



# SpinalHDL

An alternative to standard HDL

# Summary

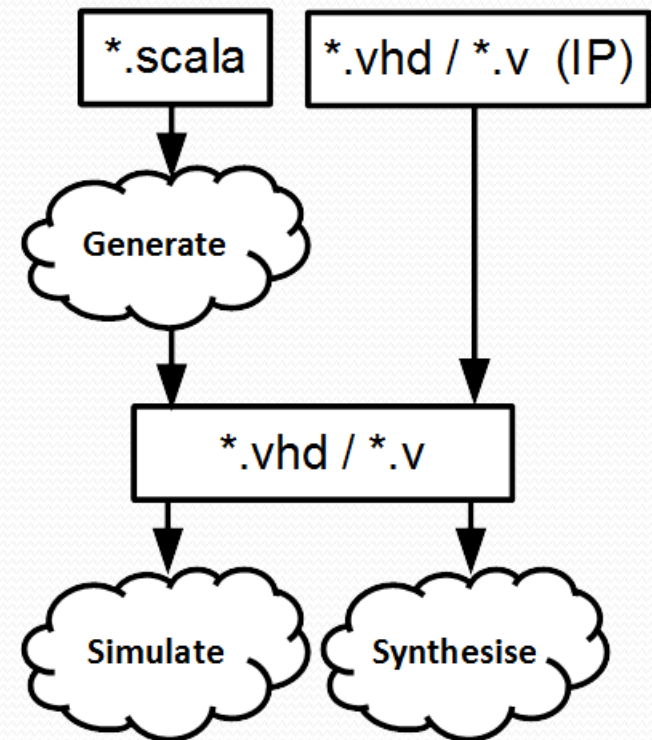
- Language introduction / flow
- Simple examples
- Advanced examples
- Meta-hardware description examples

# Language introduction

- Open source , started in december 2014
- Focus on RTL description
- Thinked to be interoperable with existing tools
  - It generate VHDL/Verilog files
  - It can integrate VHDL/Verilog IP as blackbox
- Abstraction level :
  - You can design things similary to VHDL/Verilog
  - If you want to, you can use many abstraction utils and also define new ones

# Language flow

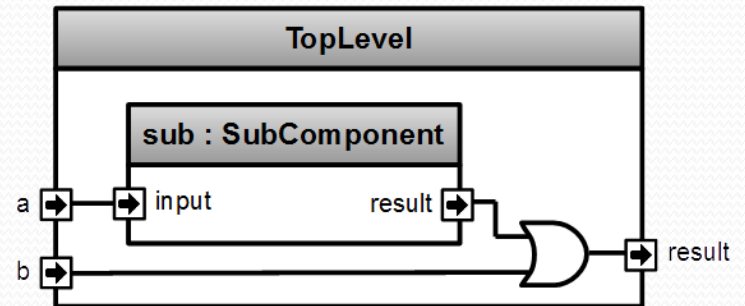
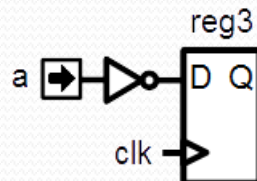
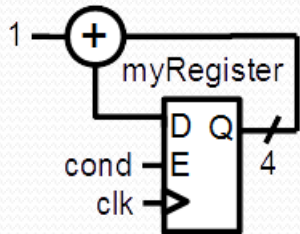
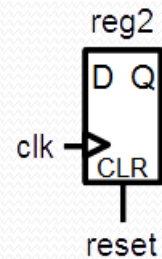
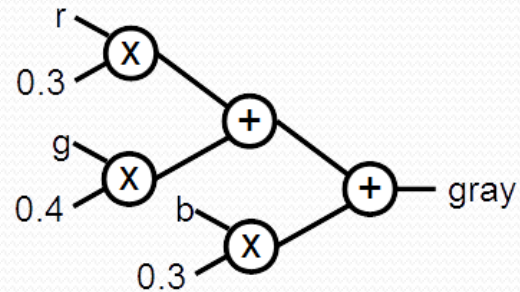
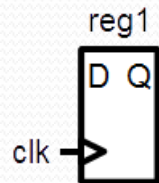
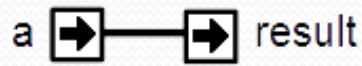
1. Describe your RTL
2. Generate the VHDL/Verilog
3. Simulate and synthesize



# 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.


# 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
```



```
    a → → result  
}
```

# 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  
  }  
}
```

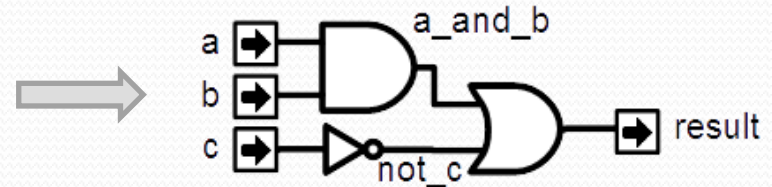
```
io.result := (io.a & io.b) | (!io.c)  
}
```





# Signals

```
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  
}
```



# Generated VHDL

```
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  
}
```

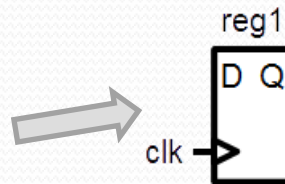


```
entity MyComponent is  
  port(  
    io_a : in std_logic;  
    io_b : in std_logic;  
    io_c : in std_logic;  
    io_result : out std_logic  
  );  
end MyComponent;  
  
architecture arch of MyComponent is  
  signal a_and_b : std_logic;  
  signal not_c : std_logic;  
begin  
  io_result 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

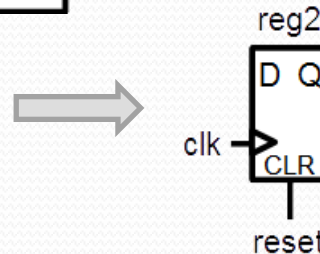
# Registers

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

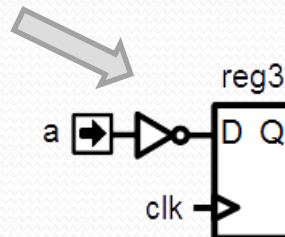
```
val reg1 = Reg(Bool)
```



