Core Communication Interface for FPGAs

José Carlos Palma, Aline Vieira de Mello, Leandro Möller, Fernando Moraes, Ney Calazans

Pontifícia Universidade Católica do Rio Grande do Sul (FACIN-PUCRS)
Av. Ipiranga, 6681 - Prédio 30 / BLOCO 4 - 90619-900 - Porto Alegre – RS – BRASIL
{ jpalma, alinev, moller, moraes, calazans }@inf.pucrs.br

Abstract

The use of pre-designed and pre-verified hardware modules, also called IP cores, is an important part of the effort to design and implement complex systems. However, many aspects of IP core manipulation are still to be developed. This paper presents an approach to solve problems related to the dynamic interconnection of hard IP cores. The approach targets system-on-a-chip designs build in a single FPGA device. The paper proposes a communication interface that allows IP cores replacement during the FPGA normal operation. The same interface also allows the communication among distinct IP cores to take place.

1. Introduction

Cores are complex pre-designed and pre-verified hardware modules, which can be considered as key component in the development of system-on-a-chip (SOC) designs. According to a recent ITRS report [1], by 2012 90% of the integrated circuits area will be implemented using cores. Table 1 shows the main features of cores in general, according to a well-established classification found in the literature [2][3].

Table 1 - Core classification.

Classification Criteria	**Hard core**	**Firm core**	**Soft core**
Structure	Pre-defined organization	Source code and netlist, technology independent.	Behavioral source code, technology independent.
Modeling	Modeled as a library component.	Synthesizable logic blocks.	Synthesizable with several technologies.
Flexibility	Cannot be modified by the designer.	Possibility to customize some parameters.	The user can modify the design.
Timing Closure	Timing ensured.	Critical paths have timing fixed by constraints.	Timing not guaranteed.
Intellectual Property Protection	Strong. Usually corresponds to a layout.	Average	Weak. Source code.
Example	FPGA Bitstream, CIF or GDS2 file for IC layout.	EDIF	VHDL, VERILOG

Cores are used (or integrated) during system design, i.e., before logical or physical synthesis. For ASIC designs, this corresponds to the normal flow. However, for FPGAs a different scenario can be envisaged: dynamic hard-core insertion and removing at run time. This idea allows inserting a hardware module into the FPGA according to the application requirements.

For instance, considering the situation where a FPGA board is connected to a host computer and a new piece of hardware may be necessary to speed up the processing in a graphic application. This piece of hardware (hard-core) could be downloaded through the Internet and sent to the FPGA.

Another example could be the "dynamic co-design". The hardware generated by a co-design tool requires a FPGA with sufficient area to implement all the application modules. Using the proposed approach (hard-core reuse), the co-design tool could generate small modules, loading them according to a timing schedule. This approach is similar to the "virtual memory" concept, found in traditional operating systems and could be named "virtual hardware" [4].

To achieve hard-core reuse some requirements have to be fulfilled: (i) FPGA allowing partial and dynamic reconfiguration; (ii) floorplanning tool to fix the core position; (iii) software for partial bitstream generation; (iv) software for partial bitstream download; (v) a core communication interface; (vi) input/output pin virtualization. Requirements (i) to (iii) can be satisfied with existing devices and tools. Works fulfilling requirements (iv) to (vi), in our knowledge, are very scarce or simply do not exist.

The core communication interface is a critical problem, since it must allow dynamic insertion and removal of cores without system interruption. This interface should be a fixed module in the FPGA, with three main functions: bus arbitration, communication between modules, and pin input/output virtualization.

This paper is organized as follows. Section 2 presents the requirements for hard-core reuse. Section 3 details the main contribution of this work, the communication interface. Preliminary results are presented in Section 4. Section 5 presents some conclusions and directions for future work.

2. Requirements for dynamic hard cores reuse

Reconfigurable computing mostly emphasizes the use of coarse grain reconfigurable arrays with path widths greater than 1 bit, like in fine-grained architectures. Fine-grained architectures, like commercial FPGAs, are much less efficient with a huge routing area overhead and poor routability [5].

