

## PROGRAMMABLE LOGIC IN PRACTICE

# Rapid FPGA Design in Python Using MyHDL

COLUMNS



MyHDL is an alternate hardware description language (HDL) that allows you to leverage the power of Python for designing, simulating, and verifying FPGA designs. Colin explains how MyHDL works and describes a FIR filter he created with C/C++ HLS in his February 2014 article to compare the toolchain flow.

*By Colin O'Flynn (Canada)*

**B**ack in February 2014, I took you through the use of C/C++ High Level Synthesis (HLS) as a design language for a FPGA. This article is designed to introduce you to another option for a design language, this time using Python. Once again I'll demonstrate that directly writing Verilog or VHDL is not always the most efficient option.

I'm going to follow the Finite Impulse Response (FIR) filter example from my February 2014 column, which allows you to directly compare the design process. One of the major advantages of using MyHDL compared to C/C++ HLS is that you can pull upon a huge library of existing Python modules to help generate and validate your design.

In the C/C++ HLS example, I used external tools to generate the FIR coefficients. In the MyHDL example, they are generated automatically from my filter specifications. This makes it easy to validate the fixed-point implementation, and compare the filter results to the "ideal" filter result. I'll get into more details later, but before that I want to present an overview of MyHDL.

I should also mention this column owes a great debt to Christopher Felton, who's presentation at DesignWest 2013 on MyHDL is what originally turned me on to the use of MyHDL. I've based the FIR filter example in this column on some of his examples. (For more of his examples, refer to [www.fpgarelated.com/blogs-1/nf/Christopher\\_Felton.php](http://www.fpgarelated.com/blogs-1/nf/Christopher_Felton.php).) I've also linked to his work from [ProgrammableLogicInPractice.com](http://ProgrammableLogicInPractice.com), which includes a few other sites besides [FPGArelated.com](http://FPGArelated.com).

## INTRODUCING MyHDL

Even if you haven't heard about MyHDL before, it's been in development for some time. It was created by Jan Decaluwe, and released to the world in September 2003. MyHDL allows you to use Python as a hardware description language (HDL). Like other high-level synthesis tools, you must remember it is not designed to convert arbitrary software code into FPGA modules. It won't make an FPGA designer out of a Python programmer, but might make a FPGA designer want to pick up Python for improved productivity.

If you are familiar with Python, you will know that it doesn't natively support all the features required in a HDL. But with a handful of extensions, we can emulate the required features such as ports, signals, and concurrent blocks. For synthesis MyHDL operates at the same Register Transfer Level (RTL) as Verilog or VHDL. This makes it easy to automatically convert from MyHDL to Verilog or VHDL. The resulting Verilog or VHDL files can either be synthesized directly by your FPGA toolchain, or integrated into your existing project (which will again be synthesized by your FPGA toolchain).

Let's jump right into a simple example.

**Listing 1** shows a simple implementation of a counter with programmable maximum. **Listing 2** shows the resulting Verilog code. The `ctr_hdl()` block is the main module. One of the first things to note is the module follows some Python-centric themes. For example, there is no explicit type (such as integer bit-width) in the module definition. Instead, the module is able to pull attributes such as the input/output port widths directly from the objects themselves.

The combinational logic (`@always_comb`) and sequential logic (`@always_seq`) blocks will be familiar to any FPGA designer. Like with Verilog or VHDL, a process sensitivity list can be used to determine when the blocks run. You will start to notice the simple use of class attributes, such as the rising edge being defined as part of the `Signal()` class from MyHDL. As well when dealing with the assignment of the future value of the signal once this block executes, we use the `.next` attribute instead of requiring a special operator (such as `<=` in Verilog).

This simple example also takes advantage of the use of the `ResetSignal()` object type. This special signal makes working with resets easier. Notice I never define the reset behavior in my MyHDL `@always_seq` block. Instead the reset signal will automatically reset any used signals to their "default" state (which was declared when I defined those signals). This helps make the code clearer. Often we don't need to see all the reset logic, but still want signals to start at a known value.

