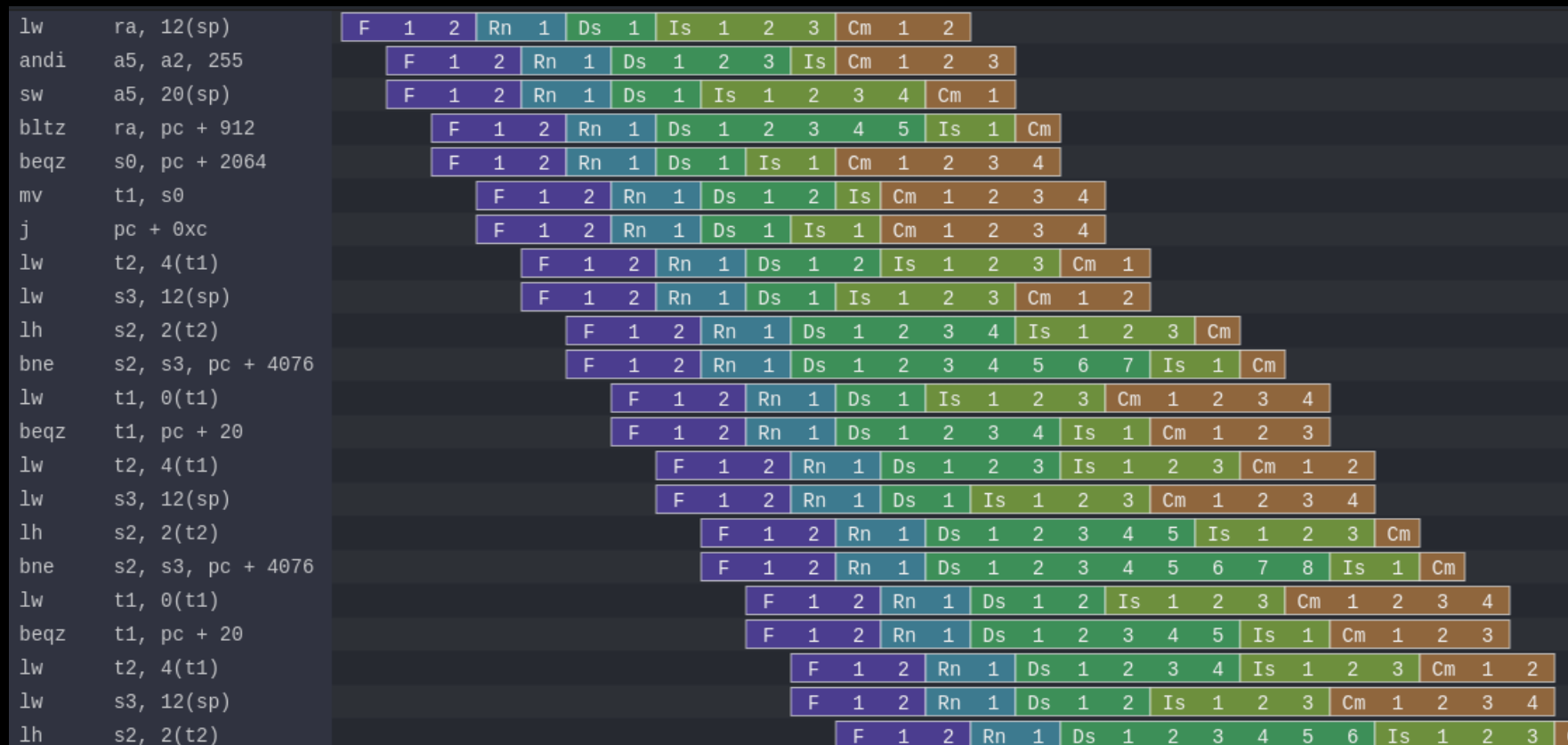


Composing an out-of-order CPU using software technics



NaxRiscv in short

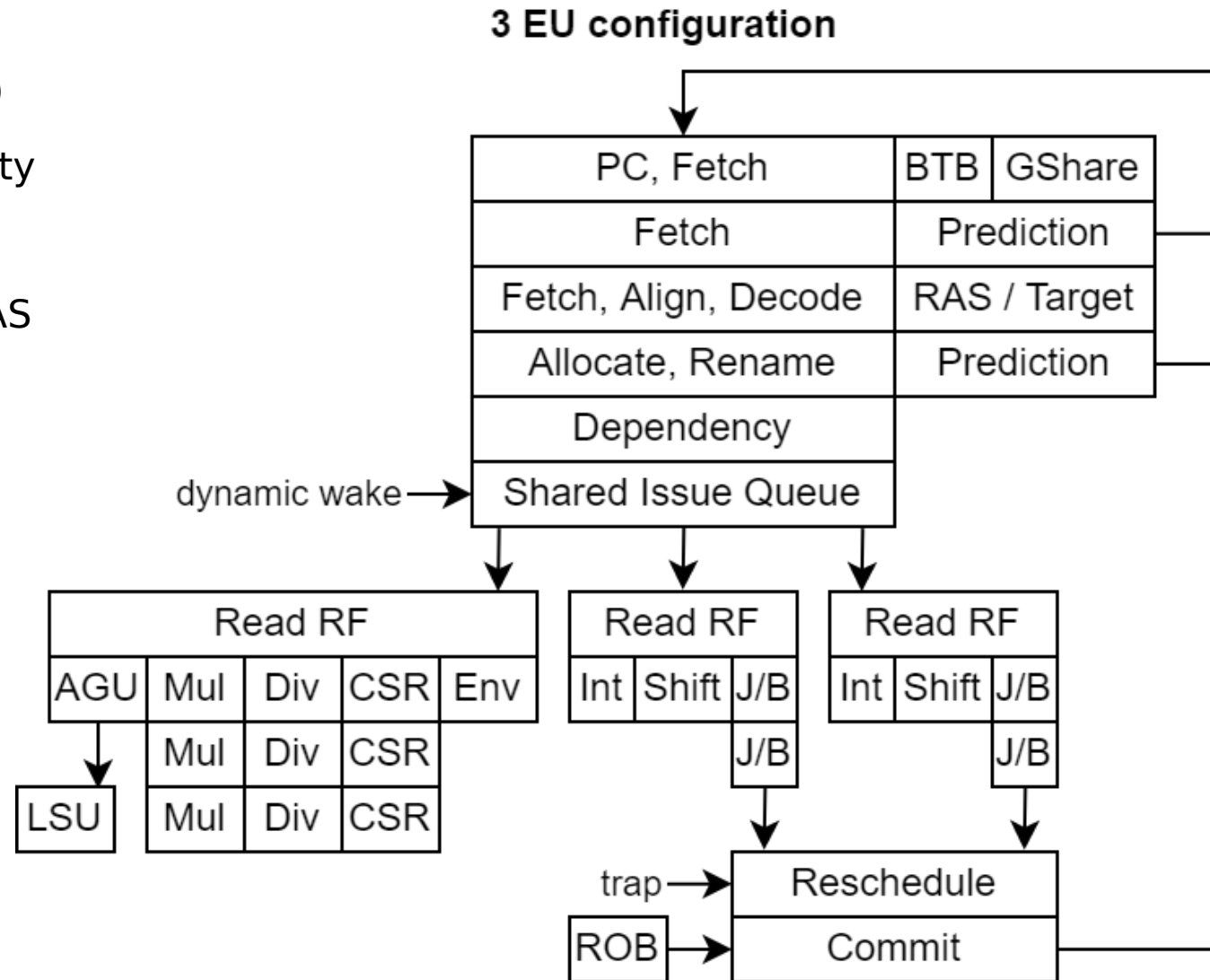
- RV32/RV64 IMAFDCSU (Linux ready)
- Out of order / superscalar
- <https://github.com/SpinalHDL/NaxRiscv>
- Not implemented in VHDL/[System]Verilog
- Software abstractions and elaboration



<https://twitter.com/i/status/1493996880593887235>

Architecture

- 2 decode, 3 issue (2 ALU, 1 shared EU)
- 10 cycles miss predicted branch penalty
- 64 physical register, 64 entry ROB
- 4KB GShare, 4KB BTB (both 1 way), RAS
- Non-blocking D\$
- I\$ D\$ each have 16 KB (4 ways)
- I\$ D\$ each have 192 TLB (2+4 ways)

