SpinalHDL

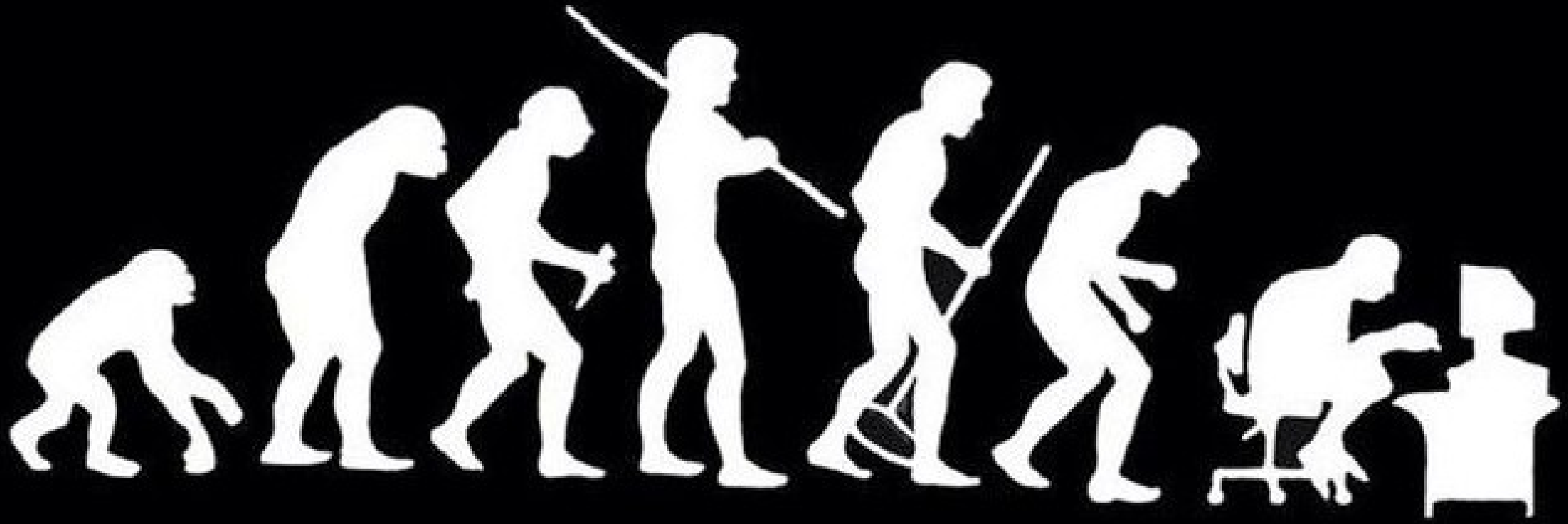
Taste

Summary

- SpinalHDL introduction
- Differences with VHDL/Verilog
- Hardware description examples

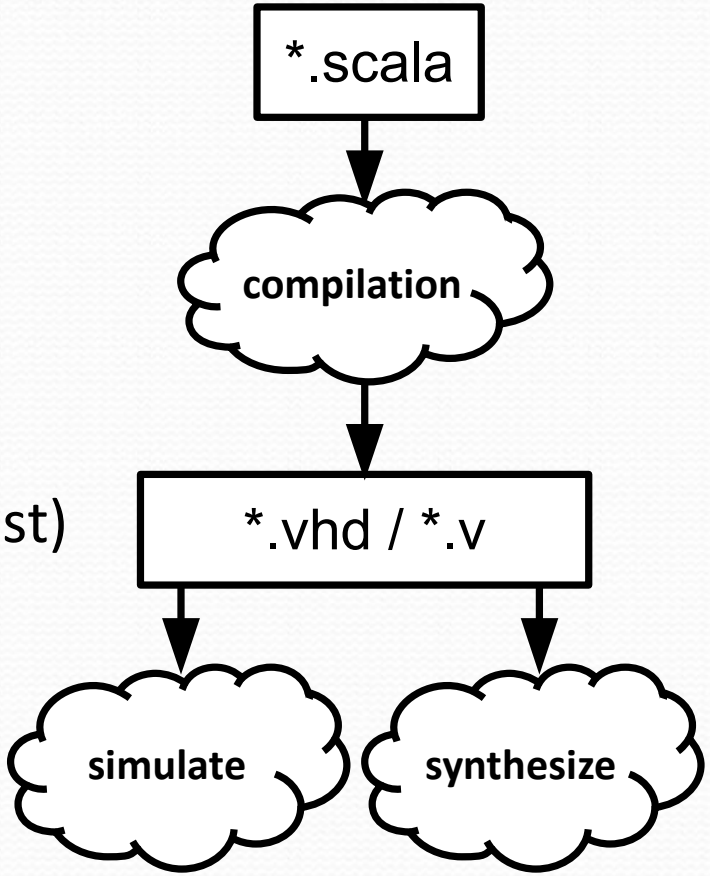
Context :

- Elaborations features of VHDL/SV are realy weak
- No hope in VHDL/SV revisions
- Using event driven paradigm to infer hardware isn't that good



SpinalHDL introduction

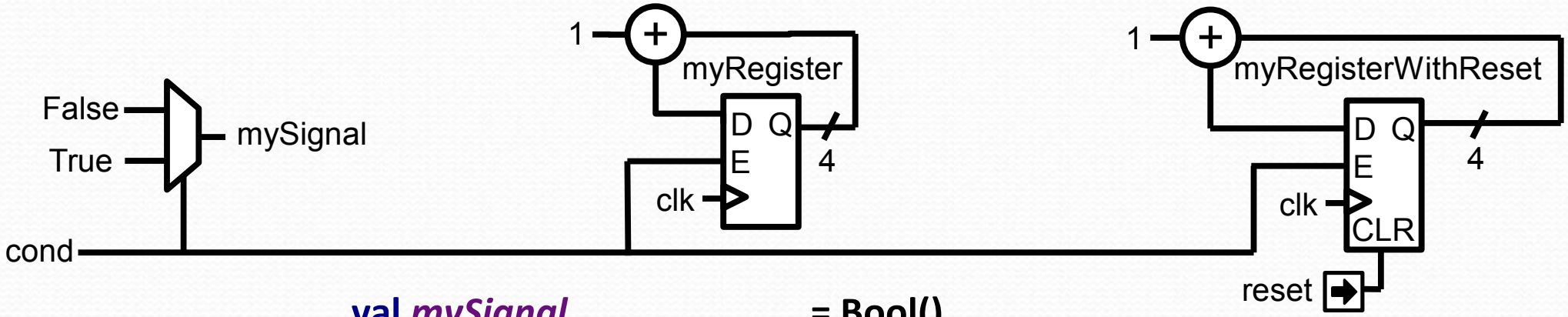
- Open source, started in December 2014
- Focus on RTL description
- Compatible with EDA tools
  - It generates simples VHDL/Verilog files (as an output netlist)
  - It can integrate VHDL/Verilog IP as blackbox
- Paradigms :
  - RTL description without behing event driven
  - Embedded into a general purpose programming language
  - General purpose programming paradigm used as an RTL elaboration tool



