



SpinalHDL

An alternative to standard HDL

Summary

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

Why a new language

- Because of current HDL :
 - Verbosity, endless wiring, copy paste
 - Wire level, can't define abstractions
 - Broken features
 - Can't parameterize records/struct
 - Can't define record's elements directions individually
 - SystemVerilog interface
 - No hardware «meta-description» capabilities
 - They were initially designed for simulation
 - Heavy legacy

Language introduction

- Open source , started in december 2014
- Focus on only on RTL
- Compatibility/interoperability is fine
 - It generate VHDL/Verilog files
 - It can integrate VHDL/Verilog IP as blackbox
- Abstraction level :
 - Start at the same level than VHDL
 - Finish between VHDL and HLS
 - The user can create new abstraction levels

Language dissection

- Spinal language is «integrated» in Scala
 - You can use all the Scala syntax / library
 - Scala IDE are helpfull and free
 - Object oriented and functional paradigms
- 2 layers
 - Core : Low level RTL
 - Lib : High level RTL, based on the Core layer
- How it work
 1. Use Spinal syntax to describe your RTL,
 2. Run Scala,
 3. VHDL/Verilog is generated.

At this point it's very important :

- To be open minded
- To forget pessimism as there is no logic overhead in the generated code.
- Not to be disturbed by the fact that Spinal HDL is only a RTL language. That's not a problem.
- Not to be afraid by the fact that you will have to simulate/synthesize a VHDL/Verilog file, while the specification is written in Spinal. That too isn't a problem.


Justification :

- There is many good verification solutions for the generated VHDL/Verilog (SystemVerilog, Formal verification, cocotb)
- The component hierarchy and all names are preserved durring the VHDL/Verilog generation. This make the navigation between the Scala code and the generated one easy.

A simple component

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

```
io.result := 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 result = out Bool  
  }  
}
```

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

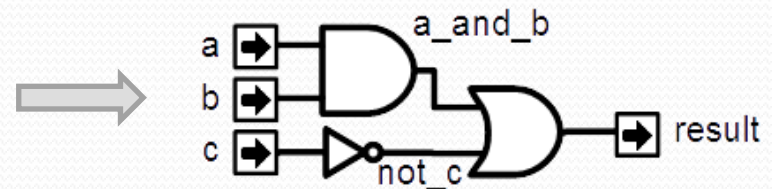


```
class MyComponent extends Component {  
  //...  
  //Latch/Loop detected, not allowed by Spinal  
  io.result := io.a | io.result  
}
```



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

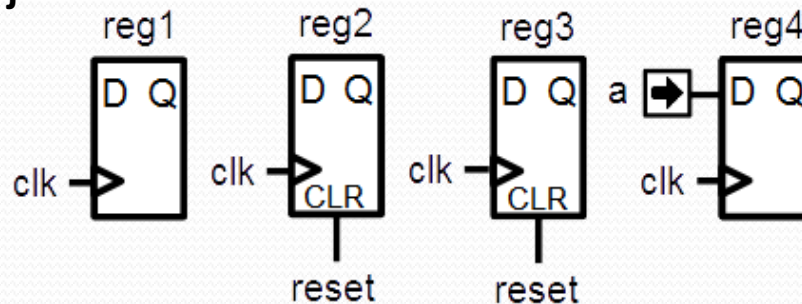
```
class MyComponent extends Component {  
  val io = new Bundle {  
    val a    = in Bool  
    val b    = in Bool  
    val c    = in Bool  
    val output = out Bool  
  }  
  val a_and_b = io.a & io.b  
  val not_c = !io.c  
  io.output := a_and_b | not_c  
}
```

```
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)  
}
```

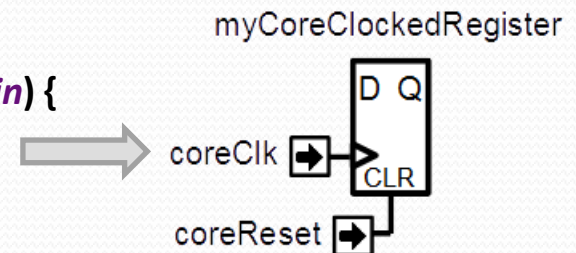


ClockDomains

```
class MyTopLevel extends Component {  
  val io = new Bundle {  
    val coreClk    = in Bool  
    val coreReset  = in Bool  
  }  
}
```

