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 complex 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 inside VLSI reconfigurable devices. The approach targets system-on-a-chip designs built in a single large FPGA. The paper proposes a communication interface that allows IP cores replacement during FPGA normal operation. The same interface also allows the communication among distinct IP cores to take place.

1. Introduction

IP cores are complex pre-designed and pre-verified hardware modules today considered as key components in the development of system-on-a-chip (SoC) designs. According to a recent ITRS report [1], by 2012, IP cores will constitute 90% of the area in state of the art integrated circuits. Table 1 shows the main features of cores in general, according to well-established classification proposals found in the literature [2][3].

Table 1 – Different core types characterization.

Criterion \ Core Type	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

Current system design techniques use to employ or integrate IP cores before logical or physical synthesis. For ASIC designs, this corresponds to the normal design flow. However, for FPGAs a different scenario is possible: dynamic hard IP core insertion and removal. This idea allows inserting a hardware module into an FPGA according to application requirements, at execution time.

For instance, consider the situation where an FPGA board connects to a host computer and a new piece of hardware may be necessary to speed up the processing in a graphic application. A reconfiguration controller can download this piece of hardware (a hard IP core) through the Internet and use it to reconfigure the FPGA, which will then present this new functionality.

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

To achieve hard IP core reuse some requirements have to be fulfilled: (i) FPGAs allowing partial and dynamic reconfiguration must be available; (ii) floorplanning tools to determine IP cores position in the IC are needed; (iii) software for partial bitstream generation must exist; (iv) software for partial bitstream download are necessary; (v) a core communication interface needs to be provided; (vi) input/output pin virtualization must be enabled. Requirements (i) to (iii) can be satisfied with existing devices and tools. To the Authors knowledge, works fulfilling requirements (iv) to (vi) 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 operation interruption. This interface is a fixed module in the FPGA, with three main functions: bus arbitration, communication between modules, and pin input/output virtualization.

This rest of this paper has the following structure. Section 2 presents the requirements for hard IP core reuse. Section 3 details the main contribution of this work, the communication interface. Section 4 presents preliminary implementation results. Finally, Section 5 draws some conclusions and directions for future work.

2. Requirements for hard IP core reuse

Research in reconfigurable computing emphasizes mostly the use of coarse grain reconfigurable arrays with datapath widths greater than 1 bit. Fine-grained

architectures, like commercial FPGAs, are much less efficient with large routing area overhead and poor routability [5].

In spite of this fact, our goal is to investigate the feasibility of implementing reconfigurable computing systems into commercial FPGAs, since such devices are more widely available than coarse-grain reconfigurable devices. The reconfiguration granularity of the proposed approach is an important point, and it classifies as coarse-grain, since it achieves reconfiguration with hard IP cores.

The next sections detail each requirement discussed in the Introduction.

2.1. FPGA support to partial reconfiguration

Examples of available FPGA devices allowing partial and dynamic reconfiguration are Xilinx Virtex [6] and Atmel At40k [7]. Selection of the target for this work felled on the Xilinx Virtex FPGAs 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 reconfiguration architecture.

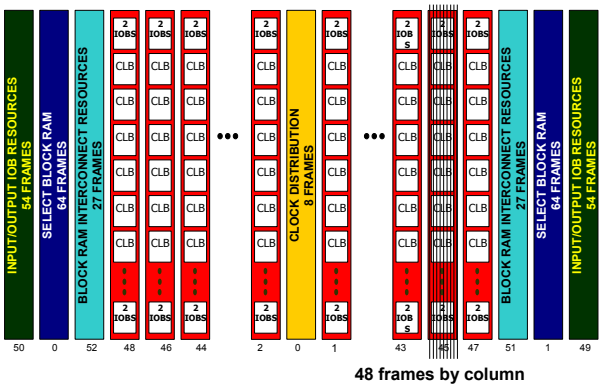

Figure 1 - Virtex FPGA example of internal reconfiguration architecture (for the XCV300 device).

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

The Virtex device is partially reconfigurable, 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, by performing an operation called *readback*. Next, the required modification is inserted over the read frames. In practice, this feature makes the structure of a Virtex device a single-dimension array of columns for reconfiguration purposes.

