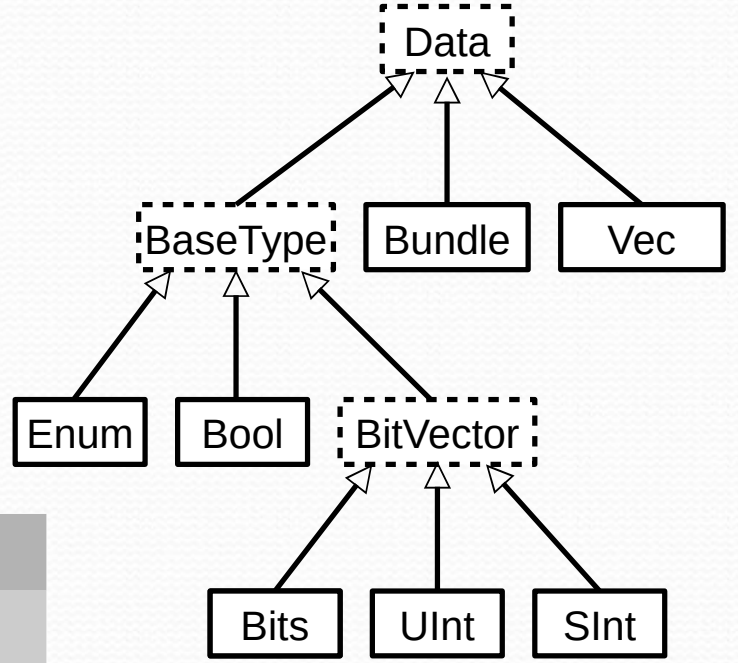
SpinalHDL

Basics and VHDL/Verilog Equivalences

Types



SpinalHDL	VHDL	Verilog
Bool()	std_logic	
Bits(8 bits)	std_logic_vector(7 downto 0)	[7:0]
UInt(8 bits)	unsigned(7 downto 0)	unsigned [7:0]
SInt(8 bits)	signed(7 downto 0)	signed [7:0]
Bundle	record	struct
Vec(SInt(8 bits), 4)	array(0 to 4) of signed(7 downto 0)	signed [7:0] … [0:3]
Data	-	-

Arithmetics returned width

Operator kind	Examples	Returned width
Boolean	&& || === =/=	1 (Bool)
Bitwise	& | ^	left, right (both width should match)
Add Sub	+ -	max(left, right)
Mult	*	left + right
Shift with width changes	>> <<	left +- maximum shift amount
Shift with fixed width	|>> |<<	left

Declaration/Assignement

Example	Description
val x = UInt(8 bits)	Hardware declaration
x := y	Hardware assignement (like <= in VHDL/Verilog)

```
val something = UInt(8 bits)
something := somethingElse
```

Semantic

- Constructing hardware by
  - Instanciating things
  - Defining rules
  - Last rule win
  - Using implicit clock domains

```
val result = UInt(8 bits)
result := 0
when(something){
    result := 42
    when(somethingElse){
        result := 0xEE
    }
}
```

```
val counter = Reg(UInt(8 bits))
when(something){
    counter := counter + 1
}
when(clearFlag){
    counter := 0
}
```

```
when(softReset){
    counter := 0
    result := 0x88
}
```

Conditional scope can be used by combinatorial and register assignements !

Variable

Example	Description
var y = UInt(8 bits)	Hardware variable declaration
y \= x	Hardware variable assignement (like variable in VHDL and = in Verilog)

```
val a      = UInt (size bits)
val b      = UInt (size bits)
val result = UInt (size bits)

var c = False
for (i <- 0 until size) {
    val x = io.a(i)
    val y = io.b(i)

    io.result(i) := x ^ y ^ c
    c \= (x && y) || (x && c) || (y && c);
}
```

OOP rules apply there

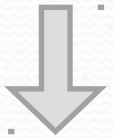
```
// !! forward reference => result is null =>  error !!
x := result
val result = UInt(8 bits)
```