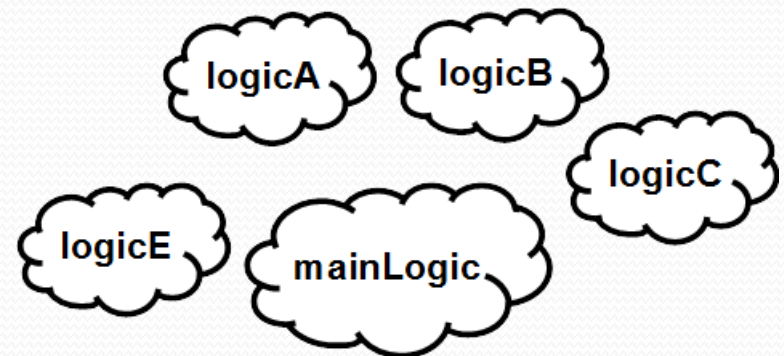
```
val coreClockDomain = ClockDomain(  
  clock = io.coreClk,  
  reset = io.coreReset,  
  config = ClockDomainConfig(  
    clockEdge      = RISING,  
    resetKind      = ASYNC,  
    resetActiveLevel = HIGH  
  )  
)
```

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



Organize things

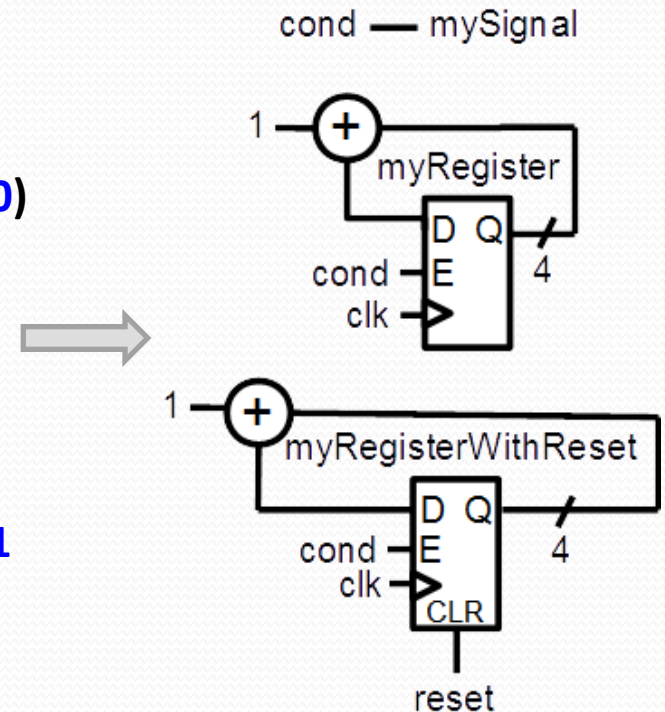
```
class UartCtrlTx extends Component {  
  val io = new Bundle {  
    // io definition  
  }  
  val timer = new Area {  
    // emit a pulse that is used as time reference  
    // in the state machine  
  }  
  val stateMachine = new Area {  
    // some logic  
  }  
}
```



Unify logic and FF

```
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
}
```



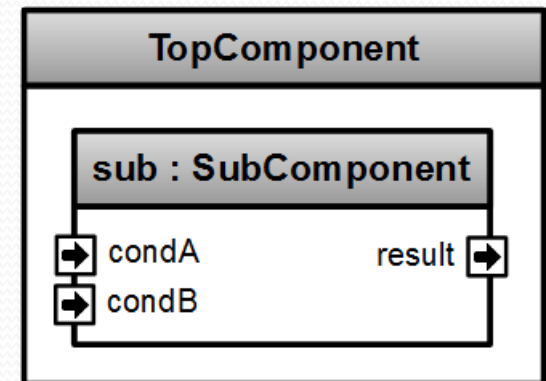
No more component binding

```
architecture arch of something is
  signal sub_condA : std_logic;
  signal sub_condB : std_logic;
  signal sub_result : std_logic;

  component SubComponent
    port(condA : in std_logic;
         condB : in std_logic;
         result : out std_logic);
  end component;
begin
  sub : SubComponent
    port map (condA => sub_condA
             condB => sub_condB
             result => sub_result);
end arch;
```



```
class TopComponent extends Component{
  val sub = new SubComponent
}
```