```
val reg2 = Reg(Bool) init (False)
```



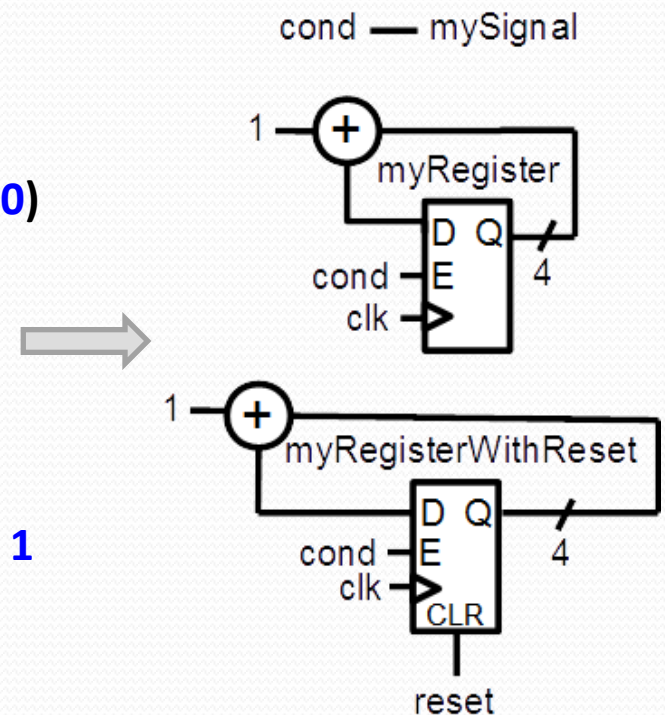
```
val reg3 = Reg(Bool)  
reg3 := !io.a  
}
```



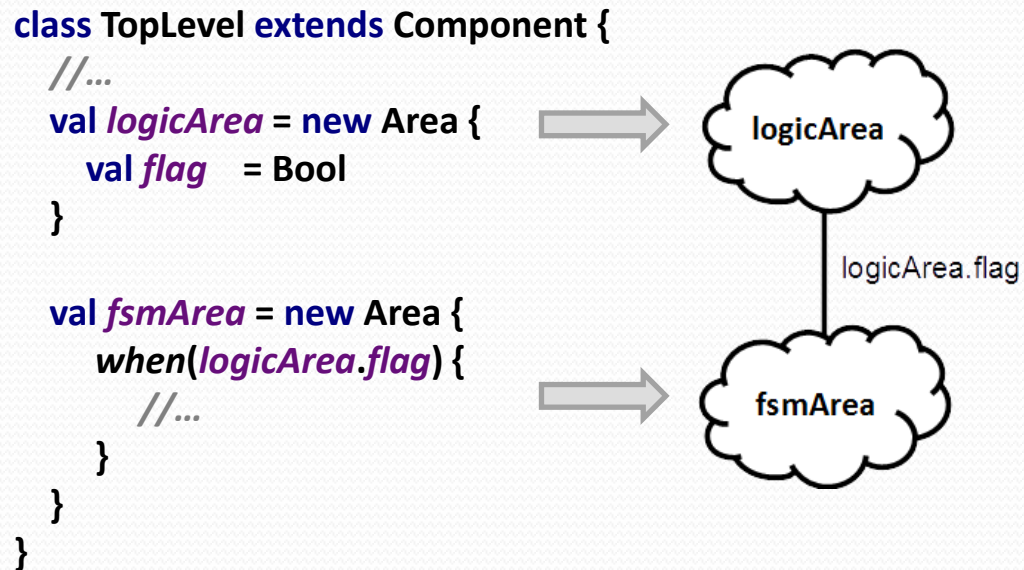
# No more Process/Always blocks

```
val cond                = Bool
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
}
```



# Component internal organisation



# Component instance

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

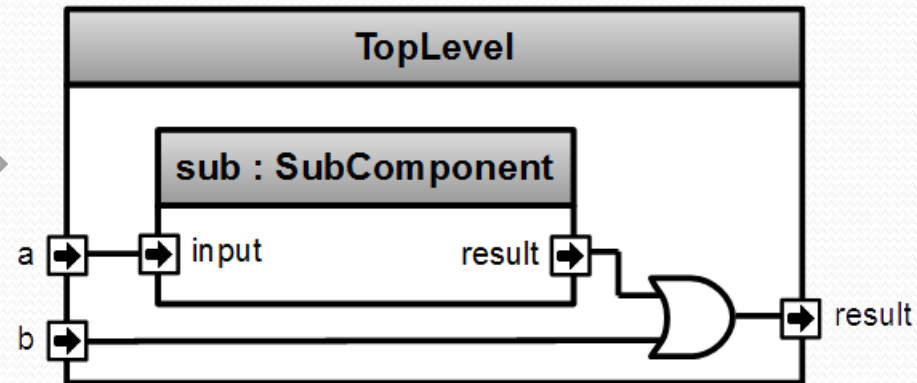
```
class TopLevel extends Component {  
  val io = new Bundle {  
    val a = in Bool  
    val b = in Bool  
    val result = out Bool  
  }  
}
```

```
val sub = new SubComponent
```

```
sub.io.input := io.a
```

```
io.result := sub.io.result | io.b
```

```
}
```



# UInt, Vec, When

```
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
  }
}
```



# Enum, Switch

```
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{  
  
    }  
  }  
}
```

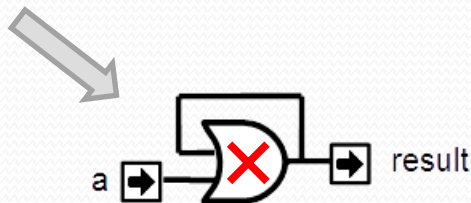


# For, Variable, Generics

```
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

```
class MyComponent extends Component {  
  val io = new Bundle {  
    val a      = in  Bool  
    val result = out Bool  
  }  
  //Latch/Loop detected => Error message will come from Spinal  
  io.result := io.a | io.result  
}
```

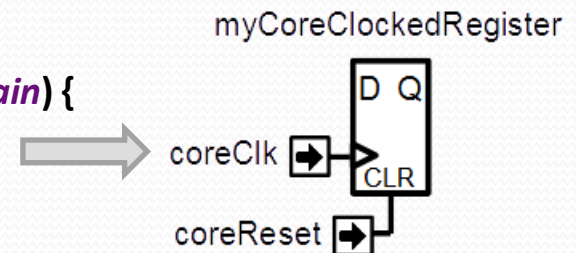


# ClockDomains

```
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(Bool, 1024)

**val asyncRam** = Mem(Bool, 1024)

*//Write them*

**syncRam(5)** := True

**asyncRam(5)** := True

*//Read them*

**val syncRam** = mem.readSync(6)

**val asyncRam** = mem.readAsync(4)

# Function

*// Input RGB color*

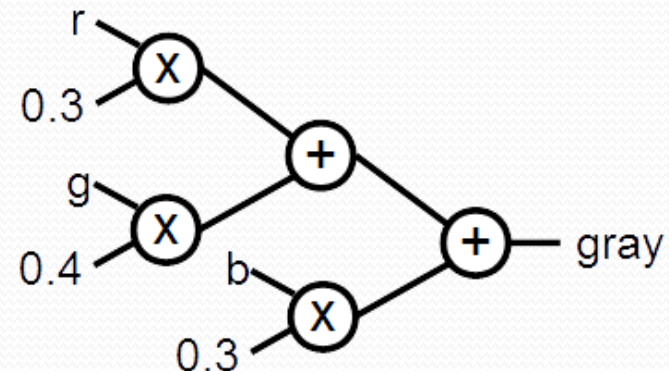
**val** *r,g,b* = UInt(**8** bits)

*// 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**)



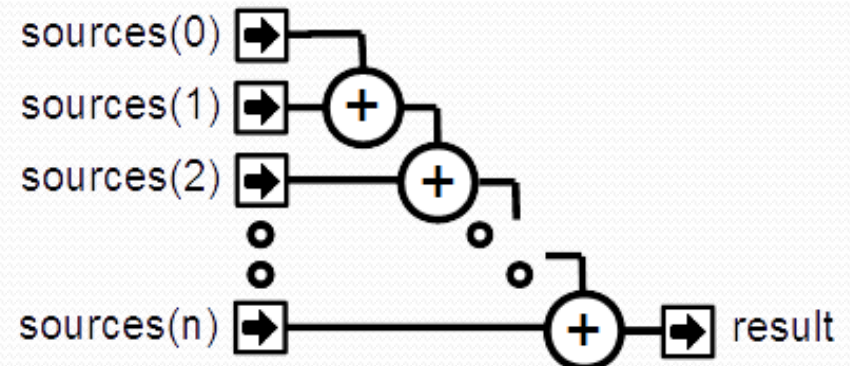
# Function, User utils (1)

