SpinalHDL

A Modern Solution for Hardware Coding

Contents

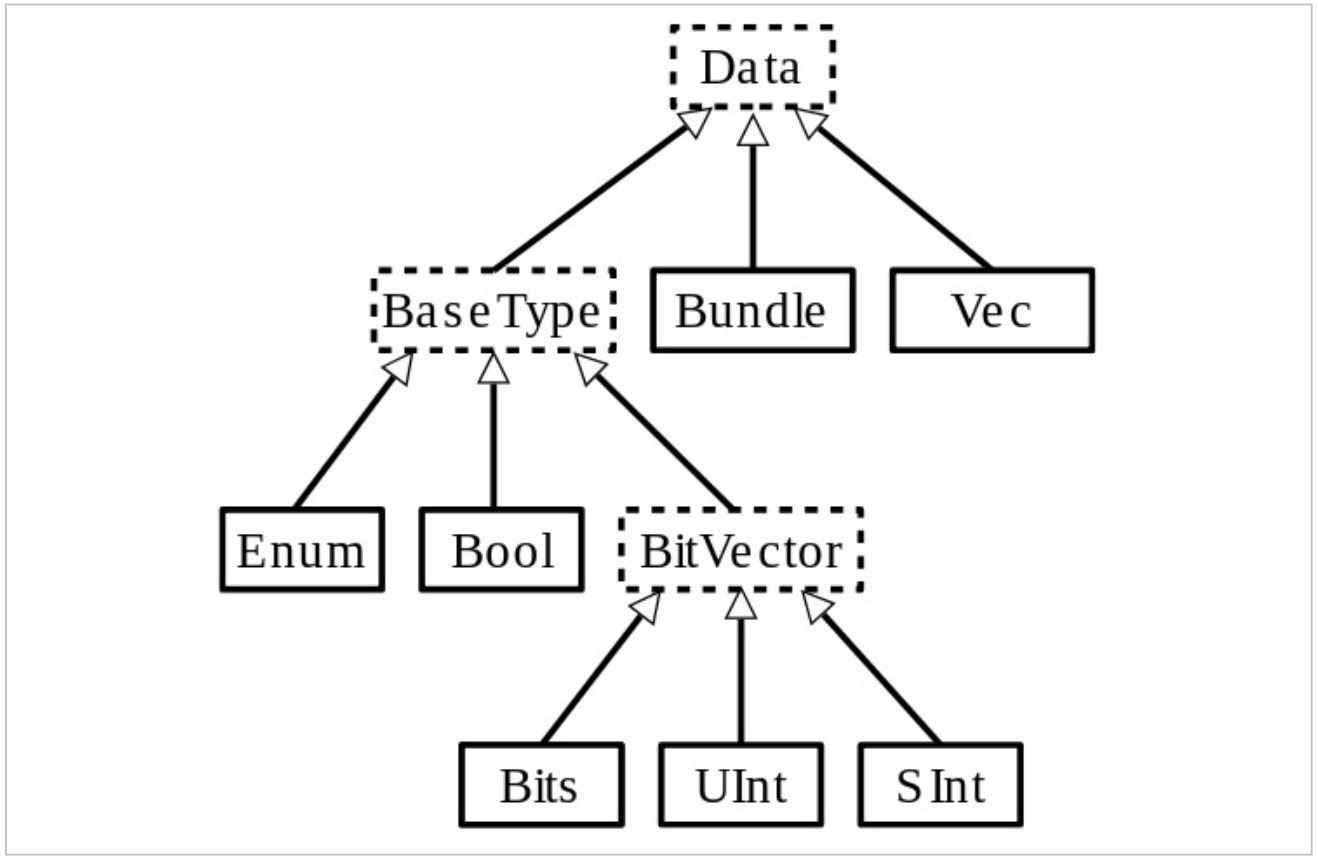
- Better Verilog
  - &#;Data Types
  - &#;Component Hierarchy
  - &#;BlackBox
- Advanced Features of SpinalHDL
- Reconstruct RTL
  - &#;Take VexRiscv Core as an Example

