HDLRuby user manual

Lovic Gauthier

Contents

1 About HDLRuby

HDLRuby is a library for describing and simulating digital electronic systems.

Warning:

- This is very preliminary work which may (will) change a lot before we release a stable version.
- It is highly recommended to have both basic knowledge of the Ruby language and hardware description languages before using HDLRuby.

2 Compiling HDLRuby descriptions

2.1 Using the HDLRuby compiler

‘hdrcc.rb’ is the HDLRuby compiler. It takes as input a HDLRuby file, checks it, and can produce as output a Verilog HDL or a YAML low-level descriptions of a HW components.

Usage:

```
hdrcc.rb [options] <input hdr file> [<output file>]
```

Where:

- `options` is a list of options
- `<input hdr file>` is the initial file to compile (mandatory)
- `<output file>` is the output file

Options	
`-y, --yaml`	Output in YAML format
`-v, --verilog`	Output in Verlog HDL format
`-s, --syntax`	Output the Ruby syntax tree
`-d, --directory`	Specify the base directory for loading the HDLRuby files
`-D, --debug`	Set the HDLRuby debug mode
`-t, --top system`	Specify the top system describing the circuit to compile
`-p, --param x,y,z`	Specify the generic parameters
`-h, --help`	Show the help message

Notice:

- If no top system is given, it is automatically looked for from the input file.
- If no option is given, simply checks the input file.
- If no output file is given, the result is given through the standard output.

Examples:

- Compile system named `adder` from `adder.hdr` input file and generate `adder.yaml` low-level YAML description:

```
hdrcc.rb --yaml --top adder adder.hdr adder.yaml
```

- Compile `adder.hdr` input file and generate `adder.v` low-level Verilog HDL description:

```
hdrcc.rb --verilog adder.hdr adder.v
```

- Check the validity of `adder.hrd` input file:

  ```
  hdrcc.rb adder.hdr
  ```

- Compile system `adder` whose bit width is generic from `adder_gen.hdr` input file to a 16-bit circuit whose low-level Verilog HDL description is dumped to the standard output:

```
hdrcc -v -t adder --param 16 adder_gen.hdr
```

- Compile system `multer` with inputs and output bit width is generic from `multer_gen.hdr` input file to a 16x16->32 bit cicruit whose low-level YAML description is saved to output file `multer_gen.yaml`

```
hdrcc -y -t multer -p 16,16,32 multer_gen.hdr multer_gen.yaml
```

2.2 Using HDLRuby within Ruby

You can also use HDLRuby in a Ruby program by loading `HDLRuby.rb` in your Ruby file:

```
require 'HDLRuby'
```

Then, you can set up Ruby for supporting high-level description of hardware components. This is done by adding the following line of code:

```
configure_high
```

After this statement, standard HDLRuby code can be written. In order to produce HW descriptions from this code a low-level hardware must then be generated from an instance of an HW module (*system* in HDLRuby). For example, assuming system 'circuitT' has been described in your Ruby program, an instance named 'circuitI' can be declared as follows:

```
circuitT :circuitI
```

From there a low-level description of the circuit is generated using the `to_low` methods as follows (in the following code, this description is assigned to Ruby variable 'circuitL'):

```
circuitL = circuitI.to_low
```

This low-level description can then be converted to a YAML format using 'to_yaml' or to a Verilog HDL format using 'to_verilog' as follows:

```
circuitY = circuitL.to_yaml
circuitV = circuitL.to_verilog
```

In the above examples, 'cricuitY' and 'cricuitV' are Ruby variables referring to the strings containing the respective YAML and Verilog HDL code.

2.3 Handling the low-level HDLRuby representation

You can include `HDLRuby::Low` for gaining access to the classes used for low-level description of hardware components.

```
include HDLRuby::Low
```

It is then possible to load a low-level representation of hardware as follows, where `stream` is a stream containing the representation.

```
hardwares = HDLRuby::from_yaml(stream)
```

For instance, you can load the sample description of an 8-bit adder as follows:

```
adder = HDLRuby::from_yaml(File.read("#{$:[0]}/HDLRuby/low_samples/adder.yaml"))
```

Note:

- A `HDLRuby::Low` description of hardware can only be built through standard Ruby class constructors, and does not include any validity check of the resulting hardware.

3 HDLRuby programming guide

HDLRuby has been designed to bring the high flexibility of the Ruby language to hardware descriptions while ensuring that they remain synthesizable. In this context, all the abstractions provided by HDLRuby are in the way of describing hardware, but not in its execution model, this latter being RTL by construction.

The second specificity of HDLRuby is that it supports natively all the features of the Ruby language.

Notes:

- It is still possible to extend HDLRuby to support hardware descriptions of higher level than RTL, please refer to section Extending HDLRuby for more details.
- In this document, HDLRuby constructs will often be compared to their Verilog HDL or VHDL equivalents for simpler explanations.

3.1 Introduction

This introduction gives a glimpse of the possibilities of the language. However, we do recommend to consult the section about the high-level programming features to have a more complete view of the advanced possibilities of this language.

At first glance, HDLRuby is similar to any other HDL languages (like Verilog HDL or VHDL), for instance the following code describes a simple D-FF:

```
system :dff do
   bit.input :clk, :rst, :d
   bit.output :q

   par(clk.posedge) do
      q <= d & ~rst
   end
end
```

As it can be seen in the code above, `system` is the keyword used for describing a digital circuit. This keyword is an equivalent of the Verilog HDL `module`. In such a system, signals are declared using a `<type>.<direction>` construct where `type` is the data type of the signal (e.g., `bit` as in the code above) and `direction` indicates if the signal is an input, an output, an inout or an inner one; and executable blocks (similar to `always` block of Verilog HDL) are described using the `par` keyword when they are parallel and `seq` when they are sequential (i.e, with respectively non-blocking and blocking assignments).

The code describing a `dff` given above is not much different from its equivalent in other HDL. However, HDLRuby provides several features for achieving a higher productivity when describing hardware. We will now describe a few of them.

First, several syntactic sugars exist that allow shorter code, for instance the following code is strictly equivalent to the previous description of `dff`:

```
system :dff do
   input :clk, :rst, :d
   output :q

   (q <= d & ~rst).at(clk.posedge)
end
```

Furthermore, generic parameters can be used for anything in HDLRuby. For instance, the following code describes an 8-bit register without any parameterization:

```
system :reg8 do
   input :clk, :rst
   [7..0].input :d
   [7..0].output :q

   (q <= d & [~rst]*8).at(clk.posedge)
end
```

But it is also possible to describe a register of arbitrary size as follows, where `n` is the parameter giving the number of bits of the register:

```
system :regn do |n|
   input :clk, :rst
   [n-1..0].input :d
   [n-1..0].output :q

   (q <= d & [~rst]*n).at(clk.posedge)
end
```

Or, even further, it is possible to describe a register of arbitrary type (not only bit vectors) as follows:

```
system :reg do |typ|
   input :clk, :rst
   typ.input :d
   typ.output :q

   (q <= d & [~rst]*typ.width).at(clk.posedge)
end
```

//: # (# Now let's generate the register //: #) (declarations.) //: # (make_reg(:dff) { bit }) //: # (make_reg(:reg8){ bit[7..0] }) //: # (make_reg(:regn){ |n| bit[n-1..0] }) //: # (make_reg(:reg) { |typ| typ }) //: # ("') //: # ( )

Wait… I have just realized: a D-FF without any inverted output does not look very serious. So let us extend the existing `dff` to provide an inverted output. There are basically three ways for doing this. First, inheritance can be used: a new system is built inheriting from `dff` as it is done in the following code.

```
system :dff_full, dff do
   output :qb
   qb <= ~q
end
```

The second possibility is to modify `dff` afterward. In HDLRuby, this achieved using the `open` method as it is done the following code:

```
dff.open do
   output :qb
   qb <= ~q
end
```

The third possibility is to modify directly a single instance of `dff` which require an inverted output, using again the `open` method, as in the following code:

```
# Declare dff0 as an instance of dff
dff :dff0

# Modify it
dff0.open do
```

```
    output :qb
    qb <= ~q
end
```

In this later case, only `dff0` will have an inverted output, the other instances of `dff` will not change.

