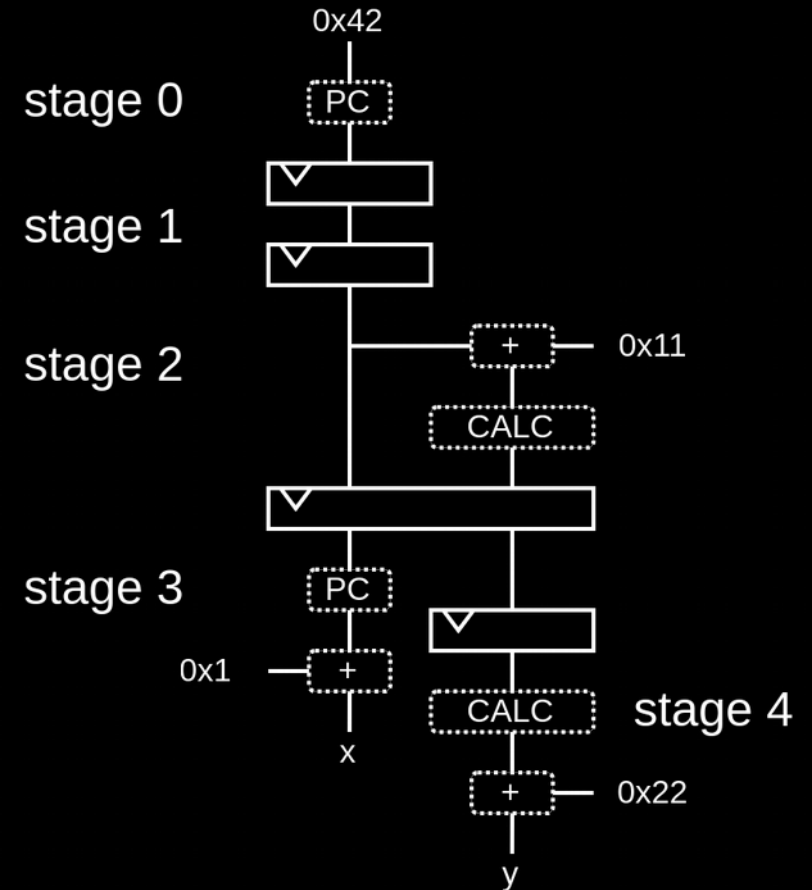
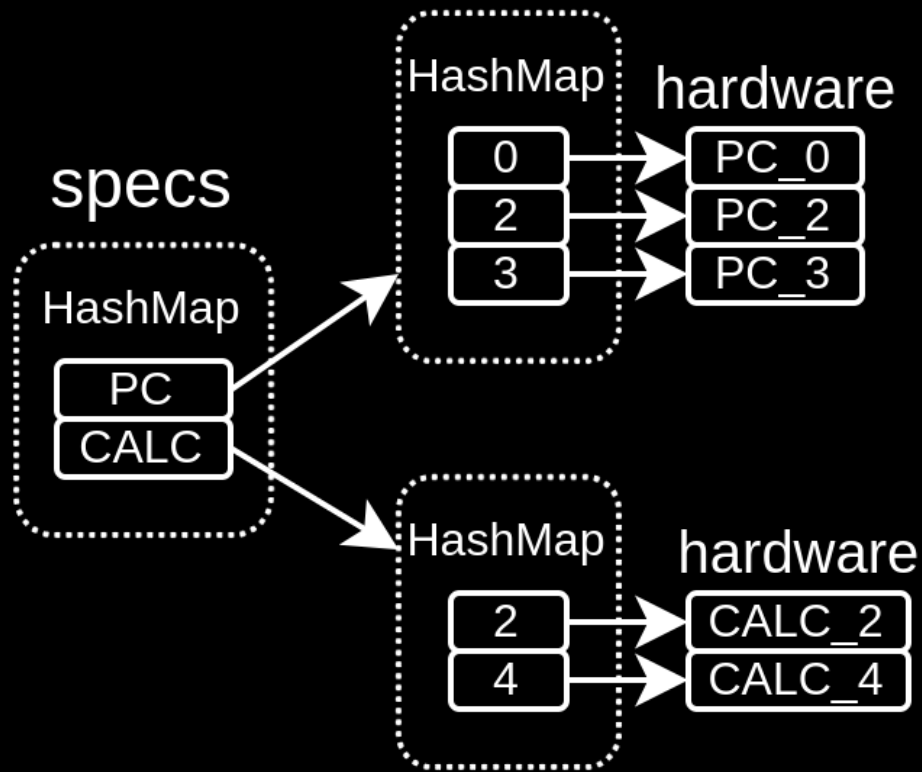


# Writing software to elaborate hardware (SpinalHDL)



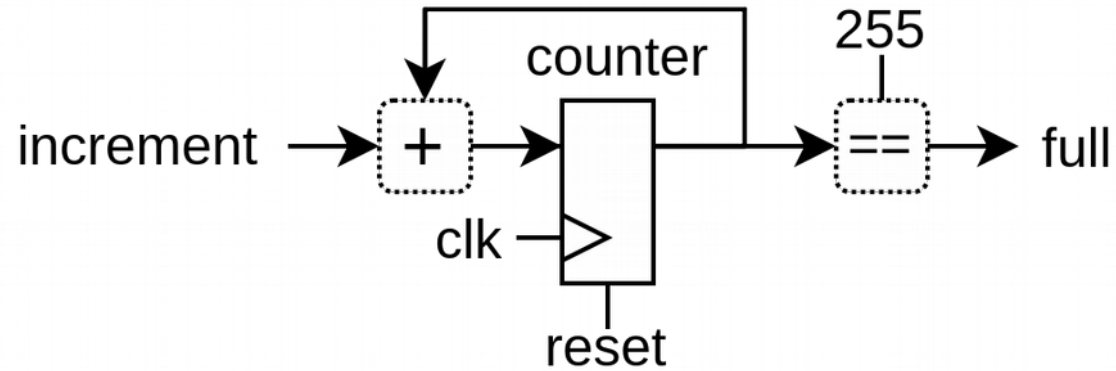
# Background / whoami

- Dolu1990 on github
- Active on open/free project
  - SpinalHDL (2015) / VexRiscv (2017) / NaxRiscv(2021) / VexiiRiscv (2023)
- Software / Hardware background
  - Industrial system / Electronic degree

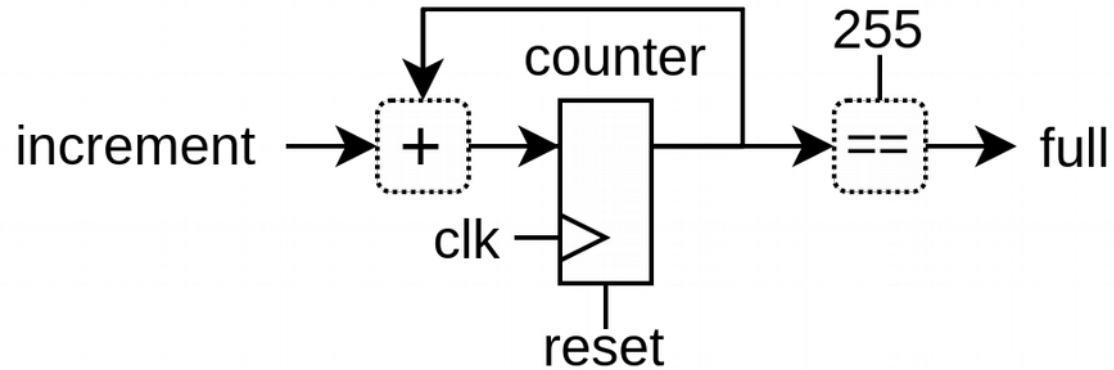
# Let's use a software language

```
object Main extends App{  
  println("Hello world")  
}
```

# Let's use an hardware description library (HDL)



# Let's use an hardware description library (HDL)

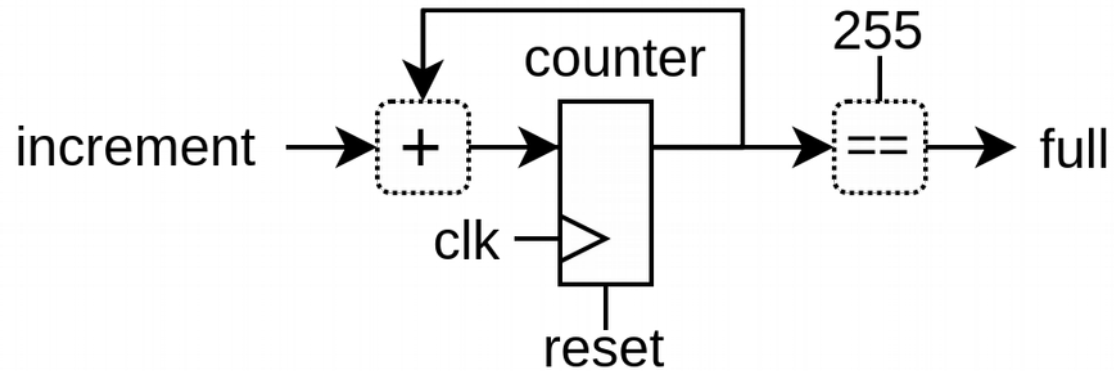


```
import spinal.core._

object Main extends App{
  SpinalVerilog(new Timer)
}

class Timer extends Component {
  val increment = in(Bool())
  val counter   = Reg(UInt(8 bits)) init(0)
  val full      = out(counter == 255)
  when(increment){
    counter := counter + 1
  }
}
```

# Let's use an hardware description library (HDL)



```
import spinal.core._
```

```
object Main extends App{
  SpinalVerilog(new Timer)
}
```

```
class Timer extends Component {
  val increment = in(Bool())
  val counter   = Reg(UInt(8 bits)) init(0)
  val full      = out(counter == 255)
  when(increment){
    counter := counter + 1
  }
}
```

```
module Timer (
  input  wire increment,
  output wire full,
  input  wire clk,
  input  wire reset
);

  reg [7:0] counter;

  assign full = (counter == 8'hff);
  always @(posedge clk or posedge reset) begin
    if(reset) begin
      counter <= 8'h00;
    end else begin
      if(increment) begin
        counter <= (counter + 8'h01);
      end
    end
  end

endmodule
```

# Data structure / parameters

```
case class Pixel(width: Int) extends Bundle{
  val r, g, b = UInt(width bits)
}

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

val bus = Stream(Pixel(8))
bus.valid := False
bus.data.r := 0x11
bus.data.g := 0x22
bus.data.b := 0x33
```

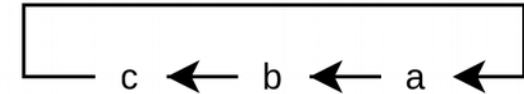
# Integrated linting

- Latches
- Combinatorial loops
- Unspecified clock crossing
- Undriven signals
- Width mismatch
- ...



