Custom processors design using Python-based high level synthesis

Diseño de procesadores a la medida usando síntesis Python de alto nivel

Jaime Parra P.
jparra(AT)admon.uniajc.edu.co
Institución Universitaria Antonio José Camacho – Colombia

ABSTRACT

It is explained a computational tool, PyHDL, conceived to assist the designer of custom processors. PyHDL takes a description given in Python, an extremely friendly language that allows writing descriptions as close to natural language descriptions as possible with current technology in programming languages. The tool carries out the generation of a synthesizable VHDL equivalent of the description given by the user, which is directly implementable in hardware, along with the necessary files to simulate and implement the design. The purpose with this tool is to facilitate the design of embedded systems and make it more accessible to a greater number of students, engineers, and users in general.

Keywords: Software development, custom processors, design tools.

RESUMEN

Se explica una herramienta computacional, PyHDL, concebida para asistir al diseñador de procesadores a la medida. PyHDL parte de una descripción dada en Python, un lenguaje extremadamente amigable que permite la escritura de descripciones tan cercanas a descripciones en lenguaje natural como es posible con la tecnología actual en lenguajes de programación. La herramienta se encarga de generar un equivalente en VHDL sintetizable de la descripción dada por el usuario, la cual es directamente implementable en hardware, junto con los archivos necesarios para simular e implementar el diseño. A través de esta herramienta se busca facilitar el diseño de sistemas embebidos y hacerlo más accesible a mayor número de estudiantes, ingenieros y usuarios en general.

Palabras clave: Desarrollo de software, procesadores a la medida, herramientas de diseño.

1 Introduction

Embedded systems are reactive systems that interact intensively with their environments and users, must run multiple tasks in parallel and must, at times, react in real time [1]. Added to this is that they should also have low power consumption and occupy little volume, both desirable characteristics to enable portability, key in mobile computation nowadays [2].

The three most popular technologies for implementing embedded systems are Application-specific integrated circuits (ASIC), microcontroller, and reconfigurable hardware (FPGA, for field-programmable gate arrays) [3]. ASICs are used when the number of units to be manufactured justifies it and/or the application is very particular and its presence in market is estimated to be from medium to long term. Microcontrollers are the usual choice for low production volumes and/or medium or simple applications [4]. FPGAs are chosen when dealing with applications with conflicting requirements and/or in the case of experimental applications [5].

A widespread way to design solutions for the FPGA case is posing a description using a Hardware Description Language (HDL), which allows the designer to think in terms of high-level abstractions related to meeting the requirements, while provides constructions associated with the hardware and the physical world which are not usually present in conventional programming languages, that are focused on microprocessors programming [6].

While designing in an HDL can achieve much more complex systems than those using only basic digital systems tools such as truth tables or schematics, it is also clear that existing HDLs have certain characteristics that hinder their wider acceptance and use. The four that stand out are: excessive verbosity, strong data typing, little encapsulation, and very explicit connection with hardware [7].

The solution proposed in this paper is the creation of a computing environment, called PyHDL, which allows the designer to describe his/her solution using Python, a contemporary programming language which is valid to represent parallel and interactive efficient solutions. Since Python is a language focused on programming conventional computers it is not directly synthesizable on hardware, hence PyHDL covers this aspect, generating a VHDL equivalent of the description given in Python, which can be easily synthesizable on ASIC or FPGA hardware [8].

PyHDL is therefore an EDA (Electronic Design Automation) tool, conceived to assist the designer of
complex digital systems, especially specific-purpose processors. The tool is a cross-compiler, whose primary purpose is to automate the tasks associated with the generation of the several files required by tools that make simulation, synthesis and implementation. Thus the tool frees the designer of those details that can be considered of low level, and lets him/her focus on the more abstract aspects of the system and express them through a contemporary high-level language, Python [9]. PyHDL receives a file written in pure conventional Python and processes in order to extract a hierarchy of structures from the usual constructions in a programming language: functions, loops, decisions and assignments. The outcome generated by PyHDL is a set of files that can be employed directly by EDA tools, for automating the electronic design.