Proceedings of the 15 th Symposium on Integrated Circuits and Systems Design (SBCCI'02)
0-7695-1807-9/02 $17.00 © 2002 IEEE

In spite of this fact, our goal is to investigate the feasibility of implementing reconfigurable computing systems into commercial FPGAs. The granularity of our approach is an important point, and it can be classified as coarse-grain, since the reconfiguration is achieved with hard cores.

Next sections detail the requirements discussed in the Introduction.

2.1. FPGA architecture

Examples of available FPGA devices allowing partial and dynamic reconfiguration are Xilinx Virtex [6] and Atmel At40k [7]. Xilinx Virtex FPGAs have been selected as the target devices for this work, because of facilities such as satisfactory CAD tools, sufficient gate counting at affordable costs, and diversity of prototyping boards available.

The main components of Virtex devices are [6]: Configurable Logic Blocks (CLBs), Input/Output Blocks (IOBs), RAM blocks, clock resources and programmable routing. Each CLB has two slices, named 'S0' and 'S1'. Each slice contains two LUTs, named 'F' and 'G', two flip-flops and carry resources. Each CLB has also local routing resources, connected to the general routing matrix (GRM). A peripheral routing ring, called VersaRing, allows additional routing to I/O blocks (IOBs). This FPGA has dedicated RAM memory blocks (BRAMs), each one with 4096 bits, and 4 to 8 DLL circuits for clock distribution and skew minimization. Figure 1 shows an abstraction of the Virtex FPGA internal architecture.

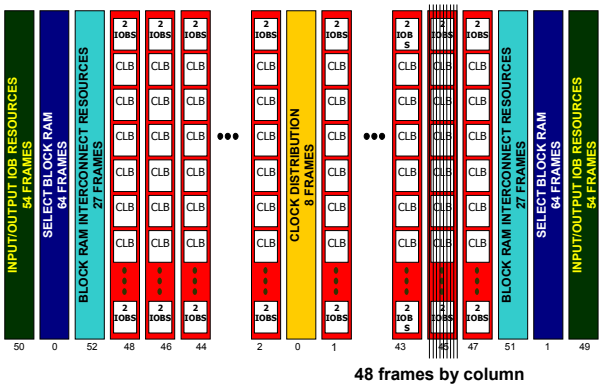

Figure 1- Virtex FPGA internal architecture (XCV300).

The Virtex configuration memory is organized as a bi-dimensional array of bits [8]. A single column of bits is named frame. One frame corresponds to one atomic unit for reconfiguration, i.e., the smallest portion that can be read from (or written to) the configuration memory. Sets of consecutive frames compose CLB, Select Block RAM, IOB and Clock columns. As can be observed in Figure 1, 48 frames configure a CLB column.

The Virtex device can be partially reconfigured since frames can be read or written individually. Note that it is not possible to configure a single CLB, since the frames belonging to a given CLB are common to all other CLBs in the same column. Therefore, if a modification to a single CLB is required, all frames belonging to the same column must be read (operation called read-back), and the modification inserted over the read frames. In practice, this feature renders the structure of the VIRTEX device a single-dimension array of columns.

2.2. Floorplanner

Partial reconfiguration is only possible if the cores to be inserted do not overlap the existing cores inside the FPGA. So, placement restrictions must be inserted during the physical synthesis.

Example of such tool is the Xilinx floorplanner [9], which is a graphical tool that allows the designer to control core position and core shape in a FPGA using "drag and drop" facilities. A browser displaying the design hierarchy allows the designer to restrict the position of each core component. Detailed placement and routing are executed after the floorplanning.

The Xilinx floorplanning tool restraints logic blocks, memory blocks, I/O blocks and tri-state buffers. This tool does not restrain detailed routing. As a result, after detailed routing some wires can be implemented outside the limited area, overlapping the area reserved for other cores. Several iterations between floorplanning and routing, and even manual user operations in the routing tool are required to solve this problem.