2.1.1. Element Addressing

To partially reconfigure a device it is necessary to address individual elements inside the configuration file, also called *bitstream*. The following equations define how to address bits inside LUTs [8], the main FPGA logic resource available for reconfiguration:

$\text{if } \left(\text{CLB}_{\text{col}} \le \frac{\boldsymbol{Chip}_{col}}{2} \right) \text{ then } \quad \text{MJA} = \boldsymbol{Chip}_{col} - (\text{CLB}_{\text{col}} * 2) + 2$ else $\text{MJA} = (\text{CLB}_{\text{col}} * 2) - \boldsymbol{Chip}_{col} - 1$
$MNA = lut\_bit + wd - slice * (2*lut\_bit + 17)$
$fm\_bit\_idx = 3 + 18*CLB_{ROW} - FG + RW*32$
$fm\_st\_wd = FL * (8 + (MJA-1)*48 + MNA) + RW*FL$
$fm\_wd = abs(fm\_bit\_idx / 32)$
$fm\_wd\_bit\_idx = 31 + 32*fm\_wd - fm\_bit\_idx$

The terms in these equations are defined as follows:

- MJA - *Major Address*: represents the column address. $Chip_{col}$ is the number of columns available in the device.
- MNA - *Minor Address*: identifies in which frame the *lut_bit* is placed. MNA assumes values between 0 and 47. "*wd*" is the number of bits per word (32) and "*slice*" is the slice number.
- fm_bit_idx – *frame bit index*: indicates the start position of the CLB being addressed. Constant 18 multiplies CLB_{row} because each CLB requires 18 bits per frame. "FG" is equal to 1 if the desired bit is in a G-Lut, and 0 if it is in an F-Lut. "RW" is equal to 0 when writing data to the FPGA and 1 when reading data from the FPGA (read-back operation).
- fm_st_wd – *frame starting word* in the bitstream (file

containing 32-bit words). “FL” designates the frame length, i.e., the number of 32-bit words needed to store a complete frame. “8” is the number of clock columns.

- fm_wd – indicates, in the bitstream, which word contains a given desired bit.
- fm_wd_bit_idx – designates the bit inside *fm_word* containing some desired information.

For example, suppose a need arise to change the 14th bit of an F-LUT, placed at slice 0 of row 1 column 1 (R1C1.S0.F), using the device XCV100, which has $Chip_{cols}$=30, FL=14 (Figure 2). Applying the above equations, it is possible to obtain: *MJA*=30, *MNA*=46, *fm_bit_idx*=21, *fm_st_wd*= 20.244, *fm_wd*=0, *fm_wd_bit_idx*=10. These results mean that the 10th bit (*fm_wd_bit_idx*) of the bitstream word 20.244 (*fm_st_wd* + *f_wd*) is the location of the desired bit. Thus, changing this bit and recomputing the bitstream CRC, it is possible to reconfigure the FPGA.

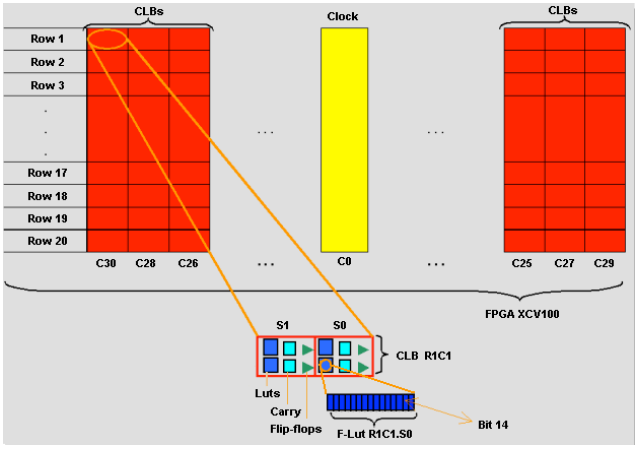

Figure 2 - Locating bit 14 from an F-LUT for a given CLB.

2.2. Availability of floorplanning tools

Partial reconfiguration is only possible if the IP cores to insert do not overlap the existing cores inside the FPGA. So, it is required to produce placement restrictions to use during physical synthesis.

Example of such a tool is the Xilinx floorplanner [9], a graphical tool that allows the designer to control IP cores position and shape in a FPGA using “drag and drop” facilities. Figure 3 displays an example of screenshot for this tool.

A browser displaying the design hierarchy allows the designer to restrict the position of each core component. Detailed placement and routing execute typically after the floorplanning step.

