SpinalHDL Documentation

SpinalHDL contributors

Jul 04, 2024

CONTENTS

Welcome to SpinalHDL's documentation!

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:

- It focuses on efficient hardware description instead of being event-driven.
- It is embedded into a general purpose programming language, enabling powerful hardware generation.

More detailed introduction of the language in *About SpinalHDL*

HTML and PDF formats of this documentation are available online:
> spinalhdl.github.io/SpinalDoc-RTD
(PDF format is accessible from the lower left corner, click `v:master` then PDF)
Chinese version of documentation:
> github.com/thuCGRA/SpinalHDL_Chinese_Doc
You can also find the API documentation:
> spinalhdl.github.io/SpinalHDL

CHAPTER

ONE

FOREWORD

Preliminary notes:

- All the following statements will be about describing digital hardware. Verification is another tasty topic.
- For conciseness, let's assume that SystemVerilog is a recent revision of Verilog.
- When reading this, we should not underestimate how much our attachment for our favourite HDL will bias our judgement.

1.1 Why moving away from traditional HDL

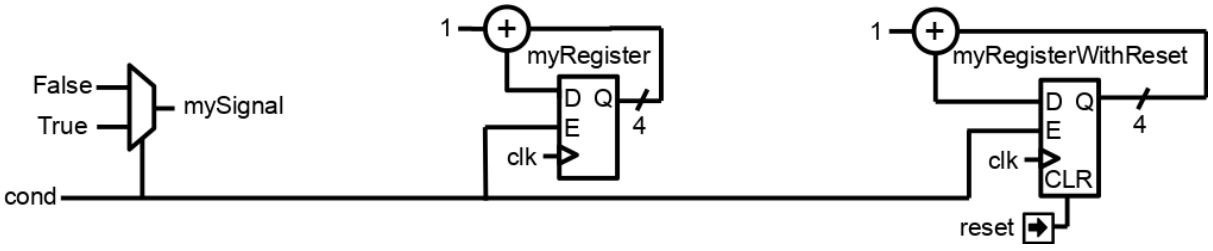
1.1.1 VHDL/Verilog aren't Hardware Description Languages

Those languages are event driven languages created initially for simulation/documentation purposes. Only in a second time they were used as inputs languages for synthesis tools. Which explain the roots of a lot of the following points.

1.1.2 Event driven paradigm doesn't make any sense for RTL

When you think about it, describing digital hardware (RTL) by using process/always blocks doesn't make any practical senses. Why do we have to worry about a sensitivity list? Why do we have to split our design between processes/always blocks of different natures (combinatorial logic / register without reset / register with async reset)?

For instance, to implement this:



Using VHDL processes you write this:

```
signal mySignal : std_logic;
signal myRegister : unsigned(3 downto 0);
signal myRegisterWithReset : unsigned(3 downto 0);

process(cond)
begin
    mySignal <= '0';
    if cond = '1' then
        mySignal <= '1';
```

(continues on next page)

(continued from previous page)

```
    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 <= 0;
    elsif rising_edge(clk) then
        if cond = '1' then
            myRegisterWithReset <= myRegisterWithReset + 1;
        end if;
    end if;
end process;
```

Using SpinalHDL you write this:

```
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
}
```

As for everything, you can get used to this event driven semantic, until you taste something better.

1.1.3 Recent revisions of VHDL and Verilog aren't usable

The EDA industry is really slow to implement VHDL 2008 and SystemVerilog synthesis capabilities in their tools. Additionally, when it's done, it appear that only a constraining subset of the language is implemented (not talking about simulation features). It result that using any interesting feature of those language revision isn't safe as:

- It will probably make your code incompatible with many EDA tools.
- Other companies will likely not accept your IP as their flow isn't ready for it.

Anyway, those revisions don't change the heart of those HDL issues: they are based on a event driven paradigm which doesn't make sense to describe digital hardware.

1.1.4 VHDL records, Verilog struct are broken (SystemVerilog is good on this, if you can use it)

You can't use them to define an interface, because you can't define their internal signal directions. Even worst, you can't give them construction parameters! So, define your RGB record/struct once, and hope you never have to use it with bigger/smaller color channels...

Also a fancy thing with VHDL is the fact that if you want to add an array of something into a component entity, you have to define the type of this array into a package... Which can't be parameterized...

For instance, below is a SpinalHDL APB3 bus definition:

```
// Class which can be instantiated to represent a given APB3 configuration
case class Apb3Config(
  addressWidth  : Int,
  dataWidth     : Int,
  selWidth      : Int     = 1,
  useSlaveError : Boolean = true
)

// Class which can be instantiated to represent a given hardware APB3 bus
case class Apb3(config: Apb3Config) extends Bundle with IMasterSlave {
  val PADDR      = UInt(config.addressWidth bits)
  val PSEL       = Bits(config.selWidth bits)
  val PENABLE    = Bool()
  val PREADY     = Bool()
  val PWRITE     = Bool()
  val PWDATA     = Bits(config.dataWidth bits)
  val PRDATA     = Bits(config.dataWidth bits)
  val PSLVERROR  = if(config.useSlaveError) Bool() else null  // Optional signal

  // Can be used to setup a given APB3 bus into a master interface of the host␣
↪component
  // `asSlave` is automatically implemented by symmetry
  override def asMaster(): Unit = {
    out(PADDR, PSEL, PENABLE, PWRITE, PWDATA)
    in(PREADY, PRDATA)
    if(config.useSlaveError) in(PSLVERROR)
  }
}
```

Then about the VHDL 2008 partial solution and the SystemVerilog interface/modport, lucky you are if your EDA tools / company flow / company policy allow you to use them.

1.1.5 VHDL and Verilog are so verbose

Really, with VHDL and Verilog, when it starts to be about component instantiation interconnection, the copy-paste god has to be invoked.

To understand it more deeply, below is a SpinalHDL example performing some peripherals instantiation and adding the APB3 decoder required to access them.

```
// Instantiate an AXI4 to APB3 bridge
val apbBridge = Axi4ToApb3Bridge(
  addressWidth = 20,
  dataWidth    = 32,
  idWidth      = 4
)
```

(continues on next page)

(continued from previous page)

```
// Instantiate some APB3 peripherals
val gpioACtrl = Apb3Gpio(gpioWidth = 32)
val gpioBCtrl = Apb3Gpio(gpioWidth = 32)
val timerCtrl = PinsecTimerCtrl()
val uartCtrl = Apb3UartCtrl(uartCtrlConfig)
val vgaCtrl = Axi4VgaCtrl(vgaCtrlConfig)

// Instantiate an APB3 decoder
// - Drived by the apbBridge
// - Map each peripheral in a memory region
val apbDecoder = Apb3Decoder(
  master = apbBridge.io.apb,
  slaves = List(
    gpioACtrl.io.apb -> (0x00000, 4 KiB),
    gpioBCtrl.io.apb -> (0x01000, 4 KiB),
    uartCtrl.io.apb  -> (0x10000, 4 KiB),
    timerCtrl.io.apb -> (0x20000, 4 KiB),
    vgaCtrl.io.apb   -> (0x30000, 4 KiB)
  )
)
```

Done. That's all. You don't have to bind each signal one by one when you instantiate a module/component because you can access their interfaces in a object-oriented manner.

Also about VHDL/Verilog struct/records, we can say that they are really dirty tricks, without true parameterization and reusability capabilities, trying to hide the fact that those languages were poorly designed.

1.1.6 Meta Hardware Description capabilities

Basically VHDL and Verilog provide some elaboration tools which aren't directly mapped into hardware as loops / generate statements / macro / function / procedure / task. But that's all.

And even then, they are really limited. For instance, one can't define process/always/component/module blocks into a task/procedure. It is really a bottleneck for many fancy things.

With SpinalHDL you can call a user-defined task/procedure on a bus like that: `myHandshakeBus.queue(depth=64)`. Below is some code including the definition.

```
// Define the concept of handshake bus
class Stream[T <: Data](dataType:  T) extends Bundle {
  val valid   = Bool()
  val ready   = Bool()
  val payload = cloneOf(dataType)

  // Define an operator to connect the left operand (this) to the right operand (that)
  def >>(that: Stream[T]): Unit = {
    that.valid := this.valid
    this.ready := that.ready
    that.payload := this.payload
  }

  // Return a Stream connected to this via a FIFO of depth elements
  def queue(depth: Int): Stream[T] = {
    val fifo = new StreamFifo(dataType, depth)
    this >> fifo.io.push
    return fifo.io.pop
```

(continues on next page)

(continued from previous page)

```
  }
}
```

Let's see further, imagine you want to define a state machine. With VHDL/Verilog you have to write a lot of raw code with some switch statements to do it. You can't define the notion of "StateMachine", which would give you a nice syntax to define each state. Else you can use a third-party tool to draw your state machine and then generate your VHDL/Verilog equivalent code...

Meta-hardware description capabilities of SpinalHDL enable you to define your own tools which then allow you to define things in abstracts ways, as for state machines.

Below is an simple example of the usage of a state machine abstraction defined on the top of SpinalHDL:

```
// Define a new state machine
val fsm = new StateMachine{
  // Define all states
  val stateA, stateB, stateC = new State

  // Set the entry point
  setEntry(stateA)

  // Define a register used into the state machine
  val counter = Reg(UInt(8 bits)) init (0)

  // Define the state machine behaviour for each state
  stateA.whenIsActive (goto(stateB))

  stateB.onEntry(counter := 0)
  stateB.onExit(io.result := True)
  stateB.whenIsActive {
    counter := counter + 1
    when(counter === 4){
      goto(stateC)
    }
  }

  stateC.whenIsActive(goto(stateA))
}
```

Imagine you want to generate the instruction decoding of your CPU. It could require some fancy elaboration time algorithms to generate the less logic possible. But in VHDL/Verilog, your only option to do these kind of things is to write a script which generates the `.vhd` and `.v` that you want.

There is really much to say about meta-hardware description, but the only true way to understand it and get its real taste is to experiment it. The goal with it is to stop playing with wires and gates, to start taking some distance with that low level stuff, to think reusable.

CHAPTER

TWO

INTRODUCTION

This section introduces the SpinalHDL project: the language, and everything around it.

2.1 About SpinalHDL

2.1.1 What is SpinalHDL?

SpinalHDL is an open source high-level hardware description language with associated tools. Its development started in December 2014.

SpinalHDL makes it possible to efficiently describe hardware, giving names to digital hardware notions; the most obvious examples are `Reg` and `Latch`. In event-driven languages such as VHDL and Verilog, to use these two common elements, the user has to describe how to simulate them with a process, so that the synthesis tool can infer what cell it is. With SpinalHDL, you just have to declare a `Reg` or a `Latch`.

SpinalHDL is a *domain-specific language* based on Scala a general-purpose language. It brings several benefits:

- There are free integrated development environments supporting it, providing many features that simple text editors don't have:
  - syntax and type errors are highlighted right in the code
  - correct renaming, even across files
  - smart auto completion / suggestions
  - navigation tools (go to definition, show all references, etc.)
- It allows to implement simple to complex hardware generators (meta-hardware description) with no need to deal with several languages.

Note: Scala is a statically-typed, functional and object-oriented language using the Java virtual machine (JVM).

2.1.2 What SpinalHDL is not

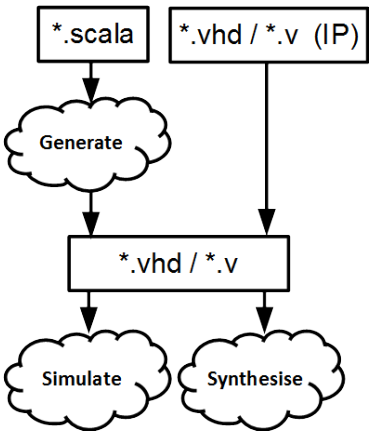
SpinalHDL is not an HLS tool: its goal is not to automagically transform an abstract algorithm into a digital circuit. Its goal is to create a new abstraction level by naming things, to help the designer reuse their code and not write the same thing over and over again.

SpinalHDL is not an analog modeling language. VHDL and Verilog make it possible for analog designers to provide a model of their IP to digital designers. SpinalHDL does not address this case, and is for digital designers to describe their own digital designs.

2.1.3 The Spinal development flow

Once code is written in *SpinalHDL*, the tool can:

- Generate VHDL, Verilog or SystemVerilog, to instantiate it in one of these languages or give it to any simulator or synthesis tool. There is no logic overhead, hierarchy and names are preserved, and it runs design checks during generation.
- Boot a simulation using Verilator or another supported simulator.



As SpinalHDL is interoperable with VHDL and (System)Verilog, you can both instantiate SpinalHDL IPs in these language (using generated code) and instantiate IPs in these languages in SpinalHDL (using `BlackBox`).

Note: SpinalHDL is *fully interoperable* with standard VHDL/Verilog-based EDA tools (simulators and synthesizers) as the output generated by the toolchain can be VHDL or Verilog.

2.1.4 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. **Design checks** - Early stage lints to check that there are eg no combinatorial loops / latches.
7. **Clock domain safety** - Early stage lints to 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.

2.2 A simple example

Below is a simple hardware description from the getting started repository.

```
case class MyTopLevel() extends Component {
  val io = new Bundle {
    val cond0 = in port Bool()
    val cond1 = in port Bool()
    val flag  = out port Bool()
    val state = out port UInt(8 bits)
  }

  val counter = Reg(UInt(8 bits)) init 0

  when(io.cond0) {
    counter := counter + 1
  }

  io.state := counter
  io.flag  := (counter === 0) | io.cond1
}
```

It is split into chunks and explained in this section.

2.2.1 Component

First, there is the structure of a SpinalHDL `Component`.

A component is a piece of logic which can be instantiated (pasted) as many times as needed, and where the only accessible signals are its inputs and outputs.

```
case class MyTopLevel() extends Component {
  val io = new Bundle {
    // port definitions go here
  }

  // component logic goes here
}
```

`MyTopLevel` is the name of the component.

In SpinalHDL, components use `UpperCamelCase`.

Note: See also *Components and hierarchy* for more information.

2.2.2 Ports

Then, the ports are defined.

```
val cond0 = in port Bool()
val cond1 = in port Bool()
val flag = out port Bool()
val state = out port UInt(8 bits)
```

Directions:

- `cond0` and `cond1` are inputs ports
- `flag` and `state` are outputs ports

Types:

- `cond0`, `cond1` and `flag` are 1 bit each (as 3 individual wires)
- `state` is an 8-bit unsigned integer (a bus of 8 wires representing an unsigned integer)

Note: This syntax is only available since SpinalHDL 1.8, see *Input / output definition* for legacy syntax and more information.

2.2.3 Internal logic

Finally, there is the component logic:

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

when(io.cond0) {
  counter := counter + 1
}

io.state := counter
io.flag := (counter === 0) | io.cond1
```

`counter` is a register containing an 8-bits unsigned integer, with the initial value 0. Assignments to change the state of a register are available for read-back only after the next clock sampling.

Note: Because of the presence of a register, two implicit signals are added to the component for the clock and the reset. See *Registers* and *Clock domains* for more information.

Then a conditional rule is described: when the input `cond0` (which is in the `io` bundle) is set, the `counter` is incremented by one, else `counter` keeps its value set in the last rule. But, there is no previous rule, you would say. With a simple signal it would be a latch, and trigger an error. But here `counter` is a register, so it has a default case: it just keeps the same value.

This creates a multiplexer: the input of the `counter` register can be its output or its output plus one depending on `io.cond0`.

Then unconditional rules (assignments) are described:

- The output `state` is connected to the output of the register `counter`.
- The output `flag` is the output of an `or` gate between a signal which is true when the output of "`counter` equals 0", and the input `cond1`.

Note: See also *Semantic* for more information.

2.3 Projects using SpinalHDL

Note that the following lists are very incompletes.

2.3.1 Repositories

- J1Sc Stack CPU
- VexRiscv CPU and SoC
- NaxRiscv CPU
- SaxonSoc
- open-rdma
- MicroRV32 SoC
- ...

2.3.2 Companies

- DatenLord, China
  - RoCE v2 hardware implementation
  - WaveBPF (wBPF): a "tightly-coupled multi-core" eBPF CPU, designed to be a high-throughput co-processor for processing in-memory data (e.g. network packets).
- Elitestek (FPGA Vendor), China

  "Elitestek has used the VexRISC-V core in FPGAs and applied in multi applications in worldwide customers."
- LeafLabs, Massachusetts, USA

  SpinalHDL To Accelerate Neuroscience (PDF slideshow)
- QsPin, Belgium
- Tiempo Secure, France

  SpinalHDL for ASIC (PDF slideshow)
- ...

2.3.3 Universities

- Universität Bremen - Fachbereich 3 - Informatik, Germany

  SpinalHDL in Computer Architecture Research and Education (PDF slideshow)
- Universität Potsdam - Embedded Systems Architectures for Signalprocessing, Germany

  A Network Attached Deep Learning Accelerator for FPGA Clusters (PDF slideshow)
- ...

2.4 Getting in touch

- For questions about SpinalHDL syntax and live talks:
  - English Matrix channel
  - Chinese Matrix channel
  - Google group
- For bug reports, feature requests and questions:
  - Open a ticket
- If you are interested in a presentation, a workshop, or consulting:
  - Contact us by email: spinalhdl@gmail.com

2.5 License

SpinalHDL uses two licenses, one for `spinal.core` and one for `spinal.lib` and everything else in the repository.

`spinal.core` (the compiler) is under the LGPL license, which can be summarized as follows:

- 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, and you wish to redistribute those modifications, you have to share those modifications to make the tool better for everybody.

`spinal.lib` (a general purpose library of components/tools/interfaces) is under the permissive MIT license so you do not have to share it, even if contributions are really appreciated.

2.6 Contributing

- Repository of the language
- Contributor guide
- Repository of this documentation
- Donation channel

2.7 FAQ

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

The overhead is nil. SpinalHDL generates the same HDL constructs found in human written VHDL/Verilog with no additional instantiated artifacts present in the resulting implementation due to its use. This makes the overhead zero when comparing the generated HDL against handwritten HDL.

Due to the powerful expressive nature of the Scala/SpinalHDL languages, the design is more concise for a given complex hardware design and has strong type safety, strong HDL safety paradigms that result in fewer lines of code, able to achieve more functionality with fewer bugs.

SpinalHDL does not take a HLS approach and is not itself a HLS solution. 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 and increase code reuse at the level the designer is working.

2.7.2 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 within a few hours.
3. Consider how much time it takes for a commercial EDA vendor to fix issues or to add new features in their closed tools. Consider also your cost and time savings achieved when using SpinalHDL and the potential for your own entity to give back to the community some of this as engineering time, open-source contribution time or donations to the project to improve its future.

2.7.3 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?

2.7.4 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.

2.7.5 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.

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

The *Foreword* is dedicated to this topic.

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

First, you need to get the repository, if you haven't cloned it yet:

```
git clone --depth 1 -b dev https://github.com/SpinalHDL/SpinalHDL.git
cd SpinalHDL
```

In the command above you can replace `dev` by the name of the branch you want to checkout. `--depth 1` prevents from downloading the repository history.

Then publish the code as it is in the directory fetched:

```
sbt clean '++ 2.12.13' publishLocal
```

Here `2.12.13` is the Scala version used. The first two numbers must match the ones of the version used in your project. You can find it in your `build.sbt` and/or `build.sc`:

```
ThisBuild / scalaVersion := "2.12.16" // in build.sbt
// or
def scalaVersion = "2.12.16" // in build.sc
```

Then in your project, update the SpinalHDL version specified in your `build.sbt` or `build.sc`: it should be set to `dev` instead of a version number.

```
val spinalVersion = "1.7.3"
// becomes
val spinalVersion = "dev"
```

Note: Here it is always `dev` no matter the branch you have checked out earlier.

2.8 Other learning materials

- A short show case (PDF slideshow)
- Presentation of the language (PDF slideshow)
- Jupyter bootcamp
- Workshop

There is also a few more specific videos online :

- On youtube
- On f-si's peertube

A few SpinalHDL webinar were made and are recorded here :

- Datenlord's youtube channel

Note: Some of those tutorials are not using the latest version of SpinalHDL, so they may lack some recent SpinalHDL features.

CHAPTER

THREE

GETTING STARTED

Let's start learning SpinalHDL! In this chapter, we will install and setup an environment, taste the language and learn how to generate VHDL and Verilog, and perform lints on the fly.

3.1 Install and setup

Spinal is a Scala library (a programming language using the Java VM) so it requires setting up a Scala environment; there are many ways to do so. Also, it generates VHDL, Verilog or SystemVerilog, which can be used by many different tools. This section describes the supported way to install a *SpinalHDL description to Simulation* flow, but there can be many variations.

3.1.1 Required/Recommended tools

Before you download the SpinalHDL tools, you need to install a Scala environment:

- Java JDK, a Java environment
- Scala 2, compiler and library
- SBT, a Scala build tool

These tools enable to use Spinal; but without any other tools, it is limited to HDL code generation.

To enable more features we recommend:

- An IDE (for instance the currently recommended IntelliJ with its Scala plugin or VSCodium with Metals extension) to get features such as:
  - Code suggestions / completion
  - Automatic build with syntax errors right in the code
  - Generate code with a single click
  - Run simulation / tests with a single click (if a supported simulator is set up)
- A supported simulator like Verilator to test the design right from SpinalHDL.
- Gtkwave to view the waves generated by Verilator during simulation.
- Git for version control system
- A C++ toolchain, needed for simulating with Verilator
- A linux shell, needed for simulating with Verilator

3.1.2 Linux Installation

At time of writing the recommended way of installing Scala and SBT is via Coursier. Coursier is able to in addition to the scala tools install a Java JDK to use, in the example below we install Java from the package manager. We recommend to install JDK 17 (LTS) because of compatibility with the used Scala version.

For Debian or Ubuntu we run:

```
sudo apt-get update
sudo apt-get install openjdk-17-jdk-headless curl git
curl -fL "https://github.com/coursier/launchers/raw/master/cs-x86_64-pc-linux.gz" |␣
↪gzip -d > cs
chmod +x cs
# should find the just installed jdk, agree to cs' questions for adding to your PATH
./cs setup
source ~/.profile
```

If you want to install the tools for simulation and/or formal proofs, we recommend oss-cad-suite. It contains a waveform viewer (gtkWave), verilog simulators (verilator and iverilog), VHDL simulator (GHDL) and other tools. In case you want to build the tools yourself have a look at the legacy simulation tool *installation instructions*.

We first install the needed C++ toolchain and download oss-cad-suite. To use it we must load the oss-cad-suite environment for each shell we want to use it in. Note that oss-cad-suite contains a Python 3 interpreter that may interfere with the system Python installation if loaded permanently.

Go to the oss-cad-suite release page to get the download link for the latest version. You can download/extract oss-cad-suite to a folder of your choice. (last tested version of oss-cad-suite is *2023-10-22*, but more recent ones will most likely also work)

```
sudo apt-get install make gcc g++ zlib1g-dev
curl -fLO <download link>
tar xzf <file that you downloaded>
```

To use oss-cad-suite in a shell you need to load it's environment, e.g. via `souce <path to oss-cad-suite>/environment`.

3.1.3 Mac OS X Installation

You can use homebrew to install on Mac OS X. By default homebrew installs Java 21, but the SpinalHDL tutorial SpinalTemplateSbt uses Scala version 2.12.16, which is not supported by Java 21 (17 is still the recommended LTS version, https://whichjdk.com/). So to install Java version 1.7 do:

```
brew install openjdk@17
```

And then add this to your path.

```
export PATH="/opt/homebrew/opt/openjdk@17/bin:$PATH"
```

To manage multiple versions of Java it is also essential to have jenv installed.

```
brew install jenv
```

Jenv added these lines to my .bash_profile

```
export PATH="$HOME/.jenv/bin:$PATH"
eval "$(jenv init -)"
```

Next you have to install scala's interactive build tool sbt.

```
brew install sbt
```

If this works for you, please let us know. If this does not work for you, you can read the github issue about Mac o SX installation here. https://github.com/SpinalHDL/SpinalHDL/issues/1216

If you want to install the tools for simulation and/or formal proofs, we recommend oss-cad-suite.

3.1.4 Windows installation

Note: While a native installation is possible the simpler and currently recommended way is to use WSL on Windows. If you want to use WSL, install it, a distribution of your choice and follow the Linux installation instructions. Data in your WSL instance can be accessed from windows under `\\wsl$`. In case you want to use IntelliJ you'll have to download the Linux version to WSL, if you want to use VSCode then the Windows version can be used to remotely edit in WSL.

At time of writing the recommended way of installing Scala and SBT is via Coursier. Coursier is able to in addition to the scala tools install a Java JDK to use, in the example below we install Java manually. We recommend to install JDK 17 (LTS) because of compatibility with Scala.

First download and install Adoptium JDK 17. Download, unzip and run the Coursier installer, when asked agree to an update of your `PATH` variable. Reboot to force an update of `PATH`.

This is sufficient for generating hardware. For simulation continue with either choice below. In case you want to build the tools yourself have a look at the legacy simulation tool *installation instructions*.

Note: An All-in-One solution offered by SpinalHDL maintainer *Readon <https://github.com/Readon>* is available to install and run SpinalHDL with Verilator simulation and formal verification via SymbiYosys. Download it and install the environment anywhere on your disk. Start the build environment by clicking on the MSYS2-MINGW64 icon in the Start menu and use the MSYS2 default console. An alternative is to use the Windows Terminal or a Tabby-like application and use the startup command `%MSYS2_ROOT%\msys2_shell.cmd -defterm -here -no-start -mingw64`, where the `%MSYS2_ROOT%` is the location of the msys2 installation. It is worth noting that if you want to use it offline, you should carefully select the libraries that the project depends on, otherwise you will need to download the packages manually. See the README for the repos for more details.

MSYS2 verilator for simulation

We recommend to install compiler/verilator through *MSYS2 <https://www.msys2.org>*. Other methods of installing gcc/make/shell (e.g. chocolatey, scoop, etc.) may also work but are untested.

SpinalHDL maintainer *Readon <https://github.com/Readon>* is maintaining a MSYS2 fork that default installs all needed officially available and custom built packages (also maintained by Readon *here <https://github.com/Readon/MINGW-SpinalHDL>*) for simulation and formal verification. It can be found *here <https://github.com/Readon/msys2-installer>*. If used then the packages installed below via `pacman` are already installed and those installation steps can be skipped.

Currently verilator 4.228 is latest available version known to work.

Download the latest installer and install MSYS2 with default settings. You should get a MSYS2 terminal at the end of the installation, there run:

```
pacman -Syuu
# will (request) close down terminal
# open 'MSYS2 MINGW64' from start menu
pacman -Syuu
pacman -S --needed base-devel mingw-w64-x86_64-toolchain mingw-w64-x86_64-iverilog␣
```

(continues on next page)

(continued from previous page)

```
↪mingw-w64-x86_64-ghdl-llvm git
curl -O https://repo.msys2.org/mingw/mingw64/mingw-w64-x86_64-verilator-4.228-1-any.
↪pkg.tar.zst
pacman -U mingw-w64-x86_64-verilator-4.228-1-any.pkg.tar.zst
```

In a MSYS2 MINGW64 terminal we need to set some environment variables to make Java/sbt available (you can make these settings persistent by adding them to `~/.bashrc` in MSYS2):

```
export VERILATOR_ROOT=/mingw64/share/verilator/
export PATH=/c/Program\ Files/Eclipse\ Adoptium/jdk-17.0.8.101-hotspot/bin:$PATH
export PATH=/c/Users/User/AppData/Local/Coursier/data/bin:$PATH
```

With this you should be able to run sbt/verilator simulations from MSYS2 terminals (sbt via calling `sbt.bat`).

MSYS2 for formal verification

In addition to the steps above we also need to install yosys, sby, z3 and yices. Both yosys(yosys-smtbmc workable) and sby are not available as official MSYS2 packages, but packages are provided by *Readon <https://github.com/Readon>*. If you used their installer then these steps are not needed (you should check if there are newer packages available).

```
pacman -S mingw-w64-x86_64-z3 mingw-w64-x86_64-yices mingw-w64-x86_64-autotools mingw-
↪w64-x86_64-python3-pip
python3 -m pip install click
curl -OL https://github.com/Readon/MINGW-SpinalHDL/releases/download/v0.4.9/mingw-w64-
↪x86_64-yosys-0.31-1-any.pkg.tar.zst
curl -OL https://github.com/Readon/MINGW-SpinalHDL/releases/download/v0.4.9/mingw-w64-
↪x86_64-python-sby-0.31-1-any.pkg.tar.zst
pacman -U *-yosys-*.pkg.tar.*
pacman -U *-python-sby-*.pkg.tar*
```

3.1.5 OCI Container

A container for SpinalHDL development is available as well. The container is hosted at `ghcr.io/spinalhdl/docker:master` and can be used with Docker/Podman/Github Codespaces. It is used for the SpinalHDL CI regression and can therefore be an easy way to run the CI commands locally.

To run the container run e.g. `podman run -v .:/workspace -it ghcr.io/spinalhdl/docker:master` in a SpinalHDL project root directory, making the project directory available in `/workspace`.

Please consult the documentation of you Distribution (Linux, WSL) or Docker (Windows) on how to install the container runtime you want to use. Multiple editors/IDEs (e.g. VSCode, IntelliJ, Neovide) allow for remote development in a container. Please consult the documentation of the editor on how to do remote development.

3.1.6 Installing SBT in an internet-free Linux environment

Note: If you are not using an air-gapped environment we recommend to go with the normal linux installation. (which is a subset of the installation for an air-gapped environment)

Normally, SBT uses online repositories to download and cache your projects dependencies. This cache is located in several folders:

- `~/.sbt`
- `~/.cache/JNA`

- `~/.cache/coursier`

To set up an internet-free environment, you can:

1. Set up an environment with internet (see above)
2. Launch a Spinal command (see *Using Spinal from CLI with SBT*) to fetch dependencies (for instance using the getting started repository)
3. Copy the caches to the internet-free environment.

3.2 Create a first SpinalHDL project

We have prepared a ready-to-go project for you the: getting started repository.

You can download it, or clone it.

The following commands clone the project into a new directory named `MySpinalProject` and initialize a fresh `git` history:

```
git clone --depth 1 https://github.com/SpinalHDL/SpinalTemplateSbt.git MySpinalProject
cd MySpinalProject
rm -rf .git
git init
git add .
git commit -m "Initial commit from template"
```

3.2.1 The directory structure of a project

Note: The structure described here is the default structure, but it can be easily modified.

In the root of the project are the following files:

File	Description
`build.sbt`	Scala configuration for `sbt`
`build.sc`	Scala configuration for `mill`, an alternative to `sbt`
`hw/`	The folder containing hardware descriptions
`project/`	More Scala configuration
`README.md`	A `text/markdown` file describing your project
`.gitignore`	List of files to ignore in versioning
`.mill-version`	More configuration for `mill`
`.scalafmt.conf`	Configuration of rules to auto-format the code

As you probably guessed it, the interesting thing here is `hw/`. It contains four folders: `spinal/`, `verilog/` and `vhdl/` for your IPs and `gen/` for IPs generated with Spinal.

`hw/spinal/` contains a folder named after your project name. This name must be set in `build.sbt` (along with the company name) and in `build.sc`; and it must be the one in `package yourprojectname` at the beginning of `.scala` files.

In `hw/spinal/yourprojectname/`, are the descriptions of your IPs, simulation tests, formal tests; and there is `Config.scala`, which contains the configuration of `Spinal`.

Note: `sbt` must be used **only** at the root of the project, in the folder containing `build.sbt`.

3.2.2 Using Spinal on SpinalHDL code

Now the tutorial shows how to use Spinal on SpinalHDL code depending on your development environment:

- *Using Spinal from CLI with SBT*
- *Using Spinal from VSCodium*
- *Using Spinal from IntelliJ IDEA*

3.3 Using Spinal from CLI with SBT

First, open a terminal in the root of the template you have downloaded earlier in *Create a first SpinalHDL project*.

Commands can be executed right from the terminal:

```
sbt "firstCommand with arguments" "secondCommand with more arguments"
```

But `sbt` has a quite long boot time so the we recommend to use its interactive mode:

```
sbt
```

Now `sbt` shows a prompt. Let's start by doing Scala compilation. It will fetch dependencies so it can take time the first time:

```
compile
```

Actually you never need to just `compile` as it is done automatically when needed. The first build time will take a few moments longer compared to future builds as the sbt tool builds the entire project from a cold start and then uses incremental building where possible from that point on. `sbt` supports autocompletion inside the interactive shell to assist discovery and usage of the available commands. You can start the interactive shell with `sbt shell` or running `sbt` with no arguments from the command line.

To run a specific HDL code-generation or simulation, the command is `runMain`. So if you type `runMain`, space, and tab, you should get this:

```
sbt:SpinalTemplateSbt> runMain
   ;                                projectname.MyTopLevelVerilog
   projectname.MyTopLevelFormal     projectname.MyTopLevelVhdl
   projectname.MyTopLevelSim
```

The autocompletion suggests all things that can be run. Let's run the Verilog generation for instance:

```
runMain projectname.MyTopLevelVerilog
```

Look at the directory ./hw/gen/: there is a new `MyTopLevel.v` file!

Now add a ~ at the beginning of the command:

```
~ runMain projectname.MyTopLevelVerilog
```

It prints this:

```
sbt:SpinalTemplateSbt> ~ runMain mylib.MyTopLevelVerilog
[info] running (fork) mylib.MyTopLevelVerilog
[info] [Runtime] SpinalHDL v1.7.3    git head :␣
↪aeaeece704fe43c766e0d36a93f2ecbb8a9f2003
[info] [Runtime] JVM max memory : 3968,0MiB
[info] [Runtime] Current date : 2022.11.17 21:35:10
[info] running (fork) projectname.MyTopLevelVerilog
```

(continues on next page)

(continued from previous page)

```
 [info] [Runtime] SpinalHDL v1.9.3    git head :␣
↪029104c77a54c53f1edda327a3bea333f7d65fd9
 [info] [Runtime] JVM max memory : 4096.0MiB
 [info] [Runtime] Current date : 2023.10.05 19:30:19
 [info] [Progress] at 0.000 : Elaborate components
 [info] [Progress] at 0.508 : Checks and transforms
 [info] [Progress] at 0.560 : Generate Verilog
[info] [Done] at 0.603
[success] Total time: 1 s, completed Oct 5, 2023, 7:30:19 PM
[info] 1. Monitoring source files for projectname/runMain projectname.
↪MyTopLevelVerilog...
[info]    Press <enter> to interrupt or '?' for more options.
```

So now, each time you save a source file, it will re-generate `MyTopLevel.v`. To do this, it automatically compiles the source files and it performs lint checks. This way you can get errors printed on the terminal almost in real-time while you are editing the source files.

You can press Enter to stop automatic generation, then Ctrl-D to exit `sbt`.

It is also possible to start it right from the terminal, without using `sbt`'s interactive prompt:

```
sbt "~ runMain mylib.MyTopLevelVerilog"
```

Now you can use your environment, let's explore the code: *A simple example*.

3.4 Using Spinal from VSCodium

Note: VSCodium is the open source build of Visual Studio Code, but without the telemetry included in Microsoft's downloadable version.

As a one-time setup task, go to view->extensions search for "Scala" and install the "Scala (Metals)" extension.

Open the workspace: `File > Open Folder...` and open the folder you have downloaded earlier in *Create a first SpinalHDL project*.

The other way to start it, is to cd into the appropriate directory and type `codium .`

Wait a little bit, a notification pop-up should appear on the bottom-right corner: "Multiple build definitions found. Which would you like to use?". Click `sbt`, then another pop-up appears, click `Import build`.

Wait while running `sbt bloopInstall`. Then a warning pop-up appears, you can ignore it (don't show again).

Find and open `hw/spinal/projectname/MyTopLevel.scala`. Wait a little bit, and see the `run | debug` line that is displayed by Metals, before each `App`. For instance, click on `run` just above `object MyTopLevelVerilog`. Alternatively, you can select Menu Bar -> Run -> Run Without Debugging. Either approach performs design checks and, as the checks pass, generates the Verilog file `./hw/gen/MyTopLevel.v`

This is all you need to do to use SpinalHDL from VSCodium. You now have the design-rule-checked Verilog and/or VHDL which you can use as input to your favorite synthesis tool.

Now that you know how to use the VSCodium development environment, let's explore the code: *A simple example*.

3.5 Using Spinal from IntelliJ IDEA

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:

- 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.

Now you can use your environment, let's explore the code: *A simple example*.

3.6 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.

3.6.1 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() }
```

3.6.2 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 bits), 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 include a dot before the keyword like methods `.elsewhen` and `.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) {

  }
```

(continues on next page)

(continued from previous page)

```
  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 bits)
  val g = UInt(c.gWidth bits)
  val b = UInt(c.bWidth bits)
}
```

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 {

}
```

3.6.3 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
}
```

(continues on next page)

(continued from previous page)

```
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 }
```

3.6.4 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.

3.7 Help for VHDL people

3.7.1 VHDL comparison

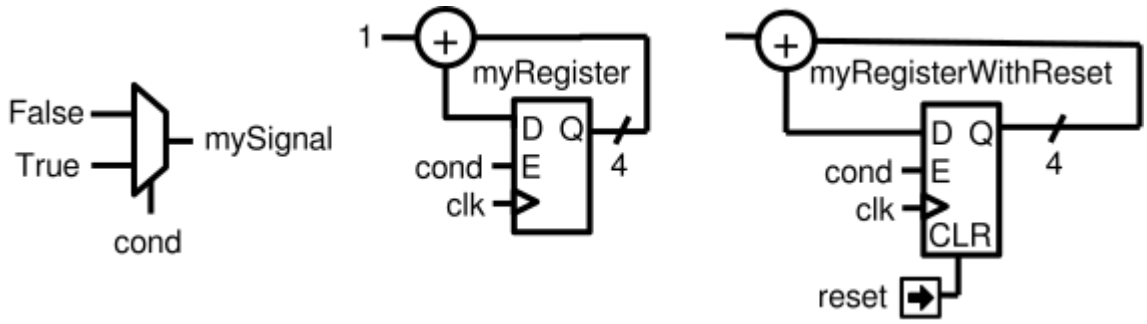
Introduction

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';
```

(continues on next page)

(continued from previous page)

```
    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 bits))
  // ...
  // 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
```

(continues on next page)

(continued from previous page)

```
}

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)
coreConfig.add(new BarrelShifterFullExtension)

val iCacheConfig = InstructionCacheConfig(
  cacheSize = 4096,
```

(continues on next page)

(continued from previous page)

```
  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.

3.7.2 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
}
```

3.8 Cheatsheets

3.8.1 Core

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

3.8.2 Lib

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

3.8.3 Symbolic

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

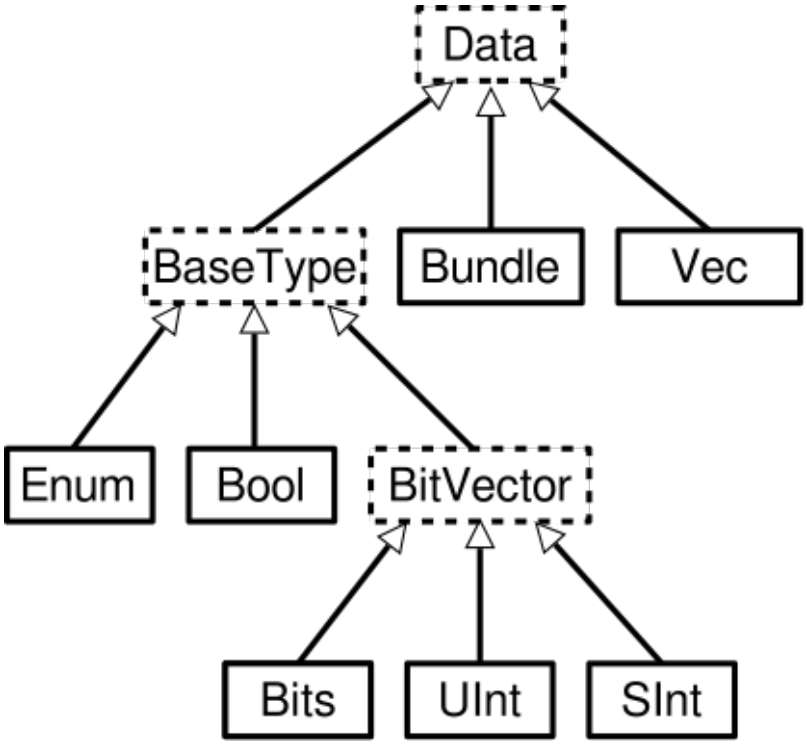
CHAPTER

FOUR

DATA TYPES

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)
- *Auto-range Fixed-point* numbers (add,sub,mul support)
- *Floating-point* numbers (experimental support)

Additionaly, if you want to assign a don't care value to some hardware, for instance, to provide a default value, you can use the assignDontCare API to do so.

```
val myBits  = Bits(8 bits)
myBits.assignDontCare() // Will assign all the bits to 'x'
```

Finally, a special type is available for checking equality between a BitVector and a bit constant pattern that contains holes defined like a bitmask (bit positions not to be compared by the equality expression).

Here is an example to show how you can achieve this (note the use of 'M' prefix) :

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

4.1 Bool

4.1.1 Description

The `Bool` type corresponds to a boolean value (True or False) or a single bit/wire used in a hardware design. While named similarly it should not be confused with Scala *Boolean* type which does not describe hardware but truth values in the Scala generator code.

An important concept and rule-of-thumb to understand is that the Scala *Boolean* type is used in places where elaboration-time HDL code-generation decision making is occuring in Scala code. Like any regular program it affects execution of the Scala program that is SpinalHDL at the time the program is being run to perform HDL code generation.

Therefore the value of a Scala *Boolean* can not be observed from hardware, because it only exists ahead-of-time in the SpinalHDL program at the time of HDL code-gen.

In scenarios where you might need this for your design, for example to pass a value (that maybe acting as a parameterized constant input) from Scala into your hardware design, you can type convert it to Bool with the constructor *Bool(value: Boolean)*.

Similarly the value of a SpinalHDL *Bool* can not be seen at code-generation, all that can be seen and manipulated is the HDL construct concerning a *wire* and how it is routed (through modules/Components), driven (sourced) and connected (sunk).

The signal direction of assignment operators *:=* is managed by SpinalHDL. The use of the Bool instance on the left-hand-side or the right-hand-side of the assignment operator *:=* dictates if it is a source (provides state) or sink (captures state) for a given assignment.

Multiple uses of the assignment operator are allowed, such that it is normal for a signal wire to act as a source (provides a value to drive HDL state) to be able to connect and drive multiple inputs of other HDL constructs. When a Bool instance used as a source the order the assignment statements appear or are executed in Scala does not matter, unlike when it is used as a sink (captures state).

When multiple assignment operators drive the Bool (the Bool is on the left-hand-side of the assignment expression), the last assignment statement wins rule; take effect. The last would be the last to execute in Scala code. This matter can affect the layout and ordering of your SpinalHDL Scala code to ensure the correct precedence order is archived in the hardware design for assigning a new state to the Bool in hardware.

It may help to understand the concept with relating the Scala/SpinalHDL *Bool* instance as a reference to a HDL *net* in the net-list. Which the assignment *:=* operator is attaching HDL constructs into the same net.

4.1.2 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 value from a Scala Boolean type (true, false). This explicitly converts to `True` or `False`.	Bool

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

val myBool_2 = False           // Equivalent to the code above

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

4.1.3 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	Logical NOT	Bool
x.set[()]	Set x to True	Unit (none)
x.clear[()]	Set x to False	Unit (none)
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)
 */
```

(continues on next page)

(continued from previous page)

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

Edge detection

All edge detection functions will instantiate an additional register via *RegNext* to get a delayed value of the `Bool` in question.

This feature does not reconfigure a D-type Flip-Flop to use an alternative CLK source, it uses two D-type Flip-Flop in series chain (with both CLK pins inheriting the default ClockDomain). It has combinational logic to perform edge detection based on the output Q states.

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
x.toggle[()]	Return True at every edge	Bool

```
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
}
```

(continues on next page)

(continued from previous page)

```
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(1 bit)
x.asUInt	Binary cast to UInt	UInt(1 bit)
x.asSInt	Binary cast to SInt	SInt(1 bit)
x.asUInt(bitCount)	Binary cast to UInt and resize, putting Bool value in LSB and padding with zeros.	UInt(bitCount bits)
x.asBits(bitCount)	Binary cast to Bits and resize, putting Bool value in LSB and padding with zeros.	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)
x #* n	Repeat x n-times	Bits(n bits)

```
val a, b, c = Bool()

// Concatenation of three Bool into a single Bits(3 bits) type
val myBits = a ## b ## c
```

MaskedBoolean

A masked boolean allows don't care values. They are usually not used on their own but through *MaskedLiteral*.

```
// first argument: Scala Boolean value
// second argument: do we care ? expressed as a Scala Boolean
val masked = new MaskedBoolean(true, false)
```

4.2 Bits

The `Bits` type is a vector of bits without conveying any arithmetic meaning.

