SpinalHDL : Software generated hardware

Subject of the talk :

- Hardware Description Library (HDL)
- Hardware Generation Library (HGL)
- Hardware Construction language (HCL)
- Hardware Description API (HDAPI)
- Hardware Description Internal Domain Specific Language (HDIDSL)
- Elaborate-Through-Execution Hardware Design Language (ETEHDL)

(Which are probably all the same)

ORConf, September 2018

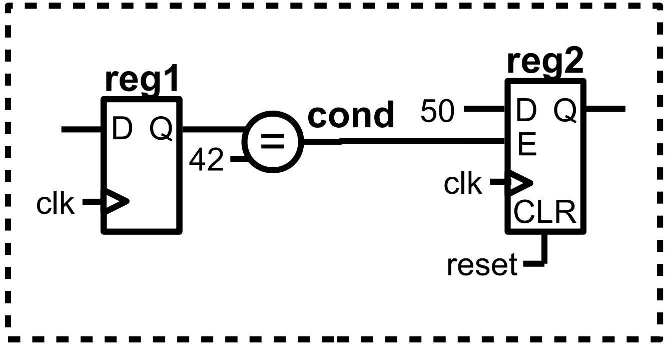
Hardware description API (HDAPI)

General purpose programming language + HDAPI

```
import spinal.core._

val reg1 = Reg(UInt(8 bits))
val cond = reg1 === 42

val reg2 = Reg(UInt(8 bits)) init(0)
when(cond){
  reg2 := 50
}
```

Execution

Hardware netlist



Netlist flush

VHDL/Verilog

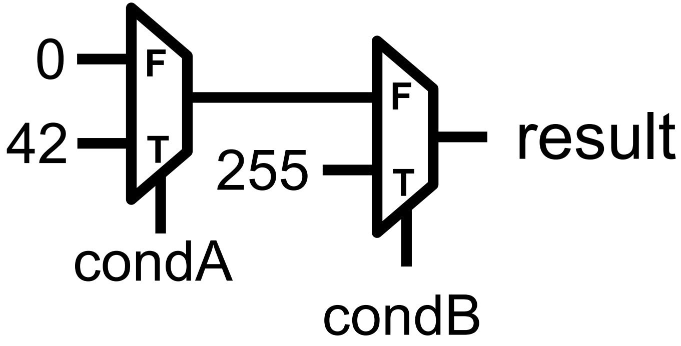
The SpinalHDL hardware API

- Hardware types (Bool, Bits, UInt, SInt, Bundle, Vec, …)
- Hardware conditional statements (when, switch)
- Hardware operators (:=, + - * / %, && || ^ ~, ...)
- And few others hardware constructs (Component, in, out, Area, ClockDomain, …)

Just some points before continuing

- About SpinalHDL
  - There is no logic overhead in the RTL
  - Will not generate broken VHDL/Verilog
    - check latches, combinatorial loop, signal width mismatch, …
  - SpinalHDL isn't a language, it's a HDAPI in Scala
- But this talk isn't realy about SpinalHDL
  - It will not introduce much of the SpinalHDL syntax
  - It will focus on giving you keys example to “feel” some of the semantic of HDAPI (SpinalHDL, Chisel, FHDL, …)

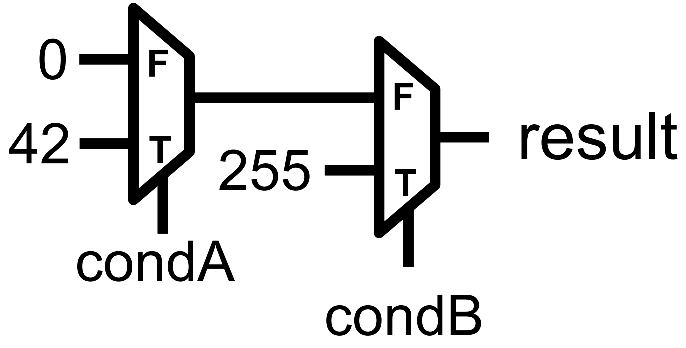
Semantic example



All the SpinalHDL API call which fill the netlist are in red
Everything else is Scala

```
//** Standard implementation**
val condA, condB = Bool()
val result = UInt(8 bits)
result := U(0)   //Can also be  := 0
when(condA) {
   result := U(42)
}
when(condB) {
   result := U(255)
}
```