Notice

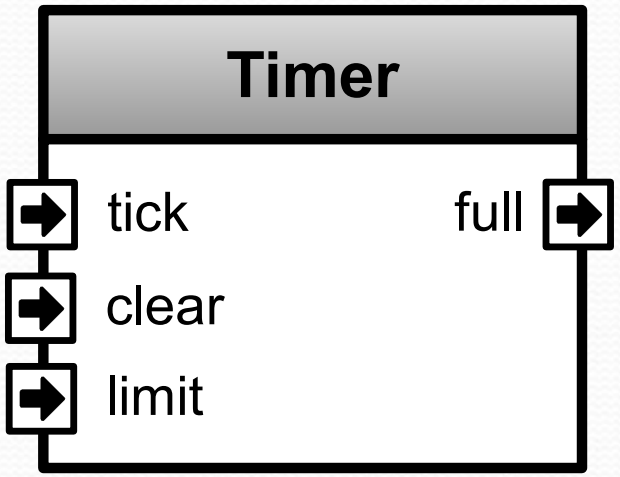
- Humans are subject to bias judgments
  - Tastes are often shaped by experencies
  - Change aversion
- Learning a language isn't a harmless process
- It's not only about the syntax, but also about features, especialy elaboration features
- Another language paradigm also mean
  - another syntax layout
  - another coding style
  - another coding guidelines

SpinalHDL basics



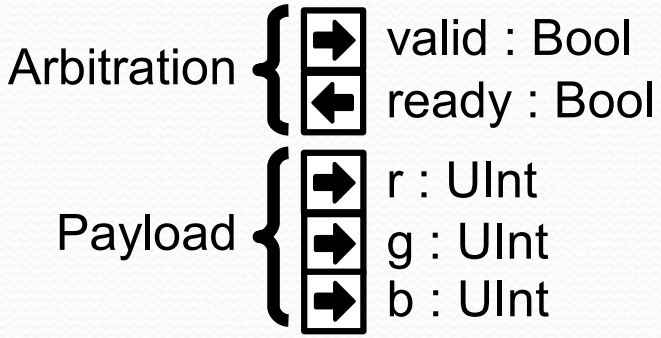

```
val mySignal                = Bool()
val myRegister              = Reg(UInt(4 bits))
val myRegisterWithReset     = Reg(UInt(4 bits)) init(0)

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

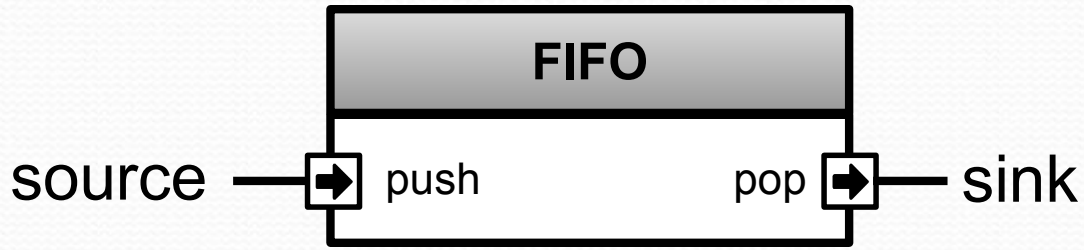
A timer implementation



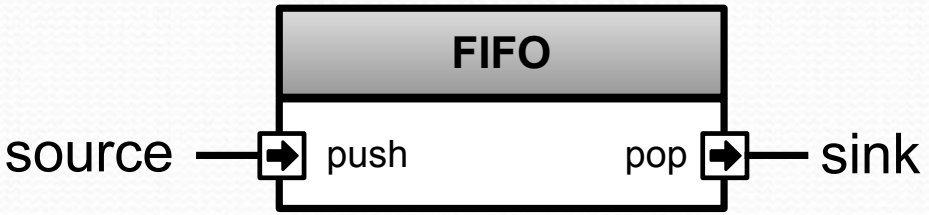

```
class Timer(width : Int) extends Component{
  val io = new Bundle{
    val tick  = in Bool
    val clear = in Bool
    val limit = in UInt(width bits)
    val full  = out Bool
  }

  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

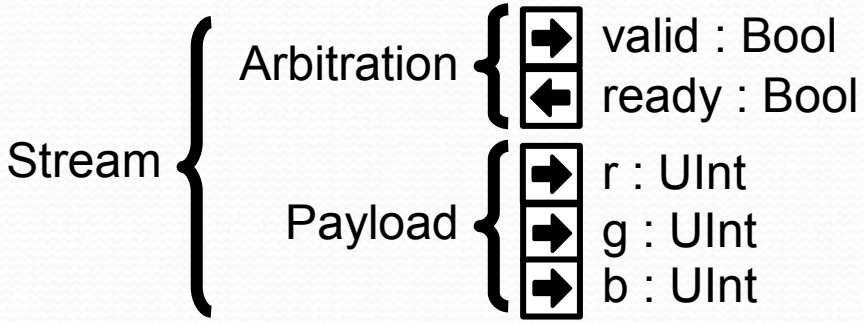


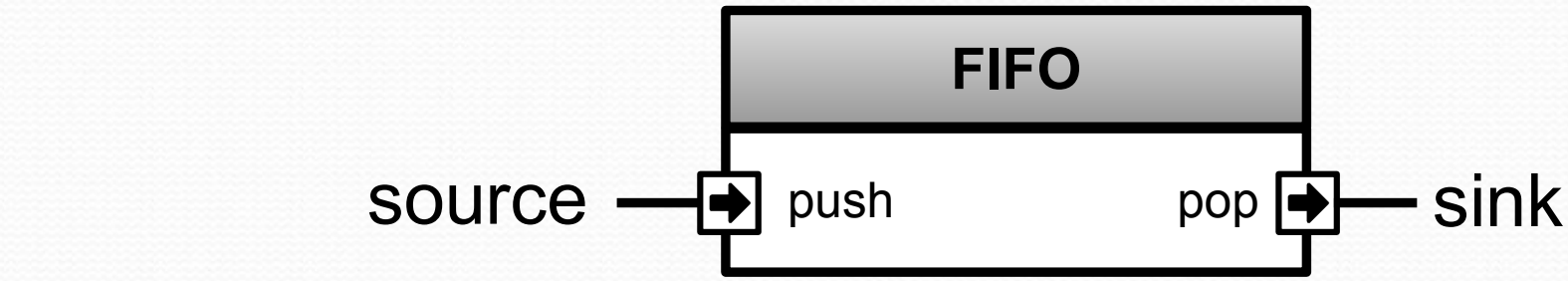

```
signal source_valid  : std_logic;
signal source_ready : std_logic;
signal source_r         : unsigned(4 downto 0);
signal source_g        : unsigned(5 downto 0);
signal source_b        : unsigned(4 downto 0);

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

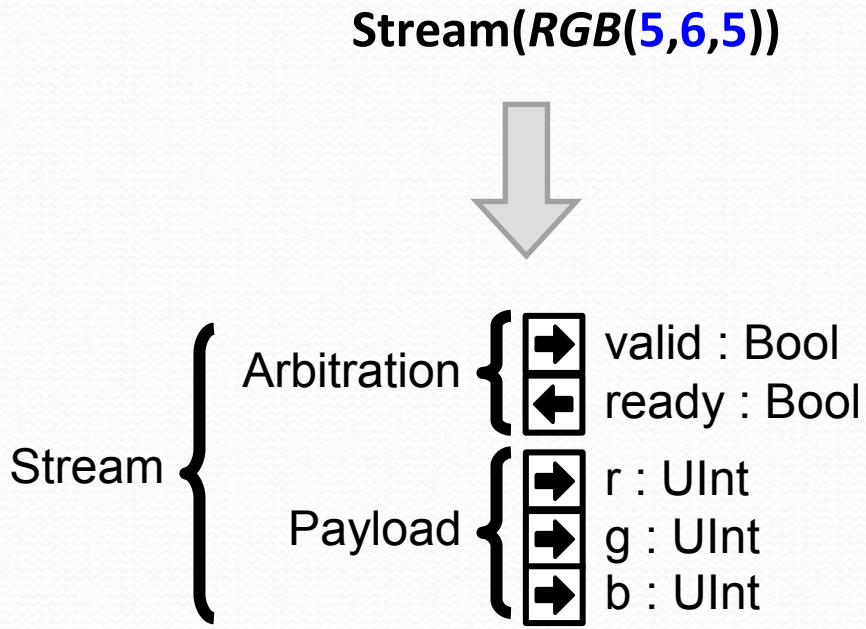
```
val source, sink = Stream(RGB(5,6,5))
val fifo = StreamFifo(
   dataType = RGB(5,6,5),
   depth      = 16
)
fifo.io.push << source
fifo.io.pop >> sink
```





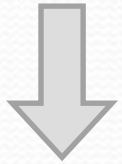
About Stream

