From the RISC-V spec to a low-tech SoC

passing by SpinalHDL, VexRiscv and OpenOCD

In short, I'm involved into :

- SpinalHDL (Alternative to VHDL/SystemVerilog for digital netlist description)
- VexRiscv (RISC-V CPU)

From RISC-V to VexRiscv (CPU implementation)

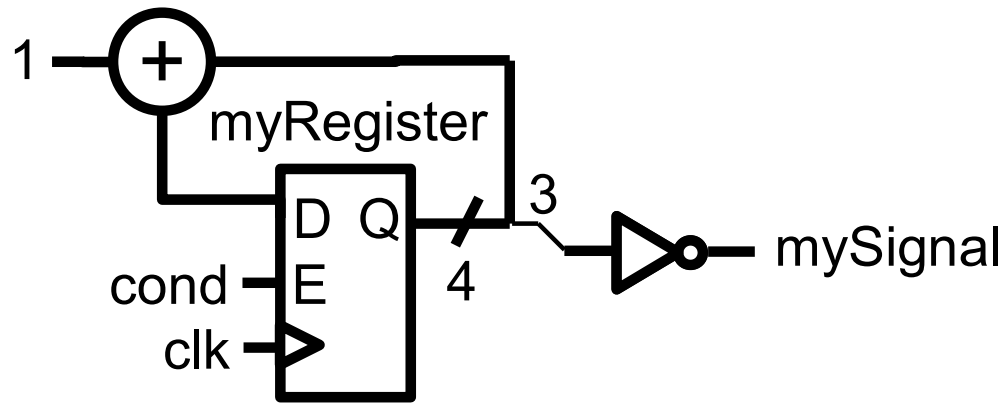
- RV32I[M][C]
- Implemented in SpinalHDL
  - Automated instruction decoding generation (Quine–McCluskey)
  - Automated pipelining (from 2 stages to 4 + X fetch stages)
  - CPU composability and customisation (The parametrization hold the behaviour)
  - Can't do that with VHDL/SystemVerilog on current synthesis tools
- Some resources about it :
  - https://github.com/SpinalHDL/VexRiscv#add-a-custom-instruction-to-the-cpu-via-the-plugin-system
  - https://tomverbeure.github.io/rtl/2018/12/06/The-VexRiscV-CPU-A-New-Way-To-Design.html

SpinalHDL, what it is, what it isn’t

- Not a even driven language
- Not a High Level Synthesis (HLS)
- Not a data flow language
- Not a language
- It is an hardware description library (SpinalHDL) implemented in a general purpose programming language (Scala)
- The hardware netlist is elaborated from the software runtime, not a compilation.

SpinalHDL API, not so far from a regular HDL

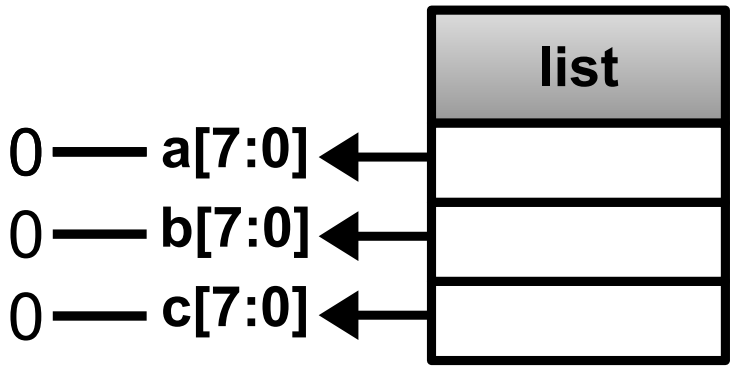
```
val myRegister = Reg(UInt(4 bits))
when(cond) {
   myRegister := myRegister + 1
}

val mySignal  = Bool
mySignal := ! myRegister.msb
```



