What’s new in MyHDL 0.6

Release 0.8

Jan Decaluwe

May 20, 2013

Contents

1 Conversion to VHDL **ii**
1.1 Rationale . . . ii
1.2 Advantages . . . ii
1.3 Solution description . . . ii
Approach . . . ii
User interface . . . iii
Type mapping . . . iii
Template transformation . . . iv

2 Conversion of lists of signals **v**

3 Conversion of test benches **v**
3.1 Background . . . v
3.2 Print statement . . . vi
3.3 Assert statement . . . vi
3.4 Delay objects . . . vii
3.5 Methodology notes . . . vii

4 Conversion output verification **vii**
4.1 Approach . . . vii
4.2 Verification interface . . . viii
4.3 HDL simulator registration . . . viii
Example: preregistered HDL simulators . . . ix

5 New modeling features **ix**
5.1 New signed() method for intbv . . . ix
5.2 always_comb and list of signals . . . x

6 Backwards incompatible changes **x**
6.1 Decorator usage . . . x
6.2 instances() function . . . x
6.3 Conversion of printing without a newline . . . x
Indexxi

Author Jan Decaluwe

1 Conversion to VHDL

1.1 Rationale

Since the MyHDL to Verilog conversion has been developed, a path to implementation from MyHDL is available. Given the widespread support for Verilog, it could thus be argued that there was no real need for a convertor to VHDL.

However, it turns out that VHDL is still very much alive and will remain so for the forseeable future. This is especially true for the FPGA market, which is especially interesting for MyHDL. It seems much more dynamic than the ASIC market. Moreover, because of the nature of FPGA's, FPGA designers may be more willing to try out new ideas.

To convince designers to use a new tool, it should integrate with their current design flow. That is why the MyHDL to VHDL convertor is needed. It should lower the threshold for VHDL designers to start using MyHDL.

1.2 Advantages

MyHDL to VHDL conversion offers the following advantages:

MyHDL integration in a VHDL-based design flow Designers can start using MyHDL and benefit from its power and flexibility, within the context of their proven design flow.

The convertor automates a number of hard tasks The convertor automates a number of tasks that are hard to do in VHDL directly. For example, when mixing `unsigned` and `signed` types it can be difficult to describe the desired behavior correctly. In contrast, a MyHDL designer can use the high-level `intbv` type, and let the convertor deal with type conversions and resizings.

MyHDL as an IP development platform The possibility to convert the same MyHDL source to equivalent Verilog and VHDL creates a novel application: using MyHDL as an IP development platform. IP developers can serve customers for both target languages from a single MyHDL code base. Moreover, MyHDL's flexibility and parametrizability make it ideally suited to this application.

1.3 Solution description

Approach

The approach followed to convert MyHDL code to VHDL is identical to the one followed for conversion to Verilog in previous MyHDL releases.

In particular, the MyHDL code analyzer in the convertor is identical for both target languages. The goal is that all MyHDL code that can be converted to Verilog can be converted to VHDL also, and vice versa. This has been achieved except for a few minor issues due to limitations of the target languages.

User interface

Conversion to VHDL is implemented by the following function in the `myhdl` package:

`toVHDL` (*func[, *args][, **kwargs]*)

Converts a MyHDL design instance to equivalent VHDL code. *func* is a function that returns an instance. `toVHDL()` calls *func* under its control and passes **args* and ***kwargs* to the call.

The return value is the same as the one returned by the call `func(*args, **kwargs)`. It can be assigned to an instance name. The top-level instance name and the basename of the Verilog output filename is `func.func_name` by default.

`toVHDL()` has the following attributes:

`toVHDL.`**`name`**

This attribute is used to overwrite the default top-level instance name and the basename of the VHDL output.

`toVHDL.`**`component_declarations`**

This attribute can be used to add component declarations to the VHDL output. When a string is assigned to it, it will be copied to the appropriate place in the output file.

Type mapping

In contrast to Verilog, VHDL is a strongly typed language. The convertor has to carefully perform type inferencing, and handle type conversions and resizings appropriately. To do this right, a well-chosen mapping from MyHDL types to VHDL types is crucial.

MyHDL types are mapped to VHDL types according to the following table:

MyHDL type	VHDL type	Notes
`int`	`integer`	
`bool`	`std_logic`	(1)
`intbv` with `min >= 0`	`unsigned`	(2)
`intbv` with `min < 0`	`signed`	(2)
`enum`	dedicated enumeration type	
`tuple` of `int`	mapped to case statement	(3)
`list` of `bool`	`array of std_logic`	
`list` of `intbv` with `min >= 0`	`array of unsigned`	(4)
`list` of `intbv` with `min < 0`	`array of signed`	(4)

Notes:

1. The VHDL `std_logic` type is defined in the standard VHDL package `IEEE.std_logic_1164`.
2. The VHDL `unsigned` and `signed` types used are those from the standard VHDL packages `IEEE.numeric_std`.
3. A MyHDL `tuple` of `int` is used for ROM inference, and can only be used in a very specific way: an indexing operation into the tuple should be the rhs of an assignment.
4. All list members should have identical value constraints.