4.2.1 Declaration

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

Syntax	Description
Bits [()]	Create Bits, bit count is inferred from the widest assignment statement after construction
Bits(x bits)	Create Bits with x bits
B(value: Int[, x bits]) B(value: BigInt[, x bits])	Create Bits with x bits assigned with 'value'
B"[[size']base]value"	Create Bits assigned with 'value' (base: 'h', 'd', 'o', 'b')
B([x bits,] elements: Element*)	Create Bits assigned with the value specified by *elements*

```
val myBits1 = Bits(32 bits)
val myBits2 = B(25, 8 bits)
val myBits3 = B"8'xFF"   // Base could be x,h (base 16)
                         //               d   (base 10)
                         //               o   (base 8)
                         //               b   (base 2)
val myBits4 = B"1001_0011"  // _ can be used for readability

// Bits with all ones ("11111111")
val myBits5 = B(8 bits, default -> True)

// initialize with "10111000" through a few elements
val myBits6 = B(8 bits, (7 downto 5) -> B"101", 4 -> true, 3 -> True, default ->␣
↪false)

// "10000000" (For assignment purposes, you can omit the B)
val myBits7 = Bits(8 bits)
myBits7 := (7 -> true, default -> false)
```

When inferring the width of a `Bits` the sizes of assigned values still have to match the final size of the signal:

```
// Declaration
val myBits = Bits()     // the size is inferred from the widest assignment
// ....
// .resized needed to prevent WIDTH MISMATCH error as the constants
// width does not match size that is inferred from assignment below
myBits := B("1010").resized  // auto-widen Bits(4 bits) to Bits(6 bits)
when(condxMaybe) {
  // Bits(6 bits) is inferred for myBits, this is the widest assignment
  myBits := B("110000")
}
```

4.2.2 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

y = 1 // Int
x >> y
	Logical shift right, y: Int
	Result may reduce width
		Bits(w(x) - y bits)

y = U(1) // UInt
x >> y
	Logical shift right, y: UInt
	Result is same width
		Bits(w(x) bits)

y = 1 // Int
x << y
	Logical shift left, y: Int
	Result may increase width
		Bits(w(x) + y bits)

y = U(1) // UInt
x << y
	Logical shift left, y: UInt
	Result may increase width
		Bits(w(x) + max(y) bits)

x |>> y	Logical shift right, y: Int/UInt Result is same width	Bits(w(x) bits)
x |<< y	Logical shift left, y: Int/UInt Result is same width	Bits(w(x) bits)
x.rotateLeft(y)	Logical left rotation, y: UInt/Int Result is same width	Bits(w(x) bits)
x.rotateRight(y)	Logical right rotation, y: UInt/Int Result is same width	Bits(w(x) bits)
x.clearAll[()]	Clear all bits	*modifies x*
x.setAll[()]	Set all bits	*modifies x*
x.setAllTo(value: Boolean)	Set all bits to the given Boolean value	*modifies x*
x.setAllTo(value: Bool)	Set all bits to the given Bool value	*modifies x*

```
// 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

// 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) {
  // ...
}

val notMySpecialValue = 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))
x.asBool	Extract LSB of `x`	Bool(x.lsb)
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)` or `B(something[, x bits])`:

```
// 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)
```

(continues on next page)

(continued from previous page)

```
// Cast the same SInt to Bits but resize to 3 bits
//  (will expand/truncate as necessary, retaining LSB)
val myBits = B(mySInt, 3 bits)
```

Bit extraction

All of the bit extraction operations can be used to read a bit / group of bits. Like in other HDLs the extraction operators can also be used to assign a part of a `Bits`.

All of the bit extraction operations can be used to read a bit / group of bits. Like in other HDLs They can also be used to select a range of bits to be written.

Operator	Description	Return
x(y: Int)	Static bit access of y-th bit	Bool
x(y: UInt)	Variable bit access of y-th bit	Bool
x(offset: Int, width bits)	Fixed part select of fixed width, offset is LSB index	Bits(width bits)
x(offset: UInt, width bits)	Variable part-select of fixed width, offset is LSB index	Bits(width bits)
x(range: Range)	Access a *range* of bits. Ex : myBits(4 downto 2)	Bits(range.size bits)
x.subdivideIn(y slices, [strict: Boolean])	Subdivide x into y slices, y: Int	Vec(Bits(...), y)
x.subdivideIn(y bits, [strict: Boolean])	Subdivide x in multiple slices of y bits, y: Int	Vec(Bits(y bit), ...)
x.msb	Access most significant bit of x (highest index)	Bool
x.lsb	Access lowest significant bit of x (index 0)	Bool

Some basic examples:

```
// get the element at the index 4
val myBool = myBits(4)
// assign element 1
myBits(1) := True

// index dynamically
val index = UInt(2 bit)
val indexed = myBits(index, 2 bit)

// range index
val myBits_8bit = myBits_16bit(7 downto 0)
val myBits_7bit = myBits_16bit(0 to 6)
val myBits_6bit = myBits_16bit(0 until 6)
// assign to myBits_16bit(3 downto 0)
myBits_8bit(3 downto 0) := myBits_4bit

// equivalent slices, no reversing occurs
val a = myBits_16bit(8 downto 4)
val b = myBits_16bit(4 to 8)

// read / assign the msb / leftmost bit / x.high bit
val isNegative = myBits_16bit.msb
myBits_16bit.msb := False
```

Subdivide details

Both overloads of `subdivideIn` have an optional parameter `strict` (i.e. `subdivideIn(slices: SlicesCount, strict: Boolean = true)`). If `strict` is `true` an error will be raised if the input could not be divided into equal parts. If set to `false` the last element may be smaller than the other (equal sized) elements.

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

 // We can also assign through subdivides
 val output8 = Bits(8 bit)
 val pieces = output8.subdivideIn(2 slices)
 // assign to output8
 pieces(0) := 0xf
 pieces(1) := 0x5
```

Misc

In contrast to the bit extraction operations listed above it's not possible to use the return values to assign to the original signal.

Operator	Description	Return
x.getWidth	Return bitcount	Int
x.bitsRange	Return the range (0 to x.high)	Range
x.valueRange	Return the range of minimum to maximum x values, interpreted as an unsigned integer (0 to 2 ** width - 1).	Range
x.high	Return the index of the MSB (highest allowed zero-based index for x)	Int
x.reversed	Return a copy of x with reverse bit order, MSB<>LSB are mirrored.	Bits(w(x) bits)
x ## y	Concatenate, x->high, y->low	Bits(w(x) + w(y) bits)
x #* n	Repeat x n-times	Bits(w(x) * n bits)
x.resize(y)	Return a resized representation of x, if enlarged, it is extended with zero padding at MSB as necessary, y: Int	Bits(y bits)
x.resized	Return a version of x which is allowed to be automatically resized were needed. The resize operation is deferred until the point of assignment later. The resize may widen or truncate, retaining the LSB.	Bits(w(x) bits)
x.resizeLeft(x)	Resize by keeping MSB at the same place, x:Int The resize may widen or truncate, retaining the MSB.	Bits(x bits)
x.getZero	Return a new instance of Bits that is assigned a constant value of zeros the same width as x.	Bits(0, w(x) bits)
x.getAllTrue	Return a new instance of Bits that is assigned a constant value of ones the same width as x.	Bits(w(x) bits).setAll()

Note: *validRange* can only be used for types where the minimum and maximum values fit into a signed 32-bit integer. (This is a limitation given by the Scala `scala.collection.immutable.Range` type which uses *Int*)

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

// Concatenation
myBits_24bits := bits_8bits_1 ## bits_8bits_2 ## bits_8bits_3
// or
myBits_24bits := Cat(bits_8bits_1, bits_8bits_2, bits_8bits_3)

// 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)
```

4.2.3 MaskedLiteral

MaskedLiteral values are bit vectors with don't care values denoted with `-`. They can be used for direct comparison or for `switch` statements and `mux` es.

```
val myBits = B"1101"

val test1 = myBits === M"1-01" // True
val test2 = myBits === M"0---" // False
val test3 = myBits === M"1--1" // True
```

4.3 UInt/SInt

The `UInt`/`SInt` types are vectors of bits interpreted as two's complement unsigned/signed integers. They can do what `Bits` can do, with the addition of unsigned/signed integer arithmetic and comparisons.

4.3.1 Declaration

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

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

```
val myUInt = UInt(8 bit)
myUInt := U(2, 8 bit)
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 = Bool()
myBool := myUInt === U(7 -> true, (6 downto 0) -> false)
myBool := myUInt === U(8 bit, 7 -> true, default -> false)
myBool := myUInt === U(myUInt.range -> true)

// For assignment purposes, you can omit the U/S
// which also allows the use of "default -> ???"
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"
```

4.3.2 Operators

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

Logic

Operator	Description	Return type
~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.xorR	XOR all bits of x (reduction operator)	Bool
x.orR	OR all bits of x (reduction operator)	Bool
x.andR	AND all bits of x (reduction operator)	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 The width of y is constrained to the width of log2Up(x) or less	T(w(x) bits)
x.rotateRight(y)	Logical right rotation, y : UInt/Int The width of y is constrained to the width of log2Up(x) or less	T(w(x) bits)
x.clearAll[()]	Clear all bits	*modifies x*
x.setAll[()]	Set all bits	*modifies x*
x.setAllTo(value : Boolean)	Set all bits to the given Boolean value	*modifies x*
x.setAllTo(value : Bool)	Set all bits to the given Bool value	*modifies x*

Note: Notice the difference in behaviour between `x >> 2` (result 2 bit narrower than x) and `x >> U(2)` (keeping width) due to the Scala type of `y`.

In the first case "2" is an `Int` (which can be seen as an "elaboration integer constant"), and in the second case it is a hardware signal (type `UInt`) that may or may not be a constant.

```
val a, b, c = SInt(32 bits)
a := S(5)
b := S(10)

// Bitwise operators
c := ~(a & b)      // Inverse(a AND b)
assert(c.getWidth == 32)

// Shift
val arithShift = UInt(8 bits) << 2      // shift left (resulting in 10 bits)
```

(continues on next page)

(continued from previous page)

```
val logicShift = UInt(8 bits) |<< 2     // shift left (resulting in 8 bits)
assert(arithShift.getWidth == 10)
assert(logicShift.getWidth == 8)

// Rotation
val rotated = UInt(8 bits) rotateLeft 3 // left bit rotation
assert(rotated.getWidth == 8)

// Set all bits of b to True when all bits of a are True
when(a.andR) { b.setAll() }
```

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 of addend with saturation (see also *T.maxValue* and *T.minValue*)	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 of subtrahend with saturation (see also *T.minValue* and *T.maxValue*)	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(min(w(x), w(y)) bits)
~x	Unary One's compliment, Bitwise NOT	T(w(x) bits)
-x	Unary Two's compliment of SInt type. Not available for UInt.	SInt(w(x) bits)

```
val a, b, c = UInt(8 bits)
a := U"xf0"
b := U"x0f"

c := a + b
assert(c === U"8'xff")

val d = a +^ b
assert(d === U"9'x0ff")

// 0xf0 + 0x20 would overflow, the result therefore saturates
val e = a +| U"8'x20"
assert(e === U"8'xff")
```

Note: Notice how simulation assertions are made here (with ===), as opposed to elaboration assertions in the previous example (with ==).

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

```
val a = U(5, 8 bits)
val b = U(10, 8 bits)
val c = UInt(2 bits)

when (a > b) {
  c := U"10"
} elsewhen (a =/= b) {
  c := U"01"
} elsewhen (a === U(0)) {
  c.setAll()
} otherwise {
  c.clearAll()
}
```

Note: When comparing `UInt` values in a way that allows for "wraparound" behavior, meaning that the values will "wrap around" to the minimum value when they exceed the maximum value. The `wrap` method of `UInt` can be used as `x.wrap < y` for `UInt` variables `x, y`, the result will be true if `x` is less than `y` in the wraparound sense.

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))
x.asBool	Extract LSB of `x`	Bool(x.lsb)
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 expanding sign bit	SInt(w(x) + 1 bits)
myUInt.twoComplement(en: Bool)	Generate two's complement of number if `en` is `True`, unchanged otherwise. (`en` makes result negative)	SInt(w(myUInt) + 1, bits)
mySInt.abs	Return the absolute value as a UInt value	UInt(w(mySInt) bits)
mySInt.abs(en: Bool)	Return the absolute value as a UInt value when `en` is `True`, otherwise just reinterpret bits as unsigned	UInt(w(mySInt) bits)
mySInt.absWithSym	Return the absolute value of the UInt value with symmetric, shrink 1 bit	UInt(w(mySInt) - 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)

// UInt to SInt conversion
val UInt_30 = U(30, 8 bit)

val SInt_30 = UInt_30.intoSInt
assert(SInt_30 === S(30, 9 bit))

mySInt := UInt_30.twoComplement(booleanDoInvert)
    // if booleanDoInvert is True then we get S(-30, 9 bit)
    // otherwise we get S(30, 9 bit)

// absolute values
val SInt_n_4 = S(-3, 3 bit)
val abs_en = SInt_n_3.abs(booleanDoAbs)
    // if booleanDoAbs is True we get U(3, 3 bit)
    // otherwise we get U"3'b101" or U(5, 3 bit) (raw bit pattern of -3)

val SInt_n_128 = S(-128, 8 bit)
val abs = SInt_n_128.abs
assert(abs === U(128, 8 bit))
val sym_abs = SInt_n_128.absWithSym
assert(sym_abs === U(127, 7 bit))
```

Bit extraction

All of the bit extraction operations can be used to read a bit / group of bits. Like in other HDLs the extraction operators can also be used to assign a part of a `UInt` / `SInt` .

Operator	Description	Return
x(y: Int)	Static bit access of y-th bit	Bool
x(x: UInt)	Variable bit access of y-th bit	Bool
x(offset: Int, width bits)	Fixed part select of fixed width, offset is LSB index	Bits(width bits)
x(offset: UInt, width bits)	Variable part-select of fixed width, offset is LSB index	Bits(width bits)
x(range: Range)	Access a *range* of bits. Ex : myBits(4 downto 2)	Bits(range.size bits)
x.subdivideIn(y slices, [strict: Boolean])	Subdivide x into y slices, y: Int	Vec(Bits(...), y)
x.subdivideIn(y bits, [strict: Boolean])	Subdivide x in multiple slices of y bits, y: Int	Vec(Bits(y bit), ...)
x.msb	Access most significant bit of x (highest index, sign bit for SInt)	Bool
x.lsb	Access lowest significant bit of x (index 0)	Bool
mySInt.sign	Access most sign bit, only SInt	Bool

Some basic examples:

```
// get the element at the index 4
val myBool = myUInt(4)
// assign element 1
myUInt(1) := True

// index dynamically
val index = UInt(2 bit)
val indexed = myUInt(index, 2 bit)

// range index
val myUInt_8bit = myUInt_16bit(7 downto 0)
val myUInt_7bit = myUInt_16bit(0 to 6)
val myUInt_6bit = myUInt_16bit(0 until 6)
// assign to myUInt_16bit(3 downto 0)
myUInt_8bit(3 downto 0) := myUInt_4bit

// equivalent slices, no reversing occurs
val a = myUInt_16bit(8 downto 4)
val b = myUInt_16bit(4 to 8)

// read / assign the msb / leftmost bit / x.high bit
val isNegative = mySInt_16bit.sign
myUInt_16bit.msb := False
```

Subdivide details

Both overloads of `subdivideIn` have an optional parameter `strict` (i.e. `subdivideIn(slices: SlicesCount, strict: Boolean = true)`). If `strict` is `true` an error will be raised if the input could not be divided into equal parts. If set to `false` the last element may be smaller than the other (equal sized) elements.

```
// Subdivide
val sel = UInt(2 bits)
val myUIntWord = myUInt_128bits.subdivideIn(32 bits)(sel)
    // sel = 3 => myUIntWord = myUInt_128bits(127 downto 96)
    // sel = 2 => myUIntWord = myUInt_128bits( 95 downto 64)
    // sel = 1 => myUIntWord = myUInt_128bits( 63 downto 32)
    // sel = 0 => myUIntWord = myUInt_128bits( 31 downto  0)

 // If you want to access in reverse order you can do:
 val myVector    = myUInt_128bits.subdivideIn(32 bits).reverse
 val myRevUIntWord = myVector(sel)

 // We can also assign through subdivides
 val output8 = UInt(8 bit)
 val pieces = output8.subdivideIn(2 slices)
 // assign to output8
 pieces(0) := 0xf
 pieces(1) := 0x5
```

Misc

In contrast to the bit extraction operations listed above it's not possible to use the return values to assign to the original signal.

Operator	Description	Return
x.getWidth	Return bitcount	Int
x.high	Return the index of the MSB (highest allowed index for Int)	Int
x.bitsRange	Return the range (0 to x.high)	Range
x.minValue	Lowest possible value of x (e.g. 0 for UInt)	BigInt
x.maxValue	Highest possible value of x	BigInt
x.valueRange	Return the range from minimum to maximum possible value of x (x.minValue to x.maxValue).	Range
x ## y	Concatenate, x->high, y->low	Bits(w(x) + w(y) bits)
x #* n	Repeat x n-times	Bits(w(x) * n bits)
x @@ y	Concatenate x:T with y:Bool/SInt/UInt	T(w(x) + w(y) bits)
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)
x.expand	Return x with 1 bit expand	T(w(x)+1 bits)
x.getZero	Return a new instance of type T that is assigned a constant value of zeros the same width as x.	T(0, w(x) bits).clearAll()
x.getAllTrue	Return a new instance of type T that is assigned a constant value of ones the same width as x.	T(w(x) bits).setAll()

Note: *validRange* can only be used for types where the minimum and maximum values fit into a signed 32-bit integer. (This is a limitation given by the Scala `scala.collection.immutable.Range` type which uses *Int*)

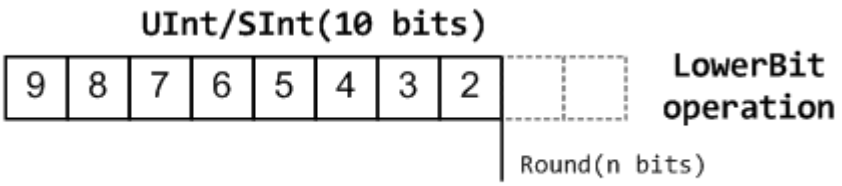
```
myBool := mySInt.lsb  // equivalent to mySInt(0)

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

// Resize
myUInt_32bits := U"32'x112233344"
myUInt_8bits  := myUInt_32bits.resized      // automatic resize (myUInt_8bits = 0x44)
val lowest_8bits = myUInt_32bits.resize(8)  // resize to 8 bits (myUInt_8bits = 0x44)
```

4.3.3 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 bits)
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.

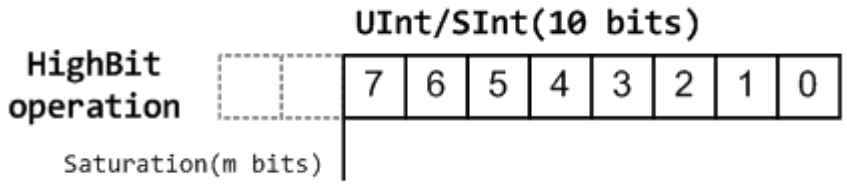
round Api

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



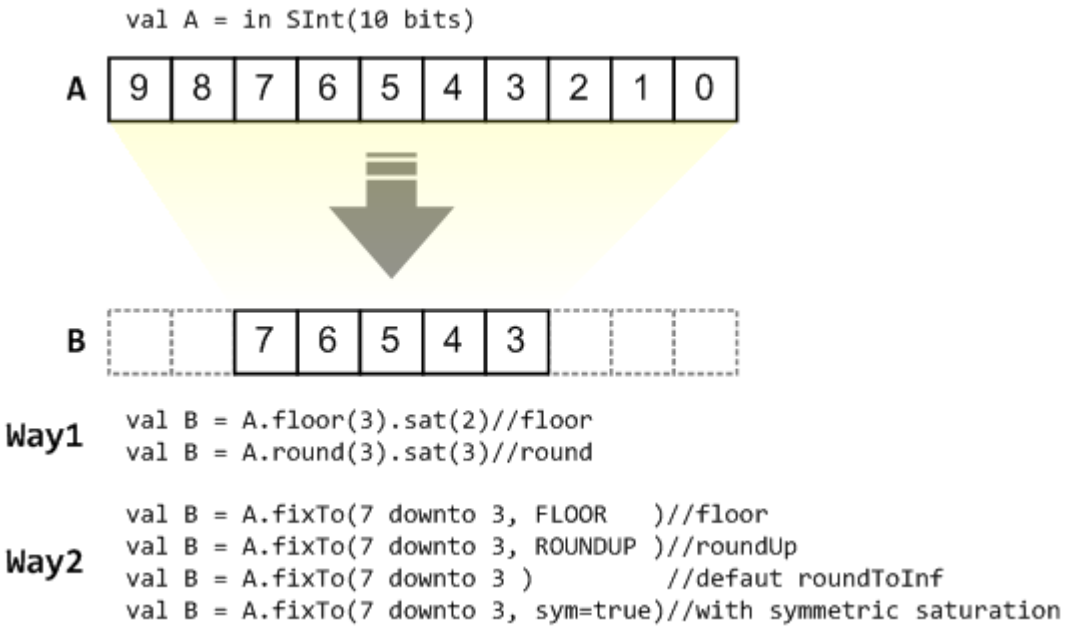
function	Operation	Positive-Op	Negative-Op
sat	Saturation	when(Top[w-1, w-n].orR) set maxValue	When(Top[w-1, w-n].andR) set minValue
trim	Discard	N/A	N/A
symmetry	Symmetric	N/A	minValue = -maxValue

Symmetric is only valid for `SInt`.

```
val A   = SInt(8 bits)
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 bits)
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
```

4.4 SpinalEnum

4.4.1 Description

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

4.4.2 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`	Hot-Count	Use Bits to store state. Each bit corresponds to one state, only one bit is set at a time in the hardware encoded state representation.
`graySequential`	`log2Up(stateCount)`	Encode index (numbers as if using `binarySequential`) as binary gray code.

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
```

4.4.3 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) {
    ...
  }
  ...
}
```

Types

In order to use your enums, for example in a function, you may need its type.

The value type (e.g. sIdle's type) is

```
spinal.core.SpinalEnumElement[UartCtrlTxState.type]
```

or equivalently

```
UartCtrlTxState.E
```

The bundle type (e.g. stateNext's type) is

```
spinal.core.SpinalEnumCraft[UartCtrlTxState.type]
```

or equivalently

```
UartCtrlTxState.C
```

Type cast

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

```
import UartCtrlTxState._

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

stateNext.assignFromBits(myBits)
```

4.5 Bundle

4.5.1 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.

4.5.2 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*.

Conditional signals

The signals in the `Bundle` can be defined conditionally. Unless `dataWidth` is greater than 0, there will be no `data` signal in elaborated `myBundle`, as demonstrated in the example below.

```
case class myBundle(dataWidth: Int) extends Bundle {
  val data = (dataWidth > 0) generate (UInt(dataWidth bits))
}
```

Note: See also *generate* for information about this SpinalHDL method.

4.5.3 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  // Compare all elements of the bundle
// is equivalent to:
//myBool := color1.r === color2.r && color1.g === color2.g && color1.b === color2.b
```

Type cast

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

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

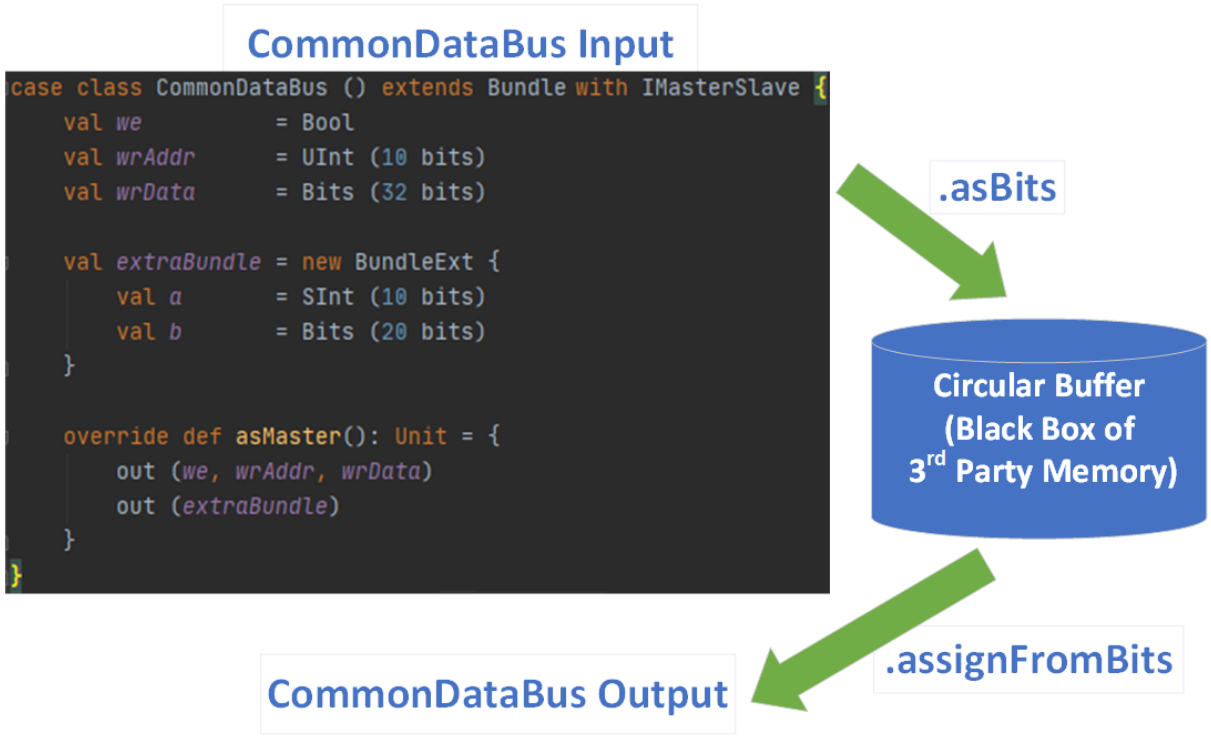
The elements of the bundle will be mapped into place in the order in which they are defined, LSB first. Thus, `r` in `color1` will occupy bits 0 to 8 of `myBits` (LSB), followed by `g` and `b` in that order, with `b.msb` also being the MSB of the resulting Bits type.

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())
```

(continues on next page)

(continued from previous page)

```
  }

  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)
}
```

4.5.4 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.

There are functions defined as toXXX, such as the `toStream` method of the `Flow` class. These functions can usually be called by the master side. In addition, the fromXXX functions are designed for the slave side. It is common that there are more functions available for the master side than for the slave side.

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)
```

(continues on next page)

(continued from previous page)

```
  }
}

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

4.6 Vec

4.6.1 Description

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

4.6.2 Declaration

The syntax to declare a vector is as follows:

Declaration	Description
Vec.fill(size: Int)(type: Data)	Create a vector of `size` elements of type `Data`
Vec(x, y, ...)	Create a vector where indexes point to the provided elements. Does not create new hardware signals. This constructor supports mixed element width.

Examples

```
// Create a vector of 2 signed integers
val myVecOfSInt = Vec.fill(2)(SInt(8 bits))
myVecOfSInt(0) := 2                    // assignment to populate index 0
myVecOfSInt(1) := myVecOfSInt(0) + 3   // assignment to populate index 1

// 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
```

4.6.3 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.fill(2)(SInt(8 bits))
val vec1 = Vec.fill(2)(SInt(8 bits))

myBool := vec2 === vec1  // Compare all elements
// is equivalent to:
//myBool := vec2(0) === vec1(0) && vec2(1) === vec1(1)
```

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.fill(2)(SInt(8 bits))

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.fill(2)(SInt(8 bits))

println(widthOf(vec1)) // 16
```

Lib helper functions

Note: You need to import `import spinal.lib._` to put these functions in scope.

Operator	Description	Return
x.sCount(condition: T => Bool)	Count the number of occurence matching a given condition in the Vec.	UInt
x.sCount(value: T)	Count the number of occurence of a value in the Vec.	UInt
x.sExists(condition: T => Bool)	Check if there is a matching condition in the Vec.	Bool
x.sContains(value: T)	Check if there is an element with a given value present in the Vec.	Bool
x.sFindFirst(condition: T => Bool)	Find the first element matching the given condition in the Vec, return if any index was successfully found and the index of that element.	(Bool, UInt)
x.reduceBalancedTree(op: (T, T) => T)	Balanced reduce function, to try to minimize the depth of the resulting circuit. op should be commutative and associative.	T
x.shuffle(indexMapping: Int => Int)	Shuffle the Vec using a function that maps the old indexes to new ones.	Vec[T]

```
import spinal.lib._

// Create a vector with 4 unsigned integers
val vec1 = Vec.fill(4)(UInt(8 bits))

// ... the vector is actually assigned somewhere

val c1: UInt = vec1.sCount(_ < 128) // how many values are lower than 128 in vec
val c2: UInt = vec1.sCount(0) // how many values are equal to zero in vec

val b1: Bool = vec1.sExists(_ > 250) // is there a element bigger than 250
val b2: Bool = vec1.sContains(0) // is there a zero in vec

val (u1Found, u1): (Bool, UInt) = vec1.sFindFirst(_ < 10) // get the index of the␣
↪first element lower than 10
val u2: UInt = vec1.reduceBalancedTree(_ + _) // sum all elements together
```

Note: The sXXX prefix is used to disambiguate with respect to identically named Scala functions that accept a lambda function as argument.

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.

4.7 UFix/SFix

4.7.1 Description

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

4.7.2 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)
```

4.7.3 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 to match assignment target
o14_m2 := i16_m2.truncated  // OK, as it is resized to match assignment target
val o18_m2 = i16_m2.truncated(18 exp, -2 exp)
val o18_22b = i16_m2.truncated(18 exp, 22 bit)
```

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
```

4.7.4 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
```

4.7.5 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.

4.8 Floating

4.8.1 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.

4.8.2 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

4.8.3 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

4.9 AFix

4.9.1 Description

Auto-ranging Fixed-Point, `AFix`, is a fixed-point class which tracks the representable range of values while preforming fixed-point operations.

Warning: Much of this code is still under development. API and function calls may change.

User feedback is appreciated!

4.9.2 Declaration

AFix can be created using bit sizes or exponents:

```
AFix.U(12 bits)             // U12.0
AFix(QFormat(12, 0, false)) // U12.0
AFix.UQ(8 bits, 4 bits)     // U8.4
AFix.U(8 exp, 12 bits)      // U8.4
AFix.U(8 exp, -4 exp)       // U8.4
AFix.U(8 exp, 4 exp)        // U8.-4
AFix(QFormat(12, 4, false)) // U8.4

AFix.S(12 bits)             // S11.0 + sign
AFix(QFormat(12, 0, true))  // S11.0 + sign
AFix.SQ(8 bits, 4 bits)     // S8.4  + sign
AFix.S(8 exp, 12 bits)      // S8.3  + sign
AFix.S(8 exp, -4 exp)       // S8.4  + sign
AFix(QFormat(12, 4, true))  // S7.4  + sign
```

These will have representable ranges for all bits.

For example:

`AFix.U(12 bits)` will have a range of 0 to 4095.

`AFix.SQ(8 bits, 4 bits)` will have a range of -4096 (-256) to 4095 (255.9375)

`AFix.U(8 exp, 4 exp)` will have a range of 0 to 256

Custom range `AFix` values can be created be directly instantiating the class.

```
class AFix(val maxValue: BigInt, val minValue: BigInt, val exp: ExpNumber)

new AFix(4096, 0, 0 exp)      // [0 to 4096, 2^0]
new AFix(256, -256, -2 exp)   // [-256 to 256, 2^-2]
new AFix(16, 8, 2 exp)        // [8 to 16, 2^2]
```

The `maxValue` and `minValue` stores what backing integer values are representable. These values represent the true fixed-point value after multiplying by `2^exp`.

`AFix.U(2 exp, -1 exp)` can represent: `0, 0.5, 1.0, 1.5, 2, 2.5, 3, 3.5`

`AFix.S(2 exp, -2 exp)` can represent: `-2.0, -1.75, -1.5, -1.25, -1, -0.75, -0.5, -0.25, 0, 0.25, 0.5, 0.75, 1, 1.25, 1.5, 1.75`

Exponent values greater 0 are allowed and represent values which are larger than 1.

`AFix.S(2 exp, 1 exp)` can represent: `-4, 2, 0, 2`

`AFix(8, 16, 2 exp)` can represent: `32, 36, 40, 44, 48, 52, 56, 60, 64`

Note: `AFix` will use 5 bits to save this type as that can store 16, its `maxValue`.

4.9.3 Mathematical Operations

`AFix` supports Addition (+), Subtraction (-), and Multiplication (*) at the hardware level. Division (\) and Modulo (%) operators are provided but are not recommended for hardware elaboration.

Operations are preformed as if the `AFix` value is a regular `Int` number. Signed and unsigned numbers are interoperable. There are no type differences between signed or unsigned values.

```
// Integer and fractional expansion
val a = AFix.U(4 bits)          // [   0 (  0.)     to  15 (15.  )]  4 bits, 2^0
val b = AFix.UQ(2 bits, 2 bits) // [   0 (  0.)     to  15 ( 3.75)]  4 bits, 2^-2
val c = a + b                   // [   0 (  0.)     to  77 (19.25)]  7 bits, 2^-2
val d = new AFix(-4, 8, -2 exp) // [-  4 (- 1.25)   to   8 ( 2.00)]  5 bits, 2^-2
val e = c * d                   // [-308 (-19.3125) to 616 (38.50)] 11 bits, 2^-4

// Integer without expansion
val aa = new AFix(8, 16, -4 exp) // [8 to 16] 5 bits, 2^-4
val bb = new AFix(1, 15, -4 exp) // [1 to 15] 4 bits, 2^-4
val cc = aa + bb                 // [9 to 31] 5 bits, 2^-4
```

`AFix` supports operations without without range expansion. It does this by selecting the aligned maximum and minimum ranges from each of the inputs.

`+|` Add without expansion. `-|` Subtract without expansion.

4.9.4 Inequality Operations

`AFix` supports standard inequality operations.

```
A === B
A =\= B
A < B
A <= B
A > B
A >= B
```

Warning: Operations which are out of range at compile time will be optimized out!

4.9.5 Bitshifting

AFix supports decimal and bit shifting

<< Shifts the decimal to the left. Adds to the exponent. >> Shifts the decimal to the right. Subtracts from the exponent. <<| Shifts the bits to the left. Adds fractional zeros. >>| Shifts the bits to the right. Removes fractional bits.

4.9.6 Saturation and Rounding

AFix implements saturation and all common rounding methods.

Saturation works by saturating the backing value range of an AFix value. There are multiple helper functions which consider the exponent.

```
val a = new AFix(63, 0, -2 exp) // [0 to 63, 2^-2]
a.sat(63, 0)                    // [0 to 63, 2^-2]
a.sat(63, 0, -3 exp)            // [0 to 31, 2^-2]
a.sat(new AFix(31, 0, -1 exp))  // [0 to 31, 2^-2]
```

AFix rounding modes:

```
// The following require exp < 0
.floor() or .truncate()
.ceil()
.floorToZero()
.ceilToInf()
// The following require exp < -1
.roundHalfUp()
.roundHalfDown()
.roundHalfToZero()
.roundHalfToInf()
.roundHalfToEven()
.roundHalfToOdd()
```

A mathematical example of these rounding modes is better explained here: Rounding - Wikipedia

All of these modes will result in an AFix value with 0 exponent. If rounding to a different exponent is required consider shifting or use an assignment with the truncated tag.

4.9.7 Assignment

AFix will automatically check and expand range and precision during assignment. By default, it is an error to assign an AFix value to another AFix value with smaller range or precision.

The .truncated function is used to control how assignments to smaller types.

```
def truncated(saturation: Boolean = false,
              overflow  : Boolean = true,
              rounding  : RoundType = RoundType.FLOOR)

def saturated(): AFix = this.truncated(saturation = true, overflow = false)
```

RoundType:

```
RoundType.FLOOR
RoundType.CEIL
RoundType.FLOORTOZERO
```

(continues on next page)

(continued from previous page)

```
RoundType.CEILTOINF
RoundType.ROUNDUP
RoundType.ROUNDDOWN
RoundType.ROUNDTOZERO
RoundType.ROUNDTOINF
RoundType.ROUNDTOEVEN
RoundType.ROUNDTOODD
```

The `saturation` flag will add logic to saturate to the assigned datatype range.

The `overflow` flag will allow assignment directly after rounding without range checking.

Rounding is always required when assigning a value with more precision to one with lower precision.

CHAPTER

FIVE

STRUCTURING

The chapters below explain:

- how to build reusable components
- alternatives to components to group hardware
- handling of clock/reset domains
- instantitation of existing VHDL and Verilog IP
- how names are assigned in SpinalHDL, and how naming can be influenced

5.1 Components and hierarchy

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 port Bool()
    val sum, cout = out port 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.

Tip: If it is better to your taste, you can use the `Module` syntax instead of `Component` (they are the same thing)

5.1.1 Input / output definition

The syntax to define inputs and outputs is as follows:

Syntax	Description	Return
`in port Bool()` `out port Bool()`	Create an input Bool/output Bool	Bool

`in Bits/UInt/SInt[(x bits)]`
`out Bits/UInt/SInt[(x bits)]`
`in Bits(3 bits)`
	Create an input/output of the corresponding type	Bits/UInt/SInt

`in(T)`
`out(T)`
`out UInt(7 bits)`
	For all other data types, you may have to add some brackets around it. Sorry, this is a Scala limitation.	T

`master(T)`
`slave(T)`
`master(Bool())`
	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 must 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()`

5.1.2 Pruned signals

SpinalHDL will generate all the named signals and their depedencies, while all the useless anonymous / zero width ones are removed from the RTL generation.

You can collect the list of all the removed ans useless signals via the `printPruned` and the `printPrunedIo` functions on the generated `SpinalReport` object:

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

5.1.3 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 port UInt(width)
    val result = out port 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)
```

(continues on next page)

(continued from previous page)

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

You can add functions inside the config, along with requirements on the config attributes:

```
case class MyBusConfig(addressWidth: Int, dataWidth: Int) {
  def bytePerWord = dataWidth / 8
  def addressType = UInt(addressWidth bits)
  def dataType = Bits(dataWidth bits)

  require(dataWidth == 32 || dataWidth == 64, "Data width must be 32 or 64")
}
```

Note: This parametrization occurs entirely within the SpinalHDL code-generation during elaboration. This generates non-generic HDL code. The methods described here do not use VHDL generics or Verilog parameters.

See also *Blackbox* for more information around support for that mechanism.

5.1.4 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`.

5.2 Area

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 bits))
    val tick = counter === 0
    counter := counter - 1
    when(tick) {
      counter := 100
    }
  }

  val tickCounter = new Area {
    val value = Reg(UInt(3 bits))
    val reset = False
    when(timer.tick) {          // Refer to the tick from timer area
```

(continues on next page)

(continued from previous page)

```
      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`

5.3 Function

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.

5.3.1 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)
```

5.3.2 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
  }

  // 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)

  // ...

}
```

5.4 Clock domains

5.4.1 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.

Please note that a register captures its clock domain when the register is created, not when it is assigned. So please make sure to create them inside the desired `ClockingArea`.

5.4.2 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. This parameter does not generate and PLL or other hardware to control the frequency	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 bits))
}
```

When an *Area* is not needed, it is also possible to apply the clock domain directly. Two syntaxes exist:

```
class Counters extends Component {
  val io = new Bundle {
    val enable = in Bool ()
    val freeCount, gatedCount, gatedCount2 = out UInt (4 bits)
  }
  val freeCounter = CounterFreeRun(16)
  io.freeCount := freeCounter.value

  // In a real design consider using a glitch free single purpose CLKGATE primitive␣
↪instead
  val gatedClk = ClockDomain.current.readClockWire && io.enable
```

(continues on next page)

(continued from previous page)

```
  val gated = ClockDomain(gatedClk, ClockDomain.current.readResetWire)

  // Here the "gated" clock domain is applied on "gatedCounter" and "gatedCounter2"
  val gatedCounter = gated(CounterFreeRun(16))
  io.gatedCount := gatedCounter.value
  val gatedCounter2 = gated on CounterFreeRun(16)
  io.gatedCount2 := gatedCounter2.value

  assert(gatedCounter.value === gatedCounter2.value, "gated count mismatch")
}
```

Configuration

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

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(
      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
```

(continues on next page)

(continued from previous page)

```
  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
  }
}
```

Warning: In other components then the one you created the ClockDomain in, you must not use `.clock` and `.reset`, but `.readClockWire` and `.readResetWire` as listed below. For the global ClockDomain you must always use those `.readXXX` functions.

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")

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

    io.result := myReg
  }
}
```

Signal priorities in HDL generation

In the current version, reset and clock enable signals have different priorities. Their order is : `asyncReset`, `clockEnable`, `syncReset` and `softReset`.

Please be careful that clockEnable has a higher priority than syncReset. If you do a sync reset when the clockEnable is disabled (especially at the beginning of a simulation), the gated registers will not be reseted.

Here is an example:

```
val clockedArea = new ClockEnableArea(clockEnable) {
  val reg = RegNext(io.input) init(False)
}
```

It will generate VerilogHDL codes like:

```
always @(posedge clk) begin
  if(clockedArea_newClockEnable) begin
    if(!resetn) begin
      clockedArea_reg <= 1'b0;
    end else begin
      clockedArea_reg <= io_input;
    end
  end
end
```

If that behaviour is problematic, one workaround is to use a when statement as a clock enable instead of using the ClockDomain.enable feature. This is open for future improvements.

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. This being the arbitrary value you configured the domain with.	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 reset signal	Bool
readSoftResetWire	Return a signal derived from the soft 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
}
```

5.4.3 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 {
```

(continues on next page)

(continued from previous page)

```
    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) {
    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. For multi-bit cases, it is recommended to use `StreamFifoCC` for high bandwidth requirements, or use `StreamCCByToggle` to reduce resource usage in cases where bandwidth is not critical.

5.4.4 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
  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)
}
```

Warning: The clock signal used in a SlowArea is the same as the parent one. The SlowArea add instead a clock-enable signal that will slow down the sampling rate inside it. In other words, `ClockDomain.current.readClockWire` will return the fast (parent domain) clock. To obtain the clock enable, use `ClockDomain.current.readClockEnableWire`

BootReset

clockDomain.withBootReset() could specify register's resetKind as BOOT. *clockDomain.withSyncReset()* could specify register's resetKind as SYNC (sync-reset).

```
class  Top extends Component {
    val io = new Bundle {
      val data = in Bits(8 bit)
      val a, b, c, d = out Bits(8 bit)
    }
    io.a  :=  RegNext(io.data) init 0
    io.b  :=  clockDomain.withBootReset()  on RegNext(io.data) init 0
    io.c  :=  clockDomain.withSyncReset()  on RegNext(io.data) init 0
    io.d  :=  clockDomain.withAsyncReset() on RegNext(io.data) init 0
}
SpinalVerilog(new Top)
```

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)
  }
}
```

5.5 Instantiate VHDL and Verilog IP

5.5.1 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.

5.5.2 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)
```

(continues on next page)

(continued from previous page)

```
  // 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) bits)
      val data = in Bits (wordWidth bits)
    }
    val rd = new Bundle {
      val en   = in Bool()
      val addr = in UInt (log2Up(wordCount) bits)
      val data = out Bits (wordWidth bits)
    }
  }

  // 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` does not change the generated verilog.

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

5.5.3 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
    }
}
```

5.5.4 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) bits)
      val data = in Bits (8 bits)
    }
    val rd = new Bundle {
      val en   = in Bool()
      val addr = in UInt (log2Up(16) bits)
      val data = out Bits (8 bits)
    }
  }

  // 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)
  }
}
```

5.5.5 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)
}
```

By default the ports of the blackbox are considered clock-less, meaning no clock crossing checks will be made on their usage. You can specify ports clock domain by using the ClockDomainTag :

```
class DemoBlackbox extends BlackBox {
  val io = new Bundle{
    val clk, rst = in Bool()
    val a = in Bool()
    val b = out Bool()
  }
  mapCurrentClockDomain(io.clk, io.rst)
  ClockDomainTag(this.clockDomain)(
    io.a,
    io.b
  )
}
```

You can also apply the tag to the whole bundle with :

```
val io = new Bundle {
  val clk, rst = in Bool()
  val a = in Bool()
  val b = out Bool()
}
ClockDomainTag(this.clockDomain)(io)
```

You can also apply the current clock domain to all the ports using (SpinalHDL 1.10.2):

```
val io = new Bundle {
  val clk, rst = in Bool()
  val a = in Bool()
  val b = out Bool()
}
setIoCd()
```

5.5.6 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) bits)
      val data = in Bits (_wordWidth bits)
    }
    val rd = new Bundle {
      val en   = in Bool()
      val addr = in UInt (log2Up(_wordCount) bits)
      val data = out Bits (_wordWidth bits)
    }
  }

  noIoPrefix()

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

5.5.7 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)
    }
  }
```

(continues on next page)

(continued from previous page)

```
  // 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().replace("portA_", "
↪") + "_A")
      if(bt.getName().contains("portB")) bt.setName(bt.getName().replace("portB_", "
↪") + "_B")
    })
  }

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

// This code generate these names:
//    clk
//    cs_A, rwn_A, dIn_A, dOut_A
//    cs_B, rwn_B, dIn_B, dOut_B
```

5.5.8 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"))
}

...
```

(continues on next page)

(continued from previous page)

```
class TopLevel() extends Component{
  //...
  val bb = new MyBlackBox()
  //...
}

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

5.5.9 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()

  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;
```

5.6 Preserving names

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.

