PYTHON AS A HARDWARE DESCRIPTION LANGUAGE: A CASE STUDY

J.I. Villar, J. Juan, M.J. Bellido, J. Viejo, D. Guerrero

ID2 Group / Department of Electronic Technology
University of Seville
E.T.S. de Ingeniería Informática
email: {jose, jjchico, bellido, julian, guerre}@dte.us.es

J. Decaluwe

Resources bvba
Leuven, Belgium
email: jan@jandecaluwe.com

ABSTRACT

Many people may see the development of software and hardware like different disciplines. However, there are great similarities between them that have been shown due to the appearance of extensions for general purpose programming languages for its use as hardware description languages. In this contribution, the approach proposed by the MyHDL package to use Python as an HDL is analyzed by making a comparative study. This study is based on the independent application of Verilog and Python based flows to the development of a real peripheral. The use of MyHDL has revealed to be a powerful and promising tool, not only because of the surprising results, but also because it opens new horizons towards the development of new techniques for modeling and verification, using the full power of one of the most versatile programming languages nowadays.

1. INTRODUCTION

The design of digital electronic systems, since its inception, was marked by a parallel and steady increase of both the complexity of tackled designs and the performance and integration level of implementation technologies.

For decades, this fact showed that low level design techniques would not be viable in the long term and therefore the development and adoption of new and efficient methodologies at a higher abstraction level would be required. These techniques should allow designers to tackle the increasing complexity of the modeling, testing and implementation tasks. In response to these needs new modeling languages [1] emerged inspired by software programming languages. These languages, called HDLs (hardware description languages), can be seen as programming languages with special abilities to describe the concurrent nature of the digital logic and electronics. HDLs model the structure and behaviour of hardware on the dimensions of time and space.

HDL descriptions are used to generate and specify the behaviour and structure of hardware. These specifications have a dual purpose: simulation of the description's behaviour and synthesis of lower level descriptions such as layouts or bitfiles, that can be finally implemented as physical devices. When used for simulation, HDL compilers generate pieces of ”executable code” that together with a simulation framework are able to emulate the behaviour of the described design so that designers can verify its correctness. These pieces of ”executable code” are the main responsibles for the fact that HDLs can be seen as a special kind of programming language. On the other hand, synthesys tools take HDL descriptions to infer structures that can be implemented in real hardware, thereby generating lower-level descriptions so that after a series of steps, a design that can be implemented on a physical device is reached. The fact that not all elements of an HDL can be extrapolated to hardware, makes HDLs have a greater expressive power for simulation than for synthesis, so it is necessary to define a synthesizable subset clearly stated [2].

In practice, there are two main standard HDLs: VHDL [3] and Verilog [4]. Nowadays their ubiquitous support from manufacturers and the lack of final arguments on each other, make both constitute the bridge that any design must pass to reach the synthesis and implementation technology, regardless of which was its original HDL. This means that any programming language that would have capacity of HDL, just being able to generate Verilog or VHDL code from its synthesizable subset could be abstracted from the lower-level stages and tools.

Moreover, both Verilog and VHDL simulators implement procedural interfaces (VPI [4] in Verilog and VHPI [5] in VHDL), through which, external software processes, developed in any programming language that have the required bindings, could connect to the simulator and govern the behavior of some signals according to the observed state. This is the fact on which underlies the ability to simulate the behavior of a hardware system using only software routines. They could be implemented in any programming language and would be connected to the simulator through VPI or VHPI.

Based on the above two points, it seems feasible to transform a very high level software programming language in an HDL [6] [7] [8] following a similar scheme to that shown in

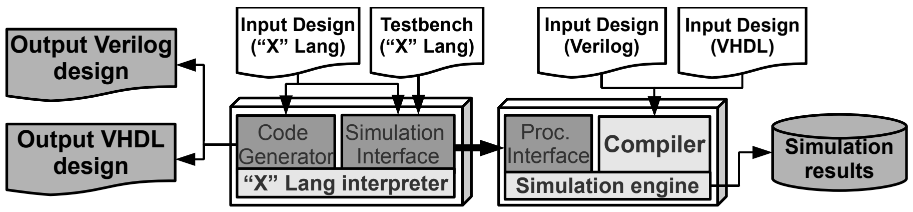

Fig. 1. Proposed adaptation scheme to use a generic interpreted programming language as an HDL

figure 1. This way it would be valid for both, physical implementation and simulation. Synthesis and implementation would use VHDL or Verilog as an intermediate language independent of the underlying technology. On the other hand, in order to use the programming language under adaptation as an HVL (Hardware Verification Language), it should be able to perform cosimulation together with other modules written in VHDL and Verilog, providing all its expressive power: high-level constructs, object orientation, third party libraries, etc...