The Xilinx floorplanning tool allows constraining logic blocks, memory blocks, I/O blocks, and tristate buffers. This tool does not allow constraining detailed routing. As a result, after routing some wires may fall outside the delimited area, overlapping 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.

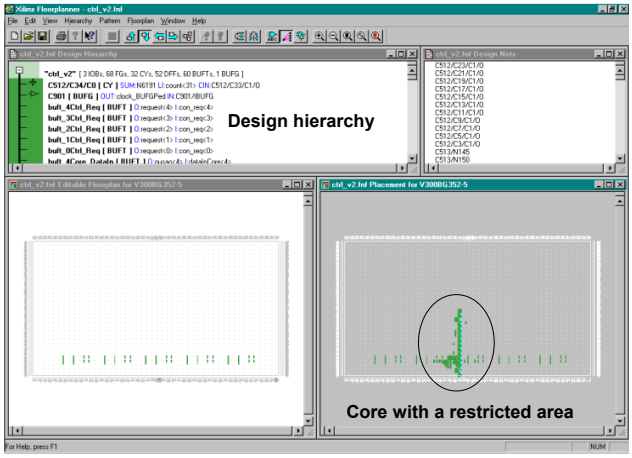

Figure 3 - Xilinx floorplanning interface.

2.3. Partial bitstream generation

Tools provided by FPGA vendors do not generate partial bitstreams. JBits [10] is a toolbox of Java classes provided by Xilinx to manipulate an abstract view of the Virtex FPGA configuration architecture.

Without using JBits, the reconfiguration tools developer has to deal with a huge 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 by readback operations.

Figure 4 shows examples of JBits 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 4- Example of JBits method call.

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

present in the bitstream have to be recomputed, new bitstream CRCs must be generated, and columns do not have a continuous addressing scheme. Even frames are in the left half of the FPGA device, while odd frames are in the right half of it.

2.4. IP core 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 [13], 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. Section 3 details this communication interface.

2.5. I/O pins virtualization

I/O pins virtualization means that cores exchange data only with the communication interface and the real off-chip communication is a function assigned to the interface.

3. Communication interface

Figure 5 outlines the basic idea to achieve communication between synthesized cores. A fixed module, named *controller*, is initially downloaded into the FPGA. Functional cores, named *slave cores*, are downloaded at run time.

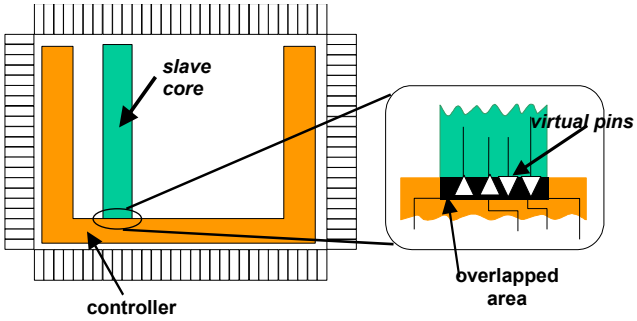

Figure 5 - Communication interface with one tristate buffer layer. Tristate control signals are not shown.

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 tristate buffers, having the position defined by the floorplanning tool. The overlapping of virtual pins creates the interconnection between slave cores and the controller.

The interface proposed in Figure 5 is not feasible due to restrictions imposed by the FPGA architecture. Such restrictions are:

- Each CLB contains LUTs, flip-flops, tristate buffers, and routing resources. It is not possible to restrain the CLB usage only to tristate buffers and routing. Therefore, overlapping slave cores over the controller could destroy some of the controller functions.
- Each CLB contains only 2 tristate buffers. The reduced number of buffers limits the IP core communication width. In addition, the buffers routing wires share common hex lines [6], restricting the routing tool. Due to this limitation, adoption of a serial bus (using 1-bit data lines) is the choice here.

In order to overcome these limitations, a communication interface with two tristate buffer layers with common routing wires is implemented. Figure 6 shows this solution. One buffer layer belongs to the controller and another buffer layer belongs to the slave IP cores.