5.6.1 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.

5.6.2 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
```

(continues on next page)

(continued from previous page)

```
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 the content's 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,
  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
```

5.6.3 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
```

5.6.4 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
);
  wire                someLogic_comparator;

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

endmodule
```

5.6.5 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
```

5.6.6 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)
  val result = out Bool()
  result := inverted(isZero(value))
}
```

Will generate :

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

(continues on next page)

(continued from previous page)

```
  wire                value_comparator_inverter;

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

endmodule
```

5.6.7 Composite in a Bundle's function

This behaviour can be very useful when implementing Bundle 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' refers to the Composite construction argument (
↪'this' in
                            //  the example). It avoids 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;
      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
```

(continues on next page)

(continued from previous page)

```
  end
endmodule
```

5.6.8 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)
  // Perform a logical OR between all the condition elements
  val result = conditions.reduce(_ || _)

  // For Bits/UInt/SInt signals the 'orR' methods implements this reduction operation
}
```

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
  }
```

(continues on next page)

(continued from previous page)

```
}
```

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) {
      ret := ret + 1
    }
    return ret
  }

  value := count(enable)
}
```

Will generate

```
module MyComponent (
  input               enable,
  output     [7:0]    value,
  input               clk,
  input               reset
```

(continues on next page)

(continued from previous page)

```
);
  // Name given to the register in last resort by looking what was driven by it
  reg        [7:0]    _zz_value;

  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)

5.7 Parametrization

There are multiple aspects to parametrization :

- Providing and the management of, elaboration time parameters provided to SpinalHDL during elaboration of the design
- Using the parameter data to allow the designer to perform any kind of hardware construction, configuration and interconnection task needed in the design. Such as optional component generation within the hardware design.

Parallels exist with the aims of HDL features such as Verilog module parameters and VHDL generics. SpinalHDL brings a far richer and more powerful set of capabilities into this area with the additional protection of Scala type safety and SpinalHDL built in HDL design rule checking.

The SpinalHDL mechanisms for parameterization of components is not built on top of any native HDL mechanism and so is not impeded by HDL language level/version support or restrictions about what can be achieved in hand written HDL.

For readers looking to interoperate with parameterized Verilog or genericized VHDL using SpinalHDL, please see the section on *BlackBox* IP for those scenarios your project requires.

5.7.1 Elaboration time parameters

You can use the whole Scala syntax to provide elaboration time parameters.

The whole syntax means you have the entire power and feature set of the Scala language at your disposal to solve parameterization requirements for your project at the level of complexity you choose.

SpinalHDL does not place any opinionated restrictions on how to achieve your parameterization goals. As such there are many Scala design patterns and a few SpinalHDL helpers that can be used to manage parameters that are suited to different parameter management scenarios.

Here are some examples and ideas of the possibilities:

- Hardwired code and constants (not strictly parameter management at all but serves to hilight the most basic mechanism, a code change, not a parameter data change)
- Constant values provided from a companion object that are static constants in Scala.
- Values provided to Scala class constructor, often a `case class` that causes Scala to capture those constructor argument values as constants.
- Regular Scala flow-control syntax, not limited to but including conditionals, looping, lambdas/monads, everything.

- Config class pattern (examples exist in library items such as UartCtrlConfig, SpiMasterCtrlConfig)
- Project defined 'Plugin' pattern (examples exist in the VexRiscV project to configure the feature set the resulting CPU IP core is built with)
- Values and information loaded from a file or network based source, using standard Scala/JVM libraries and APIs.
- *any mechanism you can create*

All of the mechanisms result in a change in resulting elaborated HDL output.

This could vary from a single constant value change all the way through to describing the entire bus and interconnection architecture of an entire SoC all without leaving the Scala programming paradigm.

Here is an example of class parameters

```
case class MyBus(width : Int) extends Bundle {
  val mySignal = UInt(width bits)
}
```

```
case class MyComponent(width : Int) extends Component {
  val bus = MyBus(width)
}
```

You can also use global variable defined in Scala objects (companion object pattern).

A *ScopeProperty* can also be used for configuration.

5.7.2 Optional hardware

So here there is more possibilities.

For optional signal :

```
case class MyComponent(flag : Boolean) extends Component {
  val mySignal = flag generate (Bool())
  // equivalent to "val mySignal = if (flag) Bool() else null"
}
```

The `generate` method is a mechanism to evaluate the expression that follows for an optional value. If the predicate is true, generate will evaluate the given expression and return the result, otherwise it returns null.

This may be used in cases to help parameterize the SpinalHDL hardware description using an elaboration-time conditional expression. Causing HDL constructs to be emitted or not-emitted in the resulting HDL. The generate method can be seen as SpinalHDL syntatic sugar reducing language clutter.

Project SpinalHDL code referencing `mySignal` would need to ensure it handles the possiblity of null gracefully. This is usually not a problem as those parts of the design can also be omitted dependant on the `flag` value. Thus the feature of parameterizing this component is demonstrated.

You can do the same in Bundle.

Note that you can also use scala Option.

If you want to disable the generation of a chunk of hardware :

```
case class MyComponent(flag : Boolean) extends Component {
  val myHardware = flag generate new Area {
    //optional hardware here
  }
}
```

You can also use scala for loops :

```
case class MyComponent(amount : Int) extends Component {
  val myHardware = for(i <- 0 until amount) yield new Area {
    // hardware here
  }
}
```

So, you can extends those scala usages at elaboration time as much as you want, including using the whole scala collections (List, Set, Map, ...) to build some data model and then converting them into hardware in a procedural way (ex iterating over those list elements).

CHAPTER

SIX

SEMANTIC

6.1 Assignments

There are multiple assignment operators:

Symbol	Description
:=	Standard assignment, equivalent to <= in VHDL/Verilog.
\=	Equivalent to := in VHDL and = in Verilog. The value is updated instantly in-place. Only works with combinational signals, does not work with registers.
<>	Automatic connection between 2 signals or two bundles of the same type. Direction is inferred by using signal direction (in/out). (Similar behavior to :=)

When muxing (for instance using `when`, see *When/Switch/Mux*.), the last valid standard assignment := wins. Else, assigning twice to the same assignee from the same scope results in an assignment overlap. SpinalHDL will assume this is a unintentional design error by default and halt elaboration with error. For special use-cases assignment overlap can be programatically permitted on a case by case basis. (see *Assignment overlap*).

```
val a, b, c = UInt(4 bits)
a := 0
b := a
//a := 1 // this would cause an `assignment overlap` error,
         // if manually overridden the assignment would take assignment priority
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 also supports Bundle assignment (convert all bit signals into a single bit-bus of suitable width of type Bits, to then use that wider form in an assignment expression). Bundle multiple signals together using `()` (Scala Tuple syntax) on both the left hand side and right hand side of an assignment expression.

```
val a, b, c = UInt(4 bits)
val d       = UInt(12 bits)
val e       = Bits(10 bits)
val f       = SInt(2  bits)
val g       = Bits()
```

(continues on next page)

(continued from previous page)

```
(a, b, c) := B(0, 12 bits)
(a, b, c) := d.asBits
(a, b, c) := (e, f).asBits           // both sides
g         := (a, b, c, e, f).asBits  // and on the right hand side
```

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
```

6.1.1 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, size inferred from x.
x := y.resize(newWidth)	Assign x with a resized copy of y `newWidth` bits wide.
x := y.resizeLeft(newWidth)	Assign x with a resized copy of y `newWidth` bits wide. Pads at the LSB if needed.

All resize methods may cause the resulting width to be wider or narrower than the original width of `y`. When widening occurs the extra bits are padded with zeros.

The inferred conversion with `x.resized` is based on the target width on the left hand side of the assignment expression being resolved and obeys the same semantics as `y.resize(someWidth)`. The expression `x := y.resized` is equivalent to `x := y.resize(x.getBitsWidth bits)`.

While the example code snippets show the use of an assignment statement, the resize family of methods can be chained like any ordinary Scala method.

There is one case where Spinal automatically resizes a value:

```
// U(3) creates an UInt of 2 bits, which doesn't match the left side (8 bits)
myUIntOf_8bits := U(3)
```

Because `U(3)` is a "weak" bit count inferred signal, SpinalHDL widens it automatically. This can be considered to be functionally equivalent to `U(3, 2 bits).resized` However rest reassured SpinalHDL will do the correct thing and continue to flag an error if the scenario would require narrowing. An error is reported if the literal required 9 bits (e.g. `U(0x100)`) when trying to assign into `myUIntOf_8bits`.

6.1.2 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.

6.1.3 CombInit

`CombInit` can be used to copy a signal and its current combinatorial assignments. The main use-case is to be able to overwrite the copied later, without impacting the original signal.

```
val a = UInt(8 bits)
a := 1

val b = a
when(sel) {
    b := 2
    // At this point, a and b are evaluated to 2: they reference the same signal
}

val c = UInt(8 bits)
c := 1

val d = CombInit(c)
// Here c and d are evaluated to 1
when(sel) {
    d := 2
    // At this point c === 1 and d === 2.
}
```

If we look at the resulting Verilog, `b` is not present. Since it is a copy of `a` by reference, these variables designate the same Verilog wire.

```
always @(*) begin
  a = 8'h01;
  if(sel) begin
    a = 8'h02;
  end
end

assign c = 8'h01;
always @(*) begin
  d = c;
  if(sel) begin
    d = 8'h02;
  end
end
```

`CombInit` is particularly helpful in helper functions to ensure that the returned value is not referencing an input.

```
// note that condition is an elaboration time constant
def invertedIf(b: Bits, condition: Boolean): Bits = if(condition) { ~b } else {␣
↪CombInit(b) }

val a2 = invertedIf(a1, c)

when(sel) {
   a2 := 0
}
```

Without `CombInit`, if `c` == false (but not if `c` == true), `a1` and `a2` reference the same signal and the zero assignment is also applied to `a1`. With `CombInit` we have a coherent behaviour whatever the `c` value.

6.2 When/Switch/Mux

6.2.1 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)
}
```

Warning: If the keyword `otherwise` is on the same line as the closing bracket `}` of the `when` condition, no dot is needed.

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

But if `.otherwise` is on another line, a dot is **required**:

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

6.2.2 WhenBuilder

Sometimes we need to generate some parameters for the when condition, and the original structure of when else is not very suitable. Therefore, we provide a 'whenBuilder' method to achieve this goal

```
import spinal.lib._

val conds = Bits(8 bits)
val result = UInt(8 bits)

val ctx = WhenBuilder()
ctx.when(conds(0)) {
  result := 0
}
ctx.when(conds(1)) {
  result := 1
}
if(true){
  ctx.when(conds(2)) {
    result := 2
  }
}
ctx.when(conds(3)) {
```

(continues on next page)

(continued from previous page)

```
  result := 3
}
```

Compared to the when/elsewhen/otherwise approach, it might be more convenient for parameterization. we can also use like this

```
for(i <- 5 to 7) ctx.when(conds(i)) {
  result := i
}

ctx.otherwise{
  result := 255
}

switch(addr) {
  for (i <- addressElements ) {
    is(i) {
      rdata :=  buffer(i)
    }
  }
}
```

This way, we can parameterize priority circuits similar to how we use 'foreach' inside 'switch()', and generate code in a more intuitive if-else format.

6.2.3 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
  }
}
```

`is` clauses can be factorized (logical OR) by separating them with a comma `is(value1, value2)`.

Example

```
switch(aluop) {
  is(ALUOp.add) {
    immediate := instruction.immI.signExtend
  }
  is(ALUOp.slt) {
    immediate := instruction.immI.signExtend
  }
  is(ALUOp.sltu) {
    immediate := instruction.immI.signExtend
  }
```

(continues on next page)

(continued from previous page)

```
  is(ALUOp.sll) {
    immediate := instruction.shamt
  }
  is(ALUOp.sra) {
    immediate := instruction.shamt
  }
}
```

is equivalent to

```
switch(aluop) {
  is(ALUOp.add, ALUOp.slt, ALUOp.sltu) {
      immediate := instruction.immI.signExtend
  }
  is(ALUOp.sll, ALUOp.sra) {
      immediate := instruction.shamt
  }
}
```

Additional options

By default, SpinalHDL will generate an "UNREACHABLE DEFAULT STATEMENT" error if a `switch` contains a `default` statement while all the possible logical values of the `switch` are already covered by the `is` statements. You can drop this error reporting by specifying `` switch(myValue, coverUnreachable = true) { ... }``.

```
switch(my2Bits, coverUnreachable = true) {
    is(0) { ... }
    is(1) { ... }
    is(2) { ... }
    is(3) { ... }
    default { ... } // This will parse and validate without error now
}
```

Note: This check is done on the logical values, not on the physical values. For instance, if you have a SpinalEnum(A,B,C) encoded in a one-hot manner, SpinalHDL will only care about the A,B,C values ("001" "010" "100"). Physical values as "000" "011" "101" "110" "111" will not be taken in account.

By default, SpinalHDL will generate a "DUPLICATED ELEMENTS IN SWITCH IS(...) STATEMENT" error if a given `is` statement provides multiple times the same value. For instance `is(42,42) { ... }` You can drop this error reporting by specifying `switch(myValue, strict = true){ ... }`. SpinalHDL will then take care of removing duplicated values.

```
switch(value, strict = false) {
    is(0) { ... }
    is(1,1,1,1,1) { ... } // This will be okay
    is(2) { ... }
}
```

6.2.4 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.

6.2.5 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
```

6.2.6 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)
)
```

`mux` checks that all possible values are covered to prevent generation of latches. If all possible values are covered, the default statement must not be added:

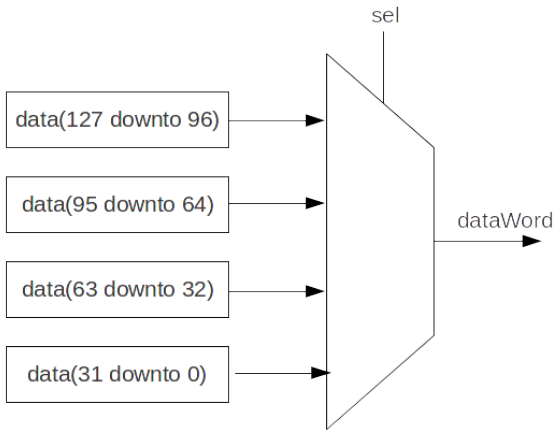
```
val bitwiseSelect = UInt(2 bits)
val bitwiseResult = bitwiseSelect.mux(
  0 -> (io.src0 & io.src1),
  1 -> (io.src0 | io.src1),
  2 -> (io.src0 ^ io.src1),
  3 -> (io.src0)
)
```

`muxList(...)` and `muxListDc(...)` are alternatives bitwise selectors that take a sequence of tuples or mappings as input.

`muxList` can be used as a direct replacement for `mux`, providing a easier to use interface in code that generates the cases. It has the same checking behavior as `mux` does, requiring full coverage and prohibiting listing a default if it is not needed.

`muxtListDc` can be used if the uncovered values are not important, they can be left unassigned by using `muxListDc`. This will add a default case if needed. This default case will generate X's during the simulation if ever encountered. `muxListDc(...)` is often a good alternative in generic code.

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)
```

Example for `muxListDc` selecting bits from a configurable width vector:

```
case class Example(width: Int = 3) extends Component {
  // 2 bit wide for default width
  val sel = UInt(log2Up(count) bit)
  val data = Bits(width*8 bit)
  // no need to cover missing case 3 for default width
  val dataByte = sel.muxListDc(for(i <- 0 until count) yield (i, data(index*8, 8␣
↪bit)))
}
```

6.3 Rules

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

6.3.1 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 an additional new rule for the left side signal/register.

6.3.2 Last valid assignment wins

If a combinational signal or register is assigned multiple times through the use of the SpinalHDL `:=` operator, the last assignment that may execute wins (and so gets to set the value as a result for this state).

It could be said that top to bottom evaluation occurs based on the state that exists at that time. If your upstream signal inputs are driven from registers and so have synchronous behaviour, then it could be said that at each clock cycle the assignments are re-evaluated based on the new state at the time.

Some reasons why an assignment statement may not get to execute in hardware this clock cycle, maybe due to it being wrapped in a `when(cond)` clause.

Another reason maybe that the SpinalHDL code never made it through elaboration because the feature was parameterized and disabled during HDL code-generation, see `paramIsFalse` use below.

As an example:

```
// Every clock cycle  evaluation starts here
val paramIsFalse = false
val x, y = Bool()           // Define two combinational signals
val result = UInt(8 bits)   // Define a combinational signal
```

(continues on next page)

(continued from previous page)

```
result := 1
when(x) {
  result := 2
  when(y) {
    result := 3
  }
}
if(paramIsFalse) {          // This assignment should win as it is last, but it was␣
↪never elaborated
  result := 4               //  into hardware due to the use of if() and it␣
↪evaluating to false at the time
}                           //  of elaboration.  The three := assignments above are␣
↪elaborated into hardware.
```

This will produce the following truth table:

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

6.3.3 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

when(inc) {
  counter := counter + 1
}
when(clear) {
  counter := 0    // If inc and clear are True, then this  assignment wins
}                 //  (last value assignment wins 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

SEVEN

SEQUENTIAL LOGIC

7.1 Registers

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.

7.1.1 Instantiation

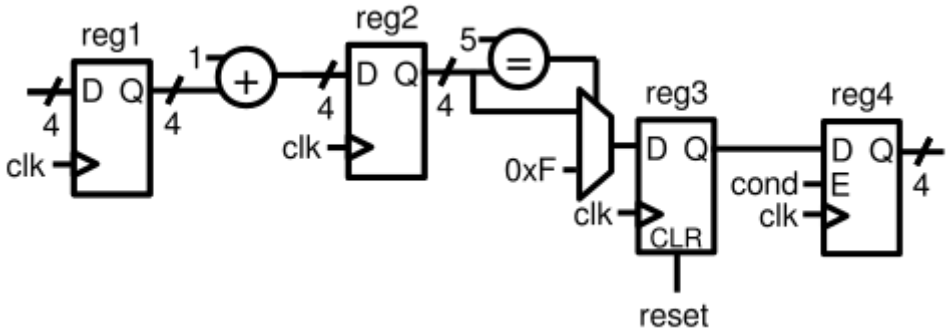
There are 4 ways to instantiate a register:

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 bits))

// Register that updates itself every cycle with a sample of reg1 incremented by 1
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
}

// 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. If the Reg is declared in a design and does not have suitable assignment and consumption it is likely to be pruned (removed from design) at some point by EDA flows after being deemed unnecessary.

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)
```

It is possible to have multiple options at the same time in other ways and so slightly more advanced compositions built on top of the basic understand of the above:

```
// UInt register of 6 bits (initialized with 42 when the reset occurs)
val reg1 = Reg(UInt(6 bits)) init(42)

// Register that samples reg1 each cycle (initialized with 0 when the reset occurs)
// using Scala named parameter argument format
val reg2 = RegNext(reg1, init=0)

// Register that has multiple features combined

// My register enable signal
val reg3Enable = Bool()
// UInt register of 6 bits (inferred from reg1 type)
//   assignment preconfigured to update from reg1
//   only updated when reg3Enable is set
//   initialized with 99 when the reset occurs
val reg3 = RegNextWhen(reg1, reg3Enable, U(99))
// when(reg3Enable) {
//   reg3 := reg1; // this expression is implied in the constructor use case
// }

when(cond2) {      // this is a valid assignment, will take priority when executed
   reg3 := U(0)    //  (due to last assignment wins rule), assignment does not require
```

(continues on next page)

(continued from previous page)

```
}                  //  reg3Enable condition, you would use `when(cond2 & reg3Enable)`␣
↪for that

// UInt register of 8 bits, initialized with 99 when the reset occurs
val reg4 = Reg(UInt(8 bits), U(99))
// My register enable signal
val reg4Enable = Bool()
// no implied assignments exist, you must use enable explicitly as necessary
when(reg4Enable) {
   reg4 := newValue
}
```

7.1.2 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 bits)) 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.
```

7.1.3 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 bits)) randBoot()
```

7.1.4 Register vectors

As for wires, it is possible to define a vector of registers with `Vec`.

```
val vecReg1 = Vec(Reg(UInt(8 bits)), 4)
val vecReg2 = Vec.fill(8)(Reg(Bool()))
```

Initialization can be done with the `init` method as usual, which can be combined with the `foreach` iteration on the registers.

```
val vecReg1 = Vec(Reg(UInt(8 bits)) init(0), 4)
val vecReg2 = Vec.fill(8)(Reg(Bool()))
vecReg2.foreach(_ init(False))
```

In case where the initialization must be deferred since the init value is not known, use a function as in the example below.

```
case class ShiftRegister[T <: Data](dataType: HardType[T], depth: Int, initFunc: T =>␣
↪Unit) extends Component {
   val io = new Bundle {
      val input  = in (dataType())
      val output = out(dataType())
   }

   val regs = Vec.fill(depth)(Reg(dataType()))
   regs.foreach(initFunc)

   for (i <- 1 to (depth-1)) {
         regs(i) := regs(i-1)
   }

   regs(0) := io.input
   io.output := regs(depth-1)
}

object SRConsumer {
   def initIdleFlow[T <: Data](flow: Flow[T]): Unit = {
      flow.valid init(False)
   }
}

class SRConsumer() extends Component {
   //...
   val sr = ShiftRegister(Flow(UInt(8 bits)), 4, SRConsumer.initIdleFlow[UInt])
}
```

7.1.5 Transforming a wire into a register

Sometimes it is useful to transform an existing wire into a register. For instance, when you are using a Bundle, if you want some outputs of the bundle to be registers, you might prefer to write `io.myBundle.PORT := newValue` without declaring registers with `val PORT = Reg(...)` and connecting their output to the port with `io.myBundle.PORT := PORT`. To do this, you just need to use `.setAsReg()` on the ports you want to control as registers:

```
val io = new Bundle {
   val apb = master(Apb3(apb3Config))
}

io.apb.PADDR.setAsReg()
io.apb.PWRITE.setAsReg() init(False)

when(someCondition) {
   io.apb.PWRITE := True
}
```

Notice in the code above that you can also specify an initialization value.

Note: The register is created in the clock domain of the wire, and does not depend on the place where `.setAsReg()` is used.

In the example above, the wire is defined in the `io` Bundle, in the same clock domain as the component. Even if

`io.apb.PADDR.setAsReg()` was written in a `ClockingArea` with a different clock domain, the register would use the clock domain of the component and not the one of the `ClockingArea`.

7.2 RAM/ROM Memory

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.

Note: Write mask width is flexible, and subdivide the memory word in as many slices of equal width as the width of the mask. For instance if you have a 32 bits memory word and provide a 4 bits mask then it will be a byte mask. If you provide a as many mask bits than you have word bits, then it is a bit mask.

Note: Manipulation of `Mem` is possible in simulation, see section *Load and Store of Memory in Simulation*.

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
)
```

7.2.1 Synchronous enable quirk

When enable signals are used in a block guarded by a conditional block like *when*, only the enable signal will be generated as the access condition: the *when* condition is ignored.

```
val rom = Mem(Bits(10 bits), 32)
when(cond){
  io.rdata := rom.readSync(io.addr, io.rdEna)
}
```

In the example above the condition *cond* will not be elaborated. Prefer to include the condition *cond* in the enable signal directly as below.

```
io.rdata := rom.readSync(io.addr, io.rdEna & cond)
```

7.2.2 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*.

7.2.3 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.

7.2.4 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`

Blackboxing can insert HDL attributes if `SpinalConfig#setDevice(Device)` has been configured for your device-vendor.

The resulting HDL attributes might look like:

```
(* ram_style = "distributed" *)
(* ramsyle = "no_rw_check" *)
```

SpinalHDL tries to support many common memory types provided by well known vendors and devices, however this is an ever moving landscape and project requirements can be very specific in this area.

If this is important to your design flow then check the output HDL for the expected attributes/generic insertion, while consulting your vendor's platform documentation.

HDL attributes can also be added manually using the *addAttribute() addAttribute* mechanism.

CHAPTER

EIGHT

DESIGN ERRORS

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.

8.1 Assignment overlap

8.1.1 Introduction

SpinalHDL will check that no signal assignment completely erases a previous one.

8.1.2 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
}
```

8.2 Clock crossing violation

8.2.1 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.

8.2.2 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*
- *setSynchronousWith 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
  regB := tmp
}
```

setSynchronousWith

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.setSynchronousWith(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) {
```

(continues on next page)

(continued from previous page)

```
    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 bits))
    val empty       = isEmpty(popPtrGray, pushPtrGray)
    ...
  }
}
```

8.3 Combinatorial loop

8.3.1 Introduction

SpinalHDL will check that there are no combinatorial loops in the design.

8.3.2 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
}
```

8.3.3 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.

8.4 Hierarchy violation

8.4.1 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.

8.4.2 Example

The following code:

```
class TopLevel extends Component {
  val io = new Bundle {
    // This is an 'in' signal of the current component 'Toplevel'
    val a = in UInt(8 bits)
  }
  val tmp = U"x42"
  io.a := tmp  // ERROR: attempting to assign to an input of current component
}
```

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  // now we are assigning to an output
}
```

8.5 IO bundle

8.5.1 Introduction

SpinalHDL will check that each `io` bundle contains only in/out/inout signals. Other kinds of signals are called directionless signals.

8.5.2 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)  // provide 'in' direction declaration
  }
}
```

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)
  }
  a.allowDirectionLessIo
}
```

8.6 Latch detected

8.6.1 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.

8.6.2 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
  }
}
```

8.6.3 Due to mux

Another reason for a latch being detected is often a non-exhaustive `mux`/`muxList` statement with a missing default:

```
val u1 = UInt(1 bit)
u1.mux(
  0 -> False,
  // case for 1 is missing
)
```

which can be fixed by adding the missing case (or a default case):

```
val u1 = UInt(1 bit)
u1.mux(
  0 -> False,
  default -> True
)
```

In e.g. width generic code it is often a better solution to use `muxListDc` as this will not generate an error for those cases were a default is not needed:

```
val u1 = UInt(1 bit)
// automatically adds default if needed
u1.muxListDc(Seq(0 -> True))
```

8.7 No driver on

8.7.1 Introduction

SpinalHDL will check that all combinational signals which have an impact on the design are assigned by something.

8.7.2 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
}
```

8.8 NullPointerException

8.8.1 Introduction

`NullPointerException` is a Scala runtime reported error which can happen when a variable is accessed before it has been initialized.

8.8.2 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`.

8.9 Out of Range Constant

8.9.1 Introduction

SpinalHDL checks that in comparisons with literals the literal is not wider than the value compared to.

8.9.2 Example

For example the following code:

```
val value = in UInt(2 bits)
val result = out(value < 42)
```

Will result in the following error:

```
OUT OF RANGE CONSTANT. Operator UInt < UInt
- Left  operand : (toplevel/value : in UInt[2 bits])
- Right operand : (U"101010" 6 bits)
 is checking a value against a out of range constant
```

8.9.3 Specifying exceptions

In some cases, because of the design parametrization, it can make sense to compare a value to a larger constant and get a statically known `True/False` result.

You have the option to specifically whitelist one instance of a comparison with an out of range constant.

```
val value = in UInt(2 bits)
val result = out((value < 42).allowOutOfRangeLiterals)
```

Alternatively, you can allow comparisons to out of range constants for the whole design.

```
SpinalConfig(allowOutOfRangeLiterals = true)
```

8.10 Register defined as component input

8.10.1 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.

8.10.2 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)
}
```

8.11 Scope violation

8.11.1 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.

8.11.2 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) {

  }
```

(continues on next page)

(continued from previous page)

```
  tmp := U"x42"
}
```

8.12 Spinal can't clone class

8.12.1 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`.

8.12.2 Example 1

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))
}
```

8.12.3 Example 2

The following code:

```
case class Xlen(val xlen: Int) {}

case class MemoryAddress()(implicit xlenConfig: Xlen) extends Bundle {
    val address = UInt(xlenConfig.xlen bits)
}

class DebugMemory(implicit config: Xlen) extends Component {
    val io = new Bundle {
        val inputAddress = in(MemoryAddress())
    }

    val someAddress = RegNext(io.inputAddress) // -> ERROR␣
↪*****************************
}
```

raises an exeption:

```
[error] *** Spinal can't clone class debug.MemoryAddress datatype
```

In this case, a solution is to override the clone function to propagate the implicit parameter.

```
case class MemoryAddress()(implicit xlenConfig: Xlen) extends Bundle {
  val address = UInt(xlenConfig.xlen bits)

  override def clone = MemoryAddress()
}
```

Note: We need to clone the hardware element, not the eventually assigned value in it.

Note: An alternative is to used *ScopeProperty*.

8.13 Unassigned register

8.13.1 Introduction

SpinalHDL will check that all registers which impact the design have been assigned somewhere.

8.13.2 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
  ***
  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
}
```

8.13.3 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
}
```

8.14 Unreachable is statement

8.14.1 Introduction

SpinalHDL will check to ensure that all `is` statements in a `switch` are reachable.

8.14.2 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 }
  }
}
```

8.15 Width mismatch

8.15.1 Introduction

SpinalHDL will check that operators and signals on the left and right side of assignments have the same widths.

8.15.2 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
}
```

8.15.3 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)
}
```

CHAPTER

NINE

OTHER LANGUAGE FEATURES

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.

9.1 Utils

9.1.1 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, from MSB to LSB, see *Cat*
`Cat(x: Iterable[Data])`	Bits	Conactenate arguments, from LSB to MSB, see *Cat*

Cat

As listed above, there are two version of `Cat`. Both versions concatenate the signals they contain, with a subtle difference:

- `Cat(x: Data*)` takes an arbitrary number of hardware signals as parameters. It mimics other HDLs and the leftmost parameter becomes the MSB of the resulting `Bits`, the rightmost the LSB side. Said differently: the input is concatenated in the order as written.
- `Cat(x: Iterable[Data])` which takes a single Scala iterable collection (Seq / Set / List / ...) containing hardware signals. This version places the first element of the list into the LSB, and the last into the MSB.

This seeming difference comes mostly from the convention that `Bits` are written from the hightest index to the lowest index, while Lists are written down starting from index 0 to the highest index. `Cat` places index 0 of both conventions at the LSB.

```
val bit0, bit1, bit2 = Bool()

val first = Cat(bit2, bit1, bit0)

// is equivalent to

val signals = List(bit0, bit1, bit2)
val second = Cat(signals)
```

9.1.2 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 provide new instance of a UInt with the same width as ``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.

```
// An example of a regular 'class' with 'override def clone()' function
class MyBundle(ppp : Int) extends Bundle {
   val a = UInt(ppp bits)
   override def clone = new MyBundle(ppp)
 }
 val x = new MyBundle(3)
 val typeDef = HardType(new MyBundle(3))
 val y = typeDef()
```

(continues on next page)

(continued from previous page)

```
cloneOf(x) // Need clone method, else it errors
cloneOf(y) // Is ok
```

9.1.3 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`.

9.1.4 Frequency and time

SpinalHDL has a dedicated syntax to define frequency and time values:

```
val frequency = 100 MHz       // infers type TimeNumber
val timeoutLimit = 3 ms       // infers type HertzNumber
val period = 100 us           // infers type TimeNumber

val periodCycles = frequency * period             // infers type BigDecimal
val timeoutCycles = frequency * timeoutLimit      // infers type BigDecimal
```

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 use scala `BigDecimal` to store numbers.

9.1.5 Binary prefix

SpinalHDL allows the definition of integer numbers using binary prefix notation according to IEC.

```
val memSize = 512 MiB       // infers type BigInt
val dpRamSize = 4 KiB       // infers type BigInt
```

The following binary prefix notations are available:

Binary Prefix	Value
Byte, Bytes	1
KiB	1024 == 1 << 10
MiB	1024^2 == 1 << 20
GiB	1024^3 == 1 << 30
TiB	1024^4 == 1 << 40
PiB	1024^5 == 1 << 50
EiB	1024^6 == 1 << 60
ZiB	1024^7 == 1 << 70
YiB	1024^8 == 1 << 80

Of course, BigInt can also be printed as a string in bytes unit. `BigInt(1024).byteUnit`.

```
val memSize = 512 MiB

println(memSize)
>> 536870912

println(memSize.byteUnit)
>> 512MiB

val dpRamSize = BigInt("123456789", 16)
```

(continues on next page)

(continued from previous page)

```
println(dpRamSize.byteUnit())
>> 4GiB+564MiB+345KiB+905Byte

println((32.MiB + 12.KiB + 223).byteUnit())
>> 32MiB+12KiB+223Byte

println((32.MiB + 12.KiB + 223).byteUnit(ceil = true))
>> 33~MiB
```

9.2 Stub

You can empty an Component Hierarchy as stub:

```
class SubSysModule extends Component {
   val io = new Bundle {
     val dx = slave(Stream(Bits(32 bits)))
     val dy = master(Stream(Bits(32 bits)))
   }
   io.dy <-< io.dx
}
class TopLevel extends Component {
   val dut = new SubSysModule().stub   //instance an SubSysModule as empty stub
}
```

It will generate the following Verilog code for example:

```
module SubSysModule (
  input               io_dx_valid,
  output              io_dx_ready,
  input      [31:0]   io_dx_payload,
  output              io_dy_valid,
  input               io_dy_ready,
  output     [31:0]   io_dy_payload,
  input               clk,
  input               reset
);


  assign io_dx_ready = 1'b0;
  assign io_dy_valid = 1'b0;
  assign io_dy_payload = 32'h0;

endmodule
```

You can also empty the top Component

```
SpinalVerilog(new Pinsec(500 MHz).stub)
```

What does *stub* do ?

- first walk all the components and find out clock, then keep clock
- then remove all children component
- then remove all assignment and logic we dont want
- tile 0 to output port

9.3 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
  )
}
```

9.4 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")
```

You can display the current simulation time using the REPORT_TIME object

```
report(L"miaou $REPORT_TIME")
```

will result in:

```
$display("NOTE miaou %t", $time);
```

9.5 ScopeProperty

A scope property is a thing which can store values locally to the current thread. Its API can be used to set/get that value, but also to apply modification to the value for a portion of the execution in a stack manner.

In other words it is a alternative to global variable, scala implicit, ThreadLocal.

- To compare with global variable, It allow to run multiple thread running the same code indepedently
- To compare with scala implicit, it is less intrusive in the code base
- To compare with ThreadLocal, it has some API to collect all ScopeProperty and restore them in the same state later on

```
object Xlen extends ScopeProperty[Int]

object ScopePropertyMiaou extends App {
  Xlen.set(1)
  println(Xlen.get) //1
  Xlen(2) {
    println(Xlen.get) //2
    Xlen(3) {
      println(Xlen.get) //3
      Xlen.set(4)
      println(Xlen.get) //4
    }
    println(Xlen.get) //2
  }
}
```

9.6 Analog and inout

9.6.1 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.

9.6.2 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()
}
```

9.6.3 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
  }
}
```

9.6.4 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;
```

9.6.5 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 bits)))
    val analog = inout(Analog(Bits(16 bits)))
  }
  io.tri.read := io.analog
  when(io.tri.writeEnable) { io.analog := io.tri.write }
}
```

9.7 VHDL and Verilog generation

9.7.1 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
```

9.7.2 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 bits)
    val resultA = out UInt(4 bits)
    val resultB = out UInt(4 bits)
  }

  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;
```

9.7.3 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;
```

CHAPTER

TEN

LIBRARIES

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

10.1 Utils

Some utils are also present in *spinal.core*

10.1.1 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`
CountLeadingZeroes(x : Bits)	UInt	Return the number of consecutive zero bits starting from the MSB
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.
PriorityMux (sel: Seq[Bool], in: Seq[T])	T	Return the first `in` element whose `sel` is `True`.
PriorityMux (in: Seq[(Bool, T)])	T	Return the first `in` element whose `sel` is `True`.

10.1.2 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*][*, init : T*]*)	Vec[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
History (that: T, range: Range *[*, when : Bool*][*, init : T*]*)	Vec[T]	Same as `History(that, length)` but return a Vec of size `range.length` where the first element is delayed by `range.low` and the last by `range.high`
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 instantiate a hardware counter.

Instantiation 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])`	Starts at zero and ends at `stateCount - 1`
`Counter(bitCount: BitCount[, inc : Bool])`	Starts at zero and ends at `(1 << bitCount) - 1`

A counter can be controlled by methods, and wires can be read:

```
val counter = Counter(2 to 9) // Creates a counter of 8 states (2 to 9)
// Methods
counter.clear()               // Resets the counter
counter.increment()           // Increments the counter
// Wires
counter.value                 // Current value
counter.valueNext             // Next value
counter.willOverflow          // True if the counter overflows this cycle
counter.willOverflowIfInc     // True if the counter would overflow this cycle if an␣
↪increment was done
// Cast
when(counter === 5){ ... }    // counter is implicitly casted to its current value
```

When a `Counter` overflows (reached end value), it restarts the next cycle to its start value.

Note: Currently, only up counter are supported.

`CounterFreeRun` builds an always running counter: `CounterFreeRun(stateCount: BigInt)`.

Timeout

The Timeout tool can be used to easily instantiate 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 instantiate 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 filtered 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 filtered value.

10.1.3 Special utilities

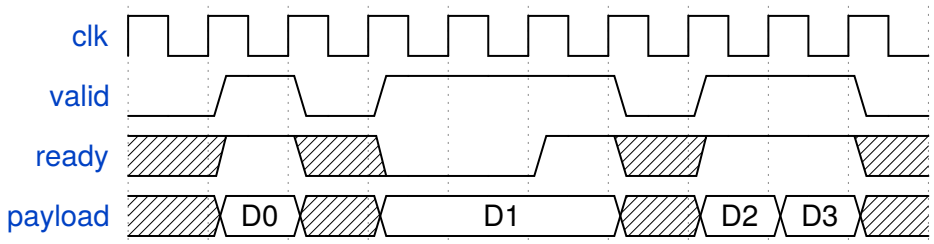
Syntax	Return	Description
LatencyAnalysis(paths : Node*)	Int	Return the shortest path, in terms of cycles, that travel through all nodes, from the first one to the last one

10.2 Stream

10.2.1 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.

10.2.2 Semantics

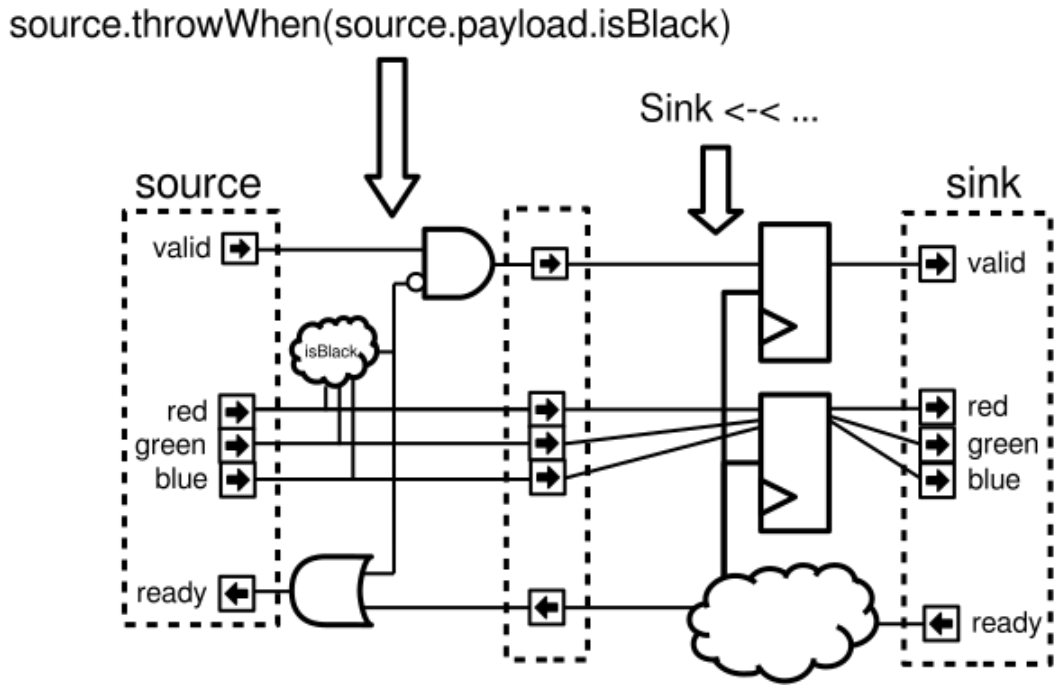
When manually reading/driving the signals of a Stream keep in mind that:

- 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.

10.2.3 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 bits)
  val green = UInt(channelWidth bits)
  val blue  = UInt(channelWidth bits)

  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)
```

10.2.4 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 instantiate 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
)
```

StreamWidthAdapter

This component adapts the width of the input stream to the output stream. When the width of the `outStream` payload is greater than the `inStream`, by combining the payloads of several input transactions into one; conversely, if the payload width of the `outStream` is less than the `inStream`, one input transaction will be split into several output transactions.

In the best case, the width of the payload of the `inStream` should be an integer multiple of the `outStream` as shown below.

```
val inStream = Stream(Bits(8 bits))
val outStream = Stream(Bits(16 bits))
val adapter = StreamWidthAdapter(inStream, outStream)
```

As in the example above, the two `inStream` transactions will be merged into one `outStream` transaction, and the payload of the first input transaction will be placed on the lower bits of the output payload by default.

If the expected order of input transaction payload placement is different from the default setting, here is an example.

```
val inStream = Stream(Bits(8 bits))
val outStream = Stream(Bits(16 bits))
val adapter = StreamWidthAdapter(inStream, outStream, order = SlicesOrder.HIGHER_
↪FIRST)
```

There is also a traditional parameter called `endianness`, which has the same effect as `ORDER`. The value of `endianness` is the same as `LOWER_FIRST` of `order` when it is `LITTLE`, and the same as `HIGHER_FIRST` when it is `BIG`. The `padding` parameter is an optional boolean value to determine whether the adapter accepts non-integer multiples of the input and output payload width.

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 utility takes multiple input streams and waits until all of them fire *valid* before letting all of them through by providing *ready*.

```
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)
```

StreamMux

A mux implementation for `Stream`. It takes a `select` signal and streams in `inputs`, and returns a `Stream` which is connected to one of the input streams specified by `select`. `StreamArbiter` is a facility works similar to this but is more powerful.

```
val inputStreams = Vec(Stream(Bits(8 bits)), portCount)
val select = UInt(log2Up(inputStreams.length) bits)
val outputStream = StreamMux(select, inputStreams)
```

Note: The `UInt` type of `select` signal could not be changed while output stream is stalled, or it might break the transaction on the fly. Use `Stream` typed `select` can generate a stream interface which only fire and change the routing when it is safe.

StreamDemux

A demux implementation for `Stream`. It takes a `input`, a `select` and a `portCount` and returns a `Vec(Stream)` where the output stream specified by `select` is connected to `input`, the other output streams are inactive. For safe transaction, refer the notes above.

