

# Spinal HDL

An alternative to standard HDL

# Summary

- Why a new language
- Language introduction / dissection
- Examples (a lot)

# Why a new language

- Because of current HDL
  - Verbosity, endless wiring, copy past
  - Wire level, can't define abstractions
  - Broken features
    - Can't parameterize records
    - Can't define record's elements directions individually
    - SystemVerilog interface parameterized datatype
  - They were initially designed for simulation
  - Heavy legacy

# Language introduction

- Open Source , started in december 2014
- Is designed for RTL
- Compatibility is assured
  - It generate a VHDL file
  - It can integrate VHDL IP as blackbox
- Abstraction level :
  - Start at the same level than VHDL
  - Finish between VHDL and HLS
  - The user can create new levels

# Language dissection

- SpinalHDL is a «Scala internal DSL»
  - You can use all Scala syntax / library
  - Scala IDE are mature and free
  - Object oriented and functional paradigms
- How it work
  - Use the library => it build internal netlist => VHDL
- 2 layers
  - Core (low level RTL)
  - Lib (high level RTL, based on Code layer)

# A simple component

```
class MyComponent extends Component {  
  val io = new Bundle {  
    val a = in Bool  
    val output = out Bool  
  }  
  
  io.output := io.a  
}
```

# Combinatorial, Latch/Loop

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

```
  io.output := (io.a & io.b) | (!io.c)  
}
```

```
class MyComponent extends Component {  
  //...  
  io.output := io.a | io.output //Latch/Loop detected, not allowed  
}
```

# Signals

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



# Generated VHDL

```
class MyComponent extends Component {  
    val io = new Bundle {  
        val a = in Bool  
        val b = in Bool  
        val c = in Bool  
        val output = out Bool  
    }  
    val a_and_b = io.a & io.b  
    val not_c = !io.c  
    io.output := 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_output : out std_logic  
    );  
end MyComponent;
```

```
architecture arch of MyComponent is  
    signal a_and_b : std_logic;  
    signal not_c : std_logic;  
begin  
    io_output <= (a_and_b or not_c);  
    a_and_b <= (io_a and io_b);  
    not_c <= (not io_c);  
end arch;
```

# Registers

```
class MyComponent extends Component {  
  val io = new Bundle {  
    val a = in Bool  
  }  
  
  val reg1 = Reg(Bool)  
  val reg2 = Reg(Bool) init(False)  
  val reg3 = RegInit(False)  
  val reg4 = RegNext(io.a)  
}
```

# Memory

```
//Memory of 1024 Bool  
val mem = Mem(Bool, 1024)  
  
//Write it  
mem(5) := True  
  
//Read it  
val read0 = mem.readAsync(4)  
val read1 = mem.readSync(6)
```

# ClockDomains, Area

```
class MyTopLevel extends Component {
  val io = new Bundle {
    val coreClk = in Bool
    val coreReset = in Bool
    val peripheralClk = in Bool
    val peripheralReset = in Bool
  }
  val coreClockDomain = ClockDomain(io.coreClk, io.coreReset)
  val peripheralClockDomain = ClockDomain(io.peripheralClk, io.peripheralReset)

  val coreArea = new ClockingArea(coreClockDomain) {
    val myCoreClockedRegister = Reg(UInt(4 bit))
  }

  val peripheralArea = new ClockingArea(peripheralClockDomain) {
    val myPeripheralClockedRegister = Reg(UInt(4 bit))
    myPeripheralClockedRegister := coreArea.myCoreClockedRegister //Not allowed
    myPeripheralClockedRegister := BufferCC(coreArea.myCoreClockedRegister)
  }
}
```

# Component instance

```
class MySubComponent extends Component {  
  val io = new Bundle {  
    val subIn = in Bool  
    val subOut = out Bool  
  }  
  ...  
}
```

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

```
val compInstance = new MySubComponent
```

```
compInstance.io.subIn := io.a
```

```
io.output := compInstance.io.subOut | io.b
```

```
}
```

