
What's new in MyHDL 0.7

Release 0.7

Jan Decaluwe

December 19, 2010

Contents

1	Conversion to VHDL/Verilog rewritten with the <code>ast</code> module	ii
2	Shadow signals	ii
2.1	Background	ii
2.2	Introducing shadow signals	ii
2.3	Concrete shadow signal subclasses	iii
2.4	Example	iii
3	Using <code>Signal</code> and <code>intbv</code> objects as indices	iv
4	New configuration attributes for conversion file headers	iv
5	Conversion propagates docstrings	v
6	New method to specify user-defined code	v
7	More powerful mapping to case statements	vi
8	Small changes	vi
8.1	<code>SignalType</code> as the base class of <code>Signals</code>	vi
8.2	Default value of <code>intbv</code> objects	vi
8.3	Combinatorial always blocks use blocking assignments	vi
8.4	No synthesis pragmas in Verilog output	vii
9	Python version	vii
10	Acknowledgments	vii
	Index	

Author Jan Decaluwe

1 Conversion to VHDL/Verilog rewritten with the `ast` module

The most important code change is a change that hopefully no-one will notice :-). The conversion code is now based on the `ast` package instead of the `compiler` package. Since Python 2.6, the `compiler` package is deprecated and replaced by the new `ast` package in the standard library. In Python 3, the `compiler` package is no longer available.

This was a considerable effort, spent on re-implementing existing behavior instead of on new interesting features. This was unfortunate, but it had to be done with priority. Now the conversion code is ready for the future.

2 Shadow signals

2.1 Background

Compared to HDLs such as VHDL and Verilog, MyHDL signals are less flexible for structural modeling. For example, slicing a signal returns a slice of the current value. For behavioral code, this is just fine. However, it implies that you cannot use such as `slice` in structural descriptions. In other words, a signal slice cannot be used as a signal. To solve these issues, a new concept was introduced: shadow signals.

The whole reasoning behind shadow signals is explained in more detail in [mep-105](#).

2.2 Introducing shadow signals

A shadow signal is related to another signal or signals as a shadow to its parent object. It follows any change to its parent signal or signals directly. However, it is not the same as the original: in particular, the user cannot assign to a shadow signal. Also, there may be a delta cycle delay between a change in the original and the corresponding change in the shadow signal. Finally, to be useful, the shadow signal performs some kind of transformation of the values of its parent signal or signals.

A shadow signal is convenient because it is directly constructed from its parent signals. The constructor infers everything that's needed: the type info, the initial value, and the transformation of the parent signal values. For simulation, the transformation is defined by a generator which is automatically created and added to the list of generators to be simulated. For conversion, the constructor defines a piece of dedicated Verilog and VHDL code which is automatically added to the convertor output.

Currently, three kinds of shadow signals have been implemented. The original inspiration for shadow signals was to have solution for structural slicing. This is the purpose of the `_SliceSignal` subclass. The opposite is also useful: a signal that shadows a composition of other signals. This is the purpose of the `ConcatSignal` subclass.

As often is the case, the idea of shadow signals had some useful side effects. In particular, I realized that the `TristateSignal` proposed in [mep-103](#) could be interpreted as a shadow signal of its drivers. With the machinery of the shadow signal in place, it became easier to support this for simulation and conversion.

2.3 Concrete shadow signal subclasses

```
class _SliceSignal (sig, left [, right=None ])
```

This class implements read-only structural slicing and indexing. It creates a new signal that shadows the slice or index of the parent signal *sig*. If the *right* parameter is omitted, you get indexing instead of slicing. Parameters *left* and *right* have the usual meaning for slice indices: in particular, *left* is non-inclusive but *right* is inclusive. *sig* should be appropriate for slicing and indexing, which means it should be based on `intbv` in practice.