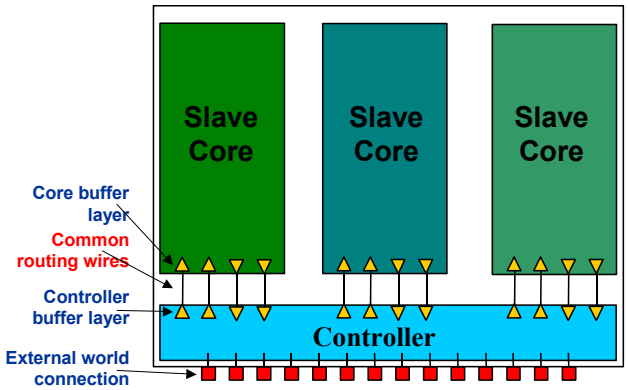

Figure 6 - Communication interface with two tristate buffer layers and a common routing wire.

In order to have common routing wires, the controller is synthesized by using *dummy cores*, which contain the buffers that will belong to the slave IP cores. The same procedure works for the slave IP cores synthesis, which include a *dummy controller*. Dummy cores are also important to avoid floating signals in the communication interface.

3.1. Controller implementation

Figure 7 illustrates the 3-module controller structure:

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

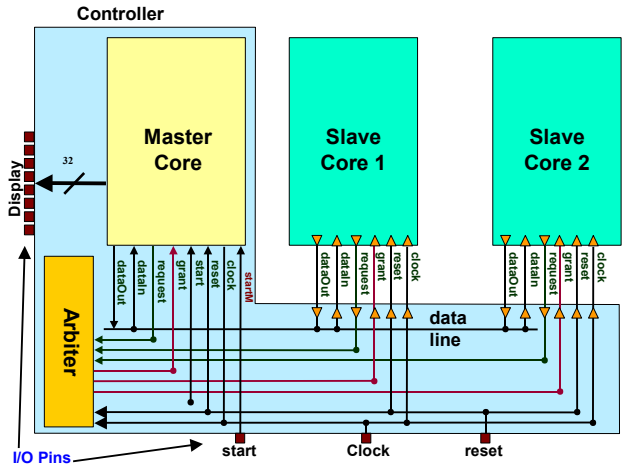

Figure 7 - Controller modules: master core, arbiter and communication bus.

The arbiter continuously reads the request lines, in a serial fashion. When a request line is active the arbiter grants 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, it is possible to serve all cores, even those with lower priority.

The implementation of the master core depends on the functions and I/O requirements of the slave IP 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. Figure 8 illustrates this structure.

When the hw-core wants to send data to another core, it activates the 'disp' line and puts a 40-bit word in the 'word_in' port. The send module activates the 'request' signal, and waits for the assertion of the 'grant' signal. When this happens, 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 that the send module is ready to send a new word. Neither buffering mechanism nor time-out controls were 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 module ignores the remaining 32 bits. 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.

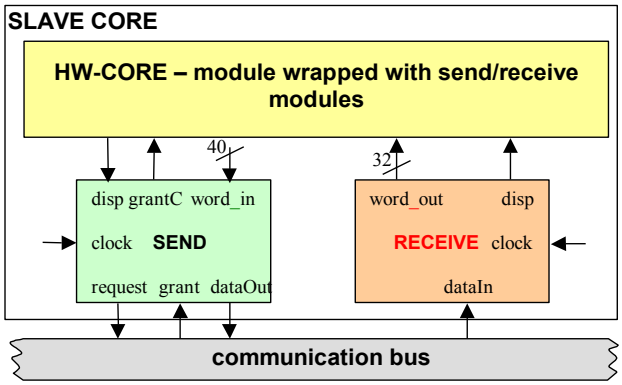

Figure 8 - Send and receive modules.

3.3. Case study

The implementation of a simple case study application validated the communication interface. Tree modules compose this application, as illustrated in Figure 7: (1) controller (2) slave1; (3) slave2.

The function of each slave core is very simple. First, it receives two 32-bit words. Then it executes an arithmetic operation over these (*slave1* executes addition and *slave2* subtraction), sending of the result to the master core. After that, it waits for two new words.

The master core has three internal registers acting as data memory, and a program memory holding the operations. Virtex LUTRAMs, a small memory resource available inside Virtex FPGAs, implement the program memory. Specialized reconfiguration tools define the program memory contents. 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 IP core address. The master core merges this address to the first register value, sending the 40-bit package to the communication bus. The procedure repeats for the second operator. After sending the two operators, the master core waits the response from the target core, storing the received value into the 'T' register. Then, it sends the result of each operation to the external world, which shows it in an 8-digit display.