```
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
  }
}
```



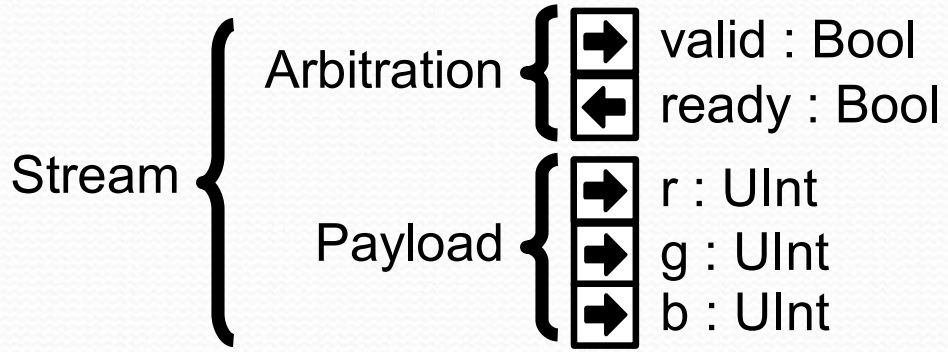
Queuing in SpinalHDL

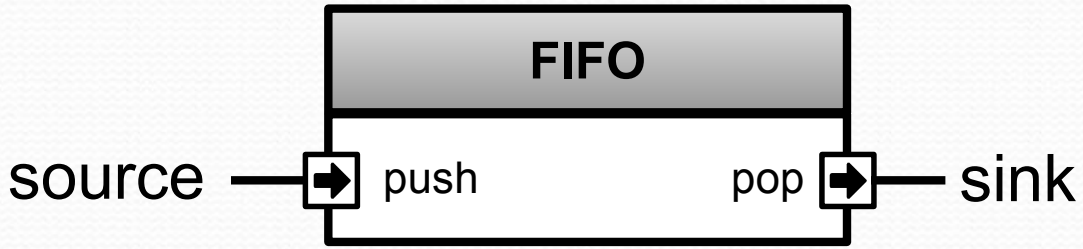
```
val source, sink = Stream(RGB(5,6,5))
val fifo = StreamFifo(
   dataType = RGB(5,6,5),
   depth = 16
)
fifo.io.push << source
fifo.io.pop >> sink
```



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

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

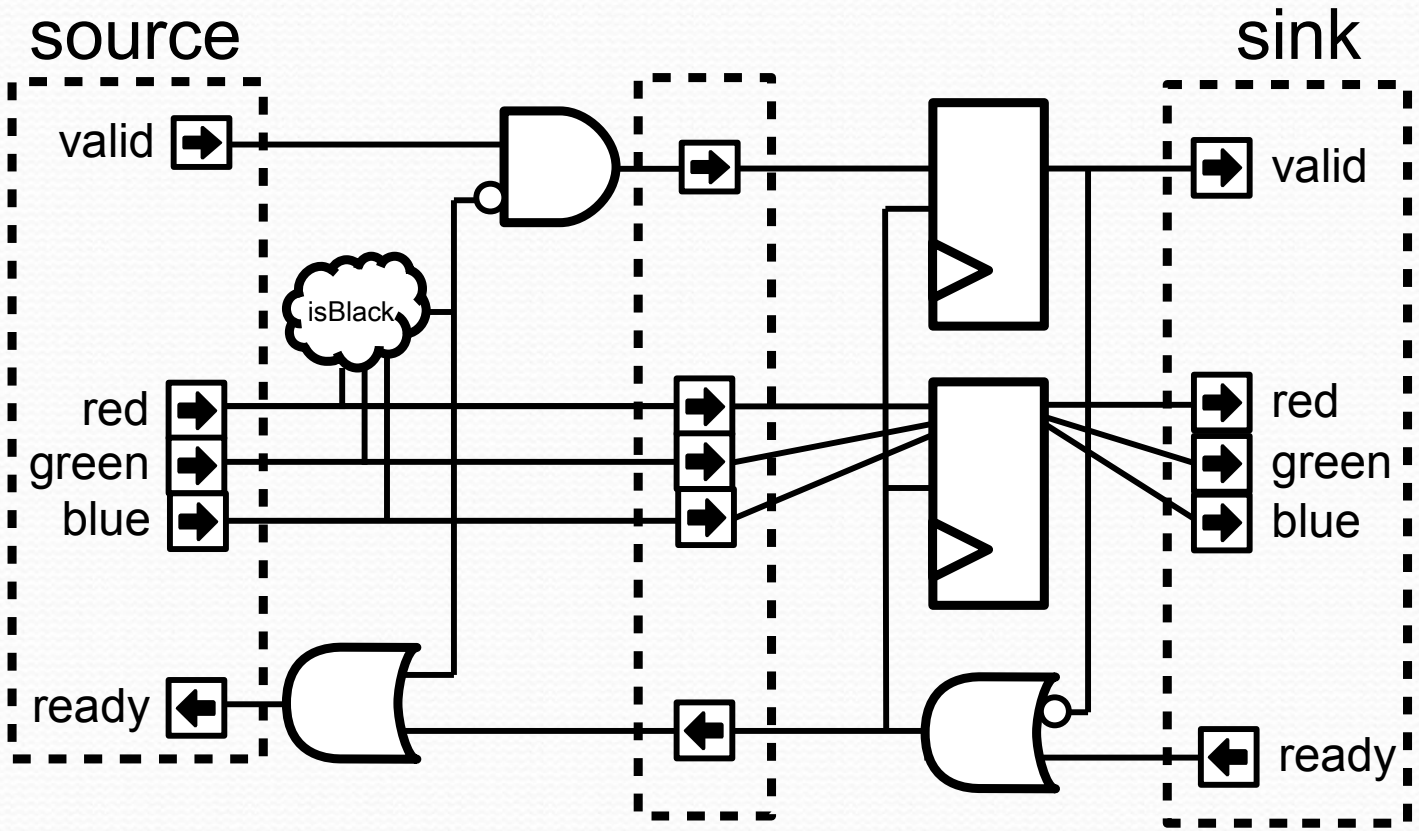
SpinalHDL => 2 lines
VHDL => 29 lines





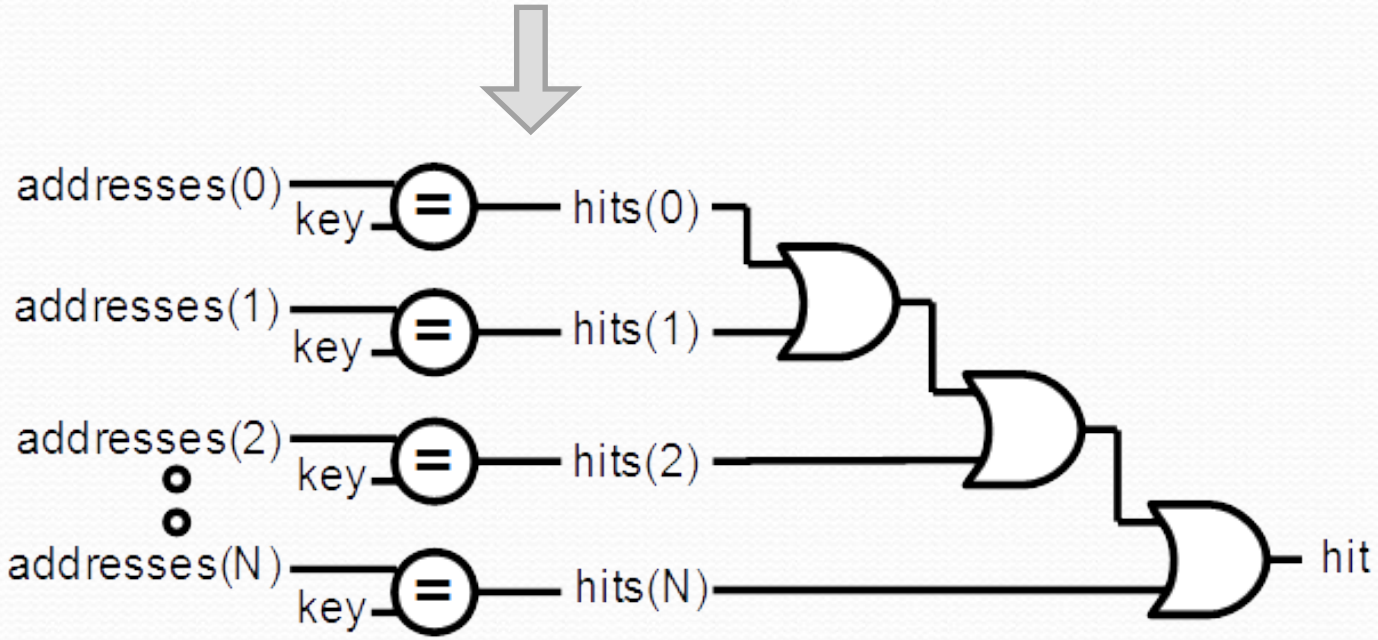
Abstract arbitration

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



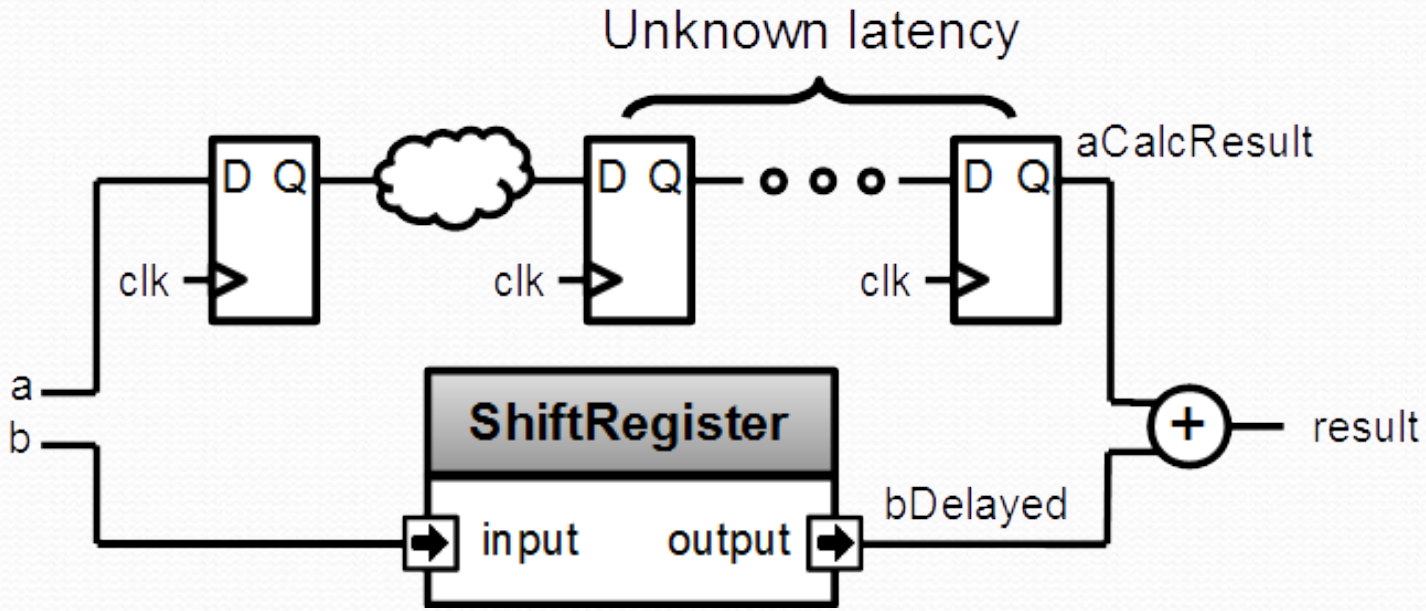
Functional programming

