MyHDL manual

Release 1.0dev

Jan Decaluwe

April 26, 2017

Contents

CHAPTER 1

Overview

The goal of the MyHDL project is to empower hardware designers with the elegance and simplicity of the Python language.

MyHDL is a free, open-source package for using Python as a hardware description and verification language. Python is a very high level language, and hardware designers can use its full power to model and simulate their designs. Moreover, MyHDL can convert a design to Verilog or VHDL. This provides a path into a traditional design flow.

Modeling

Python's power and clarity make MyHDL an ideal solution for high level modeling. Python is famous for enabling elegant solutions to complex modeling problems. Moreover, Python is outstanding for rapid application development and experimentation.

The key idea behind MyHDL is the use of Python generators to model hardware concurrency. Generators are best described as resumable functions. MyHDL generators are similar to always blocks in Verilog and processes in VHDL.

A hardware module (called a *block* in MyHDL terminology) is modeled as a function that returns generators. This approach makes it straightforward to support features such as arbitrary hierarchy, named port association, arrays of instances, and conditional instantiation. Furthermore, MyHDL provides classes that implement traditional hardware description concepts. It provides a signal class to support communication between generators, a class to support bit oriented operations, and a class for enumeration types.

Simulation and Verification

The built-in simulator runs on top of the Python interpreter. It supports waveform viewing by tracing signal changes in a VCD file.

With MyHDL, the Python unit test framework can be used on hardware designs. Although unit testing is a popular modern software verification technique, it is still uncommon in the hardware design world.

MyHDL can also be used as hardware verification language for Verilog designs, by co-simulation with traditional HDL simulators.

Conversion to Verilog and VHDL

Subject to some limitations, MyHDL designs can be converted to Verilog or VHDL. This provides a path into a traditional design flow, including synthesis and implementation. The convertible subset is restricted, but much wider than the standard synthesis subset. It includes features that can be used for high level modeling and test benches.

The converter works on an instantiated design that has been fully elaborated. Consequently, the original design structure can be arbitrarily complex. Moreover, the conversion limitations apply only to code inside generators. Outside generators, Python's full power can be used without compromising convertibility.

Finally, the converter automates a number of tasks that are hard in Verilog or VHDL directly. A notable feature is the automated handling of signed arithmetic issues.

CHAPTER 2

Background information

Prerequisites

You need a basic understanding of Python to use MyHDL. If you don't know Python, don't worry: it is is one of the easiest programming languages to learn[1]. Learning Python is one of the best time investments that engineering professionals can make[2].

For starters, http://docs.python.org/tutorial is probably the best choice for an on-line tutorial. For alternatives, see http://wiki.python.org/moin/BeginnersGuide.

A working knowledge of a hardware description language such as Verilog or VHDL is helpful.

Code examples in this manual are sometimes shortened for clarity. Complete executable examples can be found in the distribution directory at `example/manual/`.

A small tutorial on generators

Generators were introduced in Python 2.2. Because generators are the key concept in MyHDL, a small tutorial is included here.

Consider the following nonsensical function:

```
def function():
    for i in range(5):
        return i
```

You can see why it doesn't make a lot of sense. As soon as the first loop iteration is entered, the function returns:

```
>>> function()
0
```

[1] You must be bored by such claims, but in Python's case it's true.
[2] I am not biased.

Returning is fatal for the function call. Further loop iterations never get a chance, and nothing is left over from the function call when it returns.

To change the function into a generator function, we replace `return` with `yield`:

```
def generator():
    for i in range(5):
        yield i
```

Now we get:

```
>>> generator()
<generator object at 0x815d5a8>
```

When a generator function is called, it returns a generator object. A generator object supports the iterator protocol, which is an expensive way of saying that you can let it generate subsequent values by calling its `next` method:

```
>>> g = generator()
>>> g.next()
0
>>> g.next()
1
>>> g.next()
2
>>> g.next()
3
>>> g.next()
4
>>> g.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
StopIteration
```

Now we can generate the subsequent values from the for loop on demand, until they are exhausted. What happens is that the `yield` statement is like a `return`, except that it is non-fatal: the generator remembers its state and the point in the code when it yielded. A higher order agent can decide when to get the next value by calling the generator's `next` method. We say that generators are *resumable functions*.

If you are familiar with hardware description languages, this may ring a bell. In hardware simulations, there is also a higher order agent, the Simulator, that interacts with such resumable functions; they are called *processes* in VHDL and *always blocks* in Verilog. Similarly, Python generators provide an elegant and efficient method to model concurrency, without having to resort to some form of threading.

The use of generators to model concurrency is the first key concept in MyHDL. The second key concept is a related one: in MyHDL, the yielded values are used to specify the conditions on which the generator should wait before resuming. In other words, `yield` statements work as general sensitivity lists.

About decorators

Python 2.4 introduced a feature called decorators. MyHDL takes advantage of this feature by defining a number of decorators that facilitate hardware descriptions. However, some users

may not yet be familiar with decorators. Therefore, an introduction is included here.

A decorator consists of special syntax in front of a function declaration. It refers to a decorator function. The decorator function automatically transforms the declared function into some other callable object.

A decorator function `deco` is used in a decorator statement as follows:

```
@deco
def func(arg1, arg2, ...):
    <body>
```

This code is equivalent to the following:

```
def func(arg1, arg2, ...):
    <body>
func = deco(func)
```

Note that the decorator statement goes directly in front of the function declaration, and that the function name `func` is automatically reused for the final result.

MyHDL uses decorators to create ready-to-simulate generators from local function definitions. Their functionality and usage will be described extensively in this manual.

CHAPTER 3

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".

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

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*.

CHAPTER 4

Hardware-oriented types

The `intbv` class

Hardware design involves dealing with bits and bit-oriented operations. The standard Python type `int` has most of the desired features, but lacks support for indexing and slicing. For this reason, MyHDL provides the *`intbv`* class. The name was chosen to suggest an integer with bit vector flavor.

`intbv` works transparently with other integer-like types. Like class `int`, it provides access to the underlying two's complement representation for bitwise operations. However, unlike `int`, it is a mutable type. This means that its value can be changed after object creation, through methods and operators such as slice assignment.

`intbv` supports the same operators as `int` for arithmetic. In addition, it provides a number of features to make it suitable for hardware design. First, the range of allowed values can be constrained. This makes it possible to check the value at run time during simulation. Moreover, back end tools can determine the smallest possible bit width for representing the object. Secondly, it supports bit level operations by providing an indexing and slicing interface.

`intbv` objects are constructed in general as follows:

```
intbv([val=None] [, min=None]  [, max=None])
```

val is the initial value. *min* and *max* can be used to constrain the value. Following the Python conventions, *min* is inclusive, and *max* is exclusive. Therefore, the allowed value range is *min* .. *max*-1.

Let's look at some examples. An unconstrained *`intbv`* object is created as follows:

```
>>> a = intbv(24)
```

After object creation, *min* and *max* are available as attributes for inspection. Also, the standard Python function `len` can be used to determine the bit width. If we inspect the previously created object, we get:

```
>>> a
intbv(24)
>>> print(a.min)
None
>>> print(a.max)
None
>>> len(a)
0
```

As the instantiation was unconstrained, the *min* and *max* attributes are undefined. Likewise, the bit width is undefined, which is indicated by a return value `0`.

A constrained *intbv* object is created as follows:

```
>>> a = intbv(24, min=0, max=25)
```

Inspecting the object now gives:

```
>>> a
intbv(24)
>>> a.min
0
>>> a.max
25
>>> len(a)
5
```

We see that the allowed value range is 0 .. 24, and that 5 bits are required to represent the object.

The *min* and *max* bound attributes enable fine-grained control and error checking of the value range. In particular, the bound values do not have to be symmetric or powers of 2. In all cases, the bit width is set appropriately to represent the values in the range. For example:

```
>>> a = intbv(6, min=0, max=7)
>>> len(a)
3
>>> a = intbv(6, min=-3, max=7)
>>> len(a)
4
>>> a = intbv(6, min=-13, max=7)
>>> len(a)
5
```

Bit indexing

A common requirement in hardware design is access to the individual bits. The *intbv* class implements an indexing interface that provides access to the bits of the underlying two's complement representation. The following illustrates bit index read access:

```
>>> from myhdl import bin
>>> a = intbv(24)
>>> bin(a)
'11000'
```

```
>>> int(a[0])
0
>>> int(a[3])
1
>>> b = intbv(-23)
>>> bin(b)
'101001'
>>> int(b[0])
1
>>> int(b[3])
1
>>> int(b[4])
0
```

We use the *bin* function provide by MyHDL because it shows the two's complement representation for negative values, unlike Python's builtin with the same name. Note that lower indices correspond to less significant bits. The following code illustrates bit index assignment:

```
>>> bin(a)
'11000'
>>> a[3] = 0
>>> bin(a)
'10000'
>>> a
intbv(16)
>>> b
intbv(-23)
>>> bin(b)
'101001'
>>> b[3] = 0
>>> bin(b)
'100001'
>>> b
intbv(-31)
```

Bit slicing

The *intbv* type also supports bit slicing, for both read access assignment. For example:

```
>>> a = intbv(24)
>>> bin(a)
'11000'
>>> a[4:1]
intbv(4)
>>> bin(a[4:1])
'100'
>>> a[4:1] = 0b001
>>> bin(a)
'10010'
>>> a
intbv(18)
```

In accordance with the most common hardware convention, and unlike standard Python, slicing ranges are downward. As in standard Python, the slicing range is half-open: the highest

index bit is not included. Unlike standard Python however, this index corresponds to the *left-most* item.

Both indices can be omitted from the slice. If the rightmost index is omitted, it is `0` by default. If the leftmost index is omitted, the meaning is to access "all" higher order bits. For example:

```
>>> bin(a)
'11000'
>>> bin(a[4:])
'1000'
>>> a[4:] = '0001'
>>> bin(a)
'10001'
>>> a[:] = 0b10101
>>> bin(a)
'10101'
```

The half-openness of a slice may seem awkward at first, but it helps to avoid one-off count issues in practice. For example, the slice `a[8:]` has exactly `8` bits. Likewise, the slice `a[7:2]` has `7-2=5` bits. You can think about it as follows: for a slice `[i:j]`, only bits below index `i` are included, and the bit with index `j` is the last bit included.

When an *intbv* object is sliced, a new *intbv* object is returned. This new *intbv* object is always positive, and the value bounds are set up in accordance with the bit width specified by the slice. For example:

```
>>> a = intbv(6, min=-3, max=7)
>>> len(a)
4
>>> b = a[4:]
>>> b
intbv(6L)
>>> len(b)
4
>>> b.min
0
>>> b.max
16
```

In the example, the original object is sliced with a slice equal to its bit width. The returned object has the same value and bit width, but its value range consists of all positive values that can be represented by the bit width.

The object returned by a slice is positive, even when the original object is negative:

```
>>> a = intbv(-3)
>>> bin(a, width=5)
'11101'
>>> b = a[5:]
>>> b
intbv(29L)
>>> bin(b)
'11101'
```

In this example, the bit pattern of the two objects is identical within the bit width, but their values have opposite sign.

Sometimes hardware engineers prefer to constrain an object by defining its bit width directly,

instead of the range of allowed values. Using the slicing properties of the *intbv* class one can do that as follows:

```
>>> a = intbv(24)[5:]
```

What actually happens here is that first an unconstrained *intbv* is created, which is then sliced. Slicing an *intbv* returns a new *intbv* with the constraints set up appropriately. Inspecting the object now shows:

```
>>> a.min
0
>>> a.max
32
>>> len(a)
5
```

Note that the *max* attribute is 32, as with 5 bits it is possible to represent the range 0 .. 31. Creating an *intbv* in this way is convenient but has the disadvantage that only positive value ranges between 0 and a power of 2 can be specified.

The modbv class

In hardware modeling, there is often a need for the elegant modeling of wrap-around behavior. *intbv* instances do not support this automatically, as they assert that any assigned value is within the bound constraints. However, wrap-around modeling can be straightforward. For example, the wrap-around condition for a counter is often decoded explicitly, as it is needed for other purposes. Also, the modulo operator provides an elegant one-liner in many scenarios:

```
count.next = (count + 1) % 2**8
```

However, some interesting cases are not supported by the *intbv* type. For example, we would like to describe a free running counter using a variable and augmented assignment as follows:

```
count_var += 1
```

This is not possible with the *intbv* type, as we cannot add the modulo behavior to this description. A similar problem exist for an augmented left shift as follows:

```
shifter <<= 4
```

To support these operations directly, MyHDL provides the *modbv* type. *modbv* is implemented as a subclass of *intbv*. The two classes have an identical interface and work together in a straightforward way for arithmetic operations. The only difference is how the bounds are handled: out-of-bound values result in an error with *intbv*, and in wrap-around with *modbv*. For example, the modulo counter above can be modeled as follows:

```
count = Signal(modbv(0, min=0, max=2**8))
...
count.next = count + 1
```

The wrap-around behavior is defined in general as follows:

```
val = (val - min) % (max - min) + min
```

In a typical case when `min==0`, this reduces to:

```
val = val % max
```

Unsigned and signed representation

intbv is designed to be as high level as possible. The underlying value of an *intbv* object is a Python `int`, which is represented as a two's complement number with "indefinite" bit width. The range bounds are only used for error checking, and to calculate the minimum required bit width for representation. As a result, arithmetic can be performed like with normal integers.

In contrast, HDLs such as Verilog and VHDL typically require designers to deal with representational issues, especially for synthesizable code. They provide low-level types like `signed` and `unsigned` for arithmetic. The rules for arithmetic with such types are much more complicated than with plain integers.

In some cases it can be useful to interpret *intbv* objects in terms of "signed" and "unsigned". Basically, it depends on attribute *min*. if $min < 0$, then the object is "signed", otherwise it is "unsigned". In particular, the bit width of a "signed" object will account for a sign bit, but that of an "unsigned" will not, because that would be redundant. From earlier sections, we have learned that the return value from a slicing operation is always "unsigned".

In some applications, it is desirable to convert an "unsigned" *intbv* to a "signed", in other words, to interpret the msb bit as a sign bit. The msb bit is the highest order bit within the object's bit width. For this purpose, *intbv* provides the *intbv.signed* method. For example:

```
>>> a = intbv(12, min=0, max=16)
>>> bin(a)
'1100'
>>> b = a.signed()
>>> b
-4
>>> bin(b, width=4)
'1100'
```

intbv.signed extends the msb bit into the higher-order bits of the underlying object value, and returns the result as an integer. Naturally, for a "signed" the return value will always be identical to the original value, as it has the sign bit already.

As an example let's take a 8 bit wide data bus that would be modeled as follows:

```
data_bus = intbv(0)[8:]
```

Now consider that a complex number is transferred over this data bus. The upper 4 bits of the data bus are used for the real value and the lower 4 bits for the imaginary value. As real and imaginary values have a positive and negative value range, we can slice them off from the data bus and convert them as follows:

```
real.next = data_bus[8:4].signed()
imag.next = data_bus[4:].signed()
```

CHAPTER 5

Structural modeling

Introduction

Hardware descriptions need to support the concepts of module instantiation and hierarchy. In MyHDL, an instance is recursively defined as being either a sequence of instances, or a generator. Hierarchy is modeled by defining instances in a higher-level function, and returning them. The following is a schematic example of the basic case.

```
from myhdl import block

@block
def top(...):
    ...
    instance_1 = module_1(...)
    instance_2 = module_2(...)
    ...
    instance_n = module_n(...)
    ...
    return instance_1, instance_2, ... , instance_n
```

Note that MyHDL uses conventional procedural techniques for modeling structure. This makes it straightforward to model more complex cases.