4. Preliminary results

This section presents preliminary results, including the communication interface functional validation, the generation of the partial bitstream, and an implemented

tool for partial IP cores manipulation.

4.1. Functional validation

Module description occurred in the VHDL language, and simulation employed the Active-HDL simulator, from Aldec [14]. Figure 9 illustrates the system operation, using the case study presented in Section 3.3. The numbers highlighted in Figure 9 represent events with the following meaning:

1. Initially, the data line is in steady state (logic 1).
2. The master core asserts its internal send signal (connected to 'disp' in Figure 8), indicating it has data available to transfer.
3. The master send module asserts the request signal, and waits for the grant signal.
4. The grant signal is sent to the master send module.
5. Data transmission starts, the first '0' corresponds to the start bit.
6. Receive modules detect start, initiating data storage.
7. After 8 clock cycles, all receive modules compare the stored byte to their respective 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 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 to 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.

After the system functional validation, the next step is to prototype it. The prototyping board *Virtual Workbench* [15], with a Virtex XCV300, was used. In the experiment the communication bus has three "sockets" (set of tristate 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 tristate 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 avoiding high-impedance values to occur in the data line is mandatory.

The system worked correctly in the prototyping board, showing that the communication interface with tristate buffers can be used in Virtex devices to interconnect hard IP cores.

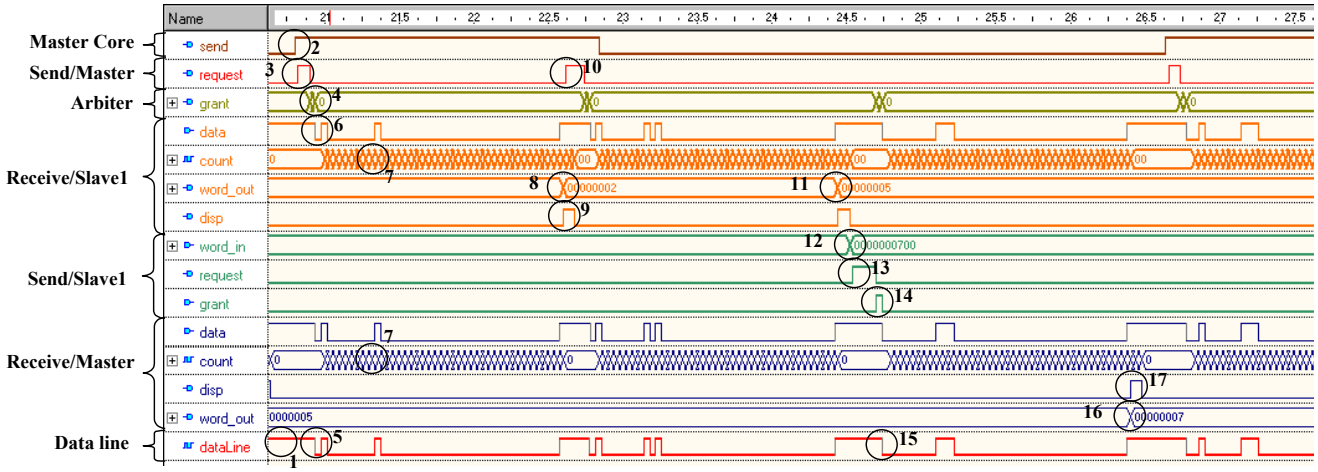

Figure 9 - Functional simulation of the communication interface.

4.2. Partial bitstream generation

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, the synthesis of each slave IP core includes a dummy controller and the synthesis of the controller includes dummy cores, to ensure the common routing wires.

The main difficulties to generate these bitstreams are:

- Placement of the tristate buffers in fixed positions. Due to the scarcity of wire resources, the routing often fails.

- The logical synthesis tool can eliminate tristate buffers connected to the FPGA outputs. Constraints inserted in the design solve this problem.
- The routing associates clock signals with global clock lines. Constraints applied to the routing tool avoid the use of such resources.
- It is hard to ensure that the routing will be constrained to the reserved module area.
- Ensure the same routing between tristate buffer layers is hard. This is the critical restriction. The routing needs to be manually executed between signals belonging to the communication interface, after all other signals are automatically routed.

