SpinalHDL Documentation

SpinalHDL contributors

Jul 20, 2021

CONTENTS

CHAPTER

ONE

FAQ

1.1 What is the overhead of SpinalHDL generated RTL compared to human written VHDL/Verilog?

The overhead is null, SpinalHDL is not an HLS approach. Its goal is not to translate any arbitrary code into RTL, but to provide a powerful language to describe RTL and raise the abstraction level.

1.2 What if SpinalHDL becomes unsupported in the future?

This question has two sides:

1. SpinalHDL generates VHDL/Verilog files, which means that SpinalHDL will be supported by all EDA tools for many decades.
2. If there is a bug in SpinalHDL and there is no longer support to fix it, it's not a deadly situation, because the SpinalHDL compiler is fully open source. For simple issues, you may be able to fix the issue yourself in few hours. Remember how much time it takes to EDA companies to fix issues or to add new features in their closed tools.

1.3 Does SpinalHDL keep comments in generated VHDL/verilog?

No, it doesn't. Generated files should be considered as a netlist. For example, when you compile C code, do you care about your comments in the generated assembly code?

1.4 Could SpinalHDL scale up to big projects?

Yes, some experiments were done, and it appears that generating hundreds of 3KLUT CPUs with caches takes around 12 seconds, which is a ridiculously short time compared to the time required to simulate or synthesize this kind of design.

1.5 How SpinalHDL came to be

Between December 2014 and April 2016, it was as a personal hobby project. But since April 2016 one person is working full time on it. Some people are also regularly contributing to the project.

1.6 Why develop a new language when there is VHDL/Verilog/SystemVerilog?

This page is dedicated to this topic.

1.7 How to use an unreleased version of SpinalHDL (but committed on git)?

For instance, if you want to try the `dev` branch, do the following in a dummy folder :

```
git clone https://github.com/SpinalHDL/SpinalHDL.git -b dev
cd SpinalHDL
sbt clean publishLocal
```

Then in your project, don't forget to update the SpinalHDL version specified in the build.sbt file, see
https://github.com/SpinalHDL/SpinalTemplateSbt/blob/master/build.sbt#L10
To know which version you have to set, look in
https://github.com/SpinalHDL/SpinalHDL/blob/dev/project/Version.scala#L7

CHAPTER

TWO

SUPPORT

2.1 Communication channels

For bug reporting and feature requests, do not hesitate to create github issues:
https://github.com/SpinalHDL/SpinalHDL/issues

For questions about SpinalHDL syntax and live talks, a Gitter channel is available:
https://gitter.im/SpinalHDL/SpinalHDL

For questions, you can also use the forum StackOverflow with the tag SpinalHDL :
https://stackoverflow.com/

A Google group is also available. Feel free to post whatever subject you want related to SpinalHDL:
https://groups.google.com/forum/#!forum/spinalhdl-hardware-description-language

2.2 Commercial support

If you are interested in a presentation, a workshop, or consulting, do not hesitate to contact us by email:
spinalhdl@gmail.com

CHAPTER

THREE

USERS

3.1 Companies

- QsPin, Belgium

3.2 Repositories

- J1Sc Stack CPU
- VexRiscv CPU and SoC

CHAPTER

FOUR

GETTING STARTED

4.1 Getting Started

SpinalHDL is a hardware description language written in Scala, a statically-typed functional language using the Java virtual machine (JVM). In order to start programming with *SpinalHDL*, you must have a JVM as well as the Scala compiler. In the next section, we will explain how to download those tools if you don't have them already.

4.1.1 Requirements / Things to download to get started

Before you download the SpinalHDL tools, you need to install:

- A Java JDK, which can be downloaded from here for instance.
- A Scala 2.11.X distribution, which can be downloaded here (not required if you use SBT).
- The SBT build tool, which can be downloaded here.

Optionally:

- An IDE (which is not compulsory). We advise you to get IntelliJ with its Scala plugin.
- Git, which is a tool for version control.

4.1.2 How to start programming with SpinalHDL

Once you have downloaded all the requirements, there are two ways to get started with SpinalHDL programming.

1. *The SBT way* : If you already are familiar with the SBT build system and/or if you don't need an IDE.
2. *The IDE way* : Get a project already set up for you in an IDE and start programming right away.

The SBT way

We have prepared a ready-to-go project for you on Github.

- Either clone or download the "getting started" repository.
- Open a terminal in the root of it and run `sbt run`. When you execute it for the first time, the process could take some time as it will download all the dependencies required to run *SpinalHDL*.

Normally, this command must generate an output file `MyTopLevel.vhd`, which corresponds to the top level *SpinalHDL* code defined in `src\main\scala\MyCode.scala`, which corresponds to the *most simple SpinalHDL example*

From a clean Debian distribution you can type the following commands into the shell. The commands will install Java, Scala, SBT, download the base project, and generate the corresponding VHDL file. Don't worry if it takes some time the first time that you run it.

```
sudo apt-get install openjdk-8-jdk
sudo apt-get install scala
echo "deb https://repo.scala-sbt.org/scalasbt/debian all main" | sudo tee /etc/apt/
↪sources.list.d/sbt.list
echo "deb https://repo.scala-sbt.org/scalasbt/debian /" | sudo tee /etc/apt/sources.
↪list.d/sbt_old.list
curl -sL "https://keyserver.ubuntu.com/pks/lookup?op=get&
↪search=0x2EE0EA64E40A89B84B2DF73499E82A75642AC823" | sudo apt-key add
sudo apt-get update
sudo apt-get install sbt
git clone https://github.com/SpinalHDL/SpinalTemplateSbt.git SpinalTemplateSbt
cd SpinalTemplateSbt
sbt run   # select "mylib.MyTopLevelVhdl" in the menu
ls MyTopLevel.vhd
```

SBT in a environnement isolated from internet

Normally, SBT uses online repositories to download and cache your projects dependencies, this cache is located in your `home/.ivy2` folder. The way to set up an internet-free environnement is to copy this cache from an internet-full environnement where the cache was already filled once, and copy it over to your internet-less environnement.

You can get a portable SBT setup here:
https://www.scala-sbt.org/download.html

The IDE way, with IntelliJ IDEA and its Scala plugin

In addition to the aforementioned *requirements* , you also need to download the IntelliJ IDEA (the free *Community edition* is enough). When you have installed IntelliJ, also check that you have enabled its Scala plugin (install information can be found here).

And do the following :

- Either clone or download the "getting started" repository.
- In *Intellij IDEA*, "import project" with the root of this repository, the choose the *Import project from external model SBT* and be sure to check all boxes.
- In addition, you might need to specify some path like where you installed the JDK to *IntelliJ*.
- In the project (Intellij project GUI), right click on `src/main/scala/mylib/MyTopLevel.scala` and select "Run MyTopLevel".

This should generate the output file `MyTopLevel.vhd` in the project directory, which implements a simple 8-bit counter.

4.1.3 A very simple SpinalHDL example

The following code generates an and gate between two one-bit inputs.

```
import spinal.core._

class AND_Gate extends Component {

  /**
    * This is the component definition that corresponds to
```

(continues on next page)

(continued from previous page)

```
    * the VHDL entity of the component
    */
  val io = new Bundle {
    val a = in Bool()
    val b = in Bool()
    val c = out Bool()
  }

  // Here we define some asynchronous logic
  io.c := io.a & io.b
}

object AND_Gate {
  // Let's go
  def main(args: Array[String]) {
    SpinalVhdl(new AND_Gate)
  }
}
```

As you can see, the first line you have to write in SpinalHDL is `import spinal.core._` which indicates that we are using the *Spinal* components in the file.

Generated code

Once you have successfully compiled your code, the compiler should have emitted the following VHDL code:

```
package pkg_enum is
  ...
end pkg_enum;

package pkg_scala2hdl is
  ...
end  pkg_scala2hdl;

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

library work;
use work.pkg_scala2hdl.all;
use work.all;
use work.pkg_enum.all;


entity AND_Gate is
  port(
    io_a : in std_logic;
    io_b : in std_logic;
    io_c : out std_logic
  );
end AND_Gate;

architecture arch of AND_Gate is

begin
```

(continues on next page)

(continued from previous page)

```
  io_c <= (io_a and io_b);
end arch;
```

4.1.4 What to do next?

It's up to you, but why not have a look at what the *types* are in SpinalHDL or discover what primitives the language provides to describe hardware components? You could also have a look at our *examples* to see some samples of what you could do next.

4.2 Motivation

Redirection to https://github.com/SpinalHDL/SpinalDoc/blob/master/presentation/en/workshop/taste.pdf

4.3 Presentation

Redirection to https://github.com/SpinalHDL/SpinalDoc/blob/master/presentation/en/presentation.pdf

4.4 Scala Guide

Important: Variables and functions should be defined into `object`, `class`, `function`. You can't define them on the root of a Scala file.

4.4.1 Basics

Types

In Scala, there are 5 major types:

Type	Literal	Description
Boolean	true, false	
Int	3, 0x32	32 bits integer
Float	3.14f	32 bits floating point
Double	3.14	64 bits floating point
String	"Hello world"	UTF-16 string

Variables

In Scala, you can define a variable by using the `var` keyword:

```
var number : Int = 0
number = 6
number += 4
println(number) // 10
```

Scala is able to infer the type automatically. You don't need to specify it if the variable is assigned at declaration:

```
var number = 0   //The type of 'number' is inferred as an Int during compilation.
```

However, it's not very common to use `var` in Scala. Instead, constant values defined by `val` are often used:

```
val two   = 2
val three = 3
val six   = two * three
```

Functions

For example, if you want to define a function which returns `true` if the sum of its two arguments is bigger than zero, you can do as follows:

```
def sumBiggerThanZero(a: Float, b: Float): Boolean = {
  return (a + b) > 0
}
```

Then, to call this function, you can write:

```
sumBiggerThanZero(2.3f, 5.4f)
```

You can also specify arguments by name, which is useful if you have many arguments:

```
sumBiggerThanZero(
  a = 2.3f,
  b = 5.4f
)
```

Return

The `return` keyword is not necessary. In absence of it, Scala takes the last statement of your function as the returned value.

```
def sumBiggerThanZero(a: Float, b: Float): Boolean = {
  (a + b) > 0
}
```

Return type inferation

Scala is able to automatically infer the return type. You don't need to specify it:

```
def sumBiggerThanZero(a: Float, b: Float) = {
  (a + b) > 0
}
```

Curly braces

Scala functions don't require curly braces if your function contains only one statement:

```
def sumBiggerThanZero(a: Float, b: Float) = (a + b) > 0
```

Function that returns nothing

If you want a function to return nothing, the return type should be set to `Unit`. It's equivalent to the C/C++ `void` type.

```
def printer(): Unit = {
  println("1234")
  println("5678")
}
```

Argument default values

You can specify a default value for each argument of a function:

```
def sumBiggerThanZero(a: Float, b: Float = 0.0f) = {
  (a + b) > 0
}
```

Apply

Functions named `apply` are special because you can call them without having to type their name:

```
class Array() {
  def apply(index: Int): Int = index + 3
}

val array = new Array()
val value = array(4)   //array(4) is interpreted as array.apply(4) and will return 7
```

This concept is also applicable for Scala `object` (static)

```
object MajorityVote {
  def apply(value: Int): Int = ...
}

val value = MajorityVote(4) // Will call MajorityVote.apply(4)
```

Object

In Scala, there is no `static` keyword. In place of that, there is `object`. Everything defined inside an `object` definition is static.

The following example defines a static function named `pow2` which takes a floating point value as parameter and returns a floating point value as well.

```
object MathUtils {
  def pow2(value: Float): Float = value * value
}
```

Then you can call it by writing:

```
MathUtils.pow2(42.0f)
```

Entry point (main)

The entry point of a Scala program (the main function) should be defined inside an object as a function named `main`.

```
object MyTopLevelMain{
  def main(args: Array[String]) {
    println("Hello world")
  }
}
```

Class

The class syntax is very similar to Java. Imagine that you want to define a `Color` class which takes as construction parameters three Float values (r,g,b) :

```
class Color(r: Float, g: Float, b: Float) {
  def getGrayLevel(): Float = r * 0.3f + g * 0.4f + b * 0.4f
}
```

Then, to instantiate the class from the previous example and use its `getGrayLevel` function:

```
val blue = new Color(0, 0, 1)
val grayLevelOfBlue = blue.getGrayLevel()
```

Be careful, if you want to access a construction parameter of the class from the outside, this construction parameter should be defined as a `val`:

```
class Color(val r: Float, val g: Float, val b: Float) { ... }
...
val blue = new Color(0, 0, 1)
val redLevelOfBlue = blue.r
```

Inheritance

As an example, suppose that you want to define two classes, `Rectangle` and `Square`, which extend the class `Shape`:

```
class Shape {
  def getArea(): Float
}

class Square(sideLength: Float) extends Shape {
  override def getArea() = sideLength * sideLength
}

class Rectangle(width: Float, height: Float) extends Shape {
  override def getArea() = width * height
}
```

Case class

Case class is an alternative way of declaring classes.

```
case class Rectangle(width: Float, height: Float) extends Shape {
  override def getArea() = width * height
}
```

Then there are some differences between `case class` and `class` :

- case classes don't need the `new` keyword to be instantiated.
- construction parameters are accessible from outside; you don't need to define them as `val`.

In SpinalHDL, this explains the reasoning behind the coding conventions: it's in general recommended to use `case class` instead of `class` in order to have less typing and more coherency.

Templates / Type parameterization

Imagine you want to design a class which is a queue of a given datatype, in that case you need to provide a type parameter to the class:

```
class  Queue[T](){
  def push(that: T) : Unit = ...
  def pop(): T = ...
}
```

If you want to restrict the `T` type to be a sub class of a given type (for example `Shape`), you can use the `<: Shape` syntax :

```
class Shape() {
    def getArea(): Float
}
class Rectangle() extends Shape { ... }

class  Queue[T <: Shape]() {
  def push(that: T): Unit = ...
  def pop(): T = ...
}
```

The same is possible for functions:

```
def doSomething[T <: Shape](shape: T): Something = { shape.getArea() }
```

4.4.2 Coding conventions

Introduction

The coding conventions used in SpinalHDL are the same as the ones documented in the Scala Style Guide.

Some additional practical details and cases are explained in next pages.

class vs case class

When you define a `Bundle` or a `Component`, it is preferable to declare it as a case class.

The reasons are:

- It avoids the use of `new` keywords. Never having to use it is better than sometimes, under some conditions.
- A `case class` provides a `clone` function. This is useful in SpinalHDL when there is a need to clone a `Bundle`, for example, when you define a new `Reg` or a new `Stream` of some kind.
- Construction parameters are directly visible from outside.

[case] class

All classes names should start with a uppercase letter

```
class Fifo extends Component {

}

class Counter extends Area {

}

case class Color extends Bundle {

}
```

companion object

A companion object should start with an uppercase letter.

```
object Fifo {
  def apply(that: Stream[Bits]): Stream[Bits] = {...}
}

object MajorityVote {
  def apply(that: Bits): UInt = {...}
}
```

An exception to this rule is when the companion object is used as a function (only `apply` inside), and these `apply` functions don't generate hardware:

```
object log2 {
  def apply(value: Int): Int = {...}
}
```

function

A function should always start with a lowercase letter:

```
def sinTable = (0 until sampleCount).map(sampleIndex => {
  val sinValue = Math.sin(2 * Math.PI * sampleIndex / sampleCount)
  S((sinValue * ((1 << resolutionWidth) / 2 - 1)).toInt, resolutionWidth bits)
})

val rom =  Mem(SInt(resolutionWidth bit), initialContent = sinTable)
```

instances

Instances of classes should always start with a lowercase letter:

```
val fifo   = new Fifo()
val buffer = Reg(Bits(8 bits))
```

if / when

Scala `if` and SpinalHDL `when` should normally be written in the following way:

```
if(cond) {
  ...
} else if(cond) {
  ...
} else {
  ...
}

when(cond) {
  ...
}.elseWhen(cond) {
  ...
}.otherwise {
  ...
}
```

Exceptions could be:

- It's fine to omit the dot before `otherwise`.
- It's fine to compress an `if`/`when` statement onto a single line if it makes the code more readable.

switch

SpinalHDL `switch` should normally be written in the following way:

```
switch(value) {
  is(key) {

  }
  is(key) {

  }
  default {

  }
}
```

It's fine to compress an `is`/`default` statement onto a single line if it makes the code more readable.

Parameters

Grouping parameters of a `Component`/`Bundle` inside a case class is generally welcome because:

- Easier to carry/manipulate to configure the design
- Better maintainability

```
case class RgbConfig(rWidth: Int, gWidth: Int, bWidth: Int) {
  def getWidth = rWidth + gWidth + bWidth
}

case class Rgb(c: RgbConfig) extends Bundle {
  val r = UInt(c.rWidth bit)
  val g = UInt(c.gWidth bit)
  val b = UInt(c.bWidth bit)
}
```

But this should not be applied in all cases. For example: in a FIFO, it doesn't make sense to group the `dataType` parameter with the `depth` parameter of the fifo because, in general, the `dataType` is something related to the design, while the `depth` is something related to the configuration of the design.

```
class Fifo[T <: Data](dataType: T, depth: Int) extends Component {

}
```

4.4.3 Interaction

Introduction

SpinalHDL is, in fact, not an language: it's a regular Scala library. This could seem strange at first glance, but it is a very powerful combination.

You can use the whole Scala world to help you in the description of your hardware via the SpinalHDL library, but to do that properly, it's important to understand how SpinalHDL interacts with Scala.

How SpinalHDL works behind the API

When you execute your SpinalHDL hardware description, each time you use SpinalHDL functions, operators, or classes, it will build an in-memory graph that represents the netlist of your design.

Then, when the elaboration is done (instantiation of your top-level `Component` classes), SpinalHDL will do some passes on the graph that was constructed, and if everything is fine, it will flush that graph into a VHDL or Verilog file.

Everything is a reference

For example, if you define a Scala function which takes a parameter of type `Bits`, when you call it, it will be passed as a reference. As consequence of that, if you assign that argument inside the function, it has the same effect on the underlying `Bits` object as if you had assigned to it outside the function.

Hardware types

Hardware data types in SpinalHDL are the combination of two things:

- An instance of a given Scala type
- The configuration of that instance

For example `Bits(8 bits)` is the combination of the Scala type `Bits` and its `8 bits` configuration (as a construction parameter).

RGB example

Let's take an Rgb bundle class as example:

```
case class Rgb(rWidth: Int, gWidth: Int, bWidth: Int) extends Bundle {
  val r = UInt(rWidth bits)
  val g = UInt(gWidth bits)
  val b = UInt(bWidth bits)
}
```

The hardware data type here is the combination of the Scala `Rgb` class and its `rWidth`, `gWidth`, and `bWidth` parameterization.

Here is an example of usage:

```
// Define an Rgb signal
val myRgbSignal = Rgb(5, 6, 5)

// Define another Rgb signal of the same data type as the preceding one
val myRgbCloned = cloneOf(myRgbSignal)
```

You can also use functions to define various kinds of type factories (typedef):

```
// Define a type factory function
def myRgbTypeDef = Rgb(5, 6, 5)

// Use that type factory to create an Rgb signal
val myRgbFromTypeDef = myRgbTypeDef
```

Names of signals in the generated RTL

To name signals in the generated RTL, SpinalHDL uses Java reflections to walk through your entire component hierarchy, collecting all references stored inside the class attributes, and naming them with their attribute name.

This is why the names of every signal defined inside a function are lost:

```
def myFunction(arg: UInt) {
  val temp = arg + 1  // You will not retrieve the `temp` signal in the generated RTL
  return temp
}

val value = myFunction(U"000001") + 42
```

One solution if you want preserve the names of the internal variables in the generated RTL, is to use `Area`:

```
def myFunction(arg: UInt) new Area {
  val temp = arg + 1  // You will not retrieve the temp signal in the generated RTL
}

val myFunctionCall = myFunction(U"000001")  // Will generate `temp` with␣
↪`myFunctionCall_temp` as the name
val value = myFunctionCall.temp  + 42
```

Scala is for elaboration, SpinalHDL for hardware description

For example, if you write a Scala for loop to generate some hardware, it will generate the unrolled result in VHDL/Verilog.

Also, if you want a constant, you should not use SpinalHDL hardware literals but the Scala ones. For example:

```
// This is wrong, because you can't use a hardware Bool as construction parameter.␣
↪(It will cause hierarchy violations.)
class SubComponent(activeHigh: Bool) extends Component {
  // ...
}

// This is right, you can use all the Scala world to parameterize your hardware.
class SubComponent(activeHigh: Boolean) extends Component {
  // ...
}
```

Scala elaboration capabilities (if, for, functional programming)

All of Scala's syntax can be used to elaborate hardware designs, for instance, a Scala `if` statement could be used to enable or disable the generation of hardware:

```
val counter = Reg(UInt(8 bits))
counter := counter + 1
if(generateAClearWhenHit42) {  // Elaboration test, like an if generate in vhdl
  when(counter === 42) {       // Hardware test
    counter := 0
  }
}
```

The same is true for Scala `for` loops:

```
val value = Reg(Bits(8 bits))
when(something) {
  // Set all bits of value by using a Scala for loop (evaluated during hardware␣
↪elaboration)
  for(idx <- 0 to 7) {
    value(idx) := True
  }
}
```

Also, functional programming techniques can be used with many SpinalHDL types:

```
val values = Vec(Bits(8 bits), 4)

val valuesAre42    = values.map(_ === 42)
val valuesAreAll42 = valuesAre42.reduce(_ && _)

val valuesAreEqualToTheirIndex = values.zipWithIndex.map{ case (value, i) => value␣
↪=== i }
```

4.4.4 Scala guide

Introduction

Scala is a very capable programming language that was influenced by a unique set of languages, but often, this set of languages doesn't cross the ones that most programmers use. That can hinder newcomers' understanding of the concepts and design choices behind Scala.

The following pages will present Scala, and try to provide enough information about it for newcomers to be comfortable with SpinalHDL.

4.5 Help for VHDL people

4.5.1 VHDL comparison

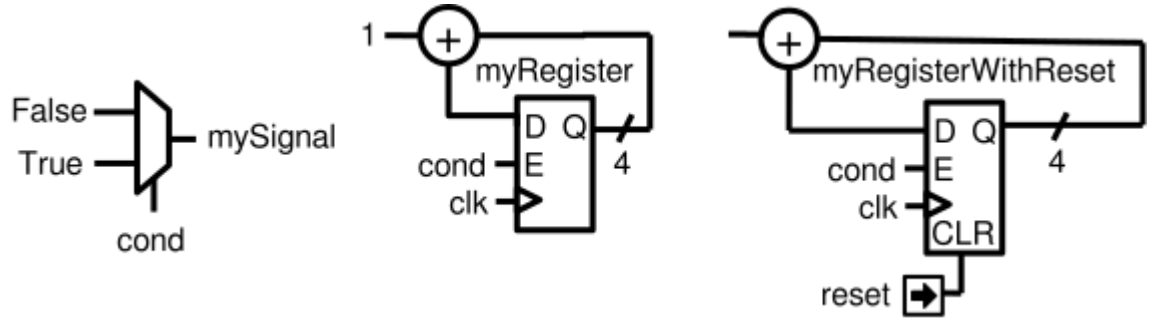
Introduction

This page will show the main differences between VHDL and SpinalHDL. Things will not be explained in depth.

Process

Processes are often needed when you write RTL, however, their semantics can be clunky to work with. Due to how they work in VHDL, they can force you to split your code and duplicate things.

To produce the following RTL:



You will have to write the following VHDL:

```
  signal mySignal : std_logic;
  signal myRegister : std_logic_vector(3 downto 0);
  signal myRegisterWithReset : std_logic_vector(3 downto 0);
begin
  process(cond)
  begin
    mySignal <= '0';
    if cond = '1' then
      mySignal <= '1';
    end if;
  end process;

  process(clk)
  begin
    if rising_edge(clk) then
      if cond = '1' then
        myRegister <= myRegister + 1;
      end if;
    end if;
  end process;

  process(clk,reset)
  begin
    if reset = '1' then
      myRegisterWithReset <= (others => '0');
    elsif rising_edge(clk) then
      if cond = '1' then
        myRegisterWithReset <= myRegisterWithReset + 1;
      end if;
    end if;
  end process;
```

While in SpinalHDL, it's:

```
val mySignal = Bool()
val myRegister = Reg(UInt(4 bits))
val myRegisterWithReset = Reg(UInt(4 bits)) init(0)

mySignal := False
when(cond) {
  mySignal := True
  myRegister := myRegister + 1
  myRegisterWithReset := myRegisterWithReset + 1
}
```

Implicit vs explicit definitions

In VHDL, when you declare a signal, you don't specify if it is a combinatorial signal or a register. Where and how you assign to it decides whether it is combinatorial or registered.

In SpinalHDL these kinds of things are explicit. Registers are defined as registers directly in their declaration.

Clock domains

In VHDL, every time you want to define a bunch of registers, you need the carry the clock and the reset wire to them. In addition, you have to hardcode everywhere how those clock and reset signals should be used (clock edge, reset polarity, reset nature (async, sync)).

In SpinalHDL you can define a `ClockDomain`, and then define the area of your hardware that uses it.

For example:

```
val coreClockDomain = ClockDomain(
  clock = io.coreClk,
  reset = io.coreReset,
  config = ClockDomainConfig(
    clockEdge = RISING,
    resetKind = ASYNC,
    resetActiveLevel = HIGH
  )
)
val coreArea = new ClockingArea(coreClockDomain) {
  val myCoreClockedRegister = Reg(UInt(4 bit))
  // ...
  // coreClockDomain will also be applied to all sub components instantiated in the␣
↪Area
  // ...
}
```

Component's internal organization

In VHDL, there is a `block` feature that allows you to define sub-areas of logic inside your component. However, almost no one uses this feature, because most people don't know about them, and also because all signals defined inside these regions are not readable from the outside.

In SpinalHDL you have an `Area` feature that does this concept much more nicely:

```
val timeout = new Area {
  val counter = Reg(UInt(8 bits)) init(0)
  val overflow = False
  when(counter =/= 100) {
    counter := counter + 1
  } otherwise {
    overflow := True
  }
}

val core = new Area {
  when(timeout.overflow) {
    timeout.counter := 0
  }
}
```

Variables and signals defined inside of an `Area` are accessible elsewhere in the component, including in other `Area` regions.

Safety

In VHDL as in SpinalHDL, it's easy to write combinatorial loops, or to infer a latch by forgetting to drive a signal in the path of a process.

Then, to detect those issues, you can use some `lint` tools that will analyze your VHDL, but those tools aren't free. In SpinalHDL the `lint` process in integrated inside the compiler, and it won't generate the RTL code until everything is fine. It also checks clock domain crossing.

Functions and procedures

Functions and procedures are not used very often in VHDL, probably because they are very limited:

- You can only define a chunk of combinational hardware, or only a chunk of registers (if you call the function/procedure inside a clocked process).
- You can't define a process inside them.
- You can't instantiate a component inside them.
- The scope of what you can read/write inside them is limited.

In SpinalHDL, all those limitations are removed.

An example that mixes combinational logic and a register in a single function:

```
def simpleAluPipeline(op: Bits, a: UInt, b: UInt): UInt = {
  val result = UInt(8 bits)

  switch(op) {
    is(0){ result := a + b }
    is(1){ result := a - b }
    is(2){ result := a * b }
  }

  return RegNext(result)
}
```

An example with the queue function inside the Stream Bundle (handshake). This function instantiates a FIFO component:

```
class Stream[T <: Data](dataType:  T) extends Bundle with IMasterSlave with␣
↪DataCarrier[T] {
  val valid = Bool()
  val ready = Bool()
  val payload = cloneOf(dataType)

  def queue(size: Int): Stream[T] = {
    val fifo = new StreamFifo(dataType, size)
    fifo.io.push <> this
    fifo.io.pop
  }
}
```

An example where a function assigns a signal defined outside of itself:

```
val counter = Reg(UInt(8 bits)) init(0)
counter := counter + 1

def clear() : Unit = {
  counter := 0
}

when(counter > 42) {
  clear()
}
```

Buses and Interfaces

VHDL is very boring when it comes to buses and interfaces. You have two options:

1) Define buses and interfaces wire-by-wire, each time and everywhere:

```
PADDR   : in unsigned(addressWidth-1 downto 0);
PSEL    : in std_logic
PENABLE : in std_logic;
PREADY  : out std_logic;
PWRITE  : in std_logic;
PWDATA  : in std_logic_vector(dataWidth-1 downto 0);
PRDATA  : out std_logic_vector(dataWidth-1 downto 0);
```

2) Use records but lose parameterization (statically fixed in the package), and you have to define one for each directions:

```
P_m : in APB_M;
P_s : out APB_S;
```

SpinalHDL has very strong support for bus and interface declarations with limitless parameterizations:

```
val P = slave(Apb3(addressWidth, dataWidth))
```

You can also use object oriented programming to define configuration objects:

```
val coreConfig = CoreConfig(
  pcWidth = 32,
  addrWidth = 32,
  startAddress = 0x00000000,
  regFileReadyKind = sync,
  branchPrediction = dynamic,
  bypassExecute0 = true,
  bypassExecute1 = true,
  bypassWriteBack = true,
  bypassWriteBackBuffer = true,
  collapseBubble = false,
  fastFetchCmdPcCalculation = true,
  dynamicBranchPredictorCacheSizeLog2 = 7
)

// The CPU has a system of plugins which allows adding new features into the core.
// Those extensions are not directly implemented in the core, but are kind of an␣
↪additive logic patch defined in a separate area.
coreConfig.add(new MulExtension)
coreConfig.add(new DivExtension)
```

(continues on next page)

(continued from previous page)

```
coreConfig.add(new BarrelShifterFullExtension)

val iCacheConfig = InstructionCacheConfig(
  cacheSize = 4096,
  bytePerLine = 32,
  wayCount = 1,  // Can only be one for the moment
  wrappedMemAccess = true,
  addressWidth = 32,
  cpuDataWidth = 32,
  memDataWidth = 32
)

new RiscvCoreAxi4(
  coreConfig = coreConfig,
  iCacheConfig = iCacheConfig,
  dCacheConfig = null,
  debug = debug,
  interruptCount = interruptCount
)
```

Signal declaration

VHDL forces you to define all signals at the top of your architecture description, which is annoying.

```
  ..
  .. (many signal declarations)
  ..
  signal a : std_logic;
  ..
  .. (many signal declarations)
  ..
begin
  ..
  .. (many logic definitions)
  ..
  a <= x & y
  ..
  .. (many logic definitions)
  ..
```

SpinalHDL is flexible when it comes to signal declarations.

```
val a = Bool
a := x & y
```

It also allows you to define and assign signals in a single line.

```
val a = x & y
```

Component instantiation

VHDL is very verbose about this, as you have to redefine all signals of your sub-component entity, and then bind them one-by-one when you instantiate your component.

```
divider_cmd_valid : in std_logic;
divider_cmd_ready : out std_logic;
divider_cmd_numerator : in unsigned(31 downto 0);
divider_cmd_denominator : in unsigned(31 downto 0);
divider_rsp_valid : out std_logic;
divider_rsp_ready : in std_logic;
divider_rsp_quotient : out unsigned(31 downto 0);
divider_rsp_remainder : out unsigned(31 downto 0);

divider : entity work.UnsignedDivider
  port map (
    clk             => clk,
    reset           => reset,
    cmd_valid       => divider_cmd_valid,
    cmd_ready       => divider_cmd_ready,
    cmd_numerator   => divider_cmd_numerator,
    cmd_denominator => divider_cmd_denominator,
    rsp_valid       => divider_rsp_valid,
    rsp_ready       => divider_rsp_ready,
    rsp_quotient    => divider_rsp_quotient,
    rsp_remainder   => divider_rsp_remainder
  );
```

SpinalHDL removes that, and allows you to access the IO of sub-components in an object-oriented way.

```
val divider = new UnsignedDivider()

// And then if you want to access IO signals of that divider:
divider.io.cmd.valid := True
divider.io.cmd.numerator := 42
```

Casting

There are two annoying casting methods in VHDL:

- boolean <> std_logic (ex: To assign a signal using a condition such as `mySignal <= myValue < 10` is not legal)
- unsigned <> integer (ex: To access an array)

SpinalHDL removes these casts by unifying things.

boolean/std_logic:

```
val value = UInt(8 bits)
val valueBiggerThanTwo = Bool
valueBiggerThanTwo := value > 2  // value > 2 return a Bool
```

unsigned/integer:

```
val array = Vec(UInt(4 bits),8)
val sel = UInt(3 bits)
val arraySel = array(sel) // Arrays are indexed directly by using UInt
```

Resizing

The fact that VHDL is strict about bit size is probably a good thing.

```
my8BitsSignal <= resize(my4BitsSignal, 8);
```

In SpinalHDL you have two ways to do the same:

```
// The traditional way
my8BitsSignal := my4BitsSignal.resize(8)

// The smart way
my8BitsSignal := my4BitsSignal.resized
```

Parameterization

VHDL prior to the 2008 revision has many issues with generics. For example, you can't parameterize records, you can't parameterize arrays in the entity, and you can't have type parameters.
Then VHDL 2008 came and fixed those issues. But RTL tool support for VHDL 2008 is really weak depending on the vendor.

SpinalHDL has full support for generics integrated natively in its compiler, and it doesn't rely on VHDL generics.

Here is an example of parameterized data structures:

```
val colorStream = Stream(Color(5, 6, 5)))
val colorFifo   = StreamFifo(Color(5, 6, 5), depth = 128)
colorFifo.io.push <> colorStream
```

Here is an example of a parameterized component:

```
class Arbiter[T <: Data](payloadType: T, portCount: Int) extends Component {
  val io = new Bundle {
    val sources = Vec(slave(Stream(payloadType)), portCount)
    val sink = master(Stream(payloadType))
  }
  // ...
}
```

Meta hardware description

VHDL has kind of a closed syntax. You can't add abstraction layers on top of it.

SpinalHDL, because it's built on top of Scala, is very flexible, and allows you to define new abstraction layers very easily.

Some examples of this flexibility are the *FSM* library, the *BusSlaveFactory* library, and also the *JTAG* library.

4.5.2 VHDL equivalences

Entity and architecture

In SpinalHDL, a VHDL entity and architecture are both defined inside a `Component`.

Here is an example of a component which has 3 inputs (`a`, `b`, `c`) and an output (`result`). This component also has an `offset` construction parameter (like a VHDL generic).

```
case class MyComponent(offset: Int) extends Component {
  val io = new Bundle{
    val a, b, c = in UInt(8 bits)
    val result  = out UInt(8 bits)
  }
  io.result := a + b + c + offset
}
```

Then to instantiate that component, you don't need to bind it:

```
case class TopLevel extends Component {
  ...
  val mySubComponent = MyComponent(offset = 5)

  ...

  mySubComponent.io.a := 1
  mySubComponent.io.b := 2
  mySubComponent.io.c := 3
  ??? := mySubComponent.io.result

  ...
}
```

Data types

SpinalHDL data types are similar to the VHDL ones:

VHDL	SpinalHDL
std_logic	Bool
std_logic_vector	Bits
unsigned	UInt
signed	SInt

In VHDL, to define an 8 bit `unsigned` you have to give the range of bits `unsigned(7 downto 0)`, whereas in SpinalHDL you simply supply the number of bits `UInt(8 bits)`.

VHDL	SpinalHDL
records	Bundle
array	Vec
enum	SpinalEnum

Here is an example of the SpinalHDL `Bundle` definition. `channelWidth` is a construction parameter, like VHDL generics, but for data structures:

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

Then for example, to instantiate a `Bundle`, you need to write `val myColor = RGB(channelWidth=8)`.

Signal

Here is an example about signal instantiations:

```
case class MyComponent(offset: Int) extends Component {
  val io = new Bundle {
    val a, b, c = UInt(8 bits)
    val result  = UInt(8 bits)
  }
  val ab = UInt(8 bits)
  ab := a + b

  val abc = ab + c            // You can define a signal directly with its value
  io.result := abc + offset
}
```

Assignments

In SpinalHDL, the `:=` assignment operator is equivalent to the VHDL signal assignment (<=):

```
val myUInt = UInt(8 bits)
myUInt := 6
```

Conditional assignments are done like in VHDL by using `if`/`case` statements:

```
val clear   = Bool()
val counter = Reg(UInt(8 bits))

when(clear) {
  counter := 0
}.elsewhen(counter === 76) {
  counter := 79
}.otherwise {
  counter(7) := ! counter(7)
}

switch(counter) {
  is(42) {
    counter := 65
  }
  default {
    counter := counter + 1
  }
}
```

Literals

Literals are a little bit different than in VHDL:

```
val myBool = Bool()
myBool := False
myBool := True
myBool := Bool(4 > 7)

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

Registers

In SpinalHDL, registers are explicitly specified while in VHDL registers are inferred. Here is an example of SpinalHDL registers:

```
val counter = Reg(UInt(8 bits))  init(0)
counter := counter + 1   // Count up each cycle

// init(0) means that the register should be initialized to zero when a reset occurs
```

Process blocks

Process blocks are a simulation feature that is unnecessary to design RTL. It's why SpinalHDL doesn't contain any feature analogous to process blocks, and you can assign what you want, where you want.

```
val cond = Bool()
val myCombinatorial = Bool()
val myRegister = UInt(8 bits)

myCombinatorial := False
when(cond) {
  myCombinatorial := True
  myRegister = myRegister + 1
}
```

4.6 Cheatsheets

4.6.1 Core

Redirection to https://github.com/SpinalHDL/SpinalDoc/blob/master/cheatsheet/cheatSheet_core_oo.pdf

4.6.2 Lib

Redirection to https://github.com/SpinalHDL/SpinalDoc/blob/master/cheatsheet/cheatSheet_lib_oo.pdf

4.6.3 Symbolic

Redirection to https://github.com/SpinalHDL/SpinalDoc/blob/master/cheatsheet/cheatSheet_symbolic.pdf

CHAPTER

FIVE

DATA TYPES

5.1 Bool

5.1.1 Description

The `Bool` type corresponds to a boolean value (True or False).

5.1.2 Declaration

The syntax to declare a boolean value is as follows: (everything between [] is optional)

Syntax	Description	Return
Bool[()]	Create a Bool	Bool
True	Create a Bool assigned with `true`	Bool
False	Create a Bool assigned with `false`	Bool
Bool(value: Boolean)	Create a Bool assigned with a Scala Boolean(true, false)	Bool

```
val myBool_1 = Bool()          // Create a Bool
myBool_1 := False             // := is the assignment operator

val myBool_2 = False          // Equivalent to the code above

val myBool_3 = Bool(5 > 12)   // Use a Scala Boolean to create a Bool
```

5.1.3 Operators

The following operators are available for the `Bool` type:

Logic

Operator	Description	Return type
!x	Logical NOT	Bool
x && y x & y	Logical AND	Bool
x || y x | y	Logical OR	Bool
x ^ y	Logical XOR	Bool
x.set[()]	Set x to True	
x.clear[()]	Set x to False	
x.setWhen(cond)	Set x when cond is True	Bool
x.clearWhen(cond)	Clear x when cond is True	Bool
x.riseWhen(cond)	Set x when x is False and cond is True	Bool
x.fallWhen(cond)	Clear x when x is True and cond is True	Bool

```
val a, b, c = Bool()
val res = (!a & b) ^ c   // ((NOT a) AND b) XOR c

val d = False
when(cond) {
  d.set()    // equivalent to d := True
}

val e = False
e.setWhen(cond) // equivalent to when(cond) { d := True }

val f = RegInit(False) fallWhen(ack) setWhen(req)
/** equivalent to
  * when(f && ack) { f := False }
  * when(req) { f := True }
  * or
  * f := req || (f && !ack)
  */

// mind the order of assignments!
val g = RegInit(False) setWhen(req) fallWhen(ack)
// equivalent to g := ((!g) && req) || (g && !ack)
```

Edge detection

Operator	Description	Return type
x.edge[()]	Return True when x changes state	Bool
x.edge(initAt: Bool)	Same as x.edge but with a reset value	Bool
x.rise[()]	Return True when x was low at the last cycle and is now high	Bool
x.rise(initAt: Bool)	Same as x.rise but with a reset value	Bool
x.fall[()]	Return True when x was high at the last cycle and is now low	Bool
x.fall(initAt: Bool)	Same as x.fall but with a reset value	Bool
x.edges[()]	Return a bundle (rise, fall, toggle)	BoolEdges
x.edges(initAt: Bool)	Same as x.edges but with a reset value	BoolEdges

```
when(myBool_1.rise(False)) {
    // do something when a rising edge is detected
}


val edgeBundle = myBool_2.edges(False)
when(edgeBundle.rise) {
    // do something when a rising edge is detected
}
when(edgeBundle.fall) {
    // do something when a falling edge is detected
}
when(edgeBundle.toggle) {
    // do something at each edge
}
```

Comparison

Operator	Description	Return type
x === y	Equality	Bool
x =/= y	Inequality	Bool

```
when(myBool) { // Equivalent to when(myBool === True)
    // do something when myBool is True
}

when(!myBool) { // Equivalent to when(myBool === False)
    // do something when myBool is False
}
```

Type cast

Operator	Description	Return
x.asBits	Binary cast to Bits	Bits(w(x) bits)
x.asUInt	Binary cast to UInt	UInt(w(x) bits)
x.asSInt	Binary cast to SInt	SInt(w(x) bits)
x.asUInt(bitCount)	Binary cast to UInt and resize	UInt(bitCount bits)
x.asBits(bitCount)	Binary cast to Bits and resize	Bits(bitCount bits)

```
// Add the carry to an SInt value
val carry = Bool()
val res = mySInt + carry.asSInt
```

Misc

Operator	Description	Return
x ## y	Concatenate, x->high, y->low	Bits(w(x) + w(y) bits)

```
val a, b, c = Bool

// Concatenation of three Bool into a Bits
val myBits = a ## b ## c
```

5.2 Bits

5.2.1 Description

The `Bits` type corresponds to a vector of bits that does not convey any arithmetic meaning.

5.2.2 Declaration

The syntax to declare a bit vector is as follows: (everything between [] is optional)

Syntax	Description	Return
Bits [()]	Create a BitVector, bits count is inferred	Bits
Bits(x bits)	Create a BitVector with x bits	Bits
B(value: Int[, x bits]) B(value: BigInt[, x bits])	Create a BitVector with x bits assigned with 'value'	Bits
B"[[size']base]value"	Create a BitVector assigned with 'value' (Base: 'h', 'd', 'o', 'b')	Bits
B([x bits,] element, ...)	Create a BitVector assigned with the value specified by elements	Bits

```
// Declaration
val myBits  = Bits()     // the size is inferred
val myBits1 = Bits(32 bits)
val myBits2 = B(25, 8 bits)
val myBits3 = B"8'xFF"   // Base could be x,h (base 16)
```

(continues on next page)

(continued from previous page)

```
                        //               d   (base 10)
                        //               o   (base 8)
                        //               b   (base 2)
val myBits4 = B"1001_0011"  // _ can be used for readability

// Element
val myBits5 = B(8 bits, default -> True) // "11111111"
val myBits6 = B(8 bits, (7 downto 5) -> B"101", 4 -> true, 3 -> True, default ->␣
↪false) // "10111000"
val myBits7 = Bits(8 bits)
myBits7 := (7 -> true, default -> false) // "10000000" (For assignment purposes, you␣
↪can omit the B)
```

5.2.3 Operators

The following operators are available for the `Bits` type:

Logic

Operator	Description	Return type
~x	Bitwise NOT	Bits(w(x) bits)
x & y	Bitwise AND	Bits(w(xy) bits)
x | y	Bitwise OR	Bits(w(xy) bits)
x ^ y	Bitwise XOR	Bits(w(xy) bits)
x.xorR	XOR all bits of x	Bool
x.orR	OR all bits of x	Bool
x.andR	AND all bits of x	Bool
x >> y	Logical shift right, y: Int	Bits(w(x) - y bits)
x >> y	Logical shift right, y: UInt	Bits(w(x) bits)
x << y	Logical shift left, y: Int	Bits(w(x) + y bits)
x << y	Logical shift left, y: UInt	Bits(w(x) + max(y) bits)
x |>> y	Logical shift right, y: Int/UInt	Bits(w(x) bits)
x |<< y	Logical shift left, y: Int/UInt	Bits(w(x) bits)
x.rotateLeft(y)	Logical left rotation, y: UInt/Int	Bits(w(x) bits)
x.rotateRight(y)	Logical right rotation, y: UInt/Int	Bits(w(x) bits)
x.clearAll[()]	Clear all bits	
x.setAll[()]	Set all bits	
x.setAllTo(value: Boolean)	Set all bits to the given Boolean value	
x.setAllTo(value: Bool)	Set all bits to the given Bool value	

```
// Bitwise operator
val a, b, c = Bits(32 bits)
c := ~(a & b) // Inverse(a AND b)

val all_1 = a.andR // Check that all bits are equal to 1

// Logical shift
val bits_10bits = bits_8bits << 2  // shift left (results in 10 bits)
val shift_8bits = bits_8bits |<< 2 // shift left (results in 8 bits)

// Logical rotation
val myBits = bits_8bits.rotateLeft(3) // left bit rotation
```

(continues on next page)

(continued from previous page)

```
// Set/clear
val a = B"8'x42"
when(cond) {
  a.setAll() // set all bits to True when cond is True
}
```

Comparison

Operator	Description	Return type
x === y	Equality	Bool
x =/= y	Inequality	Bool

```
when(myBits === 3) {
}

when(myBits_32 =/= B"32'x44332211") {
}
```

Type cast

Operator	Description	Return
x.asBits	Binary cast to Bits	Bits(w(x) bits)
x.asUInt	Binary cast to UInt	UInt(w(x) bits)
x.asSInt	Binary cast to SInt	SInt(w(x) bits)
x.asBools	Cast to an array of Bools	Vec(Bool, w(x))
B(x: T)	Cast Data to Bits	Bits(w(x) bits)

To cast a `Bool`, `UInt` or an `SInt` into a `Bits`, you can use `B(something)`:

```
// cast a Bits to SInt
val mySInt = myBits.asSInt

// create a Vector of bool
val myVec = myBits.asBools

// Cast a SInt to Bits
val myBits = B(mySInt)
```

Bit extraction

Operator	Description	Return
x(y)	Readbit, y: Int/UInt	Bool
x(offset,width bits)	Read bitfield, offset: UInt, width: Int	Bits(width bits)
x(range)	Read a range of bit. Ex : myBits(4 downto 2)	Bits(range bits)
x(y) := z	Assign bits, y: Int/UInt	Bool
x(offset, width bits) := z	Assign bitfield, offset: UInt, width: Int	Bits(width bits)
x(range) := z	Assign a range of bit. Ex : myBits(4 downto 2) := B”010”	Bits(range bits)

```
// get the element at the index 4
val myBool = myBits(4)

// assign
myBits(1) := True

// Range
val myBits_8bits = myBits_16bits(7 downto 0)
val myBits_7bits = myBits_16bits(0 to 6)
val myBits_6bits = myBits_16Bits(0 until 6)

myBits_8bits(3 downto 0) := myBits_4bits
```

Misc

Operator	Description	Return
x.getWidth	Return bitcount	Int
x.range	Return the range (x.high downto 0)	Range
x.high	Return the upper bound of the type x	Int
x.msb	Return the most significant bit	Bool
x.lsb	Return the least significant bit	Bool
x ## y	Concatenate, x->high, y->low	Bits(w(x) + w(y) bits)
x.subdivideIn(y slices)	Subdivide x in y slices, y: Int	Vec(Bits, y)
x.subdivideIn(y bits)	Subdivide x in multiple slices of y bits, y: Int	Vec(Bits, w(x)/y)
x.resize(y)	Return a resized copy of x, if enlarged, it is filled with zero, y: Int	Bits(y bits)
x.resized	Return a version of x which is allowed to be automatically resized were needed	Bits(w(x) bits)
x.resizeLeft(x)	Resize by keeping MSB at the same place, x:Int	Bits(x bits)

```
println(myBits_32bits.getWidth) // 32

myBool := myBits.lsb  // Equivalent to myBits(0)

// Concatenation
myBits_24bits := bits_8bits_1 ## bits_8bits_2 ## bits_8bits_3

// Subdivide
val sel = UInt(2 bits)
val myBitsWord = myBits_128bits.subdivideIn(32 bits)(sel)
    // sel = 0 => myBitsWord = myBits_128bits(127 downto 96)
    // sel = 1 => myBitsWord = myBits_128bits( 95 downto 64)
    // sel = 2 => myBitsWord = myBits_128bits( 63 downto 32)
    // sel = 3 => myBitsWord = myBits_128bits( 31 downto  0)

 // If you want to access in reverse order you can do:
 val myVector   = myBits_128bits.subdivideIn(32 bits).reverse
 val myBitsWord = myVector(sel)

// Resize
myBits_32bits := B"32'x112233344"
myBits_8bits  := myBits_32bits.resized       // automatic resize (myBits_8bits = 0x44)
myBits_8bits  := myBits_32bits.resize(8)     // resize to 8 bits (myBits_8bits = 0x44)
myBits_8bits  := myBits_32bits.resizeLeft(8) // resize to 8 bits (myBits_8bits = 0x11)
```

5.3 UInt/SInt

5.3.1 Description

The `UInt/SInt` type corresponds to a vector of bits that can be used for signed/unsigned integer arithmetic.

5.3.2 Declaration

The syntax to declare an integer is as follows: (everything between [] is optional)

Syntax	Description	Return
UInt[()] SInt[()]	Create an unsigned/signed integer, bits count is inferred	UInt SInt
UInt(x bits) SInt(x bits)	Create an unsigned/signed integer with x bits	UInt SInt
U(value: Int[,x bits]) U(value: BigInt[,x bits]) S(value: Int[,x bits]) S(value: BigInt[,x bits])	Create an unsigned/signed integer assigned with 'value'	UInt SInt
U"[[size']base]value" S"[[size']base]value"	Create an unsigned/signed integer assigned with 'value' (Base : 'h', 'd', 'o', 'b')	UInt SInt
U([x bits,] element, ...) S([x bits,] element, ...)	Create an unsigned integer assigned with the value specified by elements	UInt SInt

```
val myUInt = UInt(8 bits)
myUInt := U(2,8 bits)
myUInt := U(2)
myUInt := U"0000_0101"  // Base per default is binary => 5
myUInt := U"h1A"        // Base could be x (base 16)
                        //               h (base 16)
                        //               d (base 10)
                        //               o (base 8)
                        //               b (base 2)
myUInt := U"8'h1A"
myUInt := 2             // You can use a Scala Int as a literal value

val myBool := myUInt === U(7 -> true,(6 downto 0) -> false)
val myBool := myUInt === U(myUInt.range -> true)

// For assignment purposes, you can omit the U/S, which also allows the use of the␣
↪[default -> ???] feature
```

(continues on next page)

(continued from previous page)

```
myUInt := (default -> true)                          // Assign myUInt with "11111111"
myUInt := (myUInt.range -> true)                     // Assign myUInt with "11111111"
myUInt := (7 -> true, default -> false)              // Assign myUInt with "10000000"
myUInt := ((4 downto 1) -> true, default -> false) // Assign myUInt with "00011110"
```

5.3.3 Operators

The following operators are available for the `UInt` and `SInt` types:

Logic

Operator	Description	Return type
x ^ y	Logical XOR	Bool
~x	Bitwise NOT	T(w(x) bits)
x & y	Bitwise AND	T(max(w(xy) bits)
x | y	Bitwise OR	T(max(w(xy) bits)
x ^ y	Bitwise XOR	T(max(w(xy) bits)
x.xorR	XOR all bits of x	Bool
x.orR	OR all bits of x	Bool
x.andR	AND all bits of x	Bool
x >> y	Arithmetic shift right, y : Int	T(w(x) - y bits)
x >> y	Arithmetic shift right, y : UInt	T(w(x) bits)
x << y	Arithmetic shift left, y : Int	T(w(x) + y bits)
x << y	Arithmetic shift left, y : UInt	T(w(x) + max(y) bits)
x |>> y	Logical shift right, y : Int/UInt	T(w(x) bits)
x |<< y	Logical shift left, y : Int/UInt	T(w(x) bits)
x.rotateLeft(y)	Logical left rotation, y : UInt/Int	T(w(x) bits)
x.rotateRight(y)	Logical right rotation, y : UInt/Int	T(w(x) bits)
x.clearAll[()]	Clear all bits	
x.setAll[()]	Set all bits	
x.setAllTo(value : Boolean)	Set all bits to the given Boolean value	
x.setAllTo(value : Bool)	Set all bits to the given Bool value	

```
// Bitwise operator
val a, b, c = SInt(32 bits)
c := ~(a & b) // Inverse(a AND b)

val all_1 = a.andR // Check that all bits are equal to 1

// Logical shift
val uint_10bits = uint_8bits << 2  // shift left (resulting in 10 bits)
val shift_8bits = uint_8bits |<< 2 // shift left (resulting in 8 bits)

// Logical rotation
val myBits = uint_8bits.rotateLeft(3) // left bit rotation

// Set/clear
val a = B"8'x42"
when(cond) {
  a.setAll() // set all bits to True when cond is True
}
```

Arithmetic

Operator	Description	Return
x + y	Addition	T(max(w(x), w(y)), bits)
x +^ y	Addition with carry	T(max(w(x), w(y) + 1), bits)
x +| y	Addition by sat carry bit	T(max(w(x), w(y)), bits)
x - y	Subtraction	T(max(w(x), w(y)), bits)
x -^ y	Subtraction with carry	T(max(w(x), w(y) + 1), bits)
x -| y	Subtraction by sat carry bit	T(max(w(x), w(y)), bits)
x * y	Multiplication	T(w(x) + w(y)), bits)
x / y	Division	T(w(x), bits)
x % y	Modulo	T(w(x), bits)

```
// Addition
val res = mySInt_1 + mySInt_2
```

Comparison

Operator	Description	Return type
x === y	Equality	Bool
x =/= y	Inequality	Bool
x > y	Greater than	Bool
x >= y	Greater than or equal	Bool
x < y	Less than	Bool
x <= y	Less than or equal	Bool

```
// Comparison between two SInts
myBool := mySInt_1 > mySInt_2

// Comparison between a UInt and a literal
myBool := myUInt_8bits >= U(3, 8 bits)

when(myUInt_8bits === 3) {
  ..
}
```

Type cast

Operator	Description	Return
x.asBits	Binary cast to Bits	Bits(w(x), bits)
x.asUInt	Binary cast to UInt	UInt(w(x), bits)
x.asSInt	Binary cast to SInt	SInt(w(x), bits)
x.asBools	Cast into a array of Bool	Vec(Bool, w(x))
S(x: T)	Cast a Data into a SInt	SInt(w(x), bits)
U(x: T)	Cast a Data into an UInt	UInt(w(x), bits)
x.intoSInt	convert to SInt expand signbit	SInt(w(x) + 1, bits)

To cast a `Bool`, a `Bits`, or an `SInt` into a `UInt`, you can use `U(something)`. To cast things into an `SInt`, you can use `S(something)`.

