Jumble: A Hardware-in-the-Loop Simulation System for JHDL

Computer Science MPhil dissertation
(Computer Science PhD program, Microelectronics Option)
presented by **David Castells i Rufas**
and supervised by Jordi Carrabina i Bordoll
Bellaterra, September 2008

Universitat
Autònoma
de Barcelona

Jordi Carrabina i Bordoll, lecturer in the Microelectronics and Electronics Systems Department from the “Universitat Autònoma de Barcelona”,

CERTIFIES:

That the present research has been done under his direction by David Castells i Rufas as experimental work inside the Computer Science PhD program, Microelectronics option, given by the “Universitat Autònoma de Barcelona”.

Bellaterra, September 2008

Jordi Carrabina i Bordoll

Summary

This MPhil dissertation presents a new verification system for FPGA based designs described in the JHDL hardware description language. The method consists of performing hardware emulation of designer selected blocks in a co-simulation environment. Although JHDL has a Hardware execution mode it does not provide a fine control of which blocks have to be executed in Hardware and it is based on Xilinx readback technology. In this work the simulation environment is extended to control of the Hardware emulation system, instrument the design for debug, and automatically create the interface to communicate the simulator with the emulated hardware block. The resulting system does not offer 100% observability and controllability of hardware blocks. Nevertheless its interactivity provides a solid basis for incremental verification while offering the possibility of substantial simulation speedups.

Resum

Aquest treball de recerca presenta un nou sistema de verificació per dissenys descrits amb el llenguatge de descripció de Hardware JHDL. El mètode consisteix en realitzar l'emulació del bloc de Hardware seleccionat pel dissenyador en l'entorn de simulació. Malgrat que el JHDL ja disposava d'un mode d'execució, aquest no proporciona un control fi sobre quins blocs s'executen en Hardware i es basa en la tecnologia *readback* de Xilinx. En aquest treball s'amplia l'entorn de simulació per controlar el sistema d'emulació, instrumentar els dissenys per a la depuració i crea automàticament la interfície de comunicació entre el simulador i el bloc Hardware emulat. El sistema resultant no ofereix ni observabilitat ni controlabilitat completa, però suposa una sòlida base per realitzar verificació incremental i reduir el temps de simulació de manera significativa.

Resumen

Este trabajo de investigación presenta un nuevo sistema de verificación para diseños descritos mediante el lenguaje de descripción de Hardware JHDL. El método consiste en realizar la emulación del bloque de Hardware seleccionado por el diseñador dentro del entorno de simulación. A pesar de que JHDL ya disponía de un modo de ejecución, éste no proporciona un control fino sobre que bloques se ejecutan en Hardware y se basa en la tecnología *readback* de Xilinx. En este trabajo se amplía el entorno de simulación para controlar el sistema de emulación, instrumentar los diseños para su depuración y crear automáticamente la interfaz de comunicación entre el simulador i el bloque Hardware emulado. El sistema resultante no ofrece ni observabilidad ni controlabilidad completa, pero supone una sólida base para realizar verificación incremental i reducir el tiempo de simulación de manera significativa.

Acknowledgements

I would like to thank the many people that have helped me, in very different ways, to finish this never ending story. My gratitude for all that I mention and for the ones I probably forget.

To my wife Mar Soldevila for the endless hours, her infinite patience, support and love.

To my daughter Anna for her contribution of some published photos and for not erasing all my notebook hard disk in the many opportunities she had, and of course for being a so lovely kid.

To my newborn daughter Carla, I wish you can see your father getting a PhD before you get married.

To my newborn daughter Marta, who allowed us to dream of a wonderful life, a dream that was vanished too early.

To Jordi Carrabina for his guidance on this work and for giving me the chance to enjoy an academic lifestyle after so many years of being in the corporate fire front.

To Eloi Ramon, Lluis Ribas, Toni Portero, Quim Saiz, and Lluís Terés for sharing their knowledge and be always open for academic discussion.

To Brent Nelson and Francky Cathoor for giving me valuable feedback about the topics of my research.

To Jaume Joven for his optimism and for being a so easy person to work with.

To Sergi Risueño, Eduard Fernàndez, Jorge Luis Zapata, and Juan Carlos Chak for listening me and, from time to time, letting me think I can teach them something.

To Aitor Rodriguez, Eric Teruel, and Pablo Romàn for helping me so much in my everyday work and still be nice for returning a smile even when I was in bad mood (probably too often).

To Oscar Navas, David Novo, Martí Bonamusa, Josep Mesado, Jordi Escrig, and Jordi Farré, for being great people to have around.

To Alexis Morugó for his resolution to complete his project.

To Borja Martinez for revealing to me some mysteries of Quartus.

To Enric Pons for enduring the use of some of the first results of my tools.

To my brother Enric Castells for awakening my curiosity during my childhood by constant challenges and puzzles, especially in hiding things to avoid my finding of his VIC20 computer and electronic kits.

To Toni Ubieto and Pere Joan Cardona for, without knowing, showing me the little satisfactions of research, in its literal meaning.

To my grandfather Esteve Rufas, who passed away recently, for being always starting something that would never end but be so stubborn to never admit it.

To my parents, for their indispensable financial support during my youth and of course, for giving me the most important thing in the world: life.

Contents

Index of Publications

This work is a monograph, which contains some unpublished material, but is mainly based on the following publications. Copyright of the previously published material is owned by the copyright holders of the following publications.

[Castells04] D. Castells, M. Monton, R. Pla, D.Novo, A. Portero, O. Navas, J. Farré, L. Ribas, J. Carrabina "Comparing Design Flows fot Structural System Level Specifications facing FPGA Platforms" DCIS 2004.

[Castells04b] D. Castells-Rufas, J. Farré-Capel, J. Carrabina, “Experimentación con el lenguaje JHDL”, in *Proceedings of IV Jornadas de Computación Reconfigurable y Aplicaciones JCRA*. Barcelona, September, 2004.

[Castells05] D. Castells-Rufas, E. Pons, J. Carrabina. "Implementación de un sistema OCR en FPGA". in *Proceedings of V Jornadas de Computación Reconfigurable y Aplicaciones (JCRA)*. Granada, September, 2005.

[Castells06] D. Castells-Rufas, J. Carrabina, "Camera-Based Digit Recognition System". *13th International Conference on Electronics, Circuits and Systems (ICECS2006)*. Nice, France, December 10-13, 2006.

[Castells06b] D. Castells i Rufas, A. Morugó, J. Carrabina, "Traducción automática de JHDL a VHDL". *VI Jornadas sobre Computación Reconfigurable y Aplicaciones* (JCRA2006), Cáceres, Spain, September 12-14, 2006

[Castells07] D. Castells-Rufas, J. Carrabina, "Jumble: A Hardware-in-the-Loop Simulation System for JHDL". *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Napa, USA, April 23-25, 2007.

Index of Figures

Acronyms

AMR	*Automatic Meter Reading*
ASIC	*Application Specific Integrated Circuit*
CBSE	*Cycle-Based Simulation Engine*
CCM	*Custom Computing Machine*
CDFG	*Control Data Flow Graph*
CLB	*Configurable Logic Block*
COTS	*Commercial of the shelf*
DCT	*Discrete Cosine Transform*
DE	*Driving Environment*
DFG	*Data Flow Graph*
DSP	*Digital Signal Processing* *Digital Signal Processor*
DUT	*Device Under Test*
EDA	*Electronic Design Automation*
EIA	*Electronic Industries Alliance*
ESL	*Electronic System Level*
FCCM	*FPGA-based Custom Computing Machine o Field-Programmable Custom Computing Machine*
FPGA	*Field Programmable Gate Array*
FPLD	*Field Programmable Logic Device*
FSM	*Finite State Machine*
FSMD	*Finite State Machine with Data-path*
GPP	*General Purpose Processor*
HDL	*Hardware Description Language*
HIL	*Hardware In the Loop*
HLL	*High Level Language*
IC	*Integrated Circuit*
IDCT	*Inverse Discrete Cosine Transform*
IP	*Intellectual Property*
ISA	*Instruction Set Architecture*
ISS	*Instruction Set Simulator*
LE	*Logic Element*
LUT	*Look up Table*
MAC	*Multiply Accumulate*
MPSoC	*MultiProcessor System on Chip*
NoC	*Network on Chip*
NRE	*Non Recurrent Engineering*
PCB	*Printed Circuit Board*
PLA	*Programmable Logic Array*
PLD	*Programmable Logic Device*
PLI	*Programming Language Interface*
PLL	*Phase Locked Loop*
RTL	*Register Transfer Level*
SRAM	*Static Random Access Memory*
SoC	*System on Chip*
TLM	*Transaction Level Modeling*

VHDL	*VHSIC Hardware description Language*
VHSIC	*Very High Speed Integrated Circuit*

Chapter 1

Introduction

Background

Programmable logic devices are becoming essential platforms to prototype hardware-software solutions for commercial electronic systems and sometimes even a good alternative to implement them. The reconfiguration capabilities of these devices offer great advantages versus ASICs as they reduce the risks of possible design errors. The importance of flexibility increases with every passed day. Nowadays the non-recurrent-engineering costs in ASIC design are about one million dollars. Every unexpected additional iteration in the design cycle means a potentially null profit or significant loss of benefit. General Purpose Processors (GPP) and Digital Signal Processors (DSP) offer great deal of flexibility and are broadly used but they have limited hardware resources and are more energy inefficient. On the other hand, FPGAs allow designing specific hardware to maximize parallelism, offering better performance at a competitive price with less energy consumption.

The roles of programmable logic devices have been increasing with time (Figure 1). First devices where based on AND-OR planes, that were able to implement any combinational function and where used to simplify the connectivity of electronic systems (glue logic). Before its introduction, the area of printed circuit boards (PCBs) was dominated by circuits to interconnect the main operational circuits. Programmable devices assumed simple functions, like the ones offered by TTL74 family, so that area for glue-logic was greatly reduced and consequently cost was reduced as well. The steady increase of integration capacity and the technology change to SRAM-based FPGA on 1985 drove their use for ASIC prototyping.

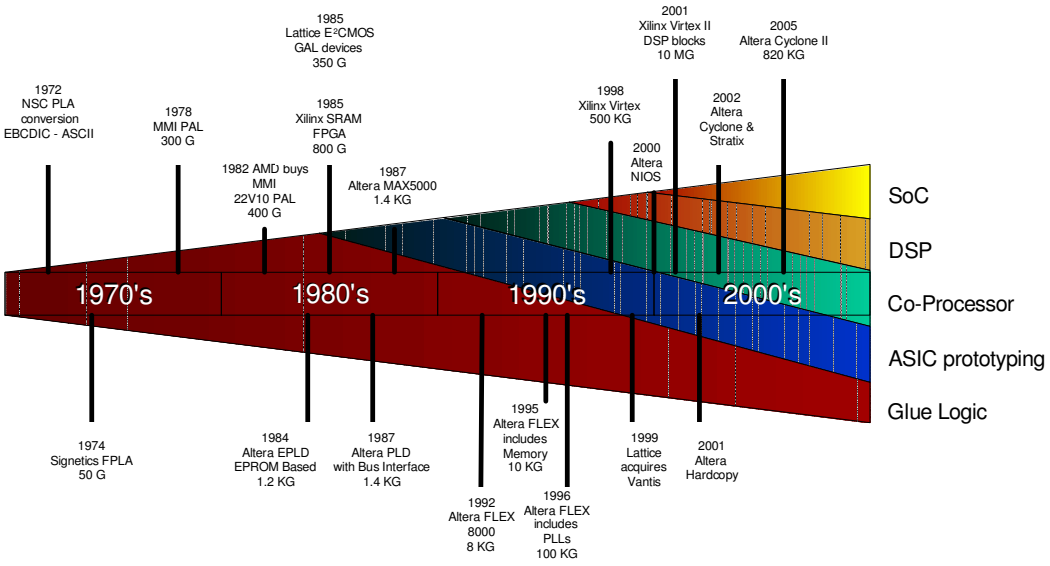

Figure 1 History and roles of FPGA devices

In early 90s people started to see FPGA devices as computing resources rather than as flexible interconnection systems. This leaded to their use as custom reconfigurable co-processor that could help to surpass limitations of the limited resources of CPUs with custom computing units. The coprocessor contribution to an application speedup is derived from Amdahl's law (1) [Amdahl67] in which α is the fraction of the application implemented in Hardware.

$$Application\ Speedup = \frac{1}{(1-\alpha) + \frac{\alpha}{Coprocessor\ Speedup}} \qquad (1)$$

So to get a significant reduction time, a large fraction of the application must be accelerated [Edwards97], and this is not always possible. In addition, the *Coprocessor Speedup* factor is defined by (2) and, since FPGA based co-processors usually are relatively far from CPUs, the overhead in communication often burdens the performance of the system [Benitez04].

$$Speedup\ Factor = \frac{Original\ Time}{Communication\ Time + Computing\ Time} \qquad (2)$$

Simultaneously, and thanks to the dynamic reconfiguration ability of some SRAM-based devices, it was feasible to think of computing machines with reconfigurable functional units. The concept was referred as Custom Computing Machines (CCM) or Field-programmable Custom Computing Machines (FCCM) [Sima00]. The difference between coprocessors and CCMs were that the first were addressed to a single function while the later were designed to exploit reconfiguration to adapt better to different application scenarios. As FPGA devices are reconfigurable in essence, in practice the difference between CCM and FPGA coprocessors has diluted overtime and both terms are often used to express the same concept.

During the 90s reconfigurable platforms were experimentally used for signal processing applications. In late 90s, FPGA manufacturers introduced specific signal processing circuitry like Phase Locked Loops (PLL) to enable multiple clock domains and Multiply-Accumulate modules (MAC) starting a battle with ASIC and DSP manufacturers that were the dominant players of that arena [Tessier01].

In early 2000s the integration capacity had increased enough to embed full microprocessors inside the FPGA device, either as a normal microprocessor sharing part of the silicon area of the FPGA (Hard-Core processor) or as an Intellectual Property block mapped in the device (Soft-Core processor). The new "Intellectual Property" (IP) concept was to Hardware what Software Components were to Software. They should enable the flourishing of a market of resources that would be ready to use for any new design. The combination of various IPs including microprocessors, peripherals and buses and their programming environments allowed the design of Systems on Chip (SoC) on an FPGA.

To summarize, today reconfigurable systems exploit the tradeoff between flexibility and performance in their various roles as glue-logic, ASIC prototyping, co-processing, DSP and SoCs.

Motivation

FPGA applications are designed using a combination of several tools. The design flows depend on the design language and the EDA tool chain. Hardware description languages like VHDL, Verilog and AHDL share a similar design flow as shown in Figure 2. The designer usually receives a specification in the form of a requirement list, which must be transformed into a HDL source code. This first deliverable can be validated with functional simulation tools. This adds the need to develop additional test code, called test-benches, to generate stimuli to the circuit under test. After validation, the synthesis process translates HDL language definitions into hierarchical definitions of the circuit structure based on basic device primitives (ands, ors, multiplexers, flip-flops, ...). There usually exist innumerable circuits structures that can implement the same function defined as HDL code. Mapping HDL into a given hardware structure is a NP complex problem.

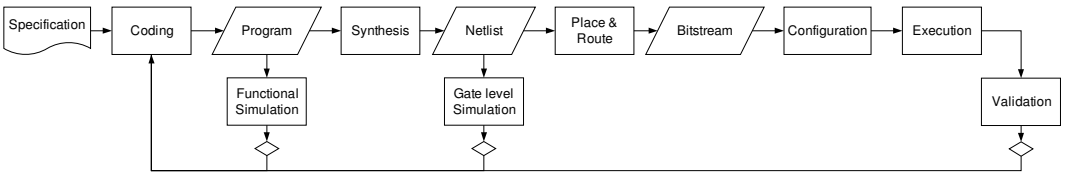

Figure 2 Typical HDL design flow

The result of synthesis, a circuit netlist, can be simulated into gate level simulators, which can use detailed information about time and other physical properties of the particular device primitives, like power consumption. Although it is possible, they are seldom used for large designs because of the long simulation time that they take.

The next step is to place the device primitives into the actual resources of the FPGA and define the interconnection between the logic elements. This function is performed by Place & Route tools that are usually provided by device vendors because of the

amount of technology information needed by them. Finally the bit-stream produced by Place & Route tools is downloaded into the device for its execution.

FPGA device manufacturers are the major providers of FPGA design flows. Since Place & Route is so technology dependent and synthesis algorithms are quite mature, there is little competition in the Synthesis Tools market. Moreover, FPGA device manufacturers try to offer design flows as a single tool and, although allowing it, do not encourage the decoupling of the process.

EDA tools that are based on higher abstraction level languages have few reasons to provide an equivalent synthesis step. Instead, they usually translate high level descriptions into HDL descriptions that can be feed into an HDL design flow as shown in Figure 3. Popular examples are SystemC compilers (like Forte Cynthesizer) that produces RTL VHDL, and model based design tools based on MATLAB Simulink like Xilinx System Generator [Hwang01] and Altera DSP Builder [Altera05].

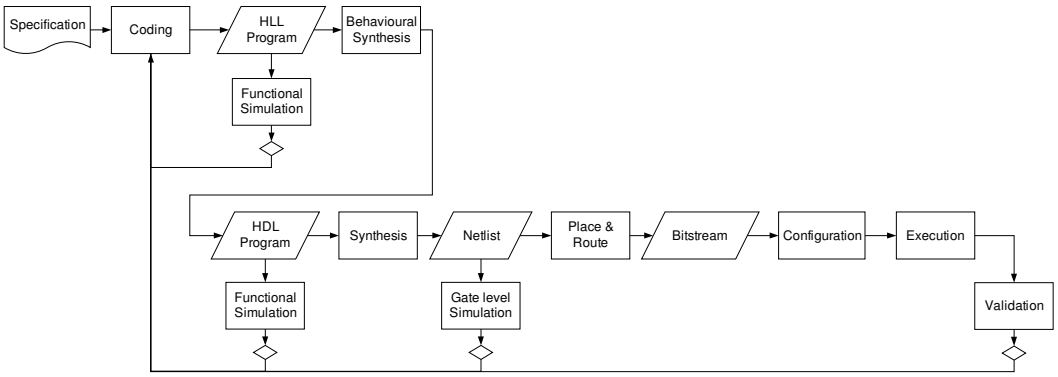

Figure 3 High Level Language Design Flow

Besides working at a higher level of abstraction, which is easier for human understanding, HLL design flows offer benefits: as the code is more abstract it should also be shorter and as a consequence a simulator working at this level should take less time to execute than its equivalent HDL simulator. In addition, the speedup in simulation is greater when cycle accuracy is not needed and one can work at TLM or ISA levels (Figure 4).

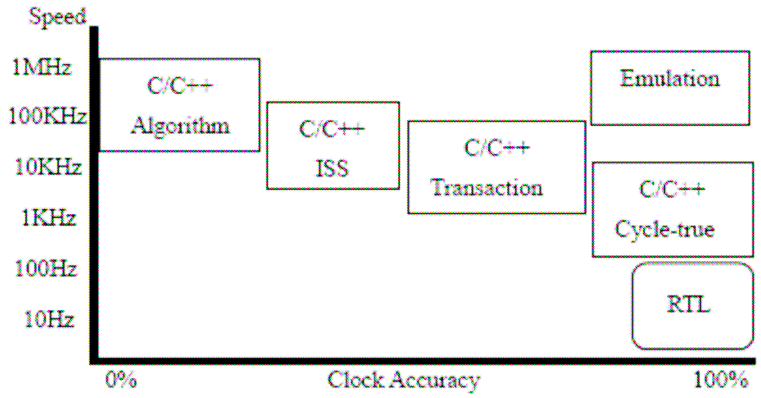

Figure 4 Trade-off between simulation accuracy and speed (from [Nakamura04])

Speeding up simulation is very important. Simulation is the most usual way of performing the verification of a digital circuit design and is usually a very time consuming task. In most projects, time spent on design is exceeded by the time spent on verification ([Hunt02],[Molina07]).