# UInt, Vec, When

```
class MyComponent extends Component {  
  val io = new Bundle {  
    val conds = in Vec(2, Bool)  
    val output = out UInt(4 bit)  
  }  
  
  when(io.conds(0)) {  
    io.output := 2  
    when(io.conds(1)) {  
      io.output := 1  
    }  
  } otherwise {  
    io.output := 0  
  }  
}
```

# Enum, Area, switch

```
object MyEnum extends SpinalEnum {  
  val state0, state1, anotherState = Value  
}  
  
abstract class MyComponent extends Component {  
  val logicOfA = new Area {  
    val flag = Bool  
    val logic = Bool  
  }  
  val fsm = new Area {  
    import MyEnum._  
    val state = Reg(MyEnum()) init (state0)  
    switch(state) {  
      is(state0) {  
        when(logicOfA.flag) {  
          state := state1  
        }  
      }  
      default {  
      }  
    }  
  }  
}
```

# For, Variable, Generics

```
class CarryAdder(size: Int) extends Component {  
  val io = new Bundle {  
    val a = in UInt (size bit)  
    val b = in UInt (size bit)  
    val result = out UInt (size bit)  
  }  
  
  var c = False  
  for (i <- 0 until size) {  
    val a = io.a(i)  
    val b = io.b(i)  
  
    io.result(i) := a ^ b ^ c  
    c = (a & b) | (a & c) | (b & c);  
  }  
}
```



# Bundle, Generics, Vec, Packing

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

class MyColorSelector(sourceCount : Int, channelWidth: Int) extends Component {
  val io = new Bundle {
    val sel = in UInt(log2Up(sourceCount) bit)
    val sources = in Vec(sourceCount, Color(channelWidth))
    val result = out Bits (3*channelWidth bit)
  }

  val selectedSource = io.sources(io.sel)
  io.result := toBits(selectedSource)
}
```

# Function, User utils (1)

```
case class Color(channelWidth: Int) extends Bundle {  
  val r = UInt(channelWidth bit)  
  val g = UInt(channelWidth bit)  
  val b = UInt(channelWidth bit)  
  
  def +(that: Color): Color = {  
    assert(that.channelWidth == this.channelWidth)  
  
    val result = cloneOf(this)  
    result.r := channelAdd(this.r, that.r)  
    result.g := channelAdd(this.g, that.g)  
    result.b := channelAdd(this.b, that.b)  
  
    def channelAdd(left: UInt, right: UInt): UInt = {  
      val (value, carry) = adderAndCarry(right, left)  
      return Mux(carry, left.maxValue, value)  
    }  
  
    return result  
  }  
}
```

# Function, User utils (2)

```
class MyColorSumming(sourceCount: Int, channelWidth: Int) extends Component {  
  val io = new Bundle {  
    val sources = in Vec(sourceCount, Color(channelWidth))  
    val result = out(Color(channelWidth))  
  }  
  
  var sum = io.sources(0)  
  for (i <- 1 until sourceCount) {  
    sum = sum + io.sources(i)  
  }  
  io.result := sum  
  
  // But you can do all this stuff by this way, balanced is bonus :  
  // io.result := io.sources.reduceBalancedSpinal(_ + _)  
}
```

# Basic abstractions from Lib

```
//Create a timeout, he become asserted if the function clear
//was not called in the last 1000 cycles
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.value      //current value
counter.valueNext  //Next value
counter.reset      //When called it reset the counter. It's not a flag
counter.inc        //When called it increment the counter. It's not a flag
counter.overflow   //Flag that indicate if the counter overflow this cycle
when(counter == 5){ } //counter is implicitly its value
```

# Flow, Handshake, Fragment

```
case class Flow[T <: Data](dataType: T) extends Bundle {  
  val valid = Bool  
  val data: T = cloneOf(dataType)  
  //..  
}
```

