SpinalHDL

An alternative hardware description language

Summary

- VHDL and Verilog are not so good
- SpinalHDL introduction
- Examples

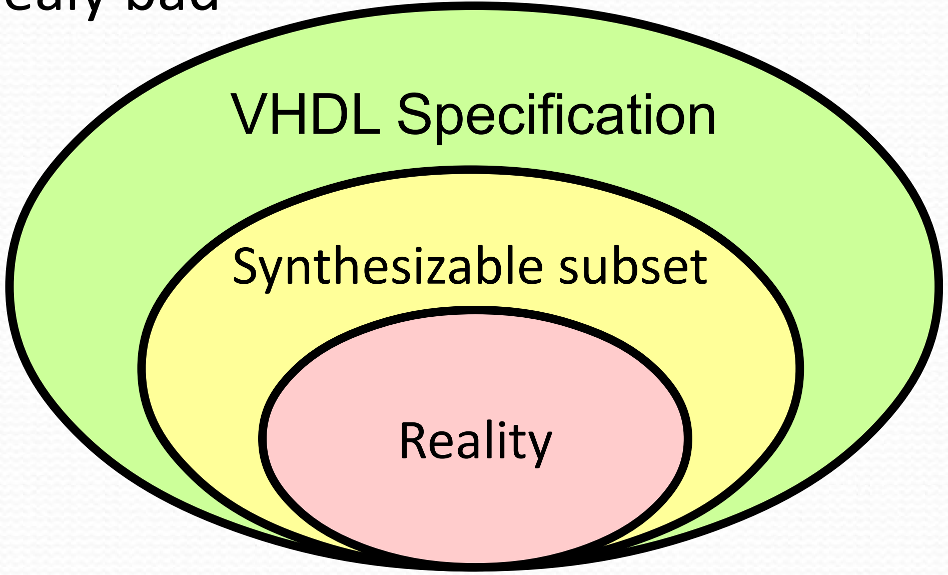
- Disclamer
  - This talk will only be about synthesizable hardware

VHDL and Verilog

- Initialy designed for simulation/documentation purposes, a long time ago
  - Process/Always blocks doesn't make sense in RTL
  - No object oriented programming, no functional programming
- Simple concepts are verbose
  - Component/Module instanciation
  - Interface instanciation
- No meta-hardware description capabilites

Blessed VHDL-2008 and SV ?

- Not realy
  - They keep the same paradigm to infer RTL (simulation constructs + event driven)
  - They didn't offer any meta-hardware description capabilities
  - VHDL-2008 and SV synthesis support could be realy bad
  - SV interface definitions are limited



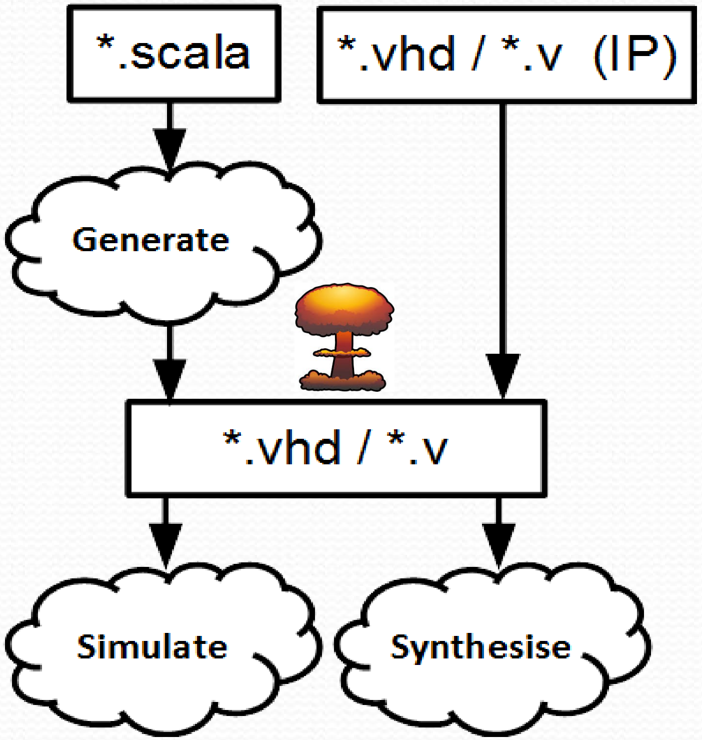
So, what to do next ?

- Lets use VHDL and Verilog as netlist language !

Source: SonicBomb.com galleries

SpinalHDL introduction

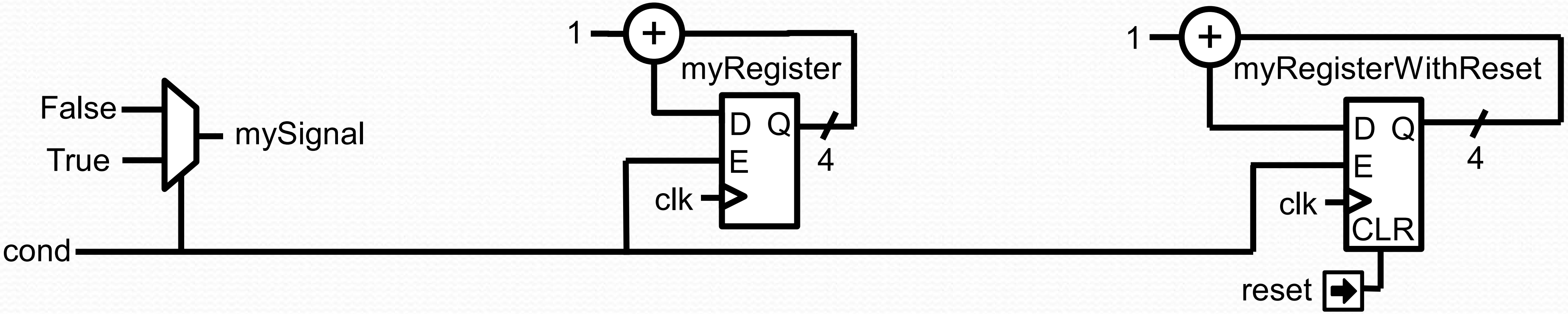
- Open source , started in december 2014
- Focus on RTL description
- Thinked to be interoperable with existing tools
  - It generates VHDL/Verilog files (as an output netlist)
  - It can integrate VHDL/Verilog IP as blackbox
- Abstraction level :
  - You can design things similary to VHDL/Verilog
 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



Some points about SpinalHDL

- There is no logic overhead in the generated code. (I swear !)
- The component hierarchy and all names are preserved during the VHDL/Verilog generation. (Good for simulations)
- It is an language hosted on the top of Scala ! (And it is a very good thing)

By using events driven constructs (VHDL)