2.3. Partial bitstream generation

Tools provided by FPGA vendors do not generate partial bitstreams. JBits classes [10] can be used as an abstraction layer hidden the Virtex FPGA configuration details.

Without JBits the tool developer has to deal with a set of details, such as frame, slice and CLB addressing [8]. JBits operates with bitstreams generated by Xilinx tools as well as bitstreams read from the hardware (read back operation).

Figure 2 shows examples of methods to read and write to LUTs.

WRITTING VALUES IN LUTS

```
SLICE 0 F LUT:  jbits.set(row, column, LUT.SLICE0_F, value)
SLICE 0 G LUT:  jbits.set(row, column, LUT.SLICE0_G, value)
SLICE 1 F LUT:  jbits.set(row, column, LUT.SLICE1_F, value)
SLICE 1 G LUT:  jbits.set(row, column, LUT.SLICE1_G, value)
```

READING VALUES FROM LUTS

```
SLICE0 FLUT:   int[] returnVal = jbits.get(row, column, LUT.SLICE0_F)
SLICE0 GLUT:   int[] returnVal = jbits.get(row, column, LUT.SLICE0_G)
SLICE0 FLUT:   int[] returnVal = jbits.get(row, column, LUT.SLICE0_F)
SLICE0 GLUT:   int[] returnVal = jbits.get(row, column, LUT.SLICE1_G)
```

Figure 2 - Example of JBits methods.

The *value* parameter is a 16-bit string specifying the function implemented by the LUT. For partial bitstream generation, complete columns are extracted from a complete bitstream, and written to a new file. This is not a simple task, since all configuration registers values (presented in the bitstream) have to be recalculated, new bitstream CRCs must be generated, and columns do not have continuous address (even frames are placed in the left part of the FPGA and odd frames in the right side).

2.4. Communication interface

In the early stages of SOC development, cores are designed with different interfaces and communication protocols. Some standard interfaces, such as Wishbone [11], AMBA [12] and CoreConnect, were created to increase core reuse. These interfaces are used during the design phase of the SOC, for ASIC and FPGA flows. After the SOC implementation, it is not possible to connect a new core to the system.

As our goal is to dynamically insert cores into an operating FPGA, a different mechanism must be created. This communication interface is detailed in Section 3.

2.5. I/O Pins virtualization

I/O pins virtualization means that cores exchange data only with the communication interface and the real input/output communication is a function of the interface.

3. Communication interface

The basic idea is outlined in Figure 3. A fixed module, named *controller*, is initially downloaded into the FPGA. Functional cores, named *slave cores*, are downloaded at run time.

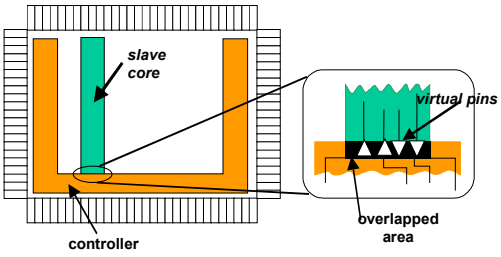

Figure 3 - Communication interface with one tri-state buffer layer (tri-state control signals are not show).

The controller is responsible for the communication with the external world (I/O pins of the device) and for the communication with slave cores. In other words, slave cores communicate with the external world only through the controller (I/O pins virtualization). Each slave core communicates with the controller through *virtual pins*. Virtual pins are in fact tri-state buffers, having the position defined by the floorplanning tool. The connection between slave cores and the controller is done by overlapping virtual pins.

The interface proposed in Figure 3 is not feasible due to restrictions imposed by the FPGA architecture. Such restrictions are:

- Input and output of tri-state buffers could be used as virtual pins by slave cores. As it is not possible to fix wire positions, the controller and slave cores must use a tri-state buffer as virtual pin.
- Each CLB contains LUTs, flip-flops, tri-state buffers and routing resources. It is not possible to restraint the CLB usage only to tri-state buffers and routing. Therefore, overlapping slave cores over the controller could destroy some controller functions.
- Each CLB contains only 2 tri-state buffers. The reduced number of buffers limits the bus width. In addition, the buffer routing wires share common hex lines [6], restricting the routing tool. Due to this limitation, we adopted a serial bus (1-bit data line).

