A Data-Path Oriented, IP-Based Framework for Flexible Design Exploration

C. Bolchini, C. Brandolese, W. Fornaciari, L. Frigerio, F. Salice
Politecnico di Milano – Milano, Italy

ABSTRACT

This paper presents a preliminary proposal for a design flow aimed at simplifying and improving the quality of the implementation of custom digital systems. The flow is based on a set of highly parametric VHDL core generators designed to enforce code readability and technology independence. The focus is on those portions of a design that are not usually covered by standard IP cores—buses, memories, etc.—and the related flow mainly targets digital signal processing or networking applications. A simple design example and the corresponding results are also presented and commented to better clarify the methodology.

1. INTRODUCTION

Designing with HDLs has been common practice for 15 years and has led to the development of efficient and reliable design flows and tools. With the increasing complexity of designs two major techniques have arose as valuable complement to classical design flow: high level synthesis and IP-based design. The former basically introduces a further, more abstract level of abstraction and the latter tends to change the level of granularity at which a design, or a portion of it, is described.

It is worth noting that while high-level synthesis has been studied for quite a long time and has not significantly helped in filling the productivity gap, the use of IPs (such as GPP or DSP cores, memories, USB or PCI buses and other complex cores) seems to be a more promising solution. Nevertheless, integration of complex blocks increases the complexity of system verification and does not completely solve the problem of customization.

The latter issue is particularly critical since standard blocks often cover only a subset of the user needs and in most cases are specifically designed for a particular target technology. Mostly due to commercial reasons, IP developers and vendors have concentrated their effort in developing cores whose functionality is clearly defined by either some standard (buses, protocols, encryption/decryption, etc.) or by very high-level mathematical formulations (FFT, DCT, filters, etc.).

This approach creates a gap between the very specific user application's needs and the available building blocks. In fact, whenever commercial IPs' functionality is too far from the user's requirements, designers tends to re-design the component from scratch since the customization of existing IPs requires long time spent in understanding their structure, in implementing the desired modifications and in verifying the new functionality. It is worth noting that in several cases the HDL source code of the IP is not even disclosed and thus is not modifiable at all. In such cases ad-hoc wrappers need to be developed to integrate the cores. Figure 1 shows the usual scenarios: in (a) the integration approach strongly relies on standardization and the involved IPs can be considered homogeneous; in (b) heterogeneous open-source IPs are integrated with designer's intervention on the core's source code; and, finally, in (c) heterogeneous black-box cores are integrated resorting to the implementation of wrappers.

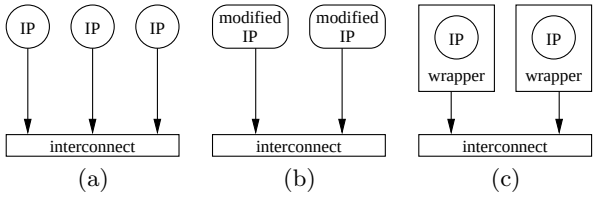

Figure 1: IPs integration approaches

According to these premises, the current IP-based methodology implicitly partitions designs into two clearly distinguished portions: a well defined set of very high-level standard blocks on one side and a portion of "sparse" logic implementing the custom functionality which constitutes the real added-value of the design, on the other side. The custom portion of the system is thus still designed according to a classical HDL-based flow and no integrated and flexible support is given to the design team.

The work presented in this paper outlines a preliminary proposal for an intermediate-level block-based design flow aimed at filling the gap discussed above. In the following, a brief summary of commercial solutions that partly face the outlined problem, also clarifying how this paper differs from the readily available approaches, is presented.

One of the key factors for the success of the IP-based design paradigm is that cores should be highly parametric and configurable. In-house development of parametric cores re-

quires more time, effort and expertise than the design of ad-hoc solutions, and for this reason—though enforcing internal reuse—it is rarely adopted. On the other hand, using commercial off-the-shelf IPs allows great flexibility but does not permit customization. IP reuse is a well-established solution in the EDA market, the most notable example being probably Synopsys DesignWare [5], a rich set of libraries that, combined with an efficient and flexible inferentiation mechanism, allows the development of rather complex systems with reasonable design effort.

A similar approach has also been followed by most of the FPGA vendors (Xilinx [6] and Altera [7] prominently): their design-entry, synthesis and back-end flows are profitably complemented by core generators for the most common functions, ranging from simple arithmetic components to complex memory and storage structures. The integration of various IPs, coming from different IP providers has been an active research field. Several solutions have been proposed, ranging from standard definitions of interfaces and buses (e.g. VSIA [12], IBM CoreConnect [9], ARM AMBA [10]) to techniques for wrappers generation (e.g. the TANGRAM [11] approach).