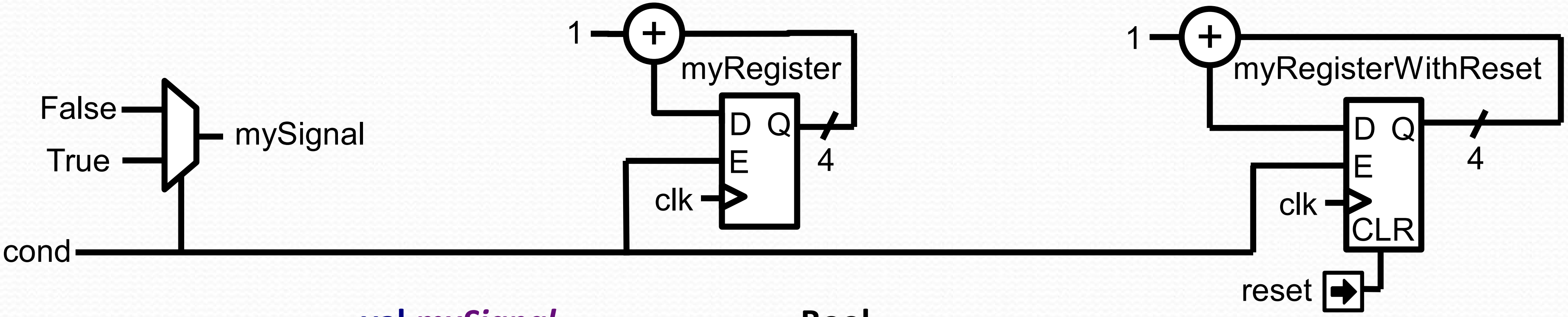


```
signal mySignal : std_logic;

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

```
signal myRegister : unsigned(3 downto 0);

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

```
signal myRegisterWithReset : unsigned(3 downto 0);

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;
```

By using an dedicated syntax (SpinalHDL)

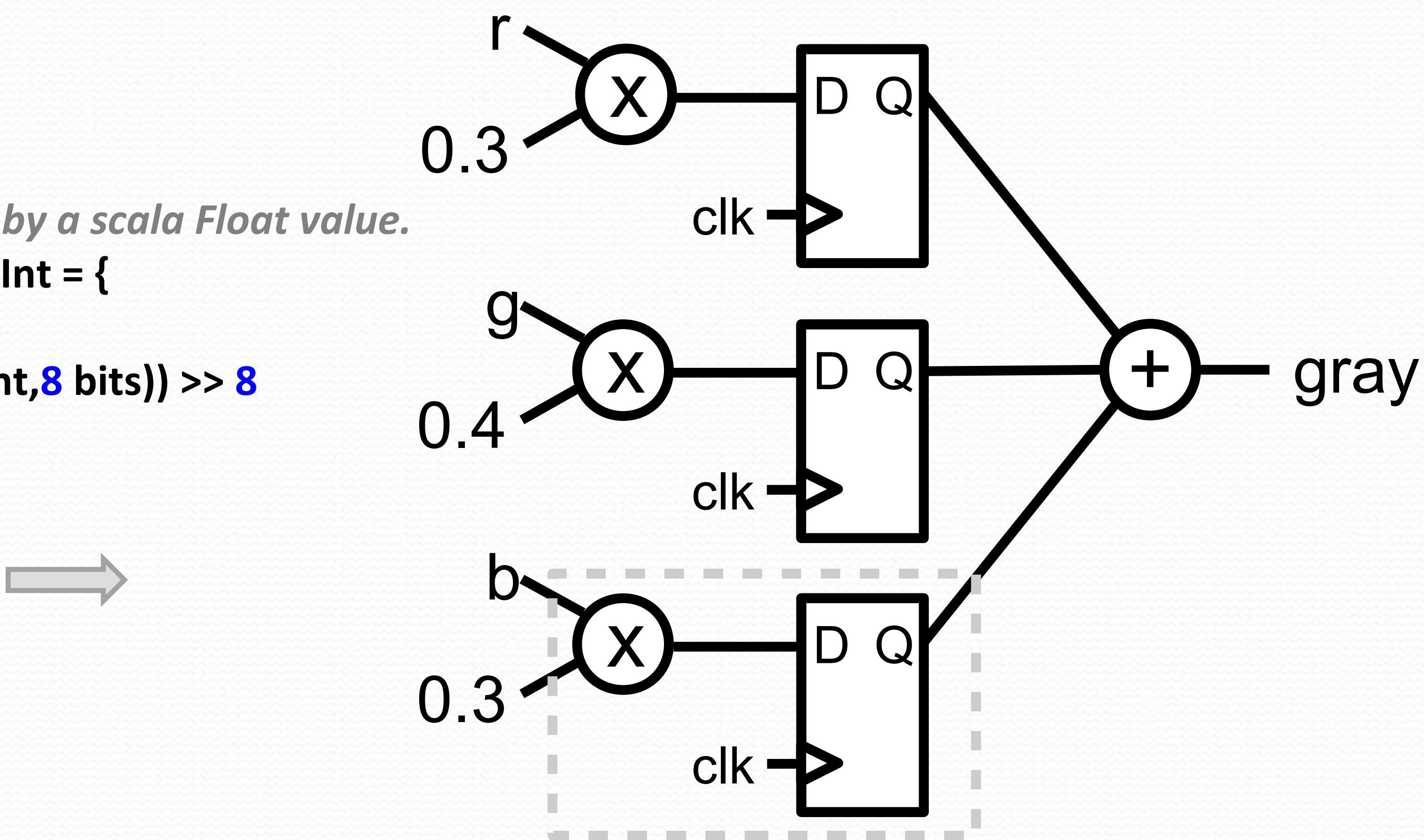


```
val mySignal                      = Bool
val myRegister                     = Reg(UInt(4 bits))
val myRegisterWithReset    = Reg(UInt(4 bits)) init(0)

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

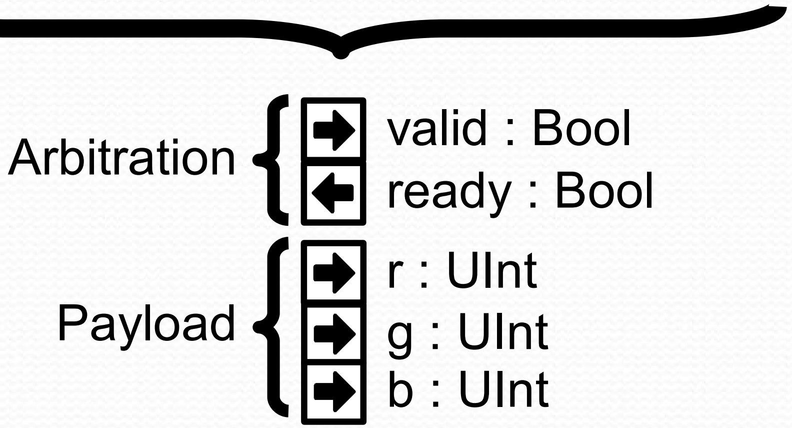
SpinalHDL => 9 lines
VHDL   => 28 lines

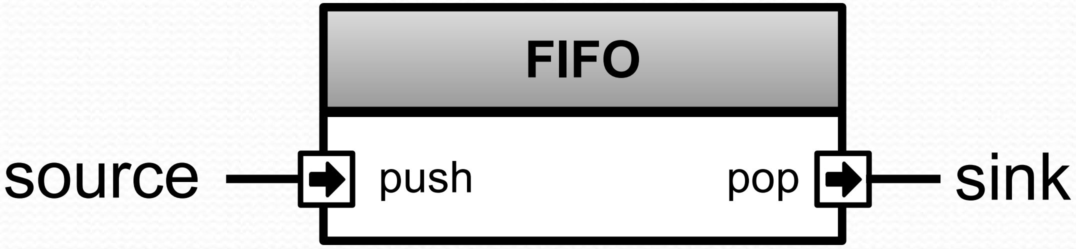
Real functions capabilities