In late 90s, there was the widespread idea that the productivity of design teams was not following the Moore's Law and that, as a consequence, there was a gap between chip capacity and design productivity that was increasing. Semagroup concluded that the number of transistors per chip was increasing by a factor of 58% per year while the productivity of designers measured in transistors per month that a design team produces was increasing by a factor of 21%. This means that although chips with many more transistors are available designing a new chip with the same size takes more time than before. To address the problem and bridge the gap some industry and research groups encourage to re-use components and design from higher levels of abstraction.

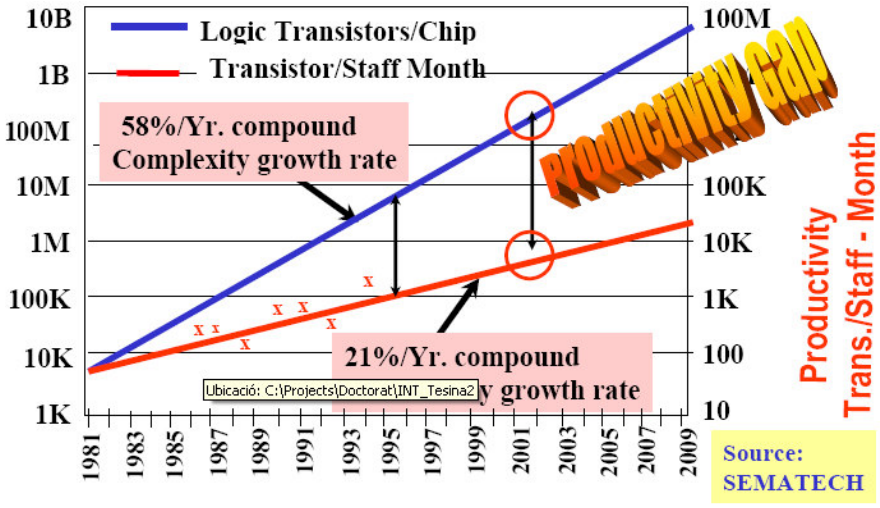

Figure 5 Productivity Gap (from [Ofner04])

However, some argued that this is not the case and that the gap has not been continuously increasing. In [Ofner04] the authors suggest that after a design

technology change happens (for instance a raise in level of abstraction) there are three phases, childhood, youth and old age, in which technology is progressively mastered to reach higher levels of productivity (Figure 6). The eclosion of a new mainstream technology causes a significant drop in productivity as tools are usually immature and designers lack the knowledge to take advantage of it. After this childhood phase, as tools mature, designers are trained, and many designs can be reused great productivity can be achieved to catch-up the Moore's Law.

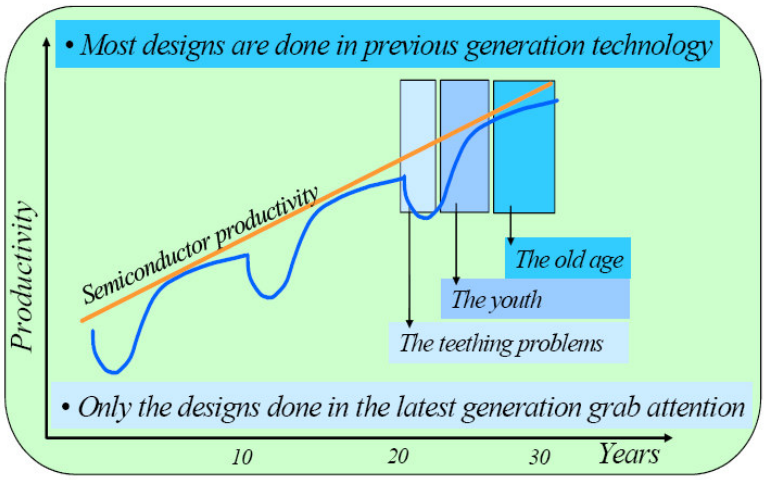

Figure 6 Expected evolution in productivity (from [Ofner04])

Even more shocking are the statements of [Bazeghi05] that conclude that the number of transistors per month produced by a design team has no correlation with the design time effort and suggest better productivity indicators as lines of HDL code and the sum of fan-ins of logic structures. As Sematech productivity gap forecast is based in the transistors per month indicator, it could be not valid at all. This is also backed by the forecast of the evolution of memory usage in SoC designs (Figure 7). As memory is a so simple design, its area occupancy expansion adds little design effort in the development process, and productivity measured in transistors per month is very easily boosted. Furthermore, having more memory on-chip instead of having it off-chip provides some additional benefits because it is usually faster and more energy efficient.

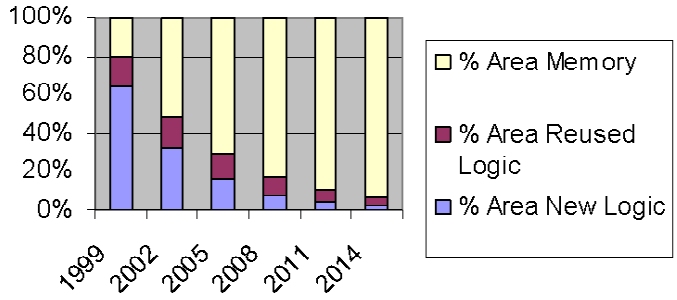

Figure 7 Area occupancy in SoC designs (from [Kahng00])

Nevertheless, there is little discussion about the fact that the complexity of future chips will increase drastically. While techniques like code reuse can reduce a lot the coding effort, they cannot eliminate the testing effort. In fact, the ratio between testing and coding effort keeps increasing steadily and by now testing is the major contribution to the overall development time (Figure 8).

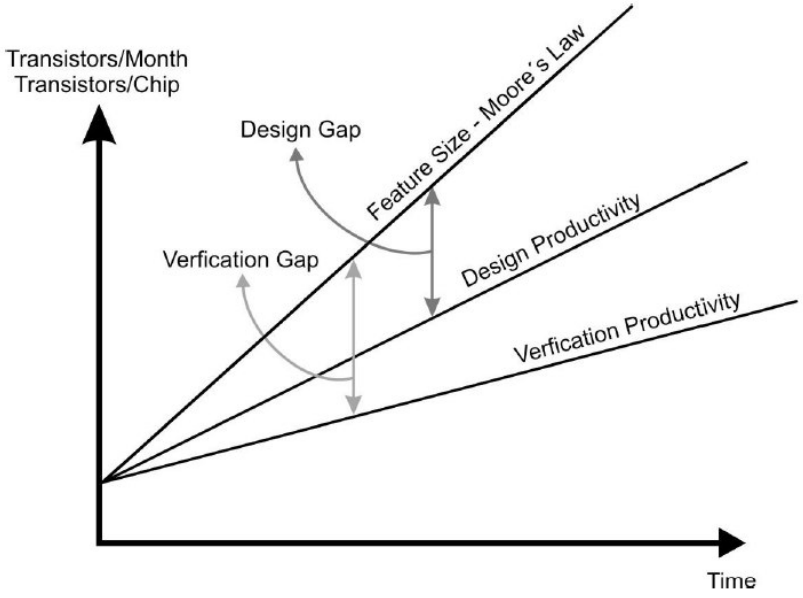

Figure 8 Verification Gap (from [Molina07])

This situation is even worse when various levels of abstraction are mixed. As explained in [Hemani04] this would happen when a HLL based design is reusing HDL blocks following the platform-based design hype. A full HLL design can be quickly verified at HLL level and synthesized assuming a correct-by-construction approach. But including a low level block, forces to include HDL verification tools in the development process, that slows it down while adding complexity in the synthesis step because of the interconnection of systems at various levels.

Hemani encourages fully adopting more abstract levels of design and improving synthesis capabilities to avoid getting stuck in the current transient productivity gap. However, to my knowledge, this is happening very slowly and HDL design flows are still in very good shape.

In this context, functional or logic simulation is still the main method to verify system correctness. As shown in Figure 4, RTL logic simulation offers cycle accuracy but its low speed slows-down the development process. Although hardware emulation based on FPGAs has been commercially available for about a decade and would provide a significant speedup in verification, it is seldom integrated in HDL design flows provided by FPGA vendors. Model based design tools such as Matlab/Simulink, Xilinx System Generator™ [Xilinx00] and Altera DSP Builder™ [Altera05] have successfully revamped hardware emulation for the DSP domain with the concept of hardware-in-the-loop (HIL) simulation and proved that it can be a convenient and easy technology.

The motivation of this work is to show that emulation can also be integrated easily in classic HDL design flows so that verification time can be greatly reduced and so productivity increased.

Objectives

I propose a method based on a developed tool, named Jumble, based on JHDL that integrates hardware emulation into the design flow following a simple approach. The principal idea is to allow designers to work in an interactive simulation environment from which they can select any block of the circuit hierarchy and instruct the tool to transparently download it into a supported hardware platform for real hardware execution. Synthesis of the custom hardware, and all the necessary communication between the simulator and the hardware implementation, is hidden to the user, greatly reducing the complexity of the process.

In the Model Design world, the concept, known as Hardware-in-the-loop simulation, usually suffers from the need of a migration phase from high-level abstraction models to hardware implementation. This process is often done manually.

It makes sense to use the same concept in HDL tools, and in fact, there exists some commercial offerings that contain some of the desired features. However, most of them are bound to a particular hardware, dependent of FPGA device or are loosely coupled with the simulation environment.

The method I propose will integrate the following capabilities:

- Integration of Hardware-in-the-loop simulation in the JHDL environment.
- Automation of synthesis, place & route and device configuration tools and operative system identification of the reconfigured system.
- Independence of the used FPGA device
- Independence of the used hardware platform.

Neither full observability nor full controllability are mandatory requirements, these would be important issues for a Hardware Debugger but are not central for a HIL simulation system. In our case, a user design is viewed as a black box that is downloaded to hardware in order to speedup the whole simulation and possibly to verify that its hardware version behaves equivalently.

Chapter 2

Previous Work

Integration of Simulation and Emulation

The integration of hardware emulation in system simulators has been a recurrent topic in EDA research and industry. An initial clean-room attempt was implemented in the JHDL project with its execution model [Bellows98]. Other attempts focused in integrating reconfigurable hardware platforms into MATLAB/Simulink [Alpha],[Lyr]. Simulink extensions have evolved much since then and become popular among the data signal processing community as they allow accelerating long and complex simulations without leaving a familiar development environment. There are many examples of integrating Hardware in the Loop for System Simulation of various applications, like Bit Error Rate calculation [Singh03][Shirazi03], Software Defined Radio [Dick01][Ramon05], Sonar Beamforming [George99], etc.

Probably because of EDA tools manufacturers and their marketing strategies, the integration of emulation in simulation has been presented under different terms. Sometimes these different flavors are caused by stressing the benefits of some of the techniques in front of others, for instance performance vs. design productivity.

Hardware Simulation

Hardware simulation [Wisniewski01] is a technology that allows speed up simulation time, turning weeks or months of simulation into days or even hours. Designer can “push” whole or a part of the design into hardware. Because it is rather a new technology every solution is different and has different features. Some vendors produce only hardware simulators, others manufacture hardware and also software simulators.

Xcite-2000 (Axis Systems, Inc.)

Xcite-2000 [Axis] offers a simulation performance up to 100K cycles/second. Their product is based on a PCI board containing an Altera FPGA that communicates with the simulator (Figure 9). Design description can be separated into three components: behavioral, RTL and gates. The Xcite compiler automatically maps sections, which can be RCC accelerated (RTL and gate level components), and builds a native compiled simulation image for behavioral sections, which need to stay within the Axis software simulator, Xsim. Using "Hierarchy Extracted" mapping technique, the Xcite compiler automatically maps the design onto arrays of FPGAs.

One of the unique capabilities of Xcite-2000 is its ability to swap software and RCC state in real time. Thus during simulation, the user may choose to swap all RCC state into Xsim in order to debug the design and continue software simulation. Once the circuit is fully diagnosed, simulation state value can be swapped back into RCC for maximum performance acceleration.

Within Xcite RCC simulation, simulation history for all nodes is compressed within RCC and stored onto the workstation. Either during or after simulation, the Xcite VCD-on-Demand capability can extract all node history values without re-simulation. Thus

design debugging has become highly efficient without the high cost of disk storage or simulation slowdown.

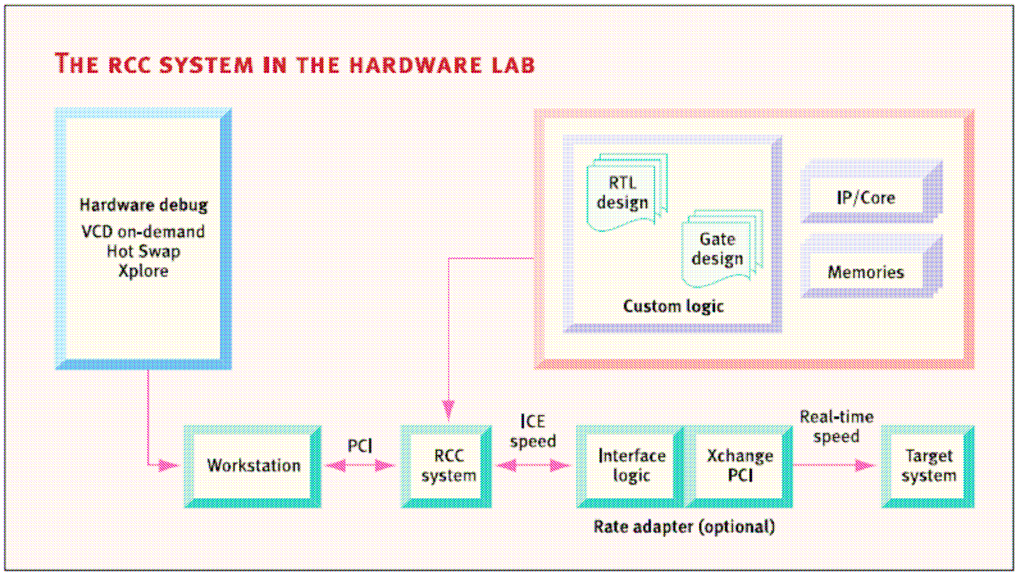

Figure 9 RCC System (from http://www.eetimes.com/isd/features/OEG20010711S0061)

Hardware Embedded Simulation (Aldec, Inc.)

Hardware Embedded Simulation (HES) [Aldec] is the technology that facilitates the incremental design verification of FPGA and ASIC devices while speeding up design verification. HES technology allows you to download selected modules of your design into an FPGA and perform hardware-software co-simulation. After a design block has been verified at the behavioral level, it is synthesized, implemented and downloaded into an FPGA residing on an accelerator board. HES technology supports up to four acceleration boards residing in one computer. The boards are the PCI cards inserted into the slots of the computer.

The entire design is simulated in the HES environment, which consists of an HDL software simulator and PCI boards. This environment assures correct communication between modules located in silicon and modules simulated in software.

Using the HES technology, verified modules of the design can be put into silicon after the synthesis of even a small part of the design. User needs to synthesize the modules that should be pushed into silicon, and the HES Design Verification Manager (DVM) will help to configure HES environment.

Aldec's simulator is based on the Incremental Prototyping. Figure 10 shows the idea of Incremental Prototyping. When module A is finished, it is synthesized and implemented and finally downloaded to the HES board. Since module A resides in the hardware simulator, the designer can prototype module B in software. When module B is verified successfully at the software level, it goes thru incremental synthesis and incremental place and route processes. Note that since module A now resides in the hardware, it is not synthesized and implemented again.

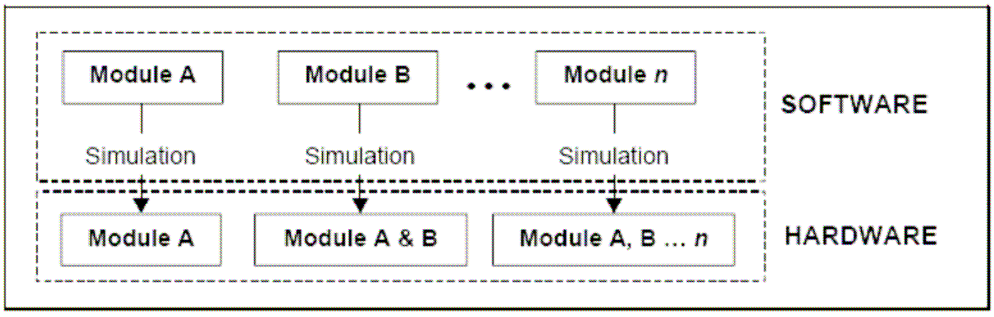

Figure 10 Hardware accelerated design simulation process

Co-Emulation

Co-emulation is a verification technique that maps portions of the design under verification into hardware while the rest is simulated in a software environment on a host computer.

The first traces of this technology are found in [Bauer98] that migrate portions of the circuit under simulation into a commercial emulation system from Quickturn. In this work the synchronization between a cycle accurate simulator and the emulation system is done at every clock cycle. This causes a bottleneck in the system performance. The maximum frequency reported in this work is 200KHz for a 35KG design and no speedup is reported.

In order to increase the overall system performance, other synchronization approaches are possible. [Fritsch99] reports poor speedups unless an enable triggered approach is used, and in this case the co-emulation system is only 3 times faster than functional simulation. [Kudlugi01] proposes a transaction-based approach to synchronize the simulation kernel with an Ikos commercial emulation system (see Figure 11). Synchronization points between the driving environment (DE) and the device under test (DUT) are more abstract than clock cycles. They involve several of them, reducing the number of total synchronization events needed. As a result this approach is faster, and authors actually report clock speeds of 700KHz and speedups of 320 for a 152KG design.

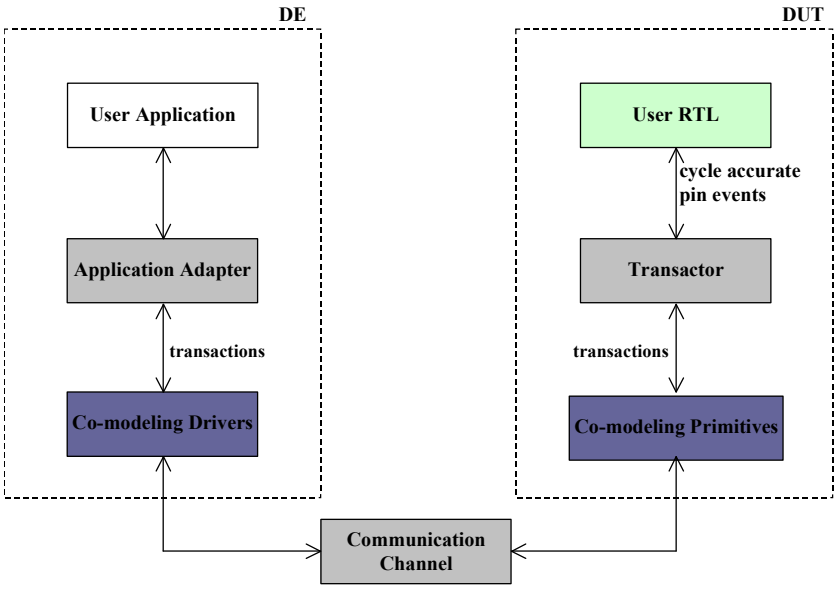

Figure 11 Co-emulation system with transaction based synchronization (from [Kudlugi01])

[Kim04] proposes to split the testbench by the parts that shown a dependence on the outputs of the DUT. The dependent part is moved into Hardware so that the communication between HW and SW parts can be buffered, allowing the use of burst transfers. This approach allows achieving a simulation speed up to 649KHz.

As the transmission of data between the simulator is main bottleneck of co-emulation systems, reducing the amount of transmitted data gives a direct frequency increase. [Nakamura04] studies various scenarios of interaction of C++ based simulators with a FPGA-based emulator via a register interface. The results of this work shows that simulation frequency is about 100KHz for some designs but it can reach to 1.1Mhz when a processor is emulated and only clock and Program Counter (PC) is transferred.

The main drawback of most of these systems is that custom code at both sides has to be developed for each device under test. [Kudlugi01] uses the Programming Language Interface (PLI), which allows a Verilog simulator to interface with external code, to provide some communication primitives based on Unix sockets to interconnect the DE with the DUT. But designers have to manually develop black boxes that use these primitives to redirect signals to the emulation system.