Semantic example

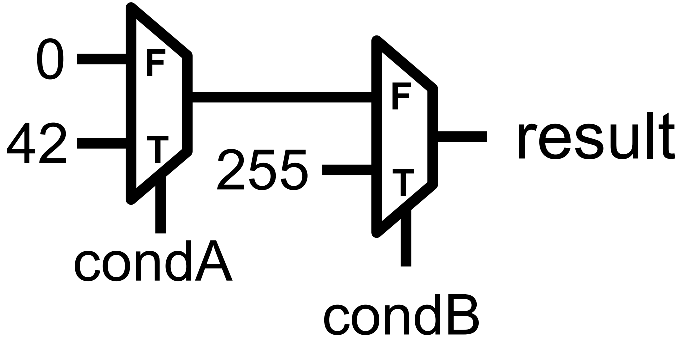


All the SpinalHDL API call which fill the netlist are in red
Everything else is Scala

```
//** Standard implementation**
val condA, condB = Bool()
val result = UInt(8 bits)
result := U(0)   //Can also be  := 0
when(condA) {
   result := U(42)
}
when(condB) {
   result := U(255)
}
```

```
//** Implementation via function/reference**
val condA, condB = Bool()
val result = UInt(8 bits)
result := U(0)
//Function call will be inlined in the netlist
whenAssign(condA, result, U(42))
whenAssign(condB, result, U(255))

def whenAssign(condition: Bool, target: UInt, value: UInt) = {
   when(condition) {
      target := value
   }
}
```

Semantic example

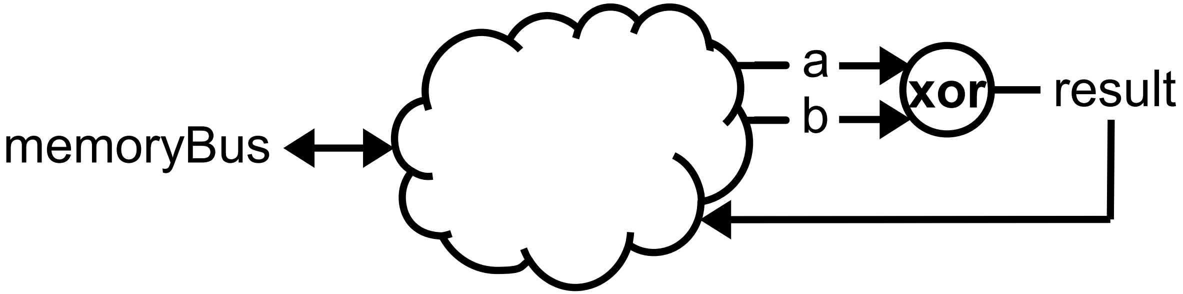


```
//** Generation via a software model **
val condA, condB = Bool()
val result = UInt(8 bits)
result := U(0)

val spec = ArrayBuffer[SpecElement]() //Scala dynamic array of SpecElement
class SpecElement(val cond: Bool, val target: UInt, val source: UInt)
spec += new SpecElement(condA, result, U(42))
spec += new SpecElement(condB, result, U(255))

//Loop iterations will be inlined in the netlist
for(e <- spec){
   when(e.cond){
      e.target := e.source
   }
}
```

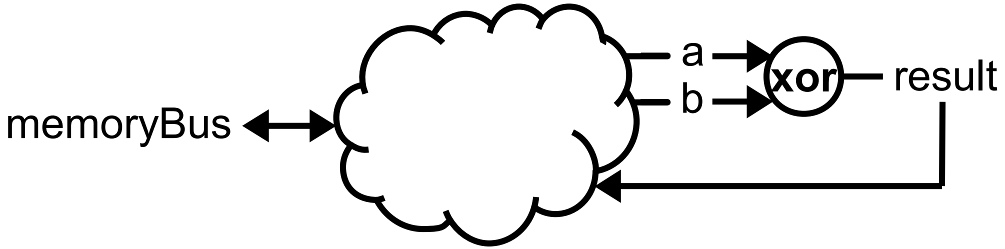
Index	Value
0	SpecElement(condA, result, U(42))
1	SpecElement(condB, result, U(255))

Simple peripheral design

- Let's imagine a simple peripheral which do XOR operations
  - We want to access it from a memory bus
  - So we need to create a bridge between the memoryBus and the peripheral signals (a, b, result)
- Let's implement it.