```
// 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 = {
  val resultReg = Reg(UInt(8 bits))
  resultReg := (value * U((255*by).toInt,8 bits)) >> 8
  return resultReg
}

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

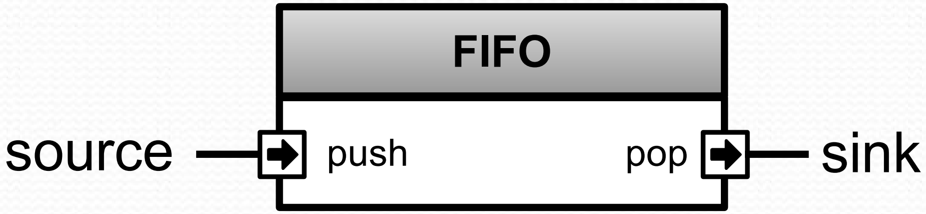


Having a Hand-shake bus of color and wanting to queue it ?





In standard VHDL

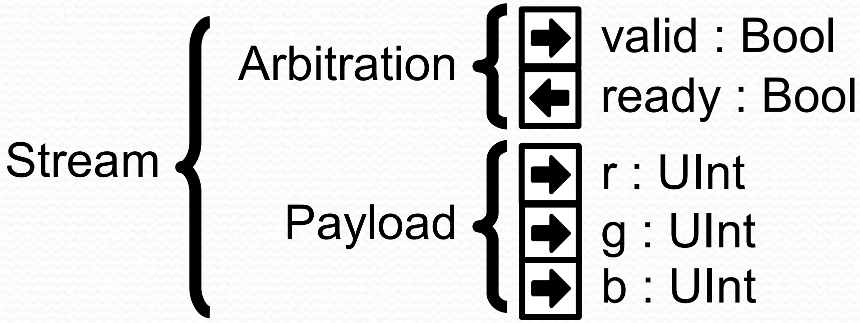


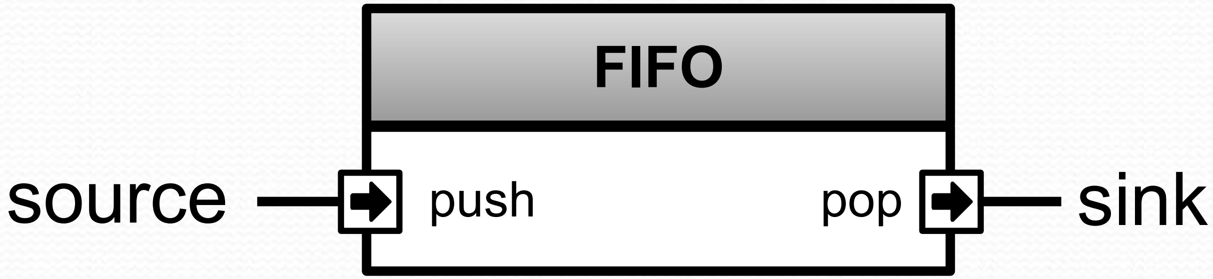
```
signal source_valid  : std_logic;
signal source_ready : std_logic;
signal source_r         : unsigned(4 downto 0);
signal source_g         : unsigned(5 downto 0);
signal source_b         : unsigned(4 downto 0);

signal sink_valid : std_logic;
signal sink_ready : std_logic;
signal sink_r         : unsigned(4 downto 0);
signal sink_g         : unsigned(5 downto 0);
signal sink_b        : unsigned(4 downto 0);
```

```
fifo_inst : entity work.Fifo
  generic map (
    depth              => 16,
    payload_width => 16
  )
  port map (
    clk => clk,
    reset => reset,
    push_valid => source_valid,
    push_ready => source_ready,
    push_payload(4  downto  0)   => source_payload_r,
    push_payload(10 downto  5)  => source_payload_g,
    push_payload(15 downto 11) => source_payload_b,
    pop_valid => sink_valid,
    pop_ready => sink_ready,
    pop_payload(4  downto  0)   => sink_payload_r,
    pop_payload(10 downto  5)  => sink_payload_g,
    pop_payload(15 downto 11) => sink_payload_b
  );
```

In SpinalHDL

```
val source, sink = Stream(RGB(5,6,5))
val fifo = StreamFifo(
   dataType = RGB(5,6,5),
   depth = 16
)
fifo.io.push << source
fifo.io.pop >> sink
```