Complete bitstreams are created, one for each module. Figure 10 shows the routing between the two buffer layers and the frontier between the controller and an IP core.

Dedicated tools were implemented to extract a partial bitstream from complete bitstreams as described in Section 2.3. This was mandatory, since the employed tools do not generate partial bitstreams.

4.3. Core unifier tool

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

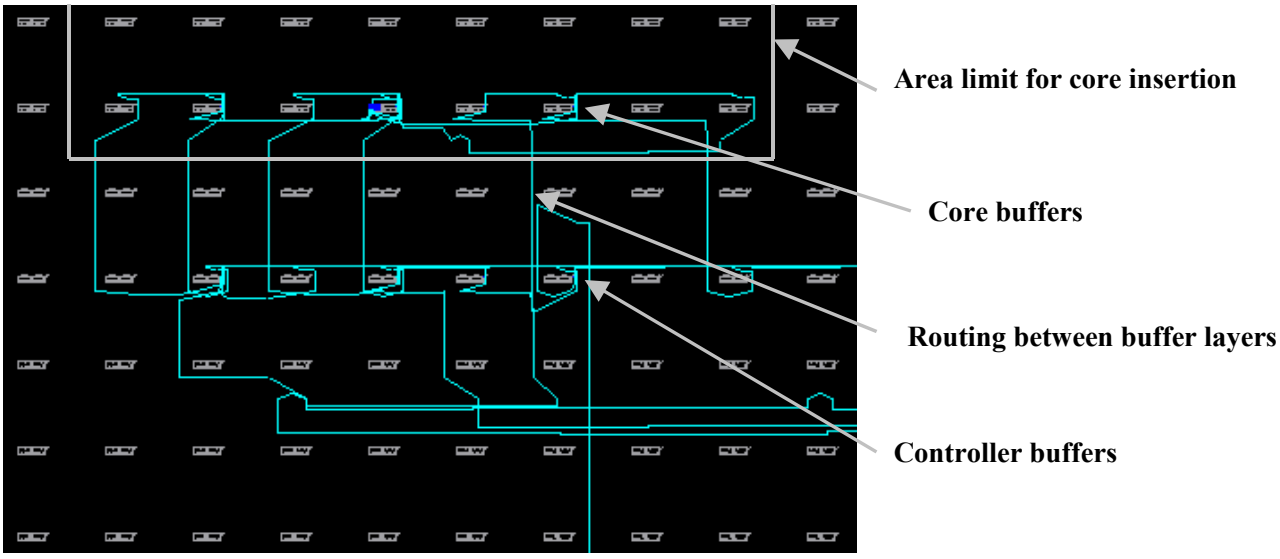

Figure 10 – Routing between buffers in the frontier between an IP core and the controller.

A tool to insert or remove cores in a FPGA, named *core unifier* was developed in the context of this work, and is used in this step. The tool works as follows. First, a *master bitstream* file, corresponding to the controller core, is opened. Then, one or more bitstreams files 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*. Lastly, the tool creates a partial bitstream, containing the modified area. Partial reconfiguration is then executed, inserting a new core into the FPGA. Figure 11 illustrates this procedure.

The Virtex address equations described in Section 2.1.1. are the basis for the development of the *core unifier* tool. This tool provides a structured form to interconnect cores, together with the possibility to dynamically replace cores in the FPGA.

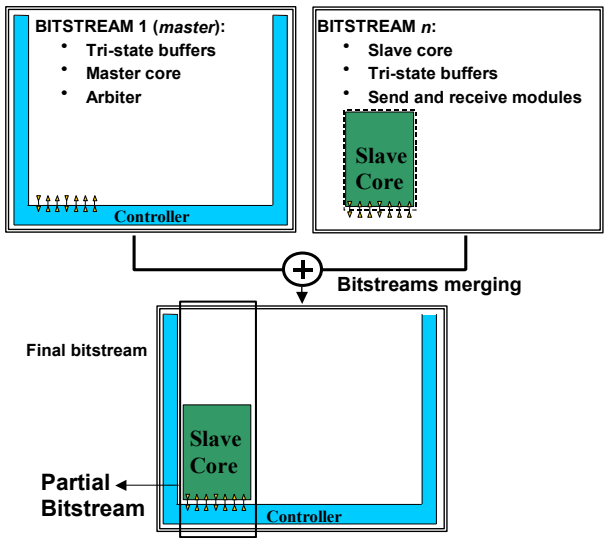

Figure 11 - Bitstream merging procedure.