```
// Cast an SInt to Bits
val myBits = mySInt.asBits

// Create a Vector of Bool
val myVec = myUInt.asBools

// Cast a Bits to SInt
val mySInt = S(myBits)
```

Bit extraction

Operator	Description	Return
x(y)	Readbit, y : Int/UInt	Bool
x(offset, width)	Read bitfield, offset: UInt, width: Int	T(width bits)
x(range)	Read a range of bits. Ex : myBits(4 downto 2)	T(range bits)
x(y) := z	Assign bits, y : Int/UInt	Bool
x(offset, width) := z	Assign bitfield, offset: UInt, width: Int	T(width bits)
x(range) := z	Assign a range of bit. Ex : myBits(4 downto 2) := U"010"	T(range bits)

```
// get the bit at index 4
val myBool = myUInt(4)

// assign bit 1 to True
mySInt(1) := True

// Range
val myUInt_8bits = myUInt_16bits(7 downto 0)
val myUInt_7bits = myUInt_16bits(0 to 6)
val myUInt_6bits = myUInt_16Bits(0 until 6)

mySInt_8bits(3 downto 0) := mySInt_4bits
```

Misc

Operator	Description	Return
x.getWidth	Return bitcount	Int
x.msb	Return the most significant bit	Bool
x.lsb	Return the least significant bit	Bool
x.range	Return the range (x.high downto 0)	Range
x.high	Return the upper bound of the type x	Int
x ## y	Concatenate, x->high, y->low	Bits(w(x) + w(y) bits)
x @@ y	Concatenate x:T with y:Bool/SInt/UInt	T(w(x) + w(y) bits)
x.subdivideIn(y slices)	Subdivide x into y slices, y: Int	Vec(T, y)
x.subdivideIn(y bits)	Subdivide x into multiple slices of y bits, y: Int	Vec(T, w(x)/y)
x.resize(y)	Return a resized copy of x, if enlarged, it is filled with zero for UInt or filled with the sign for SInt, y: Int	T(y bits)
x.resized	Return a version of x which is allowed to be automatically resized where needed	T(w(x) bits)
myUInt.twoComplement(en: Bool)	Use the two's complement to transform an UInt into an SInt	SInt(w(myUInt) + 1, bits)
mySInt.abs	Return the absolute value of the UInt value	UInt(w(mySInt), bits)
mySInt.abs(en: Bool)	Return the absolute value of the UInt value when en is True	UInt(w(mySInt), bits)
mySInt.sign	Return most significant bit	Bool
x.expand	Return x with 1 bit expand	T(w(x)+1 bit)
mySInt.absWithSym	Return the absolute value of the UInt value with symmetric, shrink 1 bit	UInt(w(mySInt) - 1, bits)

```
myBool := mySInt.lsb  // equivalent to mySInt(0)

// Concatenation
val mySInt = mySInt_1 @@ mySInt_1 @@ myBool
val myBits = mySInt_1 ## mySInt_1 ## myBool

// Subdivide
val sel = UInt(2 bits)
val mySIntWord = mySInt_128bits.subdivideIn(32 bits)(sel)
    // sel = 0 => mySIntWord = mySInt_128bits(127 downto 96)
    // sel = 1 => mySIntWord = mySInt_128bits( 95 downto 64)
    // sel = 2 => mySIntWord = mySInt_128bits( 63 downto 32)
    // sel = 3 => mySIntWord = mySInt_128bits( 31 downto  0)

 // If you want to access in reverse order you can do:
 val myVector   = mySInt_128bits.subdivideIn(32 bits).reverse
 val mySIntWord = myVector(sel)
```

(continues on next page)

(continued from previous page)

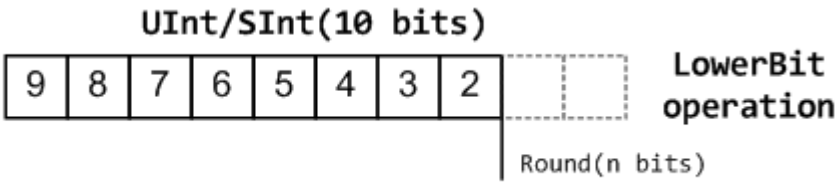
```
// Resize
myUInt_32bits := U"32'x112233344"
myUInt_8bits  := myUInt_32bits.resized       // automatic resize (myUInt_8bits = 0x44)
myUInt_8bits  := myUInt_32bits.resize(8)     // resize to 8 bits (myUInt_8bits = 0x44)

// Two's complement
mySInt := myUInt.twoComplement(myBool)

// Absolute value
mySInt_abs := mySInt.abs
```

5.3.4 FixPoint operations

For fixpoint, we can divide it into two parts:

- Lower bit operations (rounding methods)
- High bit operations (saturation operations)

Lower bit operations



About Rounding: https://en.wikipedia.org/wiki/Rounding

SpinalHDL-Name	Wikipedia-Name	API	Mathematic Algorithm	return(align=false)	Supported
FLOOR	RoundDown	floor	floor(x)	w(x)-n bits	Yes
FLOOR-TOZERO	RoundToZero	floorToZero	sign*floor(abs(x))	w(x)-n bits	Yes
CEIL	RoundUp	ceil	ceil(x)	w(x)-n+1 bits	Yes
CEILTOINF	RoundToInf	ceilToInf	sign*ceil(abs(x))	w(x)-n+1 bits	Yes
ROUNDUP	RoundHalfUp	roundUp	floor(x+0.5)	w(x)-n+1 bits	Yes
ROUNDDOWN	RoundHalfDown	roundDown	ceil(x-0.5)	w(x)-n+1 bits	Yes
ROUNDTOZERO	RoundHalfToZero	roundToZero	sign*ceil(abs(x)-0.5)	w(x)-n+1 bits	Yes
ROUNDTOINF	RoundHalfToInf	roundToInf	sign*floor(abs(x)+0.5)	w(x)-n+1 bits	Yes
ROUNDTOEVEN	RoundHalfToEven	roundToEven			No
ROUNDTOODD	RoundHalfToOdd	roundToOdd			No

Note: The **RoundToEven** and **RoundToOdd** modes are very special, and are used in some big data statistical fields with high accuracy concerns, SpinalHDL doesn't support them yet.

You will find *ROUNDUP*, *ROUNDDOWN*, *ROUNDTOZERO*, *ROUNDTOINF*, *ROUNDTOEVEN*, *ROUNTOODD* are very close in behavior, *ROUNDTOINF* is the most common. The behavior of rounding in different programming languages may be different.

Programming language	default-RoundType	Example	comments
Matlab	ROUNDTOINF	round(1.5)=2,round(2.5)=3;round(-1.5)=-2,round(-2.5)=-3	round to ±Infinity
python2	ROUNDTOINF	round(1.5)=2,round(2.5)=3;round(-1.5)=-2,round(-2.5)=-3	round to ±Infinity
python3	ROUNDTO-EVEN	round(1.5)=round(2.5)=2; round(-1.5)=round(-2.5)=-2	close to Even
Scala.math	ROUNDTOUP	round(1.5)=2,round(2.5)=3;round(-1.5)=-1,round(-2.5)=-2	always to +Infinity
SpinalHDL	ROUNDTOINF	round(1.5)=2,round(2.5)=3;round(-1.5)=-2,round(-2.5)=-3	round to ±Infinity

Note: In SpinalHDL *ROUNDTOINF* is the default RoundType (`round = roundToInf`)

```
val A  = SInt(16 bit)
val B  = A.roundToInf(6 bits) // default 'align = false' with carry, got 11 bit
val B  = A.roundToInf(6 bits, align = true) // sat 1 carry bit, got 10 bit
val B  = A.floor(6 bits)              // return 10 bit
val B  = A.floorToZero(6 bits)        // return 10 bit
val B  = A.ceil(6 bits)               // ceil with carry so return 11 bit
val B  = A.ceil(6 bits, align = true) // ceil with carry then sat 1 bit return 10 bit
val B  = A.ceilToInf(6 bits)
val B  = A.roundUp(6 bits)
val B  = A.roundDown(6 bits)
val B  = A.roundToInf(6 bits)
val B  = A.roundToZero(6 bits)
val B  = A.round(6 bits)              // SpinalHDL uses roundToInf as the default␣
↪rounding mode

val B0 = A.roundToInf(6 bits, align = true)          //  ---+
                                                     //     |--> equal
val B1 = A.roundToInf(6 bits, align = false).sat(1) //  ---+
```

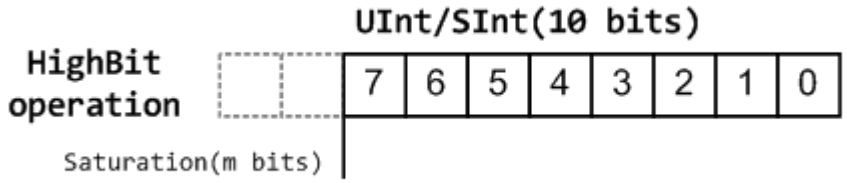
Note: Only `floor` and `floorToZero` work without the `align` option; they do not need a carry bit. Other rounding operations default to using a carry bit.

round Api

API	UInt/SInt	description	Return(align=false)	Return(align=true)
floor	Both		w(x)-n bits	w(x)-n bits
floorToZero	SInt	equal to floor in UInt	w(x)-n bits	w(x)-n bits
ceil	Both		w(x)-n+1 bits	w(x)-n bits
ceilToInf	SInt	equal to ceil in UInt	w(x)-n+1 bits	w(x)-n bits
roundUp	Both	simple for HW	w(x)-n+1 bits	w(x)-n bits
roundDown	Both		w(x)-n+1 bits	w(x)-n bits
roundToInf	SInt	most Common	w(x)-n+1 bits	w(x)-n bits
roundToZero	SInt	equal to roundDown in UInt	w(x)-n+1 bits	w(x)-n bits
round	Both	SpinalHDL chose roundToInf	w(x)-n+1 bits	w(x)-n bits

Note: Although `roundToInf` is very common, `roundUp` has the least cost and good timing, with almost no performance loss. As a result, `roundUp` is strongly recommended for production use.

High bit operations



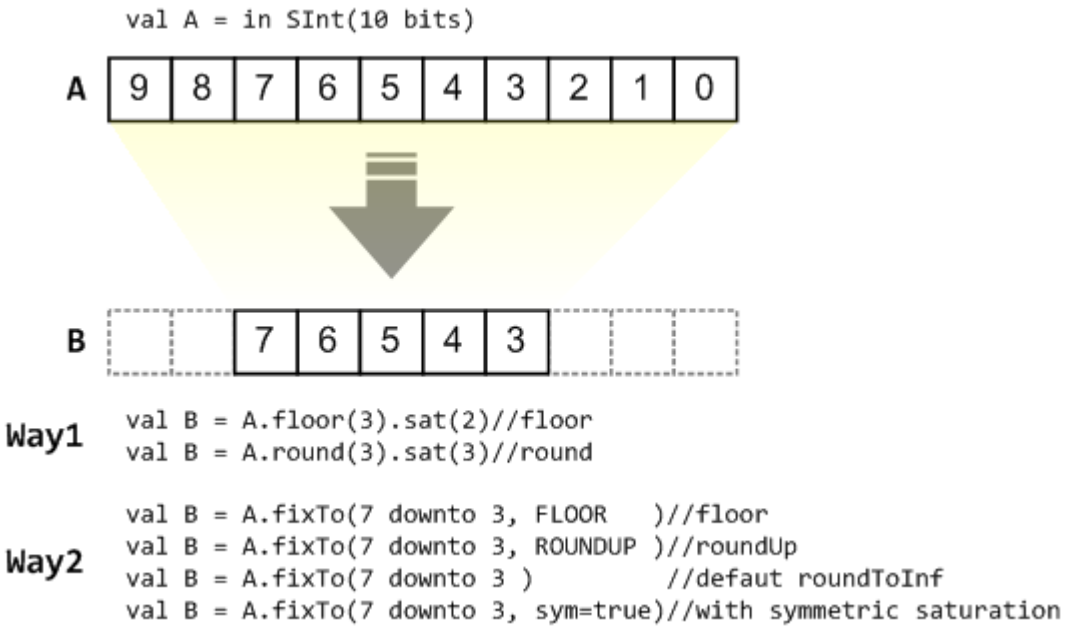
function	Operation	Positive-Op	Negative-Op
sat	Saturation	when(Top[w-1, w-n].orR) set maxValue	When(Top[w-1, w-n].andR) set minValue
trim	Discard	N/A	N/A
symmetry	Symmetric	N/A	minValue = -maxValue

Symmetric is only valid for `SInt`.

```
val A   = SInt(8 bit)
val B   = A.sat(3 bits)       // return 5 bits with saturated highest 3 bits
val B   = A.sat(3)            // equal to sat(3 bits)
val B   = A.trim(3 bits)      // return 5 bits with the highest 3 bits discarded
val B   = A.trim(3 bits)      // return 5 bits with the highest 3 bits discarded
val C   = A.symmetry          // return 8 bits and symmetry as (-128~127 to -127~127)
val C   = A.sat(3).symmetry   // return 5 bits and symmetry as (-16~15 to -15~15)
```

fixTo function

Two ways are provided in `UInt/SInt` to do fixpoint:



`fixTo` is strongly recommended in your RTL work, you don't need to handle carry bit alignment and bit width calculations manually like **Way1** in the above diagram.

Factory Fix function with Auto Saturation:

Function	Description	Return
fixTo(section, roundType, symmetric)	Factory FixFunction	section.size bits

```
val A  = SInt(16 bit)
val B  = A.fixTo(10 downto 3) // default RoundType.ROUNDTOINF, sym = false
val B  = A.fixTo( 8 downto 0, RoundType.ROUNDUP)
val B  = A.fixTo( 9 downto 3, RoundType.CEIL,       sym = false)
val B  = A.fixTo(16 downto 1, RoundType.ROUNDTOINF, sym = true )
val B  = A.fixTo(10 downto 3, RoundType.FLOOR) // floor 3 bit, sat 5 bit @ highest
val B  = A.fixTo(20 downto 3, RoundType.FLOOR) // floor 3 bit, expand 2 bit @ highest
```

5.4 SpinalEnum

5.4.1 Description

The `Enumeration` type corresponds to a list of named values.

5.4.2 Declaration

The declaration of an enumerated data type is as follows:

```
object Enumeration extends SpinalEnum {
  val element0, element1, ..., elementN = newElement()
}
```

For the example above, the default encoding is used. The native enumeration type is used for VHDL and a binary encoding is used for Verilog.

The enumeration encoding can be forced by defining the enumeration as follows:

```
object Enumeration extends SpinalEnum(defaultEncoding=encodingOfYourChoice) {
  val element0, element1, ..., elementN = newElement()
}
```

Note: If you want to define an enumeration as in/out for a given component, you have to do as following: `in(MyEnum())` or `out(MyEnum())`

Encoding

The following enumeration encodings are supported:

Encoding	Bit width	Description
native		Use the VHDL enumeration system, this is the default encoding
binarySequential	log2Up(stateCount)	Use Bits to store states in declaration order (value from 0 to n-1)
binaryOneHot	stateCount	Use Bits to store state. Each bit corresponds to one state

Custom encodings can be performed in two different ways: static or dynamic.

```
/*
 * Static encoding
 */
object MyEnumStatic extends SpinalEnum {
  val e0, e1, e2, e3 = newElement()
  defaultEncoding = SpinalEnumEncoding("staticEncoding")(
    e0 -> 0,
    e1 -> 2,
    e2 -> 3,
    e3 -> 7)
}

/*
 * Dynamic encoding with the function :  _ * 2 + 1
 *   e.g. : e0 => 0 * 2 + 1 = 1
 *          e1 => 1 * 2 + 1 = 3
 *          e2 => 2 * 2 + 1 = 5
 *          e3 => 3 * 2 + 1 = 7
 */
val encoding = SpinalEnumEncoding("dynamicEncoding", _ * 2 + 1)
```

(continues on next page)

(continued from previous page)

```
object MyEnumDynamic extends SpinalEnum(encoding) {
  val e0, e1, e2, e3 = newElement()
}
```

Example

Instantiate an enumerated signal and assign a value to it:

```
object UartCtrlTxState extends SpinalEnum {
  val sIdle, sStart, sData, sParity, sStop = newElement()
}

val stateNext = UartCtrlTxState()
stateNext := UartCtrlTxState.sIdle

// You can also import the enumeration to have visibility of its elements
import UartCtrlTxState._
stateNext := sIdle
```

5.4.3 Operators

The following operators are available for the `Enumeration` type:

Comparison

Operator	Description	Return type
x === y	Equality	Bool
x =/= y	Inequality	Bool

```
import UartCtrlTxState._

val stateNext = UartCtrlTxState()
stateNext := sIdle

when(stateNext === sStart) {
  ...
}

switch(stateNext) {
  is(sIdle) {
    ...
  }
  is(sStart) {
    ...
  }
  ...
}
```

Type cast

Operator	Description	Return
x.asBits	Binary cast to Bits	Bits(w(x) bits)
x.asUInt	Binary cast to UInt	UInt(w(x) bits)
x.asSInt	Binary cast to SInt	SInt(w(x) bits)

```
import UartCtrlTxState._

val stateNext = UartCtrlTxState()
myBits := sIdle.asBits
```

5.5 Bundle

5.5.1 Description

The `Bundle` is a composite type that defines a group of named signals (of any SpinalHDL basic type) under a single name.

A `Bundle` can be used to model data structures, buses, and interfaces.

5.5.2 Declaration

The syntax to declare a bundle is as follows:

```
case class myBundle extends Bundle {
  val bundleItem0 = AnyType
  val bundleItem1 = AnyType
  val bundleItemN = AnyType
}
```

For example, a bundle holding a color could be defined as:

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

You can find an *APB3 definition* among the *Spinal HDL examples*.

5.5.3 Operators

The following operators are available for the `Bundle` type:

Comparison

Operator	Description	Return type
x === y	Equality	Bool
x =/= y	Inequality	Bool

```
val color1 = Color(8)
color1.r := 0
color1.g := 0
color1.b := 0

val color2 = Color(8)
color2.r := 0
color2.g := 0
color2.b := 0

myBool := color1 === color2
```

Type cast

Operator	Description	Return
x.asBits	Binary cast to Bits	Bits(w(x) bits)

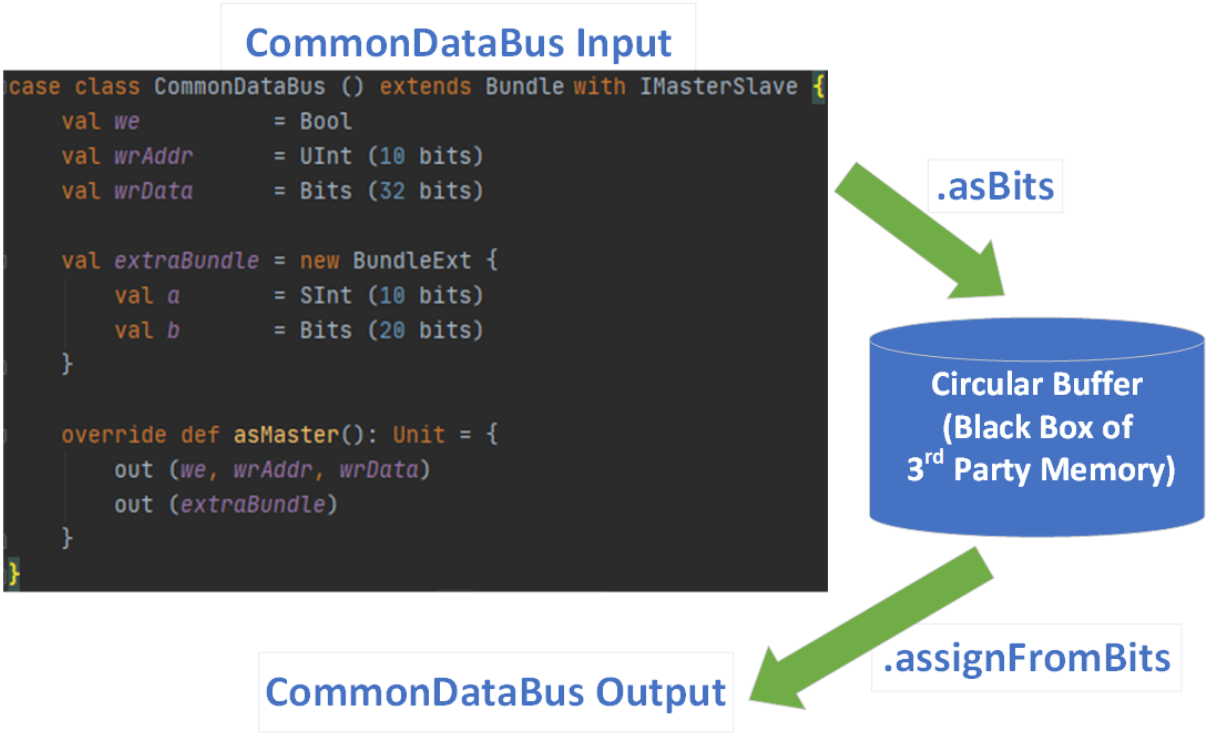
```
val color1 = Color(8)
val myBits := color1.asBits
```

Convert Bits back to Bundle

The `.assignFromBits` operator can be viewed as the reverse of `.asBits`.

Operator	Description	Return
x.assignFromBits(y)	Convert Bits (y) to Bundle(x)	Unit
x.assignFromBits(y, hi, lo)	Convert Bits (y) to Bundle(x) with high/low boundary	Unit

The following example saves a Bundle called CommonDataBus into a circular buffer (3rd party memory), reads the Bits out later and converts them back to CommonDataBus format.




```
case class TestBundle () extends Component {
  val io = new Bundle {
    val we      = in     Bool()
    val addrWr  = in     UInt (7 bits)
    val dataIn  = slave  (CommonDataBus())

    val addrRd  = in     UInt (7 bits)
    val dataOut = master (CommonDataBus())
  }

  val mm = Ram3rdParty_1w_1rs (G_DATA_WIDTH = io.dataIn.getBitsWidth,
                               G_ADDR_WIDTH = io.addrWr.getBitsWidth,
                               G_VENDOR     = "Intel_Arria10_M20K")

  mm.io.clk_in    := clockDomain.readClockWire
  mm.io.clk_out   := clockDomain.readClockWire

  mm.io.we        := io.we
  mm.io.addr_wr   := io.addrWr.asBits
  mm.io.d         := io.dataIn.asBits

  mm.io.addr_rd   := io.addrRd.asBits
  io.dataOut.assignFromBits(mm.io.q)
}
```

5.5.4 IO Element direction

When you define a `Bundle` inside the IO definition of your component, you need to specify its direction.

in/out

If all elements of your bundle go in the same direction you can use `in(MyBundle())` or `out(MyBundle())`.

For example:

```
val io = new Bundle {
  val input  = in (Color(8))
  val output = out(Color(8))
}
```

master/slave

If your interface obeys to a master/slave topology, you can use the `IMasterSlave` trait. Then you have to implement the function `def asMaster(): Unit` to set the direction of each element from the master's perspective. Then you can use the `master(MyBundle())` and `slave(MyBundle())` syntax in the IO definition.

For example:

```
case class HandShake(payloadWidth: Int) extends Bundle with IMasterSlave {
  val valid   = Bool()
  val ready   = Bool()
  val payload = Bits(payloadWidth bits)

  // You have to implement this asMaster function.
  // This function should set the direction of each signals from an master point of␣
↪view
  override def asMaster(): Unit = {
    out(valid, payload)
    in(ready)
  }
}

val io = new Bundle {
  val input  = slave(HandShake(8))
  val output = master(HandShake(8))
}
```

5.6 Vec

5.6.1 Description

A `Vec` is a composite type that defines a group of indexed signals (of any SpinalHDL basic type) under a single name.

5.6.2 Declaration

The syntax to declare a vector is as follows:

Declaration	Description
Vec(type: Data, size: Int)	Create a vector capable of holding `size` elements of type `Data`
Vec(x, y, ...)	Create a vector where indexes point to the provided elements. This constructor supports mixed element width.

Examples

```
// Create a vector of 2 signed integers
val myVecOfSInt = Vec(SInt(8 bits), 2)
myVecOfSInt(0) := 2
myVecOfSInt(1) := myVecOfSInt(0) + 3

// Create a vector of 3 different type elements
val myVecOfMixedUInt = Vec(UInt(3 bits), UInt(5 bits), UInt(8 bits))

val x, y, z = UInt(8 bits)
val myVecOf_xyz_ref = Vec(x, y, z)

// Iterate on a vector
for(element <- myVecOf_xyz_ref) {
  element := 0   // Assign x, y, z with the value 0
}

// Map on vector
myVecOfMixedUInt.map(_ := 0) // Assign all elements with value 0

// Assign 3 to the first element of the vector
myVecOf_xyz_ref(1) := 3
```

5.6.3 Operators

The following operators are available for the `Vec` type:

Comparison

Operator	Description	Return type
x === y	Equality	Bool
x =/= y	Inequality	Bool

```
// Create a vector of 2 signed integers
val vec2 = Vec(SInt(8 bits), 2)
val vec1 = Vec(SInt(8 bits), 2)

myBool := vec2 === vec1  // Compare all elements
```

Type cast

Operator	Description	Return
x.asBits	Binary cast to Bits	Bits(w(x) bits)

```
// Create a vector of 2 signed integers
val vec1 = Vec(SInt(8 bits), 2)

myBits_16bits := vec1.asBits
```

Misc

Operator	Description	Return
x.getBitsWidth	Return the full size of the Vec	Int

```
// Create a vector of 2 signed integers
val vec1 = Vec(SInt(8 bits), 2)

println(vec1.getBitsWidth) // 16
```

Warning: SpinalHDL fixed-point support is only partially used/tested, if you find any bugs with it, or you think that some functionality is missing, please create a Github issue. Also, please do not use undocumented features in your code.

5.7 UFix/SFix

5.7.1 Description

The `UFix` and `SFix` types correspond to a vector of bits that can be used for fixed-point arithmetic.

5.7.2 Declaration

The syntax to declare a fixed-point number is as follows:

Unsigned Fixed-Point

Syntax	bit width	resolution	max	min
UFix(peak: ExpNumber, resolution: ExpNumber)	peak-resolution	2^resolution	2^peak-2^resolution	0
UFix(peak: ExpNumber, width: BitCount)	width	2^(peak-width)	2^peak-2^(peak-width)	0

Signed Fixed-Point

Syntax	bit width	resolution	max	min
SFix(peak: ExpNumber, resolution: ExpNumber)	peak-resolution+1	2^resolution	2^peak-2^resolution	-(2^peak)
SFix(peak: ExpNumber, width: BitCount)	width	2^(peak-width-1)	2^peak-2^(peak-width-1)	-(2^peak)

Format

The chosen format follows the usual way of defining fixed-point number format using Q notation. More information can be found on the Wikipedia page about the Q number format.

For example Q8.2 will mean a fixed-point number of 8+2 bits, where 8 bits are used for the natural part and 2 bits for the fractional part. If the fixed-point number is signed, one more bit is used for the sign.

The resolution is defined as being the smallest power of two that can be represented in this number.

Note: To make representing power-of-two numbers less error prone, there is a numeric type in `spinal.core` called `ExpNumber`, which is used for the fixed-point type constructors. A convenience wrapper exists for this type, in the form of the `exp` function (used in the code samples on this page).

Examples

```
// Unsigned Fixed-Point
val UQ_8_2 = UFix(peak = 8 exp, resolution = -2 exp) // bit width = 8 - (-2) = 10 bits
val UQ_8_2 = UFix(8 exp, -2 exp)

val UQ_8_2 = UFix(peak = 8 exp, width = 10 bits)
val UQ_8_2 = UFix(8 exp, 10 bits)

// Signed Fixed-Point
val Q_8_2 = SFix(peak = 8 exp, resolution = -2 exp) // bit width = 8 - (-2) + 1 = 11␣
↪bits
val Q_8_2 = SFix(8 exp, -2 exp)

val Q_8_2 = SFix(peak = 8 exp, width = 11 bits)
val Q_8_2 = SFix(8 exp, 11 bits)
```

5.7.3 Assignments

Valid Assignments

An assignment to a fixed-point value is valid when there is no bit loss. Any bit loss will result in an error.

If the source fixed-point value is too big, the `.truncated` function will allow you to resize the source number to match the destination size.

Example

```
val i16_m2 = SFix(16 exp, -2 exp)
val i16_0  = SFix(16 exp,  0 exp)
val i8_m2  = SFix( 8 exp, -2 exp)
val o16_m2 = SFix(16 exp, -2 exp)
val o16_m0 = SFix(16 exp,  0 exp)
val o14_m2 = SFix(14 exp, -2 exp)

o16_m2 := i16_m2            // OK
o16_m0 := i16_m2            // Not OK, Bit loss
o14_m2 := i16_m2            // Not OK, Bit loss
o16_m0 := i16_m2.truncated  // OK, as it is resized
o14_m2 := i16_m2.truncated  // OK, as it is resized
```

From a Scala constant

Scala `BigInt` or `Double` types can be used as constants when assigning to `UFix` or `SFix` signals.

Example

```
val i4_m2 = SFix(4 exp, -2 exp)
i4_m2 := 1.25    // Will load 5 in i4_m2.raw
i4_m2 := 4       // Will load 16 in i4_m2.raw
```

5.7.4 Raw value

The integer representation of the fixed-point number can be read or written by using the `raw` property.

Example

```
val UQ_8_2 = UFix(8 exp, 10 bits)
UQ_8_2.raw := 4        // Assign the value corresponding to 1.0
UQ_8_2.raw := U(17)    // Assign the value corresponding to 4.25
```

5.7.5 Operators

The following operators are available for the `UFix` type:

Arithmetic

Operator	Description	Returned resolution	Returned amplitude
x + y	Addition	Min(x.resolution, y.resolution)	Max(x.amplitude, y.amplitude)
x - y	Subtraction	Min(x.resolution, y.resolution)	Max(x.amplitude, y.amplitude)
x * y	Multiplication	x.resolution * y.resolution)	x.amplitude * y.amplitude
x >> y	Arithmetic shift right, y : Int	x.amplitude >> y	x.resolution >> y
x << y	Arithmetic shift left, y : Int	x.amplitude << y	x.resolution << y
x >>| y	Arithmetic shift right, y : Int	x.amplitude >> y	x.resolution
x <<| y	Arithmetic shift left, y : Int	x.amplitude << y	x.resolution

Comparison

Operator	Description	Return type
x === y	Equality	Bool
x =/= y	Inequality	Bool
x > y	Greater than	Bool
x >= y	Greater than or equal	Bool
x > y	Less than	Bool
x >= y	Less than or equal	Bool

Type cast

Operator	Description	Return
x.asBits	Binary cast to Bits	Bits(w(x) bits)
x.asUInt	Binary cast to UInt	UInt(w(x) bits)
x.asSInt	Binary cast to SInt	SInt(w(x) bits)
x.asBools	Cast into a array of Bool	Vec(Bool,width(x))
x.toUInt	Return the corresponding UInt (with truncation)	UInt
x.toSInt	Return the corresponding SInt (with truncation)	SInt
x.toUFix	Return the corresponding UFix	UFix
x.toSFix	Return the corresponding SFix	SFix

Misc

Name	Return	Description
x.maxValue	Return the maximum value storable	Double
x.minValue	Return the minimum value storable	Double
x.resolution	x.amplitude * y.amplitude	Double

Warning: SpinalHDL floating-point support is under development and only partially used/tested, if you have any bugs with it, or you think that some functionality is missing, please create a Github issue. Also, please do not use undocumented features in your code.

5.8 Floating

5.8.1 Description

The `Floating` type corresponds to IEEE-754 encoded numbers. A second type called `RecFloating` helps in simplifying your design by recoding the floating-point value simplify some edge cases in IEEE-754 floating-point.

It's composed of a sign bit, an exponent field and a mantissa field. The widths of the different fields are defined in the IEEE-754 or de-facto standards.

This type can be used with the following import:

```
import spinal.lib.experimental.math._
```

IEEE-754 floating format

The numbers are encoded into IEEE-754 floating-point format.

Recoded floating format

Since IEEE-754 has some quirks about denormalized numbers and special values, Berkeley proposed another way of recoding floating-point values.

The mantissa is modified so that denormalized values can be treated the same as the normalized ones.

The exponent field is one bit larger that one of the IEEE-754 number.

The sign bit is kept unchanged between the two encodings.

Examples can be found here

Zero

The zero is encoded with the three leading zeros of the exponent field being set to zero.

Denormalized values

Denormalized values are encoded in the same way as a normal floating-point number. The mantissa is shifted so that the first one becomes implicit. The exponent is encoded as 107 (decimal) plus the index of the highest bit set to 1.

Normalized values

The recoded mantissa for normalized values is exactly the same as the original IEEE-754 mantissa. The recoded exponent is encoded as 130 (decimal) plus the original exponent value.

Infinity

The recoded mantissa value is treated as don't care. The recoded exponent three highest bits is 6 (decimal), the rest of the exponent can be treated as don't care.

NaN

The recoded mantissa for normalized values is exactly the same as the original IEEE-754 mantissa. The recoded exponent three highest bits is 7 (decimal), the rest of the exponent can be treated as don't care.

5.8.2 Declaration

The syntax to declare a floating-point number is as follows:

IEEE-754 Number

Syntax	Description
Floating(exponentSize: Int, mantissaSize: Int)	IEEE-754 Floating-point value with a custom exponent and mantissa size
Floating16()	IEEE-754 Half precision floating-point number
Floating32()	IEEE-754 Single precision floating-point number
Floating64()	IEEE-754 Double precision floating-point number
Floating128()	IEEE-754 Quad precision floating-point number

Recoded floating-point number

Syntax	Description
RecFloating(exponentSize: Int, mantissaSize: Int)	Recoded Floating-point value with a custom exponent and mantissa size
RecFloating16()	Recoded Half precision floating-point number
RecFloating32()	Recoded Single precision floating-point number
RecFloating64()	Recoded Double precision floating-point number
RecFloating128()	Recoded Quad precision floating-point number

5.8.3 Operators

The following operators are available for the `Floating` and `RecFloating` types:

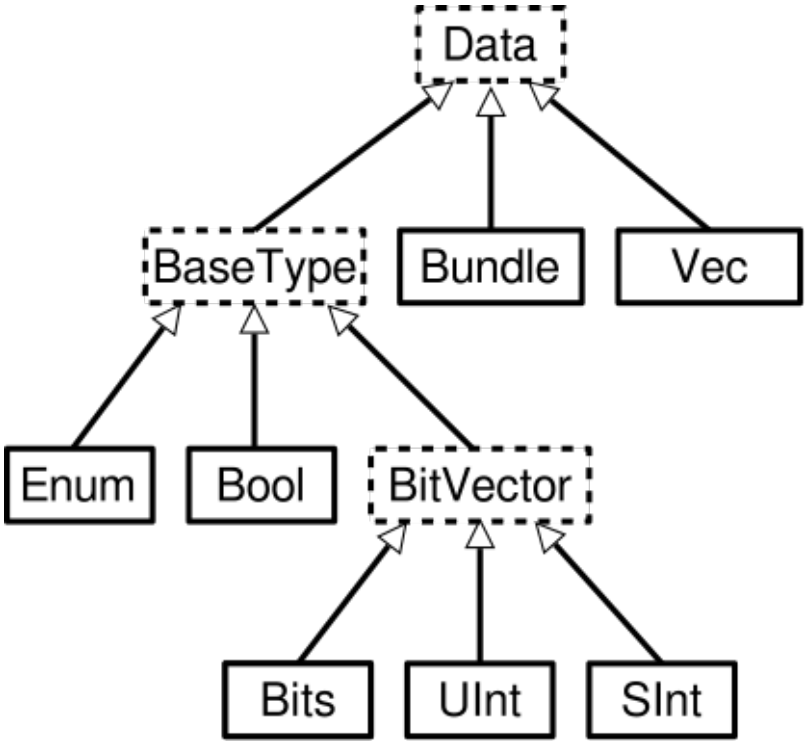
Type cast

Operator	Description	Return
x.asBits	Binary cast to Bits	Bits(w(x) bits)
x.asBools	Cast into a array of Bool	Vec(Bool,width(x))
x.toUInt(size: Int)	Return the corresponding UInt (with truncation)	UInt
x.toSInt(size: Int)	Return the corresponding SInt (with truncation)	SInt
x.fromUInt	Return the corresponding Floating (with truncation)	UInt
x.fromSInt	Return the corresponding Floating (with truncation)	SInt

5.9 Introduction

The language provides 5 base types, and 2 composite types that can be used.

- Base types: *Bool* , *Bits* , *UInt* for unsigned integers, *SInt* for signed integers and *Enum*.
- Composite types: *Bundle* and *Vec*.



In addition to the base types, Spinal has support under development for:

- *Fixed-point* numbers (partial support)
- *Floating-point* numbers (experimental support)

Finally, a special type is available for checking equality between a BitVector and a bits constant that contains holes (don't care values). An example is shown below:

```
val myBits  = Bits(8 bits)
val itMatch = myBits === M"00--10--" // - don't care value
```

CHAPTER

SIX

STRUCTURING

6.1 Component and hierarchy

6.1.1 Introduction

Like in VHDL and Verilog, you can define components that can be used to build a design hierarchy. However, in SpinalHDL, you don't need to bind their ports at instantiation:

```
class AdderCell extends Component {
  // Declaring external ports in a Bundle called `io` is recommended
  val io = new Bundle {
    val a, b, cin = in Bool()
    val sum, cout = out Bool()
  }
  // Do some logic
  io.sum := io.a ^ io.b ^ io.cin
  io.cout := (io.a & io.b) | (io.a & io.cin) | (io.b & io.cin)
}

class Adder(width: Int) extends Component {
  ...
  // Create 2 AdderCell instances
  val cell0 = new AdderCell
  val cell1 = new AdderCell
  cell1.io.cin := cell0.io.cout   // Connect cout of cell0 to cin of cell1

  // Another example which creates an array of ArrayCell instances
  val cellArray = Array.fill(width)(new AdderCell)
  cellArray(1).io.cin := cellArray(0).io.cout   // Connect cout of cell(0) to cin of␣
↪cell(1)
  ...
}
```

Tip:

`val io = new Bundle { ... }`
Declaring external ports in a `Bundle` called `io` is recommended. If you name your bundle `io`, SpinalHDL will check that all of its elements are defined as inputs or outputs.

6.1.2 Input / output definition

The syntax to define inputs and outputs is as follows:

Syntax	Description	Return
in Bool()/out Bool()	Create an input Bool/output Bool	Bool
in/out Bits/UInt/SInt[(x bit)]	Create an input/output of the corresponding type	Bits/UInt/SInt
in/out(T)	For all other data types, you may have to add some brackets around it. Sorry, this is a Scala limitation.	T
master/slave(T)	This syntax is provided by the `spinal.lib` library (If you annotate your object with the `slave` syntax, then import `spinal.lib.slave` instead). T should extend `IMasterSlave` – Some documentation is available *here*. You may not actually need the brackets, so `master T` is fine as well.	T

There are some rules to follow with component interconnection:

- Components can only **read** output and input signals of child components.
- Components can read their own output port values (unlike in VHDL).

Tip: If for some reason you need to read signals from far away in the hierarchy (such as for debugging or temporal patches), you can do it by using the value returned by `some.where.else.theSignal.pull()`

6.1.3 Pruned signals

SpinalHDL only generates things which are directly or indirectly required to drive the outputs of your top-level entity.

All other signals (the useless ones) are removed from the RTL generation and are inserted into a list of pruned signals. You can get this list via the `printPruned` and the `printPrunedIo` functions on the generated `SpinalReport` object:

```
class TopLevel extends Component {
  val io = new Bundle {
    val a,b = in UInt(8 bits)
    val result = out UInt(8 bits)
  }

  io.result := io.a + io.b

  val unusedSignal = UInt(8 bits)
  val unusedSignal2 = UInt(8 bits)

  unusedSignal2 := unusedSignal
}

object Main {
  def main(args: Array[String]) {
    SpinalVhdl(new TopLevel).printPruned()
    //This will report :
    //  [Warning] Unused wire detected : toplevel/unusedSignal : UInt[8 bits]
    //  [Warning] Unused wire detected : toplevel/unusedSignal2 : UInt[8 bits]
  }
}
```

If you want to keep a pruned signal in the generated RTL for debugging reasons, you can use the `keep` function of that signal:

```
class TopLevel extends Component {
  val io = new Bundle {
    val a, b = in UInt(8 bits)
    val result = out UInt(8 bits)
  }

  io.result := io.a + io.b

  val unusedSignal = UInt(8 bits)
  val unusedSignal2 = UInt(8 bits).keep()

  unusedSignal  := 0
  unusedSignal2 := unusedSignal
}

object Main {
  def main(args: Array[String]) {
    SpinalVhdl(new TopLevel).printPruned()
    // This will report nothing
  }
}
```

6.1.4 Parametrized Hardware ("Generic" in VHDL, "Parameter" in Verilog)

If you want to parameterize your component, you can give parameters to the constructor of the component as follows:

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

object Main {
  def main(args: Array[String]) {
    SpinalVhdl(new MyAdder(32 bits))
  }
}
```

If you have several parameters, it is a good practice to give a specific configuration class as follows:

```
case class MySocConfig(axiFrequency  : HertzNumber,
                       onChipRamSize : BigInt,
                       cpu           : RiscCoreConfig,
                       iCache        : InstructionCacheConfig)

class MySoc(config: MySocConfig) extends Component {
  ...
}
```

6.1.5 Synthesized component names

Within a module, each component has a name, called a "partial name". The "full" name is built by joining every component's parent name with "_", for example: `io_clockDomain_reset`. You can use `setName` to replace this convention with a custom name. This is especially useful when interfacing with external components. The other methods are called `getName`, `setPartialName`, and `getPartialName` respectively.

When synthesized, each module gets the name of the Scala class defining it. You can override this as well with `setDefinitionName`.

6.2 Area

6.2.1 Introduction

Sometimes, creating a `Component` to define some logic is overkill because you:

- Need to define all construction parameters and IO (verbosity, duplication)
- Split your code (more than needed)

For this kind of case you can use an `Area` to define a group of signals/logic:

```
class UartCtrl extends Component {
  ...
  val timer = new Area {
    val counter = Reg(UInt(8 bit))
    val tick = counter === 0
    counter := counter - 1
    when(tick) {
      counter := 100
    }
  }

  val tickCounter = new Area {
    val value = Reg(UInt(3 bit))
    val reset = False
    when(timer.tick) {          // Refer to the tick from timer area
      value := value + 1
    }
    when(reset) {
      value := 0
    }
  }

  val stateMachine = new Area {
    ...
  }
}
```

Tip:

In VHDL and Verilog, sometimes prefixes are used to separate variables into logical sections. It is suggested that you use `Area` instead of this in SpinalHDL.

Note: *ClockingArea* is a special kind of `Area` that allows you to define chunks of hardware which use a given

`ClockDomain`

6.3 Function

6.3.1 Introduction

The ways you can use Scala functions to generate hardware are radically different than VHDL/Verilog for many reasons:

- You can instantiate registers, combinational logic, and components inside them.
- You don't have to play with `process`/`@always` blocks that limit the scope of assignment of signals.
- Everything is passed by reference, which allows easy manipulation.
  For example, you can give a bus to a function as an argument, then the function can internally read/write to it. You can also return a Component, a Bus, or anything else from Scala and the Scala world.

6.3.2 RGB to gray

For example, if you want to convert a Red/Green/Blue color into greyscale by using coefficients, you can use functions to apply them:

```
// Input RGB color
val r, g, b = UInt(8 bits)

// Define a function to multiply a UInt by a Scala Float value.
def coef(value: UInt, by: Float): UInt = (value * U((255 * by).toInt, 8 bits) >> 8)

// Calculate the gray level
val gray = coef(r, 0.3f) + coef(g, 0.4f) + coef(b, 0.3f)
```

6.3.3 Valid Ready Payload bus

For instance, if you define a simple bus with `valid`, `ready`, and `payload` signals, you can then define some useful functions inside of it.

```
case class MyBus(payloadWidth: Int) extends Bundle with IMasterSlave {
  val valid   = Bool()
  val ready   = Bool()
  val payload = Bits(payloadWidth bits)

  // Define the direction of the data in a master mode
  override def asMaster(): Unit = {
    out(valid, payload)
    in(ready)
  }

  // Connect that to this
  def <<(that: MyBus): Unit = {
    this.valid   := that.valid
    that.ready   := this.ready
    this.payload := that.payload
  }
```

(continues on next page)

(continued from previous page)

```
  // Connect this to the FIFO input, return the fifo output
  def queue(size: Int): MyBus = {
    val fifo = new MyBusFifo(payloadWidth, size)
    fifo.io.push << this
    return fifo.io.pop
  }
}

class MyBusFifo(payloadWidth: Int, depth: Int) extends Component {

  val io = new Bundle {
    val push = slave(MyBus(payloadWidth))
    val pop  = master(MyBus(payloadWidth))
  }

  val mem = Mem(Bits(payloadWidth bits), depth)

  // ...

}
```

6.4 Clock domains

6.4.1 Introduction

In SpinalHDL, clock and reset signals can be combined to create a **clock domain**. Clock domains can be applied to some areas of the design and then all synchronous elements instantiated into those areas will then **implicitly** use this clock domain.

Clock domain application works like a stack, which means that if you are in a given clock domain you can still apply another clock domain locally.