# Integrated linting

- Latches
- Combinatorial loops
- Unspecified clock crossing
- Undriven signals
- Width mismatch
- ...



```
val a, b, c = Bool()  
c := b; b := a; a := c
```

COMBINATORIAL LOOP :

(toplevel/a : Bool)

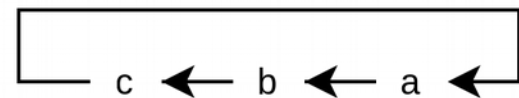
(toplevel/b : Bool)

(toplevel/c : Bool)

(toplevel/a : Bool)

# Integrated linting

- Latches
- Combinatorial loops
- Unspecified clock crossing
- Undriven signals
- Width mismatch
- ...



```
val a,b,c = Bool()  
c := b; b := a; a := c
```

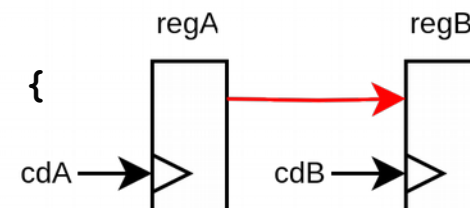
COMBINATORIAL LOOP :

(toplevel/a : Bool)

(toplevel/b : Bool)

(toplevel/c : Bool)

(toplevel/a : Bool)

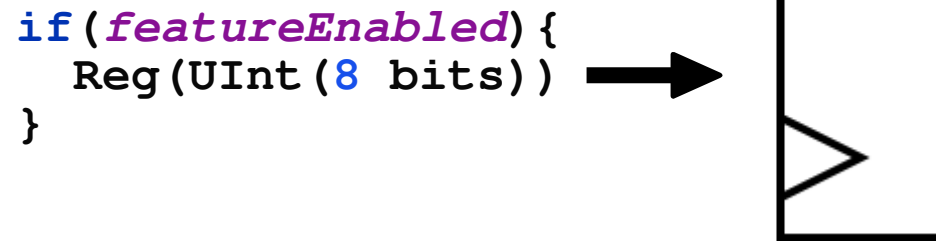


```
class Toplevel(cdA : ClockDomain, cdB : ClockDomain) extends Component {  
  val regA = cdA(Reg(Bool()))  
  val regB = cdB(Reg(Bool()))  
  regB := regA  
}
```

CLOCK CROSSING VIOLATION :

```
- Source          : (toplevel/regA : Bool) spinal.testers.code.Toplevel... (PresentationDsl.scala:1494)  
- Source clock    : (cdA_clk : Bool)  
- Destination     : (toplevel/regB : Bool) spinal.testers.code.Toplevel... (PresentationDsl.scala:1495)  
- Destination clock : (cdB_clk : Bool)
```

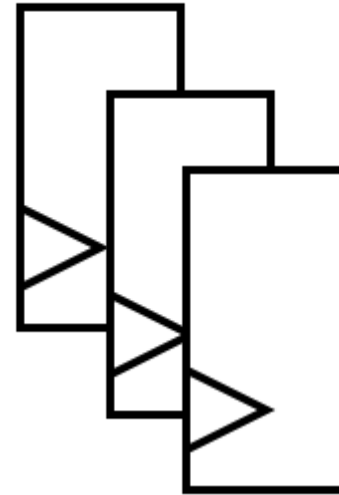
# Using software to elaborate your hardware



- Control flow : **if** / for
- Data collections : dynamic array / hash map / hash set
- Lambda function : reduce / fold / map / filter / ...
- OOP : class / software interface
- See <https://spinalhdl.github.io/NaxRiscv-Rtd/main/NaxRiscv/abstraction/index.html>

# Using software to elaborate your hardware

```
for(i <- 0 to 2){  
  Reg(UInt(8 bits))  
}
```



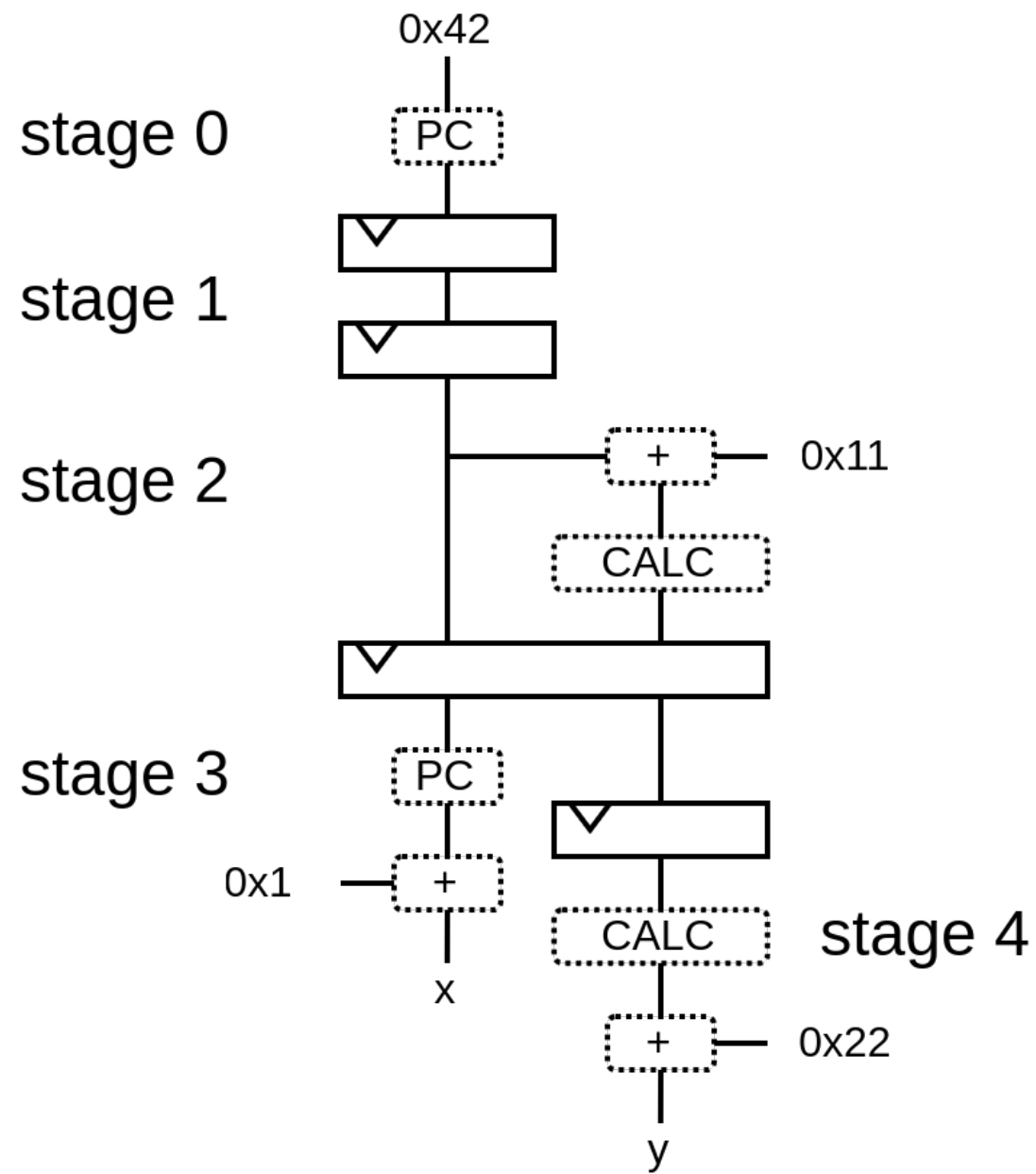
- Control flow : if / **for**
- Data collections : dynamic array / hash map / hash set
- Lambda function : reduce / fold / map / filter / ...
- OOP : class / software interface
- See <https://spinalhdl.github.io/NaxRiscv-Rtd/main/NaxRiscv/abstraction/index.html>

# Using software to elaborate your hardware

- Control flow : if / for
- Data collections : dynamic array / hash map / hash set
- Lambda function : reduce / fold / map / filter / ...
- OOP : class / software interface