```
val inputStream = Stream(Bits(8 bits))
val select = UInt(log2Up(portCount) bits)
val outputStreams = StreamDemux(inputStream, select, portCount)
```

StreamDispatcherSequencial

This util take its input stream and routes it to `outputCount` stream in a sequential order.

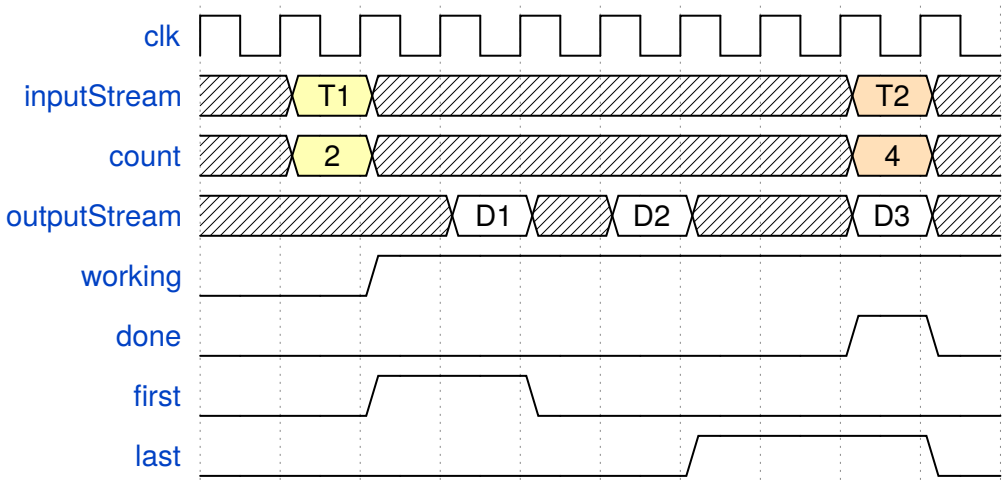
```
val inputStream = Stream(Bits(8 bits))
val dispatchedStreams = StreamDispatcherSequencial(
  input = inputStream,
  outputCount = 3
)
```

StreamTransactionExtender

This utility will take one input transfer and generate several output transfers, it provides the facility to repeat the payload value `count+1` times into output transfers. The `count` is captured and registered each time inputStream fires for an individual payload.

```
val inputStream = Stream(Bits(8 bits))
val outputStream = Stream(Bits(8 bits))
val count = UInt(3 bits)
val extender = StreamTransactionExtender(inputStream, outputStream, count) {
   // id, is the 0-based index of total output transfers so far in the current input␣
↪transaction.
   // last, is the last transfer indication, same as the last signal for extender.
   // the returned payload is allowed to be modified only based on id and last␣
↪signals, other translation should be done outside of this.
    (id, payload, last) => payload
}
```

This `extender` provides several status signals, such as `working`, `last`, `done` where `working` means there is one input transfer accepted and in-progress, `last` indicates the last output transfer is prepared and waiting to complete, `done` become valid represents the last output transfer is fireing and making the current input transaction process complete and ready to start another transaction.



Note: If only count for output stream is required then use `StreamTransactionCounter` instead.

10.2.5 Simulation support

For simulation master and slave implementations are available:

Class	Usage
StreamMonitor	Used for both master and slave sides, calls function with payload if Stream fires.
StreamDriver	Testbench master side, drives values by calling function to apply value (if available). Function must return if value was available. Supports random delays.
StreamReadyRandomizer	Randomizes `ready` for reception of data, testbench is the slave side.
ScoreboardInOrder	Often used to compare reference/dut data

```
import spinal.core._
import spinal.core.sim._
import spinal.lib._
import spinal.lib.sim.{StreamMonitor, StreamDriver, StreamReadyRandomizer,␣
↪ScoreboardInOrder}

object Example extends App {
  val dut = SimConfig.withWave.compile(StreamFifo(Bits(8 bits), 2))

  dut.doSim("simple test") { dut =>
    SimTimeout(10000)

    val scoreboard = ScoreboardInOrder[Int]()

    dut.io.flush #= false

    // drive random data and add pushed data to scoreboard
    StreamDriver(dut.io.push, dut.clockDomain) { payload =>
      payload.randomize()
      true
    }
    StreamMonitor(dut.io.push, dut.clockDomain) { payload =>
      scoreboard.pushRef(payload.toInt)
    }

    // randmize ready on the output and add popped data to scoreboard
    StreamReadyRandomizer(dut.io.pop, dut.clockDomain)
    StreamMonitor(dut.io.pop, dut.clockDomain) { payload =>
      scoreboard.pushDut(payload.toInt)
    }

    dut.clockDomain.forkStimulus(10)

    dut.clockDomain.waitActiveEdgeWhere(scoreboard.matches == 100)
  }
}
```

10.3 Flow

10.3.1 Specification

The Flow interface is a simple valid/payload protocol which means the slave can't halt the bus. It could be used 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

10.3.2 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.stage()	Equivalent to x.m2sPipe()	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	
x.setIdle()	Set the Flow in an Idle state: `valid` is `False` and don't care about `payload`.		
x.push(newPayload: T)	Assign a new valid payload to the Flow. `valid` is set to `True`.		

10.3.3 Code example

```
case class FlowExample() extends Component {
  val io = new Bundle {
    val request = slave(Flow(Bits(8 bit)))
    val answer = master(Flow(Bits(8 bit)))
  }
  val storage = Reg(Bits(8 bit))

  val fsm = new StateMachine {
    io.answer.setIdle()

    val idle: State = new State with EntryPoint {
      whenIsActive {
```

(continues on next page)

(continued from previous page)

```
        when(io.request.valid) {
          storage := io.request.payload
          goto(sendEcho)
        }
      }
    }

    val sendEcho: State = new State {
      whenIsActive {
        io.answer.push(storage)
        goto(idle)
      }
    }
  }

  // This StateMachine behaves equivalently to
  // io.answer <-< io.request
}
```

10.3.4 Simulation Support

Class	Usage
FlowMonitor	Used for both master and slave sides, calls function with payload if Flow transmits data.
FlowDriver	Testbench master side, drives values by calling function to apply value (if available). Function must return if value was available. Supports random delays.
ScoreboardInOrder	Often used to compare reference/dut data

```
package spinaldoc.libraries.flow

import spinal.core._
import spinal.core.sim._
import spinal.lib._
import spinal.lib.sim.{FlowDriver, FlowMonitor, ScoreboardInOrder}

import scala.language.postfixOps

case class SomeDUT() extends Component {
  val io = new Bundle {
    val input = slave(Flow(UInt(8 bit)))
    val output = master(Flow(UInt(8 bit)))
  }
  io.output <-< io.input
}

object Example extends App {
  val dut = SimConfig.withWave.compile(SomeDUT())

  dut.doSim("simple test") { dut =>
    SimTimeout(10000)

    val scoreboard = ScoreboardInOrder[Int]()
```

(continues on next page)

(continued from previous page)

```
    // drive random data at random intervals, and add inputted data to scoreboard
    FlowDriver(dut.io.input, dut.clockDomain) { payload =>
      payload.randomize()
      true
    }
    FlowMonitor(dut.io.input, dut.clockDomain) { payload =>
      scoreboard.pushRef(payload.toInt)
    }

    // add all data coming out of DUT to scoreboard
    FlowMonitor(dut.io.output, dut.clockDomain) { payload =>
      scoreboard.pushDut(payload.toInt)
    }

    dut.clockDomain.forkStimulus(10)
    dut.clockDomain.waitActiveEdgeWhere(scoreboard.matches == 100)
  }
}
```

10.4 Fragment

10.4.1 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)'

10.4.2 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 :

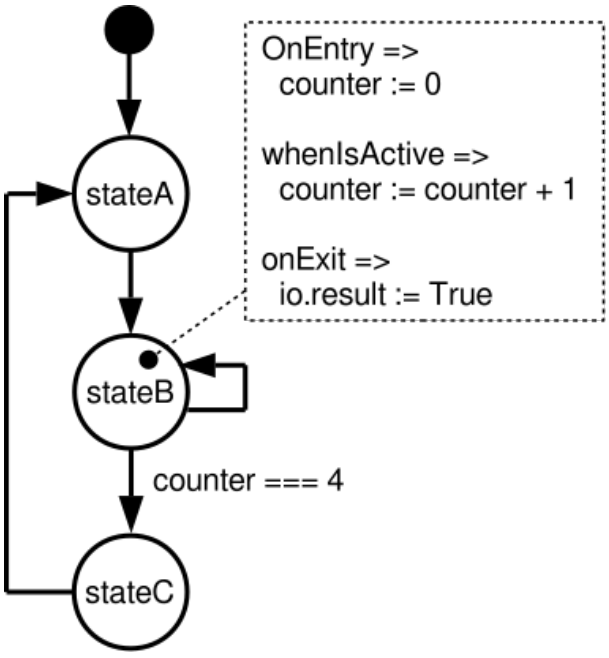
Syntax	Return	Description
x.insertHeader(header : T)	Stream[Fragment[T]]	Add the `header` to each packet on `x` and return the resulting bus

10.5 State machine

10.5.1 Introduction

In SpinalHDL you can define your state machine like in VHDL/Verilog, by using enumerations and switch/case statements. But in SpinalHDL you can also use a dedicated syntax.

The state machine below is implemented in the following examples:



Style A:

```
import spinal.lib.fsm._

class TopLevel extends Component {
  val io = new Bundle {
    val result = out Bool()
```

(continues on next page)

(continued from previous page)

```
  }

  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 {
  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
```

(continues on next page)

(continued from previous page)

```
      .whenIsActive(goto(stateA))
  }
}
```

10.5.2 StateMachine

`StateMachine` is the base class. It manages the logic of the FSM.

```
val myFsm = new StateMachine {
  // Definition of states
}
```

`StateMachine` also provides some accessors:

Name	Return	Description
`isActive(state)`	`Bool`	Returns `True` when the state machine is in the given state
`isEntering(state)`	`Bool`	Returns `True` when the state machine is entering the given state

Entry point

A state can be defined as the entry point of the state machine by extending the EntryPoint trait:

```
val stateA = new State with EntryPoint
```

Or by using `setEntry(state)`:

```
val stateA = new State
setEntry(stateA)
```

Transitions

- Transitions are represented by `goto(nextState)`, which schedules the state machine to be in `nextState` the next cycle.
- `exit()` schedules the state machine to be in the boot state the next cycle (or, in `StateFsm`, to exit the current nested state machine).

These two functions can be used inside state definitions (see below) or using `always { yourStatements }`, which always applies `yourStatements`, with a priority over states.

State encoding

By default the FSM state vector will be encoded using the native encoding of the language/tools the RTL is generated for (Verilog or VHDL). This default can be overriden by using the `setEncoding(...)` method which either takes a `SpinalEnumEncoding` or varargs of type `(State, BigInt)` for a custom encoding.

Listing 1: Using a `SpinalEnumEncoding`

```
val fsm = new StateMachine {
  setEncoding(binaryOneHot)

  ...
}
```

Listing 2: Using a custom encoding

```
val fsm = new StateMachine {
  val stateA = new State with EntryPoint
  val stateB = new State
  ...
  setEncoding((stateA -> 0x23), (stateB -> 0x22))
}
```

Warning: When using the `graySequential` enum encoding, no check is done to verify that the FSM transitions only produce single-bit changes in the state vector. The encoding is done according to the order of state definitions and the designer must ensure that only valid transitions are done if needed.

10.5.3 States

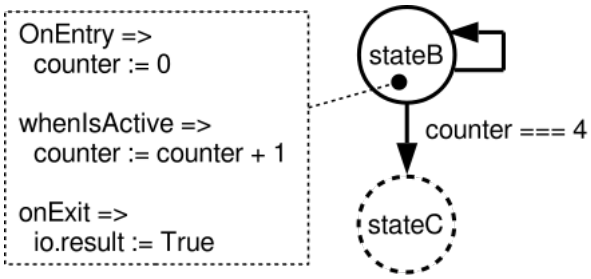
Multiple kinds of states can be used:

- `State` (the base one)
- `StateDelay`
- `StateFsm`
- `StateParallelFsm`

Each of them provides the following functions to define the logic associated to them:

Name	Description
`state.onEntry{ yourStatements }`	`yourStatements` is applied when the state machine is not in `state` and will be in `state` the next cycle
`state.onExit{ yourStatements }`	`yourStatements` is applied when the state machine is in `state` and will be in another state the next cycle
`state.whenIsActive{ yourStatements }`	`yourStatements` is applied when the state machine is in `state`
`state.whenIsNext{ yourStatements }`	`yourStatements` is executed when the state machine will be in `state` the next cycle (even if it is already in it)

`state.` is implicit in a `new State` block:




```
val stateB : State = new State {
  onEntry(counter := 0)
  whenIsActive {
    counter := counter + 1
    when(counter === 4) {
      goto(stateC)
    }
  }
  onExit(io.result := True)
}
```

StateDelay

`StateDelay` allows you to create a state which waits for a fixed number of cycles before executing statements in `whenCompleted {...}`. The preferred way to use it is:

```
val stateG : State = new StateDelay(cyclesCount=40) {
  whenCompleted {
    goto(stateH)
  }
}
```

It can also be written in one line:

```
val stateG : State = new StateDelay(40) { whenCompleted(goto(stateH)) }
```

StateFsm

`StateFsm` allows you to describe a state containing a nested state machine. When the nested state machine is done (exited), statements in `whenCompleted { ... }` are executed.

There is an example of StateFsm definition :

```
// internalFsm is a function defined below
val stateC = new StateFsm(fsm=internalFsm()) {
  whenCompleted {
    goto(stateD)
  }
}

def internalFsm() = new StateMachine {
  val counter = Reg(UInt(8 bits)) init(0)

  val stateA : State = new State with EntryPoint {
    whenIsActive {
      goto(stateB)
    }
  }

  val stateB : State = new State {
    onEntry (counter := 0)
    whenIsActive {
      when(counter === 4) {
        exit()
      }
      counter := counter + 1
    }
  }
}
```

In the example above, `exit()` makes the state machine jump to the boot state (a internal hidden state). This notifies `StateFsm` about the completion of the inner state machine.

StateParallelFsm

`StateParallelFsm` allows you to handle multiple nested state machines. When all nested state machine are done, statements in `whenCompleted { ... }` are executed.

Example:

```
val stateD = new StateParallelFsm (internalFsmA(), internalFsmB()) {
  whenCompleted {
    goto(stateE)
  }
}
```

Notes about the entry state

The way the entry state has been defined above makes it so that between the reset and the first clock sampling, the state machine is in a boot state. It is only after the first clock sampling that the defined entry state becomes active. This allows to properly enter the entry state (applying statements in `onEntry`), and allows nested state machines.

While it is usefull, it is also possible to bypass that feature and directly having a state machine booting into a user state.

To do so, use *makeInstantEntry()* instead of defining a `new State`. This function returns the boot state, active directly after reset.

Note: The `onEntry` of that state will only be called when it transitions from another state to this state and not during boot.

Note: During simulation, the boot state is always named `BOOT`.

Example:

```
// State sequance: IDLE, STATE_A, STATE_B, ...
val fsm = new StateMachine {
  // IDLE is named BOOT in simulation
  val IDLE = makeInstantEntry()
  val STATE_A, STATE_B, STATE_C = new State

  IDLE.whenIsActive(goto(STATE_A))
  STATE_A.whenIsActive(goto(STATE_B))
  STATE_B.whenIsActive(goto(STATE_C))
  STATE_C.whenIsActive(goto(STATE_B))
}
```

```
//  State sequance : BOOT, IDLE, STATE_A, STATE_B, ...
val fsm = new StateMachine {
  val IDLE, STATE_A, STATE_B, STATE_C = new State
  setEntry(IDLE)

  IDLE.whenIsActive(goto(STATE_A))
  STATE_A.whenIsActive(goto(STATE_B))
  STATE_B.whenIsActive(goto(STATE_C))
  STATE_C.whenIsActive(goto(STATE_B))
}
```

10.6 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

10.7 Bus Slave Factory

10.7.1 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*

10.7.2 Functionality

There are many implementations of the `BusSlaveFactory` tool : AHB3-lite, APB3, APB4, AvalonMM, AXI-lite 3, AXI4, BMB, Wishbone and PipelinedMemoryBus.
Each implementation of that tool take as an argument one instance of the corresponding bus and then offers the 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)		Instantiate 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

10.8 Fiber framework

Warning: This framework is not expected to be used for general RTL generation and targets large system design management and code generation. It is currently used as toplevel integration tool in SaxonSoC.

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.

Using this framework can complicate simple things but provide some strong features for complex cases :

- You can define things before even knowing all their requirements, ex : instantiating a interruption controller, before knowing how many interrupt signal lines you need
- Abstract/lazy/partial SoC architecture definition allowing the creation of SoC template for further specialisations
- Automatic requirement 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 allows the current task to announce that it will load `handle` with a value (used for scheduling)

10.8.1 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.

10.8.2 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.

10.9 BinarySystem

10.9.1 Specification

Here things have nothing to do with HDL, but they are very common in digital systems, In particular, the algorithm reference model is widely used. In addition, it is also used in build testbench.

Syntax	Description	Return
String.asHex	HexString to BigInt == BigInt(string, 16)	BigInt
String.asDec	Decimal String to BigInt == BigInt(string, 10)	BigInt
String.asOct	Octal String to BigInt == BigInt(string, 8)	BigInt
String.asBin	Binary String to BigInt == BigInt(string, 2)	BigInt

continues on next page

Table 1 – continued from previous page

Syntax	Description	Return
Byte|Int|Long|BigInt.hexString()	to HEX String	String
Byte|Int|Long|BigInt.octString()	to Oct String	String
Byte|Int|Long|BigInt.binString()	to Bin String	String
Byte|Int|Long|BigInt.hexString(bitSize)	fitSize align to bit Size, then to HEX String	String
Byte|Int|Long|BigInt.octString(bitSize)	fitSize align to bit Size, then to Oct String	String
Byte|Int|Long|BigInt.binString(bitSize)	fitSize align to bit Size, then to Bin String	String
Byte|Int|Long|BigInt.toBinInts()	to BinaryList	List[Int]
Byte|Int|Long|BigInt.toDecInts()	to DecimalList	List[Int]
Byte|Int|Long|BigInt.toOctInts()	to OctalList	List[Int]
Byte|Int|Long|BigInt.toBinInts(num)	to BinaryList, align to num size and fill 0	List[Int]
Byte|Int|Long|BigInt.toDecInts(num)	to DecimalList, align to num size and fill 0	List[Int]
Byte|Int|Long|BigInt.toOctInts(num)	to OctalList, align to num size and fill 0	List[Int]
"3F2A".hexToBinInts	Hex String to BinaryList	List[Int]
"3F2A".hexToBinIntsAlign	Hex String to BinaryList Align to times of 4	List[Int]
List(1,0,1,0,...).binIntsToHex	BinaryList to HexString	String
List(1,0,1,0,...).binIntsToOct	BinaryList to OctString	String
List(1,0,1,0,...).binIntsToHexAlignHigh	BinaryList size align to times of 4 (fill 0) then to HexString	String
List(1,0,1,0,...).binIntsToOctAlignHigh	BinaryList size align to times of 3 (fill 0) then to HexString	String
List(1,0,1,0,...).binIntsToInt	BinaryList (maxSize 32) to Int	Int
List(1,0,1,0,...).binIntsToLong	BinaryList (maxSIZE 64) to Long	Long
List(1,0,1,0,...).binIntsToBigInt	BinaryList (size no restrictions) to BigInt	BigInt
Int.toBigInt	32.toBigInt == BigInt(32)	BigInt
Long.toBigInt	3233113232L.toBigInt == BigInt(3233113232L)	BigInt
Byte.toBigInt	8.toByte.toBigInt == BigInt(8.toByte)	BigInt

10.9.2 String to Int/Long/BigInt

```
import spinal.core.lib._

$: "32FF190".asHex

$: "12384798999999".asDec

$: "123456777777700".asOct

$: "10100011100111111".asBin
```

10.9.3 Int/Long/BigInt to String

```
import spinal.core.lib._

$: "32FF190".asHex.hexString()
"32FF190"
$: "123456777777700".asOct.octString()
"123456777777700"
$: "10100011100111111".asBin.binString()
"10100011100111111"
$: 32323239988L.hexString()
7869d8034
$: 3239988L.octString()
14270064
$: 34.binString()
100010
```

10.9.4 Int/Long/BigInt to Binary-List

```
import spinal.core.lib._

$: 32.toBinInts
List(0, 0, 0, 0, 0, 1)
$: 1302309988L.toBinInts
List(0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1,␣
↪1, 0, 0, 1)
$: BigInt("100101110", 2).toBinInts
List(0, 1, 1, 1, 0, 1, 0, 0, 1)
$: BigInt("123456789abcdef0", 16).toBinInts
List(0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 0,␣
↪1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0,␣
↪0, 0, 1, 0, 0, 1)
$: BigInt("1234567", 8).toBinInts
List(1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1)
$: BigInt("123451118", 10).toBinInts
List(0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1)
```

align to a fixed width

```
import spinal.core.lib._

$: 39.toBinInts()
List(1, 1, 1, 0, 0, 1)
$: 39.toBinInts(8)    // align to 8 bit zero filled at MSB
List(1, 1, 1, 0, 0, 1, 0, 0)
```

10.9.5 Binary-List to Int/Long/BigInt

```
import spinal.core.lib._

$: List(1, 1, 1, 0, 0, 1).binIntsToInt
39
$: List(1, 1, 1, 0:, 0, 1).binIntsToLong
39
$: List(0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0,␣
↪1, 1, 0, 0, 1).binIntsToBigInt
1302309988
```

```
$: List(1, 1, 1, 0, 0, 1).binIntsToHex
27
$: List(1, 1, 1, 0, 0, 1).binIntsToHexAlignHigh
9c
$: List(1, 1, 1, 0, 0, 1).binIntsToOct
47
$: List(1, 1, 1, 0, 0, 1).binIntsToHexAlignHigh
47
```

10.9.6 BigInt enricher

```
$: 32.toBigInt
32
$: 3211323244L.toBigInt
3211323244
$: 8.toByte.toBigInt
8
```

10.10 RegIf

Register Interface Builder

- Automatic address, fields allocation and conflict detection
- 28 Register Access types (Covering the 25 types defined by the UVM standard)
- Automatic documentation generation

10.10.1 Automatic allocation

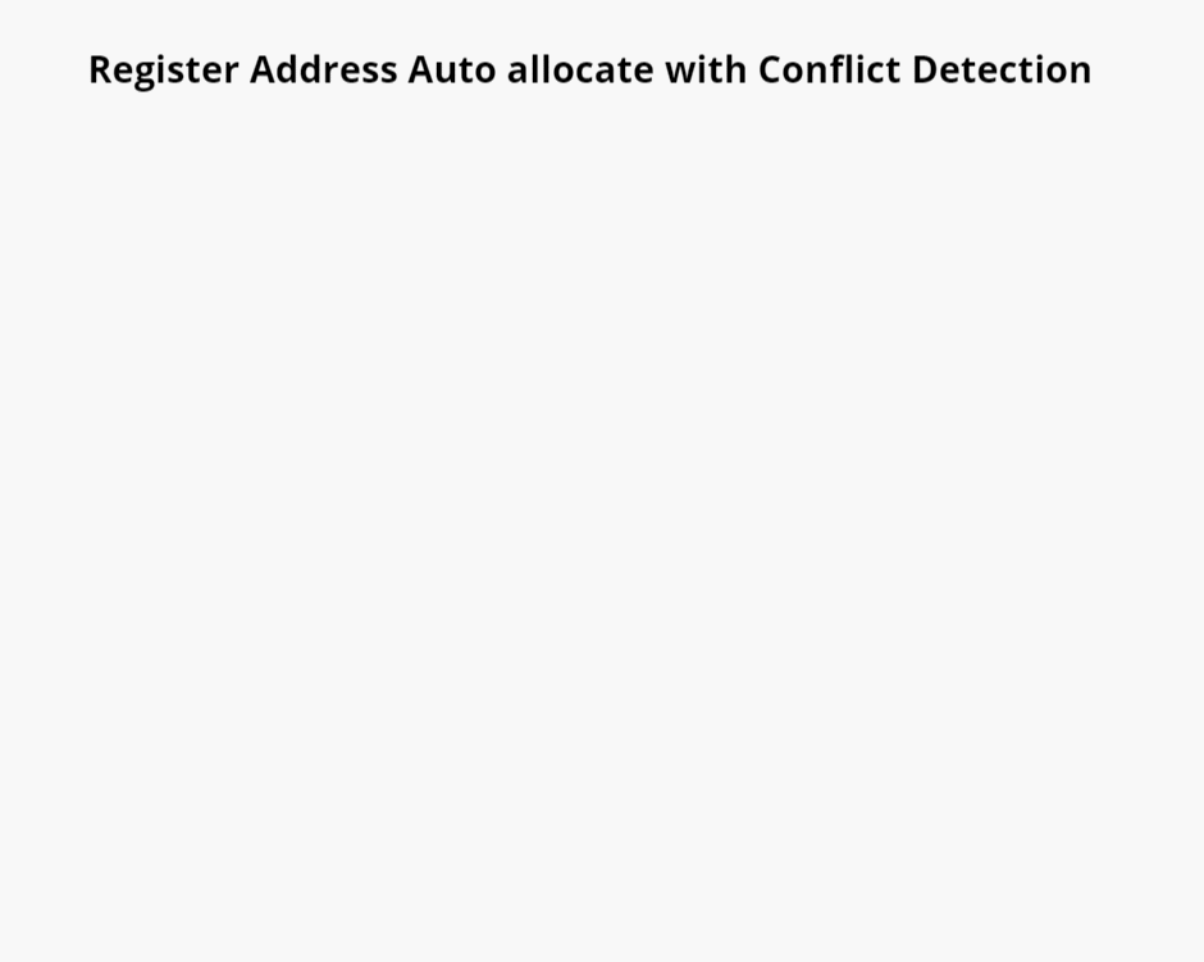
Automatic address allocation

```
class RegBankExample extends Component {
  val io = new Bundle {
    apb = slave(Apb3(Apb3Config(16,32)))
  }
  val busif = Apb3BusInterface(io.apb,(0x0000, 100 Byte))
  val M_REG0  = busif.newReg(doc="REG0")
  val M_REG1  = busif.newReg(doc="REG1")
  val M_REG2  = busif.newReg(doc="REG2")

  val M_REGn  = busif.newRegAt(address=0x40, doc="REGn")
```

(continues on next page)

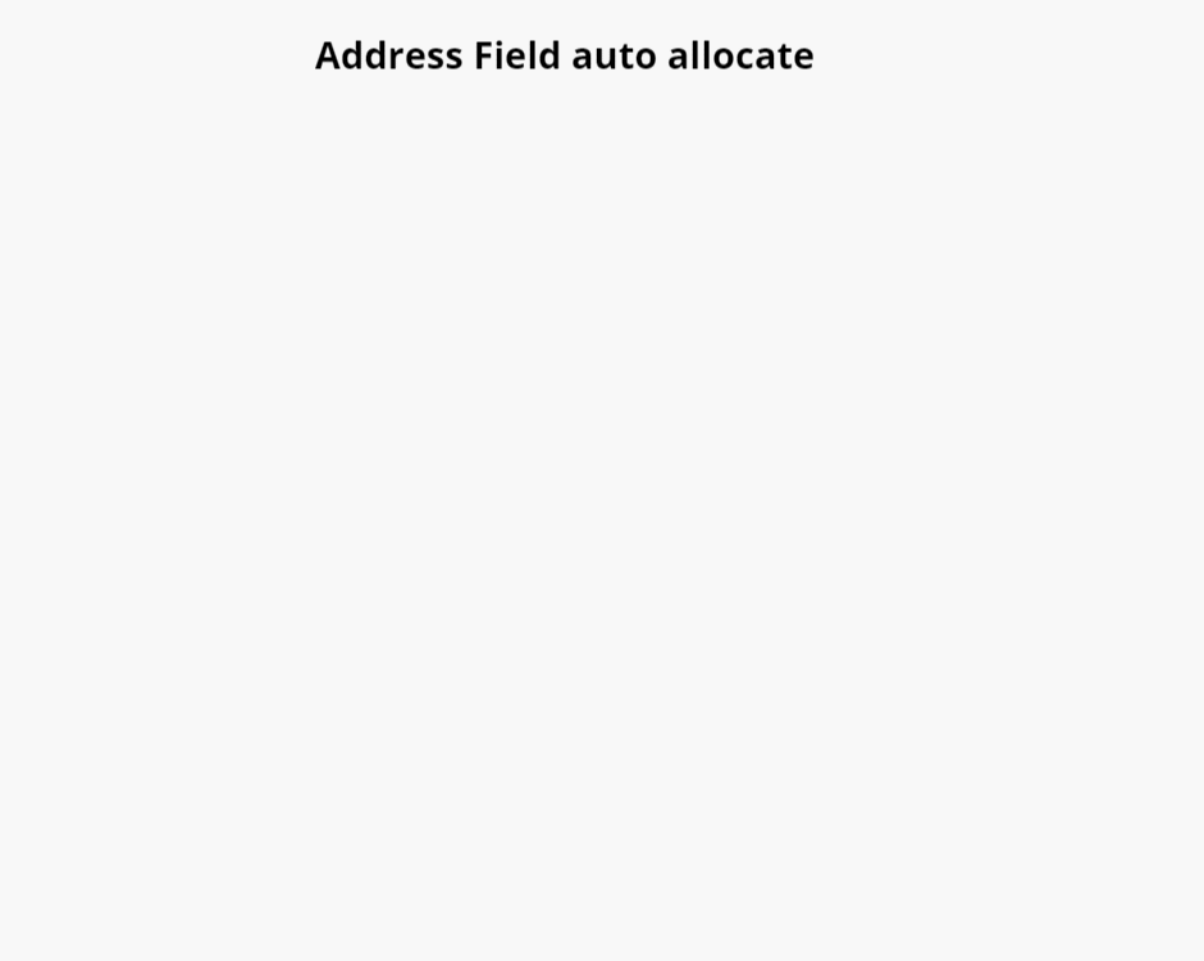
(continued from previous page)

```
  val M_REGn1 = busif.newReg(doc="REGn1")

  busif.accept(HtmlGenerator("regif", "AP"))
  // busif.accept(CHeaderGenerator("header", "AP"))
  // busif.accept(JsonGenerator("regif"))
  // busif.accept(RalfGenerator("regbank"))
  // busif.accept(SystemRdlGenerator("regif", "AP"))
}
```



Automatic fileds allocation

```
val M_REG0  = busif.newReg(doc="REG1")
val fd0 = M_REG0.field(Bits(2 bit), RW, doc= "fields 0")
M_REG0.reserved(5 bits)
val fd1 = M_REG0.field(Bits(3 bit), RW, doc= "fields 0")
val fd2 = M_REG0.field(Bits(3 bit), RW, doc= "fields 0")
//auto reserved 2 bits
val fd3 = M_REG0.fieldAt(pos=16, Bits(4 bit), doc= "fields 3")
//auto reserved 12 bits
```



conflict detection

```
val M_REG1  = busif.newReg(doc="REG1")
val r1fd0 = M_REG1.field(Bits(16 bits), RW, doc="fields 1")
val r1fd2 = M_REG1.field(Bits(18 bits), RW, doc="fields 1")
  ...
cause Exception
val M_REG1  = busif.newReg(doc="REG1")
val r1fd0 = M_REG1.field(Bits(16 bits), RW, doc="fields 1")
val r1fd2 = M_REG1.fieldAt(pos=10, Bits(2 bits), RW, doc="fields 1")
  ...
cause Exception
```

10.10.2 28 Access Types

Most of these come from UVM specification

AccessType	Description	From
RO	w: no effect, r: no effect	UVM
RW	w: as-is, r: no effect	UVM
RC	w: no effect, r: clears all bits	UVM
RS	w: no effect, r: sets all bits	UVM
WRC	w: as-is, r: clears all bits	UVM
WRS	w: as-is, r: sets all bits	UVM

continues on next page

Table 2 – continued from previous page

AccessType	Description	From
WC	w: clears all bits, r: no effect	UVM
WS	w: sets all bits, r: no effect	UVM
WSRC	w: sets all bits, r: clears all bits	UVM
WCRS	w: clears all bits, r: sets all bits	UVM
W1C	w: 1/0 clears/no effect on matching bit, r: no effect	UVM
W1S	w: 1/0 sets/no effect on matching bit, r: no effect	UVM
W1T	w: 1/0 toggles/no effect on matching bit, r: no effect	UVM
W0C	w: 1/0 no effect on/clears matching bit, r: no effect	UVM
W0S	w: 1/0 no effect on/sets matching bit, r: no effect	UVM
W0T	w: 1/0 no effect on/toggles matching bit, r: no effect	UVM
W1SRC	w: 1/0 sets/no effect on matching bit, r: clears all bits	UVM
W1CRS	w: 1/0 clears/no effect on matching bit, r: sets all bits	UVM
W0SRC	w: 1/0 no effect on/sets matching bit, r: clears all bits	UVM
W0CRS	w: 1/0 no effect on/clears matching bit, r: sets all bits	UVM
WO	w: as-is, r: error	UVM
WOC	w: clears all bits, r: error	UVM
WOS	w: sets all bits, r: error	UVM
W1	w: first one after hard reset is as-is, other w have no effects, r: no effect	UVM
WO1	w: first one after hard reset is as-is, other w have no effects, r: error	UVM
NA	w: reserved, r: reserved	New
W1P	w: 1/0 pulse/no effect on matching bit, r: no effect	New
W0P	w: 0/1 pulse/no effect on matching bit, r: no effect	New
HSRW	w: Hardware Set, SoftWare RW	New
RWHS	w: SoftWare RW, Hardware Set	New
ROV	w: ReadOnly Value, used for hardware version	New
CSTM	w: user custom Type, used for document	New

10.10.3 Automatic documentation generation

Document Type

Document	Usage	Status
HTML	`busif.accept(HtmlGenerator("regif", title = "XXX register file"))`	Y
CHeader	`busif.accept(CHeaderGenerator("header", "AP"))`	Y
JSON	`busif.accept(JsonGenerator("regif"))`	Y
RALF(UVM)	`busif.accept(RalfGenerator("header"))`	Y
SystemRDL	`busif.accept(SystemRdlGenerator("regif", "addrmap_name", Some("name"), Some("desc")))`	Y
Latex(pdf)		N
docx		N

HTML auto-doc is now complete, Example source Code:

generated HTML document:

文件 /github/SpinalHDL/tmp/regif.html

TurboRegBank Interface Document

AddressOffset	RegName	Description	Width	Section	FieldName	R/W	Reset value	Field-Description
0x0	M_TURBO_EARLY_QUIT	Turbo Hardware-mode register 1	64	[63:3]	--	NA	0x0	Reserved
				[2:1]	early_count	RW	0x2	CRC validate early quit enable
				[0]	early_quit	RW	0x0	CRC validate early quit enable
0x8	M_TURBO_THETA	Turbo Hardware-mode register 2	64	[63:5]	--	NA	0x0	Reserved
				[4]	bib_order	RW	0x1	bib in Byte, defalut. 0:[76543210] 1:[01234567]
				[3:0]	theta	RW	0x6	Back-Weight, UQ(4,3), default 0.75
0x10	M_TURBO_START	Turbo Start register	64	[63:1]	--	NA	0x0	Reserved
				[0]	turbo_start	W1P	0x0	turbo start pulse
0x18	M_TURBO_MOD	Turbo Mode	64	[63:1]	--	NA	0x0	Reserved
				[0]	umts_on	RW	0x1	1: umts mode 0: emtc mode
0x20	M_TURBO_CRC_POLY	Turbo CRC Poly	64	[63:25]	--	NA	0x0	Reserved
				[24:1]	crc_poly	RW	0x864cfb	(D24+D23+D18+D17+D14+D11+D10+D7+D6+D5+D4+D3+D+1)
				[0]	crc_mode	RW	0x1	0: CRC24; 1: CRC16
0x28	M_TURBO_K	Turbo block size	64	[63:12]	--	NA	0x0	Reserved
				[11:0]	K	RW	0x20	decode block size max: 4032
0x30	M_TURBO_F1F2	Turbo Interleave Parameter	64	[63:25]	--	NA	0x0	Reserved
				[24:16]	f2	RW	0x2f	turbo interleave parameter f2
				[15:9]	--	NA	0x0	Reserved
				[8:0]	f1	RW	0x2f	turbo interleave parameter f1
0x38	M_TURBO_MAX_ITER	Turbo Max Iter Times	64	[63:6]	--	NA	0x0	Reserved
				[5:0]	max_iter	RW	0x0	Max iter times 1~63 avaliable
0x40	M_TURBO_FILL_NUM	Turbo block-head fill number	64	[63:6]	--	NA	0x0	Reserved
				[5:0]	fill_num	RW	0x0	0~63 avaliable, Head fill Number
0x48	M_TURBO_3G_INTER_PV	Turbo UMTS Interleave Parameter P,V	64	[63:21]	--	NA	0x0	Reserved
				[20:16]	v	RW	0x0	Primitive root v
				[15:9]	--	NA	0x0	Reserved
				[8:0]	p	RW	0x0	parameter of prime
0x50	M_TURBO_3G_INTER_CRP	Turbo UMTS Interleave Parameter C,R,p	64	[63:17]	--	NA	0x0	Reserved
				[16:8]	C	RW	0x7	interlave Max Column Number
				[7:5]	--	NA	0x0	Reserved
				[4]	KeqRxC	RW	0x0	1:K=R*C else 0
				[3:2]	CPtype	RW	0x0	CP relation 0: C=P-1 1: C=p 2: C=p+1
				[1:0]	Rtype	RW	0x0	inter Row number 0:5,1:10,2:20,3:20other
0x58	M_TURBO_3G_INTER_FILL	Turbo UMTS Interleave Fill number	64	[63:18]	--	NA	0x0	Reserved
				[17:16]	fillRow	RW	0x0	interlave fill Row number, 0~2 avaliable n+1 row
				[15:9]	--	NA	0x0	Reserved
				[8:0]	fillPos	RW	0x0	interlave start Column of fill Number

Powered By SpinalHDL

10.10.4 Special Access Usage

CASE1: RO usage

RO is different from other types. It does not create registers and requires an external signal to drive it, Attention, please don't forget to drive it.

```
val io = new Bundle {
  val cnt = in UInt(8 bit)
}

val counter = M_REG0.field(UInt(8 bit), RO, 0, "counter")
counter :=  io.cnt
```

```
val xxstate = M_REG0.field(UInt(8 bit), RO, 0, "xx-ctrl state").asInput
```

```
val overflow = M_REG0.field(Bits(32 bit), RO, 0, "xx-ip paramete")
val ovfreg = Reg(32 bit)
overflow := ovfreg
```

```
val inc    = in Bool()
val couter = M_REG0.field(UInt(8 bit), RO, 0, "counter")
val cnt = Counter(100,  inc)
couter := cnt
```

CASE2: ROV usage

ASIC design often requires some solidified version information. Unlike RO, it is not expected to generate wire signals

old way:

```
val version = M_REG0.field(Bits(32 bit), RO, 0, "xx-device version")
version := BigInt("F000A801", 16)
```

new way:

```
M_REG0.field(Bits(32 bit), ROV, BigInt("F000A801", 16), "xx-device version")(Symbol(
↪"Version"))
```

CASE3: HSRW/RWHS hardware set type In some cases, such registers are not only configured by software, but also set by hardware signals

```
val io = new Bundle {
  val xxx_set = in Bool()
  val xxx_set_val = in Bits(32 bit)
}

val reg0 = M_REG0.fieldHSRW(io.xxx_set, io.xxx_set_val, 0, "xx-device version")  //
↪0x0000
val reg1 = M_REG1.fieldRWHS(io.xxx_set, io.xxx_set_val, 0, "xx-device version")  //
↪0x0004
```

```
always @(posedge clk or negedge rstn)
  if(!rstn) begin
     reg0  <= '0;
     reg0  <= '0;
  end else begin
```

(continues on next page)

(continued from previous page)

```
    if(hit_0x0000) begin
       reg0 <= wdata ;
    end
    if(io.xxx_set) begin      //HW have High priority than SW
       reg0 <= io.xxx_set_val ;
    end

    if(io.xxx_set) begin
       reg1 <= io.xxx_set_val ;
    end
    if(hit_0x0004) begin      //SW have High priority than HW
       reg1 <= wdata ;
    end
  end
```

CASE4: `CSTM` Although SpinalHDL includes 25 register types and 6 extension types, there are still various demands for private register types in practical application. Therefore, we reserve CSTM types for scalability. CSTM is only used to generate software interfaces, and does not generate actual circuits

```
val reg = Reg(Bits(16 bit)) init 0
REG.registerAtOnlyReadLogic(0, reg, CSTM("BMRW"), resetValue = 0, "custom field")

when(busif.dowrite) {
   reg :=  reg & ~busif.writeData(31 downto 16) |  busif.writeData(15 downto 0) &␣
↪busif.writeData(31 downto 16)
}
```

CASE5: `parasiteField`

This is used for software to share the same register on multiple address instead of generating multiple register entities

example1: clock gate software enable

```
val M_CG_ENS_SET = busif.newReg(doc="Clock Gate Enables")  //0x0000
val M_CG_ENS_CLR = busif.newReg(doc="Clock Gate Enables")  //0x0004
val M_CG_ENS_RO  = busif.newReg(doc="Clock Gate Enables")  //0x0008

val xx_sys_cg_en = M_CG_ENS_SET.field(Bits(4 bit), W1S, 0, "clock gate enalbes, write␣
↪1 set" )
                   M_CG_ENS_CLR.parasiteField(xx_sys_cg_en, W1C, 0, "clock gate␣
↪enalbes, write 1 clear" )
                   M_CG_ENS_RO.parasiteField(xx_sys_cg_en, RO, 0, "clock gate enables,
↪ read only"
```

example2: interrupt raw reg with foce interface for software

```
val RAW    = this.newRegAt(offset,"Interrupt Raw status Register\n set when event \n␣
↪clear raw when write 1")
val FORCE  = this.newReg("Interrupt Force  Register\n for SW debug use \n write 1 set␣
↪raw")

val raw    = RAW.field(Bool(), AccessType.W1C,    resetValue = 0, doc = s"raw,␣
↪default 0" )
             FORCE.parasiteField(raw, AccessType.W1S,   resetValue = 0, doc = s"force,
↪ write 1 set, debug use" )
```

CASE6: `SpinalEnum`

When the field type is SpinalEnum, the resetValue specifies the index of the enum elements.

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

val raw = M_REG2.field(UartCtrlTxState(), AccessType.RW, resetValue = 2, doc="state")
// raw will be init to sData
```

10.10.5 Byte Mask

withStrb

10.10.6 Typical Example

Batch create REG-Address and fields register

```
import spinal.lib.bus.regif._

class RegBank extends Component {
  val io = new Bundle {
    val apb = slave(Apb3(Apb3Config(16, 32)))
    val stats = in Vec(Bits(16 bit), 10)
    val IQ  = out Vec(Bits(16 bit), 10)
  }
  val busif = Apb3BusInterface(io.apb, (0x000, 100 Byte), regPre = "AP")

  (0 to 9).map { i =>
    //here use setName give REG uniq name for Docs usage
    val REG = busif.newReg(doc = s"Register${i}").setName(s"REG${i}")
    val real = REG.field(SInt(8 bit), AccessType.RW, 0, "Complex real")
    val imag = REG.field(SInt(8 bit), AccessType.RW, 0, "Complex imag")
    val stat = REG.field(Bits(16 bit), AccessType.RO, 0, "Accelerator status")
    io.IQ(i)( 7 downto 0) := real.asBits
    io.IQ(i)(15 downto 8) := imag.asBits
    stat := io.stats(i)
  }

  def genDocs() = {
    busif.accept(CHeaderGenerator("regbank", "AP"))
    busif.accept(HtmlGenerator("regbank", "Interupt Example"))
    busif.accept(JsonGenerator("regbank"))
    busif.accept(RalfGenerator("regbank"))
    busif.accept(SystemRdlGenerator("regbank", "AP"))
  }

  this.genDocs()
}

SpinalVerilog(new RegBank())
```

10.10.7 Interrupt Factory

Manual writing interruption

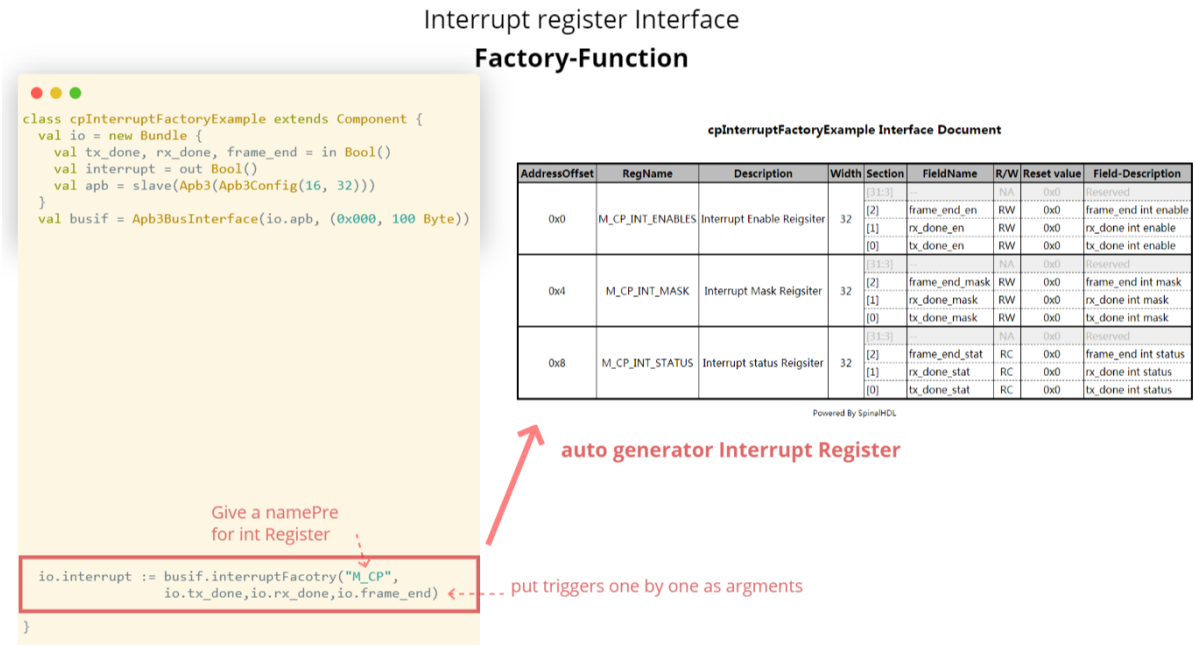
```
class cpInterruptExample extends Component {
   val io = new Bundle {
     val tx_done, rx_done, frame_end = in Bool()
     val interrupt = out Bool()
     val apb = slave(Apb3(Apb3Config(16, 32)))
   }
   val busif = Apb3BusInterface(io.apb, (0x000, 100 Byte), regPre = "AP")
   val M_CP_INT_RAW    = busif.newReg(doc="cp int raw register")
   val tx_int_raw      = M_CP_INT_RAW.field(Bool(), W1C, doc="tx interrupt enable␣
↪register")
   val rx_int_raw      = M_CP_INT_RAW.field(Bool(), W1C, doc="rx interrupt enable␣
↪register")
   val frame_int_raw   = M_CP_INT_RAW.field(Bool(), W1C, doc="frame interrupt enable␣
↪register")

   val M_CP_INT_FORCE = busif.newReg(doc="cp int force register\n for debug use")
   val tx_int_force     = M_CP_INT_FORCE.field(Bool(), RW, doc="tx interrupt enable␣
↪register")
   val rx_int_force     = M_CP_INT_FORCE.field(Bool(), RW, doc="rx interrupt enable␣
↪register")
   val frame_int_force  = M_CP_INT_FORCE.field(Bool(), RW, doc="frame interrupt␣
↪enable register")

   val M_CP_INT_MASK    = busif.newReg(doc="cp int mask register")
   val tx_int_mask      = M_CP_INT_MASK.field(Bool(), RW, doc="tx interrupt mask␣
↪register")
   val rx_int_mask      = M_CP_INT_MASK.field(Bool(), RW, doc="rx interrupt mask␣
↪register")
   val frame_int_mask   = M_CP_INT_MASK.field(Bool(), RW, doc="frame interrupt mask␣
↪register")

   val M_CP_INT_STATUS   = busif.newReg(doc="cp int state register")
   val tx_int_status      = M_CP_INT_STATUS.field(Bool(), RO, doc="tx interrupt state␣
↪register")
   val rx_int_status      = M_CP_INT_STATUS.field(Bool(), RO, doc="rx interrupt state␣
↪register")
   val frame_int_status   = M_CP_INT_STATUS.field(Bool(), RO, doc="frame interrupt␣
↪state register")

   rx_int_raw.setWhen(io.rx_done)
   tx_int_raw.setWhen(io.tx_done)
   frame_int_raw.setWhen(io.frame_end)

   rx_int_status := (rx_int_raw || rx_int_force) && (!rx_int_mask)
   tx_int_status := (tx_int_raw || rx_int_force) && (!rx_int_mask)
   frame_int_status := (frame_int_raw || frame_int_force) && (!frame_int_mask)

   io.interrupt := rx_int_status || tx_int_status || frame_int_status

}
```

this is a very tedious and repetitive work, a better way is to use the "factory" paradigm to auto-generate the documentation for each signal.

now the InterruptFactory can do that.