In order to overcome these limitations, a communication interface with two tri-state buffer layers with common routing wires is implemented. Figure 4 shows this solution. One buffer layer belongs to the controller and one buffer layer belongs to the slave cores.

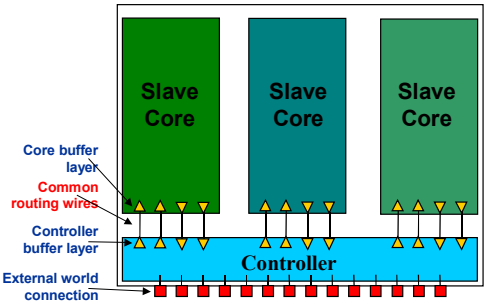

Figure 4 - Communication interface with two tri-state buffer layers and a common routing wire.

In order to have common routing wires the controller is synthesized by using “dummy cores”, which have the buffers belonging to the slave cores. The same procedure is applied to slave cores, which are synthesized with a “dummy controller”. “Dummy cores” are also important to avoid floating signals in the communication interface.

3.1. Controller implementation

The controller contains three modules, illustrated in Figure 5:

- *Communication bus*, connecting the slave cores;
- *Arbiter*, granting the data line to a given slave core;
- *Master core*, responsible for the communication to the external world.

The communication bus has the following signals: reset (global), clock (global), individual request and grant lines for each slave core, and a bi-directional data-line (global). Each slave core communicates with the bus through six tri-state buffers, connected to the following signals: clock, reset, request, grant, *datain* and *dataout*.

Datain and *dataout* are connected together to the data line. The 1-bit serial data line transports a 40-bit word, containing an 8-bit core address and a 32-bit data word. A simple protocol is employed, using a starting bit to indicate the transmission of a new word.

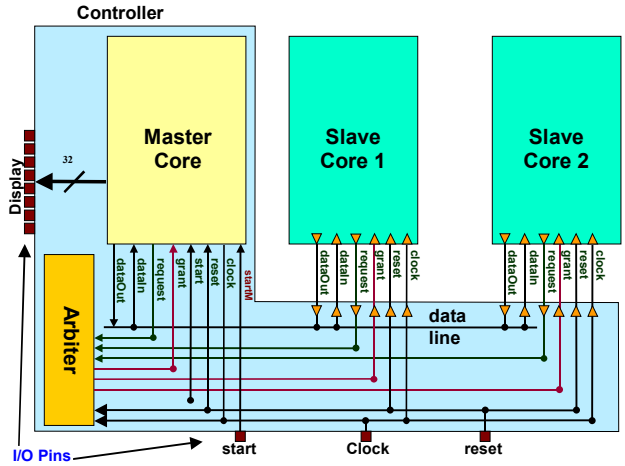

Figure 5 - Controller modules: master core, arbiter and communication bus.

The arbiter continuously reads the request lines, in a serial fashion. When a request line is active the arbiter grant the data line to the slave core requesting data for 40 clock cycles. After these 40 clock cycles, the arbiter starts reading the request line of the next core. In this way, all cores are served, even cores with lower priority.

The master core is implemented according to the functions and input/outputs requirements of the slave cores.

3.2. Send and Receive Modules

Slave cores contain the hardware implementing the function specified by the user (hw-core), plus send and receive modules. This structure is illustrated in Figure 6.

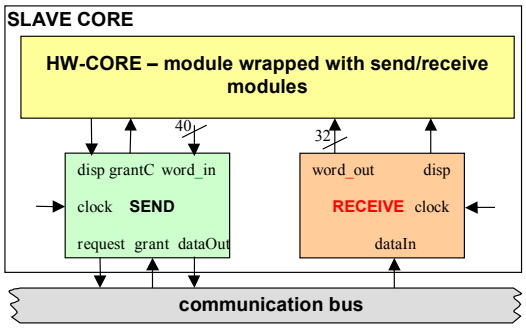

Figure 6 - Send and receive modules.

