SpinalHDL

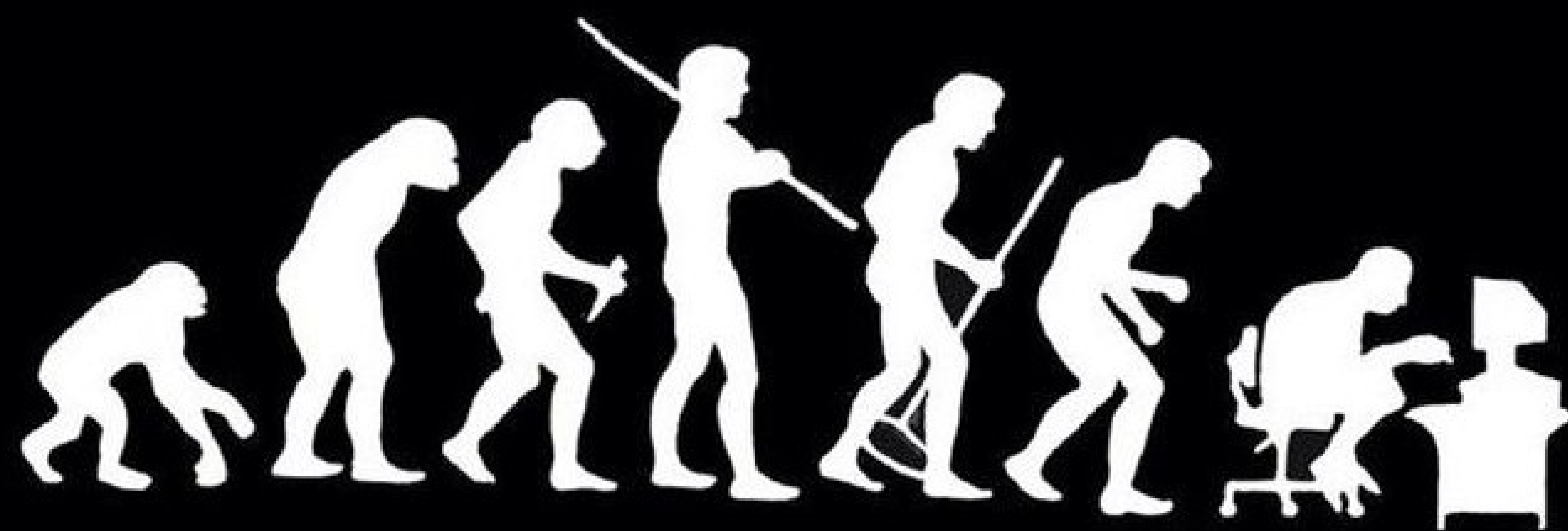
An alternative hardware description language

Summary

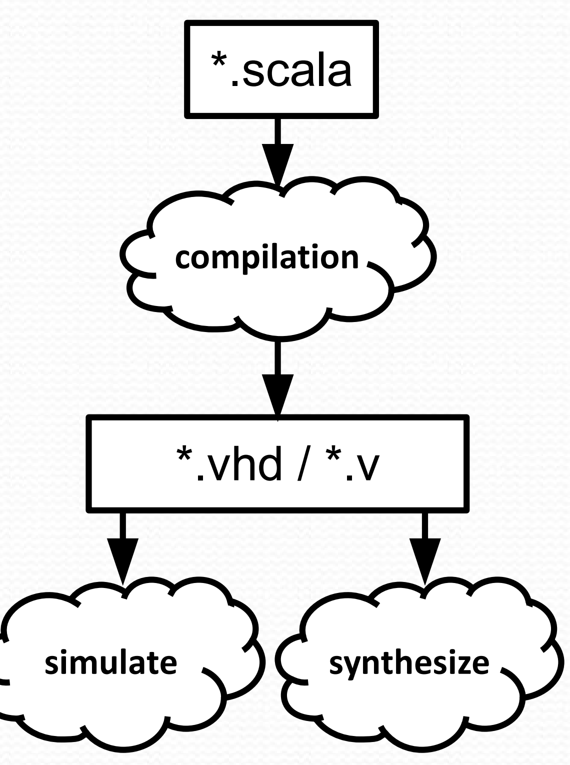
- SpinalHDL introduction
- Differences with VHDL/Verilog
- Hardware description examples
  - By using abstractions
  - By using software engineering
- Disclamer
  - This talk will only be about synthesizable hardware

Context :

- VHDL-2002 and Verilog-2005 are bottlenecks by many aspects
- VHDL-2008 and SV will not all save us (EDA support, features ...)