Easy Way create interruption:

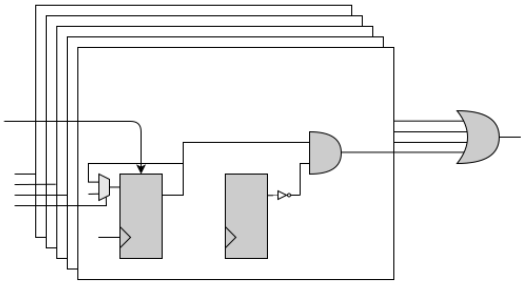
```
class EasyInterrupt extends Component {
  val io = new Bundle {
    val apb = slave(Apb3(Apb3Config(16,32)))
    val a, b, c, d, e = in Bool()
  }

  val busif = BusInterface(io.apb,(0x000,1 KiB), 0, regPre = "AP")

  busif.interruptFactory("T", io.a, io.b, io.c, io.d, io.e)

  busif.accept(CHeaderGenerator("intrreg","AP"))
  busif.accept(HtmlGenerator("intrreg", "Interupt Example"))
  busif.accept(JsonGenerator("intrreg"))
  busif.accept(RalfGenerator("intrreg"))
  busif.accept(SystemRdlGenerator("intrreg", "AP"))
}
```



AddressOffset	RegName	Description	Width	Section	FieldName	R/W	Reset value	Field-Description
0x0	M_CP_INT_ENABLES	Interrupt Enable Reigsiter	32	[31:3]		NA	0x0	Reserved
				[2]	frame_end_en	RW	0x0	frame_end int enable
				[1]	rx_done_en	RW	0x0	rx_done int enable
				[0]	tx_done_en	RW	0x0	tx_done int enable
0x4	M_CP_INT_MASK	Interrupt Mask Reigsiter	32	[31:3]		NA	0x0	Reserved
				[2]	frame_end_mask	RW	0x0	frame_end int mask
				[1]	rx_done_mask	RW	0x0	rx_done int mask
				[0]	tx_done_mask	RW	0x0	tx_done int mask
0x8	M_CP_INT_STATUS	Interrupt status Reigsiter	32	[31:3]		NA	0x0	Reserved
				[2]	frame_end_stat	RC	0x0	frame_end int status
				[1]	rx_done_stat	RC	0x0	rx_done int status
				[0]	tx_done_stat	RC	0x0	tx_done int status

IP level interrupt Factory

Register	AccessType	Description
RAW	W1C	int raw register, set by int event, clear when bus write 1
FORCE	RW	int force register, for SW debug use
MASK	RW	int mask register, 1: off; 0: open; defualt 1 int off
STATUS	RO	int status, Read Only, `status = raw && ! mask`

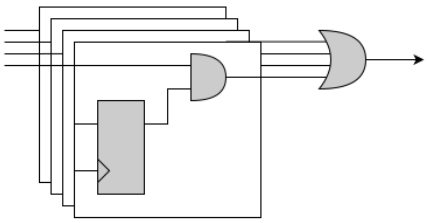


SpinalUsage:

```
busif.interruptFactory("T", io.a, io.b, io.c, io.d, io.e)
```

SYS level interrupt merge

Register	AccessType	Description
MASK	RW	int mask register, 1: off; 0: open; defualt 1 int off
STATUS	RO	int status, RO, `status = int_level && ! mask`



SpinalUsage:

```
busif.interruptLevelFactory("T", sys_int0, sys_int1)
```

Spinal Factory

BusInterface method	Description
`InterruptFactory(regNamePre: String, triggers: Bool*)`	create RAW/FORCE/MASK/STATUS for pulse event
`InterruptFactoryNoForce(regNamePre: String, triggers: Bool*)`	create RAW/MASK/STATUS for pulse event
`InterruptLevelFactory(regNamePre: String, triggers: Bool*)`	create MASK/STATUS for level_int merge
`InterruptFactoryAt(addrOffset: Int, regNamePre: String, triggers: Bool*)`	create RAW/FORCE/MASK/STATUS for pulse event at addrOffset
`InterruptFactoryNoForceAt(addrOffset: Int, regNamePre: String, triggers: Bool*)`	create RAW/MASK/STATUS for pulse event at addrOffset
`InterruptFactoryAt(addrOffset: Int, regNamePre: String, triggers: Bool*)`	create MASK/STATUS for level_int merge at addrOffset
`interrupt_W1SCmask_FactoryAt(addrOffset: BigInt, regNamePre: String, triggers: Bool*)`	creat RAW/FORCE/MASK(SET/CLR)/STATUS for pulse event at addrOffset
`interruptLevel_W1SCmask_FactoryAt(addrOffset: BigInt, regNamePre: String, levels: Bool*)`	creat RAW/FORCE/MASK(SET/CLR)/STATUS for leveel event at addrOffset

Example

```
class RegFileIntrExample extends Component {
   val io = new Bundle {
     val apb = slave(Apb3(Apb3Config(16,32)))
     val int_pulse0, int_pulse1, int_pulse2, int_pulse3 = in Bool()
     val int_level0, int_level1, int_level2 = in Bool()
     val sys_int = out Bool()
     val gpio_int = out Bool()
   }

   val busif = BusInterface(io.apb,  (0x000,1 KiB), 0, regPre = "AP")
   io.sys_int  := busif.interruptFactory("SYS",io.int_pulse0, io.int_pulse1, io.int_
↪pulse2, io.int_pulse3)
   io.gpio_int := busif.interruptLevelFactory("GPIO",io.int_level0, io.int_level1, io.
↪int_level2, io.sys_int)

   def genDoc() = {
     busif.accept(CHeaderGenerator("intrreg","Intr"))
     busif.accept(HtmlGenerator("intrreg", "Interupt Example"))
     busif.accept(JsonGenerator("intrreg"))
     busif.accept(RalfGenerator("intrreg"))
     busif.accept(SystemRdlGenerator("intrreg", "Intr"))
     this
   }

   this.genDoc()
 }
```

Interupt Example register interface

AddressOffset	RegName	Description	Width	Section	FieldName	R/W	Reset value	Field-Description
0x0	SYS_INT_RAW	Interrupt Raw status Register set when event clear when write 1	32	[31:4]	--	NA	28'b0	Reserved
				[3]	int_pulse3_raw	W1C	1'b0	raw, default 0
				[2]	int_pulse2_raw	W1C	1'b0	raw, default 0
				[1]	int_pulse1_raw	W1C	1'b0	raw, default 0
				[0]	int_pulse0_raw	W1C	1'b0	raw, default 0
0x4	SYS_INT_FORCE	Interrupt Force Register for SW debug use	32	[31:4]	--	NA	28'b0	Reserved
				[3]	int_pulse3_force	RW	1'b0	force, default 0
				[2]	int_pulse2_force	RW	1'b0	force, default 0
				[1]	int_pulse1_force	RW	1'b0	force, default 0
				[0]	int_pulse0_force	RW	1'b0	force, default 0
0x8	SYS_INT_MASK	Interrupt Mask Register 1: int off 0: int open default 1, int off	32	[31:4]	--	NA	28'b0	Reserved
				[3]	int_pulse3_mask	RW	1'h1	mask, default 1, int off
				[2]	int_pulse2_mask	RW	1'h1	mask, default 1, int off
				[1]	int_pulse1_mask	RW	1'h1	mask, default 1, int off
				[0]	int_pulse0_mask	RW	1'h1	mask, default 1, int off
0xc	SYS_INT_STATUS	Interrupt status Register status = (raw || force) && (!mask)	32	[31:4]	--	NA	28'b0	Reserved
				[3]	int_pulse3_status	RO	1'b0	stauts default 0
				[2]	int_pulse2_status	RO	1'b0	stauts default 0
				[1]	int_pulse1_status	RO	1'b0	stauts default 0
				[0]	int_pulse0_status	RO	1'b0	stauts default 0
0x10	GPIO_INT_MASK	Interrupt Mask Register 1: int off 0: int open default 1, int off	32	[31:4]	--	NA	28'b0	Reserved
				[3]	sys_int_mask	RW	1'h1	mask
				[2]	int_level2_mask	RW	1'h1	mask
				[1]	int_level1_mask	RW	1'h1	mask
				[0]	int_level0_mask	RW	1'h1	mask
0x14	GPIO_INT_STATUS	Interrupt status Register status = int_level && (!mask)	32	[31:4]	--	NA	28'b0	Reserved
				[3]	sys_int_status	RO	1'b0	stauts
				[2]	int_level2_status	RO	1'b0	stauts
				[1]	int_level1_status	RO	1'b0	stauts
				[0]	int_level0_status	RO	1'b0	stauts

Powered by SpinalHDL

Wed Apr 13 01:16:53 CST 2022

10.10.8 DefaultReadValue

When the software reads a reserved address, the current policy is to return normally, readerror=0. In order to facilitate software debugging, the read back value can be configured, which is 0 by default

```
busif.setReservedAddressReadValue(0x0000EF00)
```

```
default: begin
   busif_rdata  <= 32'h0000EF00 ;
   busif_rderr  <= 1'b0         ;
end
```

10.10.9 Developers Area

You can add your document Type by extending the *BusIfVistor* Trait

`case class Latex(fileName :  String) extends BusIfVisitor{ ... }`

BusIfVistor give access BusIf.RegInsts to do what you want

```
// lib/src/main/scala/lib/bus/regif/BusIfVistor.scala

trait BusIfVisitor {
  def begin(busDataWidth : Int) : Unit
  def visit(descr : FifoDescr)  : Unit
  def visit(descr : RegDescr)   : Unit
  def end()                     : Unit
}
```

10.11 Bus

10.11.1 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()
  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)
```

(continues on next page)

(continued from previous page)

```
  }
}
```

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`.

10.11.2 Apb3

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 bits)
  val PSEL       = Bits(config.selWidth bits)
  val PENABLE    = Bool()
  val PREADY     = Bool()
  val PWRITE     = Bool()
  val PWDATA     = Bits(config.dataWidth bits)
  val PRDATA     = Bits(config.dataWidth bits)
  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

10.11.3 Axi4

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))
```

(continues on next page)

(continued from previous page)

```
  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

10.11.4 AvalonMM

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 on the intel website.

```
case class AvalonMMConfig( addressWidth : Int,
                           dataWidth : Int,
                           burstCountWidth : Int,
                           useByteEnable : Boolean,
                           useDebugAccess : Boolean,
                           useRead : Boolean,
                           useWrite : Boolean,
                           useResponse : Boolean,
                           useLock : Boolean,
                           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)
```

10.11.5 Tilelink

Configuration and instanciation

There is a short example to define two non coherent tilelink bus instance and connect them:

```
import spinal.lib.bus.tilelink
val param = tilelink.BusParameter.simple(
  addressWidth = 32,
  dataWidth    = 64,
  sizeBytes    = 64,
  sourceWidth  = 4
)
val busA, busB = tilelink.Bus(param)
busA << busB
```

Here is the same as above, but with coherency channels

```
import spinal.lib.bus.tilelink
val param = tilelink.BusParameter(
  addressWidth = 32,
  dataWidth = 64,
  sizeBytes = 64,
  sourceWidth = 4,
  sinkWidth = 0,
  withBCE = false,
  withDataA = true,
  withDataB = false,
  withDataC = false,
  withDataD = true,
  node = null
)
```

(continues on next page)

(continued from previous page)

```
val busA, busB = tilelink.Bus(param)
busA << busB
```

Those above where for the hardware instanciation, the thing is that it is the simple / easy part. When things goes into SoC / memory coherency, you kind of need an additional layer to negociate / propagate parameters all around. That's what tilelink.fabric.Node is about.

10.11.6 tilelink.fabric.Node

tilelink.fabric.Node is an additional layer over the regular tilelink hardware instanciation which handle negociation and parameters propagation at a SoC level.

It is mostly based on the Fiber API, which allows to create elaboration time fibers (user-space threads), allowing to schedule future parameter propagation / negociation and hardware elaboration.

A Node can be created in 3 ways :

- tilelink.fabric.Node.down() : To create a node which can connect downward (toward slaves), so it would be used in a CPU / DMA / bridges agents
- tilelink.fabric.Node() : To create an intermediate nodes
- tilelink.fabric.Node.up() : To create a node which can connect upward (toward masters), so it would be used in peripherals / memories / bridges agents

Nodes mostly have the following attributes :

- bus : Handle[tilelink.Bus]; the hardware instance of the bus
- m2s.proposed : Handle[tilelink.M2sSupport]; The set of features which is proposed by the upward connections
- m2s.supported : Handle[tilelink.M2sSupport] : The set of feature supported by the downward connections
- m2s.parameter : Handle[tilelink.M2sParameter] : The final bus parameter

You can note that they all are Handles. Handle is a way in SpinalHDL to have share a value between fibers. If a fiber read a Handle while this one has no value yet, it will block the execution of that fiber until another fiber provide a value to the Handle.

There is also a set of attribues like m2s, but reversed (named s2m) which specify the parameters for the transactions initiated by the slave side of the interconnect (ex memory coherency).

There is two talks which where introducing the tilelink.fabric.Node. Those talk may not exactly follow the actual syntax, they are still follow the concepts :

- Introduction : https://youtu.be/hVi9xOGuuek
- In depth : https://peertube.f-si.org/videos/watch/bcf49c84-d21d-4571-a73e-96d7eb89e907

Example Toplevel

Here is an example of a simple fictive SoC toplevel :

```
val cpu = new CpuFiber()

val ram = new RamFiber()
ram.up at(0x10000, 0x200) of cpu.down // map the ram at [0x10000-0x101FF], the ram␣
↪will infer its own size from it

val gpio = new GpioFiber()
gpio.up at 0x20000 of cpu.down // map the gpio at [0x20000-0x20FFF], its range of 4KB␣
↪being fixed internally
```

You can also define intermediate nodes in the interconnect as following :

```
val cpu = new CpuFiber()

val ram = new RamFiber()
ram.up at(0x10000, 0x200) of cpu.down

// Create a peripherals namespace to keep things clean
val peripherals = new Area{
  // Create a intermediate node in the interconnect
  val access = tilelink.fabric.Node()
  access at 0x20000 of cpu.down

  val gpioA = new GpioFiber()
  gpioA.up at 0x0000 of access

  val gpioB = new GpioFiber()
  gpioB.up at 0x1000 of access
}
```

Example GpioFiber

GpioFiber is a simple tilelink peripheral which can read / drive a 32 bits tristate array.

```
import spinal.lib._
import spinal.lib.bus.tilelink
import spinal.core.fiber.Fiber
class GpioFiber extends Area {
  // Define a node facing upward (toward masters only)
  val up = tilelink.fabric.Node.up()

  // Define a elaboration thread to specify the "up" parameters and generate the␣
↪hardware
  val fiber = Fiber build new Area {
    // Here we first define what our up node support. m2s mean master to slave␣
↪requests
    up.m2s.supported load tilelink.M2sSupport(
      addressWidth = 12,
      dataWidth = 32,
      // Transfers define which kind of memory transactions our up node will support.
      // Here it only support 4 bytes get/putfull
      transfers = tilelink.M2sTransfers(
        get = tilelink.SizeRange(4),
        putFull = tilelink.SizeRange(4)
      )
    )
    // s2m mean slave to master requests, those are only use for memory coherency␣
↪purpose
    // So here we specify we do not need any
    up.s2m.none()

    // Then we can finally generate some hardware
    // Starting by defining a 32 bits TriStateArray (Array meaning that each pin has␣
↪its own writeEnable bit
    val pins = master(TriStateArray(32 bits))

    // tilelink.SlaveFactory is a utility allowing to easily generate the logic␣
```

(continues on next page)

(continued from previous page)

```
↪required
    // to control some hardware from a tilelink bus.
    val factory = new tilelink.SlaveFactory(up.bus, allowBurst = false)

    // Use the SlaveFactory API to generate some hardware to read / drive the pins
    val writeEnableReg = factory.drive(pins.writeEnable, 0x0) init (0)
    val writeReg = factory.drive(pins.write, 0x4) init(0)
    factory.read(pins.read, 0x8)
  }
}
```

Example RamFiber

RamFiber is the integration layer of a regular tilelink Ram component.

```
import spinal.lib.bus.tilelink
import spinal.core.fiber.Fiber
class RamFiber() extends Area {
  val up = tilelink.fabric.Node.up()

  val thread = Fiber build new Area {
    // Here the supported parameters are function of what the master would like us to␣
↪idealy support.
    // The tilelink.Ram support all addressWidth / dataWidth / burst length / get /␣
↪put accesses
    // but doesn't support atomic / coherency. So we take what is proposed to use and␣
↪restrict it to
    // all sorts of get / put request
    up.m2s.supported load up.m2s.proposed.intersect(M2sTransfers.allGetPut)
    up.s2m.none()

    // Here we infer how many bytes our ram need to be, by looking at the memory␣
↪mapping of the connected masters
    val bytes = up.ups.map(e => e.mapping.value.highestBound - e.mapping.value.
↪lowerBound + 1).max.toInt

    // Then we finaly generate the regular hardware
    val logic = new tilelink.Ram(up.bus.p.node, bytes)
    logic.io.up << up.bus
  }
}
```

Example CpuFiber

CpuFiber is an fictive example of a master integration.

```
import spinal.lib.bus.tilelink
import spinal.core.fiber.Fiber

class CpuFiber extends Area {
  // Define a node facing downward (toward slaves only)
  val down = tilelink.fabric.Node.down()

  val fiber = Fiber build new Area {
```

(continues on next page)

(continued from previous page)

```
    // Here we force the bus parameters to a specific configurations
    down.m2s forceParameters tilelink.M2sParameters(
      addressWidth = 32,
      dataWidth = 64,
      // We define the traffic of each master using this node. (one master => one␣
↪M2sAgent)
      // In our case, there is only the CpuFiber.
      masters = List(
        tilelink.M2sAgent(
          name = CpuFiber.this, // Reference to the original agent.
          // A agent can use multiple sets of source ID for different purposes
          // Here we define the usage of every sets of source ID
          // In our case, let's say we use ID [0-3] to emit get/putFull requests
          mapping = List(
            tilelink.M2sSource(
              id = SizeMapping(0, 4),
              emits = M2sTransfers(
                get = tilelink.SizeRange(1, 64), //Meaning the get access can be any␣
↪power of 2 size in [1, 64]
                putFull = tilelink.SizeRange(1, 64)
              )
            )
          )
        )
      )
    )

    // Lets say the CPU doesn't support any slave initiated requests (memory␣
↪coherency)
    down.s2m.supported load tilelink.S2mSupport.none()

    // Then we can generate some hardware (nothing usefull in this example)
    down.bus.a.setIdle()
    down.bus.d.ready := True
  }
}
```

One particularity of Tilelink, is that it assumes a master will not emit requests to a unmapped memory space. To allow a master to identify what memory access it is allowed to do, you can use the spinal.lib.system.tag.MemoryConnection.getMemoryTransfers tool as following :

```
val mappings = spinal.lib.system.tag.MemoryConnection.getMemoryTransfers(down)
// Here we just print the values out in stdout, but instead you can generate some␣
↪hardware from it.
for(mapping <- mappings){
  println(s"- ${mapping.where} -> ${mapping.transfers}")
}
```

If you run this in the Cpu's fiber, in the following soc :

```
val cpu = new CpuFiber()

val ram = new RamFiber()
ram.up at(0x10000, 0x200) of cpu.down

// Create a peripherals namespace to keep things clean
val peripherals = new Area{
```

(continues on next page)

(continued from previous page)

```
  // Create a intermediate node in the interconnect
  val access = tilelink.fabric.Node()
  access at 0x20000 of cpu.down

  val gpioA = new GpioFiber()
  gpioA.up at 0x0000 of access

  val gpioB = new GpioFiber()
  gpioB.up at 0x1000 of access
}
```

You will get :

```
- toplevel/ram_up mapped=SM(0x10000, 0x200) through=List(OT(0x10000))  -> GF
- toplevel/peripherals_gpioA_up mapped=SM(0x20000, 0x1000) through=List(OT(0x20000),␣
↪OT(0x0))  -> GF
- toplevel/peripherals_gpioB_up mapped=SM(0x21000, 0x1000) through=List(OT(0x20000),␣
↪OT(0x1000))  -> GF
```

- "through=" specify the chain of address transformations done to reach the target.
- "SM" means SizeMapping(address, size)
- "OT" means OffsetTransformer(offset)

Note that you can also add PMA (Physical Memory Attributes) to nodes and retreives them via this getMemoryTransfers utilities.

The currently defined PMA are :

```
object MAIN           extends PMA
object IO             extends PMA
object CACHABLE       extends PMA // an intermediate agent may have cached a copy of␣
↪the region for you
object TRACEABLE      extends PMA // the region may have been cached by another master,
↪ but coherence is being provided
object UNCACHABLE     extends PMA // the region has not been cached yet, but should be␣
↪cached when possible
object IDEMPOTENT     extends PMA // reads return most recently put content, but␣
↪content should not be cached
object EXECUTABLE     extends PMA // Allows an agent to fetch code from this region
object VOLATILE       extends PMA // content may change without a write
object WRITE_EFFECTS  extends PMA // writes produce side effects and so must not be␣
↪combined/delayed
object READ_EFFECTS   extends PMA // reads produce side effects and so must not be␣
↪issued speculatively
```

The getMemoryTransfers utility rely on a dedicated SpinalTag :

```
trait MemoryConnection extends SpinalTag {
  def up : Nameable with SpinalTagReady // Side toward the masters of the system
  def down : Nameable with SpinalTagReady // Side toward the slaves of the system
  def mapping : AddressMapping //Specify the memory mapping of the slave from the␣
↪master address (before transformers are applied)
  def transformers : List[AddressTransformer]  //List of alteration done to the␣
↪address on this connection (ex offset, interleaving, ...)
  def sToM(downs : MemoryTransfers, args : MappedNode) : MemoryTransfers = downs //
↪Convert the slave MemoryTransfers capabilities into the master ones
}
```

That SpinalTag can be used applied to both ends of a given memory bus connection to keep this connection discoverable at elaboration time, creating a graph of MemoryConnection. One good thing about it is that is is bus agnostic, meaning it isn't tilelink specific.

Example WidthAdapter

The width adapter is a simple example of bridge.

```
class WidthAdapterFiber() extends Area{
  val up = Node.up()
  val down = Node.down()

  // Populate the MemoryConnection graph
  new MemoryConnection {
    override def up = up
    override def down = down
    override def transformers = Nil
    override def mapping = SizeMapping(0, BigInt(1) << WidthAdapterFiber.this.up.m2s.
↪parameters.addressWidth)
    populate()
  }

  // Fiber in which we will negociate the data width parameters and generate the␣
↪hardware
  val logic = Fiber build new Area{
    // First, we propagate downward the parameter proposal, hopping that the downward␣
↪side will agree
    down.m2s.proposed.load(up.m2s.proposed)

    // Second, we will propagate upward what is actualy supported, but will take care␣
↪of any dataWidth missmatch
    up.m2s.supported load down.m2s.supported.copy(
      dataWidth = up.m2s.proposed.dataWidth
    )

    // Third, we propagate downward the final bus parameter, but will take care of␣
↪any dataWidth missmatch
    down.m2s.parameters load up.m2s.parameters.copy(
      dataWidth = down.m2s.supported.dataWidth
    )

    // No alteration on s2m parameters
    up.s2m.from(down.s2m)

    // Finaly, we generate the hardware
    val bridge = new tilelink.WidthAdapter(up.bus.p, down.bus.p)
    bridge.io.up << up.bus
    bridge.io.down >> down.bus
  }
}
```

10.12 Com

10.12.1 SPI XDR

There is a SPI controller which support :

- half/full duplex
- single/dual/quad SPI
- SDR/DDR/.. data rate

You can find its APB3 implementation here :

https://github.com/SpinalHDL/SpinalHDL/blob/68b6158700fc2440ea7980406f927262c004faca/lib/src/main/scala/spinal/lib/com/spi/xdr/Apb3SpiXdrMasterCtrl.scala#L43

Configuration

Here is an example.

```
Apb3SpiXdrMasterCtrl(
  SpiXdrMasterCtrl.MemoryMappingParameters(
    SpiXdrMasterCtrl.Parameters(
      dataWidth = 8, // Each transfer will be 8 bits
      timerWidth = 12, // The timer is used to slow down the transmition
      spi = SpiXdrParameter( //Specify the physical SPI interface
        dataWidth = 4, //Number of physical SPI data pins
        ioRate = 1, //Specify the number of transfer that each spi pin can do per␣
↪clock 1 => SDR, 2 => DDR
        ssWidth = 1 //Number of chip selects
      )
    )
    .addFullDuplex(id = 0) // Add support for regular SPI (MISO / MOSI) using the␣
↪mode id 0
    .addHalfDuplex( // Add another mode
      id = 1,  // mapped on mode id 1
      rate = 1, // When rate is 1, the clock will do up to one toggle per cycle,␣
↪divided by the (timer+1)
                // When rate bigger (ex 2), the controller will ignore the timer, and␣
↪use the SpiXdrParameter.ioRate
                // capabilities to emit up to "rate" transition per clock cycle.
      ddr = false, // sdr => 1 bit per SPI clock, DDR => 2 bits per SPI clock
      spiWidth = 4 //Number of physical SPI data pin used for serialisation
    ),
    cmdFifoDepth = 32,
    rspFifoDepth = 32,
    xip = null
  )
)
```

Software Driver

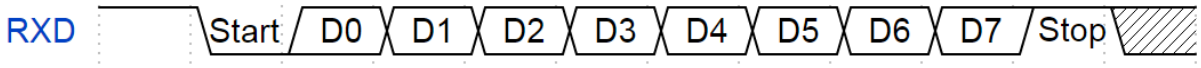
See :

https://github.com/SpinalHDL/SaxonSoc/blob/dev-0.3/software/standalone/driver/spi.h https://github.com/SpinalHDL/SaxonSoc/blob/dev-0.3/software/standalone/spiDemo/src/main.c

10.12.2 UART

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

10.12.3 USB device

Here exists a USB device controller in the SpinalHDL library.

A few bullet points to summarise support:

- Implemented to allow a CPU to configure and manage the endpoints
- A internal ram which store the endpoints states and transactions descriptors
- Up to 16 endpoints (for virtualy no price)
- Support USB host full speed (12Mbps)
- Test on linux using its own driver (https://github.com/SpinalHDL/linux/blob/dev/drivers/usb/gadget/udc/spinal_udc.c)
- Bmb memory interace for the configuration
- Require a clock for the internal phy which is a multiple of 12 Mhz at least 48 Mhz
- The controller frequency is not restricted
- No external phy required

Linux gadget tested and functional :

- Serial connection
- Ethernet connection
- Mass storage (~8 Mbps on ArtyA7 linux)

Deployments :

- https://github.com/SpinalHDL/SaxonSoc/tree/dev-0.3/bsp/digilent/ArtyA7SmpLinux
- https://github.com/SpinalHDL/SaxonSoc/tree/dev-0.3/bsp/radiona/ulx3s/smp

Architecture

The controller is composed of :

- A few control registers
- A internal ram used to store the endpoint status, the transfer descriptors and the endpoint 0 SETUP data.

A linked list of descriptors for each endpoint in order to handle the USB IN/OUT transactions and data.

The endpoint 0 manage the IN/OUT transactions like all the other endpoints but has some additional hardware to manage the SETUP transactions :

- Its linked list is cleared on each setup transactions
- The data from the SETUP transaction is stored in a fixed location (SETUP_DATA)
- It has a specific interrupt flag for SETUP transactions

Registers

Note that all registers and memories of the controller are only accessible in 32 bits word access, bytes access isn't supported.

FRAME (0xFF00)

Name	Type	Bits	Description
usbFrameId	RO	31-0	Current usb frame id

ADDRESS (0xFF04)

Name	Type	Bits	Description
address	WO	6-0	The device will only listen at tokens with the specified address This field is automaticaly cleared on usb reset events
enable	WO	8	Enable the USB address filtering if set
trigger	WO	9	Set the enable (see above) on the next EP0 IN tocken completion Cleared by the hardware after any EP0 completion

The idea here is to keep the whole register cleared until a USB SET_ADDRESS setup packet is received on EP0. At that moment, you can set the address and the trigger field, then provide the IN zero length descriptor to EP0 to finalise the SET_ADDRESS sequance. The controller will then automaticaly turn on the address filtering at the completion of that descriptor.

INTERRUPT (0xFF08)

Individual bits of this register can be cleared by writing '1' in them. Reading this register returns the current interrupt status.

Name	Type	Bits	Description
endpoints	W1C	15-0	Raised when an endpoint generates an interrupt
reset	W1C	16	Raised when a USB reset occurs
ep0Setup	W1C	17	Raised when endpoint 0 receives a setup transaction
suspend	W1C	18	Raised when a USB suspend occurs
resume	W1C	19	Raised when a USB resume occurs
disconnect	W1C	20	Raised when a USB disconnect occurs

HALT (0xFF0C)

This register allows placement of a single endpoint into a dormant state in order to ensure atomicity of CPU operations, allowing to do things as read/modify/write on the endpoint registers and descriptors. The peripheral will return NAK if the given endpoint is addressed by the usb host while halt is enabled and the endpoint is enabled.

Name	Type	Bits	Description
endpointId	WO	3-0	The endpoint you want to put in sleep
enable	WO	4	When set halt is active, when clear endpoint is unhalted.
effective enable	RO	5	After setting the enable, you need to wait for this bit to be set by the hardware itself to ensure atomicity

CONFIG (0xFF10)

Name	Type	Bits	Description
pullupSet	SO	0	Write '1' to enable the USB device pullup on the dp pin
pullupClear	SO	1	
interruptEnableSet	SO	2	Write '1' to let the present and future interrupt happening
interruptEnableClear	SO	3	

INFO (0xFF20)

Name	Type	Bits	Description
ramSize	RO	3-0	The internal ram will have (1 << this) bytes

ENDPOINTS (0x0000 - 0x003F)

The endpoints status are stored at the begining of the internal ram over one 32 bits word each.

Name	Type	Bits	Description
enable	RW	0	If not set, the endpoint will ignore all the trafic
stall	RW	1	If set, the endpoint will always return STALL status
nack	RW	2	If set, the endpoint will always return NACK status
dataPhase	RW	3	Specify the IN/OUT data PID used. '0' => DATA0. This field is also updated by the controller.
head	RW	15-4	Specify the current descriptor head (linked list). 0 => empty list, byte address = this << 4
isochronous	RW	16	
maxPacketSize	RW	31-22	

To get a endpoint responsive you need :

- Set its enable flag to 1

Then the there is a few cases : - Either you have the stall or nack flag set, and so, the controller will always respond with the corresponding responses - Either, for EP0 setup request, the controller will not use descriptors, but will instead write the data into the SETUP_DATA register, and ACK - Either you have a empty linked list (head==0) in which case it will answer NACK - Either you have at least one descriptor pointed by head, in which case it will execute it and ACK if all was going smooth

SETUP_DATA (0x0040 - 0x0047)

When endpoint 0 receives a SETUP transaction, the data of the transaction will be stored in this location.

Descriptors

Descriptors allows to specify how an endpoint needs to handle the data phase of IN/OUT transactions. They are stored in the internal ram, can be linked together via their linked lists and need to be aligned on 16 bytes boundaries

Name	Word	Bits	Description
offset	0	15-0	Specify the current progress in the transfer (in byte)
code	0	19-16	0xF => in progress, 0x0 => success
next	1	15-4	Pointer to the next descriptor 0 => nothing, byte address = this << 4
length	1	31-16	Number of bytes allocated for the data field
direction	2	16	'0' => OUT, '1' => IN
interrupt	2	17	If set, the completion of the descriptor will generate an interrupt.
completionOnFull	2	18	Normally, a descriptor completion only occurs when a USB transfer is smaller than the maxPacketSize. But if this field is set, then when the descriptor become full is also a considered as a completion event. (offset == length)
data1OnCompletion	2	19	force the endpoint dataPhase to DATA1 on the completion of the descriptor
data	...	...	

Note, if the controller receives a frame where the IN/OUT does not match the descriptor IN/OUT, the frame will be ignored.

Also, to initialise a descriptor, the CPU should set the code field to 0xF

Usage

```
import spinal.core._
import spinal.core.sim._
import spinal.lib.bus.bmb.BmbParameter
import spinal.lib.com.usb.phy.UsbDevicePhyNative
import spinal.lib.com.usb.sim.UsbLsFsPhyAbstractIoAgent
import spinal.lib.com.usb.udc.{UsbDeviceCtrl, UsbDeviceCtrlParameter}


case class UsbDeviceTop() extends Component {
  val ctrlCd = ClockDomain.external("ctrlCd", frequency = FixedFrequency(100 MHz))
  val phyCd = ClockDomain.external("phyCd", frequency = FixedFrequency(48 MHz))

  val ctrl = ctrlCd on new UsbDeviceCtrl(
    p = UsbDeviceCtrlParameter(
      addressWidth = 14
    ),
    bmbParameter = BmbParameter(
      addressWidth = UsbDeviceCtrl.ctrlAddressWidth,
      dataWidth = 32,
```

(continues on next page)

(continued from previous page)

```
      sourceWidth = 0,
      contextWidth = 0,
      lengthWidth = 2
    )
  )

  val phy = phyCd on new UsbDevicePhyNative(sim = true)
  ctrl.io.phy.cc(ctrlCd, phyCd) <> phy.io.ctrl

  val bmb = ctrl.io.ctrl.toIo()
  val usb = phy.io.usb.toIo()
  val power = phy.io.power.toIo()
  val pullup = phy.io.pullup.toIo()
  val interrupts = ctrl.io.interrupt.toIo()
}


object UsbDeviceGen extends App{
  SpinalVerilog(new UsbDeviceTop())
}
```

10.12.4 USB OHCI

Here exists a USB OHCi controller (host) in the SpinalHDL library.

A few bullet points to summarise support:

- It follow the *OpenHCI Open Host Controller Interface Specification for USB* specification (OHCI).
- It is compatible with the upstream linux / uboot OHCI drivers already. (there is also an OHCI driver on tinyUSB)
- This provides USB host full speed and low speed capabilities (12Mbps and 1.5Mbps)
- Tested on linux and uboot
- One controller can host multiple ports (up to 16)
- Bmb memory interface for DMA accesses
- Bmb memory interace for the configuration
- Requires a clock for the internal phy which is a multiple of 12 Mhz at least 48 Mhz
- The controller frequency is not restricted
- No external phy required

Devices tested and functional :

- Mass storage (~8 Mbps on ArtyA7 linux)
- Keyboard / Mouse
- Audio output
- Hub

Limitations :

- Some USB hub (had one so far) do not like having a full speed host with low speed devices attached.
- Some modern devices will not work on USB full speed (ex : Gbps ethernet adapter)
- Require memory coherency with the CPU (or the cpu need to be able to flush its data cache in the driver)

Deployments :

- https://github.com/SpinalHDL/SaxonSoc/tree/dev-0.3/bsp/digilent/ArtyA7SmpLinux
- https://github.com/SpinalHDL/SaxonSoc/tree/dev-0.3/bsp/radiona/ulx3s/smp

Usage

```
import spinal.core._
import spinal.core.sim._
import spinal.lib.bus.bmb._
import spinal.lib.bus.bmb.sim._
import spinal.lib.bus.misc.SizeMapping
import spinal.lib.com.usb.ohci._
import spinal.lib.com.usb.phy.UsbHubLsFs.CtrlCc
import spinal.lib.com.usb.phy._

class UsbOhciTop(val p : UsbOhciParameter) extends Component {
  val ohci = UsbOhci(p, BmbParameter(
    addressWidth = 12,
    dataWidth = 32,
    sourceWidth = 0,
    contextWidth = 0,
    lengthWidth = 2
  ))

  val phyCd = ClockDomain.external("phyCd", frequency = FixedFrequency(48 MHz))
  val phy = phyCd(UsbLsFsPhy(p.portCount, sim=true))

  val phyCc = CtrlCc(p.portCount, ClockDomain.current, phyCd)
  phyCc.input <> ohci.io.phy
  phyCc.output <> phy.io.ctrl

  // propagate io signals
  val irq = ohci.io.interrupt.toIo
  val ctrl = ohci.io.ctrl.toIo
  val dma = ohci.io.dma.toIo
  val usb = phy.io.usb.toIo
  val management = phy.io.management.toIo
}

object UsbHostGen extends App {
  val p = UsbOhciParameter(
    noPowerSwitching = true,
    powerSwitchingMode = true,
    noOverCurrentProtection = true,
    powerOnToPowerGoodTime = 10,
    dataWidth = 64, //DMA data width, up to 128
    portsConfig = List.fill(4)(OhciPortParameter()) //4 Ports
  )

  SpinalVerilog(new UsbOhciTop(p))
}
```

10.13 IO

10.13.1 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) {

}
```

10.13.2 TriState

Tri-state signals are difficult 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) {

}
```

10.14 Graphics

10.14.1 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
```

10.14.2 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 bits)
  val colorEnd = UInt(timingsWidth bits)
  val syncStart = UInt(timingsWidth bits)
  val syncEnd = UInt(timingsWidth bits)
}

case class VgaTimings(timingsWidth: Int) extends Bundle {
  val h = VgaTimingsHV(timingsWidth)
```

(continues on next page)

(continued from previous page)

```
  val v = VgaTimingsHV(timingsWidth)

   def setAs_h640_v480_r60 = ...
   def driveFrom(busCtrl : BusSlaveFactory,baseAddress : Int) = ...
}
```

VGA controller

An VGA controller is available. Its 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.

10.15 EDA

10.15.1 QSysify

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

10.15.2 QuartusFlow

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()`.

10.16 Pipeline

10.16.1 Introduction

spinal.lib.misc.pipeline provides a pipelining API. The main advantages over manual pipelining are :

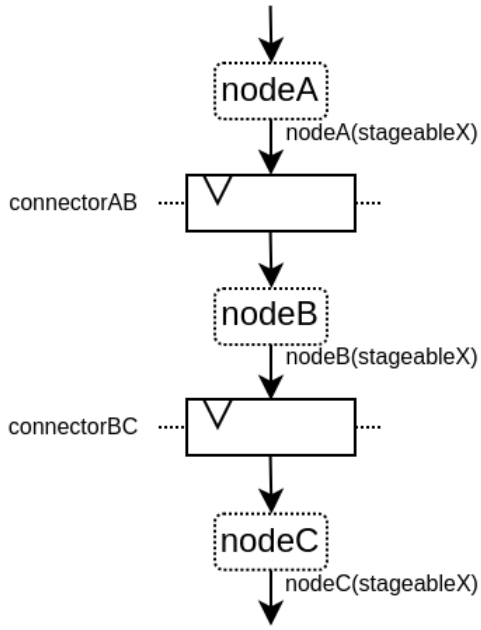
- You don't have to predefine all the signal elements needed for the entire staged system upfront. You can create and consume stagable signals in a more ad hoc fashion as your design requires - without needing to refactor all the intervening stages to know about the signal
- Signals of the pipeline can utilize the powerful parametrization capabilities of SpinalHDL and be subject to optimization/removal if a specific design build does not require a particular parametrized feature, without any need to modify the staging system design or project code base in a significant way.
- Manual retiming is much easier, as you don't have to handle the registers / arbitration manualy
- Manage the arbitration by itself

The API is composed of 4 main things :

- Node : which represents a layer in the pipeline
- Link : which allows to connect nodes to each other
- Builder : which will generate the hardware required for a whole pipeline
- Payload : which are used to retrieve hardware signals on nodes along the pipeline

It is important to understand that Payload isn't a hardware data/signal instance, but a key to retrieve a data/signal on nodes along the pipeline, and that the pipeline builder will then automatically interconnect/pipeline every occurrence of a given Payload between nodes.

Here is an example to illustrate :



Here is a video about this API :

- https://www.youtube.com/watch?v=74h_-FMWWIM

Simple example

Here is a simple example which only uses the basics of the API :

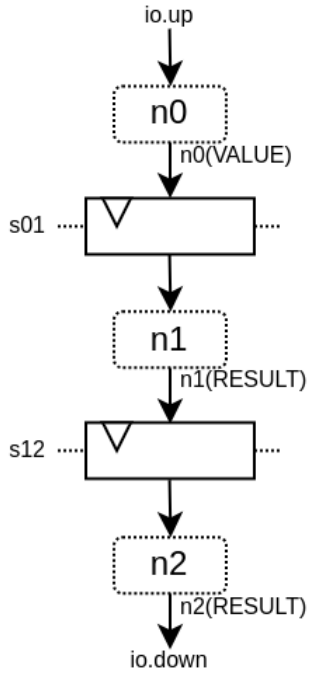
```
import spinal.core._
import spinal.core.sim._
import spinal.lib._
import spinal.lib.misc.pipeline._

class TopLevel extends Component {
  val io = new Bundle{
    val up = slave Stream (UInt(16 bits))
    val down = master Stream (UInt(16 bits))
  }

  // Let's define 3 Nodes for our pipeline
  val n0, n1, n2 = Node()

  // Let's connect those nodes by using simples registers
  val s01 = StageLink(n0, n1)
  val s12 = StageLink(n1, n2)

  // Let's define a few Payload things that can go through the pipeline
  val VALUE = Payload(UInt(16 bits))
  val RESULT = Payload(UInt(16 bits))

  // Let's bind io.up to n0
  io.up.ready := n0.ready
  n0.valid := io.up.valid
  n0(VALUE) := io.up.payload

  // Let's do some processing on n1
```

(continues on next page)

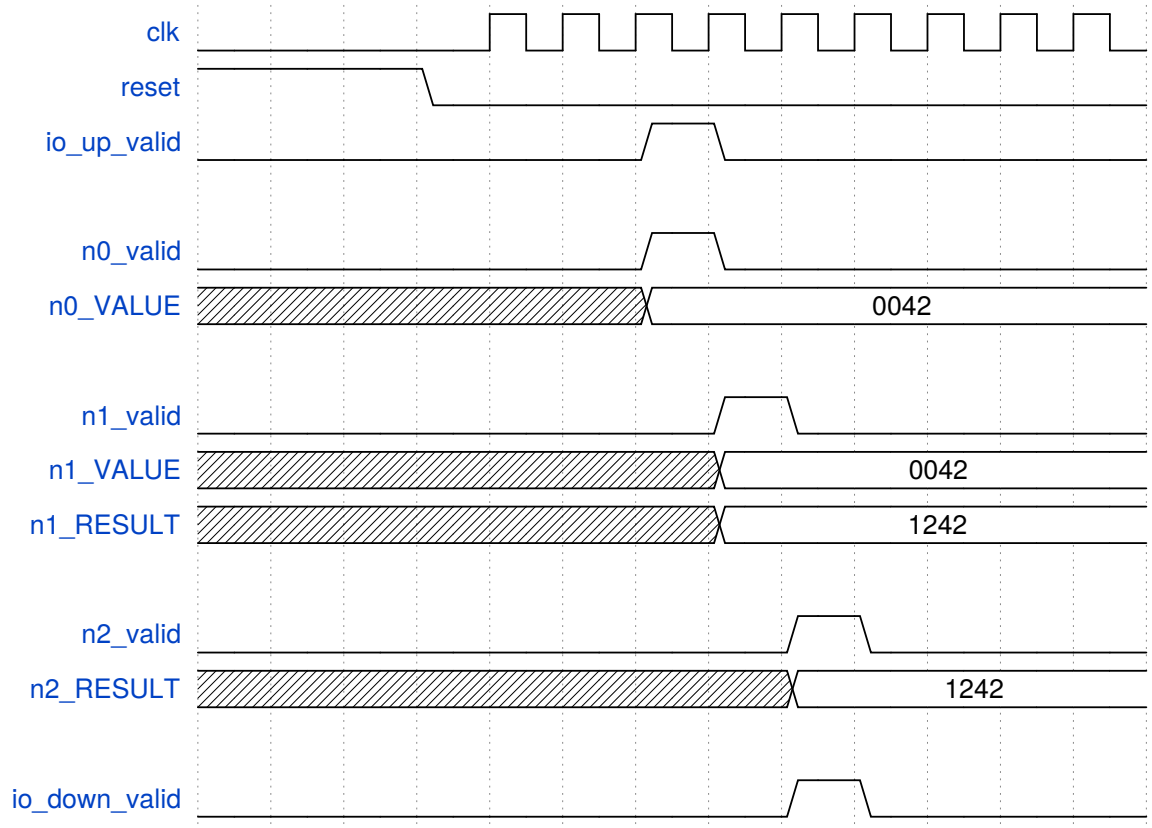
(continued from previous page)

```
  n1(RESULT) := n1(VALUE) + 0x1200

  // Let's bind n2 to io.down
  n2.ready := io.down.ready
  io.down.valid := n2.valid
  io.down.payload := n2(RESULT)

  // Let's ask the builder to generate all the required hardware
  Builder(s01, s12)
}
```

This will produce the following hardware :



Here is a simulation wave :



Here is the same example but using more of the API :

```
import spinal.core._
import spinal.core.sim._
import spinal.lib._
import spinal.lib.misc.pipeline._

class TopLevel extends Component {
  val VALUE = Payload(UInt(16 bits))

  val io = new Bundle{
    val up = slave Stream(VALUE)  //VALUE can also be used as a HardType
    val down = master Stream(VALUE)
  }

  // NodesBuilder will be used to register all the nodes created, connect them via␣
↪stages and generate the hardware
  val builder = new NodesBuilder()

  // Let's define a Node which connect from io.up
  val n0 = new builder.Node{
    arbitrateFrom(io.up)
    VALUE := io.up.payload
  }

  // Let's define a Node which do some processing
  val n1 = new builder.Node{
    val RESULT = insert(VALUE + 0x1200)
  }

  //  Let's define a Node which connect to io.down
```

(continues on next page)

(continued from previous page)

```
  val n2 = new builder.Node {
    arbitrateTo(io.down)
    io.down.payload := n1.RESULT
  }

  // Let's connect those nodes by using registers stages and generate the related␣
↪hardware
  builder.genStagedPipeline()
}
```

10.16.2 Payload

Payload objects are used to refer to data which can go through the pipeline. Technicaly speaking, Payload is a HardType which has a name and is used as a "key" to retrieve the signals in a certain pipeline stage.