Figure 12 presents the main window of the *core* unifier tool. This window has a 48*x*32 grid, representing all CLBs of a Virtex XCV300 device and it is different for distinct devices. Light and dark gray squares represent CLBs not used (default values). Red squares represent CLBs used by the *master bitstream*. Squares with different colors (e.g. yellow) represent inserted cores. Visualization of LUT values is possible using an auxiliary window, selecting the CLB with the mouse.

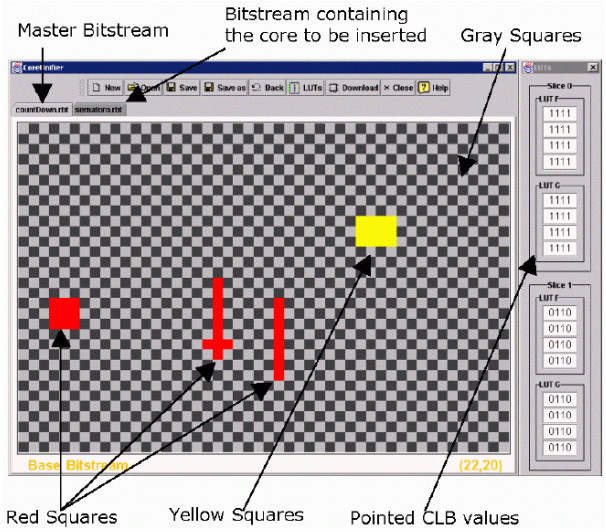

Figure 12 – Core unifier main window.

The user can insert new cores into the master bitstream, a feature that adds flexibility to the tool, allowing dynamic insertion and/or removal of IP cores.

This tool permits to implement virtual hardware, in a similar manner as dealing with virtual memory. The user may have several hard cores stored in memory. As a function of some execution scheduling, these may be partially downloaded into the FPGA.

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 relates to routing. Even with correct common routing wires, sometimes the core insertion fails. A new tool is under development to solve routing problems.

5. Conclusions and future work

The main contribution of this work is the presented method to reuse hard IP cores in FPGAs. The requirements to achieve partial and dynamic IP core reuse were presented and a communication interface was proposed and described. The communication interface is functional, validated through simulation and prototyping. A tool to merge independent bitstreams and to create partial bitstreams was developed and described.

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

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

6. References

[1] Sematech International. **International Technology Roadmap for Semiconductors – 2002 Update.** 2002. Available at 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**), Sep.-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**. In: Proceedings of Field-Programmable Logic and Applications (FPL´00), 2000.

[5] Hartenstein,R. **A decade of reconfigurable computing: a visionary retrospective**. In: Design, Automation and Test in Europe (DATE´01), 2001, pp. 642 –649.

[6] Xilinx, Inc. Virtex 2.5V Field Programmable Gate Arrays (DS003). Virtex Series Datasheet, Version 2.5, 2001.

[7] Atmel, Inc. **5K-50K gates coprocessor FPGA with FreeRAM**. Datasheet, Document doc2818. 2003. Available at http://www.atmel.com/dyn/resources/prod_documents/doc2818.pdf

[8] Xilinx, Inc. **Virtex Series Configuration Architecture User Guide**. Xilinx Application Note XAPP151, Version 1.5, Sep. 2000. Available at http://www.xilinx.com/xapp/xapp151.pdf

[9] Xilinx, Inc. **Floorplanner Guide 3.1i**. 2001.

[10] James-Roxby,P.; Guccione,S.A. **Automated extraction of run-time parameterisable cores from program-mable device configurations**. In: IEEE Symposium on Field-Program. Custom Comp. Machines (FCCM´00), 2000.

[11] OpenCores.Org. **WISHBONE SoC Interconnection**. Available at 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] IBM, Inc. **The CoreConnect™ Bus Architecture**. 2000 Available at http://www.chips.ibm.com/products/coreconnect/docs/ crco_wp.pdf.

[14] Aldec, Inc. **Active-HDL 5.1**. 2001. Available at http://www.aldec.com/ActiveHDL/default.htm

[15] Virtual Computer Corporation Inc. **The Virtual Workbench**. 2000. Available at http://www.vcc.com/vw.html.