SpinalHDL introduction

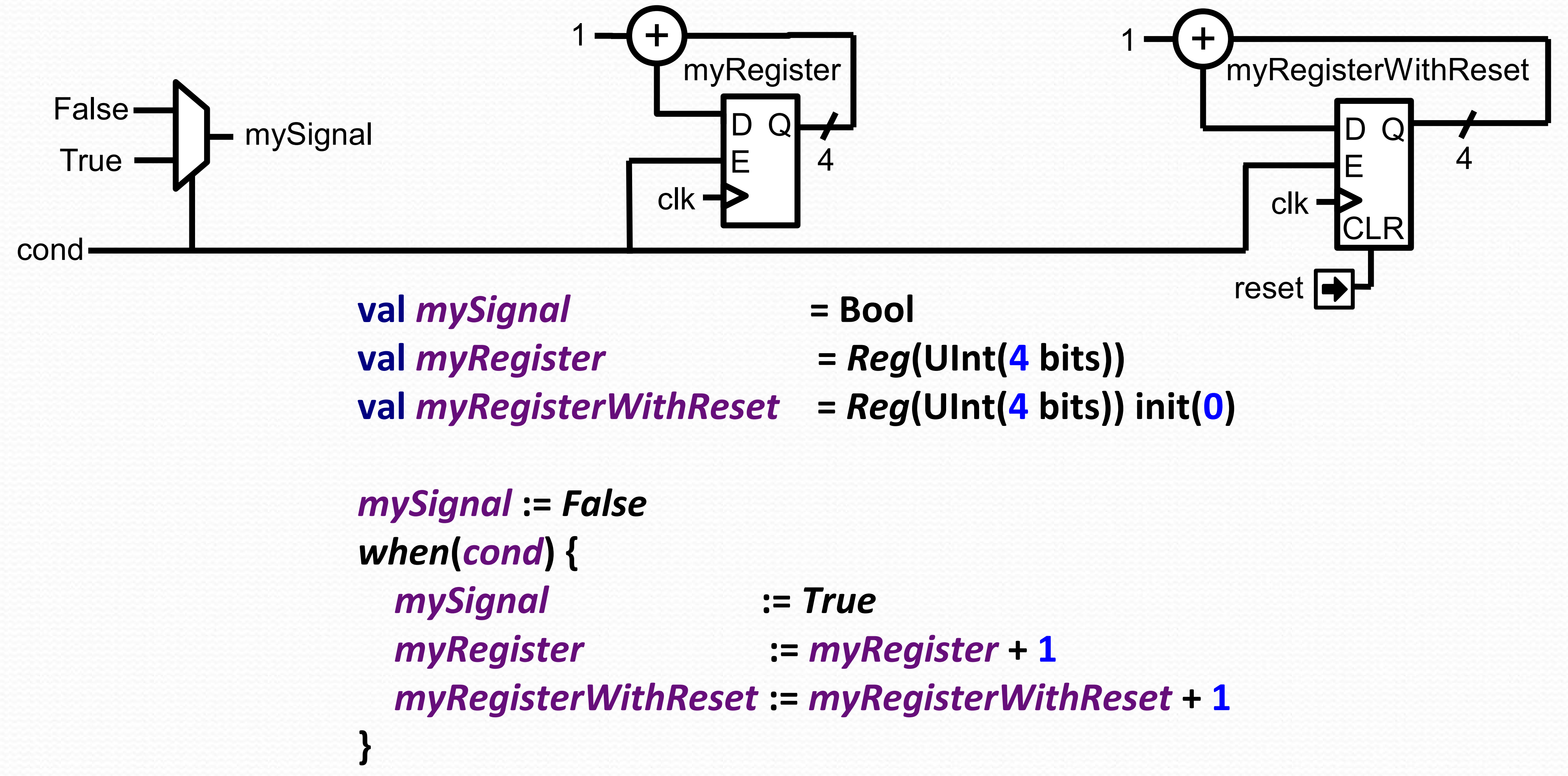


- Open source, started in december 2014
- Focus on RTL description
- Thinked to be interoperable with existing tools
  - It generates simples VHDL/Verilog files (as an output netlist)
  - It can integrate VHDL/Verilog IP as blackbox
- Abstraction level :
  - An RTL approach as VHDL/Verilog
  - If you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

Some points about SpinalHDL

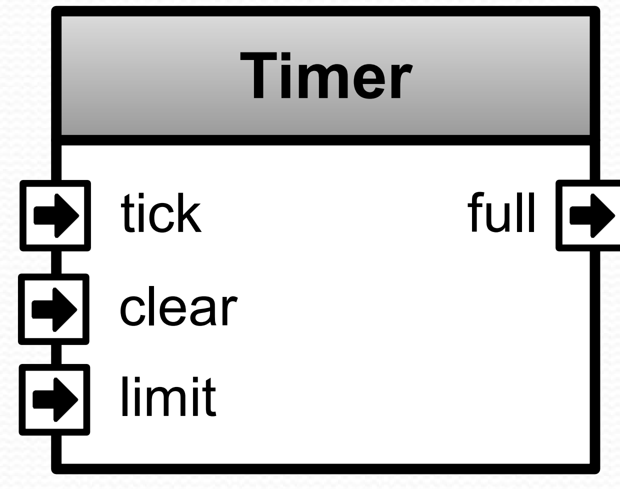
- There is no logic overhead in the generated code. (I swear !)
- The component hierarchy and all names are preserved during the VHDL/Verilog generation. (Good for simulations)
- It is an language hosted on the top of Scala ! (but don't be afraid)

By using an dedicated syntax (SpinalHDL)