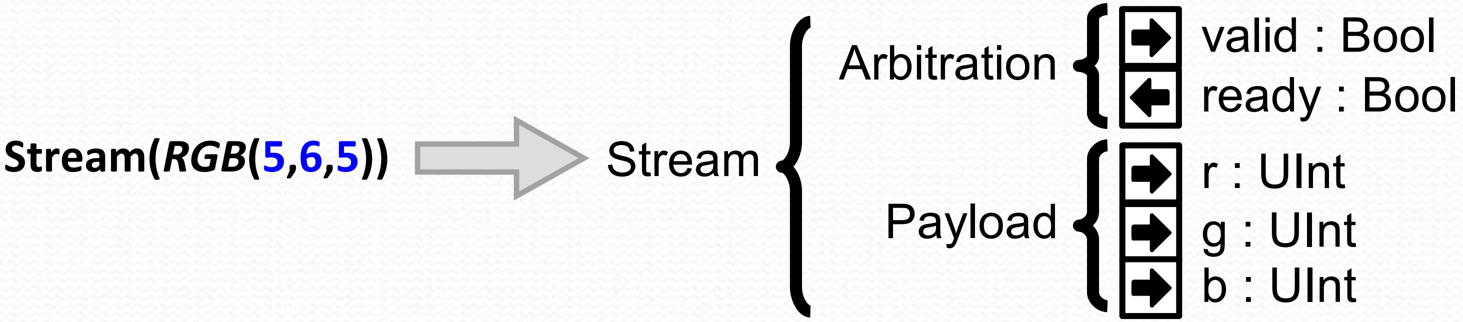
SpinalHDL => 7 lines
VHDL => 29 lines

About RGB

```
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)
}
```

About Stream

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

  override def asMaster(): Unit = {
    out(valid,payload)
    in(ready)
  }

  def >>(sink: Stream[T]): Unit ={
    sink.valid   := this.valid
    this.ready   := sink.ready
    sink.payload := this.payload
  }
  def <<(source: Stream[T]): Unit = source >> this
  def queue(depth : Int) : Stream[T] = {...}
}
```


Stream(*RGB*(5,6,5))

Stream

Arbitration

valid : Bool

ready : Bool

Payload

r : UInt

g : UInt

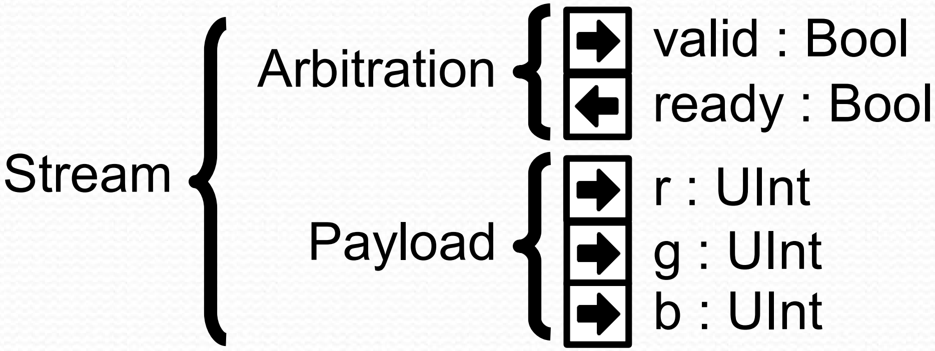
b : UInt

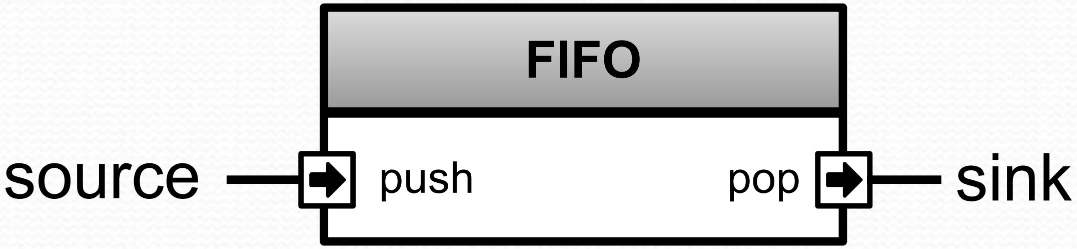
Queuing in SpinalHDL++

```
val source, sink = Stream(RGB(5,6,5))
val fifo = StreamFifo(
   dataType = RGB(5,6,5),
   depth = 16
)
fifo.io.push << source
fifo.io.pop >> sink
```

```
val source, sink = Stream(RGB(5,6,5))

source.queue(16) >> sink
```

SpinalHDL => 2 lines
VHDL   => 29 lines

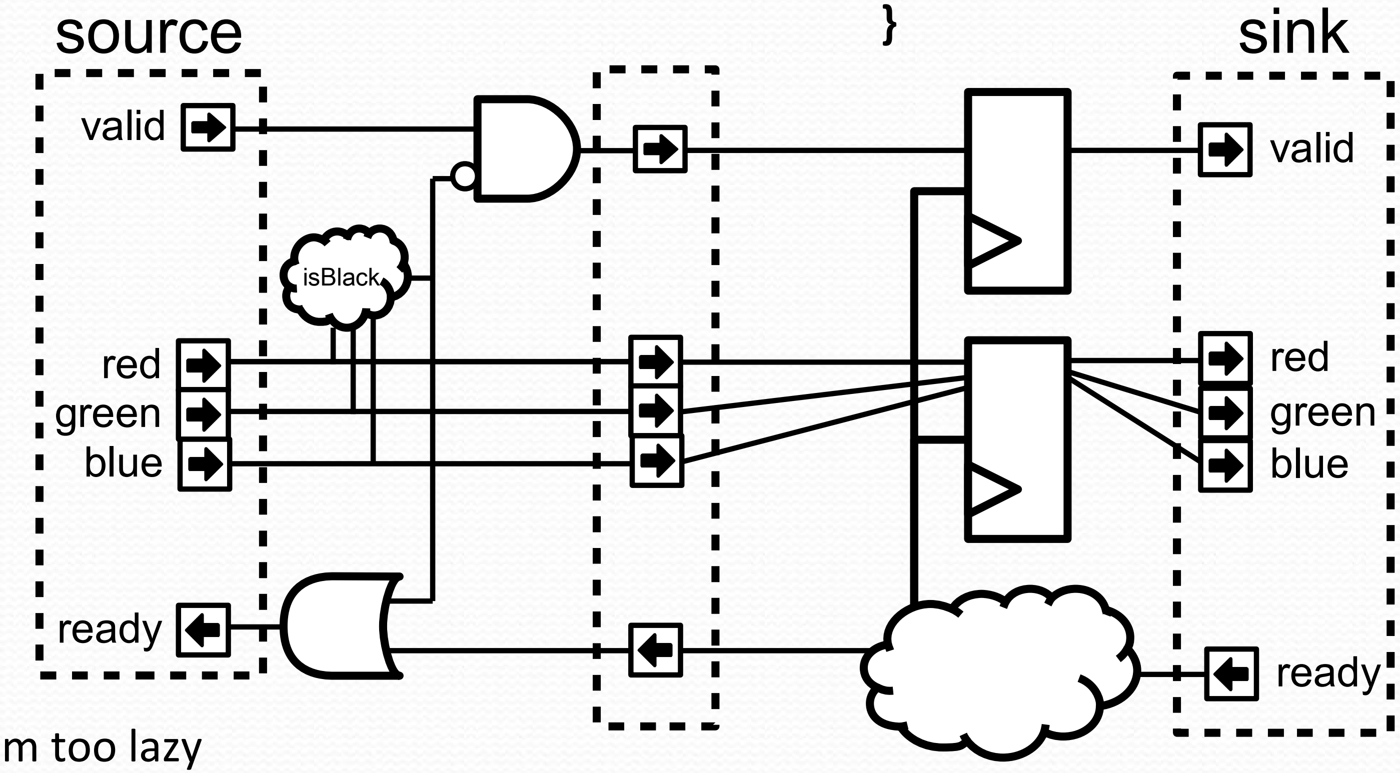




Abstract arbitration

```
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)

  def isBlack : Bool = (r === 0 && g === 0 && b === 0)
}
```

```
val source = Stream(RGB(5,6,5))
val sink = source.throwWhen(source.payload.isBlack).stage()
```



SpinalHDL => 2 lines
VHDL => Sorry, I'm too lazy

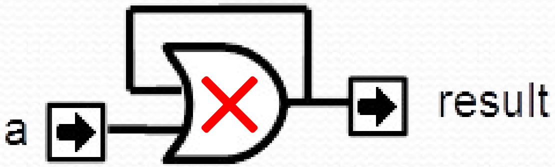
Abstract arbitration

```
def throwWhen(cond: Bool): Stream[T] = {
  val ret = Stream(dataType)

  ret << this
  when(cond) {
    ret.valid := False
    this.ready := True
  }
  return ret
}
```

```
def stage(): Stream[T] = {
  val ret = Stream(dataType)

  val rValid  = Reg(Bool) init(False)
  val rData   = Reg(dataType)
  this.ready := ! ret.valid || ret.ready

  when(this.ready) {
    rValid := this.valid
    rData := this.payload
  }

  ret.valid := rValid
  ret.payload := rData

  return ret
}
```

Safty first !