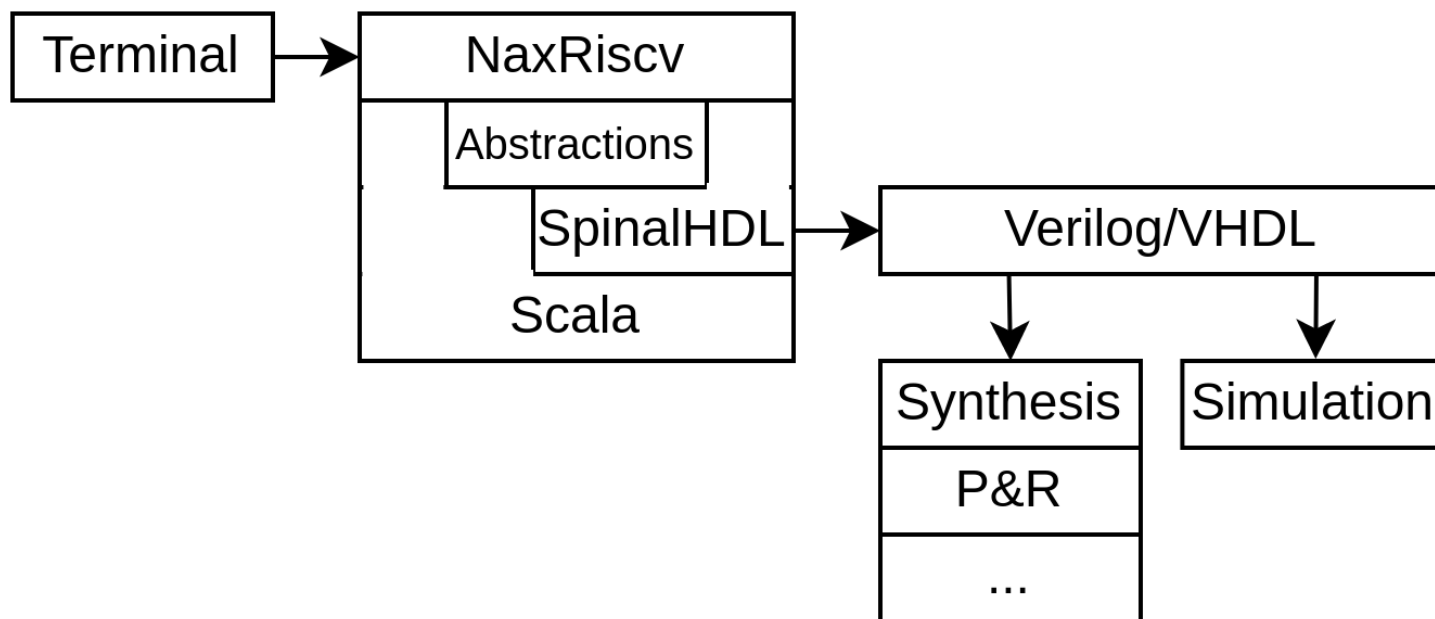


Abstraction != Overhead

- RV32IMASU, 2 decode, 3 issue (Linux ready)
 - Dhrystone : 2.94 DMIPS/Mhz 1.65 IPC (-O3 -fno-common -fno-inline)
 - Coremark : 5.00 Coremark/Mhz 1.28 IPC
 - Embench-iot : 1.68 baseline 1.42 IPC (baseline = Cortex M4)
- On Artix 7 speed grade 3 :
 - 150 Mhz (360 Mhz on Kintex ultrascale+ grade 3)
 - 14.2 KLUT, 9.7 KFF, 11.5 BRAM, 4 DSP

NaxRiscv elaboration

sbt "runMain naxriscv.Gen"



Scala in one slides

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

```
object Main extends App{  
  val verilog = new PrintWriter(new File("toplevel.v"))  
  verilog.write("module miaou{\n")  
  verilog.write("  input  clk,\n")  
  //...  
  verilog.close()  
}
```

Scala in one slides

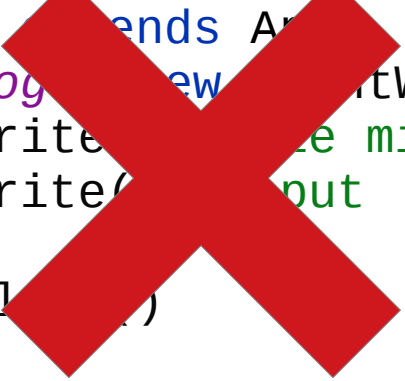
```
object Main extends App{  
  println("hello world")  
}
```

```
object Main extends App{  
  val verilog = new PrintWriter(new File("toplevel.v"))  
  verilog.write("module miaou{\n")  
  verilog.write("  input  clk,\n")  
  //...  
  verilog.close()  
}
```

Scala in one slides

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

```
object Main extends App{  
  val verilog = new PrintWriter(new File("toplevel.v"))  
  verilog.write("miaoou\n")  
  verilog.write("output clk,\n")  
  //...  
  verilog.close()  
}
```



SpinalHDL in two slides

```
import spinal.core._

object Main extends App{
  SpinalVerilog(
    new Module{
      val a,b = in UInt(8 bits)
      val result = out(a + b)
    }
  )
}
```



```
module unnamed (
  input      [7:0]  a,
  input      [7:0]  b,
  output     [7:0]  result
);
  assign result = (a + b);
endmodule
```

SpinalHDL in two slides

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

```
object Main extends App{  
  SpinalVerilog(  
    new Module{  
      val a,b = in UInt(8 bits)  
      val result = out(a + b)  
    }  
  )  
}
```



```
module unnamed (  
  input      [7:0]    a,  
  input      [7:0]    b,  
  output     [7:0]    result  
);  
  assign result = (a + b);  
endmodule
```

SpinalHDL – Scala interaction

```
new Module {  
  val a, b, c = out UInt(8 bits)
```

```
  val array = ArrayBuffer[UInt]()  
  array += a // By reference !  
  array += b  
  array += c
```

```
  for(element <- array){  
    element := 0  
  }  
}
```



```
module unnamed (  
  output [7:0] a,  
  output [7:0] b,  
  output [7:0] c  
);  
  assign a = 8'h0;  
  assign b = 8'h0;  
  assign c = 8'h0;  
endmodule
```

SpinalHDL – Scala interaction

```
new Module {  
  val a, b, c = out UInt(8 bits)  
  
  val array = ArrayBuffer[UInt]()  
  array += a // By reference !  
  array += b  
  array += c  
  
  for(element <- array){  
    element := 0  
  }  
}
```



```
module unnamed (  
  output [7:0] a,  
  output [7:0] b,  
  output [7:0] c  
);  
  assign a = 8'h0;  
  assign b = 8'h0;  
  assign c = 8'h0;  
endmodule
```

SpinalHDL – Scala interaction

```
new Module {  
  val a,b,c = out UInt(8 bits)
```

```
  val array = ArrayBuffer[UInt]()  
  array += a // By reference !  
  array += b  
  array += c
```

```
  for(element <- array){  
    element := 0  
  }  
}
```



```
module unnamed (  
  output [7:0] a,  
  output [7:0] b,  
  output [7:0] c  
);  
  assign a = 8'h0;  
  assign b = 8'h0;  
  assign c = 8'h0;  
endmodule
```

SpinalHDL – API side effects

```
new Module {  
  val a, b, c = out UInt(8 bits)
```