Peripheral implementation (with APB3)



```
val a, b = Reg(Bits(32 bits))
val result = a ^ b

//Create a new APB3 bus
val memoryBus = Apb3(addressWidth = 8, dataWidth = 32)

//Create the factory which creates the bridging logic between the bus and some hardware
val memoryMapper = new Apb3MemoryMapper(memoryBus)

//Drive 'a' and 'b' from the memoryBus
memoryMapper.write(target=a, address = 0x00)
memoryMapper.write(target=b, address = 0x04)

//Make 'result' readable by memoryBus
memoryMapper.read(source=result, address = 0x08)

memoryMapper.build()
```

Key	Value
0x00	WriteAccess(a)
0x04	WriteAccess(b)
0x08	ReadAccess(result)

Memory mapping datamodel

```
class WriteAccess(val target : Bits)
class ReadAccess(val source : Bits)

class Apb3MemoryMapper(bus : Apb3) {
  val spec = Map[Int, ArrayBuffer[Any]]()

  def write(target : Bits, address : Int)  = spec(address) += new WriteAccess(target)
  def read(source : Bits, address : Int) = spec(address) += new ReadAccess(source)

  def build() = { … } // Implementation on the next slide
}
```

Key	Value
0x00	WriteAccess(a)
0x04	WriteAccess(b)
0x08	ReadAccess(result)

Memory mapping APB3 implementation

```
class Apb3MemoryMapper(bus : Apb3) {
  override def build() = {
    ...









  }
}
```

Key	Value
0x00	WriteAccess(a)
0x04	WriteAccess(b)
0x08	ReadAccess(result)

Memory mapping APB3 implementation

```
class Apb3MemoryMapper(bus : Apb3) {
    override def build() = {
        ...
        val doWrite  = bus.PSEL(0) && bus.PENABLE && bus.PREADY &&  bus.PWRITE



    }
}
```

Key	Value
0x00	WriteAccess(a)
0x04	WriteAccess(b)
0x08	ReadAccess(result)

Memory mapping APB3 implementation

```
class Apb3MemoryMapper(bus : Apb3) {
    override def build() = {
        ...
        val doWrite  = bus.PSEL(0) && bus.PENABLE && bus.PREADY &&  bus.PWRITE
        switch(bus.PADDR) { // Hardware switch statement









        }
    }
}
```

Key	Value
0x00	WriteAccess(a)
0x04	WriteAccess(b)
0x08	ReadAccess(result)

Memory mapping APB3 implementation

```
class Apb3MemoryMapper(bus : Apb3) {
    override def build() = {
        ...
        val doWrite  = bus.PSEL(0) && bus.PENABLE && bus.PREADY &&  bus.PWRITE
        switch(bus.PADDR) {
          for ((address, jobs) <- spec) { // Software loop




          }
        }
    }
}
```

Key	Value
0x00	WriteAccess(a)
0x04	WriteAccess(b)
0x08	ReadAccess(result)

Memory mapping APB3 implementation

```
class Apb3MemoryMapper(bus : Apb3) {
    override def build() = {
        ...
        val doWrite  = bus.PSEL(0) && bus.PENABLE && bus.PREADY &&  bus.PWRITE
        switch(bus.PADDR) {
          for ((address, jobs) <- spec) {
              is(address) {  // Hardware is statement



              }
          }
        }
    }
}
```

Key	Value
0x00	WriteAccess(a)
0x04	WriteAccess(b)
0x08	ReadAccess(result)

Memory mapping APB3 implementation

```
class Apb3MemoryMapper(bus : Apb3) {
    override def build() = {
        ...
        val doWrite  = bus.PSEL(0) && bus.PENABLE && bus.PREADY &&  bus.PWRITE
        switch(bus.PADDR) {
          for ((address, jobs) <- spec) {
              is(address) {
                  for(job <- jobs)  { // Software loop statement


                  }
              }
           }
        }
    }
}
```

Key	Value
0x00	WriteAccess(a)
0x04	WriteAccess(b)
0x08	ReadAccess(result)

Memory mapping APB3 implementation

```
class Apb3MemoryMapper(bus : Apb3) {
    override def build() = {
        ...
        val doWrite  = bus.PSEL(0) && bus.PENABLE && bus.PREADY &&  bus.PWRITE
        switch(bus.PADDR) {
          for ((address, jobs) <- spec) {
              is(address) {
                  for(job <- jobs) job match { // Software switch statement
                      case w : WriteAccess => ...
                      case r   : ReadAccess => ...
                  }
              }
          }
        }
    }
}
```

Key	Value
0x00	WriteAccess(a)
0x04	WriteAccess(b)
0x08	ReadAccess(result)