This article presents a particular application of the Python language [9], based on the MyHDL package, [8] to make it a viable alternative for hardware development. To characterize this feasibility a study comparing the several aspects has been performed by designing a real peripheral using independently Verilog and MyHDL flows. Thus, it aims to reach a gut feeling about the power of Python as a hardware modeling tool as well as being able to evaluate its expressive power and the quality of automatically generated designs. We should note that, although in some aspects there may be similarities between MyHDL and other solutions such as System C or System Verilog, there is one feature which differentiates them unequivocally: whereas these other solutions are languages whose specific purpose is the description of hardware at system level, Python is a language whose development has been totally oblivious to the specific requirements of HDLs and that through an external library/framework has been given the ability to model hardware designs at RTL level.

This article is organized as follows: in section two the operation of MyHDL is analyzed highlighting its main features, advantages and disadvantages. Section three presents the design of the peripherals on which the comparison has been made. Section four details the comparison methodology used and the parameters on which both solutions were compared. Section five presents and analyzes the obtained results. Finally section six presents the most relevant conclusions derived from this experience.

2. PYTHON AS AN HDL: MYHDL

The development of HDLs based on general-purpose programming languages involves a series of design decisions concerning the way in which the differentiating characteristics of hardware, such as concurrency between operations will be modeled. In this regard, the particular characteristics of the programming language under study play a key role. They will have a great impact on such important issues as the different modeling techniques or code generation. Another key aspect of the language to use, is whether it is compiled or interpreted. Interpreted languages provide great flexibility due to their ability to self-analysis and runtime modification using introspection techniques [10]. These techniques, as discussed in the following sections, will be useful for this purpose. The Python language is, by his own philosophy, in a unique position to be used as an HDL. Throughout its development it has evolved in response to major design philosophies, becoming the language in which different paradigms: imperative, functional, object oriented, etc... are reconciled to thus have one of the most eclectic developer communities nowadays.

Since this versatility and ability to cope with almost any design philosophy is the distinguishing mark of Python, it is not surprising that it has finally approached to hardware design. A set of libraries called MyHDL have made this possible. The purpose of MyHDL, as defined by its creators is "to provide hardware designers with the simplicity and elegance of the Python language" [8]. To map the elements of a hardware design flow, we will discuss separately the three main issues: the Python hardware modeling techniques; simulation and verification; and finally the generation of VHDL or Verilog code using introspective techniques.

2.1. Hardware Modeling

The main idea behind MyHDL is the use of generators [11] and decorators [12] [13] to model concurrency. Generators are a special type of function. Their main difference with common functions is that generators remember the execution point at which they return to continue at the same point on subsequent calls.

Hardware modules are modeled as Python functions with wrapping decorators that return sets of generators. Thus, the semantics and structure of Python functions are used to support features such as arbitrary hierarchies of components, named port association, etc... Besides the above, MyHDL defines new classes to model hardware

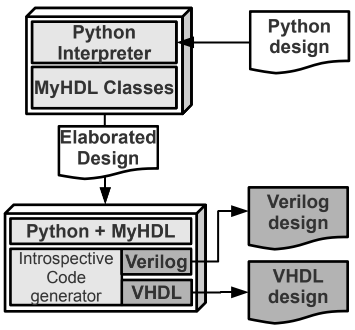

Fig. 2. VHDL and Verilog code generation scheme used by MyHDL

elements that can not be found in the core of Python such as signals (which are used to establish communication between generators), enumerated types with configurable encoding (for generating sets of states for FSMs) or new classes to work with signed or unsigned numbers at bit level transparently.