```
val a      = Bool
val result = Bool
result := a | result //Loop detected by SpinalHDL
```



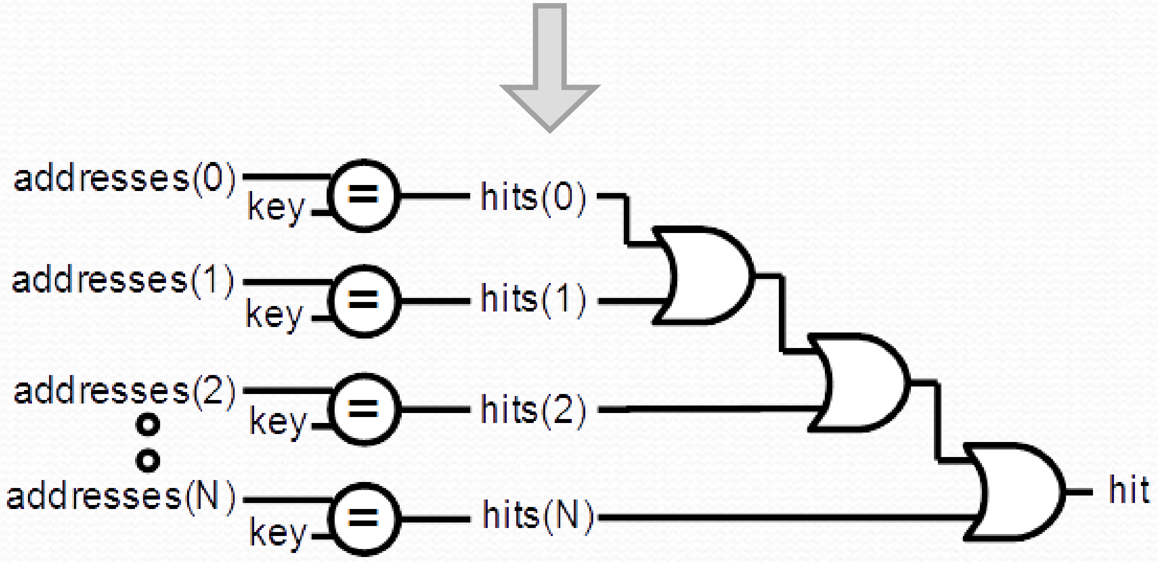
```
val result = Bool
when(cond){    //result is not assigned in all cases => Latch detected by SpinalHDL
   result := True
}
```

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){ …}
```

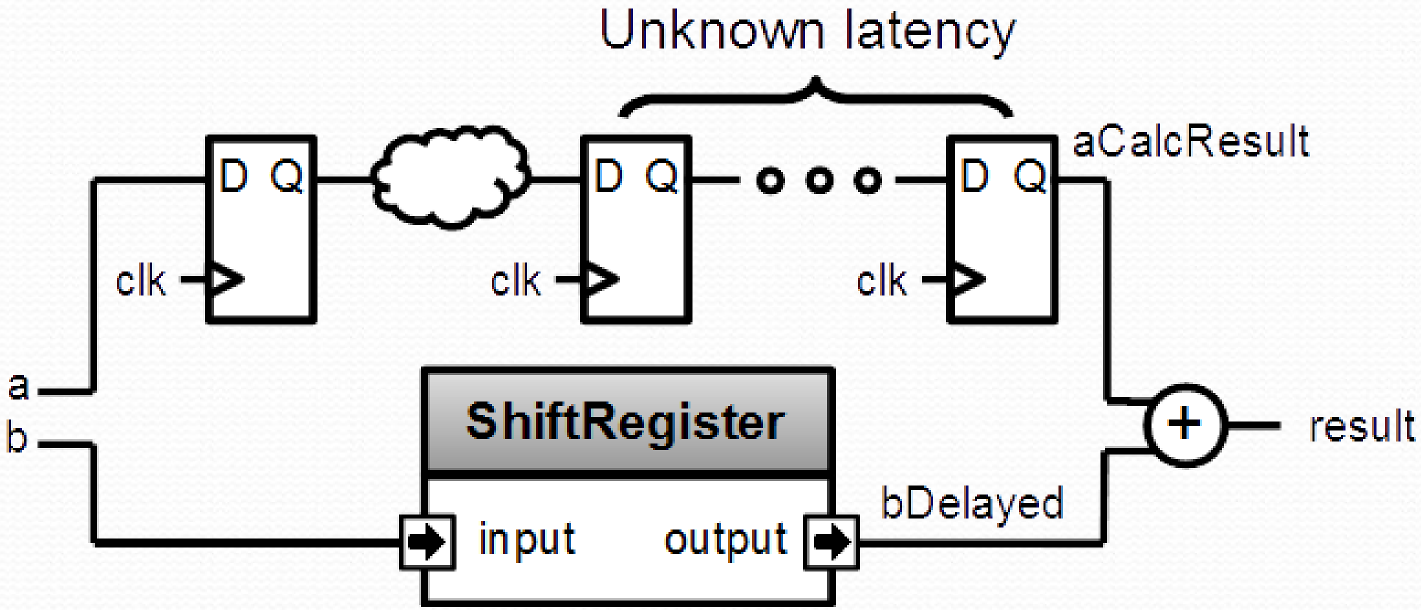
Functional programming

```
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)
```



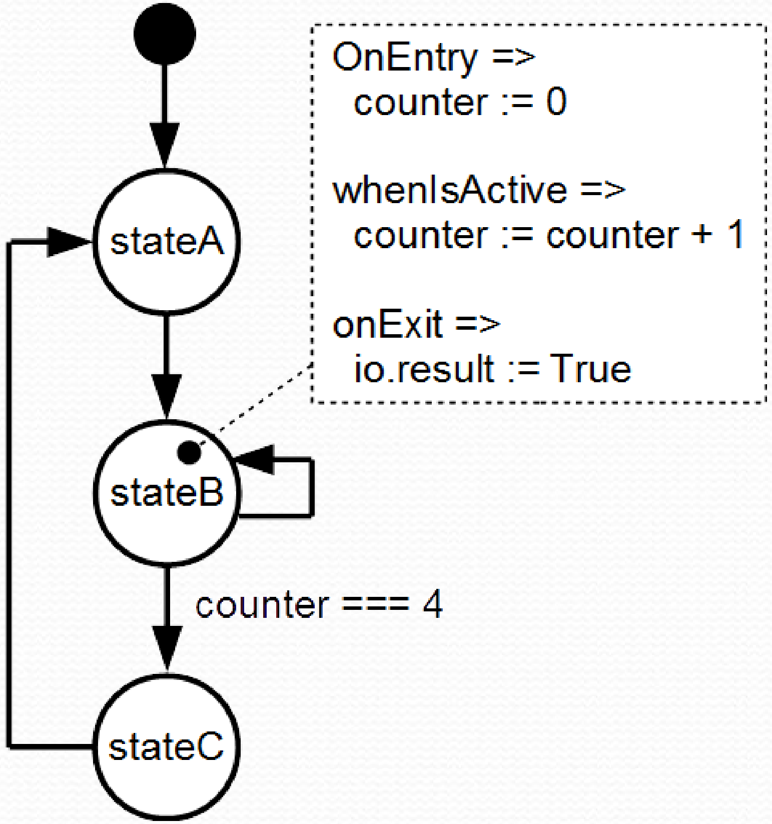
Design introspection

```
val a = UInt(8 bits)
val b = UInt(8 bits)

val aCalcResult = complicatedLogic(a)

val aLatency = LatencyAnalysis(a,aCalcResult)
val bDelayed = Delay(b,cycleCount = aLatency)

val result = aCalcResult + bDelayed
```



