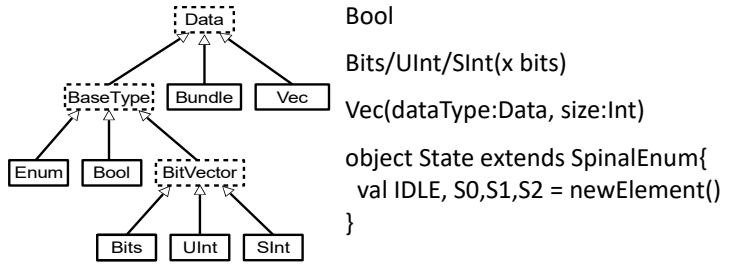


SpinalHDL CheatSheet – Core

Basic Types



Literals

Bool(boolean)	val myBool = Bool(4 < 2)
True, False	val myBool = True
B/U/S(value:Int[,x bits])	val myUInt = U(13, 32 bits)
B/U/S”[[size’]base]value”	val myBits = B"8'hA3" // h,d,b,x,o
B/U/S”binaryNumber”	val myBits = B"0110"
M”binaryNumber”	val itMatch = myBits === M"00--10--"

Register

val r = Reg(UInt(8 bits)	val r = Reg(UInt(8 bits)) init(0)
val r = RegNext(signal)	val r = RegNextWhen(signal,cond)
val r = RegInit(U"010")	

Assignements

x := y	VHDL, Verilog <=
x <> y	uartCtrl.io.uart <> io.uart //Automatic connection
x \= y	VHDL :=, Verilog =

Cast

asBits / asUInt / asSInt / asBool	
x.assignFromBits(y : Bits)	Can be used to assign a Bits into something else.
x.assignFromBits(y : Bits,hi:Int,lo:Int)	
x.assignFromBits(y : Bits, offset:Int, bitCount:BitCount)	

Range

myBits(7 downto 0) //8 bits	myBits(0 to 5) //6 bits
myBits(0 until 5) // 5 bits	myBits(5) //bit 5
myUInt := (default -> true)	myUInt := (myUInt.range -> true)
myUInt := (3 -> true, default -> false)	myUInt := ((3 downto 1) -> true, default -> false)
val myBool = myUInt === U(myUInt.range -> true)	

Units

Hz, kHz, MHz, GHz, THz	val freq: HertzNumber = 1 kHz
fs, ps, ns, us, ms, s, mn, hr	val time: TimeNumber = 2 ms
Bytes, kB, MB, GB, TB	val size:BigInt = 4MB
bits, bit	val myBits:BitCount = 3 bits

Basetype Functions

Bool	.set, .clear, .rise, .fall, .setWhen(cond), .clearWhen(cond)
Bits	resize(y:Int), .resized, .range, .high, x(hi,lo), x(offset,width bits), x(index), .msb, .lsb, xorR, .orR, .andR, .clearAll,
SInt	.setAll, .setAllTo(Booleen), setAllTo(Bool)

Conditional / Mux

myBits.mux(0 -> (io.src0 & io.src1), 1 -> (io.src0 io.src1), default -> (io.src0))	when(cond1){ }elsewhen(cond2){ }otherwise{ }	switch(x){ is(value1){ } is(value2){ } default{ } }
cond ? whenTrue whenFalse	Select(cond1 -> value1, cond2 -> value2, default -> value3)	
Mux(cond,whenTrue,whenFalse)		

Basetype Operators

	x + y x - y x * y	x < y x > y x <= y x >= y	x === y x != y	x >> y x << y x << y x >> y	X & y x y x ^ y	X && y x y	!x	~x	@@	##
Bool			✓		✓	✓	✓	✓		✓
Bits			✓	✓	✓			✓		✓
SInt UInt	✓	✓	✓	✓	✓			✓	✓	

(##, @@ for concatenation) (|>>, |<< shift without resizing)

