Introduction to MyHDL

A basic MyHDL simulation

We will introduce MyHDL with a classic `Hello World` style example. All example code can be found in the distribution directory under `example/manual/`. Here are the contents of a MyHDL simulation script called `hello1.py`:

```
from myhdl import block, delay, always, now

@block
def HelloWorld():

    @always(delay(10))
    def say_hello():
        print("%s Hello World!" % now())

    return say_hello


inst = HelloWorld()
inst.run_sim(30)
```

When we run this simulation, we get the following output:

```
$ python hello1.py
10 Hello World!
20 Hello World!
30 Hello World!
<class 'myhdl._SuspendSimulation'>: Simulated 30 timesteps
```

The first line of the script imports a number of objects from the `myhdl` package. In Python we can only use identifiers that are literally defined in the source file.

Then, we define a function called **`HelloWorld`**. In MyHDL, a hardware module is modeled by a function decorated with the **`block`** decorator. The name *block* was chosen to avoid confusion with the Python concept of a module. We will use this terminology further on.

The parameter list of the **`HelloWorld`** function is used to define the interface of the hardware block. In this first example, the interface is empty.

Inside the top level function we declare a local function called `say_hello` that defines the desired behavior. This function is decorated with an `always` decorator that has a `delay` object as its parameter. The meaning is that the function will be executed whenever the specified delay interval has expired.

Behind the curtains, the `always` decorator creates a Python *generator* and reuses the name of the decorated function for it. Generators are the fundamental objects in MyHDL, and we will say much more about them further on.

Finally, the top level function returns the `say_hello` generator. This is the simplest case of the basic MyHDL code pattern to define the contents of a hardware block. We will describe the general case further on.

In MyHDL, we create an *instance* of a hardware block by calling the corresponding function. The `block` decorator make sure that the return value is actually an instance of a block class, with a useful API. In the example, variable `inst` refers to a `HelloWorld` block instance.

To simulate the instance, we use its `run_sim` method. We can use it to run the simulation for the desired amount of timesteps.

Signals and concurrency

An actual hardware design is typically massively concurrent, which means that a large amount of functional units are running in parallel. MyHDL supports this behavior by allowing an arbitrary number of concurrently running generators.

With concurrency comes the problem of deterministic communication. Hardware languages use special objects to support deterministic communication between concurrent code. In particular, MyHDL has a `Signal` object which is roughly modeled after VHDL signals.

We will demonstrate signals and concurrency by extending and modifying our first example. We define a hardware block that contains two generators, one that drives a clock signal, and one that is sensitive to a positive edge on a clock signal:

```
from myhdl import block, Signal, delay, always, now

@block
def HelloWorld():

    clk = Signal(0)

    @always(delay(10))
    def drive_clk():
        clk.next = not clk

    @always(clk.posedge)
    def say_hello():
        print("%s Hello World!" % now())

    return drive_clk, say_hello


inst = HelloWorld()
inst.run_sim(50)
```

The clock driver function **`clk_driver`** drives the clock signal. If defines a generator that continuously toggles a clock signal after a certain delay. A new value of a signal is specified by assigning to its `next` attribute. This is the MyHDL equivalent of the VHDL signal assignment and the Verilog non-blocking assignment.

The **`say_hello`** function is modified from the first example. It is made sensitive to a rising edge of the clock signal, specified by the `posedge` attribute of a signal. The edge specifier is the argument of the `always` decorator. As a result, the decorated function will be executed on every rising clock edge.

The `clk` signal is constructed with an initial value `0`. One generator drives it, the other is sensitive to it. The result of this communication is that the generators run in parallel, but that their actions are coordinated by the clock signal.

When we run the simulation, we get:

```
$ python hello2.py
10 Hello World!
30 Hello World!
50 Hello World!
<class 'myhdl._SuspendSimulation'>: Simulated 50 timesteps
```

Parameters, ports and hierarchy

We have seen that MyHDL uses functions to model hardware blocks. So far these functions did not have parameters. However, to create general, reusable blocks we will need parameters. For example, we can create a clock driver block as follows:

```
from myhdl import block, delay, instance


@block
def ClkDriver(clk, period=20):

    lowTime = int(period / 2)
    highTime = period - lowTime

    @instance
    def drive_clk():
        while True:
            yield delay(lowTime)
            clk.next = 1
            yield delay(highTime)
            clk.next = 0

    return drive_clk
```

The block encapsulates a clock driver generator. It has two parameters.

The first parameter is *clk* is the clock signal. A asignal parameter is MyHDL's way to model a dfn:port:. The second parameter is the clock *period*, with a default value of `20`.

As the low time of the clock may differ from the high time in case of an odd period, we cannot use the `always` decorator with a single delay value anymore. Instead, the `drive_clk` function is now a generator function with an explicit definition of the desired behavior. It is decorated with the `instance` decorator. You can see that `drive_clk` is a generator function because it contains `yield` statements.

When a generator function is called, it returns a generator object. This is basically what the `instance` decorator does. It is less sophisticated than the `always` decorator, but it can be used to create a generator from any local generator function.

