An alternative to standard HDL

SpinalHDL
An alternative to standard HDL
Summary

- Language introduction / flow
- Simple examples
- Advanced examples
- Meta-hardware description examples
Language introduction

- Open source, started in December 2014
- Focus on RTL description
- Thought to be interoperable with existing tools
  - It generates VHDL/Verilog files
  - It can integrate VHDL/Verilog IP as blackbox
- Abstraction level:
  - You can design things similarly to VHDL/Verilog
  - If you want to, you can use many abstraction utilities and also define new ones
1. Describe your RTL
2. Generate the VHDL/Verilog
3. Simulate and synthesize
Some points about Spinal

- There is no logic overhead in the generated code. (I swear !)
- Spinal HDL is a RTL language. But the generated VHDL/Verilog is simulatable with all standards EDA tools.
- The component hierarchy and all names are preserved during the VHDL/Verilog generation.
Keywords

- Types:
  - Bool / Bits / UInt / SInt / SpinalEnum
  - Bundle / Vec
- Hierarchy:
  - Component
  - Area
- Misc:
  - Reg / RegInit
  - in / out / master / slave
Simple examples
A simple component

class MyComponent extends Component {
    val io = new Bundle {
        val a = in Bool
        val result = out Bool
    }

    io.result := io.a

}
Combinatorial logic

class MyComponent extends Component {
    val io = new Bundle {
        val a = in Bool
        val b = in Bool
        val c = in Bool
        val result = out Bool
    }

    io.result := (io.a & io.b) | (!io.c)
}
Signals

```scala
class MyComponent extends Component {
  val io = new Bundle {
    val a = in Bool
    val b = in Bool
    val c = in Bool
    val result = out Bool
  }
  val a_and_b = Bool
  a_and_b := io.a & io.b
  val not_c = ! io.c
  io.result := a_and_b | not_c
}
```
class MyComponent extends Component {
    val io = new Bundle {
        val a = in Bool
        val b = in Bool
        val c = in Bool
        val result = out Bool
    }
    val a_and_b = io.a & io.b
    val not_c = !io.c
    io.result := a_and_b | not_c
}

entity MyComponent is
    port(
        io_a : in std_logic;
        io_b : in std_logic;
        io_c : in std_logic;
        io_result : out std_logic
    );
end MyComponent;

architecture arch of MyComponent is
    signal a_and_b : std_logic;
    signal not_c : std_logic;
    begin
        io_result <= (a_and_b or not_c);
        a_and_b <= (io_a and io_b);
        not_c <= (not io_c);
    end arch;
class MyComponent extends Component {
    val io = new Bundle {
        val a = in Bool
    }
    val reg1 = Reg(Bool)
    val reg2 = Reg(Bool) init (False)
    val reg3 = Reg(Bool)
    reg3 := ! io.a
}
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
}
class TopLevel extends Component {
    // ...
    val logicArea = new Area {
        val flag = Bool
    }

    val fsmArea = new Area {
        when(logicArea.flag) {
            // ...
        }
    }
}

Component instance

```scala
class SubComponent extends Component{
  val io = new Bundle {
    val input = in Bool
    val result = out Bool
  }
  ...
}

class TopLevel extends Component {
  val io = new Bundle {
    val a = in Bool
    val b = in Bool
    val result = out Bool
  }

  val sub = new SubComponent

  sub.io.input := io.a
  io.result := sub.io.result | io.b
}
```
class MyComponent extends Component {
  val io = new Bundle {
    val conds = in Vec(Bool,2)
    val result = out UInt(4 bits)
  }

  when(io.conds(0)){
    io.result := 2
    when(io.conds(1)){
      io.result := 1
    }
  } otherwise {
    io.result := 0
  }
}

conds_0  conds_1  result
object MyEnum extends SpinalEnum {
    val state0, state1 = newElement()
}

class MyComponent extends Component {
    val state = Reg(MyEnum) init(MyEnum.state0)