```
val PC = Payload(UInt(32 bits))
val PC_PLUS_4 = Payload(UInt(32 bits))

val n0, n1 = Node()
val s01 = StageLink(n0, n1)

n0(PC) := 0x42
n1(PC_PLUS_4) := n1(PC) + 4
```

Note that I got used to name the Payload instances using uppercase. This is to make it very explicit that the thing isn't a hardware signal, but are more like a "key/type" to access things.

10.16.3 Node

Node mostly hosts the valid/ready arbitration signals, and the hardware signals required for all the Payload values going through it.

You can access its arbitration via :

API	Access	Description
node.valid	RW	Is the signal which specifies if a transaction is present on the node. It is driven by the upstream. Once asserted, it must only be de-asserted the cycle after which either both valid and ready or node.cancel are high. valid must not depend on ready.
node.ready	RW	Is the signal which specifies if the node's transaction can proceed downstream. It is driven by the downstream to create backpresure. The signal has no meaning when there is no transaction (node.valid being deasserted)
node.cancel	RW	Is the signal which specifies if the node's transaction in being canceled from the pipeline. It is driven by the downstream. The signal has no meaning when there is no transaction (node.valid being deasserted)
node.isValid	RO	node.valid's read only accessor
node.isReady	RO	node.ready's read only accessor
node.isCancel	RO	node.cancel's read only accessor
node.isFiring	RO	True when the node transaction is successfuly moving futher (valid && ready && !cancel). Useful to commit state changes.
node.isMoving	RO	True when the node transaction will not be present anymore on the node (starting from the next cycle), either because downstream is ready to take the transaction, or because the transaction is canceled from the pipeline. (valid && (ready || cancel)). Useful to "reset" states.
node.isCanceling	RO	True when the node transaction is being canceled. Meaning that it will not appear anywhere in the pipeline in future cycles.

Note that the node.valid/node.ready signals follows the same conventions than the *Stream*'s ones .

The Node controls (valid/ready/cancel) and status (isValid, isReady, isCancel, isFiring, ...) signals are created on demande. So for instance you can create pipeline with no backpresure by never refering to the ready signal. That's why it is important to use status signals when you want to read the status of something and only use control signals when you to drive something.

Here is a list of arbitration cases you can have on a node. valid/ready/cancel define the state we are in, while isFiring/isMoving result of those :

valid	ready	cancel	Description	isFiring	isMoving
0	X	X	No transaction	0	0
1	1	0	Going through	1	1
1	0	0	Blocked	0	0
1	X	1	Canceled	0	1

Note that if you want to model things like for instance a CPU stage which can block and flush stuff, take a look a the CtrlLink, as it provides the API to do such things.

You can access signals referenced by a Payload via:

API	Description
node(Payload)	Return the corresponding hardware signal
node(Payload, Any)	Same as above, but include a second argument which is used as a "secondary key". This eases the construction of multi-lane hardware. For instance, when you have a multi issue CPU pipeline, you can use the lane Int id as secondary key
node.insert(Data)	Return a new Payload instance which is connected to the given Data hardware signal

```
val n0, n1 = Node()

val PC = Payload(UInt(32 bits))
```

(continues on next page)

(continued from previous page)

```
n0(PC) := 0x42
n0(PC, "true") := 0x42
n0(PC, 0x666) := 0xEE
val SOMETHING = n0.insert(myHardwareSignal) //This create a new Payload
when(n1(SOMETHING) === 0xFFAA){ ... }
```

While you can manualy drive/read the arbitration/data of the first/last stage of your pipeline, there is a few utilities to connect its boundaries.

API	Description
node.arbitrateFrom(Stream[T]])	Drive a node arbitration from a stream.
node.arbitrateFrom(Flow[T]])	Drive a node arbitration from the Flow.
node.arbitrateTo(Stream[T]])	Drive a stream arbitration from the node.
node.arbitrateTo(Flow[T]])	Drive a Flow arbitration from the node.
node.driveFrom(Stream[T]])((Node, T) => Unit)	Drive a node from a stream. The provided lambda function can be use to connect the data
node.driveFrom(Flow[T]])((Node, T) => Unit)	Same as above but for Flow
node.driveTo(Stream[T]])((T, Node) => Unit)	Drive a stream from the node. The provided lambda function can be use to connect the data
node.driveTo(Flow[T]])((T, Node) => Unit)	Same as above but for Flow

```
val n0, n1, n2 = Node()

val IN = Payload(UInt(16 bits))
val OUT = Payload(UInt(16 bits))

n1(OUT) := n1(IN) + 0x42

// Define the input / output stream that will be later connected to the pipeline
val up = slave Stream(UInt(16 bits))
val down = master Stream(UInt(16 bits)) //Note master Stream(OUT) is good aswell

n0.driveFrom(up)((self, payload) => self(IN) := payload)
n2.driveTo(down)((payload, self) => payload := self(OUT))
```

In order to reduce verbosity, there is a set of implicit conversions between Payload toward their data representation which can be used when you are in the context of a Node :

```
val VALUE = Payload(UInt(16 bits))
val n1 = new Node{
    val PLUS_ONE = insert(VALUE + 1) // VALUE is implicitly converted into its␣
↪n1(VALUE) representation
}
```

You can also use those implicit conversions by importing them :

```
val VALUE = Payload(UInt(16 bits))
val n1 = Node()

val n1Stuff = new Area {
    import n1._
    val PLUS_ONE = insert(VALUE) + 1 // Equivalent to n1.insert(n1(VALUE)) + 1
}
```

There is also an API which alows you to create new Area which provide the whole API of a given node instance (including implicit convertion) without import :

```
val n1 = Node()
val VALUE = Payload(UInt(16 bits))

val n1Stuff = new n1.Area{
    val PLUS_ONE = insert(VALUE) + 1 // Equivalent to n1.insert(n1(VALUE)) + 1
}
```

Such feature is very useful when you have parametrizable pipeline locations for your hardware (see retiming example).

10.16.4 Links

There is few different Links already implemented (but you could also create your own custom one). The idea of Links is to connect two nodes together in various ways. They generally have a *up* Node and a *down* Node.

DirectLink

Very simple, it connect two nodes with wires only. Here is an example :

```
val c01 = DirectLink(n0, n1)
```

StageLink

This connect two nodes using registers on the data / valid signals and some arbitration on the ready.

```
val c01 = StageLink(n0, n1)
```

S2mLink

This connect two nodes using registers on the ready signal, which can be useful to improve backpresure combinatorial timings.

```
val c01 = S2mLink(n0, n1)
```

CtrlLink

This is kind of a special Link, as connect two nodes with optional flow control / bypass logic. Its API should be flexible enough to implement a CPU stage with it.

Here is its flow control API (The Bool arguments enable the features) :

API	Description
haltWhen(Bool)	Allows to block the current transaction (clear up.ready down.valid)
throwWhen(Bool)	Allows to cancel the current transaction from the pipeline (clear down.valid and make the transaction driver forget its current state)
forgetOneWhen(Bool)	Allows to request the upstream to forget its current transaction (but doesn't clear the down.valid)
ignoreReadyWhen(Bool)	Allows to ignore the downstream ready (set up.ready)
duplicateWhen(Bool)	Allows to duplicate the current transaction (clear up.ready)
terminateWhen(Bool)	Allows to hide the current transaction from downstream (clear down.valid)

Also note that if you want to do flow control in a conditional scope (ex in a when statement), you can call the following functions :

- haltIt(), duplicateIt(), terminateIt(), forgetOneNow(), ignoreReadyNow(), throwIt()

```
val c01 = CtrlLink(n0, n1)

c01.haltWhen(something) // Explicit halt request

when(somethingElse){
    c01.haltIt() // Conditional scope sensitive halt request, same as c01.
↪haltWhen(somethingElse)
}
```

You can retrieve which nodes are connected to the Link using node.up / node.down.

The CtrlLink also provide an API to access Payload :

API	Description
link(Payload)	Same as Link.down(Payload)
link(Payload, Any)	Same as Link.down(Payload, Any)
link.insert(Data)	Same as Link.down.insert(Data)
link.bypass(Payload)	Allows to conditionaly override a Payload value between link.up -> link.down. This can be used to fix data hazard in CPU pipelines for instance.

```
val c01 = CtrlLink(n0, n1)

val PC = Payload(UInt(32 bits))
c01(PC) := 0x42
c01(PC, 0x666) := 0xEE

val DATA = Payload(UInt(32 bits))
// Let's say Data is inserted in the pipeline before c01
when(hazard){
    c01.bypass(DATA) := fixedValue
}

// c01(DATA) and below will get the hazard patch
```

Note that if you create a CtrlLink without node arguments, it will create its own nodes internally.

```
val decode = CtrlLink()
val execute = CtrlLink()

val d2e = StageLink(decode.down, execute.up)
```

Other Links

There is also a JoinLink / ForkLink implemented.

Your custom Link

You can implement your custom links by implementing the Link base class.

```
trait Link extends Area{
  def ups : Seq[Node]
  def downs : Seq[Node]

  def propagateDown(): Unit
  def propagateUp(): Unit
  def build() : Unit
}
```

But that API may change a bit, as it is still fresh.

10.16.5 Builder

To generate the hardware of your pipeline, you need to give a list of all the Links used in your pipeline.

```
// Let's define 3 Nodes for our pipeline
val n0, n1, n2 = Node()

// Let's connect those nodes by using simples registers
val s01 = StageLink(n0, n1)
val s12 = StageLink(n1, n2)

// Let's ask the builder to generate all the required hardware
Builder(s01, s12)
```

There is also a set of "all in one" builders that you can instanciate to help yourself.

For instance there is the NodesBuilder class which can be used to create sequentially staged pipelines :

```
val builder = new NodesBuilder()

// Let's define a few nodes
val n0, n1, n2 = new builder.Node

// Let's connect those nodes by using registers and generate the related hardware
builder.genStagedPipeline()
```

10.16.6 Composability

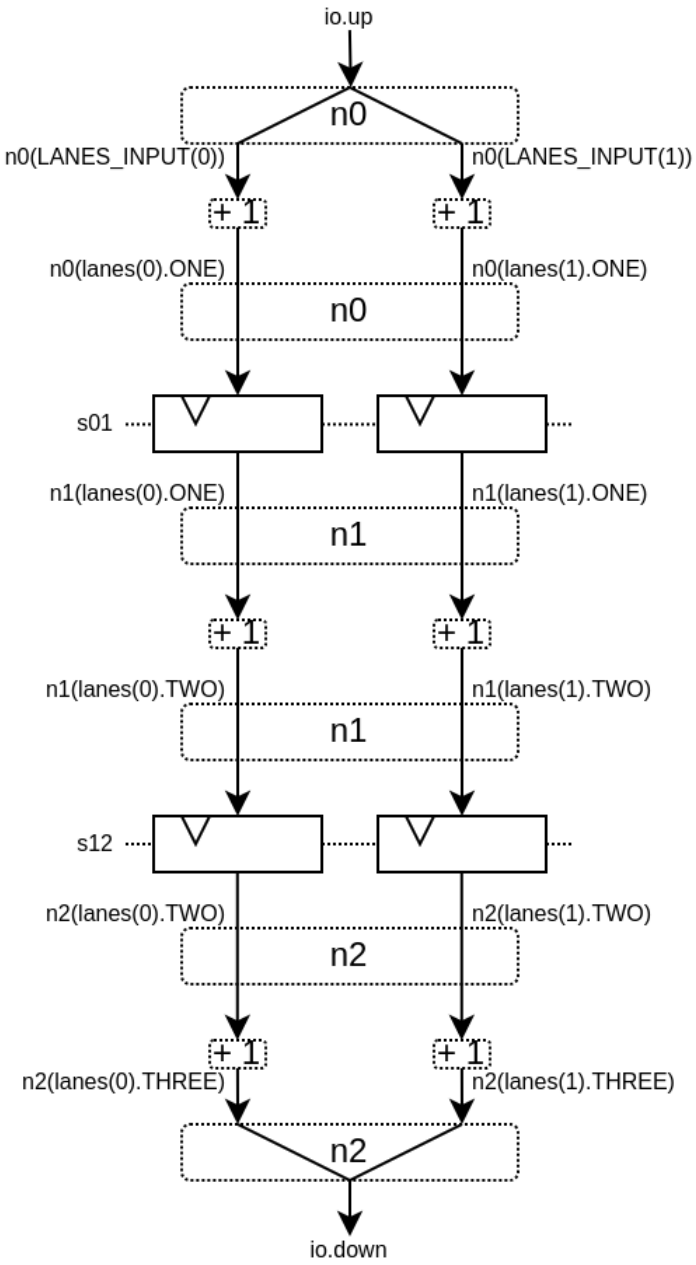
One good thing about the API is that it easily allows to compose a pipeline with multiple parallel things. What i mean by "compose" is that sometime the pipeline you need to design has parallel processing to do.

Imagine you need to do floating point multiplication on 4 pairs of numbers (to later sum them). If those 4 pairs a provided at the same time by a single stream of data, then you don't want 4 different pipelines to multiply them, instead you want to process them all in parallel in the same pipeline.

The example below show a pattern which composes a pipeline with multiple lanes to process them in parallel.

```
// This area allows to take a input value and do +1 +1 +1 over 3 stages.
// I know that's useless, but let's pretend that instead it does a multiplication␣
↪between two numbers over 3 stages (for FMax reasons)
class Plus3(INPUT: Payload[UInt], stage1: Node, stage2: Node, stage3: Node) extends␣
↪Area {
  val ONE = stage1.insert(stage1(INPUT) + 1)
  val TWO = stage2.insert(stage2(ONE) + 1)
  val THREE = stage3.insert(stage3(TWO) + 1)
}

// Let's define a component which takes a stream as input,
// which carries 'lanesCount' values that we want to process in parallel
// and put the result on an output stream
class TopLevel(lanesCount : Int) extends Component {
  val io = new Bundle{
    val up = slave Stream(Vec.fill(lanesCount)(UInt(16 bits)))
    val down = master Stream(Vec.fill(lanesCount)(UInt(16 bits)))
  }

  // Let's define 3 Nodes for our pipeline
  val n0, n1, n2 = Node()

  // Let's connect those nodes by using simples registers
  val s01 = StageLink(n0, n1)
  val s12 = StageLink(n1, n2)

  // Let's bind io.up to n0
  n0.arbitrateFrom(io.up)
  val LANES_INPUT = io.up.payload.map(n0.insert(_))

  // Let's use our "reusable" Plus3 area to generate each processing lane
  val lanes = for(i <- 0 until lanesCount) yield new Plus3(LANES_INPUT(i), n0, n1, n2)

  // Let's bind n2 to io.down
  n2.arbitrateTo(io.down)
  for(i <- 0 until lanesCount) io.down.payload(i) := n2(lanes(i).THREE)

  // Let's ask the builder to generate all the required hardware
  Builder(s01, s12)
}
```

This will produce the following data path (assuming lanesCount = 2), abitration not being shown :



10.16.7 Retiming / Variable lenth

Sometime you want to design a pipeline, but you don't really know where the critical paths will be and what the right balance between stages is. And often you can't rely on the synthesis tool doing a good job with automatic retiming.

So, you kind of need a easy way to move the logic of your pipeline around.

Here is how it can be done with this pipelining API :

```
// Define a component which will take a input stream of RGB value
// Process (~(R + G + B)) * 0xEE
// And provide that result into an output stream
class RgbToSomething(addAt : Int,
                     invAt : Int,
                     mulAt : Int,
                     resultAt : Int) extends Component {
```

(continues on next page)

(continued from previous page)

```
  val io = new Bundle {
    val up = slave Stream(spinal.lib.graphic.Rgb(8, 8, 8))
    val down = master Stream (UInt(16 bits))
  }

  // Let's define the Nodes for our pipeline
  val nodes = Array.fill(resultAt+1)(Node())

  // Let's specify which node will be used for what part of the pipeline
  val insertNode = nodes(0)
  val addNode = nodes(addAt)
  val invNode = nodes(invAt)
  val mulNode = nodes(mulAt)
  val resultNode = nodes(resultAt)

  // Define the hardware which will feed the io.up stream into the pipeline
  val inserter = new insertNode.Area {
    arbitrateFrom(io.up)
    val RGB = insert(io.up.payload)
  }

  // sum the r g b values of the color
  val adder = new addNode.Area {
    val SUM = insert(inserter.RGB.r + inserter.RGB.g + inserter.RGB.b)
  }

  // flip all the bit of the RGB sum
  val inverter = new invNode.Area {
    val INV = insert(~adder.SUM)
  }

  // multiply the inverted bits with 0xEE
  val multiplier = new mulNode.Area {
    val MUL = insert(inverter.INV*0xEE)
  }

  // Connect the end of the pipeline to the io.down stream
  val resulter = new resultNode.Area {
    arbitrateTo(io.down)
    io.down.payload := multiplier.MUL
  }

  // Let's connect those nodes sequencialy by using simples registers
  val links = for (i <- 0 to resultAt - 1) yield StageLink(nodes(i), nodes(i + 1))

  // Let's ask the builder to generate all the required hardware
  Builder(links)
}
```

If then you generate this component like this :

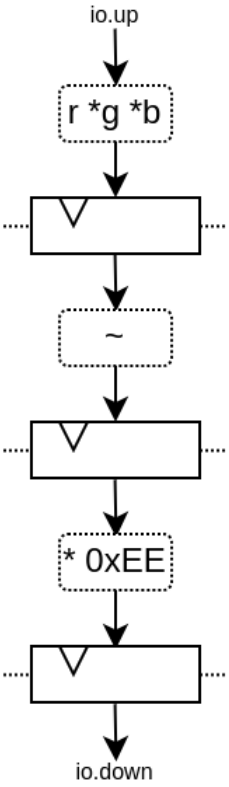
```
SpinalVerilog(
  new RgbToSomething(
    addAt    = 0,
    invAt    = 1,
```

(continues on next page)

(continued from previous page)

```
    mulAt    = 2,
    resultAt = 3
  )
)
```

You will get a 4 stages separated by 3 layer of flip flop doing your processing :



Note the generated hardware verilog is kinda clean (by my standards at least :P) :

```
// Generator : SpinalHDL dev    git head : 1259510dd72697a4f2c388ad22b269d4d2600df7
// Component : RgbToSomething
// Git hash  : 63da021a1cd082d22124888dd6c1e5017d4a37b2

`timescale 1ns/1ps

module RgbToSomething (
  input  wire          io_up_valid,
  output wire          io_up_ready,
  input  wire [7:0]    io_up_payload_r,
  input  wire [7:0]    io_up_payload_g,
  input  wire [7:0]    io_up_payload_b,
  output wire          io_down_valid,
  input  wire          io_down_ready,
  output wire [15:0]   io_down_payload,
  input  wire          clk,
  input  wire          reset
);

  wire       [7:0]    _zz_nodes_0_adder_SUM;
  reg        [15:0]   nodes_3_multiplier_MUL;
  wire       [15:0]   nodes_2_multiplier_MUL;
```

(continues on next page)

(continued from previous page)

```
reg        [7:0]    nodes_2_inverter_INV;
wire       [7:0]    nodes_1_inverter_INV;
reg        [7:0]    nodes_1_adder_SUM;
wire       [7:0]    nodes_0_adder_SUM;
wire       [7:0]    nodes_0_inserter_RGB_r;
wire       [7:0]    nodes_0_inserter_RGB_g;
wire       [7:0]    nodes_0_inserter_RGB_b;
wire                nodes_0_valid;
reg                 nodes_0_ready;
reg                 nodes_1_valid;
reg                 nodes_1_ready;
reg                 nodes_2_valid;
reg                 nodes_2_ready;
reg                 nodes_3_valid;
wire                nodes_3_ready;
wire                when_StageLink_l56;
wire                when_StageLink_l56_1;
wire                when_StageLink_l56_2;

assign _zz_nodes_0_adder_SUM = (nodes_0_inserter_RGB_r + nodes_0_inserter_RGB_g);
assign nodes_0_valid = io_up_valid;
assign io_up_ready = nodes_0_ready;
assign nodes_0_inserter_RGB_r = io_up_payload_r;
assign nodes_0_inserter_RGB_g = io_up_payload_g;
assign nodes_0_inserter_RGB_b = io_up_payload_b;
assign nodes_0_adder_SUM = (_zz_nodes_0_adder_SUM + nodes_0_inserter_RGB_b);
assign nodes_1_inverter_INV = (~ nodes_1_adder_SUM);
assign nodes_2_multiplier_MUL = (nodes_2_inverter_INV * 8'hee);
assign io_down_valid = nodes_3_valid;
assign nodes_3_ready = io_down_ready;
assign io_down_payload = nodes_3_multiplier_MUL;
always @(*) begin
  nodes_0_ready = nodes_1_ready;
  if(when_StageLink_l56) begin
    nodes_0_ready = 1'b1;
  end
end

assign when_StageLink_l56 = (! nodes_1_valid);
always @(*) begin
  nodes_1_ready = nodes_2_ready;
  if(when_StageLink_l56_1) begin
    nodes_1_ready = 1'b1;
  end
end

assign when_StageLink_l56_1 = (! nodes_2_valid);
always @(*) begin
  nodes_2_ready = nodes_3_ready;
  if(when_StageLink_l56_2) begin
    nodes_2_ready = 1'b1;
  end
end

assign when_StageLink_l56_2 = (! nodes_3_valid);
always @(posedge clk or posedge reset) begin
```

(continues on next page)

(continued from previous page)

```
    if(reset) begin
      nodes_1_valid <= 1'b0;
      nodes_2_valid <= 1'b0;
      nodes_3_valid <= 1'b0;
    end else begin
      if(nodes_0_ready) begin
        nodes_1_valid <= nodes_0_valid;
      end
      if(nodes_1_ready) begin
        nodes_2_valid <= nodes_1_valid;
      end
      if(nodes_2_ready) begin
        nodes_3_valid <= nodes_2_valid;
      end
    end
  end

  always @(posedge clk) begin
    if(nodes_0_ready) begin
      nodes_1_adder_SUM <= nodes_0_adder_SUM;
    end
    if(nodes_1_ready) begin
      nodes_2_inverter_INV <= nodes_1_inverter_INV;
    end
    if(nodes_2_ready) begin
      nodes_3_multiplier_MUL <= nodes_2_multiplier_MUL;
    end
  end


endmodule
```

Also, you can easily tweak how many stages and where you want the processing to be done, for instance you may want to move the inversion hardware in the same stage as the adder. This can be done the following way :

```
SpinalVerilog(
  new RgbToSomething(
    addAt    = 0,
    invAt    = 0,
    mulAt    = 1,
    resultAt = 2
  )
)
```

Then you may want to remove the output register stage :

```
SpinalVerilog(
  new RgbToSomething(
    addAt    = 0,
    invAt    = 0,
    mulAt    = 1,
    resultAt = 1
  )
)
```

One thing about this example is the necessity intermediate val as *addNode*. I mean :

```
val addNode = nodes(addAt)
// sum the r g b values of the color
val adder = new addNode.Area {
  ...
}
```

Unfortunatly, scala doesn't allow to replace *new addNode.Area* with *new nodes(addAt).Area*. One workaround is to define a class as :

```
class NodeArea(at : Int) extends NodeMirror(nodes(at))
val adder = new NodeArea(addAt) {
    ...
}
```

Depending the scale of your pipeline, it can payoff.

10.16.8 Simple CPU example

Here is a simple/stupid 8 bits CPU example with :

- 3 stages (fetch, decode, execute)
- embedded fetch memory
- add / jump / led /delay instructions

```
class Cpu extends Component {
  val fetch, decode, execute = CtrlLink()
  val f2d = StageLink(fetch.down, decode.up)
  val d2e = StageLink(decode.down, execute.up)

  val PC = Payload(UInt(8 bits))
  val INSTRUCTION = Payload(Bits(16 bits))

  val led = out(Reg(Bits(8 bits))) init(0)

  val fetcher = new fetch.Area{
    val pcReg = Reg(PC) init (0)
    up(PC) := pcReg
    up.valid := True
    when(up.isFiring) {
      pcReg := PC + 1
    }

    val mem = Mem.fill(256)(INSTRUCTION).simPublic
    INSTRUCTION := mem.readAsync(PC)
  }

  val decoder = new decode.Area{
    val opcode = INSTRUCTION(7 downto 0)
    val IS_ADD   = insert(opcode === 0x1)
    val IS_JUMP  = insert(opcode === 0x2)
    val IS_LED   = insert(opcode === 0x3)
    val IS_DELAY = insert(opcode === 0x4)
  }


  val alu = new execute.Area{
```

(continues on next page)

(continued from previous page)

```
    val regfile = Reg(UInt(8 bits)) init(0)

    val flush = False
    for (stage <- List(fetch, decode)) {
      stage.throwWhen(flush, usingReady = true)
    }

    val delayCounter = Reg(UInt(8 bits)) init (0)

    when(isValid) {
      when(decoder.IS_ADD) {
        regfile := regfile + U(INSTRUCTION(15 downto 8))
      }
      when(decoder.IS_JUMP) {
        flush := True
        fetcher.pcReg := U(INSTRUCTION(15 downto 8))
      }
      when(decoder.IS_LED) {
        led := B(regfile)
      }
      when(decoder.IS_DELAY) {
        delayCounter := delayCounter + 1
        when(delayCounter === U(INSTRUCTION(15 downto 8))) {
          delayCounter := 0
        } otherwise {
          execute.haltIt()
        }
      }
    }
  }

  Builder(fetch, decode, execute, f2d, d2e)
}
```

Here is a simple testbench which implement a loop which will make the led counting up.

```
SimConfig.withFstWave.compile(new Cpu).doSim(seed = 2){ dut =>
  def nop() = BigInt(0)
  def add(value: Int) = BigInt(1 | (value << 8))
  def jump(target: Int) = BigInt(2 | (target << 8))
  def led() = BigInt(3)
  def delay(cycles: Int) = BigInt(4 | (cycles << 8))
  val mem = dut.fetcher.mem
  mem.setBigInt(0, nop())
  mem.setBigInt(1, nop())
  mem.setBigInt(2, add(0x1))
  mem.setBigInt(3, led())
  mem.setBigInt(4, delay(16))
  mem.setBigInt(5, jump(0x2))

  dut.clockDomain.forkStimulus(10)
  dut.clockDomain.waitSampling(100)
}
```

10.17 Misc

10.17.1 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.

10.17.2 Plugin

Introduction

For some design, instead of implementing your Component's hardware directly in it, you may instead want to compose its hardware by using some sorts of Plugins. This can provide a few key features :

- You can extend the features of your component by adding new plugins in its parameters. For instance adding Floating point support in a CPU.
- You can swap various implementations of the same functionality just by using another set of plugins. For instance one implementation of a CPU multiplier may fit well on some FPGA, while others may fit well on ASIC.
- It avoid the very very very large hand written toplevel syndrom where everything has to be connected manualy. Instead plugins can discover their neighborhood by looking/using the software interface of other plugins.

VexRiscv and NaxRiscv projects are an example of this. Their are CPUs which have a mostly empty toplevel, and their hardware parts are injected using plugins. For instance :

- PcPlugin

- FetchPlugin
- DecoderPlugin
- RegFilePlugin
- IntAluPlugin
- ...

And those plugins will then negociate/propagate/interconnect to each others via their pool of services.

While VexRiscv use a strict synchronous 2 phase system (setup/build callback), NaxRiscv uses a more flexible approach which uses the spinal.core.fiber API to fork elaboration threads which can interlock each others in order to ensure a workable elaboration ordering.

The Plugin API provide a NaxRiscv like system to define composable components using plugins.

Execution order

The main idea is that you have multiple 2 executions phases :

- Setup phase, in which plugins can lock/retain each others. The idea is not to start negociation / elaboration yet.
- Build phase, in which plugins can negociation / elaboration hardware.

The build phase will not start before all FiberPlugin are done with their setup phase.

```
class MyPlugin extends FiberPlugin {
  val logic = during setup new Area {
    // Here we are executing code in the setup phase
    awaitBuild()
    // Here we are executing code in the build phase
  }
}

class MyPlugin2 extends FiberPlugin {
  val logic = during build new Area {
    // Here we are executing code in the build phase
  }
}
```

Simple example

Here is a simple dummy example with a SubComponent which will be composed using 2 plugins :

```
import spinal.core._
import spinal.lib.misc.plugin._

// Let's define a Component with a PluginHost instance
class SubComponent extends Component {
  val host = new PluginHost()
}

// Let's define a plugin which create a register
class StatePlugin extends FiberPlugin {
  // during build new Area { body } will run the body of code in the Fiber build␣
↪phase, in the context of the PluginHost
  val logic = during build new Area {
    val signal = Reg(UInt(32 bits))
```

(continues on next page)

(continued from previous page)

```
  }
}

// Let's define a plugin which will make the StatePlugin's register increment
class DriverPlugin extends FiberPlugin {
  // We define how to get the instance of StatePlugin.logic from the PluginHost. It␣
↪is a lazy val, because we can't evaluate it until the plugin is binded to its host.
  lazy val sp = host[StatePlugin].logic.get

  val logic = during build new Area {
    // Generate the increment hardware
    sp.signal := sp.signal + 1
  }
}

class TopLevel extends Component {
  val sub = new SubComponent()

  // Here we create plugins and embed them in sub.host
  new DriverPlugin().setHost(sub.host)
  new StatePlugin().setHost(sub.host)
}
```

Such TopLevel would generate the following Verilog code :

```
module TopLevel (
  input  wire          clk,
  input  wire          reset
);


  SubComponent sub (
    .clk   (clk  ), //i
    .reset (reset)  //i
  );

endmodule

module SubComponent (
  input  wire          clk,
  input  wire          reset
);

  reg        [31:0]   StatePlugin_logic_signal; //Created by StatePlugin

  always @(posedge clk) begin
    StatePlugin_logic_signal <= (StatePlugin_logic_signal + 32'h00000001); //
↪incremented by DriverPlugin
  end
endmodule
```

Note each “during build” fork an elaboration thread, the DriverPlugin.logic thread execution will be blocked on the “sp” evaluation until the StatePlugin.logic execution is done.

Interlocking / Ordering

Plugins can interlock each others using Retainer instances. Each plugin instance has a built in lock which can be controlled using retain/release functions.

Here is an example based on the above *Simple example* but that time, the DriverPlugin will increment the StatePlugin.logic.signal by an amount set by other plugins (SetupPlugin in our case). And to ensure that the DriverPlugin doesn't generate the hardware too early, the SetupPlugin uses the DriverPlugin.retain/release functions.

```
import spinal.core._
import spinal.lib.misc.plugin._
import spinal.core.fiber._

class SubComponent extends Component {
  val host = new PluginHost()
}

class StatePlugin extends FiberPlugin {
  val logic = during build new Area {
    val signal = Reg(UInt(32 bits))
  }
}

class DriverPlugin extends FiberPlugin {
  // incrementBy will be set by others plugin at elaboration time
  var incrementBy = 0
  // retainer allows other plugins to create locks, on which this plugin will wait␣
↪before using incrementBy
  val retainer = Retainer()

  val logic = during build new Area {
    val sp = host[StatePlugin].logic.get
    retainer.await()

    // Generate the incrementer hardware
    sp.signal := sp.signal + incrementBy
  }
}

// Let's define a plugin which will modify the DriverPlugin.incrementBy variable␣
↪because letting it elaborate its hardware
class SetupPlugin extends FiberPlugin {
  // during setup { body } will spawn the body of code in the Fiber setup phase (it␣
↪is before the Fiber build phase)
  val logic = during setup new Area {
    // *** Setup phase code ***
    val dp = host[DriverPlugin]

    // Prevent the DriverPlugin from executing its build's body (until release() is␣
↪called)
    val lock = dp.retainer()
    // Wait until the fiber phase reached build phase
    awaitBuild()

    // *** Build phase code ***
    // Let's mutate DriverPlugin.incrementBy
    dp.incrementBy += 1
```

(continues on next page)

(continued from previous page)

```
    // Allows the DriverPlugin to execute its build's body
    lock.release()
  }
}

class TopLevel extends Component {
  val sub = new SubComponent()

  sub.host.asHostOf(
    new DriverPlugin(),
    new StatePlugin(),
    new SetupPlugin(),
    new SetupPlugin() //Let's add a second SetupPlugin, because we can
  )
}
```

Here is the generated verilog

```
module TopLevel (
  input  wire          clk,
  input  wire          reset
);


  SubComponent sub (
    .clk   (clk  ), //i
    .reset (reset)  //i
  );

endmodule

module SubComponent (
  input  wire          clk,
  input  wire          reset
);

  reg        [31:0]   StatePlugin_logic_signal;

  always @(posedge clk) begin
    StatePlugin_logic_signal <= (StatePlugin_logic_signal + 32'h00000002); // + 2 as␣
↪we have two SetupPlugin
  end
endmodule
```

Clearly, those examples are overkilled for what they do, the idea in general is more about :

- Negociate / create interfaces between plugins (ex jump / flush ports)
- Schedule the elaboration (ex decode / dispatch specification)
- Provide a distributed framework which can scale up (minimal hardcoding)

CHAPTER

ELEVEN

SIMULATION

As always, you can use your standard simulation tools to simulate the VHDL/Verilog generated by SpinalHDL. However, since SpinalHDL 1.0.0, the language integrates an API to write testbenches and test your hardware directly in Scala. This API provides the capabilities to read and write the DUT signals, fork and join simulation processes, sleep and wait until a given condition is reached. Therefore, using SpinalHDL's simulation API, it is easy to integrate testbenches with the most common Scala unit-test frameworks.

To be able to simulate user-defined components, SpinalHDL uses external HDL simulators as backend. Currently, four simulators are supported:

- Verilator
- GHDL
- Icarus Verilog
- VCS **(experimental, since SpinalHDL 1.7.0)**
- XSim **(experimental, since SpinalHDL 1.7.0)**

With external HDL simulators it is possible to directly test the generated HDL sources without increasing the SpinalHDL codebase complexity.

11.1 SBT setup for simulation

To enable SpinalSim, the following lines have to be added in your build.sbt file :

```
fork := true
```

Also the following imports have to be added in testbenches sources :

```
import spinal.core._
import spinal.core.sim._
```

Also, if you need to use gmake instead of make (ex OpenBSD) you can set the SPINAL_MAKE_CMD environnement variable to "gmake"

11.1.1 Backend-dependent installation instructions

Setup and installation of GHDL

Note: If you installed the recommended oss-cad-suite during SpinalHDL *setup* you can skip the instructions below - but you need to activate the oss-cad-suite environment.

Even though GHDL is generally available in linux distributions package system, SpinalHDL depends on bugfixes of GHDL codebase that were added after the release of GHDL v0.37. Therefore it is reccomended to install GHDL

from source. The C++ library boost-interprocess, which is contained in the libboost-dev package in debian-like distributions, has to be installed too. boost-interprocess is required to generate the shared memory communication interface.

Linux

```
sudo apt-get install build-essential libboost-dev git
sudo apt-get install gnat # Ada compiler used to buid GHDL
git clone https://github.com/ghdl/ghdl.git
cd ghdl
mkdir build
cd build
../configure
make
sudo make install
```

Also the openjdk package that corresponds to your Java version has to be installed.

For more configuration options and Windows installation see https://ghdl.github.io/ghdl/getting.html

Setup and installation of Icarus Verilog

Note: If you installed the recommended oss-cad-suite during SpinalHDL *setup* you can skip the instructions below - but you need to activate the oss-cad-suite environment.

In most recent linux distributions, a recent version of Icarus Verilog is generally available through the package system. The C++ library boost-interprocess, which is contained in the libboost-dev package in debian-like distributions, has to be installed too. boost-interprocess is required to generate the shared memory communication interface.

Linux

```
sudo apt-get install build-essential libboost-dev iverilog
```

Also the openjdk package that corresponds to your Java version has to be installed. Refer to https://iverilog.fandom.com/wiki/Installation_Guide for more informations about Windows and installation from source.

VCS Simulation Configuration

Environment variable

You should have several environment variables defined before:

- `VCS_HOME`: The home path to your VCS installation.
- `VERDI_HOME`: The home path to your Verdi installation.
- Add `$VCS_HOME/bin` and `$VERDI_HOME/bin` to your `PATH`.

Prepend the following paths to your `LD_LIBRARY_PATH` to enable PLI features.

```
export LD_LIBRARY_PATH=$VERDI_HOME/share/PLI/VCS/LINUX64:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH=$VERDI_HOME/share/PLI/IUS/LINUX64:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH=$VERDI_HOME/share/PLI/lib/LINUX64:$LD_LIBRARY_PATH
```

(continues on next page)

(continued from previous page)

```
export LD_LIBRARY_PATH=$VERDI_HOME/share/PLI/Ius/LINUX64:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH=$VERDI_HOME/share/PLI/MODELSIM/LINUX64:$LD_LIBRARY_PATH
```

If you encounter the `Compilation of SharedMemIface.cpp failed` error, make sure that you have installed C++ boost library correctly. The header and library files path should be added to `CPLUS_INCLUDE_PATH`, `LIBRARY_PATH` and `LD_LIBRARY_PATH` respectively.

User defined environment setup

Sometimes a VCS environment setup file *synopsys_sim.setup* is required to run VCS simulation. Also you may want to run some scripts or code to setup the environment just before VCS starting compilation. You can do this by *withVCSSimSetup*.

```
val simConfig = SimConfig
  .withVCS
  .withVCSSimSetup(
    setupFile = "~/work/myproj/sim/synopsys_sim.setup",
    beforeAnalysis = () => { // this code block will be run before VCS analysis step.
      "pwd".!
      println("Hello, VCS")
    }
  )
```

This method will copy your own *synopsys_sim.setup* file to the VCS work directory under the *workspacePath* (default as *simWorkspace*) directory, and run your scripts.

VCS Flags

The VCS backend follows the three step compilation flow:

1. Analysis step: analysis the HDL model using `vlogan` and `vhdlan`.
2. Elaborate step: elaborate the model using `vcs` and generate the executable hardware model.
3. Simulation step: run the simulation.

In each step, user can pass some specific flags through `VCSFlags` to enable some features like SDF back-annotation or multi-threads.

`VCSFlags` takes three parameters,

Name	Type	Description
`compileFlags`	`List[String]`	Flags pass to `vlogan` or `vhdlan`.
`elaborateFlags`	`List[String]`	Flags pass to `vcs`.
`runFlags`	`List[String]`	Flags pass to executable hardware model.

For example, you pass the `-kdb` flags to both compilation step and elaboration step, for Verdi debugging,

```
val flags = VCSFlags(
    compileFlags = List("-kdb"),
    elaborateFlags = List("-kdb")
)

val config =
  SimConfig
    .withVCS(flags)
```

(continues on next page)

(continued from previous page)

```
    .withFSDBWave
    .workspacePath("tb")
    .compile(UIntAdder(8))

...
```

Waveform generation

VCS backend can generate three waveform format: `VCD`, `VPD` and `FSDB` (Verdi required).

You can enable them by the following methods of `SpinalSimConfig`,

Method	Description
`withWave`	Enable `VCD` waveform.
`withVPDWave`	Enable `VPD` waveform.
`withFSDBWave`	Enable `FSDB` waveform.

Also, you can control the wave trace depth by using `withWaveDepth(depth: Int)`.

Simulation with `Blackbox`

Sometimes, IP vendors will provide you with some design entites in Verilog/VHDL format and you want to integrate them into your SpinalHDL design. The integration can done by following two ways:

1. In a `Blackbox` definition, use `addRTLPath(path: String)` to assign a external Verilog/VHDL file to this blackbox.
2. Use the method `mergeRTLSource(fileName: String=null)` of `SpinalReport`.

Setup and installation of Verilator

Note: If you installed the recommended oss-cad-suite during SpinalHDL *setup* you can skip the instructions below - but you need to activate the oss-cad-suite environment.

SpinalSim + Verilator is supported on both Linux and Windows platforms.

It is recommended that v4.218 is the oldest Verilator version to use. While it maybe possible to use older verilator versions, some optional and Scala source dependent features that SpinalHDL can use (such as Verilog `$urandom` support) may not be supported by older Verilator versions and will cause an error when trying to simulate.

Ideally the latest v4.xxx and v5.xxx is well supported and bug reports should be opened with any issues you have.

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._
```

Linux

You will also need a recent version of Verilator installed :

```
sudo apt-get install git make autoconf g++ flex bison  # 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 v4.218   # Can use newer v4.228 and v5.xxx
autoconf              # Create ./configure script
./configure
make -j$(nproc)
sudo make install
echo "DONE"
```

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 environment variable points to the JDK installation folder (i.e.: `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 v4.218   # Can use newer v4.228 and v5.xxx
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.