6.4.2 Instantiation

The syntax to define a clock domain is as follows (using EBNF syntax):

```
ClockDomain(
  clock: Bool
  [,reset: Bool]
  [,softReset: Bool]
  [,clockEnable: Bool]
  [,frequency: IClockDomainFrequency]
  [,config: ClockDomainConfig]
)
```

This definition takes five parameters:

Argument	Description	Default
clock	Clock signal that defines the domain	
reset	Reset signal. If a register exists which needs a reset and the clock domain doesn't provide one, an error message will be displayed	null
softReset	Reset which infers an additional synchronous reset	null
clockEnable	The goal of this signal is to disable the clock on the whole clock domain without having to manually implement that on each synchronous element	null
frequency	Allows you to specify the frequency of the given clock domain and later read it in your design	UnknownFrequency
config	Specify the polarity of signals and the nature of the reset	Current config

An applied example to define a specific clock domain within the design is as follows:

```
val coreClock = Bool()
val coreReset = Bool()

// Define a new clock domain
val coreClockDomain = ClockDomain(coreClock, coreReset)

// Use this domain in an area of the design
val coreArea = new ClockingArea(coreClockDomain) {
  val coreClockedRegister = Reg(UInt(4 bit))
}
```

Configuration

In addition to *constructor parameters*, the following elements of each clock domain are configurable via a ClockDomainConfigclass:

Property	Valid values
clockEdge	RISING, FALLING
resetKind	ASYNC, SYNC, and BOOT which is supported by some FPGAs (where FF values are loaded by the bitstream)
resetActiveLevel	HIGH, LOW
softResetActiveLevel	HIGH, LOW
clockEnableActiveLevel	HIGH, LOW

```
class CustomClockExample extends Component {
  val io = new Bundle {
    val clk    = in Bool()
    val resetn = in Bool()
    val result = out UInt (4 bits)
  }

  // Configure the clock domain
  val myClockDomain = ClockDomain(
    clock  = io.clk,
    reset  = io.resetn,
    config = ClockDomainConfig(
```

(continues on next page)

(continued from previous page)

```
      clockEdge        = RISING,
      resetKind        = ASYNC,
      resetActiveLevel = LOW
    )
  )

  // Define an Area which use myClockDomain
  val myArea = new ClockingArea(myClockDomain) {
    val myReg = Reg(UInt(4 bits)) init(7)

    myReg := myReg + 1

    io.result := myReg
  }
}
```

By default, a `ClockDomain` is applied to the whole design. The configuration of this default domain is:

- Clock : rising edge
- Reset : asynchronous, active high
- No clock enable

This corresponds to the following `ClockDomainConfig`:

```
val defaultCC = ClockDomainConfig(
  clockEdge        = RISING,
  resetKind        = ASYNC,
  resetActiveLevel = HIGH
)
```

Internal clock

An alternative syntax to create a clock domain is the following:

```
ClockDomain.internal(
  name: String,
  [config: ClockDomainConfig,]
  [withReset: Boolean,]
  [withSoftReset: Boolean,]
  [withClockEnable: Boolean,]
  [frequency: IClockDomainFrequency]
)
```

This definition takes six parameters:

Argument	Description	Default
`name`	Name of *clk* and *reset* signal	
`config`	Specify polarity of signals and the nature of the reset	Current config
`withReset`	Add a reset signal	true
`withSoftReset`	Add a soft reset signal	false
`withClockEnable`	Add a clock enable	false
`frequency`	Frequency of the clock domain	UnknownFrequency

The advantage of this approach is to create clock and reset signals with a known/specified name instead of an inherited one.

Once created, you have to assign the ClockDomain's signals, as shown in the example below:

```
class InternalClockWithPllExample extends Component {
  val io = new Bundle {
    val clk100M = in Bool()
    val aReset  = in Bool()
    val result  = out UInt (4 bits)
  }
  // myClockDomain.clock will be named myClockName_clk
  // myClockDomain.reset will be named myClockName_reset
  val myClockDomain = ClockDomain.internal("myClockName")

  // Instantiate a PLL (probably a BlackBox)
  val pll = new Pll()
  pll.io.clkIn := io.clk100M

  // Assign myClockDomain signals with something
  myClockDomain.clock := pll.io.clockOut
  myClockDomain.reset := io.aReset || !pll.io.

  // Do whatever you want with myClockDomain
  val myArea = new ClockingArea(myClockDomain) {
    val myReg = Reg(UInt(4 bits)) init(7)
    myReg := myReg + 1

    io.result := myReg
  }
}
```

External clock

You can define a clock domain which is driven by the outside anywhere in your source. It will then automatically add clock and reset wires from the top level inputs to all synchronous elements.

```
ClockDomain.external(
  name: String,
  [config: ClockDomainConfig,]
  [withReset: Boolean,]
  [withSoftReset: Boolean,]
  [withClockEnable: Boolean,]
  [frequency: IClockDomainFrequency]
)
```

The arguments to the ClockDomain.external function are exactly the same as in the ClockDomain.internal function. Below is an example of a design using ClockDomain.external:

```
class ExternalClockExample extends Component {
  val io = new Bundle {
    val result = out UInt (4 bits)
  }

  // On the top level you have two signals  :
  //     myClockName_clk and myClockName_reset
  val myClockDomain = ClockDomain.external("myClockName")
```

(continues on next page)

(continued from previous page)

```
  val myArea = new ClockingArea(myClockDomain) {
    val myReg = Reg(UInt(4 bits)) init(7)
    myReg := myReg + 1

    io.result := myReg
  }
}
```

Context

You can retrieve in which clock domain you are by calling `ClockDomain.current` anywhere.

The returned `ClockDomain` instance has the following functions that can be called:

name	Description	Return
frequency.getValue	Return the frequency of the clock domain	Double
hasReset	Return if the clock domain has a reset signal	Boolean
hasSoftReset	Return if the clock domain has a soft reset signal	Boolean
hasClockEnable	Return if the clock domain has a clock enable signal	Boolean
readClockWire	Return a signal derived from the clock signal	Bool
readResetWire	Return a signal derived from the soft reset signal	Bool
readSoftResetWire	Return a signal derived from the reset signal	Bool
readClockEnableWire	Return a signal derived from the clock enable signal	Bool
isResetActive	Return True when the reset is active	Bool
isSoftResetActive	Return True when the soft reset is active	Bool
isClockEnableActive	Return True when the clock enable is active	Bool

An example is included below where a UART controller uses the frequency specification to set its clock divider:

```
val coreClockDomain = ClockDomain(coreClock, coreReset,␣
↪frequency=FixedFrequency(100e6))

val coreArea = new ClockingArea(coreClockDomain) {
  val ctrl = new UartCtrl()
  ctrl.io.config.clockDivider := (coreClk.frequency.getValue / 57.6e3 / 8).toInt
}
```

6.4.3 Clock domain crossing

SpinalHDL checks at compile time that there are no unwanted/unspecified cross clock domain signal reads. If you want to read a signal that is emitted by another `ClockDomain` area, you should add the `crossClockDomain` tag to the destination signal as depicted in the following example:

```
//             _____                          _____             _____
//            |     |  (crossClockDomain)  |     |           |     |
//  dataIn -->|     |--------------------->|     |---------->|     |--> dataOut
//            | FF  |                      | FF  |           | FF  |
//  clkA   -->|     |             clkB -->|     |   clkB -->|     |
//  rstA   -->|_____|             rstB -->|_____|   rstB -->|_____|



// Implementation where clock and reset pins are given by components' IO
class CrossingExample extends Component {
  val io = new Bundle {
    val clkA = in Bool()
    val rstA = in Bool()

    val clkB = in Bool()
    val rstB = in Bool()

    val dataIn  = in Bool()
    val dataOut = out Bool()
  }

  // sample dataIn with clkA
  val area_clkA = new ClockingArea(ClockDomain(io.clkA,io.rstA)) {
    val reg = RegNext(io.dataIn) init(False)
  }

  // 2 register stages to avoid metastability issues
  val area_clkB = new ClockingArea(ClockDomain(io.clkB,io.rstB)) {
    val buf0   = RegNext(area_clkA.reg) init(False) addTag(crossClockDomain)
    val buf1   = RegNext(buf0)          init(False)
  }

  io.dataOut := area_clkB.buf1
}


// Alternative implementation where clock domains are given as parameters
class CrossingExample(clkA : ClockDomain,clkB : ClockDomain) extends Component {
  val io = new Bundle {
    val dataIn  = in Bool()
    val dataOut = out Bool()
  }

  // sample dataIn with clkA
  val area_clkA = new ClockingArea(clkA) {
    val reg = RegNext(io.dataIn) init(False)
  }

  // 2 register stages to avoid metastability issues
  val area_clkB = new ClockingArea(clkB) {
```

(continues on next page)

(continued from previous page)

```
    val buf0   = RegNext(area_clkA.reg) init(False) addTag(crossClockDomain)
    val buf1   = RegNext(buf0)          init(False)
  }

  io.dataOut := area_clkB.buf1
}
```

In general, you can use 2 or more flip-flop driven by the destination clock domain to prevent metastability. The `BufferCC(input: T, init: T = null, bufferDepth: Int = 2)` function provided in `spinal.lib._` will instantiate the necessary flip-flops (the number of flip-flops will depends on the `bufferDepth` parameter) to mitigate the phenomena.

```
class CrossingExample(clkA : ClockDomain,clkB : ClockDomain) extends Component {
  val io = new Bundle {
    val dataIn  = in Bool()
    val dataOut = out Bool()
  }

  // sample dataIn with clkA
  val area_clkA = new ClockingArea(clkA) {
    val reg = RegNext(io.dataIn) init(False)
  }

  // BufferCC to avoid metastability issues
  val area_clkB = new ClockingArea(clkB) {
    val buf1   = BufferCC(area_clkA.reg, False)
  }

  io.dataOut := area_clkB.buf1
}
```

Warning: The `BufferCC` function is only for signals of type `Bit`, or `Bits` operating as Gray-coded counters (only 1 bit-flip per clock cycle), and can not used for multi-bit cross-domain processes.

6.4.4 Special clocking Areas

Slow Area

A `SlowArea` is used to create a new clock domain area which is slower than the current one:

```
class TopLevel extends Component {

  // Use the current clock domain : 100MHz
  val areaStd = new Area {
    val counter = out(CounterFreeRun(16).value)
  }

  // Slow the current clockDomain by 4 : 25 MHz
  val areaDiv4 = new SlowArea(4) {
    val counter = out(CounterFreeRun(16).value)
  }

  // Slow the current clockDomain to 50MHz
```

(continues on next page)

(continued from previous page)

```
  val area50Mhz = new SlowArea(50 MHz) {
    val counter = out(CounterFreeRun(16).value)
  }
}

def main(args: Array[String]) {
  new SpinalConfig(
    defaultClockDomainFrequency = FixedFrequency(100 MHz)
  ).generateVhdl(new TopLevel)
}
```

ResetArea

A `ResetArea` is used to create a new clock domain area where a special reset signal is combined with the current clock domain reset:

```
class TopLevel extends Component {

  val specialReset = Bool()

  // The reset of this area is done with the specialReset signal
  val areaRst_1 = new ResetArea(specialReset, false) {
    val counter = out(CounterFreeRun(16).value)
  }

  // The reset of this area is a combination between the current reset and the␣
↪specialReset
  val areaRst_2 = new ResetArea(specialReset, true) {
    val counter = out(CounterFreeRun(16).value)
  }
}
```

ClockEnableArea

A `ClockEnableArea` is used to add an additional clock enable in the current clock domain:

```
class TopLevel extends Component {

  val clockEnable = Bool()

  // Add a clock enable for this area
  val area_1 = new ClockEnableArea(clockEnable) {
    val counter = out(CounterFreeRun(16).value)
  }
}
```

6.5 Instantiate VHDL and Verilog IP

6.5.1 Description

A blackbox allows the user to integrate an existing VHDL/Verilog component into the design by just specifying its interfaces. It's up to the simulator or synthesizer to do the elaboration correctly.

6.5.2 Defining an blackbox

An example of how to define a blackbox is shown below:

```
// Define a Ram as a BlackBox
class Ram_1w_1r(wordWidth: Int, wordCount: Int) extends BlackBox {
  // Add VHDL Generics / Verilog parameters to the blackbox
  // You can use String, Int, Double, Boolean, and all SpinalHDL base
  // types as generic values
  addGeneric("wordCount", wordCount)
  addGeneric("wordWidth", wordWidth)

  // Define IO of the VHDL entity / Verilog module
  val io = new Bundle {
    val clk = in Bool()
    val wr = new Bundle {
      val en   = in Bool()
      val addr = in UInt (log2Up(wordCount) bit)
      val data = in Bits (wordWidth bit)
    }
    val rd = new Bundle {
      val en   = in Bool()
      val addr = in UInt (log2Up(wordCount) bit)
      val data = out Bits (wordWidth bit)
    }
  }

  // Map the current clock domain to the io.clk pin
  mapClockDomain(clock=io.clk)
}
```

In VHDL, signals of type `Bool` will be translated into `std_logic` and `Bits` into `std_logic_vector`. If you want to get `std_ulogic`, you have to use a `BlackBoxULogic` instead of `BlackBox`.
In Verilog, `BlackBoxUlogic` has no effect.

```
class Ram_1w_1r(wordWidth: Int, wordCount: Int) extends BlackBoxULogic {
  ...
}
```

6.5.3 Generics

There are two different ways to declare generics:

```
class Ram(wordWidth: Int, wordCount: Int) extends BlackBox {
    addGeneric("wordCount", wordCount)
    addGeneric("wordWidth", wordWidth)

    // OR

    val generic = new Generic {
      val wordCount = Ram.this.wordCount
      val wordWidth = Ram.this.wordWidth
    }
}
```

6.5.4 Instantiating a blackbox

Instantiating a `BlackBox` is just like instantiating a `Component`:

```
// Create the top level and instantiate the Ram
class TopLevel extends Component {
  val io = new Bundle {
    val wr = new Bundle {
      val en   = in Bool()
      val addr = in UInt (log2Up(16) bit)
      val data = in Bits (8 bit)
    }
    val rd = new Bundle {
      val en   = in Bool()
      val addr = in UInt (log2Up(16) bit)
      val data = out Bits (8 bit)
    }
  }

  // Instantiate the blackbox
  val ram = new Ram_1w_1r(8,16)

  // Connect all the signals
  io.wr.en   <> ram.io.wr.en
  io.wr.addr <> ram.io.wr.addr
  io.wr.data <> ram.io.wr.data
  io.rd.en   <> ram.io.rd.en
  io.rd.addr <> ram.io.rd.addr
  io.rd.data <> ram.io.rd.data
}

object Main {
  def main(args: Array[String]): Unit = {
    SpinalVhdl(new TopLevel)
  }
}
```

6.5.5 Clock and reset mapping

In your blackbox definition you have to explicitly define clock and reset wires. To map signals of a `ClockDomain` to corresponding inputs of the blackbox you can use the `mapClockDomain` or `mapCurrentClockDomain` function. `mapClockDomain` has the following parameters:

name	type	default	description
clockDomain	ClockDomain	ClockDomain.current	Specify the clockDomain which provides the signals
clock	Bool	Nothing	Blackbox input which should be connected to the clockDomain clock
reset	Bool	Nothing	Blackbox input which should be connected to the clockDomain reset
enable	Bool	Nothing	Blackbox input which should be connected to the clockDomain enable

`mapCurrentClockDomain` has almost the same parameters as `mapClockDomain` but without the clockDomain.

For example:

```
class MyRam(clkDomain: ClockDomain) extends BlackBox {

  val io = new Bundle {
    val clkA = in Bool()
    ...
    val clkB = in Bool()
    ...
  }

  // Clock A is map on a specific clock Domain
  mapClockDomain(clkDomain, io.clkA)
  // Clock B is map on the current clock domain
  mapCurrentClockDomain(io.clkB)
}
```

6.5.6 io prefix

In order to avoid the prefix "io_" on each of the IOs of the blackbox, you can use the function `noIoPrefix()` as shown below :

```
// Define the Ram as a BlackBox
class Ram_1w_1r(wordWidth: Int, wordCount: Int) extends BlackBox {

  val generic = new Generic {
    val wordCount = Ram_1w_1r.this.wordCount
    val wordWidth = Ram_1w_1r.this.wordWidth
  }

  val io = new Bundle {
    val clk = in Bool()

    val wr = new Bundle {
      val en   = in Bool()
      val addr = in UInt (log2Up(_wordCount) bit)
      val data = in Bits (_wordWidth bit)
    }
    val rd = new Bundle {
      val en   = in Bool()
```

(continues on next page)

(continued from previous page)

```
      val addr = in UInt (log2Up(_wordCount) bit)
      val data = out Bits (_wordWidth bit)
    }
  }

  noIoPrefix()

  mapCurrentClockDomain(clock=io.clk)
}
```

6.5.7 Rename all io of a blackbox

IOs of a `BlackBox` or `Component` can be renamed at compile-time using the `addPrePopTask` function. This function takes a no-argument function to be applied during compilation, and is useful for adding renaming passes, as shown in the following example:

```
class MyRam() extends Blackbox {

  val io = new Bundle {
    val clk = in Bool()
    val portA = new Bundle{
      val cs   = in Bool()
      val rwn  = in Bool()
      val dIn  = in Bits(32 bits)
      val dOut = out Bits(32 bits)
    }
    val portB = new Bundle{
      val cs   = in Bool()
      val rwn  = in Bool()
      val dIn  = in Bits(32 bits)
      val dOut = out Bits(32 bits)
    }
  }

  // Map the clk
  mapCurrentClockDomain(io.clk)

  // Remove io_ prefix
  noIoPrefix()

  // Function used to rename all signals of the blackbox
  private def renameIO(): Unit = {
    io.flatten.foreach(bt => {
      if(bt.getName().contains("portA")) bt.setName(bt.getName().repalce("portA_", "
↪") + "_A")
      if(bt.getName().contains("portB")) bt.setName(bt.getName().repalce("portB_", "
↪") + "_B")
    })
  }

  // Execute the function renameIO after the creation of the component
  addPrePopTask(() => renameIO())
}

// This code generate these names:
```

(continues on next page)

(continued from previous page)

```
//    clk
//    cs_A, rwn_A, dIn_A, dOut_A
//    cs_B, rwn_B, dIn_B, dOut_B
```

6.5.8 Add RTL source

With the function `addRTLPath()` you can associate your RTL sources with the blackbox. After the generation of your SpinalHDL code you can call the function `mergeRTLSource` to merge all of the sources together.

```
class MyBlackBox() extends Blackbox {

  val io = new Bundle {
    val clk   = in  Bool()
    val start = in Bool()
    val dIn   = in  Bits(32 bits)
    val dOut  = out Bits(32 bits)
    val ready = out Bool()
  }

  // Map the clk
  mapCurrentClockDomain(io.clk)

  // Remove io_ prefix
  noIoPrefix()

  // Add all rtl dependencies
  addRTLPath("./rtl/RegisterBank.v")                          // Add a verilog file
  addRTLPath(s"./rtl/myDesign.vhd")                           // Add a vhdl file
  addRTLPath(s"${sys.env("MY_PROJECT")}/myTopLevel.vhd")      // Use an environement␣
↪variable MY_PROJECT (System.getenv("MY_PROJECT"))
}

...

val report = SpinalVhdl(new MyBlackBox)
report.mergeRTLSource("mergeRTL") // Merge all rtl sources into mergeRTL.vhd and␣
↪mergeRTL.v files
```

6.5.9 VHDL - No numeric type

If you want to use only `std_logic_vector` in your blackbox component, you can add the tag `noNumericType` to the blackbox.

```
class MyBlackBox() extends BlackBox{
  val io = new Bundle {
    val clk       = in  Bool()
    val increment = in  Bool()
    val initValue = in  UInt(8 bits)
    val counter   = out UInt(8 bits)
  }

  mapCurrentClockDomain(io.clk)

  noIoPrefix()
```

(continues on next page)

(continued from previous page)

```
  addTag(noNumericType)  // Only std_logic_vector
}
```

The code above will generate the following VHDL:

```
component MyBlackBox is
  port(
    clk       : in  std_logic;
    increment : in  std_logic;
    initValue : in  std_logic_vector(7 downto 0);
    counter   : out std_logic_vector(7 downto 0)
  );
end component;
```

6.6 Preserving names

6.6.1 Introduction

This page will describe how SpinalHDL propagate names from the scala code to the generated hardware. Knowing them should enable you to preserve those names as much as possible to generate understandable netlists.

6.6.2 Nameable base class

All the things which can be named in SpinalHDL extends the Nameable base class which.

So in practice, the following classes extends Nameable :

- Component
- Area
- Data (UInt, SInt, Bundle, ...)

There is a few example of that Nameable API

```
class MyComponent extends Component{
  val a, b, c, d = Bool()
  b.setName("rawrr") // Force name
  c.setName("rawrr", weak = true) // Propose a name, will not be applied if a␣
↪stronger name is already applied
  d.setCompositeName(b, postfix = "wuff") // Force toto to be named as b.getName() + _
↪wuff"
}
```

Will generation :

```
module MyComponent (
);
  wire                a;
  wire                rawrr;
  wire                c;
  wire                rawrr_wuff;
endmodule
```

In general, you don't realy need to access that API, unless you want to do tricky stuff for debug reasons or for elaboration purposes.

6.6.3 Name extraction from Scala

First, since version 1.4.0, SpinalHDL use a scala compiler plugin which can provide a call back each time a new val is defined during the construction of an class.

There is a example showing more or less how SpinalHDL itself is implemented :

```
//spinal.idslplugin.ValCallback is the Scala compiler plugin feature which will␣
↪provide the callbacks
class Component extends spinal.idslplugin.ValCallback{
  override def valCallback[T](ref: T, name: String) : T = {
    println(s"Got $ref named $name") // Here we just print what we got as a demo.
    ref
  }
}

class UInt
class Bits
class MyComponent extends Component{
  val two = 2
  val wuff = "miaou"
  val toto = new UInt
  val rawrr = new Bits
}

object Debug3 extends App{
  new MyComponent()
  // ^ This will print :
  // Got 2 named two
  // Got miaou named wuff
  // Got spinal.tester.code.sandbox.UInt@691a7f8f named toto
  // Got spinal.tester.code.sandbox.Bits@161b062a named rawrr
}
```

Using that ValCallback "introspection" feature, SpinalHDL's Component classes are able to be aware of their content and its name.

But this also mean that if you want something to get a name, and you only rely on this automatic naming feature, the reference to your Data (UInt, SInt, ...) instances should be stored somewhere in a Component val.

For instance :

```
class MyComponent extends Component {
  val a,b = in UInt(8 bits) // Will be properly named
  val toto = out UInt(8 bits)   // same

  def doStuff(): Unit = {
    val tmp = UInt(8 bits) // This will not be named, as it isn't stored anywhere in␣
↪a component val (but there is a solution explained later)
    tmp := 0x20
    toto := tmp
  }
  doStuff()
}
```

Will generate :

```
module MyComponent (
  input      [7:0]    a,
```

(continues on next page)

(continued from previous page)

```
  input      [7:0]    b,
  output     [7:0]    toto
);
  //Note that the tmp signal defined in scala was "shortcuted" by SpinalHDL, as it␣
↪was unamed and technicaly "shortcutable"
  assign toto = 8'h20;
endmodule
```

6.6.4 Area in a Component

One important aspect in the naming system is that you can define new namespaces inside components and manipulate

For instance via Area :

```
class MyComponent extends Component {
  val logicA = new Area{   //This define a new namespace named "logicA
    val toggle = Reg(Bool) //This register will be named "logicA_toggle"
    toggle := !toggle
  }
}
```

Will generate

```
module MyComponent (
  input               clk,
  input               reset
);
  reg                 logicA_toggle;
  always @ (posedge clk) begin
    logicA_toggle <= (! logicA_toggle);
  end
endmodule
```

6.6.5 Area in a function

You can also define function which will create new Area which will provide a namespace for all its content :

```
class MyComponent extends Component {
  def isZero(value: UInt) = new Area {
    val comparator = value === 0
  }

  val value = in UInt (8 bits)
  val someLogic = isZero(value)

  val result = out Bool()
  result := someLogic.comparator
}
```

Which will generate :

```
module MyComponent (
  input      [7:0]    value,
  output              result
```

(continues on next page)

(continued from previous page)

```
);
  wire                  someLogic_comparator;

  assign someLogic_comparator = (value == 8'h0);
  assign result = someLogic_comparator;

endmodule
```

6.6.6 Composite in a function

Added in SpinalHDL 1.5.0, Composite which allow you to create a scope which will use as prefix another Nameable:

```
class MyComponent extends Component {
  //Basicaly, a Composite is an Area that use its construction parameter as namespace␣
↪prefix
  def isZero(value: UInt) = new Composite(value) {
    val comparator = value === 0
  }.comparator  //Note we don't return the Composite, but the element of the␣
↪composite that we are interested in

  val value = in UInt (8 bits)
  val result = out Bool()
  result := isZero(value)
}
```

Will generate :

```
module MyComponent (
  input      [7:0]    value,
  output              result
);
  wire                value_comparator;

  assign value_comparator = (value == 8'h0);
  assign result = value_comparator;

endmodule
```

6.6.7 Composite chains

You can also chain composites :

```
class MyComponent extends Component {
  def isZero(value: UInt) = new Composite(value) {
    val comparator = value === 0
  }.comparator


  def inverted(value: Bool) = new Composite(value) {
    val inverter = !value
  }.inverter

  val value = in UInt(8 bits)
```

(continues on next page)

(continued from previous page)

```
  val result = out Bool()
  result := inverted(isZero(value))
}
```

Will generate :

```
module MyComponent (
  input      [7:0]    value,
  output              result
);
  wire                value_comparator;
  wire                value_comparator_inverter;

  assign value_comparator = (value == 8'h0);
  assign value_comparator_inverter = (! value_comparator);
  assign result = value_comparator_inverter;

endmodule
```

6.6.8 Composite in a Bundle's function

This behaviour can be very useful when implementing Bundles utilities. For instance in the spinal.lib.Stream class is defined the following :

```
class Stream[T <: Data](val payloadType :  HardType[T]) extends Bundle {
  val valid   = Bool()
  val ready   = Bool()
  val payload = payloadType()

  def queue(size: Int): Stream[T] = new Composite(this){
    val fifo = new StreamFifo(payloadType, size)
    fifo.io.push << self    // 'self' refer to the Composite construction argument␣
↪(this in that example). It avoid having to do a boring 'Stream.this'
  }.fifo.io.pop

  def m2sPipe(): Stream[T] = new Composite(this) {
    val m2sPipe = Stream(payloadType)

    val rValid = RegInit(False)
    val rData = Reg(payloadType)

    self.ready := (!m2sPipe.valid) || m2sPipe.ready

    when(self.ready) {
      rValid := self.valid
      rData := self.payload
    }

    m2sPipe.valid := rValid
    m2sPipe.payload := rData
  }.m2sPipe
}
```

Which allow nested calls while preserving the names :

```
class MyComponent extends Component {
  val source = slave(Stream(UInt(8 bits)))
  val sink = master(Stream(UInt(8 bits)))
  sink << source.queue(size = 16).m2sPipe()
}
```

Will generate

```
module MyComponent (
  input               source_valid,
  output              source_ready,
  input      [7:0]    source_payload,
  output              sink_valid,
  input               sink_ready,
  output     [7:0]    sink_payload,
  input               clk,
  input               reset
);
  wire                source_fifo_io_pop_ready;
  wire                source_fifo_io_push_ready;
  wire                source_fifo_io_pop_valid;
  wire       [7:0]    source_fifo_io_pop_payload;
  wire       [4:0]    source_fifo_io_occupancy;
  wire       [4:0]    source_fifo_io_availability;
  wire                source_fifo_io_pop_m2sPipe_valid;
  wire                source_fifo_io_pop_m2sPipe_ready;
  wire       [7:0]    source_fifo_io_pop_m2sPipe_payload;
  reg                 source_fifo_io_pop_rValid;
  reg        [7:0]    source_fifo_io_pop_rData;

  StreamFifo source_fifo (
    .io_push_valid      (source_valid                 ), //i
    .io_push_ready      (source_fifo_io_push_ready    ), //o
    .io_push_payload    (source_payload               ), //i
    .io_pop_valid       (source_fifo_io_pop_valid     ), //o
    .io_pop_ready       (source_fifo_io_pop_ready     ), //i
    .io_pop_payload     (source_fifo_io_pop_payload   ), //o
    .io_flush           (1'b0                         ), //i
    .io_occupancy       (source_fifo_io_occupancy     ), //o
    .io_availability    (source_fifo_io_availability  ), //o
    .clk                (clk                          ), //i
    .reset              (reset                        )  //i
  );
  assign source_ready = source_fifo_io_push_ready;
  assign source_fifo_io_pop_ready = ((1'b1 && (! source_fifo_io_pop_m2sPipe_valid)) ␣
↪|| source_fifo_io_pop_m2sPipe_ready);
  assign source_fifo_io_pop_m2sPipe_valid = source_fifo_io_pop_rValid;
  assign source_fifo_io_pop_m2sPipe_payload = source_fifo_io_pop_rData;
  assign sink_valid = source_fifo_io_pop_m2sPipe_valid;
  assign source_fifo_io_pop_m2sPipe_ready = sink_ready;
  assign sink_payload = source_fifo_io_pop_m2sPipe_payload;
  always @ (posedge clk or posedge reset) begin
    if (reset) begin
      source_fifo_io_pop_rValid <= 1'b0;
    end else begin
      if(source_fifo_io_pop_ready)begin
        source_fifo_io_pop_rValid <= source_fifo_io_pop_valid;
```

(continues on next page)

(continued from previous page)

```
      end
    end
  end

  always @ (posedge clk) begin
    if(source_fifo_io_pop_ready)begin
      source_fifo_io_pop_rData <= source_fifo_io_pop_payload;
    end
  end
endmodule
```

6.6.9 Unamed signal handling

Since 1.5.0, for signal which end up without name, SpinalHDL will find a signal which is driven by that unamed signal and propagate its name. This can produce useful results as long you don't have too large island of unamed stuff.

The name attributed to such unamed signal is : _zz_ + drivenSignal.getName()

Note that this naming pattern is also used by the generation backend when they need to breakup some specific expressions or long chain of expression into multiple signals.

Verilog expression splitting

There is an instance of expressions (ex : the + operator) that SpinalHDL need to express in dedicated signals to match the behaviour with the Scala API :

```
class MyComponent extends Component {
  val a,b,c,d = in UInt(8 bits)
  val result = a + b + c + d
}
```

Will generate

```
module MyComponent (
  input      [7:0]    a,
  input      [7:0]    b,
  input      [7:0]    c,
  input      [7:0]    d
);
  wire       [7:0]    _zz_result;
  wire       [7:0]    _zz_result_1;
  wire       [7:0]    result;

  assign _zz_result = (_zz_result_1 + c);
  assign _zz_result_1 = (a + b);
  assign result = (_zz_result + d);

endmodule
```

Verilog long expression splitting

There is a instance of how a very long expression chain will be splited up by SpinalHDL :

```
class MyComponent extends Component {
  val conditions = in Vec(Bool, 64)
  val result = conditions.reduce(_ || _) // Do a logical or between all the␣
↪conditions elements
}
```

Will generate

```
module MyComponent (
  input               conditions_0,
  input               conditions_1,
  input               conditions_2,
  input               conditions_3,
  ...
  input               conditions_58,
  input               conditions_59,
  input               conditions_60,
  input               conditions_61,
  input               conditions_62,
  input               conditions_63
);
  wire                _zz_result;
  wire                _zz_result_1;
  wire                _zz_result_2;
  wire                result;

  assign _zz_result = ((((((((((((((_zz_result_1 || conditions_32) || conditions_
↪33) || conditions_34) || conditions_35) || conditions_36) || conditions_37) ||␣
↪conditions_38) || conditions_39) || conditions_40) || conditions_41) || conditions_
↪42) || conditions_43) || conditions_44) || conditions_45) || conditions_46) ||␣
↪conditions_47);
  assign _zz_result_1 = ((((((((((((((_zz_result_2 || conditions_16) || conditions_
↪17) || conditions_18) || conditions_19) || conditions_20) || conditions_21) ||␣
↪conditions_22) || conditions_23) || conditions_24) || conditions_25) || conditions_
↪26) || conditions_27) || conditions_28) || conditions_29) || conditions_30) ||␣
↪conditions_31);
  assign _zz_result_2 = ((((((((((((((conditions_0 || conditions_1) || conditions_2)␣
↪|| conditions_3) || conditions_4) || conditions_5) || conditions_6) || conditions_
↪7) || conditions_8) || conditions_9) || conditions_10) || conditions_11) ||␣
↪conditions_12) || conditions_13) || conditions_14) || conditions_15);
  assign result = ((((((((((((((_zz_result || conditions_48) || conditions_49) ||␣
↪conditions_50) || conditions_51) || conditions_52) || conditions_53) || conditions_
↪54) || conditions_55) || conditions_56) || conditions_57) || conditions_58) ||␣
↪conditions_59) || conditions_60) || conditions_61) || conditions_62) || conditions_
↪63);

endmodule
```

When statement condition

The *when(cond) { }* statements condition are generated into separated signals named *when_* + fileName + line. A similar thing will also be done for switch statements.

```
//In file Test.scala
class MyComponent extends Component {
  val value = in UInt(8 bits)
  val isZero = out(Bool())
  val counter = out(Reg(UInt(8 bits)))

  isZero := False
  when(value === 0){ //At line 117
    isZero := True
    counter := counter + 1
  }
}
```

Will generate

```
module MyComponent (
  input      [7:0]    value,
  output reg          isZero,
  output reg [7:0]    counter,
  input               clk,
  input               reset
);
  wire                when_Test_l117;

  always @ (*) begin
    isZero = 1'b0;
    if(when_Test_l117)begin
      isZero = 1'b1;
    end
  end

  assign when_Test_l117 = (value == 8'h0);
  always @ (posedge clk) begin
    if(when_Test_l117)begin
      counter <= (counter + 8'h01);
    end
  end
endmodule
```

In last resort

In last resort, if a signal has no name (anonymous signal), SpinalHDL will seek for a named signal which is driven by the anonymous signal, and use it as a name postfix :

```
class MyComponent extends Component {
  val enable = in Bool()
  val value = out UInt(8 bits)

  def count(cond : Bool): UInt = {
    val ret = Reg(UInt(8 bits)) // This register is not named (on purpose for the␣
↪example)
    when(cond){
```

(continues on next page)

(continued from previous page)

```
      ret := ret + 1
    }
    return ret
  }

  value := count(enable)
}
```

Will generate

```
module MyComponent (
  input               enable,
  output     [7:0]    value,
  input               clk,
  input               reset
);
  reg        [7:0]    _zz_value; //Name given to the register in last resort by␣
↪looking what was driven by it

  assign value = _zz_value;
  always @ (posedge clk) begin
    if(enable)begin
      _zz_value <= (_zz_value + 8'h01);
    end
  end
endmodule
```

This last resort naming skim isn't ideal in all cases, but can help out.

Note that signal starting with a underscore aren't stored in the Verilator waves (on purpose)

CHAPTER

SEVEN

SEMANTIC

7.1 Assignments

7.1.1 Assignments

There are multiple assignment operators:

Symbol	Description
:=	Standard assignment, equivalent to <= in VHDL/Verilog. The last assignment to a variable wins; the value is not updated until the next simulation delta cycle.
\=	Equivalent to := in VHDL and = in Verilog. The value is updated instantly in-place.
<>	Automatic connection between 2 signals or two bundles of the same type. Direction is inferred by using signal direction (in/out). (Similar behavior to :=)

```
// Because of hardware concurrency, `a` is always read as '1' by b and c
val a, b, c = UInt(4 bits)
a := 0
b := a
a := 1  // a := 1 "wins"
c := a

var x = UInt(4 bits)
val y, z = UInt(4 bits)
x := 0
y := x      // y read x with the value 0
x \= x + 1
z := x      // z read x with the value 1

// Automatic connection between two UART interfaces.
uartCtrl.io.uart <> io.uart
```

It is important to understand that in SpinalHDL, the nature of a signal (combinational/sequential) is defined in its declaration, not by the way it is assigned. All datatype instances will define a combinational signal, while a datatype instance wrapped with `Reg(...)` will define a sequential (registered) signal.

```
val a = UInt(4 bits) // Define a combinational signal
val b = Reg(UInt(4 bits)) // Define a registered signal
val c = Reg(UInt(4 bits)) init(0) // Define a registered signal which is set to 0␣
↪when a reset occurs
```

7.1.2 Width checking

SpinalHDL checks that the bit count of the left side and the right side of an assignment matches. There are multiple ways to adapt the width of a given BitVector (`Bits`, `UInt`, `SInt`):

Resizing techniques	Description
x := y.resized	Assign x with a resized copy of y, resize value is automatically inferred to match x
x := y.resize(newWidth)	Assign x with a resized copy of y, size is manually calculated

There is one case where Spinal automatically resizes a value:

Assignment	Problem	SpinalHDL action
myUIntOf_8bit := U(3)	U(3) creates an UInt of 2 bits, which doesn't match the left side (8 bits)	Because U(3) is a "weak" bit count inferred signal, SpinalHDL resizes it automatically

7.1.3 Combinatorial loops

SpinalHDL checks that there are no combinatorial loops (latches) in your design. If one is detected, it raises an error and SpinalHDL will print the path of the loop.

7.2 When/Switch/Mux

7.2.1 When

As in VHDL and Verilog, signals can be conditionally assigned when a specified condition is met:

```
when(cond1) {
  // Execute when cond1 is true
}.elsewhen(cond2) {
  // Execute when (not cond1) and cond2
}.otherwise {
  // Execute when (not cond1) and (not cond2)
}
```

7.2.2 Switch

As in VHDL and Verilog, signals can be conditionally assigned when a signal has a defined value:

```
switch(x) {
  is(value1) {
    // Execute when x === value1
  }
  is(value2) {
    // Execute when x === value2
  }
  default {
    // Execute if none of precedent conditions met
  }
}
```

7.2.3 Local declaration

It is possible to define new signals inside a when/switch statement:

```
val x, y = UInt(4 bits)
val a, b = UInt(4 bits)

when(cond) {
  val tmp = a + b
  x := tmp
  y := tmp + 1
} otherwise {
  x := 0
  y := 0
}
```

Note: SpinalHDL checks that signals defined inside a scope are only assigned inside that scope.

7.2.4 Mux

If you just need a `Mux` with a `Bool` selection signal, there are two equivalent syntaxes:

Syntax	Return	Description
Mux(cond, whenTrue, whenFalse)	T	Return `whenTrue` when `cond` is True, `whenFalse` otherwise
cond ? whenTrue | whenFalse	T	Return `whenTrue` when `cond` is True, `whenFalse` otherwise

```
val cond = Bool
val whenTrue, whenFalse = UInt(8 bits)
val muxOutput  = Mux(cond, whenTrue, whenFalse)
val muxOutput2 = cond ? whenTrue | whenFalse
```

7.2.5 Bitwise selection

A bitwise selection looks like the VHDL `when` syntax.

Example

```
val bitwiseSelect = UInt(2 bits)
val bitwiseResult = bitwiseSelect.mux(
  0 -> (io.src0 & io.src1),
  1 -> (io.src0 | io.src1),
  2 -> (io.src0 ^ io.src1),
  default -> (io.src0)
)
```

Also, if all possible values are covered in your mux, you can omit the default value:

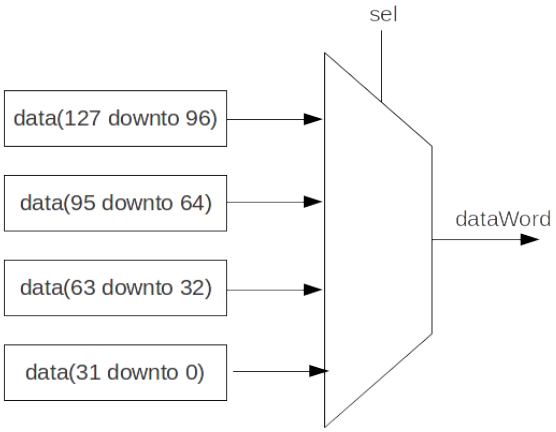
```
val bitwiseSelect = UInt(2 bits)
val bitwiseResult = bitwiseSelect.mux(
  0 -> (io.src0 & io.src1),
```

(continues on next page)

(continued from previous page)

```
  1 -> (io.src0 | io.src1),
  2 -> (io.src0 ^ io.src1),
  3 -> (io.src0)
)
```

`muxLists(...)` is another bitwise selection which takes a sequence of tuples as input. Below is an example of dividing a `Bits` of 128 bits into 32 bits:




```
val sel  = UInt(2 bits)
val data = Bits(128 bits)

// Dividing a wide Bits type into smaller chunks, using a mux:
val dataWord = sel.muxList(for (index <- 0 until 4) yield (index, data(index*32+32-1␣
↪downto index*32)))

// A shorter way to do the same thing:
val dataWord = data.subdivideIn(32 bits)(sel)
```

7.3 Rules

7.3.1 Introduction

The semantics behind SpinalHDL are important to learn, so that you understand what is really happening behind the scenes, and how to control it.

These semantics are defined by multiple rules:

- Signals and registers are operating concurrently with each other (parallel behavioral, as in VHDL and Verilog)
- An assignment to a combinational signal is like expressing a rule which is always true
- An assignment to a register is like expressing a rule which is applied on each cycle of its clock domain
- For each signal, the last valid assignment wins
- Each signal and register can be manipulated as an object during hardware elaboration in a OOP manner

7.3.2 Concurrency

The order in which you assign each combinational or registered signal has no behavioral impact.

For example, both of the following pieces of code are equivalent:

```
val a, b, c = UInt(8 bits) // Define 3 combinational signals
c := a + b  // c will be set to 7
b := 2      // b will be set to 2
a := b + 3  // a will be set to 5
```

This is equivalent to:

```
val a, b, c = UInt(8 bits) // Define 3 combinational signals
b := 2      // b will be set to 2
a := b + 3  // a will be set to 5
c := a + b  // c will be set to 7
```

More generally, when you use the `:=` assignment operator, it's like specifying a new rule for the left side signal/register.

7.3.3 Last valid assignment wins

If a combinational signal or register is assigned multiple times, the last valid one wins.

As an example:

```
val x, y = Bool()           // Define two combinational signals
val result = UInt(8 bits)   // Define a combinational signal

result := 1
when(x) {
  result := 2
  when(y) {
    result := 3
  }
}
```

This will produce the following truth table:

x	y	=>	result
False	False		1
False	True		1
True	False		2
True	True		3

7.3.4 Signal and register interactions with Scala (OOP reference + Functions)

In SpinalHDL, each hardware element is modeled by a class instance. This means you can manipulate instances by using their references, such as passing them as arguments to a function.

As an example, the following code implements a register which is incremented when `inc` is True and cleared when `clear` is True (`clear` has priority over `inc`) :

```
val inc, clear = Bool()          // Define two combinational signals/wires
val counter = Reg(UInt(8 bits))  // Define an 8 bit register
```

(continues on next page)

(continued from previous page)

```
when(inc) {
  counter := counter + 1
}
when(clear) {
  counter := 0    // If inc and clear are True, then this  assignment wins (Last␣
↪valid assignment rule)
}
```

You can implement exactly the same functionality by mixing the previous example with a function that assigns to counter:

```
val inc, clear = Bool()
val counter = Reg(UInt(8 bits))

def setCounter(value : UInt): Unit = {
  counter := value
}

when(inc) {
  setCounter(counter + 1)  // Set counter with counter + 1
}
when(clear) {
  counter := 0
}
```

You can also integrate the conditional check inside the function:

```
val inc, clear = Bool()
val counter = Reg(UInt(8 bits))

def setCounterWhen(cond : Bool,value : UInt): Unit = {
  when(cond) {
    counter := value
  }
}

setCounterWhen(cond = inc,   value = counter + 1)
setCounterWhen(cond = clear, value = 0)
```

And also specify what should be assigned to the function:

```
val inc, clear = Bool()
val counter = Reg(UInt(8 bits))

def setSomethingWhen(something : UInt, cond : Bool, value : UInt): Unit = {
  when(cond) {
    something := value
  }
}

setSomethingWhen(something = counter, cond = inc,   value = counter + 1)
setSomethingWhen(something = counter, cond = clear, value = 0)
```

All of the previous examples are strictly equivalent both in their generated RTL and also in the SpinalHDL compiler's perspective. This is because SpinalHDL only cares about the Scala runtime and the objects instantiated there, it doesn't care about the Scala syntax itself.

In other words, from a generated RTL generation / SpinalHDL perspective, when you use functions in Scala which

generate hardware, it is like the function was inlined. This is also true case for Scala loops, as they will appear in unrolled form in the generated RTL.

CHAPTER

EIGHT

SEQUENTIAL LOGIC

8.1 Registers

8.1.1 Introduction

Creating registers in SpinalHDL is very different than in VHDL or Verilog.

In Spinal, there are no process/always blocks. Registers are explicitly defined at declaration. This difference from traditional event-driven HDL has a big impact:

- You can assign registers and wires in the same scope, meaning the code doesn't need to be split between process/always blocks
- It make things much more flexible (see *Functions*)

Clocks and resets are handled separately, see the *Clock domain* chapter for details.

8.1.2 Instantiation

There are 4 ways to instantiate a register:

Syntax	Description
`Reg(type : Data)`	Register of the given type
`RegInit(resetValue : Data)`	Register loaded with the given `resetValue` when a reset occurs
`RegNext(nextValue : Data)`	Register that samples the given `nextValue` each cycle
`RegNextWhen(nextValue : Data, cond : Bool)`	Register that samples the given `nextValue` when a condition occurs

Here is an example declaring some registers:

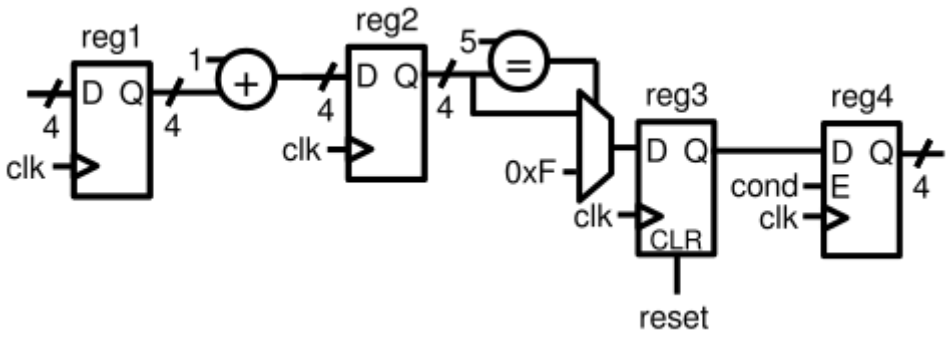
```
// UInt register of 4 bits
val reg1 = Reg(UInt(4 bit))

// Register that samples reg1 each cycle
val reg2 = RegNext(reg1 + 1)

// UInt register of 4 bits initialized with 0 when the reset occurs
val reg3 = RegInit(U"0000")
reg3 := reg2
when(reg2 === 5) {
  reg3 := 0xF
}
```

(continues on next page)

(continued from previous page)

```
// Register that samples reg3 when cond is True
val reg4 = RegNextWhen(reg3, cond)
```

The code above will infer the following logic:



Note: The `reg3` example above shows how you can assign the value of a `RegInit` register. It's possible to use the same syntax to assign to the other register types as well (`Reg`, `RegNext`, `RegNextWhen`). Just like in combinational assignments, the rule is 'Last assignment wins', but if no assignment is done, the register keeps its value.

Also, `RegNext` is an abstraction which is built over the `Reg` syntax. The two following sequences of code are strictly equivalent:

```
// Standard way
val something = Bool()
val value = Reg(Bool())
value := something

// Short way
val something = Bool()
val value = RegNext(something)
```

8.1.3 Reset value

In addition to the `RegInit(value : Data)` syntax which directly creates the register with a reset value, you can also set the reset value by calling the `init(value : Data)` function on the register.

```
// UInt register of 4 bits initialized with 0 when the reset occurs
val reg1 = Reg(UInt(4 bit)) init(0)
```

If you have a register containing a Bundle, you can use the `init` function on each element of the Bundle.

```
case class ValidRGB() extends Bundle{
  val valid   = Bool()
  val r, g, b = UInt(8 bits)
}

val reg = Reg(ValidRGB())
reg.valid init(False)  // Only the valid if that register bundle will have a reset␣
↪value.
```

8.1.4 Initialization value for simulation purposes

For registers that don't need a reset value in RTL, but need an initialization value for simulation (to avoid x-propagation), you can ask for a random initialization value by calling the `randBoot()` function.

```
// UInt register of 4 bits initialized with a random value
val reg1 = Reg(UInt(4 bit)) randBoot()
```

8.2 RAM/ROM

8.2.1 Syntax

To create a memory in SpinalHDL, the `Mem` class should be used. It allows you to define a memory and add read and write ports to it.

The following table shows how to instantiate a memory:

Syntax	Description
`Mem(type : Data, size : Int)`	Create a RAM
`Mem(type : Data, initialContent : Array[Data])`	Create a ROM. If your target is an FPGA, because the memory can be inferred as a block ram, you can still create write ports on it.

Note: If you want to define a ROM, elements of the `initialContent` array should only be literal values (no operator, no resize functions). There is an example *here*.

Note: To give a RAM initial values, you can also use the `init` function.

The following table show how to add access ports on a memory :

Syntax	Description	Return
mem(address) := data	Synchronous write	
mem(address)	Asynchronous read	T
mem.write(address data [enable] [mask])	Synchronous write with an optional mask. If no enable is specified, it's automatically inferred from the conditional scope where this function is called	
mem.readAsync(address [readUnderWrite])	Asynchronous read with an optional read-under-write policy	T
mem.readSync(address [enable] [readUnderWrite] [clockCrossing])	Synchronous read with an optional enable, read-under-write policy, and `clockCrossing` mode	T
mem.readWriteSync(address data enable write [mask] [readUnderWrite] [clockCrossing]	Infer a read/write port. `data` is written when `enable && write`. Return the read data, the read occurs when `enable` is true	T

Note: If for some reason you need a specific memory port which is not implemented in Spinal, you can always abstract over your memory by specifying a BlackBox for it.

Important: Memory ports in SpinalHDL are not inferred, but are explicitly defined. You should not use coding templates like in VHDL/Verilog to help the synthesis tool to infer memory.

Here is a example which infers a simple dual port ram (32 bits * 256):

```
val mem = Mem(Bits(32 bits), wordCount = 256)
mem.write(
  enable  = io.writeValid,
  address = io.writeAddress,
  data    = io.writeData
)

io.readData := mem.readSync(
  enable  = io.readValid,
  address = io.readAddress
)
```

8.2.2 Read-under-write policy

This policy specifies how a read is affected when a write occurs in the same cycle to the same address.

Kinds	Description
`dontCare`	Don't care about the read value when the case occurs
`readFirst`	The read will get the old value (before the write)
`writeFirst`	The read will get the new value (provided by the write)

Important: The generated VHDL/Verilog is always in the `readFirst` mode, which is compatible with `dontCare` but not with `writeFirst`. To generate a design that contains this kind of feature, you need to enable *automatic memory blackboxing*.

8.2.3 Mixed-width ram

You can specify ports that access the memory with a width that is a power of two fraction of the memory width using these functions:

Syntax	Description
mem.writeMixedWidth(address data [readUnderWrite])	Similar to `mem.write`
mem.readAsyncMixedWidth(address data [readUnderWrite])	Similar to `mem.readAsync`, but in place of returning the read value, it drives the signal/object given as the `data` argument
mem.readSyncMixedWidth(address data [enable] [readUnderWrite] [clockCrossing])	Similar to `mem.readSync`, but in place of returning the read value, it drives the signal/object given as the `data` argument
mem.readWriteSyncMixedWidth(address data enable write [mask] [readUnderWrite] [clockCrossing])	Equivalent to `mem.readWriteSync`

Important: As for read-under-write policy, to use this feature you need to enable *automatic memory blackboxing*,

because there is no universal VHDL/Verilog language template to infer mixed-width ram.