```
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)
```

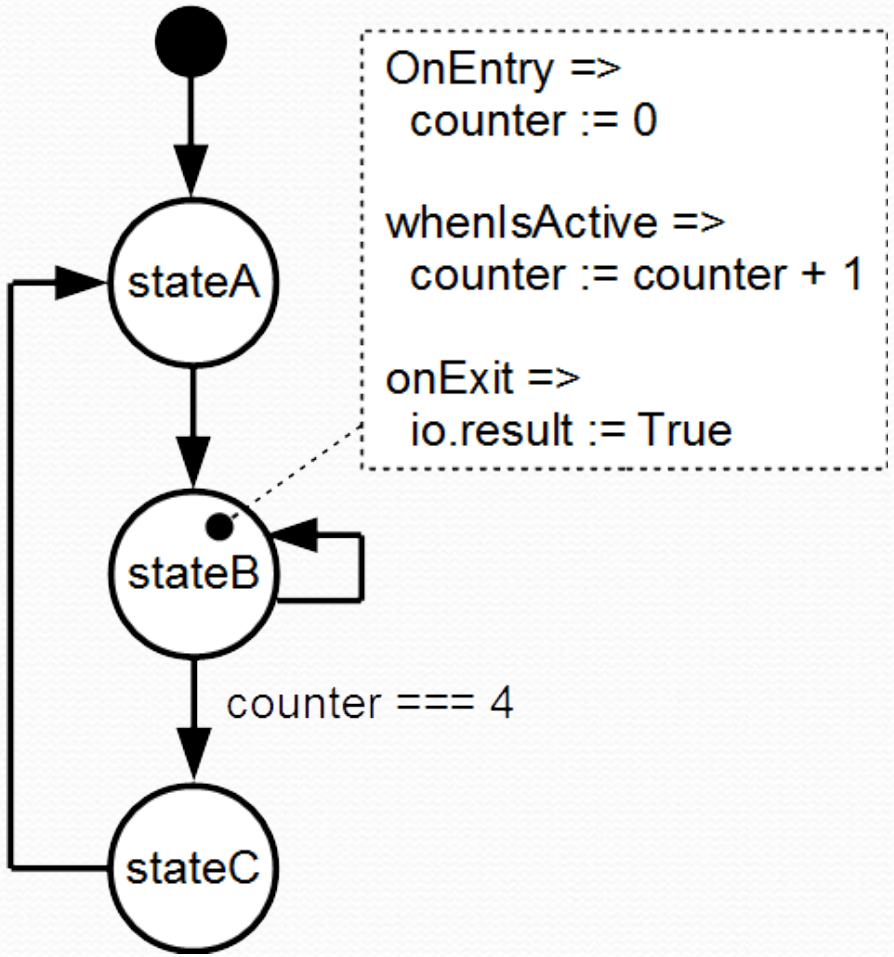


Design introspection

