Spinal CheatSheet - Lib

v0.1

Stream

Interface	valid, ready, payload
Example	val myStream = Stream(T) val myStream = master/slave Stream(T)
Connection	
slave << master master >> slave	Connect the slave to the master stream
s <-< m m >-> s	Connect with a register stage (1 lactency)
s </< m m >/> s	Connect with a register stage + mux(0 latency)
s <-/< m m >-/> s	bandwith divided by 2
Function	.haltWhen(cond), .throwWhen(cond), .queue(size:Int), .fire, .isStall
StreamFifo	val myFifo = StreamFifo(dataType=Bits(8 bits), depth=128) // .push, .pop, .flush, .occupancy
StreamFifoCC	val myFifo = StreamFifoCC(dataType=Bits(8 bits), depth=128, pushClock=clockA, popClock=clockB)
StreamCCByToggle	val bridge = StreamCCByToggle(dataType=Bits(8 bits), inputClock=clockA, outputClock=clockB)
StreamArbiter	val arbitredABC = StreamArbiterFactory.arbitration.lock .onArgs(Seq[Stream[T]])/Stream[T]*)) **Arbitration** : lowerFirst, roundRobin, sequentialOrder **Lock** : noLock, transactionLock, fragmentLock val arbitredDEF = StreamArbiterFactory.lowerFirst.noLock.onArgs(streamA, streamB, streamC)
StreamFork	val fork = new StreamFork(T, 2) fork.io.input << input // Connect the input stream fork.io.outputs(0) //Get the fork stream 0
StreamDispatcher Sequencial	val dispatchedStreams = StreamDispatcherSequencial(input=inputStream, outputCount=3)
StreamMux	val outStream = StreamMux(UInt, Seq[Stream[T]]/Vec[Stream[T]])
StreamDemux	val demux = StreamDemux(T, portCount :Int) //demux.io.select, demux.io.input, demux.io.output

StreamJoin, translateWith	val writeJoinEvent =
	StreamJoin.arg(bus.writeCmd,bus.writeData)
	val writeRsp = AxiLite4B(bus.config)
	bus.writeRsp <-< writeJoinEvent.translateWith(writeRsp)

Flow

Interface	valid, payload
Example	val myFlow = Flow(T) val myFlow = master/slave Flow(T)
Connection	s << m, m >> s, s <-< m, m >-> s
Function	.toReg(), .throwWhen(cond)

Fragment

Interface	last, payload
Example	val myStream = slave Stream (Fragment[T]) val myFlow = master Flow (Fragment[T])
Function	.first, .tail, .isFirst, .isTail, .isLast, .insertHeader(T)

State machine

```
val sm = new StateMachine{ // Style A
  always{
    when(cond) { goto (sS1) }
  }
  val sS1:State=new State with
EntryPoint{
    onEntry{ }
    whenIsActive{ goto (sS2) }
    onExit {}
  }
  val sS2:State= new State{
    whenIsActive{ goto(sSx) }
  }
}
```

```
val sm = new StateMachine{ // Style B
  val sS1 = new State with EntryPoint
  val sS2 = new State
  always{
    when(cond) { goto (sS1) }
  }
  sS1
    .onEntry()
    .whenIsActive{ goto (sS2) }
    .onExit()
  sS2
    .whenIsActive{ goto(sSx) }
}
```

Delay : val sDelay : State = new StateDelay(40) { whenCompleted(goto(stateH)) }

Inner SM : val stateC = new StateFsm(fsm=internalFsm()) { whenCompleted{ goto(stateD) }}

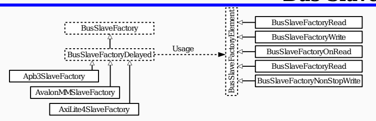
Parallel SM : val stateD = new StateParallelFsm (internalFsmA(), internalFsmB()){ whenCompleted{ goto(stateE) } }

```
def internalFsm() = new StateMachine { // Internal SM
  val counter = Reg(UInt(8 bits)) init (0)
  val stateA: State = new State with EntryPoint { whenIsActive { goto(stateB) } }
  val stateB: State = new State {
    onEntry (counter := 0)
    whenIsActive {
      when(counter === 4) { exit() }
      counter := counter + 1
    }
  }
}
```

Utils

Delay(singal, cycle)	Delay a signal of x clock	History(T, len, [when, init])	Shit register
toGray(x:UInt)	Return the gray value	fromGray(x : Bits)	Return the UInt value
toGray(x:UInt)	Return the gray value	fromGray(x : Bits)	Return the UInt value
Reverse(T)	Flip all bits	Endianness(T,[base])	Big-Endian <-> Littre-Endian
OHToUInt(Seq[Bool]/BitVector)	Index of the single bit	CountOne(Seq[Bool]/BitVector)	Number of bits set
MajorityVote(Seq[Bool]/BitVector)	True if nuber bit set is < x.size/2	BufferCC(T)	Buffer clock domain
LatencyAnalysis(Node*)	Return the shortest path	Counter(BigInt)	Counter

Bus Slave Factory



BusSlaveFactory (Base Functions) :
.busDataWidth, .read(that,address,bitOffset), .write(that,address,bitOffset),
.onWrite(address)(doThat), .onRead(address)(doThat), .nonStopWrite(that,bitOffset)

BusSlaveFactory (Derived Functions) :
.readAndWrite(that,address,bitOffset), .readMultiWord(that,address),
.writeMultiWord(that,address), .createWriteOnly(dataType,address,bitOffset),
.createReadWrite(dataType,address,bitOffset),
.createAndDriveFlow(dataType,address,bitOffset), .drive(that,address,bitOffset),
.driveAndRead(that,address,bitOffset) , .driveFlow(that,address,bitOffset),
.readStreamNonBlocking(that,address,validBitOffset,payloadBitOffset),
.doBitsAccumulationAndClearOnRead(that,address,bitOffset)

```
class AvalonUartCtrl(uartCtrlConfig : UartCtrlGenerics, rxFifoDepth : Int) extends
Component{
  val io = new Bundle{
    val bus = slave(AvalonMM(AvalonMMUartCtrl.getAvalonMMConfig)) /tab2>
    val uart = master(Uart())
  }
  val uartCtrl = new UartCtrl(uartCtrlConfig)
  io.uart <> uartCtrl.io.uart
  val busCtrl = AvalonMMSlaveFactory(io.bus)
  busCtrl.driveAndRead(uartCtrl.io.config.clockDivider,address = 0)
  busCtrl.driveAndRead(uartCtrl.io.config.frame,address = 4)
  busCtrl.createAndDriveFlow(Bits(uartCtrlConfig.dataWidthMax bits),address =
8).toStream >-> uartCtrl.io.write

  busCtrl.readStreamNonBlocking(uartCtrl.io.read.toStream.queue(rxFifoDepth),address
= 12, validBitOffset = 31, payloadBitOffset = 0) }
```

Master/Slave interface

```
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(): this.type = {
    master(addr, dataWr, cs, rw) // Master drive
these signals
    slave(dataRd) // Slave drive this signal
  }
}
```

```
val io = new Bundle{
  val masterBus =
master(Bus(BusConfig))
  val slaveBus =
slave(Bus(BusConfig))
}
```

UART Controller

```
val uartCtrl = new UartCtrl()
uartCtrl.io.config.setClockDivider(921600)
uartCtrl.io.config.frame.dataLength := 7 // 8 bits
uartCtrl.io.config.frame.parity := UartParityType.NONE // NONE, EVEN, ODD
uartCtrl.io.config.frame.stop := UartStopType.ONE // ONE, TWO
uartCtrl.io.uart <> io.uart
```