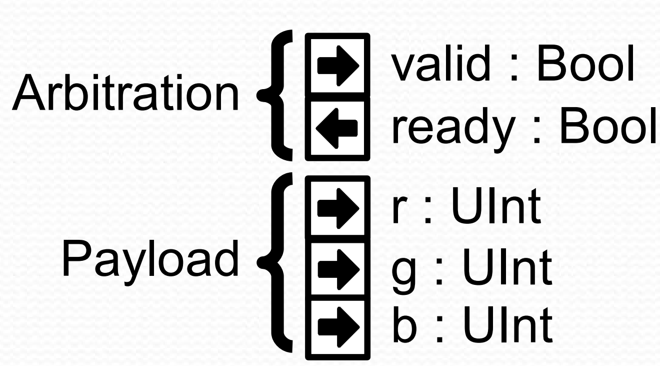

```
val mySignal              = Bool
val myRegister            = Reg(UInt(4 bits))
val myRegisterWithReset   = Reg(UInt(4 bits)) init(0)

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

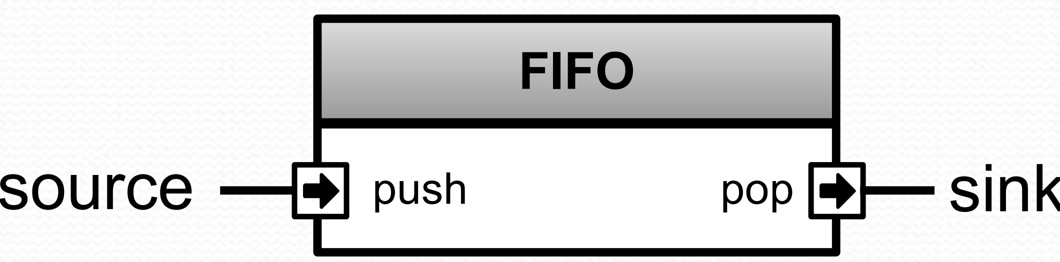
A timer implementation




```
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 counter = Reg(UInt(width bits))
  when(io.tick && !io.full){
    counter := counter + 1
  }
  when(io.clear){
    counter := 0
  }

  io.full := counter === io.limit
}
```

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





In standard VHDL-2002

FIFO

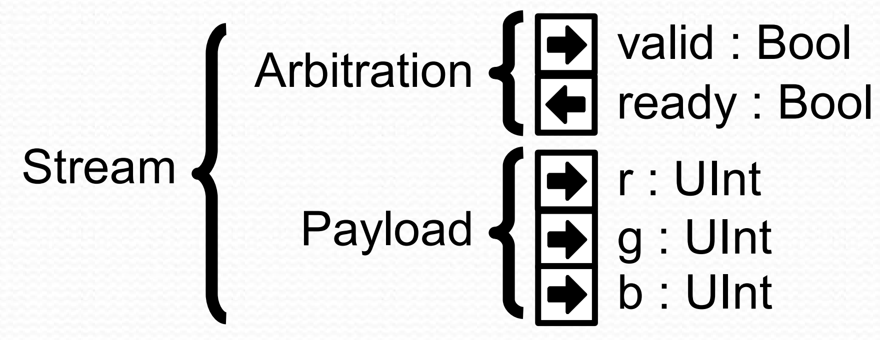
source push pop sink

```
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
```

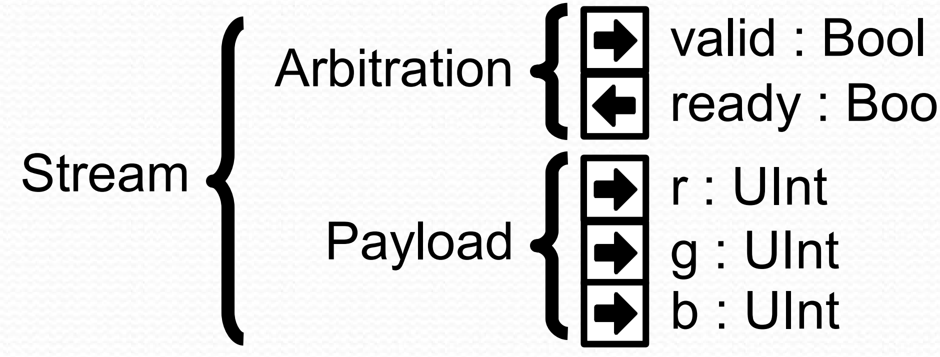


About Stream

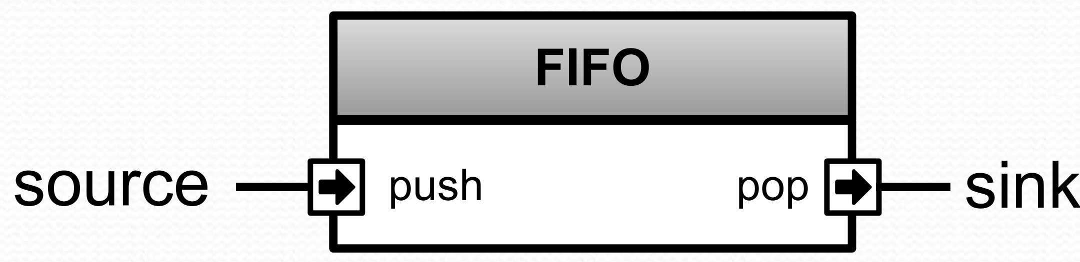
```
case class Stream[T <: Data](payloadType : HardType[T]) extends Bundle {
  val valid     = Bool
  val ready     = Bool
  val payload = payloadType()

  def >>(sink: Stream[T]): Unit ={
    sink.valid     := this.valid
    this.ready     := sink.ready
    sink.payload := this.payload
  }

  def queue(size: Int): Stream[T] = {
    val fifo = new StreamFifo(payloadType, size)
    this >> fifo.io.push
    return fifo.io.pop
  }
}
```

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
```