11.2 Boot a simulation

11.2.1 Introduction

Below is an example hardware definition + testbench:

```
import spinal.core._

// Identity takes n bits in a and gives them back in z
class Identity(n: Int) extends Component {
  val io = new Bundle {
    val a = in Bits(n bits)
    val z = out Bits(n bits)
  }

  io.z := io.a
}
```

```
import spinal.core.sim._
```

(continues on next page)

(continued from previous page)

```
object TestIdentity extends App {
  // Use the component with n = 3 bits as "dut" (device under test)
  SimConfig.withWave.compile(new Identity(3)).doSim{ dut =>
    // For each number from 3'b000 to 3'b111 included
    for (a <- 0 to 7) {
      // Apply input
      dut.io.a #= a
      // Wait for a simulation time unit
      sleep(1)
      // Read output
      val z = dut.io.z.toInt
      // Check result
      assert(z == a, s"Got $z, expected $a")
    }
  }
}
```

11.2.2 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 (default format)
`withVcdWave`	Enable simulation wave capture (VCD text format)
`withFstWave`	Enable simulation wave capture (FST binary format)
`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
`withVerilator`	Use Verilator as simulation backend (default)
`withGhdl`	Use GHDL as simulation backend
`withIVerilog`	Use Icarus Verilog as simulation backend
`withVCS`	Use Synopsys VCS as simulation backend

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:

`doSim[(simName[, seed])]{dut => /* main stimulus code */}`
: Run the simulation until the main thread runs to completion and exits/returns. It will detect and report an error if the simulation gets fully stuck. As long as e.g. a clock is running the simulation can continue forever, it is therefore recommended to use `SimTimeout(cycles)` to limit the possible runtime.

`doSimUntilVoid[(simName[, seed])]{dut => ...}`
: Run the simulation until it is ended by calling either `simSuccess()` or `simFailure()`. The main stimulus thread can continue or exit early. As long as there are events to process, the simulation will continue. The simulation will report an error if it gets fully stuck.

The following testbench template will use the following toplevel :

```
class TopLevel extends Component {
   val counter = out(Reg(UInt(8 bits)) init (0))
   counter := counter + 1
}
```

Here is a template with many simulation configurations:

```
val spinalConfig = SpinalConfig(defaultClockDomainFrequency = FixedFrequency(10 MHz))

SimConfig
  .withConfig(spinalConfig)
  .withWave
  .allOptimisation
  .workspacePath("~/tmp")
  .compile(new TopLevel)
  .doSim { dut =>
    SimTimeout(1000)
    // Simulation code here
}
```

Here is a template where the simulation ends by completing the simulation main thread execution:

```
SimConfig.compile(new TopLevel).doSim { dut =>
  SimTimeout(1000)
  dut.clockDomain.forkStimulus(10)
  dut.clockDomain.waitSamplingWhere(dut.counter.toInt == 20)
  println("done")
}
```

Here is a template where the simulation ends by explicitly calling *simSuccess()*:

```
SimConfig.compile(new TopLevel).doSimUntilVoid{ dut =>
  SimTimeout(1000)
  dut.clockDomain.forkStimulus(10)
  fork {
    dut.clockDomain.waitSamplingWhere(dut.counter.toInt == 20)
    println("done")
    simSuccess()
  }
}
```

Note is it equivalent to:

```
SimConfig.compile(new TopLevel).doSim{ dut =>
  SimTimeout(1000)
  dut.clockDomain.forkStimulus(10)
  fork {
    dut.clockDomain.waitSamplingWhere(dut.counter.toInt == 20)
    println("done")
    simSuccess()
  }
  simThread.suspend() // Avoid the "doSim" completion
}
```

The location where the simulation files will be placed is defined by default in $WORKSPACE/$COMPILED.

- $WORKSPACE being by default `simWorkspace`, you can override it with the `SPINALSIM_WORKSPACE` environnement variable.
- $COMPILED being the name of the toplevel being simulated.
- The location of the wave file depend the backend used. For verilator it will be in the folder (`$WORKSPACE/$COMPILED/$TEST` by default).
- For the verilator backend, you can override the location of the test folder via the `SimConfig.setTestPath(path)` function.

- You can retrieve the location of the test path durring simulation by calling the currentTestPath() function.

11.2.3 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
}
```

11.2.4 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.

It is possible to make a simulation fail when it is too long, for instance because the test-bench is waiting for a condition which never occurs. To do so, call `SimTimeout(maxDuration)` where `maxDuration` is the time (in simulation units of time) after the which the simulation should be considered to have failed.

For instance, to make the simulation fail after 1000 times the duration of a clock cycle:

```
val period = 10
dut.clockDomain.forkStimulus(period)
SimTimeout(1000 * period)
```

11.2.5 Capturing wave for a given window before failure

In the case you have a very long simulation, and you don't want to capture the wave on all of it (too bug, too slow), you have mostly 2 ways to do it.

Either you know already at which `simTime` the simulation failed, in which case you can do the following in your testbench :

```
disableSimWave()
delayed(timeFromWhichIWantToCapture)(enableSimWave())
```

Or you can run a dual lock-step simulation, with one running a bit delayed from the the other one, and which will start recording the wave once the leading simulation had a failure.

To do this, you can use the DualSimTracer utility, with parameters for the compiled hardware, the window of time you want to capture before failure, and a seed.

Here is an example :

```
package spinaldoc.libraries.sim

import spinal.core._
import spinal.core.sim._
import spinal.lib.misc.test.DualSimTracer

class Toplevel extends Component{
  val counter = out(Reg(UInt(16 bits))) init(0)
  counter := counter + 1
```

(continues on next page)

(continued from previous page)

```
}

object Example extends App {
  val compiled = SimConfig.withFstWave.compile(new Toplevel())

  DualSimTracer(compiled, window = 10000, seed = 42){dut=>
    dut.clockDomain.forkStimulus(10)
    dut.clockDomain.onSamplings{
      val value = dut.counter.toInt

      if(value % 0x1000 == 0){
        println(f"Value=0x$value%x at ${simTime()}")
      }

      // Throw a simulation failure after 64K cycles
      if(value == 0xFFFF){
        simFailure()
      }
    }
  }
}
```

This will generate the following file structure :

- simWorkspace/Toplevel/explorer/stdout.log : stdout of the simulation which is ahead
- simWorkspace/Toplevel/tracer/stdout.log : stdout of the simulation doing the wave tracing
- simWorkspace/Toplevel/tracer.fst : Waveform of the failure

The scala terminal will show the explorer simulation stdout.

11.3 Accessing signals of the simulation

11.3.1 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`
`Data.randomize()`	Assign a random value to a SpinalHDL value.

```
dut.io.a #= 42
dut.io.a #= 42l
```

(continues on next page)

(continued from previous page)

```
dut.io.a #= BigInt("101010", 2)
dut.io.a #= BigInt("0123456789ABCDEF", 16)
println(dut.io.b.toInt)
```

11.3.2 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
  }

  def main(args: Array[String]) {
    SimConfig.compile {
      val dut = new TopLevel
      dut.counter.simPublic()     // Call simPublic() here
      dut
    }.doSim{dut =>
      dut.clockDomain.forkStimulus(10)

      for(i <- 0 to 3) {
        dut.clockDomain.waitSampling()
        println(dut.counter.toInt)
      }
    }
  }
}
```

11.3.3 Load and Store of Memory in Simulation

It is possible to modify the contents of `Mem` hardware interface components in simulation. The *data* argument should be a word-width value with the *address* being the word-address within.

There is no API to convert address and/or individual data bits into units other than the natural word size.

There is no API to mark any memory location with simulation *X* (undefined) state.

Syntax	Description
`Mem.getBigInt(address: Long): BigInt`	Read a word from simulator at the word-address.
`Mem.setBigInt(address: Long, data: BigInt)`	Write a word to simulator at the word-address.

Using this simple example using a memory:

```
case class MemoryExample() extends Component {
  val wordCount = 64
  val io = new Bundle {
    val address = in port UInt(log2Up(wordCount) bit)
    val i = in port Bits(8 bit)
    val o = out port Bits(8 bit)
    val we = in port Bool()
  }

  val mem = Mem(Bits(8 bit), wordCount=wordCount)
  io.o := mem(io.address)
  when(io.we) {
    mem(io.address) := io.i
  }
}
```

Setting up the simulation we make the memory accessible:

```
  SimConfig.withVcdWave.compile {
    val d = MemoryExample()
    // make memory accessible during simulation
    d.mem.simPublic()
    d
  }.doSim("example") { dut =>
```

We can read data during simulation, but have to take care that the data is already available (might be a cycle late due to simulation event ordering):

```
    // do a write
    dut.io.we #= true
    dut.io.address #= 10
    dut.io.i #= 0xaf
    dut.clockDomain.waitSampling(2)
    // check written data is there
    assert(dut.mem.getBigInt(10) == 0xaf)
```

And can write to memory like so:

```
    // set some data in memory
    dut.mem.setBigInt(15, 0xfe)
    // do a read to check if it's there
    dut.io.address #= 15
```

(continues on next page)

(continued from previous page)

```
    dut.clockDomain.waitSampling(1)
    assert(dut.io.o.toBigInt == 0xfe)
```

Care has to be taken that due to event ordering in simulation e.g. the read depicted above has to be delayed to when the value is actually available in the memory.

11.4 Clock domains

11.4.1 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

11.4.2 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([cycleCount])`	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([cycleCount])`	Same as `waitRisingEdge` but for the falling edge
`waitActiveEdge([cycleCount])`	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 `waitFallingEdgeWhere`, but for the falling edge
`waitActiveEdgeWhere(condition)`	Same as `waitRisingEdgeWhere`, but for the edge level specified by the `ClockDomainConfig`
`waitSamplingWhere(condition) : Boolean`	Wait until the clockdomain sampled and the given condition is true
`waitSamplingWhere(timeout)(condition) : Boolean`	Same as `waitSamplingWhere` defined above, but will never block more than timeout cycles. Return true if the exit condition came from the timeout

Warning: All the functionality of the wait API can only be called directly from inside a thread, and not from a callback executed via the Callback API.

11.4.3 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
`onSamplingWhile { callback : Boolean }`	Same as onSampling, but you can stop it (forever) by letting the callback returning false

11.4.4 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()
  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()
```

11.4.5 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)
```

11.5 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.

11.5.1 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()
```

11.5.2 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)
```

11.6 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*

11.7 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)

11.8 Simulator specific details

11.8.1 How SpinalHDL simulates the hardware with Verilator backend

1. Behind the scenes, SpinalHDL generates a Verilog equivalent hardware model of the DUT and then uses Verilator to convert it to a C++ cycle-accurate model.
2. The C++ model is compiled into a shared object (.so), which is bound to Scala via JNI-FFI.
3. The native Verilator API is abstracted by providing a simulation multi-threaded API.

Advantages:

- Since the Verilator backend uses a compiled C++ simulation model, the simulation speed is fast compared to most of the other commercial and free simulators.

Limitations:

- Verilator accepts only synthesizable Verilog/System Verilog code. Therefore special care has to be taken when simulating Verilog blackbox components that may have non-synthesizable statements.
- VHDL blackboxes cannot be simulated.
- The simulation boot process is slow due to the necessity to compile and link the generated C++ model. Some support to incrementally compile and link exists which can provide speedups for subsequent simulations after building the first.

11.8.2 How SpinalHDL simulates the hardware with GHDL/Icarus Verilog backend

1. Depending on the chosen simulator, SpinalHDL generates a Verilog or VHDL hardware model of the DUT.
2. The HDL model is loaded in the simulator.
3. The communication between the simulation and the JVM is established through shared memory. The commands are issued to the simulator using VPI.

Advantages:

- Both GHDL and Icarus Verilog can accept non-synthesizable HDL code.
- The simulation boot process is quite faster compared to Verilator.

Limitations:

- GHDL accepts VHDL code only. Therefore only VHDL blackboxes can be used with this simulator.
- Icarus Verilog accepts Verilog code only. Therefore only Verilog blackboxes can be used with this simulator.
- The simulation speed is around one order of magnitude slower compared to Verilator.

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.

11.8.3 How SpinalHDL simulates the hardware with Synopsys VCS backend

1. SpinalHDL generates a Verilog/VHDL (depended on your choice) hardware model of the DUT.
2. The HDL model is loaded in the simulator.
3. The communication between the simulation and the JVM is established through shared memory. The commands are issued to the simulator using VPI.

Advantages:

- Support all language features of SystemVerilog/Verilog/VHDL.
- Support encrypted IP.
- Support FSDB wave format dump.
- High Performance of both compilation and simulation.

Limitations:

- Synopsys VCS is a **commercial** simulation tool. It is close source and not free. You have to own the licenses to **legally** use it.

Before using VCS as the simulation backend, make sure that you have checked your system environment as *VCS environment*.

11.8.4 How SpinalHDL simulates the hardware with Xilinx XSim backend

1. SpinalHDL generates a Verilog/VHDL (depended on your choice) hardware model of the DUT.
2. The HDL model is loaded in the simulator.
3. The communication between the simulation and the JVM is established through shared memory. The commands are issued to the simulator using XSI.

Advantages:

- Support Xilinx built-in primitives and cores.

Limitations:

- Xilinx XSim is a **commercial** tool installed with Vivado. It is closed source and subject to licensing terms to use. You have to own the licenses to **legally** use it.
- Vivado versions prior to 2019.1 do not work properly.

Before using XSim as the simulation backend, make sure that you have done following steps. 1. Define VIVADO_HOME environment variable to specify where your vivado located. ex *export VIVADO_HOME=/d/Xilinx/Vivado/2022.1* (under MSYS2). 2. Make sure two vivado path is inside the PATH. For Windows MSYS2 user, run shell command like *export PATH=$PATH:$VIVADO_HOME/bin:$VIVADO_HOME/lib/win64.o*. For Linux user just source the Vivado's settings64.sh file located at *VIVADO_HOME*.

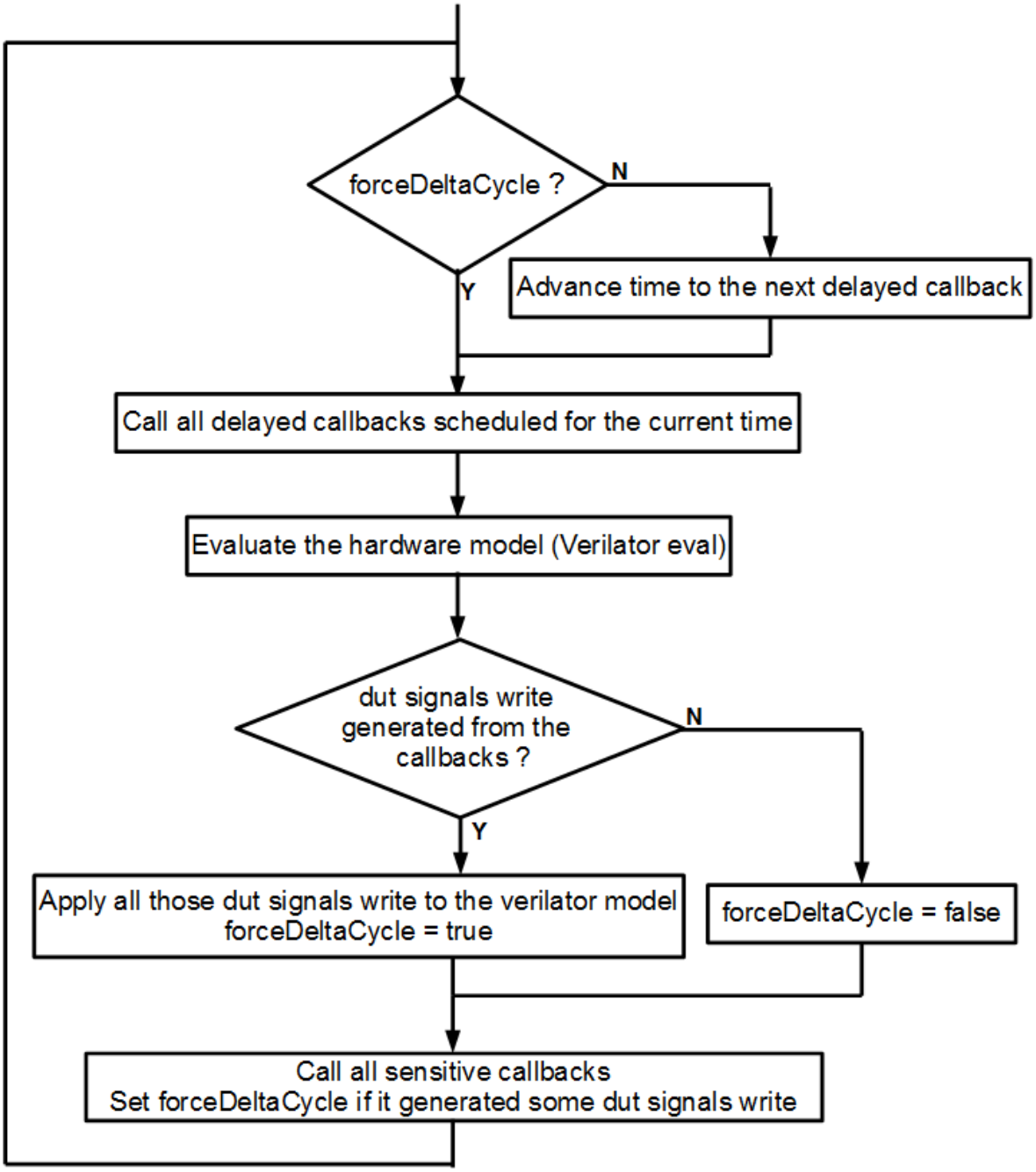
11.8.5 Performance

When a high-performance simulation is required, Verilator should be used as a backend. On a little SoC like Murax, an Intel® Core™ i7-4720HQ is capable of simulating 1.2 million clock cycles per second. However, when the DUT is simple and a maximum of few thousands clock cycles have to be simulated, using GHDL or Icarus Verilog could yield a better result, due to their lower simulation loading overhead.

11.9 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.

11.10 Examples

11.10.1 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.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
```

(continues on next page)

(continued from previous page)

```
        dut.io.c #= c
        sleep(1) // Sleep 1 simulation timestep
        assert(dut.io.result.toInt == ((a + b - c) & 0xFF))
        idx += 1
      }
    }
  }
}
```

11.10.2 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.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",withReset = false)
      )
    )

    // 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()
        sleep(0)
```

(continues on next page)

(continued from previous page)

```
        // Do the resets.
        dut.pushClock.assertReset()
        sleep(10)
        dut.pushClock.deassertReset()
        sleep(1)

        // Forever, randomly toggle one of the clocks.
        // 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()
      }
    }
  }
}
```

11.10.3 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.core._
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()
      }
    }
  }
}
```

11.10.4 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.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
      }
    }
  }
}
```

11.10.5 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)
```

(continues on next page)

(continued from previous page)

```
    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)
  }
}
```

11.10.6 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.
    }
  }
}
```

CHAPTER

TWELVE

FORMAL VERIFICATION

12.1 General

SpinalHDL allows to generate a subset of the SystemVerilog Assertions (SVA). Mostly assert, assume, cover and a few others.

In addition it provide a formal verification backend which allows to directly run the formal verification in the open-source Symbi-Yosys toolchain.

12.2 Formal backend

You can run the formal verification of a component via:

```
import spinal.core.formal._
FormalConfig.withBMC(15).doVerify(new Component {
    // Toplevel to verify
})
```

Currently, 3 modes are supported :

- withBMC(depth)
- withProve(depth)
- withCover(depth)

12.3 Installing requirements

To install the Symbi-Yosys, you have a few options. You can fetch a precompiled package at:

- https://github.com/YosysHQ/oss-cad-suite-build/releases
- https://github.com/YosysHQ/fpga-toolchain/releases (EOL - superseded by oss-cad-suite)

Or you can compile things from scratch :

- https://symbiyosys.readthedocs.io/en/latest/install.html

12.4 Example

12.4.1 External assertions

Here is an example of a simple counter and the corresponding formal testbench.

```
import spinal.core._

//Here is our DUT
class LimitedCounter extends Component {
  //The value register will always be between [2:10]
  val value = Reg(UInt(4 bits)) init(2)
  when(value < 10) {
    value := value + 1
  }
}

object LimitedCounterFormal extends App {
  // import utilities to run the formal verification, but also some utilities to␣
↪describe formal stuff
  import spinal.core.formal._

  // Here we run a formal verification which will explore the state space up to 15␣
↪cycles to find an assertion failure
  FormalConfig.withBMC(15).doVerify(new Component {
    // Instantiate our LimitedCounter DUT as a FormalDut, which ensure that all the␣
↪outputs of the dut are:
    // - directly and indirectly driven (no latch / no floating wire)
    // - allows the current toplevel to read every signal across the hierarchy
    val dut = FormalDut(new LimitedCounter())

    // Ensure that the state space start with a proper reset
    assumeInitial(ClockDomain.current.isResetActive)

    // Check a few things
    assert(dut.value >= 2)
    assert(dut.value <= 10)
  })
}
```

12.4.2 Internal assertions

If you want you can embed formal statements directly into the DUT:

```
class LimitedCounterEmbedded extends Component {
  val value = Reg(UInt(4 bits)) init(2)
  when(value < 10) {
    value := value + 1
  }

  // That code block will not be in the SpinalVerilog netlist by default. (would need␣
↪to enable SpinalConfig().includeFormal. ...
  GenerationFlags.formal {
    assert(value >= 2)
    assert(value <= 10)
```

(continues on next page)

(continued from previous page)

```
  }
}

object LimitedCounterEmbeddedFormal extends App {
  import spinal.core.formal._

  FormalConfig.withBMC(15).doVerify(new Component {
    val dut = FormalDut(new LimitedCounterEmbedded())
    assumeInitial(ClockDomain.current.isResetActive)
  })
}
```

12.4.3 External stimulus

If your DUT has inputs, you need to drive them from the testbench. You can use all the regular hardware statements to do it, but you can also use the formal *anyseq*, *anyconst*, *allseq*, *allconst* statement:

```
class LimitedCounterInc extends Component {
  //Only increment the value when the inc input is set
  val inc = in Bool()
  val value = Reg(UInt(4 bits)) init(2)
  when(inc && value < 10) {
    value := value + 1
  }
}

object LimitedCounterIncFormal extends App {
  import spinal.core.formal._

  FormalConfig.withBMC(15).doVerify(new Component {
    val dut = FormalDut(new LimitedCounterInc())
    assumeInitial(ClockDomain.current.isResetActive)
    assert(dut.value >= 2)
    assert(dut.value <= 10)

    // Drive dut.inc with random values
    anyseq(dut.inc)
  })
}
```

12.4.4 More assertions / past

For instance we can check that the value is counting up (if not already at 10):

```
FormalConfig.withBMC(15).doVerify(new Component {
  val dut = FormalDut(new LimitedCounter())
  assumeInitial(ClockDomain.current.isResetActive)

  // Check that the value is incrementing.
  // hasPast is used to ensure that the past(dut.value) had at least one sampling out␣
↪of reset
  when(pastValid() && past(dut.value) =/= 10) {
    assert(dut.value === past(dut.value) + 1)
```

(continues on next page)

(continued from previous page)

```
  }
})
```

12.4.5 Assuming memory content

Here is an example where we want to prevent the value 1 from ever being present in a memory :

```
class DutWithRam extends Component {
  val ram = Mem.fill(4)(UInt(8 bits))
  val write = slave(ram.writePort)
  val read = slave(ram.readAsyncPort)
}

object FormalRam extends App {
  import spinal.core.formal._

  FormalConfig.withBMC(15).doVerify(new Component {
    val dut = FormalDut(new DutWithRam())
    assumeInitial(ClockDomain.current.isResetActive)

    // assume that no word in the ram has the value 1
    for(i <- 0 until dut.ram.wordCount) {
      assumeInitial(dut.ram(i) =/= 1)
    }

    // Allow the write anything but value 1 in the ram
    anyseq(dut.write)
    clockDomain.withoutReset() { //As the memory write can occur during reset, we␣
↪need to ensure the assume apply there too
      assume(dut.write.data =/= 1)
    }

    // Check that no word in the ram is set to 1
    anyseq(dut.read.address)
    assert(dut.read.data =/= 1)
  })
}
```

12.5 Utilities and primitives

12.5.1 Assertions / clock / reset

Assertions are always clocked and disabled during resets. This also apply for assumes and covers.

If you want to keep your assertion enabled during reset you can do:

```
ClockDomain.current.withoutReset() {
  assert(wuff === 0)
}
```

12.5.2 Specifying the initial value of a signal

For instance, for the reset signal of the current clockdomain (usefull at the top)

```
ClockDomain.current.readResetWire initial(False)
```

12.5.3 Specifying a initial assumption

```
assumeInitial(clockDomain.isResetActive)
```

12.5.4 Memory content (Mem)

If you have a Mem in your design, and you want to check its content, you can do it the following ways :

```
// Manual access
for(i <- 0 until dut.ram.wordCount) {
  assumeInitial(dut.ram(i) =/= X) //No occurence of the word X
}

assumeInitial(!dut.ram.formalContains(X)) //No occurence of the word X

assumeInitial(dut.ram.formalCount(X) === 1) //only one occurence of the word X
```

12.5.5 Specifying assertion in the reset scope

```
ClockDomain.current.duringReset {
  assume(rawrrr === 0)
  assume(wuff === 3)
}
```

12.5.6 Formal primitives

Syntax	Returns	Description
`assert(Bool)`		
`assume(Bool)`		
`cover(Bool)`		
`past(that : T, delay : Int)` `past(that : T)`	T	Return `that` delayed by `delay` cycles. (default 1 cycle)
`rose(that : Bool)`	Bool	Return True when `that` transitioned from False to True
`fell(that : Bool)`	Bool	Return True when `that` transitioned from True to False
`changed(that : Bool)`	Bool	Return True when `that` current value changed between compared to the last cycle
`stable(that : Bool)`	Bool	Return True when `that` current value didn't changed between compared to the last cycle
`initstate()`	Bool	Return True the first cycle
`pastValid()`	Bool	Returns True when the past value is valid (False on the first cycle). Recommended to be used with each application of `past`, `rose`, `fell`, `changed` and `stable`.
`pastValidAfterReset()`	Bool	Simliar to `pastValid`, where only difference is that this would take reset into account. Can be understood as `pastValid & past(!reset)`.

Note that you can use the init statement on past:

```
when(past(enable) init(False)) { ... }
```

12.6 Limitations

There is no support for unclocked assertions. But their usage in third party formal verification examples seems mostly code style related.

12.7 Naming polices

All formal validation related functions return Area or Composite (preferred), and naming as formalXXXX. `formalContext` can be used to create formal related logic, there could be `formalAsserts`, `formalAssumes` and `formalCovers` in it.

12.7.1 For Component

The minimum required assertions internally in a `Component` for "prove" can be named as `formalAsserts`.

12.7.2 For interfaces implement IMasterSlave

There could be functions in name `formalAssertsMaster`, `formalAssertsSlave`, `formalAssumesMaster`, `formalAssumesSlave` or `formalCovers`. Master/Slave are target interface type, so that `formalAssertsMaster` can be understand as "formal verfication assertions for master interface".

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 bits)
  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 bits)
  val PRDATA = Bits(config.dataWidth bits)
  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:

```
case class Apb3User(apbConfig: Apb3Config) extends Component {
  val io = new Bundle {
    val apb = slave(Apb3(apbConfig))
  }

  io.apb.PREADY := True
  when(io.apb.PSEL(0) && io.apb.PENABLE) {
    // ...
  }

  io.apb.PRDATA := B(0)
}

object Apb3User extends App {
  val config = Apb3Config(
    addressWidth = 16,
    dataWidth = 32,
    selWidth = 1,
    useSlaveError = false
  )
  SpinalVerilog(Apb3User(config))
}
```

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.

```
case 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
```

(continues on next page)

(continued from previous page)

```
  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
    c \= (a & b) | (a & c) | (b & c);    //variable assignment
  }
}

object CarryAdderProject extends App {
  SpinalVhdl(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
    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.

```
case 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 until sourceCount) {
    sum \= sum + io.sources(i)
  }
  io.result := sum
```

(continues on next page)

(continued from previous page)

```
}
```

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.

```
case 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 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`:

```
case class PLL() extends BlackBox {
  val io = new Bundle {
    val clkIn = in Bool()
    val clkOut = out Bool()
    val isLocked = out Bool()
  }
  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:

```
case 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 {
    // Instantiate 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
  }
}
```

13.1.6 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

```
case 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 scaled(value : UInt,by : Float): UInt = value * U((255*by).toInt,8 bits) >> 8

  val gray = RegNext(
    scaled(io.r,0.3f) +
      scaled(io.g,0.4f) +
      scaled(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
  }
}
```

13.1.7 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
sinFiltered	out	SInt(resolutionWidth bits)	Output which plays the filtered version of the sine

So let's define the `Component`:

```
case class SineRom(resolutionWidth: Int, sampleCount: Int) extends Component {
  val io = new Bundle {
    val sin = out SInt(resolutionWidth bits)
    val sinFiltered = out SInt(resolutionWidth bits)
```

(continues on next page)

(continued from previous page)

```
  }
...
```

To play the sine wave on the `sin` output, you can define a ROM which contain all samples of a sine period (it could be just a quarter, but let's do things the most simple way). Then you can read that ROM with an phase counter and this will generate your sine wave.

```
  // Calculate values for the lookup table
  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 `sinFiltered`, you can for example use a first order low pass filter implementation:

```
  io.sinFiltered := RegNext(io.sinFiltered  - (io.sinFiltered  >> 5) + (io.sin >> 5))␣
↪init 0
```

Here is the complete code:

```
case class SineRom(resolutionWidth: Int, sampleCount: Int) extends Component {
  val io = new Bundle {
    val sin = out SInt(resolutionWidth bits)
    val sinFiltered = out SInt(resolutionWidth bits)
  }

  // Calculate values for the lookup table
  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)

  io.sinFiltered := RegNext(io.sinFiltered  - (io.sinFiltered  >> 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

  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
clockDivider-Width	Int	Number of bits that the clock divider has
preSampling-Size	Int	Number of samples to drop at the beginning of the sampling window
sampling-Size	Int	Number of samples use at the middle of the window to get the filtered RXD value
postSampling-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. If so we can skip resetting counters in a few places.

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 interface

Let's define a UART interface bundle without flow control.

```
case class Uart() extends Bundle with IMasterSlave {
  val txd = Bool()
  val rxd = Bool()

  override def asMaster(): Unit = {
    out(txd)
    in(rxd)
  }
}
```

UART configuration enums

Let's define parity and stop bit enumerations.

```
object UartParityType extends SpinalEnum(binarySequential) {
  val NONE, EVEN, ODD = newElement()
}

object UartStopType extends SpinalEnum(binarySequential) {
  val ONE, TWO = newElement()
  def toBitCount(that: C): UInt = (that === ONE) ? U"0" | U"1"
}
```

UartCtrl configuration Bundles

Let's define bundles that will be used as IO elements to setup `UartCtrl`.

```
case class UartCtrlFrameConfig(g: UartCtrlGenerics) extends Bundle {
  val dataLength = UInt(log2Up(g.dataWidthMax) bits) //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 bits) //see UartCtrlGenerics.
```

(continues on next page)

(continued from previous page)

```
↪clockDividerWidth for calculation

  def setClockDivider(baudrate: Double, clkFrequency: HertzNumber = ClockDomain.
↪current.frequency.getValue): Unit = {
    clockDivider := (clkFrequency.toDouble / 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 bits))
    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)
```

(continues on next page)

(continued from previous page)

```
    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) bits))
    def reset() = value := 0
    ..
  }

  val stateMachine = new Area {
    import UartCtrlTxState._

    val state = RegInit(IDLE)
    val parity = Reg(Bool())
    val txd = True
    ..
    switch(state) {
      ..
    }
  }

  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 bits)
    val txd          = out Bool()
  }

  // Provide one clockDivider.tick each rxSamplePerBit pulse of io.samplingTick
  // Used by the stateMachine as a baudrate 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(log2Up(Math.max(dataWidthMax, 2)) bits))
    def reset(): Unit = value := 0
```

(continues on next page)

(continued from previous page)

```
    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){
          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 === UartStopType.toBitCount(io.configFrame.stop)) {
            state := io.write.valid ? START | IDLE
```

(continues on next page)

(continued from previous page)

```
          }
        }
      }
    }
  }

  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 bits))
    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
```

(continues on next page)

(continued from previous page)

```
  // reset() can be called to recenter the counter over a start bit.
  val bitTimer = new Area {
    val counter = Reg(UInt(log2Up(rxSamplePerBit) bits))
    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) bits))
    def reset() = value := 0
    ...
  }

  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 bits)
    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 synchronizer = BufferCC(io.rxd)
    val samples     = spinal.lib.History(that=synchronizer, 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) bits))
    def reset(): Unit = counter := preSamplingSize + (samplingSize - 1) / 2 - 1
```

(continues on next page)

(continued from previous page)

```
    val tick = False
    when(sampler.tick) {
      counter := counter - 1
      tick := counter === 0
    }
  }

  // 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(log2Up(Math.max(dataWidthMax, 2)) bits))
    def reset(): Unit = 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) {
          state := START
          bitTimer.reset()
          bitCounter.reset()
        }
      }
      is(START) {
        when(bitTimer.tick) {
          state := DATA
          bitCounter.reset()
          parity := io.configFrame.parity === UartParityType.ODD
          when(sampler.value) {
            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) {
```

(continues on next page)

(continued from previous page)

```
              state := STOP
            } otherwise {
              state := PARITY
            }
          }
        }
      }
      is(PARITY) {
        when(bitTimer.tick) {
          state := STOP
          bitCounter.reset()
          when(parity =/= sampler.value) {
            state := IDLE
          }
        }
      }
      is(STOP) {
        when(bitTimer.tick) {
          when(!sampler.value) {
            state := IDLE
          }.elsewhen(bitCounter.value === UartStopType.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 bits)))
    val read   = master(Flow(Bits(g.dataWidthMax bits)))
    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) {
```

(continues on next page)

(continued from previous page)

```
      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
  io.uart.rxd <> rx.io.rxd
}
```

To make it easier to use the UART with fixed settings, we introduce an companion object for `UartCtrl`. It allows us to provide additional ways of instantiating a UartCtrl component with different sets of parameters. Here we define a `UartCtrlInitConfig` holding the settings for a component that is not runtime configurable. Note that it is still possible to instantiate the UartCtrl manually like all other components, which one would do if a runtime-configurable UART is needed (via `val uart = new UartCtrl()`).

```
case class UartCtrlInitConfig(baudrate: Int = 0,
                              dataLength: Int = 1,
                              parity: UartParityType.E = null,
                              stop: UartStopType.E = null
                             ) {
  require(dataLength >= 1)
  def initReg(reg : UartCtrlConfig): Unit = {
    require(reg.isReg)
    if(baudrate != 0) reg.clockDivider init((ClockDomain.current.frequency.getValue /␣
↪baudrate / reg.g.rxSamplePerBit).toInt-1)
    if(dataLength != 1) reg.frame.dataLength init (dataLength - 1)
    if(parity != null) reg.frame.parity init parity
    if(stop != null) reg.frame.stop init stop
  }
}

object UartCtrl {
  def apply(config: UartCtrlInitConfig, readonly: Boolean = false): UartCtrl = {
    val uartCtrl = new UartCtrl()
    uartCtrl.io.config.setClockDivider(config.baudrate)
    uartCtrl.io.config.frame.dataLength := config.dataLength - 1
    uartCtrl.io.config.frame.parity := config.parity
    uartCtrl.io.config.frame.stop := config.stop
    if (readonly) {
      uartCtrl.io.write.valid := False
      uartCtrl.io.write.payload := B(0)
    }
    uartCtrl
  }
}
```

Simple usage

To synthesize a `UartCtrl` as `115200-N-8-1`:

```
  val uartCtrl = UartCtrl(
    config=UartCtrlInitConfig(
      baudrate = 115200,
      dataLength = 8,
      parity = UartParityType.NONE,
      stop = UartStopType.ONE
    )
  )
```

If you are using `txd` pin only, add:

```
  uartCtrl.io.uart.rxd := True
  io.tx := uartCtrl.io.uart.txd
```

On the contrary, if you are using `rxd` pin only:

```
  val uartCtrl = UartCtrl(
    config = UartCtrlInitConfig(
      baudrate = 115200,
      dataLength = 8,
      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.

```
case class UartCtrlUsageExample() extends Component{
  val io = new Bundle{
    val uart = master(Uart())
    val switches = in Bits(8 bits)
    val leds = out Bits(8 bits)
  }

  val uartCtrl = new UartCtrl()
  // set config manually to show that this is still OK
  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

  //Assign io.led with a register loaded each time a byte is received
  io.leds := uartCtrl.io.read.toReg()
```

(continues on next page)

(continued from previous page)

```
  //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.switches
  write >-> uartCtrl.io.write
}

object UartCtrlUsageExample extends App {
  SpinalConfig(
    defaultClockDomainFrequency = FixedFrequency(100 MHz)
  ).generateVhdl(UartCtrlUsageExample())
}
```

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 queuedReads = uartCtrl.io.read.toStream.queue(16)
```

If you want to add a queue on the write interface and do some flow control:

```
  val writeCmd = Stream(Bits(8 bits))
  val stopIt = Bool()
  writeCmd.queue(16).haltWhen(stopIt) >> uartCtrl.io.write
```

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.switches
  write.last := True
  write.stage().insertHeader(0x55).toStreamOfFragment >> 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 bits)
  val g = UInt(c.gWidth bits)
  val b = UInt(c.bWidth bits)
}
```

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 bits)
  val colorEnd   = UInt(timingsWidth bits)
  val syncStart  = UInt(timingsWidth bits)
  val syncEnd    = UInt(timingsWidth bits)
}

case class VgaTimings(timingsWidth: Int) extends Bundle {
  val h = VgaTimingsHV(timingsWidth)
  val v = VgaTimingsHV(timingsWidth)
}
```

Then we can 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 bits)
  val colorEnd   = UInt(timingsWidth bits)
  val syncStart  = UInt(timingsWidth bits)
  val syncEnd    = UInt(timingsWidth bits)
}

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.

```
case 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.

```
case class VgaCtrl(rgbConfig: RgbConfig, timingsWidth: Int = 12) extends Component {
...
  case class HVArea(timingsHV: VgaTimingsHV, enable: Bool) extends Area {
    val counter = Reg(UInt(timingsWidth bits)) 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.

```
case class VgaCtrl(rgbConfig: RgbConfig, timingsWidth: Int = 12) extends Component {
...
  val colorEn = h.colorEn && v.colorEn
  io.pixels.ready := colorEn
  io.error := colorEn && ! io.pixels.valid

  io.frameStart := v.syncEnd

  io.vga.hSync := h.sync
  io.vga.vSync := v.sync
  io.vga.colorEn := colorEn
```

(continues on next page)

(continued from previous page)

```
  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.

```
case 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 explain how JTAG works. A good tutorial can be found 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 bits)
    val keys    = in  Bits(4 bits)
    val leds    = out Bits(8 bits)
  }

  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 with JtagTapAccess {
  val fsm = new JtagFsm(jtag)
  val instruction = Reg(Bits(instructionWidth bits))
  val instructionShift = Reg(Bits(instructionWidth bits))
  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
    }
  }
}
```

Note: Ignore the reference to *with JTagTapAccess* for now, it will be explained further down.

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.C
  def getInstruction(): Bits
  def setInstruction(value: Bits) : Unit
}
```

Then let the `JtagTap` implement this abstract interface:

Listing 1: Additions to `class JtagTap`

```
class JtagTap(val jtag: Jtag, ...) extends Area with JtagTapAccess{
...
  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.C = 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 allows 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 bits))

  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, cleanUpdate: Boolean, readable:␣
↪Boolean)(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 .

Listing 2: Additions to `trait JtagTapAccess`

```
trait JtagTapAccess {
...
  def idcode(value: Bits)(instructionId: Bits) =
    new JtagInstructionIdcode(value)(this, instructionId)

  def read[T <: Data](data: T)(instructionId: Bits)    =
```

(continues on next page)

(continued from previous page)

```
    new JtagInstructionRead(data)(this, instructionId)

  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 bits)
    val keys    = in  Bits(4 bits)
    val leds    = out Bits(8 bits)
  }

  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)
}
// end SimpleJtagTap
```

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

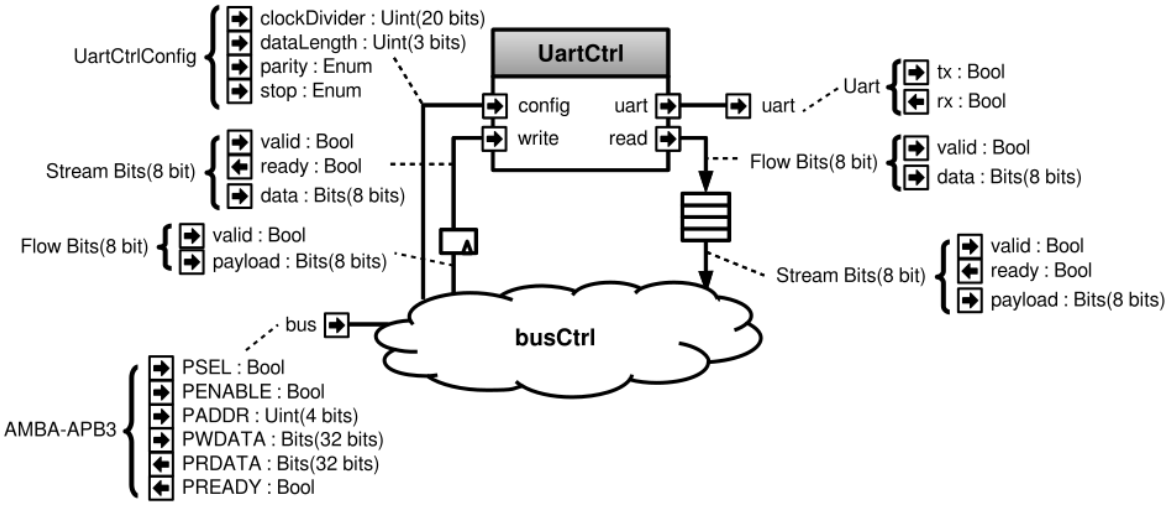
Implementation

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:




```
case class Apb3UartCtrl(uartCtrlConfig: UartCtrlGenerics, rxFifoDepth: Int) extends␣
↪Component {
  val io = new Bundle{
    val bus =  slave(Apb3(Apb3UartCtrl.getApb3Config))
    val uart = master(Uart())
  }
```

(continues on next page)

(continued from previous page)

```
  // Instantiate 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␣
↪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.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 Slots

Introduction

Let's say you have some hardware which has to keep track of multiple similar ongoing activities, you may want to implement an array of "slots" to do so. This example show how to do it using Area, OHMasking.first, onMask and reader.

Implementation

This implementation avoid the use of Vec. Instead, it use Area which allow to mix signal, registers and logic definitions in each slot.

Note that the *reader* API is for SpinalHDL version comming after 1.9.1

```
package spinaldoc.examples.advanced

import spinal.core._
import spinal.lib._
import scala.language.postfixOps

case class SlotsDemo(slotsCount : Int) extends Component {
  //...