The table as presented applies to MyHDL variables. They are mapped to VHDL variables (except for the case of a `tuple` of `int`).

The convertor also supports MyHDL signals that use `bool`, `intbv` or `enum` objects as their underlying type. These are mapped to VHDL signals with a type as specified in the table above.

The convertor supports MyHDL list of signals provided the underlying signal type is either `bool` or `intbv`. They may be mapped to a VHDL signal with a VHDL type as specified in the table. However, list of signals are not always mapped to a corresponding VHDL signal. See *Conversion of lists of signals* for more info.

Template transformation

There is a difference between VHDL and Verilog in the way in which sensitivity to signal edges is specified. In Verilog, edge specifiers can be used directly in the sensitivity list. In VHDL, this is not possible: only signals can be used in the sensitivity list. To check for an edge, one uses the `rising_edge()` or `falling_edge()` functions in the code.

MyHDL follows the Verilog scheme to specify edges in the sensitivity list. Consequently, when mapping such code to VHDL, it needs to be transformed to equivalent VHDL. This is an important issue because it affects all synthesizable templates that infer sequential logic.

We will illustrate this feature with some examples. This is the MyHDL code for a D flip-flop:

```
@always(clk.posedge)
def logic():
    q.next = d
```

It is converted to VHDL as follows:

```
DFF_LOGIC: process (clk) is
begin
    if rising_edge(clk) then
        q <= d;
    end if;
end process DFF_LOGIC;
```

The convertor can handle the more general case. For example, this is MyHDL code for a D flip-flop with asynchronous set, asynchronous reset, and preference of set over reset:

```
@always(clk.posedge, set.negedge, rst.negedge)
def logic():
    if set == 0:
        q.next = 1
    elif rst == 0:
        q.next = 0
    else:
        q.next = d
```

This is converted to VHDL as follows:

```
DFFSR_LOGIC: process (clk, set, rst) is
begin
    if (set = '0') then
        q <= '1';
    elsif (rst = '0') then
        q <= '0';
    elsif rising_edge(clk) then
        q <= d;
    end if;
end process DFFSR_LOGIC;
```

All cases with practical utility can be handled in this way. However, there are other cases that cannot be transformed to equivalent VHDL. The convertor will detect those cases and give an error.

2 Conversion of lists of signals

Lists of signals are useful for many purposes. For example, they make it easy to create a repetitive structure. Another application is the description of memory behavior.

The convertor output is non-hierarchical. That implies that all signals are declared at the top-level in VHDL or Verilog (as VHDL signals, or Verilog regs and wires.) However, some signals that are a list member at some level in the design hierarchy may be used as a plain signal at a lower level. For such signals, a choice has to be made whether to declare a Verilog memory or VHDL array, or a number of plain signal names.

If possible, plain signal declarations are preferred, because Verilog memories and arrays have some restrictions in usage and tool support. This is possible if the list syntax is strictly used outside generator code, for example when lists of signals are used to describe structure.

Conversely, when list syntax is used in some generator, then a Verilog memory or VHDL array will be declared. The typical example is the description of RAM memories.

The convertor in the previous MyHDL release had a severe restriction on the latter case: it didn't allow that, for a certain signal, list syntax was used in some generator, and plain signal syntax in another. This restriction, together with its rather obscure error message, has caused regular user complaints. In this release, this restriction has been lifted.

3 Conversion of test benches

3.1 Background

After conversion, we obviously want to verify that the VHDL or Verilog code works correctly. In previous MyHDL versions, the proposed verification technique was co-simulation: use the same MyHDL test bench to simulate the converted Verilog code and the original MyHDL code. While co-simulation works well, there are a number of issues with it:

- Co-simulation requires that the HDL simulator has an interface to its internal workings, such as `vpi` for Verilog and `vhpi` for VHDL.
- `vpi` for Verilog is well-established and available for open-source simulators such as Icarus and cver. However, `vhpi` for VHDL is much less established; it is unclear whether there is an open source solution that is powerful enough for MyHDL's purposes.
- Even though `vpi` is a "standard", there are differences between various simulators. Therefore, some customization is typically required per Verilog simulator.
- MyHDL co-simulation uses unix-style interprocess communication that doesn't work on Windows natively. This is an exception to the rest of MyHDL that should run on any Python platform.