Software as an elaboration tool

BLUE => Elaboration software
RED => SpinalHDL API

```
val a,b,c = UInt(8 bits)
val list = List(a,b,c)

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

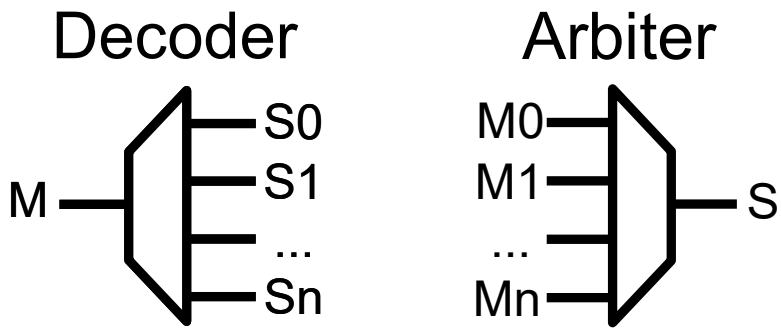


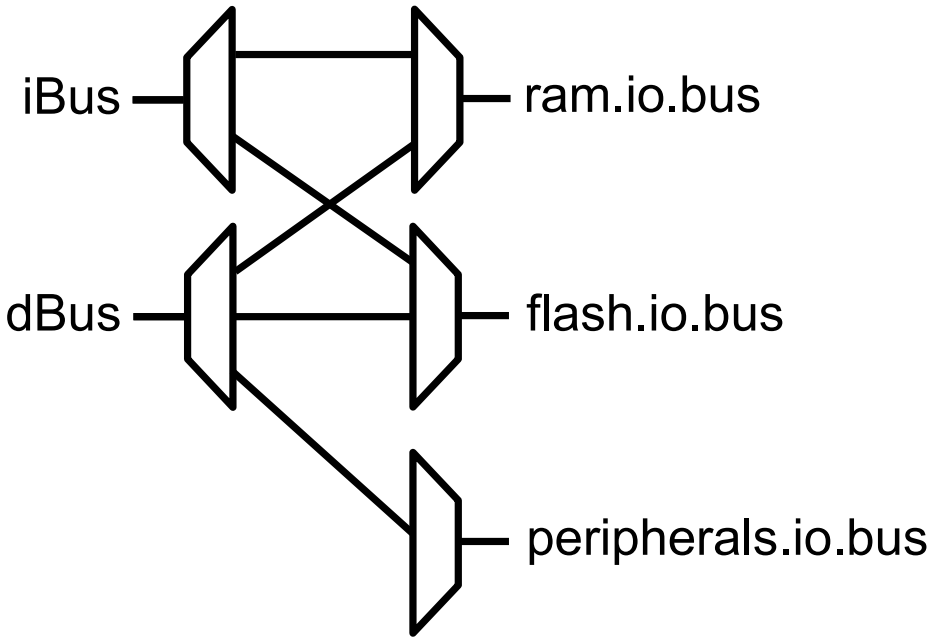
Software as an elaboration tool

- Object oriented programming
- Functional programming
- Inheritance
- Software collection (Dynamic lists, Map/Dictionary, Set)
- ...

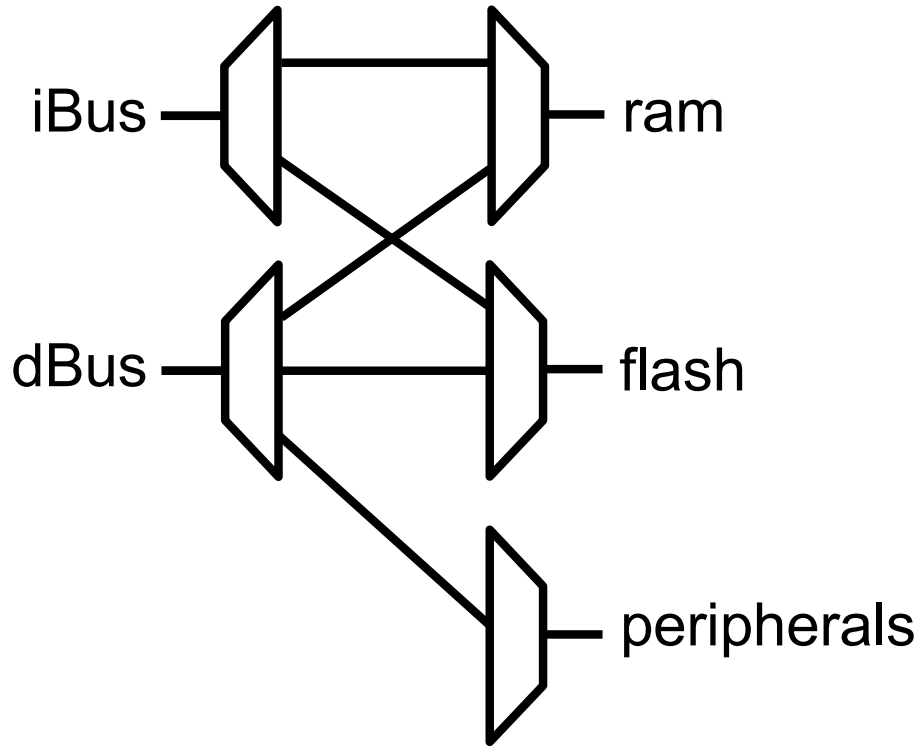
Interconnect

- Nothing complicated here
  - 1 to N (decoder)
  - N to 1 (arbiter)





