Why You Should be Using Python/MyHDL as Your HDL

Christopher Felton
Design Engineer, @ Random Research

Introduction

Hardware Description Languages (HDLs) revolutionized the digital hardware design landscape when they were introduced 30 years ago. The majority of the complex digital hardware (IC and FPGA) –that has irreversibly changed our lives– was enabled by HDLs—mainly Verilog and VHDL. Although the mainstay HDLs have had much success, they haven't fundamentally changed since their inception. The defacto HDLs, Verilog and VHDL, have evolved over time, but this is good and bad. These languages have new features but some newer language constructs don't fit well with existing constructs—not a clean design. MyHDL strives to be an HDL based on proven concepts that can be powerful yet elegantly expressed (i.e. clean design) [1].

Figure 1 shows a high-level diagram of a typical MyHDL flow.

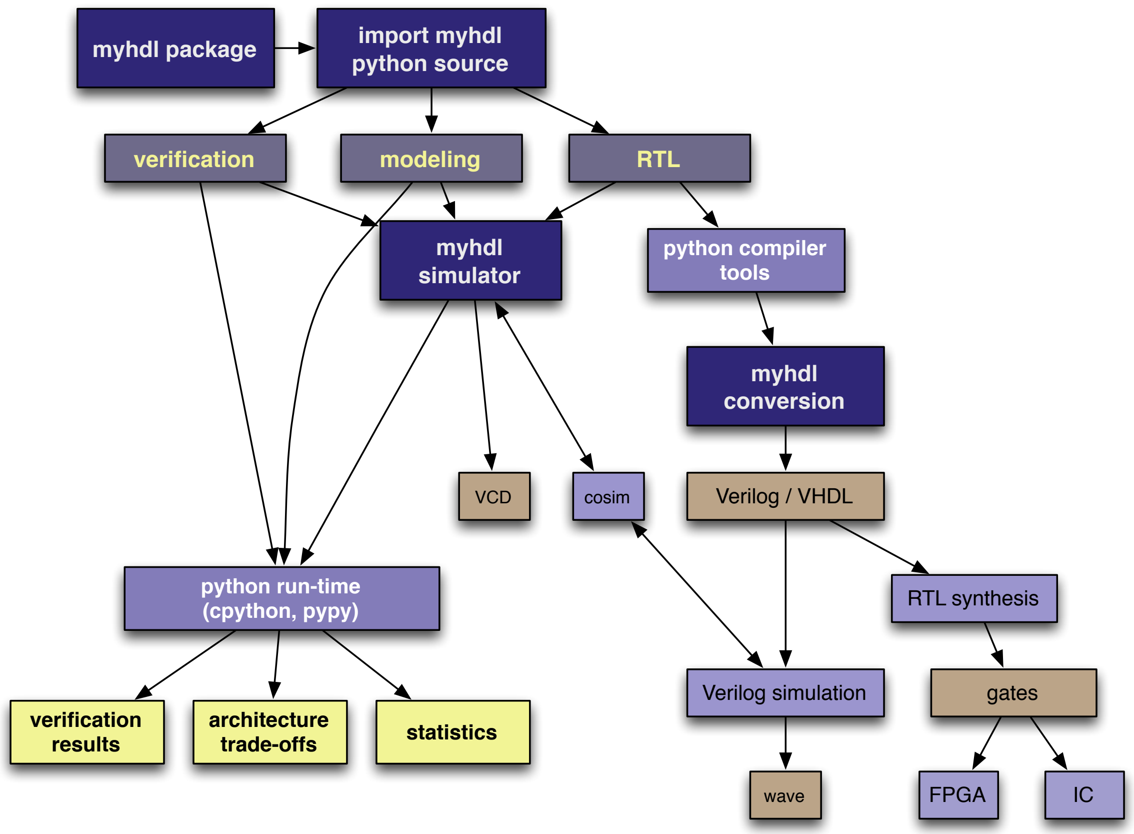


Figure 1: MyHDL Flow

What is MyHDL?

my·hdl Noun /ˈmī H-D-L/

1. A Python based hardware description language

MyHDL is a package in Python that uses existing Python language constructs to extend the Python language to support hardware description, modeling, verification, and conversion [1].