This problem is addressed by [Sarmadi02], that defines how to systematically write HDL code that uses PLI code to interact with the emulation part of the design. Nevertheless its approach is still manual and they report a maximum speedup of 56.

In [Çakir03] a semi automatic process to generate the communication layers between simulator and emulator is presented. A tool called ProtoEnvGen [Çakır01] is used for the generation. No speedup is reported.

Another usual drawback of co-emulation systems is that they are usually limited to the emulation of a single DUT. But complex designs would benefit from the possibility to

have several designs to test at the same time from a complex testbench. [Schumacher05] address this problem by defining Virtual Sockets to access multiple emulated circuits.

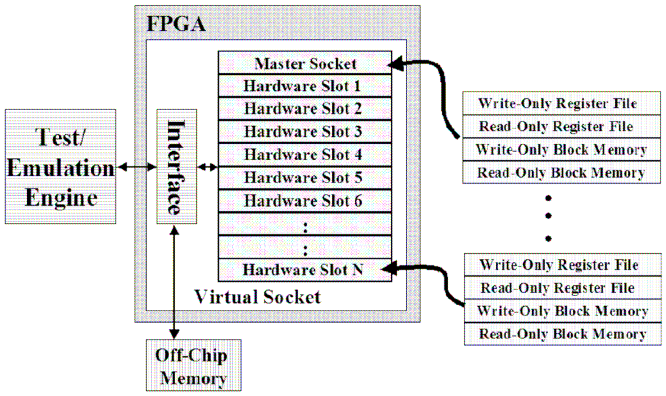

Figure 12 Virtual Socket Platform (from [Schumacher05])

Finally relatively recent works [Siripokarpirom04] have addressed the use of runtime reconfiguration to efficiently reuse the FPGA recourses to speedup large designs that would not fit in a single device.

Virtual Emulation

The term Virtual Emulation was first used in [Borgatti96] and later [Borgatti97] and [Dozza98]. This short-life term was used to define a very similar concept to Co-Emulation.

A virtual emulation system can be seen as a digital system made up of a prototype system implemented using available and off-the-shelf components and a virtual system implemented by a behavioral model running on a simulator.

These two systems communicate through a special purpose HW/SW layer implementing the Emulation Interface Border (see Figure 13). The virtual system is typically under development and its specification is not so detailed to make it synthesizable. The specialized HW/SW interface (the so-called Emulation POD) is implemented in an FPGA that converts electric signals from the prototype system in logic signals for simulator and vice-versa.

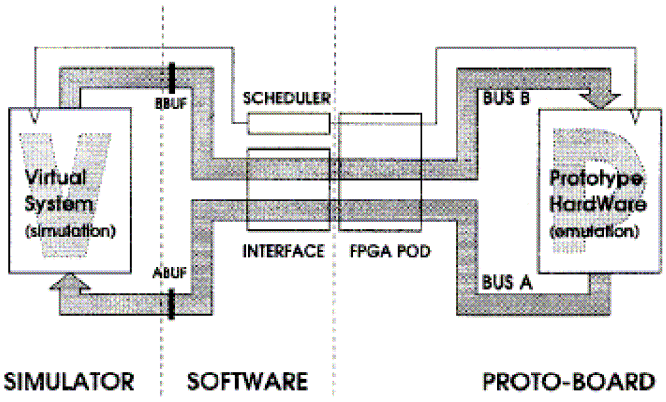

Figure 13 Virtual emulation (from [Borgatti96])

Bogartti also stresses the benefits of the incremental verification of the system. Starting from a completely behavioral description one can smoothly go to a complete hardware implementation while maintaining the same benchmarks and tools (Figure 14).

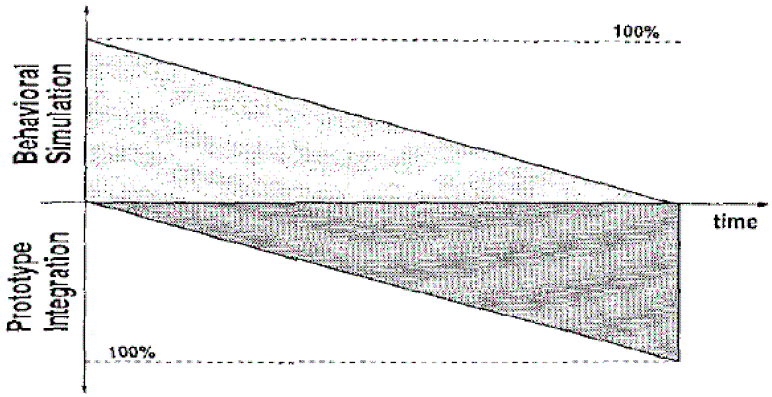

Figure 14 Path to complete hardware integration (from Borgatti97)

JHDL execution mode

As initially conceived in [Bellows98], in a JHDL design there is always a `HWSystem` object that, as its name implies, represents the whole Hardware System. It implements the simulation kernel that invokes behavioral descriptions during software simulation. In some systems it is also responsible to talk with the FPGA through calls to the necessary APIs and device drivers. The purpose of this communication could be the configuration of the device or the transmission of data, most of often to interface with a given functional unit programmed in the device. Using this transmission link, input and output ports of any JHDL circuit can be redirected to the FPGA device to interact with its real hardware implementation, allowing its execution in real Hardware (Figure 15 a).

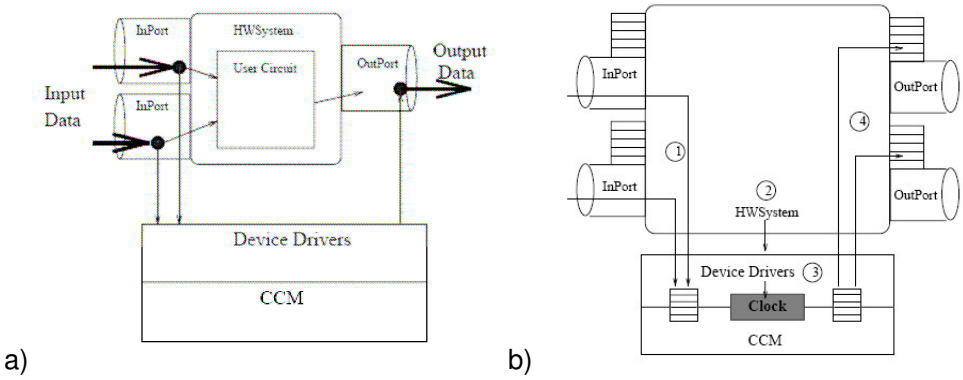

Figure 15 Operation of the Hardware Interface (from [Bellows98]). a) Dual mode dynamics b) Steps of an execution cycle

However to be able to perform this redirection the circuit has to be synthesized before and downloaded into the platform. This is done programmatically by instructing the `HWSystem` to download a given bitstream into the device. Figure 15 b shows the steps involved in advancing a clock cycle in a hardware circuit.

1. The inputs of the circuit are passed to the hardware implementation through the necessary calls to device drivers.

2. The HWSystem issues a clock step to the whole system that eventually calls the device to advance its clock.

3. The device advances a clock and buffers circuit outputs.

4. The outputs of the circuit are passed up to software and placed on the output ports software buffers.

Running the above algorithm to be able to interact with the synthesized circuit version has some price to be paid. The hardware platform must include some instrumentation so it is possible to copy input and output values and control the advance of the circuit clock.

Unfortunately [Bellows98] does not specify the details of how this is done in the Hotworks platform. Later [Hutchings99], while presenting how the support for the Annapolis Microsystems Wildforce platform is implemented, gives some clues about how the instrumentation is performed. Is not that any circuit can be downloaded into the hardware platform but just the ones that derive from a specified `pelca` class which represents a programmable element from the platform. The `pelca` class adds some instrumentation to communicate with the host system but the transmission of circuit output values after a clock cycle (step 4 of previous algorithm) is performed by using readback technology from Xilinx FPGAs. The configuration of the device is no longer programmed in the source code but a configuration utility (Figure 16) is provided to do it interactively.

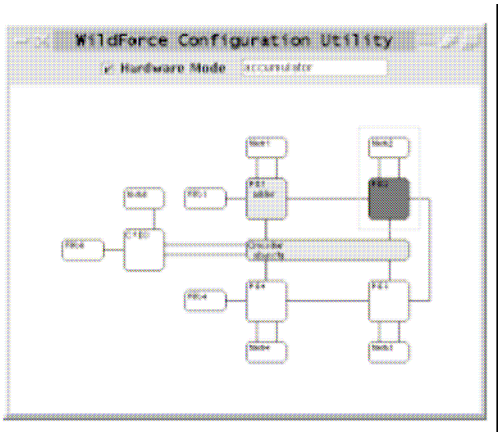

Figure 16 Wildfore configuration utility (from [Hutchings99])

Later work on JHDL [Hutchings00b], [Hutchings01], [Graham01], [Wheeler01], [Bellows04] keep on the same approach, more addressing Hardware Debugging rather than Hardware in the Loop Simulation.

Hardware in the loop with Simulink

Simulink is a graphical system modeling tool that provides a simulation environment addressed to continuous and discrete time systems modeling. Simulink was especially used for process control system modeling but with the years it has broaden its scope towards DSP and hardware systems design.

Simulink lets to graphically describe system components with interconnected modules, which can be composed of other basic modules or can be described behaviorally with Matlab S language or other languages like C and Fortran. To integrate Hardware in the loop simulation in Simulink there were to major problems to be solved.

1. How to generate Hardware from Matlab models ?

2. How to integrate generated hardware modules in the Simulation chain ?

There has been some research in creating HDL code from S code, [Banarjee99],[Banarjee00]. In fact this is a problem of behavioral synthesis from a high level language. However S language has some properties that make this task challenging. It is a dynamically typed language and programs usually rely on working on dynamically allocated multidimensional arrays and calls to a large preexisting library of functions. This approach has not been much successful. On the other hand in year 2000 Xilinx presented an alliance with MathWorks that yield to the launch of Xilinx System Generator [Xilinx00]. Its approach is not based on behavioral synthesis but on structural design and IP reuse, which is similar to a previous proposal found in [Krukowski99]. In System Generator a library of hardware blocks are provided by Xilinx. Hardware blocks are described twice: as S functions, which can be integrated in Simulink, and VHDL code, which can be synthesized to a Hardware platform. The blocks can be simple, as primitive logic cells, or complex IP cores, like an FFT circuit. By combining blocks the designer can implement and test a Hardware design from Simulink and create its Hardware implementation.

Several IP blocks are provided: FFT, FIR filters, Multipliers, etc. Users can also integrate their own existing VHDL code with a black-box model. The reported speedups (Table 1) depend on several factors like complexity of the design and the synchronization scheme.

Table 1. Simulink Hardware in the loop co-simulation results (from [Shirazi03])

Application	Software Simulation	Hardware Execution	Speedup Factor
5 x 5 Image Filter	170 s	4 s	43
Cordic Arc Tangent	187 s	27 s	7
Additive White Gaussian Noise Channel (AWGN)	600 s	80 s	7.5

It is relevant to know that Hardware in the loop simulation is provided in conjunction with platform manufacturers, because they have to provide all the middleware that allows the communication between the simulation kernel and the circuit under test. In 2000 only few hardware platforms had support for Simulink ([Alpha], [Lyr]) but nowadays there are a large number of them from different manufactures (Annanapolis, Nallatech, Lyrtech, etc.).

Ptolemy

The Ptolemy project [Lee01],[Lee01b] is focused in modeling, simulation and design of concurrent embedded systems. Ptolemy is aimed to model various technologies, like mechanical systems, analog electronics, digital systems and software. Several computing domains are defined so that the models can accommodate the different properties of the different technologies, especially regarding their notion of time.

Ptolemy is an actor-oriented design framework [Lee03], meaning that it is centered in modeling the actors that interact in a system and not constraining their models of computation. Moreover Ptolemy II allows domain-polymorphic definitions and the integration of hierarchical heterogeneous domains (Figure 17).

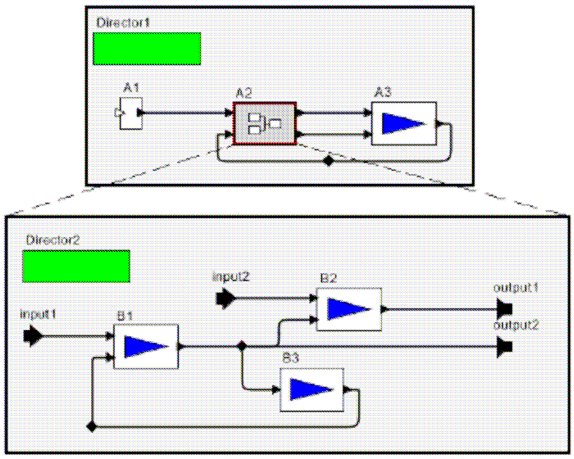

Figure 17 A hierarchical model in Ptolemy II (From [Liu04])

Ptolemy can describe actors in a variety of languages as programming languages tend to be designed for a certain computing domain. Although there is no included support for Hardware in the loop simulation in Ptolemy there is some interest by Ptolemy community in having this feature. In fact, the group of Indrusiak is working on this topic. [Indrusiak03] describes a method to integrate remote actors into a Ptolemy system. Since remote actors execute in different process contexts, their implementation can be anything that communicates successfully with their proxies, including a Hardware circuit running on an FPGA platform. This is a good approach to share a limited resource like an expensive FPGA board for educational purposes [Jimenez05] and can be also useful to design complex systems like a WCDMA receiver [Indrusiak05]. The development process is similar to the process imposed by Simulink based tools (Figure 18). A model in Ptolemy is created and it is refined in several iterations first to convert from floating point to fixed point, then to create its equivalent JHDL model and finally to netlist the resulting circuit.

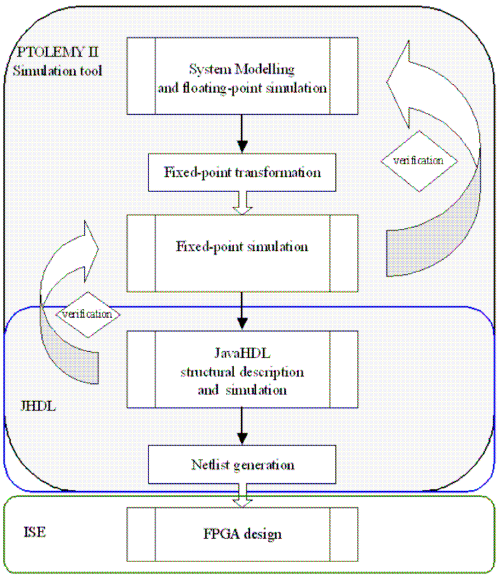

Figure 18 Design flow for Hardware integration in Ptolemy (from [Indrusiak05])

However, this kind of integration heavily relies on manual processes, especially in the interface between Ptolemy and JHDL. Figure 19 gives an idea of the coding that must be developed to intercommunicate both worlds.

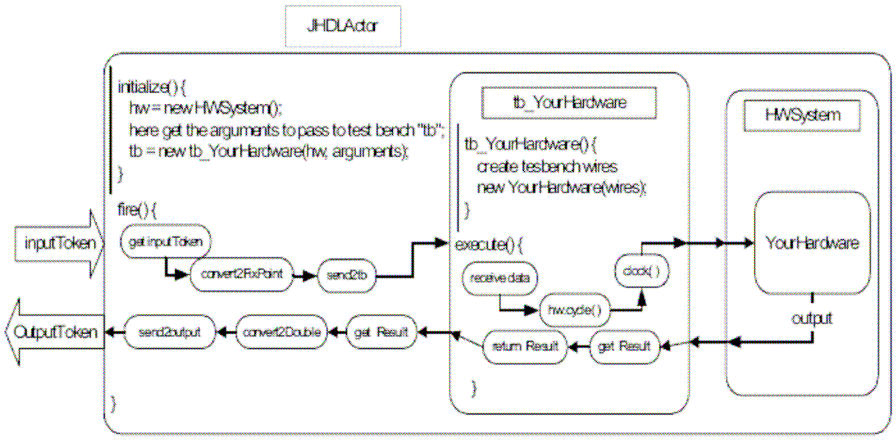

Figure 19 Interface between Ptolemy and JHDL (from [Indrusiak05])

Despite the availability of the "Hardware Mode" of JHDL that would allow the execution of JHDL based actors on FPGA platforms, so eventually achieving the goal of executing Hardware in the simulation loop, to my knowledge this has not been done.

To sum up Ptolemy provides a very powerful environment to design in various computing domains but, by now, it has an excessive manual approach to integrate Hardware in the loop.

Hardware Debug

DSP designers can aggressively shorten simulation time of complex systems with HIL simulation. However, HIL simulation it is not the only application of hardware emulation. Some ASIC prototyping systems, like the products from Quickturn [Butts92],[Quickturn] and Mentor Graphics [Mentor] (originally IKOS), emulate large ASIC designs on FPGAs. Since a single FPGA does not have enough capacity to embed a typical ASIC design, these systems use multiple interconnected FPGAs enclosed in a big case and controlled by a host computer via a high bandwidth link. The drawbacks of these systems are that they are very expensive and need some expertise to be used. Additionally the communication between hardware and simulation kernel is often based on transactions and usually not transparent to designers.

Another related topic is the Hardware Debug concept [Tombs04],[Graham01] which pursues to offer the equivalent features of Software Debuggers for Hardware design. The main goal of a Hardware Debugging system is to allow detecting and removing bugs from a design, so speed is not the central point, although an important one. Software designers debug by running step-by-step, setting breakpoints, adding traces, watching variables, and modifying values while debugging. These features can be formalized as interactivity, controllability and observability. Hardware debuggers, for instance, should allow controlling the clock (or clocks) execution, to watch any part of the circuit, to add breakpoints (triggers), to add traces, and to modify register or memory contents. Simple versions of some of these functions are being integrated in EDA design flows. Embedded logic analyzers like Altera Signal Tap [Altera01] and Xilinx Chip Scope [Xilinx00b] follow a simple approach to enable the acquisition of signal values over some time after a triggering event has been reached.

Chapter 3

The JHDL Framework

JHDL is a design environment that provides a Java API for describing FPGA circuits in a constructive way (mostly bottom-up) as well as a collection of tools and utilities for their simulation and hardware execution.

Circuit Modeling

In JHDL circuits are described as Java classes that follow a given design style. The typical different levels of abstraction used by HDLs are specified by deriving classes from specific base classes and interfaces. The basic class hierarchy is shown in Figure 20. `Cell` class is the main class that represents a hardware block with an I/O interface. There are two different Java interfaces: `Clockable` and `Propagateable`, which denote sequential and combinational logic respectively. `Cell` class has a number of subclasses that map more specifically the nature of the circuits. `CL` class is a `Cell` derived class that represents a completely combinational circuit, so it implements the `Propagateable` interface. `Synchronous` class is a `Cell` derived class that represents a completely synchronous circuit, so it implements the `Clockable` interface. It is mandatory that `CL` and `Synchronous` derived classes provide a behavioral model and they are often used internally in JHDL to represent primitive logic from the hardware devices.