8.2.4 Automatic blackboxing

Because it's impossible to infer all ram kinds by using regular VHDL/Verilog, SpinalHDL integrates an optional automatic blackboxing system. This system looks at all memories present in your RTL netlist and replaces them with blackboxes. Then the generated code will rely on third party IP to provide the memory features, such as the read-during-write policy and mixed-width ports.

Here is an example of how to enable blackboxing of memories by default:

```
def main(args: Array[String]) {
  SpinalConfig()
    .addStandardMemBlackboxing(blackboxAll)
    .generateVhdl(new TopLevel)
}
```

If the standard blackboxing tools don't do enough for your design, do not hesitate to create a Github issue. There is also a way to create your own blackboxing tool.

Blackboxing policy

There are multiple policies that you can use to select which memory you want to blackbox and also what to do when the blackboxing is not feasible:

Kinds	Description
`blackboxAll`	Blackbox all memory. Throw an error on unblackboxable memory
`blackboxAllWhatsYouCan`	Blackbox all memory that is blackboxable
`blackboxRequestedAndUninferable`	Blackbox memory specified by the user and memory that is known to be uninferable (mixed-width, ...). Throw an error on unblackboxable memory
`blackboxOnlyIfRequested`	Blackbox memory specified by the user Throw an error on unblackboxable memory

To explicitly set a memory to be blackboxed, you can use its `generateAsBlackBox` function.

```
val mem = Mem(Rgb(rgbConfig), 1 << 16)
mem.generateAsBlackBox()
```

You can also define your own blackboxing policy by extending the `MemBlackboxingPolicy` class.

Standard memory blackboxes

Shown below are the VHDL definitions of the standard blackboxes used in SpinalHDL:

```
-- Simple asynchronous dual port (1 write port, 1 read port)
component Ram_1w_1ra is
  generic(
    wordCount : integer;
    wordWidth : integer;
    technology : string;
    readUnderWrite : string;
    wrAddressWidth : integer;
    wrDataWidth : integer;
    wrMaskWidth : integer;
    wrMaskEnable : boolean;
    rdAddressWidth : integer;
    rdDataWidth : integer
  );
  port(
    clk : in std_logic;
    wr_en : in std_logic;
    wr_mask : in std_logic_vector;
    wr_addr : in unsigned;
    wr_data : in std_logic_vector;
    rd_addr : in unsigned;
    rd_data : out std_logic_vector
  );
end component;

-- Simple synchronous dual port (1 write port, 1 read port)
component Ram_1w_1rs is
  generic(
    wordCount : integer;
    wordWidth : integer;
    clockCrossing : boolean;
    technology : string;
    readUnderWrite : string;
    wrAddressWidth : integer;
    wrDataWidth : integer;
    wrMaskWidth : integer;
    wrMaskEnable : boolean;
    rdAddressWidth : integer;
    rdDataWidth : integer;
    rdEnEnable : boolean
  );
  port(
    wr_clk : in std_logic;
    wr_en : in std_logic;
    wr_mask : in std_logic_vector;
    wr_addr : in unsigned;
    wr_data : in std_logic_vector;
    rd_clk : in std_logic;
    rd_en : in std_logic;
    rd_addr : in unsigned;
    rd_data : out std_logic_vector
  );
end component;
```

(continues on next page)

(continued from previous page)

```
-- Single port (1 readWrite port)
component Ram_1wrs is
  generic(
    wordCount : integer;
    wordWidth : integer;
    readUnderWrite : string;
    technology : string
  );
  port(
    clk : in std_logic;
    en : in std_logic;
    wr : in std_logic;
    addr : in unsigned;
    wrData : in std_logic_vector;
    rdData : out std_logic_vector
  );
end component;

--True dual port (2 readWrite port)
component Ram_2wrs is
  generic(
    wordCount : integer;
    wordWidth : integer;
    clockCrossing : boolean;
    technology : string;
    portA_readUnderWrite : string;
    portA_addressWidth : integer;
    portA_dataWidth : integer;
    portA_maskWidth : integer;
    portA_maskEnable : boolean;
    portB_readUnderWrite : string;
    portB_addressWidth : integer;
    portB_dataWidth : integer;
    portB_maskWidth : integer;
    portB_maskEnable : boolean
  );
  port(
    portA_clk : in std_logic;
    portA_en : in std_logic;
    portA_wr : in std_logic;
    portA_mask : in std_logic_vector;
    portA_addr : in unsigned;
    portA_wrData : in std_logic_vector;
    portA_rdData : out std_logic_vector;
    portB_clk : in std_logic;
    portB_en : in std_logic;
    portB_wr : in std_logic;
    portB_mask : in std_logic_vector;
    portB_addr : in unsigned;
    portB_wrData : in std_logic_vector;
    portB_rdData : out std_logic_vector
  );
end component;
```

As you can see, blackboxes have a technology parameter. To set it, you can use the `setTechnology` function on the corresponding memory. There are currently 4 kinds of technologies possible:

- auto
- ramBlock
- distributedLut
- registerFile

CHAPTER

NINE

DESIGN ERRORS

9.1 Assignment overlap

9.1.1 Introduction

SpinalHDL will check that no signal assignment completely erases a previous one.

9.1.2 Example

The following code

```
class TopLevel extends Component {
  val a = UInt(8 bits)
  a := 42
  a := 66 // Erase the a := 42 assignment
}
```

will throw the following error:

```
ASSIGNMENT OVERLAP completely the previous one of (toplevel/a :  UInt[8 bits])
  ***
  Source file location of the a := 66 assignment via the stack trace
  ***
```

A fix could be:

```
class TopLevel extends Component {
  val a = UInt(8 bits)
  a := 42
  when(something) {
    a := 66
  }
}
```

But in the case when you really want to override the previous assignment (as there are times when overriding makes sense), you can do the following:

```
class TopLevel extends Component {
  val a = UInt(8 bits)
  a := 42
  a.allowOverride
  a := 66
}
```

9.2 Clock crossing violation

9.2.1 Introduction

SpinalHDL will check that every register of your design only depends (through combinational logic paths) on registers which use the same or a synchronous clock domain.

9.2.2 Example

The following code:

```
class TopLevel extends Component {
  val clkA = ClockDomain.external("clkA")
  val clkB = ClockDomain.external("clkB")

  val regA = clkA(Reg(UInt(8 bits)))   // PlayDev.scala:834
  val regB = clkB(Reg(UInt(8 bits)))   // PlayDev.scala:835

  val tmp = regA + regA                // PlayDev.scala:838
  regB := tmp
}
```

will throw:

```
CLOCK CROSSING VIOLATION from (toplevel/regA :  UInt[8 bits]) to (toplevel/regB : ␣
↪UInt[8 bits]).
- Register declaration at
  ***
  Source file location of the toplevel/regA definition via the stack trace
  ***
- through
      >>> (toplevel/regA :  UInt[8 bits]) at ***(PlayDev.scala:834) >>>
      >>> (toplevel/tmp :  UInt[8 bits]) at ***(PlayDev.scala:838) >>>
      >>> (toplevel/regB :  UInt[8 bits]) at ***(PlayDev.scala:835) >>>
```

There are multiple possible fixes, listed below:

- *crossClockDomain tags*
- *setSyncronousWith method*
- *BufferCC type*

crossClockDomain tag

The `crossClockDomain` tag can be used to communicate "It's alright, don't panic about this specific clock crossing" to the SpinalHDL compiler.

```
class TopLevel extends Component {
  val clkA = ClockDomain.external("clkA")
  val clkB = ClockDomain.external("clkB")

  val regA = clkA(Reg(UInt(8 bits)))
  val regB = clkB(Reg(UInt(8 bits))).addTag(crossClockDomain)


  val tmp = regA + regA
```

(continues on next page)

(continued from previous page)

```
  regB := tmp
}
```

setSyncronousWith

You can also specify that two clock domains are synchronous together by using the `setSynchronousWith` method of one of the `ClockDomain` objects.

```
class TopLevel extends Component {
  val clkA = ClockDomain.external("clkA")
  val clkB = ClockDomain.external("clkB")
  clkB.setSyncronousWith(clkA)

  val regA = clkA(Reg(UInt(8 bits)))
  val regB = clkB(Reg(UInt(8 bits)))


  val tmp = regA + regA
  regB := tmp
}
```

BufferCC

When exchanging single-bit signals (such as `Bool` types), or Gray-coded values, you can use `BufferCC` to safely cross different `ClockDomain` regions.

> **Warning:** Do not use `BufferCC` with multi-bit signals, as there is a risk of corrupted reads on the receiving side if the clocks are asynchronous. See the *Clock Domains* page for more details.

```
class AsyncFifo extends Component {
   val popToPushGray = Bits(ptrWidth bits)
   val pushToPopGray = Bits(ptrWidth bits)

   val pushCC = new ClockingArea(pushClock) {
     val pushPtr     = Counter(depth << 1)
     val pushPtrGray = RegNext(toGray(pushPtr.valueNext)) init(0)
     val popPtrGray  = BufferCC(popToPushGray, B(0, ptrWidth bits))
     val full        = isFull(pushPtrGray, popPtrGray)
     ...
   }

   val popCC = new ClockingArea(popClock) {
     val popPtr      = Counter(depth << 1)
     val popPtrGray  = RegNext(toGray(popPtr.valueNext)) init(0)
     val pushPtrGray = BufferCC(pushToPopGray, B(0, ptrWidth bit))
     val empty       = isEmpty(popPtrGray, pushPtrGray)
     ...
   }
}
```

9.3 Combinatorial loop

9.3.1 Introduction

SpinalHDL will check that there are no combinatorial loops in the design.

9.3.2 Example

The following code:

```
class TopLevel extends Component {
  val a = UInt(8 bits) // PlayDev.scala line 831
  val b = UInt(8 bits) // PlayDev.scala line 832
  val c = UInt(8 bits)
  val d = UInt(8 bits)

  a := b
  b := c | d
  d := a
  c := 0
}
```

will throw :

```
COMBINATORIAL LOOP :
  Partial chain :
    >>> (toplevel/a :  UInt[8 bits]) at ***(PlayDev.scala:831) >>>
    >>> (toplevel/d :  UInt[8 bits]) at ***(PlayDev.scala:834) >>>
    >>> (toplevel/b :  UInt[8 bits]) at ***(PlayDev.scala:832) >>>
    >>> (toplevel/a :  UInt[8 bits]) at ***(PlayDev.scala:831) >>>

  Full chain :
    (toplevel/a :  UInt[8 bits])
    (toplevel/d :  UInt[8 bits])
    (UInt | UInt)[8 bits]
    (toplevel/b :  UInt[8 bits])
    (toplevel/a :  UInt[8 bits])
```

A possible fix could be:

```
class TopLevel extends Component {
  val a = UInt(8 bits) // PlayDev.scala line 831
  val b = UInt(8 bits) // PlayDev.scala line 832
  val c = UInt(8 bits)
  val d = UInt(8 bits)

  a := b
  b := c | d
  d := 42
  c := 0
}
```

9.3.3 False-positives

It should be said that SpinalHDL's algorithm to detect combinatorial loops can be pessimistic, and it may give false positives. If it is giving a false positive, you can manually disable loop checking on one signal of the loop like so:

```
class TopLevel extends Component {
  val a = UInt(8 bits)
  a := 0
  a(1) := a(0) // False positive because of this line
}
```

could be fixed by :

```
class TopLevel extends Component {
  val a = UInt(8 bits).noCombLoopCheck
  a := 0
  a(1) := a(0)
}
```

It should also be said that assignments such as `(a(1) := a(0))` can make some tools like Verilator unhappy. It may be better to use a `Vec(Bool, 8)` in this case.

9.4 Hierarchy violation

9.4.1 Introduction

SpinalHDL will check that signals are never accessed outside of the current component's scope.

The following signals can be read inside a component:

- All directionless signals defined in the current component
- All in/out/inout signals of the current component
- All in/out/inout signals of child components

In addition, the following signals can be assigned to inside of a component:

- All directionless signals defined in the current component
- All out/inout signals of the current component
- All in/inout signals of child components

If a `HIERARCHY VIOLATION` error appears, it means that one of the above rules was violated.

9.4.2 Example

The following code:

```
class TopLevel extends Component {
  val io = new Bundle {
    val a = in UInt(8 bits)
  }
  val tmp = U"x42"
  io.a := tmp
}
```

will throw:

```
HIERARCHY VIOLATION : (toplevel/io_a : in UInt[8 bits]) is driven by (toplevel/tmp : ␣
↪UInt[8 bits]), but isn't accessible in the toplevel component.
  ***
  Source file location of the `io.a := tmp` via the stack trace
  ***
```

A fix could be :

```
class TopLevel extends Component {
  val io = new Bundle {
    val a = out UInt(8 bits) // changed from in to out
  }
  val tmp = U"x42"
  io.a := tmp
}
```

9.5 Io bundle

9.5.1 Introduction

SpinalHDL will check that each `io` bundle contains only in/out/inout signals.

9.5.2 Example

The following code:

```
class TopLevel extends Component {
  val io = new Bundle {
    val a = UInt(8 bits)
  }
}
```

will throw:

```
IO BUNDLE ERROR : A direction less (toplevel/io_a :  UInt[8 bits]) signal was defined␣
↪into toplevel component's io bundle
  ***
  Source file location of the toplevel/io_a definition via the stack trace
  ***
```

A fix could be:

```
class TopLevel extends Component {
  val io = new Bundle {
    val a = in UInt(8 bits)
  }
}
```

But if for meta hardware description reasons you really want `io.a` to be directionless, you can do:

```
class TopLevel extends Component {
  val io = new Bundle {
    val a = UInt(8 bits)
  }
```

(continues on next page)

(continued from previous page)

```
  a.allowDirectionLessIo
}
```

9.6 Latch detected

9.6.1 Introduction

SpinalHDL will check that no combinational signals will infer a latch during synthesis. In other words, this is a check that no combinational signals are partially assigned.

9.6.2 Example

The following code:

```
class TopLevel extends Component {
  val cond = in(Bool)
  val a = UInt(8 bits)

  when(cond) {
    a := 42
  }
}
```

will throw:

```
LATCH DETECTED from the combinatorial signal (toplevel/a :  UInt[8 bits]), defined at
  ***
  Source file location of the toplevel/io_a definition via the stack trace
  ***
```

A fix could be:

```
class TopLevel extends Component {
  val cond = in(Bool)
  val a = UInt(8 bits)

  a := 0
  when(cond) {
    a := 42
  }
}
```

9.7 No driver on

9.7.1 Introduction

SpinalHDL will check that all combinational signals which have an impact on the design are assigned by something.

9.7.2 Example

The following code:

```
class TopLevel extends Component {
  val result = out(UInt(8 bits))
  val a = UInt(8 bits)
  result := a
}
```

will throw:

```
NO DRIVER ON (toplevel/a :  UInt[8 bits]), defined at
  ***
  Source file location of the toplevel/a definition via the stack trace
  ***
```

A fix could be:

```
class TopLevel extends Component {
  val result = out(UInt(8 bits))
  val a = UInt(8 bits)
  a := 42
  result := a
}
```

9.8 NullPointerException

9.8.1 Introduction

`NullPointerException` is a Scala runtime reported error which can happen when a variable is accessed before it has been initialized.

9.8.2 Example

The following code:

```
class TopLevel extends Component {
  a := 42
  val a = UInt(8 bits)
}
```

will throw:

```
Exception in thread "main" java.lang.NullPointerException
  ***
  Source file location of the a := 42 assignment via the stack trace
  ***
```

A fix could be:

```
class TopLevel extends Component {
  val a = UInt(8 bits)
  a := 42
}
```

Issue explanation

SpinalHDL is not a language, it is a Scala library, which means that it obeys the same rules as the Scala general purpose programming language.

When running the above SpinalHDL hardware description to generate the corresponding VHDL/Verilog RTL, the SpinalHDL hardware description will be executed as a Scala program, and `a` will be a null reference until the program executes `val a = UInt(8 bits)`, so trying to assign to it before then will result in a `NullPointerException`.

9.9 Register defined as component input

9.9.1 Introduction

In SpinalHDL, you are not allowed to define a component that has a register as an input. The reasoning behind this is to prevent surprises when the user tries to drive the inputs of child components with the registered signal. If a registered input is desired, you will need to declare the unregistered input in the `io` bundle, and register the signal in the body of the component.

9.9.2 Example

The following code :

```
class TopLevel extends Component {
  val io = new Bundle {
    val a = in(Reg(UInt(8 bits)))
  }
}
```

will throw:

```
REGISTER DEFINED AS COMPONENT INPUT : (toplevel/io_a : in UInt[8 bits]) is defined as␣
↪a registered input of the toplevel component, but isn't allowed.
  ***
  Source file location of the toplevel/io_a definition via the stack trace
  ***
```

A fix could be :

```
class TopLevel extends Component {
  val io = new Bundle {
    val a = in UInt(8 bits)
  }
}
```

If a registered `a` is required, it can be done like so:

```
class TopLevel extends Component {
  val io = new Bundle {
    val a = in UInt(8 bits)
  }
  val a = RegNext(io.a)
}
```

9.10 Scope violation

9.10.1 Introduction

SpinalHDL will check that there are no signals assigned outside the scope they are defined in. This error isn't easy to trigger as it requires some specific meta hardware description tricks.

9.10.2 Example

The following code:

```
class TopLevel extends Component {
  val cond = Bool()

  var tmp : UInt = null
  when(cond) {
    tmp = UInt(8 bits)
  }
  tmp := U"x42"
}
```

will throw:

```
SCOPE VIOLATION : (toplevel/tmp :  UInt[8 bits]) is assigned outside its declaration␣
↪scope at
  ***
  Source file location of the tmp := U"x42" via the stack trace
  ***
```

A fix could be:

```
class TopLevel extends Component {
  val cond = Bool()

  var tmp : UInt = UInt(8 bits)
  when(cond) {

  }
  tmp := U"x42"
}
```

9.11 Spinal can't clone class

9.11.1 Introduction

This error happens when SpinalHDL wants to create a new datatype instance via the `cloneOf` function but isn't able to do it. The reason for this is nearly always because it can't retrieve the construction parameters of a `Bundle`.

9.11.2 Example

The following code:

```
// cloneOf(this) isn't able to retrieve the width value that was used to construct␣
↪itself
class RGB(width : Int) extends Bundle {
  val r, g, b = UInt(width bits)
}

class TopLevel extends Component {
  val tmp = Stream(new RGB(8)) // Stream requires the capability to cloneOf(new␣
↪RGB(8))
}
```

will throw:

```
*** Spinal can't clone class spinal.tester.PlayDevMessages$RGB datatype
*** You have two way to solve that :
*** In place to declare a "class Bundle(args){}", create a "case class Bundle(args){}"
*** Or override by your self the bundle clone function
  ***
  Source file location of the RGB class definition via the stack trace
  ***
```

A fix could be:

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

class TopLevel extends Component {
  val tmp = Stream(RGB(8))
}
```

9.12 Unassigned register

9.12.1 Introduction

SpinalHDL will check that all registers which impact the design have been assigned somewhere.

9.12.2 Example

The following code:

```
class TopLevel extends Component {
  val result = out(UInt(8 bits))
  val a = Reg(UInt(8 bits))
  result := a
}
```

will throw:

```
UNASSIGNED REGISTER (toplevel/a :  UInt[8 bits]), defined at
  ***
```

(continues on next page)

(continued from previous page)

```
  Source file location of the toplevel/a definition via the stack trace
  ***
```

A fix could be:

```
class TopLevel extends Component {
  val result = out(UInt(8 bits))
  val a = Reg(UInt(8 bits))
  a := 42
  result := a
}
```

9.12.3 Register with only init

In some cases, because of the design parameterization, it could make sense to generate a register which has no assignment but only an `init` statement.

```
class TopLevel extends Component {
  val result = out(UInt(8 bits))
  val a = Reg(UInt(8 bits)) init(42)

  if(something)
    a := somethingElse
  result := a
}
```

will throw:

```
UNASSIGNED REGISTER (toplevel/a :  UInt[8 bits]), defined at
  ***
  Source file location of the toplevel/a definition via the stack trace
  ***
```

To fix it, you can ask SpinalHDL to transform the register into a combinational one if no assignment is present but it has an `init` statement:

```
class TopLevel extends Component {
  val result = out(UInt(8 bits))
  val a = Reg(UInt(8 bits)).init(42).allowUnsetRegToAvoidLatch

  if(something)
    a := somethingElse
  result := a
}
```

9.13 Unreachable is statement

9.13.1 Introduction

SpinalHDL will check to ensure that all `is` statements in a `switch` are reachable.

9.13.2 Example

The following code:

```
class TopLevel extends Component {
  val sel = UInt(2 bits)
  val result = UInt(4 bits)
  switch(sel) {
    is(0){ result := 4 }
    is(1){ result := 6 }
    is(2){ result := 8 }
    is(3){ result := 9 }
    is(0){ result := 2 } // Duplicated is statement!
  }
}
```

will throw:

```
UNREACHABLE IS STATEMENT in the switch statement at
  ***
  Source file location of the is statement definition via the stack trace
  ***
```

A fix could be:

```
class TopLevel extends Component {
  val sel = UInt(2 bits)
  val result = UInt(4 bits)
  switch(sel) {
    is(0){ result := 4 }
    is(1){ result := 6 }
    is(2){ result := 8 }
    is(3){ result := 9 }
  }
}
```

9.14 Width mismatch

9.14.1 Introduction

SpinalHDL will check that operators and signals on the left and right side of assignments have the same widths.

9.14.2 Assignment example

The following code:

```
class TopLevel extends Component {
  val a = UInt(8 bits)
  val b = UInt(4 bits)
  b := a
}
```

will throw:

```
WIDTH MISMATCH on (toplevel/b :  UInt[4 bits]) := (toplevel/a :  UInt[8 bits]) at
  ***
  Source file location of the OR operator via the stack trace
  ***
```

A fix could be:

```
class TopLevel extends Component {
  val a = UInt(8 bits)
  val b = UInt(4 bits)
  b := a.resized
}
```

9.14.3 Operator example

The following code:

```
class TopLevel extends Component {
  val a = UInt(8 bits)
  val b = UInt(4 bits)
  val result = a | b
}
```

will throw:

```
WIDTH MISMATCH on (UInt | UInt)[8 bits]
- Left  operand : (toplevel/a :  UInt[8 bits])
- Right operand : (toplevel/b :  UInt[4 bits])
  at
  ***
  Source file location of the OR operator via the stack trace
  ***
```

A fix could be:

```
class TopLevel extends Component {
  val a = UInt(8 bits)
  val b = UInt(4 bits)
  val result = a | (b.resized)
}
```

9.15 Introduction

The SpinalHDL compiler will perform many checks on your design to be sure that the generated VHDL/Verilog will be safe for simulation and synthesis. Basically, it should not be possible to generate a broken VHDL/Verilog design. Below is a non-exhaustive list of SpinalHDL checks:

- Assignment overlapping
- Clock crossing
- Hierarchy violation
- Combinatorial loops
- Latches
- Undriven signals
- Width mismatch
- Unreachable switch statements

On each SpinalHDL error report, you will find a stack trace, which can be useful to accurately find out where the design error is. These design checks may look like overkill at first glance, but they becomes invaluable as soon as you start to move away from the traditional way of doing hardware description.

CHAPTER

TEN

OTHER LANGUAGE FEATURES

10.1 Utils

10.1.1 General

Many tools and utilities are present in *spinal.lib* but some are already present in the SpinalHDL Core.

Syntax	Return	Description
`widthOf(x : BitVector)`	Int	Return the width of a Bits/UInt/SInt signal
`log2Up(x : BigInt)`	Int	Return the number of bits needed to represent `x` states
`isPow2(x : BigInt)`	Boolean	Return true if `x` is a power of two
`roundUp(that : BigInt, by : BigInt)`	BigInt	Return the first `by` multiply from `that` (included)
`Cat(x : Data*)`	Bits	Concatenate all arguments, the first in MSB, the last in LSB

10.1.2 Cloning hardware datatypes

You can clone a given hardware data type by using the `cloneOf(x)` function. It will return a new instance of the same Scala type and parameters.

For example:

```
def plusOne(value : UInt) : UInt = {
  // Will recreate a UInt with the same width than ``value``
  val temp = cloneOf(value)
  temp := value + 1
  return temp
}

// treePlusOne will become a 8 bits value
val treePlusOne = plusOne(U(3, 8 bits))
```

You can get more information about how hardware data types are managed on the *Hardware types page*.

Note: If you use the `cloneOf` function on a `Bundle`, this `Bundle` should be a `case class` or should override the clone function internally.

10.1.3 Passing a datatype as construction parameter

Many pieces of reusable hardware need to be parameterized by some data type. For example if you want to define a FIFO or a shift register, you need a parameter to specify which kind of payload you want for the component.

There are two similar ways to do this.

The old way

A good example of the old way to do this is in this definition of a `ShiftRegister` component:

```
case class ShiftRegister[T <: Data](dataType: T, depth: Int) extends Component {
  val io = new Bundle {
    val input  = in (cloneOf(dataType))
    val output = out(cloneOf(dataType))
  }
  // ...
}
```

And here is how you can instantiate the component:

```
val shiftReg = ShiftRegister(Bits(32 bits), depth = 8)
```

As you can see, the raw hardware type is directly passed as a construction parameter. Then each time you want to create an new instance of that kind of hardware data type, you need to use the `cloneOf(...)` function. Doing things this way is not super safe as it's easy to forget to use `cloneOf`.

The safe way

An example of the safe way to pass a data type parameter is as follows:

```
case class ShiftRegister[T <: Data](dataType: HardType[T], depth: Int) extends␣
↪Component {
  val io = new Bundle {
    val input  = in (dataType())
    val output = out(dataType())
  }
  // ...
}
```

And here is how you instantiate the component (exactly the same as before):

```
val shiftReg = ShiftRegister(Bits(32 bits), depth = 8)
```

Notice how the example above uses a `HardType` wrapper around the raw data type `T`, which is a "blueprint" definition of a hardware data type. This way of doing things is easier to use than the "old way", because to create a new instance of the hardware data type you only need to call the `apply` function of that `HardType` (or in other words, just add parentheses after the parameter).

Additionally, this mechanism is completely transparent from the point of view of the user, as a hardware data type can be implicitly converted into a `HardType`.

10.1.4 Frequency and time

SpinalHDL has a dedicated syntax to define frequency and time values:

```
val frequency = 100 MHz
val timeoutLimit = 3 ms
val period = 100 us

val periodCycles = frequency * period
val timeoutCycles = frequency * timeoutLimit
```

For time definitions you can use following postfixes to get a `TimeNumber`:
`fs`, `ps`, `ns`, `us`, `ms`, `sec`, `mn`, `hr`

For time definitions you can use following postfixes to get a `HertzNumber`:
`Hz`, `KHz`, `MHz`, `GHz`, `THz`

`TimeNumber` and `HertzNumber` are based on the `PhysicalNumber` class which uses the Scala `BigDecimal` type to store numbers.

10.2 Assertions

In addition to Scala run-time assertions, you can add hardware assertions using the following syntax:

`assert(assertion : Bool, message : String = null, severity: AssertNodeSeverity = Error)`

Severity levels are:

Name	Description
NOTE	Used to report an informative message
WARNING	Used to report an unusual case
ERROR	Used to report an situation that should not happen
FAILURE	Used to report a fatal situation and close the simulation

One practical example could be to check that the `valid` signal of a handshake protocol never drops when `ready` is low:

```
class TopLevel extends Component {
  val valid = RegInit(False)
  val ready = in Bool

  when(ready) {
    valid := False
  }
  // some logic

  assert(
    assertion = !(valid.fall && !ready),
    message   = "Valid dropped when ready was low",
    severity  = ERROR
  )
}
```

10.3 Report

You can add debugging in RTL for simulation, using the following syntax:

```
object Enum extends SpinalEnum{
    val MIAOU, RAWRR = newElement()
}

class TopLevel extends Component {
    val a = Enum.RAWRR()
    val b = U(0x42)
    val c = out(Enum.RAWRR())
    val d = out (U(0x42))
    report(Seq("miaou ", a, b, c, d))
}
```

It will generate the following Verilog code for example:

```
$display("NOTE miaou %s%x%s%x", a_string, b, c_string, d);
```

Since SpinalHDL 1.4.4, the following syntax is also supported:

```
report(L"miaou $a $b $c $d")
```

10.4 Formal

10.4.1 General

There is limited support for SystemVerilog Assertions (SVA).

You can add formal statements (assume, assert, etc.) in the `Component` definition, like in the example below:

```
class TopLevel extends Component {
  val io = new Bundle {
    val ready = in Bool()
    val valid = out Bool()
  }
  val valid = RegInit(False)

 when(io.ready) {
   valid := False
 }
 io.valid <> valid
 // some logic

 import spinal.core.GenerationFlags._
 import spinal.core.Formal._

 GenerationFlags.formal {
   when(initstate()) {
     assume(clockDomain.isResetActive)
     assume(io.ready === False)
   }.otherwise {
     assert(!(valid.fall && !io.ready))
   }
```

(continues on next page)

(continued from previous page)

```
 }
}
```

To generate a design which includes the formal statements you can use `includeFormal`:

```
object MyToplevelSystemVerilogWithFormal {
 def main(args: Array[String]) {
   val config = SpinalConfig(defaultConfigForClockDomains =␣
↪ClockDomainConfig(resetKind=SYNC, resetActiveLevel=HIGH))
    config.includeFormal.generateSystemVerilog(new TopLevel())
  }
}
```

10.4.2 Supported features

Syntax	Returns	Creates in SystemVerilog
`assert()`		`assert()`
`cover()`		`cover()`
`past(that : T, delay : Int)` `past(that : T)`	T	`past(that)`
`rose(that : Bool)`	Bool	`rose(that)`
`fell(that : Bool)`	Bool	fell(that)
`changed(that : Bool)`	Bool	`changed(that)`
`stable(that : Bool)`	Bool	`stable(that)`
`initstate()`	Bool	`$initstate()`

10.4.3 Limitations

No support for unclocked assertions. Everything that is described in `GenerationFlags.formal` will be generated in a clocked process.

10.5 Analog and inout

10.5.1 Introduction

You can define native tristate signals by using the `Analog/inout` features. These features were added for the following reasons:

- Being able to add native tristate signals to the toplevel (it avoids having to manually wrap them with some hand-written VHDL/Verilog).
- Allowing the definition of blackboxes which contain `inout` pins.
- Being able to connect a blackbox's `inout` pin through the hierarchy to a toplevel `inout` pin.

As those features were only added for convenience, please do not try other fancy stuff with tristate logic just yet.

If you want to model a component like a memory-mapped GPIO peripheral, please use the *TriState/TriStateArray* bundles from the Spinal standard library, which abstract over the true nature of tristate drivers.

10.5.2 Analog

`Analog` is the keyword which allows a signal to be defined as something analog, which in the digital world could mean `0`, `1`, or `Z` (the disconnected, high-impedance state).

For instance:

```
case class SdramInterface(g : SdramLayout) extends Bundle {
  val DQ    = Analog(Bits(g.dataWidth bits)) // Bidirectional data bus
  val DQM   = Bits(g.bytePerWord bits)
  val ADDR  = Bits(g.chipAddressWidth bits)
  val BA    = Bits(g.bankWidth bits)
  val CKE, CSn, CASn, RASn, WEn  = Bool
}
```

10.5.3 inout

`inout` is the keyword which allows you to set an `Analog` signal as a bidirectional (both "in" and "out") signal.

For instance:

```
case class SdramInterface(g : SdramLayout) extends Bundle with IMasterSlave {
  val DQ    = Analog(Bits(g.dataWidth bits)) // Bidirectional data bus
  val DQM   = Bits(g.bytePerWord bits)
  val ADDR  = Bits(g.chipAddressWidth bits)
  val BA    = Bits(g.bankWidth bits)
  val CKE, CSn, CASn, RASn, WEn  = Bool

  override def asMaster() : Unit = {
    out(ADDR, BA, CASn, CKE, CSn, DQM, RASn, WEn)
    inout(DQ) // Set the Analog DQ as an inout signal of the component
  }
}
```

10.5.4 InOutWrapper

`InOutWrapper` is a tool which allows you to transform all `master TriState/TriStateArray/ReadableOpenDrain` bundles of a component into native `inout(Analog(...))` signals. It allows you to keep your hardware description free of any `Analog/inout` things, and then transform the toplevel to make it synthesis ready.

For instance:

```
case class Apb3Gpio(gpioWidth : Int) extends Component {
  val io = new Bundle{
    val gpio = master(TriStateArray(gpioWidth bits))
    val apb  = slave(Apb3(Apb3Gpio.getApb3Config()))
  }
  ...
}

SpinalVhdl(InOutWrapper(Apb3Gpio(32)))
```

Will generate:

```
entity Apb3Gpio is
  port(
    io_gpio : inout std_logic_vector(31 downto 0); -- This io_gpio was originally a␣
```

(continues on next page)

(continued from previous page)

```
↪TriStateArray Bundle
    io_apb_PADDR : in unsigned(3 downto 0);
    io_apb_PSEL : in std_logic_vector(0 downto 0);
    io_apb_PENABLE : in std_logic;
    io_apb_PREADY : out std_logic;
    io_apb_PWRITE : in std_logic;
    io_apb_PWDATA : in std_logic_vector(31 downto 0);
    io_apb_PRDATA : out std_logic_vector(31 downto 0);
    io_apb_PSLVERROR : out std_logic;
    clk : in std_logic;
    reset : in std_logic
  );
end Apb3Gpio;
```

Instead of:

```
entity Apb3Gpio is
  port(
    io_gpio_read : in std_logic_vector(31 downto 0);
    io_gpio_write : out std_logic_vector(31 downto 0);
    io_gpio_writeEnable : out std_logic_vector(31 downto 0);
    io_apb_PADDR : in unsigned(3 downto 0);
    io_apb_PSEL : in std_logic_vector(0 downto 0);
    io_apb_PENABLE : in std_logic;
    io_apb_PREADY : out std_logic;
    io_apb_PWRITE : in std_logic;
    io_apb_PWDATA : in std_logic_vector(31 downto 0);
    io_apb_PRDATA : out std_logic_vector(31 downto 0);
    io_apb_PSLVERROR : out std_logic;
    clk : in std_logic;
    reset : in std_logic
  );
end Apb3Gpio;
```

10.5.5 Manually driving Analog bundles

If an `Analog` bundle is not driven, it will default to being high-Z. Therefore to manually implement a tristate driver (in case the `InOutWrapper` type can't be used for some reason) you have to conditionally drive the signal.

To manually connect a `TriState` signal to an `Analog` bundle:

```
case class Example extends Component {
  val io = new Bundle {
    val tri = slave(TriState(Bits(16 bit)))
    val analog = inout Analog(Bits(16 bit))
  }
  tri.read := analog
  when(tri.writeEnable) { analog := tri.write }
}
```

10.6 VHDL and Verilog generation

10.6.1 Generate VHDL and Verilog from a SpinalHDL Component

To generate the VHDL from a SpinalHDL component you just need to call `SpinalVhdl(new YourComponent)` in a Scala `main`.

Generating Verilog is exactly the same, but with `SpinalVerilog` in place of `SpinalVHDL`

```
import spinal.core._

// A simple component definition.
class MyTopLevel extends Component {
  // Define some input/output signals. Bundle like a VHDL record or a Verilog struct.
  val io = new Bundle {
    val a = in  Bool()
    val b = in  Bool()
    val c = out Bool()
  }

  // Define some asynchronous logic.
  io.c := io.a & io.b
}

// This is the main function that generates the VHDL and the Verilog corresponding to␣
↪MyTopLevel.
object MyMain {
  def main(args: Array[String]) {
    SpinalVhdl(new MyTopLevel)
    SpinalVerilog(new MyTopLevel)
  }
}
```

Important: `SpinalVhdl` and `SpinalVerilog` may need to create multiple instances of your component class, therefore the first argument is not a `Component` reference, but a function that returns a new component.

Important: The `SpinalVerilog` implementation began the 5th of June, 2016. This backend successfully passes the same regression tests as the VHDL one (RISCV CPU, Multicore and pipelined mandelbrot, UART RX/TX, Single clock fifo, Dual clock fifo, Gray counter, ...).

If you have any issues with this new backend, please make a Github issue describing the problem.

Parametrization from Scala

Argument name	Type	Default	Description
mode	SpinalMode	null	Set the SpinalHDL hdl generation mode. Can be set to VHDL or Verilog
defaultConfigForClockDomains	ClockDomainConfig	RisingEdgeClock AsynchronousReset ResetActiveHigh ClockEnableActiveHigh	Set the clock configuration that will be used as the default value for all new ClockDomain.
onlyStdLogicVectorAtTopLevelIo	Boolean	false	Change all unsigned/signed toplevel io into std_logic_vector.
defaultClockDomainFrequency	IClockDomainFrequency	UnknownFrequency	Default clock frequency.
targetDirectory	String	Current directory	Directory where files are generated.

And this is the syntax to specify them:

```
SpinalConfig(mode=VHDL, targetDirectory="temp/myDesign").generate(new UartCtrl)

// Or for Verilog in a more scalable formatting:
SpinalConfig(
  mode=Verilog,
  targetDirectory="temp/myDesign"
).generate(new UartCtrl)
```

Parametrization from shell

You can also specify generation parameters by using command line arguments.

```
def main(args: Array[String]): Unit = {
  SpinalConfig.shell(args)(new UartCtrl)
}
```

The syntax for command line arguments is:

```
Usage: SpinalCore [options]

  --vhdl
        Select the VHDL mode
  --verilog
        Select the Verilog mode
  -d | --debug
        Enter in debug mode directly
  -o <value> | --targetDirectory <value>
        Set the target directory
```

10.6.2 Generated VHDL and Verilog

How a SpinalHDL RTL description is translated into VHDL and Verilog is important:

- Names in Scala are preserved in VHDL and Verilog.
- `Component` hierarchy in Scala is preserved in VHDL and Verilog.
- `when` statements in Scala are emitted as if statements in VHDL and Verilog.
- `switch` statements in Scala are emitted as case statements in VHDL and Verilog in all standard cases.

Organization

When you use the VHDL generator, all modules are generated into a single file which contain three sections:

1. A package that contains the definition of all Enums
2. A package that contains functions used by the architectural elements
3. All components needed by your design

When you use the Verilog generation, all modules are generated into a single file which contains two sections:

1. All enumeration definitions used
2. All modules needed by your design

Combinational logic

Scala:

```
class TopLevel extends Component {
  val io = new Bundle {
    val cond           = in  Bool()
    val value          = in  UInt(4 bits)
    val withoutProcess = out UInt(4 bits)
    val withProcess    = out UInt(4 bits)
  }
  io.withoutProcess := io.value
  io.withProcess := 0
  when(io.cond) {
    switch(io.value) {
      is(U"0000") {
        io.withProcess := 8
      }
      is(U"0001") {
        io.withProcess := 9
      }
      default {
        io.withProcess := io.value+1
      }
    }
  }
}
```

VHDL:

```
entity TopLevel is
  port(
    io_cond : in std_logic;
```

(continues on next page)

(continued from previous page)

```
    io_value : in unsigned(3 downto 0);
    io_withoutProcess : out unsigned(3 downto 0);
    io_withProcess : out unsigned(3 downto 0)
  );
end TopLevel;

architecture arch of TopLevel is
begin
  io_withoutProcess <= io_value;
  process(io_cond,io_value)
  begin
    io_withProcess <= pkg_unsigned("0000");
    if io_cond = '1' then
      case io_value is
        when pkg_unsigned("0000") =>
          io_withProcess <= pkg_unsigned("1000");
        when pkg_unsigned("0001") =>
          io_withProcess <= pkg_unsigned("1001");
        when others =>
          io_withProcess <= (io_value + pkg_unsigned("0001"));
      end case;
    end if;
  end process;
end arch;
```

Sequential logic

Scala:

```
class TopLevel extends Component {
  val io = new Bundle {
    val cond   = in Bool()
    val value  = in UInt (4 bit)
    val resultA = out UInt(4 bit)
    val resultB = out UInt(4 bit)
  }

  val regWithReset = Reg(UInt(4 bits)) init(0)
  val regWithoutReset = Reg(UInt(4 bits))

  regWithReset := io.value
  regWithoutReset := 0
  when(io.cond) {
    regWithoutReset := io.value
  }

  io.resultA := regWithReset
  io.resultB := regWithoutReset
}
```

VHDL:

```
entity TopLevel is
  port(
    io_cond : in std_logic;
```

(continues on next page)

(continued from previous page)

```
    io_value : in unsigned(3 downto 0);
    io_resultA : out unsigned(3 downto 0);
    io_resultB : out unsigned(3 downto 0);
    clk : in std_logic;
    reset : in std_logic
  );
end TopLevel;

architecture arch of TopLevel is

  signal regWithReset : unsigned(3 downto 0);
  signal regWithoutReset : unsigned(3 downto 0);
begin
  io_resultA <= regWithReset;
  io_resultB <= regWithoutReset;
  process(clk,reset)
  begin
    if reset = '1' then
      regWithReset <= pkg_unsigned("0000");
    elsif rising_edge(clk) then
      regWithReset <= io_value;
    end if;
  end process;

  process(clk)
  begin
    if rising_edge(clk) then
      regWithoutReset <= pkg_unsigned("0000");
      if io_cond = '1' then
        regWithoutReset <= io_value;
      end if;
    end if;
  end process;
end arch;
```

10.6.3 VHDL and Verilog attributes

In some situations, it is useful to give attributes for some signals in a design to modify how they are synthesized.

To do that, you can call the following functions on any signals or memories in the design:

Syntax	Description
`addAttribute(name)`	Add a boolean attribute with the given `name` set to true
`addAttribute(name, value)`	Add a string attribute with the given `name` set to `value`

Example:

```
val pcPlus4 = pc + 4
pcPlus4.addAttribute("keep")
```

Produced declaration in VHDL:

```
attribute keep : boolean;
signal pcPlus4 : unsigned(31 downto 0);
attribute keep of pcPlus4: signal is true;
```

Produced declaration in Verilog:

```
(* keep *) wire [31:0] pcPlus4;
```

10.7 Introduction

10.7.1 Introduction

The core of the language defines the syntax for many features:

- Types / Literals
- Register / Clock domains
- Component / Area
- RAM / ROM
- When / Switch / Mux
- BlackBox (to integrate VHDL or Verilog IPs inside Spinal)
- SpinalHDL to VHDL converter

Then, by using these features, you can define digital hardware, and also build powerful libraries and abstractions. It's one of the major advantages of SpinalHDL over other commonly used HDLs, because you can extend the language without having knowledge about the compiler.

One good example of this is the *SpinalHDL lib* which adds many utilities, tools, buses, and methodologies.

To use features introduced in the following chapter you need to `import spinal.core._` in your sources.

CHAPTER

ELEVEN

LIBRARIES

11.1 Utils

Some utils are also present in *spinal.core*

11.1.1 State less utilities

Syntax	Return	Description
toGray(x : UInt)	Bits	Return the gray value converted from `x` (UInt)
fromGray(x : Bits)	UInt	Return the UInt value converted value from `x` (gray)
Reverse(x : T)	T	Flip all bits (lsb + n -> msb - n)
OHToUInt(x : Seq[Bool]) OHToUInt(x : BitVector)	UInt	Return the index of the single bit set (one hot) in `x`
CountOne(x : Seq[Bool]) CountOne(x : BitVector)	UInt	Return the number of bit set in `x`
MajorityVote(x : Seq[Bool]) MajorityVote(x : BitVector)	Bool	Return True if the number of bit set is > x.size / 2
EndiannessSwap(that: T[, base:BitCount])	T	Big-Endian <-> Little-Endian
OHMasking.first(x : Bits)	Bits	Apply a mask on x to only keep the first bit set
OHMasking.last(x : Bits)	Bits	Apply a mask on x to only keep the last bit set
OHMasking.roundRobin(requests : Bits, ohPriority : Bits)	Bits	Apply a mask on x to only keep the bit set from `requests`. it start looking in `requests` from the `ohPriority` position. For example if `requests` is "1001" and `ohPriority` is "0010", the `roundRobin` function will start looking in *requests* from its second bit and will return "1000".
MuxOH (oneHot : IndexedSeq[Bool], inputs : Iterable[T])	T	Returns the muxed `T` from the `inputs` based on the `oneHot` vector.

11.1.2 State full utilities

Syntax	Return	Description
Delay(that: T, cycleCount: Int)	T	Return `that` delayed by `cycleCount` cycles
History(that: T, length: Int[,when : Bool])	List[T]	Return a Vec of `length` elements The first element is `that`, the last one is `that` delayed by `length`-1 The internal shift register sample when `when` is asserted
BufferCC(input : T)	T	Return the input signal synchronized with the current clock domain by using 2 flip flop

Counter

The Counter tool can be used to easily instanciate an hardware counter.

Instanciation syntax	Notes
Counter(start: BigInt, end: BigInt[, inc : Bool])	
Counter(range : Range[, inc : Bool])	Compatible with the `x to y` `x until y` syntaxes
Counter(stateCount: BigInt[, inc : Bool])	Start at zero and finish at `stateCount - 1`
Counter(bitCount: BitCount[, inc : Bool])	Start at zero and finish at `(1 << bitCount) - 1`

There is an example of different syntaxes which could be used with the Counter tool

```
val counter = Counter(2 to 9)  //Create a counter of 10 states (2 to 9)
counter.clear()              //When called it ask to reset the counter.
counter.increment()          //When called it ask to increment the counter.
counter.value                //current value
counter.valueNext            //Next value
counter.willOverflow         //Flag that indicate if the counter overflow this cycle
counter.willOverflowIfInc    //Flag that indicate if the counter overflow this cycle if␣
↪an increment is done
when(counter === 5){ ... }
```

When a `Counter` overflow its end value, it restart to its start value.

Note: Currently, only up counter are supported.

Timeout

The Timeout tool can be used to easily instanciate an hardware timeout.

Instanciation syntax	Notes
Timeout(cycles : BigInt)	Tick after `cycles` clocks
Timeout(time : TimeNumber)	Tick after a `time` duration
Timeout(frequency : HertzNumber)	Tick at an `frequency` rate

There is an example of different syntaxes which could be used with the Counter tool

```
val timeout = Timeout(10 ms)  //Timeout who tick after 10 ms
when(timeout){                //Check if the timeout has tick
    timeout.clear()           //Ask the timeout to clear its flag
}
```

Note: If you instanciate an `Timeout` with an time or frequency setup, the implicit `ClockDomain` should have an frequency setting.

ResetCtrl

The ResetCtrl provide some utilities to manage resets.

asyncAssertSyncDeassert

You can filter an asynchronous reset by using an asynchronously asserted synchronously deaserted logic. To do it you can use the `ResetCtrl.asyncAssertSyncDeassert` function which will return you the filtred value.

Argument name	Type	Description
input	Bool	Signal that should be filtered
clockDomain	ClockDomain	ClockDomain which will use the filtered value
inputPolarity	Polarity	HIGH/LOW (default=HIGH)
outputPolarity	Polarity	HIGH/LOW (default=clockDomain.config.resetActiveLevel)
bufferDepth	Int	Number of register stages used to avoid metastability (default=2)

There is also an `ResetCtrl.asyncAssertSyncDeassertDrive` version of tool which directly assign the `clockDomain` reset with the filtred value.

11.1.3 Special utilities

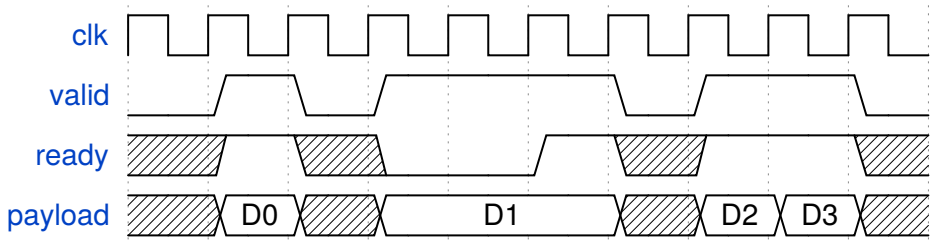
Syntax	Return	Description
LatencyAnalysis(paths : Node*)	Int	Return the shortest path,in therm of cycle, that travel through all nodes, from the first one to the last one

11.2 Stream

11.2.1 Specification

The Stream interface is a simple handshake protocol to carry payload.
It could be used for example to push and pop elements into a FIFO, send requests to a UART controller, etc.

Signal	Type	Driver	Description	Don't care when
valid	Bool	Master	When high => payload present on the interface	
ready	Bool	Slave	When low => transaction are not consumed by the slave	valid is low
payload	T	Master	Content of the transaction	valid is low



There is some examples of usage in SpinalHDL :

```
class StreamFifo[T <: Data](dataType: T, depth: Int) extends Component {
  val io = new Bundle {
    val push = slave Stream (dataType)
    val pop = master Stream (dataType)
  }
  ...
}

class StreamArbiter[T <: Data](dataType: T,portCount: Int) extends Component {
  val io = new Bundle {
    val inputs = Vec(slave Stream (dataType),portCount)
    val output = master Stream (dataType)
  }
  ...
}
```

Note: Each slave can or can't allow the payload to change when valid is high and ready is low. For examples:

- An priority arbiter without lock logic can switch from one input to the other (which will change the payload).
- An UART controller could directly use the write port to drive UART pins and only consume the transaction at the end of the transmission. Be careful with that.

11.2.2 Semantics

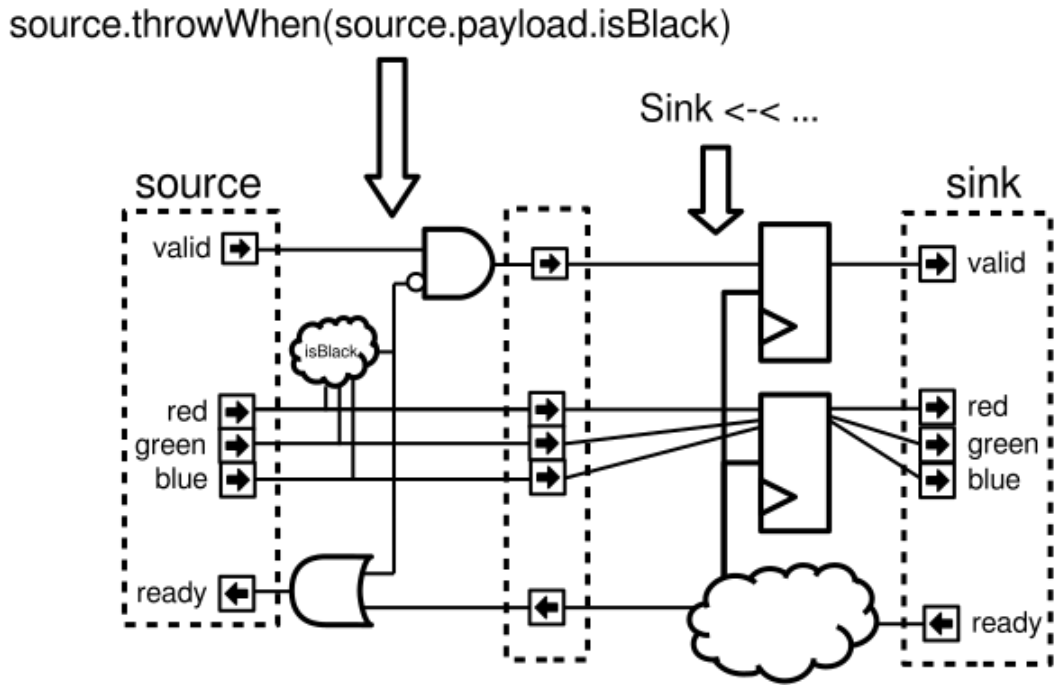
When manually reading/driving the signals of a Stream keep in mind that:

- After being asserted, `valid` may only be deasserted once the current payload was acknowleged. This means `valid` can only toggle to 0 the cycle after a the slave did a read by asserting `ready`.
- In contrast to that `ready` may change at any time.
- A transfer is only done on cycles where both `valid` and `ready` are asserted.
- `valid` of a Stream must not depend on `ready` in a combinatorial way and any path between the two must be registered.
- It is recommended that `valid` does not depend on `ready` at all.

11.2.3 Functions

Syntax	Description	Return	Latency
Stream(type : Data)	Create a Stream of a given type	Stream[T]	
master/slave Stream(type : Data)	Create a Stream of a given type Initialized with corresponding in/out setup	Stream[T]	
x.fire	Return True when a transaction is consumed on the bus (valid && ready)	Bool	
x.isStall	Return True when a transaction is stall on the bus (valid && ! ready)	Bool	
x.queue(size:Int)	Return a Stream connected to x through a FIFO	Stream[T]	2
x.m2sPipe() x.stage()	Return a Stream drived by x through a register stage that cut valid/payload paths Cost = (payload width + 1) flop flop	Stream[T]	1
x.s2mPipe()	Return a Stream drived by x ready paths is cut by a register stage Cost = payload width * (mux2 + 1 flip flop)	Stream[T]	0
x.halfPipe()	Return a Stream drived by x valid/ready/payload paths are cut by some register Cost = (payload width + 2) flip flop, bandwidth divided by two	Stream[T]	1
x << y y >> x	Connect y to x		0
x <-< y y >-> x	Connect y to x through a m2sPipe		1
x </< y y >/> x	Connect y to x through a s2mPipe		0
x <-/< y y >/-> x	Connect y to x through s2mPipe().m2sPipe() Which imply no combinatorial path between x and y		1
x.haltWhen(cond : Bool)	Return a Stream connected to x Halted when cond is true	Stream[T]	0
x.throwWhen(cond : Bool)		Stream[T]	0

The following code will create this logic :




```
case class RGB(channelWidth : Int) extends Bundle{
  val red   = UInt(channelWidth bit)
  val green = UInt(channelWidth bit)
  val blue  = UInt(channelWidth bit)

  def isBlack : Bool = red === 0 && green === 0 && blue === 0
}

val source = Stream(RGB(8))
val sink   = Stream(RGB(8))
sink <-< source.throwWhen(source.payload.isBlack)
```

11.2.4 Utils

There is many utils that you can use in your design in conjunction with the Stream bus, this chapter will document them.

StreamFifo

On each stream you can call the .queue(size) to get a buffered stream. But you can also instantiate the FIFO component itself :

```
val streamA,streamB = Stream(Bits(8 bits))
//...
val myFifo = StreamFifo(
  dataType = Bits(8 bits),
  depth    = 128
)
myFifo.io.push << streamA
myFifo.io.pop  >> streamB
```

parameter name	Type	Description
dataType	T	Payload data type
depth	Int	Size of the memory used to store elements

io name	Type	Description
push	Stream[T]	Used to push elements
pop	Stream[T]	Used to pop elements
flush	Bool	Used to remove all elements inside the FIFO
occupancy	UInt of log2Up(depth + 1) bits	Indicate the internal memory occupancy

StreamFifoCC

You can instanciate the dual clock domain version of the fifo the following way :

```
val clockA = ClockDomain(???)
val clockB = ClockDomain(???)
val streamA,streamB = Stream(Bits(8 bits))
//...
val myFifo = StreamFifoCC(
  dataType  = Bits(8 bits),
  depth     = 128,
  pushClock = clockA,
  popClock  = clockB
)
myFifo.io.push << streamA
myFifo.io.pop  >> streamB
```

parameter name	Type	Description
dataType	T	Payload data type
depth	Int	Size of the memory used to store elements
pushClock	ClockDomain	Clock domain used by the push side
popClock	ClockDomain	Clock domain used by the pop side

io name	Type	Description
push	Stream[T]	Used to push elements
pop	Stream[T]	Used to pop elements
pushOccupancy	UInt of log2Up(depth + 1) bits	Indicate the internal memory occupancy (from the push side perspective)
popOccupancy	UInt of log2Up(depth + 1) bits	Indicate the internal memory occupancy (from the pop side perspective)

StreamCCByToggle

Component that connects Streams across clock domains based on toggling signals.
This way of implementing a cross clock domain bridge is characterized by a small area usage but also a low bandwidth.