Section 1

Better RTL

Data Types

- Strong typing
  - Bits / UInt / SInt
  - Enum / Bool
- Structural types
  - Bundle
  - Vec
  - ...
- Register
  - Reg(Data) init 1



Object-Oriented Types

```
class Color() extends Bundle {
  val r, g, b = Bits( width = 8 bits)
  def isBlack: Bool = r === 0 & g === 0 & b === 0
  def isWhite: Bool = r === 255 & g === 255 & b === 255
}

trait Transparent {
  val a = Bits( width = 8 bits)
  def isTransparent: Bool = a === 255
}

case class MyRGBA() extends Color with Transparent

case class MyRGB() extends Color

class MyTop extends Component {
  val io = new Bundle {
    val videoIn  = in  (MyRGBA())
    val videoOut = out (MyRGB())
    val colorful = out Bool()
  }

  val vin = io.videoIn
  val vBuf = RegNextWhen(vin.r ## vin.g ## vin.b, ! vin.isTransparent) in
  val colorful = RegNextWhen((! vin.isBlack) & ! vin.isWhite, ! vin.isTra

  io.videoOut.r := vBuf(23 downto 16)
  io.videoOut.g := vBuf(15 downto 8)
  io.videoOut.b := vBuf(7 downto 0)
  io.colorful := colorful
}
```

```
module MyTop (
      input  [7:0] vin_r,
      input  [7:0] vin_g,
      input  [7:0] vin_b,
      input  [7:0] vin_a,
      output [7:0] io_videoOut_r,
      output [7:0] io_videoOut_g,
      output [7:0] io_videoOut_b,
      output  io_colorful,
      input   clk,
      input   reset);
  reg [23:0] vBuf;
  reg  colorful;
  assign io_videoOut_r = vBuf[23 : 16];
  assign io_videoOut_g = vBuf[15 : 8];
  assign io_videoOut_b = vBuf[7 : 0];
  assign io_colorful = colorful;
  always @ (posedge clk or posedge reset) begin
    if (reset) begin
      vBuf <= (24'b000000000000000000000000);
    end else begin
      if((! (vin_a == (8'b11111111))))begin
        vBuf <= {{vin_r,vin_g},vin_b};
      end
    end
  end

  always @ (posedge clk) begin
    if((! (vin_a == (8'b11111111))))begin
      colorful <= ((! (((vin_r == (8'b00000000)) && (vin_g == (8'b00000000))) &&
    end
  end

endmodule
```

Component Hierarchy

- Component
  &#;Statement "module" in Verilog
  &#;Explicit IO declaration
    &#;in / out / master / slave
- Area
  - Flexible and light
  - Nested module
  - Clock domain is a special "Area"

```
47  class UartCtrl extends Component {
48
49    val timer = new Area {
50      val counter = Reg(UInt( width = 8 bit))
51      val tick = counter === 0
52      counter := counter - 1
53      when(tick) {
54        counter := 100
55      }
56    }
57
58    val tickCounter = new Area {
59      val value = Reg(UInt( width = 3 bit))
60      val reset = False
61      when(timer.tick) {          // Refer to the tick from timer area
62        value := value + 1
63      }
64      when(reset) {
65        value := 0
66      }
67    }
68
69    val stateMachine = new Area {
70
71    }
72  }
```

Interact with Verilog/VHDL

```
80    class Ram(wordCount: Int) extends BlackBox{
81      val io = new Bundle{
82        val clk = in Bool
83        val addr = in UInt(32 bits)
84          // ...
85      }
86      addGeneric( name= "wordcount", wordCount)
87      noIoPrefix() // remove io_ prefix
88      mapClockDomain(clock=io.clk) // map to the current clockDomai
89    }
```