Of course, MyHDL doesn't force its reset handling down your throat. Another form of the sequential block allows you to explicitly define the reset behavior. This allows you to reset signals to other values or perform additional actions within the reset block.

So far, I've concentrated mostly on the synthesizable aspects of MyHDL. But much of the "more interesting" aspects of MyHDL are the ability to use it for both simulation and verification of your hardware cores. Whereas Verilog or VHDL have somewhat limited I/O facilities and external libraries, Python has

almost limitless potential when it comes to I/O facilities and external libraries.

In fact, MyHDL can even be used in combination with a Verilog simulator. This means you are not simulating the MyHDL code, but actually simulating the Verilog code generated by MyHDL. The advantage is that by using MyHDL (and Python), you are able to perform complex verification tasks with ease, while still validating your Verilog implementation.

MyHDL also makes problems such as conditional instantiation (selecting which version of a core to use) trivial. MyHDL passes instances of the HDL object, and doesn't

```
from myhdl import *

def ctr_hdl(clk,reset,prog_max,cnt):
    #Define local signal with sizes based on port
    intcnt = Signal(intbv(0,min=cnt.min, max=cnt.max))

    #Example combinational block
    @always_comb
    def copy_out():
        cnt.next = intcnt

    #Example sequential block - reset code generated
    #automatically
    @always_seq(clk.posedge, reset=reset)
    def cnt_main():
        if cnt < prog_max:
            intcnt.next = (intcnt + 1)
        else:
            intcnt.next = 0

    return instances()

##Example of instantiating module, here used just
##for Verilog conversion

#Simple boolean signal
clk = Signal(False)

#Reset signal gets special treatment, makes it easier
#to change reset parameters around
reset = ResetSignal(0, active=1, async=True)

#bit-vector types, specify default value along with min/max
prog_max = Signal(intbv(0,min=0, max=4000))
cnt = Signal(intbv(0,min=0, max=4000))

toVerilog(ctr_hdl, clk, reset, prog_max, cnt)
```

#### LISTING 1

A simple counter implemented in MyHDL. This code is sufficient to describe the counter and convert it to Verilog, the resulting Verilog being shown in Listing 2.

```

module cntr_hdl (
    clk,
    reset,
    prog_max,
    cnt
);

input clk;
input reset;
input [11:0] prog_max;
output [11:0] cnt;
wire [11:0] cnt;

reg [11:0] intcnt;

always @(posedge clk, posedge reset) begin:
    CNTR_HDL_CNT_MAIN
        if (reset == 1) begin
            intcnt <= 0;
        end
        else begin
            if ((cnt < prog_max)) begin
                intcnt <= (intcnt + 1);
            end
            else begin
                intcnt <= 0;
            end
        end
    end
end

assign cnt = intcnt;

endmodule

```

**LISTING 2**

The Verilog output of MyHDL for the input given in Listing 1. The direct conversion can easily be seen in this case, although MyHDL has handled some features for us such as resetting signals to default values that Verilog requires us to explicitly specify.

**ABOUT THE AUTHOR**

Colin O'Flynn (coflynn@newae.com) has been building and breaking electronic devices for many years. He is currently completing a PhD at Dalhousie University in Halifax, NS, Canada. His most recent work focuses on embedded security, but he still enjoys everything from FPGA development to hand-soldering prototype circuits. Some of his work is posted on his website at [www.colinoflynn.com](http://www.colinoflynn.com).

require you to define the entire port map as Verilog or VHDL would need.

While I don't have time to cover all these aspects, I want to talk you through at least a simple example of MyHDL simulation and implementation. To do this I'll be replicating the FIR filter from the February 2014 column.

**ANOTHER FIRRY EXAMPLE**

The FIR filter is not particularly exciting, but it does show off the use of MyHDL and Python to simplify your entire development. If you want to follow along, the easiest method is using a Python distribution such as WinPython on Windows. You can then install MyHDL using the pip tools, as described in the MyHDL documentation. This is all that is required to run the examples, which will be posted on [ProgrammableLogicInPractice.com](http://ProgrammableLogicInPractice.com) if you don't want to type everything in from the listing.