```
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  
  
    return result  
  }  
}
```

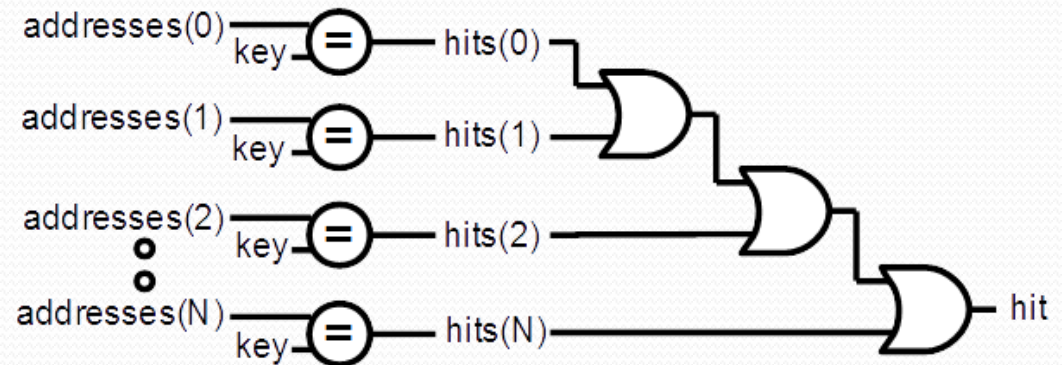
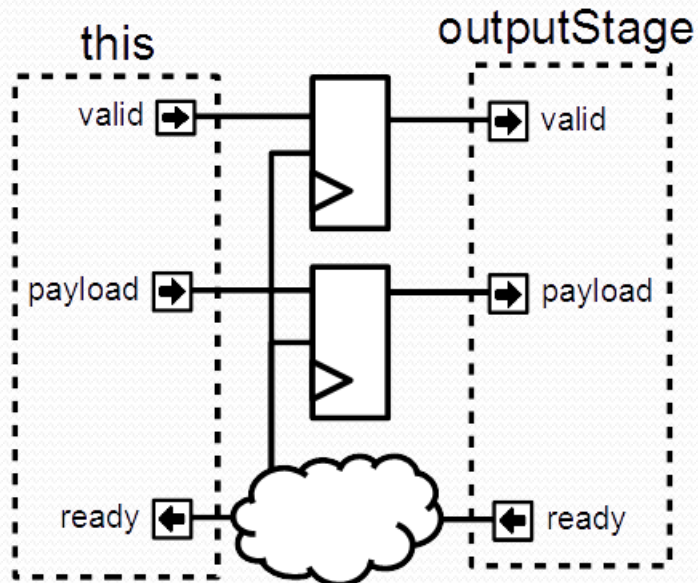
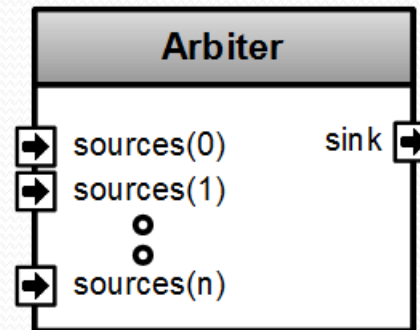
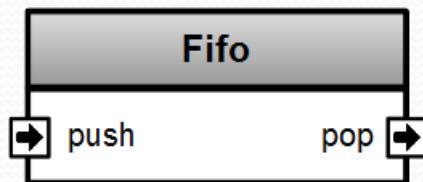
# Function, User utils (2)

```
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  
}
```



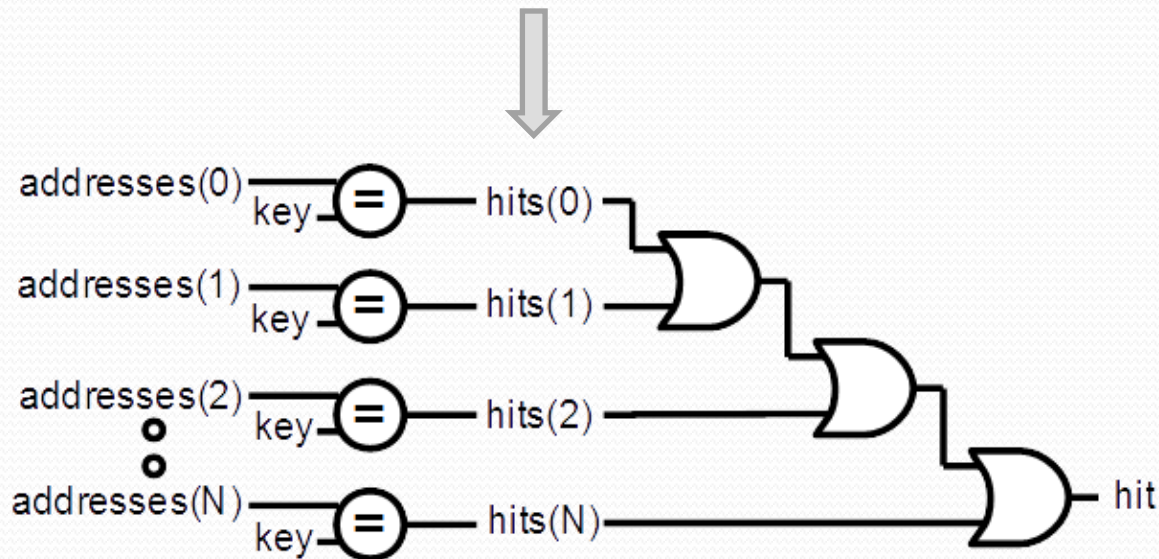
# Advanced examples





# Functional programming

```
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

```
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

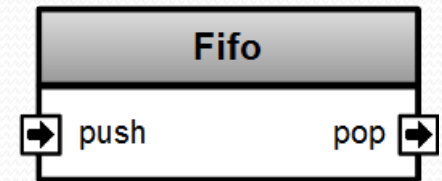
```
case class Flow[T <: Data](payloadType: T) extends Bundle {  
  val valid      = Bool  
  val payload : T = Flow(payloadType)  
}
```

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

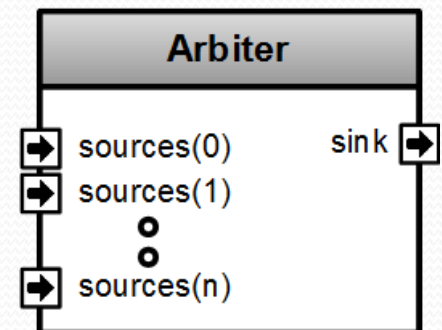
```
val myStreamOfRGB = Stream(RGB(8,8,8))
```

# 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)  
  }  
  //...  
}
```