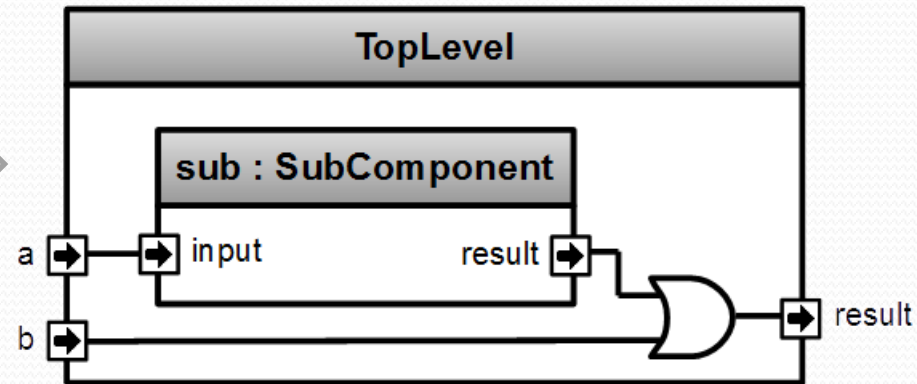
Component instance

```
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 output = out Bool  
  }  
}
```

```
val sub = new SubComponent
```

```
sub.io.input := io.a  
io.output := 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)  
  }  
}
```

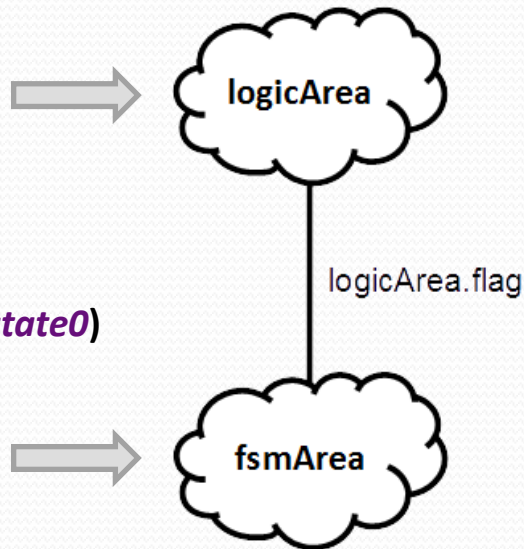
```
when(io.conds(0)){  
  io.result := 2  
  when(io.conds(1)){  
    io.result := 1  
  }  
} otherwise {  
  io.result := 0  
}  
}
```



Enum, Area, switch

```
class TopLevel extends Component {  
  //...  
  val logicArea = new Area {  
    val flag = Bool  
    //...  
  }
```

```
  val fsmArea = new Area {  
    val state = RegInit(MyEnum.state0)  
    switch(state) {  
      is(MyEnum.state0) {  
        when(logicArea.flag) {  
          state := MyEnum.state1  
        }  
      }  
    }  
    default {  
      //...  
    }  
  }  
}
```



```
object MyEnum extends SpinalEnum {  
  val state0, state1, anotherState = newElement  
}
```

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)  
  }  
}
```

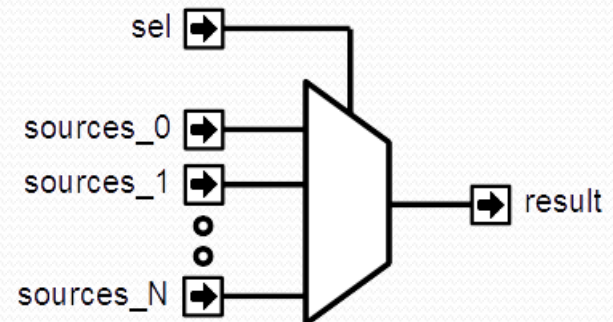
```
  var c = False  
  for (i <- 0 until size) {  
    val a = io.a(i)  
    val b = io.b(i)  
  
    io.result(i) := a ^ b ^ c  
    c |= (a & b) | (a & c) | (b & c);  
  }  
}
```

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