The MyHDL code for the FIR filter is shown in **Listing 3**, and the Verilog code generated by this is shown in **Listing 4**. Note the code in Listing 3 doesn't show the external interface or coefficient generation. I'll talk about that in a moment.

Comparing Listing 3 and Listing 4, you can note the similarity between the two code bases. One difference between the HLS C/C++ example from my previous column is that loop unrolling is not handled by MyHDL. Instead as MyHDL is operating at a similar level to Verilog or VHDL it relies on the synthesizer to perform the loop unrolling. Future version of MyHDL may support loop unrolling, but one could argue that perhaps this is not the job of the HDL, but instead the job of the hardware designer using the HDL.

Regardless of philosophical arguments, this does mean you are unable to automatically perform tasks such as tuning the trade-off between usage of hardware resources and throughput by asking the tools to unroll or not unroll a specific loop. The C/C++ HLS examples from my previous columns could be optimized for area or speed by a simple `#pragma` due to the support of C/C++ HLS to tune loop unrolling.

One thing I haven't explicitly mentioned until now is that MyHDL is entirely open-source (and free), whereas the C/C++ HLS has a \$2,000 yearly license fee and is proprietary. Thus, while I will compare the two for regular usage, it's worthwhile to also consider both the up-front cost, and the ability to modify the tools for your own use. MyHDL easily wins on both of those fronts!

But the real triumph of MyHDL can be seen once I introduce the complete simulation and generation environment. This is shown in **Listing 5**. The HDL code from Listing 3 is

not repeated, but you can consider the two listings are combined in the final program. In the C/C++ FIR example I required the use of external tools for filter design—with MyHDL, it's built right into the tools.

MyHDL is really just calling standard Python libraries, which have extensive tools for filter design. Thus, I could easily generate FIR or IIR filters of almost any order and type. The filter implementation itself is

fixed-point, and the Python code converts the floating-point types to the integer (fixed-point) notation in use. Full fixed-point support is still not present in the latest MyHDL release as of this column (0.8), but is on the roadmap for a future version.

Even without fixed-point support, the simulation environment of MyHDL pulls it ahead of C/C++ HLS. This makes it easy to verify correct operation of complex modules

### LISTING 3

The core of the FIR module in MyHDL is given here. Note this snippet requires instantiation to declare signal widths and the filter constants.

```
# Based on IIR Filter code, which is Copyright Christopher Felton
# and released under the LGPL license.

from myhdl import *

def sfir_hdl(
    # ~~ Ports ~~
    clk,          # Synchronous clock
    x,            # Input word, fixed-point format described by "W"
    y,            # Output word, fixed-point format described by "W"

    # ~~ Parameters ~~
    B=None,       # Numerator coefficients, in fixed-point specified
    W=(24,0)      # Fixed-point description, tuple,
                  # W[0] = word length (wl)
                  # W[1] = integer word length (iwl)
                  # fraction word length (fwl) = wl-iwl-1
):
    # Make sure all coefficients are int, the class wrapper handles all float to
    # fixed-point conversion.
    rB = [isinstance(B[ii], int) for ii in range(len(B))]
    assert False not in rB, "All B coefficients must be type int (fixed-point)"

    # We use a double-precision parameters as the result of the multiplication
    # will be 2x the input bit width. Define double width (precision ) max and min
    _max = 2**(2*W[0])
    _min = -1*_max

    Q = W[0]-1
    Qd = 2*W[0]

    # Delay elements, list of signals (double precision for all)
    ffd = [Signal(intbv(0, min=_min, max=_max)) for ii in range(len(B))]

    @always(clk.posedge)
    def rtl_fir():
        ffd[0].next = x
        for i in range(1, len(B)):
            ffd[i].next = ffd[i-1]

        yacc = 0
        for i in range(0, len(B)):
            b = B[i]
            yacc += b * ffd[i]

        # Double precision accumulator
        y.next = yacc >> Q

    return instances()
```