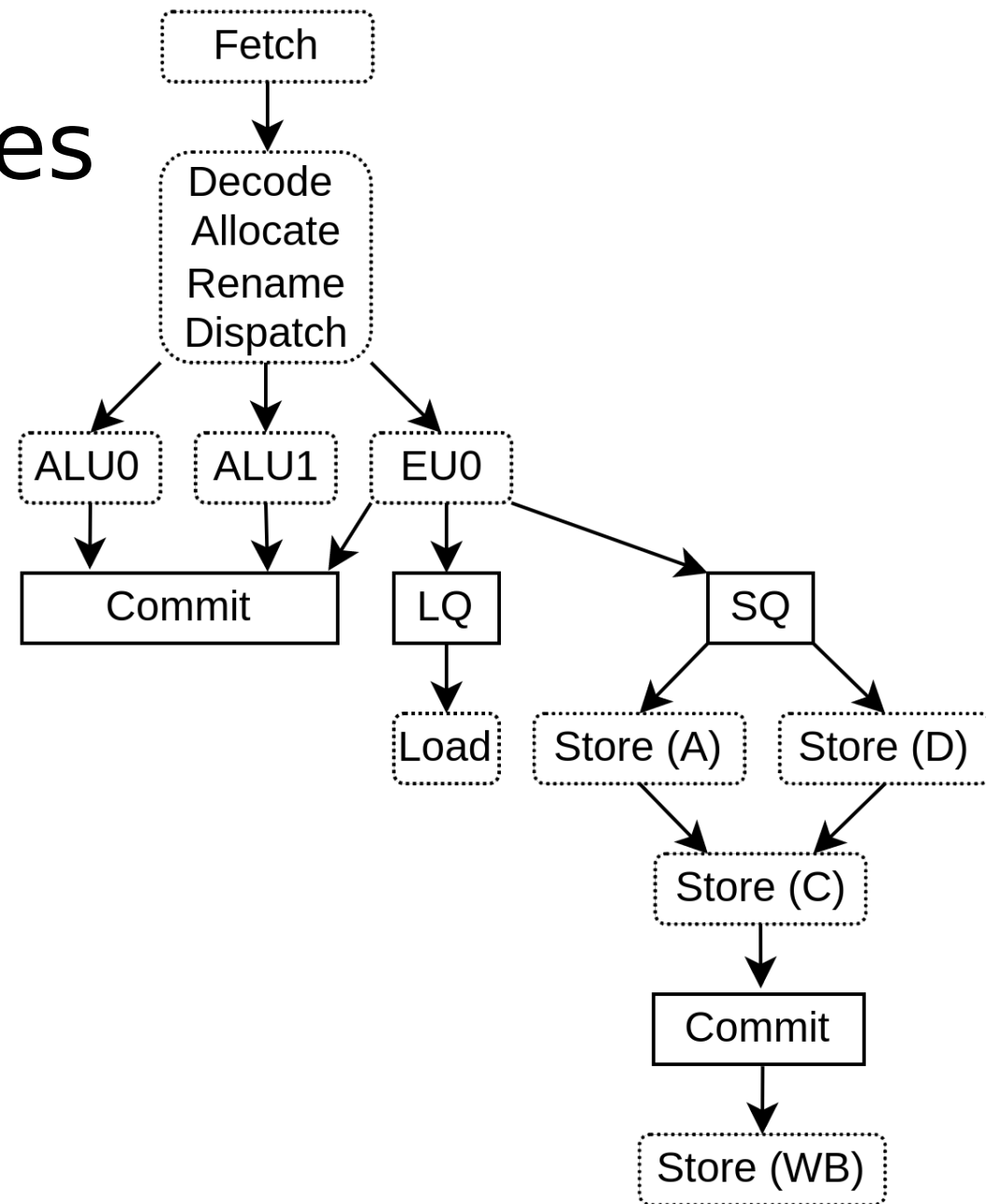
```
  val array = ArrayBuffer[UInt]()  
  array += a // By reference !  
  array += b  
  array += c
```

```
  for(element <- array){  
    element := 0  
  }  
}
```



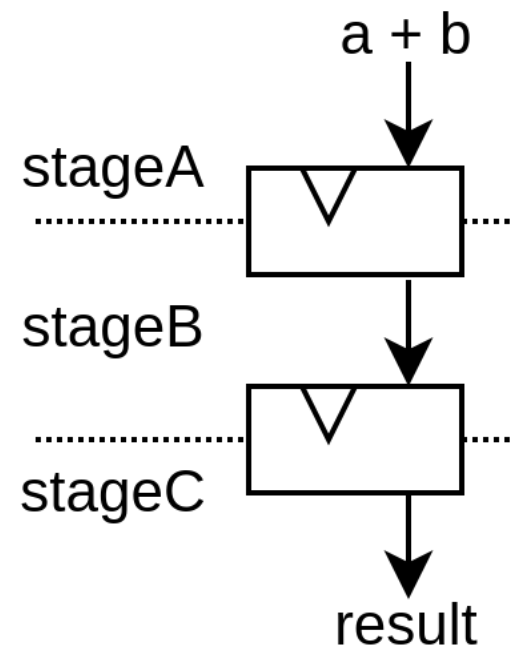
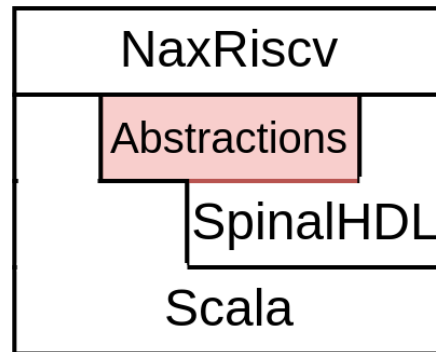
```
module unnamed (  
  output [7:0] a,  
  output [7:0] b,  
  output [7:0] c  
);  
  assign a = 8'h0;  
  assign b = 8'h0;  
  assign c = 8'h0;  
endmodule
```

NaxRiscv : Many pipelines



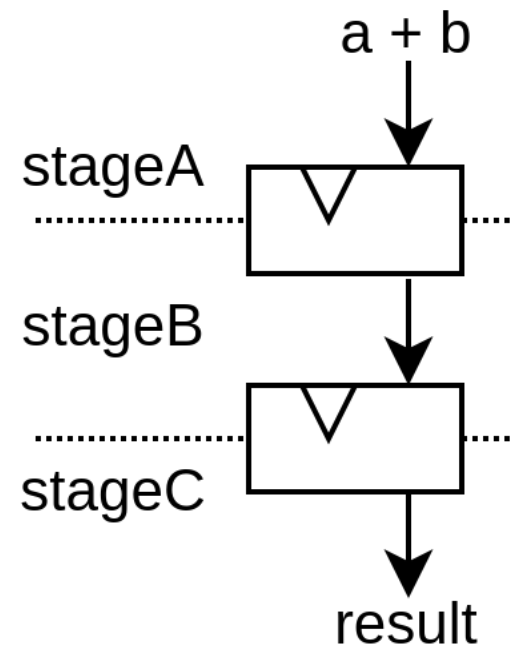
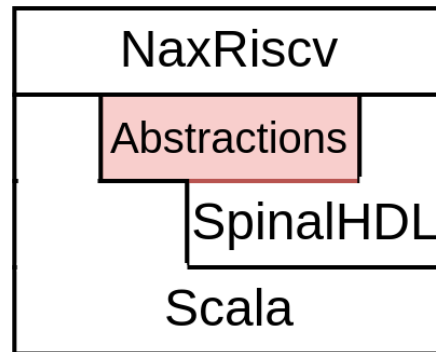
Pipeline API

```
new Module{  
  val a, b    = in  UInt(8 bits)  
  val result  = out UInt(8 bits)  
  
  implicit val pip = new Pipeline  
  val stageA = new Stage()  
  val stageB = new Stage(connection = M2S())  
  val stageC = new Stage(connection = M2S())  
  
  val SUM = stageA.insert(a + b)  
  result := stageC(SUM)  
  
  pip.build()  
}
```