Now assuming we opted for the first solution, we have now `dff_full`, a highly advanced D-FF with such unique features as an inverted output. So we would like to use it in other designs, for example a shift register of `n` bits. Such a system will include a generic number of `dff_full` instances, and can be described as follows making use of the native Ruby method `each_cons` for connecting them together:

```
system :shifter do |n|
    input :clk, :rst
    input :i0
    output :o0, :o0b

    # Instantiating n D-FF
    [n].dff_full :dffIs

    # Connect the clock and the reset.
    dffIs.each { |ff| ff.clk <= clk ; ff.rst <= rst }

    # Interconnect them as a shift register
    dffIs[0..-1].each_cons(2) { |ff0,ff1| ff1.d <= ff0.q }

    # Connects the input and output of the circuit
    dffIs[0].d <= i0
    o0 <= dffIs[-1].q
    o0b <= dffIs[-1].qb
end
```

As it can be seen in the above examples, in HDLRuby, any construct is an object and therefore include methods. For instance, declaring a signal of a given `type` and direction (input, output or inout) is done as follows, so that `direction` is actually a method of the type, and the signal names are actually the arguments of this method (symbols or string are supported.)

```
<type>.<direction> <list of symbols representing the signal>
```

Of course, if you do not need to use the specific component `dff_full` you can describe a shift register more simply as follows:

```
system :shifter do |n|
    input :clk, :rst
    input :i0
    output :o0
```

```
   [n].inner :sh

   par (clk.posedge) do
      hif(rst) { sh <= 0 }
      helse { sh <= [sh[n-2..0], i0] }
   end

   o0 <= sh[n-1]
end
```

Now, let us assume you what to design a circuit that performs a sum of products of several inputs with constant coefficients. For the case of 4 16-bit signed inputs and given coefficient as 3, 4, 5 and 6. The corresponding basic code could be as follows:

```
system :sumprod_16_3456 do
   signed[16].input :i0, :i1, :i2, :i3
   signed[16].output :o

   o <= i0*3 + i1*4 + i2*5 + i3*6
end
```

The description above is straight forward, but it would be necessary to rewrite it if another circuit with different bit width or coefficients is to be designed. Moreover, if the number of coefficient is large an error in the expression will be easy to make and hard to find. A better approach would be to use a generic description of such a circuit as follows:

```
system :sumprod do |typ,coefs|
   typ[coefs.size].input ins
   typ.output :o

   o <= coefs.each_with_index.reduce(_0) do
   |sum,(coef,i)|
      sum + ins[i]*coef
   end
end
```

In the code above, there are two generic parameters, `typ` that indicates the data type of the circuit and `coefs` that is assumed to be an array of coefficients. Since the number of inputs depends on the number of provided coefficients, it is declared as an array of `width` bit signed whose size is equal to the number of coefficients.

The description of the sum of product maybe more difficult to understand for people not familiar with the Ruby language. The `each_with_index` method iterates over the coefficients adding their index as iteration variable, the resulting operation (i.e., the iteration loop) is then modified by the `reduce` method that accumulates the code passed as arguments. This code, starting by `|sum,coef,i|`

simply performs the addition of the current accumulation result (`sum`) with the product of the current coefficient (`coef`) and input (`ins[i]`, where `i` is the index) in the iteration. The argument `_0` initializes the sum to `0`.

While slightly longer than the previous description, this description allows to declare a circuit implementing a sum of product with any bit width and any number of coefficients. For instance, the following code describes a signed 32-bit sum of product with 16 coefficients (actually just random numbers here).

```
sumprod [:my_circuit],
        signed[32],
        [3,78,43,246, 3,67,1,8,
         47,82,99,13, 5,77,2,4]
```

As seen in the code above, when passing generic argument for instantiating a generic system, the name of the instance is put between brackets for avoiding confusion.

While description `sumprod` is already usable in a wide range of cases, it still uses the standard addition and multiplication. However, there are cases where specific components are to be used for these operations, either for sake of performance, compliance with constraints, or because functionally different operations are required (e.g., saturated computations). This can be solved by using functions implementing such computation in place of operators, for example as follows:

```
system :sumprod_func do |typ,coefs|
   typ[coefs.size].input ins
   typ.output :o

   o <= coefs.each_with_index.reduce(_0) do
   |sum,(coef,i)|
      add(sum, mult(ins[i]*coef))
   end
end
```

Where `add` and `mult` are functions implementing the required specific operations. HDLRuby functions are similar to Verilog HDL ones. In our example, an addition that saturates at 1000 could be described as follows:

```
function :add do |x,y|
   inner :res
   seq do
      res <= x + y
      (res <= 1000).hif(res > 1000)
   end
end
```

With HDLRuby functions, the result of the last statement in the return value, in this case that will be the value of res. The code above is also an example of the usage of the postfixed if statement, it an equivalent of the following code:

```
      hif(res>1000) { res <= 1000 }
```

With functions, it is enough to change their content to obtained a new kind of circuit without change the main code. This approach suffers for two drawbacks though: first, the level of saturation is hardcoded in the function, second, it would be preferable to be able to select the function to execute instead of modifying its code. For the first problem a simple approach is to add an argument to the function given the saturation level. Such an add function would therefore be as follows:

```
function :add do |max, x, y|
   inner :res
   seq do
      res <= x + y
      (res <= max).hif(res > max)
   end
end
```

It would however be necessary to add this argument when invoking the function, e.g., `add(1000,sum,mult(...))`. While this argument is relevant for addition with saturation, it is not for the other kind of addition operations, and hence, the code of `sumprod` is not general any longer.

HDLRuby provides two ways to address such issues. First, it is possible to pass code as argument. In the case of `sumprod` it would then be enough to add two arguments that perform the required addition and multipliction. The example is bellow:

```
system :sumprod_proc do |add,mult,typ,coefs|
   typ[coefs.size].input ins
   typ.output :o

   o <= coefs.each_with_index.reduce(_0) do
   |sum,(coef,i)|
      add.(sum, mult.(ins[i]*coef))
   end
end
```

Note:

- With HDLRuby, when some code is passed as argument, it is invoked using the `.()` operator, and not simple parenthesis like functions.

Assuming the addition with saturation is now implemented by a function named `add_sat` and a multiplication with saturation is implemented by a function named `mult_sat` (with similar arguments), a circuit implementing a signed 16-bit sum of product saturating at 1000 with 16 coefficients could be described as follows:

```
sumprod_proc(
        proc { |x,y| add_sat(1000,x,y) },
        proc { |x,y| mult_sat(1000,x,y) },
        signed[64],
        [3,78,43,246, 3,67,1,8,
         47,82,99,13, 5,77,2,4]).(:my_circuit)
```

As seen in the example above, a piece of code is passed as argument using the proc keyword.

A second possible approach provided by HDLRuby is to declare a new data type with redefined addition and multiplication operators. For the case of a 16-bit saturated addition and multiplication the following generic data type can be defined (for signed computations):

```
signed[16].typedef(:sat16_1000)

sat16_1000.define_operator(:+) do |x,y|
   [16].inner :res
   seq do
      res <= x + y
      ( res <= 1000 ).hif(res > 1000)
   end
end
```

In the code above, the first line define the new type `sat16_1000` to be 16-bit signed and the remaining overloads (redefines) the `+` operator for this type (the same should be done for the `*` operator). Then, the initial version of `sumprod` can be used with this type to achieve saturated computations as follows:

```
sumprod(sat16_1000,
        [3,78,43,246, 3,67,1,8,
        47,82,99,13, 5,77,2,4]).(:my_circuit)
```

It is also possible to declare a generic type. For instance a generic signed type with saturation can be declared as follows:

```
typedef :sat do |width, max|
   signed[width]
end


sat.define_operator(:+) do |width,max, x,y|
   [width].inner :res
   seq do
      res <= x + y
      ( res <= max ).hif(res > max)
   end
end
```

Note:

- The generic parameters have also to be declared for the operator redefinitions.

With this generic type, the circuit can be declared as follows:

```
sumprod(sat(16,1000),
        [3,78,43,246, 3,67,1,8,
         47,82,99,13, 5,77,2,4]).(:my_circuit)
```

3.2 How does HDLRuby work

Contrary to descriptions in high-level HDL like SystemVerilog, VHDL or SystemC, HDLRuby descriptions are not software-like description of hardware, but are programs meant to produce hardware descriptions. In other words, while the execution of a common HDL code will result in some simulation of the described hardware, the execution of HDLRuby code will result in some low-level hardware description. This low-level description is synthesizable, and can also be simulated like any standard hardware description. This decoupling of the representation of the hardware from the point of view of the user (HDLRuby), and the actual hardware description (HDLRuby::Low) makes it possible to provide the user with any advanced software features without jeopardizing the synthesizability of the actual hardware description.

For that purpose, each construct in HDLRuby is not a direct description of some hardware construct, but a program which generates the corresponding description. For example, let us consider the following line of code of HDLRuby describing the connection between signal `a` and signal `b`:

```
a <= b
```

Its execution will produce the actual hardware description of this connection as an object of the HDLRuby::Low library — in this case an instance of the `HDLRuby::Low::Connection` class. Concretely, a HDLRuby system is described by a Ruby block, and the instantiation of this system is actually performed by executing this block. The actual synthesizable description of this hardware is the execution result of this instantiation.