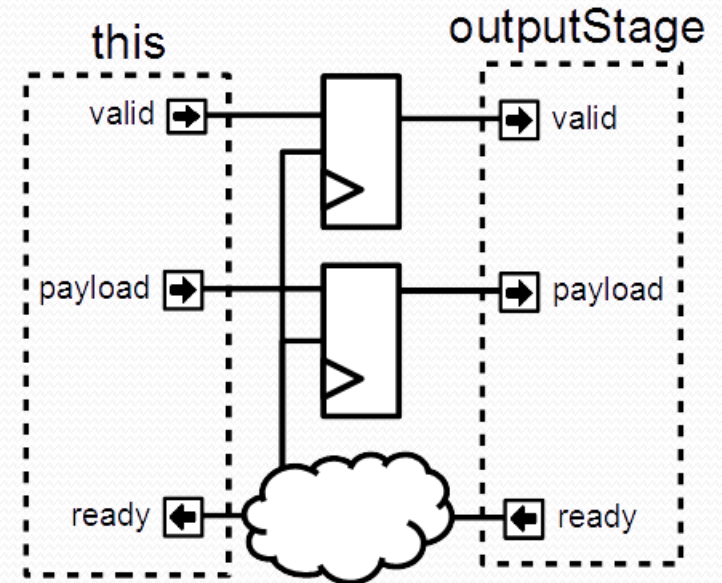


```
class Arbiter[T <: Data](payloadType: T, portCount: Int) extends Component {  
  val io = new Bundle {  
    val sources = Vec(slave(Stream(payloadType)), portCount)  
    val sink = master(Stream(payloadType))  
  }  
  //...  
}
```



# 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(dataType)  
    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

```
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)  
}
```

# Netlist analyser / Latency analysis

```
class MyComponentWithLatencyAssert extends Component {  
  val io = new Bundle {  
    val slavePort = slave Stream (UInt(8 bits))  
    val masterPort = master Stream (UInt(8 bits))  
  }
```

*//These 3 line are equivalent to io.slavePort.queue(16) >/-> io.masterPort*

```
  val fifo = new StreamFifo((UInt(8 bits)),16)
```

*fifo.io.push << io.slavePort // << is a connection operator without decoupling*

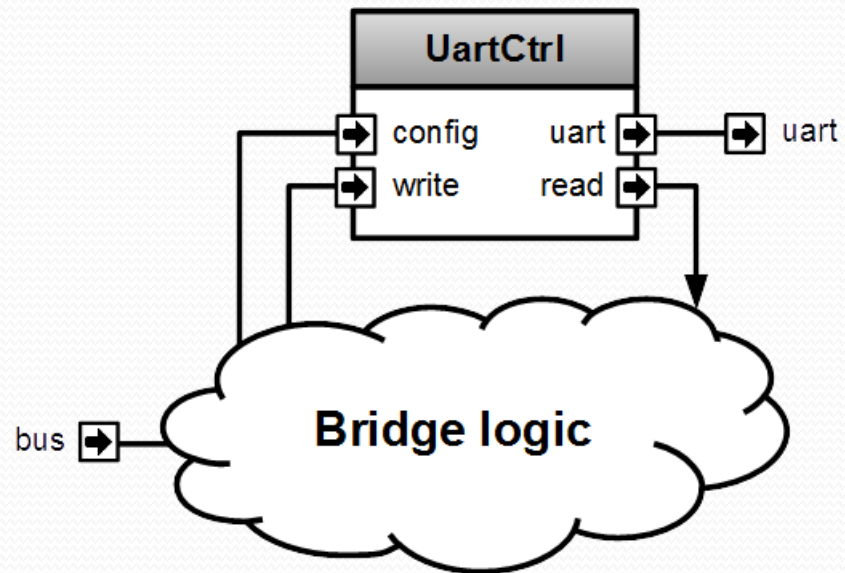
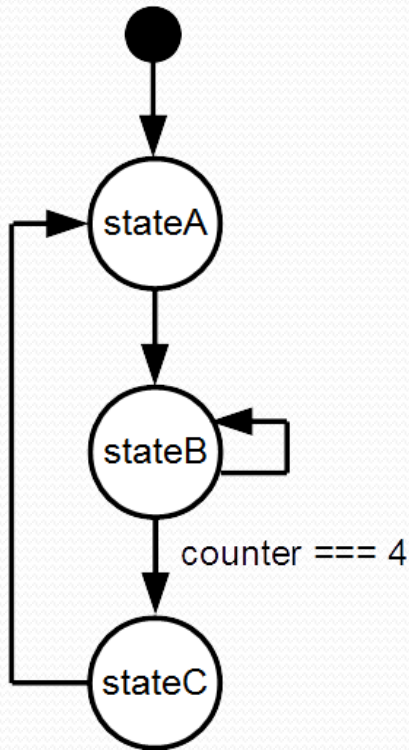
*fifo.io.pop >/-> io.masterPort // >/-> is a connection operator with decoupling*

```
  assert(3 == latencyAnalysis(io.slavePort.payload, io.masterPort.payload))
```

```
  assert(2 == latencyAnalysis(io.masterPort.ready, io.slavePort.ready))
```

```
}
```