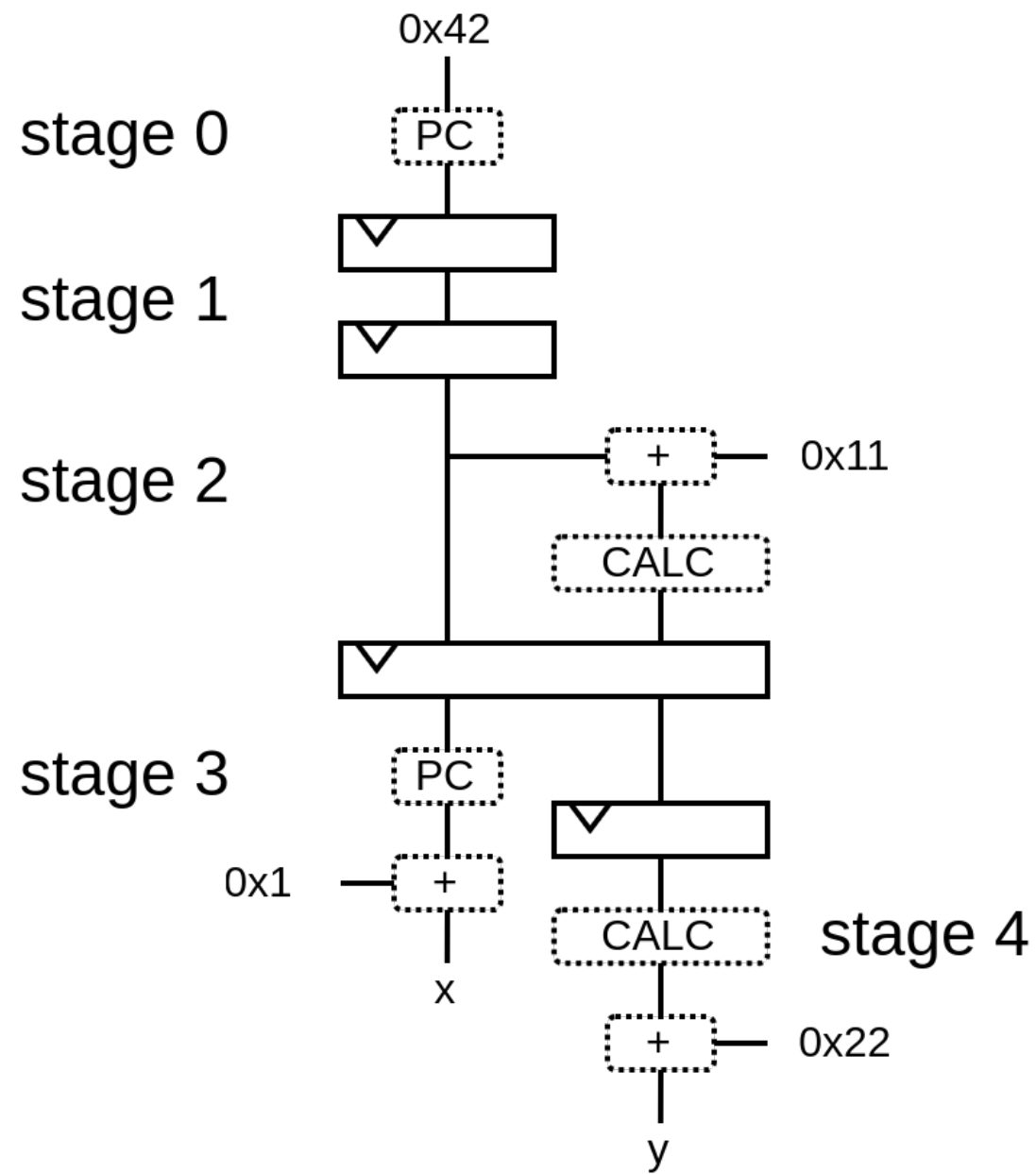


# Pipeline API



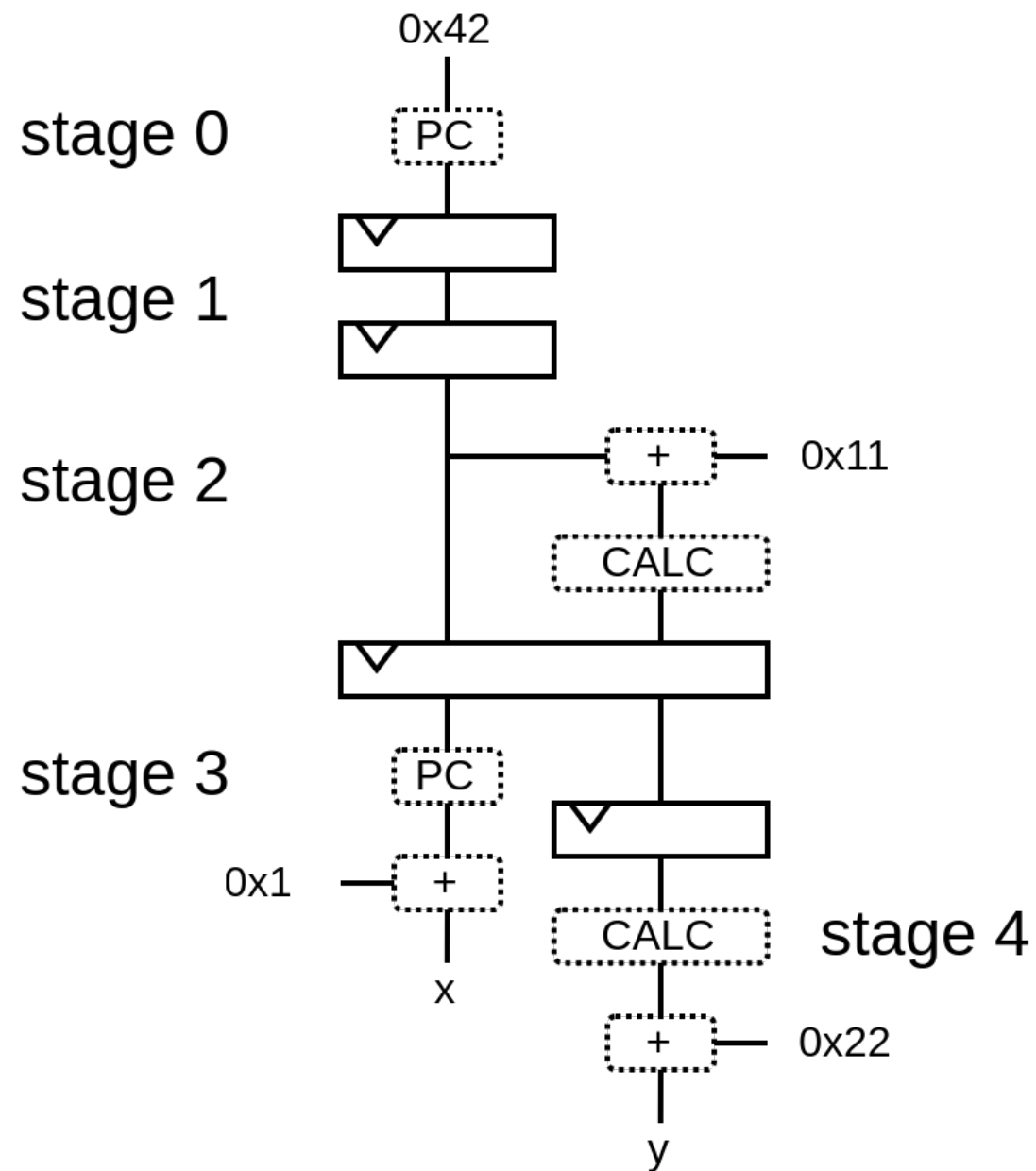
# Pipeline API

```
val pip = new Pipeline()
```



# Pipeline API

```
val pip = new Pipeline()  
  
val PC = NamedType(UInt(32 bits))  
pip(PC, 0) := 0x42
```



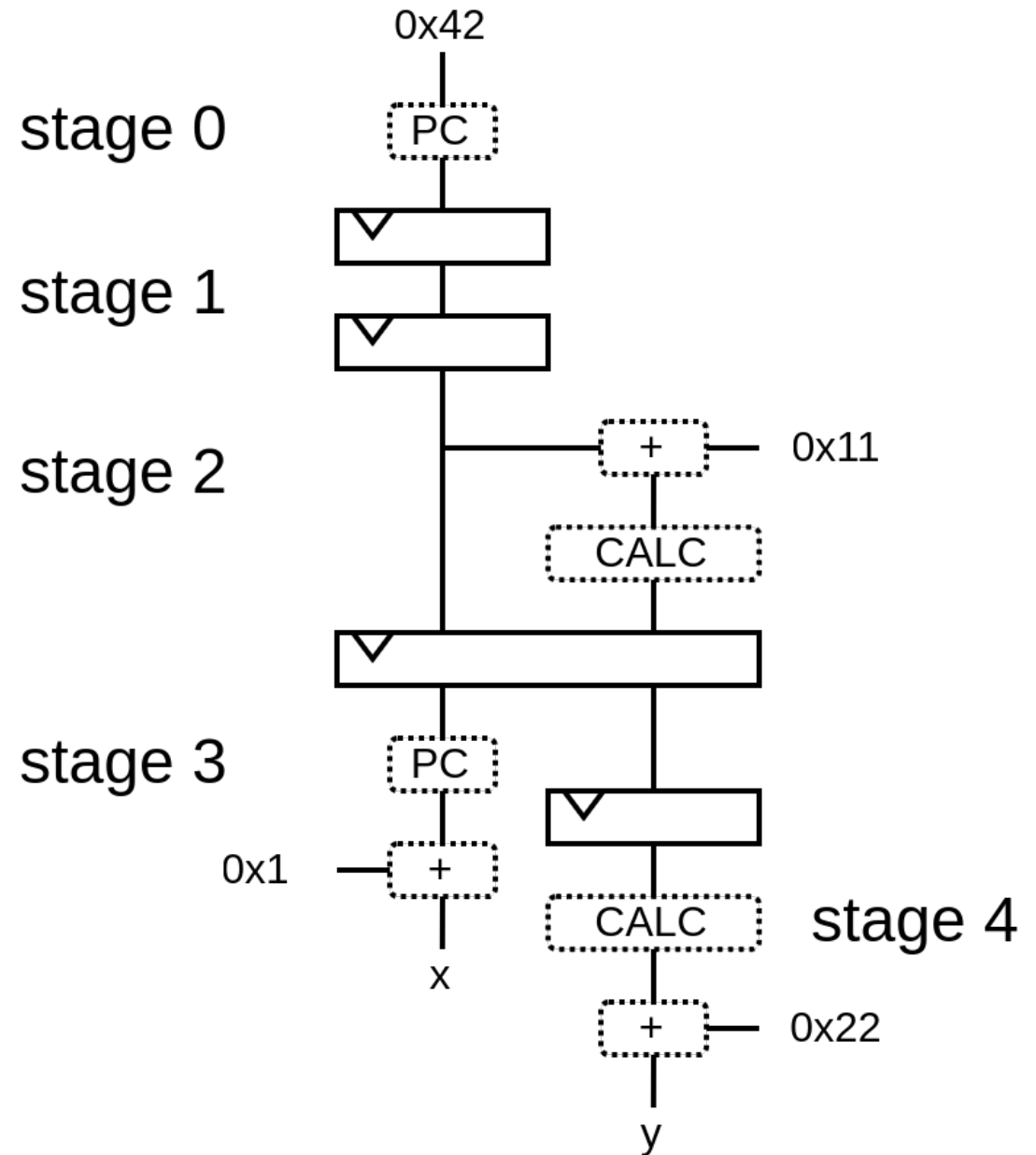


# Pipeline API

```
val pip = new Pipeline()

val PC = NamedType(UInt(32 bits))
pip(PC, 0) := 0x42

val CALC = NamedType(UInt(32 bits))
pip(CALC, 2) := pip(PC, 2) + 0x11
```