  //Create the hardware for each slot
  //Note each slot is an Area, not a Bundle
  val slots = for(i <- 0 until slotsCount) yield new Area{
    //Because the slot is an Area, we can define mix signal, registers, logic␣
↪definitions
    //Here are the registers for each slots
    val valid = RegInit(False)
    val address = Reg(UInt(8 bits))
    val age = Reg(UInt(16 bits)) //Will count since how many cycles the slot is valid

    //Here is some hardware behaviour for each slots
    //Implement the age logic
    when(valid){
      age := age + 1
    }

    //removeIt will be used as a slot interface later on
    val removeIt = False
    when(removeIt){
      valid := False
    }
  }

  //Logic to allocate a new slot
  val insert = new Area{
    val cmd = Stream(UInt(8 bits)) //interface to issue requests
    val free = slots.map(!_.valid)
```

(continues on next page)

(continued from previous page)

```
    val freeOh = OHMasking.first(free) //Get the first free slot (on hot mask)
    cmd.ready := free.orR //Only allow cmd when there is a free slot
    when(cmd.fire){
      //slots.onMask(freeOh)(code) will execute the code for each slot where the␣
↪corresponding freeOh bit is set
      slots.onMask(freeOh){slot =>
        slot.valid := True
        slot.address := cmd.payload
        slot.age := 0
      }
    }
  }

  //Logic to remove the slots which match a given address (assuming there is not more␣
↪than one match)
  val remove = new Area{
    val cmd = Flow(UInt(8 bits))//interface to issue requests
    val oh = slots.map(s => s.valid && s.address === cmd.payload) //oh meaning "one␣
↪hot"
    when(cmd.fire){
      slots.onMask(oh){ slot =>
        slot.removeIt := True
      }
    }

    val reader = slots.reader(oh) //Create a facility to read the slots using "oh" as␣
↪index
    val age = reader(_.age) //Age of the slot which is selected by "oh"
  }

  //...
}

object SlotsDemo extends App {
  SpinalVerilog(SlotsDemo(4))
}
```

In practice

For instance, this kind of slot pattern is used in Tilelink coherency hub to keep track of all ongoing memory probes in flight:

https://github.com/SpinalHDL/SpinalHDL/blob/008c73f1ce18e294f137efe7a1442bd3f8fa2ee0/lib/src/main/scala/spinal/lib/bus/tilelink/coherent/Hub.scala#L376

As well in the DRAM / SDR / DDR memory controller to implement the handeling of multiple memory transactions at once (having multiple precharge / active / read / write running at the same time to improve performances) :

https://github.com/SpinalHDL/SpinalHDL/blob/1edba1890b5f629b28e5171b3c449155337d2548/lib/src/main/scala/spinal/lib/memory/sdram/xdr/Tasker.scala#L202

As well in the NaxRiscv (out of order CPU) load-store-unit to handle the store-queue / load-queue hardware (a bit too scary to show here in the doc XD)

13.3.5 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
```

(continues on next page)

(continued from previous page)

```
}
```

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 {
...
  // 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 {
    // Offset 0 => clear/tick masks + bus
    val ticksEnable = busCtrl.createReadAndWrite(Bits(ticks.length bits), baseAddress␣
↪+ 0,0) init(0)
    val clearsEnable = busCtrl.createReadAndWrite(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

    // Offset 4 => read/write limit (+ auto clear)
    busCtrl.driveAndRead(io.limit, baseAddress + 4)
    busClearing.setWhen(busCtrl.isWriting(baseAddress + 4))

    // Offset 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 = slave(Apb3(Apb3Config(addressWidth=8, dataWidth=32)))
    val interrupt = out Bool()
    val external = new Bundle {
      val tick = in Bool()
      val clear = in Bool()
    }
  }

  //Prescaler is very similar to the timer, it mainly integrates a piece of auto␣
↪reload logic.
  val prescaler = Prescaler(width = 16)
```

(continues on next page)

(continued from previous page)

```
  val timerA = Timer(width = 32)
  val timerB,timerC,timerD = Timer(width = 16)

  val busCtrl = Apb3SlaveFactory(io.apb)

  prescaler.driveFrom(busCtrl, 0x00)

  timerA.driveFrom(busCtrl, 0x40)(
    ticks=List(True, prescaler.io.overflow),
    clears=List(timerA.io.full)
  )
  timerB.driveFrom(busCtrl, 0x50)(
    ticks=List(True, prescaler.io.overflow, io.external.tick),
    clears=List(timerB.io.full, io.external.clear)
  )
  timerC.driveFrom(busCtrl, 0x60)(
    ticks=List(True, prescaler.io.overflow, io.external.tick),
    clears=List(timerC.io.full, io.external.clear)
  )
  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)
  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
}
```

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 repositories using SpinalHDL *there*

13.4 Getting started

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
  }
}
```

CHAPTER

FOURTEEN

LEGACY

14.1 RiscV

Warning: This page only documents the first generation of RISC-V CPU created in SpinalHDL. This page does not document the VexRiscV CPU, which is the second generation of this CPU and is available here and offers better performance/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 consider the VexRiscv instead.

14.1.4 How to debug

You can find the openOCD fork here :

https://github.com/Dolu1990/openocd_riscv

An example target configuration file could be find here :

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 Introduction

Note: This page only documents the SoC implemented with the first generation of RISC-V CPU created in SpinalHDL. This page does not document the VexRiscV CPU, which is the second generation of this SoC (and CPU) is available here and offers 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*

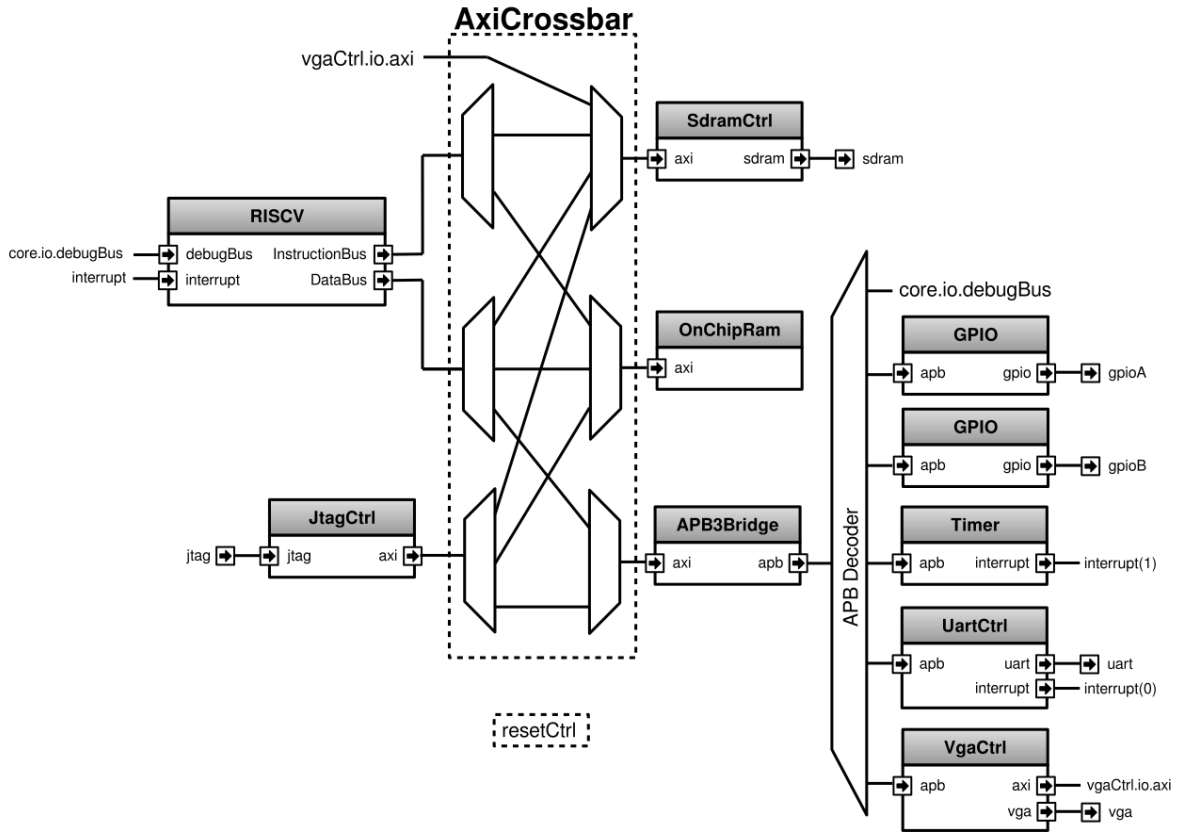
Board support

A DE1-SOC FPGA project can be find here with some demo binaries.

14.2.2 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 integrates an AXI4 bus fabric. AXI4 is not the easiest bus to work with 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 cores
- A handshake methodology that fits with SpinalHDL Stream.

From an Area utilization perspective, AXI4 is for sure not the lightest solution, but some techniques could dramatically reduce that concern :

- Using Read-Only/Write-Only AXI4 variations where that is possible
- Introducing an Axi4-Shared variation where a new ARW channel is introduced to replace and combine AR and AW channels. This solution reduces resource usage by a factor of two for the address decoding and the address arbitration.
- Timing relaxation is possible depending upon the interconnect implementation, and if all masters never stall the R/B channel (RREADY and BREADY are strapped to 1). Both xREADY signals can be removed by synthesis in this case, relaxing timings.
- As the AXI4 spec suggests, the interconnect can expand the transactions ID by aggregating the corresponding input port ID. This allows the interconnect to have an infinite number of pending requests 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 cores

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.

14.2.3 SoC toplevel (Pinsec)

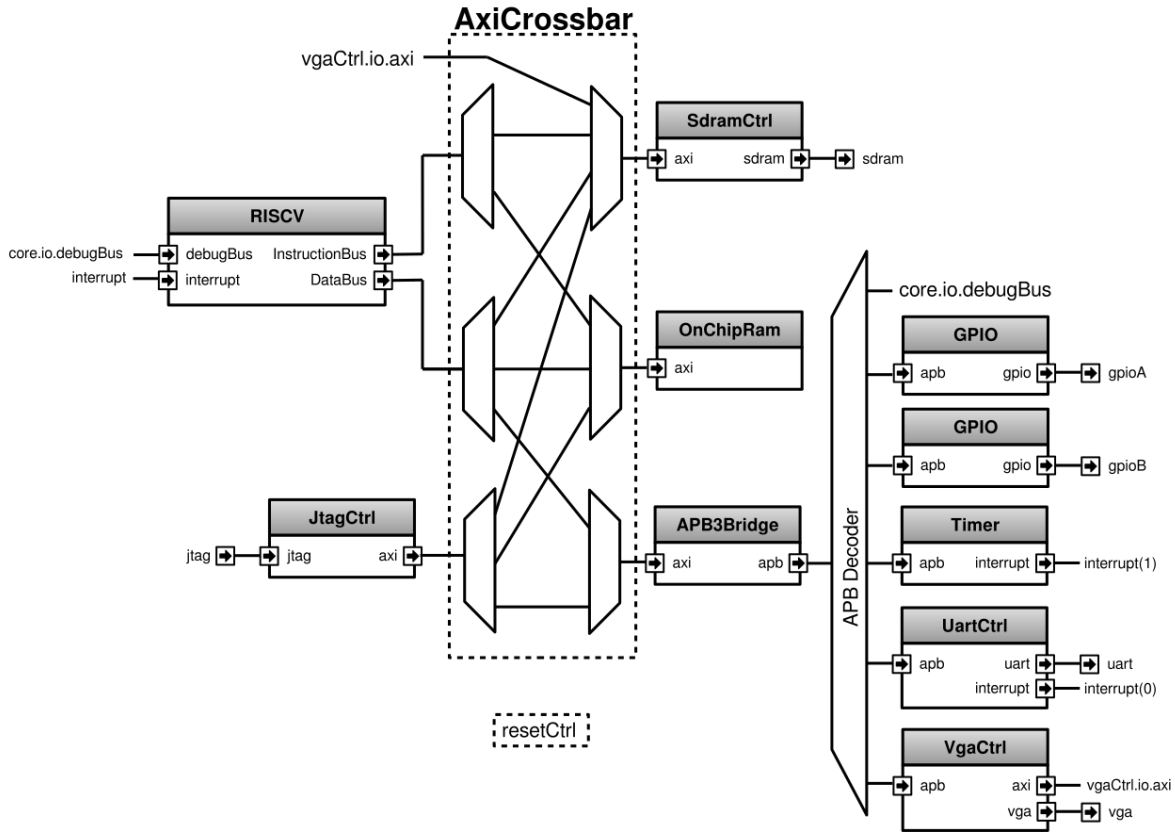
Introduction

`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 is a mix some design patterns that make it very easy to modify. Adding a new master or a new peripheral to the bus fabric could be done with little effort.

The toplevel implementation could be consulted at the links here : https://github.com/SpinalHDL/SpinalHDL/blob/master/lib/src/main/scala/spinal/lib/soc/pinsec/Pinsec.scala

This 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 an 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)
}
```

Clock domain setup for each system

Now that the reset controller is implemented, we can define clock domain for all sub-systems 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 instantiation 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 instantiation 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 uses less area while being fully interoperable with full AXI4.

```
val ram = Axi4SharedOnChipRam(
  dataWidth = 32,
  byteCount = 4 KiB,
  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 has some integrated peripherals :

- GPIO
- Timer
- UART
- VGA

GPIO

```
val gpioACtrl = Apb3Gpio(
  gpioWidth = 32
)

val gpioBCtrl = Apb3Gpio(
  gpioWidth = 32
)
```

Timer

The Pinsec timer module consists 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 KiB),
  sdramCtrl.io.axi -> (0x40000000L,  64 MiB),
  apbBridge.io.axi -> (0xF0000000L,   1 MiB)
)
```

Then you need to populate a matrix of interconnections between slaves and masters (this sets up visibility) :

```
//         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 KiB),
    gpioBCtrl.io.apb -> (0x01000, 4 KiB),
    uartCtrl.io.apb  -> (0x10000, 4 KiB),
    timerCtrl.io.apb -> (0x20000, 4 KiB),
    vgaCtrl.io.apb   -> (0x30000, 4 KiB),
    core.io.debugBus -> (0xF0000, 4 KiB)
  )
)
```

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
}
```

14.2.4 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

MISCELLANEOUS

This section includes content that may be:

- out-of-date
- could be better curated
- may contain duplicate information (better found elsewhere here or in another repo)
- drafts of documentation and works in progress

So please consider the information in this section with caution and a best effort on the author to provide documentation.

15.1 Frequent Errors

This page will talk about errors which could happen when people are using SpinalHDL.

15.2 Exception in thread “main” java.lang.NullPointerException

Console symptoms :

```
Exception in thread "main" java.lang.NullPointerException
```

Code Example :

```
val a = b + 1         //b can't be read at that time, because b isn't instantiated yet
val b = UInt(4 bits)
```

Issue explanation :

SpinalHDL is not a language, it is an Scala library, which mean, it obey to the same rules than the Scala general purpose programming language. When you run your SpinalHDL hardware description to generate the corresponding VHDL/Verilog RTL, your SpinalHDL hardware description will be executed as a Scala programm, and b will be a `null` reference until the programm execution come to that line, and it’s why you can’t use it before.

15.3 Hierarchy violation

The SpinalHDL compiler check that all your assignments are legal from an hierarchy perspective. Multiple cases are elaborated in following chapters

15.3.1 Signal X can't be assigned by Y

Console symptoms :

```
Hierarchy violation : Signal X can't be assigned by Y
```

Code Example :

```
class ComponentX extends Component{
  ...
  val X = Bool()
  ...
}

class ComponentY extends Component{
  ...
  val componentX = new ComponentX
  val Y = Bool()
  componentX.X := Y //This assignment is not legal
  ...
}
```

```
class ComponentX extends Component{
  val io = new Bundle{
    val X = Bool() //Forgot to specify an in/out direction
  }
  ...
}

class ComponentY extends Component{
  ...
  val componentX = new ComponentX
  val Y = Bool()
  componentX.io.X := Y //This assignment will be detected as not legal
  ...
}
```

Issue explanation :

You can only assign input signals of subcomponents, else there is an hierarchy violation. If this issue happend, you probably forgot to specify the X signal's direction.

15.3.2 Input signal X can't be assigned by Y

Console symptoms :

```
Hierarchy violation : Input signal X can't be assigned by Y
```

Code Example :

```
class ComponentXY extends Component{
  val io = new Bundle{
    val X = in Bool()
  }
  ...
  val Y = Bool()
  io.X := Y //This assignment is not legal
  ...
}
```

Issue explanation :

You can only assign an input signals from the parent component, else there is an hierarchy violation. If this issue happend, you probably mixed signals direction declaration.

15.3.3 Output signal X can't be assigned by Y

Console symptoms :

```
Hierarchy violation : Output signal X can't be assigned by Y
```

Code Example :

```
class ComponentX extends Component{
  val io = new Bundle{
    val X = out Bool()
  }
  ...
}

class ComponentY extends Component{
  ...
  val componentX = new ComponentX
  val Y = Bool()
  componentX.X := Y //This assignment is not legal
  ...
}
```

Issue explanation :

You can only assign output signals of a component from the inside of it, else there is an hierarchy violation. If this issue happend, you probably mixed signals direction declaration.

15.4 The `spinal.core` components

The core components of the language are described in this document. It is part of the general

The core language components are as follows:

- **Clock domains**, which allow to define and interoperate multiple clock domains within a design
- *Memory instantiation*, which permit the automatic instantiation of RAM and ROM memories.
- *IP instantiation*, using either existing VHDL or Verilog component.
- Assignments
- When / Switch
- Component hierarchy
- Area
- Functions
- Utility functions
- VHDL generator

15.4.1 Clock domains definitions

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

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

Clock domain syntax

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

```
ClockDomain(clock :  Bool[,reset :  Bool[,enable :  Bool]])
```

This definition takes three parameters:

1. The clock signal that defines the domain
2. An optional `reset`signal. If a register which need a reset and his clock domain didn't provide one, an error message happen
3. An optional `enable` signal. 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.

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 bits))
}
```

Clock configuration

In addition to the constructor parameters given *here* , the following elements of each clock domain are configurable via a `ClockDomainConfig` class :

Property	Valid values
`clockEdge`	`RISING`, `FALLING`
`ResetKind`	`ASYNC`, `SYNC`
`resetActiveHigh`	`true`, `false`
`clockEnableActiveHigh`	`true`, `false`

```
class CustomClockExample extends Component {
  val io = new Bundle {
    val clk = in Bool()
    val resetn = in Bool()
    val result = out UInt (4 bits)
  }
  val myClockDomainConfig = ClockDomainConfig(
    clockEdge = RISING,
    resetKind = ASYNC,
    resetActiveLevel = LOW
  )
  val myClockDomain = ClockDomain(io.clk,io.resetn,config = myClockDomainConfig)
  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 one is :

- clock : rising edge
- reset: asynchronous, active high
- no enable signal

External clock

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

```
class ExternalClockExample extends Component {
  val io = new Bundle {
    val result = out UInt (4 bits)
  }
  val myClockDomain = ClockDomain.external("myClockName")
  val myArea = new ClockingArea(myClockDomain){
    val myReg = Reg(UInt(4 bits)) init(7)
    myReg := myReg + 1

    io.result := myReg
  }
}
```

Cross Clock Domain

SpinalHDL checks at compile time that there is 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:

```
val asynchronousSignal = UInt(8 bits)
...
val buffer0 = Reg(UInt(8 bits)).addTag(crossClockDomain)
val buffer1 = Reg(UInt(8 bits))
buffer0 := asynchronousSignal
buffer1 := buffer0   // Second register stage to be avoid metastability issues
```

```
// Or in less lines:
val buffer0 = RegNext(asynchronousSignal).addTag(crossClockDomain)
val buffer1 = RegNext(buffer0)
```

15.4.2 Assignments

There are multiple assignment operator :

Symbole	Description
:=	Standard assignment, equivalent to '<=' in VHDL/Verilog last assignment win, value updated at next delta cycle
/=	Equivalent to := in VHDL and = in Verilog value updated instantly
<>	Automatic connection between 2 signals. Direction is inferred by using signal direction (in/out) Similar behavioural than :=

```
//Because of hardware concurrency is always read with the value '1' by b and c
val a,b,c = UInt(4 bits)
a := 0
b := a
a := 1  //a := 1 win
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
```

SpinalHDL check that bitcount of left and right assignment side match. There is multiple ways to adapt bitcount of BitVector (Bits, UInt, SInt) :

Resizing ways	Description
x := y.resized	Assign x wit 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 are 2 cases where spinal automaticly resize things :

Assignement	Problem	SpinalHDL action
myUIntOf_8bit := U(3)	U(3) create an UInt of 2 bits, which don't match with left side	Because U(3) is a "weak" bit inferred signal, SpinalHDL resize it automatically
myUIntOf_8bit := U(2 -> False default -> true)	The right part infer a 3 bit UInt, which doesn't match with the left part	SpinalHDL reapply the default value to bit that are missing

15.4.3 When / Switch

As VHDL and Verilog, wire and register can be conditionally assigned by using when and switch syntaxes

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

switch(x){
  is(value1){
    //execute when x === value1
  }
  is(value2){
    //execute when x === value2
  }
  default{
    //execute if none of precedent condition meet
  }
}
```

You can also define new signals into a when/switch statement. It's useful if you want to calculate an intermediate value.

```
val toto,titi = UInt(4 bits)
val a,b = UInt(4 bits)

when(cond){
  val tmp = a + b
  toto := tmp
  titi := tmp + 1
```

(continues on next page)

(continued from previous page)

```
} otherwise {
  toto := 0
  titi := 0
}
```

15.4.4 Component/Hierarchy

Like in VHDL and Verilog, you can define components that could be used to build a design hierarchy. But unlike them, you don't need to bind them at instantiation.

```
class AdderCell extends Component {
  //Declaring all in/out in an io Bundle is probably a good practice
  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
  val cell0 = new AdderCell
  val cell1 = new AdderCell
  cell1.io.cin := cell0.io.cout //Connect carrys
  ...
  val cellArray = Array.fill(width)(new AdderCell)
  ...
}
```

Syntax to define in/out is the following :

Syntax	Description	Return
in/out(x : Data)	Set x an input/output	x
in/out Bool()	Create an input/output Bool	Bool
in/out Bits/UInt/SInt(x bits)	Create an input/output of the corresponding type	T

There is some rules about component interconnection :

- Components can only read outputs/inputs signals of children components
- Components can read outputs/inputs ports values
- If for some reason, you need to read a signals from far away in the hierarchy (debug, temporal patch) you can do it by using the value returned by some.where.else.theSignal.pull().

15.4.5 Area

Sometime, creating a component to define some logic is overkill and to much verbose. For this kind of cases you can use Area :

```
class UartCtrl extends Component {
  ...
  val timer = new Area {
    val counter = Reg(UInt(8 bits))
    val tick = counter === 0
    counter := counter - 1
    when(tick) {
      counter := 100
    }
  }
  val tickCounter = new Area {
    val value = Reg(UInt(3 bits))
    val reset = False
    when(timer.tick) {          // Refer to the tick from timer area
      value := value + 1
    }
    when(reset) {
      value := 0
    }
  }
  val stateMachine = new Area {
    ...
  }
}
```

15.4.6 Function

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

- You can instantiate register, combinatorial logic and component inside them.
- You don't have to play with `process`/`@always` that limit the scope of assignment of signals
- Everything work by reference, which allow many manipulation.
  For example you can give to a function an bus as argument, then the function can internaly read/write it.
  You can also return a Component, a Bus, are anything else from scala the scala world.

RGB to gray

For example if you want to convert a Red/Green/Blue color into a gray one by using coefficient, 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) +
```

(continues on next page)

(continued from previous page)

```
            coef(g,0.4f) +
            coef(b,0.3f)
```

Valid Ready Payload bus

For instance if you define a simple Valid Ready Payload bus, you can then define useful function inside it.

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

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

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

15.4.7 VHDL generation

There is a small component and a `main` that generate the corresponding VHDL.

```
// spinal.core contain all basics (Bool, UInt, Bundle, Reg, Component, ..)
import spinal.core._

//A simple component definition
class MyTopLevel extends Component {
  //Define some input/output. 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 of the project. It create a instance of MyTopLevel and
//call the SpinalHDL library to flush it into a VHDL file.
object MyMain {
  def main(args: Array[String]) {
    SpinalVhdl(new MyTopLevel)
  }
}
```

15.4.8 Instantiate VHDL and Verilog IP

In some cases, it could be useful to instantiate a VHDL or a Verilog component into a SpinalHDL design. To do that, you need to define BlackBox which is like a Component, but its internal implementation should be provided by a separate VHDL/Verilog file to the simulator/synthesis tool.

```
class Ram_1w_1r(_wordWidth: Int, _wordCount: Int) extends BlackBox {
  val generic = new Generic {
    val wordCount = _wordCount
    val wordWidth = _wordWidth
  }

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

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

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

15.4.9 Utils

The SpinalHDL core contain some utils :

Syntax	Description	Return
log2Up(x : BigInt)	Return the number of bit needed to represent x states	Int
isPow2(x : BigInt)	Return true if x is a power of two	Boolean

Much more tool and utils are present in spinal.lib

15.5 Element

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"xyz" one
default -> y : Boolean/Bool	Set all unconnected bits with the y value. This feature could only be use to do assignments without the B prefix or with the B prefix combined with the bits specification

15.6 Range

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

CHAPTER

SIXTEEN

DEVELOPERS AREA

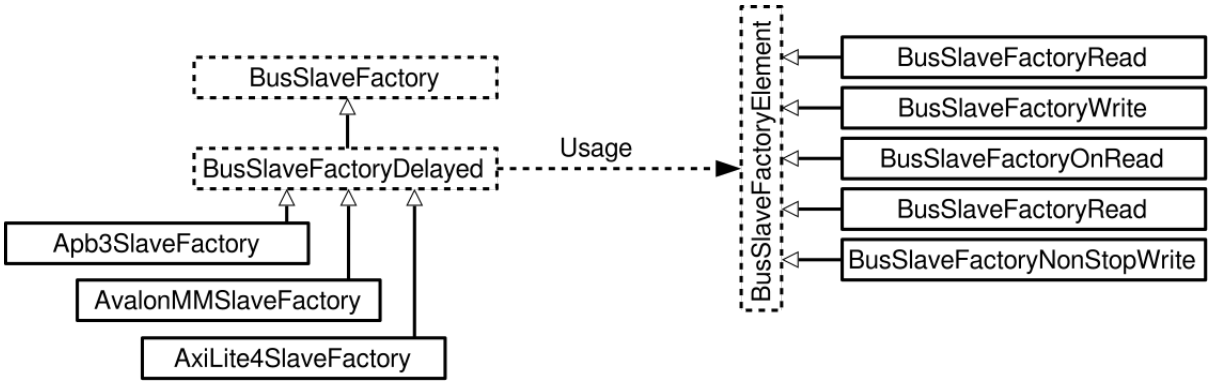
16.1 Bus Slave Factory Implementation

16.1.1 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 *here*.

16.1.2 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)		Instantiate 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.

16.1.3 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 bits))
    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	Bits(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
}
```

16.1.4 Conclusion

That's all, you can check one example that use this `Apb3SlaveFactory` to create an Apb3UartCtrl *here*.

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

16.2 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 which file, and in what order to put it in the documentation tree.

16.2.1 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:

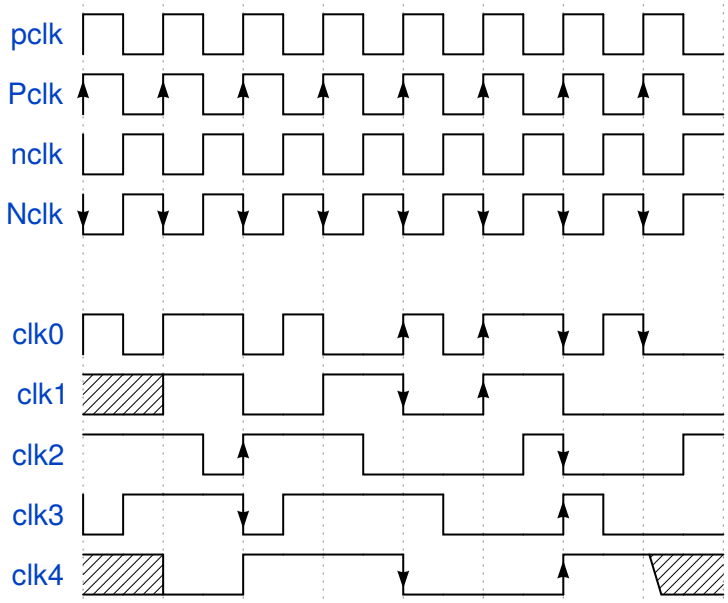
- `=` over and underline for section titles
- `=` underline for titles
- `-` underline for paragraph
- `^` for subparagraph

16.2.2 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" }
   ]}
```

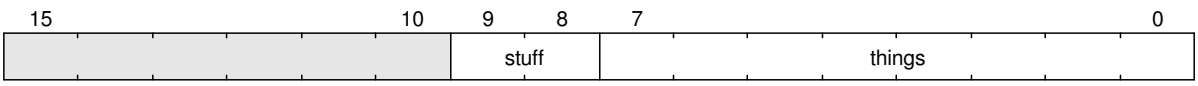
and you get:



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 }
 }
```



16.2.3 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.

16.3 Build through Mill

SpinalHDL itself can be built with Mill. This is an alternative to the Sbt build tool that can be found at Introduction_to_Mill. It can compile/test/publishLocal the existing modules. Build through mill can be much faster than Sbt, which is useful while debugging.

16.3.1 Compile the library

```
mill __.compile
sbt compile # equivalent alternatives
```

16.3.2 Run all test suites

```
mill __.test
sbt test # equivalent alternatives
```

16.3.3 Run a specified test suite

```
mill tester.test.testOnly spinal.xxxxx.xxxxx
sbt "tester/testOnly spinal.xxxxx.xxxxx" # equivalent alternatives
```

16.3.4 Run a specified App

```
mill tester.runMain spinal.xxxxx.xxxxx
sbt "tester/runMain spinal.xxxxx.xxxxx" # equivalent alternatives
```

16.3.5 Publish locally

Mill can also publish the library to the local ivy2 repository as a `dev` version.

```
mill __.publishLocal
sbt publishLocal # equivalent alternatives
```

16.4 SpinalHDL internal datamodel

16.4.1 Introduction

This page provides documentation on the internal data structure utilized by SpinalHDL for storing and modifying the netlist described by users via the SpinalHDL API.

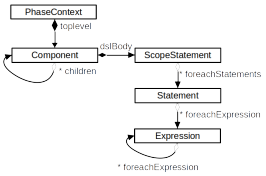
16.4.2 General structure

The following diagrams follow the UML nomenclature :

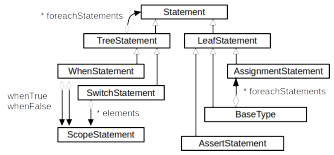
- A link with a white arrow mean “base extend target”
- A link with a black diamond mean “base contains target”
- A link with a white diamond mean “base has a reference to target”
- The * symbol mean “multiple”

The majority of the data structures are stored using double-linked lists, which facilitate the insertion and removal of elements.

There is a diagram of the global data structure :



And here more details about the *Statement* class :



In general, when an element within the data model utilizes other expressions or statements, that element typically includes functions for iterating over these usages. For example, each Expression is equipped with a *foreachExpression* function.

When using these iteration functions, you have the option to remove the current element from the tree.

Additionally, as a side note, while the *foreachXXX* functions iterate only one level deep, there are often corresponding *walkXXX* functions that perform recursive iteration. For instance, using *myExpression.walkExpression* on *((a+b)+c)+d* will traverse the entire tree of addition operations.

There are also utilities like *myExpression.remapExpressions(Expression => Expression),* which iterate through all the expressions used within *myExpression* and replace them with the one you provide.

More generaly, most of the graph checks and transformations done by SpinalHDL are located in <https://github.com/SpinalHDL/SpinalHDL/blob/dev/core/src/main/scala/spinal/core/internals/Phase.scala>

16.4.3 Exploring the datamodel

Here is an example that identifies all adders within the netlist without utilizing shortcuts. :

```
object FindAllAddersManualy {
  class Toplevel extends Component{
    val a,b,c = in UInt(8 bits)
    val result = out(a + b + c)
  }

  import spinal.core.internals._

  class PrintBaseTypes(message : String) extends Phase{
    override def impl(pc: PhaseContext) = {
      println(message)

      recComponent(pc.topLevel)

      def recComponent(c: Component): Unit = {
        c.children.foreach(recComponent)
        c.dslBody.foreachStatements(recStatement)
      }

      def recStatement(s: Statement): Unit = {
        s.foreachExpression(recExpression)
        s match {
          case ts: TreeStatement => ts.foreachStatements(recStatement)
          case _ =>
        }
      }

      def recExpression(e: Expression): Unit = {
        e match {
          case op: Operator.BitVector.Add => println(s"Found ${op.left} + ${op.right}
↪")
          case _ =>
        }
        e.foreachExpression(recExpression)
      }

    }
    override def hasNetlistImpact = false

    override def toString = s"${super.toString} - $message"
  }

  def main(args: Array[String]): Unit = {
    val config = SpinalConfig()

    //Add a early phase
    config.addTransformationPhase(new PrintBaseTypes("Early"))

    //Add a late phase
    config.phasesInserters += {phases =>
      phases.insert(phases.indexWhere(_.isInstanceOf[PhaseVerilog]), new␣
↪PrintBaseTypes("Late"))
    }
```

(continues on next page)

(continued from previous page)

```
    config.generateVerilog(new Toplevel())
  }
}
```

Which will produces :

```
[Runtime] SpinalHDL v1.6.1    git head : 3100c81b37a04715d05d9b9873c3df07a0786a9b
[Runtime] JVM max memory : 8044.0MiB
[Runtime] Current date : 2021.10.16 20:31:33
[Progress] at 0.000 : Elaborate components
[Progress] at 0.163 : Checks and transforms
Early
Found (toplevel/a : in UInt[8 bits]) + (toplevel/b : in UInt[8 bits])
Found (toplevel/??? :  UInt[? bits]) + (toplevel/c : in UInt[8 bits])
[Progress] at 0.191 : Generate Verilog
Late
Found (UInt + UInt)[8 bits] + (toplevel/c : in UInt[8 bits])
Found (toplevel/a : in UInt[8 bits]) + (toplevel/b : in UInt[8 bits])
[Done] at 0.218
```

Please note that in many cases, shortcuts are available. All the recursive processes mentioned earlier could have been replaced by a single one. :

```
override def impl(pc: PhaseContext) = {
  println(message)
  pc.walkExpression{
    case op: Operator.BitVector.Add => println(s"Found ${op.left} + ${op.right}")
    case _ =>
  }
}
```

16.4.4 Compilation Phases

Here is the complete list of default phases, arranged in order, that are employed to modify, check, and generate Verilog code from a top-level component. :

<https://github.com/SpinalHDL/SpinalHDL/blob/ec8cd9f513566b43cbbdb08d0df4dee1f0fee655/core/src/main/scala/spinal/core/internals/Phase.scala#L2487>

If you, as a user, add a new compilation phase by using *SpinalConfig.addTransformationPhase(new MyPhase())*, this phase will be inserted immediately after the user component elaboration process, which is relatively early in the compilation sequence. During this phase, you can still make use of the complete SpinalHDL user API to introduce elements into the netlist.

If you choose to use the SpinalConfig.phasesInserters API, it's essential to exercise caution and ensure that any modifications made to the netlist align with the phases that have already been executed. For instance, if you insert your phase after the *PhaseInferWidth*, you must specify the width of each node you introduce.

16.4.5 Modifying a netlist as a user without plugins

There are several user APIs that enable you to make modifications during the user elaboration phase. :

- mySignal.removeAssignments : Will remove all previous := affecting the given signal
- mySignal.removeStatement : Will void the existance of the signal
- mySignal.setAsDirectionLess : Will turn a in / out signal into a internal signal
- mySignal.setName : Enforce a given name on a signal (there is many other variants)
- mySubComponent.mySignal.pull() : Will provide a readable copy of the given signal, even if that signal is somewhere else in the hierarchy
- myComponent.rework{ myCode } : Execute *myCode* in the context of *myComponent*, allowing modifying it with the user API

For example, the following code can be used to modify a top-level component by adding a three-stage shift register to each input and output of the component. This is particularly useful for synthesis testing.

```
def ffIo[T <: Component](c : T): T ={
  def buf1[T <: Data](that : T) = KeepAttribute(RegNext(that)).addAttribute("DONT_
↪TOUCH")
  def buf[T <: Data](that : T) = buf1(buf1(buf1(that)))
  c.rework{
    val ios = c.getAllIo.toList
    ios.foreach{io =>
      if(io.getName() == "clk"){
        //Do nothing
      } else if(io.isInput){
        io.setAsDirectionLess().allowDirectionLessIo  //allowDirectionLessIo is to␣
↪disable the io Bundle linting
        io := buf(in(cloneOf(io).setName(io.getName() + "_wrap")))
      } else if(io.isOutput){
        io.setAsDirectionLess().allowDirectionLessIo
        out(cloneOf(io).setName(io.getName() + "_wrap")) := buf(io)
      } else ???
    }
  }
  c
}
```

You can use the code in the following manner: :

```
SpinalVerilog(ffIo(new MyToplevel))
```

Here is a function that enables you to execute the body code as if the current component's context did not exist. This can be particularly useful for defining new signals without the influence of the current conditional scope (such as when or switch).

```
def atBeginingOfCurrentComponent[T](body : => T) : T = {
  val body = Component.current.dslBody  // Get the head of the current component␣
↪symbols tree (AST in other words)
  val ctx = body.push()                 // Now all access to the SpinalHDL API will␣
↪be append to it (instead of the current context)
  val swapContext = body.swap()         // Empty the symbol tree (but keep a␣
↪reference to the old content)
  val ret = that                        // Execute the block of code (will be added␣
↪to the recently empty body)
  ctx.restore()                         // Restore the original context in which this␣
```

(continues on next page)

(continued from previous page)

```
↪function was called
  swapContext.appendBack()              // append the original symbols tree to the␣
↪modified body
  ret                                   // return the value returned by that
}

val database = mutable.HashMap[Any, Bool]()
def get(key : Any) : Bool = {
  database.getOrElseUpdate(key, atBeginingOfCurrentComponent(False)
}

object key

when(something){
  if(somehow){
    get(key) := True
  }
}
when(database(key)){
   ...
}
```

This kind of functionality is, for instance, employed in the VexRiscv pipeline to dynamically create components or elements as needed.

16.4.6 User space netlist analysis

The SpinalHDL data model is also accessible and can be read during user-time elaboration. Here's an example that can help find the shortest logical path (in terms of clock cycles) to traverse a list of signals. In this specific case, it is being used to analyze the latency of the VexRiscv FPU design.

```
println("cpuDecode to fpuDispatch " + LatencyAnalysis(vex.decode.arbitration.isValid,␣
↪logic.decode.input.valid))
println("fpuDispatch to cpuRsp    " + LatencyAnalysis(logic.decode.input.valid,␣
↪plugin.port.rsp.valid))

println("cpuWriteback to fpuAdd   " + LatencyAnalysis(vex.writeBack.input(plugin.FPU_
↪COMMIT), logic.commitLogic(0).add.counter))

println("add                      " + LatencyAnalysis(logic.decode.add.rs1.mantissa,␣
↪logic.get.merge.arbitrated.value.mantissa))
println("mul                      " + LatencyAnalysis(logic.decode.mul.rs1.mantissa,␣
↪logic.get.merge.arbitrated.value.mantissa))
println("fma                      " + LatencyAnalysis(logic.decode.mul.rs1.mantissa,␣
↪logic.get.decode.add.rs1.mantissa, logic.get.merge.arbitrated.value.mantissa))
println("short                    " + LatencyAnalysis(logic.decode.shortPip.rs1.
↪mantissa, logic.get.merge.arbitrated.value.mantissa))
```

Here you can find the implementation of that LatencyAnalysis tool : <https://github.com/SpinalHDL/SpinalHDL/blob/3b87c898cb94dc08456b4fe2b1e8b145e6c86f63/lib/src/main/scala/spinal/lib/Utils.scala#L620>

16.4.7 Enumerating every ClockDomain in use

In this case, this is accomplished after the elaboration process by utilizing the SpinalHDL report.

```
object MyTopLevelVerilog extends App{
  class MyTopLevel extends Component {
    val cdA = ClockDomain.external("rawrr")
    val regA = cdA(RegNext(False))

    val sub = new Component {
      val cdB = ClockDomain.external("miaou")
      val regB = cdB(RegNext(False))

      val clkC = CombInit(regB)
      val cdC = ClockDomain(clkC)
      val regC = cdC(RegNext(False))
    }
  }

  val report = SpinalVerilog(new MyTopLevel)

  val clockDomains = mutable.LinkedHashSet[ClockDomain]()
  report.toplevel.walkComponents(c =>
    c.dslBody.walkStatements(s =>
      s.foreachClockDomain(cd =>
        clockDomains += cd
      )
    )
  )

  println("ClockDomains : " + clockDomains.mkString(", "))
  val externals = clockDomains.filter(_.clock.component == null)
  println("Externals : " + externals.mkString(", "))
}
```

Will print out

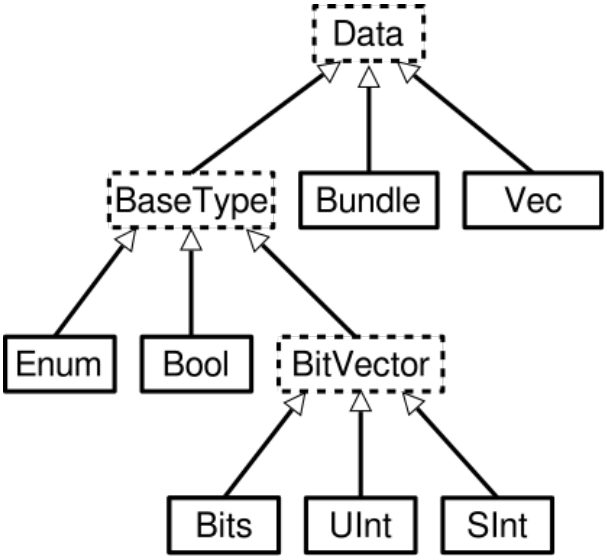
```
ClockDomains : rawrr_clk, miaou_clk, clkC
Externals : rawrr_clk, miaou_clk
```

16.5 Types

16.5.1 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.

Fixed point support is documented *Fixed-Point*

16.5.2 Bool

This is the standard *boolean* type that corresponds to a single 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

16.5.3 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 arithmetic 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 the table below)	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 follows the same rules as 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 can only be used 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)	Read bitfield, 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 an array of Bool	Vec(Bool(),width(x))

Masked comparison

Sometimes you need to check equality between a `BitVector` and a bits constant that contain holes defined as a bitmask (bit positions not to be compared by the equality expression).

An example demonstrating how to do that (note the use of 'M' prefix) :

```
val myBits = Bits(8 bits)
val itMatch = myBits === M"00--10--"
```

16.5.4 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 bits as necessary at the MSB to widen, may also truncate width retaining at the LSB side, y : Int	T(y bits)
x.resizeLeft(y)	Return a resized copy of x, filled with zero bits as necessary at the LSB to widen, may also truncate width retraining at the MSB side, y : Int	T(y bits)

16.5.5 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)

16.5.6 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)
x.asBool	Binary cast in Bool	Bool(x.lsb)

16.5.7 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 supports 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
```

16.5.8 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 instantiate 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 bits)
  val PSEL       = Bits(selWidth bits)
  val PENABLE    = Bool()
  val PREADY     = Bool()
  val PWRITE     = Bool()
  val PWDATA     = Bits(dataWidth bits)
  val PRDATA     = Bits(dataWidth bits)
  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 bits)
  val PSEL       = Bits(config.selWidth bits)
  val PENABLE    = Bool()
  val PREADY     = Bool()
  val PWRITE     = Bool()
  val PWDATA     = Bits(config.dataWidth bits)
  val PRDATA     = Bits(config.dataWidth bits)
  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 bits)
  val PSEL       = Bits(config.selWidth bits)
  val PENABLE    = Bool()
  val PREADY     = Bool()
  val PWRITE     = Bool()
  val PWDATA     = Bits(config.dataWidth bits)
  val PRDATA     = Bits(config.dataWidth bits)
  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 integrates a small master slave utility 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 in 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)
}
```

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 bits)
  val PSEL       = Bits(selWidth bits)
  val PENABLE    = Bool()
  val PREADY     = Bool()
  val PWRITE     = Bool()
  val PWDATA     = Bits(dataWidth bits)
  val PRDATA     = Bits(dataWidth bits)
  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.
}
```

16.5.9 Enum

SpinalHDL supports 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 position corresponds to one state, only one bit is active at a time when encoded.

Define an enumeration type:

```
object UartCtrlTxState extends SpinalEnum { // Or␣
↪SpinalEnum(defaultEncoding=encodingOfYourChoice)
  val sIdle, sStart, sData, sParity, sStop = newElement()
}
```

Instantiate a signal to store the enumeration encoded value and assign it a value :

```
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
```

16.5.10 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

16.5.11 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
- Setup inferred type: UInt(4 bits)
- Clock cycles where *cond =/= True* will result in the constant being reinstated
- Clock cycles where *cond === True* will result in the signal having the value of *red* due to the last statement wins rule.

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
}
```

16.5.12 Continuous Assignment Literals as signal declaration

You can also use them in expressions to do three things at once :

- Define a wire
- Maintain the result of an equality operation in the hardware logic implementation with the constant value and another signal
- Setup inferred type: Bool due to use of === equality operator having a result of type Bool

There is an example :

```
val done = Bool(False)
val blue = in UInt(4 bits)
...
val value = blue === U"0001"  // inferred type is Bool due to use of === operator
when(value) {
  done := True
}
```