A further extension and improvement to such design flows and tools has been realized (Xilinx EDK, Altera Quartus and others) to cope with the complexity of FPGA-based SoCs such as Xilinx Virtex-II Pro or Altera Excalibur families. In these scenarios the overall architecture of the target platform is almost completely predefined (buses, component wrapping, component interfacing, etc.) and their nature is more suitable for system-level design rather than for custom logic development. It is worth noting that the major difference between such approaches and the one proposed in this paper is the strong link these design tools have with the target technology.

A more technology independent and very interesting commercial proposal is the Synplify toolchain [8] based on a Simulink design entry complemented by a custom (DSP) blockset, an architectural-level optimization engine (Synplify DSP), a set of code generators associated with the blockset and a classical RT synthesis flow (Synplify Pro). Though really valuable, this approach concentrates more on the analysis of the performance of a given implementation—the Simulink model—of a DSP problem, posing its focus on very high-level properties such as numerical convergence, quantization, parallelization and so on. In our opinion, this methodology operates with too coarse a grain to cope with the design challenges addressed by the approach proposed in this paper.

2. METHODOLOGY OVERVIEW

Core-based design is a promising way to deal with the increasing system design complexity. Exploiting the availability of pre-verified cores can greatly speed-up the design, as only an integration step need to be performed. However, this mechanism often leads to limited flexibility, especially if IPs are purchased from third-party vendors, because no visibility of the internal structure is usually allowed, and therefore no internal modifications are possible. IPs are not usually provided with the source code and therefore they can only be integrated as black boxes.

Internal development of IPs is also possible, and can guarantee a greater flexibility and control, even if an extra effort has to be planned. Furthermore, a framework for IPs development should be used. To develop a flexible and parametric core, the HDL features (for example the VHDL generics and generate statements, functions, etc.) are not always sufficient and often lead to poor code readability.

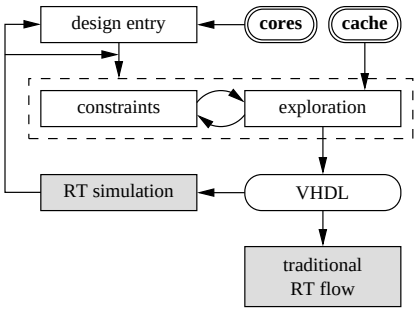

Figure 2: The proposed design methodology

The proposed methodology aims at providing a design framework to exploit both the advantages of IP-based design and design for reuse. The design of a complex data path can be performed with the integration of already preverified blocks that can also be accessed and modified by the designer. The development of blocks, both designed from scratch and defined starting from other cores is possible. Finally, the exploration of the solution space is also possible and a final VHDL model is then produced.

Figure 2 shows a representation of the proposed flow. The entry point is a high-level schematic that allows the designer to define and connect the blocks composing the system. After that, constraints can be set and an exploration phase is performed to determine the most suitable configurations. A VHDL model is available for every chosen configuration and can then be simulated. In principle, it could be possible and even advisable, to evaluate and simulate more than one design. Finally, after defining all the necessary details of the target technology, synthesis and back-end phases are performed according to traditional flows.

The framework is based on the RoadRunner (RR) toolset, composed by the following elements:

RRCore – a collection of parametric core generators;

RRCache – a database storing core characterization (e.g., area, critical path delay, latency, ...) integrated with a neural network based estimation engine used to extrapolate figures for uncharacterized modules.

Details of each phase composing the design flow are described in the following paragraphs.

2.1 System definition

System definition is realized through a high-level schematic entry, exploiting a collection of pre-verified, technology independent cores available in the RRCore. Each core is parametric and exposes several parameters. The designer must set all those parameters that will not be involved in the exploration phase while leaving unassigned all those parameters defining aspects of the modules that will be tuned during exploration. It is worth noting that some parameters are automatically inherited by cascaded modules. For example, the setting of the output width of a module chained with another one, causes the latter to have the input width fixed, and therefore the parameter "width" is defined at top level

and can be propagated when the modules are connected to each other. Not only the available cores can be used to define a system, but it is likely that some portions of the design will need to be customized. The designer has different strategies to define the parts that should be not generated through RRCore. First of all, as the VHDL code of the library cores is available and human-readable, every modification can be performed both in a parametric way (by modifying the generator) and in a fixed way (by editing the generated VHDL code).