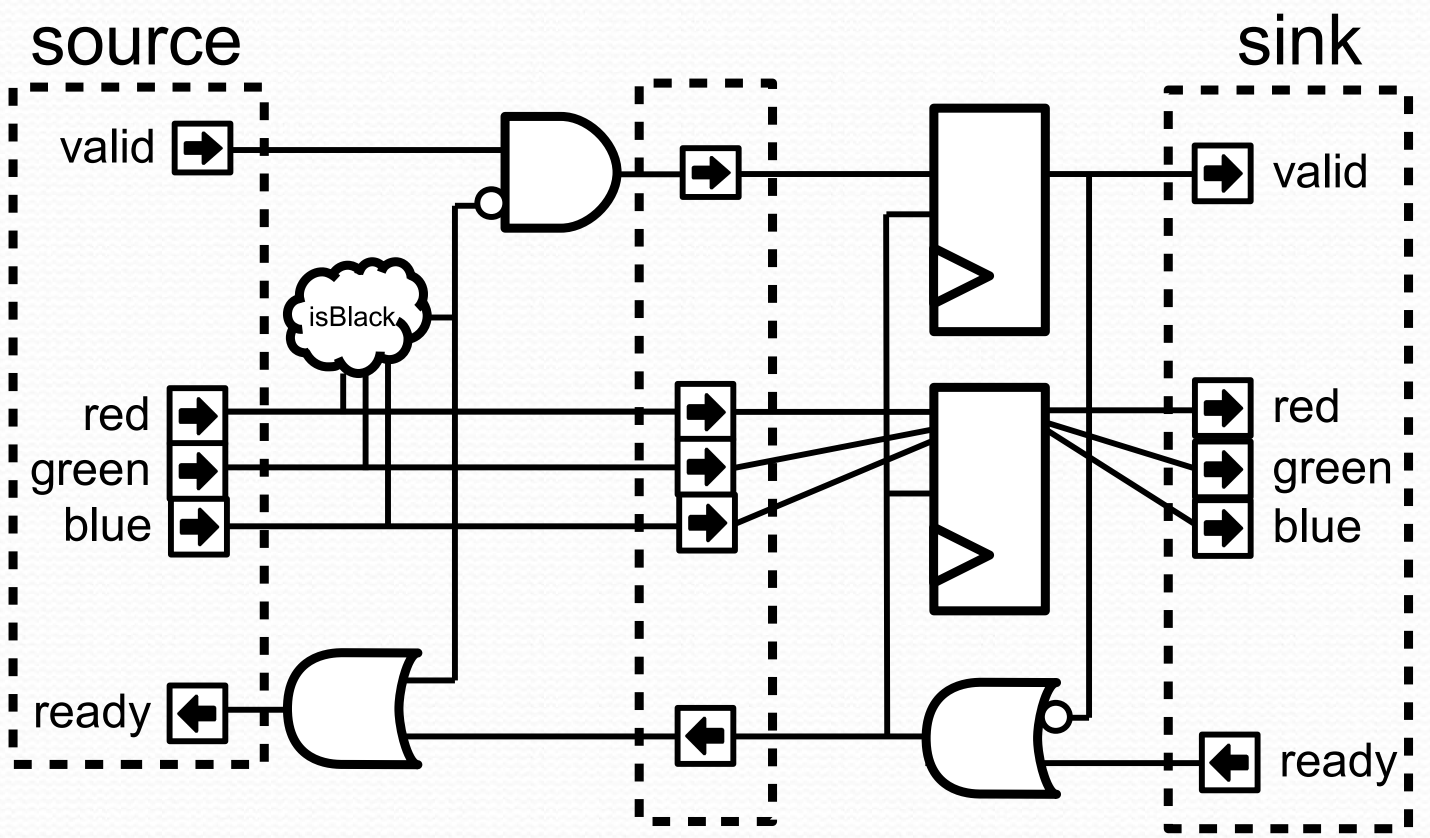

```
val source, sink = Stream(RGB(5,6,5))

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



SpinalHDL => 2 lines
VHDL => 29 lines

Abstract arbitration

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

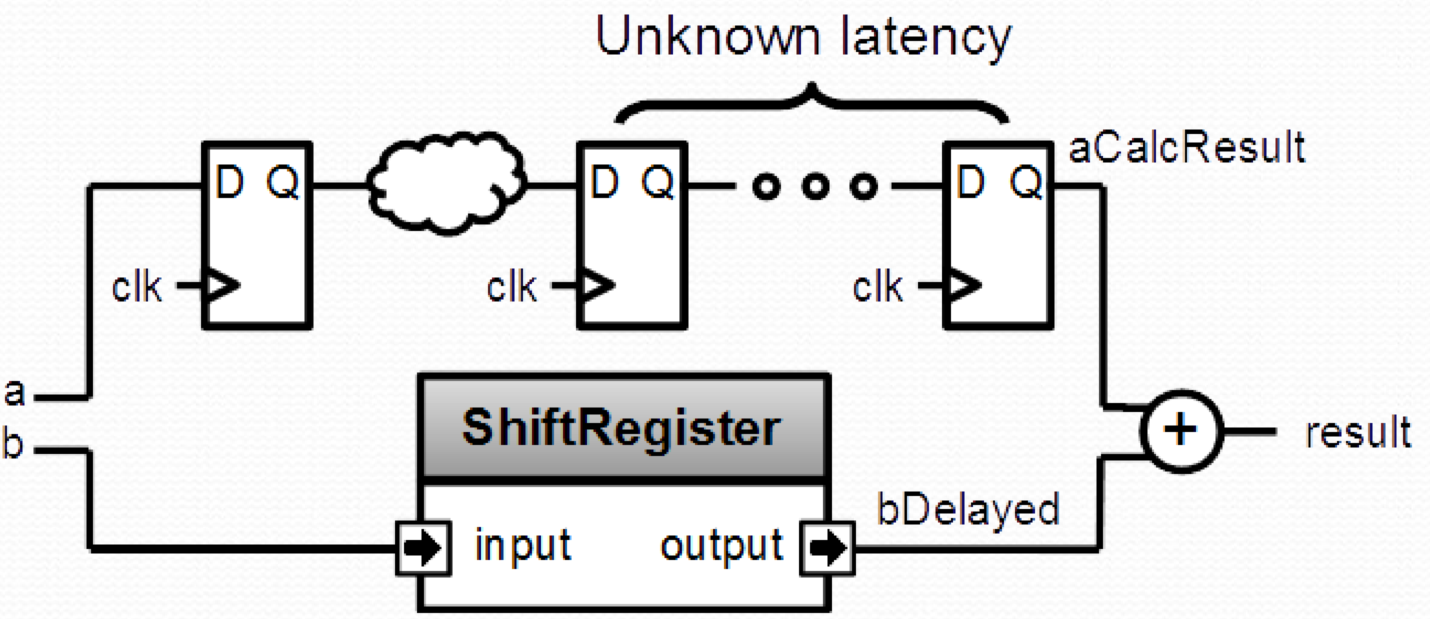


Functional programming