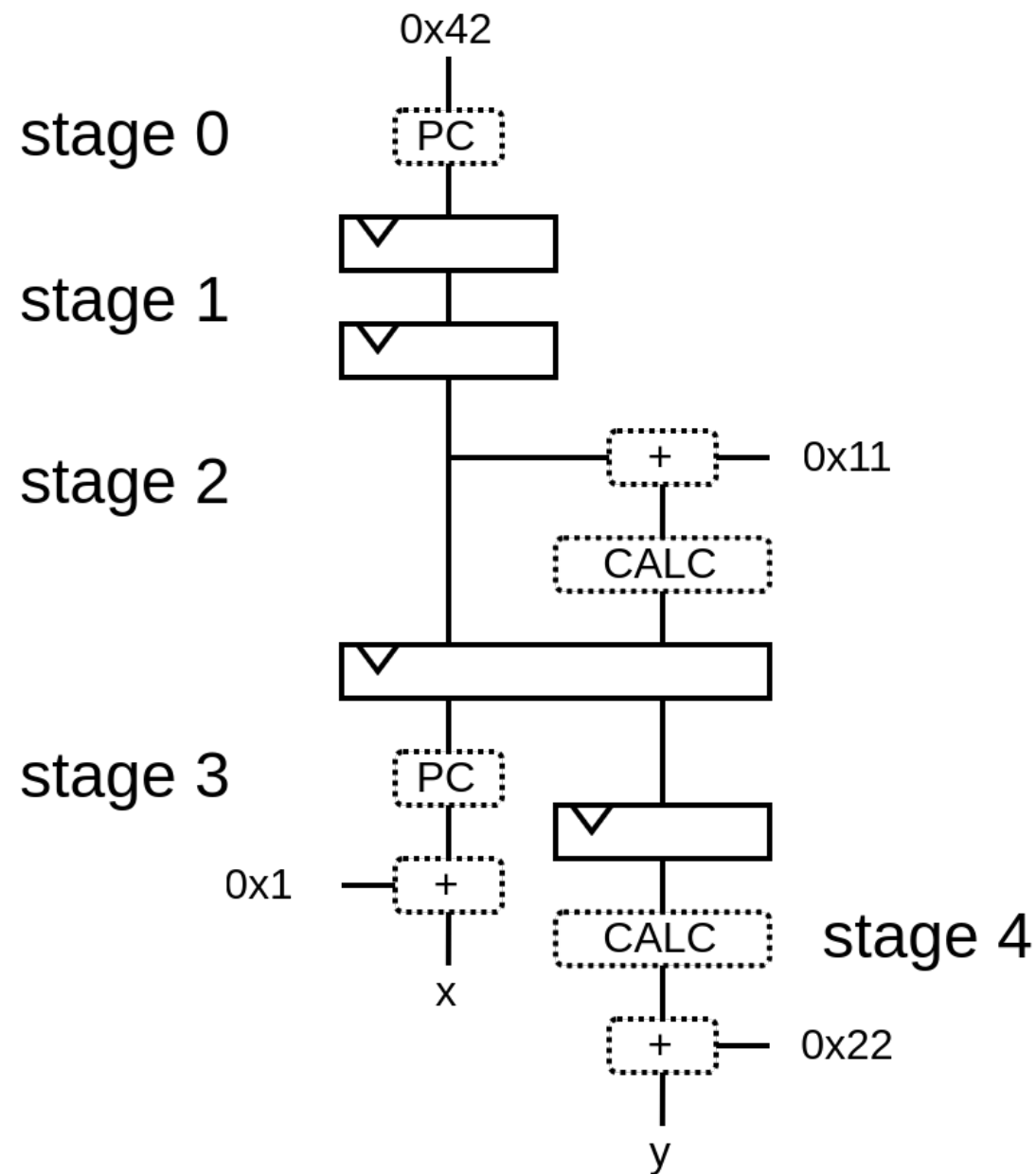
# Pipeline API

```
val pip = new Pipeline()

val PC = NamedType(UInt(32 bits))
pip(PC, 0) := 0x42

val CALC = NamedType(UInt(32 bits))
pip(CALC, 2) := pip(PC, 2) + 0x11

val x = pip(PC, 3) + 1
val y = pip(CALC, 4) + 0x22
```



# Pipeline API

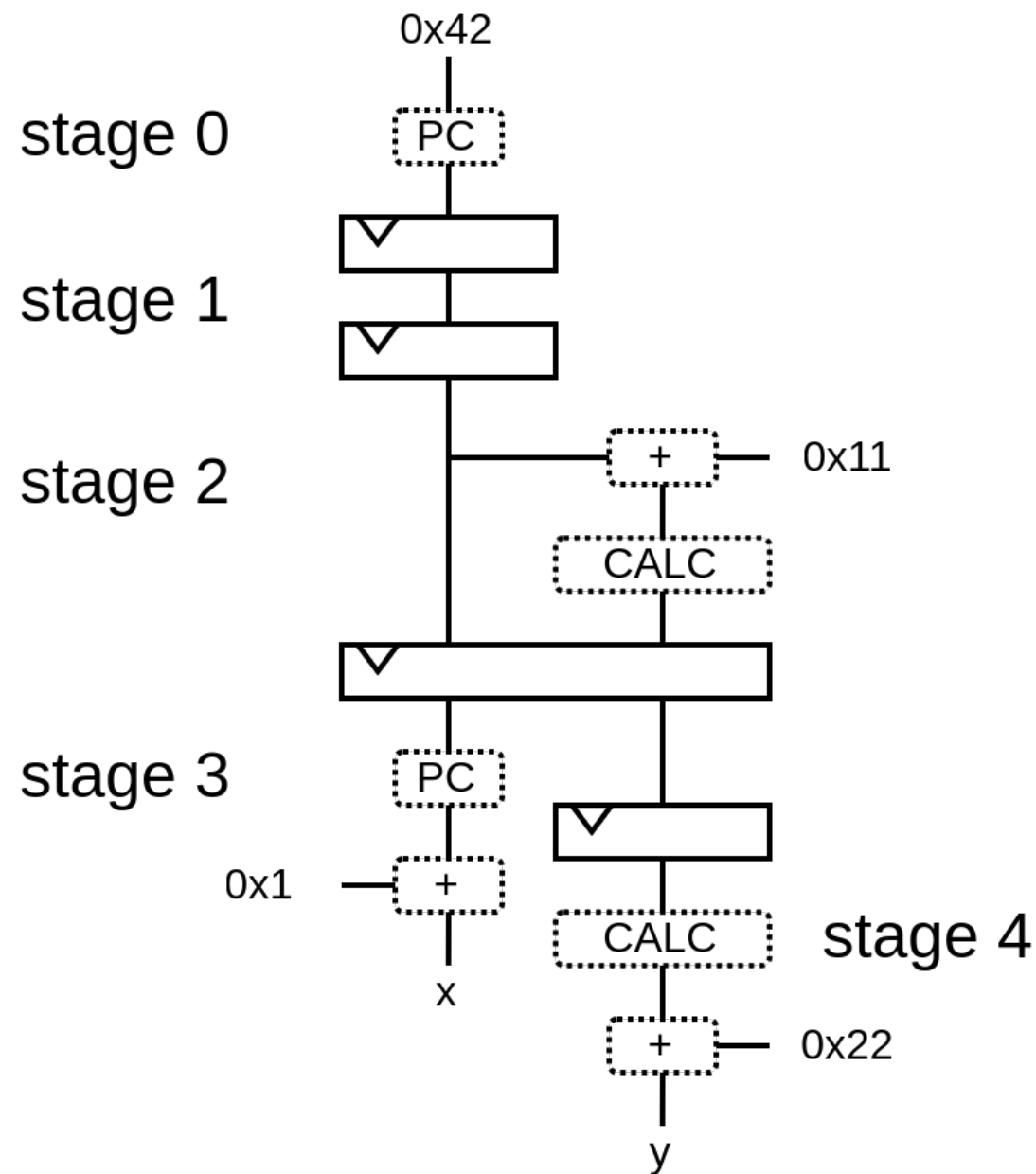
```
val pip = new Pipeline()

val PC = NamedType(UInt(32 bits))
pip(PC, 0) := 0x42

val CALC = NamedType(UInt(32 bits))
pip(CALC, 2) := pip(PC, 2) + 0x11

val x = pip(PC, 3) + 1
val y = pip(CALC, 4) + 0x22

pip.build()
```



# Pipeline API

```
val pip = new Pipeline()

val PC = NamedType(UInt(32 bits))
pip(PC, 0) := 0x42

val CALC = NamedType(UInt(32 bits))
pip(CALC, 2) := pip(PC, 2) + 0x11

val x = pip(PC, 3) + 1
val y = pip(CALC, 4) + 0x22

pip.build()
```

```
wire      [31:0]  PC_0;
wire      [31:0]  PC_3;
wire      [31:0]  x;
wire      [31:0]  CALC_2;
wire      [31:0]  PC_2;
wire      [31:0]  CALC_4;
wire      [31:0]  y;
wire      [31:0]  PC_1;
reg       [31:0]  PC_0_regNext;
reg       [31:0]  PC_1_regNext;
reg       [31:0]  PC_2_regNext;
wire      [31:0]  CALC_3;
reg       [31:0]  CALC_2_regNext;
reg       [31:0]  CALC_3_regNext;

assign PC_0 = 32'h00000042;
assign x = (PC_3 + 32'h00000001);
assign CALC_2 = (PC_2 + 32'h00000011);
assign y = (CALC_4 + 32'h00000022);
assign PC_1 = PC_0_regNext;
assign PC_2 = PC_1_regNext;
assign PC_3 = PC_2_regNext;
assign CALC_3 = CALC_2_regNext;
assign CALC_4 = CALC_3_regNext;
always @(posedge clk) begin
    PC_0_regNext <= PC_0;
    PC_1_regNext <= PC_1;
    PC_2_regNext <= PC_2;
    CALC_2_regNext <= CALC_2;
    CALC_3_regNext <= CALC_3;
end
```