Section 2

Advanced Features

Bus: a More Complex Bundle

- Abstract Bus Interface
- Stream / Flow
  &#;StreamFIFO (CDC)
- Built-in Bus Protocol
  &#;Avalon
  &#;AHB / APB / AXI

```scala
import spinal.lib.bus.amba4.axi._
import spinal.lib._

case class Axi4(config: Axi4Config) extends Bundle with IMasterSlave {
  val aw = Stream(Axi4Aw(config))
  val w  = Stream(Axi4W(config))
  val b  = Stream(Axi4B(config))
  val ar = Stream(Axi4Ar(config))
  val r  = Stream(Axi4R(config))

  override def asMaster(): Unit = {
    master(ar,aw,w)
    slave(r,b)
  }
}

class Top extends Component {
  val axiConfig = Axi4Config(
    addressWidth = 32,
    dataWidth    = 32,
    idWidth      = 4
  )
  val axiX = Axi4(axiConfig)
  val axiY = Axi4(axiConfig)

  when(axiY.aw.valid){
    // ...
  }
}
```

Finite State Machine

- FSMs could be defined with regular syntax (Enum, Switch)
- You can also use a much more friendly syntax, fully integrated, with following features :

  &#;onEntry / onExit / whenIsActive / whenIsNext blocs

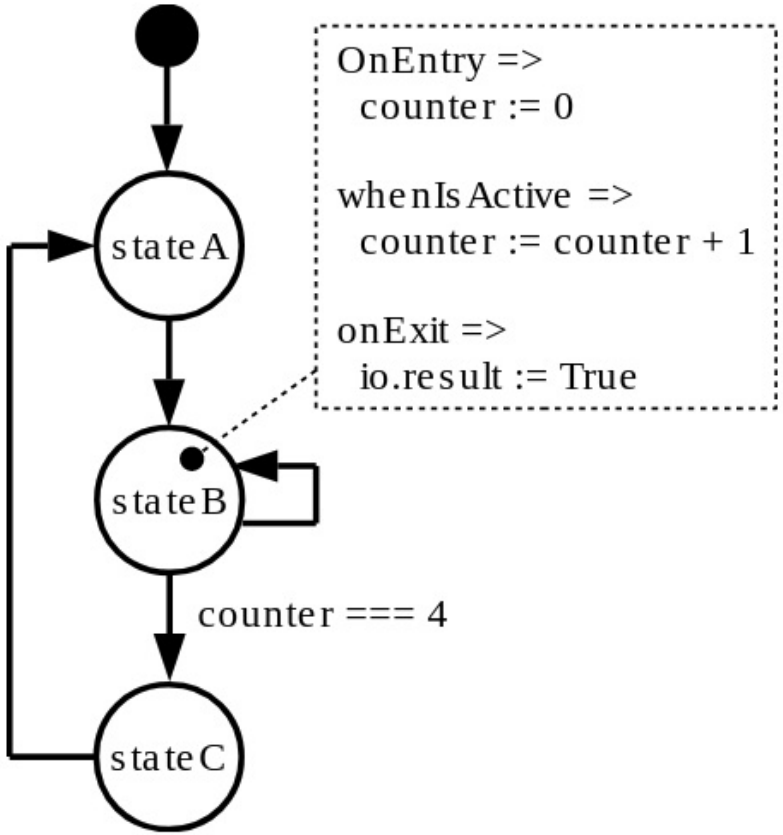
  &#;State with inner FSM

  &#;State with multiple inner FSM (parallel execution)