From there, we will describe into more details each construct of HDLRuby.

3.3 Naming rules

Several constructs in HDLRuby are referred to by name, e.g., systems and signals. When such constructs are declared, their names are to be specified by Ruby symbols starting with a lower case. For example, `:hello` is a valid name declaration, but `:Hello` is not.

After being declared, the construct can be referred to by using the name directly (i.e., without the `:` of Ruby symbols). For example, if a construct has been declared with `:hello` as name, it will be afterward referred by `hello`.

3.4 Systems and signals

A system represents a digital system and corresponds to a Verilog HDL module. A system has an interface comprising input, output, and inout signals, and includes of structural and behavioral descriptions.

A signal represents a state in a system. It has a data type and a value, the latter varying with time. Similarly to VHDL, HDLRuby signals can be viewed as abstractions of both wires and registers in a digital circuit. As a general rule, a signal whose value is explicitly set all the time models a wire, otherwise it models a register.

3.4.1 Declaring an empty system

A system is declared using the keyword `system`. It must be given a Ruby symbol for name and a block that describe its content. For instance, the following code describes an empty system named `box`:

```
system(:box) {}
```

Notes:

- Since this is Ruby code, the body can also be delimited by the `do` and `end` Ruby keywords (in which case the parentheses can be omitted) as follows:

```
system :box do
end
```

- Names in HDLRuby are natively stored as Ruby symbols, but strings can also be used, e.g., `system("box") {}` is also valid.

3.4.2 Declaring a system with an interface

The interface of a system can be described anywhere in its body, but it is recommended to do it at its beginning. This is done by declaring input, output or inout signals of given data types as follows:

```
<data type>.<direction> <list of colon-preceded names>
```

For example, declaring a 1-bit input signal named `clk` can be declared as follows:

```
bit.input :clk
```

Now, since `bit` is the default data type in HDLRuby, it can be omitted as follows:

```
input :clk
```

The following is a more complete example: it is the code of a system describing an 8-bit data, 16-bit address memory whose interface includes a 1-bit input clock (`clk`), a 1-bit signal for selecting reading or writing access (`rwb`), a 16-bit address input (`addr`) and an 8-bit data inout — the remaining of the code describes the content and the behavior of the memory.

```
system :mem8_16 do
    input :clk, :rwb
    [15..0].input :addr
    [7..0].inout :data

    bit[7..0][2**16].inner :content

    par(clk.posedge) do
        hif(rwb) { data <= content[addr] }
        helse    { content[addr] <= data }
    end
end
```

3.4.3 Structural description in a system

In a system, structural descriptions consist of subsystems and interconnections among them.

A subsystem is obtained by instantiating an existing system as follows, where `<system name>` is the name of the system to instantiate (without any colon):

```
<system name> :<instance name>
```

For example, system `mem8_16` declared in the previous section can be instantiated as follows:

```
mem8_16 :mem8_16I
```

It is also possible to declare multiple instances of a same system at time as follows:

```
<system name> [list of colon-separated instance names]
```

For example, the following code declares two instances of system `mem8_16`:

```
mem8_16 [ :mem8_16I0, :mem8_16I1 ]
```

Interconnecting instances may require internal signals in the system. Such signals are declared using the `inner` direction. For example, the following code declares a 1-bit inner signal named `w1` and a 2-bit inner signal named `w2`:

```
inner :w1
[1..0].inner :w2
```

A connection between signals is done using the arrow operator `<=` as follows:

```
<destination> <= <source>
```

The `<destination>` must be a reference to a signal, and the `<source>` can be any expression.

For example the following code, connects signal `w1` to signal `ready` and signal `clk` to the first bit of signal `w2`:

```
ready <= w1
w2[0] <= clk
```

As another example, the following code connects to the second bit of `w2` the output of an AND operation between `clk` and `rst` as follows:

```
w2[1] <= clk & rst
```

The signals of an instance can be connected through the arrow operator too, provided they are properly referred to. One way to refer them is to use the dot operator `.` on the instance as follows:

```
<instance name>.<signal name>
```

For example, the following code connects signal `clk` of instance `mem8_16I` to signal `clk` of the current system:

```
mem8_16I.clk <= clk
```

It is also possible to connect multiple signals of an instance using the call operator `.()` as follows, where each target can be any expression:

```
<intance name>.(<signal name0>: <target0>, ...)
```

For example, the following code connects signals `clk` and `rst` of instance `mem8_16I` to signals `clk` and `rst` of the current system. As seen in this example, this method allows partial connection since the address and the data buses are not connected yet.

```
mem8_16I.(clk: clk, rst: rst)
```

This last connection method can be used directly while declaring an instance. For example, `mem8_16I` could have been declared and connected to `clk` and `rst` as follows:

```
mem8_16(:mem8_16I).(clk: clk, rst: rest)
```

To summarize this section, here is a structural description of a 16-bit memory made of two 8-bit memories (or equivalent) sharing the same address bus, and using respectively the lower and the higher 8-bits of the data bus:

```
system :mem16_16 do
   input :clk, :rwb
   [15..0].input :addr
   [15..0].inout :data
   mem8_16(:memL).(clk: clk, rwb: rwb, addr: addr, data: data[7..0])
   mem8_16(:memH).(clk: clk, rwb: rwb, addr: addr, data: data[15..8])
end
```

And here is an equivalent code using the arrow operator:

```
system :mem16_16 do
   input :clk, :rwb
   [15..0].input :addr
   [15..0].inout :data

   mem8_16 [:memL, :memH]

   memL.clk  <= clk
   memL.rwb  <= rwb
   memL.addr <= addr
   memL.data <= data[7..0]

   memH.clk  <= clk
   memH.rwb  <= rwb
   memH.addr <= addr
   memH.data <= data[15..8]
end
```

3.4.4 Scope in a system

3.4.4.1 General scopes

The signals of the interface of signals are accessible from anywhere in a HDLRuby description. This is not the case for inner signals and instances: they are accessible only within the scope they are declared in.

A scope is a region of the code where locally declared objects are accessible. Each system has its own scope that cannot be accessible from other part of an HDLRuby description. For example in the following code, signals `d` and `qb` as well as instance `dffI` cannot be accessed from outside system `div2`:

```
system :div2 do
   input :clk
   output :q

   inner :d, :qb
   d <= qb

   dff_full(:dffI).(clk: clk, d: d, q: q, qb: qb)
```

For robustness or, readability purpose, it is possible to add inner scope inside existing scope using the `sub` keyword as follows:

```
sub do
   <code>
end
```

For example, in the code bellow, signal `sig` is not accessible from outside the additional inner scope of system `sys`

```
system :sys do
   ...
   sub
      inner :sig
      <sig is accessible here>
   end
   <sig is not accessible from here>
end
```

It is also possible to add an inner scope within another inner scope as follows:

```
system :sys do
   ...
   sub
      inner :sig0
      <sig0 is accessible here>
      sub
         inner :sig1
         <sig0 and sig1 are accessible here>
      end
      <sig1 is not accessible here>
   end
   <neither sig0 nor sig1 are accessible here>
end
```

Within a same scope it is not possible to declared multiple signals or instances with a same name. However, it is possible to declare a signal or an instance with a name identical to one previously declared outside the scope: the inner-most declaration will be used.

3.4.4.2 Named scopes

It is possible to declare a scope with a name as follows:

```
sub :<name> do
   <code>
end
```

Where:

- `<name>` is the name of the scope.
- `<code>` is the code within the scope.

Contrary to the case of scopes without name, signals and instances declared within a named scope can be accessed outside using this name as reference. For example in the code bellow signal `sig` declared within scope named `scop` is accessed outside it using `scop.sig`:

```
sub :scop do
   inner :sig
   ...
end
...
scop.sig <= ...
```

3.4.5 Behavioral description in a system.

In a system, parallel behavioral descriptions are declared using the `par` keyword, and sequential behavioral descriptions are declared using the `seq` keyword. They are the equivalent of the Verilog HDL `always` blocks.

A behavior is made of a list of events (the sensitivity list) upon which it is activated, and a list of statements. A behavior is declared as follows:

```
par <list of events> do
   <list of statements>
end
```

In addition, it is possible to declare inner signals within an execution block. While such signals will be physically linked to the system, they are only accessible within the block they are declared into. This permits a tighter scope for signals, which improves the readability of the code and make it possible to declare several signals with identical names provided their respective scopes are different.

An event represents a specific change of state of a signal. For example, a rising edge of a clock signal named `clk` will be represented by event `clk.posedge`. In HDLRuby, events are obtained directly from expressions using the following methods: `posedge` for rising edge, `negedge` for falling edge, and `edge` for any edge. Events are described in more detail in section Events.