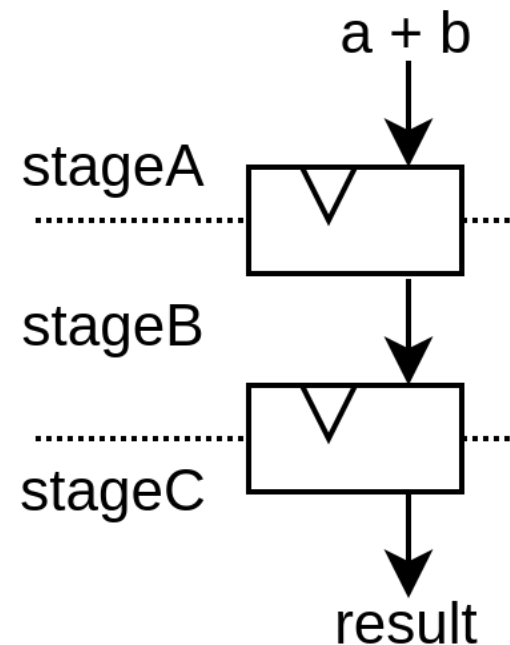
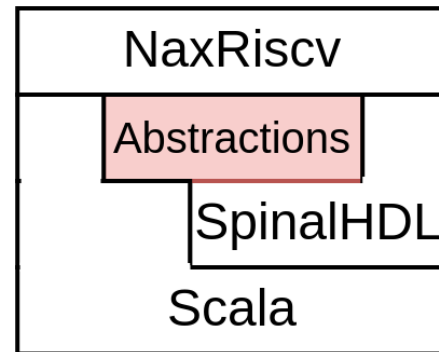
Pipeline API

```
new Module{  
  val a, b    = in  UInt(8 bits)  
  val result  = out UInt(8 bits)  
  
  implicit val pip = new Pipeline  
  val stageA = new Stage()  
  val stageB = new Stage(connection = M2S())  
  val stageC = new Stage(connection = M2S())  
  
  val SUM = stageA.insert(a + b)  
  result := stageC(SUM)  
  
  pip.build()  
}
```



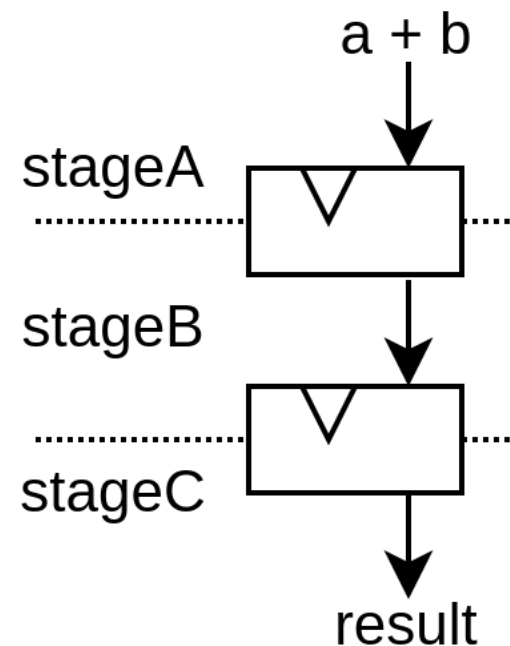
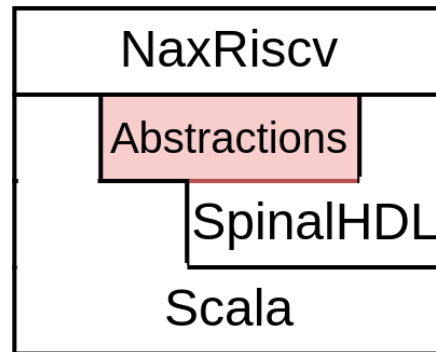
Pipeline API

```
new Module{  
  val a, b    = in  UInt(8 bits)  
  val result  = out UInt(8 bits)  
  
  implicit val pip = new Pipeline  
  val stageA = new Stage()  
  val stageB = new Stage(connection = M2S())  
  val stageC = new Stage(connection = M2S())  
  
  val SUM = stageA.insert(a + b)  
  result := stageC(SUM)  
  
  pip.build()  
}
```



Pipeline API

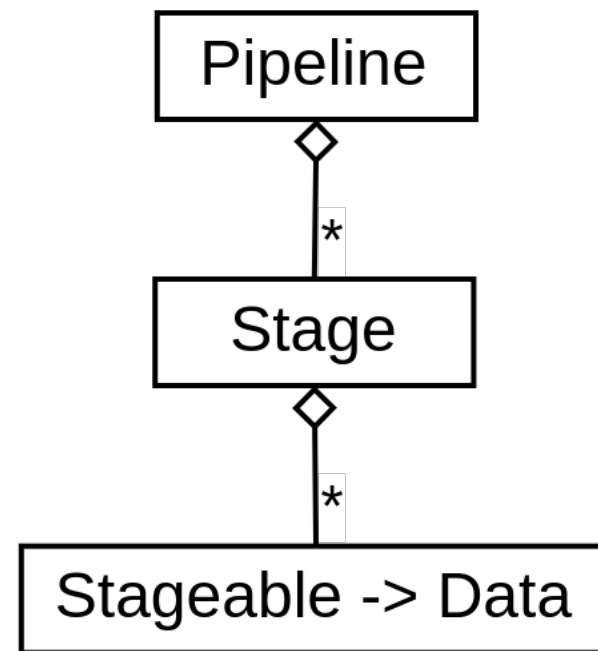
```
new Module{  
  val a, b    = in  UInt(8 bits)  
  val result  = out UInt(8 bits)  
  
  implicit val pip = new Pipeline  
  val stageA = new Stage()  
  val stageB = new Stage(connection = M2S())  
  val stageC = new Stage(connection = M2S())  
  
  val SUM = stageA.insert(a + b)  
  result := stageC(SUM)  
  
  pip.build()  
}
```



Pipeline internals

```
class Pipeline extends Area {  
  val stages = ArrayBuffer[Stage]()  
  //...  
  def build(): Unit = {  
    for(s <- stages){  
      for(... <- s.stageableToData){  
      }  
    }  
  }  
}
```

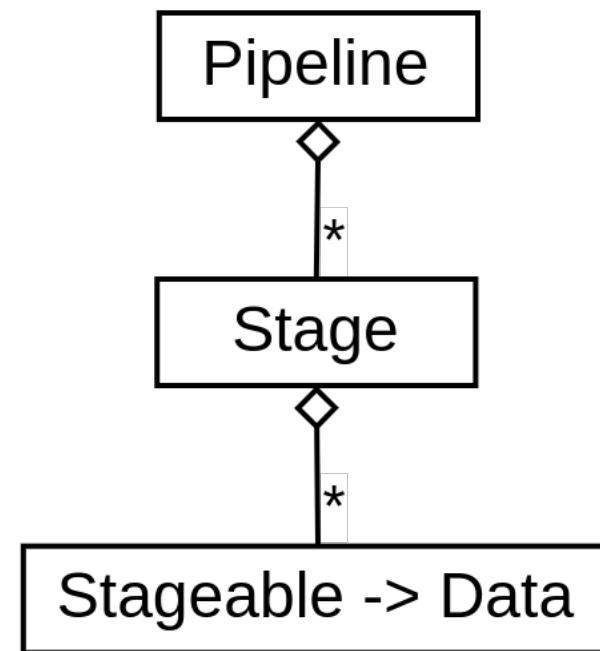
```
class Stage extends Area {  
  val stageableToData = HashMap[Stageable, Data]()  
  //...  
  pip.stages += this  
}
```



Pipeline internals

```
class Pipeline extends Area {  
  val stages = ArrayBuffer[Stage]()  
  //...  
  def build(): Unit = {  
    for(s <- stages){  
      for(... <- s.stageableToData){  
      }  
    }  
  }  
}
```

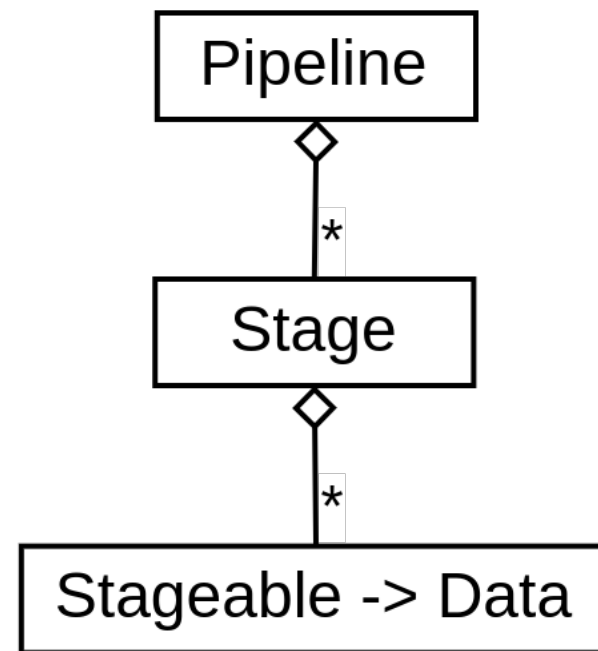
```
class Stage extends Area {  
  val stageableToData = HashMap[Stageable, Data]()  
  //...  
  pip.stages += this  
}
```



