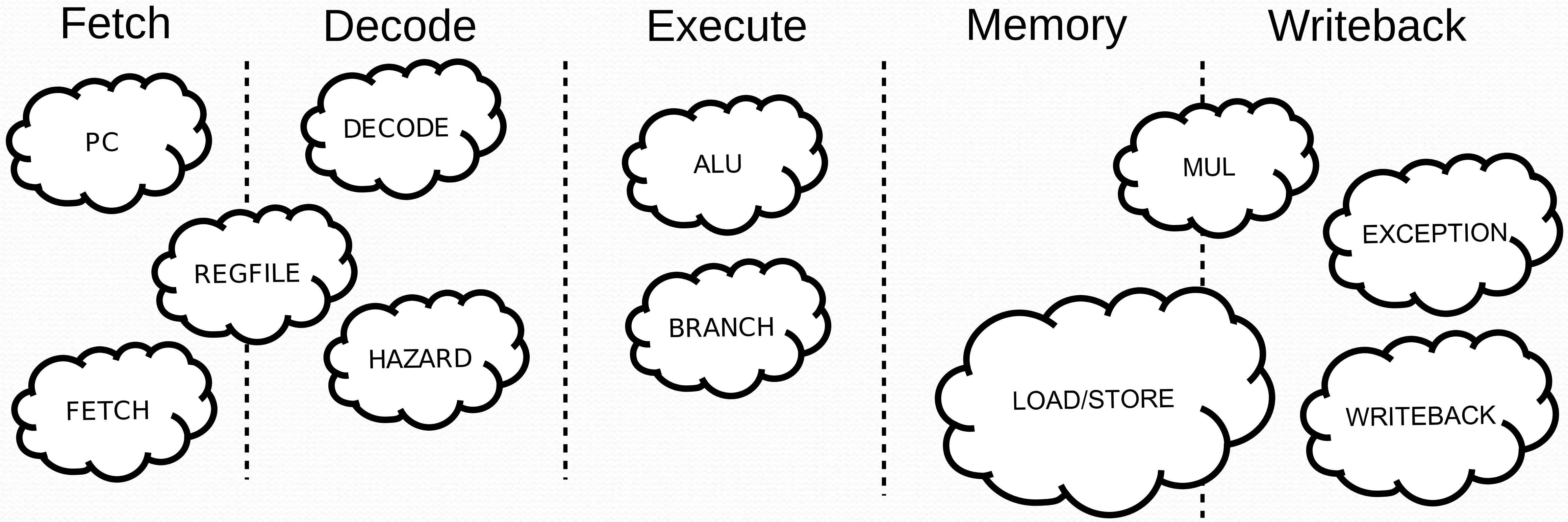
SpinalHDL

VexRiscv

In short

- Started in 2017
- Initial goals
  - Modular and extensible hardware description
  - Test and demonstrate SpinalHDL
  - Free open source NIOS II / Microblaze alternative
  - FPGA synthesis friendly
  - Baremetal only
- Current state
  - RV32I[M][A][C][S], recently added [F][D]
  - JTAG via openocd
  - Can run upstream Linux with SMP