When one of the events of the sensitivity list of a behavior occurs, the behavior is executed, i.e., each of its statements is executed in sequence. A statement can represent a data transmission to a signal, a control flow, a nested execution block or the declaration of an inner signal (as stated earlier). Statements are described in more detail in section statements. In this section, we focus on the transmission statements and the block statements.

A transmission statement is declared using the arrow operator `<=` as follows:

```
<destination> <= <source>
```

The `<destination>` must be a reference to a signal, and the `<source>` can be any expression. A transmission has therefore exactly the same structure as a connection. However, its execution model is different: whereas a connection is continuously executed, a transmission is only executed during the execution of its block.

A block comprises a list of statements. It is used for adding hierarchy within a behavior. Blocks can be either parallel or sequential, i.e., their transmission statements are respectively non-blocking or blocking. By default, a top block is created when declaring a behavior, and it inherits from its execution mode. For example, with the following code, the top block of the behavior is sequential.

```
system :with_sequential_behavior do
   seq do
      <list of statements>
   end
end
```

It is possible to declare new blocks within an existing block. For declaring a sub block with the same execution mode as the upper one, the keyword `sub` is used. For example, the following code declare a sub block within a sequential block, with the same execution mode:

```
system :with_sequential_behavior do
   seq do
      <list of statements>
      sub do
         <list of statements>
      end
   end
end
```

A sub block can also have a different execution mode if it is declared using `seq`, that will force sequential execution mode, and `par` that will force parallel execution mode. For example in the following code, a parallel sub block is declared within a sequential one:

```
system :with_sequential_behavior do
   seq do
      <list of statements>
      par do
         <list of statements>
      end
   end
end
```

Sub blocks have their own scope so that it is possible to declare signals without colliding with existing ones. For example it is possible to declare three different inner signals all called `sig` as follows:

```
...
par(<sensibility list>) do
   inner :sig
   ...
   sub do
      inner :sig
      ...
      sub do
         inner :sig
         ...
      end
   end
   ...
end
```

To summarize this section, here is a behavioral description of a 16-bit shift register with asynchronous reset (`hif` and `helse` are keywords used for specifying hardware *if* and *else* control statements).

```
system :shift16 do
   input :clk, :rst, :din
   output :dout

   [15..0].inner :reg

   dout <= reg[15] # The output is the last bit of the register.

   par(clk.posedge) do
      hif(rst) { reg <= 0 }
      helse do
         reg[0] <= din
         reg[15..1] <= reg[14..0]
      end
   end
end
```

In the example above, the order of the transmission statements is of no consequence. This is not the case for the following example, that implements the same register using a sequential block. In this second example, putting statement `reg[0] <= din` in the last place would have lead to an invalid functionality for a shift register.

```
system :shift16 do
   input :clk, :rst, :din
   output :dout
   [15..0].inner :reg

   dout <= reg[15] # The output is the last bit of the register.

   par(clk.posedge) do
      hif(rst) { reg <= 0 }
      helse seq do
         reg[0] <= din
         reg <= reg[14..0]
      end
   end
end
```

Note:

- `helse seq` ensures that the block of the hardware else is in sequential mode.
- `hif(rst)` could also have been set to sequential mode as follows:

  ```
  hif rst, seq do
     reg <= 0
  end
  ```

- Parallel mode can be set the same way using `par`.

Finally, it often happens that a behavior contains only one statement. In such a case, the description can be shortened using the `at` operator as follows:

```
( statement ).at(<list of events>)
```

For example the following two code samples are equivalent:

```
par(clk.posedge) do
   a <= b+1
end
```

```
( a <= b+1 ).at(clk.posedge)
```

For sake of consistency, this operator can also be applied on block statements as follows, but it is probably less readable than the standard declaration of behaviors:

```
( seq do
     a <= b+1
     c <= d+2
  end ).at(clk.posedge)
```

3.5 Events

Each behavior of a system is associated with a list of events, called sensibility list, that specifies when the behavior is to be executed. An event is associated with a signal and represents the instants when the signal reaches a given state.

There are three kinds of event: positive edge events represent the instants when their corresponding signals vary from 0 to 1, negative edge events represent the instants when their corresponding signals vary from 1 to 0 and the change events represent the instants when their corresponding signals vary. Events are declared directly from the signals, using the `posedge` operator for positive edge, the `negedge` operator for negative edge, and the `change` operator for change. For example the following code declares 3 behaviors activated respectively on the positive edge, the negative edge and any change of the `clk` signal.

```
inner :clk

par(clk.posedge) do
...
end

par(clk.negedge) do
...
end

par(clk.change) do
...
end
```

Note: - The `change` keyword can be omitted.

3.6 Statements

Statements are the basic elements of a behavioral description. They are regrouped in blocks that specify their execution mode (parallel or sequential). There are four kinds of statements: the transmit statement that computes expressions and send the result to the target signals, the control statement that changes the execution flow of the behavior, the block statement (described earlier) and the inner signal declaration.

Note:

- There is actually a fifth type of statement, the time statement. It will be discussed in section Time.

3.6.1 Transmit statement

A transmit statement is declared using the arrow operator `<=` within a behavior. Its right value is the expression to compute and its left value is a reference to the target signals (or parts of signals), i.e., the signals (or part of signals) that receive the computation result.

For example following code transmits the value `3` to signal `s0` and the sum of the values of signals `i0` and `i1` to the first four bits of signal `s1`:

```
s0 <= 3
s1[3..0] <= i0 + i1
```

The comportment of a transmit statement depends on the execution mode of the enclosing block:

- If the mode is parallel, the target signals are updated when all the statements of the current block are processed.
- If the mode is sequential, the target signals are updated immediately after the right value of the statement is computed.

3.6.2 Control statements

There are only two possible control statements: the hardware if `hif` and the hardware case `hcase`.

3.6.2.1 hif

The `hif` construct is made of a condition and a block that is executed if and only if the condition is met. It is declared as follows, where the condition can be any expression:

```
hif <condition> do
   <block contents>
end
```

3.6.2.2 hcase

The `hcase` construct is made of an expression and a list of value-block pairs. A block is executed when the corresponding value is equal to the value of the expression of the `hcase`. This construct is declared as follows:

```
hcase <expression>
hwhen <value 0> do
   <block contents 0>
end
hwhen <value 1> do
```

```
   <block contents 1>
end
...
```

3.6.2.3 helse

It is possible to add a block that is executed when the condition of an `hif` is not met, or when no case matches the expression of a `hcase`, using the `helse` keyword as follows:

```
<hif or hcase construct>
helse do
   <block contents>
end
```

3.6.3 helsif

In addition to `helse` it is possible to set additional conditions to an `hif` using the `helsif` keyword as follows:

```
hif <condition 0> do
   <block contents 0>
end
helsif <condition 1> do
   <block contents 1>
end
...
```

3.6.3.1 About loops

HDLRuby does not include any hardware construct for describing loops. This might look poor compared to the other HDL, but it is important to understand that the current synthesis tools do not really synthesize hardware from such loops but instead preprocess them (e.g., unroll them) to synthesizable loopless hardware. In HDLRuby, such features are natively supported by the Ruby loop constructs (`for`, `while`, and so on), but also by advanced Ruby constructs like the enumerators (`each`, `times`, and so on).

Notes:

- HDLRuby being based on Ruby, it is highly recommended to avoid `for` or `while` constructs and to use enumerators instead.
- The Ruby `if` and `case` statements can also be used, but they do not represent nay hardware. Actually, they are executed when the corresponding system is instantiated. For example, the following code will display `Hello`

`world!` when the described system is instantiated, provided the generic parameter `param` is not nil.

```
system :say_hello do |param = nil|
   if param != nil then
      puts "Hello world!"
   end
end
```

3.7 Types

Each signal and expression is associated with a data type which describes the kind of value it can represent. In HDLRuby, the data types represent basically bit vectors associated with the way they should be interpreted, i.e., as bit strings, unsigned values, signed values, or hierarchical contents.

3.7.1 Type construction

There are five basic types, `bit`, `signed`, `unsigned`, `integer` and `float` that represent respectively single bit logical values, single bit unsigned values, single bit signed values, Ruby integer values and Ruby floating point values (double precision). The first three types are HW and support four-valued logic, whereas the two last ones are SW (but are compatible with HW) and only support boolean logic. Ruby integers can represent any element of **Z** (the mathematical integers), and have for that purpose a variable bit-width.

The other types are built from them using a combination of the two following type operators.

