状态。每个位对应一种状态，编码时一次只有一位有效。

定义一个枚举类型：

```
object UartCtrlTxState extends SpinalEnum { // Or␣
↪SpinalEnum(defaultEncoding=encodingOfYourChoice)
  val sIdle, sStart, sData, sParity, sStop = newElement()
}
```

实例化一个信号来存储枚举编码值并为其赋值：

```
val stateNext = UartCtrlTxState() // Or␣
↪UartCtrlTxState(encoding=encodingOfYouChoice)
stateNext := UartCtrlTxState.sIdle

// You can also import the enumeration to have the visibility on its elements
import UartCtrlTxState._
stateNext := sIdle
```

16.6.10 Data (Bool, Bits, UInt, SInt, Enum, Bundle, Vec)

所有硬件类型都扩展了 Data 类，这意味着它们都提供以下运算符：

运算符	描述	返回类型
x === y	等价性判断	Bool
x =/= y	不等价判断运算	Bool
x.getWidth	返回位数	Int
x ## y	连接 Bits，x-> 高位，y-> 低位	Bits(width(x) + width(y) bits)
Cat(x)	连接列表，第一个元素放置在 lsb 上，x : Array[Data]	Bits(sumOfWidth bits)
Mux(cond,x,y)	if cond ? x : y	T(max(w(x), w(y) bits)
x.asBits	类型转换到 Bits	Bits(width(x) bits)
x.assignFromBits(bits)	从 Bits 获取信号来赋值	
x.assignFromBits(bits,hi,lo)	赋值位域，hi : Int，lo : Int	T(hi-lo+1 bits)
x.assignFromBits(bits,offset,width)	赋值位域，offset: UInt，width: Int	T(width bits)
x.getZero	获取等效类型且赋值 0	T

16.6.11 使用字面量声明信号

字面量通常用作常量值。但您也可以使用它们一次完成两件事：

- 定义一条分配有常量值的连线
- 设置推断类型：UInt(4 bits)
- 当条件 *cond =/= True* 满足的时钟周期，将信号重新设置为常量
- 由于最后一条语句获胜规则，满足 *cond === True* 条件的时钟周期将导致信号具有“red”值。

实例：

```
val cond = in Bool()
val red = in UInt(4 bits)
...
val valid = False           // Bool wire which is by default assigned with False
val value = U"0100"         // UInt wire of 4 bits which is by default assigned␣
↪with 4
when(cond) {
  valid := True
  value := red
}
```

16.6.12 用连续赋值字面量作来声明信号

您还可以在表达式中使用它们来同时达成三件事：

- 定义一个信号（一条线）
- 在硬件逻辑实现中，保持等价性比较运算结果的常数值和另一信号的结果
- 设置推断类型：Bool，因为使用 === 相等运算符，其结果为 Bool 类型

这里是一个例子：

```
val done = Bool(False)
val blue = in UInt(4 bits)
...
val value = blue === U"0001"  // inferred type is Bool due to use of === operator
```

(续下页)

(接上页)

```
when(value) {
  done := True
}
```