```
case class Handshake[T <: Data](dataType: T) extends Bundle {  
  val valid = Bool  
  val ready = Bool  
  val data: T = cloneOf(dataType)  
  //..  
}
```

```
case class Fragment[T <: Data](dataType: T) extends Bundle {  
  val last = Bool  
  val data: T = cloneOf(dataType)  
  //..  
}
```

# Handshake examples

```
class HandshakeFifo[T <: Data](dataType: T, depth: Int) extends Component{
  val io = new Bundle {
    val pushPort = slave Handshake (dataType)
    val popPort = master Handshake (dataType)
    val occupancy = out UInt (log2Up(depth + 1) bit)
  } //...
}

class HandshakeArbiter[T <: Data](dataType: T, portCount: Int) extends Component{
  val io = new Bundle {
    val inputs = Vec(portCount, slave Handshake (dataType))
    val output = master Handshake (dataType)
    val chosen = out UInt (log2Up(portCount) bit)
  } //...
}

class HandshakeFork[T <: Data](dataType: T, portCount: Int) extends Component{
  val io = new Bundle {
    val input = slave Handshake (dataType)
    val output = Vec(portCount, master Handshake (dataType))
  } //...
}
```

# Handshake functions

```
val cond = Bool
val inPort = Handshake(Bits(32 bit))
val outPort = Handshake(Bits(32 bit))

outPort << inPort
outPort <-< inPort
outPort </< inPort
outPort <-/< inPort
val haltedPort = inPort.haltWhen(cond)
val filteredPort = inPort.throwWhen(inPort.data === 0)
val outPortWithMsb = inPort.translateWith(inPort.data.msb)

val mem = Mem(Bool, 1024)
val memReadCmd = Handshake(UInt(10 bit))
val memReadPort = mem.handshakeReadSync(memReadCmd, memReadCmd.data)
memReadPort.valid //arbitration
memReadPort.ready //arbitration
memReadPort.data.value //Readed value
memReadPort.data.linked //Linked value (memReadCmd.data)
```

# Flow of Fragment example

```
case class LogicAnalyserConfig() extends Bundle{
  val trigger = new Bundle{
    val delay = UInt(32 bit)
    //...
  }
  val logger = new Bundle{
    val samplesLeftAfterTrigger = UInt(8 bit)
    //...
  }
}

class LogicAnalyser extends Component {
  val io = new Bundle {
    val cfgPort = slave Flow Fragment(Bits(8 bit))
  }
  val waitTrigger = io.cfgPort filterHeader (0x01) toRegOf (Bool) init (False)
  val userTrigger = io.cfgPort pulseOn (0x02)
  val configs      = io.cfgPort filterHeader (0x0F) toRegOf (LogicAnalyserConfig())
}
```



# Generator, Logic Analyser

```
val logicAnalyser = LogicAnalyserBuilder()  
    .setSampleCount(256)  
    .exTrigger(somewhere.inThe.hierarchy.trigger)  
    .probe(somewhere.inThe.hierarchy.signalA)  
    .probe(somewhere.inThe.hierarchy.signalB)  
    .probe(somewhere.signalC)  
    .build
```

```
val uartCtrl = new UartCtrl()  
uartCtrl.read >> logicAnalyser.io.slavePort  
uartCtrl.write << logicAnalyser.io.masterPort
```

# Netlist analyser / Latency analysis

```
class MyComponentWithLatencyAssert extends Component {  
  val io = new Bundle {  
    val slavePort = slave Handshake (UInt(8 bit))  
    val masterPort = master Handshake (UInt(8 bit))  
  }  
  
  //These 3 line are equivalent to io.slavePort.queue(16) >/-> io.masterPort  
  val fifo = new HandshakeFifo((UInt(8 bit)),16)  
  fifo.io.push << io.slavePort  
  fifo.io.pop >/-> io.masterPort  
  
  assert(3 == latencyAnalysis(io.slavePort.data,io.masterPort.data))  
  assert(2 == latencyAnalysis(io.masterPort.ready,io.slavePort.ready))  
}
```