The vector operator `[]` is used for building types representing vectors of single or multiple other types. A vector whose elements have all the same type are declared as follows:

```
<type>[<range>]
```

The `<range>` of a vector type indicates the position of the starting and ending bits relatively to the radix point. If the position of the starting bit is on the left side of the range, the vector is big endian, otherwise it is little endian. Negative values in a range are also possible and indicate positions bellow the radix point. For example the following code describes a big endian fixed point type with 8 bits above the radix point and 4 bits bellow:

```
bit[7..-4]
```

A `n..0` range can also be abbreviated to `n+1`. For instance the two following types are identical:

```
bit[7..0]
bit[8]
```

A vector of multiple types, also called tuple, is declared as follows:

```
[<type 0>, <type 1>, ... ]
```

For example the following code declares the type of the vectors made of a 8-bit logical, a 16-bit signed and a 16-bit unsigned values:

```
[ bit[8], signed[16], unsigned[16] ]
```

The structure opertor `{}` is used for building hierarchical types made of named subtypes. This operator is used as follows:

```
{ <name 0>: <type 0>, <name 1>: <type 1>, ... }
```

For instance, the following code declares a hierarchical type with an 8-bit sub type named `header` and a 24-bit sub type named `data`:

```
{ header: bit[7..0], data: bit[23..0] }
```

3.7.2 Type definition

It is possible to give names to type constructs using the `typedef` keywords as follows:

```
<type construct>.typedef :<name>
```

For example the followings gives the name `char` to a 8-bit vector:

```
[7..0].typedef :char
```

After this statement, `char` can be used like any other type. For example, the following code sample declares a new input signal `sig` whose type is `char`:

```
char.input :sig
```

3.7.3 Type compatibility and conversion

HDLRuby is strongly typed which means that when two types are not compatible together, operations, connection or transmission between two expressions of these types are not permitted. The compatibility rules between two types are the followings:

1. The basic types are not compatible with one another.
2. Two vector types are compatible if and only if they have the same range and the same subtype (i.e., the type of their elements).
3. Hierarchical types are compatible if and only if they have the same subtypes names and each subtype of same name are compatible together.

The type an expression can be converted to one with another type using a conversion operator. Please refer to section Conversion operators for more details about such an operator.

Note:

- For the unambiguous cases, conversion operators will be implicitly added, please refer to section Implicit conversions for more details.

3.8 Expressions

Expressions are any construct that represents a value associated with a type. They include immediate values, reference to signals and operations among other expressions using expression operators.

3.8.1 Immediate values

The immediate values of HDLRuby can represent vectors of `bit`, `unsigned` and `signed`, and integer or floating point numbers. They are prefixed by a `_` character and include a header that indicates the vector type and the base used for representing the value, followed by a numeral representing the value. The bit width of a value is obtained by default from the width of the numeral, but it is also possible to enforce it in the header.

The vector type specifiers are the followings:

- `b`: `bit` type, can be omitted,
- `u`: `unsigned` type,
- `s`: `signed` type, the last figure is sign extended if required by the binary, octal and hexadecimal bases, but not for the decimal base.

The base specifiers are the followings:

- `b`: binary, can be omitted,
- `o`: octal,
- `d`: decimal,
- `h`: hexadecimal.

For example, all the following immediate values represent an 8-bit `100` (either in unsigned or signed representation):

```
_bb01100100
_b8b1100100
_b01100100
_01100100
```

```
_u8d100
_s8d100
_uh64
_s8o144
```

Notes:

- Ruby immediate values can also be used, their bit width is automatically adjusted to match the data type of the expression they are used in. Please notice this adjusting may change the value of the immediate, for example the following code will actually set `sig` to 4 instead of 100:

  ```
  [3..0].inner :sig
  sig <= 100
  ```

3.8.2 References

References are expressions used to designate signals, or a part of signals.

The most simple reference is simply the name of a signal. It designates the signal corresponding to this name in the current scope. For instance, in the following code, inner signal `sig0` is declared, and therefore the name *sig0* becomes a reference to designate this signal.

```
# Declaration of signal sig0.
inner :sig0

# Access to signal sig0 using a name reference.
sig0 <= 0
```

For designating a signal of another system, or a sub signal in a hierarchical signal, you can use the `.` operator as follows:

```
<parent name>.<signal name>
```

For example, in the following code, input signal `d` of system instance `dff0` is connected to sub signal `sub0` of hierarchical signal `sig`.

```
system :dff do
   input :clk, :rst, :d
   output :q

   par(clk.posedge) { q <= d & ~rst }
end

system :my_system do
   input :clk, :rst
   { sub0: bit, sub1: bit}.inner :sig
```

```
    dff(:dff0).(clk: clk, rst: rst)
    dff0.d <= sig.sub0
    ...
end
```

3.8.3 Expression operators

The following table gives a summary of the operators available in HDLRuby. More details are given for each group of operator in the subsequent sections.

Assignment operators (left-most operator of a statement):

symbol	description
:<=	connection, if outside behavior
:<=	transmission, if inside behavior

Arithmetic operators:

symbol	description
:+	addition
:-	subtraction
:*	multiplication
:/	division
:%	modulo
:**	power
:+@	positive sign
:-@	negation

Comparison operators:

symbol	description
:==	equality
:!=	difference
:>	greater than
:<	smaller than
:>=	greater or equal
:<=	smaller or equal

Logic and shift operators:

symbol	description
:&	bitwise / logical and
:	
:~	bitwise / logical not
:mux	multiplex
:<< / :ls	left shift
:>> / :rs	right shift
:lr	left rotate
:rr	right rotate

Conversion operators:

symbol	description
:to_bit	cast to bit vector
:to_unsigned	cast to unsigned vector
:to_signed	cast to signed vector
:to_big	cast to big endian
:to_little	cast to little endian
:reverse	reverse the bit order
:ljust	increase width from the left, preserves the sign
:rjust	increase width from the right, preserves the sign
:zext	zero extension, converts to unsigned if signed
:sext	sign extension, converts to sign

Selection /concatenation operators:

symbol	description
:[]	sub vector selection
:@[]	concatenation operator
:.	field selection

Notes:

- The operator precedence is the one of Ruby.
- Ruby does not allow to override the `&&`, the `||` and the `?:` operators so that they are not present in HDLRuby. Instead of the `?:` operator, HDLRuby provides the more general multiplex operator `mux`. However, HDLRuby does not provides any replacement for the `&&` and the `||` operators, please refer to section Logic operators for a justification about this issue.

3.8.3.1 Assignment operators

The assignment operators can be used with any type. They are actually the connection and the transmission operators, both being represented by `<=`.

Note:

- The first operator of a statement is necessarily an assignment operator, while the other occurrences of `<=` represent the usual `less than or equal to` operators.

3.8.3.2 Arithmetic operators

The arithmetic operators can only be used on vectors of `bit`, `unsigned` or `signed` values, `integer` or `float` values. These operators are `+`, `-`, `*`, `%` and the unary arithmetic operators are `-` and `+`. They have the same meaning as their Ruby equivalents.

3.8.3.3 Comparison operators

Comparison operators are the operators whose result is either true or false. In HDLRuby, true and false are represented by respectively `bit` value 1 and `bit` value 0. This operators are `==`, `!=`, `<`, `>`, `<=`, `>=` . They have the same meaning as their Ruby equivalents.

Notes:

- The `<`, `>`, `<=` and `>=` operators can only be used on vectors of `bit`, `unsigned` or `signed` values, `integer` or `float` values.
- When compared, values of type different from vector of `signed` and from `float` are considered as vectors of `unsigned`.

3.8.3.4 Logic and shift operators

In HDLRuby, the logic operators are all bitwise. For performing boolean computations it is necessary to use single bit values. The bitwise logic binary operators are `&`, `|`, and `^`, and the unary one is `~`. They have the same meaning as their Ruby equivalents.

Note: there is two reasons why there is no boolean operators

1. Ruby language does not support redefinition of the boolean operators
2. In Ruby, each value which is not `false` nor `nil` is considered to be true. This is perfectly relevant for software, but not for hardware where the basic data types are bit vectors. Hence, it seemed preferable to support boolean computation for one-bit values only, which can be done through bitwise operations.

The shift operators are `<<` and `>>` and have the same meaning as their Ruby equivalent. They do not change the bit width, and preserve the sign for `signed` values.

The rotation operators are `rl` and `rr` for respectively left and right bit rotations. Like the shifts, they do not change the bit width and preserve the sign for the `signed` values. However, since such operators do not exist in Ruby, they are actually used like methods as follows:

```
<expression>.rl(<other expression>)
<expression>.rr(<other expression>)
```

For example, for rotating left signal `sig` 3 times, the following code can be used:

```
sig.rl(3)
```

It is possible to perform other kinds of shifts or rotations using the selection and the concatenation operators. Please refer to section Concatenation and selection operators for more details about these operators.

3.8.3.5 Conversion operators

The conversion operators are used to change the type of an expression. There are two kinds of such operators: the type pun that do not change the raw value of the expression and the type cast that changes the raw value.