What is Python?

py·thon Noun /ˈpīˌTHän/ /ˈpīTHən/

1. A general-purpose high-level programming language whose design philosophy emphasizes code readability
2. The Pythons (also known as Monty Python) were a British comedy group that created the influential Monty Python's Flying Circus
3. A large heavy-bodied nonvenomous constrictor snake occurring throughout the Old World tropics

Python is a modern, high-level, dynamic, and general purpose programming language. The syntax is said to be clear and expressive.[1] Python has a large and comprehensive standard library, which has led to the statement "batteries included".

Python developers and users, for the most part, don't take themselves too seriously. The name for the programming language, Python, was inspired by the "Monty Python" comedy group. Python creator Guido van Rossum decided to name his language after the troupe. Even in the Python environment a user can experience the light-heartedness. Witticisms exist in the Python environment [2]. For example, simply start up the interactive Python environment and type:

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
…
```

This light-heartedness should not detract from Python's power and quality. For example, in the mid 1990s the first version of Google's crawler and indexer was written in Python (it was not Google then). Page and Brin were collaborating with a

[1] Although subjective, this is a resounding claim frequently encountered

programmer named Scott Hassan and Scott abandoned Java because it was too buggy at the time. Hassan transitioned to Python because it was his preferred language [3].

The principles inherent in Python: 'easy to learn', 'ugly code matters', and 'batteries included', are crucial for engineers and scientists. An engineer does not need to have invested years in a particular language or have a computer science degree to use Python. This opens up programming to a wider world of engineers with limited programming background, just as the Laplace transform allowed engineers to solve differential equations without investing years wading through the minutiae of differential calculus.

Python is often used as a scripting language and too frequently referred to as a scripting language. A script is commonly defined as a small piece of software that automates the execution of tasks—frequently a one-time use. Python can be employed in this manner because it doesn't require compilation, but Python is a full featured programming language with an impressive ecosystem. Python fully supports object-oriented programming, has support for functional programming (in the LISP tradition), and allows for structured and imperative programming. Python is a multi-paradigm programming language.

Register Transfer Abstraction Level