The library can be updated not only by modifying existing cores, but also by adding new blocks either defined from scratch or combining existing cores into a more complex module. The proposed methodology gives the necessary support to perform this step, through a set of support C++ classes that helps the generators' development.

Finally, if some components are specific for the application under development and are not expected to be reused in other designs, or if the development of a parametric core requires too much effort, the designer can integrate VHDL code directly in the system.

The schematic entry allows to graphically identify the topology of the system, to define how ports are connected and how modules work together. At the end of the design phase, a model of the system is available. This model is not completely defined as some parameters are left unspecified for the exploration phase. In other words, the functionality is fully specified while the definition of the architecture of individual cores is delayed to a later design phase.

2.2 Constraints Setting

After the design phase is completed the designer usually may want to define some constraints in order to guide the exploration phase. Different types of constraints are available according to the information stored in the RRCache for each module, area and delay constraints being the most common. Each module of the library usually has different architectures, each one suited to satisfy specific constraints. For example, a CORDIC module might be realized through an iterative solution, suitable if area occupation has to be minimized, but less performing with respect to execution time. Alternatively, a pipelined solution requires more area but is usually faster. The designer can impose constraints both on single modules, and on the top level design, thus guiding the exploration phase with the desired degree of detail.

2.3 Space Analysis

During design space analysis, different solutions for the design are analyzed and compared to meet the user constraints and the most suitable solutions are proposed. The designer has the possibility to change the parameters in order to modify the proposed solutions. Modification of the parameters at top level leads to changes in all of the modules that are sensible to these parameters. In this way, for example, the bit-width of certain signals can easily be modified and propagated to the involved modules. The RRCache database, where cost figures of the modules are stored, is used to support choices during exploration. Whenever the required data are missing, the estimation engine is run and an estimate is generated.

2.4 Simulation, Synthesis and back-end

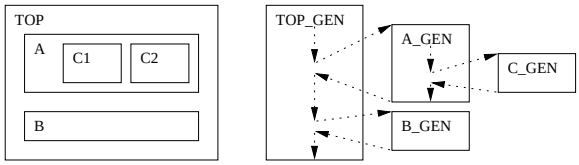

Figure 3: Modules and generators hierarchy

The simulation is performed to verify adherence of the system to the functional requirements. To perform this step an external simulation engine is used (e.g., Mentor Graphics Modelsim). If the solution is not correct or does not meet some constraints, another iteration needs to be carried out.

Finally, synthesis and back-end are performed with commercial tools according to traditional FPGA or ASIC flows.

3. PARAMETRIC CORES

In this section, the features of generic IP cores constituting the base of the methodology are detailed by outlining their structure and parameters and by describing the fundamental ideas behind the exploration phase.

3.1 Atoms and Molecules

Modules in the RoadRunner library can be classified according to their internal hierarchy. Very simple modules exist whose internal structure need not to be split over two or more levels of hierarchy. Such modules correspond to a purely RT VHDL description requiring no components at all. We will call such modules *Atomic IP*s or, for short, *atoms*. As all the modules of the RoadRunner library, the structure and functionality of atoms may depend on a certain number of parameters.

The structure of the RoadRunner library allows easy integration of already designed modules into more complex cores, with no limits, in principle, to this nesting mechanism. We will refer to composite hierarchical modules as *Molecular IPs* or *molecules*. It is worth noting that since modules are always generated, hierarchy is obtained by combining the generators and not the modules themselves. This situation is depicted graphically in Figure 3, where the left diagram shows the hardware (VHDL) hierarchy of a molecule and the right side shows the structure of the generator with emphasis on the calls to external generators for each submodule.

Similarly to atoms, molecules also are characterized by parameters, but molecules have an additional property: *opacity*. The degree of opacity of a molecule changes the way its structure is visible from the outside. A perfectly transparent molecule exposes all its constituents (atoms or other molecules) at all levels of hierarchy. The exposing of the hierarchy means that the parameters of each submodule are inherited by the top-level module and can be tuned by the exploration mechanism. At the opposite side of the spectrum there are perfectly opaque molecules, i.e., modules exposing only those parameters inherently pertaining to the top-level architecture. With respect to parameters, a perfectly opaque molecule is identical to an atom. In between these two limiting cases there are partially opaque molecules, i.e., molecules exposing the parameters of submodules down to a given level of hierarchy.