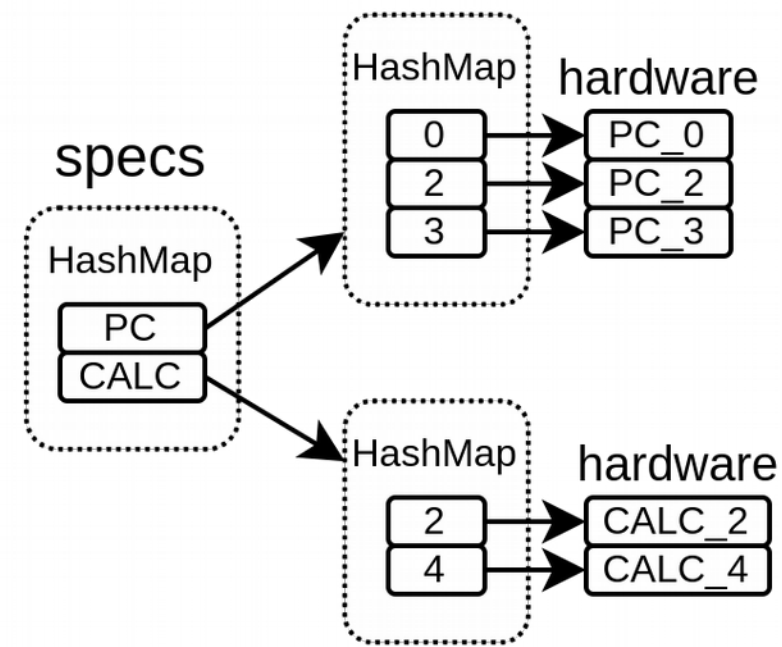
# Pipeline API

```
class Pipeline{
  //Define the pipeline data model
  val specs = LinkedHashMap[NamedType[Data], LinkedHashMap[Int, Data]]()

  //Define how we can access the pipeline
  def apply[T <: Data](what: NamedType[T], stageId: Int) = {
    val spec = specs.getOrElseUpdate(what.asInstanceOf[NamedType[Data]], new LinkedHashMap[Int, Data])
    spec.getOrElseUpdate(stageId, what().setName(what.getName + "_" + stageId)).asInstanceOf[T]
  }

  //Translate specs into hardware
  def build(): Unit = {
    for ((what, nodes) <- specs) {
      for (i <- nodes.keys.min until nodes.keys.max) {
        apply(what, i + 1) := RegNext(apply(what, i))
      }
    }
  }
}
```

# Pipeline API

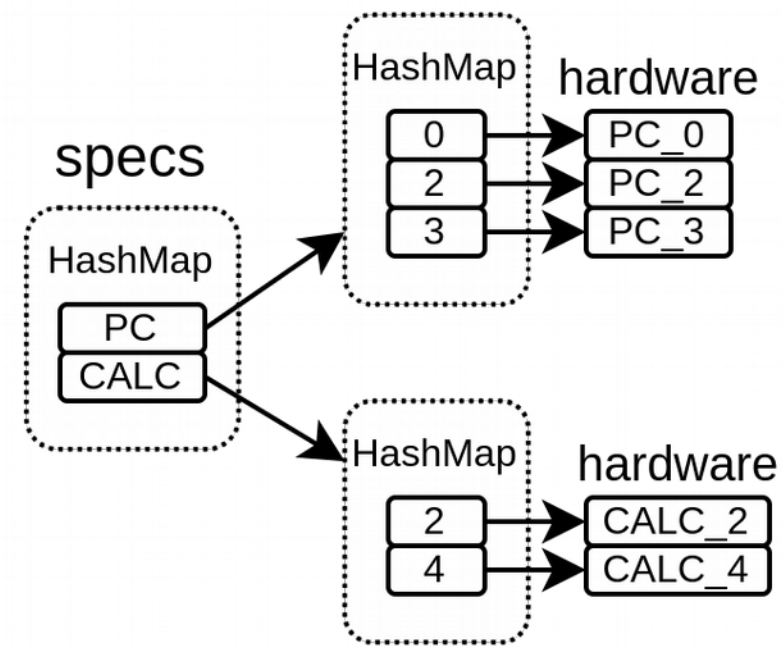


```
class Pipeline{
  //Define the pipeline data model
  val specs = LinkedHashMap[NamedType[Data], LinkedHashMap[Int, Data]]()

  //Define how we can access the pipeline
  def apply[T <: Data](what: NamedType[T], stageId: Int) = {
    val spec = specs.getOrElseUpdate(what.asInstanceOf[NamedType[Data]], new LinkedHashMap[Int, Data])
    spec.getOrElseUpdate(stageId, what().setName(what.getName + "_" + stageId)).asInstanceOf[T]
  }

  //Translate specs into hardware
  def build(): Unit = {
    for ((what, nodes) <- specs) {
      for (i <- nodes.keys.min until nodes.keys.max) {
        apply(what, i + 1) := RegNext(apply(what, i))
      }
    }
  }
}
```

# Pipeline API

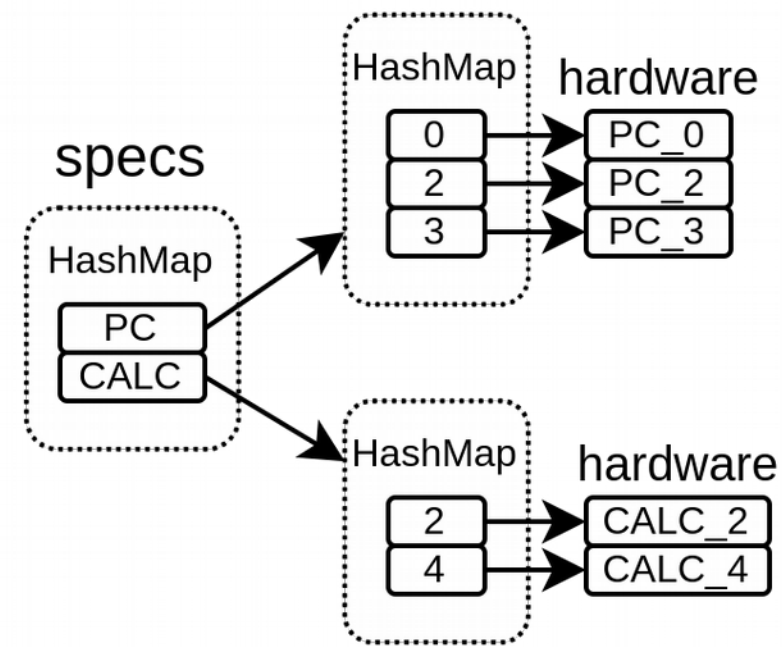


```
class Pipeline{
  //Define the pipeline data model
  val specs = LinkedHashMap[NamedType[Data], LinkedHashMap[Int, Data]]()

  //Define how we can access the pipeline
  def apply[T <: Data](what: NamedType[T], stageId: Int) = {
    val spec = specs.getOrElseUpdate(what.asInstanceOf[NamedType[Data]], new LinkedHashMap[Int, Data])
    spec.getOrElseUpdate(stageId, what().setName(what.getName + "_" + stageId)).asInstanceOf[T]
  }

  //Translate specs into hardware
  def build(): Unit = {
    for ((what, nodes) <- specs) {
      for (i <- nodes.keys.min until nodes.keys.max) {
        apply(what, i + 1) := RegNext(apply(what, i))
      }
    }
  }
}
```

# Pipeline API



```
class Pipeline{
  //Define the pipeline data model
  val specs = LinkedHashMap[NamedType[Data], LinkedHashMap[Int, Data]]()

  //Define how we can access the pipeline
  def apply[T <: Data](what: NamedType[T], stageId: Int) = {
    val spec = specs.getOrElseUpdate(what.asInstanceOf[NamedType[Data]], new LinkedHashMap[Int, Data])
    spec.getOrElseUpdate(stageId, what().setName(what.getName + "_" + stageId)).asInstanceOf[T]
  }

  //Translate specs into hardware
  def build(): Unit = {
    for ((what, nodes) <- specs) {
      for (i <- nodes.keys.min until nodes.keys.max) {
        apply(what, i + 1) := RegNext(apply(what, i))
      }
    }
  }
}
```



# Pipeline API

```
val pip = new Pipeline()

val PC = NamedType(UInt(32 bits))
pip(PC, 0) := 0x42

val CALC = NamedType(UInt(32 bits))
pip(CALC, 2) := pip(PC, 2) + 0x11

val x = pip(PC, 3) + 1
val y = pip(CALC, 4) + 0x22

pip.build()
```

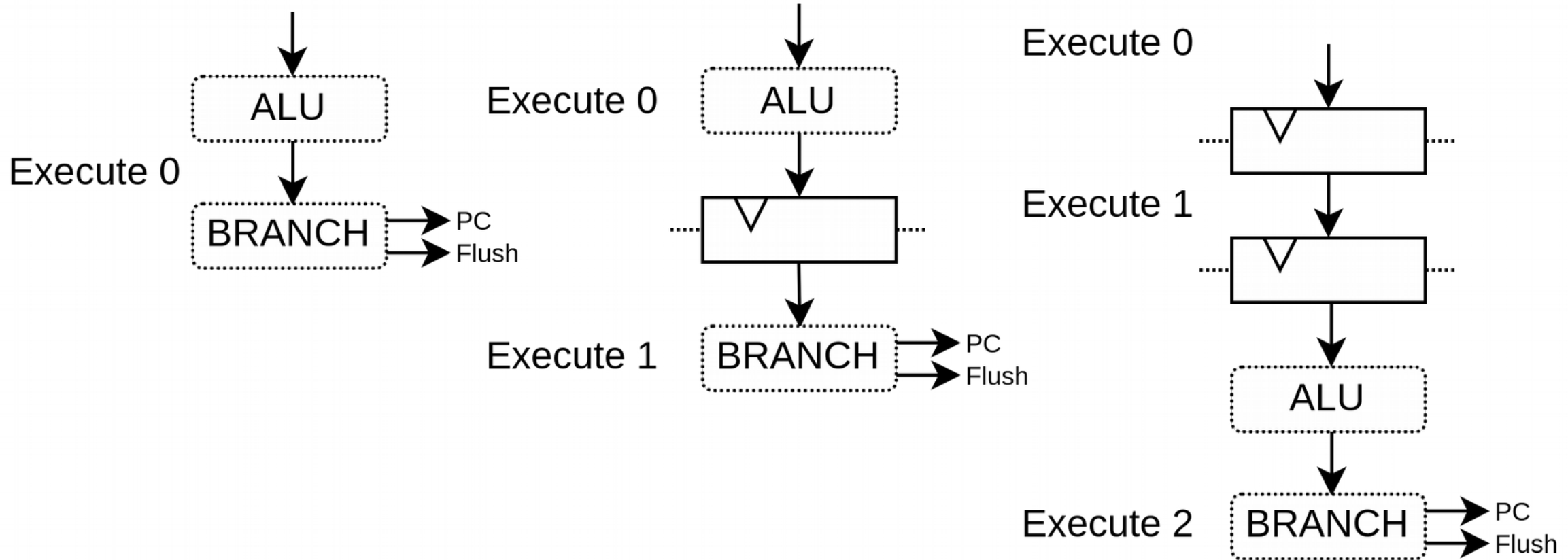
```
wire      [31:0]  PC_0;
wire      [31:0]  PC_3;
wire      [31:0]  x;
wire      [31:0]  CALC_2;
wire      [31:0]  PC_2;
wire      [31:0]  CALC_4;
wire      [31:0]  y;
wire      [31:0]  PC_1;
reg       [31:0]  PC_0_regNext;
reg       [31:0]  PC_1_regNext;
reg       [31:0]  PC_2_regNext;
wire      [31:0]  CALC_3;
reg       [31:0]  CALC_2_regNext;
reg       [31:0]  CALC_3_regNext;

assign PC_0 = 32'h00000042;
assign x = (PC_3 + 32'h00000001);
assign CALC_2 = (PC_2 + 32'h00000011);
assign y = (CALC_4 + 32'h00000022);
assign PC_1 = PC_0_regNext;
assign PC_2 = PC_1_regNext;
assign PC_3 = PC_2_regNext;
assign CALC_3 = CALC_2_regNext;
assign CALC_4 = CALC_3_regNext;
always @(posedge clk) begin
    PC_0_regNext <= PC_0;
    PC_1_regNext <= PC_1;
    PC_2_regNext <= PC_2;
    CALC_2_regNext <= CALC_2;
    CALC_3_regNext <= CALC_3;
end
```