FSM

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



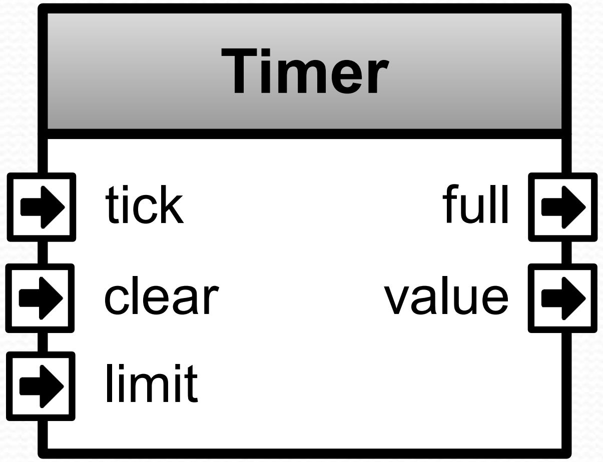
OnEntry =>
  counter := 0

whenIsActive =>
  counter := counter + 1

onExit =>
  io.result := True

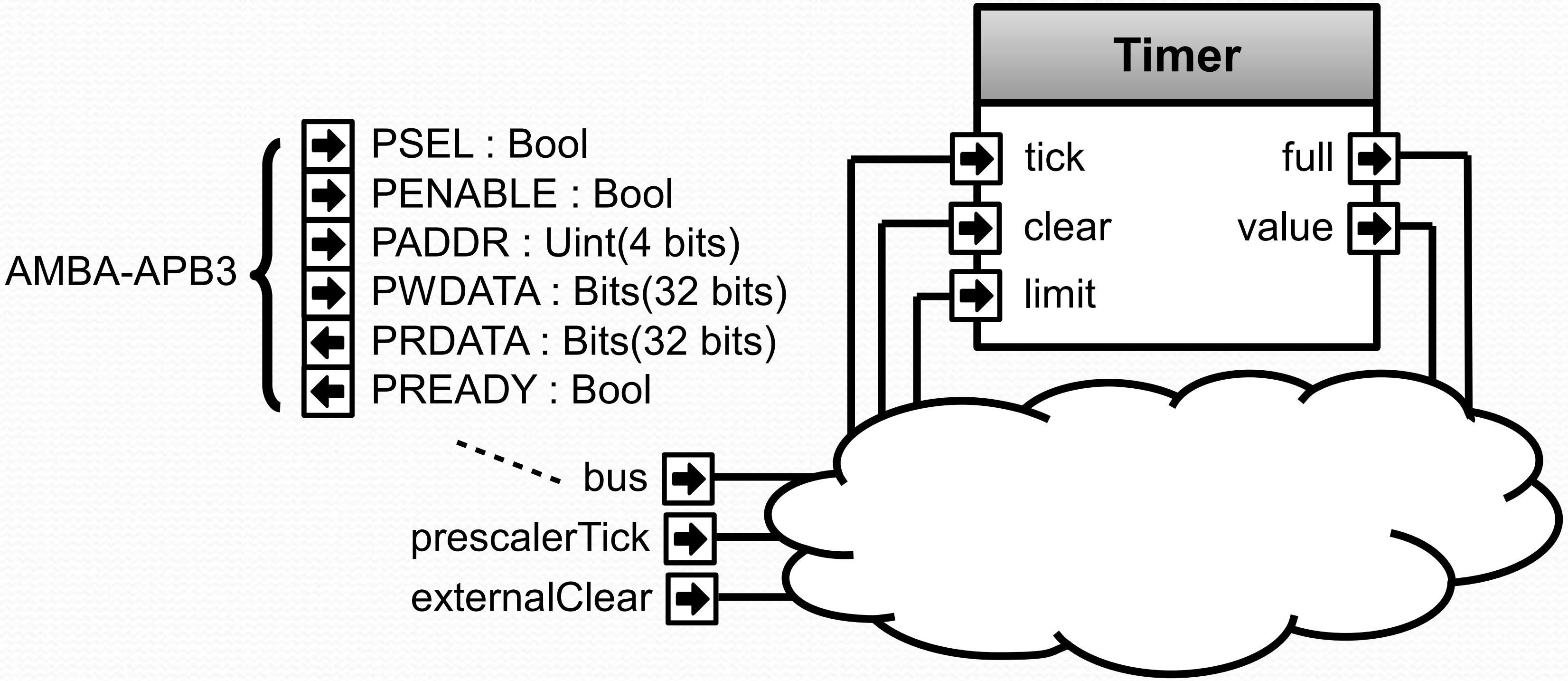
```
val fsm = new StateMachine{
    val stateA = new State with EntryPoint
    val stateB = new State
    val stateC = new State

    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))
}
```

Imagine a simple timer



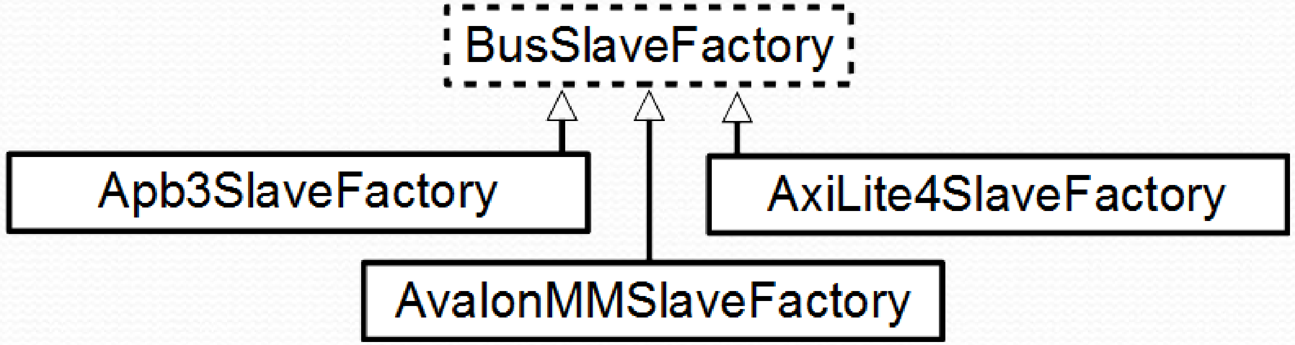
```
case class Timer(width : Int) extends Component{
  val io = new Bundle{
    val tick      = in Bool
    val clear     = in Bool
    val limit     = in UInt(width bits)

    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.value := counter
}
```

Imagine you want to connect it



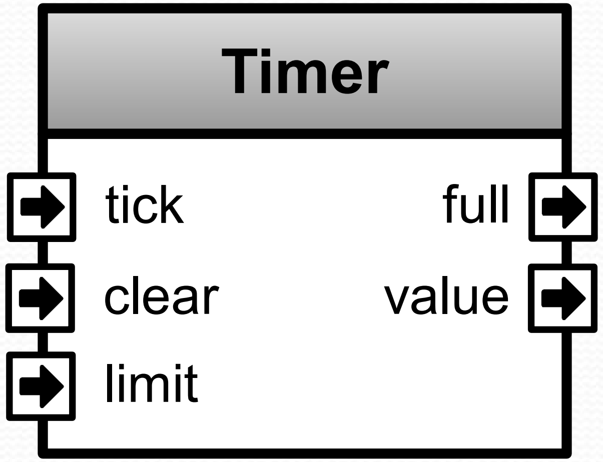
Bus Slave Factory

- OOP meet Hardware description !
- BusSlaveFactory implementations are able to create register banks by using an abstract way. Let’s use it to bind our Timer !