    switch(state) {
        is(MyEnum.state0) {

        }
        is(MyEnum.state1) {

        }
        default{

        }
    }
}
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)
    }

    var c = False
    for (i <- 0 until size) {
        val x = io.a(i)
        val y = io.b(i)

        io.result(i) := x ^ y ^ c
        c |= (x & y) | (x & c) | (y & c);
    }
}
Latch/Loop

```scala
val a = Bool
val result = Bool
result := a | result // Loop detected

val result = Bool
when(cond){  // result is not assigned in all cases => Latch detected
  result := True
}
```
class MyTopLevel extends Component {
    val io = new Bundle {
        val coreClk = in Bool
        val coreReset = in Bool
    }

    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 bit))
        //...
    }
}
Memory

//Memory of 1024 Bool
val syncRam = Mem(Boolean, 1024)
val asyncRam = Mem(Boolean, 1024)

//Write them
syncRam(5) := True
asyncRam(5) := True

//Read them
val syncRam = mem.readSync(6)
val asyncRam = mem.readAsync(4)
// Input RGB color
val r, g, b = UInt(8 bits)

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

// Calculate the gray level
val gray = coefMul(r, 0.3f) +
            coefMul(g, 0.4f) +
            coefMul(b, 0.3f)
case class Color(channelWidth: Int) extends Bundle {
  val r, g, 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

    return result
  }
}
class ColorSumming(sourceCount : Int, channelWidth : Int) extends Component {
  val io = new Bundle {
    val sources = in Vec(Color(channelWidth), sourceCount)
    val result = out(Color(channelWidth))
  }

  var sum = io.sources(0)
  for (i <- 1 until sourceCount) {
    sum += sum + io.sources(i)
  }

  io.result := sum
}
Advanced examples

Fifo

push  pop

Arbiter

sources(0)  sink

sources(1)

sources(n)

this

valid  valid

payload  payload

ready  ready

outputStage

addresses(0)  key  hits(0)

addresses(1)  key  hits(1)

addresses(2)  key  hits(2)

addresses(N)  key  hits(N)

hit
Functional programming

```scala
val addresses = Vec(UInt(8 bits), 4)
val key = UInt(8 bits)
val hits = addresses.map(address => address === key)
val hit = hits.reduce((a, b) => a || b)
```
Basic abstractions

```scala
val timeout = Timeout(1000)
when(timeout){ // implicit conversion to Bool
  timeout.clear() // Clear the flag and the internal counter
}

// Create a counter of 10 states (0 to 9)
val counter = Counter(10)
counter.clear() // When called it reset the counter. It's not a flag
counter.increment() // When called it increment the counter. It's not a flag
counter.value // Current value
counter.valueNext // Next value
counter.willOverflow // Flag that indicate if the counter overflow this cycle
when(counter == 5){ ...}
```
Flow, Stream

```scala
case class Flow[T <: Data](payloadType: T) extends Bundle {
    val valid = Bool
    val payload = cloneOf(payloadType)
}

case class Stream[T <: Data](payloadType: T) extends Bundle {
    val valid = Bool
    val ready = Bool
    val payload = cloneOf(payloadType)
}

val myStreamOfRGB = Stream(RGB(8,8,8))
```
Stream components

class Fifo[T <: Data](payloadType: T, depth: Int) extends Component {
    val io = new Bundle {
      val push = slave Stream (payloadType)
      val pop  = master Stream (payloadType)
    }
    //...
}