Abstractions for electronic circuits are commonly broken into three domains: physical/geometrical, structural, and behavioral [4]. This is further broken into different levels: Circuit, Logic, Register Transfer, Algorithmic, and System. The mainstream HDLs, Verilog and VHDL, operate at the Register Transfer Level (RTL) of abstraction. MyHDL operates at the Register Transfer Level of abstraction as well.

The Register Transfer Level taken literally, describes the operations on data when transferred from one register to the next. A system is represented at the RT-Level when specified by: 1) set of registers in the system, 2) operations performed on data in the registers, 3) the control that steers operations in the system [5]. MyHDL, Verilog, and VHDL actually support slightly higher abstractions levels than the basic register transfer but they are typically still considered RTL abstraction.

HDLs include semantics to capture register transfer operations as well as structural organization. HDL language constructs also exist to parameterize and configure process/always blocks.

Why Use MyHDL?

Python and MyHDL help a designer manage complexity far better than the alternatives. Although the implementer is describing the behavior at the RTL level, the other features of Python allow the developer to manage the complexity of the design. The engineer has a single language for algorithm exploration, modeling, designing, simulating, testing, and scripting [6]. Python shifts the focus from the programming language to the design. This is particularly useful for beginners, but even more important for experienced engineers. As designs increase in complexity Python and MyHDL scales to tackle the design.

Substantial thought and consideration have gone into the design of Python and MyHDL by their respective creators. The result is an elegant language and package to implement digital hardware description. MyHDL, like Python, is "batteries included". An engineer new to HDL design has a language, runtime, and simulator at their fingertips. A new user (or any user) can write their description, run simulations, verify, and review the results all in the same language and environment. This is demonstrated with the examples in this paper.

When using Python and MyHDL the developer leapfrogs existing HDLs. Python and its ecosystem have greater support than the traditional HDL languages. Existing HDLs have warts each new designer needs to work through. MyHDL takes the strengths of its predecessors and combines the best features of each to create a well-designed HDL, enabling engineers to be more productive and produce higher quality of results (QoR).

What MyHDL is Not

As discussed, MyHDL is many things and there are compelling reasons to use it. It is also worth reviewing what MyHDL is not. MyHDL is not a tool to take arbitrary Python code and create working hardware [7]. MyHDL is similar to existing HDLs; the convertible subset of the language describes hardware behavior at the Register Transfer Level (RTL) of abstraction. Clearly, this indicates MyHDL is not a High-Level Synthesis (HLS) language.