# Meta-hardware description examples



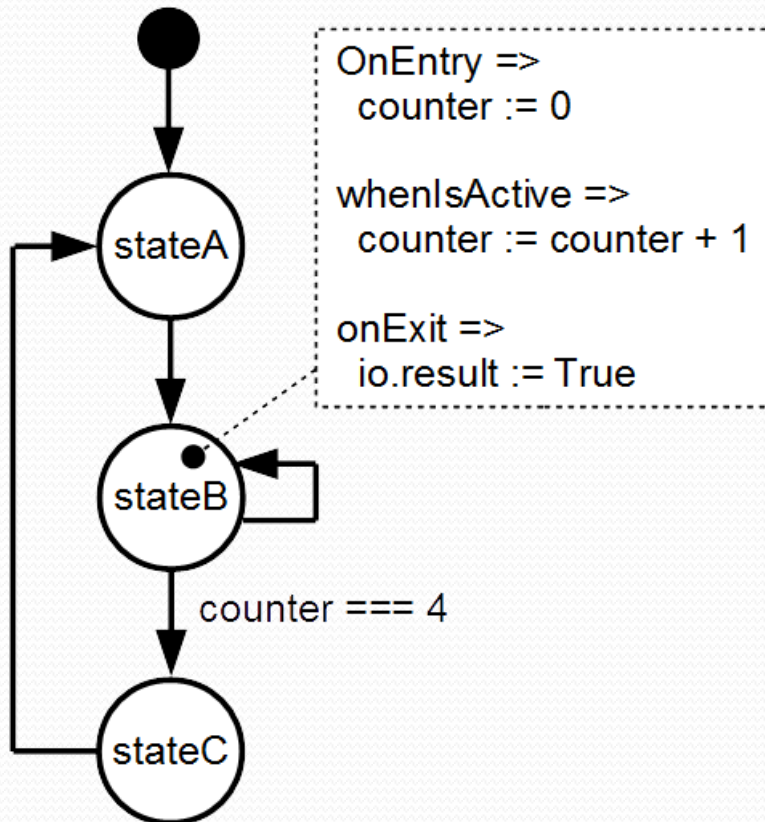


# FSM

- 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

# FSM style A

```
val io = new Bundle{  
  val result = out Bool  
}
```



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

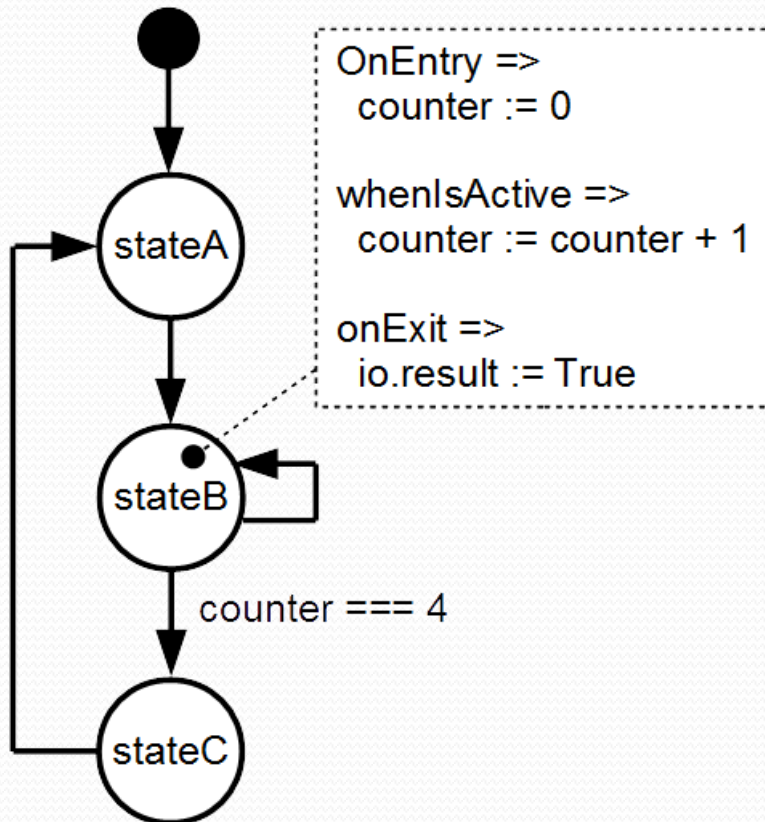
```
  val stateA : State = new State with EntryPoint{  
    whenIsActive (goto(stateB))  
  }
```

```
  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))  
  }  
}
```

# FSM style B

```
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))
```

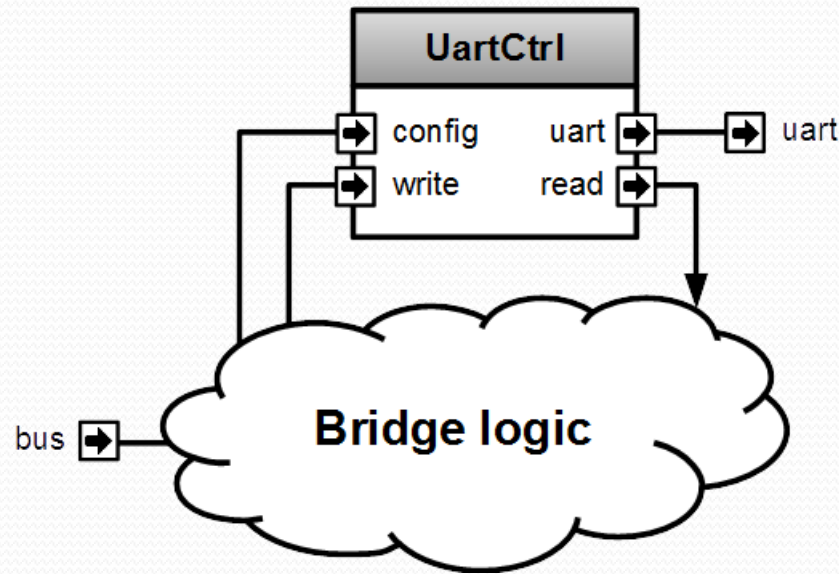
```
  stateB  
    .onEntry(counter := 0)  
    .whenIsActive {  
      counter := counter + 1  
      when(counter === 4){  
        goto(stateC)  
      }  
    }  
    .onExit(io.result := True)
```