```
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
```



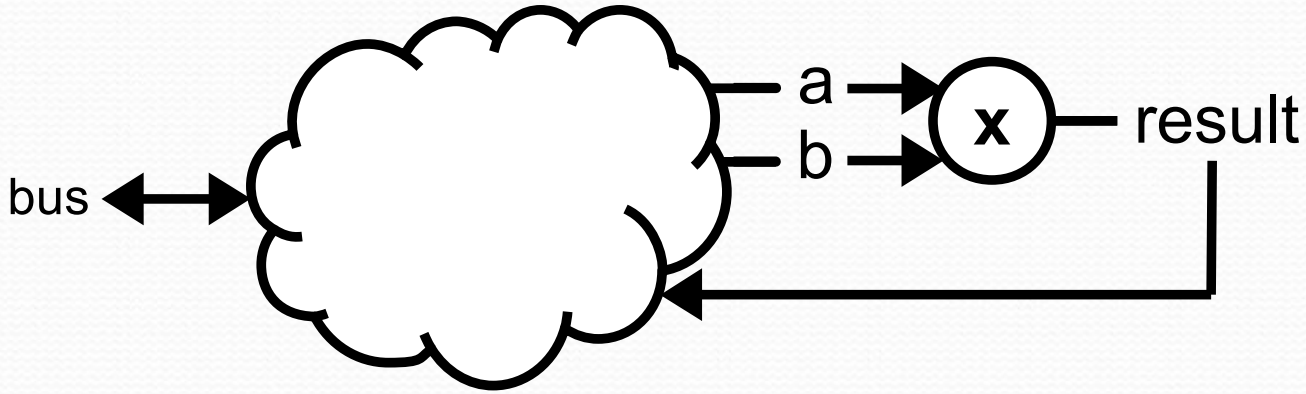
FSM




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

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

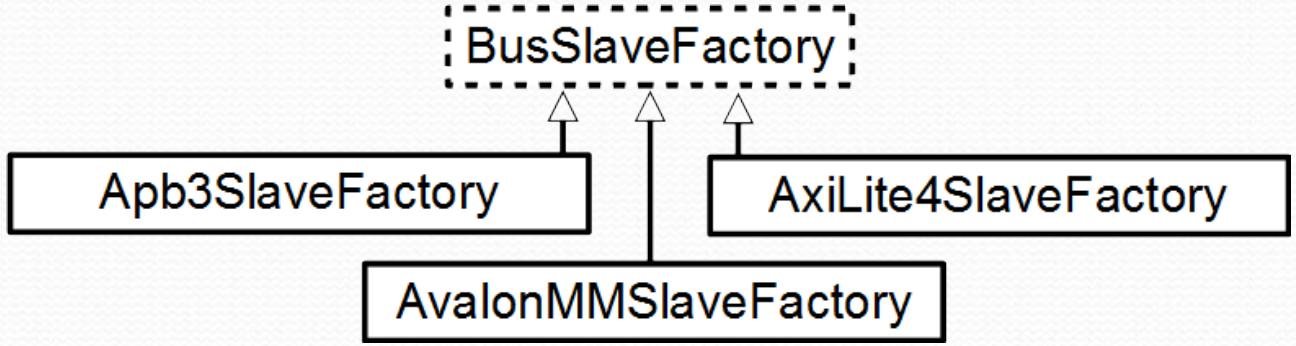
Abstract bus mapping

```
//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)
```



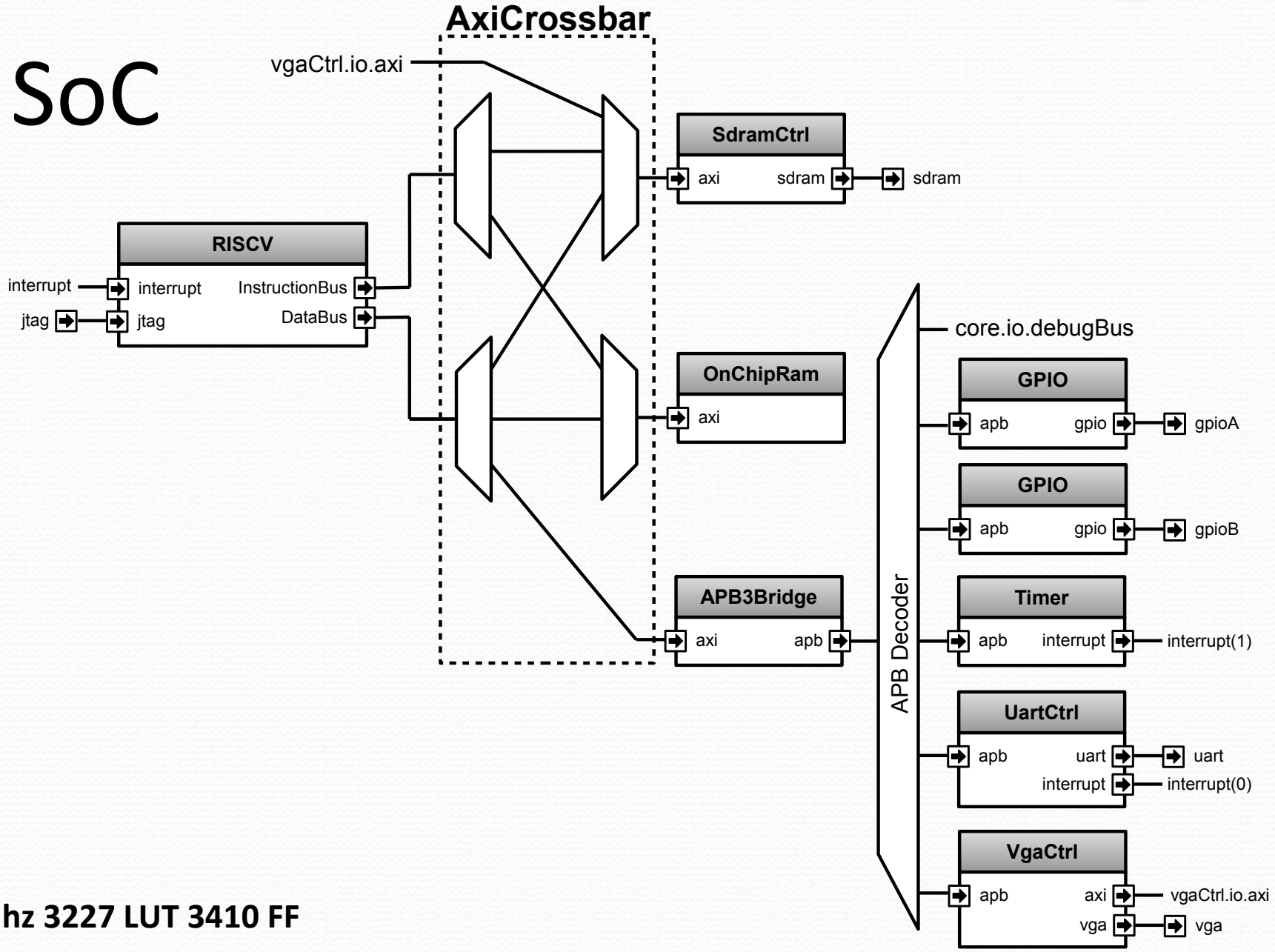
SlaveFactory

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