  &#;Delay state

  &#;You can extends the syntax by defining new state types

Finite State Machine



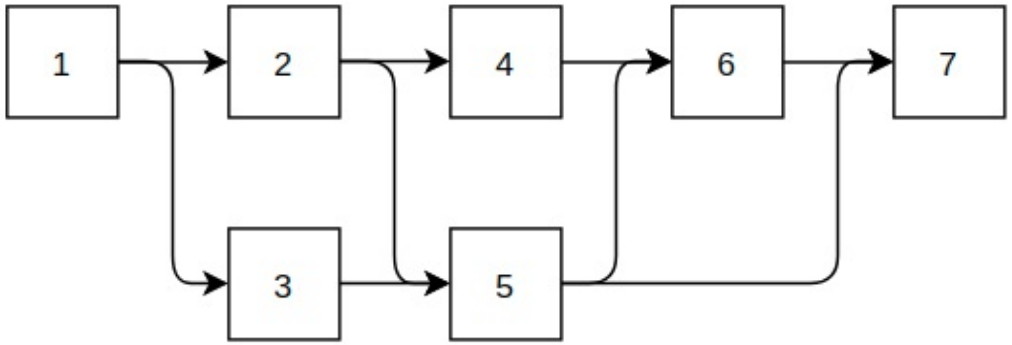

```scala
import spinal.lib.fsm._

class TopLevel extends Component {
  val io = new Bundle{
    val result = out Bool
  }

  val fsm = new StateMachine{
    val stateA = new State with EntryPoint
    val stateB = new State
    val stateC = new State

    val counter = Reg(UInt( width = 8 bits)) init (0)
    io.result := False

    stateA
      .whenIsActive (goto(stateB))

    stateB
      .onEntry(counter := 0)
      .whenIsActive {
        counter := counter + 1
        when(counter === 4){
          goto(stateC)
        }
      }
      .onExit(io.result := True)

    stateC
      .whenIsActive (goto(stateA))
  }
}
```

Signal Latency Automatic Inference

- LatencyAnalysis(paths : Node*)

  &#;return: Int

  &#;Return the shortest path,in therm of cycle, that travel through all nodes, from the first one to the last one



Signal Latency Automatic Inference

```scala
object MatchLatency {

  // Match arrival time of 2 signals with _vld
  def apply[A <: Data, B <: Data](common_vld: Bool, a_vld : Bool, a : A, b_vld : Bool, b : B) : (Bool, A, B) = {

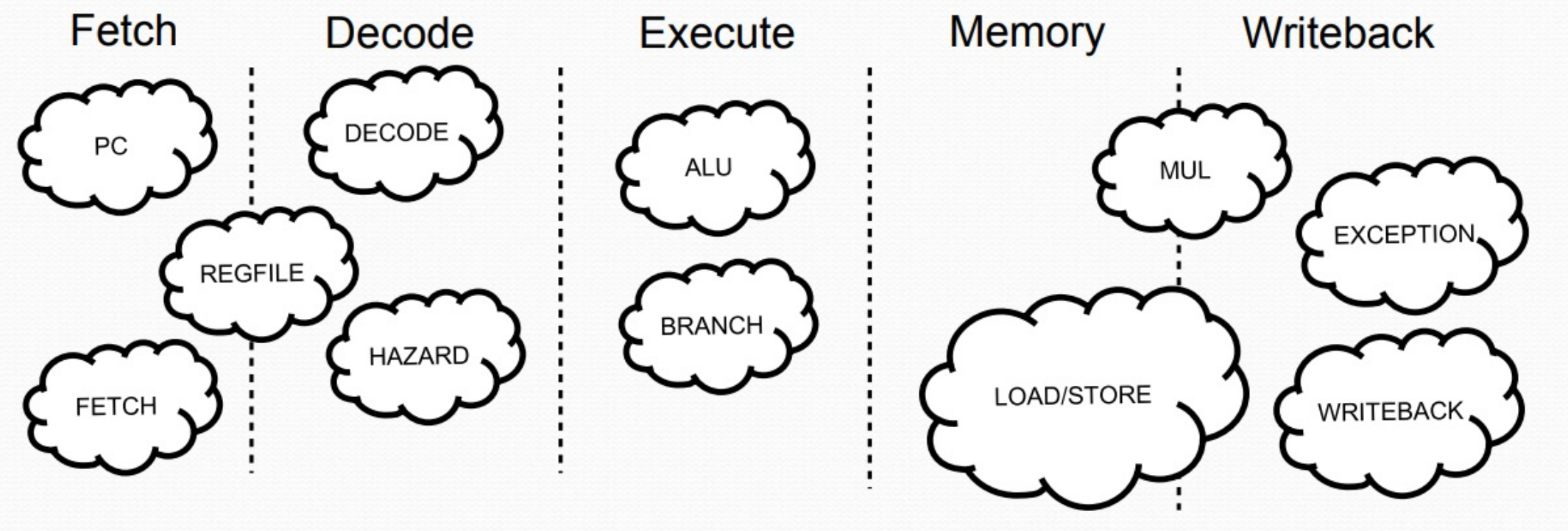
    val a_latency = LatencyAnalysis(common_vld, a_vld)
    val b_latency = LatencyAnalysis(common_vld, b_vld)

    if (a_latency > b_latency) {
      (a_vld, a, Delay(b, cycleCount = a_latency - b_latency) )
    }
    else if (b_latency > a_latency) {
      (b_vld, Delay(a, cycleCount = b_latency - a_latency), b )
    }
    else {
      (a_vld, a, b)
    }
  }
}
```

