Accelerating Algorithm Implementation in FPGA/ASIC Using Python

Tom Dillon, Jeremy Paatela, Guenter Dannoritzer, Scott Hussong

[tdillon, jpaatela, guenter, shussong]@DillonEng.com

Dillon Engineering, Inc.

Introduction

There are many white spots on the map when looking at the road from algorithm to ASIC/FPGA implementation. Venerable tools like MATLAB cover the algorithm development side. On the logic side, many vendors offer capable simulators and synthesis tools. However, after an algorithm has been developed, there are still some steps needed to arrive at the final implementation in a Hardware Description Language (HDL) that can be verified for proper functionality. New tools and languages constantly emerge to nibble at pieces or promise to deliver everything, but there is still no clear one-size-fits-all vendor solution to pave the way from algorithm to logic.

At Dillon Engineering Inc. we have been using the Python scripting language for many years now to fill some of the white spots in the development cycle. We are essentially hardware engineers always looking for ways to speed our development cycles. Python has become a foundation of all aspects of our logic development flow, allowing us to become a world-wide HPEC FPGA/ASIC leader for our clients, with increasingly shorter schedules.

Figure 1 shows a typical design flow for the implementation of an algorithm, starting with the specification or an actual implementation of the algorithm in floating point representation. If the HDL implementation is meant for fixed point representation, the next step is to create an equivalent model with fixed point. That model will then be part of the verification environment, but also needs to be refined to a certain extent so that the logic developers can start their job.

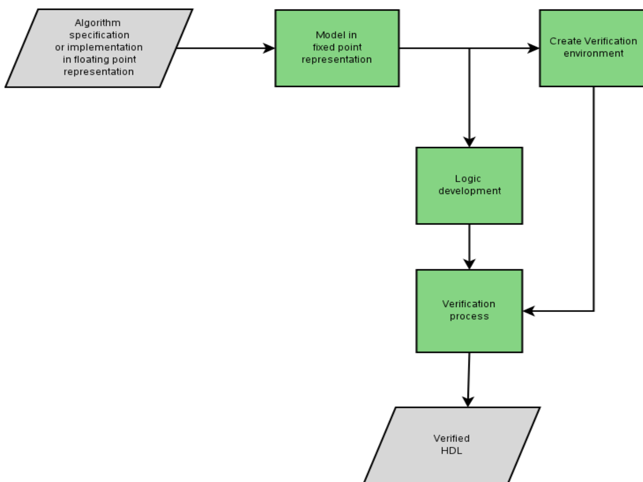

Figure 1: Development Flow

What can be seen from this flow chart is that the model is a critical point in the development process. Reducing its development time will reduce the overall development time. The more accurate the model, the less risk there will be in the logic implementation. To accomplish this, we use Python to model the algorithm and other Python-based helpers to improve efficiency in the rest of the development steps.

The remaining document will be divided in chapters with a short introduction to Python, modeling bit true logic, building logic, and finally using Python to drive the verification process.

Brief Introduction to Python

Python is an open-source design language, encompassing a lightweight scripting style liked by hardware engineers, but with extensions that include powerful array and scientific processing via NumPy / SciPy. Its concise, explicit format is easy to code, read, and maintain, resulting in excellent software quality. Object-oriented programming aspects promote powerful class creation and code reuse. Python is available for all major computer platforms and operating systems.

Based upon our experience, an engineer of any discipline can do more in less time with Python than with any other language. That is a bold statement, one that we believe is supported by our success implementing very complex HPEC algorithms.

HPEC Algorithm Model

Python shines for HPEC algorithm modeling, with math capabilities on par with MATLAB and all the features of a modern general purpose object oriented scripting language at your fingertips.

Algorithm Development with Python

To effectively implement an HPEC algorithm in logic, it first must be modeled in a format that permits closer comparison to HDL building blocks, yet retains the high-level algorithmic structure. We will typically start with development of a Python model using regular floating point numbers.

Python has a number of useful built-in features for algorithm development:

- full array support
- linear algebra
- built in complex number support
- array plotting
- canned FFT/IFFT and other math functions
- easy integration of control and math functions
- ease of model re-use via modules
- object oriented programming for the non-programmer

Fixed vs. Float

Most HPEC algorithms will require some or all fixed point math conversion to reduce size and power in an ASIC or FPGA implementation to make the design resulting product viable.