As an example, let us consider an FIR filter: it is built

with multipliers and adders at the first level of hierarchy and, in turn, multipliers are built with adders at a second level of hierarchy. Tuning the opacity of such a molecule means exposing the parameters of submodules at the different levels, as summarized below:

opaque – number of taps, filter weights, word width, active clock edge, active reset level;

semitransparent – all previous parameters plus the architecture of the multipliers (e.g., Wallace or Dadda) and architecture of the adders used in the taps (e.g., ripple-carry or carry look-ahead);

transparent – all previous parameters plus the architecture of the adders used to build multipliers.

It is worth noting that the opacity is itself a parameter of a molecule but has the peculiarity of modifying the "interface" of the corresponding generator. This issue is further discussed in Section 3.3.

3.2 Parameters

Both atoms and molecules are characterized by a set of parameters that make them flexible and customizable. Mutuating the terms from object oriented programming, it can be said that an atom or a molecule, i.e., its corresponding generator, is a *class* while an instance of an atom or a molecule is an *object*. According to this definition the difference between a class and an object is that in the latter all parameters have been assigned a specific value.

When structuring the architecture of a circuit, the designer must instantiate atoms and molecules, must connect them and must assign values to their parameters. If all parameters of all modules are fixed, no design space exploration is possible and the design is complete. On the other hand, the designer usually makes some choices at design-time and leaves other parameters unassigned for automatic optimization.

To clarify the exploration process and, namely, to define the design space the following definitions are necessary.

Definition 1. *A parameter can be either* explorable *or* non-explorable. *Explorable parameters can (but not necessarily must) be left unassigned in the design phase and automatically determined during exploration in such a way to optimize a predefined figure of merit. Non-explorable parameters must be assigned at design time. Furthermore, explorable parameters may only have a finite (usually small) number of different values.*

For example explorable parameters are the internal precision of a floating-point adder, the bit-width of a multiplier or the architecture of an adder (chosen between a set of predefined alternatives) while non-explorable parameters are the coefficients of a FIR filter, the active front of a clock signal or the number of inputs of a multiplexer.

Definition 2. *The value of a parameter can be either* fixed *or* variable. *A fixed value is assigned explicitly by the designer while a variable parameter is left free for exploration.*

An additional definition is necessary to completely define the concept of value in this context.

			PARAMETER	
			explorable	non-explorable
VALUE	fixed	module	×	×
		top-level	×	×
	variable	module	×	
		top-level	×	

Table 1: Parameters and values properties

Definition 3. *A value can be defined at* module level *or at* top-level. *It is defined at module level when its value is assigned explicitly to the corresponding parameter, whereas it is defined at top-level when it is assigned indirectly through the definition of a new system-wide explorable parameter.*

While Definitions 1 and 2 are rather straightforward, Definition 3 deserves more attention and its meaning is better clarified by the following example. In a synchronous design it is reasonable that the reset signal of all bistables is active on the same level (high or low): this suggests the definition of a new, top-level parameter indicating the active reset level and to associate this parameter with the value of all submodule's reset active level. The reset active level of a module is in this case a fixed, top-level value associated with a non-explorable parameter. Similarly, the number of pipeline stages of two modules connected in parallel should reasonably be the same but the designer might want to investigate different choices. To do this, the designer defines a top-level parameter indicating the number of stages and assigns it as the value of the parameters of the two modules. Such a new parameter is explorable and has a variable, top-level value.

The relations between parameters and values properties and all allowed combinations are summarized in Table 1.

The opacity of a module is currently defined as non-explorable, since the preliminary optimization algorithm used for design space exploration cannot treat it conveniently. Nevertheless there is no theoretical reason to allow this parameter to be explorable. As a rule of thumb, it can be stated that making a molecule opaque limits the design space reducing the optimization run-times but may lead to unsatisfactory results while making the molecule perfectly transparent greatly widens the design space and allows finer tuning of the overall architecture, opening new optimization opportunities.

3.3 Exploration

A complete design is composed of modules (atoms or molecules) and a number of module-level and top-level parameters. Some of the parameters are non-explorable or fixed, while others are explorable and variable. The set of the latter defines the dimensionality of the design space. Since, according to Definition 1, explorable parameters can only range over a finite set of values, the design space is guaranteed to be finite.