```
io.result := io.sources(io.sel).asBits  
}
```



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)

Less scope limitations

```
val valid = Bool  
val regA = Reg(UInt(4 bit))
```

```
def doSomething(value : Int) = {  
  valid := True  
  regA := value  
}
```

```
when(???) {  
  doSomething(4)  
}
```

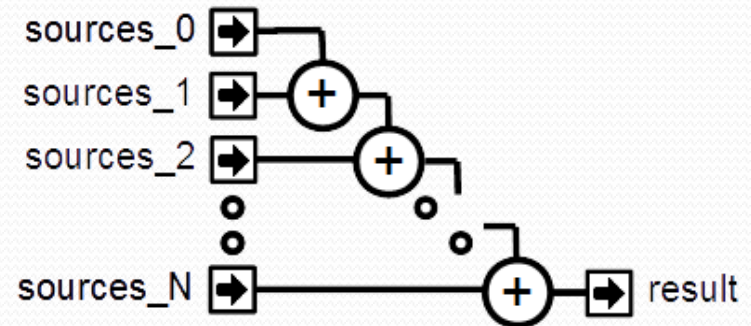
Function, User utils (1)

```
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 = {  
    val result = cloneOf(this)  
    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(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  
// io.result := io.sources.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](dataType: T) extends Bundle {  
  val valid = Bool  
  val data: T = cloneOf(dataType)  
}
```

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

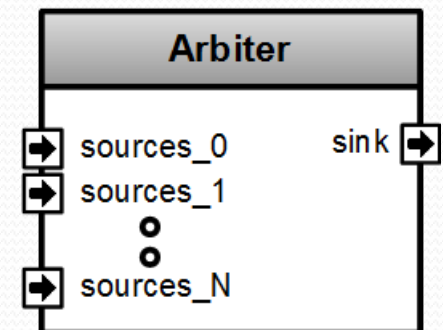
```
val myStreamOfRGB = Stream(RGB(8,8,8))
```

Stream components

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

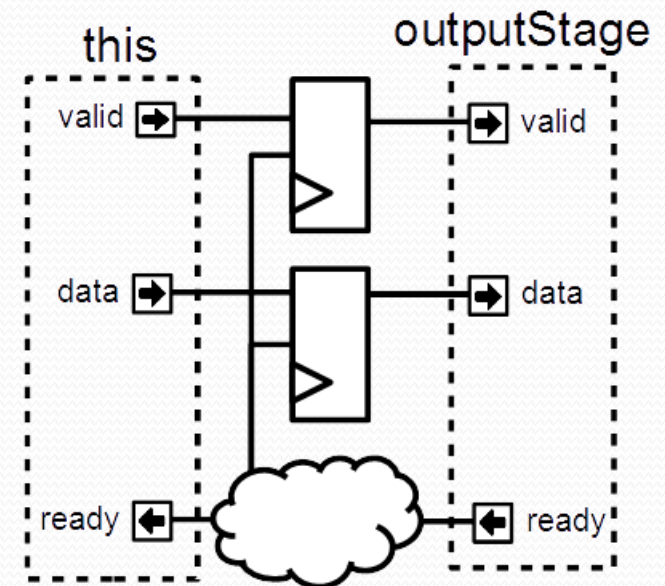


```
class Arbiter[T <: Data](dataType: T, portCount: Int) extends Component {  
  val io = new Bundle {  
    val sources = Vec(slave(Stream(dataType)), portCount)  
    val sink = master(Stream(dataType))  
  }  
  //...  
}
```



Stream functions

```
case class Stream[T <: Data](dataType: T) extends Bundle {  
  // ...  
  def connectFrom(that: Stream[T]) = {  
    // some connections between this and that  
  }  
  def m2sPipe(): Stream[T] = {  
    val outputStage = cloneOf(this)  
    val validReg = RegInit(False)  
    val dataReg = Reg(dataType)  
    // some logic  
    return outputStage  
  }  
  def << (that: Stream[T]) = this.connectFrom(that)  
  def <-< (that: Stream[T]) = this << that.m2sPipe()  
}
```



