evercodeML: a formal language for SoC integration

José I. Villar, Jorge Juan, David Guerrero, Manuel J. Bellido, Julián Viejo
Departamento de Tecnología Electrónica
Universidad de Sevilla
Spain
E-mail: {jose, jjchico, guerre, bellido, julian}@dte.us.es

***Abstract*—Complex SoC design devote a great part of the developing time to module integration tasks. The necessity of automating system integration at high-level has yield to the development of module description languages like IP-XACT. However, the available options today still lack advanced parametrization capabilities needed to design complex systems with very heterogeneous IP-cores and module providers. This contribution introduces a formal language for SoC integration that overcomes these limitations.**

***Keywords*.- FPGA, SoC, IP-core, IP-XACT.**

I. INTRODUCTION

In the last few years FPGA devices have experienced a great increment in the level of integration and performance. This development of configurable devices has brought a remarkable variant of the SoC [1] (System on Chip) approach so-called SoPC (System on Programmable Chip) that has made it necessary the adoption of new methodologies tailored to the highly increased flexibility of the programmable platforms and the growing complexity of tackled designs.

On the other hand, the rising success of programmable devices, has also made hardware development extremely popular, reaching new target audiences as small/medium enterprises, academic environments and even individual developers that until FPGA popularization could not afford designing in the SoC arena. Some initiatives have brought successful ideas from Free Software [2] to the hardware area. The most relevant of these initiatives is the Opencores portal [3], which offers over 500 ready-to-use projects, including a variety of CPUs, arithmetic cores, peripheral controllers and complete SoC systems among others.

Because of these the available hardware core portfolio is huge but extremely heterogeneous in quality, packaging, distribution methods, and licensing conditions. It makes system integration a tough task. In particular, the potential of many good quality and useful open cores is not exploited not only because of the absence of publicity but also due to the fact that they were conceived for a particular application and later released to the community with little or no documentation.

Additionally, to build a complete SoC, developers also have to deal with specific design constrains like clock frequency, reset generation, interconnection strategies, etc. which makes it even harder to integrate third party cores even if the core design is available and can be modified. These limitations have been identified by the industry in recent years and some effort have been devoted to create formal core description and packaging languages like IP-XACT [4] and CHREC [5].

IP-XACT has become an international standard in 2009 [4]. While IP-XACT does an excellent job at describing the functional characteristics of a SoC design, it lacks the flexibility needed by reconfigurable cores, where ~~a~~ complex parameter interdependencies are common. Another XML schema named CHREC has been proposed in [5]. CHREC is more FPGA friendly with strong support for parametrization including parameter evaluation. However, both formats lack some characteristics to tackle the increasingly complex integration problems like repository handling or complex parametrization found in descriptions intended for core-generation tools.

This contribution is specifically devoted to the description of the second iteration over a novel core description markup language which is part of a larger project aimed at developing a general strategy to facilitate the reuse and integration of available hardware cores in complex scenarios.

In section II there is an overview of the type of system the proposed format is intended to. In section III the core description's most relevant capabilities are described in some detail. Section IV comments some sample descriptions that highlights the potential of the language and section V derives some conclusion and future work.

II. SYSTEM OVERVIEW

Figure 1 depicts an overview of a fully-automated, high-level SoC integration system. The platform should be able to locate the required cores in a set of repositories, fetch them and produce a HDL source that can be further processed by conventional vendor-specific tools to produce a final implementation. A brief description of the main tasks involved follows.

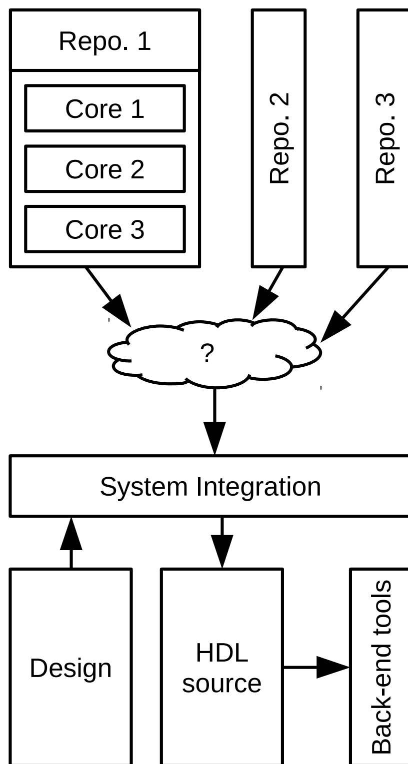

Figure 1. *SoC integration overview.*

A. *Core Packaging and Distribution*

A set of package repositories (either remote or local) plays the role of a distributed library of cores. Every package includes the original design of the core that may include HDL source code, configuration and synthesis scripts, core generators configuration and/or already compiled netlists. Every packages also contains machine readable meta-information about the design to allow for the automatic acquisition and integration of these components in larger designs. This meta-information ranges from the semantics of its interfaces to implementation and functional constraints, dependencies with other cores, configuration and parametrization capabilities and any other details needed by other tools to integrate cores with none or little human interaction. This may be seen as an analogy to the way that software components are distributed in several GNU/Linux distributions using package repositories [6].