Bundle

case class RGB(width:Int) extends Bundle{ val red, green, blue = UInt(width bits) def isBlack = red === 0 & green === 0 & blue === 0 }	val io = new Bundle{ val a = in Bits(32 bits) val b = in(Rgb(config)) val c = out UInt(8 bits) }
--	--

class Bus(val config: BusConfig) extends Bundle with IMasterSlave{ val addr = UInt(config.addrWidth bits) val dataWr, dataRd = Bits(config.dataWidth bits) val cs,rw = Bool def asMaster(): Unit = { out(addr, dataWr, cs, rw) in(dataRd) } }	
val io = new Bundle{ val masterBus = master(Bus(BusConfig)) val slaveBus = slave(Bus(BusConfig)) }	

Blackbox

class Ram(wordCount: Int) extends BlackBox{ val io = new Bundle{ val clk = in Bool val addr = in UInt(32 bits) ... }	
addGeneric("wordcount", wordCount) noloPrefix() // remove io_ prefix mapClockDomain(clock=io.clk) // map to the current clockDomai }	

Component

class AndGate(width : Int) extend Component{ val io = new Bundle{ val value = out Bits(width bits) val in1,in2 = in Bits(width bits) } io.value := io.in1 & io.in2 }
--

Directions

in/out(T)	in/out Bool/Bits/UInt/SInt(x bits)
master/slave(T)	master/slave Stream/Flow(T)

Area

val myCounter = new Area{ val tick = Bool ... }
io.output := myCounter.tick

ClockDomain

	val config = ClockDomainConfig(clockEdge = RISING, // FALLING resetKind = ASYNC, // SYNC, BOOT resetActiveLevel = LOW, // HIGH softResetActiveLevel = LOW, // HIGH clockEnableActiveLevel = LOW // HIGH)
Configuration	
Clock Domain	val myCD = ClockDomain(ioClock,ioReset, config)
Area	val coreArea = new ClockingArea(myCD){ val myReg = Reg(UInt(32 bits)) // myCD clocked ... }
External Clock	val myCD = ClockDomain.external("clockName")
ClockDomain.current // Get the current clock domain ClockDomain.current.readClockWire // read current clock	
Misc	ClockEnableArea, SlowArea, ResetArea

RAM

Declaration	val myRAM = Mem(type,size:Int) val myROM = Mem(type,initialContent : Array[Data])
Write	mem(address) := data mem.write(address, data, [mask])
Read	myOutput := mem(x) // Asynchronous read mem.readAsync(address,[readUnderWrite]) mem.readSync(address,[enable],[readUnderWrite])
Read/Write	mem.readWriteSync(address,data,enable,write)

HDL Generation

SpinalVhdl(new MyTopLevel())	SpinalVerilog(new MyTopLevel())
SpinalConfig(mode = Verilog, // VHDL targetDirectory="temp/myDesign",).generate(new myComponent())	
val report = SpinalVhdl(new myTopLevel()) report.printPruned()	

Template

import spinal.core._ // import the core class MyTopLevel() extends Component { //Define a Component val io = new Bundle { val a,b = in Bool val c = out Bool } io.c := io.a & io.b }
object MyMain { def main(args: Array[String]) { SpinalVhdl(new MyTopLevel()) //Generate a VHDL file }

Utils

log2Up(x : BigInt)	Number of bit needed to represent x
isPow2(x : BigInt)	Return true if x is a power of two
Cat(x: Data*)	Concatenate all arguments

Function

// Function to multiply an UInt by a scala Float value. def coef(value : UInt,by : Float) : UInt = (value * U((255*by).toInt,8 bits) >> 8)
def clear() : Unit = counter := 0 // Clear the counter
def sinus(size:Int, res:BitCount) = { (0 to size).map (i => U((((Math.sin(i)+1) * Math.pow(2,res.value)/2).toInt)) } // memory initialised with a sinus val mySinus = Mem(UInt(16 bits), sinus(1024, 16 bits))

Assertion

assert(assertion = cond, message = "My message", severity = ERROR //WARNING, NOTE, FAILURE)

Enum

object MyEnum extends SpinalEnum{ val s0, s1, ..., sn = newElement }
val state = RegInit(MyEnum.s0)