Conditional instantiation

To model conditional instantiation, we can select the returned instance under parameter control. For example:

```
from myhdl import block

SLOW, MEDIUM, FAST = range(3)

@block
```

```
def top(..., speed=SLOW):
    ...
    def slowAndSmall():
        ...
    ...
    def fastAndLarge():
        ...
    if speed == SLOW:
        return slowAndSmall()
    elif speed == FAST:
        return fastAndLarge()
    else:
        raise NotImplementedError
```

Lists of instances and signals

Python lists are easy to create. We can use them to model lists of instances.

Suppose we have a top module that instantiates a single `channel` submodule, as follows:

```
from myhdl import block, Signal

@block
def top(...):

    din = Signal(0)
    dout = Signal(0)
    clk = Signal(bool(0))
    reset = Signal(bool(0))

    channel_inst = channel(dout, din, clk, reset)

    return channel_inst
```

If we wanted to support an arbitrary number of channels, we can use lists of signals and a list of instances, as follows:

```
from myhdl import block, Signal

@block
def top(..., n=8):

    din = [Signal(0) for i in range(n)]
    dout = [Signal(0) for in range(n)]
    clk = Signal(bool(0))
    reset = Signal(bool(0))
    channel_inst = [None for i in range(n)]

    for i in range(n):
        channel_inst[i] = channel(dout[i], din[i], clk, reset)

    return channel_inst
```

Converting between lists of signals and bit vectors

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.

In MyHDL, you can address such cases by a concept called shadow signals. A shadow signal is constructed out of other signals and follows their value changes automatically. For example, a *_SliceSignal* follows the value of an index or a slice from another signal. Likewise, A *ConcatSignal* follows the values of a number of signals as a concatenation.

As an 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*.

Inferring the list of instances

In MyHDL, instances have to be returned explicitly by a top level function. It may be convenient to assemble the list of instances automatically. For this purpose, MyHDL provides the function *instances*. Using the first example in this section, it is used as follows:

```
from myhdl import block, instances

@block
def top(...):
    ...
    instance_1 = module_1(...)
    instance_2 = module_2(...)
    ...
    instance_n = module_n(...)
    ...
    return instances()
```

Function *instances* uses introspection to inspect the type of the local variables defined by the calling function. All variables that comply with the definition of an instance are assembled in a list, and that list is returned.

CHAPTER 6

RTL modeling

Introduction

RTL (Register Transfer Level) is a modeling abstraction level that is typically used to write synthesizable models. *Synthesis* refers to the process by which an HDL description is automatically compiled into an implementation for an ASIC or FPGA. This chapter describes how MyHDL supports it.

Combinatorial logic

Template

Combinatorial logic is described with a code pattern as follows:

```
from myhdl import block, always_comb

@block
def top(<parameters>):
    ...
    @always_comb
    def comb_logic():
        <functional code>
    ...
    return comb_logic, ...
```

The *always_comb* decorator describes combinatorial logic. The name refers to a similar construct in SystemVerilog. The decorated function is a local function that specifies what happens when one of the input signals of the logic changes. The *always_comb* decorator infers the input signals automatically. It returns a generator that is sensitive to all inputs, and that executes the function whenever an input changes.

Example

The following is an example of a combinatorial multiplexer

```
from myhdl import block, always_comb, Signal

@block
def mux(z, a, b, sel):

    """ Multiplexer.

    z -- mux output
    a, b -- data inputs
    sel -- control input: select a if asserted, otherwise b

    """

    @always_comb
    def comb():
        if sel == 1:
            z.next = a
        else:
            z.next = b

    return comb
```

To verify it, we will simulate the logic with some random patterns. The `random` module in Python's standard library comes in handy for such purposes. The function `randrange(n)` returns a random natural integer smaller than n. It is used in the test bench code to produce random input values.

```
import random
from myhdl import block, instance, Signal, intbv, delay
from mux import mux

random.seed(5)
randrange = random.randrange

@block
def test_mux():

    z, a, b, sel = [Signal(intbv(0)) for i in range(4)]

    mux_1 = mux(z, a, b, sel)

    @instance
    def stimulus():
        print("z a b sel")
        for i in range(12):
            a.next, b.next, sel.next = randrange(8), randrange(8),␣
↪randrange(2)
            yield delay(10)
            print("%s %s %s %s" % (z, a, b, sel))

    return mux_1, stimulus

tb = test_mux()
```

```
tb.run_sim()
```

It is often useful to keep the random values reproducible. This can be accomplished by providing a seed value as in the code. The run produces the following output:

```
$ python test_mux.py
z a b sel
5 4 5 0
3 7 3 0
2 2 1 1
7 7 3 1
3 1 3 0
3 3 6 1
6 2 6 0
1 1 2 1
2 2 2 0
3 0 3 0
2 2 2 1
3 5 3 0
<class 'myhdl.StopSimulation'>: No more events
```

Sequential logic

Template

Sequential RTL models are sensitive to a clock edge. In addition, they may be sensitive to a reset signal. The *always_seq* decorator supports this model directly:

```
from myhdl import block, always_seq

@instance
def top(<parameters>, clock, ..., reset, ...):
    ...
    @always_seq(clock.posedge, reset=reset)
    def seq_logic():
        <functional code>
    ...
    return seq_logic, ...
```

The *always_seq* decorator automatically infers the reset functionality. It detects which signals need to be reset, and uses their initial values as the reset values. The reset signal itself needs to be specified as a *ResetSignal* object. For example:

```
reset = ResetSignal(0, active=0, async=True)
```

The first parameter specifies the initial value. The *active* parameter specifies the value on which the reset is active, and the *async* parameter specifies whether it is an asychronous (`True`) or a synchronous (`False`) reset. If no reset is needed, you can assign `None` to the *reset* parameter in the *always_seq* parameter.

Example

The following code is a description of an incrementer with enable, and an asynchronous reset.

```
from myhdl import block, always_seq

@block
def inc(count, enable, clock, reset):
    """ Incrementer with enable.

    count -- output
    enable -- control input, increment when 1
    clock -- clock input
    reset -- asynchronous reset input
    """

    @always_seq(clock.posedge, reset=reset)
    def seq():
        if enable:
            count.next = count + 1

    return seq
```

For the test bench, we will use an independent clock generator, stimulus generator, and monitor. After applying enough stimulus patterns, we can raise the *StopSimulation* exception to stop the simulation run. The test bench for a small incrementer and a small number of patterns is a follows

```
import random
from myhdl import block, always, instance, Signal, \
    ResetSignal, modbv, delay, StopSimulation
from inc import inc

random.seed(1)
randrange = random.randrange

ACTIVE_LOW, INACTIVE_HIGH = 0, 1

@block
def testbench():
    m = 3
    count = Signal(modbv(0)[m:])
    enable = Signal(bool(0))
    clock  = Signal(bool(0))
    reset = ResetSignal(0, active=0, async=True)

    inc_1 = inc(count, enable, clock, reset)

    HALF_PERIOD = delay(10)

    @always(HALF_PERIOD)
    def clockGen():
        clock.next = not clock

    @instance
    def stimulus():
        reset.next = ACTIVE_LOW
```

```
        yield clock.negedge
        reset.next = INACTIVE_HIGH
        for i in range(16):
            enable.next = min(1, randrange(3))
            yield clock.negedge
        raise StopSimulation()

    @instance
    def monitor():
        print("enable  count")
        yield reset.posedge
        while 1:
            yield clock.posedge
            yield delay(1)
            print("   %s      %s" % (int(enable), count))

    return clockGen, stimulus, inc_1, monitor

tb = testbench()
tb.run_sim()
```

The simulation produces the following output

```
$ python test_inc.py
enable  count
   0      0
   1      1
   0      1
   1      2
   0      2
   1      3
   1      4
   1      5
   1      6
   1      7
   0      7
   0      7
   1      0
   0      0
   1      1
   1      2
```

Alternative template

The template with the *always_seq* decorator is convenient as it infers the reset functionality automatically. Alternatively, you can use a more explicit template as follows:

```
from myhdl import block, always

@block
def top(<parameters>, clock, ..., reset, ...):
    ...
    @always(clock.posedge, reset.negedge)
    def seq_logic():
       if not reset:
```

```
            <reset code>
        else:
            <functional code>

    return seq_logic,...
```

With this template, the reset values have to be specified explicitly.

Finite State Machine modeling

Finite State Machine (FSM) modeling is very common in RTL design and therefore deserves special attention.

For code clarity, the state values are typically represented by a set of identifiers. A standard Python idiom for this purpose is to assign a range of integers to a tuple of identifiers, like so

```
>>> SEARCH, CONFIRM, SYNC = range(3)
>>> CONFIRM
1
```

However, this technique has some drawbacks. Though it is clearly the intention that the identifiers belong together, this information is lost as soon as they are defined. Also, the identifiers evaluate to integers, whereas a string representation of the identifiers would be preferable. To solve these issues, we need an *enumeration type*.

MyHDL supports enumeration types by providing a function *enum*. The arguments to *enum* are the string representations of the identifiers, and its return value is an enumeration type. The identifiers are available as attributes of the type. For example

```
>>> from myhdl import enum
>>> t_State = enum('SEARCH', 'CONFIRM', 'SYNC')
>>> t_State
<Enum: SEARCH, CONFIRM, SYNC>
>>> t_State.CONFIRM
CONFIRM
```

We can use this type to construct a state signal as follows:

```
state = Signal(t_State.SEARCH)
```

As an example, we will use a framing controller FSM. It is an imaginary example, but similar control structures are often found in telecommunication applications. Suppose that we need to find the Start Of Frame (SOF) position of an incoming frame of bytes. A sync pattern detector continuously looks for a framing pattern and indicates it to the FSM with a `syncFlag` signal. When found, the FSM moves from the initial `SEARCH` state to the `CONFIRM` state. When the `syncFlag` is confirmed on the expected position, the FSM declares `SYNC`, otherwise it falls back to the `SEARCH` state. This FSM can be coded as follows

```
from myhdl import block, always_seq, Signal, intbv, enum

ACTIVE_LOW = 0
FRAME_SIZE = 8
t_state = enum('SEARCH', 'CONFIRM', 'SYNC')
```

```
@block
def framer_ctrl(sof, state, sync_flag, clk, reset_n):

    """ Framing control FSM.

    sof -- start-of-frame output bit
    state -- FramerState output
    sync_flag -- sync pattern found indication input
    clk -- clock input
    reset_n -- active low reset

    """

    index = Signal(intbv(0, min=0, max=FRAME_SIZE)) # position in frame

    @always_seq(clk.posedge, reset=reset_n)
    def FSM():
        if reset_n == ACTIVE_LOW:
            sof.next = 0
            index.next = 0
            state.next = t_state.SEARCH

        else:
            index.next = (index + 1) % FRAME_SIZE
            sof.next = 0

            if state == t_state.SEARCH:
                index.next = 1
                if sync_flag:
                    state.next = t_state.CONFIRM

            elif state == t_state.CONFIRM:
                if index == 0:
                    if sync_flag:
                        state.next = t_state.SYNC
                    else:
                        state.next = t_state.SEARCH

            elif state == t_state.SYNC:
                if index == 0:
                    if not sync_flag:
                        state.next = t_state.SEARCH
                sof.next = (index == FRAME_SIZE-1)

            else:
                raise ValueError("Undefined state")

    return FSM
```

At this point, we will use the example to demonstrate the MyHDL support for waveform viewing. During simulation, signal changes can be written to a VCD output file. The VCD file can then be loaded and viewed in a waveform viewer tool such as **gtkwave**.

The user interface of this feature consists of a single function, *traceSignals*. To explain how it works, recall that in MyHDL, an instance is created by assigning the result of a function call to an instance name. For example:

```
tb_fsm = testbench()
```

To enable VCD tracing, the instance should be created as follows instead:

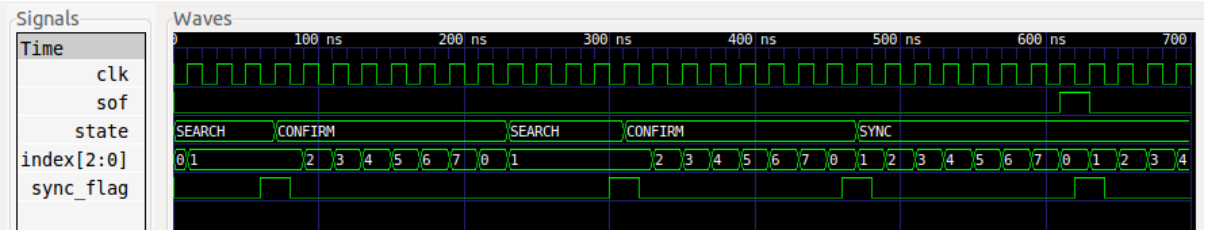
```
tb_fsm = traceSignals(testbench)
```

Note that the first argument of *traceSignals* consists of the uncalled function. By calling the function under its control, *traceSignals* gathers information about the hierarchy and the signals to be traced. In addition to a function argument, *traceSignals* accepts an arbitrary number of non-keyword and keyword arguments that will be passed to the function call.

A small test bench for our framing controller example, with signal tracing enabled, is shown below:

```
import myhdl
from myhdl import block, always, instance, Signal, ResetSignal, delay,␣
↪StopSimulation
from fsm import framer_ctrl, t_state

ACTIVE_LOW = 0

@block
def testbench():

    sof = Signal(bool(0))
    sync_flag = Signal(bool(0))
    clk = Signal(bool(0))
    reset_n = ResetSignal(1, active=ACTIVE_LOW, async=True)
    state = Signal(t_state.SEARCH)

    frame_ctrl_0 = framer_ctrl(sof, state, sync_flag, clk, reset_n)

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

    @instance
    def stimulus():
        for i in range(3):
            yield clk.negedge
        for n in (12, 8, 8, 4):
            sync_flag.next = 1
            yield clk.negedge
            sync_flag.next = 0
            for i in range(n-1):
                yield clk.negedge
        raise StopSimulation()

    return frame_ctrl_0, clkgen, stimulus

tb = testbench()
tb.config_sim(trace=True)
tb.run_sim()
```

When we run the test bench, it generates a VCD file called `testbench.vcd`. When we load this file into **gtkwave**, we can view the waveforms:



Signals are dumped in a suitable format. This format is inferred at the *Signal* construction time, from the type of the initial value. In particular, `bool` signals are dumped as single bits. (This only works starting with Python 2.3, when `bool` has become a separate type). Likewise, *intbv* signals with a defined bit width are dumped as bit vectors. To support the general case, other types of signals are dumped as a string representation, as returned by the standard `str` function.

Warning: Support for literal string representations is not part of the VCD standard. It is specific to **gtkwave**. To generate a standard VCD file, you need to use signals with a defined bit width only.

CHAPTER 7

High level modeling

Introduction

To write synthesizable models in MyHDL, you should stick to the RTL templates shown in *RTL modeling*. However, modeling in MyHDL is much more powerful than that. Conceptually, MyHDL is a library for general event-driven modeling and simulation of hardware systems.

There are many reasons why it can be useful to model at a higher abstraction level than RTL. For example, you can use MyHDL to verify architectural features, such as system throughput, latency and buffer sizes. You can also write high level models for specialized technology-dependent cores that are not going through synthesis. Last but not least, you can use MyHDL to write test benches that verify a system model or a synthesizable description.

This chapter explores some of the options for high level modeling with MyHDL.

Modeling with bus-functional procedures

A *bus-functional procedure* is a reusable encapsulation of the low-level operations needed to implement some abstract transaction on a physical interface. Bus-functional procedures are typically used in flexible verification environments.

Once again, MyHDL uses generator functions to support bus-functional procedures. In MyHDL, the difference between instances and bus-functional procedure calls comes from the way in which a generator function is used.