```
val clockA = ClockDomain(???)
val clockB = ClockDomain(???)
val streamA,streamB = Stream(Bits(8 bits))
//...
val bridge = StreamCCByToggle(
  dataType    = Bits(8 bits),
  inputClock  = clockA,
  outputClock = clockB
)
bridge.io.input  << streamA
bridge.io.output >> streamB
```

parameter name	Type	Description
dataType	T	Payload data type
inputClock	ClockDomain	Clock domain used by the push side
outputClock	ClockDomain	Clock domain used by the pop side

io name	Type	Description
input	Stream[T]	Used to push elements
output	Stream[T]	Used to pop elements

Alternatively you can also use a this shorter syntax which directly return you the cross clocked stream:

```
val clockA = ClockDomain(???)
val clockB = ClockDomain(???)
val streamA = Stream(Bits(8 bits))
val streamB = StreamCCByToggle(
  input       = streamA,
  inputClock  = clockA,
  outputClock = clockB
)
```

StreamArbiter

When you have multiple Streams and you want to arbitrate them to drive a single one, you can use the StreamArbiterFactory.

```
val streamA, streamB, streamC = Stream(Bits(8 bits))
val arbitredABC = StreamArbiterFactory.roundRobin.onArgs(streamA, streamB, streamC)

val streamD, streamE, streamF = Stream(Bits(8 bits))
val arbitredDEF = StreamArbiterFactory.lowerFirst.noLock.onArgs(streamD, streamE,␣
↪streamF)
```

Arbitration functions	Description
lowerFirst	Lower port have priority over higher port
roundRobin	Fair round robin arbitration
sequentialOrder	Could be used to retrieve transaction in a sequancial order First transaction should come from port zero, then from port one, ...

Lock functions	Description
noLock	The port selection could change every cycle, even if the transaction on the selected port is not consumed.
transactionLock	The port selection is locked until the transaction on the selected port is consumed.
fragmentLock	Could be used to arbitrate Stream[Flow[T]]. In this mode, the port selection is locked until the selected port finish is burst (last=True).

Generation functions	Return
on(inputs : Seq[Stream[T]])	Stream[T]
onArgs(inputs : Stream[T]*)	Stream[T]

StreamJoin

This utile takes multiple input streams and wait until all of them fire before letting all of them through.

```
val cmdJoin = Stream(Cmd())
cmdJoin.arbitrationFrom(StreamJoin.arg(cmdABuffer, cmdBBuffer))
```

StreamFork

A StreamFork will clone each incoming data to all its output streams. If synchronous is true, all output streams will always fire together, which means that the stream will halt until all output streams are ready. If synchronous is false, output streams may be ready one at a time, at the cost of an additional flip flop (1 bit per output). The input stream will block until all output streams have processed each item regardlessly.

```
val inputStream = Stream(Bits(8 bits))
val (outputstream1, outputstream2) = StreamFork2(inputStream, synchronous=false)
```

or

```
val inputStream = Stream(Bits(8 bits))
val outputStreams = StreamFork(inputStream,portCount=2, synchronous=true)
```

StreamDispatcherSequencial

This util take its input stream and routes it to `outputCount` stream in a sequential order.

```
val inputStream = Stream(Bits(8 bits))
val dispatchedStreams = StreamDispatcherSequencial(
  input = inputStream,
  outputCount = 3
)
```

11.3 Flow

11.3.1 Specification

The Flow interface is a simple valid/payload protocol which mean the slave can't halt the bus.
It could be used, for example, to represent data coming from an UART controller, requests to write an on-chip memory, etc.

Signal	Type	Driver	Description	Don't care when
valid	Bool	Master	When high => payload present on the interface	
payload	T	Master	Content of the transaction	valid is low

11.3.2 Functions

Syntax	Description	Return	Latency
Flow(type : Data)	Create a Flow of a given type	Flow[T]	
master/slave Flow(type : Data)	Create a Flow of a given type Initialized with corresponding in/out setup	Flow[T]	
x.m2sPipe()	Return a Flow drived by x through a register stage that cut valid/payload paths	Flow[T]	1
x << y y >> x	Connect y to x		0
x <-< y y >-> x	Connect y to x through a m2sPipe		1
x.throwWhen(cond : Bool)	Return a Flow connected to x When cond is high, transaction are dropped	Flow[T]	0
x.toReg()	Return a register which is loaded with `payload` when valid is high	T	

11.4 Fragment

11.4.1 Specification

The `Fragment` bundle is the concept of transmitting a "big" thing by using multiple "small" fragments. For examples :

- A picture transmitted with width*height transaction on a `Stream[Fragment[Pixel]]`
- An UART packet received from an controller without flow control could be transmitted on a `Flow[Fragment[Bits]]`
- An AXI read burst could be carried by an `Stream[Fragment[AxiReadResponse]]`

Signals defined by the `Fragment` bundle are :

Signal	Type	Driver	Description
fragment	T	Master	The "payload" of the current transaction
last	Bool	Master	High when the fragment is the last of the current packet

As you can see with this specification and precedent example, the `Fragment` concept doesn't specify how transaction are transmitted (You can use Stream,Flow or any other communication protocol). It only add enough information (`last`) to know if the current transaction is the first one, the last one or one in the middle of a given packet.

Note: The protocol didn't carry a 'first' bit because it can be generated at any place by doing 'RegNextWhen(bus.last, bus.fire) init(True)'

11.4.2 Functions

For `Stream[Fragment[T]]` and `Flow[Fragment[T]]`, following function are presents :

Syntax	Return	Description
x.first	Bool	Return True when the next or the current transaction is/would be the first of a packet
x.tail	Bool	Return True when the next or the current transaction is/would be not the first of a packet
x.isFirst	Bool	Return True when an transaction is present and is the first of a packet
x.isTail	Bool	Return True when an transaction is present and is the not the first/last of a packet
x.isLast	Bool	Return True when an transaction is present and is the last of a packet

For `Stream[Fragment[T]]`, following function are also accessible :

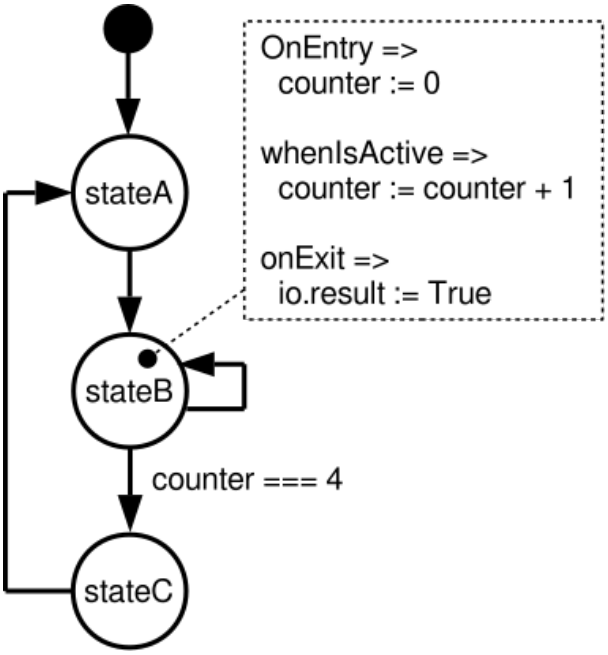
Syntax	Return	Description
x.insertHeader(header : T)	Stream[Fragment[T]]	Add the `header` to each packet on `x` and return the resulting bus

11.5 State machine

11.5.1 Introduction

In SpinalHDL you can define your state machine like in VHDL/Verilog, by using enumerations and switch cases statements. But in SpinalHDL you can also use a dedicated syntax.

The following state machine is implemented in following examples :



Style A :

```
import spinal.lib.fsm._

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

  val fsm = new StateMachine{
    val counter = Reg(UInt(8 bits)) init (0)
    io.result := False

    val stateA : State = new State with EntryPoint{
      whenIsActive (goto(stateB))
    }
    val stateB : State = new State{
      onEntry(counter := 0)
      whenIsActive {
        counter := counter + 1
        when(counter === 4){
          goto(stateC)
        }
      }
      onExit(io.result := True)
    }
    val stateC : State = new State{
      whenIsActive (goto(stateA))
    }
  }
}
```

Style B :

```
import spinal.lib.fsm._

class TopLevel extends Component {
```

(continues on next page)

(continued from previous page)

```
  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(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))
  }
}
```

11.5.2 StateMachine

StateMachine is the base class that will manage the logic of your FSM.

```
val myFsm = new StateMachine{
  // Here will come states definition
}
```

The StateMachine class also provide some utils :

Name	Return	Description
isActive(state)	Bool	Return True when the state machine is in the given state
isEntering(state)	Bool	Return True when the state machine is entering the given state

11.5.3 States

There is multiple kinds of states that you can use.

- State (the base one)
- StateDelay
- StateFsm
- StateParallelFsm

In each of them you have access the following utilities :

Name	Description

onEntry{
yourStatements
}
	yourStatements is executed the cycle before entering the state

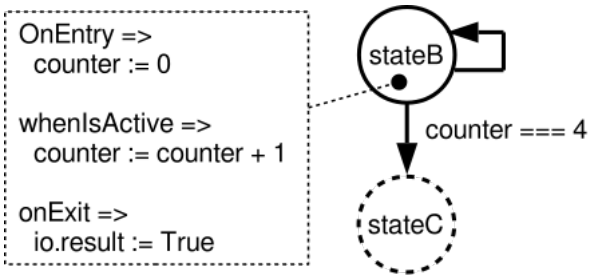
onExit{
yourStatements
}
	yourStatements is executed when the state machine will be in another state the next cycle

whenIsActive{
yourStatements
}
	yourStatements is executed when the state machine is in the state

whenIsNext{
yourStatements
}
	yourStatements is executed when the state machine will be in the state the next cycle

goto(nextState)	Set the state of the state machine by nextState
exit()	Set the state of the state machine to the boot one

For example, the following state could be defined in SpinalHDL by using the following syntax :



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

You can also define your state as the entry point of the state machine by extends the EntryPoint trait.

```
val stateA: State = new State with EntryPoint {
  whenIsActive {
    goto(stateB)
  }
}
```

StateDelay

StateDelay allow you to create a state which wait a fixed number of cycles before executing statments in your `whenCompleted{...}`. The standard way to write it is :

```
val stateG : State = new StateDelay(cyclesCount=40){
  whenCompleted{
    goto(stateH)
  }
}
```

But you can also write it like that :

```
val stateG : State = new StateDelay(40){whenCompleted(goto(stateH))}
```

StateFsm

StateFsm Allow you to describe a state which contains a nested state machine. When the nested state machine is done, your statments in `whenCompleted{...}` are executed.

There is an example of StateFsm definition :

```
val stateC = new StateFsm(fsm=internalFsm()){
  whenCompleted{
    goto(stateD)
  }
}
```

As you can see in the precedent code, it use a `internalFsm` function to create the inner state machine. There is an example of definition bellow :

```
def internalFsm() = new StateMachine {
  val counter = Reg(UInt(8 bits)) init (0)

  val stateA: State = new State with EntryPoint {
    whenIsActive {
      goto(stateB)
```

(continues on next page)

(continued from previous page)

```
    }
  }

  val stateB: State = new State {
    onEntry (counter := 0)
    whenIsActive {
      when(counter === 4) {
        exit()
      }
      counter := counter + 1
    }
  }
}
```

In the precedent example, the `exit()` call will make the state machine jump to the boot state (a internal hidden state). This notify the StateFsm about the completion of the inner state machine.

StateParallelFsm

This state is able to handle multiple nested state machines. When all nested state machine are done, your statments in `whenCompleted{...}` are executed.

There is an example of declaration :

```
val stateD = new StateParallelFsm (internalFsmA(), internalFsmB()){
  whenCompleted{
    goto(stateE)
  }
}
```

11.6 VexRiscv (RV32IM CPU)

VexRiscv is an fpga friendly RISC-V ISA CPU implementation with following features :

- RV32IM instruction set
- Pipelined on 5 stages (Fetch, Decode, Execute, Memory, WriteBack)
- 1.44 DMIPS/Mhz when all features are enabled
- Optimized for FPGA
- Optional MUL/DIV extension
- Optional instruction and data caches
- Optional MMU
- Optional debug extension allowing eclipse debugging via an GDB >> openOCD >> JTAG connection
- Optional interrupts and exception handling with the Machine and the User mode from the riscv-privileged-v1.9.1 spec.
- Two implementation of shift instructions, Single cycle / shiftNumber cycles
- Each stage could have bypass or interlock hazard logic
- FreeRTOS port https://github.com/Dolu1990/FreeRTOS-RISCV

Much more information there : https://github.com/SpinalHDL/VexRiscv

11.7 Bus Slave Factory

11.7.1 Introduction

In many situation it's needed to implement a bus register bank. The `BusSlaveFactory` is a tool that provide an abstract and smooth way to define them.

To see capabilities of the tool, an simple example use the Apb3SlaveFactory variation to implement an *memory mapped UART*. There is also another example with an *Timer* which contain a memory mapping function.

You can find more documentation about the internal implementation of the `BusSlaveFactory` tool *there*

11.7.2 Functionality

Currently there is three implementation of the `BusSlaveFactory` tool : APB3, AXI-lite 3 and Avalon.
Each implementation of that tool take as argument one instance of the corresponding bus and then offer following functions to map your hardware into the memory mapping :

Name	Return	Description
busDataWidth	Int	Return the data width of the bus
read(that,address,bitOffset)		When the bus read the `address`, fill the response with `that` at `bitOffset`
write(that,address,bitOffset)		When the bus write the `address`, assign `that` with bus's data from `bitOffset`
onWrite(address)(doThat)		Call `doThat` when a write transaction occur on `address`
onRead(address)(doThat)		Call `doThat` when a read transaction occur on `address`
nonStopWrite(that,bitOffset)		Permanently assign `that` by the bus write data from `bitOffset`
readAndWrite(that,address,bitOffset)		Make `that` readable and writable at `address` and placed at `bitOffset` in the word
readMultiWord(that,address)		Create the memory mapping to read `that` from 'address'. If `that` is bigger than one word it extends the register on followings addresses
writeMultiWord(that,address)		Create the memory mapping to write `that` at 'address'. If `that` is bigger than one word it extends the register on followings addresses
createWriteOnly(dataType,address,bitOffset)	T	Create a write only register of type `dataType` at `address` and placed at `bitOffset` in the word
createReadWrite(dataType,address,bitOffset)	T	Create a read write register of type `dataType` at `address` and placed at `bitOffset` in the word
createAndDriveFlow(dataType,address,bitOffset)	Flow[T]	Create a writable Flow register of type `dataType` at `address` and placed at `bitOffset` in the word
drive(that,address,bitOffset)		Drive `that` with a register writable at `address` placed at `bitOffset` in the word
driveAndRead(that,address,bitOffset)		Drive `that` with a register writable and readable at `address` placed at `bitOffset` in the word
driveFlow(that,address,bitOffset)		Emit on `that` a transaction when a write happen at `address` by using data placed at `bitOffset` in the word
readStreamNonBlocking(that, address, validBitOffset, payloadBitOffset)		Read `that` and consume the transaction when a read happen at `address`. valid <= validBitOffset bit payload <= payloadBitOffset+widthOf(payload) downto `payloadBitOffset`
doBitsAccumulationAndClearOnRead(that, address, bitOffset)		Instanciate an internal register which at each cycle do : reg := reg | that Then when a read occur, the register is cleared. This register is readable at `address` and placed at `bitOffset` in the word

11.8 Fiber framework

Currently in developpement.

The Fiber to run the hardware elaboration in a out of order manner, a bit similarly to Makefile, where you can define rules and dependencies which will then be solved when you run a make command. It is very similar to the Scala Future feature.

Such framework complexify simple things but provide some strong feature for complex cases :

- You can define things before even knowing all their requirements, ex : instanciating a interruption controller, before knowing how many lines of interrupt you need
- Abstract/lazy/partial SoC architecture definition allowing the creation of SoC template for further specialisations
- Automatic requirements negotiation between multiple agents in a decentralized way, ex : between masters and slaves of a memory bus

The framework is mainly composed of :

- Handle[T], which can be used later to store a value of type T.
- handle.load which allow to set the value of a handle (will reschedule all tasks waiting on it)
- handle.get, which return the value of the given handle. Will block the task execution if that handle isn't loaded yet
- Handle{ code }, which fork a new task which will execute the given code. The result of that code will be loaded into the Handle
- soon(handle), which allow the current task to announce that soon it will load that handle with a value (used to track which handle will

Warning, this is realy not usual RTL description and aim large system generation. It is currently used as toplevel integration tool in SaxonSoC.

11.8.1 Simple dummy example

There is a simple example :

```
import spinal.core.fiber._

// Create two empty Handles
val a, b = Handle[Int]

// Create a Handle which will be loaded asynchronously by the given body result
val calculator = Handle {
    a.get + b.get // .get will block until they are loaded
}

// Same as above
val printer = Handle {
    println(s"a + b = ${calculator.get}") // .get is blocking until the calculator␣
↪body is done
}

// Synchronously load a and b, this will unblock a.get and b.get
a.load(3)
b.load(4)
```

Its runtime will be :

- create a and b
- fork the calculator task, but is blocked when executing a.get
- fork the printer task, but is blocked when executing calculator.get
- load a and b, which reschedule the calculator task (as it was waiting on a)
- calculator do its a + b sum, and load its Handle with that result, which reschedule the printer task
- printer task print its stuff
- everything done

So, the main point of that example is to show that we kind of overcome the sequential execution of things, as a and b are loaded after the definition of the calculator.

11.8.2 Handle[T]

Handle[T] are a bit like scala's Future[T], they allow to talk about something before it is even existing, and wait on it.

```
val x,y = Handle[Int]
val xPlus2 : Handle[Int] = x.produce(x.get + 2) //x.produce can be used to generate a␣
↪new Handle when x is loaded
val xPlus3 : Handle[Int] = x.derivate(_ + 3)    //x.derivate is as x.produce, but␣
↪also provide the x.get as argument of the lambda function
x.load(3) //x will now contain the value 3
```

soon(handle)

In order to maintain a proper graph of dependencies between tasks and Handle, a task can specify in advance that it will load a given handle. This is very usefull in case of a generation starvation/deadlock for SpinalHDL to report accuratly where is the issue.

11.9 Bus

11.9.1 AHB-Lite3

Configuration and instanciation

First each time you want to create a AHB-Lite3 bus, you will need a configuration object. This configuration object is an `AhbLite3Config` and has following arguments :

Parameter name	Type	Default	Description
addressWidth	Int		Width of HADDR (byte granularity)
dataWidth	Int		Width of HWDATA and HRDATA

There is in short how the AHB-Lite3 bus is defined in the SpinalHDL library :

```
case class AhbLite3(config: AhbLite3Config) extends Bundle with IMasterSlave{
  //  Address and control
  val HADDR = UInt(config.addressWidth bits)
  val HSEL = Bool()
  val HREADY = Bool()
  val HWRITE = Bool()
```

(continues on next page)

(continued from previous page)

```
  val HSIZE = Bits(3 bits)
  val HBURST = Bits(3 bits)
  val HPROT = Bits(4 bits)
  val HTRANS = Bits(2 bits)
  val HMASTLOCK = Bool()

  //  Data
  val HWDATA = Bits(config.dataWidth bits)
  val HRDATA = Bits(config.dataWidth bits)

  //  Transfer response
  val HREADYOUT = Bool()
  val HRESP = Bool()

  override def asMaster(): Unit = {
    out(HADDR,HWRITE,HSIZE,HBURST,HPROT,HTRANS,HMASTLOCK,HWDATA,HREADY,HSEL)
    in(HREADYOUT,HRESP,HRDATA)
  }
}
```

There is a short example of usage :

```
val ahbConfig = AhbLite3Config(
  addressWidth = 12,
  dataWidth    = 32
)
val ahbX = AhbLite3(ahbConfig)
val ahbY = AhbLite3(ahbConfig)

when(ahbY.HSEL){
  //...
}
```

Variations

There is an AhbLite3Master variation. The only difference is the absence of the `HREADYOUT` signal. This variation should only be used by masters while the interconnect and slaves use `AhbLite3`.

11.9.2 Apb3

Introduction

The AMBA3-APB bus is commonly used to interface low bandwidth peripherals.

Configuration and instanciation

First each time you want to create a APB3 bus, you will need a configuration object. This configuration object is an `Apb3Config` and has following arguments :

Parameter name	Type	Default	Description
addressWidth	Int		Width of PADDR (byte granularity)
dataWidth	Int		Width of PWDATA and PRDATA
selWidth	Int	1	With of PSEL
useSlaveError	Boolean	false	Specify the presence of PSLVERROR

There is in short how the APB3 bus is defined in the SpinalHDL library :

```
case class Apb3(config: Apb3Config) extends Bundle with IMasterSlave {
  val PADDR      = UInt(config.addressWidth bit)
  val PSEL       = Bits(config.selWidth bits)
  val PENABLE    = Bool()
  val PREADY     = Bool()
  val PWRITE     = Bool()
  val PWDATA     = Bits(config.dataWidth bit)
  val PRDATA     = Bits(config.dataWidth bit)
  val PSLVERROR  = if(config.useSlaveError) Bool() else null
  //...
}
```

There is a short example of usage :

```
val apbConfig = Apb3Config(
  addressWidth = 12,
  dataWidth    = 32
)
val apbX = Apb3(apbConfig)
val apbY = Apb3(apbConfig)

when(apbY.PENABLE){
  //...
}
```

Functions and operators

Name	Return	Description
X >> Y		Connect X to Y. Address of Y could be smaller than the one of X
X << Y		Do the reverse of the >> operator

11.9.3 Axi4

Introduction

The AXI4 is a high bandwidth bus defined by ARM.

Configuration and instanciation

First each time you want to create a AXI4 bus, you will need a configuration object. This configuration object is an `Axi4Config` and has following arguments :

Note : useXXX specify if the bus has XXX signal present.

Parameter name	Type	Default
addressWidth	Int	
dataWidth	Int	
idWidth	Int	
userWidth	Int	
useId	Boolean	true
useRegion	Boolean	true
useBurst	Boolean	true
useLock	Boolean	true
useCache	Boolean	true
useSize	Boolean	true
useQos	Boolean	true
useLen	Boolean	true
useLast	Boolean	true
useResp	Boolean	true
useProt	Boolean	true
useStrb	Boolean	true
useUser	Boolean	false

There is in short how the AXI4 bus is defined in the SpinalHDL library :

```
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)
  }
}
```

There is a short example of usage :

```
val axiConfig = Axi4Config(
  addressWidth = 32,
  dataWidth    = 32,
  idWidth      = 4
)
val axiX = Axi4(axiConfig)
val axiY = Axi4(axiConfig)

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

Variations

There is 3 other variation of the Axi4 bus :

Type	Description
Axi4ReadOnly	Only AR and R channels are present
Axi4WriteOnly	Only AW, W and B channels are present
Axi4Shared	This variation is a library initiative. It use 4 channels, W, B ,R and also a new one which is named AWR. The AWR channel can be used to transmit AR and AW transactions. To dissociate them, a signal `write` is present. The advantage of this Axi4Shared variation is to use less area, especially in the interconnect.

Functions and operators

Name	Return	Description
X >> Y		Connect X to Y. Able infer default values as specified in the AXI4 specification, and also to adapt some width in a safe manner.
X << Y		Do the reverse of the >> operator
X.toWriteOnly	Axi4WriteOnly	Return an Axi4WriteOnly bus drive by X
X.toReadOnly	Axi4ReadOnly	Return an Axi4ReadOnly bus drive by X

11.9.4 AvalonMM

Introduction

The AvalonMM bus fit very well in FPGA. It is very flexible :

- Able of the same simplicity than APB
- Better for than AHB in many application that need bandwidth because AvalonMM has a mode that decouple read response from commands (reduce latency read latency impact).
- Less performance than AXI but use much less area (Read and write command use the same handshake channel. The master don't need to store address of pending request to avoid Read/Write hazard)

Configuration and instanciation

The `AvalonMM` Bundle has a construction argument `AvalonMMConfig`. Because of the flexible nature of the Avalon bus, the `AvalonMMConfig` as many configuration elements. For more information the Avalon spec could be find there.

```
case class AvalonMMConfig( addressWidth : Int,
                           dataWidth : Int,
                           burstCountWidth : Int,
                           useByteEnable : Boolean,
                           useDebugAccess : Boolean,
                           useRead : Boolean,
                           useWrite : Boolean,
                           useResponse : Boolean,
                           useLock : Boolean,
```

(continues on next page)

(continued from previous page)

```
                        useWaitRequestn : Boolean,
                        useReadDataValid : Boolean,
                        useBurstCount : Boolean,
                        //useEndOfPacket : Boolean,

                        addressUnits : AddressUnits = symbols,
                        burstCountUnits : AddressUnits = words,
                        burstOnBurstBoundariesOnly : Boolean = false,
                        constantBurstBehavior : Boolean = false,
                        holdTime : Int = 0,
                        linewrapBursts : Boolean = false,
                        maximumPendingReadTransactions : Int = 1,
                        maximumPendingWriteTransactions : Int = 0, // unlimited
                        readLatency : Int = 0,
                        readWaitTime : Int = 0,
                        setupTime : Int = 0,
                        writeWaitTime : Int = 0
                        )
```

This configuration class has also some functions :

Name	Return	Description
getReadOnlyConfig	AvalonMMConfig	Return a similar configuration but with all write feature disabled
getWriteOnlyConfig	AvalonMMConfig	Return a similar configuration but with all read feature disabled

This configuration companion object has also some functions to provide some `AvalonMMConfig` templates :

Name	Return	Description
fixed(addressWidth,dataWidth,readLatency)	AvalonMMConfig	Return a simple configuration with fixed read timings
pipelined(addressWidth,dataWidth)	AvalonMMConfig	Return a configuration with variable latency read (readDataValid)
bursted(addressWidth,dataWidth,burstCountWidth)	AvalonMMConfig	Return a configuration with variable latency read and burst capabilities

```
// Create a write only AvalonMM configuration with burst capabilities and byte enable
val myAvalonConfig =  AvalonMMConfig.bursted(
                        addressWidth = addressWidth,
                        dataWidth = memDataWidth,
                        burstCountWidth = log2Up(burstSize + 1)
                      ).copy(
                        useByteEnable = true,
                        constantBurstBehavior = true,
                        burstOnBurstBoundariesOnly = true
                      ).getWriteOnlyConfig

// Create an instance of the AvalonMM bus by using this configuration
val bus = AvalonMM(myAvalonConfig)
```

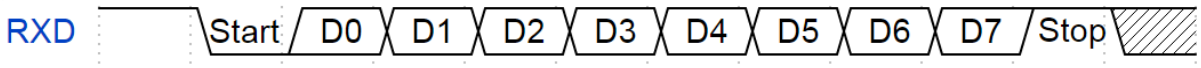
11.10 Com

11.10.1 UART

Introduction

The UART protocol could be used, for instance, to emit and receive RS232 / RS485 frames.

There is an example of an 8 bits frame, with no parity and one stop bit :



Bus definition

```
case class Uart() extends Bundle with IMasterSlave {
  val txd = Bool() // Used to emit frames
  val rxd = Bool() // Used to receive frames

  override def asMaster(): Unit = {
    out(txd)
    in(rxd)
  }
}
```

UartCtrl

An Uart controller is implemented in the library. This controller has the specificity to use a sampling window to read the `rxd` pin and then to using an majority vote to filter its value.

IO name	direction	type	Description
config	in	UartCtrlConfig	Used to set the clock divider/parity/stop/data length of the controller
write	slave	Stream[Bits]	Stream port used to request a frame transmission
read	master	Flow[Bits]	Flow port used to receive decoded frames
uart	master	Uart	Interface to the real world

The controller could be instantiated via an `UartCtrlGenerics` configuration object :

Attribute	type	Description
dataWidthMax	Int	Maximal number of bit inside a frame
clockDividerWidth	Int	Width of the internal clock divider
preSamplingSize	Int	Specify how many samplingTick are drop at the beginning of a UART baud
samplingSize	Int	Specify how many samplingTick are used to sample `rxd` values in the middle of the UART baud
postSamplingSize	Int	Specify how many samplingTick are drop at the end of a UART baud

11.11 IO

11.11.1 ReadableOpenDrain

ReadableOpenDrain

The ReadableOpenDrain bundle is defined as following :

```
case class ReadableOpenDrain[T<: Data](dataType : HardType[T]) extends Bundle with␣
↪IMasterSlave{
  val write,read : T = dataType()

  override def asMaster(): Unit = {
    out(write)
    in(read)
  }
}
```

Then, as a master, you can use the `read` signal to read the outside value and use the `write` to set the value that you want to drive on the output.

There is an example of usage :

```
val io = new Bundle{
  val dataBus = master(ReadableOpenDrain(Bits(32 bits)))
}

io.dataBus.write := 0x12345678
when(io.dataBus.read === 42){

}
```

11.11.2 TriState

Introduction

Tri-state signals are weird to handle in many cases:

- They are not really kind of digital things
- And except for IO, they aren't used for digital design
- The tristate concept doesn't fit naturally in the SpinalHDL internal graph.

SpinalHDL provides two different abstractions for tristate signals. The `TriState` bundle and *Analog and inout* signals. Both serve different purposes:

- TriState should be used for most purposes, especially within a design. The bundle contains an additional signal to carry the current direction.
- `Analog` and `inout` should be used for drivers on the device boundary and in some other special cases. See the referenced documentation page for more details.

As stated above, the recommended approach is to use `TriState` within a design. On the top-level the `TriState` bundle is then assigned to an analog inout to get the synthesis tools to infer the correct I/O driver. This can be done automatically done via the *InOutWrapper* or manually if needed.

TriState

The TriState bundle is defined as following :

```
case class TriState[T <: Data](dataType : HardType[T]) extends Bundle with␣
↪IMasterSlave{
  val read,write : T = dataType()
  val writeEnable = Bool()

  override def asMaster(): Unit = {
    out(write,writeEnable)
    in(read)
  }
}
```

A master can use the `read` signal to read the outside value, the `writeEnable` to enable the output, and finally use `write` to set the value that is driven on the output.

There is an example of usage:

```
val io = new Bundle{
  val dataBus = master(TriState(Bits(32 bits)))
}

io.dataBus.writeEnable := True
io.dataBus.write := 0x12345678
when(io.dataBus.read === 42){

}
```

TriStateArray

In some case, you need to have the control over the output enable of each individual pin (Like for GPIO). In this range of cases, you can use the TriStateArray bundle.

It is defined as following :

```
case class TriStateArray(width : BitCount) extends Bundle with IMasterSlave{
  val read,write,writeEnable = Bits(width)

  override def asMaster(): Unit = {
    out(write,writeEnable)
    in(read)
  }
}
```

It is the same than the TriState bundle, except that the `writeEnable` is an Bits to control each output buffer.

There is an example of usage :

```
val io = new Bundle{
  val dataBus = master(TriStateArray(32 bits)
}

io.dataBus.writeEnable := 0x87654321
io.dataBus.write := 0x12345678
when(io.dataBus.read === 42){

}
```

11.12 Graphics

11.12.1 Colors

RGB

You can use an Rgb bundle to model colors in hardware. This Rgb bundle take as parameter an RgbConfig classes which specify the number of bits for each channels :

```
case class RgbConfig(rWidth : Int,gWidth : Int,bWidth : Int){
  def getWidth = rWidth + gWidth + bWidth
}

case class Rgb(c: RgbConfig) extends Bundle{
  val r = UInt(c.rWidth bits)
  val g = UInt(c.gWidth bits)
  val b = UInt(c.bWidth bits)
}
```

Those classes could be used as following :

```
val config = RgbConfig(5,6,5)
val color = Rgb(config)
color.r := 31
```

11.12.2 VGA

VGA bus

An VGA bus definition is available via the Vga bundle.

```
case class Vga (rgbConfig: RgbConfig) extends Bundle with IMasterSlave{
  val vSync = Bool()
  val hSync = Bool()

  val colorEn = Bool()  //High when the frame is inside the color area
  val color = Rgb(rgbConfig)

  override def asMaster() = this.asOutput()
}
```

VGA timings

VGA timings could be modeled in hardware by using an VgaTimings bundle :

```
case class VgaTimingsHV(timingsWidth: Int) extends Bundle {
  val colorStart = UInt(timingsWidth bit)
  val colorEnd = UInt(timingsWidth bit)
  val syncStart = UInt(timingsWidth bit)
  val syncEnd = UInt(timingsWidth bit)
}

case class VgaTimings(timingsWidth: Int) extends Bundle {
  val h = VgaTimingsHV(timingsWidth)
  val v = VgaTimingsHV(timingsWidth)

   def setAs_h640_v480_r60 = ...
   def driveFrom(busCtrl : BusSlaveFactory,baseAddress : Int) = ...
}
```

VGA controller

An VGA controller is available. It's definition is the following :

```
case class VgaCtrl(rgbConfig: RgbConfig, timingsWidth: Int = 12) extends Component {
  val io = new Bundle {
    val softReset = in Bool()
    val timings   = in(VgaTimings(timingsWidth))

    val frameStart = out Bool()
    val pixels     = slave Stream (Rgb(rgbConfig))
    val vga        = master(Vga(rgbConfig))

    val error      = out Bool()
  }
  // ...
}
```

`frameStart` is a signals that pulse one cycle at the beginning of each new frame.

`pixels` is a stream of color used to feed the VGA interface when needed.
`error` is high when a transaction on the pixels is needed, but nothing is present.

11.13 EDA

11.13.1 QSysify

Introduction

QSysify is a tool which is able to generate a QSys IP (tcl script) from a SpinalHDL component by analysing its IO definition. It currently implement the following interfaces features :

- Master/Slave AvalonMM
- Master/Slave APB3
- Clock domain input
- Reset output
- Interrupt input
- Conduit (Used in last resort)

Example

In the case of a UART controller :

```
case class AvalonMMUartCtrl(...) extends Component{
  val io = new Bundle{
    val bus =  slave(AvalonMM(AvalonMMUartCtrl.getAvalonMMConfig))
    val uart = master(Uart())
  }

  //...
}
```

The following `main` will generate the Verilog and the QSys TCL script with io.bus as an AvalonMM and io.uart as a conduit :

```
object AvalonMMUartCtrl{
  def main(args: Array[String]) {
    //Generate the Verilog
    val toplevel = SpinalVerilog(AvalonMMUartCtrl(UartCtrlMemoryMappedConfig(...))).
↪toplevel

    //Add some tags to the avalon bus to specify it's clock domain (information used␣
↪by QSysify)
    toplevel.io.bus addTag(ClockDomainTag(toplevel.clockDomain))

    //Generate the QSys IP (tcl script)
    QSysify(toplevel)
  }
}
```

tags

Because QSys require some information that are not specified in the SpinalHDL hardware specification, some tags should be added to interface:

AvalonMM / APB3

```
io.bus addTag(ClockDomainTag(busClockDomain))
```

Interrupt input

```
io.interrupt addTag(InterruptReceiverTag(relatedMemoryInterfacei,␣
↪interruptClockDomain))
```

Reset output

```
io.resetOutput addTag(ResetEmitterTag(resetOutputClockDomain))
```

Adding new interface support

Basically, the QSysify tool can be setup with a list of interface `emitter` (as you can see here)

You can create your own emitter by creating a new class extending QSysifyInterfaceEmiter

11.13.2 QuartusFlow

Introduction

A compilation flow is an Altera-defined sequence of commands that use a combination of command-line executables. A full compilation flow launches all Compiler modules in sequence to synthesize, fit, analyze final timing, and generate a device programming file.

Tools in this file help you get rid of redundant Quartus GUI.

For a single rtl file

The object `spinal.lib.eda.altera.QuartusFlow` can automatically report the used area and maximum frequency of a single rtl file.

Example

```
val report = QuartusFlow(
   quartusPath="/eda/intelFPGA_lite/17.0/quartus/bin/",
   workspacePath="/home/spinalvm/tmp",
   toplevelPath="TopLevel.vhd",
   family="Cyclone V",
   device="5CSEMA5F31C6",
   frequencyTarget = 1 MHz
)
println(report)
```

The code above will create a new Quartus project with `TopLevel.vhd`.

Warning: This operation will remove the folder `workspacePath`!

Note: The `family` and `device` values are passed straight to the Quartus CLI as parameters. Please check the Quartus documentation for the correct value to use in your project.

Tip

To test a component that has too many pins, set them as `VIRTUAL_PIN`.

```
val miaou: Vec[Flow[Bool]] = Vec(master(Flow(Bool())), 666)
miaou.addAttribute("altera_attribute", "-name VIRTUAL_PIN ON")
```

For an existing project

The class `spinal.lib.eda.altera.QuartusProject` can automatically find configuration files in an existing project. Those are used for compilation and programming the device.

Example

Specify the path that contains your project files like `.qpf` and `.cdf`.

```
val prj = new QuartusProject(
   quartusPath = "F:/intelFPGA_lite/20.1/quartus/bin64/",
   workspacePath = "G:/"
)
prj.compile()
prj.program()  // automatically find Chain Description File of the project
```

Important: Remember to save the `.cdf` of your project before calling `prj.program()`.

11.14 Misc

11.14.1 Plic Mapper

The PLIC Mapper defines the register generation and access for a PLIC (Platform Level Interrupt Controller.

PlicMapper.apply

`(bus: BusSlaveFactory, mapping: PlicMapping)(gateways : Seq[PlicGateway], targets : Seq[PlicTarget])`

args for PlicMapper:

- **bus**: bus to which this ctrl is attached
- **mapping**: a mapping configuration (see above)
- **gateways**: a sequence of PlicGateway (interrupt sources) to generate the bus access control
- **targets**: the sequence of PlicTarget (eg. multiple cores) to generate the bus access control

It follows the interface given by riscv: https://github.com/riscv/riscv-plic-spec/blob/master/riscv-plic.adoc

As of now, two memory mappings are available :

PlicMapping.sifive

Follows the SiFive PLIC mapping (eg. E31 core complex Manual), basically a full fledged PLIC

PlicMapping.light

This mapping generates a lighter PLIC, at the cost of some missing optional features:

- no reading the intrerrupt's priority
- no reading the interrupts's pending bit (must use the claim/complete mechanism)
- no reading the target's threshold

The rest of the registers & logic is generated.

11.15 Introduction

11.15.1 Introduction

The spinal.lib package goals are :

- Provide things that are commonly used in hardware design (FIFO, clock crossing bridges, useful functions)
- Provide simple peripherals (UART, JTAG, VGA, ..)
- Provide some bus definition (Avalon, AMBA, ..)
- Provide some methodology (Stream, Flow, Fragment)
- Provide some example to get the spirit of spinal
- Provide some tools and facilities (latency analyser, QSys converter, . . .)

To use features introduced in followings chapter you need, in most of cases, to `import spinal.lib._` in your sources.

Important:

This package is currently under construction. Documented features could be considered as stable.
Do not hesitate to use github for suggestions/bug/fixes/enhancements

---

CHAPTER

TWELVE

SIMULATION

12.1 Setup and installation

SpinalSim + Verilator is supported on both Linux and Windows platforms.

12.1.1 Scala

Don't forget to add the following in your `build.sbt` file:

```
fork := true
```

And you will always need the following imports in your Scala testbench:

```
import spinal.core._
import spinal.core.sim._
```

12.1.2 Linux

You will also need a recent version of Verilator installed :

```
sudo apt-get install git make autoconf g++ flex bison -y  # First time prerequisites
git clone http://git.veripool.org/git/verilator   # Only first time
unsetenv VERILATOR_ROOT  # For csh; ignore error if on bash
unset VERILATOR_ROOT  # For bash
cd verilator
git pull        # Make sure we're up-to-date
git checkout v4.040
autoconf        # Create ./configure script
./configure
make -j$(nproc)
sudo make install
echo "DONE"
```

12.1.3 Windows

In order to get SpinalSim + Verilator working on Windows, you have to do the following:

- Install MSYS2
- Via MSYS2 get gcc/g++/verilator (for Verilator you can compile it from the sources)
- Add `bin` and `usr\bin` of MSYS2 into your windows `PATH` (ie : `C:\msys64\usr\bin;C:\msys64\mingw64\bin`)
- Check that the JAVA_HOME environnement variable point to the JDK installation folder (ie : `C:\Program Files\Java\jdk-13.0.2`)

Then you should be able to run SpinalSim + Verilator from your Scala project without having to use MSYS2 anymore.

From a fresh install of MSYS2 MinGW 64-bit, you will have to run the following commands inside the MSYS2 MinGW 64-bits shell (enter commands one by one):

From the MinGW package manager

```
pacman -Syuu
# Close the MSYS2 shell once you're asked to
pacman -Syuu
pacman -S --needed base-devel mingw-w64-x86_64-toolchain \
                   git flex\
                   mingw-w64-x86_64-cmake

pacman -U http://repo.msys2.org/mingw/x86_64/mingw-w64-x86_64-verilator-4.032-1-any.
↪pkg.tar.xz

# Add C:\msys64\usr\bin;C:\msys64\mingw64\bin to your Windows PATH
```

From source

```
pacman -Syuu
# Close the MSYS2 shell once you're asked to
pacman -Syuu
pacman -S --needed base-devel mingw-w64-x86_64-toolchain \
                   git flex\
                   mingw-w64-x86_64-cmake

git clone http://git.veripool.org/git/verilator
unset VERILATOR_ROOT
cd verilator
git pull
git checkout v4.040
autoconf
./configure
export CPLUS_INCLUDE_PATH=/usr/include:$CPLUS_INCLUDE_PATH
export PATH=/usr/bin/core_perl:$PATH
cp /usr/include/FlexLexer.h ./src

make -j$(nproc)
make install
echo "DONE"
# Add C:\msys64\usr\bin;C:\msys64\mingw64\bin to your Windows PATH
```

Important: Be sure that your `PATH` environnement variable is pointing to the JDK 1.8 and doesn't contain a JRE installation.

Important: Adding the MSYS2 `bin` folders into your windows `PATH` could potentialy have some side effects. This is why it is safer to add them as the last elements of the `PATH` to reduce their priority.

12.2 Boot a simulation

12.2.1 Introduction

There is an example hardware definition + testbench :

```
//Your hardware toplevel
import spinal.core._
class TopLevel extends Component {
  ...
}

// Your toplevel tester
import spinal.sim._
import spinal.core.sim._

object DutTests {
  def main(args: Array[String]): Unit = {
    SimConfig.withWave.compile(new TopLevel).doSim{ dut =>
      // Simulation code here
    }
  }
}
```

12.2.2 Configuration

`SimConfig` will return a default simulation configuration instance on which you can call multiple functions to configure your simulation:

Syntax	Description
`withWave`	Enable simulation wave capture
`withConfig(SpinalConfig)`	Specify the `SpinalConfig` that should be use to generate the hardware
`allOptimisation`	Enable all the RTL compilation optimizations to reduce simulation time (will increase compilation time)
`workspacePath(path)`	Change the folder where the sim files are generated

Then you can call the `compile(rtl)` function to compile the hardware and warm up the simulator. This function will return a `SimCompiled` instance.

On this `SimCompiled` instance you can run your simulation with the following functions:

Syntax	Description
`doSim[(simName[, seed])]{dut => ...}`	Run the simulation until the main thread is done (doesn't wait on forked threads) or until all threads are stuck
`doSimUntilVoid[(simName[, seed])]{dut => ...}`	Run the simulation until all threads are done or stuck

For example :

```
val spinalConfig = SpinalConfig(defaultClockDomainFrequency = FixedFrequency(10 MHz))

SimConfig
  .withConfig(spinalConfig)
  .withWave
  .allOptimisation
  .workspacePath("~/tmp")
  .compile(new TopLevel)
  .doSim { dut =>
    // Simulation code here
}
```

Note that by default, the simulation files will be placed into the `simWorkspace/xxx` folders. You can override the simWorkspace location by setting the `SPINALSIM_WORKSPACE` environnement variable.

12.2.3 Running multiple tests on the same hardware

```
val compiled = SimConfig.withWave.compile(new Dut)

compiled.doSim("testA") { dut =>
   // Simulation code here
}

compiled.doSim("testB") { dut =>
   // Simulation code here
}
```

12.2.4 Throw Success or Failure of the simulation from a thread

At any moment during a simulation you can call `simSuccess` or `simFailure` to end it.

12.3 Accessing signals of the simulation

12.3.1 Read and write signals

Each interface signal of the toplevel can be read and written from Scala:

Syntax	Description
Bool.toBoolean	Read a hardware `Bool` as a Scala `Boolean` value
Bits/UInt/SInt.toInt	Read a hardware `BitVector` as a Scala `Int` value
Bits/UInt/SInt.toLong	Read a hardware `BitVector` as a Scala `Long` value
Bits/UInt/SInt.toBigInt	Read a hardware `BitVector` as a Scala `BigInt` value
SpinalEnumCraft.toEnum	Read a hardware `SpinalEnumCraft` as a Scala `SpinalEnumElement` value
Bool #= Boolean	Assign a hardware `Bool` from an Scala `Boolean`
Bits/UInt/SInt #= Int	Assign a hardware `BitVector` from a Scala `Int`
Bits/UInt/SInt #= Long	Assign a hardware `BitVector` from a Scala `Long`
Bits/UInt/SInt #= BigInt	Assign a hardware `BitVector` from a Scala `BigInt`
SpinalEnumCraft #= SpinalEnumElement	Assign a hardware `SpinalEnumCraft` from a Scala `SpinalEnumElement`

```
dut.io.a #= 42
dut.io.a #= 42l
dut.io.a #= BigInt("101010", 2)
dut.io.a #= BigInt("0123456789ABCDEF", 16)
println(dut.io.b.toInt)
```

12.3.2 Accessing signals inside the component's hierarchy

To access signals which are inside the component's hierarchy, you have first to set the given signal as `simPublic`.