Using fixed point data (enabled for example by DeFixedInt class, available via download at www.DillonEng.com), trade offs between numeric performance and fixed point bit widths are done quickly and early in the development process. Bit accurate results are possible at this stage of the development, before a single line of HDL has been written.

With minimal changes to the floating point model, a fixed point bit accurate model is created. In most cases, the algorithm portion is common between fixed and floating point, with the only different the array types passed and returned.

Fixed Point Model

Once the fixed point model is functional, a final pass can be taken to test different bit widths and rounding schemes at various places in the algorithm. Again this is done before a single line of HDL is written and the resulting model will exactly predict the HDL representation that is eventually implemented in logic.

Building Logic with Python

We have also used Python to enable efficient logic generation, test-benching, and areas where scripting increases productivity.

ParaCore

Dillon Engineering's IP development suite is completely written in Python. More information on this tool is available on our website (www.DillonEng.com).

MyHDL

MyHDL is a Hardware Description Language using Python to describe the logic. Logic can be designed and verified in Python. MyHDL also has automatic conversion to Verilog or VHDL.

Scripting

There are always many repetitious tasks in any logic build flow:

- logic generation
- synthesis
- place and route

All of these tasks can be simply automated using Python scripts. Automation ensures proper builds and alleviates wasted effort of troubleshooting mistakes in the build process.

Of course other scripting languages exist, but writing these scripts in Python keeps us from having to master other scripting languages, increasing our productivity.

A good example of a time saving Python script is gen_ise_sh.py (available at www.DillonEng.com). With a few simple parameters, it will build all structures needed to synthesize and place/route a design with Xilinx ISE.

Verification with Python

Figure 2 shows a typical verification setup for an algebraic logic implementation. A test data generator generates some stimulus data that are fed through the model to get the expected data out. The generated data are also fed through the HDL test bench into the device under test. In the test bench there is a checker that uses output from the device under test and compares it with the expected output from the model.

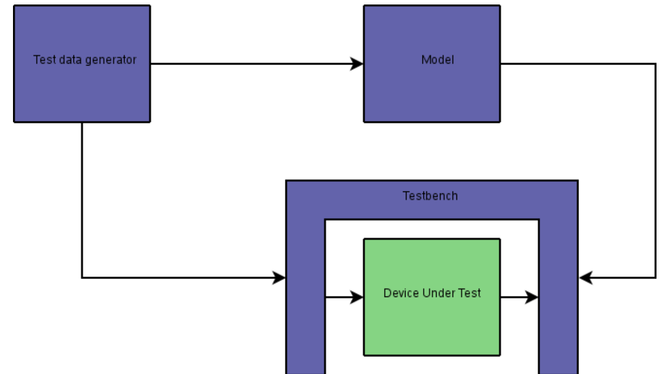

Figure 2: Verification

Verifying an algorithm implementation requires stimulating the logic with data that is generated by math functions. Here Python comes in handy again with its built in math module, allowing signal generation with random or trigonometric nature. Another benefit of building the model in Python is now we can feed the data right through it without the need for file I/O.

The remaining question is how to get the generated data to the test bench, as here usually a commercial simulator is used, supporting only HDL. Our approach is to use MyHDL, a Python module that allows Python/Verilog co-simulation through the Verilog PLI. We build the test bench in Python, and in connection with another Python base module, we can create assertion based test cases. This allows us to feed data through a Verilog device under test right from our Python code. The output from the logic is verified with the output from the model and the assertion will throw an exception and cancel the simulation if there is a data mismatch.

Conclusion

As demonstrated by the efficiency at which Dillon Engineering implements HPEC algorithms in ASICs and FPGAs, Python can be used in many ways to improve productivity and reliability of the logic design flow. The Python language scales nicely from scripting up to complex algorithm development, allowing engineers the opportunity to master only one language to greatly increase their everyday productivity.

Reference

[1] http://www.python.org – Python web page

[2] http://www.scipy.org – SciPy web page

[3] http://www.numpy.org – NumPy web page

[4] http://myhdl.jandecaluwe.com – MyHDL web page

[5] http://matplotlib.sourceforge.net – matplotlib web page

[6] http://www.DillonEng.com – Dillon Engineer web page