**LISTING 4**

The resulting FIR filter in Verilog, based on Listing 3. Again, note the fairly straightforward conversion from MyHDL to Verilog.

```

`timescale 1ns/10ps

module sfir_hdl (
    clk,
    x,
    y
);

input clk;
input signed [9:0] x;
output signed [9:0] y;
reg signed [9:0] y;

reg signed [20:0] ffd [0:19-1];

always @(posedge clk) begin: SFIR_HDL_RTL_FIR
    integer i;
    integer yacc;
    integer b;
    ffd[0] <= x;
    for (i=1; i<19; i=i+1) begin
        ffd[i] <= ffd[(i - 1)];
    end
    yacc = 0;
    for (i=0; i<19; i=i+1) begin
        case (i)
            0: b = 1;
            1: b = 2;
            2: b = 1;
            3: b = (-5);
            4: b = (-15);
            5: b = (-15);
            6: b = 13;
            7: b = 69;
            8: b = 128;
            9: b = 153;
            10: b = 128;
            11: b = 69;
            12: b = 13;
            13: b = (-15);
            14: b = (-15);
            15: b = (-5);
            16: b = 1;
            17: b = 2;
            default: b = 1;
        endcase
        yacc = yacc + (b * ffd[i]);
    end
    y <= $signed(yacc >>> 9);
end

endmodule

```

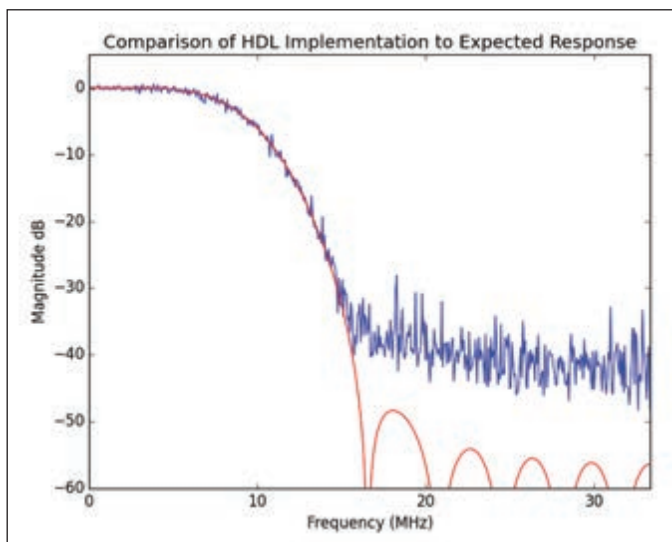
compared to C/C++ HLS, mostly as MyHDL is able to use the huge selection of Python modules to do everything from FFTs to graphing to I/O handling.

The simulation itself is performed in the `TestFreqResponse()` function. You will notice again the MyHDL-specific extensions

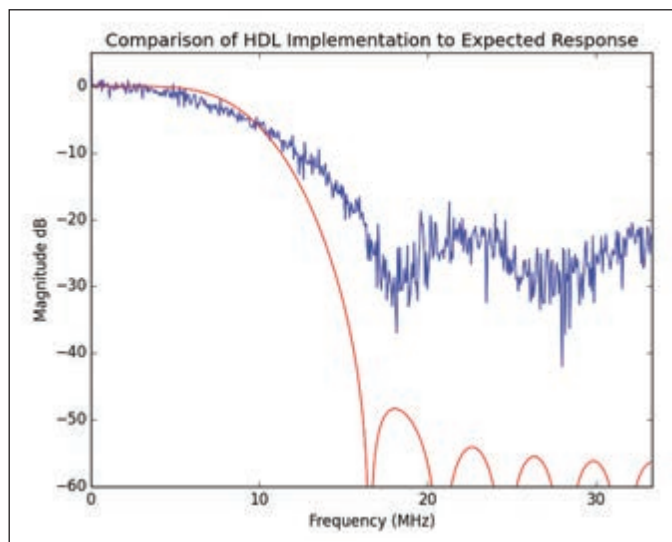
used here (such as the `@always` block to generate a clock signal). But we can use Python libraries as part of our test bench—appending data to lists or performing FFTs of data before saving.