B. *Automatic system integration*

As depicted in Fig. 1, a system integration tool would be responsible from reading a design specification and producing a RTL HDL source and/or compiled design that can be fed to the regular vendor-specific platform in order to produce a final implementation. The design is mostly a high-level structural description of the system where very parameterized cores can be instantiated in many different configurations. The integration tool would be responsible of rendering the appropriate instances of the core by not only substituting design parameters, but also by fetching adequate core versions, executing available core generators, automating bus connections and so on.

C. *Core meta-information*

All the system pivots on the meta-information associated with the core, so that the integration tasks that can be automated will directly depend on the capabilities of the design that can be captured in the meta-information language. An obvious candidate for core description is IP-XACT, but it lacks the flexibility needed by the described scenario as already mentioned in the introduction of this contribution. Like CHREC, the proposed platform also aims to strong parametrization support, including not only parameters used by core generators or specific designer's scripts, but also the non-functional data required to support the package system.

III. XML Core Description Language

evercodeML (Exchangeable VERsatile COre DEscription Markup Language) is a XML-based core description language intended to support the special characteristics of the core integration system described in Section II. *evercodeML* includes basic functional description capabilities that are similar to those found in IP-XACT like component naming, port and bus description, parameter definition, etc., but it also includes some unique and/or improved facilities which are highlighted in the rest of this section.

A. *Javascript*

evercodeML can embed Javascript code like it is commonly found in XML world wide web applications. This is a powerful aid to support complex parametrization and other unique features of *evercodeML* that are detailed below. This functionality has been implemented by incorporating a Javascript interpreter into the *evercodeML* processor.

B. *Parametrization*

All the parameters from an underneath HDL description can be incorporated to *evercodeML*. In the case of Verilog, both *DEFINE* macros and in-module *parameter* definitions can be captured. In the case of *DEFINE* macros in Verilog, the same value is typically applied to all the instances of the affected modules. *evercodeML* automatically overcomes this limitation by replicating the module definition if a different set of parameters is used in two *evercodeML* instantiations when the underlying implementation of the parameters is a *DEFINE* macro. However, the use of Verilog's *parameter* is the preferred method to parametrize a SoC design.

Additionally, *evercodeML* can also define the so-called *master parameters* which live in the description's name space. Design parameters or other master parameters can be calculated from an equation involving other (master) parameters. Javascript is used to resolve the parameters so all the power of Javascript is available: mathematical expressions, control structures and so on. This makes it possible to handle

complex parameter relationships where many core parameters need to be calculated from a few master parameters even using some algorithmic resolution. Parameter value constrains can also be specified.

C. Algorithmic labels

Design labels like port names can be described algorithmically using Javascript control structures. It makes it possible to describe a core which interface topology varies depending on design parameters, like typically happens with cores produced by core generators. This way, the same description can be used even if the number or even the names of the input/output ports of the core changes. E.g. a core generator that generates a variable number *n* of address buses named *addr_1*, *addr_2*, … *addr_n* could be described with a label name of *addr_$(i)* inside a loop that makes it go from 1 to n.

D. Non-functional meta-data

evercodeML also supports a number of non-functional information which is not strictly related to the core's implementation but to various administration tasks like:

- Core's authorship and copyright information.
- Digital signature (planned): in order to authenticate the core's authorship.
- Version control and dependencies: a core may depend or use another core with some version restrictions.
- Name spaces: a core can be part of some absolute name space that identifies specific vendor, project, library, etc.
- URL (Universal Resource Locator): used to automatically fetch a core from a repository.

IV. EXAMPLES

Figure 2 shows an *evercodeML* description of a parameterizable floating point divider. The corresponding implementation calculates an approximation of the divisor inverse in the first stage. Then the inverse is calculated iteratively with the required precision using the Newton-Raphson method in the second stage. The quotient is calculated by multiplying the divined by the divisor inverse in the third stage. In the last stage the correctly rounded representation of the result is obtained. One of the parameters of the module is the maximum latency “max_latency”. Since the implementation has four stages this parameter must not be lower than four. This restriction is reflected in the description. The master parameter *max_iterations* will be the maximum number of iterations of the second stage and is calculated just by subtracting 3 to the *max_latency* parameters. The parameters *significand_width*, *exponent_width* and *max_iterations* of the *evercodeML* description of the module correspond to the underlaying HDL parameters *fractional_width*, *exponent_width* and *max_iterations* of the implementation respectively.

```
<parameters>

  <masterparameter datatype="integer()" id="significand_width"
         sourcename="fractional_width">
    <description>
         Number of bits used to code the significand
    </description>
    <inputvalue><defaultvalue>23</defaultvalue></inputvalue>
  </masterparameter>

  <masterparameter datatype="integer()" id="exponent_width"
         sourcename="exponent_width">
    <description>
         Number of bits used to code the exponent</description>
    <inputvalue><defaultvalue>8</defaultvalue></inputvalue>
  </masterparameter>

  <masterparameter datatype="integer()" id="max_latency">
    <description>An upper bound on the maximum number of clock
cycles that will be required to calculate the division</description>
    <inputvalue><defaultvalue>8</defaultvalue></inputvalue>
    <constraint
alert="A minumum of four cycles are required to compute a division">
         param.max_latency>=4
    </constraint>

  </masterparameter>
  <parameter datatype="integer()" id="max_iterations"
         sourcename="max_iterations">
    <value>
      param.max_latency-3
    </value>
  </masterparameter>

<parameters>
```