As an example, we will design a bus-functional procedure of a simplified UART transmitter. We assume 8 data bits, no parity bit, and a single stop bit, and we add print statements to follow the simulation behavior:

```
T_9600 = int(1e9 / 9600)

def rs232_tx(tx, data, duration=T_9600):

```

```
    """ Simple rs232 transmitter procedure.

    tx -- serial output data
    data -- input data byte to be transmitted
    duration -- transmit bit duration

    """

    print "-- Transmitting %s --" % hex(data)
    print "TX: start bit"
    tx.next = 0
    yield delay(duration)

    for i in range(8):
        print "TX: %s" % data[i]
        tx.next = data[i]
        yield delay(duration)

    print "TX: stop bit"
    tx.next = 1
    yield delay(duration)
```

This looks exactly like the generator functions in previous sections. It becomes a bus-functional procedure when we use it differently. Suppose that in a test bench, we want to generate a number of data bytes to be transmitted. This can be modeled as follows:

```
testvals = (0xc5, 0x3a, 0x4b)

def stimulus():
    tx = Signal(1)
    for val in testvals:
        txData = intbv(val)
        yield rs232_tx(tx, txData)
```

We use the bus-functional procedure call as a clause in a `yield` statement. This introduces a fourth form of the `yield` statement: using a generator as a clause. Although this is a more dynamic usage than in the previous cases, the meaning is actually very similar: at that point, the original generator should wait for the completion of a generator. In this case, the original generator resumes when the `rs232_tx(tx, txData)` generator returns.

When simulating this, we get:

```
-- Transmitting 0xc5 --
TX: start bit
TX: 1
TX: 0
TX: 1
TX: 0
TX: 0
TX: 0
TX: 1
TX: 1
TX: stop bit
-- Transmitting 0x3a --
TX: start bit
TX: 0
TX: 1
```

```
TX: 0
TX: 1
...
```

We will continue with this example by designing the corresponding UART receiver bus-functional procedure. This will allow us to introduce further capabilities of MyHDL and its use of the `yield` statement.

Until now, the `yield` statements had a single clause. However, they can have multiple clauses as well. In that case, the generator resumes as soon as the wait condition specified by one of the clauses is satisfied. This corresponds to the functionality of sensitivity lists in Verilog and VHDL.

For example, suppose we want to design an UART receive procedure with a timeout. We can specify the timeout condition while waiting for the start bit, as in the following generator function:

```
def rs232_rx(rx, data, duration=T_9600, timeout=MAX_TIMEOUT):

    """ Simple rs232 receiver procedure.

    rx -- serial input data
    data -- data received
    duration -- receive bit duration

    """

    # wait on start bit until timeout
    yield rx.negedge, delay(timeout)
    if rx == 1:
        raise StopSimulation, "RX time out error"

    # sample in the middle of the bit duration
    yield delay(duration // 2)
    print "RX: start bit"

    for i in range(8):
        yield delay(duration)
        print "RX: %s" % rx
        data[i] = rx

    yield delay(duration)
    print "RX: stop bit"
    print "-- Received %s --" % hex(data)
```

If the timeout condition is triggered, the receive bit `rx` will still be `1`. In that case, we raise an exception to stop the simulation. The `StopSimulation` exception is predefined in MyHDL for such purposes. In the other case, we proceed by positioning the sample point in the middle of the bit duration, and sampling the received data bits.

When a `yield` statement has multiple clauses, they can be of any type that is supported as a single clause, including generators. For example, we can verify the transmitter and receiver generator against each other by yielding them together, as follows:

```
def test():
    tx = Signal(1)
```

```
    rx = tx
    rxData = intbv(0)
    for val in testvals:
        txData = intbv(val)
        yield rs232_rx(rx, rxData), rs232_tx(tx, txData)
```

Both forked generators will run concurrently, and the original generator will resume as soon as one of them finishes (which will be the transmitter in this case). The simulation output shows how the UART procedures run in lockstep:

```
-- Transmitting 0xc5 --
TX: start bit
RX: start bit
TX: 1
RX: 1
TX: 0
RX: 0
TX: 1
RX: 1
TX: 0
RX: 0
TX: 0
RX: 0
TX: 0
RX: 0
TX: 1
RX: 1
TX: 1
RX: 1
TX: stop bit
RX: stop bit
-- Received 0xc5 --
-- Transmitting 0x3a --
TX: start bit
RX: start bit
TX: 0
RX: 0
...
```

For completeness, we will verify the timeout behavior with a test bench that disconnects the `rx` from the `tx` signal, and we specify a small timeout for the receive procedure:

```
def testTimeout():
    tx = Signal(1)
    rx = Signal(1)
    rxData = intbv(0)
    for val in testvals:
        txData = intbv(val)
        yield rs232_rx(rx, rxData, timeout=4*T_9600-1), rs232_tx(tx,␣
↪txData)
```

The simulation now stops with a timeout exception after a few transmit cycles:

```
-- Transmitting 0xc5 --
TX: start bit
TX: 1
TX: 0
```

```
TX: 1
StopSimulation: RX time out error
```

Recall that the original generator resumes as soon as one of the forked generators returns. In the previous cases, this is just fine, as the transmitter and receiver generators run in lockstep. However, it may be desirable to resume the caller only when *all* of the forked generators have finished. For example, suppose that we want to characterize the robustness of the transmitter and receiver design to bit duration differences. We can adapt our test bench as follows, to run the transmitter at a faster rate:

```
T_10200 = int(1e9 / 10200)

def testNoJoin():
    tx = Signal(1)
    rx = tx
    rxData = intbv(0)
    for val in testvals:
        txData = intbv(val)
        yield rs232_rx(rx, rxData), rs232_tx(tx, txData, duration=T_10200)
```

Simulating this shows how the transmission of the new byte starts before the previous one is received, potentially creating additional transmission errors:

```
-- Transmitting 0xc5 --
TX: start bit
RX: start bit
...
TX: 1
RX: 1
TX: 1
TX: stop bit
RX: 1
-- Transmitting 0x3a --
TX: start bit
RX: stop bit
-- Received 0xc5 --
RX: start bit
TX: 0
```

It is more likely that we want to characterize the design on a byte by byte basis, and align the two generators before transmitting each byte. In MyHDL, this is done with the *join* function. By joining clauses together in a `yield` statement, we create a new clause that triggers only when all of its clause arguments have triggered. For example, we can adapt the test bench as follows:

```
def testJoin():
    tx = Signal(1)
    rx = tx
    rxData = intbv(0)
    for val in testvals:
        txData = intbv(val)
        yield join(rs232_rx(rx, rxData), rs232_tx(tx, txData, duration=T_
↪10200))
```

Now, transmission of a new byte only starts when the previous one is received:

```
-- Transmitting 0xc5 --
TX: start bit
RX: start bit
...
TX: 1
RX: 1
TX: 1
TX: stop bit
RX: 1
RX: stop bit
-- Received 0xc5 --
-- Transmitting 0x3a --
TX: start bit
RX: start bit
TX: 0
RX: 0
```

Modeling memories with built-in types

Python has powerful built-in data types that can be useful to model hardware memories. This can be merely a matter of putting an interface around some data type operations.

For example, a *dictionary* comes in handy to model sparse memory structures. (In other languages, this data type is called *associative array*, or *hash table*.) A sparse memory is one in which only a small part of the addresses is used in a particular application or simulation. Instead of statically allocating the full address space, which can be large, it is better to dynamically allocate the needed storage space. This is exactly what a dictionary provides. The following is an example of a sparse memory model:

```
def sparseMemory(dout, din, addr, we, en, clk):

    """ Sparse memory model based on a dictionary.

    Ports:
    dout -- data out
    din -- data in
    addr -- address bus
    we -- write enable: write if 1, read otherwise
    en -- interface enable: enabled if 1
    clk -- clock input

    """

    memory = {}

    @always(clk.posedge)
    def access():
        if en:
            if we:
                memory[addr.val] = din.val
            else:
                dout.next = memory[addr.val]

    return access
```

Note how we use the `val` attribute of the `din` signal, as we don't want to store the signal object itself, but its current value. Similarly, we use the `val` attribute of the `addr` signal as the dictionary key.

In many cases, MyHDL code uses a signal's current value automatically when there is no ambiguity: for example, when a signal is used in an expression. However, in other cases, such as in this example, you have to refer to the value explicitly: for example, when the Signal is used as a dictionary key, or when it is not used in an expression. One option is to use the `val` attribute, as in this example. Another possibility is to use the `int()` or `bool()` functions to typecast the Signal to an integer or a boolean value. These functions are also useful with *intbv* objects.

As a second example, we will demonstrate how to use a list to model a synchronous fifo:

```
def fifo(dout, din, re, we, empty, full, clk, maxFilling=sys.maxint):

    """ Synchronous fifo model based on a list.

    Ports:
    dout -- data out
    din -- data in
    re -- read enable
    we -- write enable
    empty -- empty indication flag
    full -- full indication flag
    clk -- clock input

    Optional parameter:
    maxFilling -- maximum fifo filling, "infinite" by default

    """

    memory = []

    @always(clk.posedge)
    def access():
        if we:
            memory.insert(0, din.val)
        if re:
            dout.next = memory.pop()
        filling = len(memory)
        empty.next = (filling == 0)
        full.next = (filling == maxFilling)

    return access
```

Again, the model is merely a MyHDL interface around some operations on a list: `insert` to insert entries, `pop` to retrieve them, and `len` to get the size of a Python object.

Modeling errors using exceptions

In the previous section, we used Python data types for modeling. If such a type is used inappropriately, Python's run time error system will come into play. For example, if we access an address in the `sparseMemory` model that was not initialized before, we will get a traceback similar to the following (some lines omitted for clarity):

```
Traceback (most recent call last):
...
  File "sparseMemory.py", line 31, in access
    dout.next = memory[addr.val]
KeyError: Signal(51)
```

Similarly, if the `fifo` is empty, and we attempt to read from it, we get:

```
Traceback (most recent call last):
...
  File "fifo.py", line 34, in fifo
    dout.next = memory.pop()
IndexError: pop from empty list
```

Instead of these low level errors, it may be preferable to define errors at the functional level. In Python, this is typically done by defining a custom `Error` exception, by subclassing the standard `Exception` class. This exception is then raised explicitly when an error condition occurs.

For example, we can change the `sparseMemory` function as follows (with the doc string is omitted for brevity):

```
class Error(Exception):
    pass

def sparseMemory2(dout, din, addr, we, en, clk):

    memory = {}

    @always(clk.posedge)
    def access():
        if en:
            if we:
                memory[addr.val] = din.val
            else:
                try:
                    dout.next = memory[addr.val]
                except KeyError:
                    raise Error, "Uninitialized address %s" % hex(addr)

    return access
```

This works by catching the low level data type exception, and raising the custom exception with an appropriate error message instead. If the `sparseMemory` function is defined in a module with the same name, an access error is now reported as follows:

```
Traceback (most recent call last):
...
  File "sparseMemory.py", line 61, in access
    raise Error, "Uninitialized address %s" % hex(addr)
Error: Uninitialized address 0x33
```

Likewise, the `fifo` function can be adapted as follows, to report underflow and overflow errors:

```
class Error(Exception):
    pass


def fifo2(dout, din, re, we, empty, full, clk, maxFilling=sys.maxint):

    memory = []

    @always(clk.posedge)
    def access():
        if we:
            memory.insert(0, din.val)
        if re:
            try:
                dout.next = memory.pop()
            except IndexError:
                raise Error, "Underflow -- Read from empty fifo"
        filling = len(memory)
        empty.next = (filling == 0)
        full.next = (filling == maxFilling)
        if filling > maxFilling:
            raise Error, "Overflow -- Max filling %s exceeded" % maxFilling

    return access
```

In this case, the underflow error is detected as before, by catching a low level exception on the list data type. On the other hand, the overflow error is detected by a regular check on the length of the list.

Object oriented modeling

The models in the previous sections used high-level built-in data types internally. However, they had a conventional RTL-style interface. Communication with such a module is done through signals that are attached to it during instantiation.

A more advanced approach is to model hardware blocks as objects. Communication with objects is done through method calls. A method encapsulates all details of a certain task performed by the object. As an object has a method interface instead of an RTL-style hardware interface, this is a much higher level approach.

As an example, we will design a synchronized queue object. Such an object can be filled by producer, and independently read by a consumer. When the queue is empty, the consumer should wait until an item is available. The queue can be modeled as an object with a `put(item)` and a `get` method, as follows:

```
from myhdl import *

def trigger(event):
    event.next = not event

class queue:
    def __init__(self):
       self.l = []
       self.sync = Signal(0)
```

```
        self.item = None
    def put(self,item):
        # non time-consuming method
        self.l.append(item)
        trigger(self.sync)
    def get(self):
        # time-consuming method
        if not self.l:
            yield self.sync
        self.item = self.l.pop(0)
```

The `queue` object constructor initializes an internal list to hold items, and a *sync* signal to synchronize the operation between the methods. Whenever `put` puts an item in the queue, the signal is triggered. When the `get` method sees that the list is empty, it waits on the trigger first. `get` is a generator method because it may consume time. As the `yield` statement is used in MyHDL for timing control, the method cannot "yield" the item. Instead, it makes it available in the *item* instance variable.

To test the queue operation, we will model a producer and a consumer in the test bench. As a waiting consumer should not block a whole system, it should run in a concurrent "thread". As always in MyHDL, concurrency is modeled by Python generators. Producer and consumer will thus run independently, and we will monitor their operation through some print statements:

```
q = queue()

def Producer(q):
    yield delay(120)
    for i in range(5):
        print "%s: PUT item %s" % (now(), i)
        q.put(i)
        yield delay(max(5, 45 - 10*i))

def Consumer(q):
    yield delay(100)
    while 1:
        print "%s: TRY to get item" % now()
        yield q.get()
        print "%s: GOT item %s" % (now(), q.item)
        yield delay(30)

def main():
    P = Producer(q)
    C = Consumer(q)
    return P, C

sim = Simulation(main())
sim.run()
```

Note that the generator method `get` is called in a `yield` statement in the `Consumer` function. The new generator will take over from `Consumer`, until it is done. Running this test bench produces the following output:

```
% python queue.py
100: TRY to get item
120: PUT item 0
```

```
120: GOT item 0
150: TRY to get item
165: PUT item 1
165: GOT item 1
195: TRY to get item
200: PUT item 2
200: GOT item 2
225: PUT item 3
230: TRY to get item
230: GOT item 3
240: PUT item 4
260: TRY to get item
260: GOT item 4
290: TRY to get item
StopSimulation: No more events
```

CHAPTER 8

Unit testing

Introduction

Many aspects in the design flow of modern digital hardware design can be viewed as a special kind of software development. From that viewpoint, it is a natural question whether advances in software design techniques can not also be applied to hardware design.

One software design approach that deserves attention is *Extreme Programming* (XP). It is a fascinating set of techniques and guidelines that often seems to go against the conventional wisdom. On other occasions, XP just seems to emphasize the common sense, which doesn't always coincide with common practice. For example, XP stresses the importance of normal workweeks, if we are to have the fresh mind needed for good software development.

It is not my intention nor qualification to present a tutorial on Extreme Programming. Instead, in this section I will highlight one XP concept which I think is very relevant to hardware design: the importance and methodology of unit testing.

The importance of unit tests

Unit testing is one of the corner stones of Extreme Programming. Other XP concepts, such as collective ownership of code and continuous refinement, are only possible by having unit tests. Moreover, XP emphasizes that writing unit tests should be automated, that they should test everything in every class, and that they should run perfectly all the time.

I believe that these concepts apply directly to hardware design. In addition, unit tests are a way to manage simulation time. For example, a state machine that runs very slowly on infrequent events may be impossible to verify at the system level, even on the fastest simulator. On the other hand, it may be easy to verify it exhaustively in a unit test, even on the slowest simulator.

It is clear that unit tests have compelling advantages. On the other hand, if we need to test everything, we have to write lots of unit tests. So it should be easy and pleasant to create, manage and run them. Therefore, XP emphasizes the need for a unit test framework that

supports these tasks. In this chapter, we will explore the use of the `unittest` module from the standard Python library for creating unit tests for hardware designs.