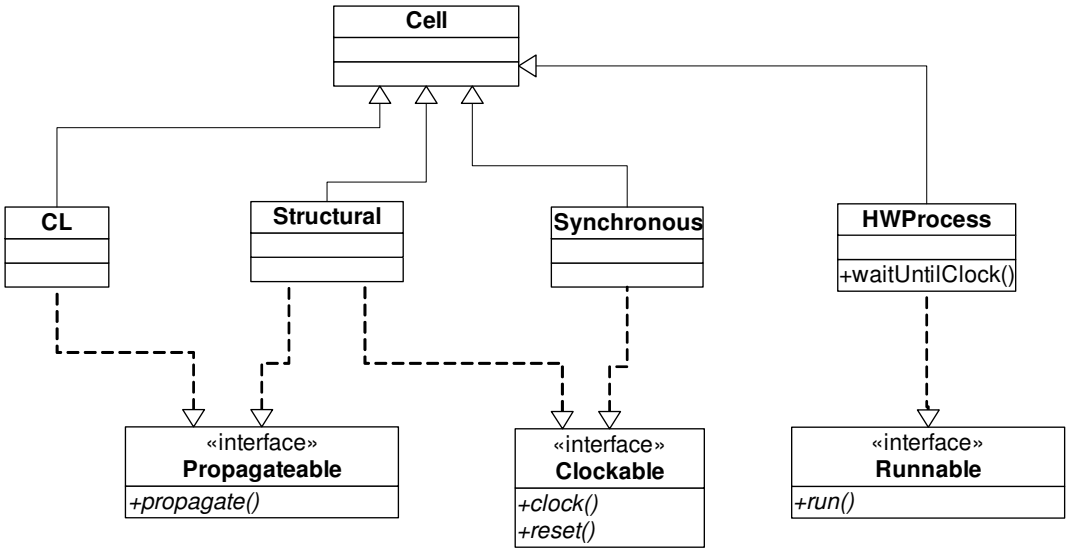

Figure 20 Hierarchy of JHDL circuits

User defined designs consist on the instantiation of existing cells following a constructional method. Moreover, their behavior can be inferred from their components. For this reason, there is an additional Cell derived class named `Structural`. As building blocks can be either combinational or synchronous `Structural` class implements both `Clockable` and `Propagateable` interfaces but it is not mandatory to provide a behavioral model since it can be inferred from containing cells.

The additional `HWProcess` class allows to describe circuits by only providing a behavioral model based on a sequential description in which timing is specified by calling the `waitUntilClock` function. Since there is only a behavioral model, and they never map to primitive logic from the hardware device, `HWProcess` derived classes are not synthesizable.

Whatever is the base class of a circuit, a behavioral model consists in having a programmatically way of driving the circuit outputs, i.e. assigning values to the output of the system depending on the values of the inputs and an internal state. Inputs and outputs in JHDL are represented by a unique class named `Wire`. Wires also connect different circuits. One can examine the value of an input wire by calling a `get` method and can assign a value to an output wire by calling a `put` method. There are a number of variants of `get` and `put` methods depending on the width of the wires.

Circuit Verification by Simulation

Once a circuit is compiled it can be simulated by two methods. First, a custom testbench can be developed by implementing the `TestBench` Java interface. Simulation kernel is exposed as an object and can be easily controlled from testbenches to feed stimuli to the circuit under test, extract and display outputs and control execution by explicit invocation of clock advance functions.

In the following example code a testbench is build to verify a median filter design. In the `execute` method `HWSystem clock` method is called to advance the system clock after the data for the inputs of the system have been updated.

```
package org.cephis.MedianFilter;

...

public class tb_MedianFilter extends Logic implements TestBench
{
      static HWSystem hw;

      Wire in[] = new Wire[9];
      int v[] = new int[9];

      ...


      public static void main(String argv[])
      {
            hw = new HWSystem();

            tb_MedianFilter tb = new tb_MedianFilter(hw);
            tb.execute();
      }

      public tb_MedianFilter(Node parent)
      {
            super(parent);

            for (int i=0; i < 9 ; i++) in[i] = wire(8, "in"+1);
            median = wire(8, "median");

            design = new MedianValue(this, in[0], in[1], in[2], in[3], in[4], in[5], in[6], in[7], in[8], median);
      }

      public void execute()
      {
            FileInputStream fis = new FileInputStream("c:\\test.jpg");
            JPEGImageDecoder decoder = JPEGCodec.createJPEGDecoder(fis);
            BufferedImage img = decoder.decodeAsBufferedImage();
            BufferedImage dst = new BufferedImage(img.getWidth(), img.getHeight(), BufferedImage.TYPE_INT_RGB);

            int shift = 0;

            addSaltAndPepperNoise(img, 0.2);

            for (int i=0; i < 9 ; i++)
                  v[i] = 0;

            getSystem().cycle(1);

            for (int mask = 0xFF; shift <= 16; mask <<=8, shift +=8)
                  for (int y=1; y < img.getHeight()-1; y++)
```

```
                    for (int x=1; x < img.getWidth()-1; x++);
                    {
                         fillKernelValues(v, img, x, y);

                         getSystem().cycle(1);

                         int rgb = dst.getRGB(x, y);

                         rgb &= ~mask;
                         rgb |= (median.get(this) << shift) & mask;

                         dst.setRGB(x,y,rgb);
                    }
          }

          public void reset()
          {
               for (int i=0; i < 9; i++)
                    in[i].put(this, 0);
          }

          public void clock()
          {
               for (int i=0; i < 9; i++)
                    in[i].put(this, v[i]);
          }

          ...
}
```

Second, a circuit can be loaded into the interactive simulation environment (`DynamicTestBench`, DTB), which provides several facilities for exercising and viewing the state of the circuit during simulation. DTB includes a hierarchical circuit browser with a tabular view of signals, a schematic viewer that include values of signals, a waveform viewer, a memory viewer and a command line interpreter (see Figure 21).

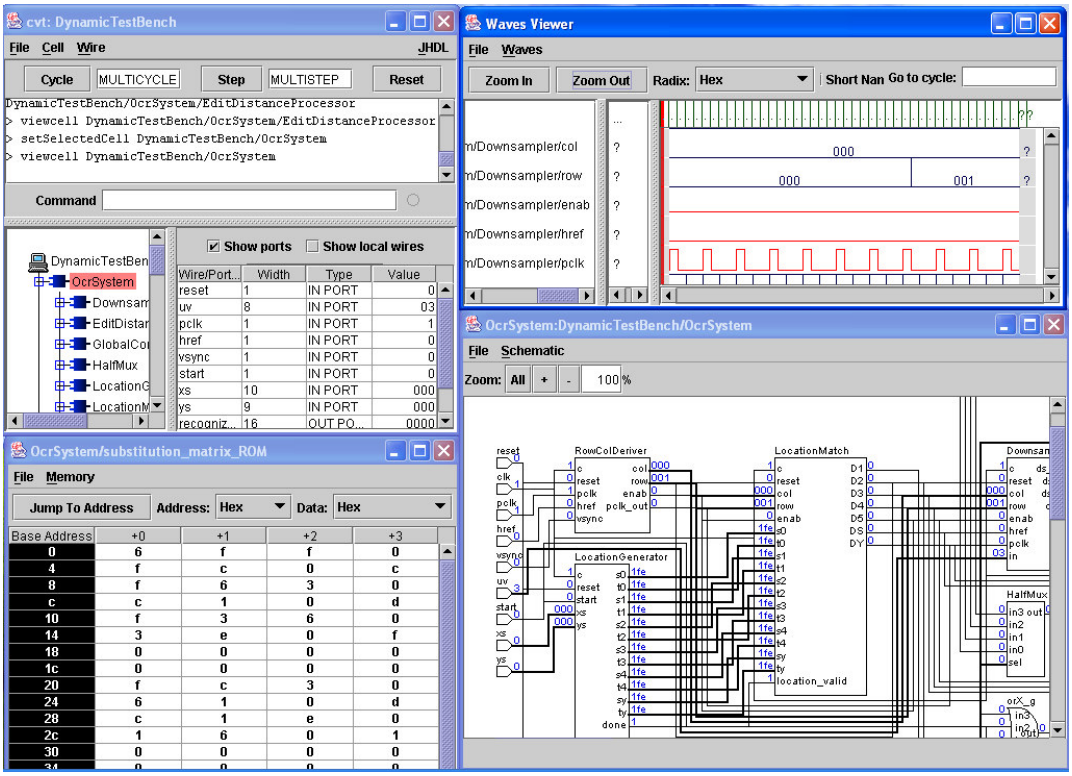

Figure 21 From left to right and up to down: a) DTB which includes a hirerarchical circuit browser, interface description table and command line interpreter b) Waveform viewer c) Memory viewer d) Schematic Viewer

Since all tools are public and available from the designer perspective, more complex hybrid testbenches can be developed. For instance, a `TestBench` derived class could instantiate the schematic viewer and waveform viewer for easy visual inspection of results while programmatically feeding stimuli. On the other hand, DTB based simulations could include some behavioral modules that generate complex stimuli or display results in a custom way.

Simulation Engine

Most HDL simulators are based on an event-driven approach. In fact, as stated in [Kulmala], VHDL and Verilog languages semantics assume there is an underlying event driven simulator. An event driven simulator is based on the existence of a queue of pending events, called *Time Wheel*. An event has two components: value of a signal and time. At each simulation iteration the simulator takes the head of the *Time Wheel* and evaluates all the dependant circuits that the event could trigger. Circuits that depend on a signal are obtained from the sensitivity list that is always defined in VHDL and Verilog designs. The evaluation process can probably cause the insertion of new events in the *Time Wheel*. Events are always inserted in the *Time Wheel* in time order. Simulation ends when the *Time Wheel* contains no pending events.

A simple example of the simulation dynamics is depicted in Figure 22. In this example the *Time Wheel* is initialized with the test vectors for signals a, b, c. The first event in the *Time Wheel* indicates that signal b is changed at time t1. When the simulator takes this event it must look for all circuits that depend of signal b. The sensitivity list of the behavioral model of the NAND2 gate includes the signal b, so it is processed and a new event, associated to the output, is inserted in the *Time Wheel*. The same algorithm is repeated until no events are pending in the *Time Wheel*.

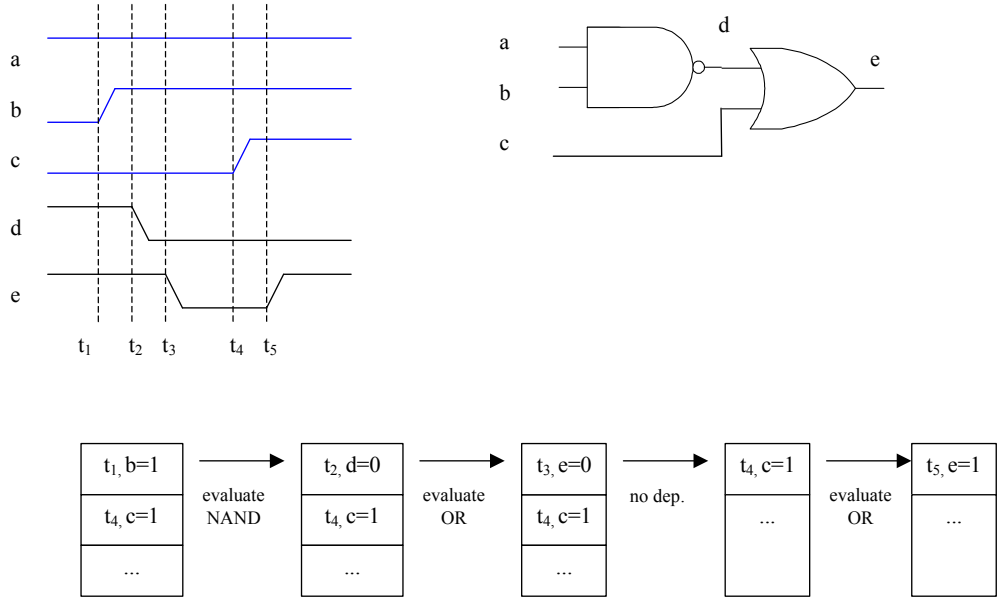

Figure 22 Event driven simulation of a simple circuit

Several events can happen at the same time, so the order of evaluation of events is critical for the accuracy of the results. If behavioral models of the circuit to test include

time semantics the system can be verified with some time precision. On other occasions a zero delay is assumed.

On the other hand JHDL has a cycle-based simulation engine (CBSE). In cycle based simulators time advances at discrete intervals, i.e. clock cycles. Combinational logic is assumed to be zero delay and synchronous logic has a delay of one clock cycle.

The JHDL simulator has to differentiate between synchronous and asynchronous circuits, and as a consequence between synchronous and asynchronous wires. This allows the simulator to perform the correct method of value propagation to each circuit wire. The model that allows the simulation of a JHDL circuit is build at the same time that the circuit is build. In fact the simulation system is tightly coupled with the circuit modeling and simulation structures are created and maintained even though there is no intention of simulation. This approach is totally different to other simulators like ModelSim that are totally uncoupled to circuit modeling.

A fundamental class in the JHDL framework is the `ValuePropagater` class that models a channel that can propagate a value between two endpoints. A `ValuePropagater` is associated to each line of each wire. During circuit building the `BuildListManager` class is responsible for keeping track of all propagators of the system that are stored in the `all_value_propagaters` member variable. This array is populated progressively as wires are connected to Cells (Figure 23).

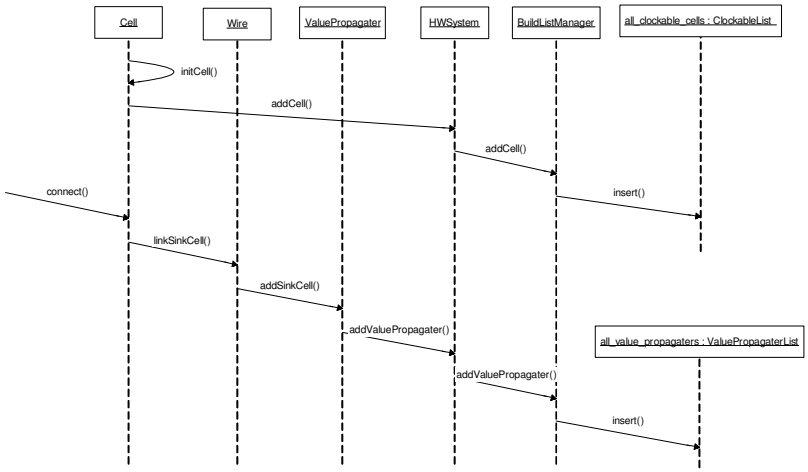

Figure 23 UML Sequence diagram that causes the population of clockable and value propagaters lists

When simulation is initialized all the `ValuePropagaters` are classified depending on the nature of the cells that drive them. This process is performed by `PropagateManager.topologicalSort` (Figure 24).

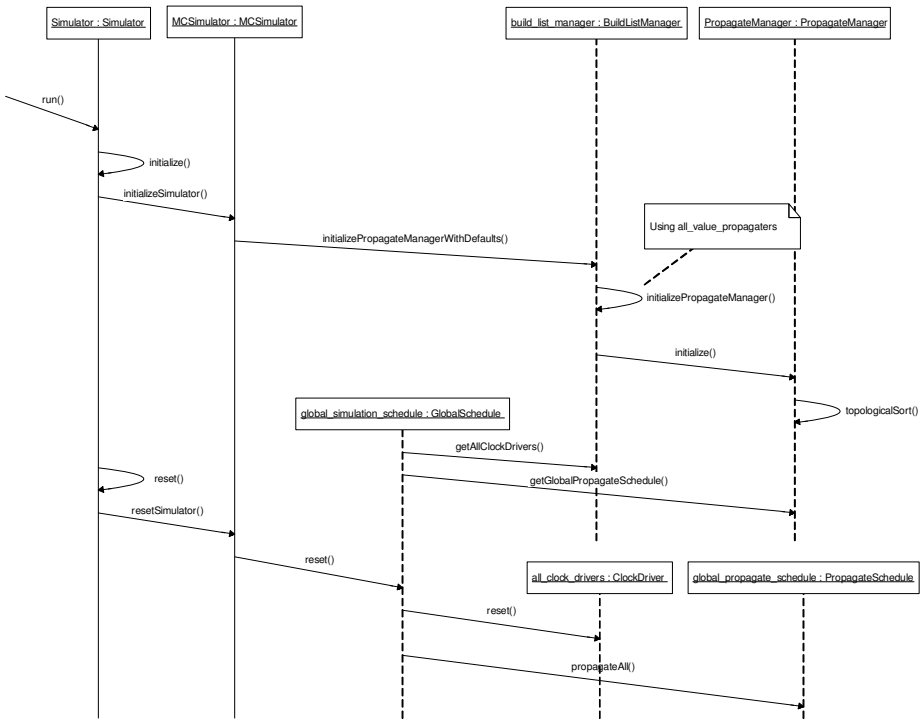

Figure 24 UML Sequence diagram of simulation initialization

This classification is important to handle in different ways the different types of sources. For instance, constant cells do not vary and there is no reason to evaluate their value at every clock cycle, so they are handled different from the rest of the elements of the circuit. Besides classification, the `topologicalSort` function also builds a directed graph with the dependency of the different `ValuePropagaters`. This is useful for propagating the results of combinational logic since all cells are evaluated in the order of the list. This is also the major reason why asynchronous loops are not allowed in JHDL.

The following example illustrates how the topological sort is performed and how this affects the simulation. Let's consider we have a very simple code in a JHDL circuit that creates a couple of registers and some simple logic gates. We assume that signals `in0`, and `nor` are the input and output signals of the circuit respectively. The schematic view of this very simple circuit is shown in Figure 25.

```
...
Wire in0 = wire("in0");
new Stimulator(this, new Wire[]{in0});

Wire xor0 = wire();
Wire nor = wire();

Wire reg1 = reg(in0);
Wire reg2 = reg(nor);

xor_o(reg1, reg2, xor0);

Wire or0 = or(reg1, reg2);
Wire and0 = and(in0, or0);

nor_o(reg1, and0, nor);
...
```

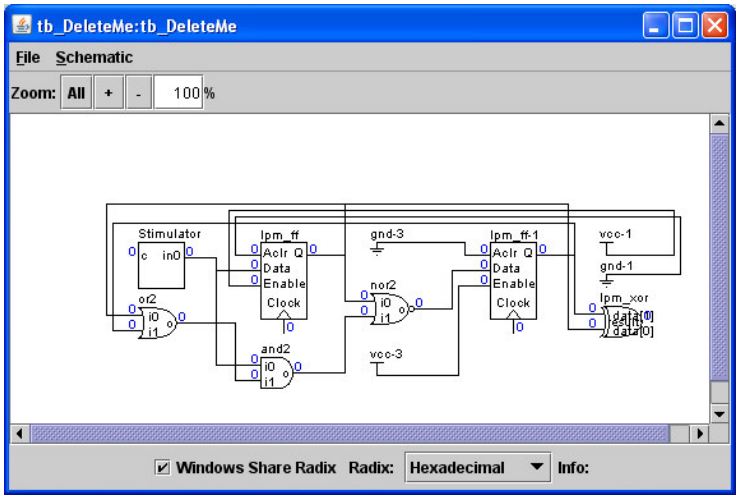

Figure 25 Simple circuit diagram

As mentioned before, the topological sort during simulation initialization would classify the different elements of the circuit, including Wires and Cells. The resulting graph is shown in Figure 26, Cells are yellow colored while Wires are white colored. Three main groups would be created for this circuit: constant cells, clockable cells, and propagatable elements. Constant cells would contain all the constants of this circuit, just power (VCC) and ground (GND) connections. Since there are only two flip-flops in this circuit, clockable elements would contain only references to lpm_ff and lpm_ff-1. Finally, we would have the list of the propagatable elements of the circuit. As the list has been build taking dependencies into account the simulator can evaluate each element in order and be sure that no inconsistency occurs.

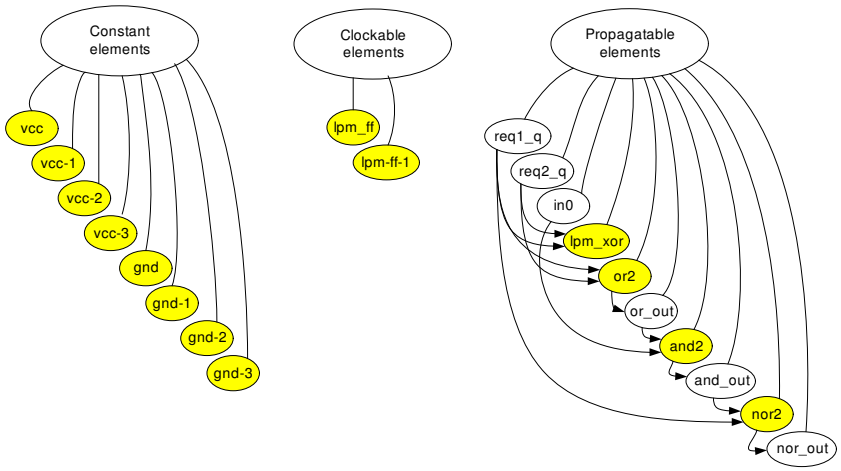

Figure 26 Resulting graph from topological sort

The simulator evaluates all synchronous blocks before propagating the asynchronous elements (see Figure 27). The order of evaluation of the synchronous blocks is irrelevant since the values they compute is not made public to the rest of the circuits until Wire propagation occurs as part of the propagation of asynchronous elements.