Pipeline internals

```
class Pipeline extends Area {  
  val stages = ArrayBuffer[Stage]()  
  //...  
  def build(): Unit = {  
    for(s <- stages){  
      for(... <- s.stageableToData){  
      }  
    }  
  }  
}
```

```
class Stage extends Area {  
  val stageableToData = HashMap[Stageable, Data]()  
  //...  
  pip.stages += this  
}
```



```
val plugins = ArrayBuffer[Plugin]()

plugins += List(
    new PcPlugin(),
    new FetchCachePlugin(16 KB),
    new DecoderPlugin(),
    new DispatchPlugin(slotCount = 32),
    new CommitPlugin()
    ...
)

plugins += List(
    new ExecutionUnitBase("ALU0"),
    new IntAluPlugin("ALU0"),
    new ShiftPlugin("ALU0"),
    new BranchPlugin("ALU0"),
    new IntFormatPlugin("ALU0"),
    new SrcPlugin("ALU0"),
)

new NaxRiscv(plugins) ← Create NaxRiscv
```

`val plugins = ArrayBuffer[Plugin]()` ← Elaboration time list of Plugin

```
plugins += List(  
    new PcPlugin(),  
    new FetchCachePlugin(16 KB),  
    new DecoderPlugin(),  
    new DispatchPlugin(slotCount = 32),  
    new CommitPlugin()  
    ...  
)
```

```
plugins += List(  
    new ExecutionUnitBase("ALU0"),  
    new IntAluPlugin("ALU0"),  
    new ShiftPlugin("ALU0"),  
    new BranchPlugin("ALU0"),  
    new IntFormatPlugin("ALU0"),  
    new SrcPlugin("ALU0"),  
)
```

```
new NaxRiscv(plugins)
```



```
val plugins = ArrayBuffer[Plugin]()
```

```
plugins += List(  
    new PcPlugin(),  
    new FetchCachePlugin(16 KB),  
    new DecoderPlugin(),  
    new DispatchPlugin(slotCount = 32),  
    new CommitPlugin()  
    ...  
)
```

```
plugins += List(  
    new ExecutionUnitBase("ALU0"),  
    new IntAluPlugin("ALU0"),  
    new ShiftPlugin("ALU0"),  
    new BranchPlugin("ALU0"),  
    new IntFormatPlugin("ALU0"),  
    new SrcPlugin("ALU0"),  
)
```

```
new NaxRiscv(plugins)
```

```
plugins += List(  
    new ExecutionUnitBase("ALU1"),  
    new IntAluPlugin("ALU1"),  
    new ShiftPlugin("ALU1"),  
    new BranchPlugin("ALU1"),  
    new IntFormatPlugin("ALU1"),  
    new SrcPlugin("ALU1"),  
)
```

Wanna a second execution unit ?



Plugin - Inheritance

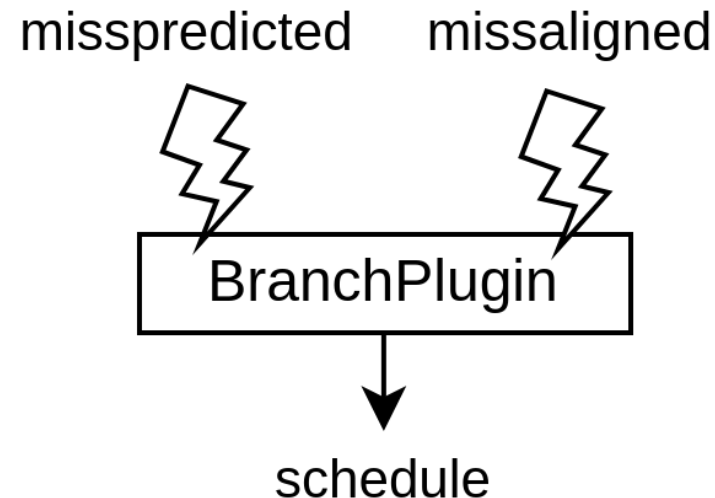
```
class BranchPlugin(euId : String) extends Plugin {  
    val setup = create early new Area{  
        // ...  
    }  
  
    val logic = create late new Area{  
        // ...  
    }  
}
```

Plugin - Phases

```
class BranchPlugin(euId : String) extends Plugin {  
    val setup = create early new Area{  
        // ...  
    }  
  
    val logic = create late new Area{  
        // ...  
    }  
}
```

Plugin - Discovery

```
class BranchPlugin(euId : String) extends Plugin {  
  val setup = create early new Area{  
    // ...  
    val commit = getService[CommitService]  
    val schedule = commit.newSchedulePort(canJump = true, canTrap = true)  
  }  
  
  val logic = create late new Area{  
    // ...  
    setup.schedule.valid := isFireing && SEL && (MISSPREDICTED || MISSALIGNED)  
    setup.schedule.trap := MISSALIGNED  
    setup.schedule.pcTarget := target  
  }  
}
```

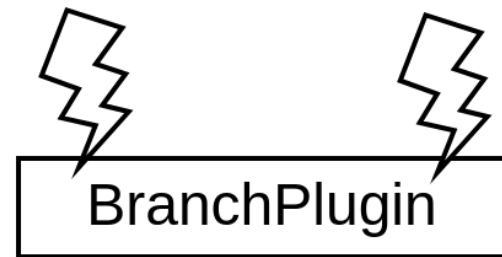


Plugin - Discovery

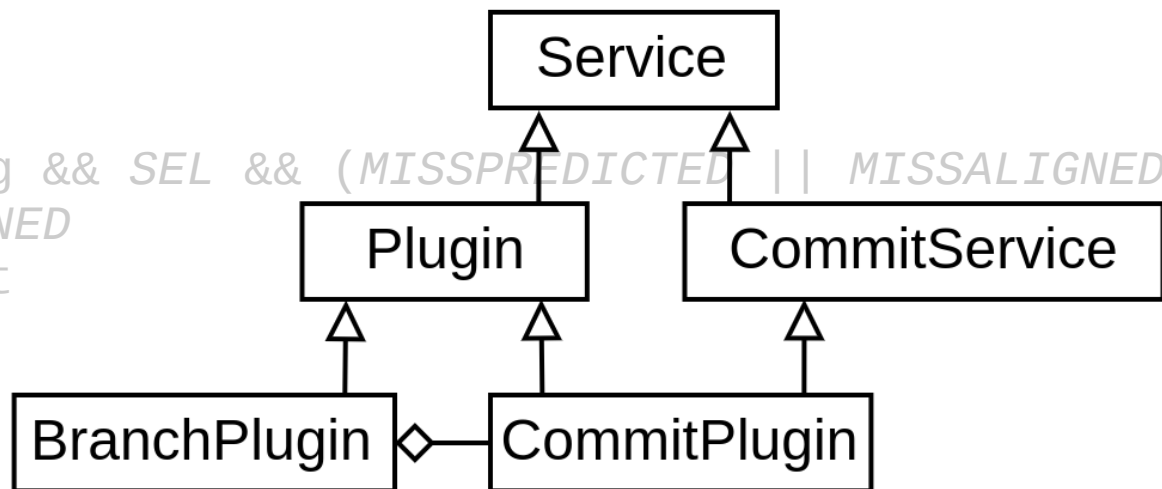
```
class BranchPlugin(euId : String) extends Plugin {  
  val setup = create early new Area{  
    // ...  
    val commit = getService[CommitService]  
    val schedule = commit.newSchedulePort(canJump = true, canTrap = true)  
  }  
}
```

```
val logic = create late new Area{  
  // ...  
  setup.schedule.valid := isFireing && SEL && (MISSPREDICTED || MISSALIGNED)  
  setup.schedule.trap := MISSALIGNED  
  setup.schedule.pcTarget := target  
}  
}
```

