Hardware Reuse Improvement through the Domain Specific Language dHDL

Miguel A. Sánchez, Marisa López-Vallejo

Carlos A. Iglesias

***Abstract*—The dHDL language has been defined to improve hardware design productivity. This is achieved through the definition of a better reuse interface (including parameters, attributes and macroports) and the creation of control structures that help the designer in the hardware generation process.**

I. Rationale: Why a new DSL is defined

Current technologies deep in the nanometer regime allow the integration of millions of transistors in a single FPGA. Consequently, the development of parallel machines with hardware co-processors has now become a reality. However, the complexity of hardware design makes the design cycle too long and complicated. A serious improvement in productivity is required, and this is only affordable by component reuse, which makes possible to cope with even greater designs, simultaneously reducing the time to market. Reused components, known as cores, may come from former designs or from third parties. In the last case, they are known as Intellectual Property (IP), based on licensing reasons.

System design is thus simplified by proper assembly of such parts, and the identification of the best ones and their parameters is typically a time consuming task. Hence, IP-based design actually needs tools to simplify component assembly and core generation, and also to provide quality assessment measures (area, speed, power, accuracy) that ease the selection. These goals can only be accomplished if the cores are specified with a language that allows subcomponent customization, instantiation and interconnection, with the corresponding simplification of the design of hierarchical collections of modules.

In this sense, conventional hardware description languages, as VHDL or Verilog, do not satisfy all the requirements that IP reuse may have. For instance, these languages exhibit serious limitations in parameterizable design or do not allow the specification of additional attributes which could drive a design space exploration process.

To overcome these limitations, in this work we present dHDL (dynamic HDL), a domain specific language [1] that helps the designer in the difficult task of IP core description and reuse. The idea behind dHDL is to extend the features of conventional HDLs (currently VHDL) following the component-based design paradigm[2] to ease both the parameterization of designs and the reuse of previous works. In this way, dHDL provides a simple and clear way to describe complex systems, while being easy to learn and use.

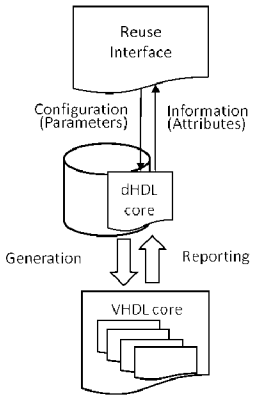

Fig. 1. Role of dHDL description.

The language is accompanied by a compiler that generates VHDL source code that can be input to commercial synthesis tools. dHDL provides a wrapper to VHDL cores extending their conventional reuse capabilities. Therefore, not only core configuration and generation is allowed, but also information regarding the core implementation can be reported to the new core interface in the form of attributes (Figure 1).

II. dHDL Essencials

dHDL seeks to improve the generalization, generation and reuse of traditional HDLs. The possibilities offered by VHDL parameterization are rather limited, largely due to the rigidity imposed by the language itself.

We want to improve the information offered by VHDL cores first by extending the interface, as shown in Fig. 2. More complex configuration parameters can be defined. Furthermore, attributes reporting on particular implementation data can be also declared, what cannot be done in VHDL. Finally, to simplify the core interconnection and instantiation processes we have defined macroports.

Regarding the language itself it provides great flexibility when defining variables, instantiating components or creating structures. Additionally, these mechanisms allow the collection of information during the generation phase that is fed back that

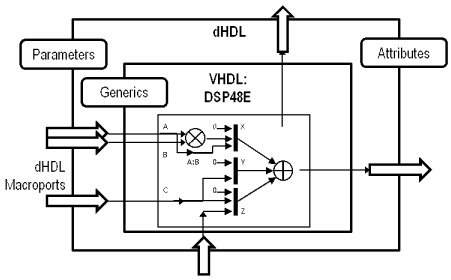

Fig. 2. dHDL interface.

to the interface as design attributes.

A. Interface

The dHDL interface allows to declare three basic elements:

- **Parameters**, which can be considered as inputs to configure the core. Differently from VHDL, where the generics are interpreted by the synthesizer, in dHDL parameters disappear after code generation, being substituted by their values.
- **Attributes** are elements that inform on the component characteristics (output information). When VHDL code is generated, attributes behave as global variables which can be read and modified. After code generation, attribute values are returned to the interface to inform about the result of the generation process (area, latency, etc.).
- **Macroports** constitute the input-output for data interfacing. Each macroport gathers an entire set of signals, which are also tied to a protocol. Macroports can be conditionally declared based on the evaluation of particular parameter values.It is also possible to use a unique identifier to declare an array of macroports. Macroports have been designed to simplify component interconnection.

B. Core Description

dHDL cores are oriented to describe a core based on other cores, where structural description is more suitable than behavioral description: a core is a set of instances of components, the interconnections between these components and the ports of the core itself.

Types declared in dHDL allow the designer to control how an when VHDL code is generated, so which elements will construct the structure defined by control structures. Control structures defined in dHDL are more flexible, and use interface configuration to guide the structural description.

It is also possible to include behavioral descriptions with dHDL. VHDL cores can be instantiated, and the language includes a mechanism to insert customized VHDL code.

The language constructs for the core description are:

- **Control Structures**, that allow the designer to control the flow of generation of the *core* and thus the component structural description. These structures also bring visibility to the types defined and can be chained and nested together. dHDL control structures are of two types: conditional structures, based on the`if-else` construct, and iterative following the `while` construct. Both structures are based on the evaluation of a condition. Conditions can be simple (evaluated with relational operators) or complex conditions, which are constructed with logical operators allowing the evaluation of a chain of simple conditions.
- **Declarations**: dHDL code can access to elements declared in the interface (parameters, attributes and macroports). But it can also declare and use new types: variables, signals, ports, entities and components. Declarations are allowed all along the code. Scope of variables and signals is also considered.
- **Code insertions**. To properly describe a component, it is also necessary to perform VHDL code insertions in the language. Those insertions can be customized by using data types, through the substitution of their actual values during generation. In code insertions, VHDL is interpreted as plain text, but accepts substitutions anywhere, triggered by brackets around language expressions, which use formerly defined types (e.g. parameters, functions, properties, variables). This allows extensive code customization.

III. Design examples

To check the usefulness of dHDL, showing its customization and interconnection capabilities, several cores have been implemented and integrated into different dHDL libraries. These libraries contain from simple components, as adders or registers, to more sophisticated ones, as general KCMs, CORDIC rotators, filters of FFTs. Additionally, basic dHDL structures/modules have been built, such as input/ouput macroports, connectors, etc.

IV. Conclusions

The use of FPGA-based hardware accelerators is characterized by very long design cycles. We have conceived the dHDL language to improve hardware reuse targetting the increase of design productivity. The main advantages of dHDL are the following:

- It is conceived to emphasize reuse.
- It allows a high degree of parameterization of the cores.
- It simplifies the specification with constructs that ease the designer's work.
- No conditions are imposed to the HDL used to describe the designs.
- It is easy to learn, since it stays close to conventional HDLs used by designers.
- It allows design space exploration based on the mechanisms defined to provide feedback information to the interface in the form of attributes.

References

[1] A. Raja and D. Lakshmanan, "Domain Specific Languages," *International Journal of Computer Applications IJCA*, vol. 1, no. 21, pp. 105–111, 2010.

[2] E. A. Lee and A. L. Sangiovanni-Vincentelli, "Component-based design for the future," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2011, pp. 1–5.