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

```
}
```

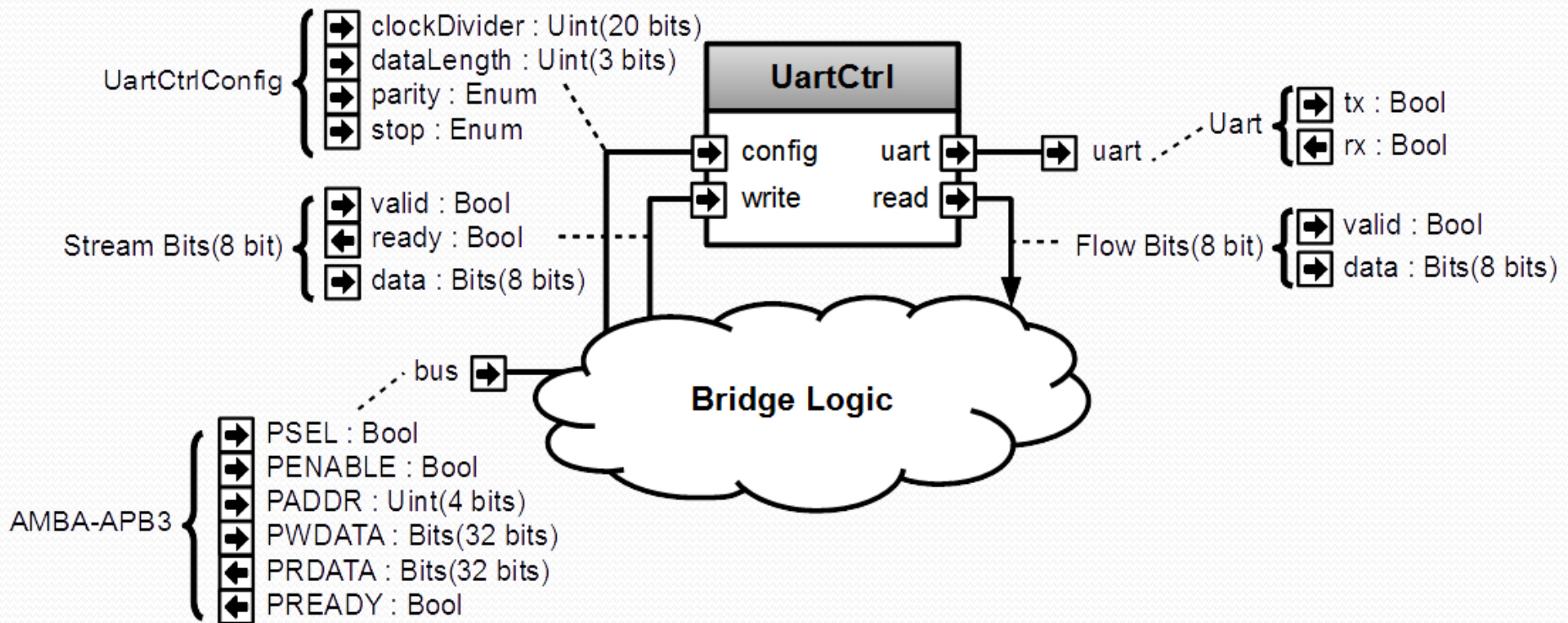
# Bus Slave Factory

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



# Bus Slave Factory

- Let's detail the situation



# Bus Slave Factory

- IO / instances / direct connections

```
class Apb3UartCtrl_A(rxFifoDepth : Int) extends Component {  
  val io = new Bundle {  
    val bus = slave(Apb3(addressWidth = 4, dataWidth = 32))  
    val uart = master(Uart())  
  }
```

```
// Instantiate an simple UART controller
```

```
val uartCtrl = new UartCtrl()
```

```
//Connect its UART bus
```

```
io.uart <> uartCtrl.io.uart
```

```
//Here we have to implement the "bridge logic".
```

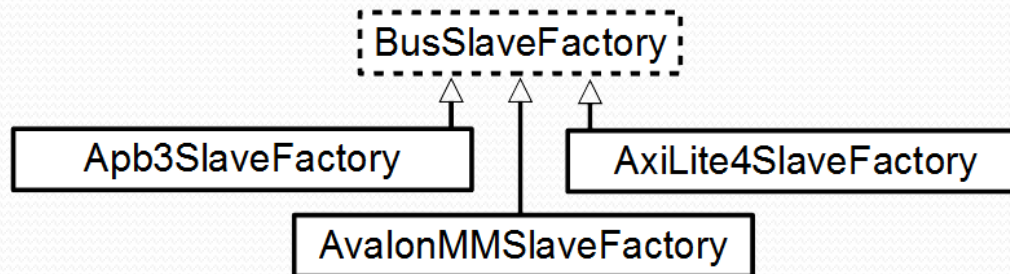
```
//All the code of next slides should be inserted there
```

```
//...
```

```
}
```

# Bus Slave Factory

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



```
val busCtrl = Apb3SlaveFactory(io.bus)
```