Let's define a function !

```
case class Timer(width : Int) extends Component{
  val io = new Bundle {
    // ...
     def driveFrom(busCtrl : BusSlaveFactory,baseAddress : BigInt)
                  (ticks : Seq[Bool],clears : Seq[Bool]) = new Area {
       clear := False

       //Address 0 => read/write limit (+ auto clear)
       busCtrl.driveAndRead(limit,baseAddress + 0)
       clear.setWhen(busCtrl.isWriting(baseAddress + 0))

       //Address 4 => read timer value / write => clear timer value
       busCtrl.read(value,baseAddress + 4)
       clear.setWhen(busCtrl.isWriting(baseAddress + 4))

        //Address 8 => clear/tick masks + bus
       // ...
     }
  }
  // ...
}
```

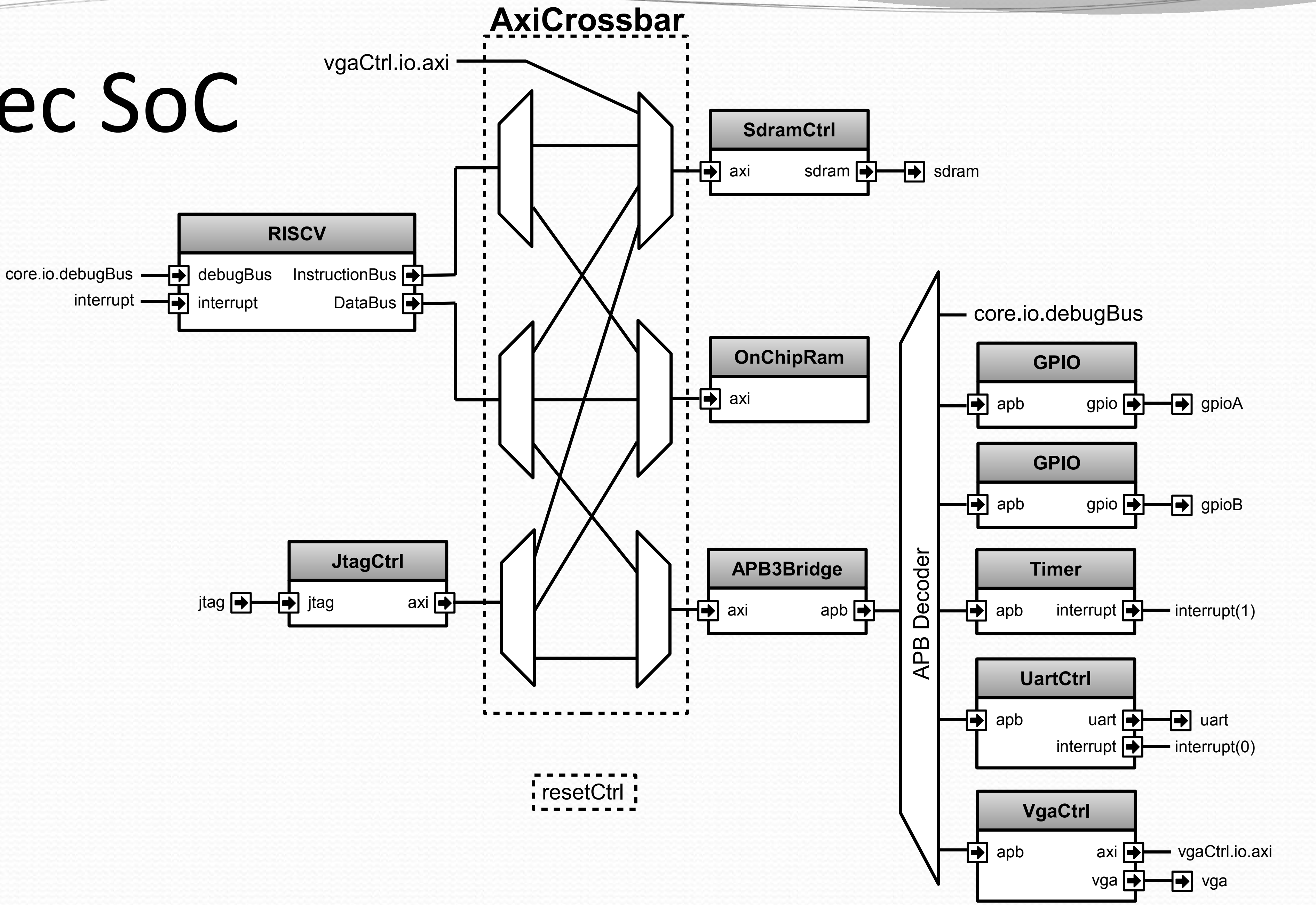


Let's use it :

```
val apb = Apb3(addressWidth = 8, dataWidth = 32)
val external = new Bundle{
   val clear,tick = Bool
}

val prescaler       = Prescaler(16)
val timerA          = Timer(32)
val timerB,timerC   = Timer(16)

val busCtrl = Apb3SlaveFactory(apb)
```