Unit test development

In this section, we will informally explore the application of unit test techniques to hardware design. We will do so by a (small) example: testing a binary to Gray encoder as introduced in section *Bit indexing*.

Defining the requirements

We start by defining the requirements. For a Gray encoder, we want to the output to comply with Gray code characteristics. Let's define a *code* as a list of *codewords*, where a codeword is a bit string. A code of order `n` has `2**n` codewords.

A well-known characteristic is the one that Gray codes are all about:

Consecutive codewords in a Gray code should differ in a single bit.

Is this sufficient? Not quite: suppose for example that an implementation returns the lsb of each binary input. This would comply with the requirement, but is obviously not what we want. Also, we don't want the bit width of Gray codewords to exceed the bit width of the binary codewords.

Each codeword in a Gray code of order n must occur exactly once in the binary code of the same order.

With the requirements written down we can proceed.

Writing the test first

A fascinating guideline in the XP world is to write the unit test first. That is, before implementing something, first write the test that will verify it. This seems to go against our natural inclination, and certainly against common practices. Many engineers like to implement first and think about verification afterwards.

But if you think about it, it makes a lot of sense to deal with verification first. Verification is about the requirements only — so your thoughts are not yet cluttered with implementation details. The unit tests are an executable description of the requirements, so they will be better understood and it will be very clear what needs to be done. Consequently, the implementation should go smoother. Perhaps most importantly, the test is available when you are done implementing, and can be run anytime by anybody to verify changes.

Python has a standard `unittest` module that facilitates writing, managing and running unit tests. With `unittest`, a test case is written by creating a class that inherits from `unittest.TestCase`. Individual tests are created by methods of that class: all method names that start with `test` are considered to be tests of the test case.

We will define a test case for the Gray code properties, and then write a test for each of the requirements. The outline of the test case class is as follows:

```
import unittest

class TestGrayCodeProperties(unittest.TestCase):

    def testSingleBitChange(self):
     """Check that only one bit changes in successive codewords."""
     ....


    def testUniqueCodeWords(self):
    """Check that all codewords occur exactly once."""
    ....
```

Each method will be a small test bench that tests a single requirement. To write the tests, we don't need an implementation of the Gray encoder, but we do need the interface of the design. We can specify this by a dummy implementation, as follows:

```
from myhdl import block

@block
def bin2gray(B, G):
    # DUMMY PLACEHOLDER
    """ Gray encoder.

    B -- binary input
    G -- Gray encoded output
    """
    pass
```

For the first requirement, we will test all consecutive input numbers, and compare the current output with the previous one For each input, we check that the difference is exactly a single bit. For the second requirement, we will test all input numbers and put the result in a list. The requirement implies that if we sort the result list, we should get a range of numbers. For both requirements, we will test all Gray codes up to a certain order `MAX_WIDTH`. The test code looks as follows:

```
import unittest

from myhdl import Simulation, Signal, delay, intbv, bin

from bin2gray import bin2gray

MAX_WIDTH = 11

class TestGrayCodeProperties(unittest.TestCase):

    def testSingleBitChange(self):
        """Check that only one bit changes in successive codewords."""

        def test(B, G):
            w = len(B)
            G_Z = Signal(intbv(0)[w:])
            B.next = intbv(0)
            yield delay(10)
            for i in range(1, 2**w):
                G_Z.next = G
```

```
                B.next = intbv(i)
                yield delay(10)
                diffcode = bin(G ^ G_Z)
                self.assertEqual(diffcode.count('1'), 1)

        self.runTests(test)

    def testUniqueCodeWords(self):
        """Check that all codewords occur exactly once."""

        def test(B, G):
            w = len(B)
            actual = []
            for i in range(2**w):
                B.next = intbv(i)
                yield delay(10)
                actual.append(int(G))
            actual.sort()
            expected = list(range(2**w))
            self.assertEqual(actual, expected)

        self.runTests(test)


    def runTests(self, test):
        """Helper method to run the actual tests."""
        for w in range(1, MAX_WIDTH):
            B = Signal(intbv(0)[w:])
            G = Signal(intbv(0)[w:])
            dut = bin2gray(B, G)
            check = test(B, G)
            sim = Simulation(dut, check)
            sim.run(quiet=1)


if __name__ == '__main__':
    unittest.main(verbosity=2)
```

Note how the actual check is performed by a `self.assertEqual` method, defined by the `unittest.TestCase` class. Also, we have factored out running the tests for all Gray codes in a separate method `runTests`.

Test-driven implementation

With the test written, we begin with the implementation. For illustration purposes, we will intentionally write some incorrect implementations to see how the test behaves.

The easiest way to run tests defined with the `unittest` framework, is to put a call to its `main` method at the end of the test module:

```
unittest.main()
```

Let's run the test using the dummy Gray encoder shown earlier:

```
% python test_gray_properties.py
testSingleBitChange (__main__.TestGrayCodeProperties)
Check that only one bit changes in successive codewords. ... ERROR
testUniqueCodeWords (__main__.TestGrayCodeProperties)
Check that all codewords occur exactly once. ... ERROR
```

As expected, this fails completely. Let us try an incorrect implementation, that puts the lsb of in the input on the output:

```
from myhdl import block, always_comb

@block
def bin2gray(B, G):
    # INCORRECT IMPLEMENTATION
    """ Gray encoder.

    B -- binary input
    G -- Gray encoded output
    """

    @always_comb
    def logic():
        G.next = B[0]

    return logic
```

Running the test produces:

```
python test_gray_properties.py
testSingleBitChange (__main__.TestGrayCodeProperties)
Check that only one bit changes in successive codewords. ... ok
testUniqueCodeWords (__main__.TestGrayCodeProperties)
Check that all codewords occur exactly once. ... FAIL

======================================================================
FAIL: testUniqueCodeWords (__main__.TestGrayCodeProperties)
Check that all codewords occur exactly once.
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_gray_properties.py", line 42, in testUniqueCodeWords
    self.runTests(test)
  File "test_gray_properties.py", line 53, in runTests
    sim.run(quiet=1)
  File "/home/jand/dev/myhdl/myhdl/_Simulation.py", line 154, in run
    waiter.next(waiters, actives, exc)
  File "/home/jand/dev/myhdl/myhdl/_Waiter.py", line 127, in next
    clause = next(self.generator)
  File "test_gray_properties.py", line 40, in test
    self.assertEqual(actual, expected)
AssertionError: Lists differ: [0, 0, 1, 1] != [0, 1, 2, 3]

First differing element 1:
0
1

- [0, 0, 1, 1]
+ [0, 1, 2, 3]
```

```
----------------------------------------------------------------------
Ran 2 tests in 0.083s

FAILED (failures=1)
```

Now the test passes the first requirement, as expected, but fails the second one. After the test feedback, a full traceback is shown that can help to debug the test output.

Finally, we use a correct implementation:

```
from myhdl import block, always_comb

@block
def bin2gray(B, G):
    """ Gray encoder.

    B -- binary input
    G -- Gray encoded output
    """

    @always_comb
    def logic():
        G.next = (B>>1) ^ B

    return logic
```

Now the tests pass:

```
$ python test_gray_properties.py
testSingleBitChange (__main__.TestGrayCodeProperties)
Check that only one bit changes in successive codewords. ... ok
testUniqueCodeWords (__main__.TestGrayCodeProperties)
Check that all codewords occur exactly once. ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.369s

OK
```

Additional requirements

In the previous section, we concentrated on the general requirements of a Gray code. It is possible to specify these without specifying the actual code. It is easy to see that there are several codes that satisfy these requirements. In good XP style, we only tested the requirements and nothing more.

It may be that more control is needed. For example, the requirement may be for a particular code, instead of compliance with general properties. As an illustration, we will show how to test for *the* original Gray code, which is one specific instance that satisfies the requirements of the previous section. In this particular case, this test will actually be easier than the previous one.

We denote the original Gray code of order `n` as `Ln`. Some examples:

```
L1 = ['0', '1']
L2 = ['00', '01', '11', '10']
L3 = ['000', '001', '011', '010', '110', '111', '101', 100']
```

It is possible to specify these codes by a recursive algorithm, as follows:

1. L1 = ['0', '1']
2. Ln+1 can be obtained from Ln as follows. Create a new code Ln0 by prefixing all codewords of Ln with '0'. Create another new code Ln1 by prefixing all codewords of Ln with '1', and reversing their order. Ln+1 is the concatenation of Ln0 and Ln1.

Python is well-known for its elegant algorithmic descriptions, and this is a good example. We can write the algorithm in Python as follows:

```
def nextLn(Ln):
    """ Return Gray code Ln+1, given Ln. """
    Ln0 = ['0' + codeword for codeword in Ln]
    Ln1 = ['1' + codeword for codeword in Ln]
    Ln1.reverse()
    return Ln0 + Ln1
```

The code `['0' + codeword for ...]` is called a *list comprehension*. It is a concise way to describe lists built by short computations in a for loop.

The requirement is now that the output code matches the expected code Ln. We use the `nextLn` function to compute the expected result. The new test case code is as follows:

```
import unittest

from myhdl import Simulation, Signal, delay, intbv, bin

from bin2gray import bin2gray
from next_gray_code import nextLn

MAX_WIDTH = 11

class TestOriginalGrayCode(unittest.TestCase):

    def testOriginalGrayCode(self):
        """Check that the code is an original Gray code."""

        Rn = []

        def stimulus(B, G, n):
            for i in range(2**n):
                B.next = intbv(i)
                yield delay(10)
                Rn.append(bin(G, width=n))

        Ln = ['0', '1'] # n == 1
        for w in range(2, MAX_WIDTH):
            Ln = nextLn(Ln)
            del Rn[:]
            B = Signal(intbv(0)[w:])
            G = Signal(intbv(0)[w:])
            dut = bin2gray(B, G)
            stim = stimulus(B, G, w)
```

```
            sim = Simulation(dut, stim)
            sim.run(quiet=1)
            self.assertEqual(Ln, Rn)


if __name__ == '__main__':
    unittest.main(verbosity=2)
```

As it happens, our implementation is apparently an original Gray code:

```
$ python test_gray_original.py
testOriginalGrayCode (__main__.TestOriginalGrayCode)
Check that the code is an original Gray code. ... ok

----------------------------------------------------------------------
Ran 1 test in 0.173s

OK
```

CHAPTER 9

Co-simulation with Verilog

Introduction

One of the most exciting possibilities of MyHDLis to use it as a hardware verification language (HVL). A HVL is a language used to write test benches and verification environments, and to control simulations.

Nowadays, it is generally acknowledged that HVLs should be equipped with modern software techniques, such as object orientation. The reason is that verification it the most complex and time-consuming task of the design process. Consequently, every useful technique is welcome. Moreover, test benches are not required to be implementable. Therefore, unlike with synthesizable code, there are no constraints on creativity.

Technically, verification of a design implemented in another language requires co-simulation. MyHDL is enabled for co-simulation with any HDL simulator that has a procedural language interface (PLI). The MyHDLside is designed to be independent of a particular simulator, On the other hand, for each HDL simulator a specific PLI module will have to be written in C. Currently, the MyHDL release contains a PLI module for two Verilog simulators: Icarus and Cver.

The HDL side

To introduce co-simulation, we will continue to use the Gray encoder example from the previous chapters. Suppose that we want to synthesize it and write it in Verilog for that purpose. Clearly we would like to reuse our unit test verification environment.

To start, let's recall how the Gray encoder in MyHDL looks like:

```
from myhdl import block, always_comb

@block
def bin2gray(B, G):
    """ Gray encoder.
```

```
    B -- binary input
    G -- Gray encoded output
    """

    @always_comb
    def logic():
        G.next = (B>>1) ^ B

    return logic
```

To show the co-simulation flow, we don't need the Verilog implementation yet, but only the interface. Our Gray encoder in Verilog would have the following interface:

```
module bin2gray(B, G);

   parameter width = 8;
   input [width-1:0]  B;
   output [width-1:0] G;
   ....
```

To write a test bench, one creates a new module that instantiates the design under test (DUT). The test bench declares nets and regs (or signals in VHDL) that are attached to the DUT, and to stimulus generators and response checkers. In an all-HDL flow, the generators and checkers are written in the HDL itself, but we will want to write them in MyHDL. To make the connection, we need to declare which regs & nets are driven and read by the MyHDLsimulator. For our example, this is done as follows:

```
module dut_bin2gray;

   reg [`width-1:0] B;
   wire [`width-1:0] G;

   initial begin
      $from_myhdl(B);
      $to_myhdl(G);
   end

   bin2gray dut (.B(B), .G(G));
   defparam dut.width = `width;

endmodule
```

The `$from_myhdl` task call declares which regs are driven by MyHDL, and the `$to_myhdl` task call which regs & nets are read by it. These tasks take an arbitrary number of arguments. They are defined in a PLI module written in C and made available in a simulator-dependent manner. In Icarus Verilog, the tasks are defined in a `myhdl.vpi` module that is compiled from C source code.

The MyHDL side

MyHDL supports co-simulation by a `Cosimulation` object. A `Cosimulation` object must know how to run a HDL simulation. Therefore, the first argument to its constructor is a com-

mand string to execute a simulation.

The way to generate and run an simulation executable is simulator dependent. For example, in Icarus Verilog, a simulation executable for our example can be obtained obtained by running the `iverilog` compiler as follows:

```
% iverilog -o bin2gray -Dwidth=4 bin2gray.v dut_bin2gray.v
```

This generates a `bin2gray` executable for a parameter `width` of 4, by compiling the contributing Verilog files.

The simulation itself is run by the `vvp` command:

```
% vvp -m ./myhdl.vpi bin2gray
```

This runs the `bin2gray` simulation, and specifies to use the `myhdl.vpi` PLI module present in the current directory. (This is just a command line usage example; actually simulating with the `myhdl.vpi` module is only meaningful from a `Cosimulation` object.)

We can use a `Cosimulation` object to provide a HDL version of a design to the MyHDL simulator. Instead of a generator function, we write a function that returns a `Cosimulation` object. For our example and the Icarus Verilog simulator, this is done as follows:

```
import os

from myhdl import Cosimulation

cmd = "iverilog -o bin2gray.o -Dwidth=%s " + \
      "../../test/verilog/bin2gray.v " + \
      "../../test/verilog/dut_bin2gray.v "

def bin2gray(B, G):
    width = len(B)
    os.system(cmd % width)
    return Cosimulation("vvp -m ../myhdl.vpi bin2gray.o", B=B, G=G)
```

After the executable command argument, the `Cosimulation` constructor takes an arbitrary number of keyword arguments. Those arguments make the link between MyHDL Signals and HDL nets, regs, or signals, by named association. The keyword is the name of an argument in a `$to_myhdl` or `$from_myhdl` call; the argument is a MyHDL Signal.

With all this in place, we can now use the existing unit test to verify the Verilog implementation. Note that we kept the same name and parameters for the the `bin2gray` function: all we need to do is to provide this alternative definition to the existing unit test.

Let's try it on the Verilog design:

```
module bin2gray(B, G);

   parameter width = 8;
   input [width-1:0]  B;
   output [width-1:0] G;

   assign G = (B >> 1) ^ B;

endmodule // bin2gray
```

When we run our unit tests, we get:

```
% python test_gray.py
testSingleBitChange (test_gray_properties.TestGrayCodeProperties)
Check that only one bit changes in successive codewords. ... ok
testUniqueCodeWords (test_gray_properties.TestGrayCodeProperties)
Check that all codewords occur exactly once. ... ok
testOriginalGrayCode (test_gray_original.TestOriginalGrayCode)
Check that the code is an original Gray code. ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.706s

OK
```

Restrictions

In the ideal case, it should be possible to simulate any HDL description seamlessly with MyHDL. Moreover the communicating signals at each side should act transparently as a single one, enabling fully race free operation.