In this example the function `PlotResponse()` generates the “expected”





**FIGURE 1** This shows the comparison of the expected FIR filter (in red) to the 10-bit fixed point implementation in blue. The fixed-point frequency response is obtained through a simulation in MyHDL.



**FIGURE 2** Compared to Figure 1, this shows what happens if we use only a 5-bit signal instead of a 10-bit signal. The loss of correct filter response can be seen by comparing the fixed-point response (in blue) to the expected response (in red).

frequency response of the filter based entirely on tried-and-true Python libraries, and compares it to our fixed-point results. The results of this are shown in **Figure 1**. Notice that the frequency response generally follows the expected response. This is using 10-bit inputs (the same as the ADC/DAC on my test board) and 20-bit intermediate values.

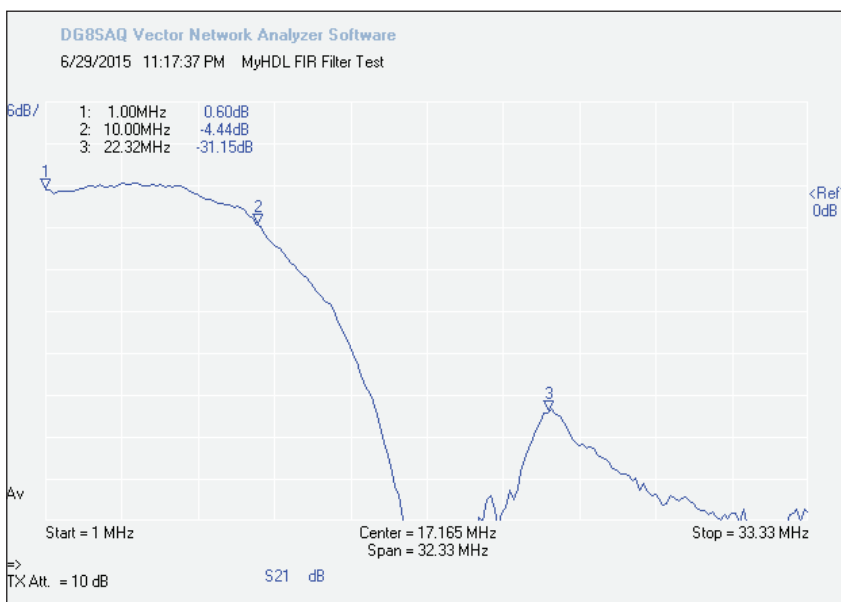
I could easily change the HDL to use 5-bit integers for the input values, which causes some additional divergence of my filter frequency response to the ideal filter. This frequency response of this fixed-point implementation is shown in **Figure 2**.

As a final test I've implemented the FIR filter in a Spartan 3 device, with an ADC and DAC running at 66.67 MHz. The FIR filter has been inserted between the ADC and DAC, and the frequency response is plotted in **Figure 3**. The analog path isn't perfect here which accounts for some of the errors, but you can see the response falls within "expected" bands compared to Figure 1.

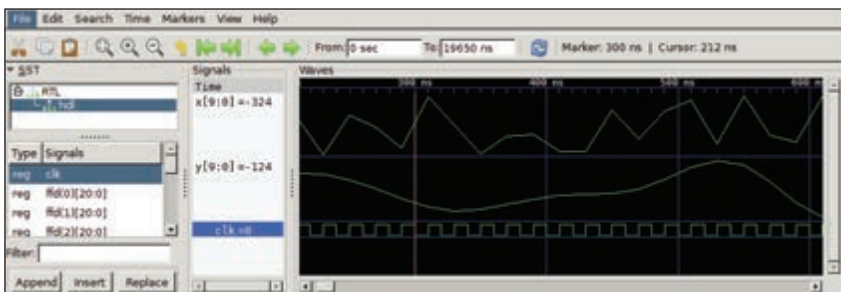
MyHDL made it trivial to entirely describe a filter which can be synthesized onto a FPGA. Unlike the C/C++ HLS example, I was able to use Python tools to include the entire filter design specifications into the source file.