The type puns include `to_bit`, `to_unsigned` and `to_signed` that convert expressions of any type type to vectors of respectively `bit`, `unsigned` and `signed` elements. For example, the following code converts an expression of hierarchical type to an 8-bit signed vector:

```
[ up: signed[3..0], down: unsigned[3..0] ].inner :sig
sig.to_bit <= b01010011
```

The type casts change both the type and the value and are used to adjust the width of the types. They can only be applied to vectors of `bit`, `signed` or `unsinged` and can only increase the bit width (bit width can be truncated using the selection operator, please refer to the next section). These operators comprise the bit width conversions: `ljust`, `rjust`, `zext` and `sext`; they also comprise the bit endianness conversions: `to_big`, `to_little` and `reverse`.

More precisely, the bit width conversions operate as follows:

- `ljust` and `rjust` increase the size from respectively the left or the right side of the bit vector. They take as argument the width of the new type and the value (0 or 1) of the bits to add. For example the following code increases the size of `sig0` to 12 bits by adding 1 on the right:

  ```
  [7..0].inner :sig0
  [11..0].inner :sig1
  ```

```
sig0 <= 25
sig1 <= sig0.ljust(12,1)
```

- `zext` increases the size by adding several 0 bits on the most significant bit side, this side depending on the endianness of the expression. This conversion takes as argument the width of the resulting type. For example, the following code increases the size of `sig0` to 12 bits by adding 0 on the left:

  ```
  signed[7..0].inner :sig0
  [11..0].inner :sig1
  sig0 <= -120
  sig1 <= sig0.zext(12)
  ```

- `sext` increases the size by duplicating the most significant bit, the side of the extension depending on the endianness of the expression. This conversion takes as argument the width of the resulting type. For example, the following code increases the size of `sig0` to 12 bits by adding 1 on the right:

  ```
  signed[0..7].inner :sig0
  [0..11].inner :sig1
  sig0 <= -120
  sig1 <= sig0.sext(12)
  ```

Finally, the bit endianness conversions operate as follows:

- `to_big` ensures the type of the converted expression is big endian. If the initial expression is already big endian, it is left as is, otherwise its bits are reversed.
- `to_little` ensures the type of the converted expression is little endian. If the initial expression is already little endian, it is left as is, otherwise its bits are reversed.
- `reverse` always reverses the bit order of the expression.

3.8.3.6 Concatenation and selection operators

Concatenation and selection are done using the `[]` operator as follows:

- when this operator takes as arguments several expressions, it concatenates them. For example, the following code concatenates `sig0` to `sig1`:

  ```
  [3..0].inner :sig0
  [7..0].inner :sig1
  [11..0].inner :sig2
  sig0 <= 5
  sig1 <= 6
  sig2 <= [sig0, sig1]
  ```

- when this operator is applied to an expression of `bit`, `unsigned` or `signed` vector type while taking as argument a range, it selects the bits corresponding to this range. If only one bit is to select, the offset of this bit can be used instead. For example, the following code selects bits from 3 to 1 of `sig0` and bit 4 of `sig1`:

```
[7..0].inner :sig0
[7..0].inner :sig1
[3..0].inner :sig2
bit.inner    :sig3
sig0 <= 5
sig1 <= 6
sig2 <= sig0[3..1]
sig3 <= sig1[4]
```

3.8.3.7 Implicit conversions

When there is no ambiguity with bit vector types of same endianness, HDLRuby will automatically insert conversion operators when two types are not compatible with one another. The cases where such implicit conversions are applied are summarized in the following tables where:

- `operator` is the operator in use
- `result width` is the width of the result's type
- `result base` is the base type of the result's type
- `S` is the shortest operand
- `L` is the longest operand
- `S operand type` is the base type of the shortest operand
- `L operand type` is the base type of the longest operand
- `operand conversion` is the conversions added to make the operands compatible.
- `w` is the width of the operands after conversion
- `lw` is the width of the left operand's type before conversion
- `rw` is the width of the right operand's type before conversion

Additive and logical operators:

operator	result's width
<= (assign)	w (error is raised if L.width < R.width)
+, -	w+1
&, |, ^	w
==	1
<	1
>	1
<= (comp.)	1
>=	1

S operand base	L operand base	result base	operand conversion
bit	bit	bit	S.zext(L.width)
bit	unsigned	unsigned	S.zext(L.width).to_unsigned
bit	signed	signed	S.zext(max(S.width+1,L.width).to_signed
unsigned	bit	unsigned	S.zext(L.width), L.to_unsigned
unsigned	unsigned	unsigned	S.zext(L.width)
unsigned	signed	signed	S.zext(max(S.width+1,L.width).to_signed
signed	bit	signed	S.sext(L.width+1), L.zext(L.width+1).to_signed
signed	unsigned	signed	S.sext(L.width+1), L.zext(L.width+1).to_signed
signed	signed	signed	S.sext(L.width)

Multiplicative operators:

operator	result width
*	lw * rw
/	lw
%	rw
	rw
<< / ls	lw
>> / rs	lw
lr	lw
rr	lw

S operand base	L operand base	result base	operand conversion
bit	bit	bit	
bit	unsigned	unsigned	S.to_unsigned
bit	signed	signed	S.zext(S.width+1).to_signed
unsigned	bit	unsigned	L.to_unsigned
unsigned	unsigned	unsigned	
unsigned	signed	signed	S.zext(S.width).to_signed
signed	bit	signed	L.zext(L.width+1).to_signed
signed	unsigned	signed	L.zext(L.width+1).to_signed
signed	signed	signed	

3.9 Functions

3.9.1 HDLRuby functions

Similarly to Verilog HDL, HDLRuby provides function constructs for reusing code. HDLRuby functions are declared as follows:

```
function :<function name> do |<arguments>|
<code>
end
```

Where:

- `function name` is the name of the function.
- `arguments` is the list of arguments of the function.
- `code` is the code of the function.

Notes:

- Functions have their own scope, so that any declaration within a function is local. It is also forbidden to declare interface signals (input, output or inout) within a function.
- Similarly to Ruby proc objects, the last statement of a function's code serves as return value. For instance the following function returns `1` (in this example the function does not have any argument):

  ```
  function :one { 1 }
  ```

- Functions can accept any kind of object as argument, including variadic arguments or blocks of code as shown bellow with a function which apply the code passed as argument to all the variadic arguments of `args`:

  ```
  function :apply do |*args, &code|
     args.each { |arg| code.call(args) }
  end
  ```

  Such a function can be used for example for connecting a signal to a set of other signals as follows (where `sig` is connected to `x`, `y` and `z`): `ruby apply(x,y,z) { |v| v <= sig }`

A function can be invoked anywhere in the code using its name and passing its argument between parentheses as follows:

```
<function name>(<list of values>)
```

3.9.2 Ruby functions

HDLRuby functions are useful for reusing code, but they cannot interact with the code they are called in. For example, it is not possible to add interface signals through a function nor to modify a control statement (e.g., `hif`) with them. These high-level generic operations can however be performed using the functions of the Ruby language declared as follows:

```
def <function name>(<arguments>)
   <code>
end
```

Where:

- is the name of the function.
- is the list of arguments of the function.
- is the code of the function.

These functions are called the same way HDLRuby functions are called, but this operation actually pastes the code of the function as is within the code. Moreover, these function do not have any scope so that any inner signal or instance declared within them will actually added to the object they are invoked in.

For example, the following function will add input `in0` to any system where it is invoked:

```
def add_in0
   input :in0
end
```

This function can be used as follows:

```
system :sys do
   ...
   add_in0
   ...
end
```

As another example, following function will add an alternative code that generates a reset to a condition statement (`hif` or `hcase`):

```
def too_bad
   helse { $rst <= 1 }
end
```

This function can be used as follows:

```
system :sys do
   ...
   par do
      hif(sig == 1) do
         ...
      end
      too_bad
   end
end
```

Ruby functions can be compared to the macros of the C languages: they have more flexible since they actually edit the code they are invoked in, but are also dangerous to use. In general, it is not recommended to use them, unless when designing a library of generic code for HDLRuby.