misspredicted missaligned

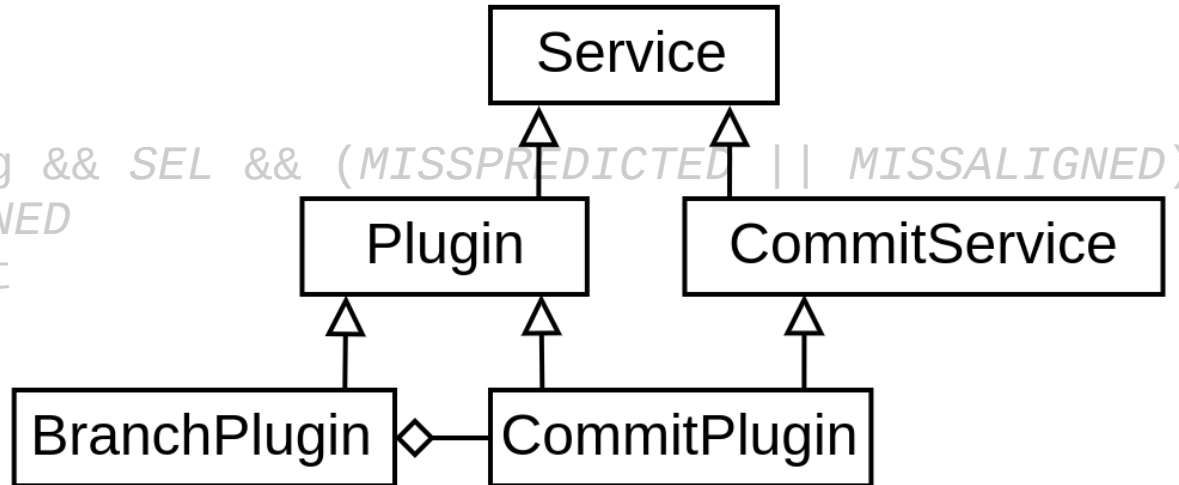


schedule



Plugin – Software interface

```
class BranchPlugin(euId : String) extends Plugin {  
  val setup = create early new Area{  
    // ...  
    val commit = getService[CommitService]  
    val schedule = commit.newSchedulePort(canJump = true, canTrap = true)  
  }  
  
  val logic = create late new Area{  
    // ...  
    setup.schedule.valid := isFireing && SEL && (MISSPREDICTED || MISSALIGNED)  
    setup.schedule.trap := MISSALIGNED  
    setup.schedule.pcTarget := target  
  }  
}
```



Plugin - Implementation

```
class BranchPlugin(euId : String) extends Plugin {  
  val setup = create early new Area{  
    // ...  
    val commit = getService[CommitService]  
    val schedule = commit.newSchedulePort(canJump = true, canTrap = true)  
  }  
  
  val logic = create late new Area{  
    // ...  
    setup.schedule.valid := isFireing && SEL && (MISSPREDICTED || MISSALIGNED)  
    setup.schedule.trap := MISSALIGNED  
    setup.schedule.pcTarget := target  
  }  
}
```

Plugin – Instruction requirements

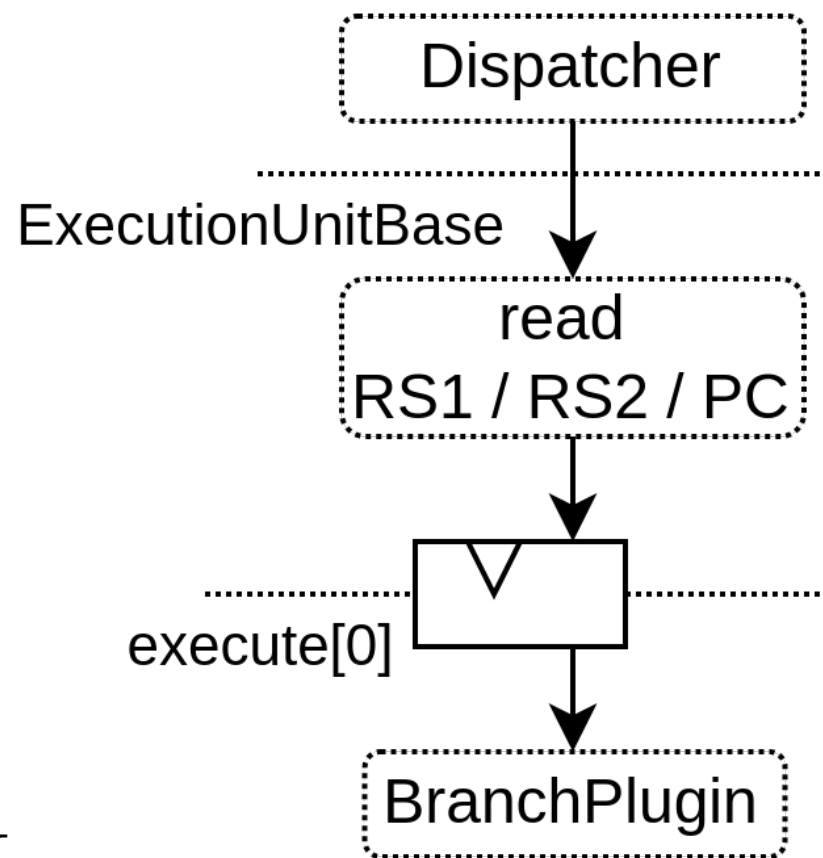
```
val BEQ = SingleDecoding(  
  opcode = M"-----000-----1100011",  
  resources = List(  
    IntRegFile -> RS1,  
    IntRegFile -> RS2,  
    PC_READ,  
    INSTRUCTION_SIZE  
  )  
)  
  
class BranchPlugin(euId : String) extends Plugin{  
  val setup = create early new Area{  
    val eu = findService[ExecutionUnitBase](_.euId == euId)  
    eu.add(BEQ)  
    // ...  
  }  
  //...  
}
```


Plugin – Instruction registration

```
val BEQ = SingleDecoding(  
  opcode = M"-----000-----1100011",  
  resources = List(  
    IntRegFile -> RS1,  
    IntRegFile -> RS2,  
    PC_READ,  
    INSTRUCTION_SIZE  
  )  
)  
  
class BranchPlugin(euId : String) extends Plugin{  
  val setup = create early new Area{  
    val eu = findService[ExecutionUnitBase](_.euId == euId)  
    eu.add(BEQ)  
    // ...  
  }  
  //...  
}
```