### EVEN MORE THROWN IN

While this brief introduction to MyHDL won't do it full justice, there are a few more things worth mentioning. One thing I can't miss is highlighting the ability to perform unit testing in MyHDL. When designing hardware modules it can be a hassle to ensure you have tests for every module, and let alone attempting to script those tests to continuously run.



**FIGURE 3** The implemented FIR filter using 10-bit integer inputs is tested on a FPGA, where the input and output of the filter is an ADC and DAC respectively sampling at 66.67 MHz. This figure has numerous sources of error due to the introduction of an analog signal path, but it can be seen to generally match the expected FIR filter response.



**FIGURE 4** MyHDL also makes it easy to trace signal changes with time, by writing all signal changes to a VCD file. Here I'm inspecting the input and output of the filter for the frequency-bandwidth test used in generating Figures 1 and 2.

```

# Based on IIR Filter code, which is Copyright Christopher Felton
# and released under the LGPL license.

from myhdl import *
import numpy as np
from numpy import pi, log10
from numpy.fft import fft
from numpy.random import uniform
from scipy.signal import firwin, freqz
import pylab

class SFIR():
    def __init__(self,
                 Fc=10E6,      # cutoff frequency
                 Fs=66.66E6,   # sample rate
                 W=(24,0)     # Fixed-point to use
                 ):
        # The W format, intended to be (total bits, integer bits,
        # fractional bits) is not fully support.
        # Determine the max and min for the word-widths specified
        self.W = W
        self.max = int(2**(W[0]-1))
        self.min = int(-1*self.max)

        # Filter Design
        N = 19
        Wn = 0

        # Define the cutoff as a fraction of the nyquist
        Wn = float(Fc)/(Fs/2.0)
        self.b = firwin(N, Wn)

        # fixed-point Coefficients for the FIR filter
        self.fxb = np.round(self.b * self.max)/self.max

        # Create the integer (fixed-point) version
        self.fxb = tuple(map(int, self.fxb*self.max))

        print "FIR w,b", Wn, self.b
        print "FIR fixed-point b", self.fxb

    def Convert(self, W=None):
        clk = Signal(False)
        x = Signal(intbv(0,min=-2**(self.W[0]-1), max=2**(self.W[0]-1)))
        y = Signal(intbv(0,min=-2**(self.W[0]-1), max=2**(self.W[0]-1)))

        toVerilog(sfir_hdl, clk, x, y, B=self.fxb, W=self.W)

    def TestFreqResponse(self, Nloops=3, Nfft=1024):
        self.Nfft = Nfft
        Q = self.W[0]-1
        clk = Signal(False)
        x = Signal(intbv(0,min=-2**Q,max=2**Q))
        y = Signal(intbv(0,min=-2**Q,max=2**Q))
        xf = Signal(0.0)

        dut = traceSignals(self.RTL, clk, x, y)

        @always(delay(10))
        def clkgen():

```

```

        clk.next = not clk

    @always(clk.posedge)
    def ist():
        xi      = uniform(-1,1)
        x.next  = int(self.max*xi)
        xf.next  = xi

    @instance
    def stimulus():
        ysave    = np.zeros(Nfft)
        xsave    = np.zeros(Nfft)

        self.yfavg = np.zeros(Nfft)
        self.xfavg = np.zeros(Nfft)

        for ii in range(Nloops):
            for jj in range(Nfft):
                yield clk.posedge
                xsave[jj] = float(xf)
                ysave[jj] = float(y)/self.max

                self.yfavg = self.yfavg + (abs(fft(ysave, Nfft)) / Nfft)
                self.xfavg = self.xfavg + (abs(fft(xsave, Nfft)) / Nfft)

            raise StopSimulation

    return instances()

def RTL(self, clk, x, y):
    hdl = sfir_hdl(clk, x, y, B=self.fxb, W=self.W)
    return hdl

def PlotResponse(self):
    # Plot the designed filter response
    pylab.ioff()

    Fs = 66.66E6

    # plot the simulated response
    # -- Fixed Point Sim --
    xa = (2*pi * np.arange(self.Nfft)/self.Nfft) / (2*pi) * Fs
    H = self.yfavg / self.xfavg
    pylab.plot(xa, 20*log10(H), 'b' )

    w, h = freqz(self.b)
    # pylab.hold(True)
    pylab.plot((w/(2*pi))*Fs, 20 * np.log10(abs(h)), 'r')

    pylab.ylabel('Magnitude dB');
    pylab.xlabel('Frequency (MHz)')
    pylab.axis([0, Fs/2, -60, 5])
    pylab.xticks([0,10E6,20E6,30E6], ['0', '10', '20', '30'])
    pylab.title('Comparison of HDL Implementation to Expected Response')

