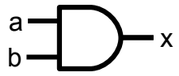
SpinalHDL CheatSheet – Symbolic

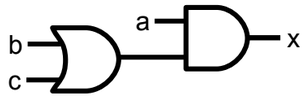
Combinatorial

```
val x = ~a
```

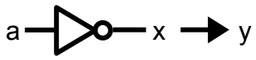

```
val x = a && b
```



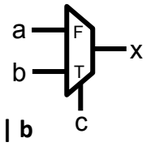

```
val x = a && (b || c)

val tmp = b || c
val x = a && tmp
```




```
val x, y = Bool()
x := ~a
y := x
```




```
val x = Mux(c, a, b)
```



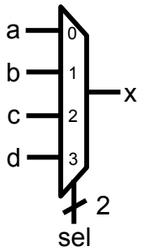

```
val x = c ? a | b
```

```
val x = Bool()
when(c){
  x := b
} otherwise {
  x := a
}
```

```
val sel = UInt(2 bits)
val x = Bool()
switch(sel){
  is(0) {x := a}
  is(1) {x := b}
  is(2) {x := c}
  is(3) {x := d}
}
```

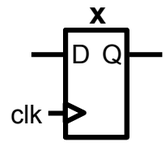
```
val sel = UInt(2 bits)
val vec = Vec(a,b,c,d)
val x = vec(sel)
```

```
val sel = UInt(2 bits)
val x = sel.mux(
  0 -> a,
  1 -> b,
  2 -> c,
  3 -> d
)
```

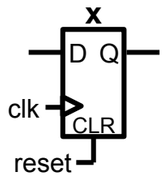


Register

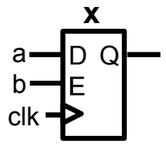
```
val x = Reg(Bool)
```



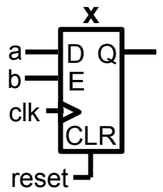

```
val x = Reg(Bool) init(False)
```



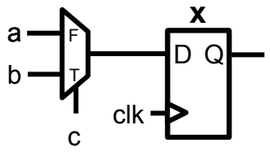

```
val x = Reg(Bool)
when(b){
  x := a
}
```



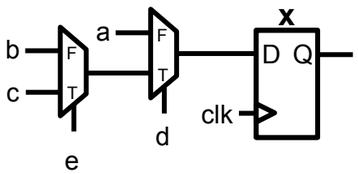

```
val x = Reg(Bool) init(False)
when(b){
  x := a
}
```




```
val x = Reg(Bool)
x := a
when(c){
  x := b
}
```

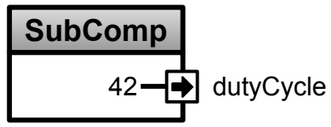



```
val x = Reg(Bool)
x := a
when(d){
  when(e){
    x := c
  } otherwise {
    x := b
  }
}
```

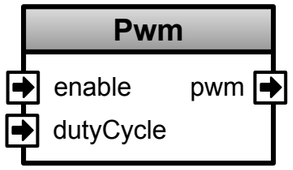


Component

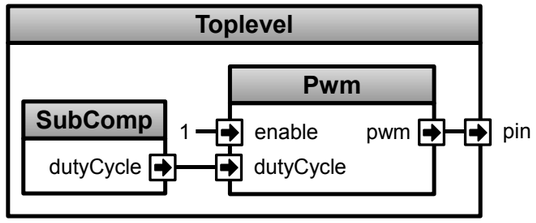
```
class SubComp extends Component{
  val io = new Bundle {
    val dutyCycle = out UInt(16 bits)
  }
  io.dutyCycle := 42
}
```



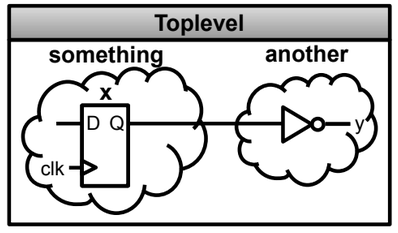

```
class Pwm(width : Int) extends Component{
  val io = new Bundle{
    val dutyCycle = in UInt(width bits)
    val enable    = in Bool
    val pwm       = out Bool
  }
  // ...
}
```



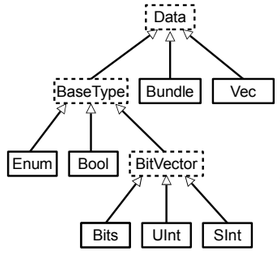

```
class Toplevel extends Component{
  val io = new Bundle{
    val pin = out Bool
  }
  val subComp = new SubComp
  val ctrl = new Pwm(width = 10)
  ctrl.io.enable := True
  ctrl.io.dutyCycle := subComp.io.dutyCycle << 6
  ctrl.io.pwm <> io.pin  //Autoconnect
}
```




```
class Toplevel extends Component{
  //...
  val something = new Area{
    val x = Reg(Bool)
  }
  val another = new Area{
    val y = ~something.x
  }
  //..
}
```



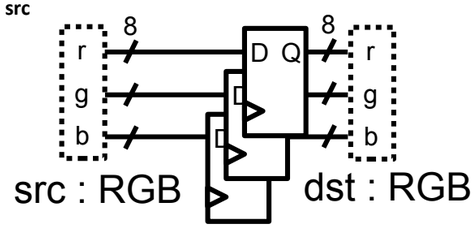
Datatype/Interface



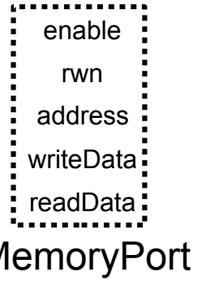

```
val x = Bool()
val x = Bits(8 bits)
val x = UInt(8 bits)
val x = SInt(8 bits)
val x = Vec(UInt(8 bits), size = 4)

object State extends SpinalEnum{
  val IDLE, S0,S1,S2 = newElement()
}
val x = State()
```

```
case class RGB(channelWidth : Int) extends Bundle{
  val r,g,b = UInt(channelWidth bits)
  def isBlack : Bool = r === 0 && g === 0 && b == 0
}
```


```
val src = RGB(8)
val dst = Reg(RGB(channelWidth = 8))
dst := src
```




```
case class MemoryPort( addressWidth : Int,
                       dataWidth : Int) extends Bundle
                                        with IMasterSlave {
  val enable    = Bool
  val rwn       = Bool
  val address   = Bits(addressWidth bits)
  val writeData = Bits(dataWidth bits)
  val readData  = Bits(dataWidth bits)

  override def asMaster(): Unit = {
    out(enable,rwn,address,writeData)
    in(readData)
  }
}
```




```
class MappedFifo( packetWidth : Int,
                  fifoDepth : Int) extends Component{
  val io = new Bundle{
    val apb = slave(MemoryPort(
      addressWidth = 32,
      dataWidth = 32
    ))
    val pop = master(Stream(Bits(packetWidth bits)))
  }
  // ...
}
```