```
val myStreamA, myStreamB = Stream(UInt(8 bit))  
myStreamA <-< myStreamB
```

Functional programming

```
Case class LineTag extends Bundle {
```

```
  val valid = Bool
```

```
  val address = UInt(32 bit)
```

```
  val dirty = Bool
```

```
  def hit(targetAddress : UInt) : Bool = valid && address === targetAddress  
}
```

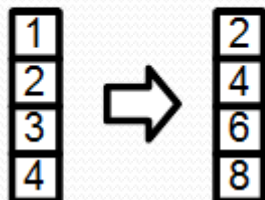
```
val lineTags = Vec(LineTag(), 8)
```

```
val lineHits = lineTags.map(lineTag => lineTag.hit(targetAddress))
```

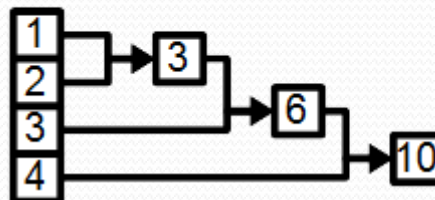
```
val lineHitValid = lineHits.reduce((a,b) => a || b)
```

```
val lineHitIndex = OHToUInt(lineHits)
```

map($x \times 2$)



reduce($x + y$)



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 bit), initialContent = sinTable)  
  val phase = CounterFreeRun(sampleCount)  
  val 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

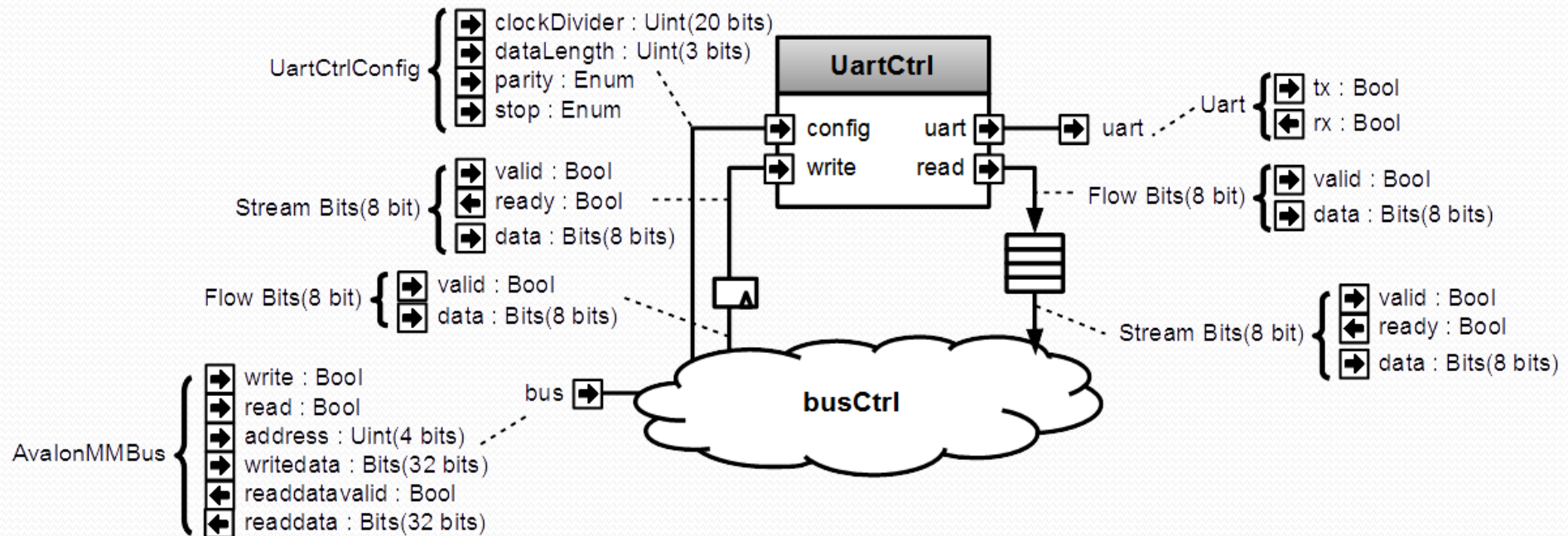
fifo.io.pop >/-> io.masterPort //>/-> is a connection operator with decoupling

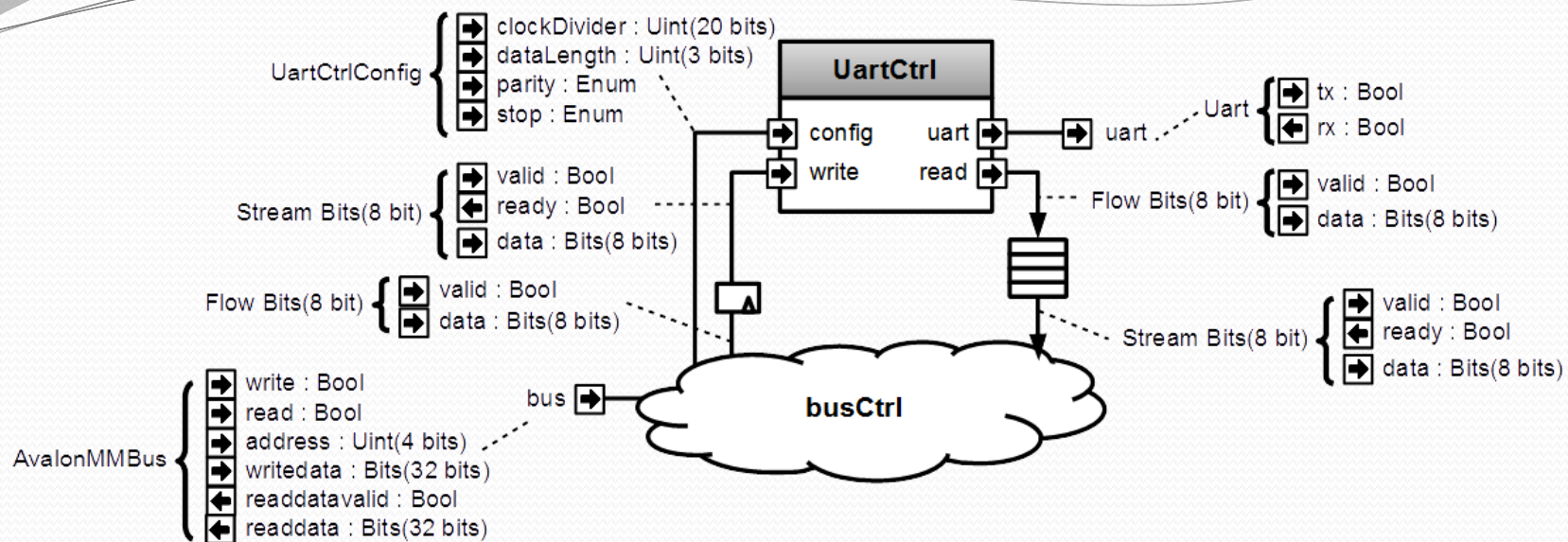
```
  assert(3 == latencyAnalysis(io.slavePort.data,io.masterPort.data))
```

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

```
}
```

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	Bits ## Bool	R	12	Bit 0 => read data valid Bit 8 downto 1 => read data

```

class AvalonUartCtrl(uartCtrlConfig : UartCtrlGenerics, rxFifoDepth : Int) extends Component{
  val io = new Bundle{
    val bus = slave(AvalonMM(AvalonMMUartCtrl.getAvalonMMConfig))
    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)
}

```

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
 - <https://gitter.im/SpinalHDL/SpinalHDL>
 - <https://github.com/SpinalHDL/SpinalHDL/issues>



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
```

```
}
```

Flow, Stream, Fragment

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

```
case class Stream[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)  
  //..  
}
```


Stream functions

```
val cond = Bool
val inPort = Stream(Bits(32 bit))
val outPort = Stream(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 = Stream(UInt(10 bit))
val memReadPort = mem.streamReadSync(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
```