When the hw-core wants to send data to another core, it activates de 'disp' line and puts a 40-bit word in the 'word_in' port. The send module activates the 'request' signal, and waits the 'grant' signal. Once the 'grant' signal is asserted, the available 40-bit word is stored by the send module, and put serially into the dataOut port. After 40 clock cycles the 'grantC' is asserted, signaling to the hw-core the send module is ready to send a new word. No buffer mechanism neither time-out control was implemented in this first version of the send module.

The receive module constantly reads the data line. The 8 bits following the start bit are compared against the address of the receive module (unique for each slave core). If the address does not match the module address, the remaining 32 bits are ignored. If the data is addressed to the slave core, the receive module sends the data to the hw-core 32 clock cycles after address matching, asserting the 'disp' line during one clock cycle.

3.3. Case Study

A simple case study was implemented to validate the communication interface. Tree modules compose this application, as illustrated in Figure 5: (1) controller (2) slave1; (3) slave2.

The function of each slave core is very simple: reception of two 32-bit words, execution of an arithmetic operation over these words (*slave1* executes addition and *slave2* subtraction), sends the result to the master core, waits two new words.

The master core has three internal registers, acting as data memory, and a program memory holding the operations. The program memory is implemented in LUTRAM, being its contents defined by specialized reconfiguration tools. The actions specified in the program memory can be register initialization or arithmetic operation between two registers. The target register (T) is always the same. Example of operations stored in LUTRAM can be A=8, B=7, T=A+B, T=T+A, etc. A special code is used to indicate the last operation. When an arithmetic operation is found, the arithmetic operation is translated to the slave core address. The master core merges this address to the first register value, sending the 40-bit package to the communication bus. The same procedure is done for the second operator. Once the two operators are sent, the master core waits the response from the target core, storing the received value into the 'T' register. The result of each operation is sent to the external world, and shown in an 8-digit display.

4. Preliminary results

This section presents preliminary results: functional communication interface validation, partial bitstream generation, and CAD tool for core manipulation.

4.1. Functional validation

All modules are described in VHDL language, and simulated using the Active-HDL simulator, from Aldec [13]. Figure 7 illustrates the system operation, using the case study presented in section 3.3. The numbers highlighted in Figure 7 are explained below:

1. Initially, the data line is in steady state (logic state 'HIGH').
2. The master core asserts its internal send signal (connect to 'disp' in Figure 6), indicating it has available data to transfer.
3. The master's send module asserts the request signal, and waits for the grant signal.
4. The grant signal is sent to the master's send module.
5. Data transmission starts, and the first '0' correspond to the start bit.
6. Receive modules detect the start bit, starting data storage.
7. After 8 clock cycles, all receive modules compares the stored byte to the internal address. If they match, the receive module continues to store the arriving bits. If they do not match, the receive module waits 32 clock cycles to read again the data line.
8. 40 clock cycles after the starting bit a new 32-bit word is available.
9. The slave1 receive module signs an available word to the hw-core.
10. At the end of the first transmission, the master's send signal is still in logic '1', indicating that it wants to send the second operator. A new request is sent to the arbiter and operations 4-9 are repeated.

Events 13-14 indicate data transmission from slave1 to the master core (result of the arithmetic operation). The master core, in event 17, receives the sent data.

Once the system is functionally validated by simulation, the next step is to prototype it. The prototyping board *Virtual Workbench* [14], with a Virtex XCV300, was used. In our experiment the communication bus has three "sockets" (set of tri-state buffers to slave core connections), and the application has two slave cores.

Two bitstreams were created. The first bitstream contains the controller and two slave cores. Floating signals induced by unused tri-state buffers in the third socket made the system instable.

In the second bitstream, a "dummy core" was connected to the empty socket to avoid floating signals, mainly in the *request* and *dataOut* lines.

As several independent modules compose this system, it is important a careful design of the FSMs initialization and high-impedance value in the data line must be avoided.