You can add this `simPublic` tag directly in the hardware description:

```
object SimAccessSubSignal {
  import spinal.core.sim._

  class TopLevel extends Component {
    val counter = Reg(UInt(8 bits)) init(0) simPublic() // Here we add the simPublic␣
↪tag on the counter register to make it visible
    counter := counter + 1
  }

  def main(args: Array[String]) {
    SimConfig.compile(new TopLevel).doSim{dut =>
      dut.clockDomain.forkStimulus(10)

      for(i <- 0 to 3) {
        dut.clockDomain.waitSampling()
        println(dut.counter.toInt)
      }
    }
  }
}
```

Or you can add it later, after having instantiated your toplevel for the simulation:

```
object SimAccessSubSignal {
  import spinal.core.sim._
  class TopLevel extends Component {
    val counter = Reg(UInt(8 bits)) init(0)
    counter := counter + 1
  }
```

(continues on next page)

(continued from previous page)

```
  def main(args: Array[String]) {
    SimConfig.compile {
      val dut = new TopLevel
      dut.counter.simPublic()
      dut
    }.doSim{dut =>
      dut.clockDomain.forkStimulus(10)

      for(i <- 0 to 3) {
        dut.clockDomain.waitSampling()
        println(dut.counter.toInt)
      }
    }
  }
}
```

12.4 Clock domains

12.4.1 Stimulus API

Below is a list of `ClockDomain` stimulation functions:

ClockDomain stimulus functions	Description
`forkStimulus(period)`	Fork a simulation process to generate the clockdomain stimulus (clock, reset, softReset, clockEnable signals)
`forkSimSpeedPrinter(printPeriod)`	Fork a simulation process which will periodically print the simulation speed in kilo-cycles per real time second. `printPeriod` is in realtime seconds
`clockToggle()`	Toggle the clock signal
`fallingEdge()`	Clear the clock signal
`risingEdge()`	Set the clock signal
`assertReset()`	Set the reset signal to its active level
`deassertReset()`	Set the reset signal to its inactive level
`assertClockEnable()`	Set the clockEnable signal to its active level
`deassertClockEnable()`	Set the clockEnable signal to its active level
`assertSoftReset()`	Set the softReset signal to its active level
`deassertSoftReset()`	Set the softReset signal to its active level

12.4.2 Wait API

Below is a list of `ClockDomain` utilities that you can use to wait for a given event from the domain:

ClockDomain wait functions	Description
`waitSampling([cyclesCount])`	Wait until the `ClockDomain` makes a sampling, (active clock edge && deassertReset && assertClockEnable)
`waitRisingEdge([cyclesCount])`	Wait cyclesCount rising edges on the clock; cycleCount defaults to 1 cycle if not otherwise specified. Note, cyclesCount = 0 is legal, and the function is not sensitive to reset/softReset/clockEnable
`waitFallingEdge([cyclesCount])`	Same as `waitRisingEdge` but for the falling edge
`waitActiveEdge([cyclesCount])`	Same as `waitRisingEdge` but for the edge level specified by the `ClockDomainConfig`
`waitRisingEdgeWhere(condition)`	Same as `waitRisingEdge`, but to exit, the boolean `condition` must be true when the rising edge occurs
`waitFallingEdgeWhere(condition)`	Same as `waitRisingEdgeWhere`, but for the falling edge
`waitActiveEdgeWhere(condition)`	Same as `waitRisingEdgeWhere`, but for the edge level specified by the `ClockDomainConfig`

Warning: All the functionalities of the wait API can only be called from inside of a thread, and not from a callback.

12.4.3 Callback API

Below is a list of `ClockDomain` utilities that you can use to wait for a given event from the domain:

ClockDomain callback functions	Description
`onNextSampling { callback }`	Execute the callback code only once on the next `ClockDomain` sample (active edge + reset off + clock enable on)
`onSamplings { callback }`	Execute the callback code each time the `ClockDomain` sample (active edge + reset off + clock enable on)
`onActiveEdges { callback }`	Execute the callback code each time the `ClockDomain` clock generates its configured edge
`onEdges { callback }`	Execute the callback code each time the `ClockDomain` clock generates a rising or falling edge
`onRisingEdges { callback }`	Execute the callback code each time the `ClockDomain` clock generates a rising edge
`onFallingEdges { callback }`	Execute the callback code each time the `ClockDomain` clock generates a falling edge

12.4.4 Default ClockDomain

You can access the default `ClockDomain` of your toplevel as shown below:

```
// Example of thread forking to generate a reset, and then toggling the clock each 5␣
↪time units.
// dut.clockDomain refers to the implicit clock domain created during component␣
↪instantiation.
fork {
  dut.clockDomain.assertReset()
```

(continues on next page)

(continued from previous page)

```
  dut.clockDomain.fallingEdge()
  sleep(10)
  while(true) {
    dut.clockDomain.clockToggle()
    sleep(5)
  }
}
```

Note that you can also directly fork a standard reset/clock process:

```
dut.clockDomain.forkStimulus(period = 10)
```

An example of how to wait for a rising edge on the clock:

```
dut.clockDomain.waitRisingEdge()
```

12.4.5 New ClockDomain

If your toplevel defines some clock and reset inputs which aren't directly integrated into their `ClockDomain`, you can define their corresponding `ClockDomain` directly in the testbench:

```
// In the testbench
ClockDomain(dut.io.coreClk, dut.io.coreReset).forkStimulus(10)
```

12.5 Thread-full API

In SpinalSim, you can write your testbench by using multiple threads in a similar way to SystemVerilog, and a bit like VHDL/Verilog process/always blocks. This allows you to write concurrent tasks and control the simulation time using a fluent API.

12.5.1 Fork and join simulation threads

```
// Create a new thread
val myNewThread = fork {
  // New simulation thread body
}

// Wait until `myNewThread` is execution is done.
myNewThread.join()
```

12.5.2 Sleep and waitUntil

```
// Sleep 1000 units of time
sleep(1000)

// waitUntil the dut.io.a value is bigger than 42 before continuing
waitUntil(dut.io.a > 42)
```

12.6 Thread-less API

There are some functions that you can use to avoid the need for threading, but which still allow you to control the flow of simulation time.

Threadless functions	Description
`delayed(delay) { callback }`	Register the callback code to be called at a simulation time `delay` steps after the current timestep.

The advantages of the `delayed` function over using a regular simulation thread + sleep are:

- Performance (no context switching)
- Memory usage (no native JVM thread memory allocation)

Some other thread-less functions related to `ClockDomain` objects are documented as part of the *Callback API*, and some others related with the delta-cycle execution process are documented as part of the *Sensitive API*

12.7 Sensitive API

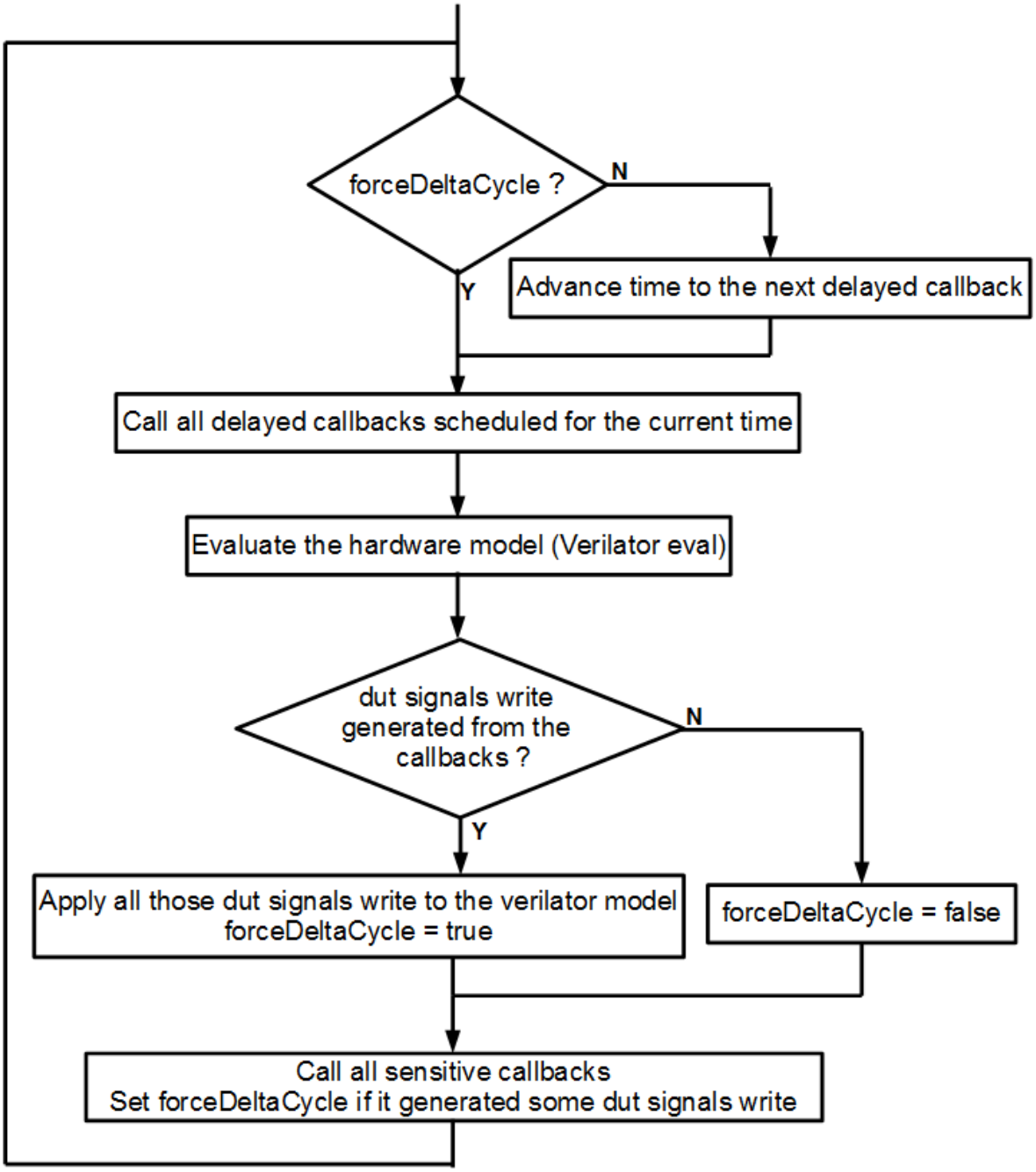
You can register callback functions to be called on each delta-cycle of the simulation:

Sensitive functions	Description
`forkSensitive { callback }`	Register the callback code to be called at each delta-cycle of the simulation
`forkSensitiveWhile { callback }`	Register the callback code to be called at each delta-cycle of the simulation, while the callback return value is true (meaning it should be rescheduled for the next delta-cycle)

12.8 Simulation engine

This page explains the internals of the simulation engine.

The simulation engine emulates an event-driven simulator (VHDL/Verilog like) by applying the following simulation loop on the top of the Verilator C++ simulation model:



At a low level, the simulation engine manages the following primitives:

- *Sensitive callbacks*, which allow users to call a function on each simulation delta cycle.
- *Delayed callbacks*, which allow users to call a function at a future simulation time.
- *Simulation threads*, which allow users to describe concurrent processes.
- *Command buffer*, which allows users to delay write access to the DUT (Device Under Test) until the end of the current delta cycle.

There are some practical uses of those primitives:

- Sensitive callbacks can be used to wake up a simulation thread when a given condition happens, like a rising edge on a clock.
- Delayed callbacks can be used to schedule stimuli, such as deasserting a reset after a given time, or toggling the clock.

- Both sensitive and delayed callbacks can be used to resume a simulation thread.
- A simulation thread can be used (for instance) to produce stimulus and check the DUT's output values.
- The command buffer's purpose is mainly to avoid all concurrency issues between the DUT and the testbench.

12.9 Examples

12.9.1 Asynchronous adder

This example creates a `Component` out of combinational logic that does some simple arithmetic on 3 operands.

The test bench performs the following steps 100 times:

- Initialize `a`, `b`, and `c` to random integers in the 0..255 range.
- Stimulate the DUT's matching `a`, `b`, `c` inputs.
- Wait 1 simulation timestep (to allow the inputs to propagate).
- Check for correct output.

```
import spinal.sim._
import spinal.core._
import spinal.core.sim._

import scala.util.Random


object SimAsynchronousExample {
  class Dut extends Component {
    val io = new Bundle {
      val a, b, c = in UInt (8 bits)
      val result = out UInt (8 bits)
    }
    io.result := io.a + io.b - io.c
  }

  def main(args: Array[String]): Unit = {
    SimConfig.withWave.compile(new Dut).doSim{ dut =>
      var idx = 0
      while(idx < 100){
        val a, b, c = Random.nextInt(256)
        dut.io.a #= a
        dut.io.b #= b
        dut.io.c #= c
        sleep(1) // Sleep 1 simulation timestep
        assert(dut.io.result.toInt == ((a + b - c) & 0xFF))
        idx += 1
      }
    }
  }
}
```

12.9.2 Dual clock fifo

This example creates a `StreamFifoCC`, which is designed for crossing clock domains, along with 3 simulation threads.

The threads handle:

- Management of the two clocks
- Pushing to the FIFO
- Popping from the FIFO

The FIFO push thread randomizes the inputs.

The FIFO pop thread handles checking the the DUT's outputs against the reference model (an ordinary `scala.collection.mutable.Queue` instance).

```
import spinal.sim._
import spinal.core._
import spinal.core.sim._

import scala.collection.mutable.Queue


object SimStreamFifoCCExample {
  def main(args: Array[String]): Unit = {
    // Compile the Component for the simulator.
    val compiled = SimConfig.withWave.allOptimisation.compile(
      rtl = new StreamFifoCC(
        dataType = Bits(32 bits),
        depth = 32,
        pushClock = ClockDomain.external("clkA"),
        popClock = ClockDomain.external("clkB")
      )
    )

    // Run the simulation.
    compiled.doSimUntilVoid{dut =>
      val queueModel = mutable.Queue[Long]()

      // Fork a thread to manage the clock domains signals
      val clocksThread = fork {
        // Clear the clock domains' signals, to be sure the simulation captures their␣
↪first edges.
        dut.pushClock.fallingEdge()
        dut.popClock.fallingEdge()
        dut.pushClock.deassertReset()
        dut.popClock.deassertReset()
        sleep(0)

        // Do the resets.
        dut.pushClock.assertReset()
        dut.popClock.assertReset()
        sleep(10)
        dut.pushClock.deassertReset()
        dut.popClock.deassertReset()
        sleep(1)

        // Forever, randomly toggle one of the clocks.
```

(continues on next page)

(continued from previous page)

```
        // This will create asynchronous clocks without fixed frequencies.
        while(true) {
          if(Random.nextBoolean()) {
            dut.pushClock.clockToggle()
          } else {
            dut.popClock.clockToggle()
          }
          sleep(1)
        }
      }

      // Push data randomly, and fill the queueModel with pushed transactions.
      val pushThread = fork {
        while(true) {
          dut.io.push.valid.randomize()
          dut.io.push.payload.randomize()
          dut.pushClock.waitSampling()
          if(dut.io.push.valid.toBoolean && dut.io.push.ready.toBoolean) {
            queueModel.enqueue(dut.io.push.payload.toLong)
          }
        }
      }

      // Pop data randomly, and check that it match with the queueModel.
      val popThread = fork {
        for(i <- 0 until 100000) {
          dut.io.pop.ready.randomize()
          dut.popClock.waitSampling()
          if(dut.io.pop.valid.toBoolean && dut.io.pop.ready.toBoolean) {
            assert(dut.io.pop.payload.toLong == queueModel.dequeue())
          }
        }
        simSuccess()
      }
    }
  }
}
```

12.9.3 Single clock fifo

This example creates a `StreamFifo`, and spawns 3 simulation threads. Unlike the *Dual clock fifo* example, this FIFO does not need complex clock management.

The 3 simulation threads handle:

- Managing the clock/reset
- Pushing to the FIFO
- Popping from the FIFO

The FIFO push thread randomizes the inputs.

The FIFO pop thread handles checking the the DUT's outputs against the reference model (an ordinary `scala.collection.mutable.Queue` instance).

```
import spinal.sim._
import spinal.core._
```

(continues on next page)

(continued from previous page)

```
import spinal.core.sim._

import scala.collection.mutable.Queue


object SimStreamFifoExample {
  def main(args: Array[String]): Unit = {
    // Compile the Component for the simulator.
    val compiled = SimConfig.withWave.allOptimisation.compile(
      rtl = new StreamFifo(
        dataType = Bits(32 bits),
        depth = 32
      )
    )

    // Run the simulation.
    compiled.doSimUntilVoid{dut =>
      val queueModel = mutable.Queue[Long]()

      dut.clockDomain.forkStimulus(period = 10)
      SimTimeout(1000000*10)

      // Push data randomly, and fill the queueModel with pushed transactions.
      val pushThread = fork {
        dut.io.push.valid #= false
        while(true) {
          dut.io.push.valid.randomize()
          dut.io.push.payload.randomize()
          dut.clockDomain.waitSampling()
          if(dut.io.push.valid.toBoolean && dut.io.push.ready.toBoolean) {
            queueModel.enqueue(dut.io.push.payload.toLong)
          }
        }
      }

      // Pop data randomly, and check that it match with the queueModel.
      val popThread = fork {
        dut.io.pop.ready #= true
        for(i <- 0 until 100000) {
          dut.io.pop.ready.randomize()
          dut.clockDomain.waitSampling()
          if(dut.io.pop.valid.toBoolean && dut.io.pop.ready.toBoolean) {
            assert(dut.io.pop.payload.toLong == queueModel.dequeue())
          }
        }
        simSuccess()
      }
    }
  }
}
```

12.9.4 Synchronous adder

This example creates a `Component` out of sequential logic that does some simple arithmetic on 3 operands.

The test bench performs the following steps 100 times:

- Initialize `a`, `b`, and `c` to random integers in the 0..255 range.
- Stimulate the DUT's matching `a`, `b`, `c` inputs.
- Wait until the simulation samples the DUT's signals again.
- Check for correct output.

The main difference between this example and the *Asynchronous adder* example is that this `Component` has to use `forkStimulus` to generate a clock signal, since it is using sequential logic internally.

```
import spinal.sim._
import spinal.core._
import spinal.core.sim._

import scala.util.Random


object SimSynchronousExample {
  class Dut extends Component {
    val io = new Bundle {
      val a, b, c = in UInt (8 bits)
      val result = out UInt (8 bits)
    }
    io.result := RegNext(io.a + io.b - io.c) init(0)
  }

  def main(args: Array[String]): Unit = {
    SimConfig.withWave.compile(new Dut).doSim{ dut =>
      dut.clockDomain.forkStimulus(period = 10)

      var resultModel = 0
      for(idx <- 0 until 100){
        dut.io.a #= Random.nextInt(256)
        dut.io.b #= Random.nextInt(256)
        dut.io.c #= Random.nextInt(256)
        dut.clockDomain.waitSampling()
        assert(dut.io.result.toInt == resultModel)
        resultModel = (dut.io.a.toInt + dut.io.b.toInt - dut.io.c.toInt) & 0xFF
      }
    }
  }
}
```

12.9.5 Uart decoder

```
// Fork a simulation process which will analyze the uartPin and print transmitted␣
↪bytes into the simulation terminal.
fork {
  // Wait until the design sets the uartPin to true (wait for the reset effect).
  waitUntil(uartPin.toBoolean == true)

  while(true) {
    waitUntil(uartPin.toBoolean == false)
    sleep(baudPeriod/2)

    assert(uartPin.toBoolean == false)
    sleep(baudPeriod)

    var buffer = 0
    for(bitId <- 0 to 7) {
      if(uartPin.toBoolean)
        buffer |= 1 << bitId
      sleep(baudPeriod)
    }

    assert(uartPin.toBoolean == true)
    print(buffer.toChar)
  }
}
```

12.9.6 Uart encoder

```
// Fork a simulation process which will get chars typed into the simulation terminal␣
↪and transmit them on the simulation uartPin.
fork{
  uartPin #= true
  while(true) {
    // System.in is the java equivalent of the C's stdin.
    if(System.in.available() != 0) {
      val buffer = System.in.read()
      uartPin #= false
      sleep(baudPeriod)

      for(bitId <- 0 to 7) {
        uartPin #= ((buffer >> bitId) & 1) != 0
        sleep(baudPeriod)
      }

      uartPin #= true
      sleep(baudPeriod)
    } else {
      sleep(baudPeriod * 10) // Sleep a little while to avoid polling System.in too␣
↪often.
    }
  }
}
```

12.10 Introduction

As always, you can use your standard simulation tools to simulate the VHDL/Verilog files generated by SpinalHDL, but since SpinalHDL 1.0.0 the language integrates an API that allows you to write testbenches and test your hardware directly in Scala.

The simulation API provides the capabilities to:

- Read and write the DUT's signals
- Fork and join simulation processes
- Sleep and wait until a given condition is filled

12.11 How does SpinalHDL simulate the hardware?

Behind the scenes, SpinalHDL generates a C++ cycle-accurate model of your hardware by generating the equivalent Verilog, and then using Verilator to convert it into a C++ model.

Then SpinalHDL uses GCC to compile the C++ model into a shared object (.so) file, and binds it back to Scala via JNI.

Finally, as the native Verilator API is rather crude, SpinalHDL abstracts over it by providing both single and multi-threaded simulation APIs to help the user construct testbench implementations.

This simulation methodology has several advantages:

- The C++ simulation model processes simulation steps very quickly
- It tests the generated Verilog hardware instead of the SpinalHDL internal model
- It doesn't require SpinalHDL to be able to simulate the hardware itself (This keeps the codebase smaller, and reduces bugs, since Verilator is a reliable tool)

However, there are some limitations:

- Verilator will only accept synthesizable Verilog code

12.12 Performance

As Verilator is the current simulation backend, the simulation speed is very fast.

On a small SoC like Murax a modern laptop can simulate 1.2 million clock cycles or more, per realtime second.

CHAPTER

THIRTEEN

EXAMPLES

13.1 Simple ones

13.1.1 APB3 definition

Introduction

This example will show the syntax to define an APB3 `Bundle`.

Specification

The specification from ARM could be interpreted as follows:

Signal Name	Type	Driver side	Comment
PADDR	UInt(addressWidth bits)	Master	Address in byte
PSEL	Bits(selWidth)	Master	One bit per slave
PENABLE	Bool	Master	
PWRITE	Bool	Master	
PWDATA	Bits(dataWidth bits)	Master	
PREADY	Bool	Slave	
PRDATA	Bits(dataWidth bits)	Slave	
PSLVERROR	Bool	Slave	Optional

Implementation

This specification shows that the APB3 bus has multiple possible configurations. To represent that, we can define a configuration class in Scala:

```
case class Apb3Config(
  addressWidth  : Int,
  dataWidth     : Int,
  selWidth      : Int     = 1,
  useSlaveError : Boolean = true
)
```

Then we can define the APB3 `Bundle` which will be used to represent the bus in hardware:

```
case class Apb3(config: Apb3Config) extends Bundle with IMasterSlave {
  val PADDR   = UInt(config.addressWidth bit)
  val PSEL    = Bits(config.selWidth bits)
```

(continues on next page)

(continued from previous page)

```
  val PENABLE    = Bool()
  val PREADY     = Bool()
  val PWRITE     = Bool()
  val PWDATA     = Bits(config.dataWidth bit)
  val PRDATA     = Bits(config.dataWidth bit)
  val PSLVERROR  = if(config.useSlaveError) Bool else null

  override def asMaster(): Unit = {
    out(PADDR,PSEL,PENABLE,PWRITE,PWDATA)
    in(PREADY,PRDATA)
    if(config.useSlaveError) in(PSLVERROR)
  }
}
```

Usage

Here is a usage example of this definition:

```
val apbConfig = Apb3Config(
  addressWidth  = 16,
  dataWidth     = 32,
  selWidth      = 1,
  useSlaveError = false
)

val io = new Bundle{
  val apb = slave(Apb3(apbConfig))
}

io.apb.PREADY := True
when(io.apb.PSEL(0) && io.apb.PENABLE){
  //...
}
```

13.1.2 Carry adder

This example defines a component with inputs `a` and `b`, and a `result` output. At any time, `result` will be the sum of `a` and `b` (combinatorial). This sum is manually done by a carry adder logic.

```
class CarryAdder(size : Int) extends Component{
  val io = new Bundle{
    val a = in UInt(size bits)
    val b = in UInt(size bits)
    val result = out UInt(size bits)      //result = a + b
  }

  var c = False                   //Carry, like a VHDL variable
  for (i <- 0 until size) {
    //Create some intermediate value in the loop scope.
    val a = io.a(i)
    val b = io.b(i)

    //The carry adder's asynchronous logic
    io.result(i) := a ^ b ^ c
```

(continues on next page)

(continued from previous page)

```
    c \= (a & b) | (a & c) | (b & c);    //variable assignment
  }
}


object CarryAdderProject {
  def main(args: Array[String]) {
    SpinalVhdl(new CarryAdder(4))
  }
}
```

13.1.3 Color summing

First let's define a Color `Bundle` with an addition operator.

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

  def +(that: Color): Color = {
    val result = Color(channelWidth)
    result.r := this.r + that.r
    result.g := this.g + that.g
    result.b := this.b + that.b
    return result
  }

  def clear(): Color ={
    this.r := 0
    this.g := 0
    this.b := 0
    this
  }
}
```

Then let's define a component with a `sources` input which is a vector of colors, and a `result` output which is the sum of the `sources` input.

```
class ColorSumming(sourceCount: Int, channelWidth: Int) extends Component {
  val io = new Bundle {
    val sources = in Vec(Color(channelWidth), sourceCount)
    val result = out(Color(channelWidth))
  }

  var sum = Color(channelWidth)
  sum.clear()
  for (i <- 0 to sourceCount - 1) {
    sum \= sum + io.sources(i)
  }
  io.result := sum
}
```

13.1.4 Counter with clear

This example defines a component with a `clear` input and a `value` output. Each clock cycle, the `value` output is incrementing, but when `clear` is high, `value` is cleared.

```
class Counter(width : Int) extends Component{
  val io = new Bundle{
    val clear = in Bool()
    val value = out UInt(width bits)
  }
  val register = Reg(UInt(width bits)) init(0)
  register := register + 1
  when(io.clear){
    register := 0
  }
  io.value := register
}
```

13.1.5 Introduction

All examples assume that you have the following imports on the top of your scala file:

```
import spinal.core._
import spinal.lib._
```

To generate VHDL for a given component, you can place the following at the bottom of your scala file:

```
object MyMainObject {
  def main(args: Array[String]) {
    SpinalVhdl(new TheComponentThatIWantToGenerate(constructionArguments))    //Or␣
↪SpinalVerilog
  }
}
```

13.1.6 PLL BlackBox and reset controller

Let's imagine you want to define a `TopLevel` component which instantiates a PLL `BlackBox`, and create a new clock domain from it which will be used by your core logic. Let's also imagine that you want to adapt an external asynchronous reset into this core clock domain to a synchronous reset source.

The following imports will be used in code examples on this page:

```
import spinal.core._
import spinal.lib._
```

The PLL BlackBox definition

This is how to define the PLL `BlackBox`:

```
class PLL extends BlackBox{
  val io = new Bundle{
    val clkIn    = in Bool()
    val clkOut   = out Bool()
    val isLocked = out Bool()
  }
```

(continues on next page)

(continued from previous page)

```
  noIoPrefix()
}
```

This will correspond to the following VHDL component:

```
component PLL is
  port(
    clkIn    : in std_logic;
    clkOut   : out std_logic;
    isLocked : out std_logic
  );
end component;
```

TopLevel definition

This is how to define your `TopLevel` which instantiates the PLL, creates the new `ClockDomain`, and also adapts the asynchronous reset input to a synchronous reset:

```
class TopLevel extends Component{
  val io = new Bundle {
    val aReset    = in Bool()
    val clk100Mhz = in Bool()
    val result    = out UInt(4 bits)
  }

  // Create an Area to manage all clocks and reset things
  val clkCtrl = new Area {
    //Instanciate and drive the PLL
    val pll = new PLL
    pll.io.clkIn := io.clk100Mhz

    //Create a new clock domain named 'core'
    val coreClockDomain = ClockDomain.internal(
      name = "core",
      frequency = FixedFrequency(200 MHz)  // This frequency specification can be used
    )                                      // by coreClockDomain users to do some␣
↪calculations

    //Drive clock and reset signals of the coreClockDomain previously created
    coreClockDomain.clock := pll.io.clkOut
    coreClockDomain.reset := ResetCtrl.asyncAssertSyncDeassert(
      input = io.aReset || ! pll.io.isLocked,
      clockDomain = coreClockDomain
    )
  }

  //Create a ClockingArea which will be under the effect of the clkCtrl.
↪coreClockDomain
  val core = new ClockingArea(clkCtrl.coreClockDomain){
    //Do your stuff which use coreClockDomain here
    val counter = Reg(UInt(4 bits)) init(0)
    counter := counter + 1
    io.result := counter
  }
}
```

13.1.7 RGB to gray

Let's imagine a component that converts an RGB color into a gray one, and then writes it into external memory.

io name	Direction	Description
clear	in	Clear all internal registers
r,g,b	in	Color inputs
wr	out	Memory write
address	out	Memory address, incrementing each cycle
data	out	Memory data, gray level

```
class RgbToGray extends Component{
  val io = new Bundle{
    val clear = in Bool()
    val r,g,b = in UInt(8 bits)

    val wr = out Bool()
    val address = out UInt(16 bits)
    val data = out UInt(8 bits)
  }

  def coef(value : UInt,by : Float) : UInt = (value * U((255*by).toInt,8 bits) >> 8)
  val gray = RegNext(
    coef(io.r,0.3f) +
    coef(io.g,0.4f) +
    coef(io.b,0.3f)
  )

  val address = CounterFreeRun(stateCount = 1 << 16)
  io.address := address
  io.wr := True
  io.data := gray

  when(io.clear){
    gray := 0
    address.clear()
    io.wr := False
  }
}
```

13.1.8 Sinus rom

Let's imagine that you want to generate a sine wave and also have a filtered version of it (which is completely useless in practical, but let's do it as an example).

Parameters name	Type	Description
resolutionWidth	Int	Number of bits used to represent numbers
sampleCount	Int	Number of samples in a sine period

IO name	Direction	Type	Description
sin	out	SInt(resolutionWidth bits)	Output which plays the sine wave
sinFiltred	out	SInt(resolutionWidth bits)	Output which plays the filtered version of the sine

So let's define the Component:

```
class TopLevel(resolutionWidth : Int,sampleCount : Int) extends Component {
  val io = new Bundle {
    val sin = out SInt(resolutionWidth bits)
    val sinFiltred = out SInt(resolutionWidth bits)
  }
  // Here will come the logic implementation
}
```

To play the sine wave on the sin output, you can define a ROM which contain all samples of a sine period (tt could be just a quarter, but let's do things by the simplest way).
Then you can read that ROM with an phase counter and this will generate your sine wave.

```
//Function used to generate the rom (later)
def sinTable = for(sampleIndex <- 0 until sampleCount) yield {
  val sinValue = Math.sin(2 * Math.PI * sampleIndex / sampleCount)
  S((sinValue * ((1<<resolutionWidth)/2-1)).toInt,resolutionWidth bits)
}

val rom =  Mem(SInt(resolutionWidth bits),initialContent = sinTable)
val phase = Reg(UInt(log2Up(sampleCount) bits)) init(0)
phase := phase + 1

io.sin := rom.readSync(phase)
```

Then to generate sinFiltred, you can for example use a first order low pass filter implementation:

```
io.sinFiltred := RegNext(io.sinFiltred  - (io.sinFiltred  >> 5) + (io.sin >> 5))␣
↪init(0)
```

Here is the complete code:

```
class TopLevel(resolutionWidth : Int,sampleCount : Int) extends Component {
  val io = new Bundle {
    val sin = out SInt(resolutionWidth bits)
    val sinFiltred = out SInt(resolutionWidth bits)
  }

  def sinTable = for(sampleIndex <- 0 until sampleCount) yield {
    val sinValue = Math.sin(2 * Math.PI * sampleIndex / sampleCount)
    S((sinValue * ((1<<resolutionWidth)/2-1)).toInt,resolutionWidth bits)
  }

  val rom =  Mem(SInt(resolutionWidth bits),initialContent = sinTable)
  val phase = Reg(UInt(log2Up(sampleCount) bits)) init(0)
```

(continues on next page)

(continued from previous page)

```
  phase := phase + 1

  io.sin := rom.readSync(phase)
  io.sinFiltred := RegNext(io.sinFiltred  - (io.sinFiltred  >> 5) + (io.sin >> 5))␣
↪init(0)
}
```

13.2 Intermediates ones

13.2.1 Fractal calculator

Introduction

This example will show a simple implementation (without optimization) of a Mandelbrot fractal calculator by using data streams and fixed point calculations.

Specification

The component will receive one `Stream` of pixel tasks (which contain the XY coordinates in the Mandelbrot space) and will produce one `Stream` of pixel results (which contain the number of iterations done for the corresponding task).

Let's specify the IO of our component:

IO Name	Direction	Type	Description
cmd	slave	Stream[PixelTask]	XY coordinates to process
rsp	master	Stream[PixelResult]	Iteration count needed for the corresponding cmd transaction

Let's specify the PixelTask `Bundle`:

Element Name	Type	Description
x	SFix	Coordinate in the Mandelbrot space
y	SFix	Coordinate in the Mandelbrot space

Let's specify the PixelResult `Bundle`:

Element Name	Type	Description
iteration	UInt	Number of iterations required to solve the Mandelbrot coordinates

Elaboration parameters (Generics)

Let's define the class that will provide construction parameters of our system:

```
case class PixelSolverGenerics(fixAmplitude : Int,
                               fixResolution : Int,
                               iterationLimit : Int){
  val iterationWidth = log2Up(iterationLimit+1)
  def iterationType = UInt(iterationWidth bits)
  def fixType = SFix(
    peak = fixAmplitude exp,
    resolution = fixResolution exp
  )
}
```

Note: iterationType and fixType are functions that you can call to instantiate new signals. It's like a typedef in C.

Bundle definition

```
case class PixelTask(g : PixelSolverGenerics) extends Bundle{
  val x,y = g.fixType
}

case class PixelResult(g : PixelSolverGenerics) extends Bundle{
  val iteration = g.iterationType
}
```

Component implementation

And now the implementation. The one below is a very simple one without pipelining / multi-threading.

```
case class PixelSolver(g : PixelSolverGenerics) extends Component{
  val io = new Bundle{
    val cmd = slave  Stream(PixelTask(g))
    val rsp = master Stream(PixelResult(g))
  }

  import g._

  //Define states
  val x,y       = Reg(fixType) init(0)
  val iteration = Reg(iterationType) init(0)

  //Do some shared calculation
  val xx = x*x
  val yy = y*y
  val xy = x*y

  //Apply default assignment
  io.cmd.ready := False
  io.rsp.valid := False
  io.rsp.iteration := iteration
```

(continues on next page)

(continued from previous page)

```
  when(io.cmd.valid) {
    //Is the mandelbrot iteration done ?
    when(xx + yy >= 4.0 || iteration === iterationLimit) {
      io.rsp.valid := True
      when(io.rsp.ready){
        io.cmd.ready := True
        x := 0
        y := 0
        iteration := 0
      }
    } otherwise {
      x := (xx - yy + io.cmd.x).truncated
      y := (((xy) << 1) + io.cmd.y).truncated
      iteration := iteration + 1
    }
  }
}
```

13.2.2 UART

Specification

This UART controller tutorial is based on this implementation.

This implementation is characterized by:

- ClockDivider/Parity/StopBit/DataLength configs are set by the component inputs.
- RXD input is filtered by using a sampling window of N samples and a majority vote.

Interfaces of this UartCtrl are:

Name	Type	Description
config	UartCtrlConfig	Give all configurations to the controller
write	Stream[Bits]	Port used by the system to give transmission order to the controller
read	Flow[Bits]	Port used by the controller to notify the system about a successfully received frame
uart	Uart	Uart interface with rxd / txd

Data structures

Before implementing the controller itself we need to define some data structures.

Controller construction parameters

Name	Type	Description
dataWidth-Max	Int	Maximum number of data bits that could be sent using a single UART frame
clockDi-vider-Width	Int	Number of bits that the clock divider has
pre-Sam-pling-Size	Int	Number of samples to drop at the beginning of the sampling window
sam-pling-Size	Int	Number of samples use at the middle of the window to get the filtered RXD value
post-Sam-pling-Size	Int	Number of samples to drop at the end of the sampling window

To make the implementation easier let's assume that `preSamplingSize + samplingSize + postSamplingSize` is always a power of two.

Instead of adding each construction parameters (generics) to `UartCtrl` one by one, we can group them inside a class that will be used as single parameter of `UartCtrl`.

```
case class UartCtrlGenerics( dataWidthMax: Int = 8,
                             clockDividerWidth: Int = 20, // baudrate = Fclk /␣
↪rxSamplePerBit / clockDividerWidth
                             preSamplingSize: Int = 1,
                             samplingSize: Int = 5,
                             postSamplingSize: Int = 2) {
  val rxSamplePerBit = preSamplingSize + samplingSize + postSamplingSize
  assert(isPow2(rxSamplePerBit))
  if ((samplingSize % 2) == 0)
    SpinalWarning(s"It's not nice to have a odd samplingSize value (because of the␣
↪majority vote)")
}
```

UART bus

Let's define a UART bus without flow control.

```
case class Uart() extends Bundle with IMasterSlave {
  val txd = Bool()
  val rxd = Bool()

  override def asMaster(): Unit = {
    out(txd)
```

(continues on next page)

(continued from previous page)

```
    in(rxd)
  }
}
```

UART configuration enums

Let's define parity and stop bit enumerations.

```
object UartParityType extends SpinalEnum(sequancial) {
  val NONE, EVEN, ODD = newElement()
}

object UartStopType extends SpinalEnum(sequancial) {
  val ONE, TWO = newElement()
  def toBitCount(that : T) : UInt = (that === ONE) ? U"0" | U"1"
}
```

UartCtrl configuration Bundles

Let's define `Bundle`s that will be used as IO elements to setup `UartCtrl`.

```
case class UartCtrlFrameConfig(g: UartCtrlGenerics) extends Bundle {
  val dataLength = UInt(log2Up(g.dataWidthMax) bit) //Bit count = dataLength + 1
  val stop       = UartStopType()
  val parity     = UartParityType()
}

case class UartCtrlConfig(g: UartCtrlGenerics) extends Bundle {
  val frame        = UartCtrlFrameConfig(g)
  val clockDivider = UInt (g.clockDividerWidth bit) //see UartCtrlGenerics.
↪clockDividerWidth for calculation

  def setClockDivider(baudrate : Double,clkFrequency : Double = ClockDomain.current.
↪frequency.getValue) : Unit = {
    clockDivider := (clkFrequency / baudrate / g.rxSamplePerBit).toInt
  }
}
```

Implementation

In `UartCtrl`, 3 things will be instantiated:

- One clock divider that generates a tick pulse at the UART RX sampling rate.
- One `UartCtrlTx` `Component`
- One `UartCtrlRx` `Component`

UartCtrlTx

The interfaces of this Component are the following :

Name	Type	Description
configFrame	UartCtrlFrameConfig	Contains data bit width count and party/stop bits configurations
samplingTick	Bool	Time reference that pulses rxSamplePerBit times per UART baud
write	Stream[Bits]	Port used by the system to give transmission orders to the controller
txd	Bool	UART txd pin

Let's define the enumeration that will be used to store the state of UartCtrlTx:

```
object UartCtrlTxState extends SpinalEnum {
  val IDLE, START, DATA, PARITY, STOP = newElement()
}
```

Let's define the skeleton of UartCtrlTx:

```
class UartCtrlTx(g : UartCtrlGenerics) extends Component {
  import g._

  val io = new Bundle {
    val configFrame  = in(UartCtrlFrameConfig(g))
    val samplingTick = in Bool()
    val write        = slave Stream (Bits(dataWidthMax bit))
    val txd          = out Bool
  }

  // Provide one clockDivider.tick each rxSamplePerBit pulses of io.samplingTick
  // Used by the stateMachine as a baud rate time reference
  val clockDivider = new Area {
    val counter = Reg(UInt(log2Up(rxSamplePerBit) bits)) init(0)
    val tick = False
    ..
  }

  // Count up each clockDivider.tick, used by the state machine to count up data bits␣
↪and stop bits
  val tickCounter = new Area {
    val value = Reg(UInt(Math.max(dataWidthMax, 2) bit))
    def reset() = value := 0
    ..
  }

  val stateMachine = new Area {
    import UartCtrlTxState._

    val state = RegInit(IDLE)
    val parity = Reg(Bool)
    val txd = True
    ..
    switch(state) {
```

(continues on next page)

(continued from previous page)

```
      ..
    }
  }

  io.txd := RegNext(stateMachine.txd) init(True)
}
```

And here is the complete implementation:

```
class UartCtrlTx(g : UartCtrlGenerics) extends Component {
  import g._

  val io = new Bundle {
    val configFrame  = in(UartCtrlFrameConfig(g))
    val samplingTick = in Bool
    val write        = slave Stream (Bits(dataWidthMax bit))
    val txd          = out Bool
  }

  // Provide one clockDivider.tick each rxSamplePerBit pulse of io.samplingTick
  // Used by the stateMachine as a baud rate time reference
  val clockDivider = new Area {
    val counter = Reg(UInt(log2Up(rxSamplePerBit) bits)) init(0)
    val tick = False
    when(io.samplingTick) {
      counter := counter - 1
      tick := counter === 0
    }
  }

  // Count up each clockDivider.tick, used by the state machine to count up data bits␣
↪and stop bits
  val tickCounter = new Area {
    val value = Reg(UInt(Math.max(dataWidthMax, 2) bit))
    def reset() = value := 0

    when(clockDivider.tick) {
      value := value + 1
    }
  }

  val stateMachine = new Area {
    import UartCtrlTxState._

    val state = RegInit(IDLE)
    val parity = Reg(Bool)
    val txd = True

    when(clockDivider.tick) {
      parity := parity ^ txd
    }

    io.write.ready := False
    switch(state) {
      is(IDLE){
        when(io.write.valid && clockDivider.tick){
```

(continues on next page)

(continued from previous page)

```
          state := START
        }
      }
      is(START) {
        txd := False
        when(clockDivider.tick) {
          state := DATA
          parity := io.configFrame.parity === UartParityType.ODD
          tickCounter.reset()
        }
      }
      is(DATA) {
        txd := io.write.payload(tickCounter.value)
        when(clockDivider.tick) {
          when(tickCounter.value === io.configFrame.dataLength) {
            io.write.ready := True
            tickCounter.reset()
            when(io.configFrame.parity === UartParityType.NONE) {
              state := STOP
            } otherwise {
              state := PARITY
            }
          }
        }
      }
      is(PARITY) {
        txd := parity
        when(clockDivider.tick) {
          state := STOP
          tickCounter.reset()
        }
      }
      is(STOP) {
        when(clockDivider.tick) {
          when(tickCounter.value === toBitCount(io.configFrame.stop)) {
            state := io.write.valid ? START | IDLE
          }
        }
      }
    }
  }

  io.txd := RegNext(stateMachine.txd, True)
}
```

UartCtrlRx

The interfaces of this Component are the following:

Name	Type	Description
configFrame	UartCtrlFrameConfig	Contains data bit width and party/stop bits configurations
samplingTick	Bool	Time reference that pulses rxSamplePerBit times per UART baud
read	Flow[Bits]	Port used by the controller to notify the system about a successfully received frame
rxd	Bool	UART rxd pin, not synchronized with the current clock domain

Let's define the enumeration that will be used to store the state of UartCtrlTx:

```
object UartCtrlRxState extends SpinalEnum {
  val IDLE, START, DATA, PARITY, STOP = newElement()
}
```

Let's define the skeleton of the UartCtrlRx :

```
class UartCtrlRx(g : UartCtrlGenerics) extends Component {
  import g._
  val io = new Bundle {
    val configFrame  = in(UartCtrlFrameConfig(g))
    val samplingTick = in Bool
    val read         = master Flow (Bits(dataWidthMax bit))
    val rxd          = in Bool
  }

  // Implement the rxd sampling with a majority vote over samplingSize bits
  // Provide a new sampler.value each time sampler.tick is high
  val sampler = new Area {
    val syncroniser = BufferCC(io.rxd)
    val samples     = History(that=syncroniser,when=io.samplingTick,
↪length=samplingSize)
    val value       = RegNext(MajorityVote(samples))
    val tick        = RegNext(io.samplingTick)
  }

  // Provide a bitTimer.tick each rxSamplePerBit
  // reset() can be called to recenter the counter over a start bit.
  val bitTimer = new Area {
    val counter = Reg(UInt(log2Up(rxSamplePerBit) bit))
    def reset() = counter := preSamplingSize + (samplingSize - 1) / 2 - 1)
    val tick = False
    ...
  }

  // Provide bitCounter.value that count up each bitTimer.tick, Used by the state␣
↪machine to count data bits and stop bits
  // reset() can be called to reset it to zero
  val bitCounter = new Area {
    val value = Reg(UInt(Math.max(dataWidthMax, 2) bit))
    def reset() = value := 0
```

(continues on next page)

(continued from previous page)

```
    ...
  }

  val stateMachine = new Area {
    import UartCtrlRxState._

    val state   = RegInit(IDLE)
    val parity  = Reg(Bool)
    val shifter = Reg(io.read.payload)
    ...
    switch(state) {
      ...
    }
  }
}
```

And here is the complete implementation:

```
class UartCtrlRx(g : UartCtrlGenerics) extends Component {
  import g._
  val io = new Bundle {
    val configFrame  = in(UartCtrlFrameConfig(g))
    val samplingTick = in Bool
    val read         = master Flow (Bits(dataWidthMax bit))
    val rxd          = in Bool
  }

  // Implement the rxd sampling with a majority vote over samplingSize bits
  // Provide a new sampler.value each time sampler.tick is high
  val sampler = new Area {
    val syncroniser = BufferCC(io.rxd)
    val samples     = History(that=syncroniser,when=io.samplingTick,
↪length=samplingSize)
    val value       = RegNext(MajorityVote(samples))
    val tick        = RegNext(io.samplingTick)
  }

  // Provide a bitTimer.tick each rxSamplePerBit
  // reset() can be called to recenter the counter over a start bit.
  val bitTimer = new Area {
    val counter = Reg(UInt(log2Up(rxSamplePerBit) bit))
    def reset() = counter := preSamplingSize + (samplingSize - 1) / 2 - 1
    val tick = False
    when(sampler.tick) {
      counter := counter - 1
      when(counter === 0) {
        tick := True
      }
    }
  }

  // Provide bitCounter.value that count up each bitTimer.tick, Used by the state␣
↪machine to count data bits and stop bits
  // reset() can be called to reset it to zero
  val bitCounter = new Area {
    val value = Reg(UInt(Math.max(dataWidthMax, 2) bit))
```

(continues on next page)

(continued from previous page)

```
    def reset() = value := 0

    when(bitTimer.tick) {
      value := value + 1
    }
  }

  val stateMachine = new Area {
    import UartCtrlRxState._

    val state   = RegInit(IDLE)
    val parity  = Reg(Bool)
    val shifter = Reg(io.read.payload)

    //Parity calculation
    when(bitTimer.tick) {
      parity := parity ^ sampler.value
    }

    io.read.valid := False
    switch(state) {
      is(IDLE) {
        when(sampler.value === False) {
          state := START
          bitTimer.reset()
        }
      }
      is(START) {
        when(bitTimer.tick) {
          state := DATA
          bitCounter.reset()
          parity := io.configFrame.parity === UartParityType.ODD
          when(sampler.value === True) {
            state := IDLE
          }
        }
      }
      is(DATA) {
        when(bitTimer.tick) {
          shifter(bitCounter.value) := sampler.value
          when(bitCounter.value === io.configFrame.dataLength) {
            bitCounter.reset()
            when(io.configFrame.parity === UartParityType.NONE) {
              state := STOP
            } otherwise {
              state := PARITY
            }
          }
        }
      }
      is(PARITY) {
        when(bitTimer.tick) {
          state := STOP
          bitCounter.reset()
          when(parity =/= sampler.value) {
            state := IDLE
```

(continues on next page)

(continued from previous page)

```
          }
        }
      }
      is(STOP) {
        when(bitTimer.tick) {
          when(!sampler.value) {
            state := IDLE
          }.elsewhen(bitCounter.value === toBitCount(io.configFrame.stop)) {
            state := IDLE
            io.read.valid := True
          }
        }
      }
    }
  }
  io.read.payload := stateMachine.shifter
}
```

UartCtrl

Let's write `UartCtrl` that instantiates the `UartCtrlRx` and `UartCtrlTx` parts, generate the clock divider logic, and connect them to each other.

```
class UartCtrl(g : UartCtrlGenerics = UartCtrlGenerics()) extends Component {
  val io = new Bundle {
    val config = in(UartCtrlConfig(g))
    val write  = slave(Stream(Bits(g.dataWidthMax bit)))
    val read   = master(Flow(Bits(g.dataWidthMax bit)))
    val uart   = master(Uart())
  }

  val tx = new UartCtrlTx(g)
  val rx = new UartCtrlRx(g)

  //Clock divider used by RX and TX
  val clockDivider = new Area {
    val counter = Reg(UInt(g.clockDividerWidth bits)) init(0)
    val tick = counter === 0

    counter := counter - 1
    when(tick) {
      counter := io.config.clockDivider
    }
  }

  tx.io.samplingTick := clockDivider.tick
  rx.io.samplingTick := clockDivider.tick

  tx.io.configFrame := io.config.frame
  rx.io.configFrame := io.config.frame

  tx.io.write << io.write
  rx.io.read >> io.read

  io.uart.txd <> tx.io.txd
```

(continues on next page)

(continued from previous page)

```
  io.uart.rxd <> rx.io.rxd
}
```

Simple usage

To synthesize a `UartCtrl` as `115200-N-8-1`:

```
val uartCtrl: UartCtrl = UartCtrl(
  config = UartCtrlInitConfig(
    baudrate = 115200,
    dataLength = 7,  // 8 bits
    parity = UartParityType.NONE,
    stop = UartStopType.ONE
  )
)
```

If you are using `txd` pin only:

```
uartCtrl.io.uart.rxd := True  // High is the idle state for UART
txd := uartCtrl.io.uart.txd
```

On the contrary, if you are using `rxd` pin only:

```
val uartCtrl: UartCtrl = UartCtrl(
  config = UartCtrlInitConfig(
    baudrate = 115200,
    dataLength = 7,  // 8 bits
    parity = UartParityType.NONE,
    stop = UartStopType.ONE
  ),
  readonly = true
)
```

Example with test bench

Here is a top level example that does the followings things:

- Instantiate `UartCtrl` and set its configuration to 921600 baud/s, no parity, 1 stop bit.
- Each time a byte is received from the UART, it writes it on the leds output.
- Every 2000 cycles, it sends the switches input value to the UART.