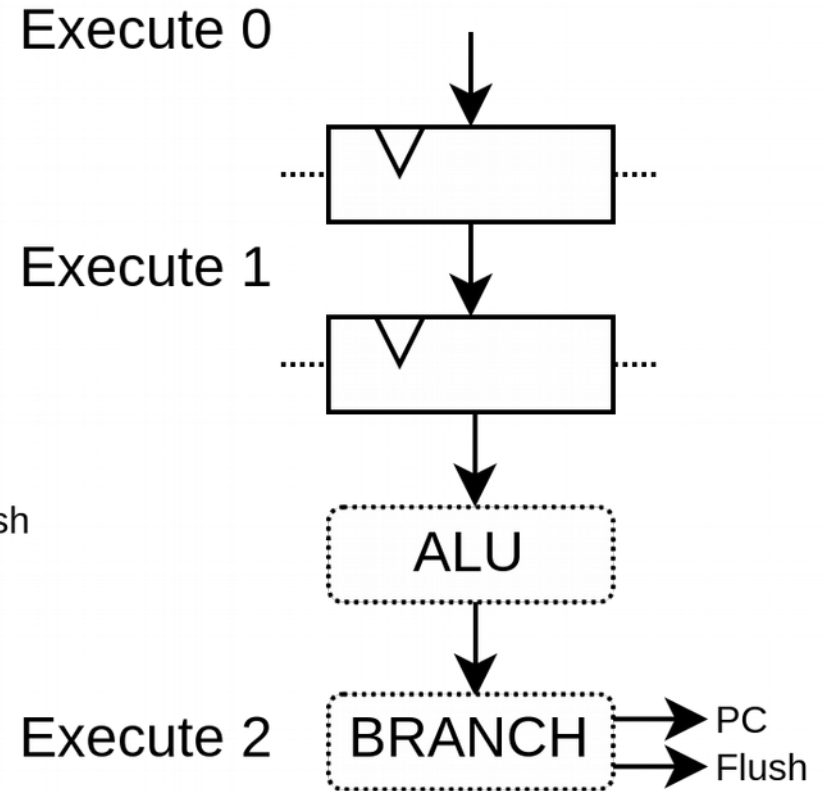
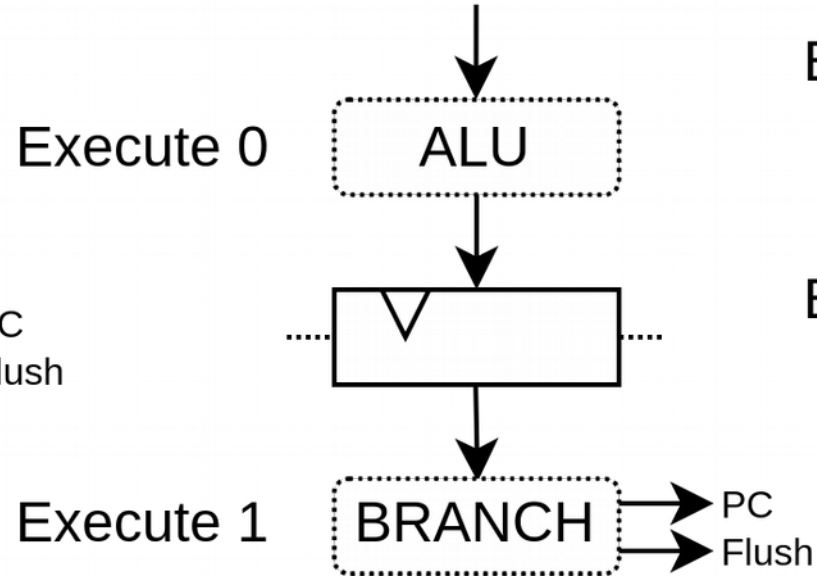
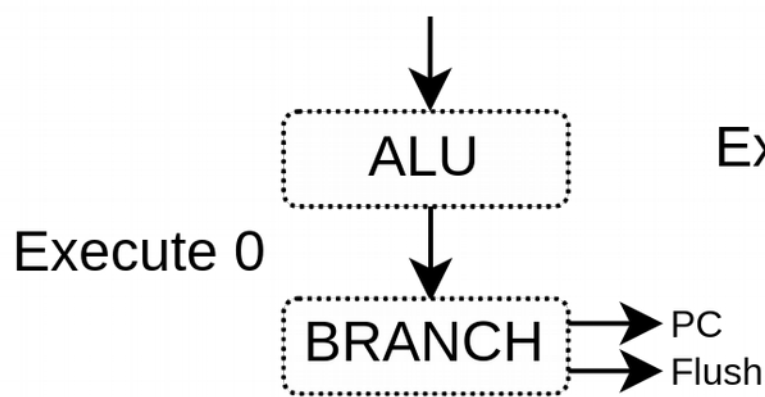
# VexiiRiscv

- RISC-V softcore
- Very large design space
- Can run linux (RV32 / RV64 IMACSU)
- Multiple issue
- Early + late ALU
- 5.24 coremark/Mhz 2.50 dhystone/Mhz
- <https://github.com/SpinalHDL/VexiiRiscv>

# VexiiRiscv : Pipeline API

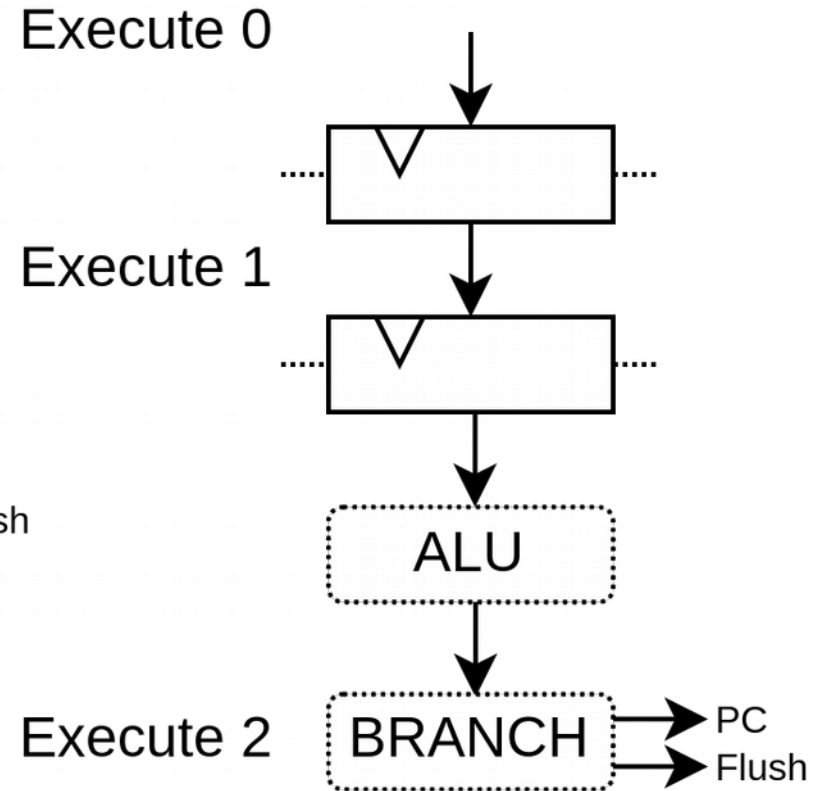
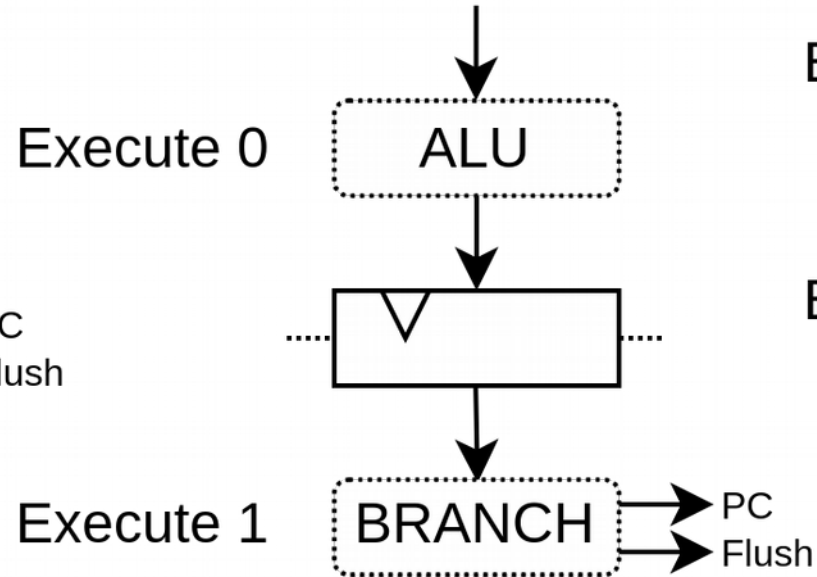
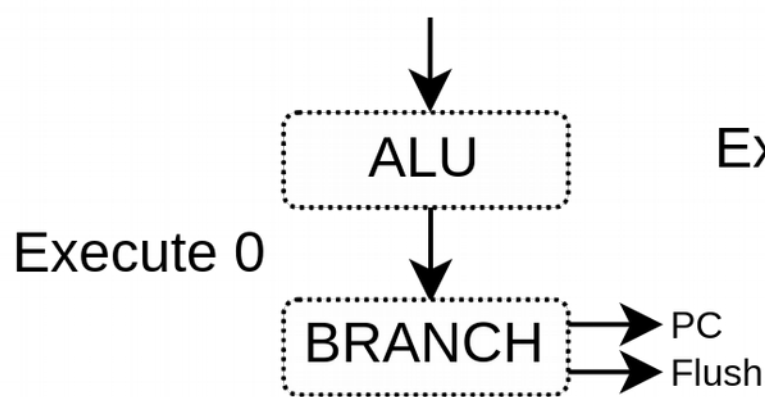