2.2. Simulation and Verification

While Python can be a useful tool for implementation, simulation and verification are the areas where it provides a further increase in benefits over VHDL or Verilog.

In the traditional hardware development cycle, the costs associated with testing and verification tasks vary between 30% and 80% [14], this means that improvements in this area have a great impact on the total cost of a project. Python, being a general purpose programming language especially designed for rapid application development [15], provides all its power and efficiency to make the testing phase a economic task in time and cost.

Moreover the simulation does not have any restriction inherent to the synthesizable subset, so an overwhelming amount of Python functions and extensions can be used. This opens the door to new hardware testing methodologies inspired in software engineering [16], such as unit testing; testsbenchs with the ability to interact with network elements; generation and analysis of data by using packages for scientific computation as SciPy [17], etc ...

MyHDL implements an embedded simulator that can generate visualizations of waveforms using the VCD standard [4]. It can also be used as a HVL [8] (hardware verification language) along with external simulators to verify designs through the use of its cosimulation features.

2.3. Verilog and VHDL Code Generation

As in other hardware description languages, the generation of synthesizable code in MyHDL has several limitations. Since the converter generates Verilog or VHDL code, the limit of the expressive power of MyHDL is limited to the elements that are common to the synthesizable subset of both languages. However, this expressive power is sufficient to define a convertible subset considerably more extensive than the standard synthesizable subset. This is possible because a design elaboration process is done prior to the code conversion stage. In this elaboration phase, the converter automatically performs tasks such as signal direction analysis (in, out or inout), transparent handling of signed and unsigned arithmetic or size and type inference among others.

The converter operation follows the scheme depicted in figure 2. In a first step it performs an elaboration of the design, where no convertibility limitations are applied, to obtain a hierarchy of Python generators. In a later stage, the code generator will analyze this hierarchy using the Python introspection api and will translate it into VHDL or Verilog. Thus the limitations of the synthesizable subset are solely applied to the internal parts of the generators allowing the full power of Python outside of them. This approach to code generation enables hardware modeling using elements such as lists of instances, conditional instantiation, etc ...

3. DESIGN UNDER STUDY

To make a comparison of two design flows with significative results, the development on which the two flows will be contrasted should be representative of most hardware developments and therefore it should include the most common elements: combinational and sequential blocks, memory elements, finite state machines, etc...

To choose the design on which the comparison was carried out, several applications have been assessed in different ranges of complexity. One of the most attractive designs for this purpose was a driver for LCD character displays compatible with the Sitronix ST7066U [18], Samsung S6A0069X or KS0066U, Hitachi HD44780 and SMOS SED1278. This controller is ideal because there are two communication modes between the controller and the chip: one based on eight data lines and a more complex one based only on four. The four-line communication protocol has been chosen to raise the complexity of the design and justify the inclusion of nested FSMs. Moreover, these chips require a pre-initialization and configuration stage that has also been implemented as another nested FSM. Thus, the core of the design consists of a main state machine that relies on two inner FSMs: one for managing the data transmission protocol and another one for the initialization process.

To expand the focus of the comparison, two solutions have been implemented at the protocol level, and two more at the storage level. Respecting to the protocol, we have implemented two approaches: one connecting the controller directly to a memory bus and one wrapping the controller to provide an interface compliant with the Wishbone standard [19]. Regarding the character storage we have also implemented two approaches: one storing them in an internal RAM so that transactions take a single clock cycle, and one storing them in the external memory of the LCD chip, that has a latency of the order of microseconds. These two storage methods, which we will call RAM and RAMless respectively, in combination with the two communication protocols result in a set of four different designs on which we will compare results.

4. DEVELOPMENT AND COMPARISON METHODOLOGY

To perform the comparison, we have previously established a methodology for both, the development of the designs to compare and for the extraction of the parameters to be measured.

The specifications of the lcd driver chip have been taken as the starting point to develope a first functional design, coded in Verilog and using the internal RAM storage approach without any kind of optimization. This design has been iteratively optimized till a point of quiescence where if no protocol changes were added we could not reach any further improvements. This optimized Verilog design was used to get an equivalent Python design by making a direct translation. From this point we have worked with both flows independently. Using the Python design, an iterative optimization process was started again, as it was previosusly done with the Verilog core, until we could not achieve any futher improvements.