EDA tools focus primarily on one or more of the four fundamental aspects that are part of the design flow: modeling, simulation, synthesis, and implementation [10]. In modeling the designer takes the ideas from which he/she conceives the solution to a given problem and expresses them through some form of representation. In the case of digital systems, there are different types of representations, ranging from the closest to hardware such as schematics to those of algorithmic type, passing through intermediate forms such as truth tables, state diagrams or Petri nets.

Simulation involves running an application that provides an equivalent of the environment in which the solution would run with the same input or excitation vectors, to which the solution generates answers according to the modeling given to its behavior. The simulator presents this information to the user or can take care of carrying out some analysis in order to identify possible flaws in the system behavior. Synthesis consists in mapping the representation of the solution to some kind of hardware. The usual alternatives are either a custom-made device, or ASIC, or a prototype using reconfigurable hardware, or FPGA. In both cases the job of the synthesis is splitting the representation into small modules and fit them in such a way that matches the alternatives present in a component library. Finally, implementation materializes that synthesis, leading to a real hardware on which the solution can be run [11].

Of the four stages mentioned, the last three (simulation, synthesis, and implementation) are fairly mature, to a lesser extent the first one. Moreover, given the enormous complexity of today's digital systems, it is almost impossible to perform these tasks by a human being. That is the reason why improvement is almost an exclusive domain of large manufacturers of EDA tools. That is not the case with the first stage, modeling, which, because of being the closest to human beings, and in fact the most critical, since from its outcome depends the others, still has much improvement work.

PyHDL intervenes in this stage of modeling, providing an interface for the user to translate their ideas into an algorithmic model written in the Python programming language, taking advantage of its remarkable educational characteristics, which have proved to be of great influence on the quality of the model generated [12]. Therefore it seeks that obtaining a functional solution be achieved in less time and that the developed product be more robust and easier to analyze, with a view to future improvements or additions or expansions.

2 Processors

Computation is physically performed by processors. A processor takes certain input data and transforms them into their respective output data according to the rules dictated by the algorithm that directs the activities of the processor. In general, the most widely spread model for the processor represents it as made of two modules: a controller and a data path [13]. The controller is a finite state machine that follows the algorithm, while the data path is an arithmetic/logic execution unit (ALU) where data transformations are carried out. According to the way these two modules are related, processors are classified into one of three groups: general purpose, application specific or custom or specific purpose [14].

General purpose processors are those processors that are commonly found in commercial computers. Their controller receives as input an instruction from the algorithm encoded in a particular machine language the processor can recognize, decodes and prepares them, also generic, data path to transform the data contained or indicated by the instruction (Figure 1).

Specific application processors are designed for a particular application segment and their architectures are optimized to efficiently carry out typical operations in the implementation in case. Examples of these cases can be audio or video processors, where it is common to use transforms; or processors for scientific applications, where it is often to use numbers in floating point notation and where it is required to maintain accuracy between calculations [15]. The biggest difference with the previous case is that the ALU is optimized to carry out the transformations required in the specific field of application for which processor was designed (Figure 2).
Specific purpose processors are designed to explicitly match the requirements of a particular behavior. Their controller executes a single algorithm and their ALU executes only the transformations present in the target behavior [16]. As such, they are optimal in the sense that there is no general-purpose or application-specific processor that can overcome them with respect to the particular behavior defined (Figure 3).

![Figure 3: Application-specific processor](image)

### 3 Cross-compilers

A compiler is a program that transforms a code written in a source programming language into another functionally equivalent code written in object, binary, or machine language. The primary reason for the existence of compilers is to allow human programmers to write programs using high level languages whose primitives are focused on abstract concepts of algorithmic kind, such as assignments, decisions, loops, and functions, but when running on a real machine, it can make use of an optimized version for it, which is written using as primitives the basic operations of the machine, which are actually quite modest: assignment, addition, conditional and unconditional jump.

Since a high level instruction usually translates in several low level instructions, the compiler must use a technique called parsing, by which the instruction is broken into logical components. Modern compilers not only translate the code, but can also account for errors in it and make improvements. The tasks typically done by compilers are: lexical analysis, preprocessing, parsing, semantic analysis, code generation, and code optimization [17].