```
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

OnEntry =>
  counter := 0

whenIsActive =>
  counter := counter + 1

onExit =>
  io.result := True

stateA

stateB

counter === 4

stateC

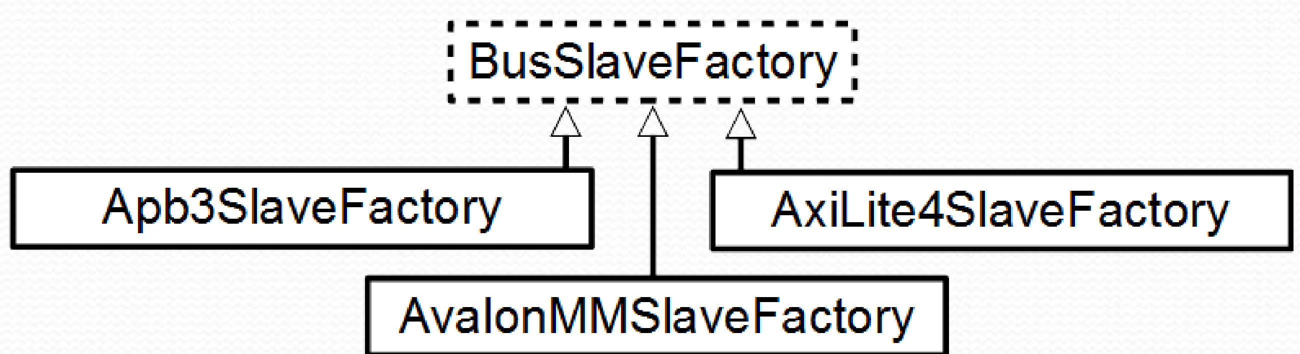
```
val fsm = new StateMachine{
  val stateA = new State with EntryPoint
  val stateB = new State
  val stateC = new State

  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))
}
```

Abstract bus mapping

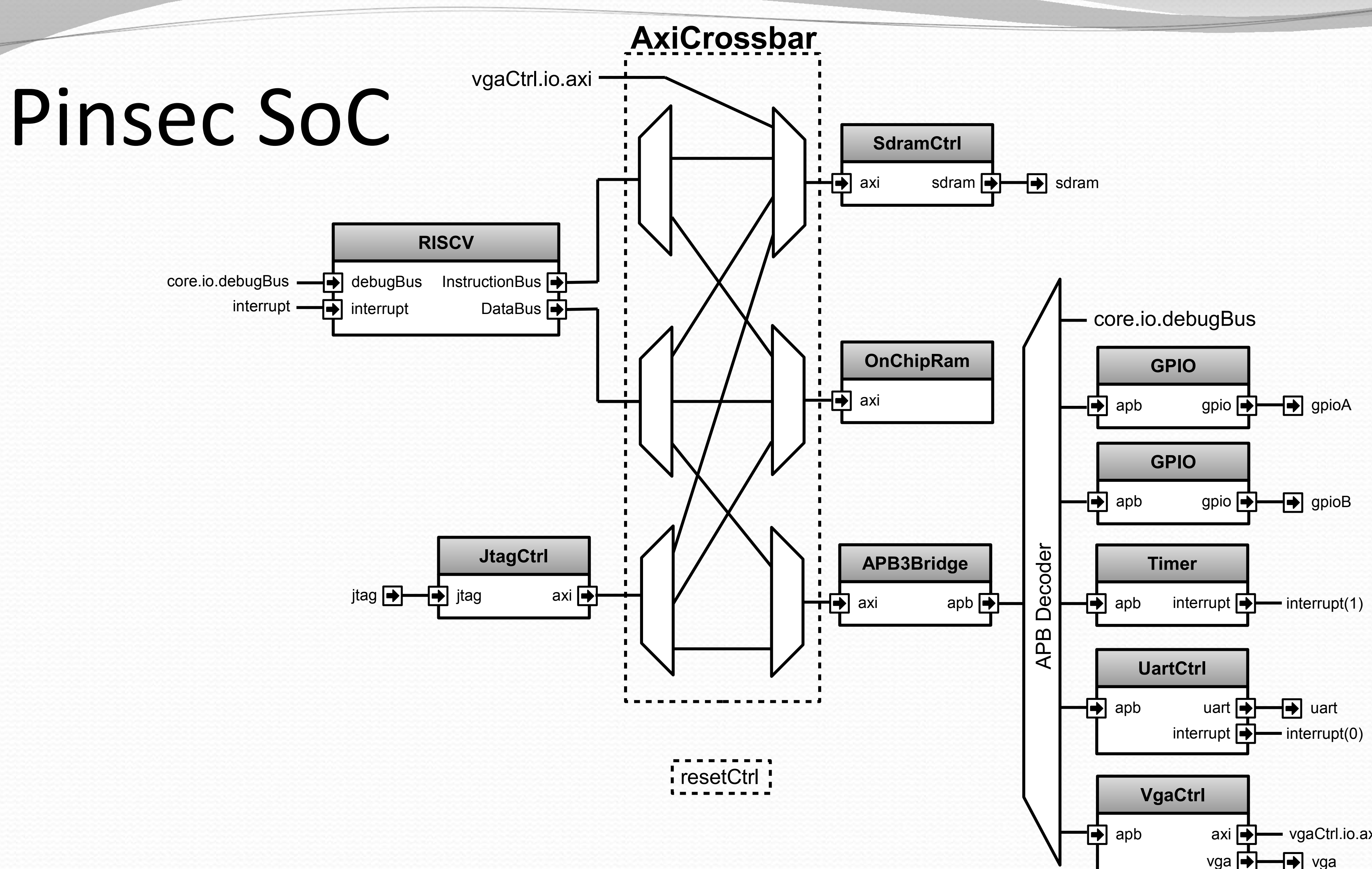
```
//Create a new AxiLite4 bus
val bus = AxiLite4(addressWidth =  12, dataWidth = 32)

//Create the factory which is able to create some bridging logic between the bus and some hardware
val factory = new AxiLite4SlaveFactory(bus)

//Create 'a' and 'b' as write only register
val a = factory.createWriteOnly(UInt(32 bits), address = 0)
val b = factory.createWriteOnly(UInt(32 bits), address = 4)

//Do some calculation
val result = a * b

//Make 'result' readable by the bus
factory.read(result(31 downto 0), address = 8)
```