MyHDL is not a synthesis tool; it will not directly create netlists (structural descriptions) from the MyHDL sources. MyHDL needs to be converted to Verilog and/or VHDL, which then needs to be synthesized into an actual circuit realization [7]. The conversion of MyHDL to Verilog and VHDL is pragmatic; synthesis technology is a huge beast and can require tens of thousands of developer hours to get a high quality results. MyHDL, like VHDL and Verilog, is a hardware description language. MyHDL does not include "IP"[2] or cores directly [7].

MyHDL a Short History

Jan Decaluwe is the creator, implementer, and maintainer of MyHDL. Jan has twenty plus years experience with IC design, digital logic, and HDLs. Jan has brought a very well thought design and approach to the MyHDL hardware description language. Jan created MyHDL around 2002 and the first release was in 2003. The mailing list first message was in September 2003. Then in late 2004 Jan published the first article on MyHDL in the Linux Journal [8].

[2] IP refers to intellectual property, in HDL world a digital system is a collection of IP blocks or logic cores

Over the years MyHDL has grown and evolved. MyHDL, like Python, has a methodical design and release process. The MyHDL releases have very high quality, in similar vein to Python. MyHDL has been used in the design of seven ASICs (at last count) and numerous FPGA projects.

Jan created MyHDL to fill a gap in existing HDLs. He wanted an HDL that could elegantly be expressed, have a clean design, and be based on proven concepts. This includes the clear use of signals and variables [9]. MyHDL's use of signals and variables is clear and removes the confusion provided by current HDLs.

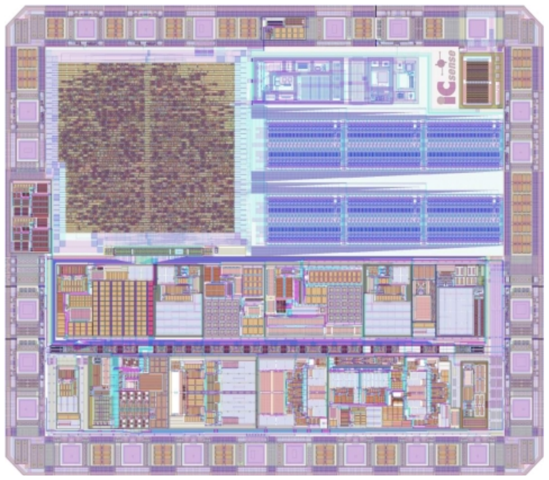


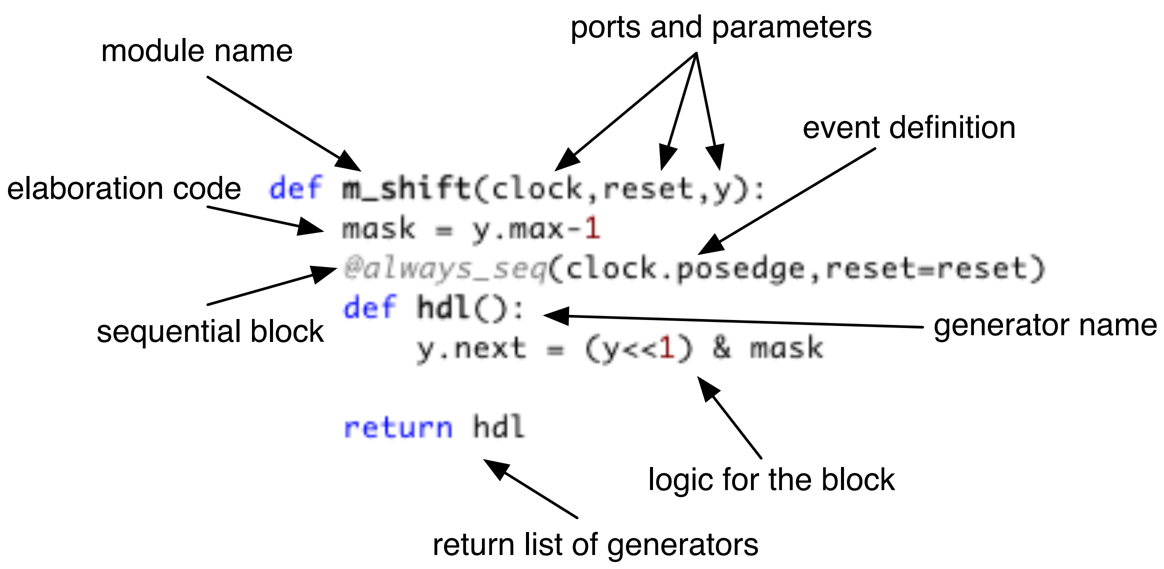
Figure 2: First MyHDL ASIC, image from www.jandecaluwe.com

The Python language was a perfect home to realize Jan's vision of an improved HDL. Python had the extensibility and meta-programming available to design the HDL package. Python was already known for its **not** ugly code. Implementing in Python removed the need to define a new language syntax and create new tools to interpret it.

The Anatomy of a MyHDL Module

MyHDL modules are similar to modules in Verilog or an entity/architecture in VHDL. The use of the word *module* is overloaded with Python's use of module. In the Python context *module* refers to a file that contains code and in MyHDL a *module* is a Python function that returns a generator and uses the MyHDL objects to describe hardware behavior. Figure 3 outlines the anatomy of a MyHDL module.

Generators in Python are resumable functions, i.e. functions that can maintain state from one call to another. Generators don't provide a return but `yield` the current execution (i.e., return from the function at the yield and return to this point when next is called on the generator).