The conclusion is that co-simulation is probably not a viable solution for the VHDL case, and it has some disadvantages for Verilog as well.

The proposed alternative is to convert the test bench itself, so that both test bench and design can be run in the HDL simulator. Of course, this is not a fully general solution either, as there are important restrictions on the kind of code that can be converted. However, with the additional features that have been developed, it should be a useful solution for verifying converted code.

3.2 Print statement

In previous MyHDL versions, `print` statement conversion to Verilog was supported in a quick and dirty way, by merely copying the format string without checks. With the advent of VHDL conversion, this has now been implemented more rigorously. This was necessary because VHDL doesn't work with format strings. Rather, the format string specification has to be converted to a sequence of VHDL `write` and `writeline` calls.

A `print` statement with multiple arguments:

```
print arg1, arg2, ...
```

is supported. However, there are restrictions on the arguments. First, they should be of one of the following forms:

```
arg
formatstring % arg
formatstring % (arg1, arg2, ...)
```

where `arg` is a `bool`, `int`, `intbv`, `enum`, or a `Signal` of these types.

The `formatstring` contains ordinary characters and conversion specifiers as in Python. However, the only supported conversion specifiers are `%s` and `%d`. Justification and width specification are thus not supported.

Printing without a newline:

```
print arg1 ,
```

is not supported. This is because the solution is based on `std.textio`. In VHDL `std.textio`, subsequent `write()` calls to a line are only printed upon a `writeline()` call. As a normal `print` implies a newline, the correct behavior can be guaranteed, but for a `print` without newline this is not possible. In the future, other techniques may be used and this restriction may be lifted.

3.3 Assert statement

An `assert` statement in Python looks as follow:

```
assert test_expression
```

It can be converted provided `test_expression` is convertible.

3.4 Delay objects

Delay objects are constructed as follows:

```
delay(t)
```

with `t` an integer. They are used in `yield` statements and as the argument of `always()` decorators, to specify delays. They can now be converted.

3.5 Methodology notes

The question is whether the conversion restrictions permit to develop sufficiently complex test benches. In this section, we present some insights about this.

The most important restrictions are the types that can be used. These remain "hardware-oriented" as before.

Even in the previous MyHDL release, the "convertible subset" was much wider than the "synthesis subset". For example, `while` and `raise` statement were already convertible.

The support for `delay()` objects is the most important new feature to write high-level models and test benches.

With the `print` statement, simple debugging can be done.

Of particular interest is the `assert` statement. Originally, `assert` statements were only intended to insert debugging assertions in code. Recently, there is a tendency to use them to write self-checking unit tests, controlled by unit test frameworks such as `py.test`. In particular, they are a powerful way to write self-checking test benches for MyHDL designs. As `assert` statements are now convertible, a whole test suite in MyHDL can be converted to an equivalent test suite in Verilog and VHDL.

Finally, the same techniques as for synthesizable code can be used to master complexity. In particular, any code outside generators is executed during elaboration, and therefore not considered in the conversion process. This feature can for example be used for complex calculations that set up constants or expected results. Furthermore, a tuple of ints can be used to hold a table of values that will be mapped to a case statement in Verilog and VHDL.

4 Conversion output verification

Note: This functionality is not needed in a typical design flow. It is only relevant to debug the MyHDL convertor itself.

4.1 Approach

To verify the convertor output, a methodology has been developed and implemented that doesn't rely on co-simulation and works for both Verilog and VHDL.

The solution builds on the features explained in section *Conversion of test benches*. The idea is basically to convert the test bench as well as the functional code. In particular, `print` statements in MyHDL are converted to equivalent statements in the HDL. The verification process consists of running both the MyHDL and the HDL simulation, comparing the simulation output, and reporting any differences.

The goal is to make the verification process as easy as possible. The use of `print` statements to debug a design is a very common and simple technique. The verification process itself is implemented in a single function with an interface that is identical to `toVHDL` and `toVerilog`.

As this is a native Python solution, it runs on any platform on which the HDL simulator runs. Moreover, any HDL simulator can be used as no `vpi` or `vhpi` capabilities are needed. Of course, per HDL simulator some customization is required to define the details on how it is used. This needs to be done once per HDL simulator and is fully under user control.

4.2 Verification interface

All functions related to conversion verification are implemented in the `myhdl.conversion` package. (To keep the `myhdl` namespace clean, they are not available from the `myhdl` namespace directly.)

`verify`(*func[, *args][, **kwargs]*)
Used like `toVHDL()`. It converts MyHDL code, simulates both the MyHDL code and the HDL code and reports any differences. The default HDL simulator is GHDL.

`analyze`(*func[, *args][, **kwargs]*)
Used like `toVHDL()`. It converts MyHDL code, and analyzes the resulting HDL. Used to verify whether the HDL output is syntactically correct.