SlaveFactory

- AxiLite4SlaveFactory is only a part of something bigger and more abstract.

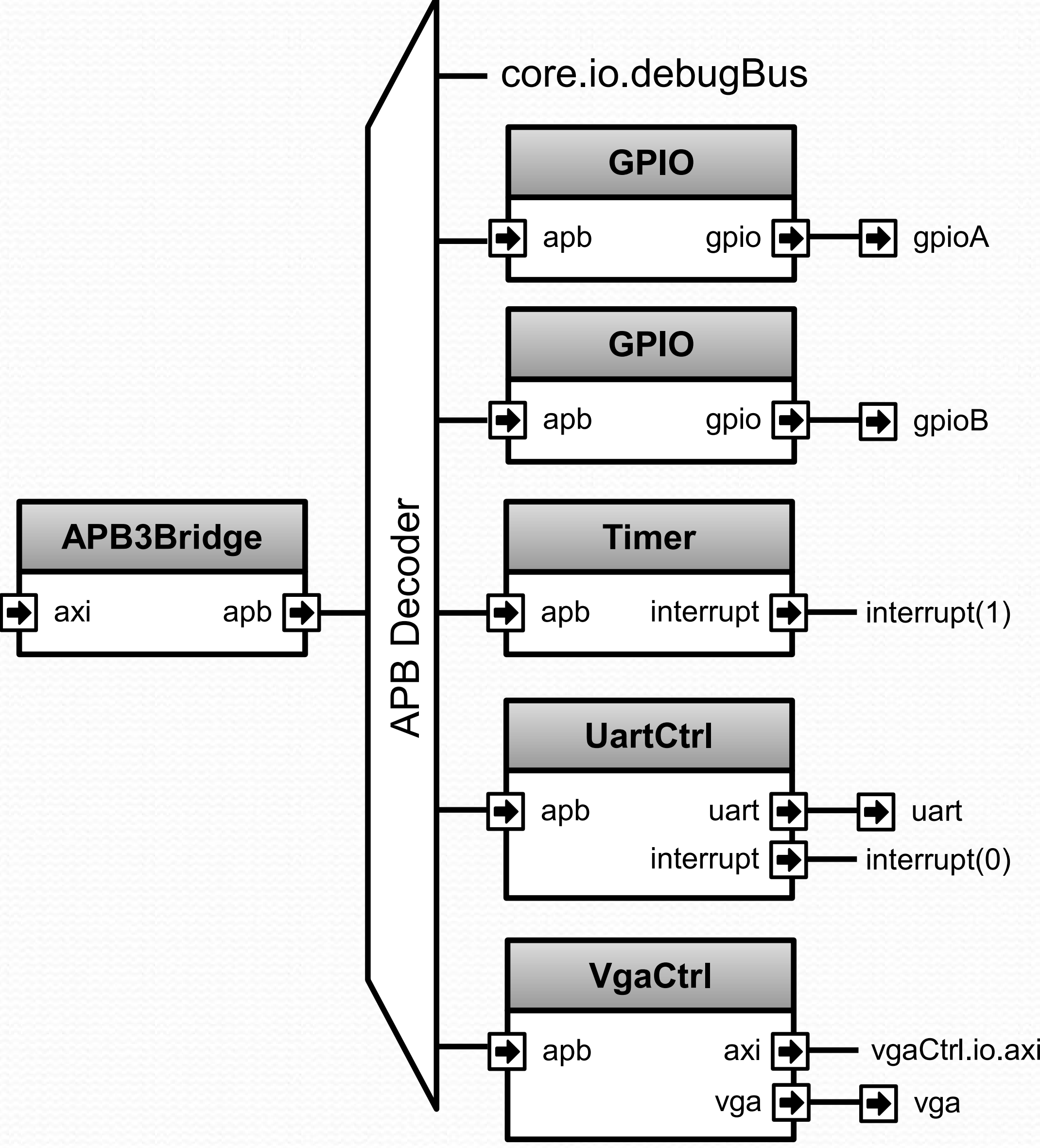


Pinsec SoC

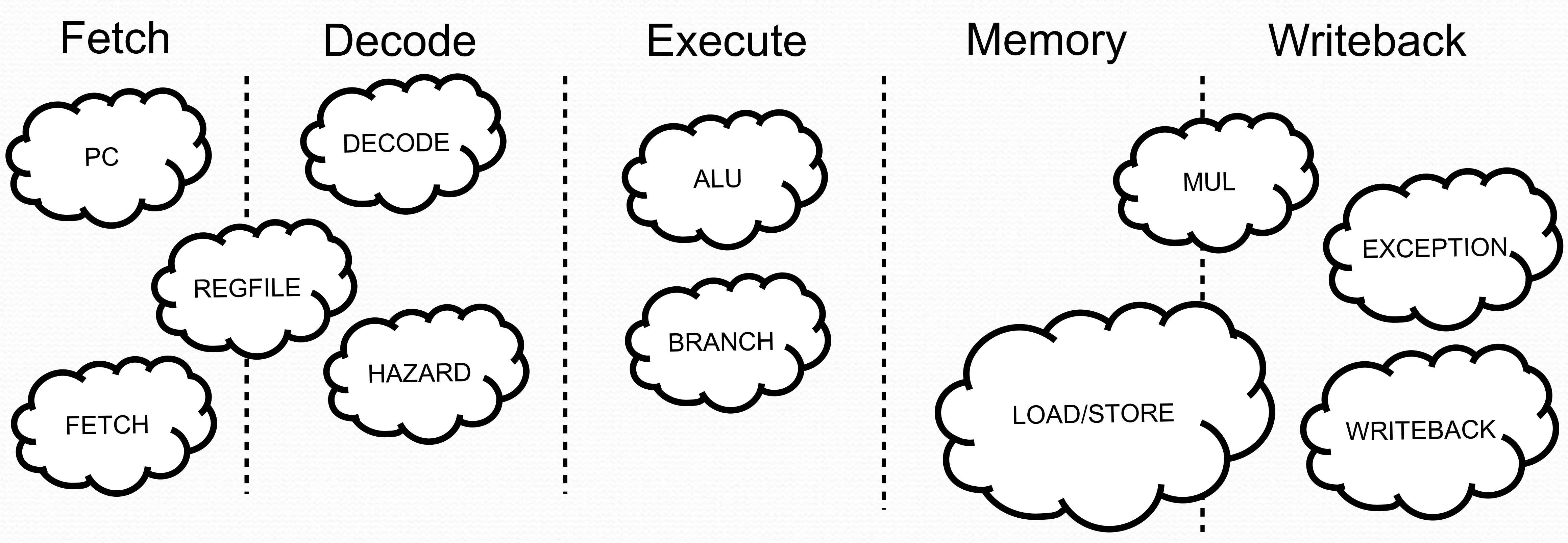


Peripheral side

```
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)
  )
)
```



CPU pipelined over 5 stages



Modular CPU framework

```
val plugins = List(
  new PcManagerSimplePlugin(resetVector = 0x00000000l),
  new IBusSimplePlugin(catchAccessFault = false),
  new DBusSimplePlugin(catchAccessFault = false),
  new DecoderSimplePlugin(catchIllegalInstruction = false),
  new RegFilePlugin,
  new IntAluPlugin,
  new FullBarrielShifterPlugin,
  new HazardSimplePlugin(
    bypassExecute     = false,
    bypassMemory      = false,
    bypassWriteBack   = false
  ),
  new MulPlugin,
  new DivPlugin,
  new MachineCsr(...),
  new BranchPlugin(
    catchAddressMisaligned = false,
    prediction = DYNAMIC
  )
)

val cpu = new Cpu5Stage(plugins)
```