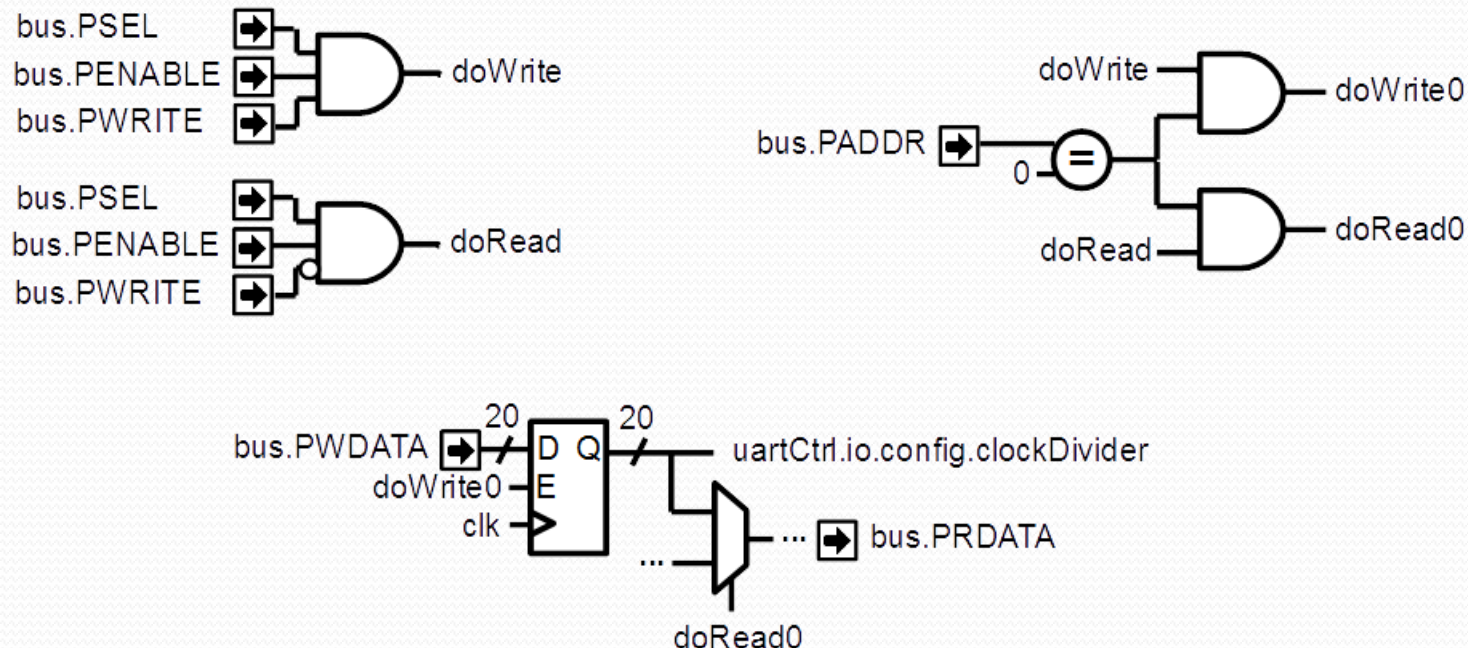
# Bus Slave Factory

- 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)***

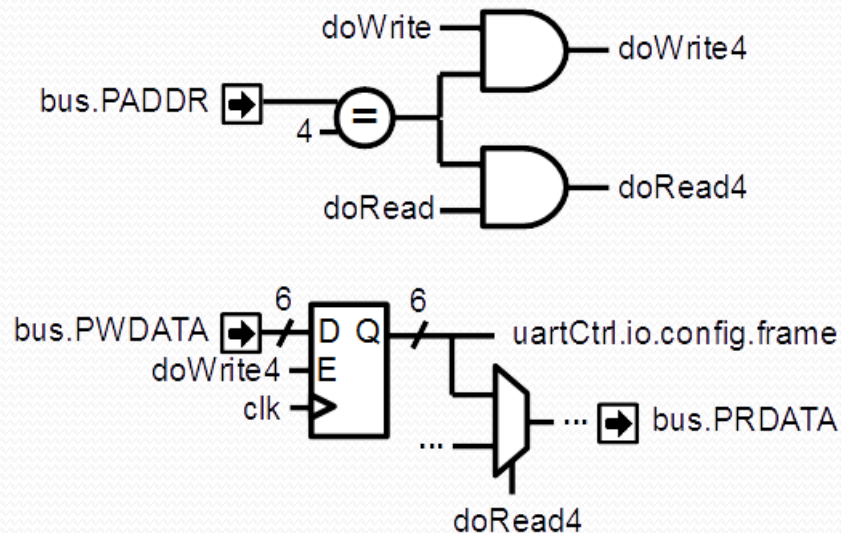




# Bus Slave Factory

- Make the frame config readable/writable by the bus

*// 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

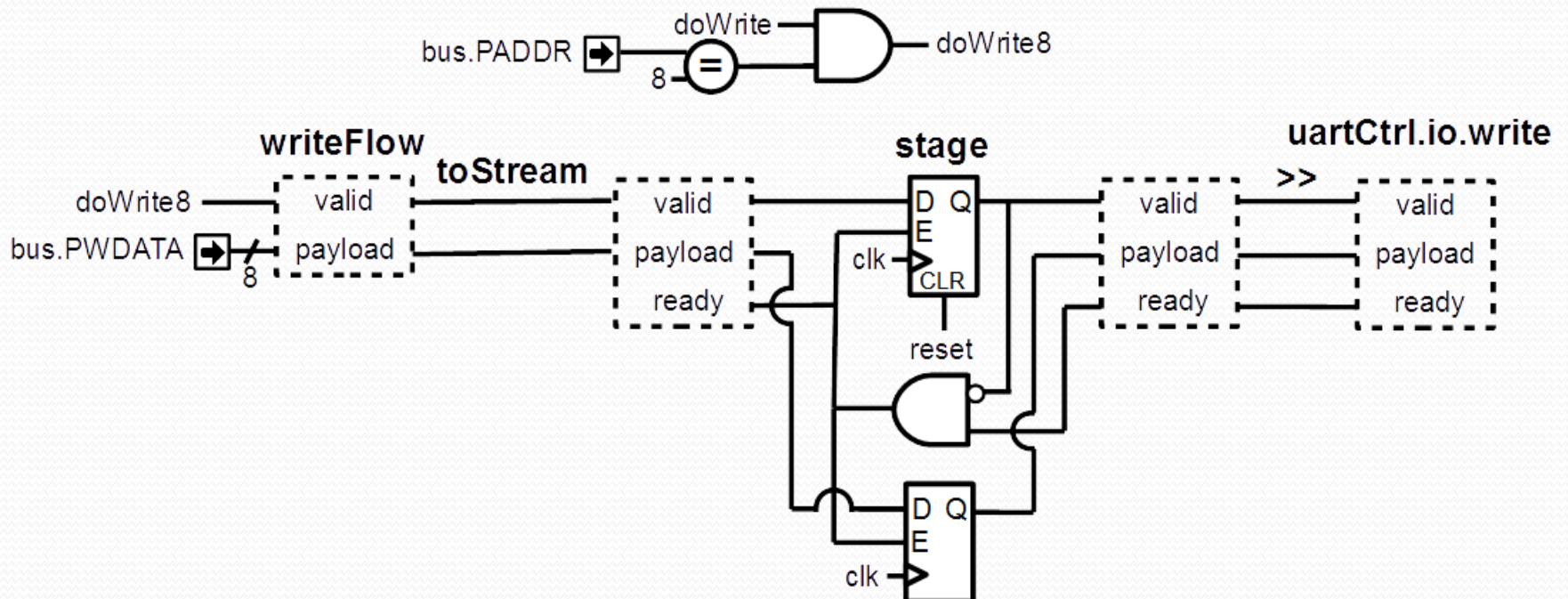
- Allow the bus to emit UART write requests

*// Ask the busCtrl to create a writable Flow[Bits] (valid/payload) at the address 8.*