Memory mapping APB3 implementation

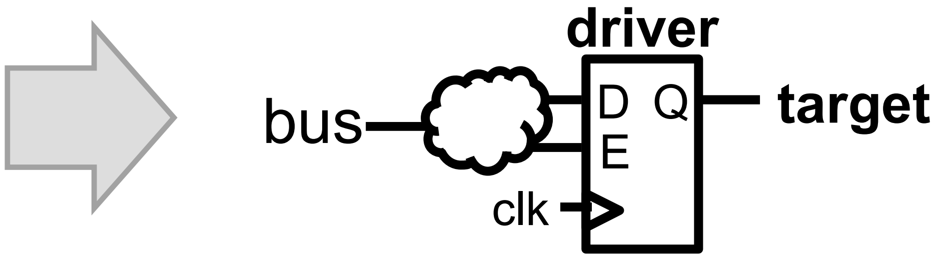
```
class Apb3MemoryMapper(bus : Apb3) {
    override def build() = {
        ...
        val doWrite  = bus.PSEL(0) && bus.PENABLE && bus.PREADY &&  bus.PWRITE
        switch(bus.PADDR) {
          for ((address, jobs) <- spec) {
              is(address) {
                  for(job <- jobs) job match {
                      case w : WriteAccess => when(doWrite) { w.target := bus.PWDATA }
                      case r   : ReadAccess => bus.PRDATA := r.source
                  }
              }
          }
        }
    }
}
```

Key	Value
0x00	WriteAccess(a)
0x04	WriteAccess(b)
0x08	ReadAccess(result)

And the drive primitive ?

- Build on the top of the write primitive

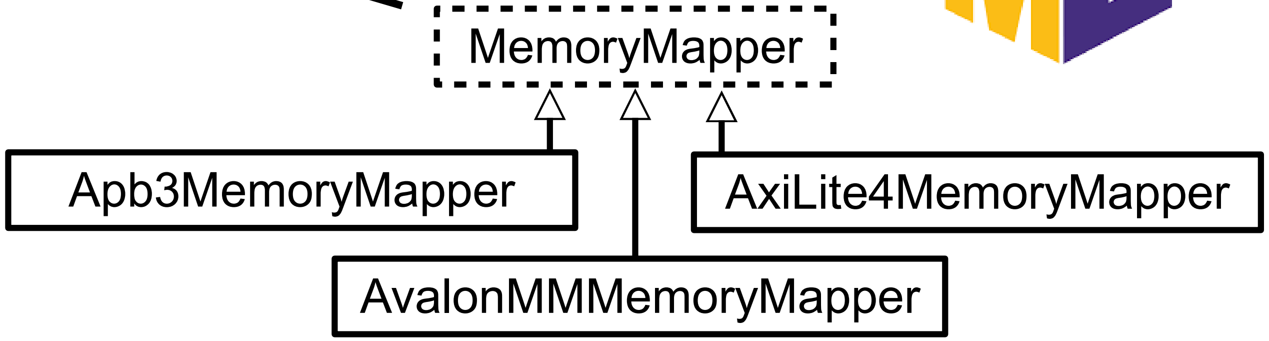
```
class Apb3MemoryMapper(bus : Apb3) {
    ...
    def write(target : Bits, address : Int) = ...
    def read(source : Bits, address : Int) = ...

    def drive(target : Bits, address : Int) = {
        val driver = Reg(Bits(widthOf(target) bits))
        target := driver
        write(driver, address)
    }
}
```



- You can do the same for many other primitives, ex : driveAndRead

Let's do proper software enginnering

```scala
abstract class MemoryMapper{
  val spec = HashMap[Int, ArrayBuffer[Any]]()

  def write(target : Bits, address : Int)  =...
  def read(source : Bits, address : Int) = ...

  def build() = ??? // Unimplemented
}
```



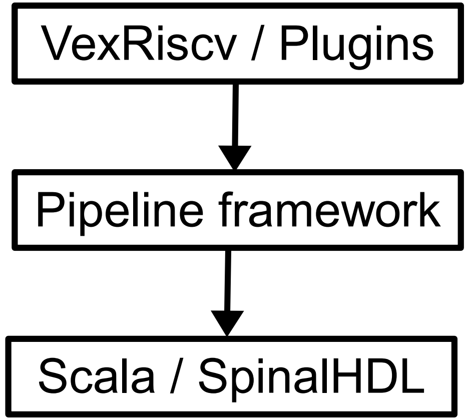
```scala
class Apb3MemoryMapper(bus : Apb3) extends MemoryMapper{
    override def build() = {
        ...
    }
}
```