If a compiler is a program that translates code written in a high-level programming language into a low-level or machine equivalent program, a cross-compiler performs a similar task, but has as target another high-level programming language or, as it is in this case, a different-domain language. Figure 4 shows the steps that are commonly carried out by a compiler during its job [18]. The tasks of a cross-compiler usually end with the translation task or, in some cases, with the optimization task; the code generation step is not required, since no low-level code is implied.

![Figure 4: Usual steps carried out by a compiler](image)

The activities of a compiler can be divided into two large stages:

- Analysis of the source code to be compiled
- Synthesis of a target functionally-equivalent code.

In more detail, the main steps during a compilation process are:

- **Scanning:** It consists of reading the input text (usually present in a file) and grouping individual characters into tokens such as identifiers, integers, reserved words, or delimiters. The scanning task builds a referential structure, known as AST for Abstract Syntax Tree, where the different elements that are identified in the source code are put [19]. The successive steps will enrich the information associated with the elements present in the AST. Since Python employs tab characters as delimiters, PyHDL also stores information about the level of each sentence during this step.

- **Parsing:** In this step, the structures created in the previous scanning step related to the organization of tokens are analyzed and distilled to assign meaning and hierarchy dependences to them. Although during parsing is common to verify the code for correct syntax, PyHDL assumes that the code is syntactically correct. Nonetheless, some simple checking is performed for instance to detect improper tabbing or the use of reserved words as identifiers.

- **Type checking:** It is the checking of the semantics of each AST node. That is, it verifies that the construct the node represents is legal and meaningful, which is mainly associated with determining that all identifiers involved are declared, that their types are correct for the expression being evaluated, among other things. Since data in hardware are stored in predefined registers, PyHDL also checks if the expression implies a change of type for a given variable.

- **Translation:** In this step the abstract representation contained as nodes in the AST is converted into explicit code according to the rules that govern the target language. PyHDL uses for this step a template-driven approach, i.e., the target VHDL code is presented as a generic, static text with some special, replaceable characters that are built on-line as the template is written.

- **Optimization:** PyHDL takes care of possible assignments that can be done in parallel. To do that, it determines the dependencies between the variables involved in assignment instructions immediately followed by other assignment instructions. If no dependencies are detected, those assignments are all put in the same VHDL state.

PyHDL builds the AST identifying the next general syntax construction rules:

```plaintext
# Identifier := var | fun
```


PyHDL is a cross-compiler tool conceived to efficiently describe designs of custom processors. Its purposes are: to allow the user writing his/her designs in standard Python code; to translate such a code into HDL code, automatically managing all the low-level aspects such as clocks, state machines, conversion functions, and concurrency; to generate test benches to simulate such descriptions; and to generate constraint files to ease implementation of the designs in reconfigurable hardware.

PyHDL intentionally hides the particularities of hardware, such as clocks, gates, LUTs, flip-flops, signals, bits, and so on, which, although essential for the professional of computation, are not easy to get for people coming from other disciplines. In this way, the user is invited to think in a high-level domain with respect to his/her solution and let PyHDL take care of the low-level details concerning implementation on physical devices.

PyHDL is a high-level design tool. With it, the designer no longer needs to think in terms of hardware but of the abstract structures of his/her problem. But, why using PyHDL if there are other high-level design tools? The answer is that, in general, most of these tools preserve an explicit link with hardware.

MyHDL [20] is one popular environment at this respect. It is a Python package that allows description in full detail of hardware systems. However, its flavor is still too hardware-based, where concepts such as clock, flip-flop, or edge must be understood. Tools such as Matlab© has, as a part of its set of tools, some interaction with hardware. Simulink, one of such tools, allows designing parametric filters and, for implementation in reconfigurable hardware, permits exportation in VHDL format [21].

A complement is a tool that automates specific aspects of digital design. For instance, PyVHDL [22] is a Python module intended to add scan structures to a hardware design in order to make it testable by standard industrial protocols. In the case of SystemC [23], the user gets a set of classes and macros written in C++. Since it is capable of modeling systems having concurrent processes, it makes part of some EDA tools. Nonetheless, it still has the unpleasant verbose characteristic.