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))
    }
    //...
}
Stream functions

case class Stream[T <: Data](payloadType: T) extends Bundle {
    // ...
    def connectFrom(that: Stream[T]) = {
        // some connections between this and that
    }
    def stage(): Stream[T] = {
        val outputStage = Stream(payloadType)
        val validReg = RegInit(False)
        val payloadReg = Reg(payloadType)
        // some logic
        return outputStage
    }
    def << (that: Stream[T]) = this.connectFrom(that)
    def <-> (that: Stream[T]) = this << that.stage()
}

val myStreamA, myStreamB = Stream(UInt(8 bits))
myStreamA <-> myStreamB
Scala is here to help you

```scala
class SinusGenerator(resolutionWidth : Int,sampleCount : Int) extends Component {
  val io = new Bundle {
    val sin = out SInt (resolutionWidth bits)
  }

  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)
  val phase = CounterFreeRun(sampleCount)
  io.sin := rom.readSync(phase)
}
```
Design introspection

val \( a = \text{UInt}(8 \text{ bits}) \)
val \( b = \text{UInt}(8 \text{ bits}) \)

val \( a_{\text{CalcResult}} = \text{complicatedLogic}(a) \)

val \( a_{\text{Latency}} = \text{LatencyAnalysis}(a,a_{\text{CalcResult}}) \)
val \( b_{\text{Delayed}} = \text{Delay}(b, \text{cycleCount} = a_{\text{Latency}}) \)

val \( \text{result} = a_{\text{CalcResult}} + b_{\text{Delayed}} \)
Meta-hardware description examples

diagram with states and logical components
They could be defined with regular syntax (Enum, Switch)
You can also use a much more friendly syntax, fully integrated, with following features:
- onEntry / onExit / whenIsActive / whenIsNext blocs
- State with inner FSM
- State with multiple inner FSM (parallel execution)
- Delay state
- You can extends the syntax by defining new state types
val io = new Bundle{
    val result = out Bool
}

val fsm = new StateMachine{
    io.result := False
    val counter = Reg(UInt(8 bits)) init (0)

    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)
            }
        }
        onExit(io.result := True)
    }

    val stateC : State = new State{
        whenIsActive (goto(stateA))
    }
}
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))
}
Bus Slave Factory

Imagine you want to control an UART controller from a bus (for example AMBA-APB), you will have to implement a “bridge logic”.

![Diagram showing UART controller and bridge logic connections]
Bus Slave Factory
• Let’s detail the situation
Bus Slave Factory

• IO / instances / direct connections

```scala
class Apb3UartCtrl(rxFifoDepth : Int) extends Component {
  val io = new Bundle {
    val bus = slave(Apb3(addressWidth = 4, dataWidth = 32))
    val uart = master(Uart())
  }

  // Instantiate a simple UART controller
  val uartCtrl = new UartCtrl()

  // Connect its UART bus
  io.uart <> uartCtrl.io.uart

  // Here we have to implement the “bridge logic”.
  // All the code of next slides should be inserted there
  // ...
}
```
Bus Slave Factory

- Apb3SlaveFactory is able to create some “bridge logic” by using an abstract way. Let’s use it!

```scala
val busCtrl = Apb3SlaveFactory(io.bus)
```
Bus Slave Factory

Make the clockDivider readable/writable by the 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)
Bus Slave Factory

- Make the frame config readable/writable by the bus

```c
// Do the same thing than previously but for uartCtrl.io.config.frame at the address 4
busCtrl.driveAndRead(uartCtrl.io.config.frame,address = 4)
```
Bus Slave Factory

- Allow the bus to emit UART write requests

```scala
// Ask the busCtrl to create a writable Flow[Bits] (valid/payload) at the address 8.
// Then convert it into a stream, add a register register stage and connect it to the uartCtrl.io.write
val writeFlow = busCtrl.createAndDriveFlow(Bits(8 bits), address = 8)
writeFlow.toStream.stage >> uartCtrl.io.write
```
Bus Slave Factory

- Allow the bus to get the occupancy of the write buffer

// 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

```c
busCtrl.read(uartCtrl.io.write.valid, address = 8)
```
Bus Slave Factory