The class constructor is not intended to be used explicitly. Instead, regular signals now have a call interface that returns a `_SliceSignal`:

```
Signal.__call__ (left [, right=None ])
```

Therefore, instead of:

```
s1 = _SliceSignal(sig, left, right)
```

you can do:

```
s1 = sig(left, right)
```

Obviously, the call interface was intended to be similar to a slicing interface. Of course, it is not exactly the same which may seem inconvenient. On the other hand, there are Python functions with a similar slicing functionality and a similar interface, such as the `range` function. Moreover, the call interface conveys the notion that something is being constructed, which is what really happens.

```
class ConcatSignal (*args)
```

This class creates a new signal that shadows the concatenation of its parent signal values. You can pass an arbitrary number of signals to the constructor. The signal arguments should be bit-oriented with a defined number of bits.

```
class TristateSignal (val)
```

This class is used to construct a new tristate signal. The underlying type is specified by the *val* parameter. It is a `Signal` subclass and has the usual attributes, with one exception: it doesn't support the `next` attribute. Consequently, direct signal assignment to a tristate signal is not supported. The initial value is the tristate value `None`. The current value of a tristate is determined by resolving the values from its drivers. When exactly one driver value is different from `None`, that is the resolved value; otherwise it is `None`. When more than one driver value is different from `None`, a contention warning is issued.

This class has the following method:

```
driver ()
```

Returns a new driver to the tristate signal. It is initialized to `None`. A driver object is an instance of a special `SignalType` subclass. In particular, its `next` attribute can be used to assign a new value to it.

2.4 Example

A typical application of shadow signals is conversion of list of signals to bit vectors and vice versa.

For example, suppose we have a system with N requesters that need arbitration. Each requester has a request output and a grant input. To connect them in the system, we can use list of signals. For example, a list of request signals can be constructed as follows:

```
request_list = [Signal(bool()) for i in range(M)]
```

Suppose that an arbiter module is available that is instantiated as follows:

```
arb = arbiter(grant_vector, request_vector, clock, reset)
```

The `request_vector` input is a bit vector that can have any of its bits asserted. The `grant_vector` is an output bit vector with just a single bit asserted, or none. Such a module is typically based on bit vectors because they are easy to process in RTL code. In MyHDL, a bit vector is modeled using the `intbv` type.

We need a way to “connect” the list of signals to the bit vector and vice versa. Of course, we can do this with explicit code, but shadow signals can do this automatically. For example, we can construct a `request_vector` as a `ConcatSignal` object:

```
request_vector = ConcatSignal(*reversed(request_list))
```

Note that we reverse the list first. This is done because the index range of lists is the inverse of the range of `intbv` bit vectors. By reversing, the indices correspond to the same bit.

The inverse problem exist for the `grant_vector`. It would be defined as follows:

```
grant_vector = Signal(intbv(0) [M:])
```

To construct a list of signals that are connected automatically to the bit vector, we can use the `Signal` call interface to construct `_SliceSignal` objects:

```
grant_list = [grant_vector(i) for i in range(M)]
```

Note the round brackets used for this type of slicing. Also, it may not be necessary to construct this list explicitly. You can simply use `grant_vector(i)` in an instantiation.

To decide when to use normal or shadow signals, consider the data flow. Use normal signals to connect to *outputs*. Use shadow signals to transform these signals so that they can be used as *inputs*.

3 Using Signal and intbv objects as indices

Previously, it was necessary convert `Signal` and `intbv` objects explicitly to `int` when using them as indices for indexing and slicing. This conversion is no longer required; the objects can be used directly. The corresponding classes now have an `__index__()` method that takes care of the type conversion automatically. This feature is fully supported by the VHDL/Verilog convertor.

4 New configuration attributes for conversion file headers

New configuration attributes are available to control the file headers of converted output files.

`toVerilog.no_myhdl_header`

Specifies that MyHDL conversion to Verilog should not generate a default header. Default value is *False*.

`toVHDL.no_myhdl_header`

Specifies that MyHDL conversion to VHDL should not generate a default header. Default value is *False*.

`toVerilog.header`

Specifies an additional custom header for Verilog output.

`toVHDL.header`

Specifies an additional custom header for VHDL output.

The value for the custom headers should be a string that is suitable for the standard `string.Template` constructor. A number of variables (indicated by a `$` prefix) are available for string interpolation. For example, the standard header is defined as follows:

```
myhdl_header = """\
-- File: $filename
-- Generated by MyHDL $version
-- Date: $date
"""
```

The same interpolation variables are available in custom headers.

5 Conversion propagates docstrings

The convertor now propagates comments under the form of Python docstrings.