The system works correctly in the prototyping board, showing that the communication interface, with tri-state buffers, can be used in Virtex devices to connect hard cores.

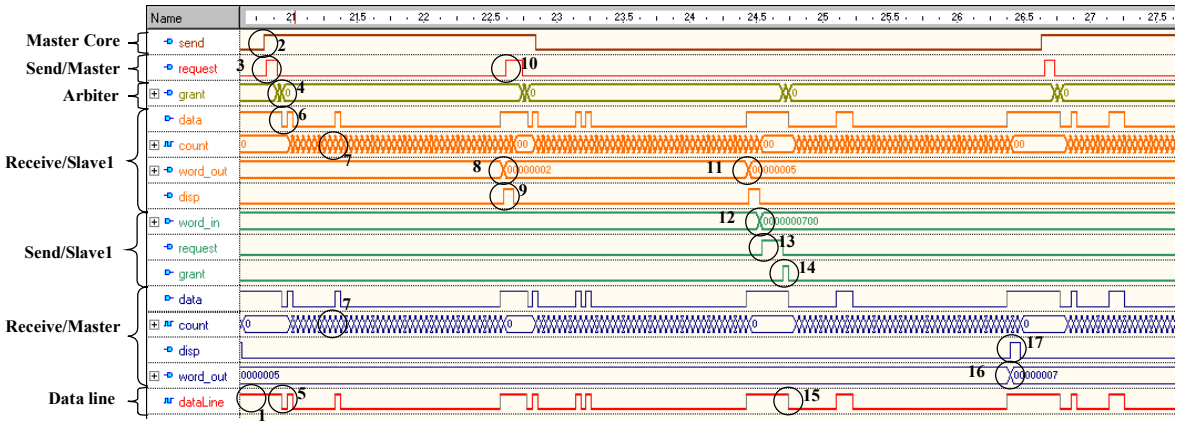

Figure 7- Functional simulation of the communication interface.

4.2. Partial bitstream generation

Once the communication interface is functionally validated, the next step is to generate an individual bitstream for each module. The floorplanning tool is used to restrict the physical position of each module. As mentioned before, each slave core is synthesized with a "dummy controller" and the controller is synthesized with "dummy cores", to ensure the common routing wires.

The main difficulties to generated these bitstreams are:

- Placement of the tri-state buffers in fixed positions. Due to the scarceness wire resources, the routing may fail.
- The logical synthesis tool can eliminate tri-state buffers connected to the FPGA outputs. Problem solved with constraints inserted in the design.
- Clock signals are routed with global clock lines. Constraints are applied to the routing tool to avoid the usage of such resources.
- Ensure that the routing will be restraint to the reserved module area.
- Ensure the same routing between tri-state buffer layers. This is the critical restriction. The routing is manually executed between signals belonging to the communication interface, and after all other signals have been automatically routed.

Complete bitstreams are created, one for each module. Dedicated tools were implemented to extract a partial bitstream from complete bitstreams (see Section 2.3), since the employed CAD tools do not generate partial bitstreams.

4.3. Core unifier tool

The last validation step is to insert the slave core into the controller core, achieving partial and dynamic reconfiguration.

A CAD tool to insert or to remove cores in a FPGA, named *core unifier* is used. This tool works as follows:

1. A *master bitstream*, corresponding to the controller core, is opened.
2. One or more bitstreams containing slave cores to be inserted into the *master bitstream* are opened. The user selects the area corresponding to the core, and all FPGA components inside this area are inserted into the master *bitstream*.
3. The tool creates a partial bitstream, containing the modified area. Partial reconfiguration is then executed, inserting a new core into the FPGA.

This procedure is illustrated in Figure 8.