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

    def readFrom(factory : MemoryMapper) = {
        factory.read(a, address = 0x00)
        factory.read(b, address = 0x04)
    }
}
```

We are now bus agnostic !

VexRiscv : A CPU implemented with SpinalHDL

- RV32I[M][C], 5 stages
- Optional I$ D$
- Optional JTAG with openocd port
- On Artix7 FPGA :
  - 500 to 2000 LUT
  - 200 to 340 Mhz
  - 0.51 to 1.44 DMIPS/Mhz (Drystone 2.1, -O3 -fno-inline)
- But the implementation isn't usual



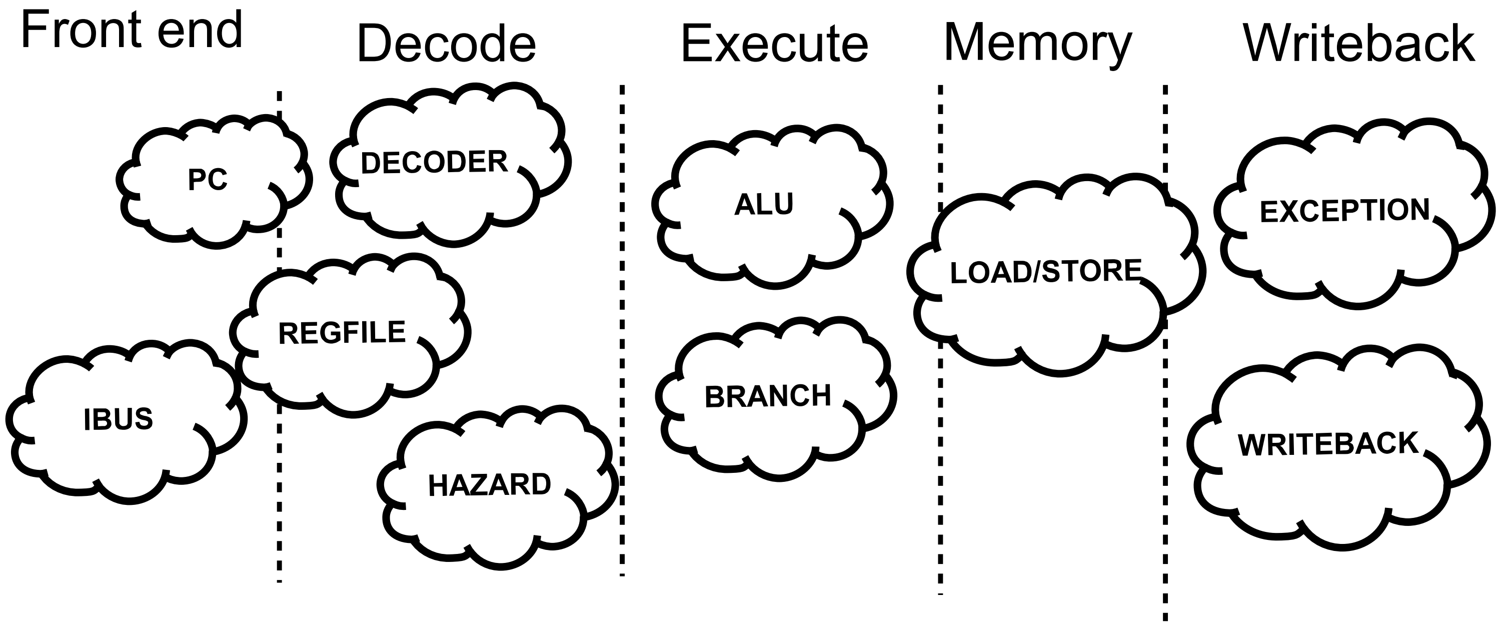
VexRiscv / Plugins

Pipeline framework

Scala / SpinalHDL

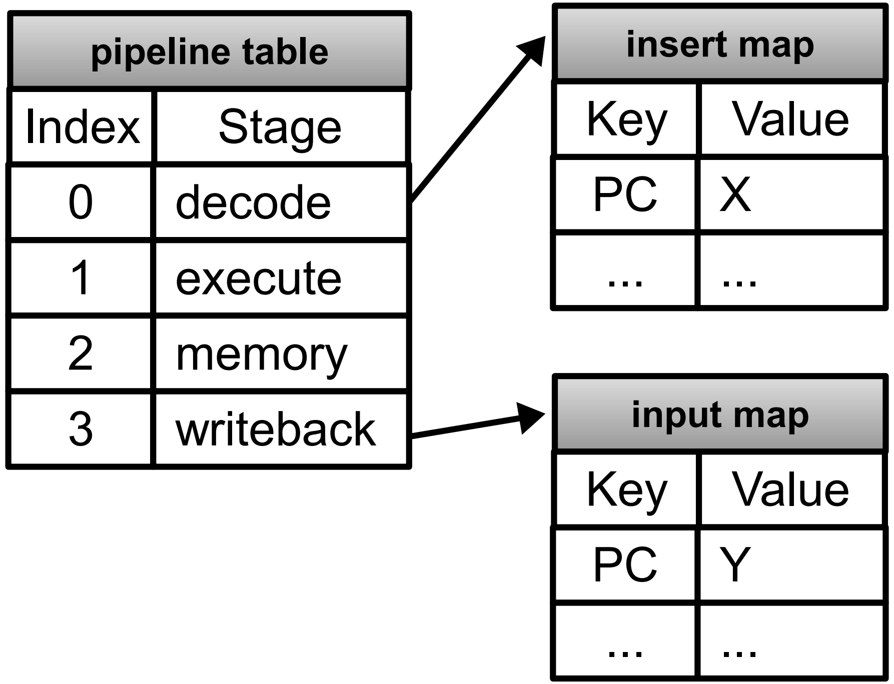
Modular CPU framework