```



```

pylab.savefig("firtest.png")

if __name__ == '__main__':
    # Instantiate the filter and define the Signal
    W = (10,0)
    flt = SFIR(W=W)

    flt.Convert()

    tb = flt.TestFreqResponse(Nloops=3, Nfft=1024)
    sim = Simulation(tb)
    print "Run Simulation"
    sim.run()
    print "Plot Response"
    flt.PlotResponse()

```

**LISTING 5**

The real power of MyHDL occurs once we start building systems around our cores. Here I'm using the SciPy library to automatically generate FIR filter coefficients, given the sampling frequency and desired cut-off. I can also compare the output of the fixed-point filter implementation to an idea FIR filter for the selected fixed-point bit width.


Once again MyHDL pulls on existing work in Python to simplify our test cases. It uses the `unittest` module from Python to allow you to easily generate test cases. Such test cases can validate a range of inputs—including testing options such as various bit widths to your module. These tests can be strung together with regular Python code, a task it excels at.

When it comes to debugging or documenting the code, MyHDL can automatically trace into a module and save signal waveforms to a `.vcd` file. Such a file can be opened by a universal viewer, such as `gtkwave`, which I show plotting the input and output of my FIR filter in **Figure 4**. The trace statement itself can be seen in **Listing 5**, as the call to `traceSignals()`. This was all done without any additional Verilog simulator, but as part of the regular MyHDL development process.

**MYHDL SUPERHERO**

MyHDL presents a number of credible reasons it can be taken seriously as a hardware description language (HDL). Most critically, it doesn't try to be "too clever", but instead inserts itself at about the same level as your existing Verilog or VHDL code. But by using the power of Python, MyHDL greatly simplifies aspects such as simulation and unit testing of your design, while improving many aspects that affect your synthesizable module such as clarifying reset signal handling and improving parametrized port definitions.

If you want to learn more about MyHDL, I'll have some examples (such as the FIR filter in this column) and links to other resources at the `ProgrammableLogicInPractice.com` website. But there is extensive documentation online at the MyHDL project webpage (`MyHDL.org`), which also includes a few examples. Christopher Felton has a number of additional well-documented examples such as FFTs, IIR filters, and more. You'll find further examples on various webpages such as everything from simple counters to Kalman filters implemented in MyHDL.

Considering MyHDL is free and easily available, there's nothing to stop you from giving it a spin. I think you'll find it has the right combination of familiar constructs that get you up to speed quickly with the new language, but adds enough new functionality to improve your overall productivity and enjoyment of FPGA development. Have fun! 

MEASUREMENT COMPUTING

## MCC Continues to Lower the Cost of DAQ

- Easy to Use
- Easy to Integrate
- Easy to Support



### USB-230 Series From \$249

- 8 SE/4 DIFF analog inputs
- 16-bit resolution
- Up to 100 kS/s sample rate
- 8 digital I/O
- One 32-bit counter
- Two analog outputs
- Included software and drivers



OEM board-only versions are also available

[mccdaq.com/USB-230-Series](http://mccdaq.com/USB-230-Series)

**MEASUREMENT  
COMPUTING™**

Contact us

**1.800.234.4232**

©2015 Measurement Computing Corporation  
info@mccdaq.com