Figure 2. *evercodeML description of a floating point divider.*

Figure 3 shows an excerpt of a sample *evercodeML* description that corresponds to a Wishbone [7] interconnection matrix and includes some of the most noteworthy capabilities of the core description language. The matrix has a configurable number of slave devices. It is assumed that a matrix core generator exists that will generate a matrix core with the desired number of slave ports. The example includes five code blocks which are commented below:

- First block defines the *required_slaves* master parameter. Default value is 8 and this value can be changed at core's instantiation time. Note that “instantiation” here is considered at a high level so every instantiation of the core will trigger the matrix core generation process to produce a core with the desired number of slave ports.
- Second block defines the *address_width* master parameter corresponding to the desired with of the address bus. Default value is 32 and a constrain is defined to force the value to be a multiple of 8.
- Next two blocks define the *actual_slaves* and *decoding_width* parameters that corresponds to the *n_slaves* and *dec_width* parameter definitions in the underlying HDL code. This is an example of how a design parameters can be calculated from master parameters previously defined using a complex mathematical expression.
- Last block uses a *for* loop to describe the variable number of ports of the core as a function of the

number of slaves. Note how the decoding address and first address is automatically calculated for every slave port, and how these are linked to the actual design parameters that will be generated by the code generator.

```
<parameters>

  <masterparameter datatype="integer()" id="required_slaves">
    <description>Number of required slave slots</description>
    <inputvalue><defaultvalue>8</defaultvalue></inputvalue>
  </masterparameter>

  <masterparameter datatype="integer()"
    id="address_width" sourcename="aw">
    <description>Address bus width</description>
    <inputvalue><defaultvalue>32</defaultvalue></inputvalue>
    <constraint
      alert="Address bus width must be a multiple of 8">
      param.address_width%8 == 0
    </constraint>
  </masterparameter>

  <parameter datatype="integer()" id="actual_slaves"
    sourcename="n_slaves">
    <value>
      pow(2,(param.address_width-floor(
        log2(2^param.address_width/param.required_slaves))))
    </value>
  </masterparameter>

  <parameter datatype="integer()" id="decoding_width"
    sourcename="dec_width">
    <value>log2(param.required_slaves)</value>
  </parameter>

  <var name="i" />
    <for pre="i=0" test="lesser(i,param.actual_slaves)" post="i++">
    <parameter datatype="logic_vector(param.decoding_width-1,0)"
      id="slave_$(i)_decoding_address" sourcename="s$(i)_dec_addr">
      <value>i</value>
    </parameter>
    <parameter datatype=
      "logic_vector(param.address_width-param.decoding_width-1,0)"
      id="slave_$(i)_first_address" sourcename="s$(i)_first_addr">
      <value>
        pow(2,(param.address_width-param.decoding_width))*i
      </value>
    </parameter>
  </for>
  ...
<parameters>
```

Figure 3. *evercodeML description of a Wishbone interconnection matrix.*

This code not only describes the core independently of the variations that will be produced by changing the core parameters, but it also plays a fundamental roll at system integration time. When used manually, the designer has to calculate by hand the decoding and first addresses of every slave in the core as the function of the number of slaves and the bus width. But the *evercodeML* description gives the necessary information to a integration tool to be able to automatically calculate all these parameters so the designer only need to specify the number of slaves and the bus width when using the interconnection matrix from the *evercodeML* description.

Since the production of *evercodeML* descriptions can be very time-consuming, a tool has been developed to parse Verilog designs and generate an initial *evercodeML* description for all the modules found. This initial description can be then easily improved to use all the extended capabilities of the format.

V. CONCLUSIONS

Module integration in state-of-the-art SoC design is a tough task which is poorly automated, specially in the most heterogeneous environments. Current formal core description languages lack the parametrization flexibility and meta-information capabilities needed in the most complex scenarios. A novel high-level core description language has been introduced that overcomes these limitations by introducing advance capabilities like Javascript automation, algorithmic signal labeling, advance parametrization and repository management support.

ACKNOWLEDGMENT

This work has been partially supported by the Ministerio de Ciencia e Innovación of the Spanish Government under project TEC2011-27936 (HIPERSYS) and by the European Regional Development Found (ERDF).

REFERENCES

[1] System-on-Chip Designs: Stategy for Success. Wipro Technologies White paper (2001).

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

[3] Opencores, “Opencores website”. http://www.opencores.org.

[4] IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows. IEEE std. 1685-2009.

[5] A. Arnesen, N. Rollings, M. Wirthlin. A Multi-Layered XML Schema and Design Tool for Reusing and Integrating FPGA IP. Proc. FPG 2009, pp. 472-475. Sept. 2009.

[6] I. Murdock, “How package management changed everything”, July 2007. http://ianmurdock.com.

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