```
val config = VexRiscvConfig()

config.plugins ++= List(
  new IBusSimplePlugin(resetVector = 0x80000000l),
  new DBusSimplePlugin,
  new CsrPlugin(CsrPluginConfig.smallest),
  new DecoderSimplePlugin,
  new RegFilePlugin(regFileReadyKind = plugin.SYNC),
  new SrcPlugin,
  new IntAluPlugin,
  new BranchPlugin(earlyBranch = false),
  new MulDivIterativePlugin(
    mulUnrollFactor = 4,
    divUnrollFactor = 1
  ),
  new FullBarrelShifterPlugin,
  new HazardSimplePlugin,
  new YamlPlugin("cpu0.yaml")
)

new VexRiscv(config)
```



CPU framework - Connections

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

//Somewere in the plugin which manage the program counter
decode.insert(PC) := X

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



pipeline table

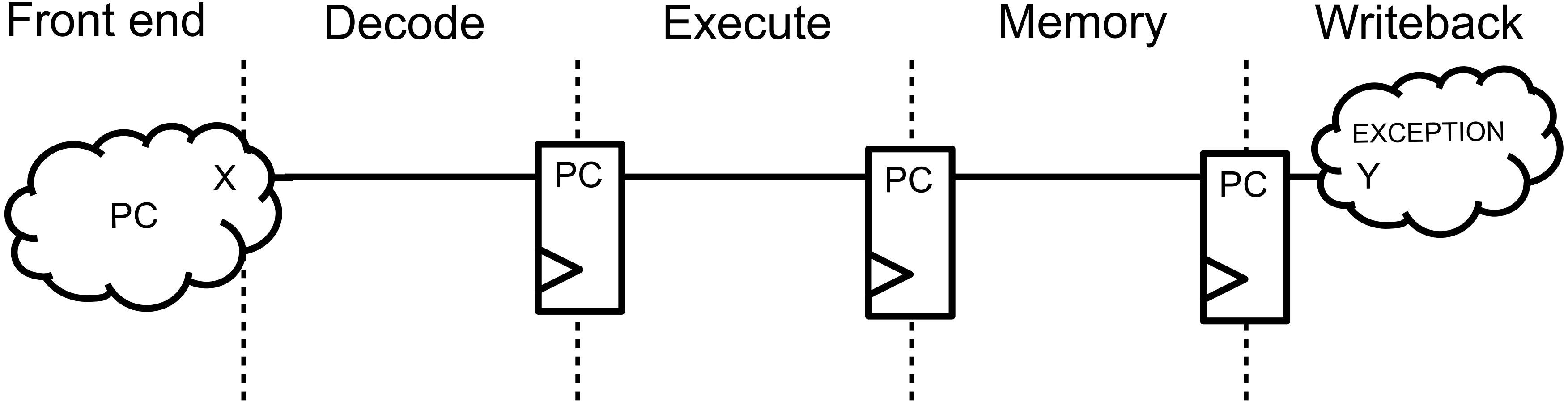
Index	Stage
0	decode
1	execute
2	memory
3	writeback

insert map

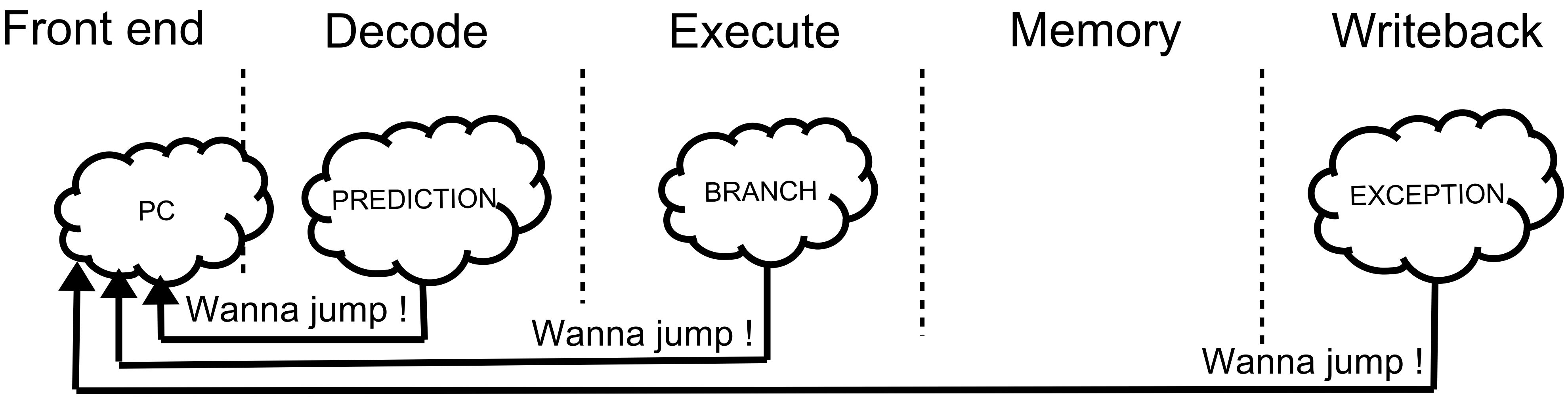
Key	Value
PC	X
...	...

input map

Key	Value
PC	Y
...	...



CPU framework - Connections

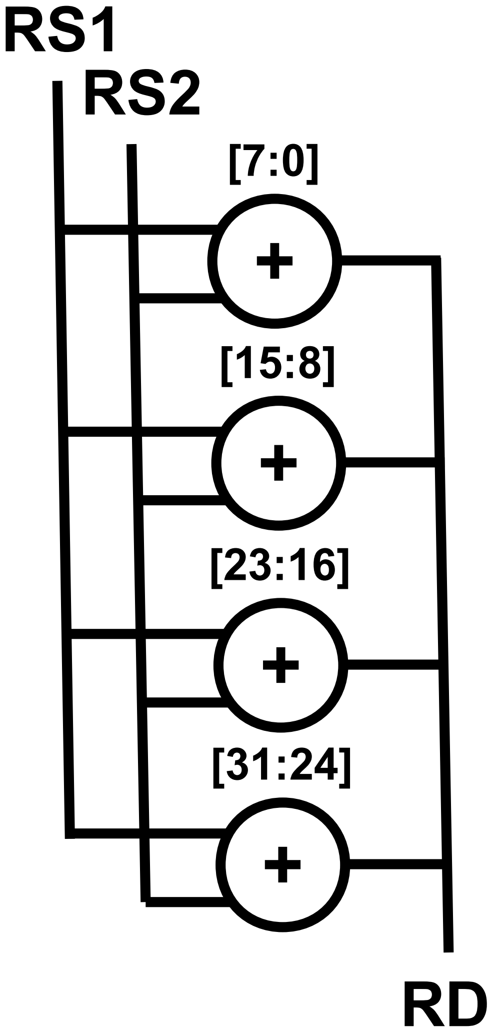


```
//Somewhere in the Branch plugin
val jumpInterface  = pcPlugin.createJumpInterface(stage = execute)

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

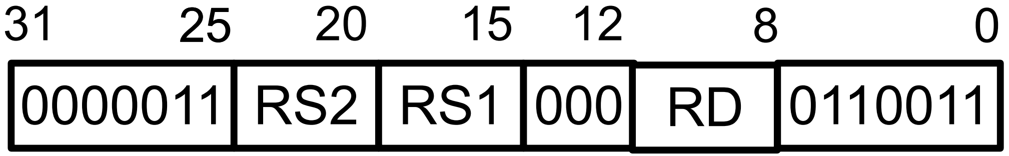
Plugin example : 4x8bits add (SIMD)