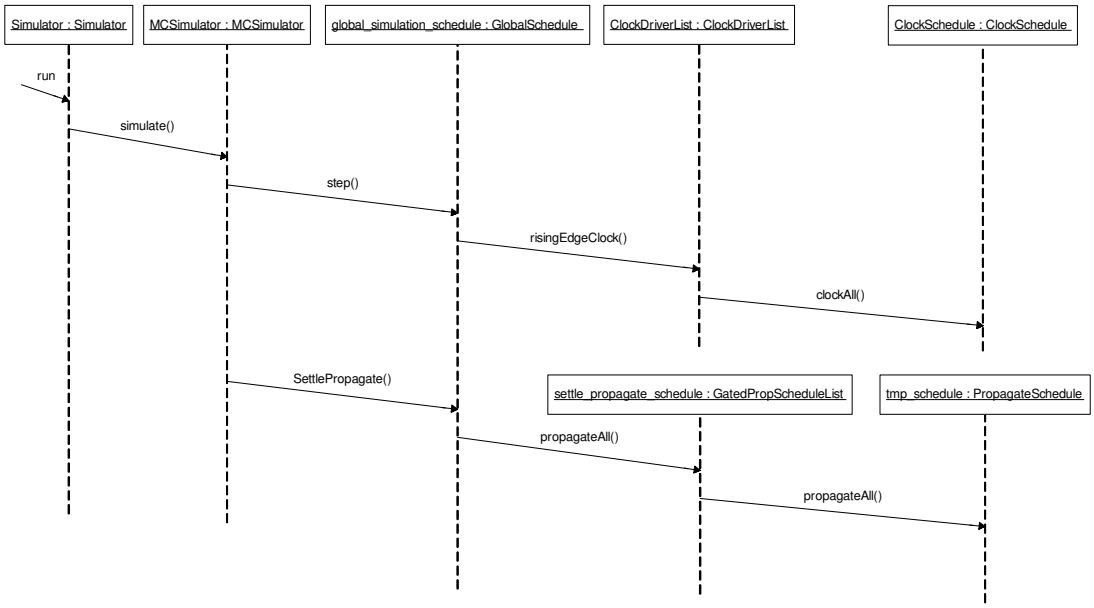

Figure 27 UML Sequence diagram of clock cycle simulation

Synthesis

JHDL allows synthesizing circuits created by the user. The synthesis method is based on generating an EDIF [Edif] netlist from the circuit model to be used by FPGA provider for final Place & Route.

The EDIF (Electronic Design Interchange Format) format is a data interchange format defined by the Electronic Industries Alliance (EIA) and US based industry association to make CAD tools interoperable.

As previously described, the internal circuit model consists of a hierarchical tree of cells connected by wires. The EDIF format completely matches this model, so the generation of EDIF files from the model is straightforward.

The EDIF system describes interconnections in text format by using reserved keywords (or tags) that are organized hierarchically. Being a text based hierarchical format, it has similarities with XML and HTML.

The following example code shows the netlist of a simple cell in EDIF 2 0 0. As seen below the andX_g_1 cell defines two input ports and one output port and instantiates a primitive cell and_2, which is part of the primitive elements of the FPGA library.

```
(cell (rename andX_g_1 "andX_g_1")
   (cellType GENERIC)
      (view view_1
         (viewType NETLIST)
         (interface
         (port in1  (direction INPUT))
         (port in0  (direction INPUT))
         (port out  (direction OUTPUT))
      )
      (contents
         (instance andX
            (viewRef view_1 (cellRef and_2))
         )
         (net (rename in1 "in1")
            (joined
               (portRef (member i 1) (instanceRef andX))
               (portRef in1)
            )
         )
         (net (rename in0 "in0")
            (joined
               (portRef (member i 0) (instanceRef andX))
               (portRef in0)
            )
         )
         (net (rename out "out")
            (joined
               (portRef o (instanceRef andX))
               (portRef out)
            )
         )
      )
   )
)
```

JHDL cannot create the final bit-stream to program the FPGA, it is mandatory to use the tools provided by the manufacturer of the device, e.g. ISE for Xilinx devices. This is not a problem of JHDL but the result from the industry tactics who is very reluctant to make the bit-stream format publicly available.

Execution

JHDL offers an integrated simulation/execution environment [Hutchings01] meaning that designer can use the same facilities when working in simulation mode and when working in hardware mode. For instance, clock control and schematic viewer, whose signal value annotation should be available on both modes. These features are based on the following facts:

1) Xilinx devices allow retrieving the state of the complete configuration memory, including flip-flop states, through readback.

2) JHDL classes that represent stateful device primitives, like flip-flops, implement the `ExternallyUpdateable` interface, so when the simulator kernel is running in hardware mode only updates their value after retrieving readback data.

A drawback of this approach is that is limited to devices that support readback or an equivalent technology, so at the end is limited to few Xilinx devices. A more general approach consists in instrumenting the designs with scan chains [Wheeler01b] to be able to access all circuit flip-flops independently from the kind of used device. Unfortunately, the cost in area overhead can be very high, from 30% to 100%, and speed is degraded by 20% in average.

JHDL hardware execution model provides a method to transparently update state of the model from the executing hardware but lacks a method to update the state in the other direction, which might be based on JBits [Ballagh01], [Poetter04]. This drawback is solved by providing a transaction-based model for each current supported hardware platform, i.e. testbenches communicate with circuits through register read/write operations. This makes difficult to incrementally test parts of the design on its hardware implementation because the interface should be redesigned in each iteration.

Additionally JHDL hardware execution model requires having a bitstream of the design to be downloaded into the hardware platform but the invocation of Place & Route tools to produce this bitstream is not included in the design flow.

Supported Platforms

JHDL has support for few hardware boards. Some info of supported platforms can be obtained from http://splish.ee.byu.edu/lab/ but most of the detailed info is spread in several research papers that describe applications implemented on them.

The HotWorks Platform

The initial paper from JHDL [Bellows98] has references of support for the HotWorks platform, a PCI board from Virtual Computer Corp.

Unfortunately, there is very little information about the details of how this platform was supported in JHDL.

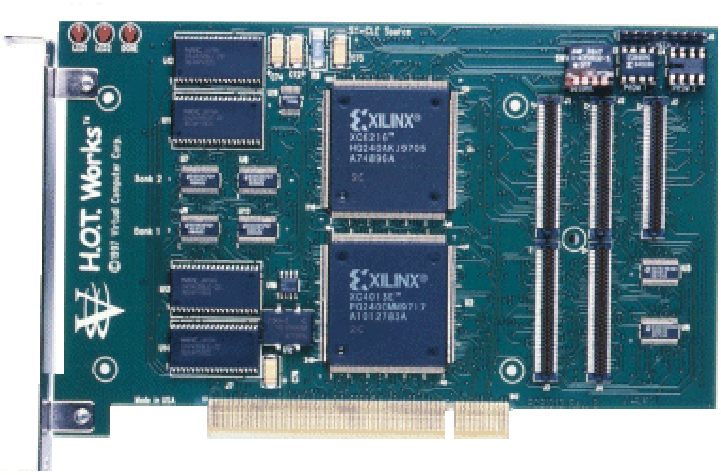

Figure 28 VCC's Hotworks platform

The SLAAC1 Platform

The *Systems Level Applications of Adaptive Computing* project was leaded by Information Sciences Institute of the University of Southern California. As stated in their website, the mission of the project was to create an open, standards-based, scalable, COTS based reference-platform that could be used for high performance demanding defense applications.

The SLAAC1 platform (Figure 29, Figure 30) was build as part of the project. It consists of an FPGA-based accelerator on a full-sized 64-bit PCI board containing a user-programmable Xilinx 4085 device, two user-programmable Xilinx 40150 devices, and ten 256Kx18 100MHz ZBT synchronous SRAMs

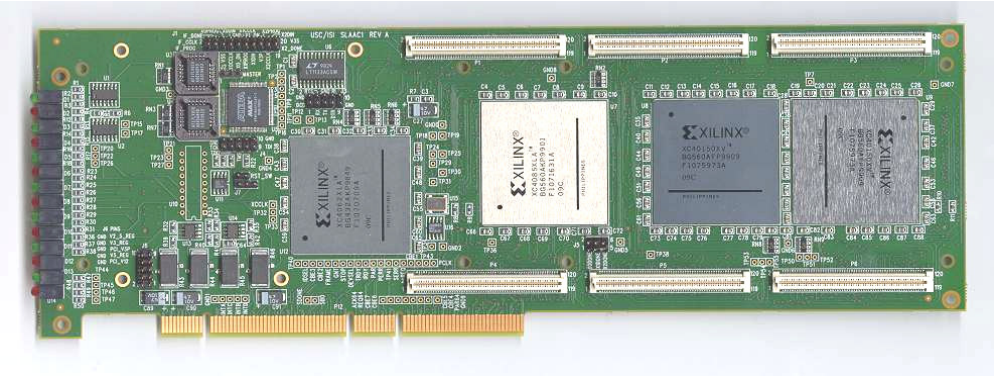

Figure 29 SLAAC1 platform

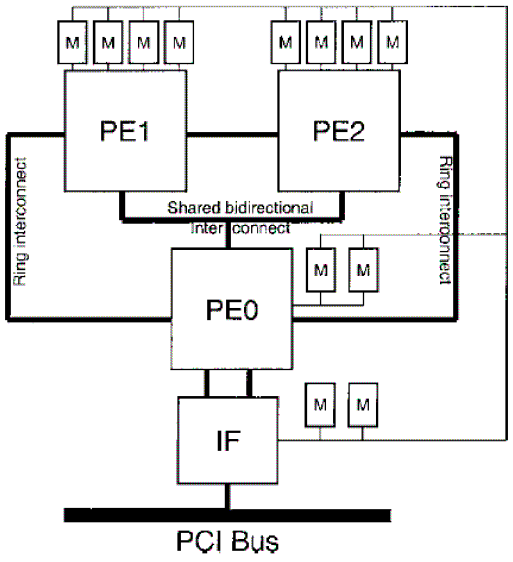

Figure 30 SLAAC1 block diagram (from [Hutchings04])

To implement a hardware design using in the SLAAC platform, and be able to simulate and netlist it, the design class must extend the super class `pelca` and define the input/output port of the circuit first.

Simulations and executions can be controlled automatically from programmatic testbenches or manually through a graphical user interface (Figure 31). Communication with the host is possible through the IF FPGA.

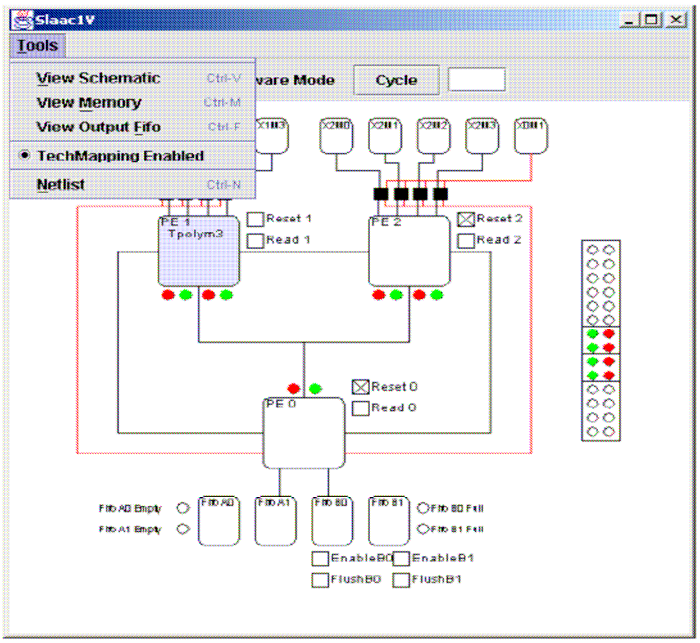

Figure 31 JHDL graphical interface to control SLAAC1-V (from [Ma03])

The Wildcard Platform

Annapolis Micro Systems Inc. manufactures various FPGA based boards for rapid prototyping and educational purposes. The Wildcard board (Figure 32) is a CardBus board for which there is a JHDL execution model. However the available model does not support readback.

Figure 32 Wildcard board

As shown in the logic block diagram (Figure 33) the board contains a processing element (Virtex FPGA) connected to two memory chips, and has two I/O banks and a bus (LAD, Local Address Data Bus) connected to the CardBus interface.

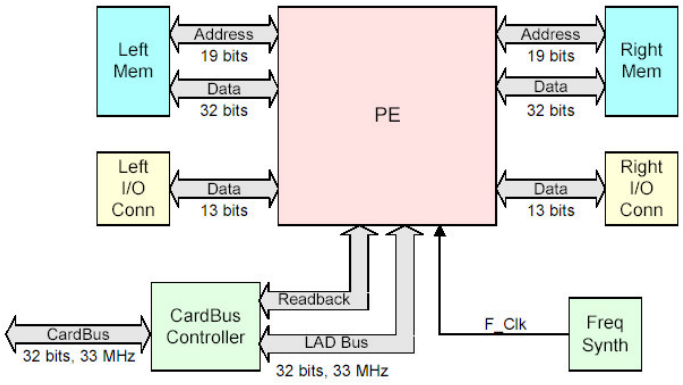

Figure 33 Wildcard logic diagram

The JHDL platform model contains the `WCBoard` class describing the board as a whole. The application specific circuits contained inside the board (like memories and bus controller) are exposed to the JHDL user as behavioral modeled circuits. In fact the user designs can only implemented into the processing element (PE).

To do so, the user must implement a Java class extending the `LogicCore` class. The elements external to the PE can be accessed through the PE interface, i.e. their input/output pins. Some helper interfaces are made available to ease the design.

The communication with the external world is achieved by going through the LAD bus, which is transaction based. Since all the fixed functionality hardware circuits (all but the PE) have alternative behavioral models, the user can simulate a host/board system. When execution mode is used the real hardware is used obtaining a significant speedup. Since the Wildcard platform does not support readback when executing in hardware mode the visibility of the total circuit state is lost.

The Osiris Platform

The Osiris platform (Figure 34) is another internal platform developed by USC/ISI and later commercialized by CoreTech, a division of Atlantic Coast Telesys.

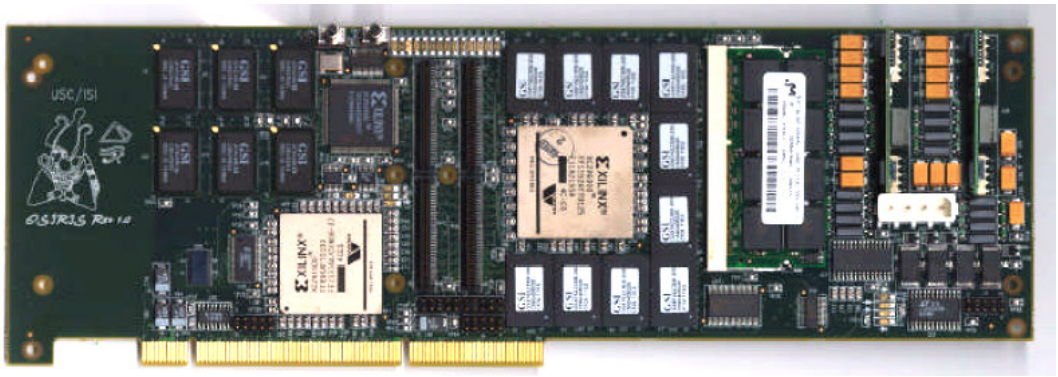

Figure 34 The Osiris platform

The Osiris platform uses a large FPGA that connects with large SDRAM memory and some ZBT RAM modules. It also integrates a current and thermal monitoring system.

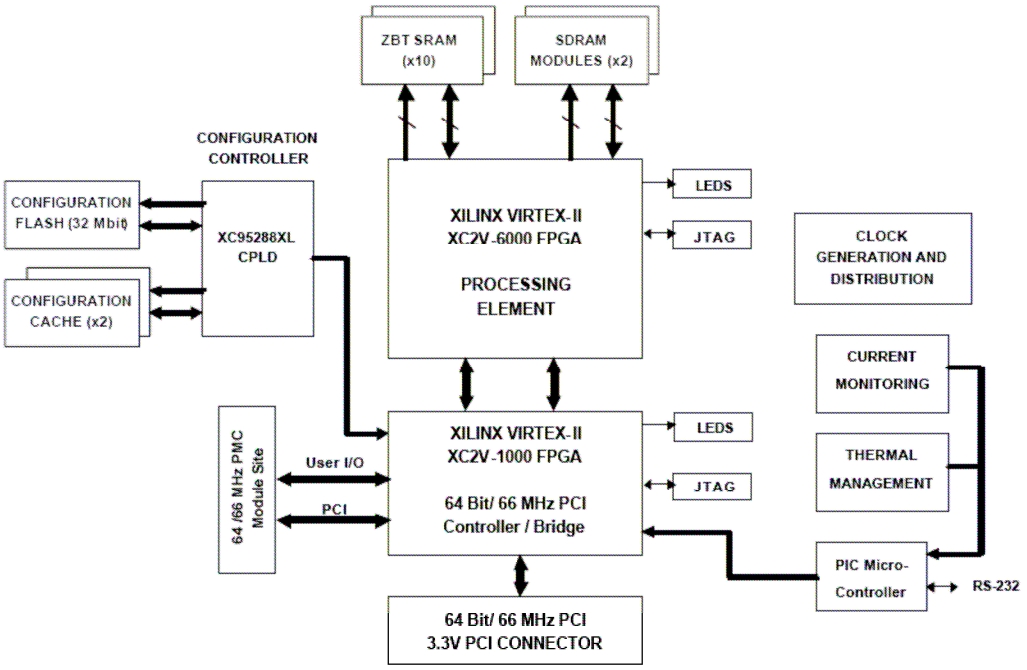

Figure 35 Osiris logic diagram

The platform is supported by JHDL [Osiris] by providing the typical behavioral models of the non-programmable blocks of the board. The user is not forced to extend a particular class to implement a circuit, but only to conform to a given interface, i.e. a list of the defined input and output pins. The readback support is missing so the state of the system can be obtained indirectly through the transactional interface.

Summing up, several platforms are available that support the execution mode of the JHDL framework. They provide a smooth method to go from circuit simulation to real system execution while offering a good level of observability when using the readback technology. JHDL execution mode aim is to offer a final execution environment for hardware designs. However, the JHDL execution mode is unacceptably dependent on technologies non-universal to different FPGA manufacturers (like readback). Another great problem is that it usually forces to identify the elements that where produced by the final Place & Route process and reference each one to their original design entities. Place & Route processes are often tightly integrated with the Synthesis process, and they are doing a better and better job to get rid of unused logic or refactor circuits to more efficient ones. So at the end you can download a bitstream that implements a functionally equivalent circuit but use different resources that you initially planned. In this situation the tools have trouble to offer valuable information.

In the following section I propose a different approach to provide hardware execution in a broader range of platforms. However the aim is not to offer the final execution platform but one that you can use to speedup simulations during your design process. It comes for free that you can eventually use it as the final execution platform.

Chapter 4

Jumble: a proposed Hardware Execution Model

The execution model proposed by JHDL has an important drawback: it is tightly coupled to the underlying hardware platform. The user has to design specific testbenches for the given platform, in which the platform is explicitly referenced. The user often knows the FPGA pin-out and uses it to access external resources or communicate with the Host. Moreover, often the circuit to execute in hardware must extend a particular class, e.g. `pelca` or `LogicCore`.

This approach can be examined from the following point of view: the platform manufacturer provides good simulation models of the board, and simulation environment is augmented so it can switch from board simulation to board execution in a very easy way as depicted in Figure 36. In this model the user is designing a board application, and since all the examined platforms are connected to a PC host through a particular flavor of PCI, probably a PC accelerator.