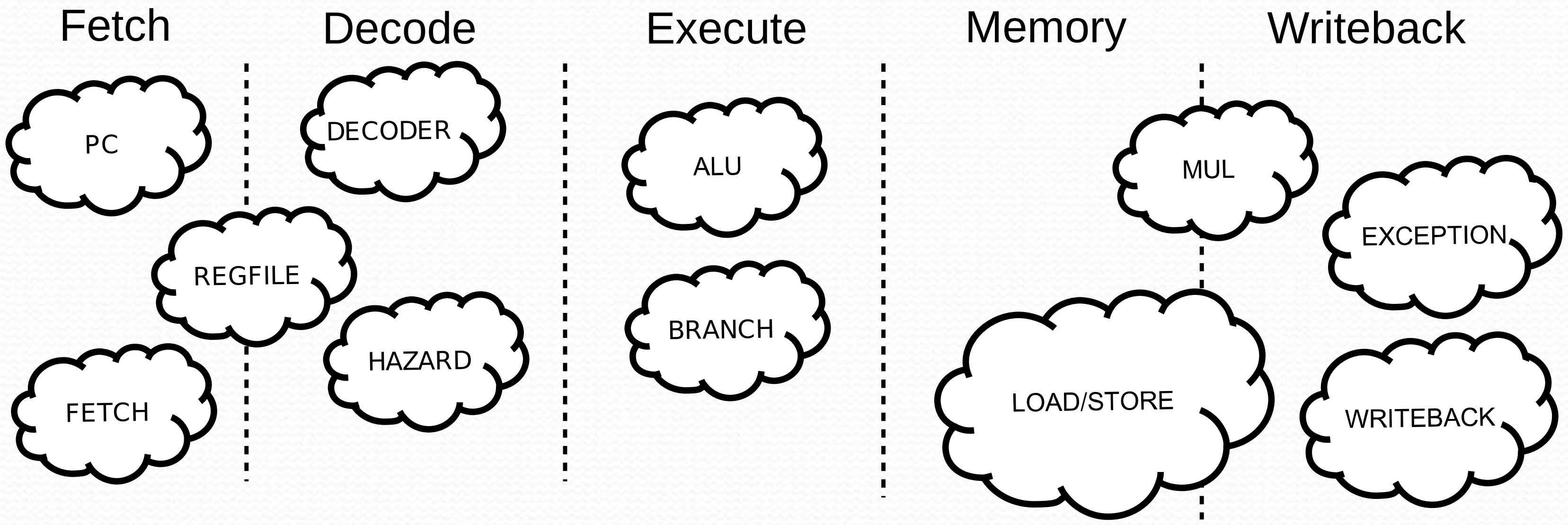
A standard pipeline



But a “empty” toplevel

```
class VexRiscv(val config : VexRiscvConfig) extends Component with Pipeline{
  type  T = VexRiscv
  import config._

  //Define stages
  def newStage(): Stage = { val s = new Stage; stages += s; s }
  val decode    = newStage()
  val execute   = newStage()
  val memory    = ifGen(config.withMemoryStage)    (newStage())
  val writeBack = ifGen(config.withWriteBackStage) (newStage())

  def stagesFromExecute = stages.dropWhile(_ != execute)

  plugins ++= config.plugins

  … + 15 lines of stuff related to testing
}
```

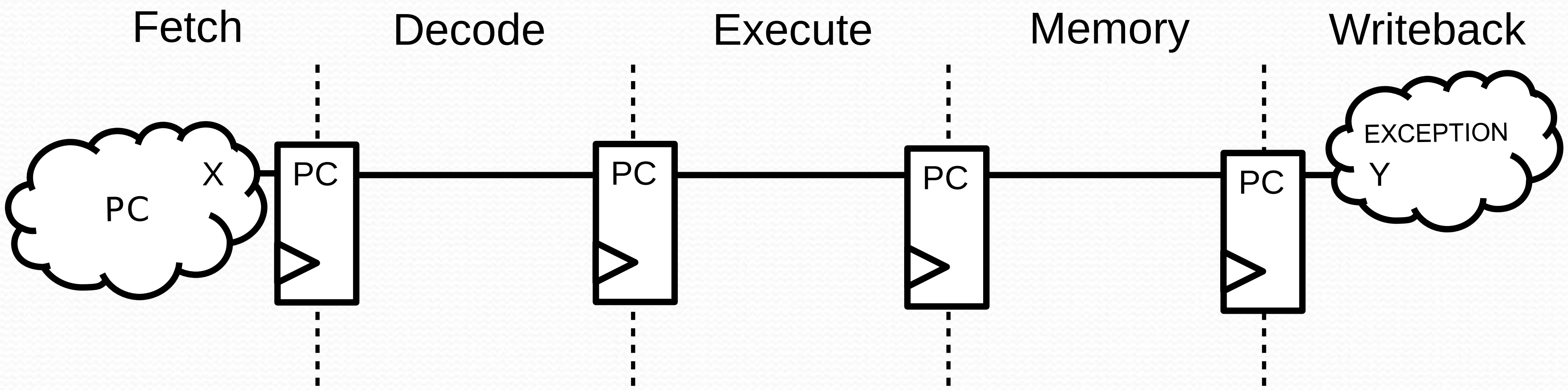
```
val plugins = VexRiscvConfig(
  plugins = List(
    new IBusSimplePlugin(catchAccessFault = false, resetVector = 0x00000000l),
    new DBusSimplePlugin(catchAccessFault = false),
    new DecoderSimplePlugin(catchIllegalInstruction = false),
    new RegFilePlugin,
    new IntAluPlugin,
    new FullBarrielShifterPlugin,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

val cpu = new VexRiscv(plugins)
```