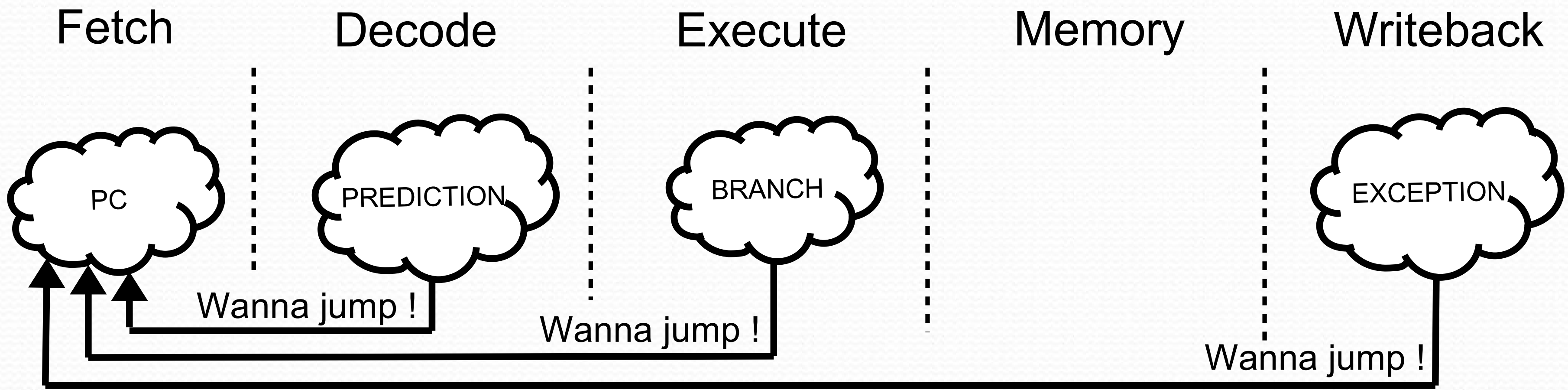
CPU framework - Connections

```
//Global definition of the Programm Counter concept
object PC extends Stageable(UInt(32 bits))

//Somewere in the PcManager plugin
fetch.insert(PC) := X

//Somewere in the MachineCsr plugin
Y := writeBack.input(PC)
```

Fetch Decode Execute Memory Writeback

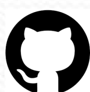
PC X | PC | PC | PC | PC | EXCEPTION Y

CPU framework - Connections



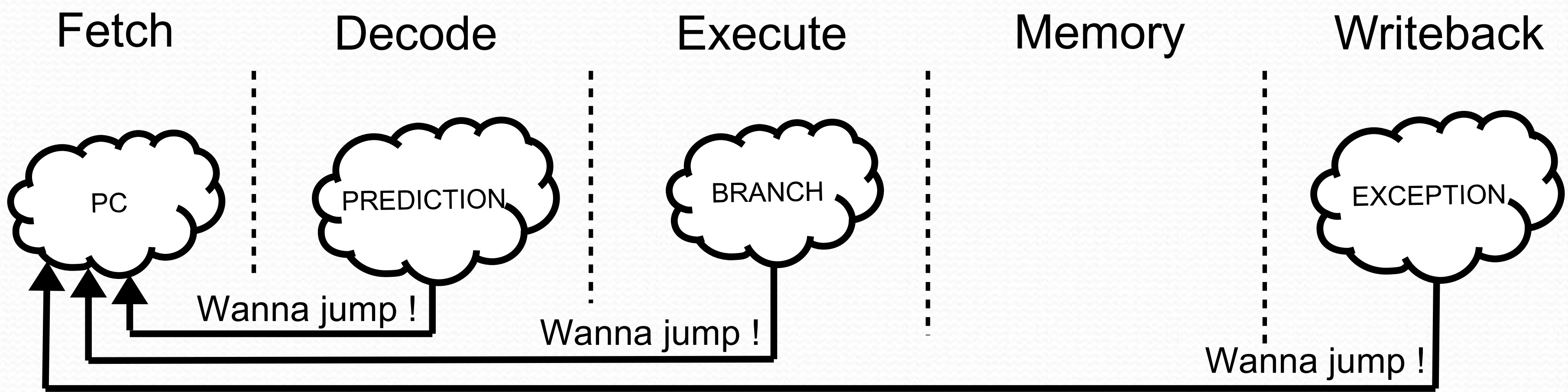

```
//Somewhere in the Branch plugin
val jumpInterface  = pcPlugin.createJumpInterface(stage = execute)

//Later in the branch plugin
jumpInterface.valid := ???
jumpInterface.payload := execute.input(PC) + execute.input(INSTRUCTION)(31 downto 20)
```

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 / removed slides

CPU framework




```
trait JumpService{
  def createJumpInterface(stage : Stage) : Flow[UInt]
}
abstract class PcManagerPlugin() extends JumpService

//Somewhere in the Branch plugin
val jumpService = pipeline.service(classOf[JumpService]) //Will get the PcManagerPlugin plugin reference dynamicaly
val jumpInterfaceForPredictions = jumpService.createJumpInterface(stage = decode)
val jumpInterfaceForBranches     = jumpService.createJumpInterface(stage = execute)

//Later in the branch plugin
jumpInterfaceForBranches.valid := ???
jumpInterfaceForBranches.payload := execute.input(PC) + execute.input(INSTRUCTION)(31 downto 20)
```

CPU infrastructure

- Nearly empty toplevel
- Plugin system to add logic
- Plugins interconnection/communication :
  - Shared signal accessed by their key
  - Automatic pipelining between stages
  - Abstract software interface
    - PC jump
    - Decoding services
    - Exception

```
val cpuConfig = VexRiscvConfig(pcWidth = 32)
cpuConfig.plugins ++= List(
  new PcManagerSimplePlugin(resetVector = 0x00000000l),
  new IBusSimplePlugin(catchAccessFault = false),
  new DBusSimplePlugin(catchAccessFault = false),
  new DecoderSimplePlugin(catchIllegalInstruction = false),
  new RegFilePlugin,
  new IntAluPlugin,
  new FullBarrielShifterPlugin,
  new HazardSimplePlugin(
    bypassExecute      = false,
    bypassMemory       = false,
    bypassWriteBack    = false
  ),
  new MulPlugin,
  new DivPlugin,
  new MachineCsr(...),
  new BranchPlugin(
    catchAddressMisaligned = false,
    prediction = DYNAMIC
  )
)
val cpu = new VexRiscv(cpuConfig)
```