This kind of board application uses an `HWSystem` object, which can be executed in either hardware mode or simulation mode. This causes to switch between behavioral circuit models or the interfaces to the real hardware. In most JHDL supported platforms the necessary synthesis, bitstream generation and configuration steps are not fully automated and user intervention is also needed in this step.

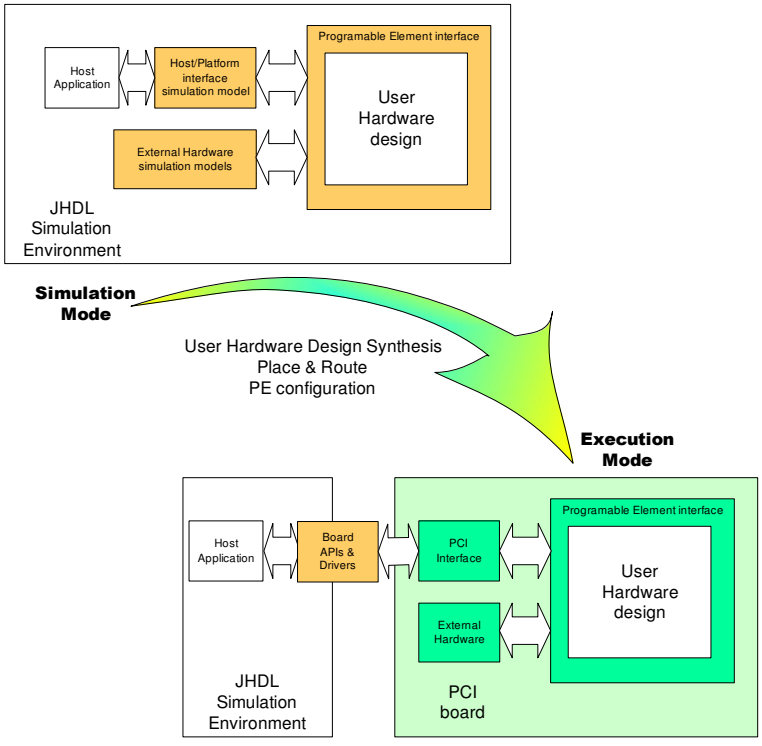

Figure 36 JHDL execution mode

Jumble proposal is more radical in the sense that it tries to hide as much as possible the existence of a Hardware platform to the designer. Instead of developing a hardware accelerator bound to the platform, the user implements a hardware design totally unrelated with the platform. The user is not forced to conform to a given interface and creates the exact same design that would create without having the Jumble simulation feature. When a certain block (target) of the design is desired to run in hardware the Jumble tool automatically creates the logic to implement the selected block in the programmable element of the available platform. The target circuit is synthesized and downloaded to the platform, but some logic is added to make the communication with the simulation possible. On the software side, the target simulation block is substituted by a redirector that performs the communication with the hardware implementation of the target.

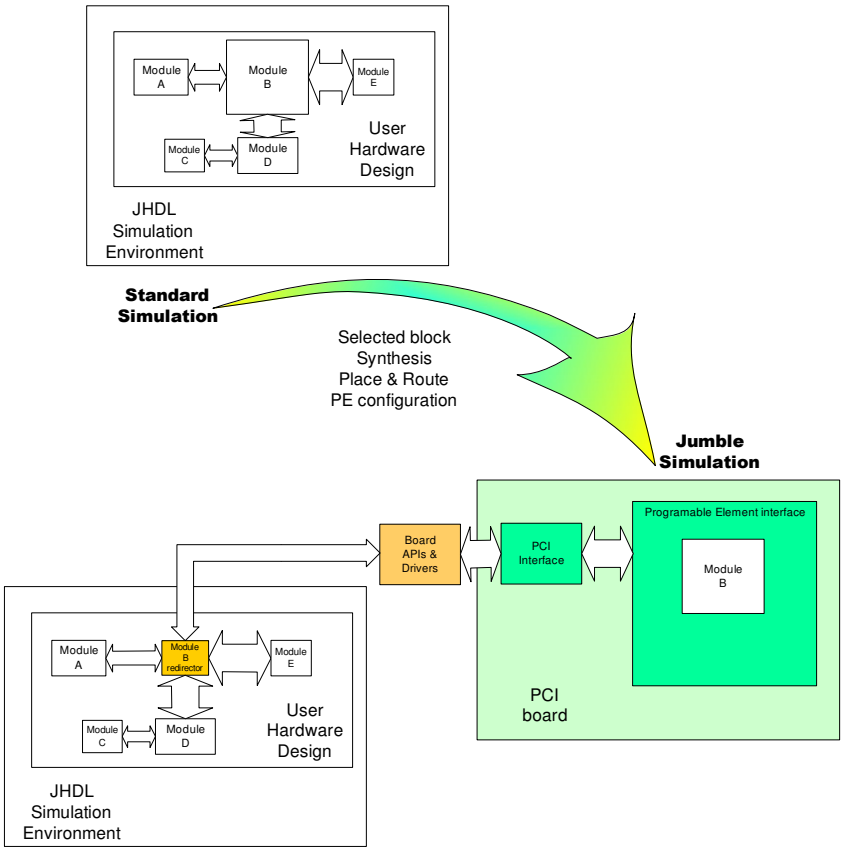

Figure 37 Jumble Simulation as user selects module B to execute in hardware

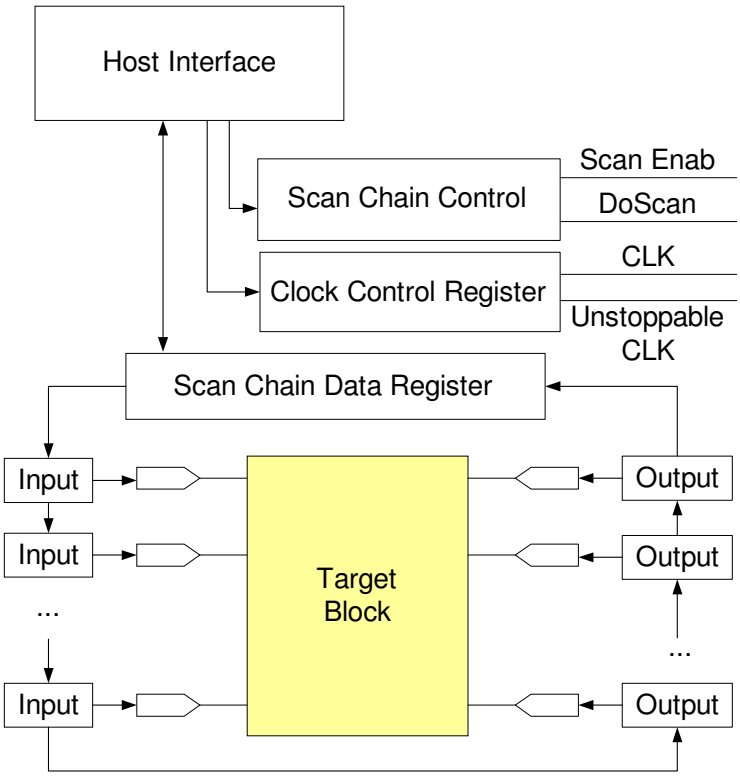

Figure 38. Scan chain and clock control

The added logic to communicate with the simulator has to be able to put inputs and get outputs the circuit under test and control its clock advance. To do so the target design is wrapped around a boundary scan chain (Figure 38), which is build as a circular register that has a window accessible from the Host. This window is a 32 bits wide register called *Scan Chain Data* (SC). A *Scan Chain Control* (SCC) register is used to control how to shift the scan chain. The SCC controls the two important signals: `DoScan` and `EnabScan`. `EnabScan` indicates that the scan chain should shift one bit. The less significant bits of SCC contain a counter value to instruct how many bits should be shifted in the scan chain. The `DoScan` flag indicates that a scan operation is being performed. It keeps activated during several transactions of the PCI bus until all the data has been correctly shifted on the scan chain. While `DoScan` is active the circuit under test does not see any change on its inputs. Only when `DoScan` gets down the inputs reflect the values that have been feed to the system through the scan chain.

After all data has been shifted in the inputs a clock cycle can be scheduled. Clock control register (CC) accepts a number of clocks to be run in the hardware system. A gated clock circuit is controlled by a countdown counter that stops running when zero is reached. Its design is shown in Figure 39.

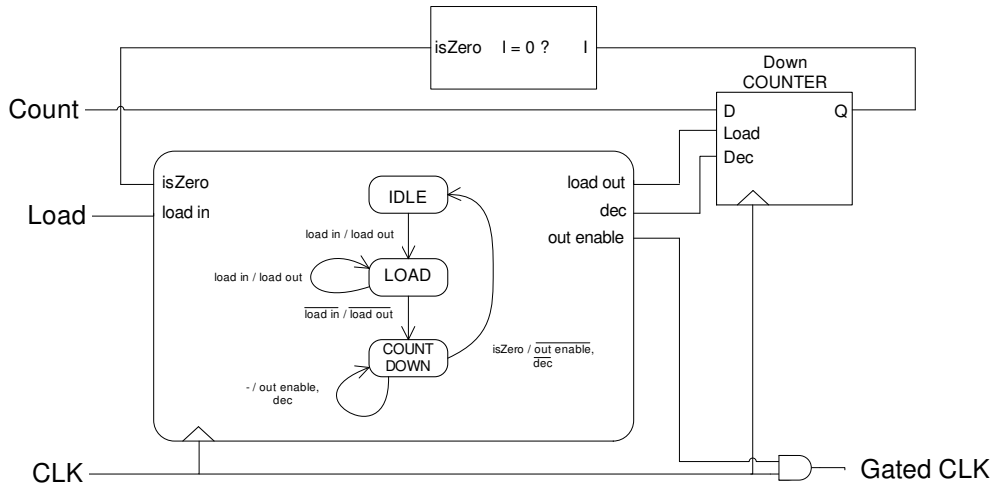

Figure 39 Block diagram of countdown clock

When the clock cycles are complete all output registers are shifted out through the scan chain.

Full scan chain registers uses the ScanOut value for chain connection and for regular Q. However the dangling of register output during shift operations could change the state of possible asynchronous designs. So an asynchronous safe boundary scan chain node is used as shown in Figure 40.

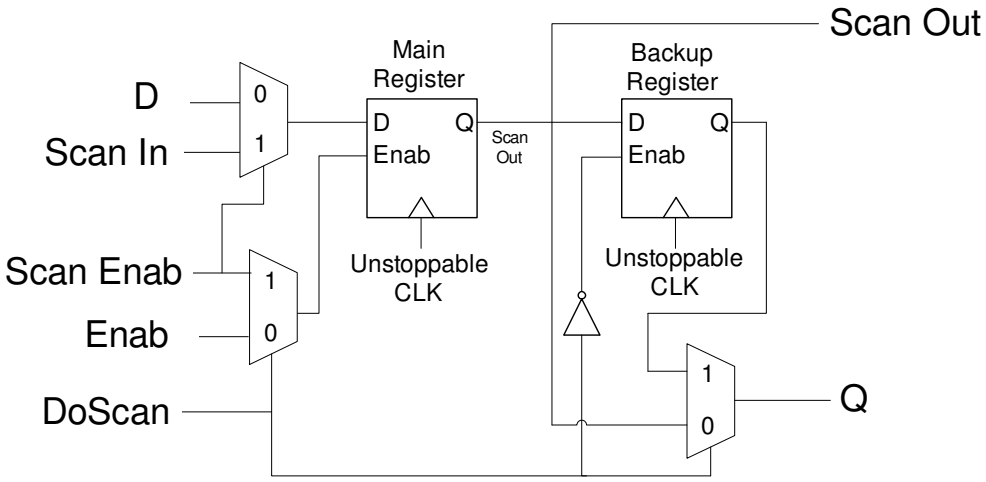

Figure 40. Boundary scan register

The simulation time reduction, or speedup, that I expect is mainly determined by the percentage of circuit design that is implemented in hardware. This is quite an evident conclusion if we recall Amdahl's law. The more circuit is implemented in hardware the faster the simulation. However in our design we have to take into account the width of the interface as well, since we spend quite a lot of "slow" cycles to place the correct values into the inputs and outputs of the hardware version of the target circuit.

From here after we can consider only a simulation of a single clock run, multiple clock runs can be generalized from the values that we get by a simple multiplication. Speedup is determined by the factor between the standard simulation time and the time used by Hardware in the loop simulation. Of course, in the later case we have to consider that there is a part that is implemented in Hardware and another part that remains unmodified. Their contributions to simulation time are T_{Jumble} and T_{Rest} respectively.

$$Speedup = \frac{T_{Std}}{T_{HIL}} \tag{3}$$

$$T_{HIL} = T_{\mathrm{Re}\,st} + T_{Jumble} \tag{4}$$

$$T_{Std} = T_{\mathrm{Re}\,st} + T_{SWT\,\mathrm{arg}\,et} \tag{5}$$

The part of the circuit implemented in hardware needs to get input data from the simulator, run a clock cycle, and send the output data again to the simulator. This time can be grouped by transfer time ($T_{I/O\ Transfer}$) and clock run ($T_{HWTarget}$).

$$T_{Jumble} = T_{I/O\ Transfer} + T_{HWT\,\mathrm{arg}\,et} \tag{6}$$

Input/Ouput data transfer is performed by reading and writing the SC and SCC registers. We do several PCI bus operations to complete a complete clock cycle but the $T_{I/O\ Transfer}$ is related with the width of the circuit interface. As the slowest part of the process is the shift of the SC register and runs at PCI clock speed we can do the simplification to make $T_{I/O\ Transfer}$ linear to the width of the circuit interface.

$$T_{I/O\ Transfer} = W_{\mathrm{int}\,erface} \cdot T_{PCI} \tag{7}$$

Since we only run a clock cycle we can totally ignore the contribution of $T_{HWTarget}$ to T_{Jumble}.

$$T_{Jumble} = W_{\mathrm{int}\,erface} \cdot T_{PCI} \tag{8}$$

The last simplification that we can make is to consider that the clock period of the host computer is a fraction of the period of the PCI clock. And then, express the time of the Software parts of the simulation in CPU operations.

$$T_{CPU} = \frac{T_{PCI}}{60} \tag{9}$$

$$T_{\mathrm{Re}\,st} = \frac{T_{PCI}}{60} \cdot Op_{\mathrm{Re}\,st} \tag{10}$$

$$T_{SWT\,arget} = \frac{T_{PCI}}{60} \cdot Op_{T\,arget} \qquad (11)$$

Obviously to get significant speedup T_{Jumble} should be smaller than $T_{SWTarget}$, but also greater than T_{Rest} to avoid the effects of an small α in the Amdahl's law.

Let's consider some situations to see how the numbers affect to the expected speedup. In all the cases we will consider a target circuit with a 1000 wires interface.

If we had a big part of the circuit as Hardware we would expect $\boldsymbol{T_{Rest} << T_{Jumble}}$. The unaccelerated software model of the circuit should use less than 60K operations per cycle of the host processor (13).

$$\frac{T_{PCI}}{60} \cdot Op_{Re\,st} << W_{int\,erface} \cdot T_{PCI} = 1K \cdot T_{PCI} \qquad (12)$$

$$Op_{Re\,st} << 60K \qquad (13)$$

In this case the Speedup would be given by

$$Speedup = \frac{T_{SWT\,arget}}{T_{Jumble}} \qquad (14)$$

$$Speedup = \frac{\frac{T_{PCI}}{60} \cdot Op_{T\,arget}}{W_{int\,erface} \cdot T_{PCI}} = \frac{Op_{T\,arget}}{60 \cdot 1K} \qquad (15)$$

For instance, if we are looking for a speedup factor of 100 we should have a target that uses more than 6M host operations to simulate each clock cycle.

$$Op_{T\,arget} > Speedup \cdot 60K > 100 \cdot 60K \qquad (16)$$

Comparing this value with the rest of the circuit it is clear that moving a large part of the circuit to the target gives an opportunity to achieve some important speedup. If rest part is small enough (much less than 60K operations) the factor of operations used by the rest part versus the target part is approximately equal to the obtained speedup.

Consider now opposed case, having $\boldsymbol{T_{Jumble} << T_{Rest}}$. This does not necessary mean that target is small compared with the rest of the circuit. As T_{Jumble} is constant for a given interface size this means that the rest of the circuit model is much more complex

than 60K operations per cycle (18), but gives to information about the complexity of the target.

$$T_{\mathrm{Re}st} = \frac{T_{PCI}}{60} \cdot Op_{\mathrm{Re}st} >> T_{Jumble} = W_{\mathrm{int}erface} \cdot T_{PCI} \qquad (17)$$

$$Op_{\mathrm{Re}st} >> W_{\mathrm{int}erface} \cdot 60 >> 60K \qquad (18)$$

Again, in this case the speedup is determined by the complexity of the target software model.

$$Speedup = \frac{T_{\mathrm{Re}st} + T_{SWT\,\mathrm{arg}et}}{T_{\mathrm{Re}st}} \qquad (19)$$

$$Speedup = 1 + \frac{\frac{T_{PCI}}{60} \cdot Op_{T\,\mathrm{arg}et}}{\frac{T_{PCI}}{60} \cdot Op_{\mathrm{Re}st}} = 1 + \frac{Op_{T\,\mathrm{arg}et}}{Op_{\mathrm{Re}st}} \qquad (20)$$

For instance, if we are looking for a speedup factor of 100 we should have a target that uses much more than 6M host operations to simulate each clock cycle (21).

$$Op_{T\,\mathrm{arg}et} > (Speedup - 1) \cdot Op_{\mathrm{Re}st} >> 100 \cdot 60K \qquad (21)$$

Until now we have considered two extreme cases, and in both cases we need a complex target to achieve a significant speedup. But what speedups can we get in more balanced cases? By **balanced** I mean having a similar complexity of the software models of the target part and the rest part.

Let's consider such a case. In this case we have a circuit in which the target part needs 100K operations and the rest part needs 100K operations as well. As previous examples the interface has 1000 wires.

$$Speedup = \frac{T_{\mathrm{Re}st} + T_{SWT\,\mathrm{arg}et}}{T_{\mathrm{Re}st} + T_{Jumble}} \qquad (22)$$

$$Speedup = \frac{\frac{T_{PCI}}{60} \cdot Op_{\mathrm{Re}st} + \frac{T_{PCI}}{60} \cdot Op_{T\,\mathrm{arg}et}}{\frac{T_{PCI}}{60} \cdot Op_{\mathrm{Re}st} + W_{\mathrm{int}erface} \cdot T_{PCI}} = \frac{\frac{100K}{60} + \frac{100K}{60}}{\frac{100K}{60} + 1000} = 1.25 \qquad (23)$$

This math troughs a rather moderate value for the obtained speedup, but we should be expecting this kind of results after looking at Amdahl's law. In this case we could improve the speedup if we had a smaller interface, but even with an interface of a single wire the speedup would be just almost 2. The only way of having significant speedups is to have a good percentage of circuit implemented in the target, or in Amdahl's terminology, to have a value of α very close to 1.

Chapter 5

Implementation

Extending JHDL framework

The JHDL framework had several drawbacks that limited its potential for circuit design. One of the drawbacks of JHDL was its limited support for FPGA devices; this was a serious drawback as our research group has traditionally been working with Altera devices. Another important drawback was the lack of behavioral synthesis. Complex control schemes are better implemented with behavioral code (at the RTL level) and behavioral synthesis is required to transform this code into hardware blocks. Finally, another detected drawback was the faulty support of sequential behavioral model. This feature was present in initial versions of JHDL through the `HWProcess` class but somehow was unsupported by newer versions.

The following subsections describe the work undergone to solve these problems and overcome these limitations. Parts of this works were done with Jordi Farré and Alexis Morugó as part of their respective final year projects and were later published in [Castells04b] and [Castells06b].