For various reasons, it may not be possible or desirable to achieve full generality. As anyone that has developed applications with the Verilog PLI can testify, the restrictions in a particular simulator, and the differences over various simulators, can be quite frustrating. Moreover, full generality may require a disproportionate amount of development work compared to a slightly less general solution that may be sufficient for the target application.

Consequently, I have tried to achieve a solution which is simple enough so that one can reasonably expect that any PLI-enabled simulator can support it, and that is relatively easy to verify and maintain. At the same time, the solution is sufficiently general to cover the target application space.

The result is a compromise that places certain restrictions on the HDL code. In this section, these restrictions are presented.

Only passive HDL can be co-simulated

The most important restriction of the MyHDL co-simulation solution is that only "passive" HDL can be co-simulated. This means that the HDL code should not contain any statements with time delays. In other words, the MyHDL simulator should be the master of time; in particular, any clock signal should be generated at the MyHDL side.

At first this may seem like an important restriction, but if one considers the target application for co-simulation, it probably isn't.

MyHDL supports co-simulation so that test benches for HDL designs can be written in Python. Let's consider the nature of the target HDL designs. For high-level, behavioral models that are not intended for implementation, it should come as no surprise that I would recommend to write them in MyHDL directly; that is one of the goals of the MyHDL effort. Likewise, gate level designs with annotated timing are not the target application: static timing analysis is a much better verification method for such designs.

Rather, the targeted HDL designs are naturally models that are intended for implementation, most likely through synthesis. As time delays are meaningless in synthesizable code, the restriction is compatible with the target application.

Race sensitivity issues

In a typical RTL code, some events cause other events to occur in the same time step. For example, when a clock signal triggers some signals may change in the same time step. For race-free operation, an HDL must differentiate between such events within a time step. This is done by the concept of "delta" cycles. In a fully general, race free co-simulation, the co-simulators would communicate at the level of delta cycles. However, in MyHDL co-simulation, this is not entirely the case.

Delta cycles from the MyHDL simulator toward the HDL co-simulator are preserved. However, in the opposite direction, they are not. The signals changes are only returned to the MyHDL simulator after all delta cycles have been performed in the HDL co-simulator.

What does this mean? Let's start with the good news. As explained in the previous section, the concept behind MyHDL co-simulation implies that clocks are generated at the MyHDL side. *When using a MyHDL clock and its corresponding HDL signal directly as a clock, co-simulation is race free.* In other words, the case that most closely reflects the MyHDL co-simulation approach, is race free.

The situation is different when you want to use a signal driven by the HDL (and the corresponding MyHDL signal) as a clock. Communication triggered by such a clock is not race free. The solution is to treat such an interface as a chip interface instead of an RTL interface. For example, when data is triggered at positive clock edges, it can safely be sampled at negative clock edges. Alternatively, the MyHDL data signals can be declared with a delay value, so that they are guaranteed to change after the clock edge.

Implementation notes

This section requires some knowledge of PLI terminology.

Enabling a simulator for co-simulation requires a PLI module written in C. In Verilog, the PLI is part of the "standard". However, different simulators implement different versions and portions of the standard. Worse yet, the behavior of certain PLI callbacks is not defined on some essential points. As a result, one should plan to write or at least customize a specific PLI module for any simulator. The release contains a PLI module for the open source Icarus and Cver simulators.

This section documents the current approach and status of the PLI module implementation and some reflections on future implementations.

Icarus Verilog

Delta cycle implementation

To make co-simulation work, a specific type of PLI callback is needed. The callback should be run when all pending events have been processed, while allowing the creation of new events in the current time step (e.g. by the MyHDL simulator). In some Verilog simulators, the

`cbReadWriteSync` callback does exactly that. However, in others, including Icarus, it does not. The callback's behavior is not fully standardized; some simulators run the callback before non- blocking assignment events have been processed.

Consequently, I had to look for a workaround. One half of the solution is to use the `cbReadOnlySync` callback. This callback runs after all pending events have been processed. However, it does not permit to create new events in the current time step. The second half of the solution is to map MyHDL delta cycles onto real Verilog time steps. Note that fortunately I had some freedom here because of the restriction that only passive HDL code can be co-simulated.

I chose to make the time granularity in the Verilog simulator a 1000 times finer than in the MyHDL simulator. For each MyHDL time step, 1000 Verilog time steps are available for MyHDL delta cycles. In practice, only a few delta cycles per time step should be needed. Exceeding this limit almost certainly indicates a design error; the limit is checked at run-time. The factor 1000 also makes it easy to distinguish "real" time from delta cycle time when printing out the Verilog time.

Passive Verilog check

As explained before, co-simulated Verilog should not contain delay statements. Ideally, there should be a run-time check to flag non-compliant code. However, there is currently no such check in the Icarus module.

The check can be written using the `cbNextSimTime` VPI callback in Verilog. However, Icarus 0.7 doesn't support this callback. In the meantime, support for it has been added to the Icarus development branch. When Icarus 0.8 is released, a check will be added.

In the mean time, just don't do this. It may appear to "work" but it really won't as events will be missed over the co-simulation interface.

Cver

MyHDL co-simulation is supported with the open source Verilog simulator Cver. The PLI module is based on the one for Icarus and basically has the same functionality. Only some cosmetic modifications were required.

Other Verilog simulators

The Icarus module is written with VPI calls, which are provided by the most recent generation of the Verilog PLI. Some simulators may only support TF/ACC calls, requiring a complete redesign of the interface module.

If the simulator supports VPI, the Icarus module should be reusable to a large extent. However, it may be possible to improve on it. The workaround to support delta cycles described in Section *Delta cycle implementation* may not be necessary. In some simulators, the `cbReadWriteSync` callback occurs after all events (including non-blocking assignments) have been processed. In that case, the functionality can be supported without a finer time granularity in the Verilog simulator.

There are also Verilog standardization efforts underway to resolve the ambiguity of the `cbReadWriteSync` callback. The solution will be to introduce new, well defined callbacks.

From reading some proposals, I conclude that the `cbEndOfSimTime` callback would provide the required functionality.

The MyHDL project currently has no access to commercial Verilog simulators, so progress in co-simulation support depends on external interest and participation. Users have reported that they are using MyHDL co-simulation with the simulators from Aldec and Modelsim.

Interrupted system calls

The PLI module uses `read` and `write` system calls to communicate between the co-simulators. The implementation assumes that these calls are restarted automatically by the operating system when interrupted. This is apparently what happens on the Linux box on which MyHDL is developed.

It is known how non-restarted interrupted system calls should be handled, but currently such code cannot be tested on the MyHDL development platform. Also, it is not clear whether this is still a relevant issue with modern operating systems. Therefore, this issue has not been addressed at this moment. However, assertions have been included that should trigger when this situation occurs.

Whenever an assertion fires in the PLI module, please report it. The same holds for Python exceptions that cannot be easily explained.

What about VHDL?

It would be nice to have an interface to VHDL simulators such as the Modelsim VHDL simulator. Let us summarize the requirements to accomplish that:

- We need a procedural interface to the internals of the simulator.
- The procedural interface should be a widely used industry standard so that we can reuse the work in several simulators.
- MyHDL is an open-source project and therefore there should be also be an open-source simulator that implements the procedural interface.

`vpi` for Verilog matches these requirements. It is a widely used standard and is supported by the open-source Verilog simulators Icarus and cver.

However, for VHDL the situation is different. While there exists a standard called `vhpi`, it much less popular than `vpi`. Also, to our knowledge there is only one credible open source VHDL simulator (GHDL) and it is unclear whether it has `vhpi` capabilities that are powerful enough for MyHDL's purposes.

Consequently, the development of co-simulation for VHDL is currently on hold. For some applications, there is an alternative: see *Conversion of test benches*.

CHAPTER 10

Conversion to Verilog and VHDL

Introduction

Subject to some limitations, MyHDL supports the automatic conversion of MyHDL code to Verilog or VHDL code. This feature provides a path from MyHDL into a standard Verilog or VHDL based design environment.

This chapter describes the concepts of conversion. Concrete examples can be found in the companion chapter *Conversion examples*.

Solution description

To be convertible, the hardware description should satisfy certain restrictions, defined as the *convertible subset*. This is described in detail in *The convertible subset*.

A convertible design can be converted to an equivalent model in Verilog or VHDL, using the function `toVerilog` or `toVHDL` from the MyHDL library.

When the design is intended for implementation a third-party *synthesis tool* is used to compile the Verilog or VHDL model into an implementation for an ASIC or FPGA. With this step, there is an automated path from a hardware description in Python to an FPGA or ASIC implementation.

The conversion does not start from source files, but from an instantiated design that has been *elaborated* by the Python interpreter. The converter uses the Python profiler to track the interpreter's operation and to infer the design structure and name spaces. It then selectively compiles pieces of source code for additional analysis and for conversion.

Features

Conversion after elaboration *Elaboration* refers to the initial processing of a hardware description to achieve a representation of a design instance that is ready for simulation or synthesis. In particular, structural parameters and constructs are processed in this step. In MyHDL, the Python interpreter itself is used for elaboration. A `Simulation` object is constructed with elaborated design instances as arguments. Likewise, conversion works on an elaborated design instance. The Python interpreter is thus used as much as possible.

Arbitrarily complex structure As the conversion works on an elaborated design instance, any modeling constraints only apply to the leaf elements of the design structure, that is, the co-operating generators. In other words, there are no restrictions on the description of the design structure: Python's full power can be used for that purpose. Also, the design hierarchy can be arbitrarily deep.

Generator are mapped to Verilog or VHDL constructs The converter analyzes the code of each generator and maps it to equivalent constructs in the target HDL. For Verilog, it will map generators to `always` blocks, continuous assignments or `initial` blocks. For VHDL, it will map them to `process` statements or concurrent signal assignments.

The module ports are inferred from signal usage In MyHDL, the input or output direction of ports is not explicitly declared. The converter investigates signal usage in the design hierarchy to infer whether a signal is used as input, output, or as an internal signal.

Interfaces are convertible An *interface*: an object that has a number of `Signal` objects as its attributes. The convertor supports this by name expansion and mangling.

Function calls are mapped to Verilog or VHDL subprograms The converter analyzes function calls and function code. Each function is mapped to an appropriate subprogram in the target HDL: a function or task in Verilog, and a function or procedure in VHDL. In order to support the full power of Python functions, a unique subprogram is generated per Python function call.

If-then-else structures may be mapped to case statements Python does not provide a case statement. However, the converter recognizes if-then-else structures in which a variable is sequentially compared to items of an enumeration type, and maps such a structure to a Verilog or VHDL case statement with the appropriate synthesis attributes.

Choice of encoding schemes for enumeration types The `enum` function in MyHDL returns an enumeration type. This function takes an additional parameter *encoding* that specifies the desired encoding in the implementation: binary, one hot, or one cold. The converter generates the appropriate code for the specified encoding.

RAM memory Certain synthesis tools can map Verilog memories or VHDL arrays to RAM structures. To support this interesting feature, the converter maps lists of signals to Verilog memories or VHDL arrays.

ROM memory Some synthesis tools can infer a ROM from a case statement. The converter does the expansion into a case statement automatically, based on a higher level description. The ROM access is described in a single line, by indexing into a tuple of integers.

Signed arithmetic In MyHDL, working with negative numbers is trivial: one just uses an `intbv` object with an appropriate constraint on its values. In contrast, both Verilog and VHDL make a difference between an unsigned and a signed representation. To work

with negative values, the user has to declare a signed variable explicitly. But when signed and unsigned operands are mixed in an expression, things may become tricky.

In Verilog, when signed and unsigned operands are mixed, all operands are interpreted as *unsigned*. Obviously, this leads to unexpected results. The designer will have to add sign extensions and type casts to solve this.

In VHDL, mixing signed and unsigned will generally not work. The designer will have to match the operands manually by adding resizings and type casts.

In MyHDL, these issues don't exist because `intbv` objects simply work as (constrained) integers. Moreover, the convertor automates the cumbersome tasks that are required in Verilog and VHDL. It uses signed or unsigned types based on the value constraints of the intbv objects, and automatically performs the required sign extensions, resizings, and type casts.

User-defined code If desired, the user can bypass the conversion process and describe user-defined code to be inserted instead.

The convertible subset

Introduction

Unsurprisingly, not all MyHDL code can be converted. Although the restrictions are significant, the convertible subset is much broader than the RTL synthesis subset which is an industry standard. In other words, MyHDL code written according to the RTL synthesis rules, should always be convertible. However, it is also possible to write convertible code for non-synthesizable models or test benches.

The converter attempts to issue clear error messages when it encounters a construct that cannot be converted.

Recall that any restrictions only apply to the design after elaboration. In practice, this means that they apply only to the code of the generators, that are the leaf functional blocks in a MyHDL design.

Coding style

A natural restriction on convertible code is that it should be written in MyHDL style: cooperating generators, communicating through signals, and with sensitivity specify resume conditions.

For pure modeling, it doesn't matter how generators are created. However, in convertible code they should be created using one of the MyHDL decorators: *instance*, *always*, *always_seq*, or *always_comb*.

Supported types

The most important restriction regards object types. Only a limited amount of types can be converted. Python `int` and `long` objects are mapped to Verilog or VHDL integers. All other supported types need to have a defined bit width. The supported types are the Python `bool` type, the MyHDL *intbv* type, and MyHDL enumeration types returned by function *enum*.

intbv objects must be constructed so that a bit width can be inferred. This can be done by specifying minimum and maximum values, e.g. as follows:

```
index = intbv(0, min=MIN, max=MAX)
```

The Verilog converter supports *intbv* objects that can take negative values.

Alternatively, a slice can be taken from an *intbv* object as follows:

```
index = intbv(0)[N:]
```

Such as slice returns a new *intbv* object, with minimum value `0` , and maximum value `2**N`.

In addition to the scalar types described above, the convertor also supports a number of tuple and list based types. The mapping from MyHDL types is summarized in the following table.

MyHDL type	VHDL type	Notes	Verilog type	Notes
`int`	`integer`		`integer`	
`bool`	`std_logic`	(1)	`reg`	
`intbv` with `min >= 0`	`unsigned`	(2)	`reg`	
`intbv` with `min < 0`	`signed`	(2)	`reg signed`	
`enum`	dedicated enumeration type		`reg`	
`tuple` of `int`	mapped to case statement	(3)	mapped to case statement	(3)
`list` of `bool`	`array of std_logic`		`reg`	(5)
`list` of `intbv` with `min >= 0`	`array of unsigned`	(4)	`reg`	(4)(5)
`list` of `intbv` with `min < 0`	`array of signed`	(4)	`reg signed`	(4)(5)

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.
5. Lists are mapped to Verilog memories.

The table as presented applies to MyHDL variables. The convertor also supports MyHDL signals that use `bool`, `intbv` or `enum` objects as their underlying type. For VHDL, these are mapped to VHDL signals with an underlying type as specified in the table above. Verilog doesn't have the signal concept. For Verilog, a MyHDL signal is mapped to a Verilog `reg` as in the table above, or to a Verilog `wire`, depending on the signal usage.

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, or to a Verilog memory. However, list of signals are not always mapped to a corresponding VHDL or Verilog object. See *Conversion of lists of signals* for more info.

Supported statements

The following is a list of the statements that are supported by the Verilog converter, possibly qualified with restrictions or usage notes.

assert An `assert` statement in Python looks as follow:

```
assert test_expression
```

It can be converted provided `test_expression` is convertible.

`break`

`continue`

`def`

for The only supported iteration scheme is iterating through sequences of integers returned by built-in function `range` or MyHDL function *`downrange`*. The optional `else` clause is not supported.

if `if`, `elif`, and `else` clauses are fully supported.

`pass`

`print`

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.

raise This statement is mapped to statements that end the simulation with an error message.

`return`

yield A *yield* expression is used to specify a sensitivity list. The yielded expression can be a *`Signal`*, a signal edge as specified by MyHDL functions *`Signal.posedge`* or *`Signal.negedge`*, or a tuple of signals and edge specifications. It can also be a *`delay`* object.

while The optional `else` clause is not supported.