Docstrings are typically used in Python to document certain objects in a standard way. Such “official” docstrings are put into the converted output on an appropriate locations. The convertor supports official docstrings for the top level module and for generators.

Within generators, “nonofficial” docstrings are propagated also. These are strings (triple quoted by convention) that can occur anywhere between statements.

Regular Python comments are ignored by the Python parser, and they are not present in the parse tree. Therefore, these are not propagated. With docstrings, you have an elegant way to specify which comments should be propagated and which not.

6 New method to specify user-defined code

The current way to specify user-defined code for conversion is through the `__vhdl__` and `__verilog__` hooks. This method has a number of disadvantages.

First, the use of “magic” variables (whose names start and end with double underscores) was a bad choice. According to Python conventions, such variables should be reserved for the Python language itself. Moreover, when new hooks would become desirable, we would have to specify additional magic variables.

A second problem that standard Python strings were used to define the user-defined output. These strings can contain the signal names from the context for interpolation. Typically, these are multiple-line strings that may be quite lengthy. When something goes wrong with the

string interpolation, the error messages may be quite cryptic as the line and column information is not present.

For these reasons, a new way to specify user-defined code has been implemented that avoids these problems.

The proper way to specify meta-information of a function is by using function attributes. Suppose a function `<func> ()` defines a hardware module. We can now specify user-defined code for it with the following function attributes:

`<func>.vhdl_code`

A template string for user-defined code in the VHDL output.

`<func>.verilog_code`

A template string for user-defined code in the Verilog output.

When such a function attribute is defined, the normal conversion process is bypassed and the user-defined code is inserted instead. The template strings should be suitable for the standard `string.Template` constructor. They can contain interpolation variables (indicated by a `$` prefix) for all signals in the context. Note that the function attribute can be defined anywhere where `<func> ()` is visible, either outside or inside the function itself.

The old method for user-defined code is still available but is deprecated and will be unsupported in the future.

7 More powerful mapping to case statements

The convertor has become more powerful to map if-then-else structures to case statements in VHDL and Verilog. Previously, only if-then-else structures testing enumerated types were considered. Now, integer tests are considered also.

8 Small changes

8.1 `SignalType` as the base class of Signals

`Signal()` has become a function instead of a class. It returns different `Signal` subtypes depending on parameters. This implies that you cannot use `Signal()` for type checking.

The base type of all Signals is now `SignalType`. This type can be used to check whether an object is a `Signal` instance.

8.2 Default value of `intbv` objects

The default initial value of an `intbv` object has been changed from `None` to `0`. Though this is backward-incompatible, the `None` value never has been put to good use, so this is most likely not an issue.

8.3 Combinatorial always blocks use blocking assignments

The convertor now uses blocking assignments for combinatorial always blocks in Verilog. This is in line with generally accepted Verilog coding conventions.

8.4 No synthesis pragmas in Verilog output

The convertor no longer outputs the synthesis pragmas `full_case` and `parallel_case`. These pragmas do more harm than good as they can cause simulation-synthesis mismatches. Synthesis tools should be able to infer the appropriate optimizations from the source code directly.

9 Python version

MyHDL 0.7 requires Python 2.6, mainly because of its dependency on the new `ast` package.

10 Acknowledgments

Several people have contributed to MyHDL 0.7 by giving feedback, making suggestions, fixing bugs and implementing features. In particular, I would like to thank Benoit Allard, Günter Dannoritzer, Tom Dillon, Knut Eldhuset, Angel Ezquerra, Christopher Felton, and Jian LUO.

Thanks to Francesco Balau for packaging MyHDL for Ubuntu.

I would also like to thank [Easics](#) for the opportunity to use MyHDL in industrial projects.

Index

Symbols

`_SliceSignal` (built-in class), [iii](#)

`__call__()` (Signal method), [iii](#)

C

`ConcatSignal` (built-in class), [iii](#)

D

`driver()`, [iii](#)

H

`header` (toVerilog attribute), [v](#)

`header` (toVHDL attribute), [v](#)

N

`no_myhdl_header` (toVerilog attribute), [iv](#)

`no_myhdl_header` (toVHDL attribute), [v](#)

T

`TristateSignal` (built-in class), [iii](#)