Adding support for Altera devices

Most of current FPGA and CPLDs are based on the use of simple called computational blocks (CLBs) or logic elements (LEs). These blocks are often built up from LUTs, registers and multiplexers. Any logic circuit, either combinational or sequential, can be build by the combination of several of these blocks. Every FPGA has a different CLB design. The goal of any design tool is to make the best use of the available CLB resources. This process is known as technology mapping [Cong94].

JHDL includes specific TechMappers for every supported FPGA device. Their function is to translate logic functions in their equivalent optimal structures for every FPGA device. Figure 41 is a clear example illustrating the objective of this process. To the left side, there is the structure of a CLB of the Virtex family devices from Xilinx. To the right, the result of mapping a 9-input and gate, performed by the `VirtexTechMapper` class, is shown. To make a good use of resources the `VirtexTechMapper` has divided the `and9` function in two `and4` that can be implemented by two LUT4 present in the CLB and has completed the function by using two multiplexers also present in the CLB structure.

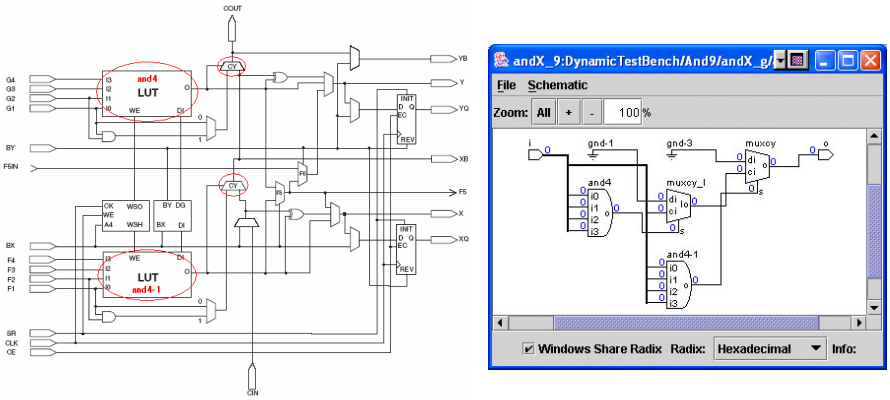

Figure 41 Virtex CLB and mapping of a 9 input and gate as done by *VirtexTechMapper*

To add support for Altera devices in JHDL, we need to know the structure of the LE (equivalent to CLB in Altera technology) for all of their devices and then implement a TechMapper that performs an optimal adaptation of the logic functions to the LE structure. Instead of this, we took a simpler approach: use the LPM standard. The LPM standard [Altera96] allows including high level elements in the netlist file, like EDIF format [Edif]. The LPM elements can include parameters and need a logic synthesis step before the Place & Route. The previous logic synthesis process ensures an optimal mapping to the LE structure for Altera devices but causes a lose of control about the number of used FPGA resources from JHDL viewpoint.

The first step consists in implementing the classes of primitive logic elements based on LPMs. The primitive logic elements are not decomposable in other simpler elements, they are the leaves in the circuit hierarchy. They also need a behavioral model for simulation. Either `propagate`, or `clock` and `reset` functions must be defined for combinational or sequential logic respectively.

We have implemented LPM_AND, LPM_OR, LPM_XOR, LPM_INV, LPM_MUX, LPM_FF, LPM_ADD, LPM_ADD_SUB and LPM_CONSTANT in a new `com.Altera.lpm` package. These primitives are similar to primitives from Xilinx devices but, as they have a higher level of abstraction, they make more use of generic parameters.

Once the primitives are implemented, the next step is to develop some technology mapping classes (`ApexTechMapper`, `CycloneTechMapper`, `StratixTechMapper`) that make use of the new available LPM primitives. Thanks to the parametric nature of LPM primitives the mapping process is much simpler than the equivalent process for other technologies like Xilinx.

Finally, it is necessary to develop a custom Netlister due to the differences in the interpretation of EDIF files between Altera and Xilinx tools. The main problem is how GND and VCC signals are handled. Xilinx tools define two primitive logic elements for this purpose. They are like logic gates that have no inputs and drive a constant value. Each VCC or GND connection in a JHDL circuit ends up in an instantiation of one of these custom gates connected each target. Altera tools do not define such primitives and assume VCC and GND as being global networks of the circuit. In addition LPMs

make heavy use of bidimensional signal arrays, which are not directly supported by the EDIF standard.

For these reasons, a custom Netlister called `CephisNetlister` has been developed to address the particularities of Altera *Place & Route* tools.

Enabling behavioral synthesis through VHDL generation

Behavioral JHDL code has to main advantages over structural code: it is much more human readable when describing reactive systems and is faster to simulate.

Behavioral synthesis from Java code was proposed in previous works such GALADRIEL [Cardoso98], NENYA [Cardoso99], Wirthlin's work [Wirthlin01] and Sea Cucumber [Tripp02].

Most of these approaches are based on the analysis of sequential code and do not match the usual RTL like descriptions used in JHDL behavioral models. All methods are based on the analysis of the CFG and DFG derived from the java code to build either EDIF or VHDL code.

The VHDL language [VHDL98] offers a great flexibility to model digital electronic circuits. Designs can be described in various levels of abstraction (sequential behavior, RTL and structural) and even mix them in the same source code. Since not all descriptions are synthesizable, designers have to know which subset to use in order to avoid rewriting.

```
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

ENTITY count_a IS
 PORT(clk, rst, updn : in std_logic;
 q : out std_logic_vector(15 downto 0));
END count_a;
ARCHITECTURE logic OF count_a IS
BEGIN
 PROCESS(rst, clk)
  VARIABLE tmp_q : std_logic_vector(15 downto
0);
 BEGIN
   IF rst = '0' THEN
    q <= 0;
   ELSIF rising_edge(clk) THEN
    IF updn = '1' THEN
     tmp_q := tmp_q + 1;
    ELSE
     tmp_q := tmp_q - 1;
    END IF;
    q <= tmp_q;
  END IF;
 END PROCESS;
END logic;
```

```
import byucc.jhdl.Logic.*

public class Count extends Logic {
  public static CellInterface[]
cell_interface={
    clk("clk"), in("rst",1),
    in("updn",1),out("q",16)};
  Wire out, updn;
  int tmp;
  public Count(Wire clk, Wire rst, Wire updn,
    Wire out){
    connect("clk", clk);
    connect("rst",rst);
    this.updn = connect("updn", updn);
    this.out = connect("out", out);
  }
  public void reset() {
    q.put(this, 0);
  }
  public void clock() {
    if (updn.getB(this)) {
      tmp = tmp+1;
    } else {
      tmp = tmp-1;
    }
    out.put(this, tmp);
  }
```

The following code shows (left column) a fragment of VHDL code mixing RTL and behavioral coding styles. The same circuit coded in behavioral JHDL (right column) has many similarities. Both programs contain a section where the interface, i.e. the inputs and outputs of the circuit, is defined. Since this is a synchronous circuit, the VHDL process sensitivity list only contains the clock and reset signals. This definition is

implicit in JHDL synchronous circuits. The reaction of the circuit to reset and clock signals is clearly separated in both descriptions and is very similar with minor syntax differences.

Not all VHDL and JHDL circuits are suitable for such comparison. However it is applicable to a large number of designs like FSMDs and reactive systems.

In this case, a new netlister that produces VHDL has been build to substitute the default EDIF netlister. The EDIF format only allows describing the circuit structure, but the VHDL language allows descriptions of both structure and behavior in a single language.

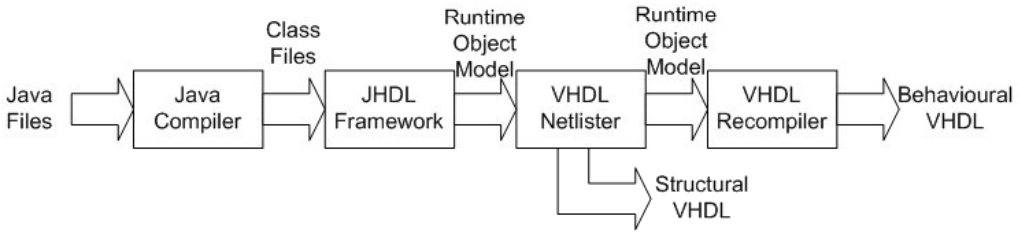

Figure 42 Tool Flow to generate VHDL

The runtime model of designed circuits can be manipulated, by the implemented VHDL netlister, to generate the desired output. The netlist generation for structural circuits is straightforward and mimics the approach followed by the EDIF netlister. Behavioral circuits are decompiled to extract the original Java code and translate it into its equivalent VHDL. Advantages of decompiling over parsing source code are: decompilation is simpler than parsing and can assume there are not syntax errors in the input, moreover source code localization is not needed.

The selected decompilation framework is the open source project JODE [Hoenicke01].

The translation to VHDL has three main blocks: the interface declaration, the variable declaration and the description of the behavioral process. The interface declaration contains the ENTITY clause and can be easily created from the runtime model. JHDL offers methods like `getPortRecords` that enable a full exploration of any circuit interface.

The variable declaration section cannot be neither directly derived from runtime information nor from the member variables of the decompiled class, because some member variables could not be used by the behavioral model. So, a deeper analysis of the member variables usage in the behavioral model is needed to determine them and create the final VHDL section.

VHDL behavioral circuits contain the PROCESS keyword with a sensitivity list containing a list of the signals that trigger a change of state in the circuit. In synchronous circuits, the sensitivity list always contains the clock and reset signal and its body contains the functional description derived from the translation of clock and reset methods.

```
if (reset = '1') then
 <reset method translation>
elseif  clk'event and clk = '1' then
 <clock method translation>
end if;
```

On the other hand combinational circuits can be expressed as a process with all the input signals as part of the sensitivity list. The body of the process is translated from the propagate method.

```
<propagate method translation >
```

The structure of the control instructions in VHDL and Java are quite similar so the translation process consists in adapting the final rendering process of Jode to generate VHDL instead of Java.

Besides expressions and blocks, there are significant differences in how both languages handle variables and signals. In VHDL, signals are assigned by using the <= operator and variables are assigned with :=. VHDL is a strong typed language so variables and signals have to have the same type and width to interoperate. To bypass these rules, conversion functions can be used. JHDL uses get and put to obtain and assign values of signals. The behavioral models can use few primitive types like `boolean`, `int`, `long` and the bit-vector (BV) class. There are multiple versions of get and put methods that accept these primitive types.

It is necessary to keep track of the variables and signals that are used in the behavioral model and their type and size. This information is used to know the conversion function that has to be applied in each situation during translation process.

Signal width information is very important in VHDL. For instance, when assigning a constant to a signal, the constant must have the same exact length as the signal it is assigned to, i.e. the same number of binary digits. Obtaining the width of design elements is crucial for a correct conversion. Design wires can be handled easily since wire width is a fundamental property of JHDL designs and can be easily obtained. Design variables are a little bit trickier. Variables are part of the behavioral description and have often Java types like `int`, `long` or `boolean`. We need to define an VHDL equivalent data type for each possible variable type so we can propose a width for design variables.

Table 2 Data types equivalence between JHDL and VHDL

JHDL data type	VHDL data type
boolean	std_logic
int	integer
long	std_logic_vector(63 downto 0)
BV(n)	std_logic_vector(n-1 downto 0)

The behavioral models can use expressions that contain operation involving variables and signals. Both languages have little common operators and an equivalence table definition is needed to perform the translation.

Table 3 Operators equivalence between Java and VHDL

Java operator	VHDL operator
=	:=
==	=
!=	/=
&	and
|	or
^	xor
!	not

as an example the following code has been created with the translator.

```
package org.cephis;
import byucc.jhdl.base.*;
import byucc.jhdl.Logic.*;
public class SAdd extends Logic
implements
com.Altera.BehaviourallyModeled {
 public static CellInterface[]
cell_interface={
  in("start",1), in("ops",8), out("sum",
8) };
 Wire start,ops,sum;     // Wires
 int a,b, state = 0;     // FSM state
 public SAdd(Node parent, Wire start,
   Wire ops, Wire sum) {
   super(parent);
   this.start = connect("start", start);
   this.ops = connect("ops", ops);
   this.sum = connect("sum", sum);
  }
  public void reset(){
   state = 0;
   sum.put(this, 0);
  }
  public void clock(){
   switch (state) {
    case 0: // idle
     if (start.getB(this)) state = 1;
break;
    case 1: // fetch A
     a = ops.get(this); state = 2;
break;
    case 2: // fetch B
     b = ops.get(this); state = 3;
break;
    case 3: // output
     sum.put(this, a + b); state = 0;
break;
   }
   setDefaultValues();
 }
 public void setDefaultValues() {
  if (!sum.hasBeenPut())
   sum.put(this, sum.get(this));
 }
}
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
entity SAdd is port ( c :  in std_logic ;
    start :  in std_logic ;
    ops :  in std_logic_vector(7 downto 0) ;
    sum :  out std_logic_vector(7 downto 0)
;
    reset :  in std_logic );
end SAdd;
architecture JHDL of SAdd is begin
 SAdd:process(reset, c)
 variable b :integer;
 variable a :integer;
 variable state :integer;
begin
if (reset = '1') then
 state := 0;
 sum <= conv_std_logic_vector(0,sum'length);
elsif c'event and c ='1' then
 case state is
  WHEN 0=>
    if (conv_integer(start) /= 0) then
    state := 1;
    end if;
 WHEN 1=>
    a :=
ieee.std_logic_unsigned.conv_integer(ops);
    state := 2;
 WHEN 2=>
    b :=
ieee.std_logic_unsigned.conv_integer(ops);
    state := 3;
 WHEN 3=>
    sum <= conv_std_logic_vector(a +
b,sum'length);
    state := 0;
 WHEN OTHERS =>
    end case;
end if;
end process;
end JHDL;
```

Reviving sequential design style for behavioral models

Behavioral JHDL descriptions are intuitive and convenient but were limited to the model known as RTL in most hardware design languages. To effectively follow a refinement process from a Software specification, it is much more convenient to describe behavior in a sequential way with some extensions to incorporate the notion of time and parallel execution of statements.

Most high level languages, as HandleC, SystemC, VHDL and Verilog, allow using a programmer friendly sequential description model which adds time semantics by using wait statements. This level of abstraction is called “behavioral model” in most HDL languages, a bad choice from my point of view since RTL is also behavioral. It would be less ambiguous to use the term “sequential model”. Original JHDL implementations include this design style with the HWProcess class, but its support was bound to the Single Clock Simulator and was lost when the Multi Clock Simulator was introduced.

SystemC SC_CTHREAD constructs forces to describe processes in a sequential way, which is more programmer-friendly. Since regular sequential descriptions have no time semantics, they must be incorporated by a language extension. The special instruction wait is used for this purpose. Any code between two consecutive wait statements must be executed in the same clock cycle. A SystemC SC_CTHREAD usually involves the creation of a real Thread of the Simulator process. During system execution, when wait function is called the Thread passes to a suspended state. All the SC_THREADs of the system have the same behavior. The SystemC simulator has to wait until all SC_THREADs are suspended to advance the clock value and propagate signals. After this step SC_THREADs return to execution state.

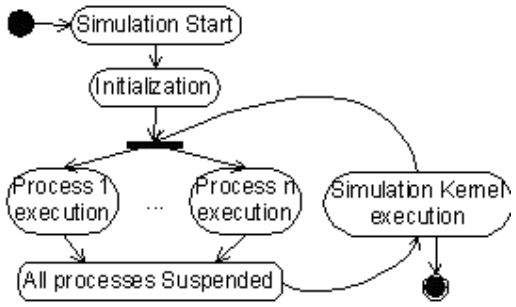

Figure 43 SystemC simulation cycles

To revive sequential design style, I have created a new class `ThreadedLogic` that supports the wait statement. All classes derived from `ThreadedLogic` must implement the functions `thread_clock`, `thread_reset` and `thread_run`.

When an object of a `ThreadedLogic` derived class is instantiated, the `ThreadedLogic` constructor automatically creates a worker thread that will call to the `thread_run` method. The `thread_run` method should use the different flavors of `sc_wait` function to advance the clock run.

All operations between two consecutive calls to the `sc_wait` function should occur in the same clock cycle as it happens in SystemC or HandleC. To achieve this goal, we

will use the synchronization primitives of Java. The `sc_wait` function will place a wait operation on the current object that will cause the thread to be blocked. A `sc_notify` will be called when the JHDL simulator call the `clock` method of the `ThreadedLogic` class, after calling `thread_clock`.

Special care must be taken with the synchronization of the worker thread with the rest of the system. Different conditions depending on the order of the calls to `sc_wait` and `thread_clock` can produce unexpected results. A formal approach is followed to avoid this kind of undeterminism. We define a global invariant as described in [Mueller01] in the following way, being *bW* an indication that there are pending clocks to run,

clock method
⟨**await** *bW* = true⟩
assign output values
⟨bW = false⟩

worker thread sc_wait
⟨**await** bW = false → *bW* = true⟩
⟨**await** bW = false⟩

The resulting `ThreadedLogic` class ensures the no race conditions will occur and reopens the richness of design style to allow sequential descriptions.

Platform Support

JHDL allows defining platform models [Bellows04]. There are several supported platforms models: Wildcard [Wildcard], SLAAC [Slaac], and Osiris [Osiris]. A platform model includes the description of the details of a hardware board that can host a design. These details include characteristics of the resources mounted on the PCB, like memories and oscillators; characteristics of the programming element (FPGA) like pin details; and possible optional IP cores like external interfaces. A hardware model also includes some API to be able to communicate with the instantiated hardware from Java applications.

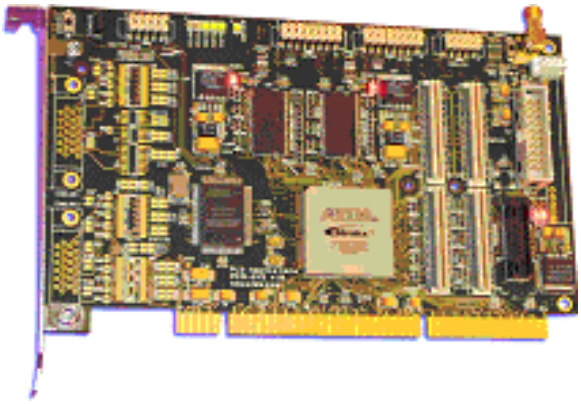

Figure 44 PCI-X board from PLD Applications

A hardware model for the PLD Applications PCI-X board (Figure 44) has been created. The board contains an Altera Stratix S30 FPGA device, four LEDs, a 100Mhz oscillator and two DDR memory banks. The model (Figure 45) includes the description of the external devices (related to the FPGA) and some IP cores that implement interfaces to them (PCI controller, DDR controller, Led Interface and Clock interface).

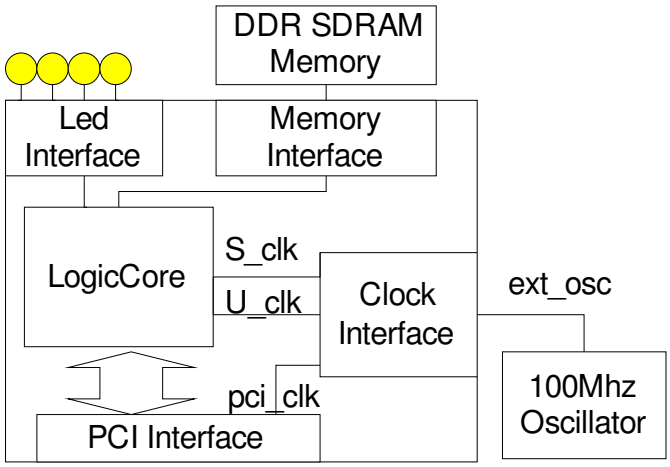

Figure 45. PLD Applications PCI-X hardware model

Wrapper Architecture and Infrastructure

One of the nice features of JHDL is that the circuit object model can be manipulated in real time. In fact, one of the original goals of JHDL was to support Runtime Reconfiguration [Bellows98], which had to be addressed by using Java object construction/destruction as a method to dynamically program and release circuits on the FPGA. We manipulate the circuit design by replacing a designer selected circuit by an implementation that redirects its input/output wires to its real hardware implementation.