Supported built-in functions

The following is a list of the built-in functions that are supported by the converter.

bool This function can be used to typecast an object explicitly to its boolean interpretation.

len For *Signal* and *intbv* objects, function len returns the bit width.

int This function can be used to typecast an object explicitly to its integer interpretation.

Docstrings

The convertor 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 at 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.

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 MyHDL 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.

Conversion of Interfaces

Complex designs often have many signals that are passed to different levels of hierarchy. Typically, many signals logically belong together. This can be modelled by an *interface*: an object that has a number of *Signal* objects as its attributes. Grouping signals into an interface simplifies the code, improves efficiency, and reduces errors.

The convertor supports interface using hierarchical name expansion and name mangling.

Assignment issues

Name assignment in Python

Name assignment in Python is a different concept than in many other languages. This point is very important for effective modeling in Python, and even more so for synthesizable MyHDL code. Therefore, the issues are discussed here explicitly.

Consider the following name assignments:

```
a = 4
a = ``a string''
a = False
```

In many languages, the meaning would be that an existing variable *a* gets a number of different values. In Python, such a concept of a variable doesn't exist. Instead, assignment merely creates a new binding of a name to a certain object, that replaces any previous binding. So in the example, the name *a* is bound a number of different objects in sequence.

The converter has to investigate name assignment and usage in MyHDL code, and to map names to Verilog or VHDL objects. To achieve that, it tries to infer the type and possibly the bit width of each expression that is assigned to a name.

Multiple assignments to the same name can be supported if it can be determined that a consistent type and bit width is being used in the assignments. This can be done for boolean expressions, numeric expressions, and enumeration type literals.

In other cases, a single assignment should be used when an object is created. Subsequent value changes are then achieved by modification of an existing object. This technique should be used for *Signal* and *intbv* objects.

Signal assignment

Signal assignment in MyHDL is implemented using attribute assignment to attribute `next`. Value changes are thus modeled by modification of the existing object. The converter investigates the *Signal* object to infer the type and bit width of the corresponding Verilog or VHDL object.

`intbv` objects

Type *intbv* is likely to be the workhorse for synthesizable modeling in MyHDL. An *intbv* instance behaves like a (mutable) integer whose individual bits can be accessed and modified. Also, it is possible to constrain its set of values. In addition to error checking, this makes it possible to infer a bit width, which is required for implementation.

As noted before, it is not possible to modify value of an *intbv* object using name assignment. In the following, we will show how it can be done instead. Consider:

```
a = intbv(0)[8:]
```

This is an *intbv* object with initial value `0` and bit width 8. To change its value to `5`, we can use slice assignment:

```
a[8:] = 5
```

The same can be achieved by leaving the bit width unspecified, which has the meaning to change "all" bits:

```
a[:] = 5
```

Often the new value will depend on the old one. For example, to increment an *intbv* with the technique above:

```
a[:] = a + 1
```

Python also provides *augmented* assignment operators, which can be used to implement in-place operations. These are supported on *intbv* objects and by the converter, so that the increment can also be done as follows:

```
a += 1
```

Excluding code from conversion

For some tasks, such as debugging, it may be useful to insert arbitrary Python code that should not be converted.

The convertor supports this by ignoring all code that is embedded in a `if __debug__` test. The value of the `__debug__` variable is not taken into account.

User-defined code

MyHDL provides a way to include user-defined code during the conversion process. There are special function attributes that are understood by the converter but ignored by the simulator. The attributes are `verilog_code` for Verilog and `vhdl_code` for VHDL. They operate like a special return value. When defined in a MyHDL function, the convertor will use their value instead of the regular return value. Effectively, it will stop converting at that point.

The value of `verilog_code` or `vhdl_code` should be a Python template string. A template string supports `$`-based substitutions. The `$name` notation can be used to refer to the variable names in the context of the string. The convertor will substitute the appropriate values in the string and then insert it instead of the regular converted output.

There is one more issue with user-defined code. Normally, the converter infers inputs, internal signals, and outputs. It also detects undriven and multiple driven signals. To do this, it assumes that signals are not driven by default. It then processes the code to find out which signals are driven from where.

Proper signal usage inference cannot be done with user-defined code. Without user help, this will result in warnings or errors during the inference process, or in compilation errors from invalid code. The user can solve this by setting the `driven` or `read` attribute for signals that are driven or read from the user-defined code. These attributes are `False` by default. The allowed "true" values of the `driven` attribute are `True`, `'wire'` and `'reg'`. The latter two values specifies how the user-defined Verilog code drives the signal in Verilog. To decide which

value to use, consider how the signal should be declared in Verilog after the user-defined code is inserted.

For an example of user-defined code, see *User-defined code*.

Template transformation

Note: This section is only relevant for VHDL.

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.

Conversion output verification by co-simulation

Note: This section is only relevant for Verilog.

To verify the converted Verilog output, co-simulation can be used. To make this task easier, the converter also generates a test bench that makes it possible to simulate the Verilog design using the Verilog co-simulation interface. This permits to verify the Verilog code with the same test bench used for the MyHDL code.

Conversion of test benches

After conversion, we obviously want to verify that the VHDL or Verilog code works correctly. For Verilog, we can use co-simulation as discussed earlier. However, for VHDL, co-simulation is currently not supported.

An 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, as there are important constraints on the kind of code that can be converted. Thus, 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 regard the types that can be used, as discussed earlier in this chapter. However, the "convertible subset" is wider than the "synthesis subset". We will present a number of non-synthesizable feature that are of interest for test benches.

the `while` loop `while` loops can be used for high-level control structures.

the `raise` statement A `raise` statement can stop the simulation on an error condition.

***`delay`* objects** Delay modeling is essential for test benches.

the `print` statement `print` statements can be used for simple debugging.

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 convertible, a whole unit test suite in MyHDL can be converted to an equivalent test suite in Verilog and VHDL.

Additionally, 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.

Methodology notes

Simulate first

In the Python philosophy, the run-time rules. The Python compiler doesn't attempt to detect a lot of errors beyond syntax errors, which given Python's ultra-dynamic nature would be an almost impossible task anyway. To verify a Python program, one should run it, preferably using unit testing to verify each feature.

The same philosophy should be used when converting a MyHDL description to Verilog: make sure the simulation runs fine first. Although the converter checks many things and attempts to issue clear error messages, there is no guarantee that it does a meaningful job unless the simulation runs fine.

Handling hierarchy

Recall that conversion occurs after elaboration. A consequence is that the converted output is non-hierarchical. In many cases, this is not an issue. The purpose of conversion is to provide a path into a traditional design flow by using Verilog and VHDL as a "back-end" format. Hierarchy is quite relevant to humans, but much less so to tools.

However, in some cases hierarchy is desirable. For example, if you convert a test bench you may prefer to keep its code separate from the design under test. In other words, conversion should stop at the design under test instance, and insert an instantiation instead.

There is a workaround to accomplish this with a small amount of additional work. The workaround is to define user-defined code consisting of an instantiation of the design under test. As discussed in *User-defined code*, when the convertor sees the hook it will stop converting and insert the instantiation instead. Of course, you will want to convert the design under test itself also. Therefore, you should use a flag that controls whether the hook is defined or not and set it according to the desired conversion.

Known issues

Verilog and VHDL integers are 32 bit wide Usually, Verilog and VHDL integers are 32 bit wide. In contrast, Python is moving toward integers with undefined width. Python `int` and `long` variables are mapped to Verilog integers; so for values wider than 32 bit this mapping is incorrect.

Synthesis pragmas are specified as Verilog comments. The recommended way to specify synthesis pragmas in Verilog is through attribute lists. However, the Icarus simulator doesn't support them for `case` statements (to specify `parallel_case` and `full_case` pragmas). Therefore, the old but deprecated method of synthesis pragmas in Verilog comments is still used.

Inconsistent place of the sensitivity list inferred from `always_comb`. The semantics of `always_comb`, both in Verilog and MyHDL, is to have an implicit sensitivity list at the end of the code. However, this may not be synthesizable. Therefore, the inferred sensitivity list is put at the top of the corresponding `always` block or `process`. This may cause inconsistent behavior at the start of the simulation. The workaround is to create events at time 0.

CHAPTER 11

Conversion examples

Introduction

In this chapter, we will demonstrate the conversion process with a number of examples. For the concepts of MyHDL conversion, read the companion chapter *Conversion to Verilog and VHDL*.

A small sequential design

Consider the following MyHDL code for an incrementer block:

```
from myhdl import block, always_seq

@block
def inc(count, enable, clock, reset):
    """ Incrementer with enable.

    count -- output
    enable -- control input, increment when 1
    clock -- clock input
    reset -- asynchronous reset input
    """

    @always_seq(clock.posedge, reset=reset)
    def seq():
        if enable:
            count.next = count + 1

    return seq
```

This design can be converted to Verilog and VHDL. The first step is to elaborate it, just as we do for simulation. Then we can use the `convert` method on the elaborated instance.

```
from myhdl import Signal, ResetSignal, modbv

from inc import inc

def convert_inc(hdl):
    """Convert inc block to Verilog or VHDL."""

    m = 8

    count = Signal(modbv(0)[m:])
    enable = Signal(bool(0))
    clock  = Signal(bool(0))
    reset = ResetSignal(0, active=0, async=True)

    inc_1 = inc(count, enable, clock, reset)

    inc_1.convert(hdl=hdl)


convert_inc(hdl='Verilog')
convert_inc(hdl='VHDL')
```

For flexibility, we wrap the conversion in a `convert_inc` function. `inc_1` is an elaborated design instance that provides the conversion method.

The conversion to Verilog generates an equivalent Verilog module in file `inc.v`. The Verilog code looks as follows:

```
// File: inc.v
// Generated by MyHDL 1.0dev
// Date: Sun May 22 18:46:37 2016


`timescale 1ns/10ps

module inc (
    count,
    enable,
    clock,
    reset
);
// Incrementer with enable.
//
// count -- output
// enable -- control input, increment when 1
// clock -- clock input
// reset -- asynchronous reset input

output [7:0] count;
reg [7:0] count;
input enable;
input clock;
input reset;



always @(posedge clock, negedge reset) begin: INC_SEQ
```

```
    if (reset == 0) begin
        count <= 0;
    end
    else begin
        if (enable) begin
            count <= (count + 1);
        end
    end
end

endmodule
```

The convertor infers a proper Verilog module interface and maps the MyHDL generator to a Verilog always block.

Similarly, the conversion to VHDL generates a file `inc.vhd` with the following content:

```
-- File: inc.vhd
-- Generated by MyHDL 1.0dev
-- Date: Sun May 22 18:46:37 2016


library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use std.textio.all;

use work.pck_myhdl_10.all;

entity inc is
    port (
        count: inout unsigned(7 downto 0);
        enable: in std_logic;
        clock: in std_logic;
        reset: in std_logic
    );
end entity inc;
-- Incrementer with enable.
--
-- count -- output
-- enable -- control input, increment when 1
-- clock -- clock input
-- reset -- asynchronous reset input

architecture MyHDL of inc is



begin




INC_SEQ: process (clock, reset) is
begin
    if (reset = '0') then
        count <= to_unsigned(0, 8);
```

```
    elsif rising_edge(clock) then
        if bool(enable) then
            count <= (count + 1);
        end if;
    end if;
end process INC_SEQ;

end architecture MyHDL;
```

The MyHDL generator is mapped to a VHDL process in this case.

Note that the VHDL file refers to a VHDL package called `pck_myhdl_<version>`. This package contains a number of convenience functions that make the conversion easier.

Note also the use of an `inout` in the interface. This is not recommended VHDL design practice, but it is required here to have a valid VHDL design that matches the behavior of the MyHDL design. As this is only an issue for ports and as the convertor output is non-hierarchical, the issue is not very common and has an easy workaround.

A small combinatorial design

The second example is a small combinatorial design, more specifically the binary to Gray code converter from previous chapters:

```
from myhdl import block, always_comb

@block
def bin2gray(B, G):
    """ Gray encoder.

    B -- binary input
    G -- Gray encoded output
    """

    @always_comb
    def logic():
        G.next = (B>>1) ^ B

    return logic
```

As before, you can create an instance and convert to Verilog and VHDL as follows:

```
from myhdl import Signal, intbv

from bin2gray import bin2gray

def convert(hdl, width=8):

    B = Signal(intbv(0)[width:])
    G = Signal(intbv(0)[width:])

    inst = bin2gray(B, G)
    inst.convert(hdl=hdl)
```

```
convert(hdl='Verilog')
convert(hdl='VHDL')
```

The generated Verilog code looks as follows:

```
// File: bin2gray.v
// Generated by MyHDL 1.0dev
// Date: Mon May 23 16:09:27 2016


`timescale 1ns/10ps

module bin2gray (
    B,
    G
);
// Gray encoder.
//
// B -- binary input
// G -- Gray encoded output

input [7:0] B;
output [7:0] G;
wire [7:0] G;




assign G = ((B >>> 1) ^ B);

endmodule
```

The generated VHDL code looks as follows:

```
-- File: bin2gray.vhd
-- Generated by MyHDL 1.0dev
-- Date: Mon May 23 16:09:27 2016


library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use std.textio.all;

use work.pck_myhdl_10.all;

entity bin2gray is
    port (
        B: in unsigned(7 downto 0);
        G: out unsigned(7 downto 0)
    );
end entity bin2gray;
-- Gray encoder.
--
-- B -- binary input
-- G -- Gray encoded output
```

```
architecture MyHDL of bin2gray is


begin




G <= (shift_right(B, 1) xor B);

end architecture MyHDL;
```

A hierarchical design

The converter can handle designs with an arbitrarily deep hierarchy.