```
def m_shift(clock,reset,y):
mask = y.max-1
@always_seq(clock.posedge,reset=reset)
def hdl():
    y.next = (y<<1) & mask

return hdl
```

Figure 3: MyHDL Module Anatomy

MyHDL adds types for the hardware description, notably the *Signal* and *intbv* types (objects). The *Signal* enables concurrent synchronization and communication between generators (process/behavioral blocks). The *intbv* is a range constrained integer to model the bit vectors in digital hardware. For a thorough introduction and definition of MyHDL refer to the MyHDL manual [10].

A MyHDL module is a Python function that contains a Python generator with a MyHDL decorator. The generators are the behavioral blocks and contain the circuit's description. The function arguments are the ports and parameters.

```
from myhdl import *

def m_shift(clock,reset,y):
    mask = y.max-1
    @always_seq(clock.posedge,reset=reset)
    def hdl():
        y.next = (y<<1) & mask

    return hdl
```

Code Listing 1: Shift code listing

The following code exercises the shift module. The module is fully parameterizable, it will be valid for any length of *y* greater than zero.

```
import argparse
from mysig import *

def run_shift(args):
    clock = Clock(0)
```

```python
    reset = Reset(0,active=0,async=True)
    y = Signal(intbv(1)[args.N:])

    tb_clk = clock.gen()
    tb_dut = m_shift(clock,reset,y)

    @instance
    def tb_stim():
        yield reset.pulse(10)
        yield clock.posedge
        for ii in range(args.N+2):
            print('%3d: %s'%(ii,bin(y).rjust(args.N,'0')))
            yield clock.posedge
        raise StopSimulation

    Simulation((tb_clk,tb_dut,tb_stim)).run()

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('N', type=int, default=8,
                        help="Number of bits in y")
    args = parser.parse_args()
    run_shift(args)
```

Code Listing 2: Shift Exercise Code

The following output is produced by the above exercise code.

```
In [1]: run m_shift.py 8
  0: 00000001
  1: 00000010
  2: 00000100
  3: 00001000
  4: 00010000
  5: 00100000
  6: 01000000
  7: 10000000
  8: 00000000
  9: 00000000

In [2]: run m_shift.py 27
  0: 000000000000000000000000001
  1: 000000000000000000000000010
  2: 000000000000000000000000100
  3: 000000000000000000000001000
  4: 000000000000000000000010000
  5: 000000000000000000000100000
  6: 000000000000000000001000000
  7: 000000000000000000010000000
  8: 000000000000000000100000000
  9: 000000000000000001000000000
 10: 000000000000000010000000000
 11: 000000000000000100000000000
 12: 000000000000001000000000000
```

```
13: 000000000000010000000000000
14: 000000000000100000000000000
15: 000000000001000000000000000
16: 000000000010000000000000000
17: 000000000100000000000000000
18: 000000001000000000000000000
19: 000000010000000000000000000
20: 000000100000000000000000000
21: 000001000000000000000000000
22: 000010000000000000000000000
23: 000100000000000000000000000
24: 001000000000000000000000000
25: 010000000000000000000000000
26: 100000000000000000000000000
27: 000000000000000000000000000
28: 000000000000000000000000000
```

It can be seen that the shift operation is indeed occurring.

A First Example (second?)

As a first example, we'll look at a common building block in typical designs— a counter that generates a periodic strobe.   This counter counts clock cycles; in register transfer speak, a counter increments once for every positive edge of the synchronous clock.

Before jumping to the counter description let's first describe a method to verify the counter. Many texts introduce simple examples but often fail to introduce how to verify the example.   Our module has a simple interface: an input clock, reset, and an output strobe.   The module also has a parameter: how frequent to strobe the output in milliseconds.

```python
def m_strober(clock, reset, strobe, ms=33):
```

To test the module it is first instantiated, a clock and reset is provided, and the strobe generation is verified as directed. The following Python code is used to test the *m_strober* module. Note that only the interface (port and parameter definitions) has been defined and not any internal logic, yet.

```python
from random import randint
from myhdl import *
from mysig import Clock,Reset

from strober import m_strober

def test_strober():
"""Test the m_strober module"""
    ms=randint(7,111)
    clock = Clock(0, frequency=1e3)
    reset = Reset(0, active=0, async=False)
    strobe = Signal(bool(0))

```

```
14      tb_clock = clock.gen()
15      tb_dut = traceSignals(m_strober, clock, reset, strobe, ms=ms)
16
17      @instance
18      def tb_stim():
19          yield reset.pulse(13)
20          for ii in range(113):
21              yield delay(clock.hticks*2*ms)
22              assert strobe == True
23
24          yield delay(23)
25          # stop simulation
26          raise StopSimulation
27
28      # run the verification
29      Simulation((tb_clock, tb_dut, tb_stim)).run()
30      # run the converters
31      toVerilog(m_strober, clock, reset, strobe)
32      toVHDL(m_strober, clock, reset, strobe)
```

Code Listing 3: Testbench for the strobe module

The first couple lines import the packages *random* and *myhdl*. The *random* package is used to generate random integers for testing. Line 5 imports the *m_strober* module. Line 9 generates a random value for this test. Lines 10-12 instantiate the signals used. Lines 14-15 instantiate the modules. Lines 17-26 define the test algorithm. Line 29 runs the verification simulation and lines 31 and 32 convert the module to Verilog and VHDL.

Before defining the logic, the outputs are set to a constant zero. We then verify that the test fails by executing the above test code. To assign the output to a constant value the following code is used:

```
from myhdl import *

def m_strober(clock, reset, strobe, ms=33):
    """Create a strobe every ms +/- a couple ticks"""
    @always_seq(clock.posedge, reset=reset)
    def hdl():
        strobe.next = False
    return hdl
```

Code Listing 4: Failing strobe logic

After the test is defined and fails, the module logic is implemented to make the test pass. The following is the MyHDL description for the strobe logic:

```
from myhdl import *

def m_strober(clock, reset, strobe, ms=333):
    """Create a strobe every “ms” millisecond +/- a couple ticks"""
```

```
5
6          max_cnt = int(round((clock.frequency/1000.)*ms))
7          cnt = Signal(intbv(1, min=0, max=max_cnt+1))
8
9          @always_seq(clock.posedge, reset=reset)
10         def hdl():
11             if cnt >= max_cnt:
12                 cnt.next = 1
13                 strobe.next = True
14             else:
15                 cnt.next = cnt + 1
16                 strobe.next = False
17
18         return hdl
```

Code Listing 5: Passing strobe logic

Line 3 is the Python function (MyHDL module) definition with ports and parameters. Line 4 is a document string for the module. Line 6 calculates the number of ticks required to meet the desired milliseconds based off the system clock frequency and stores this number in a local constant. Line 7 creates the local count (*cnt*) signal. Line 9 is the start of the behavioral block description. The *@always_seq* in line 9 indicates a sequential description synchronous to the *clock* positive edge and assigns the reset for the sequential block. Lines 11-16 define the logic and line 18 returns the generators in the module.

Figure 4 shows the waveforms from the verification simulation. It can be seen that the strobe (pulse) is created and the testbench verified the pulse was generated as expected.

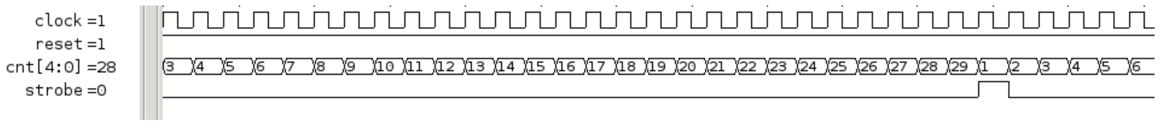


Figure 4: Strobe simulation waveform

Ecosystem

When a colleague first introduced to me to MyHDL via the Linux Journal article I was not impressed. And at the time I was not impressed with Python. Fortunately, my reluctance was solely based on the fact that I was implementing my first large HDL design and I was excited to use what I had recently learned, which wasn't MyHDL. This was unfortunate because I could have used the power of Python and MyHDL!

As the years passed, I was more open to MyHDL and eventually found a home in it. The catalyst for my migration was constant irritation with switching between Matlab and VHDL. I wanted a more integrated environment to explore signal-processing algorithms and associated digital circuit implementations. This environment for me was MyHDL.

Consider the following digital filter example. In it, we create a convertible and synthesizable description of an IIR filter and simulate the frequency response, all in the same environment.[3]

```
def m_iir_type1(clock,reset,x,y,ts,B=None,A=None):
    # make sure B and A are ints and make it a ROM (tuple of ints)
    b0,b1,b2 = map(int,B)
    a0,a1,a2 = map(int,A)

    ffd = [Signal(intbv(0, min=x.min, max=x.max)) for ii in (0,0)]
    fbd = [Signal(intbv(0, min=x.min, max=x.max)) for ii in (0,0)]
    # intermediate result, resize from here
    ysop = Signal(intbv(0, min=dmin, max=dmax))