- Allow the bus to read received UART frames through a FIFO

```scala
// Take uartCtrl.io.read, convert it into a Stream, then connect it to the input of a FIFO
// Then make the output of the FIFO readable at the address 12 by using a non blocking protocol
// (bit 31 => data valid, bits 7 downto 0 => data)
val readStream = uartCtrl.io.read.toStream.queue(rxFifoDepth)
busCtrl.readStreamNonBlocking(readStream,address = 12,validBitOffset = 31,payloadBitOffset = 0)
```
About FSM and Apb3SlaveFactory

Both aren’t part of Spinal core but are implemented on the top of it in the Spinal lib. Which mean these tools were created without any special interaction or special knowledge of the Spinal compiler. They are only a mix of Scala OOP/FP with some Spinal basic syntax to generate the right hardware!
About Scala

- Free Scala IDE (eclipse, intellij)
  - Highlight syntax error
  - Renaming flexibility
  - Intelligent auto completion
  - Code’s structure overview
  - Navigation tools
- Allow you to extend the language
- Provide many libraries
Spinal work perfectly on FPGA

- RISCV CPU, 5 stages, 1.15 DMIPS/Mhz
  - MUL/DIV
  - Instruction/Data cache
  - Interrupts
  - JTAG debugging
- AXI/APB interconnect
- Avalon/APB UART
- Avalon/AXI VGA
- Pipelined and multi-core fractal accelerator
About Spinal project

• Completely open source :
  • https://github.com/SpinalHDL/SpinalHDL

• Online documentation :
  • https://spinalhdl.github.io/SpinalDoc/

• Ready to use base project :
  • https://github.com/SpinalHDL/SpinalBaseProject

• Communication channels :
  • spinalhdl@gmail.com
  • https://gitter.im/SpinalHDL/SpinalHDL
  • https://github.com/SpinalHDL/SpinalHDL/issues
End / reserve slides
Meta-hardware description
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Access</th>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clockDivider</td>
<td>UInt</td>
<td>RW</td>
<td>0</td>
<td>Set the UartCtrl clock divider</td>
</tr>
<tr>
<td>frame</td>
<td>UartCtrlFrameConfig</td>
<td>RW</td>
<td>4</td>
<td>Set the dataLength, the parity and the stop bit configuration</td>
</tr>
<tr>
<td>writeCmd</td>
<td>Bits</td>
<td>W</td>
<td>8</td>
<td>Send a write command to the UartCtrl</td>
</tr>
<tr>
<td>writeBusy</td>
<td>Bool</td>
<td>R</td>
<td>8</td>
<td>Bit 0 =&gt; zero when a new writeCmd could be sent</td>
</tr>
<tr>
<td>read</td>
<td>Bool / Bits</td>
<td>R</td>
<td>12</td>
<td>Bit 7 downto 0 =&gt; fifo pop payload</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Bit 31 =&gt; fifo pop valid</td>
</tr>
</tbody>
</table>
class AvalonUartCtrl(uartCtrlConfig : UartCtrlGenerics, rxFifoDepth : Int) extends Component{
  val io = new Bundle{
    val bus  = slave(AvalonMM(...))
    val uart = master(Uart())
  }

  val uartCtrl = new UartCtrl(uartCtrlConfig)
  io.uart <> uartCtrl.io.uart

  val busCtrl = AvalonMMSlaveFactory(io.bus)

  //Make clockDivider register
  busCtrl.driveAndRead(uartCtrl.io.config.clockDivider, address = 0)

  //Make frame register
  busCtrl.driveAndRead(uartCtrl.io.config.frame, address = 4)

  //Make writeCmd register
  val writeFlow = busCtrl.createAndDriveFlow(Bits(uartCtrlConfig.dataWidthMax bits), address = 8)
  writeFlow.toStream.stage() >> uartCtrl.io.write

  //Make writeBusy register
  busCtrl.read(uartCtrl.io.write.valid, address = 8)

  //Make read register
  busCtrl.readStreamNonBlocking(uartCtrl.io.read.toStream.queue(rxFifoDepth),
                                address = 12, validBitOffset = 31, payloadBitOffset = 0)
}