The two previous functions have the following attribute:

`analyze.`**`simulator`**
Used to set the name of the HDL analyzer. GHDL is the default.

`verify.`**`simulator`**
Used to set the name of the HDL simulator. GHDL is the default.

4.3 HDL simulator registration

To be able to use a HDL simulator to verify conversions, it needs to be registered first. This is needed once per simulator (or rather, per set of analysis and simulation commands). Registering is done with the following function:

`registerSimulator`(*name=None, hdl=None, analyze=None, elaborate=None, simulate=None, offset=0*)
Registers a particular HDL simulator to be used by `verify()` and `analyze()`. *name* is the name of the simulator. *hdl* specifies the HDL: `"VHDL"` or `"Verilog"`. *analyze* is a command string to analyze the HDL source code. *elaborate* is a command string to elaborate the HDL code. This command is optional. *simulate* is a command string to simulate the HDL code. *offset* is an integer specifying the number of initial lines to be ignored from the HDL simulator output.

The command strings should be string templates that refer to the `topname` variable that specifies the design name. The templates can also use the `unitname` variable which is the lower case version of `topname`. The command strings can assume that a subdirectory called `work` is available in the current working directory. Analysis and elaboration results can be put there if desired.

The `analyze()` function runs the *analyze* command. The `verify()` function runs the *analyze* command, then the *elaborate* command if any, and then the *simulate* command.

The GHDL simulator is registered by default, but its registration can be overwritten if required.

Example: preregistered HDL simulators

A number of open-source HDL simulators are preregistered in the MyHDL distribution. If they are installed in the typical way, they are readily available for conversion verification. We will illustrate the registration process by showing the registrations of these simulators.

GHDL registration:

```
registerSimulator(
    name="GHDL",
    hdl="VHDL",
    analyze="ghdl -a --workdir=work pck_myhdl_%(version)s.vhd %(topname)s.vhd",
    elaborate="ghdl -e --workdir=work -o %(unitname)s_ghdl %(topname)s",
    simulate="ghdl -r %(unitname)s_ghdl"
    )
```

Icarus registration:

```
registerSimulator(
    name="icarus",
    hdl="Verilog",
    analyze="iverilog -o %(topname)s.o %(topname)s.v",
    simulate="vvp %(topname)s.o"
    )
```

cver registration:

```
registerSimulator(
    name="cver",
    hdl="Verilog",
    analyze="cver -c -q %(topname)s.v",
    simulate="cver -q %(topname)s.v",
    offset=3
    )
```

5 New modeling features

5.1 New signed() method for intbv

The `intbv` object has a new method `signed()` that implements sign extension. The extended bit is the msb bit of the bit representation of the object.

Clearly, this method only has an effect for `intbv` objects whose valid values are a finite range of positive integers.

This method can be converted to VHDL and Verilog.

5.2 always_comb and list of signals

In the previous MyHDL release, one could use lists of signals in an `always_comb()` block, but they were not considered to infer the sensitivity list. To several users, this was unexpected behavior, or even a bug.

In the present release, lists of signals are considered and the corresponding signals are added to the sensitivity list. The convertor to Verilog and VHDL is adapted accordingly.

6 Backwards incompatible changes

6.1 Decorator usage

The basic building block of a MyHDL design is a specialized Python generator.

In MyHDL 0.5, decorators were introduced to make it easier to create useful MyHDL generators. Moreover, they make the code clearer. As a result, they are now the de facto standard to describe hardware modules in MyHDL.

The implementation of certain tasks, such a signal tracing and conversion, can be simplified significantly if decorators are used to create the generators. These simplifications have now been adopted in the code. This means that decorator usage is assumed. Legacy code written for the mentioned purposes without decorators, can always be easily converted into code with decorators.

For pure modeling, it doesn't matter how generators are created and this will remain so. Therefore, designers can continue to experiment with innovative modeling concepts in the fullest generality.

6.2 instances() function

The `instances()` function can be used to automatically lookup and return the instances that are defined in a MyHDL module. In accordance with the section *Decorator usage*, its functionality has been changed. Only generators created by decorators are considered when looking up instances.

6.3 Conversion of printing without a newline

Printing without a newline (a print statement followed by a comma) is no longer supported by the convertor to Verilog. This is done to be compatible with the convertor to VHDL. Currently, the VHDL solution relies on `std.textio` and this implies that printing without a newline cannot be reliably converted.

Index

A

analyze() (in module myhdl), viii

C

component_declarations (toVHDL attribute), iii

N

name (toVHDL attribute), iii

R

registerSimulator() (in module myhdl), viii

S

simulator (analyze attribute), viii
simulator (verify attribute), viii

V

verify() (in module myhdl), viii