The process of design space exploration is basically an efficient analysis of all possible assignments of values to parameters under a given set of constraints. Typical constraints are expressed as maximum clock period, maximum combinatorial delay, maximum average power dissipation or, mostly for FPGAs, maximum area. To determine constraint violation and to assign an overall assessment of a tentative solution the exploration relies on area, timing and power figures

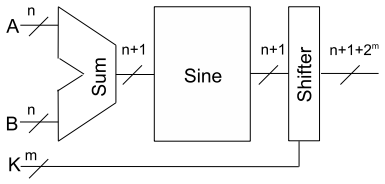

Figure 4: System functionality

either stored in a database or interpolated from a set of available values in the database. This evaluation, of course, requires that modules are pre-characterized and it can be regarded as a limitation of the proposed approach. For this reason the database is accessed through an intelligent tool, the RRCache, that, each time a module (atom or molecule) is actually used with a new set of parameters for a design, stores the relevant figures into the database and simultaneously re-trains a predictor based on a neural paradigm. This strategy does not eliminate the drawbacks related to pre-characterization but significantly improves the usability of the proposed approach.

4. EXPERIMENTAL RESULTS

To clarify the proposed methodology, let us consider the realization of a simple data-path, computing the function $f(A, B, K) = \sin(A + B) \cdot 2^K$. As entry point, the designer defines the system functionality through a schematic and fixes the system topology as in Figure 4.

The design is composed of three functional blocks: an adder, a block for the computation of the sine function and a shifter to multiply the result by 2^K. Three input values are defined (A, B, K), and their bit-widths are considered as system parameters. Widths of input values influence the dimension of the internal signals and of the output signal. Let `top::n` be the width of A and B and `top::m` the width of K; this implies that the width of the adder output and the sine output are `top::n+1` (the same precision is maintained), while the width of the final output is $\texttt{top::n+1+2}^{\texttt{top::m}}$. The prefix `top::` indicates that these values are not fixed for individual modules but are rather defined at top level.

Every block can be implemented in different ways according to the available architectures in RRCore. The block performing the sum can be implemented using either a carry look-ahead adder or a ripple-carry adder; to perform the sine computation a LUT-based or a CORDIC module, either pipelined or iterative, can be used; finally, the shifter can be realized with an arithmetic or a logarithmic architecture.

Tables 2 and 3 show the values saved in RRCache for area and timing optimization for the available modules with `top::n=8` and `top::m=3`. Two technologies are shown: LSI Logic LSI_10k at 0.5μm and Virtex-II Pro Xilinx FPGA fg456, speed grade -7.

The design space analysis is performed through a simple exploration algorithm, that takes as input the area and time figures of each module (as shown in Tables 2 and 3) and possibly other characteristics available in RRCache. In particular, adders, shifters and LUT-based sine calculation module are combinatorial, therefore they are fully characterized by the delay of the critical path, while CORDIC is a sequential module, and therefore information about throughput and latency are necessary for its complete characterization. The iterative architecture of CORDIC requires 9 clock cycles to compute the output data (one clock cycle per input bit), while the pipelined architecture provides a result at every clock cycle, after an initial latency of 9 cycles.

	LSI_10k (mils) Optimization for		Virtex-II Pro(LUT) Optimization for	
	Area	Time	Area	Time
Adder-cla	81	273	9	9
Adder-rc	80	443	9	9
Cordic-it	1565	1632	107	116
Cordic-pl	6992	7146	179	184
LUT-sine	732	2020	119	119
Shifter-arith	143	376	27	27
Shifter-log	131	265	27	27

Table 2: Synthesis results for area occupation

	LSI_10k (ns) Optimization for		Virtex-II Pro (ns) Optimization for	
	Area	Time	Area	Time
Adder-cla	11.75	4.62	9.87	9.87
Adder-rc	14.71	4.89	9.87	9.87
Cordic-it	3.30	1.81	6.62	5.78
Cordic-pl	2.72	0.97	3.30	3.22
LUT-sine	11.82	5.11	6.74	6.74
Shifter-arith	7.08	2.51	6.36	6.36
Shifter-log	6.55	2.29	6.36	6.36

Table 3: Synthesis results for path delay

The design has been evaluated for three different types of constraints: area optimization, time optimization and area/timing tradeoff.

Area optimization – If area optimization is required, and no—or very loose—timing constraints are specified, the smallest blocks are chosen for each functionality. In this case the proposed solution is dependent on the chosen technology: a ripple-carry adder, a logarithmic shifter, and an iterative CORDIC are chosen if the target technology is an FPGA, while if the deign is targeted to ASIC, a LUT is preferred for the sine computation. Results are shown in table 4.