```
class UartCtrlUsageExample extends Component{
  val io = new Bundle{
    val uart = master(Uart())
    val switchs = in Bits(8 bits)
    val leds = out Bits(8 bits)
  }

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

(continues on next page)

(continued from previous page)

```
  //Assign io.led with a register loaded each time a byte is received
  io.leds := uartCtrl.io.read.toReg()

  //Write the value of switch on the uart each 2000 cycles
  val write = Stream(Bits(8 bits))
  write.valid := CounterFreeRun(2000).willOverflow
  write.payload := io.switchs
  write >-> uartCtrl.io.write
}


object UartCtrlUsageExample{
  def main(args: Array[String]) {
    SpinalVhdl(new UartCtrlUsageExample,
↪defaultClockDomainFrequency=FixedFrequency(50e6))
  }
}
```

The following example is just a "mad one" but if you want to send a 0x55 header before sending the value of switches, you can replace the write generator of the preceding example by:

```
val write = Stream(Fragment(Bits(8 bits)))
write.valid := CounterFreeRun(4000).willOverflow
write.fragment := io.switchs
write.last := True
write.stage().insertHeader(0x55).toStreamOfFragment >> uartCtrl.io.write
```

Here you can get a simple VHDL testbench for this small `UartCtrlUsageExample`.

Bonus: Having fun with Stream

If you want to queue data received from the UART:

```
val uartCtrl = new UartCtrl()
val queuedReads = uartCtrl.io.read.toStream.queue(16)
```

If you want to add a queue on the write interface and do some flow control:

```
val uartCtrl = new UartCtrl()
val writeCmd = Stream(Bits(8 bits))
val stopIt = Bool
writeCmd.queue(16).haltWhen(stopIt) >> uartCtrl.io.write
```

13.2.3 VGA

Introduction

VGA interfaces are becoming an endangered species, but implementing a VGA controller is still a good exercise.

An explanation about the VGA protocol can be found here.

This VGA controller tutorial is based on this implementation.

Data structures

Before implementing the controller itself we need to define some data structures.

RGB color

First, we need a three channel color structure (Red, Green, Blue). This data structure will be used to feed the controller with pixels and also will be used by the VGA bus.

```
case class RgbConfig(rWidth : Int,gWidth : Int,bWidth : Int){
  def getWidth = rWidth + gWidth + bWidth
}

case class Rgb(c: RgbConfig) extends Bundle{
  val r = UInt(c.rWidth bit)
  val g = UInt(c.gWidth bit)
  val b = UInt(c.bWidth bit)
}
```

VGA bus

io name	Driver	Description
vSync	master	Vertical synchronization, indicate the beginning of a new frame
hSync	master	Horizontal synchronization, indicate the beginning of a new line
colorEn	master	High when the interface is in the visible part
color	master	Carry the color, don't care when colorEn is low

```
case class Vga (rgbConfig: RgbConfig) extends Bundle with IMasterSlave{
  val vSync = Bool()
  val hSync = Bool()

  val colorEn = Bool()
  val color   = Rgb(rgbConfig)

  override def asMaster() : Unit = this.asOutput()
}
```

This Vga `Bundle` uses the `IMasterSlave` trait, which allows you to create master/slave VGA interfaces using the following:

```
master(Vga(...))
slave(Vga(...))
```

VGA timings

The VGA interface is driven by using 8 different timings. Here is one simple example of a `Bundle` that is able to carry them.

```
case class VgaTimings(timingsWidth: Int) extends Bundle {
  val hSyncStart  = UInt(timingsWidth bits)
  val hSyncEnd    = UInt(timingsWidth bits)
  val hColorStart = UInt(timingsWidth bits)
  val hColorEnd   = UInt(timingsWidth bits)
  val vSyncStart  = UInt(timingsWidth bits)
  val vSyncEnd    = UInt(timingsWidth bits)
  val vColorStart = UInt(timingsWidth bits)
  val vColorEnd   = UInt(timingsWidth bits)
}
```

But this not a very good way to specify it because it is redundant for vertical and horizontal timings.

Let's write it in a clearer way:

```
case class VgaTimingsHV(timingsWidth: Int) extends Bundle {
  val colorStart = UInt(timingsWidth bit)
  val colorEnd   = UInt(timingsWidth bit)
  val syncStart  = UInt(timingsWidth bit)
  val syncEnd    = UInt(timingsWidth bit)
}

case class VgaTimings(timingsWidth: Int) extends Bundle {
  val h = VgaTimingsHV(timingsWidth)
  val v = VgaTimingsHV(timingsWidth)
}
```

Then we could add some some functions to set these timings for specific resolutions and frame rates:

```
case class VgaTimingsHV(timingsWidth: Int) extends Bundle {
  val colorStart = UInt(timingsWidth bit)
  val colorEnd   = UInt(timingsWidth bit)
  val syncStart  = UInt(timingsWidth bit)
  val syncEnd    = UInt(timingsWidth bit)
}

case class VgaTimings(timingsWidth: Int) extends Bundle {
  val h = VgaTimingsHV(timingsWidth)
  val v = VgaTimingsHV(timingsWidth)

  def setAs_h640_v480_r60: Unit = {
    h.syncStart := 96 - 1
    h.syncEnd := 800 - 1
    h.colorStart := 96 + 16 - 1
    h.colorEnd := 800 - 48 - 1
    v.syncStart := 2 - 1
    v.syncEnd := 525 - 1
    v.colorStart := 2 + 10 - 1
    v.colorEnd := 525 - 33 - 1
  }

  def setAs_h64_v64_r60: Unit = {
    h.syncStart := 96 - 1
```

(continues on next page)

(continued from previous page)

```
    h.syncEnd := 800 - 1
    h.colorStart := 96 + 16 - 1 + 288
    h.colorEnd := 800 - 48 - 1 - 288
    v.syncStart := 2 - 1
    v.syncEnd := 525 - 1
    v.colorStart := 2 + 10 - 1 + 208
    v.colorEnd := 525 - 33 - 1 - 208
  }
}
```

VGA Controller

Specification

io name	Direction	Description
softReset	in	Reset internal counters and keep the VGA interface inactive
timings	in	Specify VGA horizontal and vertical timings
pixels	slave	Stream of RGB colors that feeds the VGA controller
error	out	High when the pixels stream is too slow
frameStart	out	High when a new frame starts
vga	master	VGA interface

The controller does not integrate any pixel buffering. It directly takes them from the `pixels Stream` and puts them on the `vga.color` out at the right time. If `pixels` is not valid then `error` becomes high for one cycle.

Component and io definition

Let's define a new VgaCtrl `Component`, which takes as `RgbConfig` and `timingsWidth` as parameters. Let's give the bit width a default value of 12.

```
class VgaCtrl(rgbConfig: RgbConfig, timingsWidth: Int = 12) extends Component {
  val io = new Bundle {
    val softReset = in Bool
    val timings = in(VgaTimings(timingsWidth))
    val pixels = slave Stream (Rgb(rgbConfig))

    val error = out Bool
    val frameStart = out Bool
    val vga = master(Vga(rgbConfig))
  }
  ...
}
```

Horizontal and vertical logic

The logic that generates horizontal and vertical synchronization signals is quite the same. It kind of resembles ~PWM~. The horizontal one counts up each cycle, while the vertical one use the horizontal syncronization signal as to increment.

Let's define `HVArea`, which represents one ~PWM~ and then instantiate it two times: one for both horizontal and vertical syncronization.

```
class VgaCtrl(rgbConfig: RgbConfig, timingsWidth: Int = 12) extends Component {
  val io = new Bundle {...}

  case class HVArea(timingsHV: VgaTimingsHV, enable: Bool) extends Area {
    val counter = Reg(UInt(timingsWidth bit)) init(0)

    val syncStart  = counter === timingsHV.syncStart
    val syncEnd    = counter === timingsHV.syncEnd
    val colorStart = counter === timingsHV.colorStart
    val colorEnd   = counter === timingsHV.colorEnd

    when(enable) {
      counter := counter + 1
      when(syncEnd) {
        counter := 0
      }
    }

    val sync    = RegInit(False) setWhen(syncStart) clearWhen(syncEnd)
    val colorEn = RegInit(False) setWhen(colorStart) clearWhen(colorEnd)

    when(io.softReset) {
      counter := 0
      sync    := False
      colorEn := False
    }
  }
  val h = HVArea(io.timings.h, True)
  val v = HVArea(io.timings.v, h.syncEnd)
}
```

As you can see, it's done by using `Area`. This is to avoid the creation of a new `Component` which would have been much more verbose.

Interconnections

Now that we have timing generators for horizontal and vertical synchronization, we need to drive the outputs.

```
class VgaCtrl(rgbConfig: RgbConfig, timingsWidth: Int = 12) extends Component {
  val io = new Bundle {...}

  case class HVArea(timingsHV: VgaTimingsHV, enable: Bool) extends Area {...}
  val h = HVArea(io.timings.h, True)
  val v = HVArea(io.timings.v, h.syncEnd)

  val colorEn = h.colorEn && v.colorEn
  io.pixels.ready := colorEn
  io.error := colorEn && ! io.pixels.valid
```

(continues on next page)

(continued from previous page)

```
  io.frameStart := v.syncEnd

  io.vga.hSync := h.sync
  io.vga.vSync := v.sync
  io.vga.colorEn := colorEn
  io.vga.color := io.pixels.payload
}
```

Bonus

The VgaCtrl that was defined above is generic (not application specific). We can imagine a case where the system provides a `Stream` of `Fragment` of RGB, which means the system transmits pixels between start/end of picture indications.

In this case we can automatically manage the `softReset` input by asserting it when an `error` occurs, then wait for the end of the current `pixels` picture to deassert `error`.

Let's add a function to `VgaCtrl` that can be called from the parent component to feed `VgaCtrl` by using this `Stream` of `Fragment` of RGB.

```
class VgaCtrl(rgbConfig: RgbConfig, timingsWidth: Int = 12) extends Component {
  ...
  def feedWith(that : Stream[Fragment[Rgb]]): Unit ={
    io.pixels << that.toStreamOfFragment

    val error = RegInit(False)
    when(io.error){
      error := True
    }
    when(that.isLast){
      error := False
    }

    io.softReset := error
    when(error){
      that.ready := True
    }
  }
}
```

13.3 Advanced ones

13.3.1 JTAG TAP

Introduction

Important: The goal of this page is to show the implementation of a JTAG TAP (a slave) by a non-conventional way.

Important:

This implementation is not a simple one, it mix object oriented programming, abstract interfaces decoupling, hardware generation and hardware description.
Of course a simple JTAG TAP implementation could be done only with a simple hardware description, but the goal here is really to going forward and creating an very reusable and extensible JTAG TAP generator

---

Important: This page will not explains how JTAG work. A good tutorial could be find there.

One big difference between commonly used HDL and Spinal, is the fact that SpinalHDL allow you to define hardware generators/builders. It's very different than describing hardware. Let's take a look into the example bellow because the difference between generate/build/describing could seem "playing with word" or could be interpreted differently.

The example bellow is a JTAG TAP which allow the JTAG master to read `switchs`/`keys` inputs and write `leds` outputs. This TAP could also be recognized by a master by using the UID 0x87654321.

```
class SimpleJtagTap extends Component {
  val io = new Bundle {
    val jtag    = slave(Jtag())
    val switchs = in  Bits(8 bit)
    val keys    = in  Bits(4 bit)
    val leds    = out Bits(8 bit)
  }

  val tap = new JtagTap(io.jtag, 8)
  val idcodeArea  = tap.idcode(B"x87654321") (instructionId=4)
  val switchsArea = tap.read(io.switchs)     (instructionId=5)
  val keysArea    = tap.read(io.keys)        (instructionId=6)
  val ledsArea    = tap.write(io.leds)       (instructionId=7)
}
```

As you can see, a JtagTap is created but then some Generator/Builder functions (idcode,read,write) are called to create each JTAG instruction. This is what i call "Hardware generator/builder", then these Generator/Builder are used by the user to describing an hardware. And there is the point, in commonly HDL you can only describe your hardware, which imply many donkey job.

This JTAG TAP tutorial is based on this implementation.

JTAG bus

First we need to define a JTAG bus bundle.

```
case class Jtag() extends Bundle with IMasterSlave {
  val tms = Bool()
  val tdi = Bool()
  val tdo = Bool()

  override def asMaster() : Unit = {
    out(tdi, tms)
    in(tdo)
  }
}
```

As you can see this bus don't contain the TCK pin because it will be provided by the clock domain.

JTAG state machine

Let's define the JTAG state machine as explained here

```
object JtagState extends SpinalEnum {
  val RESET, IDLE,
      IR_SELECT, IR_CAPTURE, IR_SHIFT, IR_EXIT1, IR_PAUSE, IR_EXIT2, IR_UPDATE,
      DR_SELECT, DR_CAPTURE, DR_SHIFT, DR_EXIT1, DR_PAUSE, DR_EXIT2, DR_UPDATE =␣
↪newElement()
}

class JtagFsm(jtag: Jtag) extends Area {
  import JtagState._
  val stateNext = JtagState()
  val state = RegNext(stateNext) randBoot()

  stateNext := state.mux(
    default    -> (jtag.tms ? RESET     | IDLE),           //RESET
    IDLE       -> (jtag.tms ? DR_SELECT | IDLE),
    IR_SELECT  -> (jtag.tms ? RESET     | IR_CAPTURE),
    IR_CAPTURE -> (jtag.tms ? IR_EXIT1  | IR_SHIFT),
    IR_SHIFT   -> (jtag.tms ? IR_EXIT1  | IR_SHIFT),
    IR_EXIT1   -> (jtag.tms ? IR_UPDATE | IR_PAUSE),
    IR_PAUSE   -> (jtag.tms ? IR_EXIT2  | IR_PAUSE),
    IR_EXIT2   -> (jtag.tms ? IR_UPDATE | IR_SHIFT),
    IR_UPDATE  -> (jtag.tms ? DR_SELECT | IDLE),
    DR_SELECT  -> (jtag.tms ? IR_SELECT | DR_CAPTURE),
    DR_CAPTURE -> (jtag.tms ? DR_EXIT1  | DR_SHIFT),
    DR_SHIFT   -> (jtag.tms ? DR_EXIT1  | DR_SHIFT),
    DR_EXIT1   -> (jtag.tms ? DR_UPDATE | DR_PAUSE),
    DR_PAUSE   -> (jtag.tms ? DR_EXIT2  | DR_PAUSE),
    DR_EXIT2   -> (jtag.tms ? DR_UPDATE | DR_SHIFT),
    DR_UPDATE  -> (jtag.tms ? DR_SELECT | IDLE)
  )
}
```

Note: The `randBoot()` on `state` make it initialized with a random state. It's only for simulation purpose.

JTAG TAP

Let's implement the core of the JTAG TAP, without any instruction, just the base manage the instruction register (IR) and the bypass.

```
class JtagTap(val jtag: Jtag, instructionWidth: Int) extends Area{
  val fsm = new JtagFsm(jtag)
  val instruction = Reg(Bits(instructionWidth bit))
  val instructionShift = Reg(Bits(instructionWidth bit))
  val bypass = Reg(Bool)

  jtag.tdo := bypass

  switch(fsm.state) {
    is(JtagState.IR_CAPTURE) {
      instructionShift := instruction
    }
```

(continues on next page)

(continued from previous page)

```
    is(JtagState.IR_SHIFT) {
      instructionShift := (jtag.tdi ## instructionShift) >> 1
      jtag.tdo := instructionShift.lsb
    }
    is(JtagState.IR_UPDATE) {
      instruction := instructionShift
    }
    is(JtagState.DR_SHIFT) {
      bypass := jtag.tdi
    }
  }
}
```

Jtag instructions

Now that the JTAG TAP core is done, we can think about how to implement JTAG instructions by an reusable way.

JTAG TAP class interface

First we need to define how an instruction could interact with the JTAG TAP core. We could of course directly take the JtagTap area, but it's not very nice because is some situation the JTAG TAP core is provided by another IP (Altera virtual JTAG for example).

So let's define a simple and abstract interface between the JTAG TAP core and instructions :

```
trait JtagTapAccess {
  def getTdi : Bool()
  def getTms : Bool()
  def setTdo(value : Bool) : Unit

  def getState : JtagState.T
  def getInstruction() : Bits
  def setInstruction(value : Bits) : Unit
}
```

Then let's the JtagTap implement this abstract interface :

```
class JtagTap(val jtag: Jtag, ...) extends Area with JtagTapAccess{
  ...

  //JtagTapAccess impl
  override def getTdi: Bool = jtag.tdi
  override def setTdo(value: Bool): Unit = jtag.tdo := value
  override def getTms: Bool = jtag.tms

  override def getState: JtagState.T = fsm.state
  override def getInstruction(): Bits = instruction
  override def setInstruction(value: Bits): Unit = instruction := value
}
```

Base class

Let's define a useful base class for JTAG instruction that provide some callback (doCapture/doShift/doUpdate/doReset) depending the selected instruction and the state of the JTAG TAP :

```
class JtagInstruction(tap: JtagTapAccess,val instructionId: Bits) extends Area {
  def doCapture(): Unit = {}
  def doShift(): Unit = {}
  def doUpdate(): Unit = {}
  def doReset(): Unit = {}

  val instructionHit = tap.getInstruction === instructionId

  Component.current.addPrePopTask(() => {
    when(instructionHit) {
      when(tap.getState === JtagState.DR_CAPTURE) {
        doCapture()
      }
      when(tap.getState === JtagState.DR_SHIFT) {
        doShift()
      }
      when(tap.getState === JtagState.DR_UPDATE) {
        doUpdate()
      }
    }
    when(tap.getState === JtagState.RESET) {
      doReset()
    }
  })
}
```

Note:

About the Component.current.addPrePopTask(...) :

This allow you to call the given code at the end of the current component construction. Because of object oriented nature of JtagInstruction, doCapture, doShift, doUpdate and doReset should not be called before children classes construction (because children classes will use it as a callback to do some logic)

Read instruction

Let's implement an instruction that allow the JTAG to read a signal.

```
class JtagInstructionRead[T <: Data](data: T) (tap: JtagTapAccess,instructionId:␣
↪Bits)extends JtagInstruction(tap,instructionId) {
  val shifter = Reg(Bits(data.getBitsWidth bit))

  override def doCapture(): Unit = {
    shifter := data.asBits
  }

  override def doShift(): Unit = {
    shifter := (tap.getTdi ## shifter) >> 1
    tap.setTdo(shifter.lsb)
  }
}
```

Write instruction

Let's implement an instruction that allow the JTAG to write a register (and also read its current value).

```
class JtagInstructionWrite[T <: Data](data: T) (tap: JtagTapAccess,instructionId:␣
↪Bits) extends JtagInstruction(tap,instructionId) {
  val shifter,store = Reg(Bits(data.getBitsWidth bit))

  override def doCapture(): Unit = {
    shifter := store
  }
  override def doShift(): Unit = {
    shifter := (tap.getTdi ## shifter) >> 1
    tap.setTdo(shifter.lsb)
  }
  override def doUpdate(): Unit = {
    store := shifter
  }

  data.assignFromBits(store)
}
```

Idcode instruction

Let's implement the instruction that return a idcode to the JTAG and also, when a reset occur, set the instruction register (IR) to it own instructionId.

```
class JtagInstructionIdcode[T <: Data](value: Bits)(tap: JtagTapAccess,␣
↪instructionId: Bits)extends JtagInstruction(tap,instructionId) {
  val shifter = Reg(Bits(32 bit))

  override def doShift(): Unit = {
    shifter := (tap.getTdi ## shifter) >> 1
    tap.setTdo(shifter.lsb)
  }

  override def doReset(): Unit = {
    shifter := value
    tap.setInstruction(instructionId)
  }
}
```

User friendly wrapper

Let's add some user friendly function to the JtagTapAccess to make instructions instantiation easier .

```
trait JtagTapAccess {
  ...

  def idcode(value: Bits)(instructionId: Bits) =
    new JtagInstructionIdcode(value)(this,instructionId)

  def read[T <: Data](data: T)(instructionId: Bits)    =
    new JtagInstructionRead(data)(this,instructionId)
```

(continues on next page)

(continued from previous page)

```
  def write[T <: Data](data: T,  cleanUpdate: Boolean = true, readable: Boolean =␣
↪true)(instructionId: Bits) =
    new JtagInstructionWrite[T](data,cleanUpdate,readable)(this,instructionId)
}
```

Usage demonstration

And there we are, we can now very easily create an application specific JTAG TAP without having to write any logic or any interconnections.

```
class SimpleJtagTap extends Component {
  val io = new Bundle {
    val jtag    = slave(Jtag())
    val switchs = in  Bits(8 bit)
    val keys    = in  Bits(4 bit)
    val leds    = out Bits(8 bit)
  }

  val tap = new JtagTap(io.jtag, 8)
  val idcodeArea  = tap.idcode(B"x87654321") (instructionId=4)
  val switchsArea = tap.read(io.switchs)     (instructionId=5)
  val keysArea    = tap.read(io.keys)        (instructionId=6)
  val ledsArea    = tap.write(io.leds)       (instructionId=7)
}
```

This way of doing things (Generating hardware) could also be applied to, for example, generating an APB/AHB/AXI bus slave.

13.3.2 Memory mapped UART

Introduction

This example will take the `UartCtrl` component implemented in the previous *example* to create a memory mapped UART controller.

Specification

The implementation will be based on the APB3 bus with a RX FIFO.

Here is the register mapping table:

Name	Type	Access	Address	Description
clockDivider	UInt	RW	0	Set the UartCtrl clock divider
frame	UartCtrlFrameConfig	RW	4	Set the dataLength, the parity and the stop bit configuration
writeCmd	Bits	W	8	Send a write command to UartCtrl
writeBusy	Bool	R	8	Bit 0 => zero when a new writeCmd can be sent
read	Bool / Bits	R	12	Bits 7 downto 0 => rx payload Bit 31 => rx payload valid

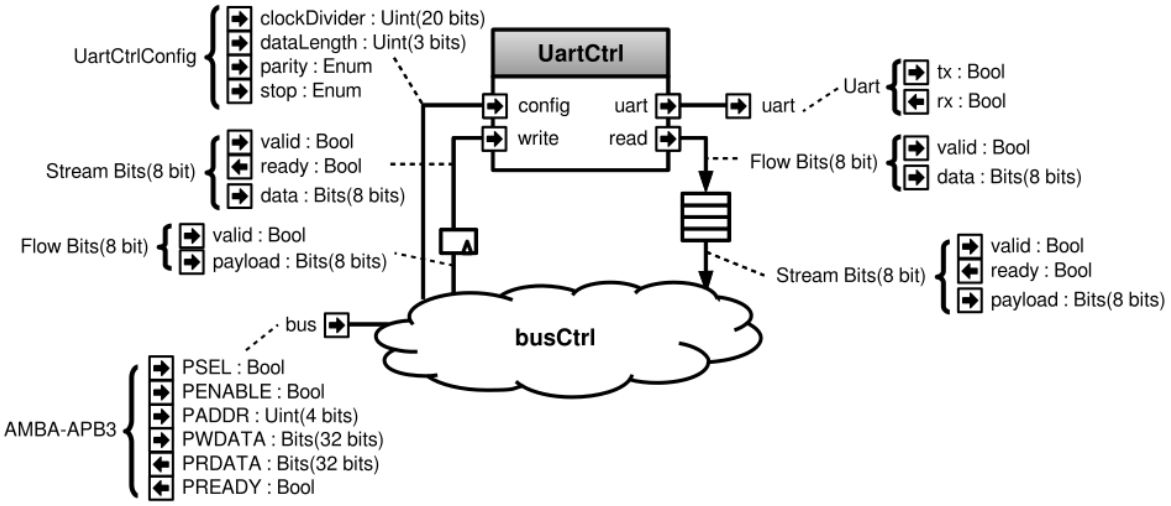
Implementation

For this implementation, the Apb3SlaveFactory tool will be used. It allows you to define a APB3 slave with a nice syntax. You can find the documentation of this tool *there*.

First, we just need to define the `Apb3Config` that will be used for the controller. It is defined in a Scala object as a function to be able to get it from everywhere.

```
object Apb3UartCtrl{
  def getApb3Config = Apb3Config(
    addressWidth = 4,
    dataWidth    = 32
  )
}
```

Then we can define a `Apb3UartCtrl` component which instantiates a `UartCtrl` and creates the memory mapping logic between it and the APB3 bus:




```
class Apb3UartCtrl(uartCtrlConfig : UartCtrlGenerics, rxFifoDepth : Int) extends␣
↪Component{
  val io = new Bundle{
    val bus =  slave(Apb3(Apb3UartCtrl.getApb3Config))
    val uart = master(Uart())
  }

  // Instanciate an simple uart controller
  val uartCtrl = new UartCtrl(uartCtrlConfig)
  io.uart <> uartCtrl.io.uart

  // Create an instance of the Apb3SlaveFactory that will then be used as a slave␣
↪factory drived by io.bus
  val busCtrl = Apb3SlaveFactory(io.bus)

  // Ask the busCtrl to create a readable/writable register at the address 0
  // and drive uartCtrl.io.config.clockDivider with this register
  busCtrl.driveAndRead(uartCtrl.io.config.clockDivider,address = 0)

  // Do the same thing than above but for uartCtrl.io.config.frame at the address 4
  busCtrl.driveAndRead(uartCtrl.io.config.frame,address = 4)

  // Ask the busCtrl to create a writable Flow[Bits] (valid/payload) at the address 8.
  // Then convert it into a stream and connect it to the uartCtrl.io.write by using␣
```

(continues on next page)

(continued from previous page)

```
↪an register stage (>->)
  busCtrl.createAndDriveFlow(Bits(uartCtrlConfig.dataWidthMax bits),address = 8).
↪toStream >-> uartCtrl.io.write

  // To avoid losing writes commands between the Flow to Stream transformation just␣
↪above,
  // make the occupancy of the uartCtrl.io.write readable at address 8
  busCtrl.read(uartCtrl.io.write.valid,address = 8)

  // Take uartCtrl.io.read, convert it into a Stream, then connect it to the input of␣
↪a FIFO of 64 elements
  // Then make the output of the FIFO readable at the address 12 by using a non␣
↪blocking protocol
  // (Bit 7 downto 0 => read data <br> Bit 31 => read data valid )
  busCtrl.readStreamNonBlocking(uartCtrl.io.read.toStream.queue(rxFifoDepth),
                                address = 12, validBitOffset = 31, payloadBitOffset =␣
↪0)
}
```

Important:

Yes, that's all it takes. It's also synthesizable.
The Apb3SlaveFactory tool is not something hard-coded into the SpinalHDL compiler. It's something implemented with SpinalHDL regular hardware description syntax.

13.3.3 Pinesec

Remember to add it

13.3.4 Timer

Introduction

A timer module is probably one of the most basic pieces of hardware. But even for a timer, there are some interesting things that you can do with SpinalHDL. This example will define a simple timer component which integrates a bus bridging utile.

Timer

So let's start with the `Timer` component.

Specification

The `Timer` component will have a single construction parameter:

Parameter Name	Type	Description
width	Int	Specify the bit width of the timer counter

And also some inputs/outputs:

IO Name	Direction	Type	Description
tick	in	Bool	When `tick` is True, the timer count up until `limit`.
clear	in	Bool	When `tick` is True, the timer is set to zero. `clear` has priority over `tick`.
limit	in	UInt(width bits)	When the timer value is equal to `limit`, the `tick` input is inhibited.
full	out	Bool	`full` is high when the timer value is equal to `limit` and `tick` is high.
value	out	UInt(width bits)	Wire out the timer counter value.

Implementation

```
case class Timer(width : Int) extends Component{
  val io = new Bundle{
    val tick      = in Bool()
    val clear     = in Bool()
    val limit     = in UInt(width bits)

    val full  = out Bool()
    val value     = out UInt(width bits)
  }

  val counter = Reg(UInt(width bits))
  when(io.tick && !io.full){
    counter := counter + 1
  }
  when(io.clear){
    counter := 0
  }

  io.full := counter === io.limit && io.tick
  io.value := counter
}
```

Bridging function

Now we can start with the main purpose of this example: defining a bus bridging function. To do that we will use two techniques:

- Using the `BusSlaveFactory` tool documented *here*
- Defining a function inside the `Timer` component which can be called from the parent component to drive the `Timer`'s IO in an abstract way.

Specification

This bridging function will take the following parameters:

Parameter Name	Type	Description
busCtrl	BusSlaveFactory	The `BusSlaveFactory` instance that will be used by the function to create the bridging logic.
baseAddress	BigInt	The base address where the bridging logic should be mapped.
ticks	Seq[Bool]	A list of Bool sources that can be used as a tick signal.
clears	Seq[Bool]	A list of Bool sources that can be used as a clear signal.

The register mapping assumes that the bus system is 32 bits wide:

Name	Access	Width	Address offset	Bit offset	Description
ticksEnable	RW	len(ticks)	0	0	Each `ticks` bool can be actived if the corresponding `ticksEnable` bit is high.
clearsEnable	RW	len(clears)	0	16	Each `clears` bool can be actived if the corresponding `clearsEnable` bit is high.
limit	RW	width	4	0	Access the limit value of the timer component. When this register is written to, the timer is cleared.
value	R	width	8	0	Access the value of the timer.
clear	W		8		When this register is written to, it clears the timer.

Implementation

Let's add this bridging function inside the `Timer` component.

```
case class Timer(width : Int) extends Component{
  val io = new Bundle{
    val tick      = in Bool()
    val clear     = in Bool()
    val limit     = in UInt(width bits)

    val full  = out Bool()
    val value     = out UInt(width bits)
  }

  // Logic previously defined
  // ....

  // The function prototype uses Scala currying funcName(arg1,arg2)(arg3,arg3)
  // which allow to call the function with a nice syntax later
  // This function also returns an area, which allows to keep names of inner signals␣
↪in the generated VHDL/Verilog.
  def driveFrom(busCtrl : BusSlaveFactory,baseAddress : BigInt)(ticks : Seq[Bool],
↪clears : Seq[Bool]) = new Area {
```

(continues on next page)

(continued from previous page)

```
    //Address 0 => clear/tick masks + bus
    val ticksEnable  = busCtrl.createReadWrite(Bits(ticks.length bits),baseAddress +␣
↪0,0) init(0)
    val clearsEnable = busCtrl.createReadWrite(Bits(clears.length bits),baseAddress +␣
↪0,16) init(0)
    val busClearing  = False

    io.clear := (clearsEnable & clears.asBits).orR | busClearing
    io.tick  := (ticksEnable  & ticks.asBits ).orR

    //Address 4 => read/write limit (+ auto clear)
    busCtrl.driveAndRead(io.limit,baseAddress + 4)
    busClearing setWhen(busCtrl.isWriting(baseAddress + 4))

    //Address 8 => read timer value / write => clear timer value
    busCtrl.read(io.value,baseAddress + 8)
    busClearing setWhen(busCtrl.isWriting(baseAddress + 8))
  }
}
```

Usage

Here is some demonstration code which is very close to the one used in the Pinsec SoC timer module. Basically it instantiates following elements:

- One 16 bit prescaler
- One 32 bit timer
- Three 16 bit timers

Then by using an `Apb3SlaveFactory` and functions defined inside the `Timer`s, it creates bridging logic between the APB3 bus and all instantiated components.

```
val io = new Bundle{
  val apb = Apb3(ApbConfig(addressWidth = 8, dataWidth = 32))
  val interrupt = in Bool()
  val external = new Bundle{
    val tick  = Bool()
    val clear = Bool()
  }
}

//Prescaler is very similar to the timer, it mainly integrates a piece of auto reload␣
↪logic.
val prescaler = Prescaler(width = 16)

val timerA = Timer(width = 32)
val timerB,timerC,timerD = Timer(width = 16)

val busCtrl = Apb3SlaveFactory(io.apb)
val prescalerBridge = prescaler.driveFrom(busCtrl,0x00)

val timerABridge = timerA.driveFrom(busCtrl,0x40)(
  // The first element is True, which allows you to have a mode where the timer is␣
↪always counting up.
  ticks  = List(True, prescaler.io.overflow),
```

(continues on next page)

(continued from previous page)

```
  // By looping the timer full to the clears, it allows you to create an autoreload␣
↪mode.
  clears = List(timerA.io.full)
)

val timerBBridge = timerB.driveFrom(busCtrl,0x50)(
  //The external.tick could allow to create an impulsion counter mode
  ticks  = List(True, prescaler.io.overflow, io.external.tick),
  //external.clear could allow to create an timeout mode.
  clears = List(timerB.io.full, io.external.clear)
)

val timerCBridge = timerC.driveFrom(busCtrl,0x60)(
  ticks  = List(True, prescaler.io.overflow, io.external.tick),
  clears = List(timerC.io.full, io.external.clear)
)

val timerDBridge = timerD.driveFrom(busCtrl,0x70)(
  ticks  = List(True, prescaler.io.overflow, io.external.tick),
  clears = List(timerD.io.full, io.external.clear)
)

val interruptCtrl = InterruptCtrl(4)
val interruptCtrlBridge = interruptCtrl.driveFrom(busCtrl,0x10)
interruptCtrl.io.inputs(0) := timerA.io.full
interruptCtrl.io.inputs(1) := timerB.io.full
interruptCtrl.io.inputs(2) := timerC.io.full
interruptCtrl.io.inputs(3) := timerD.io.full
io.interrupt := interruptCtrl.io.pendings.orR
```

13.4 Introduction

Examples are split into three kinds:

- Simple examples that could be used to get used to the basics of SpinalHDL.
- Intermediate examples which implement components by using a traditional approach.
- Advanced examples which go further than traditional HDL by using object-oriented programming, functional programming, and meta-hardware description.

They are all accessible in the sidebar under the corresponding sections.

Important: The SpinalHDL workshop contains many labs with their solutions. See here.

Note: You can also find a list of repostitories using SpinalHDL *here*

CHAPTER

FOURTEEN

LEGACY

14.1 RiscV

Warning: This page document the first RISC-V cpu iteration done in SpinalHDL. The second iteration of this CPU is available there and already offer better perforance/area/features.

14.1.1 Features

RISC-V CPU

- Pipelined on 5 stages (Fetch Decode Execute0 Execute1 WriteBack)
- Multiple branch prediction modes : (disable, static or dynamic)
- Data path parameterizable between fully bypassed to fully interlocked

Extensions

- One cycle multiplication
- 34 cycle division
- Iterative shifter (N shift -> N cycles)
- Single cycle shifter
- Interruption controller
- Debugging module (with JTAG bridge, openOCD port and GDB)
- Instruction cache with wrapped burst memory interface, one way
- Data cache with instructions to evict/flush the whole cache or a given address, one way

Performance/Area (on cyclone II)

- small core -> 846 LE, 0.6 DMIPS/Mhz
- debug module (without JTAG) -> 240 LE
- JTAG Avalon master -> 238 LE
- big core with MUL/DIV/Full shifter/I$/Interrupt/Debug -> 2200 LE, 1.15 DMIPS/Mhz, at least 100 Mhz (with default synthesis option)

14.1.2 Base FPGA project

You can find a DE1-SOC project which integrate two instance of the CPU with MUL/DIV/Full shifter/I$/Interrupt/Debug there :

https://drive.google.com/drive/folders/0B-CqLXDTaMbKNkktb2k3T3lzcUk?usp=sharing

CPU/JTAG/VGA IP are pre-generated. Quartus Prime : 15.1.

14.1.3 How to generate the CPU VHDL

Warning: This avalon version of the CPU isn't present in recent releases of SpinalHDL. Please considarate the VexRiscv instead.

14.1.4 How to debug

You can find the openOCD fork there :

https://github.com/Dolu1990/openocd_riscv

An example target configuration file could be find there :

https://github.com/Dolu1990/openocd_riscv/blob/riscv_spinal/tcl/target/riscv_spinal.cfg

Then you can use the RISCV GDB.

14.1.5 Todo

- Documentation
- Optimise instruction/data caches FMax by moving line hit condition forward into combinatorial paths.

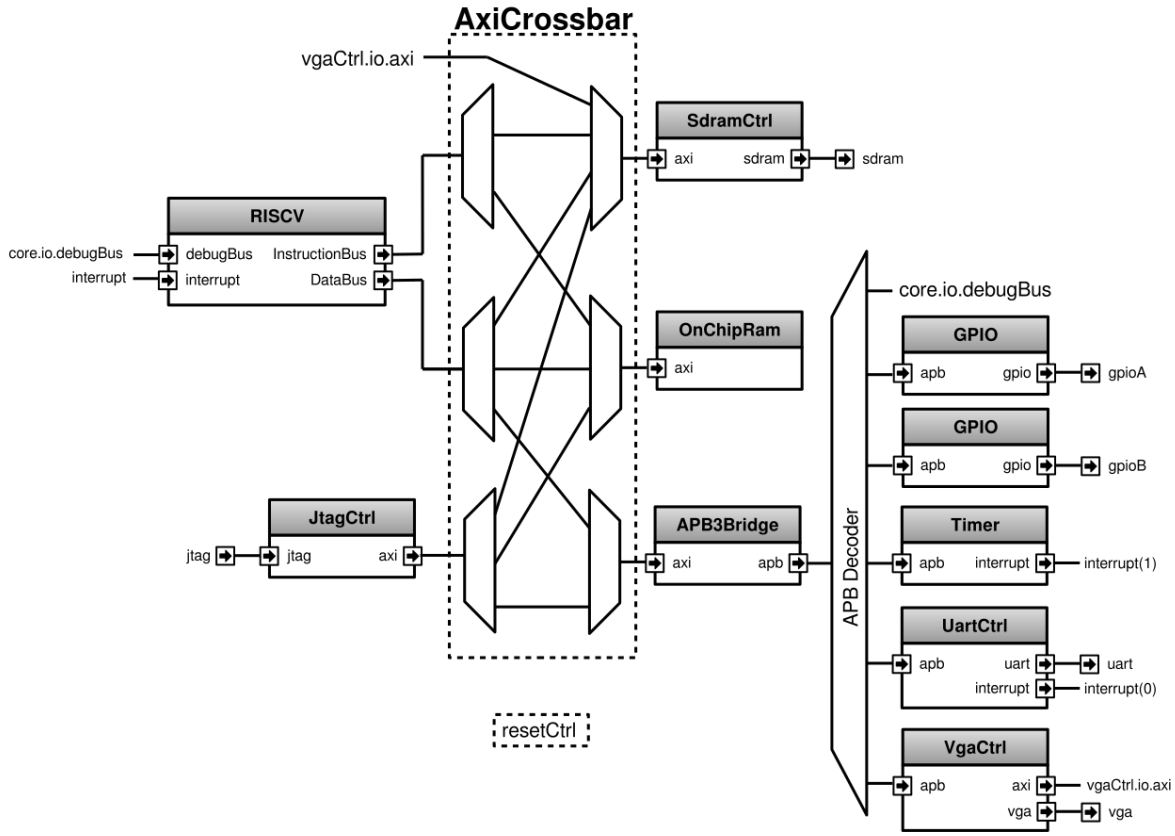
Contact spinalhdl@gmail.com for more information

14.2 pinsec

14.2.1 Hardware

Introduction

There is the Pinsec toplevel hardware diagram :



RISCV

The RISCV is a 5 stage pipelined CPU with following features :

- Instruction cache
- Single cycle Barrel shifter
- Single cycle MUL, 34 cycle DIV
- Interruption support
- Dynamic branch prediction
- Debug port

AXI4

As previously said, Pinsec integrate an AXI4 bus fabric. AXI4 is not the easiest bus on the Earth but has many advantages like :

- A flexible topology
- High bandwidth potential
- Potential out of order request completion
- Easy methods to meets clocks timings
- Standard used by many IP
- An hand-shaking methodology that fit with SpinalHDL Stream.

From an Area utilization perspective, AXI4 is for sure not the lightest solution, but some techniques could dramatically reduce that issue :

- Using Read-Only/Write-Only AXI4 variations where it's possible
- Introducing an Axi4-Shared variation where a new ARW channel is introduced to replace AR and AW channels. This solution divide resources usage by two for the address decoding and the address arbitration.
- Depending the interconnect implementation, if masters doesn't use the R/B channels ready, this path will be removed until each slaves at synthesis, which relax timings.
- As the AXI4 spec suggest, the interconnect can expand the transactions ID by aggregating the corresponding input port id. This allow the interconnect to have an infinite number of pending request and also to support out of order completion with a negligible area cost (transaction id expand).

The Pinsec interconnect doesn't introduce latency cycles.

APB3

In Pinsec, all peripherals implement an APB3 bus to be interfaced. The APB3 choice was motivated by following reasons :

- Very simple bus (no burst)
- Use very few resources
- Standard used by many IP

Generate the RTL

To generate the RTL, you have multiple solutions :

You can download the SpinalHDL source code, and then run :

```
sbt "project SpinalHDL-lib" "run-main spinal.lib.soc.pinsec.Pinsec"
```

Or you can create your own main into your own SBT project and then run it :

```
import spinal.lib.soc.pinsec._

object PinsecMain{
  def main(args: Array[String]) {
    SpinalVhdl(new Pinsec(100 MHz))
    SpinalVerilog(new Pinsec(100 MHz))
  }
}
```

Note: Currently, only the verilog version was tested in simulation and in FPGA because the last release of GHDL is not compatible with cocotb.

14.2.2 SoC toplevel (Pinsec)

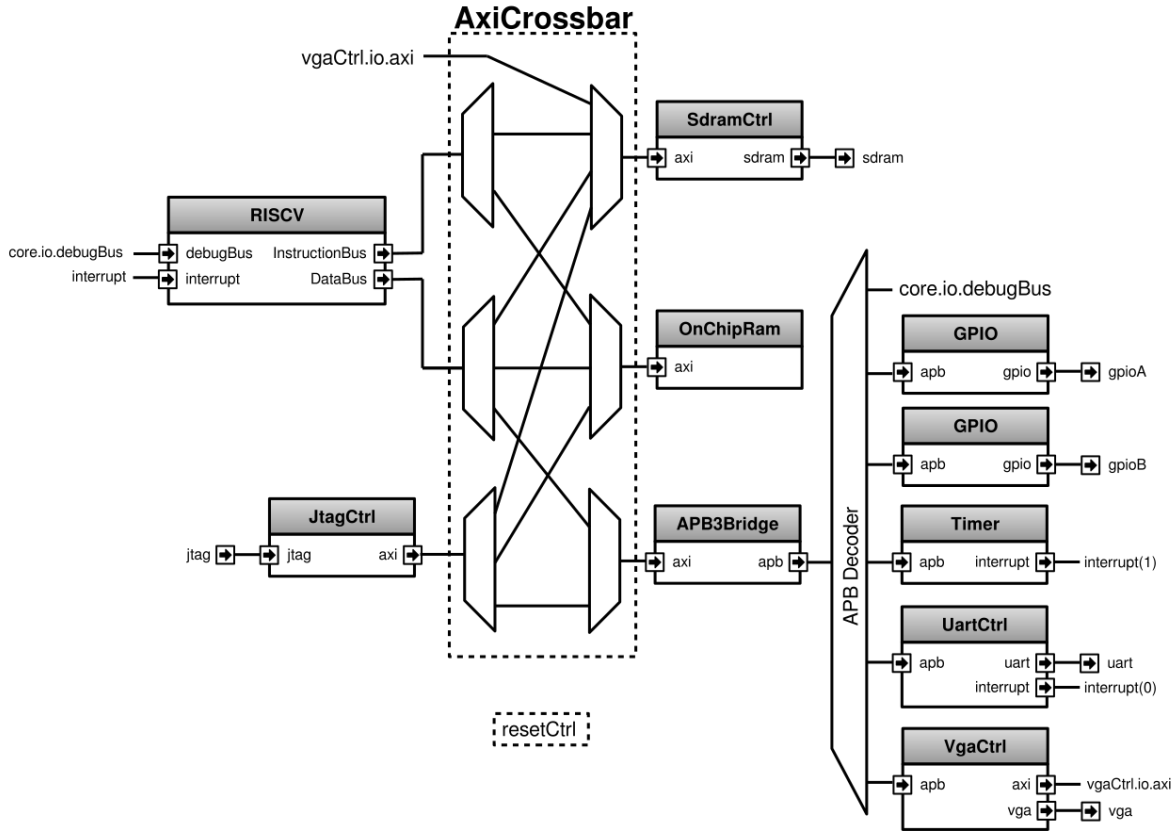
Introduction

`Pinsec` is a little SoC designed for FPGA. It is available in the SpinalHDL library and some documentation could be find *there*

Its toplevel implementation is an interesting example, because it mix some design pattern that make it very easy to modify. Adding a new master or a new peripheral to the bus fabric could be done in the seconde.

This toplevel implementation could be consulted there : https://github.com/SpinalHDL/SpinalHDL/blob/master/lib/src/main/scala/spinal/lib/soc/pinsec/Pinsec.scala

There is the Pinsec toplevel hardware diagram :



Defining all IO

```
val io = new Bundle{
  //Clocks / reset
  val asyncReset = in Bool()
  val axiClk     = in Bool()
  val vgaClk     = in Bool()

  //Main components IO
  val jtag       = slave(Jtag())
  val sdram      = master(SdramInterface(IS42x320D.layout))

  //Peripherals IO
  val gpioA      = master(TriStateArray(32 bits))   //Each pin has it's individual␣
↪output enable control
  val gpioB      = master(TriStateArray(32 bits))
  val uart       = master(Uart())
  val vga        = master(Vga(RgbConfig(5,6,5)))
}
```

Clock and resets

Pinsec has three clocks inputs :

- axiClock
- vgaClock
- jtag.tck

And one reset input :

- asyncReset

Which will finally give 5 ClockDomain (clock/reset couple) :

Name	Clock	Description
resetCtrlClockDomain	axiClock	Used by the reset controller, Flops of this clock domain are initialized by the FPGA bitstream
axiClockDomain	axiClock	Used by all component connected to the AXI and the APB interconnect
coreClockDomain	axiClock	The only difference with the axiClockDomain, is the fact that the reset could also be asserted by the debug module
vgaClockDomain	vgaClock	Used by the VGA controller backend as a pixel clock
jtagClockDomain	jtag.tck	Used to clock the frontend of the JTAG controller

Reset controller

First we need to define the reset controller clock domain, which has no reset wire, but use the FPGA bitstream loading to setup flipflops.

```
val resetCtrlClockDomain = ClockDomain(
  clock = io.axiClk,
  config = ClockDomainConfig(
    resetKind = BOOT
  )
)
```

Then we can define a simple reset controller under this clock domain.

```
val resetCtrl = new ClockingArea(resetCtrlClockDomain) {
  val axiResetUnbuffered  = False
  val coreResetUnbuffered = False

  //Implement an counter to keep the reset axiResetOrder high 64 cycles
  // Also this counter will automaticly do a reset when the system boot.
  val axiResetCounter = Reg(UInt(6 bits)) init(0)
  when(axiResetCounter =/= U(axiResetCounter.range -> true)){
    axiResetCounter := axiResetCounter + 1
    axiResetUnbuffered := True
  }
  when(BufferCC(io.asyncReset)){
    axiResetCounter := 0
  }

  //When an axiResetOrder happen, the core reset will as well
  when(axiResetUnbuffered){
```

(continues on next page)

(continued from previous page)

```
    coreResetUnbuffered := True
  }

  //Create all reset used later in the design
  val axiReset  = RegNext(axiResetUnbuffered)
  val coreReset = RegNext(coreResetUnbuffered)
  val vgaReset  = BufferCC(axiResetUnbuffered)
}
```

Systems clock domains

Now that the reset controller is implemented, we can define clock domain for all part of Pinsec :

```
val axiClockDomain = ClockDomain(
  clock     = io.axiClk,
  reset     = resetCtrl.axiReset,
  frequency = FixedFrequency(50 MHz) //The frequency information is used by the SDRAM␣
↪controller
)

val coreClockDomain = ClockDomain(
  clock = io.axiClk,
  reset = resetCtrl.coreReset
)

val vgaClockDomain = ClockDomain(
  clock = io.vgaClk,
  reset = resetCtrl.vgaReset
)

val jtagClockDomain = ClockDomain(
  clock = io.jtag.tck
)
```

Also all the core system of Pinsec will be defined into a `axi` clocked area :

```
val axi = new ClockingArea(axiClockDomain) {
  //Here will come the rest of Pinsec
}
```

Main components

Pinsec is constituted mainly by 4 main components :

- One RISCV CPU
- One SDRAM controller
- One on chip memory
- One JTAG controller

RISCV CPU

The RISCV CPU used in Pinsec as many parametrization possibilities :

```
val core = coreClockDomain {
  val coreConfig = CoreConfig(
    pcWidth = 32,
    addrWidth = 32,
    startAddress = 0x00000000,
    regFileReadyKind = sync,
    branchPrediction = dynamic,
    bypassExecute0 = true,
    bypassExecute1 = true,
    bypassWriteBack = true,
    bypassWriteBackBuffer = true,
    collapseBubble = false,
    fastFetchCmdPcCalculation = true,
    dynamicBranchPredictorCacheSizeLog2 = 7
  )

  //The CPU has a systems of plugin which allow to add new feature into the core.
  //Those extension are not directly implemented into the core, but are kind of␣
↪additive logic patch defined in a separated area.
  coreConfig.add(new MulExtension)
  coreConfig.add(new DivExtension)
  coreConfig.add(new BarrelShifterFullExtension)

  val iCacheConfig = InstructionCacheConfig(
    cacheSize =4096,
    bytePerLine =32,
    wayCount = 1,  //Can only be one for the moment
    wrappedMemAccess = true,
    addressWidth = 32,
    cpuDataWidth = 32,
    memDataWidth = 32
  )

  //There is the instanciation of the CPU by using all those construction parameters
  new RiscvAxi4(
    coreConfig = coreConfig,
    iCacheConfig = iCacheConfig,
    dCacheConfig = null,
    debug = true,
    interruptCount = 2
  )
}
```

On chip RAM

The instanciation of the AXI4 on chip RAM is very simple.