3.10 Time

3.10.1 Time values

In HDLRuby, time values can be created using the time operators: `s` for seconds, `ms` for millisecond, `us` for microsecond, `ns` for nano second, `ps` for pico second and `fs` for femto second. For example, the followings are all indicating one second of time:

```
1.s
1000.ms
1000000.us
1000000000.ns
1000000000000.ps
1000000000000000.fs
```

3.10.2 Time behaviors and time statements

Similarly to the other HDL, HDLRuby provides specific statements that models the advance of time. These statements are not synthesizable and are used for simulating the environment of a hardware component. For sake of clarity, such statements are only allowed in explicitly non-synthesizable behavior declared using the `timed` keyword as follows.

```
timed do
   <statements>
end
```

A time behavior do not have any sensitivity list but it can include any statement supported by a standard behavior in addition to the time statements. There are two kinds of such statements:

- The `wait` statements: such a statement blocks the execution of the behavior for the amount of time given in argument. For example the following code waits 10ns before proceeding:

  ```
  wait(10.ns)
  ```

  This statement can also be abbreviated using the `!` operator as follows:

  ```
  !10.ns
  ```

- The `repeat` statements: such a statement takes as argument a time value and a block. The execution of the block is repeated until the delay given by the time value argument expires. For example, the following code executes repeatedly the inversion of the `clk` signal every 10 nanoseconds for 10 seconds (i.e., it simulates a clock signal for 10 seconds):

```
repeat(10.s) do
   !10.ns
   clk <= ~clk
end
```

3.10.3 Parallel and sequential execution

Time behaviors are by default sequential but they can include both parallel and sequential blocks. The execution semantic is the following:

- A sequential block in a time behavior is executed sequentially.
- A parallel block in a time behavior is executed in semi-parallel fashion as follows:
  1. Statements are grouped in sequence until a time statement is met.
  2. The grouped sequence are executed in parallel.
  3. The time statement is executed.
  4. The subsequent statements are processed the same way.

3.11 High-level programming features

3.11.1 Using Ruby in HDLRuby

Since HDLRuby is pure Ruby code, the constructs of Ruby can be freely used without any compatibility issue. Moreover, this Ruby code will not interfere with the synthesizability of the design. It is then possible to define Ruby classes, methods or modules whose execution generates constructs of HDLRuby.

3.11.2 Generic programming

3.11.2.1 Declaring

Systems can be declared with generic parameters. For that purpose, the parameters must be given as follows:

```
system :<system name> do |<list of generic parameters>|
   ...
end
```

For example, the following code describes an empty system with two generic parameters named respectively `a` and `b`:

```
system(:nothing) { |a,b| }
```

The generic parameters can be anything: values, data types, systems, Ruby variables, and so on. For example, the following system uses generic argument `t` as a type for an input signal, generic argument `w` as a bit range for an output signal and generic argument `s` as a system used for creating instance `sI` whose input and output signals `i` and `o` are connected respectively to signals `isig` and `osig`.

```
system :something do |t,w,s|
   t.input isig
   [w].output osig

   s :sI.(i: isig, o: osig)
end
```

It is also possible to use a variable number of generic parameters using the variadic operator `*` like in the following example. In this examples, `args` is an array containing an indefinite number of parameters.

```
system(:variadic) { |*args| }
```

3.11.2.2 Specializing

A generic system is specialized by invoking its name and passing as argument the values corresponding to the generic arguments as follows:

```
<system name>(<generic argument value's list>)
```

If less values are provided than the number of generic arguments, the system is partially specialized.

A specialized system can be used for inheritance. For example, assuming system `sys` has 2 generic arguments, it can be specialized and used for building system `subsys` as follows:

```
system :subsys, sys(1,2) do
   ...
end
```

This way of inheriting can only be done with fully specialized systems though. For partially specialized systems, `include` must be used instead. For example, if `sys` specialized with only one value, can be used in generic `subsys_gen` as follows:

```
system :subsys_gen do |param|
   include sys(1,param)
   ...
end
```

Note:

- In the example above, generic parameter `param` of `sybsys_gen` is used for specializing system `sys`.

3.11.2.3 Instantiating

When instantiating a system, the values of its generic parameters must be provided after the name of the new instance as follows:

```
<system name>(<generic argument value's list>).(:<instance name>)
```

If some arguments are omitted, an exception will be raised even if the arguments are not actually used in the system's body.

For example, in the previous section, system `nothing` did not used the generic arguments, but the following instantiation is invalid:

```
nothing(1).(:nothingI)
```

However the following is valid since a value is provided for each generic argument.

```
nothing(1,2).(:nothingI)
```

The validity of the generic value itself is checked when the body of the system is executed for generating the content of the instance. For the user's point of view, this happens at instantiation time, just like the check of the number of generic parameters' values. For example, the following instantiation of previous system `something` will raise an exception since the first generic value is not a type:

```
something(1,7..0).(:somethingI)
```

However, the following is valid:

```
something(bit,7..0).(:somethingI)
```

3.11.3 Inheritance

3.11.3.1 Basics

In HDLRuby, a system can inherit from the content of one or several other parent systems using the `include` command as follows: `include <list of systems>`. Such an include can be put anywhere in the body of a system, but the resulting content will be accessible only after this command.

For example, the following code describes first a simple D-FF, and then use it to described a FF with an additional reversed output (`qb`):

```
system :dff do
   input :clk, :rst, :d
   output :q

   par(clk.posedge) { q <= d & ~rst }
```

```
end

system :dff_full do
    output :qb

    include dff

    qb <= ~q
end
```

It is also possible to declare inheritance in a more object oriented fashion by listing the parents of a system just after declaring its name as follows:

```
system :<new system name>, <list of parent systems> do
   <additional system code>
end
```

For example, the following code is another to describe `dff_full`:

```
system :dff_full, dff do
   output :qb

   qb <= ~q
end
```

Note:

- As a matter of implementation, HDLRuby systems can be seen as set of methods used for accessing various constructs (signals, instances). Hence inheritance in HDLRuby is actually closer the Ruby mixin mechanism than to a true software inheritance.

3.11.3.2 About inner signals and system instances

By default, inner signals and instances of a parent system are not accessible by its child systems. They can be made accessible using the `export` keyword as follows: `export <symbol 0>, <symbol 1>, ...` . For example the following code exports signals `clk` and `rst` and instance `dff0` of system `exporter` so that they can be accessed in child system `importer`.

```
system :exporter do
   input :d
   inner :clk, :rst

   dff(:dff0).(clk: clk, rst: rst, d: d)

   export :clk, :rst, :dff0
end
```

```
system :importer, exporter do
   input :clk0, :rst0
   output :q

   clk <= clk0
   rst <= rst0
   dff0.q <= q
end
```

Note: - export takes as arguments the symbols (or the strings) representing the name of the components to export *and not* a reference to them. For instance, the following code is invalid:

```
system :exporter do
   input :d
   inner :clk, :rst

   dff(:dff0).(clk: clk, rst: rst, d: d)

   export clk, rst, dff0
end
```

3.11.3.3 Conflicts when inheriting

Signals and instances cannot be overridden, this is also the case for signals and instances accessible through inheritance. For example the following code is invalid since `rst` has already been defined in `dff`:

```
   system :dff_bad, dff do
      input :rst
   end
```

Conflicts among several inherited systems can be avoided by renaming the signals and instances that collide with one another as shown in the next section.

3.11.3.4 Shadowed signals and instances

It is possible in HDLRuby to declare a signal or an instance whose name is identical to one used in one of the included systems. In such a case, the corresponding construct of the included system is still present, but is not directly accessible even if exported, they are said to be shadowed.

In order to access to the shadowed signals or instances, a system must be reinterpreted as the relevant parent system using the `as` operator as follows: `as(system)`.

For example, in the following code signal `db` of system `dff_db` is shadowed by signal `db` of system `dff_shadow`, but is accessed using the `as` operator.

```
system :dff_db do
   input :clk,:rst,:d
   inner :db
   output :q

   db <= ~d
   (q <= d & ~rst).at(clk.posedge)
end

system :dff_shadow, dff_db do
   output :qb, :db

   db <= ~d
   qb <= as(dff_db).db
end
```

3.11.4 Opening a system

It is possible to pursue the definition of a system after it has been declared using the `open` methods as follows:

```
<system>.open do
   <additional system description>
end
```

For example `dff`, a system describing a D-FF, can be modified to have an inverted output as follows:

```
dff.open do
   output :qb

   qb <= ~q
end
```

3.11.5 Opening an instance

When there is a modification to apply to an instance, it is sometimes preferable to modify this sole instance rather than declaring a all new system to derivate the instance from. For that purpose it is possible to open an instance for modification as follows:

```
<instance name>.open do
   <additional description for the instance>
end
```

For example, an instance of the previous `dff` system can be extended with an inverted output as follows:

```
system :some_system do
   ...
   dff :dff0
   dff0.open do
      output :qb
      qb <= ~q
   end
   ...
end
```

3.11.6 Opening a single signal, or the totality of the signals

Contrary to systems and instances, signals dot not have any inner structure. Its however sometimes useful to add features to them (cf. hooks). Again, this is done using the `open` method as follows where signal `sig` is opened:

```
sig.open do
   <some code>
end
```

It is also possible to modify the totality of the signals of the design as follows:

```
signal.open do
   <some code>
end
```

3.11.7 Predicate and access methods

In order to get information about the current state of the hardware description HDLRuby provides the following predicates:

predicate name	predicate type	predicate meaning
`is_block?`	bit	tells if in execution block
`is_par?`	bit	tells if current parallel block is parallel
`is_seq?`	bit	tells if current parallel block is sequential
`is_clocked?`	bit	tells if current behavior is clocked (activated on a sole rising or falling edge of a signal)
`cur_block`	block	gets the current block
`cur_behavior`	behavior	gets the current behavior

predicate name	predicate type	predicate meaning
`cur_system`	system	gets the current system
`one_up`	block/system	gets the upper construct (block or system)
`last_one`	any	last declared construct

Several enumerators are also provided for accessing the internals of the current construct (in the current state):

enumerator name	accessed elements
`each_input`	input signals of the current system
`each_output`	output signals of the current system
`each_inout`	inout signals of the current system
`each_behavior`	behaviors of the current system
`each_event`	events of the current behavior
`each_block`	blocks of the current behavior
`each_statement`	statements of the current block
`each_inner`	inner signals of the current block (or system if not within a block)

3.11.8 Global signals

HDLRuby allows to declare global signals the same way system's signals are declared, but outside the scope of any system. After being declared, these signals are accessible directly from within any hardware construct.