    @always_seq(clock.posedge, reset=reset)
    def hdl():
        if ts:
            ffd[1].next = ffd[0]
            ffd[0].next = x

            fbd[1].next = fbd[0]
            fbd[0].next = ysop//Am  # truncate (>>)

        # extra pipeline on the output at clock
        ysop.next = (b0*x) + (b1*ffd[0]) + (b2*ffd[1]) - \
                    (a1*fbd[0]) - (a2*fbd[1])

        # truncate to the output word format
        y.next = ysop//Am  # truncate (>>)

    return hdl
```

Code Listing 6: Straight-forward IIR digital filter

Figure 5 shows the IIR simulation waveform. Figure 6 shows the frequency response of the digital IIR filter.

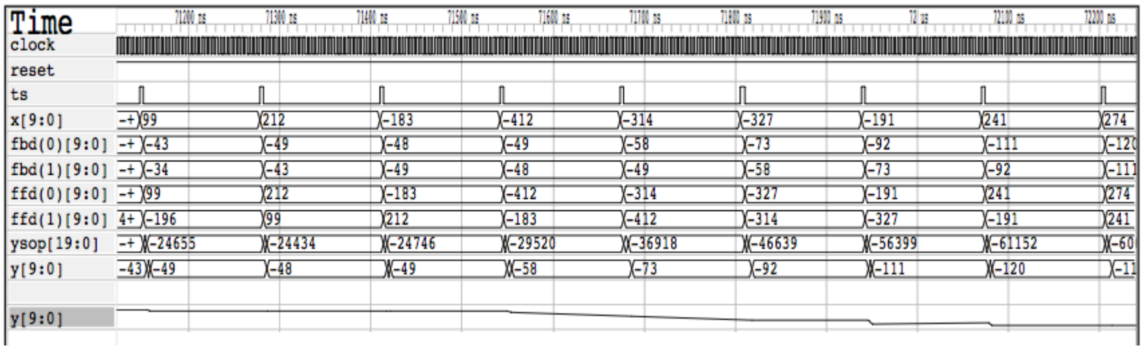


Figure 5: IIR simulation waveform

[3] The complete code example is available at: http://www.bitbucket.com/cfelton/dw2013_examples

In addition to the generating the digital waveforms from the MyHDL simulation, higher-level system analysis can be performed at the same time in the same environment. Figure 6 shows the frequency response of the digital filter. The digital filter simulation inputs random integers and captured the outputs. A frequency domain output over input is generated from the test code.

Regression tests can be defined to check the frequency response within some error bound. This can be useful as the implementation is modified and/or enhanced. The test will verify the incarnation of the digital filter's frequency response meets the requirements.

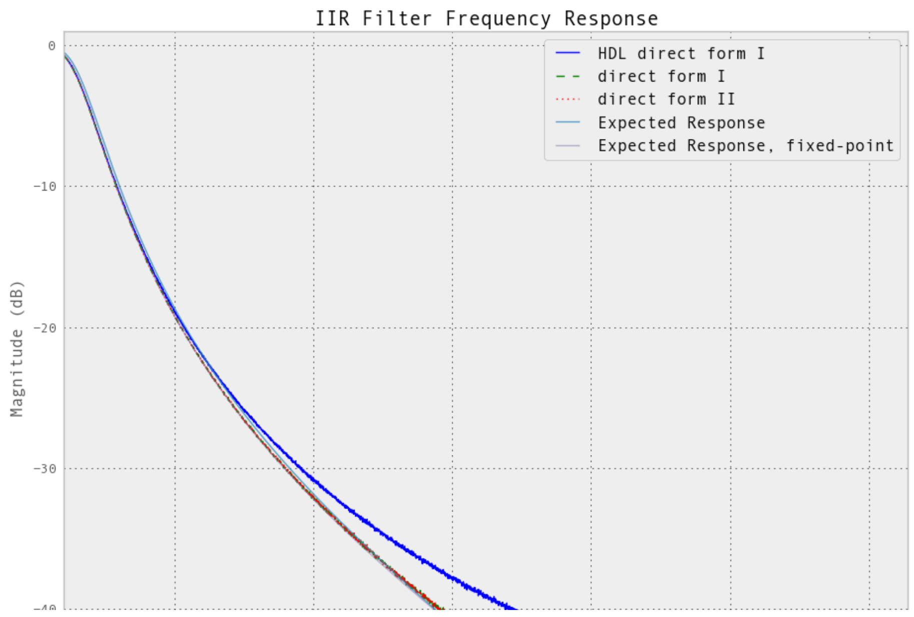


Figure 6: IIR digital filter frequency response

The HDL module can be embedded in a class. The following code listing is a usage example when the above IIR filter RTL is included in an object.[4] Embedding the RTL in a class allows a design to access different attributes of the design, as mentioned, the frequency response, fixed-point quantization analysis, etc.