After obtaining optimum designs in Verilog and Python for the cores using FPGA integrated RAM, the necessary modifications were made to store characters on the external LCD display memory, thus eliminating the need for memory inside our designs. With this RAMless approach, a new independent optimization process for each design was started to reach a new stalemate point in the parameters measured.

Another studied aspect was the impact of adding a wrapping layer to modify the core interface. A wrapper was developed to implement a Wishbone interface in both, Verilog and Python. All measures have been taken using both the core controller with the memory mapped interface and with the Wishbone interface. The main aim is to be able to evaluate how wrapping layers affect the final design.

For each design, functional equivalence with respect to the rest of designs has been checked by simulating and contrasting the generated VCD waveforms.

Within the three dimensions in which designs are classically compared: occupation, speed and power consumption, we have focused mainly on the resource occupation as an indirect measure of the expressive power of Python as an HDL since performance and maximum frequency achievable met requirements by a safe margin and had almost no diferences on both development flows. XST has been used as synthesis tool with enabled area optimization directives and effort level two. Designs have been generated for a Spartan 3E 500 FPGA. From each of the tests we have measured the usage of: LUTs, FFs and BRAMs.

This methodology is intended to compare both design flows by applying them to a real problem. They are compared from the perspective of the ability to optimize resource usage (as an indirect measure) by using different modeling techniques and expressive power offered by each language. This way our aim is to evaluate the quality of MyHDL autogenerated code and the ability to make optimizations in the Python level have a direct impact on the final design. They have also been evaluated the expressive power of Python as a modeling language and the code quality from the standpoint of readability, code size productivity (SLOCs) and potential benefits in contrast with non-automatic generated code.

5. APLICATION AND RESULT

To apply the described methodology, we obtained a first functional design implementing internal RAM in Verilog. This first design was developed without applying any kind of optimization, in the clearest and modular manner possible as described by the specifications of the device. This design was then improved over several iterations. Among the improvements, we applied techniques such as resource sharing, FSM unification, output values dependent on the status register bits, etc... This optimization process produced resource savings of around 50% of both FFs and LUTs reducing the initial design occupation from 435 to 211 LUTs and from 212 to 118 FFs. An item on which there were no savings during this process was the use of a single BRAM block, inferred by XST to implement the internal character memory.

To check whether the series of optimizations which had been applied using Verilog were also applicable to a design made in Python, a first version was developed using MyHDL modeling techniques. This first version was a translation from the core in Verilog, which tried to respect as far as possible the structure and philosophy of the original core.

The results obtained by the synthesis tool from this first Python version, as we can see in figure 3, were surprising, not only because they were not worse than those obtained by the conscientious application of optimization techniques, but because the synthesis tool did not infer elements in the same way. The Python version reduced by 27% the usage of FFs and increased by 13% the usage of LUTs with respect to the optimal Verilog core. This increase of LUTs can be explained because they were used to implement the internal RAM instead of using a BRAM block, thus saving on BRAM blocks in this case was 100%. With this first Python design, any further optimization was achieved leading to an improvement in overall occupancy.

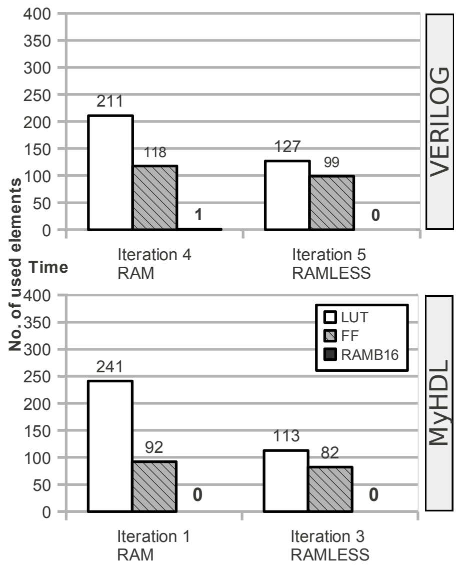

Fig. 3. Occupation comparison of the best obtained results using RAM and RAMLESS approaches for Verilog (top) and MyHDL (bottom) flows

At this point, there were no improvements in any of the two flows without removing the memory. To continue the optimization process, the two cores were modified to eliminate the need of an internal memory, replacing it by the external chip's memory at the cost of a higher latency. This change resulted in dramatic improvements regarding to resource usage. In the case of the Verilog design, the BRAM block wasn't used anymore and the reduction in the use of LUTs and FFs was respectively of 40% and 16%. In the case of the Python design, there was a reduction of 53% of LUTs and 12% of FFs.