For example, suppose we want to design an incrementer with Gray code output. Using the designs from previous sections, we can proceed as follows:

```
from myhdl import block, Signal, modbv

from bin2gray import bin2gray
from inc import inc

@block
def gray_inc(graycnt, enable, clock, reset, width):

    bincnt = Signal(modbv(0)[width:])

    inc_0 = inc(bincnt, enable, clock, reset)
    bin2gray_0 = bin2gray(B=bincnt, G=graycnt)

    return inc_0, bin2gray_0
```

According to Gray code properties, only a single bit will change in consecutive values. However, as the `bin2gray` module is combinatorial, the output bits may have transient glitches, which may not be desirable. To solve this, let's create an additional level of hierarchy and add an output register to the design. (This will create an additional latency of a clock cycle, which may not be acceptable, but we will ignore that here.)

```
from myhdl import block, always_seq, Signal, modbv

from gray_inc import gray_inc

@block
def gray_inc_reg(graycnt, enable, clock, reset, width):

    graycnt_comb = Signal(modbv(0)[width:])

    gray_inc_0 = gray_inc(graycnt_comb, enable, clock, reset, width)
```

```
    @always_seq(clock.posedge, reset=reset)
    def reg_0():
        graycnt.next = graycnt_comb

    return gray_inc_0, reg_0
```

We can convert this hierarchical design as follows:

```
from myhdl import Signal, ResetSignal, modbv

from gray_inc_reg import gray_inc_reg

def convert_gray_inc_reg(hdl, width=8):
    graycnt = Signal(modbv(0)[width:])
    enable = Signal(bool())
    clock = Signal(bool())
    reset = ResetSignal(0, active=0, async=True)

    inst = gray_inc_reg(graycnt, enable, clock, reset, width)
    inst.convert(hdl)

convert_gray_inc_reg(hdl='Verilog')
convert_gray_inc_reg(hdl='VHDL')
```

The Verilog output code looks as follows:

```
// File: gray_inc_reg.v
// Generated by MyHDL 1.0dev
// Date: Thu Jun 23 19:06:43 2016


`timescale 1ns/10ps

module gray_inc_reg (
    graycnt,
    enable,
    clock,
    reset
);


output [7:0] graycnt;
reg [7:0] graycnt;
input enable;
input clock;
input reset;

wire [7:0] graycnt_comb;
reg [7:0] gray_inc_1_bincnt;



always @(posedge clock, negedge reset) begin: GRAY_INC_REG_GRAY_INC_1_INC_
↪1_SEQ
    if (reset == 0) begin
```

```
        gray_inc_1_bincnt <= 0;
    end
    else begin
        if (enable) begin
            gray_inc_1_bincnt <= (gray_inc_1_bincnt + 1);
        end
    end
end



assign graycnt_comb = ((gray_inc_1_bincnt >>> 1) ^ gray_inc_1_bincnt);


always @(posedge clock, negedge reset) begin: GRAY_INC_REG_REG_0
    if (reset == 0) begin
        graycnt <= 0;
    end
    else begin
        graycnt <= graycnt_comb;
    end
end

endmodule
```

The VHDL output code looks as follows:

```
-- File: gray_inc_reg.vhd
-- Generated by MyHDL 1.0dev
-- Date: Thu Jun 23 19:06:43 2016


library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use std.textio.all;

use work.pck_myhdl_10.all;

entity gray_inc_reg is
    port (
        graycnt: out unsigned(7 downto 0);
        enable: in std_logic;
        clock: in std_logic;
        reset: in std_logic
    );
end entity gray_inc_reg;


architecture MyHDL of gray_inc_reg is


signal graycnt_comb: unsigned(7 downto 0);
signal gray_inc_1_bincnt: unsigned(7 downto 0);

begin
```

```
GRAY_INC_REG_GRAY_INC_1_INC_1_SEQ: process (clock, reset) is
begin
    if (reset = '0') then
        gray_inc_1_bincnt <= to_unsigned(0, 8);
    elsif rising_edge(clock) then
        if bool(enable) then
            gray_inc_1_bincnt <= (gray_inc_1_bincnt + 1);
        end if;
    end if;
end process GRAY_INC_REG_GRAY_INC_1_INC_1_SEQ;


graycnt_comb <= (shift_right(gray_inc_1_bincnt, 1) xor gray_inc_1_bincnt);

GRAY_INC_REG_REG_0: process (clock, reset) is
begin
    if (reset = '0') then
        graycnt <= to_unsigned(0, 8);
    elsif rising_edge(clock) then
        graycnt <= graycnt_comb;
    end if;
end process GRAY_INC_REG_REG_0;

end architecture MyHDL;
```

Note that the output is a flat "net list of blocks", and that hierarchical signal names are generated as necessary.

Optimizations for finite state machines

As often in hardware design, finite state machines deserve special attention.

In Verilog and VHDL, finite state machines are typically described using case statements. Python doesn't have a case statement, but the converter recognizes particular if-then-else structures and maps them to case statements. This optimization occurs when a variable whose type is an enumerated type is sequentially tested against enumeration items in an if-then-else structure. Also, the appropriate synthesis pragmas for efficient synthesis are generated in the Verilog code.

As a further optimization, function *enum* was enhanced to support alternative encoding schemes elegantly, using an additional parameter *encoding*. For example:

```
t_State = enum('SEARCH', 'CONFIRM', 'SYNC', encoding='one_hot')
```

The default encoding is `'binary'`; the other possibilities are `'one_hot'` and `'one_cold'`. This parameter only affects the conversion output, not the behavior of the type. The generated Verilog code for case statements is optimized for an efficient implementation according to the encoding. Note that in contrast, a Verilog designer has to make nontrivial code changes to implement a different encoding scheme.

As an example, consider the following finite state machine, whose state variable uses the enu-

meration type defined above:

```
ACTIVE_LOW = bool(0)
FRAME_SIZE = 8
t_State = enum('SEARCH', 'CONFIRM', 'SYNC', encoding="one_hot")

def FramerCtrl(SOF, state, syncFlag, clk, reset_n):

    """ Framing control FSM.

    SOF -- start-of-frame output bit
    state -- FramerState output
    syncFlag -- sync pattern found indication input
    clk -- clock input
    reset_n -- active low reset

    """

    index = Signal(intbv(0)[8:]) # position in frame

    @always(clk.posedge, reset_n.negedge)
    def FSM():
        if reset_n == ACTIVE_LOW:
            SOF.next = 0
            index.next = 0
            state.next = t_State.SEARCH
        else:
            index.next = (index + 1) % FRAME_SIZE
            SOF.next = 0
            if state == t_State.SEARCH:
                index.next = 1
                if syncFlag:
                    state.next = t_State.CONFIRM
            elif state == t_State.CONFIRM:
                if index == 0:
                    if syncFlag:
                        state.next = t_State.SYNC
                    else:
                        state.next = t_State.SEARCH
            elif state == t_State.SYNC:
                if index == 0:
                    if not syncFlag:
                        state.next = t_State.SEARCH
                SOF.next = (index == FRAME_SIZE-1)
            else:
                raise ValueError("Undefined state")

    return FSM
```

The conversion is done as before:

```
SOF = Signal(bool(0))
syncFlag = Signal(bool(0))
clk = Signal(bool(0))
reset_n = Signal(bool(1))
state = Signal(t_State.SEARCH)
toVerilog(FramerCtrl, SOF, state, syncFlag, clk, reset_n)
toVHDL(FramerCtrl, SOF, state, syncFlag, clk, reset_n)
```

The Verilog output looks as follows:

```
module FramerCtrl (
    SOF,
    state,
    syncFlag,
    clk,
    reset_n
);

output SOF;
reg SOF;
output [2:0] state;
reg [2:0] state;
input syncFlag;
input clk;
input reset_n;

reg [7:0] index;



always @(posedge clk, negedge reset_n) begin: FRAMERCTRL_FSM
    if ((reset_n == 0)) begin
        SOF <= 0;
        index <= 0;
        state <= 3'b001;
    end
    else begin
        index <= ((index + 1) % 8);
        SOF <= 0;
        // synthesis parallel_case full_case
        casez (state)
            3'b??1: begin
                index <= 1;
                if (syncFlag) begin
                    state <= 3'b010;
                end
            end
            3'b?1?: begin
                if ((index == 0)) begin
                    if (syncFlag) begin
                        state <= 3'b100;
                    end
                    else begin
                        state <= 3'b001;
                    end
                end
            end
            3'b1??: begin
                if ((index == 0)) begin
                    if ((!syncFlag)) begin
                        state <= 3'b001;
                    end
                end
                SOF <= (index == (8 - 1));
```

```
            end
            default: begin
                $finish;
            end
        endcase
    end
end

endmodule
```

The VHDL output looks as follows:

```
package pck_FramerCtrl is

    type t_enum_t_State_1 is (
    SEARCH,
    CONFIRM,
    SYNC
);
attribute enum_encoding of t_enum_t_State_1: type is "001 010 100";

end package pck_FramerCtrl;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use std.textio.all;

use work.pck_myhdl_06.all;

use work.pck_FramerCtrl.all;

entity FramerCtrl is
    port (
        SOF: out std_logic;
        state: inout t_enum_t_State_1;
        syncFlag: in std_logic;
        clk: in std_logic;
        reset_n: in std_logic
    );
end entity FramerCtrl;

architecture MyHDL of FramerCtrl is

signal index: unsigned(7 downto 0);

begin


FRAMERCTRL_FSM: process (clk, reset_n) is
begin
    if (reset_n = '0') then
        SOF <= '0';
        index <= "00000000";
        state <= SEARCH;
    elsif rising_edge(clk) then
        index <= ((index + 1) mod 8);
```

```
        SOF <= '0';
        case state is
            when SEARCH =>
                index <= "00000001";
                if to_boolean(syncFlag) then
                    state <= CONFIRM;
                end if;
            when CONFIRM =>
                if (index = 0) then
                    if to_boolean(syncFlag) then
                        state <= SYNC;
                    else
                        state <= SEARCH;
                    end if;
                end if;
            when SYNC =>
                if (index = 0) then
                    if (not to_boolean(syncFlag)) then
                        state <= SEARCH;
                    end if;
                end if;
                SOF <= to_std_logic(signed(resize(index, 9)) = (8 - 1));
            when others =>
                assert False report "End of Simulation" severity Failure;
        end case;
    end if;
end process FRAMERCTRL_FSM;

end architecture MyHDL;
```

RAM inference

Certain synthesis tools can infer RAM structures. To support this feature, the converter maps lists of signals in MyHDL to Verilog memories and VHDL arrays.

The following MyHDL example is a ram model that uses a list of signals to model the internal memory.

```
def RAM(dout, din, addr, we, clk, depth=128):
    """  Ram model """

    mem = [Signal(intbv(0)[8:]) for i in range(depth)]

    @always(clk.posedge)
    def write():
        if we:
            mem[addr].next = din

    @always_comb
    def read():
        dout.next = mem[addr]

    return write, read
```

With the appropriate signal definitions for the interface ports, it is converted to the following

Verilog code. Note how the list of signals `mem` is mapped to a Verilog memory.

```
module ram (
    dout,
    din,
    addr,
    we,
    clk
);

output [7:0] dout;
wire [7:0] dout;
input [7:0] din;
input [6:0] addr;
input we;
input clk;


reg [7:0] mem [0:128-1];


always @(posedge clk) begin: RAM_1_WRITE
    if (we) begin
        mem[addr] <= din;
    end
end


assign dout = mem[addr];

endmodule
```

In VHDL, the list of MyHDL signals is modeled as a VHDL array signal:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

use work.pck_myhdl_06.all;

entity ram is
    port (
        dout: out unsigned(7 downto 0);
        din: in unsigned(7 downto 0);
        addr: in unsigned(6 downto 0);
        we: in std_logic;
        clk: in std_logic
    );
end entity ram;

architecture MyHDL of ram is

type t_array_mem is array(0 to 128-1) of unsigned(7 downto 0);
signal mem: t_array_mem;

begin

RAM_WRITE: process (clk) is
```

```
begin
    if rising_edge(clk) then
        if to_boolean(we) then
            mem(to_integer(addr)) <= din;
        end if;
    end if;
end process RAM_WRITE;


dout <= mem(to_integer(addr));

end architecture MyHDL;
```

ROM inference

Some synthesis tools can infer a ROM memory from a case statement. The Verilog converter can perform the expansion into a case statement automatically, based on a higher level description. The ROM access is described in a single line, by indexing into a tuple of integers. The tuple can be described manually, but also by programmatical means. Note that a tuple is used instead of a list to stress the read-only character of the memory.

The following example illustrates this functionality. ROM access is described as follows:

```
def rom(dout, addr, CONTENT):

    @always_comb
    def read():
        dout.next = CONTENT[int(addr)]

    return read
```

The ROM content is described as a tuple of integers. When the ROM content is defined, the conversion can be performed:

```
CONTENT = (17, 134, 52, 9)
dout = Signal(intbv(0)[8:])
addr = Signal(intbv(0)[4:])

toVerilog(rom, dout, addr, CONTENT)
toVHDL(rom, dout, addr, CONTENT)
```

The Verilog output code is as follows:

```
module rom (
    dout,
    addr
);

output [7:0] dout;
reg [7:0] dout;
input [3:0] addr;

always @(addr) begin: ROM_READ
    // synthesis parallel_case full_case
```

```
    case (addr)
        0: dout <= 17;
        1: dout <= 134;
        2: dout <= 52;
        default: dout <= 9;
    endcase
end

endmodule
```

The VHDL output code is as follows:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use std.textio.all;

use work.pck_myhdl_06.all;

entity rom is
    port (
        dout: out unsigned(7 downto 0);
        addr: in unsigned(3 downto 0)
    );
end entity rom;

architecture MyHDL of rom is


begin

ROM_READ: process (addr) is
begin
    case to_integer(addr) is
        when 0 => dout <= "00010001";
        when 1 => dout <= "10000110";
        when 2 => dout <= "00110100";
        when others => dout <= "00001001";
    end case;
end process ROM_READ;

end architecture MyHDL;
```

User-defined code

MyHDL provides a way to include user-defined code during the conversion process, using the special function attributes `vhdl_code` and `verilog_code`.

For example:

```
def inc_comb(nextCount, count, n):

    @always(count)
    def logic():
        # do nothing here
```

```
        pass

    nextCount.driven = "wire"

    return logic

inc_comb.verilog_code =\
"""
assign $nextCount = ($count + 1) % $n;
"""

inc_comb.vhdl_code =\
"""
$nextCount <= ($count + 1) mod $n;
"""
```

The converted code looks as follows in Verilog:

```
module inc_comb (
    nextCount,
    count
);

output [7:0] nextCount;
wire [7:0] nextCount;
input [7:0] count;

assign nextCount = (count + 1) % 256;

endmodule
```

and as follows in VHDL:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

use work.pck_myhdl_06.all;

entity inc_comb is
    port (
        nextCount: out unsigned(7 downto 0);
        count: in unsigned(7 downto 0)
    );
end entity inc_comb;

architecture MyHDL of inc_comb is

begin

nextCount <= (count + 1) mod 256;

end architecture MyHDL;
```

In this example, conversion of the `inc_comb` function is bypassed and the user-defined code is inserted instead. The user-defined code is a Python template string that can refer to signals and parameters in the MyHDL context through `$`-based substitutions. During conversion, the

appropriate hierarchical names and parameter values will be substituted.

The MyHDL code contains the following assignment:

```
nextCount.driven = "wire"
```

This specifies that the nextCount signal is driven as a Verilog wire from this module.

For more info about user-defined code, see *User-defined code*.

CHAPTER 12

Reference

MyHDL is implemented as a Python package called `myhdl`. This chapter describes the objects that are exported by this package.

Simulation

The `Simulation` class

class `Simulation`(*arg*[, *arg ...*])
Class to construct a new simulation. Each argument should be a MyHDL instance. In MyHDL, an instance is recursively defined as being either a sequence of instances, or a MyHDL generator, or a Cosimulation object. See section *MyHDL generators and trigger objects* for the definition of MyHDL generators and their interaction with a `Simulation` object. See Section *Co-simulation* for the `Cosimulation` object. At most one `Cosimulation` object can be passed to a `Simulation` constructor.

A `Simulation` object has the following method:

`Simulation.`**`run`**([*duration*])
Run the simulation forever (by default) or for a specified duration.

`Simulation.`**`quit`**()
Quit the simulation after it has run for a specified duration. The method should be called (the simulation instance must be quit) before another simulation instance is created. The method is called by default when the simulation is run forever.

Simulation support functions

`now`()
Returns the current simulation time.

exception StopSimulation
Base exception that is caught by the `Simulation.run()` method to stop a simulation.

Waveform tracing

traceSignals (*func [, *args] [, **kwargs]*)
Enables signal tracing to a VCD file for waveform viewing. *func* is a function that returns an instance. *traceSignals* calls *func* under its control and passes **args* and ***kwargs* to the call. In this way, it finds the hierarchy and the signals to be traced.

The return value is the same as would be returned by the call `func(*args, **kwargs)`. The top-level instance name and the basename of the VCD output filename is `func.func_name` by default. If the VCD file exists already, it will be moved to a backup file by attaching a timestamp to it, before creating the new file.

The `traceSignals` callable has the following attribute:

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

directory
This attribute is used to set the directory to which VCD files are written. By default, the current working directory is used.

filename
This attribute is used to set the filename to which VCD files are written. By default, the name attribbute is used.

timescale
This attribute is used to set the timescale corresponding to unit steps, according to the VCD format. The assigned value should be a string. The default timescale is "1ns".

Modeling

Signals

The SignalType type

class SignalType
This type is the abstract base type of all signals. It is not used to construct signals, but it can be used to check whether an object is a signal.

Regular signals

class Signal (*[val=None] [, delay=0]*)
This class is used to construct a new signal and to initialize its value to *val*. Optionally, a delay can be specified.

A *Signal* object has the following attributes:

posedge
Attribute that represents the positive edge of a signal, to be used in sensitivity lists.

negedge
Attribute that represents the negative edge of a signal, to be used in sensitivity lists.

next
Read-write attribute that represents the next value of the signal.

val
Read-only attribute that represents the current value of the signal.

This attribute is always available to access the current value; however in many practical case it will not be needed. Whenever there is no ambiguity, the Signal object's current value is used implicitly. In particular, all Python's standard numeric, bit-wise, logical and comparison operators are implemented on a Signal object by delegating to its current value. The exception is augmented assignment. These operators are not implemented as they would break the rule that the current value should be a read-only attribute. In addition, when a Signal object is assigned to the `next` attribute of another Signal object, its current value is assigned instead.

min
Read-only attribute that is the minimum value (inclusive) of a numeric signal, or `None` for no minimum.

max
Read-only attribute that is the maximum value (exclusive) of a numeric signal, or `None` for no maximum.

driven
Writable attribute that can be used to indicate that the signal is supposed to be driven from the MyHDL code, and possibly how it should be declared in Verilog after conversion. The allowed values are `'reg'`, `'wire'`, `True` and `False`.

This attribute is useful when the converter cannot infer automatically whether and how a signal is driven. This occurs when the signal is driven from user-defined code. `'reg'` and `'wire'` are "true" values that permit finer control for the Verilog case.

read
Writable boolean attribute that can be used to indicate that the signal is read.

This attribute is useful when the converter cannot infer automatically whether a signal is read. This occurs when the signal is read from user-defined code.

A *Signal* object also has a call interface:

__call__(*left*[, *right=None*])
This method returns a *_SliceSignal* shadow signal.

class ResetSignal(*val*, *active*, *async*)
This Signal subclass defines reset signals. *val*, *active*, and *async* are mandatory arguments.

val is a boolean value that specifies the intial value, *active* is a boolean value that specifies the active level. *async* is a boolean value that specifies the reset style: asynchronous (`True`) or synchronous (`False`).

This class should be used in conjunction with the *`always_seq`* decorator.

Shadow signals

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

This class implements read-only structural slicing and indexing. It creates a new shadow signal of 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, use the call interface of a regular signal.The following calls are equivalent:

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

sl = sig(left, right)
```

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

This class creates a new shadow signal of the concatenation of its arguments.