Plugin – Side effects

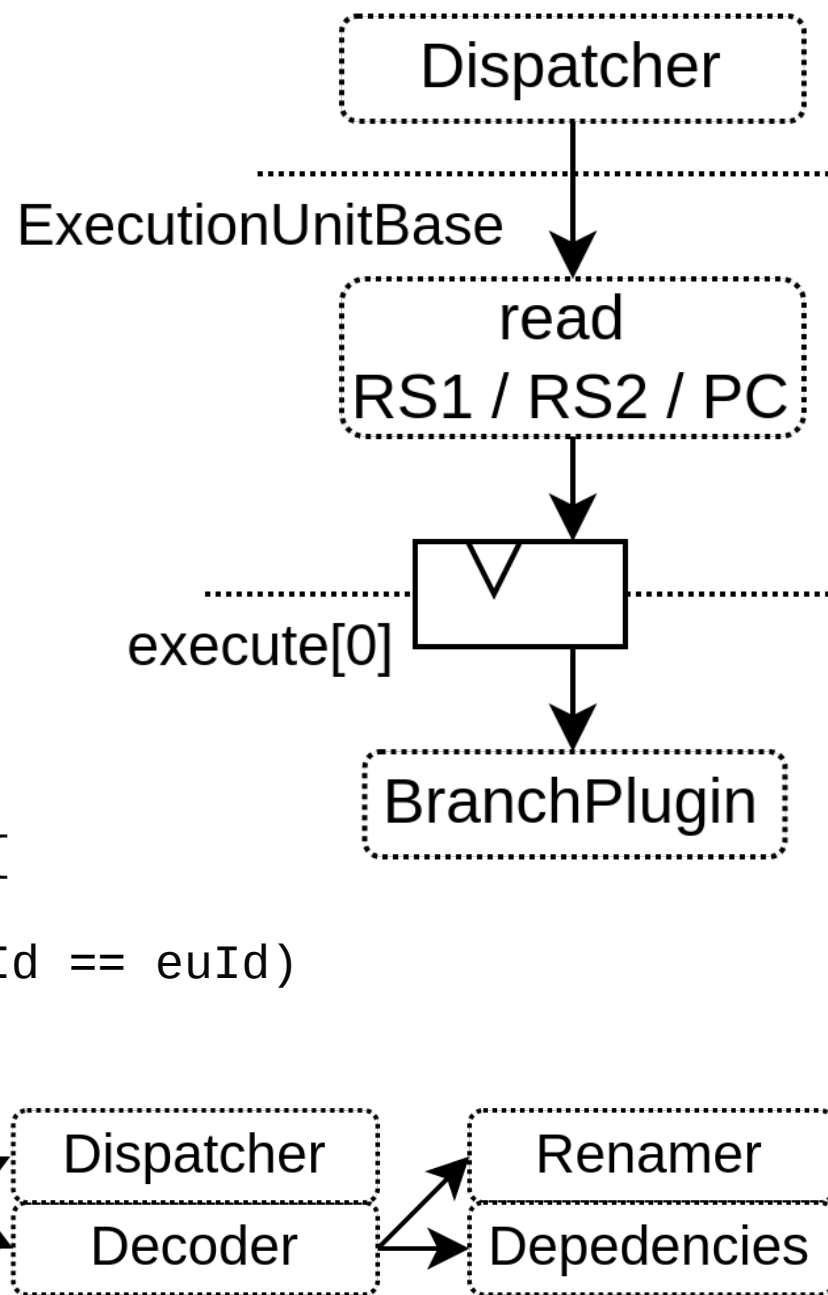
```
val BEQ = SingleDecoding(  
  opcode = M"-----000-----1100011",  
  resources = List(  
    IntRegFile -> RS1,  
    IntRegFile -> RS2,  
    PC_READ,  
    INSTRUCTION_SIZE  
  )  
)  
  
class BranchPlugin(euId : String) extends Plugin{  
  val setup = create early new Area{  
    val eu = findService[ExecutionUnitBase](_.euId == euId)  
    eu.add(BEQ)  
    // ...  
  }  
  //...  
}
```



Plugin – Side effects

```
val BEQ = SingleDecoding(  
  opcode = M"-----000-----1100011",  
  resources = List(  
    IntRegFile -> RS1,  
    IntRegFile -> RS2,  
    PC_READ,  
    INSTRUCTION_SIZE  
  )  
)
```

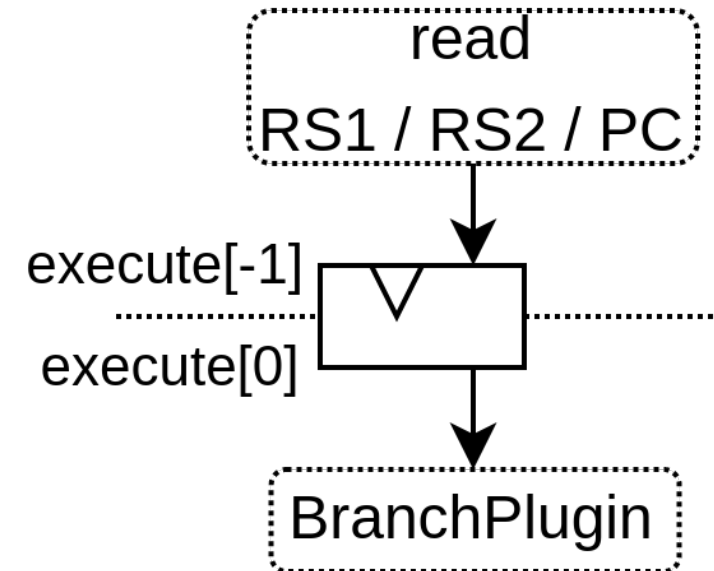
```
class BranchPlugin(euId : String) extends Plugin{  
  val setup = create early new Area{  
    val eu = findService[ExecutionUnitBase](_.euId == euId)  
    eu.add(BEQ)  
    // ...  
  }  
  //...  
}
```



Plugin – Composable pipeline

```
class BranchPlugin(euId : String) extends Plugin{
  //...
  val logic = create late new Area{
    val eu = findService[ExecutionUnitBase](_.euId == euId)
    val stage = eu.getExecute(stageId = 0)
    val target = stage(PC) + ???
    // ...
    setup.schedule.pcTarget := target
  }
}
```

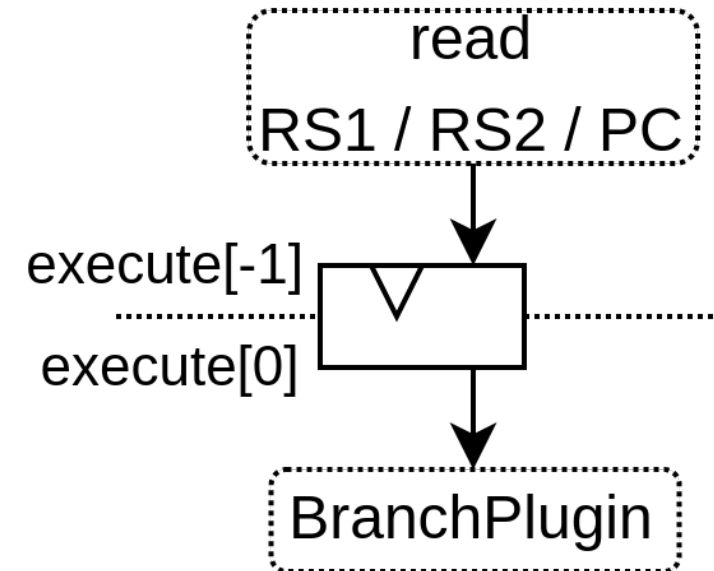
ExecutionUnitBase



Plugin – Composable pipeline

```
class BranchPlugin(euId : String) extends Plugin{
  //...
  val logic = create late new Area{
    val eu = findService[ExecutionUnitBase](_.euId == euId)
    val stage = eu.getExecute(stageId = 0)
    val target = stage(PC) + ???
    // ...
    setup.schedule.pcTarget := target
  }
}
```

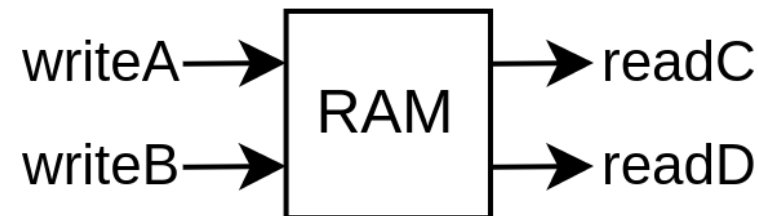
ExecutionUnitBase



Memory inference

```
new Module {  
  val ram = Mem.fill(256)(UInt(16 bits))  
  val writeA = slave(ram.writePort())  
  val writeB = slave(ram.writePort())  
  val readC  = slave(ram.readAsyncPort())  
  val readD  = slave(ram.readAsyncPort())
```

```
  this.dslBody.walkDeclarations{  
    case mem : Mem[_] => {  
      println(s"found $mem")  
      mem.dlcForeach[Any]{  
        case write : MemWrite      => println(s"  
        case read  : MemReadAsync => println(s"  
      }  
      mem.removeStatement()  
      //...  
    }  
    case _ =>  
  }  
}
```



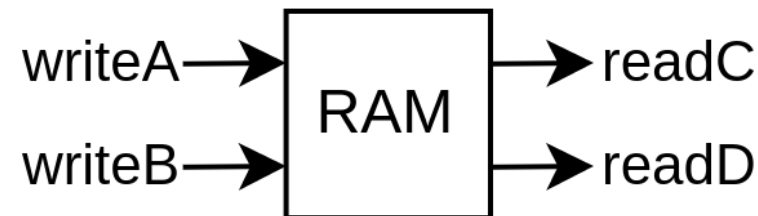
```
found one write port")  
found one read port")
```

```
found toplevel/ram : Mem[256*16 bits]  
found one write port  
found one write port  
found one read port  
found one read port
```