Although further global usage savings were not achieved, since FPGAs have a fixed proportion between logic and storage elements, such as the Spartan 3E on which the comparison was made, we tried to balance the use of LUTs and FFs to prevent an increase in overall occupation if we use slices or CLBs as measure unit. Based on the optimum designs achieved in the previous stage, in the case of Verilog, it did not get any improvements in the balance of the two types of elements without increasing the overall occupancy. In the case of the Python design, the optimal balance was achieved using 108 LUTs and 108 FFs.

Another evaluated area was the impact of adding wrapper modules to the core to adapt its interface. In this case an interface based on the Wishbone standard was developed, mainly due to its widespread use in the world of System-on-Chip [20]. The specific way that MyHDL uses to generate code, flattening the hierarchy of generators in a single module, provides certain advantages as we can infer from the obtained results. We have measured occupation of all designs using both the native memory mapped interface and the wishbone wrapper. In all cases the Verilog design increased slightly its occupation, becoming minimal in the last iteration with an overload of only 3 LUTs. In the case of the MyHDL based flow, the addition of this interface not only has not brought an extra usage, but has saved in the best case 4 LUTs and 3 FFs with respect to the memory mapped one. This result is possible because the entire hierarchy is flattened within a single module so the synthesis tool has been able to simplify those elements that are no longer useful when everything is perceived as a single system. With this proof of concept, we must not forget that we are not trying to demonstrate the advantage of self-generated code against Verilog code, but the expressiveness of Python with the ability to generate functionally equivalent results in the same range of benefits to those obtained manually. Probably, savings on the same range would have been obtained in Verilog if the synthesis tool had used hierarchy flattening directives.

Another important aspect that we have been able to assess is tightly related to development productivity and quality. Over all designs we have measured the number of SLOCS (source lines of code) by adapting an Open Source software called SlocCount to support VHDL and Verilog. This kind of measure is interesting from the perspective of development time, cost and quality. Languages that require less SLOCS are more time and cost productive and in addition designs with less SLOCs are likely to be less bug prone. As we can see in figure 4, we have compared the SLOCs of hand coded Python and Verilog together with atomatically generated Verilog and VHDL for each of the four designs. There is a high correlation along the four designs so if we analyze these results, on average, we can see that Hand coded MyHDL designs are 281 SLOCs long against Hand coded Verilog designs that are 419 SLOCs long what means that an increase of around 50%. Automatically generated Verilog and VHDL designs are not directly comparable since their hierarchies are flattened in a single component.

6. CONCLUSION

With this contribution, the results of a comparison between a high level Pyton based flow and a low level traditional development flow is being presented for the first time. We have been able to obtain an initial evaluation of MyHDL as an alternative to traditional development flows by its application to a real development. Although the results have been obtained using a small sample, they have been good enough (in some cases better than those obtained with hand coded Verilog) to justify further research.

The main explanation about what makes Python a good HDL and the mechanisms that make it so effective is that

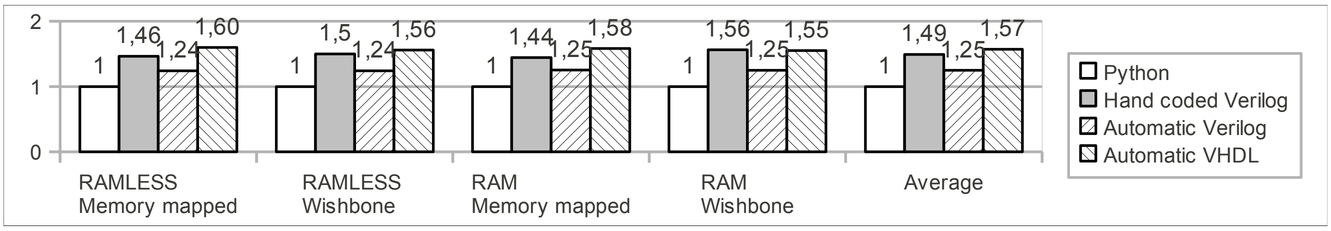

Fig. 4. Normalized source lines of code (SLOCS) of hand coded Verilog, hand coded MyHDL and autogenerated outputs for every design