```
class Something extends Bundle{
  val a, b = UInt(32 bits)

  def driveFrom(factory : BusSlaveFactory) = new Area {
    factory.driveAndRead(a, address = 0x00)
    factory.driveAndRead(b, address = 0x04)
  }
}
```

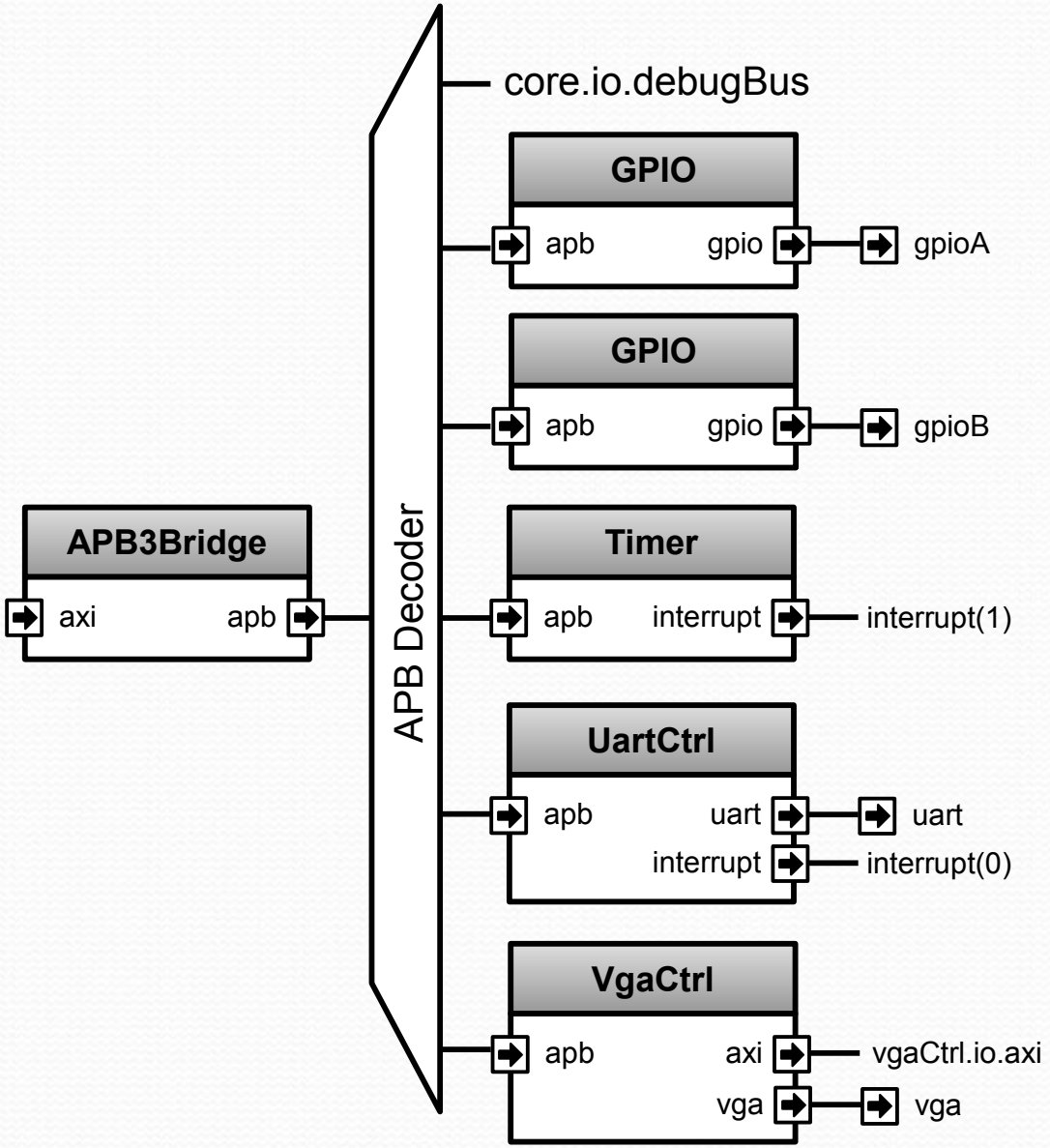
Briey SoC



Artix 7 -> 239 Mhz 3227 LUT 3410 FF

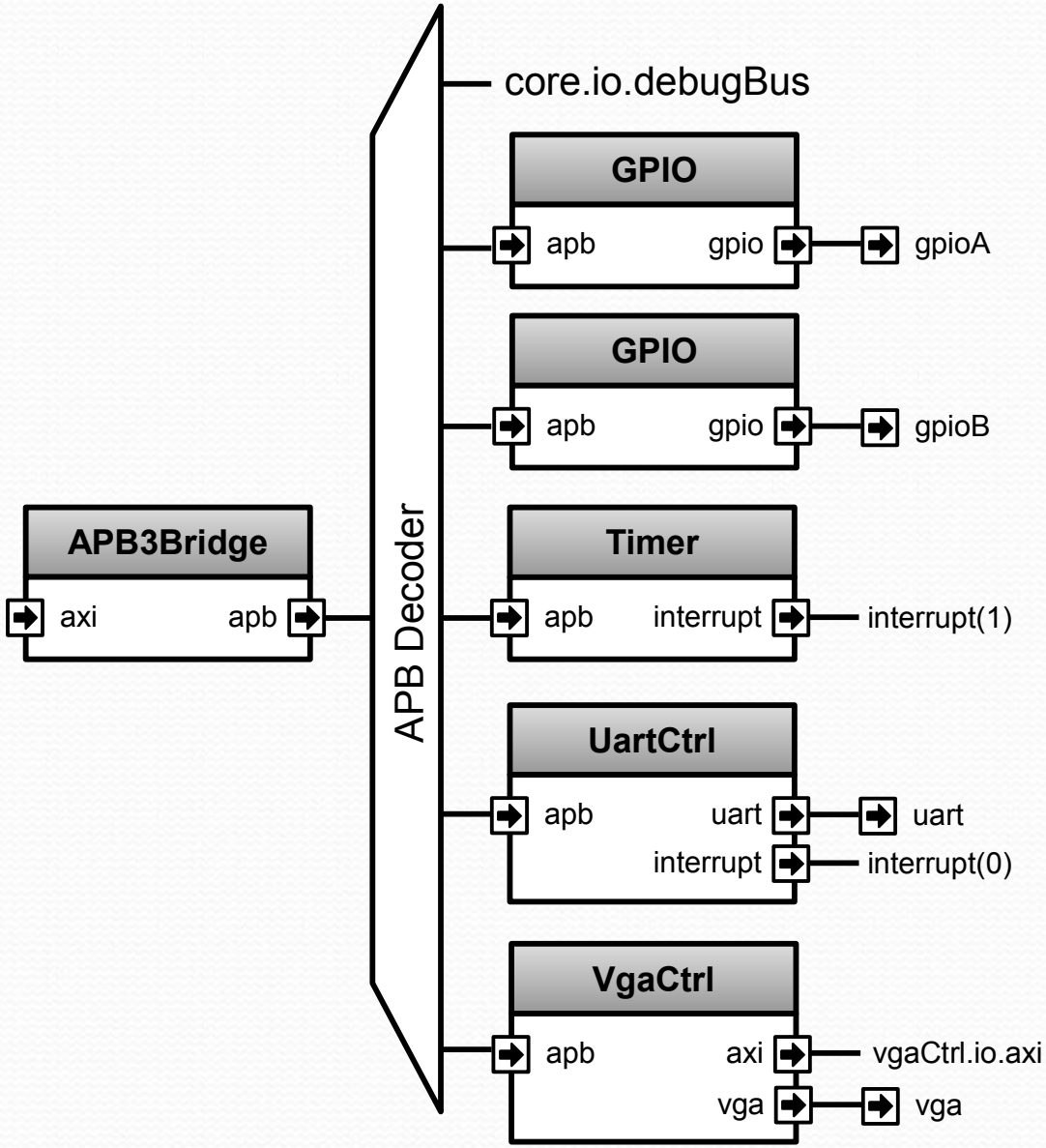
Peripheral side