In fact it's not an AXI4 but an Axi4Shared, which mean that a ARW channel replace the AR and AW ones. This solution use less area while being fully interoperable with full AXI4.

```
val ram = Axi4SharedOnChipRam(
  dataWidth = 32,
  byteCount = 4 kB,
  idWidth = 4     //Specify the AXI4 ID width.
)
```

SDRAM controller

First you need to define the layout and timings of your SDRAM device. On the DE1-SOC, the SDRAM device is an IS42x320D one.

```
object IS42x320D {
  def layout = SdramLayout(
    bankWidth   = 2,
    columnWidth = 10,
    rowWidth    = 13,
    dataWidth   = 16
  )

  def timingGrade7 = SdramTimings(
    bootRefreshCount =   8,
    tPOW             = 100 us,
    tREF             =  64 ms,
    tRC              =  60 ns,
    tRFC             =  60 ns,
    tRAS             =  37 ns,
    tRP              =  15 ns,
    tRCD             =  15 ns,
    cMRD             =   2,
    tWR              =  10 ns,
    cWR              =   1
  )
}
```

Then you can used those definition to parametrize the SDRAM controller instantiation.

```
val sdramCtrl = Axi4SharedSdramCtrl(
  axiDataWidth = 32,
  axiIdWidth   = 4,
  layout       = IS42x320D.layout,
  timing       = IS42x320D.timingGrade7,
  CAS          = 3
)
```

JTAG controller

The JTAG controller could be used to access memories and debug the CPU from an PC.

```
val jtagCtrl = JtagAxi4SharedDebugger(SystemDebuggerConfig(
  memAddressWidth = 32,
  memDataWidth    = 32,
  remoteCmdWidth  = 1,
  jtagClockDomain = jtagClockDomain
))
```

Peripherals

Pinsec integrate some peripherals :

- GPIO
- Timer
- UART
- VGA

GPIO

```
val gpioACtrl = Apb3Gpio(
  gpioWidth = 32
)

val gpioBCtrl = Apb3Gpio(
  gpioWidth = 32
)
```

Timer

The Pinsec timer module is constituted of :

- One prescaler
- One 32 bits timer
- Three 16 bits timers

All of them are packed into the PinsecTimerCtrl component.

```
val timerCtrl = PinsecTimerCtrl()
```

UART controller

First we need to define a configuration for our UART controller :

```
val uartCtrlConfig = UartCtrlMemoryMappedConfig(
  uartCtrlConfig = UartCtrlGenerics(
    dataWidthMax      = 8,
    clockDividerWidth = 20,
    preSamplingSize   = 1,
    samplingSize      = 5,
    postSamplingSize  = 2
  ),
  txFifoDepth = 16,
  rxFifoDepth = 16
)
```

Then we can use it to instantiate the UART controller

```
val uartCtrl = Apb3UartCtrl(uartCtrlConfig)
```

VGA controller

First we need to define a configuration for our VGA controller :

```
val vgaCtrlConfig = Axi4VgaCtrlGenerics(
  axiAddressWidth = 32,
  axiDataWidth    = 32,
  burstLength     = 8,           //In Axi words
  frameSizeMax    = 2048*1512*2, //In byte
  fifoSize        = 512,         //In axi words
  rgbConfig       = RgbConfig(5,6,5),
  vgaClock        = vgaClockDomain
)
```

Then we can use it to instantiate the VGA controller

```
val vgaCtrl = Axi4VgaCtrl(vgaCtrlConfig)
```

Bus interconnects

There is three interconnections components :

- AXI4 crossbar
- AXI4 to APB3 bridge
- APB3 decoder

AXI4 to APB3 bridge

This bridge will be used to connect low bandwidth peripherals to the AXI crossbar.

```
val apbBridge = Axi4SharedToApb3Bridge(
  addressWidth = 20,
  dataWidth    = 32,
  idWidth      = 4
)
```

AXI4 crossbar

The AXI4 crossbar that interconnect AXI4 masters and slaves together is generated by using an factory. The concept of this factory is to create it, then call many function on it to configure it, and finaly call the `build` function to ask the factory to generate the corresponding hardware :

```
val axiCrossbar = Axi4CrossbarFactory()
// Where you will have to call function the the axiCrossbar factory to populate its␣
↪configuration
axiCrossbar.build()
```

First you need to populate slaves interfaces :

```
//          Slave  -> (base address,  size) ,

axiCrossbar.addSlaves(
  ram.io.axi       -> (0x00000000L,   4 kB),
  sdramCtrl.io.axi -> (0x40000000L,  64 MB),
  apbBridge.io.axi -> (0xF0000000L,   1 MB)
)
```

Then you need to populate interconnections between slaves and masters :

```
//         Master -> List of slaves which are accessible

axiCrossbar.addConnections(
  core.io.i       -> List(ram.io.axi, sdramCtrl.io.axi),
  core.io.d       -> List(ram.io.axi, sdramCtrl.io.axi, apbBridge.io.axi),
  jtagCtrl.io.axi -> List(ram.io.axi, sdramCtrl.io.axi, apbBridge.io.axi),
  vgaCtrl.io.axi  -> List(            sdramCtrl.io.axi)
)
```

Then to reduce combinatorial path length and have a good design FMax, you can ask the factory to insert pipelining stages between itself a given master or slave :

Note:

`halfPipe` / >> / << / >/-> in the following code are provided by the Stream bus library.
Some documentation could be find *there*. In short, it's just some pipelining and interconnection stuff.

```
//Pipeline the connection between the crossbar and the apbBridge.io.axi
axiCrossbar.addPipelining(apbBridge.io.axi,(crossbar,bridge) => {
  crossbar.sharedCmd.halfPipe() >> bridge.sharedCmd
  crossbar.writeData.halfPipe() >> bridge.writeData
  crossbar.writeRsp             << bridge.writeRsp
```

(continues on next page)

(continued from previous page)

```
  crossbar.readRsp              << bridge.readRsp
})

//Pipeline the connection between the crossbar and the sdramCtrl.io.axi
axiCrossbar.addPipelining(sdramCtrl.io.axi,(crossbar,ctrl) => {
  crossbar.sharedCmd.halfPipe()  >>  ctrl.sharedCmd
  crossbar.writeData            >/-> ctrl.writeData
  crossbar.writeRsp              <<  ctrl.writeRsp
  crossbar.readRsp               <<  ctrl.readRsp
})
```

APB3 decoder

The interconnection between the APB3 bridge and all peripherals is done via an APB3Decoder :

```
val apbDecoder = Apb3Decoder(
  master = apbBridge.io.apb,
  slaves = List(
    gpioACtrl.io.apb -> (0x00000, 4 kB),
    gpioBCtrl.io.apb -> (0x01000, 4 kB),
    uartCtrl.io.apb  -> (0x10000, 4 kB),
    timerCtrl.io.apb -> (0x20000, 4 kB),
    vgaCtrl.io.apb   -> (0x30000, 4 kB),
    core.io.debugBus -> (0xF0000, 4 kB)
  )
)
```

Misc

To connect all toplevel IO to components, the following code is required :

```
io.gpioA <> axi.gpioACtrl.io.gpio
io.gpioB <> axi.gpioBCtrl.io.gpio
io.jtag  <> axi.jtagCtrl.io.jtag
io.uart  <> axi.uartCtrl.io.uart
io.sdram <> axi.sdramCtrl.io.sdram
io.vga   <> axi.vgaCtrl.io.vga
```

And finally some connections between components are required like interrupts and core debug module resets

```
core.io.interrupt(0) := uartCtrl.io.interrupt
core.io.interrupt(1) := timerCtrl.io.interrupt

core.io.debugResetIn := resetCtrl.axiReset
when(core.io.debugResetOut){
  resetCtrl.coreResetUnbuffered := True
}
```

14.2.3 Introduction

Note: This page document the SoC implemented with the first RISC-V cpu iteration done in SpinalHDL. The second iteration of this SoC (and CPU) is available there and offer better perforance/area/features.

Introduction

Pinsec is the name of a little FPGA SoC fully written in SpinalHDL. Goals of this project are multiple :

- Prove that SpinalHDL is a viable HDL alternative in non-trivial projects.
- Show advantage of SpinalHDL meta-hardware description capabilities in a concrete project.
- Provide a fully open source SoC.

Pinsec has followings hardware features:

- AXI4 interconnect for high speed busses
- APB3 interconnect for peripherals
- RISCV CPU with instruction cache, MUL/DIV extension and interrupt controller
- JTAG bridge to load binaries and debug the CPU
- SDRAM SDR controller
- On chip ram
- One UART controller
- One VGA controller
- Some timer module
- Some GPIO

The toplevel code explanation could be find *there*

Board support

A DE1-SOC FPGA project can be find there with some demo binaries.

14.2.4 Software

RISCV tool-chain

Binaries executed by the CPU can be defined in ASM/C/C++ and compiled by the GCC RISCV fork. Also, to load binaries and debug the CPU, an OpenOCD fork and RISCV GDB can be used.

RISCV tools : https://github.com/riscv/riscv-wiki/wiki/RISC-V-Software-Status
OpenOCD fork : https://github.com/Dolu1990/openocd_riscv
Software examples : https://github.com/Dolu1990/pinsecSoftware

OpenOCD/GDB/Eclipse configuration

About the OpenOCD fork, there is the configuration file that could be used to connect the Pinsec SoC : https://github.com/Dolu1990/openocd_riscv/blob/riscv_spinal/tcl/target/riscv_spinal.cfg

There is an example of arguments used to run the OpenOCD tool :

```
openocd -f ../tcl/interface/ftdi/ft2232h_breakout.cfg -f ../tcl/target/riscv_spinal.
↪cfg -d 3
```

To debug with eclipse, you will need the Zylin plugin and then create an "Zynlin embedded debug (native)".

Initialize commands :

```
target remote localhost:3333
monitor reset halt
load
```

Run commands :

```
continue
```

CHAPTER

FIFTEEN

DEVELOPERS AREA

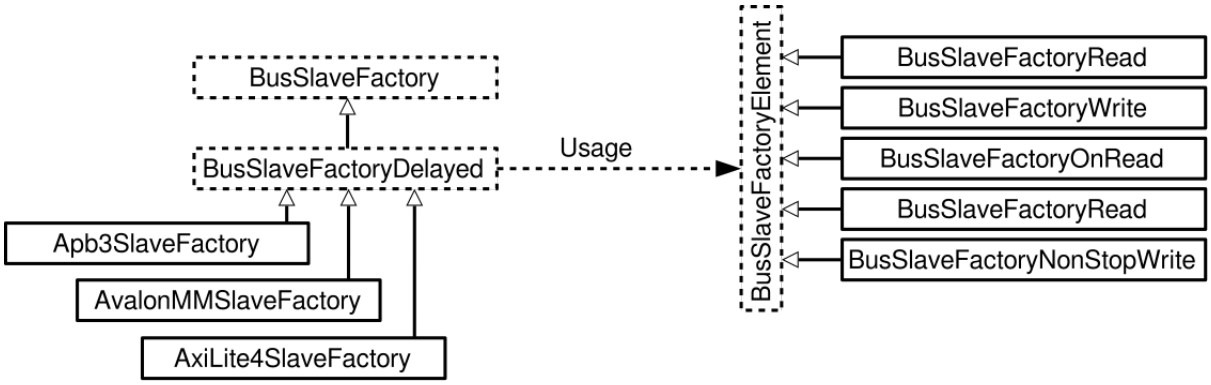
15.1 Bus Slave Factory Implementation

15.1.1 Introduction

This page will document the implementation of the BusSlaveFactory tool and one of those variant. You can get more information about the functionality of that tool *there*.

15.1.2 Specification

The class diagram is the following :



The `BusSlaveFactory` abstract class define minimum requirements that each implementation of it should provide :

Name	Description
busDataWidth	Return the data width of the bus
read(that,address,bitOffset)	When the bus read the `address`, fill the response with `that` at `bitOffset`
write(that,address,bitOffset)	When the bus write the `address`, assign `that` with bus's data from `bitOffset`
onWrite(address)(doThat)	Call `doThat` when a write transaction occur on `address`
onRead(address)(doThat)	Call `doThat` when a read transaction occur on `address`
nonStopWrite(that,bitOffset)	Permanently assign `that` by the bus write data from `bitOffset`

By using them the `BusSlaveFactory` should also be able to provide many utilities :

Name	Return	Description
readAndWrite(that,address,bitOffset)		Make `that` readable and writable at `address` and placed at `bitOffset` in the word
readMultiWord(that,address)		Create the memory mapping to read `that` from 'address'. : If `that` is bigger than one word it extends the register on followings addresses
writeMultiWord(that,address)		Create the memory mapping to write `that` at 'address'. : If `that` is bigger than one word it extends the register on followings addresses
createWriteOnly(dataType,address,bitOffset)	T	Create a write only register of type `dataType` at `address` and placed at `bitOffset` in the word
createReadWrite(dataType,address,bitOffset)	T	Create a read write register of type `dataType` at `address` and placed at `bitOffset` in the word
createAndDriveFlow(dataType,address,bitOffset)	Flow[T]	Create a writable Flow register of type `dataType` at `address` and placed at `bitOffset` in the word
drive(that,address,bitOffset)		Drive `that` with a register writable at `address` placed at `bitOffset` in the word
driveAndRead(that,address,bitOffset)		Drive `that` with a register writable and readable at `address` placed at `bitOffset` in the word
driveFlow(that,address,bitOffset)		Emit on `that` a transaction when a write happen at `address` by using data placed at `bitOffset` in the word
readStreamNonBlocking(that,address,validBitOffset,payloadBitOffset)		Read `that` and consume the transaction when a read happen at `address`. valid <= validBitOffset bit payload <= payloadBitOffset+widthOf(payload) downto `payloadBitOffset`
doBitsAccumulationAndClearOnRead(that,address,bitOffset)		Instanciate an internal register which at each cycle do : reg := reg | that Then when a read occur, the register is cleared. This register is readable at `address` and placed at `bitOffset` in the word

About `BusSlaveFactoryDelayed`, it's still an abstract class, but it capture each primitives (BusSlaveFactoryElement) calls into a data-model. This datamodel is one list that contain all primitives, but also a HashMap that link each address used to a list of primitives that are using it. Then when they all are collected (at the end of the current component), it do a callback that should be implemented by classes that extends it. The implementation of this callback should implement the hardware corresponding to all primitives collected.

15.1.3 Implementation

BusSlaveFactory

Let's describe primitives abstract function :

```
trait BusSlaveFactory  extends Area{

  def busDataWidth : Int

  def read(that : Data,
           address : BigInt,
           bitOffset : Int = 0) : Unit

  def write(that : Data,
            address : BigInt,
            bitOffset : Int = 0) : Unit

  def onWrite(address : BigInt)(doThat : => Unit) : Unit
  def onRead (address : BigInt)(doThat : => Unit) : Unit

  def nonStopWrite( that : Data,
                    bitOffset : Int = 0) : Unit

  //...
}
```

Then let's operate the magic to implement all utile based on them :

```
trait BusSlaveFactory  extends Area{
  //...
  def readAndWrite(that : Data,
                   address: BigInt,
                   bitOffset : Int = 0): Unit = {
    write(that,address,bitOffset)
    read(that,address,bitOffset)
  }

  def drive(that : Data,
            address : BigInt,
            bitOffset : Int = 0) : Unit = {
    val reg = Reg(that)
    write(reg,address,bitOffset)
    that := reg
  }

  def driveAndRead(that : Data,
                   address : BigInt,
                   bitOffset : Int = 0) : Unit = {
    val reg = Reg(that)
    write(reg,address,bitOffset)
    read(reg,address,bitOffset)
    that := reg
  }

  def driveFlow[T <: Data](that : Flow[T],
                           address: BigInt,
                           bitOffset : Int = 0) : Unit = {
```

(continues on next page)

(continued from previous page)

```
    that.valid := False
    onWrite(address){
      that.valid := True
    }
    nonStopWrite(that.payload,bitOffset)
  }

  def createReadWrite[T <: Data](dataType: T,
                                 address: BigInt,
                                 bitOffset : Int = 0): T = {
    val reg = Reg(dataType)
    write(reg,address,bitOffset)
    read(reg,address,bitOffset)
    reg
  }

  def createAndDriveFlow[T <: Data](dataType : T,
                                 address: BigInt,
                                 bitOffset : Int = 0) : Flow[T] = {
    val flow = Flow(dataType)
    driveFlow(flow,address,bitOffset)
    flow
  }

  def doBitsAccumulationAndClearOnRead(   that : Bits,
                                          address : BigInt,
                                          bitOffset : Int = 0): Unit = {
    assert(that.getWidth <= busDataWidth)
    val reg = Reg(that)
    reg := reg | that
    read(reg,address,bitOffset)
    onRead(address){
      reg := that
    }
  }

  def readStreamNonBlocking[T <: Data] (that : Stream[T],
                                        address: BigInt,
                                        validBitOffset : Int,
                                        payloadBitOffset : Int) : Unit = {
    that.ready := False
    onRead(address){
      that.ready := True
    }
    read(that.valid  ,address,validBitOffset)
    read(that.payload,address,payloadBitOffset)
  }

  def readMultiWord(that : Data,
                address : BigInt) : Unit  = {
    val wordCount = (widthOf(that) - 1) / busDataWidth + 1
    val valueBits = that.asBits.resize(wordCount*busDataWidth)
    val words = (0 until wordCount).map(id => valueBits(id * busDataWidth ,␣
↪busDataWidth bit))
    for (wordId <- (0 until wordCount)) {
      read(words(wordId), address + wordId*busDataWidth/8)
```

(continues on next page)

(continued from previous page)

```
    }
  }

  def writeMultiWord(that : Data,
                     address : BigInt) : Unit  = {
    val wordCount = (widthOf(that) - 1) / busDataWidth + 1
    for (wordId <- (0 until wordCount)) {
      write(
        that = new DataWrapper{
          override def getBitsWidth: Int =
            Math.min(busDataWidth, widthOf(that) - wordId * busDataWidth)

          override def assignFromBits(value : Bits): Unit = {
            that.assignFromBits(
              bits     = value.resized,
              offset   = wordId * busDataWidth,
              bitCount = getBitsWidth bits)
          }
        },address = address + wordId * busDataWidth / 8,0
      )
    }
  }
}
```

BusSlaveFactoryDelayed

Let's implement classes that will be used to store primitives :

```
trait BusSlaveFactoryElement

// Ask to make `that` readable when a access is done on `address`.
// bitOffset specify where `that` is placed on the answer
case class BusSlaveFactoryRead(that : Data,
                               address : BigInt,
                               bitOffset : Int) extends BusSlaveFactoryElement

// Ask to make `that` writable when a access is done on `address`.
// bitOffset specify where `that` get bits from the request
case class BusSlaveFactoryWrite(that : Data,
                                address : BigInt,
                                bitOffset : Int) extends BusSlaveFactoryElement

// Ask to execute `doThat` when a write access is done on `address`
case class BusSlaveFactoryOnWrite(address : BigInt,
                                  doThat : () => Unit) extends BusSlaveFactoryElement

// Ask to execute `doThat` when a read access is done on `address`
case class BusSlaveFactoryOnRead( address : BigInt,
                                  doThat : () => Unit) extends BusSlaveFactoryElement

// Ask to constantly drive `that` with the data bus
// bitOffset specify where `that` get bits from the request
case class BusSlaveFactoryNonStopWrite(that : Data,
                                       bitOffset : Int) extends BusSlaveFactoryElement
```

Then let's implement the `BusSlaveFactoryDelayed` itself :

```
trait BusSlaveFactoryDelayed extends BusSlaveFactory{
  // elements is an array of all BusSlaveFactoryElement requested
  val elements = ArrayBuffer[BusSlaveFactoryElement]()


  // elementsPerAddress is more structured than elements, it group all␣
↪BusSlaveFactoryElement per requested addresses
  val elementsPerAddress = collection.mutable.HashMap[BigInt,
↪ArrayBuffer[BusSlaveFactoryElement]]()

  private def addAddressableElement(e : BusSlaveFactoryElement,address : BigInt) = {
    elements += e
    elementsPerAddress.getOrElseUpdate(address,␣
↪ArrayBuffer[BusSlaveFactoryElement]()) += e
  }

  override def read(that : Data,
            address : BigInt,
            bitOffset : Int = 0) : Unit  = {
    assert(bitOffset + that.getBitsWidth <= busDataWidth)
    addAddressableElement(BusSlaveFactoryRead(that,address,bitOffset),address)
  }

  override def write(that : Data,
             address : BigInt,
             bitOffset : Int = 0) : Unit  = {
    assert(bitOffset + that.getBitsWidth <= busDataWidth)
    addAddressableElement(BusSlaveFactoryWrite(that,address,bitOffset),address)
  }

  def onWrite(address : BigInt)(doThat : => Unit) : Unit = {
    addAddressableElement(BusSlaveFactoryOnWrite(address,() => doThat),address)
  }
  def onRead (address : BigInt)(doThat : => Unit) : Unit = {
    addAddressableElement(BusSlaveFactoryOnRead(address,() => doThat),address)
  }

  def nonStopWrite( that : Data,
                    bitOffset : Int = 0) : Unit = {
    assert(bitOffset + that.getBitsWidth <= busDataWidth)
    elements += BusSlaveFactoryNonStopWrite(that,bitOffset)
  }

  //This is the only thing that should be implement by class that extends␣
↪BusSlaveFactoryDelayed
  def build() : Unit

  component.addPrePopTask(() => build())
}
```

AvalonMMSlaveFactory

First let's implement the companion object that provide the compatible AvalonMM configuration object that correspond to the following table :

Pin name	Type	Description
read	Bool	High one cycle to produce a read request
write	Bool	High one cycle to produce a write request
address	UInt(addressWidth bits)	Byte granularity but word aligned
writeData	Bits(dataWidth bits)	
readDataValid	Bool	High to respond a read command
readData	Bool(dataWidth bits)	Valid when readDataValid is high

```
object AvalonMMSlaveFactory{
  def getAvalonConfig( addressWidth : Int,
                       dataWidth : Int) = {
    AvalonMMConfig.pipelined(   //Create a simple pipelined configuration of the␣
↪Avalon Bus
      addressWidth = addressWidth,
      dataWidth = dataWidth
    ).copy(                     //Change some parameters of the configuration
      useByteEnable = false,
      useWaitRequestn = false
    )
  }

  def apply(bus : AvalonMM) = new AvalonMMSlaveFactory(bus)
}
```

Then, let's implement the AvalonMMSlaveFactory itself.

```
class AvalonMMSlaveFactory(bus : AvalonMM) extends BusSlaveFactoryDelayed{
  assert(bus.c == AvalonMMSlaveFactory.getAvalonConfig(bus.c.addressWidth,bus.c.
↪dataWidth))

  val readAtCmd = Flow(Bits(bus.c.dataWidth bits))
  val readAtRsp = readAtCmd.stage()

  bus.readDataValid := readAtRsp.valid
  bus.readData := readAtRsp.payload

  readAtCmd.valid := bus.read
  readAtCmd.payload := 0

  override def build(): Unit = {
    for(element <- elements) element match {
      case element : BusSlaveFactoryNonStopWrite =>
        element.that.assignFromBits(bus.writeData(element.bitOffset, element.that.
↪getBitsWidth bits))
      case _ =>
    }

    for((address,jobs) <- elementsPerAddress){
      when(bus.address === address){
        when(bus.write){
          for(element <- jobs) element match{
            case element : BusSlaveFactoryWrite => {
```

(continues on next page)

(continued from previous page)

```
              element.that.assignFromBits(bus.writeData(element.bitOffset, element.
↪that.getBitsWidth bits))
            }
            case element : BusSlaveFactoryOnWrite => element.doThat()
            case _ =>
          }
        }
        when(bus.read){
          for(element <- jobs) element match{
            case element : BusSlaveFactoryRead => {
              readAtCmd.payload(element.bitOffset, element.that.getBitsWidth bits) :=␣
↪element.that.asBits
            }
            case element : BusSlaveFactoryOnRead => element.doThat()
            case _ =>
          }
        }
      }
    }
  }

  override def busDataWidth: Int = bus.c.dataWidth
}
```

15.1.4 Conclusion

That's all, you can check one example that use this `Apb3SlaveFactory` to create an Apb3UartCtrl *there*.

If you want to add the support of a new memory bus, it's very simple you just need to implement another variation of the `BusSlaveFactoryDelayed` trait. The `Apb3SlaveFactory` is probably a good starting point :D

15.2 How to HACK this documentation

If you want to add your page to this documentation you need to add your source file in the appropriate section. I opted to create a structure that resample the various section of the documentation, this is not strictly necessary, but for clarity sake, highly encourage.

This documentation uses a recursive index tree: every folder have a special `index.rst` files that tell sphinx witch file, and in what order put it in the documentation tree.

15.2.1 Title convention

Sphinx is very smart, the document structure is deduced from how you use non alphanumerical characters (like: `= - ` : ' " ~ ^ _ * + # < >`), you only need to be consistent. Still, for consistency sakes we use this progression:

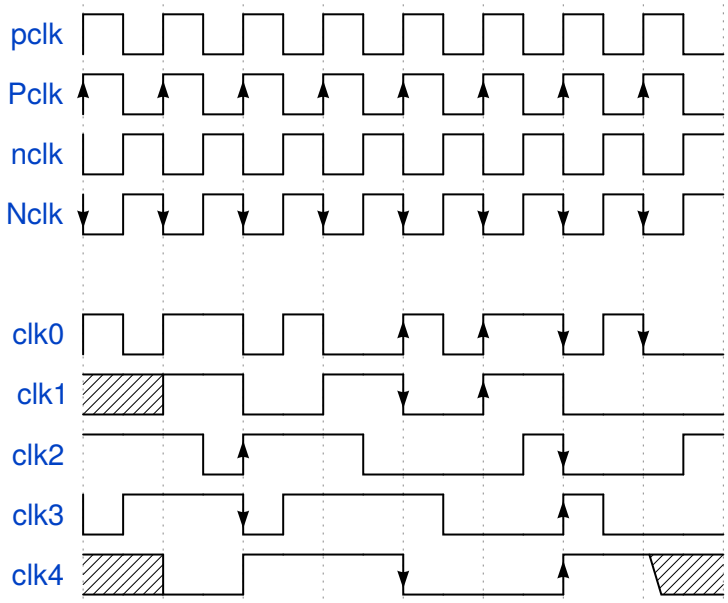
- `=` over and underline for section titles
- `=` underline for titles
- `-` underline for paragraph
- `^` for subparagraph

15.2.2 Wavedrom integration

This documentation makes use of the `sphinxcontrib-wavedrom` plugin, So you can specify a timing diagram, or a register description with the WaveJSON syntax like so:

```
.. wavedrom::

   { "signal": [
      { "name": "pclk", "wave": 'p.......' },
      { "name": "Pclk", "wave": 'P.......' },
      { "name": "nclk", "wave": 'n.......' },
      { "name": "Nclk", "wave": 'N.......' },
      {},
      { "name": 'clk0', "wave": 'phnlPHNL' },
      { "name": 'clk1', "wave": 'xhlhLHl.' },
      { "name": 'clk2', "wave": 'hpHplnLn' },
      { "name": 'clk3', "wave": 'nhNhplPl' },
      { "name": 'clk4', "wave": 'xlh.L.Hx' },
   ]}
```

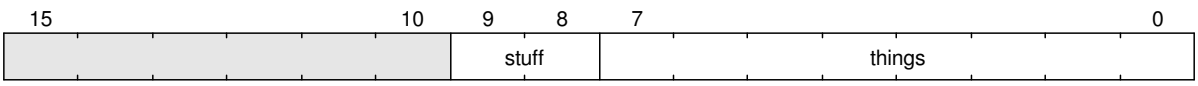
and you get:



Note: if you want the Wavedrom diagram to be present in the pdf export, you need to use the "non relaxed" JSON dialect. long story short, no javascript code and use " around key value (Eg. `"name"`).

you can describe register mapping with the same syntax:

```
{"reg":[
  {"bits": 8, "name": "things"},
  {"bits": 2, "name": "stuff" },
  {"bits": 6},
 ],
 "config": { "bits":16,"lanes":1 }
 }
```



15.2.3 New section

if you want to add a new section you need to specify in the top index, the index file of the new section. I suggest to name the folder like the section name, but is not required; Sphinx will take the name of the section from the title of the index file.

example

I want to document the new feature in SpinalHDL, and I want to create a section for it; let's call it `Cheese`

So I need to create a folder named `Cheese` (name is not important), and in it create a index file like:

```
======
Cheese
======

.. toctree::
:glob:

introduction
*
```

Note: The `.. toctree::` directive accept some parameters, in this case `:glob:` makes so you can use the `*` to include all the remaining files.

Note: The file path is relative to the index file, if you want to specify the absolute path, you need to prepend `/`

Note: `introduction.rst` will be always the first on the list because it's specified in the index file. Other files will be included in alphabetical order.

Now I can add the `introduction.rst` and other files like `cheddar.rst`, `stilton.rst`, etc.

The only thing remaining to do is to add cheese to the top index file like so:

```
Welcome to SpinalHDL's documentation!
=====================================

.. toctree::
   :maxdepth: 2
   :titlesonly:

   rst/About SpinalHDL/index
   rst/Getting Started/index
   rst/Data types/index
   rst/Structuring/index
   rst/Semantic/index
   rst/Sequential logic/index
   rst/Design errors/index
   rst/Other language features/index
   rst/Libraries/index
   rst/Simulation/index
   rst/Examples/index
   rst/Legacy/index
```

(continues on next page)

(continued from previous page)

```
   rst/Developers area/index
   rst/Cheese/index
```

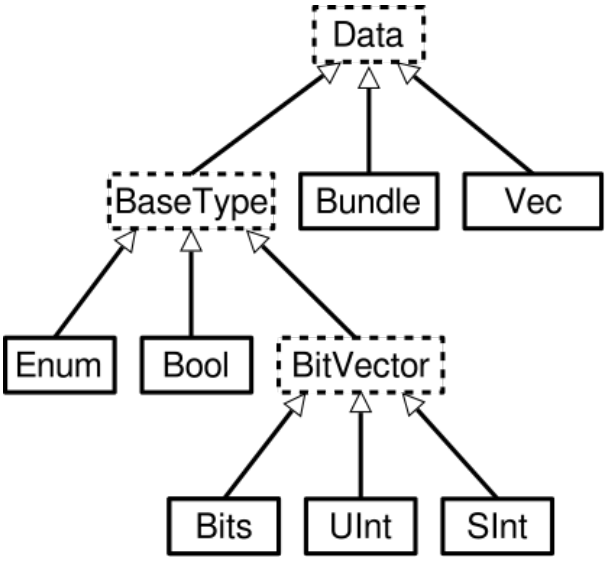
that's it, now you can add all you want in cheese and all pages will show up in the documentation.

15.3 Types

15.3.1 Introduction

The language provides 5 base types and 2 composite types that can be used.

- Base types : `Bool`, `Bits`, `UInt` for unsigned integers, `SInt` for signed integers, `Enum`.
- Composite types : Bundle, Vec.



Those types and their usage (with examples) are explained hereafter.

About the fixed point support it's documented *there*

15.3.2 Bool

This is the standard *boolean* type that correspond to a bit.

Declaration

The syntax to declare such as value is as follows:

Syntax	Description	Return
Bool()	Create a Bool	Bool
True	Create a Bool assigned with `true`	Bool
False	Create a Bool assigned with `false`	Bool
Bool(value : Boolean)	Create a Bool assigned with a Scala Boolean	Bool

Using this type into SpinalHDL yields:

```
val myBool = Bool()
myBool := False          // := is the assignment operator
myBool := Bool(false)    // Use a Scala Boolean to create a literal
```

Operators

The following operators are available for the `Bool` type

Operator	Description	Return type
!x	Logical NOT	Bool
x && y x & y	Logical AND	Bool
x || y x | y	Logical OR	Bool
x ^ y	Logical XOR	Bool
x.set[()]	Set x to True	
x.clear[()]	Set x to False	
x.rise[()]	Return True when x was low at the last cycle and is now high	Bool
x.rise(initAt : Bool)	Same as x.rise but with a reset value	Bool
x.fall[()]	Return True when x was high at the last cycle and is now low	Bool
x.fall(initAt : Bool)	Same as x.fall but with a reset value	Bool
x.setWhen(cond)	Set x when cond is True	Bool
x.clearWhen(cond)	Clear x when cond is True	Bool

15.3.3 The BitVector family - (`Bits`, `UInt`, `SInt`)

`BitVector` is a family of types for storing multiple bits of information in a single value. This type has three subtypes that can be used to model different behaviours:
`Bits` do not convey any sign information whereas the `UInt` (unsigned integer) and `SInt` (signed integer) provide the required operations to compute correct results if signed / unsigned arithmetics is used.

Declaration syntax

Syntax	Description	Return
Bits/UInt/SInt [()]	Create a BitVector, bits count is inferred	Bits/UInt/SInt
Bits/UInt/SInt(x bits)	Create a BitVector with x bits	Bits/UInt/SInt
B/U/S(value : Int[,width : BitCount])	Create a BitVector assigned with 'value'	Bits/UInt/SInt
B/U/S"[[size']base]value"	Create a BitVector assigned with 'value'	Bits/UInt/SInt
B/U/S([x bits], element, ...)	Create a BitVector assigned with the value specified by elements (see bellow table)	Bits/UInt/SInt

Elements could be defined as follows:

Element syntax	Description
x : Int -> y : Boolean/Bool	Set bit x with y
x : Range -> y : Boolean/Bool	Set each bits in range x with y
x : Range -> y : T	Set bits in range x with y
x : Range -> y : String	Set bits in range x with y The string format follow same rules than B/U/S"xyz" one
x : Range -> y : T	Set bits in range x with y
default -> y : Boolean/Bool	Set all unconnected bits with the y value. This feature could only be use to do assignments without the U/B/S prefix

You can define a Range values

Range syntax	Description	Width
(x downto y)	[x:y] x >= y	x-y+1
(x to y)	[x:y] x <= y	y-x+1
(x until y)	[x:y[x < y	y-x

```
val myUInt = UInt(8 bits)
myUInt := U(2,8 bits)
myUInt := U(2)
myUInt := U"0000_0101"  // Base per default is binary => 5
myUInt := U"h1A"        // Base could be x (base 16)
                        //               h (base 16)
                        //               d (base 10)
                        //               o (base 8)
                        //               b (base 2)
myUInt := U"8'h1A"
myUInt := 2             // You can use scala Int as literal value

val myBool := myUInt === U(7 -> true,(6 downto 0) -> false)
val myBool := myUInt === U(myUInt.range -> true)

//For assignment purposes, you can omit the B/U/S, which also alow the use of the␣
↪[default -> ???] feature
myUInt := (default -> true)                     //Assign myUInt with "11111111"
myUInt := (myUInt.range -> true)                //Assign myUInt with "11111111"
myUInt := (7 -> true,default -> false)          //Assign myUInt with "10000000"
myUInt := ((4 downto 1) -> true,default -> false) //Assign myUInt with "00011110"
```

Operators

Operator	Description	Return
~x	Bitwise NOT	T(w(x) bits)
x & y	Bitwise AND	T(max(w(x), w(y) bits)
x | y	Bitwise OR	T(max(w(x), w(y) bits)
x ^ y	Bitwise XOR	T(max(w(x), w(y) bits)
x(y)	Readbit, y : Int/UInt	Bool
x(hi,lo)	Read bitfield, hi : Int, lo : Int	T(hi-lo+1 bits)
x(offset,width)	Read bitfield, offset: UInt, width: Int	T(width bits)
x(y) := z	Assign bits, y : Int/UInt	Bool
x(hi,lo) := z	Assign bitfield, hi : Int, lo : Int	T(hi-lo+1 bits)
x(offset,width) := z	Assign bitfield, offset: UInt, width: Int	T(width bits)
x.msb	Return the most significant bit	Bool
x.lsb	Return the least significant bit	Bool
x.range	Return the range (x.high downto 0)	Range
x.high	Return the upper bound of the type x	Int
x.xorR	XOR all bits of x	Bool
x.orR	OR all bits of x	Bool
x.andR	AND all bits of x	Bool
x.clearAll[()]	Clear all bits	T
x.setAll[()]	Set all bits	T
x.setAllTo(value : Boolean)	Set all bits to the given Boolean value	
x.setAllTo(value : Bool)	Set all bits to the given Bool value	
x.asBools	Cast into a array of Bool	Vec(Bool,width(x))

Masked comparison

Some time you need to check equality between a `BitVector` and a bits constant that contain hole (don't care values).

There is an example about how to do that :

```
val myBits = Bits(8 bits)
val itMatch = myBits === M"00--10--"
```

15.3.4 Bits

Operator	Description	Return
x >> y	Logical shift right, y : Int	T(w(x) - y bits)
x >> y	Logical shift right, y : UInt	T(w(x) bits)
x << y	Logical shift left, y : Int	T(w(x) + y bits)
x << y	Logical shift left, y : UInt	T(w(x) + max(y) bits)
x.rotateLeft(y)	Logical left rotation, y : UInt	T(w(x))
x.resize(y)	Return a resized copy of x, filled with zero, y : Int	T(y bits)

15.3.5 UInt, SInt

Operator	Description	Return
x + y	Addition	T(max(w(x), w(y) bits)
x - y	Subtraction	T(max(w(x), w(y) bits)
x * y	Multiplication	T(w(x) + w(y) bits)
x > y	Greater than	Bool
x >= y	Greater than or equal	Bool
x < y	Less than	Bool
x <= y	Less than or equal	Bool
x >> y	Arithmetic shift right, y : Int	T(w(x) - y bits)
x >> y	Arithmetic shift right, y : UInt	T(w(x) bits)
x << y	Arithmetic shift left, y : Int	T(w(x) + y bits)
x << y	Arithmetic shift left, y : UInt	T(w(x) + max(y) bits)
x.resize(y)	Return an arithmetic resized copy of x, y : Int	T(y bits)

15.3.6 Bool, Bits, UInt, SInt

Operator	Description	Return
x.asBits	Binary cast in Bits	Bits(w(x) bits)
x.asUInt	Binary cast in UInt	UInt(w(x) bits)
x.asSInt	Binary cast in SInt	SInt(w(x) bits)

15.3.7 Vec

Declaration	Description
Vec(type : Data, size : Int)	Create a vector of size time the given type
Vec(x,y,..)	Create a vector where indexes point to given elements. this construct support mixed element width

Operator	Description	Return
x(y)	Read element y, y : Int/UInt	T
x(y) := z	Assign element y with z, y : Int/UInt	

```
val myVecOfSInt = Vec(SInt(8 bits),2)
myVecOfSInt(0) := 2
myVecOfSInt(1) := myVecOfSInt(0) + 3

val myVecOfMixedUInt = Vec(UInt(3 bits), UInt(5 bits), UInt(8 bits))

val x,y,z = UInt(8 bits)
val myVecOf_xyz_ref = Vec(x,y,z)
for(element <- myVecOf_xyz_ref){
  element := 0   //Assign x,y,z with the value 0
}
myVecOf_xyz_ref(1) := 3    //Assign y with the value 3
```

15.3.8 Bundle

Bundles could be used to model data structure line buses and interfaces.
All attributes that extends Data (Bool, Bits, UInt, ...) that are defined inside the bundle are considered as part of the bundle.

Simple example (RGB/VGA)

The following example show an RGB bundle definition with some internal function.

```
case class RGB(channelWidth : Int) extends Bundle{
  val red   = UInt(channelWidth bits)
  val green = UInt(channelWidth bits)
  val blue  = UInt(channelWidth bits)

  def isBlack : Bool = red === 0 && green === 0 && blue === 0
  def isWhite : Bool = {
    val max = U((channelWidth-1 downto 0) -> true)
    return red === max && green === max && blue === max
  }
}
```

Then you can also incorporate a Bundle inside Bundle as deeply as you want:

```
case class VGA(channelWidth : Int) extends Bundle{
  val hsync = Bool
  val vsync = Bool
  val color = RGB(channelWidth)
}
```

And finaly instanciate your Bundles inside the hardware :

```
val vgaIn  = VGA(8)          //Create a RGB instance
val vgaOut = VGA(8)
vgaOut := vgaIn             //Assign the whole bundle
vgaOut.color.green := 0     //Fix the green to zero
val vgaInRgbIsBlack = vgaIn.rgb.isBlack   //Get if the vgaIn rgb is black
```

If you want to specify your bundle as an input or an output of a Component, you have to do it by the following way :

```
class MyComponent extends Component{
  val io = Bundle{
    val cmd = in(RGB(8))    //Don't forget the bracket around the bundle.
    val rsp = out(RGB(8))
  }
}
```

Interface example (APB)

If you want to define an interface, let's imagine an APB interface, you can also use bundles :

```
class APB(addressWidth: Int,
          dataWidth: Int,
          selWidth : Int,
          useSlaveError : Boolean) extends Bundle {

  val PADDR      = UInt(addressWidth bit)
  val PSEL       = Bits(selWidth bits)
  val PENABLE    = Bool
  val PREADY     = Bool
  val PWRITE     = Bool
  val PWDATA     = Bits(dataWidth bit)
  val PRDATA     = Bits(dataWidth bit)
  val PSLVERROR  = if(useSlaveError) Bool() else null   //This wire is created only␣
↪when useSlaveError is true
}

// Example of usage :
val bus = APB(addressWidth = 8,
              dataWidth = 32,
              selWidth = 4,
              useSlaveError = false)
```

One good practice is to group all construction parameters inside a configuration class. This could make the parametrization much easier later in your components, especially if you have to reuse the same configuration at multiple places. Also if one time you need to add another construction parameter, you will only have to add it into the configuration class and everywhere this one is instantiated:

```
case class APBConfig(addressWidth: Int,
                     dataWidth: Int,
                     selWidth : Int,
                     useSlaveError : Boolean)

class APB(val config: APBConfig) extends Bundle {   //[val] config, make the␣
↪configuration public
  val PADDR      = UInt(config.addressWidth bit)
  val PSEL       = Bits(config.selWidth bits)
  val PENABLE    = Bool
  val PREADY     = Bool
  val PWRITE     = Bool
  val PWDATA     = Bits(config.dataWidth bit)
  val PRDATA     = Bits(config.dataWidth bit)
  val PSLVERROR  = if(config.useSlaveError) Bool() else null
}

// Example of usage
val apbConfig = APBConfig(addressWidth = 8,dataWidth = 32,selWidth = 4,useSlaveError␣
↪= false)
val busA = APB(apbConfig)
val busB = APB(apbConfig)
```

Then at some points, you will probably need to use the APB bus as master or as slave interface of some components. To do that you can define some functions :

```
import spinal.core._

case class APBConfig(addressWidth: Int,
                     dataWidth: Int,
                     selWidth : Int,
                     useSlaveError : Boolean)

class APB(val config: APBConfig) extends Bundle {
  val PADDR      = UInt(config.addressWidth bit)
  val PSEL       = Bits(config.selWidth bits)
  val PENABLE    = Bool
  val PREADY     = Bool
  val PWRITE     = Bool
  val PWDATA     = Bits(config.dataWidth bit)
  val PRDATA     = Bits(config.dataWidth bit)
  val PSLVERROR  = if(config.useSlaveError) Bool() else null

  def asMaster(): this.type = {
    out(PADDR,PSEL,PENABLE,PWRITE,PWDATA)
    in(PREADY,PRDATA)
    if(config.useSlaveError) in(PSLVERROR)
    this
  }

  def asSlave(): this.type = this.asMaster().flip() //Flip reverse all in out␣
↪configuration.
}

// Example of usage
val apbConfig = APBConfig(addressWidth = 8,dataWidth = 32,selWidth = 4,useSlaveError␣
↪= false)
val io = new Bundle{
  val masterBus = APB(apbConfig).asMaster()
  val slaveBus = APB(apbConfig).asSlave()
}
```

Then to make that better, the spinal.lib integrate a small master slave utile named IMasterSlave. When a bundle extends IMasterSlave, it should implement/override the asMaster function. It give you the ability to setup a master or a slave interface by a smoother way :

```
val apbConfig = APBConfig(addressWidth = 8,dataWidth = 32,selWidth = 4,useSlaveError␣
↪= false)
val io = new Bundle{
  val masterBus = master(apbConfig)
  val slaveBus  = slave(apbConfig)
}
```

There is an example of an APB bus that implement this IMasterSlave :

```
//You need to import spinal.lib._ to use IMasterSlave
import spinal.core._
import spinal.lib._

case class APBConfig(addressWidth: Int,
                     dataWidth: Int,
                     selWidth : Int,
                     useSlaveError : Boolean)
```

(continues on next page)

(continued from previous page)

```
class APB(val config: APBConfig) extends Bundle with IMasterSlave {
  val PADDR      = UInt(addressWidth bit)
  val PSEL       = Bits(selWidth bits)
  val PENABLE    = Bool
  val PREADY     = Bool
  val PWRITE     = Bool
  val PWDATA     = Bits(dataWidth bit)
  val PRDATA     = Bits(dataWidth bit)
  val PSLVERROR  = if(useSlaveError) Bool() else null   //This wire is created only␣
↪when useSlaveError is true

  override def asMaster() : Unit = {
    out(PADDR,PSEL,PENABLE,PWRITE,PWDATA)
    in(PREADY,PRDATA)
    if(useSlaveError) in(PSLVERROR)
  }
  //The asSlave is by default the flipped version of asMaster.
}
```

15.3.9 Enum

SpinalHDL support enumeration with some encodings :

Encoding	Bit width	Description
native		Use the VHDL enumeration system, this is the default encoding
binarySequancial	log2Up(stateCount)	Use Bits to store states in declaration order (value from 0 to n-1)
binaryOneHot	stateCount	Use Bits to store state. Each bit correspond to one state

Define a enumeration type:

```
object UartCtrlTxState extends SpinalEnum { // Or␣
↪SpinalEnum(defaultEncoding=encodingOfYouChoice)
  val sIdle, sStart, sData, sParity, sStop = newElement()
}
```

Instantiate a enumeration signal and assign it :

```
val stateNext = UartCtrlTxState() // Or UartCtrlTxState(encoding=encodingOfYouChoice)
stateNext := UartCtrlTxState.sIdle

//You can also import the enumeration to have the visibility on its elements
import UartCtrlTxState._
stateNext := sIdle
```

15.3.10 Data (Bool, Bits, UInt, SInt, Enum, Bundle, Vec)

All hardware types extends the Data class, which mean that all of them provide following operators :

Operator	Description	Return
x === y	Equality	Bool
x =/= y	Inequality	Bool
x.getWidth	Return bitcount	Int
x ## y	Concatenate, x->high, y->low	Bits(width(x) + width(y) bits)
Cat(x)	Concatenate list, first element on lsb, x : Array[Data]	Bits(sumOfWidth bits)
Mux(cond,x,y)	if cond ? x : y	T(max(w(x), w(y) bits)
x.asBits	Cast in Bits	Bits(width(x) bits)
x.assignFromBits(bits)	Assign from Bits	
x.assignFromBits(bits,hi,lo)	Assign bitfield, hi : Int, lo : Int	T(hi-lo+1 bits)
x.assignFromBits(bits,offset,width)	Assign bitfield, offset: UInt, width: Int	T(width bits)
x.getZero	Get equivalent type assigned with zero	T

15.3.11 Literals as signal declaration

Literals are generally use as a constant value. But you can also use them to do two things in a single one :

- Define a wire which is assigned with a constant value

There is an example :

```
val cond = in Bool
val red = in UInt(4 bits)
...
val valid = False           //Bool wire which is by default assigned with False
val value = U"0100"         //UInt wire of 4 bits which is by default assigned with 4
when(cond){
  valid := True
  value := red
}
```

CHAPTER

SIXTEEN

WELCOME TO SPINALHDL'S DOCUMENTATION!

16.1 Site purpose and structure

This site presents the *SpinalHDL* language and how to use it on concrete examples.

If you are learning the language from scratch, this presentation is probably a good starting point.

16.2 What is SpinalHDL ?

SpinalHDL is an open source high-level hardware description language. It can be used as an alternative to VHDL or Verilog and has several advantages over them.

Also, SpinalHDL is not an HLS approach. Its goal is not to push something abstract into flip-flops and gates, but by using simple elements (flip-flops, gates, if / case statments) create a new abstraction level and help the designer to reuse their code and not write the same thing over and over again.

Note: SpinalHDL is *fully interoperable* with standard VHDL/Verilog-based EDA tools (simulators and synthetizers) as the output generated by the toolchain could be VHDL or Verilog. It also enables mixed designs where SpinalHDL components inter-operate with VHDL or Verilog IPs.

16.2.1 Advantages of using SpinalHDL over VHDL / Verilog

As SpinalHDL is based on a high-level language, it provides several advantages to improve your hardware coding:

1. **No more endless wiring** - Create and connect complex buses like AXI in one single line.
2. **Evolving capabilities** - Create your own bus definitions and abstraction layers.
3. **Reduce code size** - By a high factor, especially for wiring. This enables you to have a better overview of your code base, increase your productivity and create fewer headaches.
4. **Free and user friendly IDE** - Thanks to Scala tools for auto-completion, error highlighting, navigation shortcuts, and many others.
5. **Powerful and easy type conversions** - Bidirectional translation between any data type and bits. Useful when loading a complex data structure from a CPU interface.
6. **Loop detection** - Tools check that there are no combinatorial loops / latches.
7. **Clock domain safety** - The tools inform you that there are no unintentional clock domain crossings.
8. **Generic design** - There are no restrictions to the genericity of your hardware description by using Scala constructs.

16.2.2 License

SpinalHDL uses two licenses, one for spinal.core, and one for spinal.lib.

spinal.core (the compiler) is under the LGPL license, which could be summarized with following statements:

- You can make money with your SpinalHDL description and its generated RTL.
- You don't have to share your SpinalHDL description and its generated RTL.
- There are no fees and no royalties.
- If your make improvements to the SpinalHDL core, please share your modifications to make the tool better for everybody.

spinal.lib (a general purpose library of components/tools/interfaces) is under the permissive MIT license.

16.3 Getting started

Want to try it for yourself? Then jump to the *getting started section* and have fun!

16.4 Links

SpinalHDL repository:
https://github.com/SpinalHDL/SpinalHDL

A short show case (PDF):
motivation.pdf

Presentation of the language (PDF):
presentation.pdf

SBT base project:
https://github.com/SpinalHDL/SpinalTemplateSbt

Jupyter bootcamp:
https://github.com/SpinalHDL/Spinal-bootcamp

Workshop:
https://github.com/SpinalHDL/SpinalWorkshop

VexRiscv CPU and SoC:
https://github.com/SpinalHDL/VexRiscv

StackOverflow (tag: SpinalHDL) :
StackOverflow

Google group:

https://groups.google.com/forum/#!forum/spinalhdl-hardware-description-language