Literals

```
val myBool = Bool()
myBool := False
myBool := True

val myUInt = UInt(8 bits)
myUInt := "0001_1100"
myUInt := "xEE"
myUInt := 42
myUInt := 0x42
myUInt := U(54, 8 bits)
myUInt := ((3 downto 0) -> myBool,default -> true)
when(myUInt === U(myUInt.range -> true)){
    myUInt(3) := False
}

val myConstant = U"0001_1100"
```

but not only

```
val myBool = False
when(something){
    myBool := True
}
```



```
process(something)
begin
   myBool <= '0';
   if something = '1' then
      myBool <= '1';
   end if;
end process;
```

Type manipulations

```
val uint8 = UInt(8 bits)
val uint12 = UInt(12 bits)
uint8 := uint12 //Error
uint12 := uint8 //Error
uint8 := uint12.resize(8)
uint8 := uint12(7 downto 0)
uint8 := uint12(uint8.range)
uint8 := uint12.resized

val bits4 = Bits(4 bits)
uint8 := bits4.asUInt.resized
```

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

val bitsIn, bitsOut = Bits(12 bits)
val color = RGB(4)
color.assignFromBits(bitsIn)
bitsOut := color.asBits
```

Component / Entity / Architecture / Module

```
import spinal.core._

class Adder(width : Int) extends Component{
    val io = new Bundle {
        val a,b = in UInt(width bits)
        val result = out UInt(width bits)
    }
    val adder = UInt(width bits)
    adder := io.a + io.b
    io.result := adder
}
```

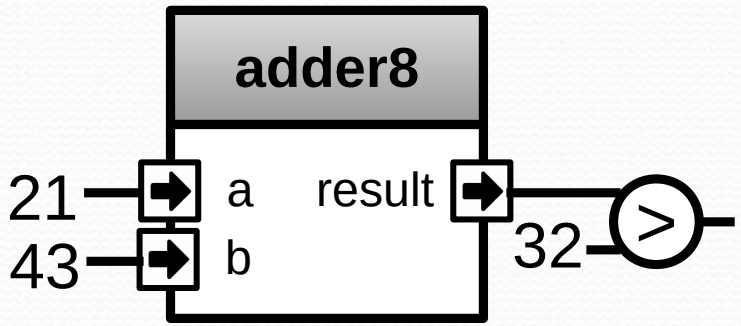
```
module Adder #(parameter width) (
    input  [width-1:0] io_a,
    input  [width-1:0] io_b,
    output [width-1:0] io_result
  );
  wire [width-1:0] adder;
  assign adder = (io_a + io_b);
  assign io_result = adder;
endmodule
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity Adder is
  generic (
    width : integer
  );
  port(
    io_a : in unsigned(width-1 downto 0);
    io_b : in unsigned(width-1 downto 0);
    io_result : out unsigned(width-1 downto 0)
  );
end Adder;

architecture arch of Adder is
  signal adder : unsigned(width-1 downto 0);
begin
  io_result <= adder;
  adder <= io_a + io_b;
end arch;
```

No component binding required, access things in a OOP manner

```
class Adder(width : Int) extends Component{
    val io = new Bundle {
        val a,b = in UInt(width bits)
        val result = out UInt(width bits)
    }
    val adder = UInt(width bits)
    adder := io.a + io.b
    io.result := adder
}
```

```
val adder8 = new Adder(8)
adder8.io.a := 21
adder8.io.b := 43
when(adder8.io.result > 32){
    //...
}
```



Function and namespaces

```
val isZero = Bool()
val shifted = UInt(9 bits)
val square  = UInt(16 bits)
def load(value : UInt) : Unit = {
    require(widthOf(value) == 8)
    isZero := value === 0
    shifted := value << 1
    square := value * value
}

when(something){
    load(U"1110_0000")
} otherwise {
    load(U"0010_0010")
}
```

```
val landaLogic = new Area{
    val a,b = UInt(8 bits)
    val result = a + b
    val flag = False
}

val somethereElse = new Area{
    val flag = False || landaLogic.result.lsb
}

when(landaLogic.result === 88){
    somethereElse.flag := False
}
```