# VexiiRiscv : Pipeline API



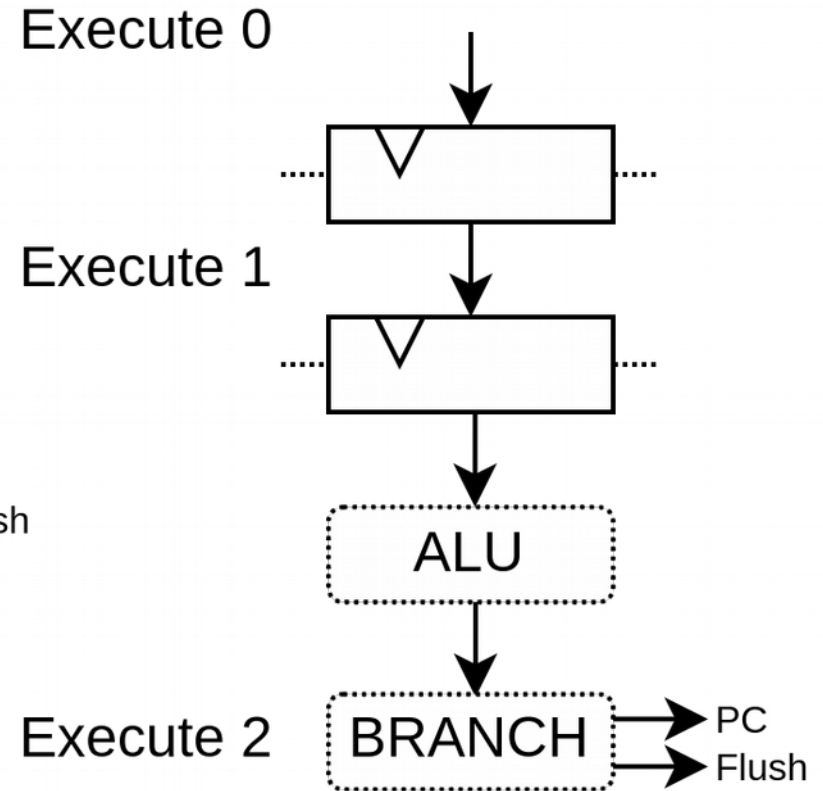
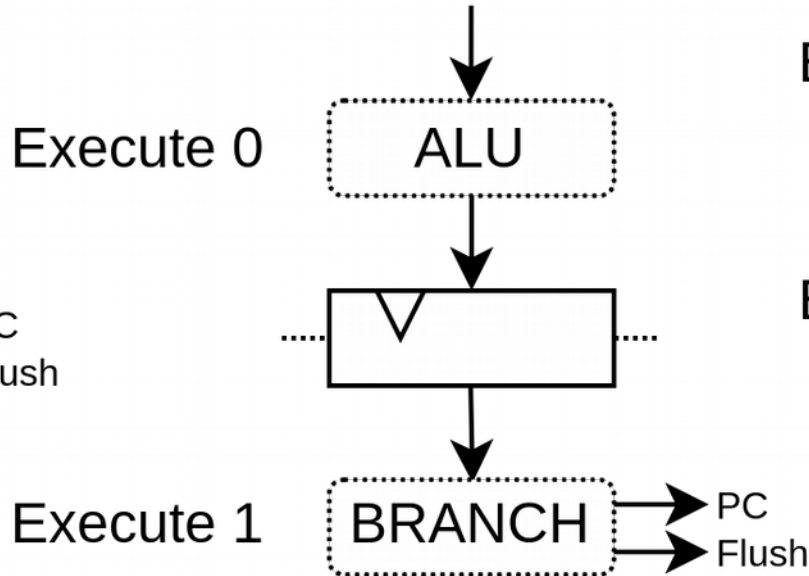
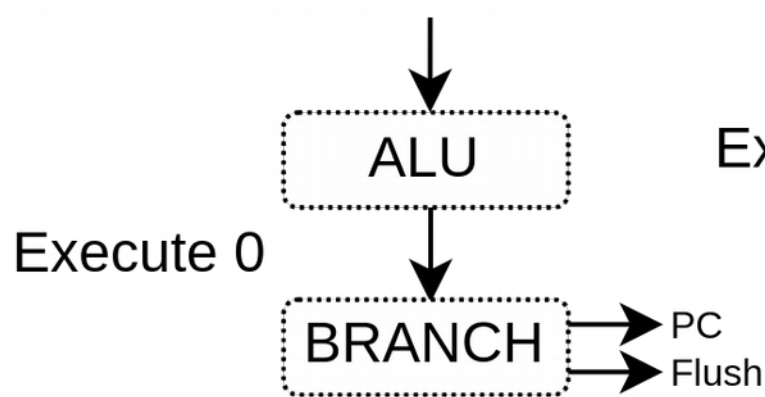
```
val alu = new pipeline.Execute(0) {  
  val TAKEN = insert(RS1 == RS2);  
}
```

# VexiiRiscv : Pipeline API



```
val alu = new pipeline.Execute(0) {  
  val TAKEN = insert(RS1 == RS2);  
}  
val branch = new pipeline.Execute(1) {  
  flushPort.valid := alu.TAKEN  
  
  pcPort.valid := alu.TAKEN  
  pcPort.pc := ..  
}
```

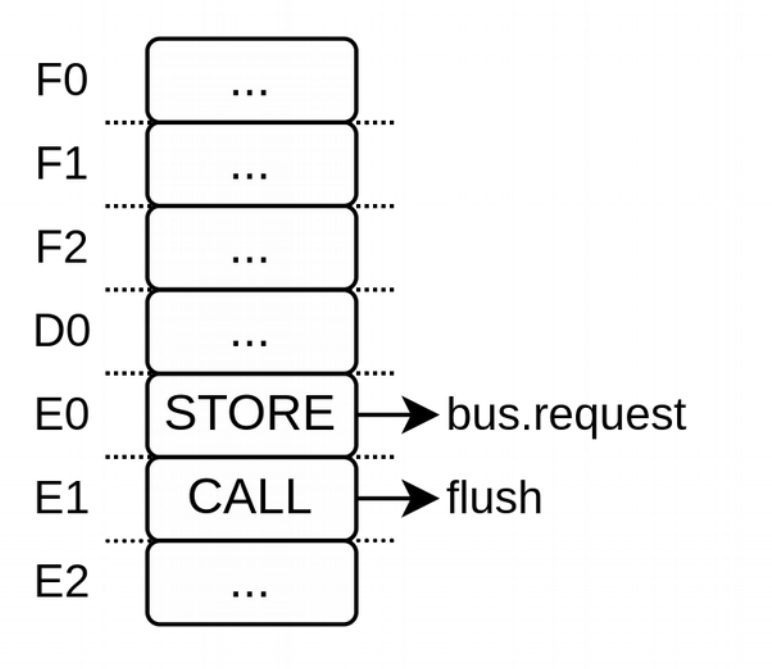
# VexiiRiscv : Pipeline API



```
val aluAt = 0
val branchAt = 1
...
val alu = new pipeline.Execute(aluAt) {
  val TAKEN = insert(RS1 == RS2);
}
val branch = new pipeline.Execute(branchAt) {
  flushPort.valid := alu.TAKEN

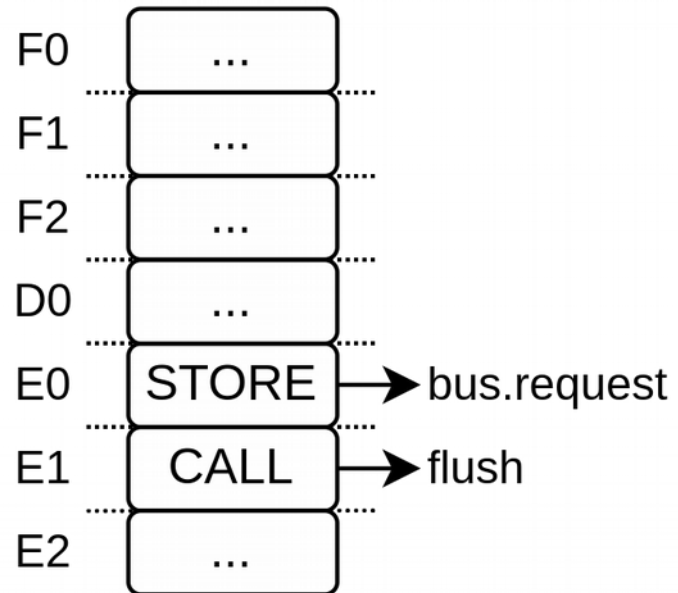
  pcPort.valid := alu.TAKEN
  pcPort.pc := ..
}
```

# VexiiRiscv : Decoding / Scheduling



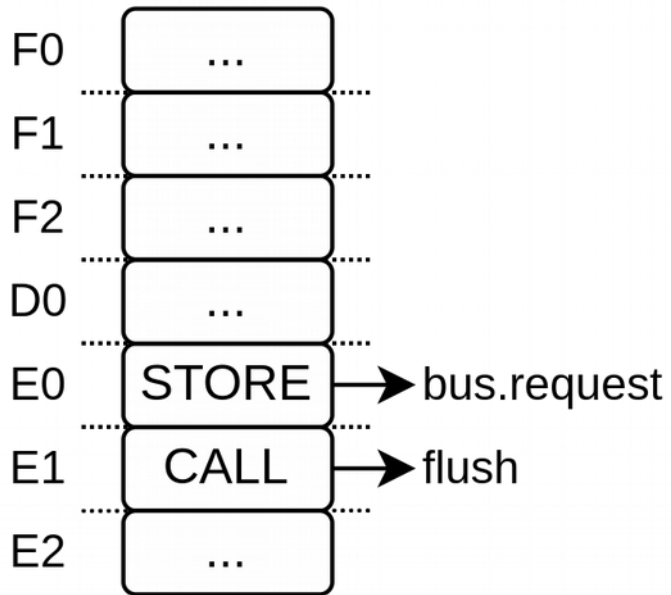
# VexiiRiscv : Specification structure

```
case class OpSpec(  
  encoding    : String,  
  mayFlush    : Boolean,  
  sideEffect  : Boolean,  
)
```