```
val gpioACtrl = Apb3Gpio(gpioWidth = 32)
val gpioBCtrl = Apb3Gpio(gpioWidth = 32)
…
val apbDecoder = Apb3Decoder(
  master = apbBridge.io.apb,
  slaves = List(
    gpioACtrl.io.apb    -> (0x0000, 4 kB),
    gpioBCtrl.io.apb    -> (0x1000, 4 kB),
    uartCtrl.io.apb     -> (0x4000, 4 kB),
    timerCtrl.io.apb    -> (0x5000, 4 kB),
    vgaCtrl.io.apb      -> (0x6000, 4 kB)
  )
)
```



Peripheral side

```
val apbMapping = ArrayBuffer[(Apb3, SizeMapping)]()

val gpioACtrl = Apb3Gpio(gpioWidth = 32)
apbMapping += gpioACtrl.io.apb -> (0x0000, 4 kB)

val gpioBCtrl = Apb3Gpio(gpioWidth = 32)
apbMapping += gpioBCtrl.io.apb -> (0x1000, 4 kB)
...
val apbDecoder = Apb3Decoder(
  master = apbBridge.io.apb,
  slaves = apbMapping
)
```



Parameters

```
val apbMapping = ArrayBuffer[(Apb3, SizeMapping)]()
```

```
val gpioACtrl = Apb3Gpio(gpioWidth = 32)
apbMapping += gpioACtrl.io.apb -> (0x0000, 4 kB)
```

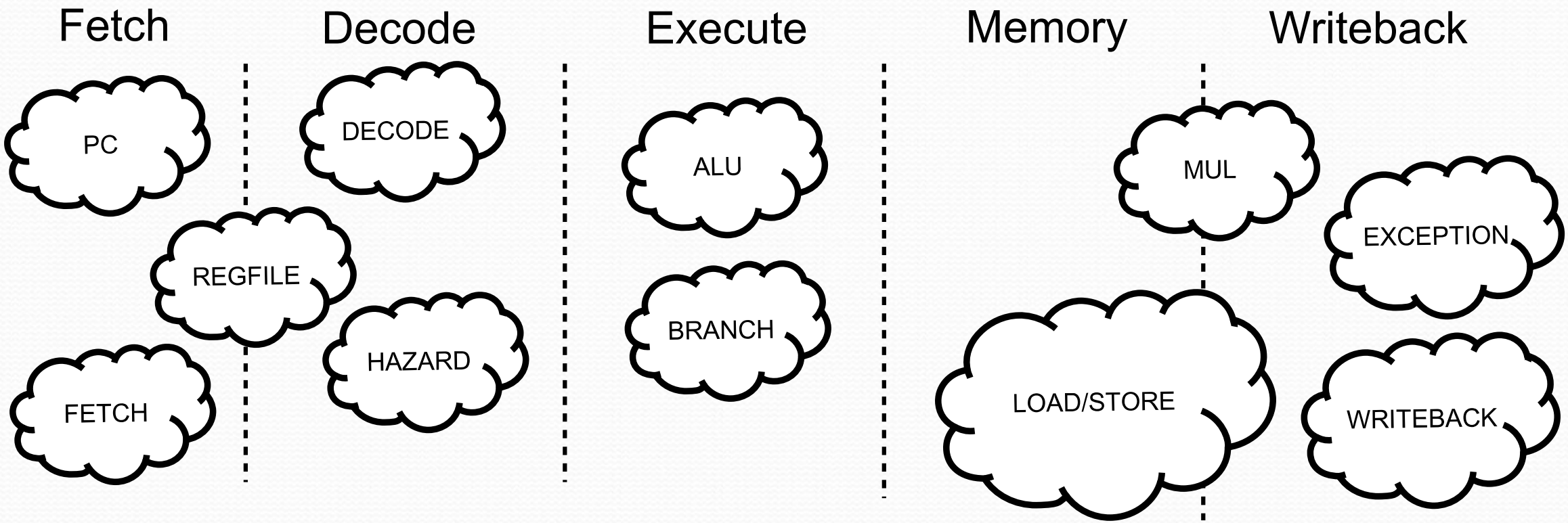
```
//   val gpioBCtrl = Apb3Gpio(gpioWidth = 32)
//   apbMapping += gpioBCtrl.io.apb -> (0x01000, 4 kB)
```

```
val gpioC = if(genGpioC) new Area{
   val ctrl = Apb3Gpio(gpioWidth = 32)
  apbMapping += peripheral.io.apb -> (0x1000, 4 kB)
}
```