In order to ease the design of standardized libraries, the following global signals are defined by default:

signal name	signal type	signal function
`$reset`	bit	global reset
`$resetb`	bit	global reset complement
`$clk`	bit	global clock
`$err`	bit	used to indicate if an error occurred
`$errno`	bit[7..0]	indicates the error number

Note:

- When not used, the global signals are discarded.

3.11.9 Defining and executing Ruby methods within HDLRuby constructs

Like with any Ruby program it is possible to define and execute methods anywhere in HDLRuby using the standard Ruby syntax. When defined, a method is attached to the enclosing HDLRuby construct. For instance, when defining a method when declaring a system, it will be usable within this system, while when defining a method outside any construct, it will be usable everywhere in the HDLRuby description.

A method can include HDLRuby code in which case the resulting hardware is appended to the current construct. For example the following code adds a connection between `sig0` and `sig1` in system `sys0`, and transmission between `sig0` and `sig1` in the behavior of `sys1`.

```
def some_arrow
   sig1 <= sig0
end

system :sys0 do
   input :sig0
   output :sig1

   some_arrow
end

system :sys1 do
   input :sig0, :clk
   output :sig1

   par(clk.posedge) do
      some_arrow
   end
end
```

Warning:

- In the above example, the semantic of `some_arrow` changes depending on where it is invoked from: within a system, it is a connection, within a behavior it is a transmission.
- Using Ruby methods for describing hardware might lead to weak code, for example the in following code, the method declares `in0` as input signal. Hence, while used in `sys0` no problems happens, an exception will be raised for `sys1` because a signal `in0` is already declare, and will also be raised for `sys2` because it is not possible to declare an input from within a behavior.

  ```
  def in_decl
  ```

```
   input :in0
end

system :sys0 do
   in_decl
end

system :sys1 do
   input :in0
   in_decl
end

system :sys2 do
   par do
      in_decl
   end
end
```

Like any other Ruby method, methods defined in HDLRuby support variadic arguments, named arguments and block arguments. For example, the following method can be used to connects a driver to multiple signals:

```
def mconnect(driver, *signals)
   signals.each do |signal|
      signal <= driver
   end
end

system :sys0 do
   input :i0
   input :o0, :o1, :o2, :o3

   mconnect(i0,o0,o1,o2,o3)
end
```

While requiring care, properly designed method can be very useful for clean code reuse. For example the following method allows to start the execution of a block after a given number of cycles:

```
def after(cycles,rst = $rst, &code)
   sub do
      inner :count
      hif rst == 1 do
         count <= 0
      end
      helse do
         hif count < cycles do
            count <= count + 1
         end
         helse do
            instance_eval(&code)
         end
      end
   end
end
```

In the code above:

- the default initialization of `rst` to `$rst` allows to reset the counter even if no such signal it provided as argument.
- `sub` ensures that the `count` signal do not conflict with another signal with the same name.
- the `instance_eval` keyword is a standard Ruby method that executes the block passed as argument in context.

The following is an example that switches a LED on after 1000000 clock cycles using the previously defined `after` ruby method:

```
system :led_after do
   output :led
   input :clk

   par(clk.posedge) do
      (led <= 0).hif($rst)
      after(100000) { led <= 1 }
   end
end
```

Note:

- Ruby's closure still applies in HDLRuby, hence, the block sent to `after` can use the signals and instances of the current block. Moreover, the signal declared in this method will not collide with them.

3.11.10 Dynamic description

When describing a system, it is possible to disconnect or to completely undefine a signal or an instance.

3.12 Extending HDLRuby

Like any Ruby classes, the constructs of HDLRuby can be dynamically extended. If it is not recommended to change their internal structure, it is possible to add methods to them for extension.

3.12.1 Extending HDLRuby constructs globally

By gobal extension of hardware constructs we actually mean the classical extension of Ruby classes by monkey patching the corresponding class. For example, it is possible to add a methods giving the number of signals in the interface of a system instance as follows:

```
class SystemI
   def interface_size
      return each_input.size + each_output.size + each_inout.size
   end
end
```

From there, the method `interface_size` can be used on any system instance as follows: `<system instance>.interface_size`.

The following table gives the class of each construct of HDLRuby.

construct	class
data type	Type
system	SystemT
scope	Scope
system instance	SystemI
signal	Signal
connection	Connection
par/seq	Behavior
timed	TimeBehavior
event	Event
par/seq/sub	Block
transmit	Transmit
hif	If
hcase	Case

3.12.2 Extending HDLRuby constructs locally

By local extension of a hardware construct, we mean that while the construct will be changed, all the other constructs will remain unchanged. This is achieved like in Ruby by accessing the eigen class using the `singleton_class` method, and extending it using the `class_eval` method. For example, with the following code, only system `dff` will respond to method `interface_size`:

```
dff.singleton_class.class_eval do
   def interface_size
      return each_input.size + each_output.size + each_inout.size
   end
end
```

It is also possible to extend locally an instance using the same methods. For example, with the following code, only instance `dff0` will respond to method `interface_size`:

```
dff :dff0

dff0.singleton_class.class_eval do
   def interface_size
      return each_input.size + each_output.size + each_inout.size
   end
end
```

Finally, it is possible to extend locally all the instances of a system using method `singleton_instance` in place of method `singleton_class`. For example, with the following code, all the instances of system `dff` will respond to method `interface_size`:

```
dff.singleton_instance.class_eval do
   def interface_size
      return each_input.size + each_output.size + each_inout.size
   end
end
```

3.12.3 Modifying the generation behavior

The main purpose of allowing global and local extensions for hardware constructs is to give the user the possibility implements its own synthesis methods. For example, one may want to implement some algorithm for a given kind of system. For that purpose, the user can define an abstract system (without any hardware content), that holds the specific algorithm as follows:

```
system(:my_base) {}

my_base.singleton_instance.class_eval do
   def my_generation
      <some code>
   end
end
```

Then, when this system named `my_base` is included into another system, this latter will inherit from the algorithms implemented inside method `my_generation` as shown in the following code:

```
system :some_system, my_base do
   <some system description>
end
```

However, when generation the low-level description of this system, code similar to the following will have to be written for applying `my_generation`:

```
some_system :instance0
instance0.my_generation
low = instance0.to_low
```

This can be avoided by redefining the `to_low` method as follows:

```
system(:my_base) {}

my_base.singleton_instance.class_eval do
   def my_generation
      <some code>
   end

   alias :_to_low :to_low
   def to_low
      my_generation
      _to_low
   end
end
```

This way, calling directly `to_low` will automatically use `my_generation`.

4 Standard library

The standard libraries are included into the module `Std`. They can be loaded as follows, where `<library name>` is the name of the library:

```
require 'std/<library name>'
```

After the libraries are loaded, the module `Std` must be included as follows:

```
include HDLRuby::High::Std
```

4.1 Channel

This library provides a unified interface to complex communication protocols. The interface consists of channel objects that can be written or read for transmission.

4.2 Clocks

This library provides utilities for an easier handling of clock synchronizations.

4.3 Counters

This library provides various construct with implicit counters for implementing synthesizable wait statements.

4.4 Pipeline

This library provides a construct for an easy description of pipeline architectures.

5 Development

After checking out the repo, run `bin/setup` to install dependencies. Then, run `rake test` to run the tests. You can also run `bin/console` for an interactive prompt that will allow you to experiment.

To install this gem onto your local machine, run `bundle exec rake install`. To release a new version, update the version number in `version.rb`, and then run `bundle exec rake release`, which will create a git tag for the version, push git commits and tags, and push the `.gem` file to rubygems.org.

6 Contributing

Bug reports and pull requests are welcome on GitHub at https://github.com/LovicGauthier/HDLRuby.

7 License

The gem is available as open source under the terms of the MIT License.