31 25	20	15	12	8	0
0000011	RS2	RS1	000	RD	0110011

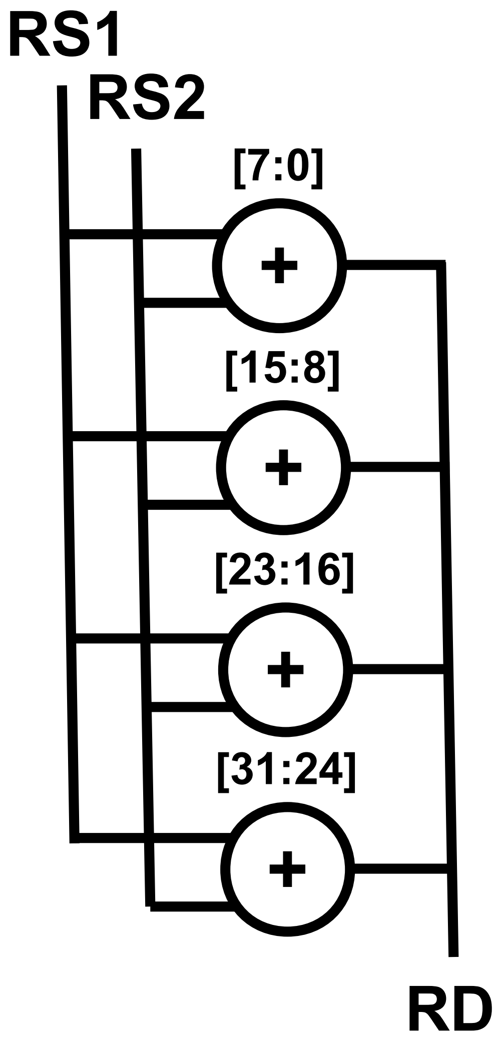
```
class SimdAddPlugin extends Plugin[VexRiscv]{
   override def setup(pipeline: VexRiscv): Unit = {
      ...
   }

   override def build(pipeline: VexRiscv): Unit = {
      ..
   }
}
```



```scala
class SimdAddPlugin extends Plugin[VexRiscv]{
  object IS_SIMD_ADD extends Stageable(Bool)

  override def setup(pipeline: VexRiscv): Unit = {
    val decoderService = pipeline.service(classOf[DecoderService])
    decoderService.addDefault(IS_SIMD_ADD, False)
    decoderService.add(
      key = M"0000011----------000-----0110011",
      List(
        IS_SIMD_ADD                     -> True,
        REGFILE_WRITE_VALID             -> True,
        BYPASSABLE_EXECUTE_STAGE -> True,
        BYPASSABLE_MEMORY_STAGE  -> True,
        RS1_USE                         -> True,
        RS2_USE                         -> True
      )
    )
  }

  override def build(pipeline: VexRiscv): Unit = { .. }
}
```



```
class SimdAddPlugin extends Plugin[VexRiscv]{
  object IS_SIMD_ADD extends Stageable(Bool)

  override def setup(pipeline: VexRiscv): Unit = { .. }

  override def build(pipeline: VexRiscv): Unit = {
    execute plug new Area {
      val rs1 = U(execute.input(RS1))
      val rs2 = U(execute.input(RS2))
      val rd = UInt(32 bits)

      rd(7 downto 0)    := rs1(7 downto 0)    + rs2(7 downto 0)
      rd(16 downto 8)   := rs1(16 downto 8)   + rs2(16 downto 8)
      rd(23 downto 16) := rs1(23 downto 16) + rs2(23 downto 16)
      rd(31 downto 24) := rs1(31 downto 24) + rs2(31 downto 24)

      when(execute.input(IS_SIMD_ADD)) {
        execute.output(REGFILE_WRITE_DATA) := B(rd)
      }
    }
  }
}
```



The end, tips to think about all of this

- When you execute the scala code, each SpinalHDL API call will add information in the SpinalHDL netlist
- All the elaborated features come from Scala (host general purpose language)
- None of the previous slide require SpinalHDL modifications, they are all implemented on the top of SpinalHDL/Scala
- Don't take the previous slide as a “You should use this framework” but as demonstration / proof of concept

Some links

- SpinalHDL library 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