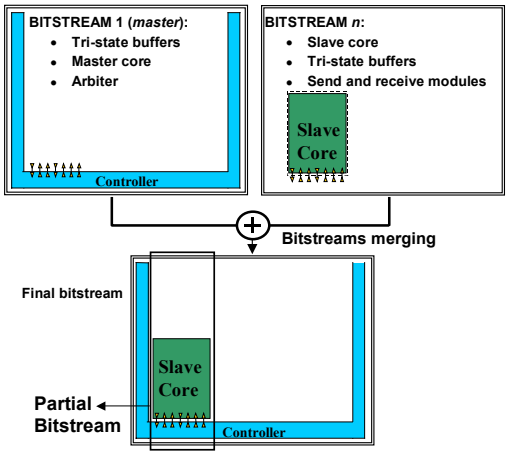

Figure 8 - Bitstream merging procedure.

At present, the tool creates complete and partial bitstreams. Complete bitstream download with core insertion was achieved successfully, however partial bitstream fails due to the lack of partial download tools. Another observed problem is related to routing. Even with correct common routing wires, sometimes the core insertion fails. A CAD tool is under development to solve problems related to routing.

5. Conclusion and future work

The main contribution of this work is the presented method to reuse hard cores in FPGAs. The requirements to achieve reuse were presented and the proposed communication interface is described. The communication interface is functionally validated, both by simulation and by prototyping. Customized tools were developed to merge independent bitstreams and to create partial bitstreams.

This work shows that it is possible to implement "virtual hardware" with commercial FPGA devices. The main obstacles to develop automatic CAD tools are the internal architecture of these devices, which are quite difficult to be used for partial reconfiguration, requiring complex manual operations (manual placement and manual routing). Intellectual IP distribution of hard cores through the Internet is also possible, since the communication interface standardize the input/output protocol.

As suggestions for future work we can enumerate: (i) to extend the bus structure to more bit lines and different bus arbitration schemes; (ii) to develop CAD tools to automate the manual steps mentioned above; (iii) to develop techniques for core relocation.

6. References

[1] International Technology Roadmap for Semiconductors, 2001, http://public.itrs.net.

[2] Bergamaschi,R.A.; et.al. **Automating the design of SOCs using cores.** IEEE Design & Test of Computers, Volume: 18(**5**), Sept.-Oct. 2001, pp. 32-45.

[3] Keating,M.; Bricaud,P. **Reuse methodology manual for system-on-a-chip designs**. Second Edition. Kluwer Academic Publishers, Norwell, MA, 1999, 286p.

[4] Dehon, A.; Caspi, E.; Chu, M.; Huang, R.; Yeh, J.; Markovsky, Y.; Wawrzynek,J. **Stream computations organized for reconfigurable execution (SCORE): Introduction and Tutorial**. Proceedings of Field-Programmable Logic and Applications, 2000.

[5] Hartenstein,R. **A decade of reconfigurable computing: a visionary retrospective** Design, Automation and Test in Europe, 2001, pp. 642 –649.

[6] XILINX**. Virtex 2.5V Field Programmable Gate Arrays (DS003)**. Virtex Series datasheet, 2000.

[7] ATMEL. **At40k Series Configuration**. http://www.atmel.com/atmel/acrobat/0896s.pdf.

[8] XILINX. **Virtex Series Configuration Architecture User Guide Application Note**, 2000. http://www.xilinx.com/xapp/xapp151.pdf

[9] XILINX. **Floorplanner Guide 3.1i**. 2001.

[10] James-Roxby,P.; Guccione,S.A. **Automated extraction of run-time parameterisable cores from program-mable device configurations.** IEEE Symposium on Field-Program. Custom Comp. Machines, 2000, pp. 153 -161.

[11] Opencores. **WISHBONE SOC Interconnection**. http://www.opencores.org/wishbone

[12] Flynn,D. **AMBA: enabling reusable on-chip designs.** IEEE Micro, Volume: 17(**4**), July-Aug. 1997, pp. 20-27.

[13] ALDEC, INC. **Active-HDL 5.1**. 2001. http://www.aldec.com/ActiveHDL/default.htm

[14] VCC Inc. **The Virtual Workbench**. 2000. http://www.vcc.com/vw.html

Acknowledgements: Fernando Moraes gratefully acknowledges the support of the CNPq through research grant number 522939/96-1.