Many high-level synthesis tools use a similar approach. Chisel [24] builds on the Scala language and is described as a set of special class definitions, predefined objects, and usage conventions within Scala, which, again, are thought of in terms of hardware. PyCPU [25] provides a set of Python modules to generate a general-purpose processor and to program it using a limited subset of the Python language.

In order to show how PyHDL works, let us consider a didactic example, the design of a custom processor that performs the multiplying operation through successive additions. The Python code is shown in Figure 5.

![Figure 5: Python code multiplier example](image)

```python
# header := def fun(parameters)
#
# parameters := " [ , var ... ] 
#
# instruction := var = expression | return expression | if condition: | else: | while condition: 
#
# expression := term |
# term arithOp term |
# fun(expressions) |
# term := constant | var |
# arithOp := + | - | * 
# condition := term relOp term |
# relOp := < | <= | == | != | >= |

An identifier represents the name of either a variable or a function. Some standard rules apply to the proper construction of an identifier:

- It can only contain letters, numbers and the "_" symbol.
- It cannot have spaces in between.
- It cannot start with a number.

Additionally, VHDL imposes other restrictions that by extension a Python identifier must fulfill:

- It cannot start with the "_" symbol.
- It cannot end with the "_" symbol.
- It cannot have more than one successive "_" symbol.

A function definition has the structure: def fun(parameters)

Where def is the keyword indicating it is a function definition, fun is the identifier for that function, and (parameters) is an optional list of input variables.

An instruction can be any of these statements: assignment, if, else, while, or return. An assignment has the form: var = expression

Where var is a variable identifier and expression is an arithmetic expression that produces a number, and that can be a term, an operation between two terms, or a function calling. A term is either a variable or a constant. An operation is an addition, a subtraction, or a multiplication. A return has the form: return expression

Where return is the keyword and expression is as before. An if and a while have the form:

if condition:
  while condition:

Where if and while are the keywords and condition is a logical expression of the form term relOp term, in which relOp is a relational operator that indicates the kind of comparison to be done (less than, less than of equal, equal, different, greater than or equal, and greater than).

4 PyHDL

PyHDL is a cross-compiler tool conceived to efficiently describe designs of custom processors. Its
In order to generate a custom processor with the desired behavior, it is necessary to associate high-level instructions with low-level machine states. The source code has six instructions, which could be translated to a six-state machine. However, PyHDL does a better job by optimizing the instruction to state mapping. This is done determining what instructions can be executed in parallel. Table 1 presents these concepts applied to the multiplier example.

Table 1: High-level instructions to low-level states mapping

<table>
<thead>
<tr>
<th>State</th>
<th>Line number</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td># PyHDL: multiplier example.</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td># set initial values</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td># idle5</td>
<td>5</td>
<td>accum = 0</td>
</tr>
<tr>
<td># while8</td>
<td>7</td>
<td># while count=num2:</td>
</tr>
<tr>
<td>assign9</td>
<td>9</td>
<td>accum += num1</td>
</tr>
<tr>
<td>ret11</td>
<td>11</td>
<td>return accum</td>
</tr>
</tbody>
</table>

The code the tool generates is a VHDL-based state machine. The interface with the module is described as an entity (Figure 6). PyHDL automatically adds three signals: start, ready, and result. Start is a signal that allows the machine proceeding with its calculation. Ready is a signal the machine asserts to indicate it has finished and the result obtained is present in signal Result.

In order to manage assignments, PyHDL builds a custom data path taking into account the local variables and constants (see Figure 8). Note the parallelization of instructions fruit of the optimization process performed by PyHDL. The logic the algorithm must follow is provided by the sequencing of instruction in charge of the controller. PyHDL generates the next-state logic based on the kinds of instructions and the conditions present in the source code (Figure 9). Observe how PyHDL detects what signals should go in the sensitivity list of the process, in this case only count, which is the only one that is both modified inside the loop and used in a condition.

Figure 9: VHDL next-state logic multiplier example

The multiplier example is just a proof of concept. More elaborated designs require multiple iterations and block nesting. In order to test the tool capabilities for these alternatives, three more challenging specifications were presented to two groups of people including students, teachers, and engineers having different levels of expertise. The specifications, in incremental order of complexity were: a 3-number comparator, an integer squared root, and a universal synchronous/asynchronous receiver/transmitter (USART). Figure 10 shows a solution given by a student for the squared root problem.