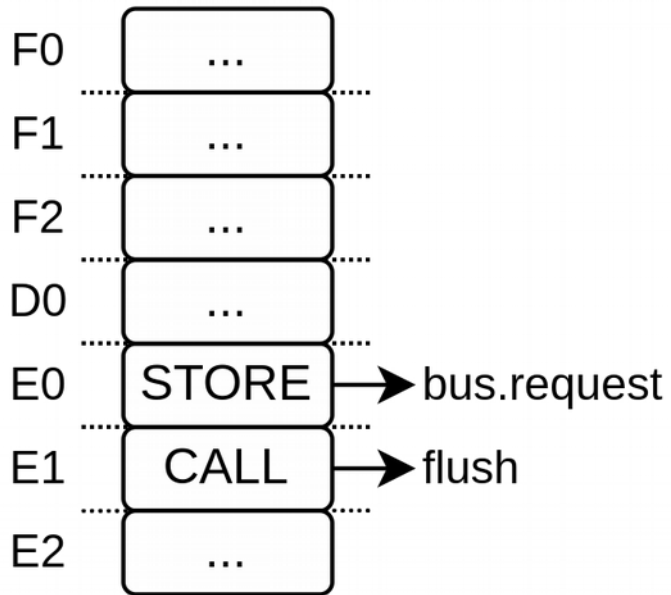


# VexiiRiscv : Define instructions spec



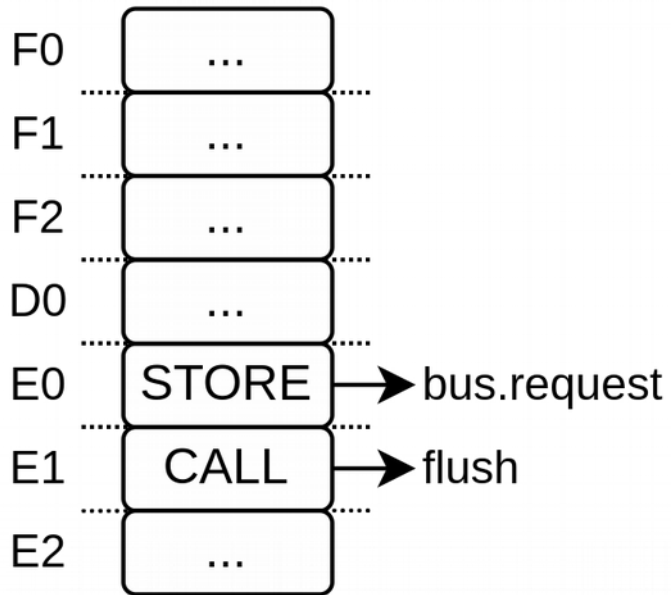
```
case class OpSpec(  
  encoding    : String,  
  mayFlush    : Boolean,  
  sideEffect  : Boolean,  
)  
  
val ADD = OpSpec("000-----0110011", false, false)  
val CALL = OpSpec("-----1101111", true, false)  
val SW = OpSpec("100-----1100011", false, true)
```

# VexiiRiscv : Collect the specifications



```
case class OpSpec(  
  encoding    : String,  
  mayFlush    : Boolean,  
  sideEffect  : Boolean,  
)  
  
val ADD = OpSpec("000-----0110011", false, false)  
val CALL = OpSpec("-----1101111", true, false)  
val SW = OpSpec("100-----1100011", false, true)  
  
val opsSpec = List(ADD, CALL, SW)
```

# VexiiRiscv : Decode

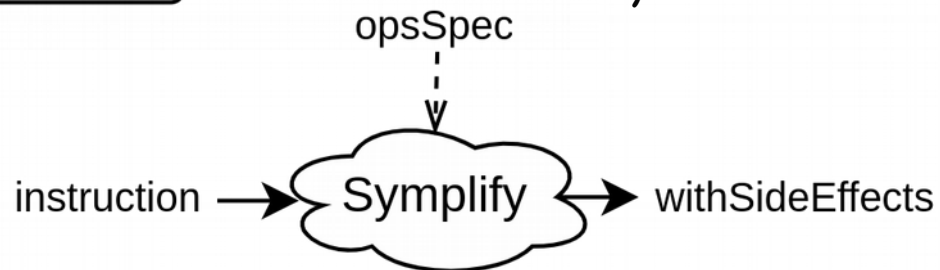


```
case class OpSpec(  
  encoding : String,  
  mayFlush : Boolean,  
  sideEffect: Boolean,  
)
```

```
val ADD  = OpSpec("000-----0110011", false, false)  
val CALL = OpSpec("-----1101111",  true, false)  
val SW   = OpSpec("100-----1100011", false, true)
```

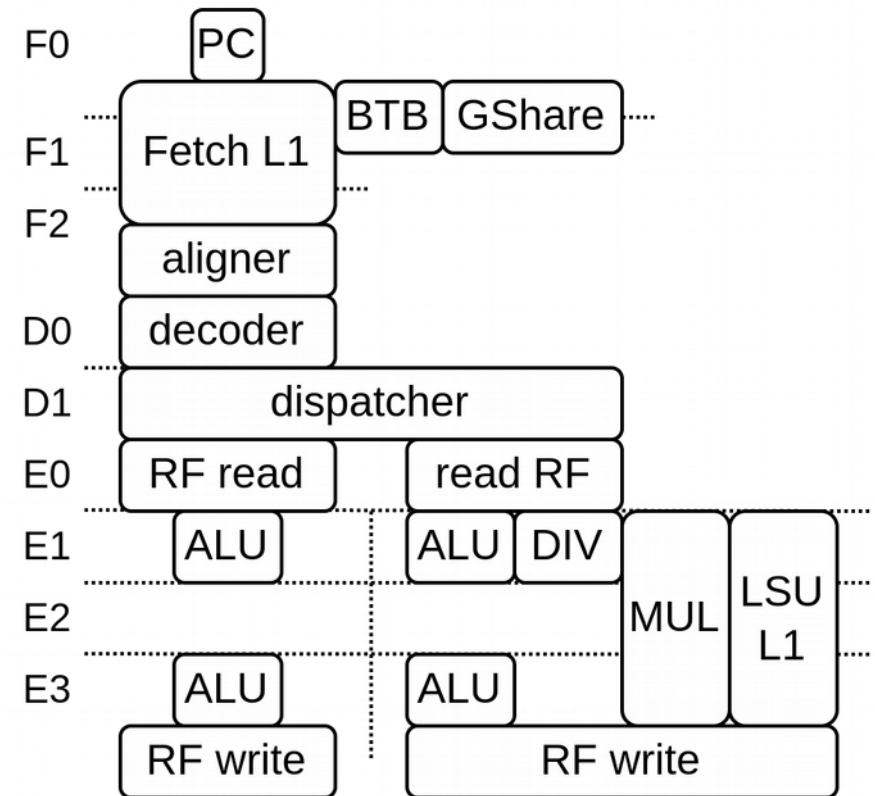
```
val opsSpec = List(ADD, CALL, SW)
```

```
val instruction = in Bits(32 bits)  
val withSideEffects = Simplify(  
  input = instruction,  
  trueTerms = opsSpec.filter(_._sideEffect).map(_._encoding),  
  falseTerms = opsSpec.filter(!_._sideEffect).map(_._encoding)  
)
```



# No more toplevel / design space

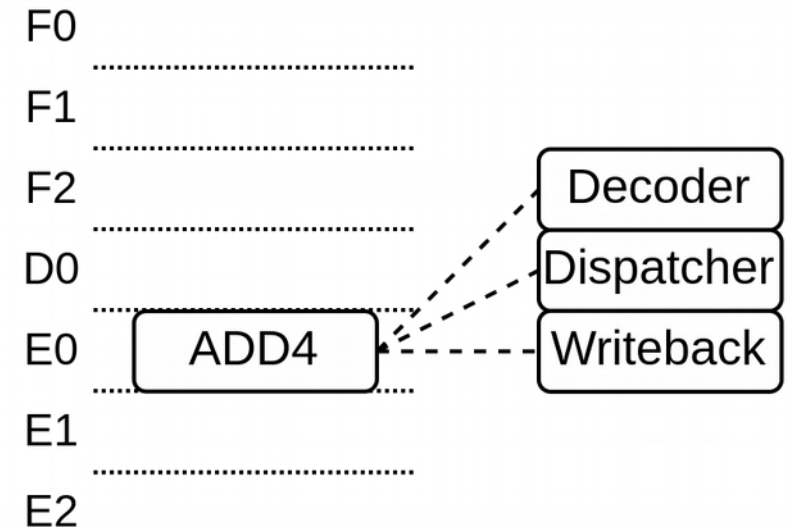
```
class VexiiRiscv(...) extends Component{  
  val database = ...  
  val host      = ...  
}  
  
val plugins = ArrayBuffer[Hostable]()  
plugins += new fetch.FetchPipelinePlugin()  
plugins += new fetch.PcPlugin(resetVector)  
plugins += new fetch.FetchL1Plugin(...)  
plugins += new prediction.BtbPlugin(...)  
plugins += new prediction.GSharePlugin(...)  
plugins += new prediction.HistoryPlugin(...)  
...  
plugins += new execute.SimdAddPlugin(...)  
  
val cpu = VexiiRiscv(plugins)
```



# Custom instruction

```
object SimdAddPlugin {
  val ADD4 = IntRegFile.TypeR(M"0000000-----000-----0001011")
}
class SimdAddPlugin(val layer: LaneLayer) extends ExecutionUnitElementSimple(layer) {
  val logic = during setup new Logic {
    awaitBuild()
    val wb = newWriteback(ifp, 0)
    val add4 = add(SimdAddPlugin.ADD4).spec
    add4.addRsSpec(RS1, executeAt = 0)
    add4.addRsSpec(RS2, executeAt = 0)
    uopRetainer.release()

    val process = new el.Execute(id = 0) {
      val rs1 = el(IntRegFile, RS1).asUInt
      val rs2 = el(IntRegFile, RS2).asUInt
      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)
      wb.valid := SEL
      wb.payload := rd.asBits
    }
  }
}
```



# How to lower barriers to hardware design ?

- VHDL / [System]Verilog alternatives
  - SpinalHDL, Chisel, Migen, Amaranth, ...
- Hardware design can leverage software engineering
  - API / Tooling / abstraction
  - New pool of people profiles
- Closed industry tools limits the scale of free/open-source hardware
  - Alternatives (Verilator, GHDL, IVerilog, openroad, ..)

Question ?