```
# Instantiate the SIIR object.  Pass the cutoff frequency
# Fc and the sample rate Fs in Hz.  Also define the input
# and output fixed-point type.  W=(wl, iwl) where
# wl = word-length and iwl = integer word-length.  This
# example uses 23 fraction bits and 1 sign bit.
>>> from siir import SIIR
>>> flt = SIIR(Fstop=1333, Fs=48000, W=(24,0))
```

[4] http://www.dsprelated.com/showcode/211.php

```

# Plot the response of the fixed-point coefficients
>>> plot(flt.hz, 20*log10(flt.h))

# Create a testbench and run a simulation
# (get the simulated response)
>>> from myhdl import Simulation
>>> tb = flt.TestFreqResponse(Nloops=128, Nfft=1024)
>>> Simulation(tb).run()
>>> flt.PlotResponse()   # generates a plot like above

# Happy with results generate the Verilog and VHDL
>>> flt.Convert()
```

Combining the MyHDL RTL description with the flexibility of Python provides different opportunities to manage the ever-increasing design complexities.

Conclusion

As discussed, MyHDL adds a package to Python to enable hardware description at the register transfer level of abstraction. The Python ecosystem provides support that cannot be found in other HDL environments and the verification opportunities are boundless.

Python and MyHDL are currently being used to implement FPGA and ASIC designs. MyHDL provides a solid platform for architecture exploration, model design, RTL entry, and verification.

The author can be contacted at cfelton@ieee.org for questions or comments.

Acknowledgements

I would to thank Jan Decaluwe for the creation of MyHDL and the information generated in the MyHDL manual, MyHDL wiki, and the All Programmable Planet blogs. I would also like to thank Clive Maxfield, Tom Dillon (Dillon Engineering), Linda Cash, and Seth Benton for reviewing this paper.

References

[1] Decaluwe, Jan. http://www.myhdl.org/doc/current/manual/preface.html

[2] Warsaw, Barry. http://www.wefearchange.org/2010/06/import-this-and-zen-of-python.html

[3] Levy, Steven. *In the Plex*. New York: Simon & Schuster, 2011

[4] Decaluwe, Jan. http://www.myhdl.org/doku.php/why

[5] Decaluwe, Jan. http://www.myhdl.org/doku.php/whatitisnot

[6] Decaluwe, Jan. "MyHDL: a Python-Based Hardware Description Language". Linux Journal, 2004

[7] Decaluwe, Jan. "MyHDL: Why Do We Need Signal Assignments". All Programmable Planet, 2012

[8] Decaluwe, Jan. http://www.myhdl.org/doc/current/

[9] Ashenden, Peter. *The Designer's Guide to VHDL.* San Francisco: Morgan Kaufmann, 2002, 2nd Edition

[10] Mano, Morris and Ciletti, Michael. *Digital Design.* Upper Saddle River: Pearson Prentice Hall, 2007, 4th Edition

[11] Lyons, Richard. *Understanding Digital Signal Processing.* Boston: Pearson Education, 2011, 3rd Edition.