```
val interconnect = PipelinedMemoryBusInterconnect()

interconnect.addSlaves(
  ram.io.bus           -> SizeMapping(0x00000,   64 kB),
  flash.io.bus         -> SizeMapping(0x80000,  512 kB),
  peripherals.io.bus -> SizeMapping(0x70000,   64 kB)
)

interconnect.addMasters(
  cpu.io.iBus  -> List(ram.io.bus, flash.io.bus),
  cpu.io.dBus -> List(ram.io.bus, flash.io.bus, peripherals.io.bus)
)

interconnect.build()
```



```
class PipelinedMemoryBusInterconnect{
  class SlaveInfo(bus : PipelinedMemoryBus, mapping : SizeMapping)
  class MasterInfo(bus : PipelinedMemoryBus, slaves : List[PipelinedMemoryBus])

  val slaves = ArrayBuffer[SlaveInfo]()
  val masters = ArrayBuffer[MasterInfo]()

  def addSlave(s : PipelinedMemoryBus, mapping : SizeMapping) = {
    slaves += new SlaveInfo(s, mapping)
  }
  def addMaster(m : PipelinedMemoryBus, slaves : List[PipelinedMemoryBus]) = {
    masters += new MasterInfo(m, slaves)
  }

  def build(): Unit ={  … }
}
```

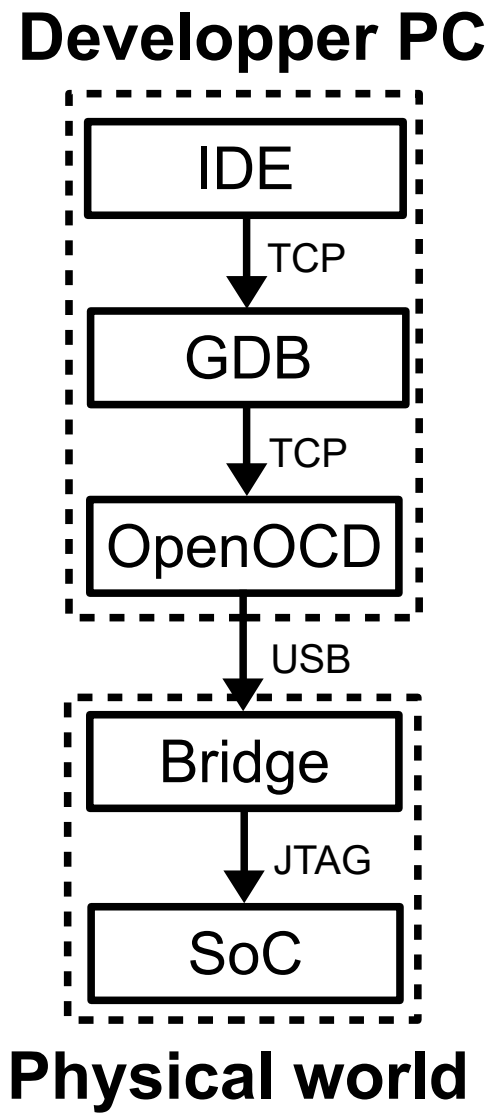
```
def build(): Unit ={
  for(m <- masters){
    // Generate decoders
  }
  for(s <- slaves){
    // Generate arbiters
  }
  // Connect decoders to arbiters
}
```

CPU : Debug interface

- A CPU without a debug interface can be a nightmare
- Many opensource CPU do not implement one :(
- Minimal requirements
  - CPU controls (reset, run, halt, step)
  - CPU status (halted, breakpointed)
  - CPU god-mode (push instruction, read the result)
  - External interface (JTAG)

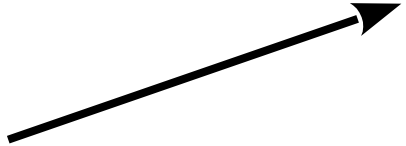
OpenOCD

- What it do :
  - Abstract the hardware to the debugger
  - Provide many JTAG drivers
- What is required to support a new CPU :
  - A set of C functions



OpenOCD : Adding a new target

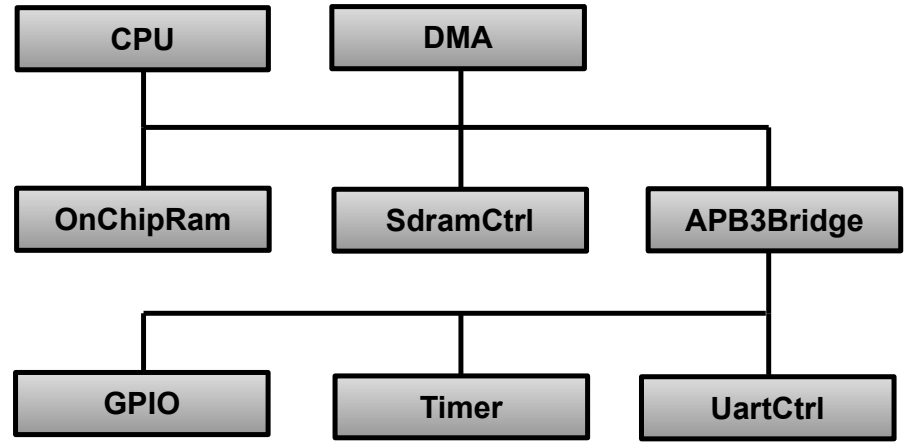
```
struct target_type vexriscv_target = {
    .name = "vexriscv",