The `yield` statement is a general Python construct, but MyHDL uses it in a specific way. It has a similar meaning as the wait statement in VHDL: the statement suspends execution of a generator, and its clauses specify the conditions on which the generator should wait before resuming. In this case, the generator waits for a certain delay.

Note that to make sure that the generator runs “forever”, we wrap its behavior in a `while True` loop.

Similarly, we can define a general `Hello` function as follows:

```
from myhdl import block, always, now


@block
def Hello(clk, to="World!"):

    @always(clk.posedge)
    def say_hello():
        print("%s Hello %s" % (now(), to))

    return say_hello
```

We can create any number of instances by calling the functions with the appropriate parameters. Hierarchy can be modeled by defining the instances in a higher-level function, and returning them. This pattern can be repeated for an arbitrary number of hierarchical levels. Consequently, the general definition of a MyHDL instance is recursive: an instance is either a sequence of instances, or a generator.

As an example, we will create a higher-level function with four instances of the lower-level functions, and simulate it:

```
from myhdl import block, Signal

from ClkDriver import ClkDriver
from Hello import Hello


@block
def Greetings():

    clk1 = Signal(0)
    clk2 = Signal(0)

    clkdriver_1 = ClkDriver(clk1)  # positional and default association
    clkdriver_2 = ClkDriver(clk=clk2, period=19)  # named association
    hello_1 = Hello(clk=clk1)  # named and default association
    hello_2 = Hello(to="MyHDL", clk=clk2)  # named association

    return clkdriver_1, clkdriver_2, hello_1, hello_2


inst = Greetings()
inst.run_sim(50)
```

As in standard Python, positional or named parameter association can be used in instantiations, or a mix of both [1]. All these styles are demonstrated in the example above. Named association can be very useful if there are a lot of parameters, as the argument order in the call does not matter in that case.

The simulation produces the following output:

```
$ python greetings.py
9 Hello MyHDL
10 Hello World!
28 Hello MyHDL
30 Hello World!
47 Hello MyHDL
50 Hello World!
<class 'myhdl._SuspendSimulation'>: Simulated 50 timesteps
```

Terminology review

Some commonly used terminology has different meanings in Python versus hardware design. For a good understanding, it is important to make these differences explicit.

A *module* in Python refers to all source code in a particular file. A module can be reused by other modules by importing it. In hardware design on the other hand, a module typically refers to a reusable unit of hardware with a well defined interface. Because these meanings are so different, the terminology chosen for a hardware module in MyHDL is *block* instead, as explained earlier in this chapter.

A hardware block can can be reused in another block by *instantiating* it.

An *instance* in Python (and other object-oriented languages) refers to the object created by a class constructor. In hardware design, an instance is a particular incarnation of a hardware block, created by *instantiating* the block. In MyHDL, such as block instance is actually an instance of a particular class. Therefore, the two meanings are not exactly the same, but they coincide nicely.

Normally, the meaning the words “block” and “instance” should be clear from the context. Sometimes, we qualify them with the words “hardware” or “MyHDL” for clarity.

Some remarks on MyHDL and Python

To conclude this introductory chapter, it is useful to stress that MyHDL is not a language in itself. The underlying language is Python, and MyHDL is implemented as a Python package called `myhdl`. Moreover, it is a design goal to keep the `myhdl` package as minimalistic as possible, so that MyHDL descriptions are very much “pure Python”.

To have Python as the underlying language is significant in several ways:

- Python is a very powerful high level language. This translates into high productivity and elegant solutions to complex problems.
- Python is continuously improved by some very clever minds, supported by a large user base. Python profits fully from the open source development model.
- Python comes with an extensive standard library. Some functionality is likely to be of direct interest to MyHDL users: examples include string handling, regular expressions,

random number generation, unit test support, operating system interfacing and GUI development. In addition, there are modules for mathematics, database connections, networking programming, internet data handling, and so on.

Summary and perspective

Here is an overview of what we have learned in this chapter:

- Generators are the basic building blocks of MyHDL models. They provide the way to model massive concurrency and sensitivity lists.
- MyHDL provides decorators that create useful generators from local functions and a decorator to create hardware blocks.
- Hardware structure and hierarchy is described with Python functions, decorated with the `block` decorator.
- `Signal` objects are used to communicate between concurrent generators.
- A block instance provides a method to simulate it.

These concepts are sufficient to start modeling and simulating with MyHDL.

However, there is much more to MyHDL. Here is an overview of what can be learned from the following chapters:

- MyHDL supports hardware-oriented types that make it easier to write typical hardware models. These are described in Chapter Hardware-oriented types.
- MyHDL supports sophisticated and high level modeling techniques. This is described in Chapter High level modeling.
- MyHDL enables the use of modern software verification techniques, such as unit testing, on hardware designs. This is the topic of Chapter Unit testing.
- It is possible to co-simulate MyHDL models with other HDL languages such as Verilog and VHDL. This is described in Chapter Co-simulation with Verilog.
- Last but not least, MyHDL models can be converted to Verilog or VHDL, providing a path to a silicon implementation. This is the topic of Chapter Conversion to Verilog and VHDL.

Footnotes

[1] All positional parameters have to go before any named parameter.