Section 3

Reconstruct RTL - VexRiscv

A Traditional CPU with Many Stages



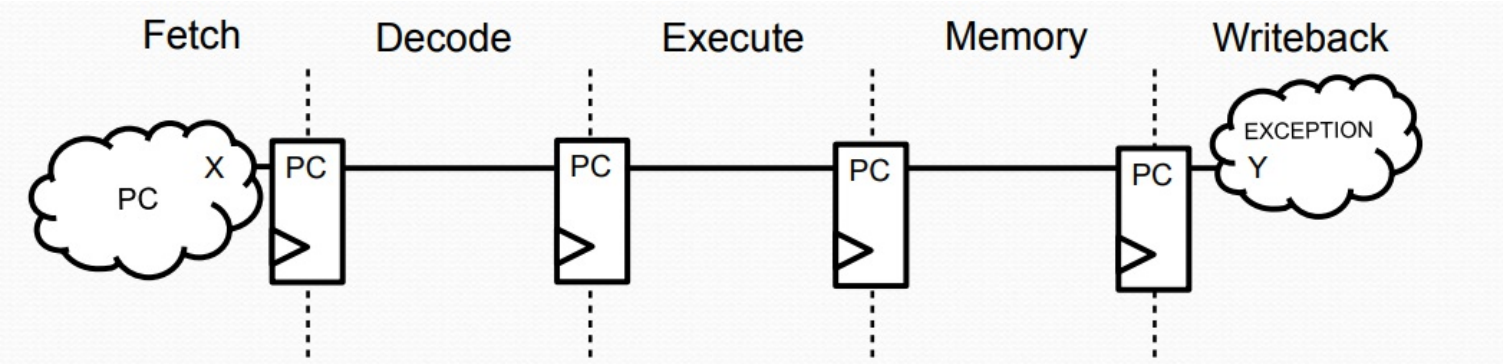
pipeline.v in RIDECORE has 1931 lines.

Stage & Stageable

- `trait Stageable[T<: Data](_dataType: => T) extends ...`
- `class Stage() extends Area`
  - `def outsideCondScope[T](that: => T): T = ...`
  - `def input[T<: Data](key: Stageable[T]): T = ...`
  - `def output[T <: Data](key : Stageable[T]) : T = ...`
  - `def insert[T<: Data](key: Stageable[T]): T = ...`
  - `val arbitration = ...`
  - `def inputInit[T<: BaseType](stageable: Stageable[T], initValue: T) = ...`

Stage & Stageable

```
//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)
```



Pipeline

- ```
  trait Pipeline
  &#; val pulgins = ArrayBuffer[Plugin[T]]()
  &#; val stages = ArrayBuffer[Stage]()
  &#; def build(): Unit = ...
  ```
  ```

RegFilePlugin

- ```
  trait Plugin[T <: Pipeline]
    &#; def plug[T <: Area](area : T) : T =
        {area.setCompositeName(stage,getName()).reflectNames();area}
  ```

- ```
  val readStage = if(readInExecute) execute else decode
  ```
- ```
  val writeStage = if(writeRfInMemoryStage) memory else stages.last
  ```
- ```
  val global = pipeline plug new Area { ... }
  ```
- ```
  readStage plug new Area { ... }
  ```
- ```
  writeStage plug new Area { ... }
  ```

Beyond the Code

Why must the **front-end RTL** and the **circuit** have the **same architecture**?