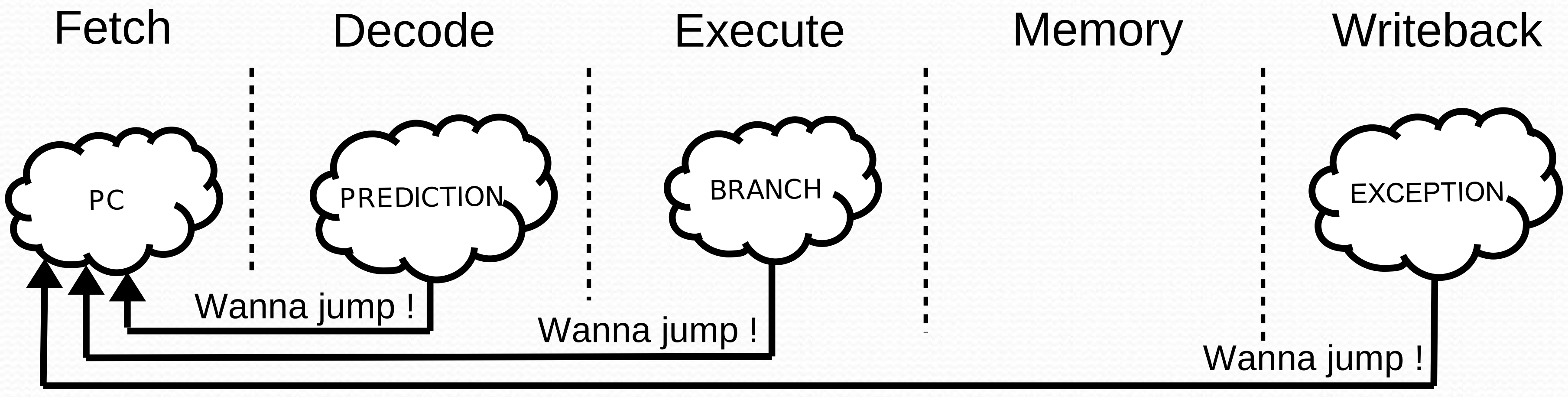
Modular CPU framework

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)
```

Plugin base class

```
trait Plugin[T <: Pipeline] extends Nameable{
    // Used to setup things with other plugins
    def setup(pipeline: T) : Unit

    //Used to flush out the required hardware (called after setup)
    def build(pipeline: T) : Unit

    ...
}
```

SIMD add (4 parallel 8 bits add)

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

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

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

SIMD add (4 parallel 8 bits add)

```
override def setup(pipeline: VexRiscv): Unit = {
  import pipeline.config._

  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
    )
  )
}
```

SIMD add (4 parallel 8 bits add)

```
override def build(pipeline: VexRiscv): Unit = {
    import pipeline._
    import pipeline.config._

    execute plug new Area {
      val rs1 = execute.input(RS1).asUInt
      val rs2 = execute.input(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)


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

Many plugins

```
AesPlugin.scala                         IntAluPlugin.scala
BranchPlugin.scala                      MemoryTranslatorPlugin.scala
CfuPlugin.scala                         Misc.scala
CsrPlugin.scala                         MmuPlugin.scala
DBusCachedPlugin.scala                  Mul16Plugin.scala
DBusSimplePlugin.scala                  MulDivIterativePlugin.scala
DebugPlugin.scala                       MulPlugin.scala
DecoderSimplePlugin.scala               MulSimplePlugin.scala
DivPlugin.scala                         NoPipeliningPlugin.scala
DummyFencePlugin.scala                  PcManagerSimplePlugin.scala
ExternalInterruptArrayPlugin.scala      Plugin.scala
Fetcher.scala                           PmpPlugin.scala
FormalPlugin.scala                      RegFilePlugin.scala
FpuPlugin.scala                         ShiftPlugins.scala
HaltOnExceptionPlugin.scala             SingleInstructionLimiterPlugin.scala
HazardPessimisticPlugin.scala           SrcPlugin.scala
HazardSimplePlugin.scala                StaticMemoryTranslatorPlugin.scala
IBusCachedPlugin.scala                  YamlPlugin.scala
IBusSimplePlugin.scala
```

DataCache features

- Write-through
- Atomic via load reserve and store conditional
- Coherency via invalidation
- Multi way, no more than 4KB per way if the MMU is used
- Flexible refill data width

Other features to setup

- Multiplication : Iterative in 32/2^n cycles | single cycle 17x17x4
- Register file : Sync | Async read, X0 init kind
- Branch : Prediction, late
- Hazard : Interlock | bypass
- Fetch : pipelining levels, RVC
- Memory, Writeback : can be optional
- Many other timings related things

Ram blackboxing

```
object MuraxAsicBlackBox extends App{
  val config = SpinalConfig()
  config.addStandardMemBlackboxing(blackboxAll)
  config.generateVerilog(Murax(MuraxConfig.default()))
}
```

- Use blackbox from spinal/core/MemBlackBox.scala to get ram
  - Ram_1w_1ra
  - Ram_1w_1rs
  - Ram_2c_1w_1rs
  - Ram_1wors
  - Ram_1wrs
- SpinalHDL “compiler” phases can be extended with another blackboxing implementation