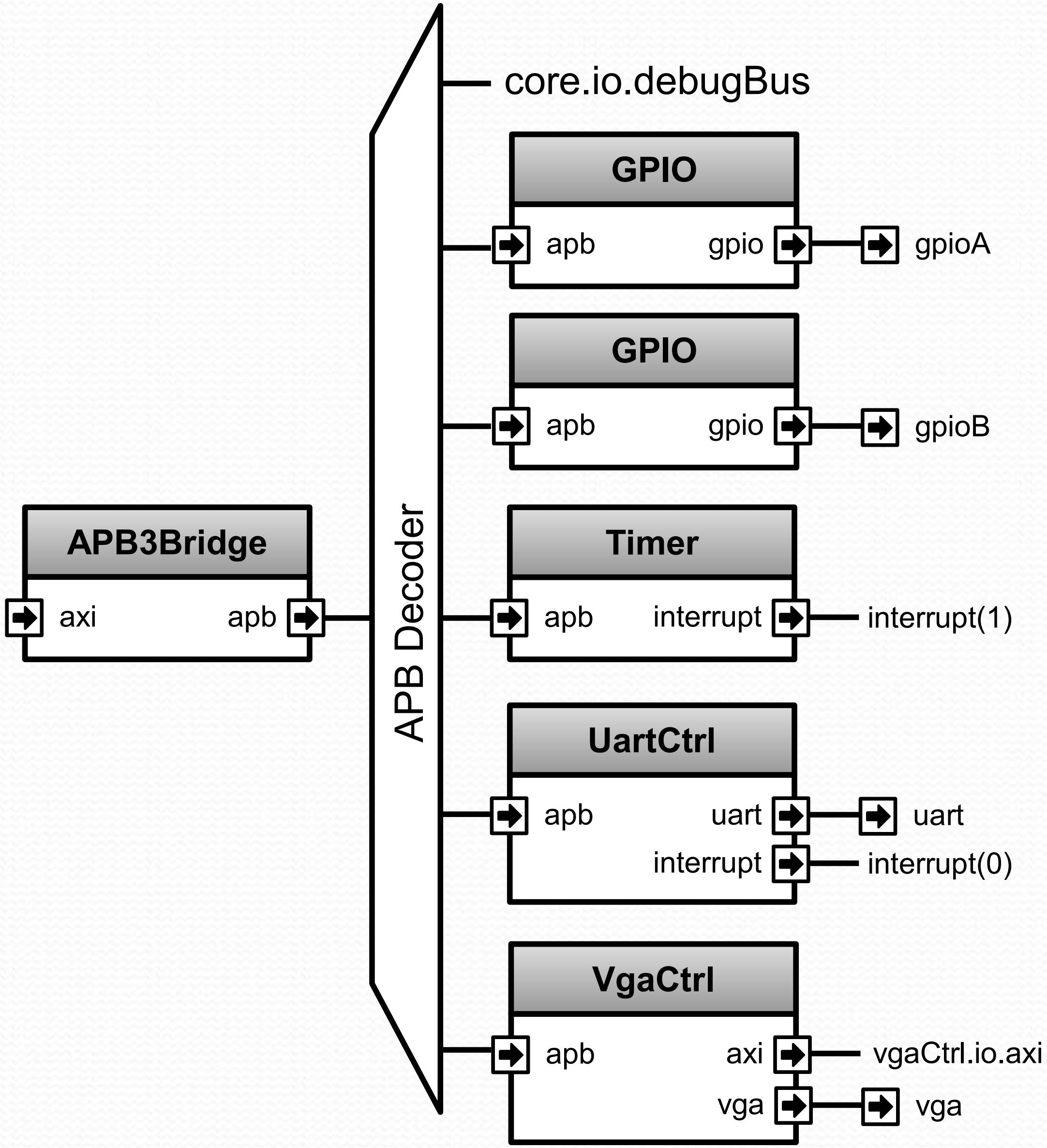
```
val prescalerBridge = prescaler.io.driveFrom(busCtrl,0x00)

val timerABridge = timerA.io.driveFrom(busCtrl,0x40)(
   ticks  = List(True, prescaler.io.overflow),
   clears = List(timerA.io.full)
)

val timerBBridge = timerB.io.driveFrom(busCtrl,0x50)(
   ticks  = List(True, prescaler.io.overflow, external.tick),
   clears = List(timerB.io.full, external.clear)
)

val timerCBridge = timerC.io.driveFrom(busCtrl,0x60)(
   ticks  = List(True, prescaler.io.overflow, external.tick),
   clears = List(timerC.io.full, external.clear)
)
```

Pinsec SoC


AxiCrossbar
vgaCtrl.io.axi
SdramCtrl
axi
sdram
sdram
RISCV
core.io.debugBus
interrupt
debugBus
interrupt
InstructionBus
DataBus
OnChipRam
axi
core.io.debugBus
GPIO
apb
gpio
gpioA
GPIO
apb
gpio
gpioB
APB3Bridge
axi
apb
APB Decoder
Timer
apb
interrupt
interrupt(1)
JtagCtrl
jtag
jtag
axi
UartCtrl
apb
uart
uart
interrupt
interrupt(0)
resetCtrl
VgaCtrl
apb
axi
vgaCtrl.io.axi
vga
vga

Peripheral side

```
val apbBridge = Axi4ToApb3Bridge(
  addressWidth = 20,
  dataWidth    = 32,
  idWidth      = 4
)

val apbDecoder = Apb3Decoder(
  master = apbBridge.io.apb,
  slaves = List(
    gpioACtrl.io.apb    -> (0x00000, 4 kB),
    gpioBCtrl.io.apb    -> (0x01000, 4 kB),
    uartCtrl.io.apb     -> (0x10000, 4 kB),
    timerCtrl.io.apb    -> (0x20000, 4 kB),
    vgaCtrl.io.apb      -> (0x30000, 4 kB),
    core.io.debugBus  -> (0xF0000, 4 kB)
  )
)
```



AXI4 side (OOP builder pattern)

```
val  axiCrossbar  =  Axi4CrossbarFactory()

axiCrossbar.addSlaves(
   ram.io.axi             ->  (0x00000000L,   onChipRamSize),
   sdramCtrl.io.axi   ->  (0x40000000L,   sdramLayout.capacity),
   apbBridge.io.axi  ->  (0xF0000000L,   1  MB)
)

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)
)

axiCrossbar.build()
```

About Scala

- Free Scala IDE (eclipse, intelij)
  - Highlight syntax error
  - Renaming flexibility
  - Intelligent auto completion
  - Code structure overview
  - Navigation tools
- Emacs plugin
- Allow you to extend the language
- Provide many libraries

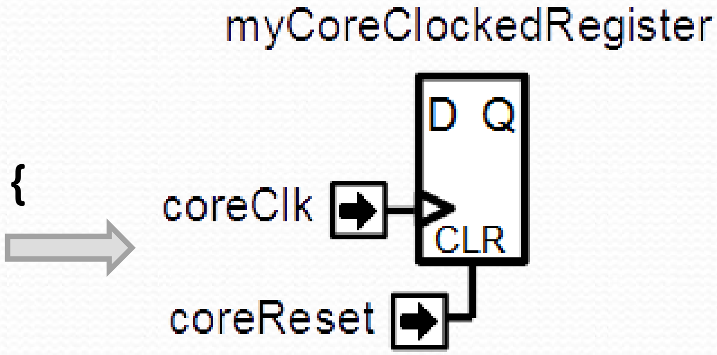
About SpinalHDL project

- 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

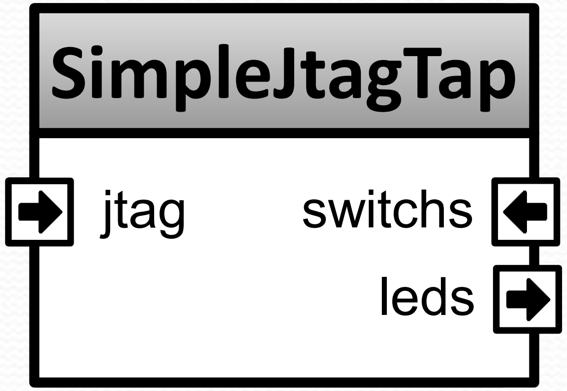
ClockDomains

```
val coreClk      = Bool
val coreReset  = Bool

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

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



JTAG slave (tap)

```
class SimpleJtagTap extends Component {
  val io = new Bundle {
    val jtag       = slave(Jtag())
    val switchs  = in   Bits(8 bit)
    val leds       = out Bits(8 bit)
  }

  val tap = new JtagTap(io.jtag, 8)
  val idcodeArea    = tap.idcode(B"x87654321")(instructionId=4)
  val switchsArea  = tap.read (io.switchs)         (instructionId=5)
  val ledsArea       = tap.write(io.leds)             (instructionId=6)
}
```


SimpleJtagTap
jtag
switchs
leds