Figure 10: Python code squared root example

In order to manage assignments, PyHDL builds a custom data path taking into account the local variables and constants (see Figure 8). Note the parallelization of instructions fruit of the optimization process performed by PyHDL. The logic the algorithm must follow is provided by the sequencing of instruction in charge of the controller. PyHDL generates the next-state logic based on the kinds of instructions and the conditions present in the source code (Figure 9). Observe how PyHDL detects what signals should go in the sensitivity list of the process, in this case only count, which is the only one that is both modified inside the loop and used in a condition.

In order to manage assignments, PyHDL builds a custom data path taking into account the local variables and constants (see Figure 8). Note the parallelization of instructions fruit of the optimization process performed by PyHDL. The logic the algorithm must follow is provided by the sequencing of instruction in charge of the controller. PyHDL generates the next-state logic based on the kinds of instructions and the conditions present in the source code (Figure 9). Observe how PyHDL detects what signals should go in the sensitivity list of the process, in this case only count, which is the only one that is both modified inside the loop and used in a condition.
PyHDL for designers with different levels of expertise. As expected, the greater the user knowledge, the lesser the resources required. However, the notable aspect to consider here is the relative comparison between designers with the same level of expertise with respect to such a feature. In all cases, the users that employed PyHDL outperformed those that used VHDL directly for the same task.

However, hardware description languages have a very inefficient syntax that does not match modern standards and a very explicit hardware-oriented approach, both factors that become important obstacles for many users to successfully implement their designs. High-level synthesis tools offer a more abstract approach for designing, but most of them are still too focused to hardware constructions, which reduces their target audience to hardware specialists.

PyHDL is a tool that covers such a gap, giving users the opportunity to describe his/her designs in a modern, more friendly language such as Python, and automating generation of the files required by simulation and synthesis tools. PyHDL properly manages Python files containing any arbitrary number of functions, interrelated through any arbitrary hierarchical relationship.

Or course, it is possible to add more functionality. Concurrency may be further optimized by deciding when a condition can be safely put in a transition instead of in a state, reducing the number of states required. Also, data types may be extended to accept fixed-point numbers, taking into account that a general implementation of floating-point numbers is not recommended for reconfigurable hardware devices because of the prohibited amount of resources it requires in absence of built-in floating-point blocks.

The tool can be enhanced by adding libraries of predefined VHDL and/or Python modules, for operations such as division or exponentiation, or others required for particular tasks.

5 Conclusions

Modern designs in hardware require considering very complex systems. Reconfigurable hardware devices permit a rapid prototyping and exploration of possibilities for such complex digital systems designs. In order to take advantage of the resources available on a reconfigurable hardware device, a powerful description language is mandatory, and languages such as VHDL are expressive enough to meet the requirements.

However, hardware description languages have a very inefficient syntax that does not match modern standards and a very explicit hardware-oriented approach, both factors that become important obstacles for many users to successfully implement their designs. High-level synthesis tools offer a more abstract approach for designing, but most of them are still too focused to hardware constructions, which reduces their target audience to hardware specialists.

PyHDL is a tool that covers such a gap, giving users the opportunity to describe his/her designs in a modern, more friendly language such as Python, and automating generation of the files required by simulation and synthesis tools. PyHDL properly manages Python files containing any arbitrary number of functions, interrelated through any arbitrary hierarchical relationship.

Or course, it is possible to add more functionality. Concurrency may be further optimized by deciding when a condition can be safely put in a transition instead of in a state, reducing the number of states required. Also, data types may be extended to accept fixed-point numbers, taking into account that a general implementation of floating-point numbers is not recommended for reconfigurable hardware devices because of the prohibited amount of resources it requires in absence of built-in floating-point blocks.

The tool can be enhanced by adding libraries of predefined VHDL and/or Python modules, for operations such as division or exponentiation, or others required for particular tasks.

References