    .target_create = vexriscv_target_create,
    .examine = vexriscv_examine,

    .halt = vexriscv_halt,
    .resume = vexriscv_resume,
    .step = vexriscv_step,

    .add_breakpoint = vexriscv_add_breakpoint,
    .remove_breakpoint = vexriscv_remove_breakpoint,

    .read_memory = vexriscv_read_memory,
    .write_memory = vexriscv_write_memory,

    ...
};
```



Write 0x89abcdef @ 0x1234567

Push instruction :
lui 0x89abc, x1
ori x1, x1, 0xdef
lui 0x01234, x2
sw x1, x2(0x567)

SoC toplevel hell



- Imagine X did a SoC, then Y want to extend it for it’s own purposes, this usualy result in :
  - Y forking X SoC, and modify the toplevel (no way back, painefull pullrequest)
  - Y adding parametrization into X SoC to cover its use cases (messy SoC toplevel)
- Now Z want to extend the X SoC for it’s own purposes, ...
- Parametrization isn’t good enough !
- Solution attempt
  - Composing a SoC by having generators and dependencies

A stairway to the heaven (I believe)

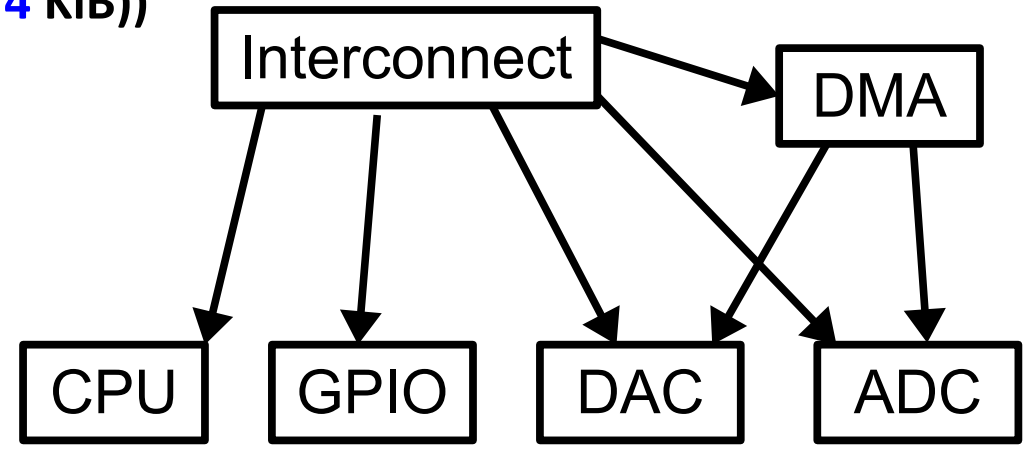
```
class SocX extends SocBase{
   val gpioA = addGpio(0xA000, 32)
}
```

```
class SocY extends SocBase {
   val gpioA = addGpio(0xA000, 12)
   val gpioB = addGpio(0xA100, 16)
   val gpioC = addGpio(0xA200, 20)
}
```

```
class SocZ extends SocBase {
   val gpioA = addGpio(0xA000, 32)

   val dma = addDma(0xB000, ...)

   val adc = addAdc(0xC000, ...)
   dma.addInputStream(adc.stream)

   val dac = addDac(0xC100, ...)
   dma.addOutputStream(dac.stream)
}
```

Multipass elaboration

```
class SocBase extends Generator{
  val cpu = new Generator { ... }
  val interconnect = new Generator {
    val builder  = PipelinedMemoryBusInterconnect()
    add task { builder.build() }
  }

  def addGpio(address : Int, width : Int) = new Generator {
    interconnect.dependencies += this

    add task {
      // Hardware generation here
      interconnect.addSlave(bus → SizeMapping(address, 4 KiB))
    }
  }
  def addAdc(address : Int) = new Generator { … }
  def addDac(address : Int) = new Generator { … }
  def addDma(address : Int) = new Generator { … }
}
```



There is very much to do !

- New design patterns to invents
  - To squeeze more out of your ASIC/FPGA and our limited man power
  - To make hardware description agile, we aren't coding monkies
- Implementing hardware by using those design patterns
  - CPU, SoC, Interconnect, Peripherals, …
- Emerging HDL to support
  - SpinalHDL, Chisel, Migen, Clash, …