You can pass an arbitrary number of arguments to the constructor. The arguments should be bit-oriented with a defined number of bits. The following argument types are supported: *`intbv`* objects with a defined bit width, `bool` objects, signals of the previous objects, and bit strings.

The new signal follows the value changes of the signal arguments. The non-signal arguments are used to define constant values in the concatenation.

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.

MyHDL generators and trigger objects

MyHDL generators are standard Python generators with specialized `yield` statements. In hardware description languages, the equivalent statements are called *sensitivity lists*. The general format of `yield` statements in in MyHDL generators is:

`yield` clause [, clause ...]

When a generator executes a `yield` statement, its execution is suspended at that point. At the same time, each *clause* is a *trigger object* which defines the condition upon which the generator should be resumed. However, per invocation of a `yield` statement, the generator resumes exactly once, regardless of the number of clauses. This happens on the first trigger that occurs.

In this section, the trigger objects and their functionality will be described.

Some MyHDL objects that are described elsewhere can directly be used as trigger objects. In particular, a *`Signal`* can be used as a trigger object. Whenever a signal changes value, the generator resumes. Likewise, the objects referred to by the signal attributes `posedge` and `negedge` are trigger objects. The generator resumes on the occurrence of a positive or a negative edge on the signal, respectively. An edge occurs when there is a change from false to true (positive) or vice versa (negative). For the full description of the *`Signal`* class and its attributes, see section *Signals*.

Furthermore, MyHDL generators can be used as clauses in `yield` statements. Such a generator is forked, and starts operating immediately, while the original generator waits for it to complete. The original generator resumes when the forked generator returns.

In addition, the following functions return trigger objects:

`delay`(*t*)
: Return a trigger object that specifies that the generator should resume after a delay *t*.

`join`(*arg*[, *arg* ...])
: Join a number of trigger objects together and return a joined trigger object. The effect is that the joined trigger object will trigger when *all* of its arguments have triggered.

Finally, as a special case, the Python `None` object can be present in a `yield` statement. It is the do-nothing trigger object. The generator immediately resumes, as if no `yield` statement were present. This can be useful if the `yield` statement also has generator clauses: those generators are forked, while the original generator resumes immediately.

Decorator functions

MyHDL defines a number of decorator functions, that make it easier to create generators from local generator functions.

`instance`()
: The *`instance`* decorator is the most general decorator. It automatically creates a generator by calling the decorated generator function.

  It is used as follows:

```
def top(...):
    ...
    @instance
    def inst():
        <generator body>
    ...
    return inst, ...
```

  This is equivalent to:

```
def top(...):
    ...
    def _gen_func():
        <generator body>
    ...
    inst = _gen_func()
    ...
    return inst, ...
```

always(*arg*[, **args*])

The *always* decorator is a specialized decorator that targets a widely used coding pattern. It is used as follows:

```
def top(...):
    ...
    @always(event1, event2, ...)
    def inst()
        <body>
    ...
    return inst, ...
```

This is equivalent to the following:

```
def top(...):
    ...
    def _func():
        <body>

    def _gen_func()
        while True:
            yield event1, event2, ...
            _func()
    ...
    inst = _gen_func()
    ...
    return inst, ...
```

The argument list of the decorator corresponds to the sensitivity list. Only signals, edge specifiers, or delay objects are allowed. The decorated function should be a classic function.

always_comb()

The *always_comb* decorator is used to describe combinatorial logic.

```
def top(...):
    ...
    @always_comb
    def comb_inst():
        <combinatorial body>
    ...
    return comb_inst, ...
```

The *always_comb* decorator infers the inputs of the combinatorial logic and the corresponding sensitivity list automatically. The decorated function should be a classic function.

always_seq(*edge*, *reset*)
The *always_seq* decorator is used to describe sequential (clocked) logic.

The *edge* parameter should be a clock edge (`clock.posedge` or `clock.negedge`). The *reset* parameter should a *ResetSignal* object.

MyHDL data types

MyHDL defines a number of data types that are useful for hardware description.

The intbv class

class intbv(*[val=0] [, min=None] [, max=None]*)
This class represents `int`-like objects with some additional features that make it suitable for hardware design.

The *val* argument can be an `int`, a `long`, an *intbv* or a bit string (a string with only '0's or '1's). For a bit string argument, the value is calculated as in `int(bitstring, 2)`. The optional *min* and *max* arguments can be used to specify the minimum and maximum value of the *intbv* object. As in standard Python practice for ranges, the minimum value is inclusive and the maximum value is exclusive.

The minimum and maximum values of an *intbv* object are available as attributes:

min
Read-only attribute that is the minimum value (inclusive) of an *intbv*, or *None* for no minimum.

max
Read-only attribute that is the maximum value (exclusive) of an *intbv*, or *None* for no maximum.

signed()
Interpretes the msb bit as as sign bit and extends it into the higher-order bits of the underlying object value. The msb bit is the highest-order bit within the object's bit width.

Return type integer

Unlike `int` objects, *intbv* objects are mutable; this is also the reason for their existence. Mutability is needed to support assignment to indexes and slices, as is common in hardware design. For the same reason, *intbv* is not a subclass from `int`, even though `int` provides most of the desired functionality. (It is not possible to derive a mutable subtype from an immutable base type.)

An *intbv* object supports the same comparison, numeric, bitwise, logical, and conversion operations as `int` objects. See http://www.python.org/doc/current/lib/typesnumeric.html for more information on such operations. In all binary operations, *intbv* objects can work together with `int` objects. For mixed-type numeric operations, the result type is an `int` or a `long`. For mixed-type bitwise operations, the result type is an *intbv*.

In addition, *intbv* supports a number of sequence operators. In particular, the `len` function returns the object's bit width. Furthermore, *intbv* objects support indexing and slicing operations:

Operation	Result	Notes
`bv[i]`	item *i* of *bv*	(1)
`bv[i] = x`	item *i* of *bv* is replaced by *x*	(1)
`bv[i:j]`	slice of *bv* from *i* downto *j*	(2)(3)
`bv[i:j] = t`	slice of *bv* from *i* downto *j* is replaced by *t*	(2)(4)

1. Indexing follows the most common hardware design conventions: the lsb bit is the rightmost bit, and it has index 0. This has the following desirable property: if the *intbv* value is decomposed as a sum of powers of 2, the bit with index *i* corresponds to the term `2**i`.
2. In contrast to standard Python sequencing conventions, slicing range are downward. This is a consequence of the indexing convention, combined with the common convention that the most significant digits of a number are the leftmost ones. The Python convention of half-open ranges is followed: the bit with the highest index is not included. However, it is the *leftmost* bit in this case. As in standard Python, this takes care of one-off issues in many practical cases: in particular, `bv[i:]` returns *i* bits; `bv[i:j]` has `i-j` bits. When the low index *j* is omitted, it defaults to `0`. When the high index *i* is omitted, it means "all" higher order bits.
3. The object returned from a slicing access operation is always a positive *intbv*; higher order bits are implicitly assumed to be zero. The bit width is implicitly stored in the return object, so that it can be used in concatenations and as an iterator. In addition, for a bit width w, the *min* and *max* attributes are implicitly set to `0` and `2**w`, respectively.
4. When setting a slice to a value, it is checked whether the slice is wide enough.

In addition, an *intbv* object supports the iterator protocol. This makes it possible to iterate over all its bits, from the high index to index 0. This is only possible for *intbv* objects with a defined bit width.

The `modbv` class

class modbv(*[val=0] [, min=None] [, max=None]*)

The *modbv* class implements modular bit vector types.

It is implemented as a subclass of *intbv* and supports the same parameters and operators. The difference is in the handling of the *min* and *max* boundaries. Instead of throwing an exception when those constraints are exceeded, the value of *modbv* objects wraps around according to the following formula:

```
val = (val - min) % (max - min) + min
```

This formula is a generalization of modulo wrap-around behavior that is often useful when describing hardware system behavior.

The `enum` factory function

enum(*arg [, arg ...] [, encoding='binary']*)

Returns an enumeration type.

The arguments should be string literals that represent the desired names of the enumeration type attributes. The returned type should be assigned to a type name. For example:

```
t_EnumType = enum('ATTR_NAME_1', 'ATTR_NAME_2', ...)
```

The enumeration type identifiers are available as attributes of the type name, for example: `t_EnumType.ATTR_NAME_1`

The optional keyword argument *encoding* specifies the encoding scheme used in Verilog output. The available encodings are `'binary'`, `'one_hot'`, and `'one_cold'`.

Modeling support functions

MyHDL defines a number of additional support functions that are useful for hardware description.

bin

bin (*num*[, *width*])

Returns a bit string representation. If the optional *width* is provided, and if it is larger than the width of the default representation, the bit string is padded with the sign bit.

This function complements the standard Python conversion functions `hex` and `oct`. A binary string representation is often useful in hardware design.

Return type string

concat

concat (*base*[, *arg ...*])

Returns an *intbv* object formed by concatenating the arguments.

The following argument types are supported: *intbv* objects with a defined bit width, `bool` objects, signals of the previous objects, and bit strings. All these objects have a defined bit width.

The first argument *base* is special as it does not need to have a defined bit width. In addition to the previously mentioned objects, unsized *intbv*, `int` and `long` objects are supported, as well as signals of such objects.

Return type *intbv*

downrange

downrange (*high*[, *low=0*])

Generates a downward range list of integers.

This function is modeled after the standard `range` function, but works in the downward direction. The returned interval is half-open, with the *high* index not included. *low* is optional and defaults to zero. This function is especially useful in conjunction with the *intbv* class, that also works with downward indexing.

instances

instances()
Looks up all MyHDL instances in the local name space and returns them in a list.

Return type list

Co-simulation

MyHDL

class Cosimulation(*exe*, ***kwargs*)
Class to construct a new Cosimulation object.

The *exe* argument is the command to execute an HDL simulation, which can be either a string of the entire command line or a list of strings. In the latter case, the first element is the executable, and subsequent elements are program arguments. Providing a list of arguments allows Python to correctly handle spaces or other characters in program arguments.

The *kwargs* keyword arguments provide a named association between signals (regs & nets) in the HDL simulator and signals in the MyHDL simulator. Each keyword should be a name listed in a `$to_myhdl` or `$from_myhdl` call in the HDL code. Each argument should be a *Signal* declared in the MyHDL code.

Verilog

$to_myhdl(arg, [, arg ...])
Task that defines which signals (regs & nets) should be read by the MyHDL simulator. This task should be called at the start of the simulation.

$from_myhdl(arg, [, arg ...])
Task that defines which signals should be driven by the MyHDL simulator. In Verilog, only regs can be specified. This task should be called at the start of the simulation.

Conversion to Verilog and VHDL

Conversion

toVerilog(*func [, *args] [, **kwargs]*)

Converts a MyHDL design instance to equivalent Verilog code, and also generates a test bench to verify it. *func* is a function that returns an instance. *toVerilog* calls *func* under its control and passes **args* and ***kwargs* to the call.

The return value is the same as would be returned by the call `func(*args, **kwargs)`. It should 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.

For more information about the restrictions on convertible MyHDL code, see section *The convertible subset* in Chapter *Conversion to Verilog and VHDL*.

`toVerilog` has the following attribute:

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

directory
This attribute is used to set the directory to which converted verilog files are written. By default, the current working directory is used.

timescale
This attribute is used to set the timescale in Verilog format. The assigned value should be a string. The default timescale is "1ns/10ps".

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 would be 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:

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

directory
This attribute is used to set the directory to which converted VHDL files are written. By default, the current working directory is used.

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.

library
This attribute can be used to set the library in the VHDL output file. The assigned value should be a string. The default library is `work`.

std_logic_ports
This boolean attribute can be used to have only `std_logic` type ports on the top-level interface (when `True`) instead of the default `signed/unsigned` types (when `False`, the default).

User-defined Verilog and VHDL code

User-defined code can be inserted in the Verilog or VHDL output through the use of function attributes. Suppose a function `<func>` defines a hardware module. User-defined code can be specified for the function 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.

These function attributes cannot be used with generator functions or decorated local functions, as these are not elaborated before simulation or conversion. In other words, they can only be used with functions that define structure.

Conversion output verification

MyHDL provides an interface to verify converted designs. This is used extensively in the package itself to verify the conversion functionality. This capability is exported by the package so that users can use it also.

Verification interface

All functions related to conversion verification are implemented in the *myhdl.conversion* package.

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

This function has the following attribute:

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

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

This function has the following attribute:

simulator
Used to set the name of the HDL simulator used to analyze the code. `"GHDL"` is the default.

HDL simulator registration

To use a HDL simulator to verify conversions, it needs to be registered first. This is needed once per simulator.

A number of HDL simulators are preregistered in the MyHDL distribution, as follows:

Identifier	Simulator
`"GHDL"`	The GHDL VHDL simulator
`"vsim"`	The ModelSim VHDL simulator
`"icarus"`	The Icarus Verilog simulator
`"cver"`	The cver Verilog simulator
`"vlog"`	The Modelsim VHDL simulator

Of course, a simulator has to be installed before it can be used.

If another simulator is required, it has to be registered by the user. This is done with the function `registerSimulation` that lives in the module `myhdl.conversion._verify`. The same module also has the registrations for the predefined simulators.

The verification functions work by comparing the HDL simulator output with the MyHDL simulator output. Therefore, they have to deal with the specific details of each HDL simulator output, which may be somewhat tricky. This is reflected in the interface of the `registerSimulation` function. As registration is rarely needed, this interface is not further described here.

Please refer to the source code in `myhdl.conversion._verify` to learn how registration works. If you need help, please contact the MyHDL community.

Python Module Index

m

Index

N

P

Q

R

S

T

U

V

W