high level descriptions gives MyHDL better information than what it can be understood from low level descriptions. Although it's not a behavioral-synthesis tool, it is more intelligent than just a translator. This way it is able to choose better and more optimal elements automatically. As design complexity increases, low level descriptions become exponentially more complex. This is equivalent to what happens in software where automatic optimization techniques have become increasingly more productive and successfull than human optimizations.

The fact that both Python and MyHDL are Free Software [21] makes them a useful hardware design tool for HDL research because of its possibility to be modified and adapted to new scenarios. With MyHDL, new ideas and methodologies within the field of HDLs can be developed and applied to real designs.

All generated cores discussed in this contribution are availaible for public review as a Free Hardware project hosted at Opencores [22].

7. REFERENCES

[1] G. DeMichelli, *Modeling Languages and Abstract Models*. Stanford University, 2006.

[2] R. Damasevicius, "A subset-based comparison of main design languages," *Informacines technologijos ir valdymas*, no. 3 (6), pp. 20–29, 2004.

[3] *IEEE standard VHDL language reference manual (integrated with VHDL-AMS changes), IEEE std 1076.1*. IEEE Standards Board, 1997.

[4] *IEEE standard for verilog hardware description language, IEEE std. 1364-2005*. IEEE Standards Board, 2006.

[5] *IEEE Standard VHDL Language Reference Manual Amendment 1: Procedural Language Application Interface, IEEE std 1076c-2007*. IEEE Standards Board, 2007.

[6] S. Guo and W. Luk, "Compiling ruby into fpgas," in *Field-Programmable Logic and Applications*, 1995.

[7] P. Bellows and B. Hutchings, "Jhdl - an hdl for reconfigurable systems," in *IEEE Workshop on FPGAs for Custom Computing Machines*, no. 3 (6), 1998, pp. 175–184.

[8] J. Decaluwe, *MyHDL Manual 0.6*, 2009.

[9] G. Van Rossum, *The Python Language Reference Manual*. Network Theory Ltd, 2003.

[10] J. Sobel and D. Friedman, *An Introduction to Reflection-Oriented Programming*. University of Indiana, 1996.

[11] *PEP 289: Generator Expressions*. Python Enhancement Proposals, 2002.

[12] *PEP 318: Decorators for Functions and Methods*. Python Enhancement Proposals, 2003.

[13] D. Mertz, "Charming python: Decorators make magic easy; a look at the newest python facility for metaprogramming," *IBM developerWorks*, 2006.

[14] S. Ur and A. Ziv, "Cross-fertilization between hardware verification and software testing," in *6th IASTED International Conference on Software Engineering and Applications (SEA2002)*, 2002.

[15] A. Cockburn, *Agile Software Development: The Cooperative Game (2nd Edition)*, ser. The Agile Software Development Series. Addison-Wesley Professional, 2006.

[16] P. Runeson, "A survey of unit testing practices," *IEEE Software Magazine*, no. V23n4, pp. 22–29, 2006.

[17] *Scipy 0.7 Reference Guide, http://docs.scipy.org/doc/*. SciPy Project, 2009.

[18] Sitronix, *16Cx40S Dot Matrix LCDController/Driver Specification Sheet*. Sitronix, 2008.

[19] R. Herveille, *WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*. Opencores, 2002.

[20] U. Kamth and R. Kaudin, "System-on-chip designs: Stategy for success," June 2001.

[21] R. Stallman, *Free Software, Free Society: Selected Essays of Richard M. Stallman*. Free Software Foundation, 2002.

[22] *WB LCD Source code http://www.opencores.org/projects/wb_lcd*. Opencores, 2010.