Memory inference

```
new Module {  
  val ram = Mem.fill(256)(UInt(16 bits))  
  val writeA = slave(ram.writePort())  
  val writeB = slave(ram.writePort())  
  val readC = slave(ram.readAsyncPort())  
  val readD = slave(ram.readAsyncPort())  
}
```

```
this.dslBody.walkDeclarations{  
  case mem : Mem[_] => {  
    println(s"found $mem")  
    mem.dlcForeach[Any]{  
      case write : MemWrite      => println(s"  
      case read  : MemReadAsync => println(s"  
    }  
    mem.removeStatement()  
    //...  
  }  
  case _ =>  
}
```

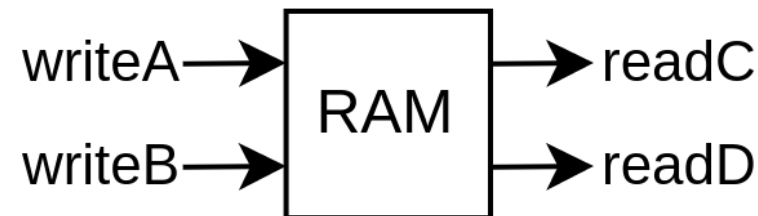


```
found one write port")  
found one read port")
```

```
found toplevel/ram : Mem[256*16 bits]  
found one write port  
found one write port  
found one read port  
found one read port
```

Memory inference / Netlist inspection

```
new Module {  
  val ram = Mem.fill(256)(UInt(16 bits))  
  val writeA = slave(ram.writePort())  
  val writeB = slave(ram.writePort())  
  val readC = slave(ram.readAsyncPort())  
  val readD = slave(ram.readAsyncPort())  
}
```



```
this.ds1Body.walkDeclarations{
```

```
  case mem : Mem[_] => {  
    println(s"found $mem")
```

```
    mem.dlcForeach[Any]{
```

```
      case write : MemWrite      => println(s"found one write port")
```

```
      case read  : MemReadAsync => println(s"found one read port")
```

```
    }
```

```
    mem.removeStatement()
```

```
    //...
```

```
  }
```

```
  case _ =>
```

```
}
```

```
}
```

```
found toplevel/ram : Mem[256*16 bits]
```

```
found one write port
```

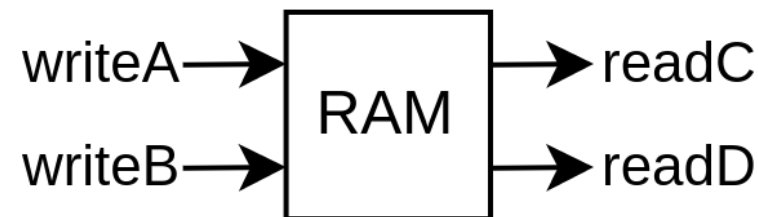
```
found one write port
```

```
found one read port
```

```
found one read port
```


Memory inference / Netlist inspection

```
new Module {  
  val ram = Mem.fill(256)(UInt(16 bits))  
  val writeA = slave(ram.writePort())  
  val writeB = slave(ram.writePort())  
  val readC = slave(ram.readAsyncPort())  
  val readD = slave(ram.readAsyncPort())  
}
```



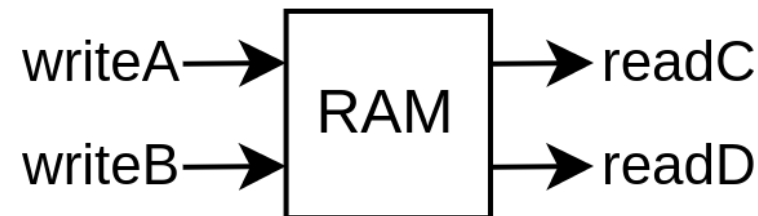
```
this.ds1Body.walkDeclarations{  
  case mem : Mem[_] => {  
    println(s"found $mem")  
    mem.dlcForeach[Any]{  
      case write : MemWrite      => println(s"  
      case read  : MemReadAsync => println(s"  
    }  
    mem.removeStatement()  
    // ...  
  }  
  case _ =>  
}
```

```
found one write port")  
found one read port")
```

```
found toplevel/ram : Mem[256*16 bits]  
found one write port  
found one write port  
found one read port  
found one read port
```

Memory inference / Netlist inspection

```
new Module {  
  val ram = Mem.fill(256)(UInt(16 bits))  
  val writeA = slave(ram.writePort())  
  val writeB = slave(ram.writePort())  
  val readC = slave(ram.readAsyncPort())  
  val readD = slave(ram.readAsyncPort())  
}
```

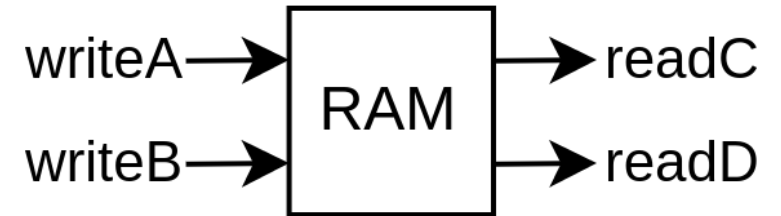


```
this.ds1Body.walkDeclarations{  
  case mem : Mem[_] => {  
    println(s"found $mem")  
    mem.dlcForeach[Any]{  
      case write : MemWrite      => println(s"found one write port")  
      case read  : MemReadAsync => println(s"found one read port")  
    }  
    mem.removeStatement()  
    // ...  
  }  
  case _ =>  
}
```

```
found toplevel/ram : Mem[256*16 bits]  
found one write port  
found one write port  
found one read port  
found one read port
```

Memory inference / Netlist inspection

```
new Module {  
  val ram = Mem.fill(256)(UInt(16 bits))  
  val writeA = slave(ram.writePort())  
  val writeB = slave(ram.writePort())  
  val readC = slave(ram.readAsyncPort())  
  val readD = slave(ram.readAsyncPort())  
}
```

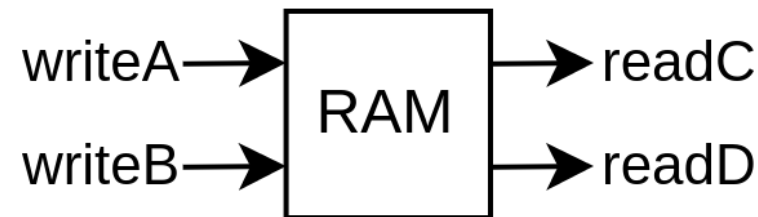


```
this.dslBody.walkDeclarations{  
  case mem : Mem[_] => {  
    println(s"found $mem")  
    mem.dlcForeach[Any]{  
      case write : MemWrite      => println(s"found one write port")  
      case read  : MemReadAsync => println(s"found one read port")  
    }  
    mem.removeStatement()  
    //...  
  }  
  case _ =>  
}
```

```
found toplevel/ram : Mem[256*16 bits]  
found one write port  
found one write port  
found one read port  
found one read port
```

Memory inference / Netlist inspection

```
new Module {  
  val ram = Mem.fill(256)(UInt(16 bits))  
  val writeA = slave(ram.writePort())  
  val writeB = slave(ram.writePort())  
  val readC = slave(ram.readAsyncPort())  
  val readD = slave(ram.readAsyncPort())  
}
```



```
this.ds1Body.walkDeclarations{  
  case mem : Mem[_] => {  
    println(s"found $mem")  
    mem.dlcForeach[Any]{  
      case write : MemWrite      => println(s"found one write port")  
      case read  : MemReadAsync => println(s"found one read port")  
    }  
    mem.removeStatement()  
    //...  
  }  
  case _ =>  
}
```

```
found toplevel/ram : Mem[256*16 bits]  
found one write port  
found one write port  
found one read port  
found one read port
```

Questions

- SpinalHDL : <https://spinalhdl.github.io/SpinalDoc-RTD/master/index.html>
- NaxRiscv : <https://github.com/SpinalHDL/NaxRiscv>
- Roadmap :
 - Debian testing
 - Memory coherency / multicore
 - ASIC targeting ?
- Thanks Nlnet for the funding