*// Then convert it into a stream, add a 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
```

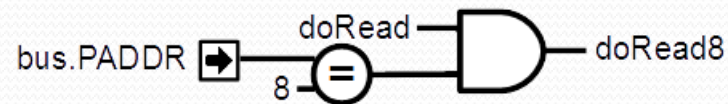


# Bus Slave Factory

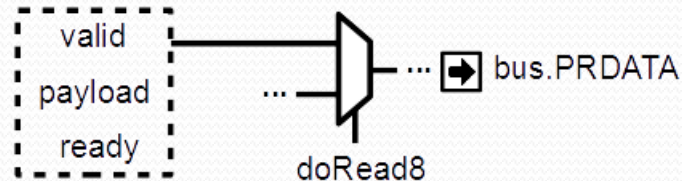
- Allow the bus to get the occupancy of the write buffer

*// 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*

*busCtrl.read(uartCtrl.io.write.valid,address = 8)*



**uartCtrl.io.write**



# Bus Slave Factory

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

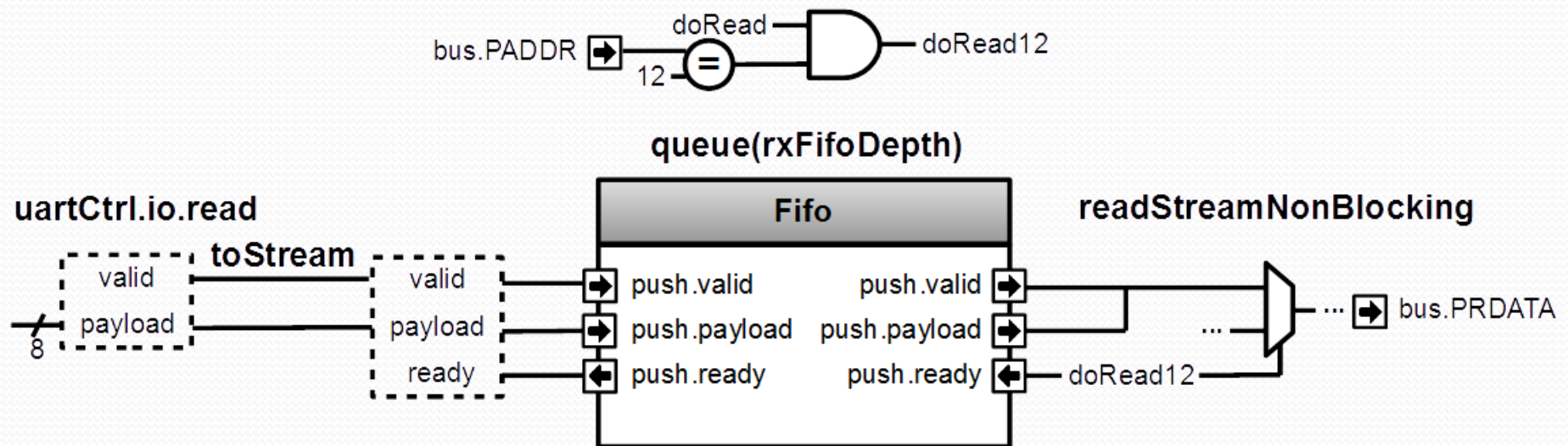
*// Take uartCtrl.io.read, convert it into a Stream, then connect it to the input of a FIFO*

*// 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, intelij)
  - 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

- RISC-V CPU, 5 stages, 1.15 DMIPS/Mhz
  - MUL/DIV
  - Instruction/Data cache
  - Interrupts
  - JTAG debugging
- Avalon/APB UART
- Avalon VGA
- Pipelined and multi-core fractal accelerator

# About Spinal project

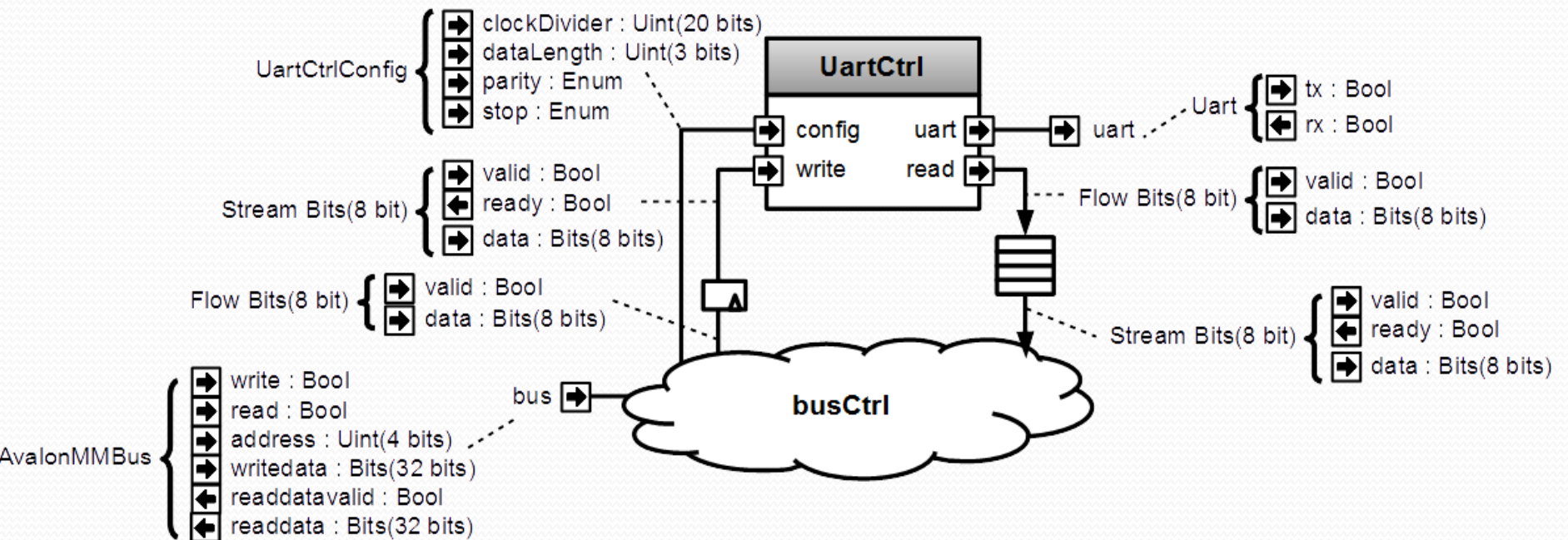
- Completely open source :
  - <https://github.com/SpinalHDL/SpinalHDL>
- Online documentation :
  - <https://spinalhdl.github.io/SpinalDoc/>
- Ready to use base project :
  - <https://github.com/SpinalHDL/SpinalBaseProject>
- Communication channels :
  - [spinalhdl@gmail.com](mailto:spinalhdl@gmail.com)
  - <https://gitter.im/SpinalHDL/SpinalHDL>
  - <https://github.com/SpinalHDL/SpinalHDL/issues>

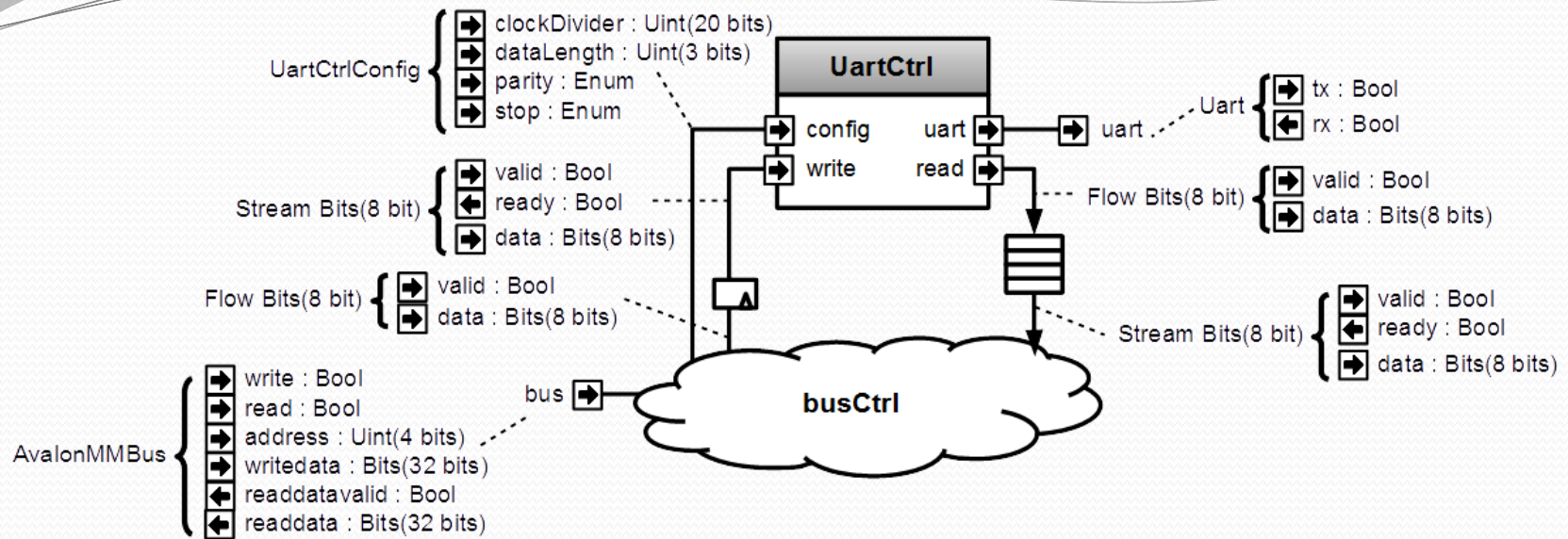




End / reserve slides

# Meta-hardware description





Name	Type	Access	Address	Description
clockDivider	UInt	RW	0	Set the UartCtrl clock divider
frame	UartCtrlFrameConfig	RW	4	Set the dataLength, the parity and the stop bit configuration
writeCmd	Bits	W	8	Send a write command to the UartCtrl
writeBusy	Bool	R	8	Bit 0 => zero when a new writeCmd could be sent
read	Bool / Bits	R	12	Bit 7 downto 0 => fifo pop payload Bit 31 => fifo pop valid

```

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)
}

```