```
val gpioX = for(i <- 0 until gpioXCtrlCount) yield new Area{
   val ctrl = Apb3Gpio(gpioWidth = 32)
   apbMapping += ctrl.io.apb -> (0x3000 + i * 0x2000, 4 kB)
}
```

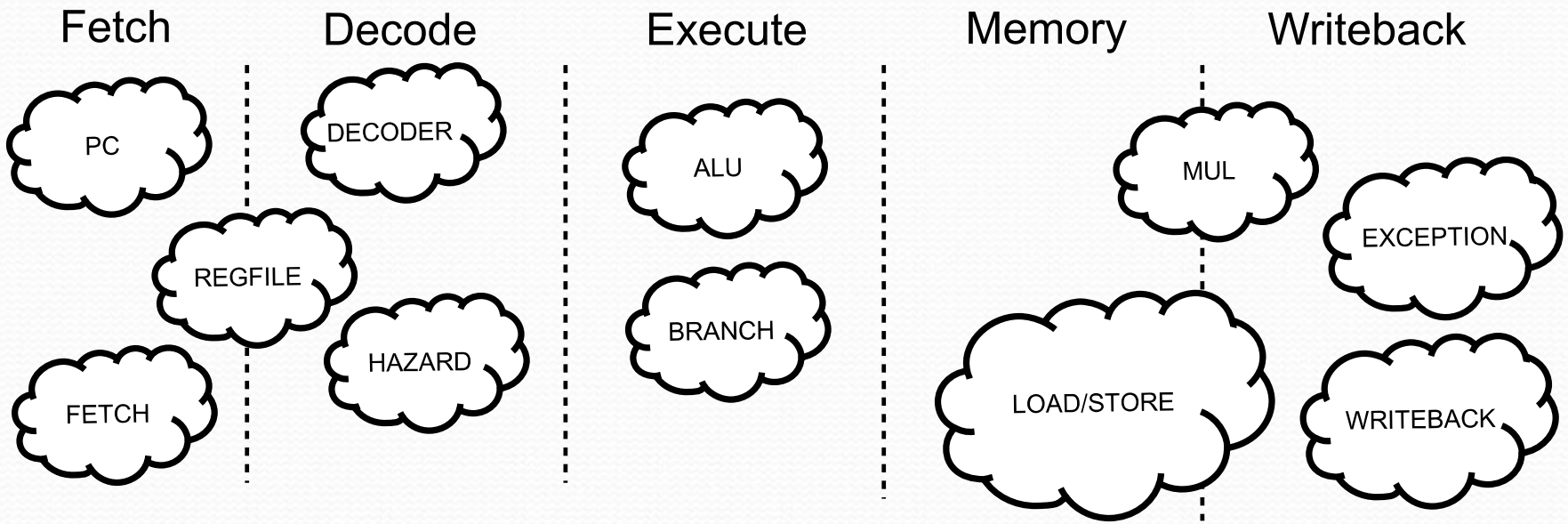
```
val apbDecoder = Apb3Decoder(
  master = apbBridge.io.apb,
  slaves = apbMapping
)
```

CPU pipelined over 5 stages



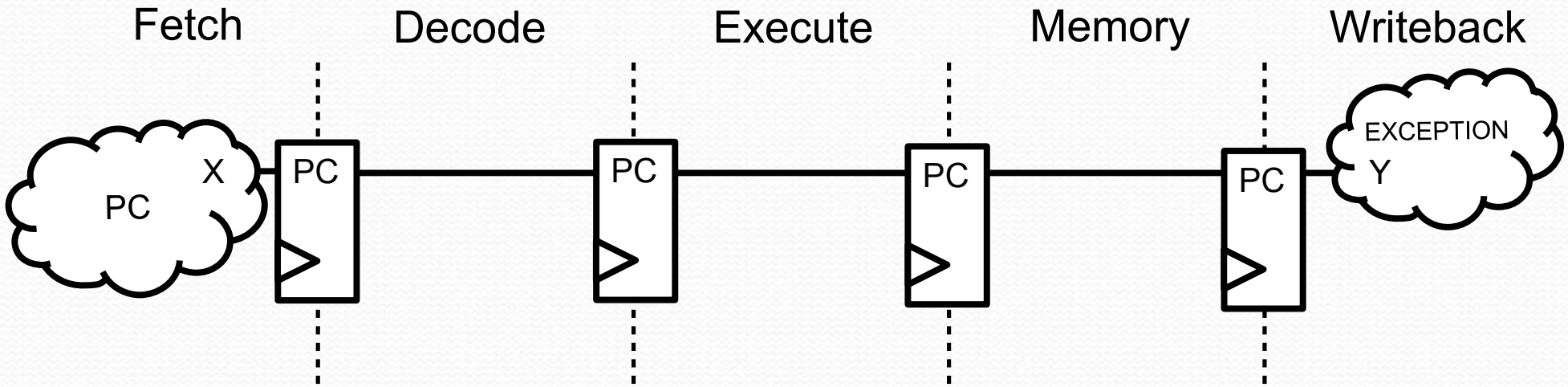
Modular CPU framework

```
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)
```

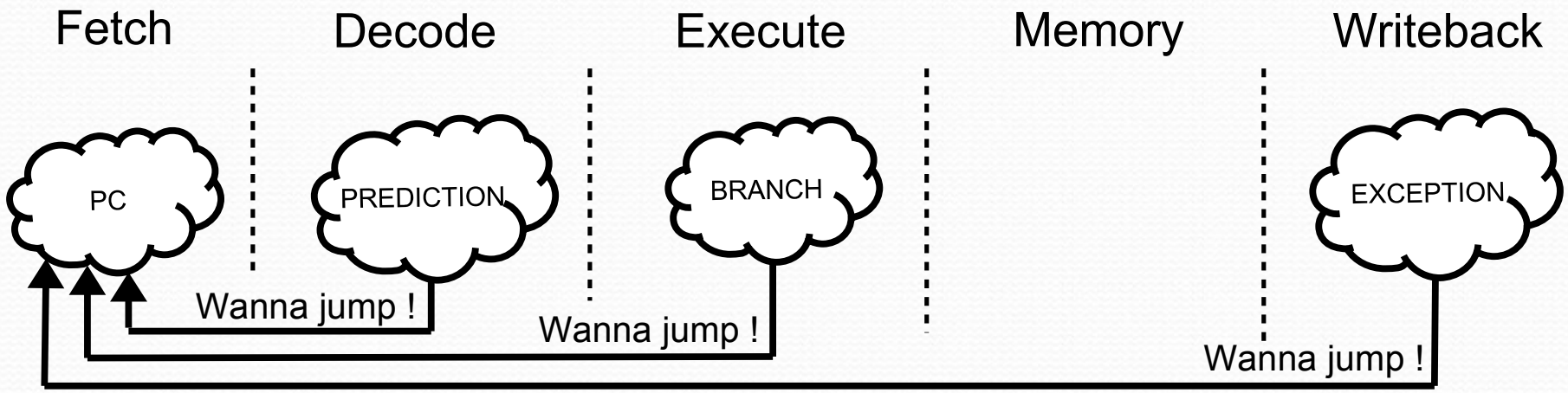


CPU framework - Connections

```
//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)
```



CPU framework - Connections




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

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

Some links

- Compiler sources :
  - https://github.com/SpinalHDL/SpinalHDL 
- Online documentation :
  - https://spinalhdl.github.io/SpinalDoc/
- Ready to use base project :
  - https://github.com/SpinalHDL/SpinalTemplateSbt
- Communication channels :
  - spinalhdl@gmail.com
  - https://gitter.im/SpinalHDL/SpinalHDL 
  - https://github.com/SpinalHDL/SpinalHDL/issues