Module	FPGA Tech.	Asic Tech.
Adder	Adder-RC	Adder-RC
Sine Comp.	Cordic-it	LUT
Shifter	Shifter-log	Shifter-log

Table 4: Solution for area optimization

Area/timing tradeoff – The previous solution can be modified adding a time constraint to the sine module. In this case a tradeoff between area and timing is required, and therefore for the FPGA technology also the LUT turns out to be preferred to CORDIC, as shown in Table 5.

Module	FPGA and Asic Tech.
Adder	Adder-RC
Sine Comp.	LUT
Shifter	Shifter-log

Table 5: Solution for area/timing tradeoff

Timing optimization – If timing optimization is required, and no area constraints are specified, the fastest blocks are selected for each functionality. Solutions vary according to the nature of the time constraint. If maximum throughput is required the solution is composed of a carry-look-ahead adder, a pipelined CORDIC for the sine computation and a logarithmic shifter, while if minimum delay is the goal, then a full combinatorial solution is proposed, therefore the pipelined CORDIC is replaced by a LUT, as shown in Table 6.

Module	Max throughput	Min delay
Adder	Adder-CLA	Adder-CLA
Sine Comp.	Cordic-pl	LUT
Shifter	Shifter-log	Shifter-log

Table 6: Solution for time optimization

It is worth noting that, in all cases, the designer can easily evaluate the results for different values of input widths by simply changing the value of such parameters in the top module.

5. CONCLUSIONS

This paper presented the fundamental ideas of a new methodology for the design of complex data-paths. In the authors' opinion the major novelties proposed by this work are:

1. A rich library of module (VHDL code) generators ranging in complexity from very simple ones (e.g., adders, shifters, muxes, ...) to rather complex ones (e.g., FFT, DCT, CORDIC, ...);

2. Modules are highly parametric and are designed in such a way that almost only the functionality needs to be specified allowing thus to exploiting all architectural details as degrees of freedom for a subsequent design space exploration;

3. The generated VHDL code is very clear and nicely formatted to improve readability both for self documentation purposes and for a better maintainability and/or customizability;

4. The exploration algorithm (currently a rather generic one) strongly relies on figures either derived from pre-characterization of the modules (stored into a growable database) or from a neural network based, adaptive estimation engine.

The example presented, yet very simple, has shown the application of the methodology to a data-path involving combinatorial, pipelined and iterative modules and has demonstrated the key ideas behind this proposal.

The current design flow relies on very prototypical implementations of the two modules RRCore and RRCache. The RRCore library contains more than 60 parametric modules, tested and pre-characterized for the two technologies mentioned in the paper. Although the implementation is preliminary, most of the methodological and formal aspects have already been defined and thus most of the future effort will be directed to implement new generators and to improve the two RoadRunner components.

6. REFERENCES

[1] C. Barna, W. Rosenstiel, "Object-Oriented Reuse Methodology for VHDL," *Proceeedings of IEEE Design, Automation and Test in Europe,* DATE'99, Munich, Germany, March 1999.

[2] P. Schaumont et al., "Hardware Reuse at the Behavioral Level," *Proceedings IEEE/ACM Design Automation Conference,* DAC'99, New Orleans, USA, June 1999.

[3] D.D. Gajski et al., "Essential issues for IP reuse," *Proceedings of ASP-DAC'2000* pp. 37–42, 2000.

[4] M. Vaupel, T. Grotker, H. Meyr, "Combox: library-based generation of VHDL modules," *Proceedings of the IX Workshop on VLSI Signal Processing*, pp. 293–302, 1996.

[5] http://www.synopsys.com/products/designware/

[6] http://www.xilinx.com/

[7] http://www.altera.com/

[8] http://www.synplicity.com/

[9] "IBM CoreConnect Bus Architecture," http://www-3.ibm.-com/chips/ products/coreconnect/index.html

[10] "ARM AMBA," http://www.arm.com/

[11] U.R.F. Souza, J.K. Sperb, B.A. Mello, F.R. Wagner, "Tangram Virtual Integration of HeterogeneousIP Components in a Distributed Co-Simulation Environment," *Proceedings of the 16th Symposium on IntegratedCircuits and Systems Design,* SBCCI'03, Sao Paulo, Brazil, September 2003.

[12] "Virtual Socket Interface Alliance," http://www.vsi.org