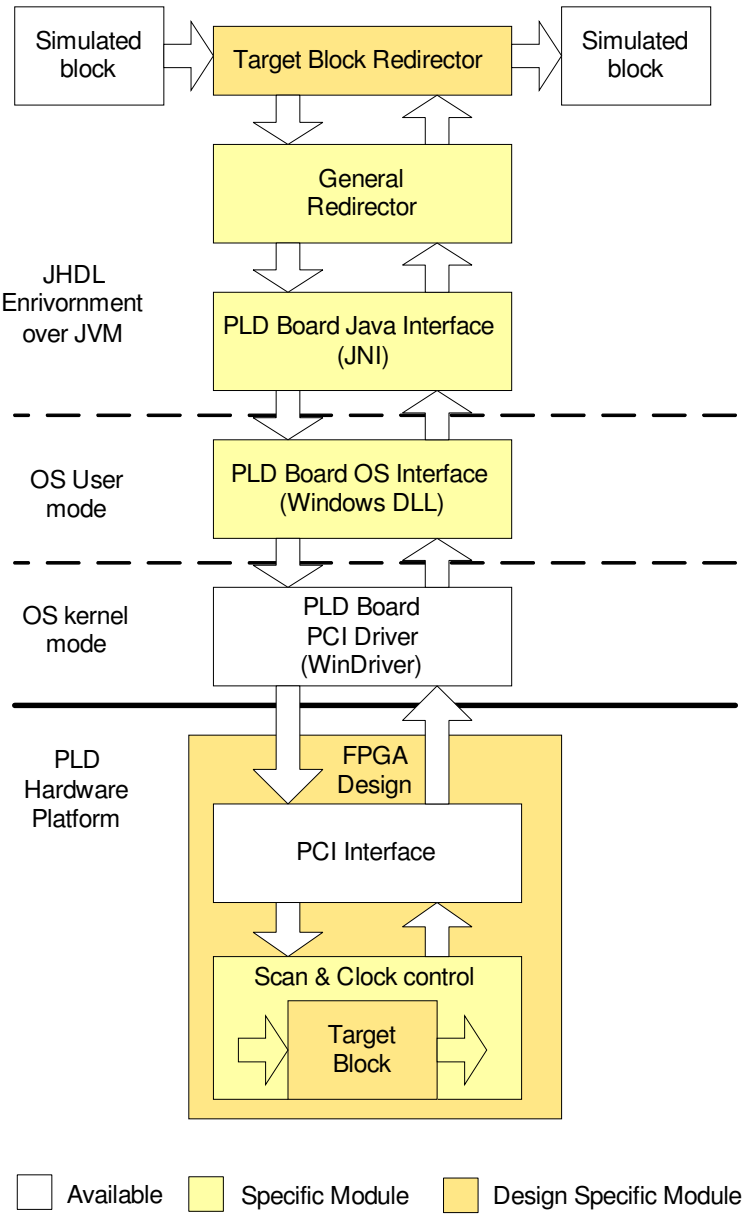

Figure 46. Wrapping Software Architecture

Although designer is hidden from the underlying details a number of systems are involved in this process:

JHDL Simulator

The JHDL simulator is the application that hosts the circuit blocks that are being simulated. It presents a GUI that includes some buttons to control clock advance, and command line interface. The command line is used by the user to instruct which block has to be executed by the hardware platform.

Target Block Redirector

When the user orders to execute a block into hardware, the original block is removed and replaced by a redirector. The feature of dynamically modify the hierarchy of the circuit by adding, removing or substituting design entities during a simulation session is unique to JHDL and becomes crucial for this work.

The redirector copies the interface of the substitute class to keep an accounting of the input and output pins and their widths. This block contains a behavioral model that redirects inputs and outputs and controls the real hardware clock advance using a general redirector.

General Redirector

The general redirector has a low level control the hardware platform interface by reading and writing its memory mapped registers. On the one hand controls the transfer of inputs and outputs by commanding the scan chain. On the other hand, it controls the clock advance. In our case, we always advance the clock with one step, but future applications could use a different approach.

PLD Board Java interface

The developed PLD board platform model include a Java Native Interface (JNI) class to access to OS dependent communication primitive operations such as hardware detection, and read and write to memory mapped registers.

The `PCIXNative` class main functions are `open`, `close`, `writeMem` and `readMem`. In addition, another class (`RenablePLDA`) has been developed to control the execution of a utility application that forces PCI reenumeration.

PLD Board OS Interface

The final software interaction with the hardware platform is performed by a kernel driver. The development of a kernel device driver is a complex task and error prone. The bugs in this kind of software are catastrophic since their cause the reboot of the machine. So, for this kind of work, it is better to use a commercial driver as WinDriver. The basic low-level functions are encapsulated into a Windows DLL to mitigate the hassles of kernel mode programming.

Nevertheless, there is an issue that is not covered by the DLL, which is the resource negotiation with the OS. PCI devices are Plug & Play. This means that their OS needed resources (memory ranges, I/O ranges, and interrupts) are flexible and are determined by a central resource manager, which is part of the OS. This avoids conflicting address spaces or interrupts like we had in the old ISA days. There is a special PCI configuration mode that allows the resource manager to resolve the resource needs for each plugged device during a process that is known as PCI enumeration. However, enumeration is only done at boot up or after device insertion for hot swappable devices. The user can force this process for new devices from Windows Device Manager (as seen in Figure 47). But to rerun the negotiation for an already connected device, you need to manually disable and enable the device.

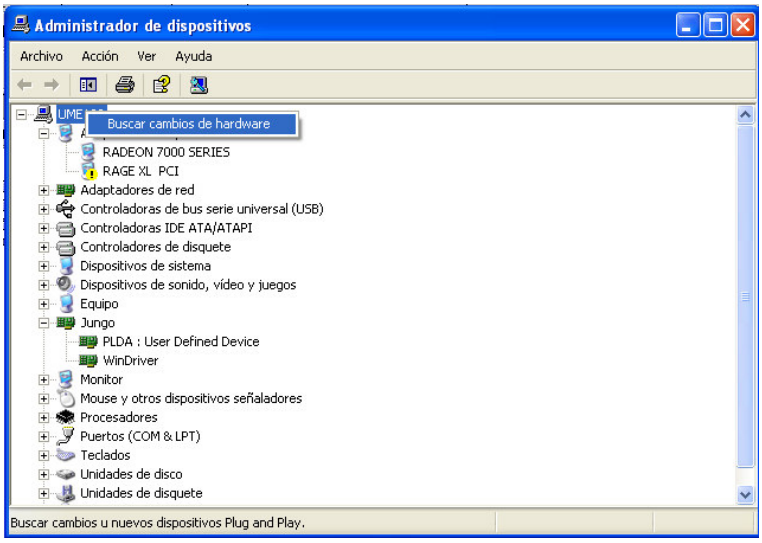

Figure 47 Forcing Windows Device Manager to dectet new devices

When the FPGA device of the PLD is configured through JTAG, the host computer does not receive any event that causes a reenumeration of the PCI. A reenumeration is needed to reassign the resources needed by the device. Since the previously described way of forcing the reenumeration is not practical for an automated framework, an utility application has been developed to programmatically perform the disabling and enabling of the device, and resolving its resource needs.

PLD Board PCI Driver

A Windows kernel driver exposes very simple functions via IOCTLs to user mode applications. These functions are basically read/write operations to memory.

FPGA Design (Wrapper)

The FPGA design is based on the developed Hardware Platform. It combines a set of predefined blocks, like the PCI-X interface and clock interface, with the user block wrapped by the scan chain and all the register based control interface.

PCI-X Interface

PLDApplications provides a PCI-X core together with the PCI-X platform. The core is highly configurable via a Core Configuration Wizard (see Figure 48) supporting various advanced PCI features. A specific set of parameters have been used and a VHDL implementation of the version of the core has been included as part of the platform model.

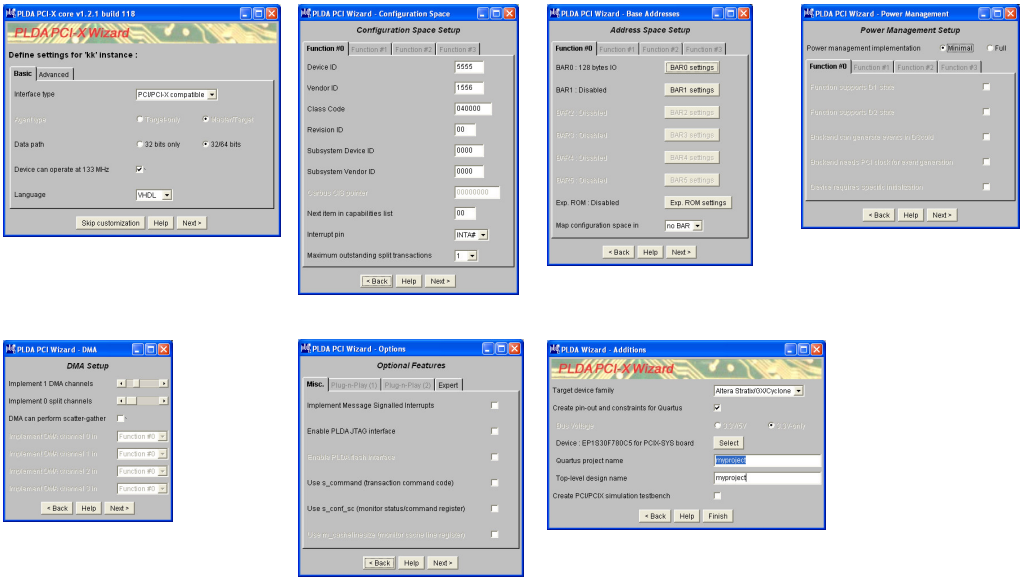

Figure 48 PLD Applications PCIX IP Wizard

Register Interface

The host interface consists of three registers accessible from the PCI-X bus: Scan Chain Control (SCC) register, Scan Chain Data (SC) register and Clock Control (CC) register.

Target Block

The original block, selected by the user for hardware execution, is the central part of the FPGA. The design inputs and outputs are wrapped around by boundary scan registers controlled by the register interface.

The target block runs in a different clock domain.

Interactive Command-line operations

The process of hardware substitution is not implemented in a single push button but is separated into four processes as shown in Figure 49.

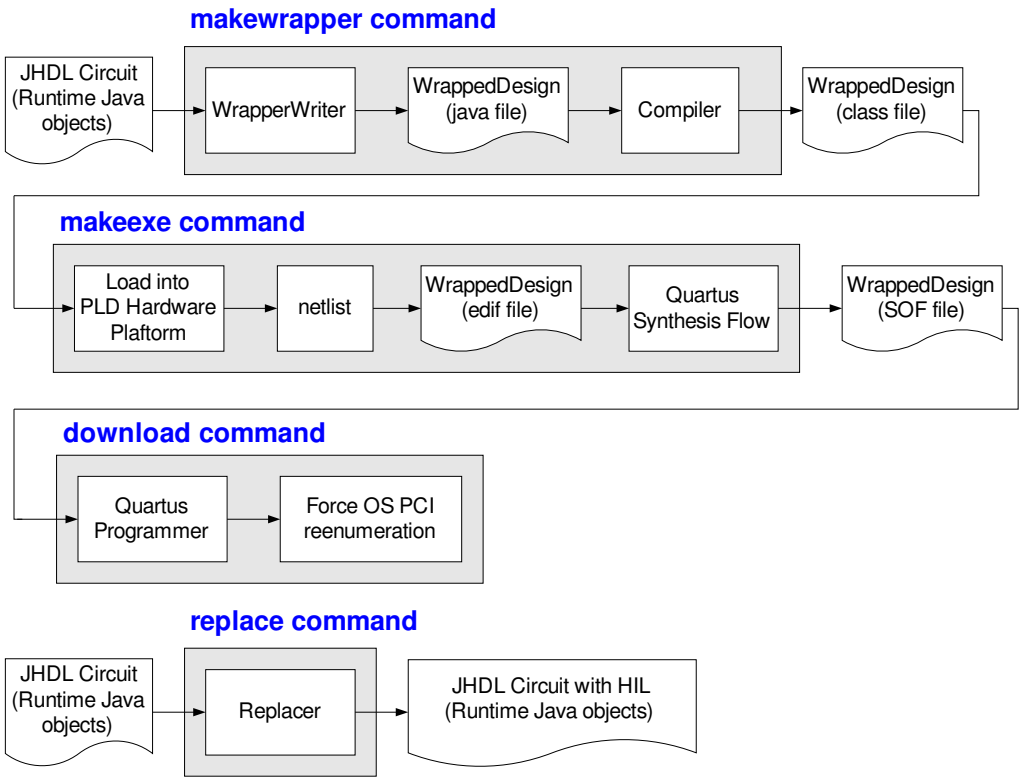

Figure 49. Commands involved in HIL automation

The `makewrapper` command writes the Java Source code of the JHDL circuit that will go into the FPGA and compiles it. This circuit contains the target block wrapped around a scan chain with a clock control unit and the commercial PCI interface.

The `makeexe` command loads the compiled class into the PLD applications platform environment produces the netlist, and call the Quartus tools to end up with a binary file.

The `download` command uses Quartus programmer tool to download the created .SOF file into the board. It also calls a custom application that enable and disable the driver so that PCI reenumeration is done.

Finally, replace command substitutes the selected block by a redirector to its implementation into the hardware platform.

Chapter 6

Applications and Results

In order to experiment the expected benefits from Jumble simulation I have developed three different designs having increasing complexity. In each example design I have selected different parts of the circuit for Jumble replacement (hardware execution) and measured the simulation speedups achieved by this way.

Median Filter

The first test is performed with a simple system: A Median Filter application based on [Maheshwari97]. In this example the powerful testbench facilities can be clearly shown.

We have developed two custom modules to integrate picture viewing on the schematic viewer. `SchematicImageSource` takes the path of a JPEG image in the host file system, decompresses and renders the image, in the schematic viewer. The behavioral model outputs a pixel of the image in RGB every clock following a row scan basis. `SchematicImageSink` receives the RGB value and coordinates of a pixel each clock and renders them into an image viewable from the schematic view.

This allows a straightforward environment for the verification of the system. A comparable testbench with VHDL or Verilog would be extremely complex, in case it is possible. Figure 50 shows the schematic view of the system. The original image with added noise generates the signals that are sent to the median filter, which in turn send the results to the image sink. Simulation is completed by instructing to run 90000 clock cycles using the interpreter command line. The simulation is run twice, the first time with the default behavior and the second one replacing the median filter with its hardware version.

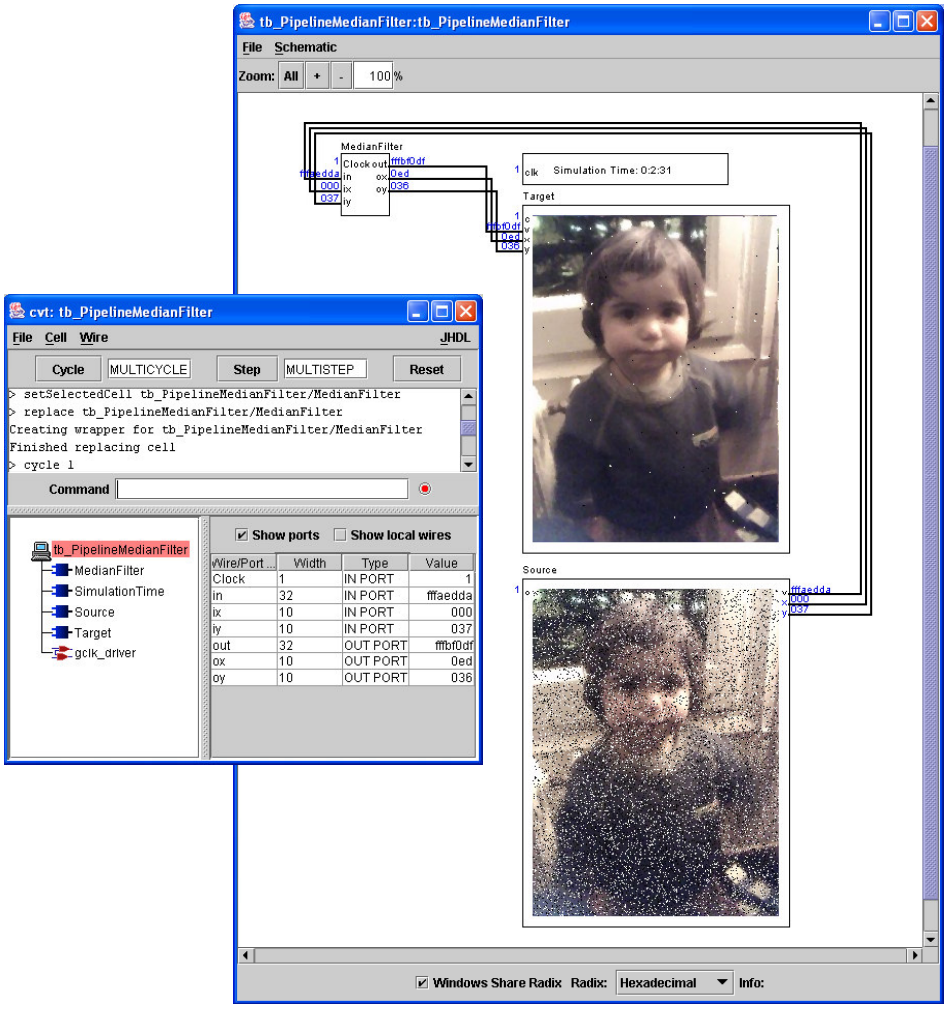

Figure 50. Advanced simulation environment

The speedup factor is between one and more orders of magnitude depending on the complexity of the circuit being replaced and the capacity of the host computer. Table 4 shows some example circuits and the speedup factor achieved when Jumble HIL simulation is used instead of a regular simulation. The host system consist of a PC with a hyperthreaded Pentium IV CPU running at 2.80Ghz with 512MB of RAM. The first design is the example shown above consisting of a simple median filter circuit applied to a noisy input image. The standard simulation uses 150 seconds to simulate 90000 cycles, enough cycles to produce the final filtered image. The jumble simulation takes only 3 seconds, what produces a speedup factor of 50. Second and third examples consist on a different design that chains a number of median filter units. In the first case the number of chained filters is 4 and in the second case the number is 10. Jumble simulation time keeps constant, as the interface to the hardware implementation is equivalent in all cases. The hardware implementation complexity increases with each design but anyhow it runs at the same speed because the clock frequency remains the same and hardware is inherently parallel.

Table 4. 90K cycle simulation for different designs

Design	Std. Simulation	Jumble Simulation	Speedup Factor
1 Median Filter	150 s	3 s	50
4 Median Filter Datapath	754 s	3 s	251.3
10 Median Filter Datapath	1902 s	3 s	634

The system is limited by the capacity of the FPGA. Obviously designs that do not fit in a single FPGA cannot be completely simulated with Jumble. As an alternative, not the whole system but a subset can often be downloaded to achieve some speedup.

Table 5 depicts the details of the resource usage of the second design example. The design uses a small fraction of the FPGA resources. The most important contribution to resource usage comes from the circuit under test followed by the PCI-X core instance. Clock control and LED interface have a very low contribution in resource usage. The third larger contribution comes from the scan and control system, which anyway supposes an acceptable overhead.

Table 5. Resource usage of the S30 device

Block of the design	LEs	Memory bits
PCI-X interface	2253 (39%)	1152 (2%)
Clock interface	57 (1%)	0 (0%)
LED interface	76 (1%)	0 (0%)
4 Median Filter Data Path	3127 (54%)	61156 (98%)
Scan & Control	309 (5%)	0 (0%)
TOTAL	**5826** (17% of device total)	**62308** (1% of device total)

Optical Digit Recognition System

The second set of tests is performed with an OCR system based on [Castells05],[Castells06]. The aim of the design is to include it into a commercial system (Figure 51) that will allow the remote reading of water meters by attaching a device on top of conventional mechanical meters, which will periodically take a picture, extract the counter reading and transmit it to a remote system through a wireless connection. The system has been patented [Ayuso06] and is currently commercialized by the company Mirakonta.

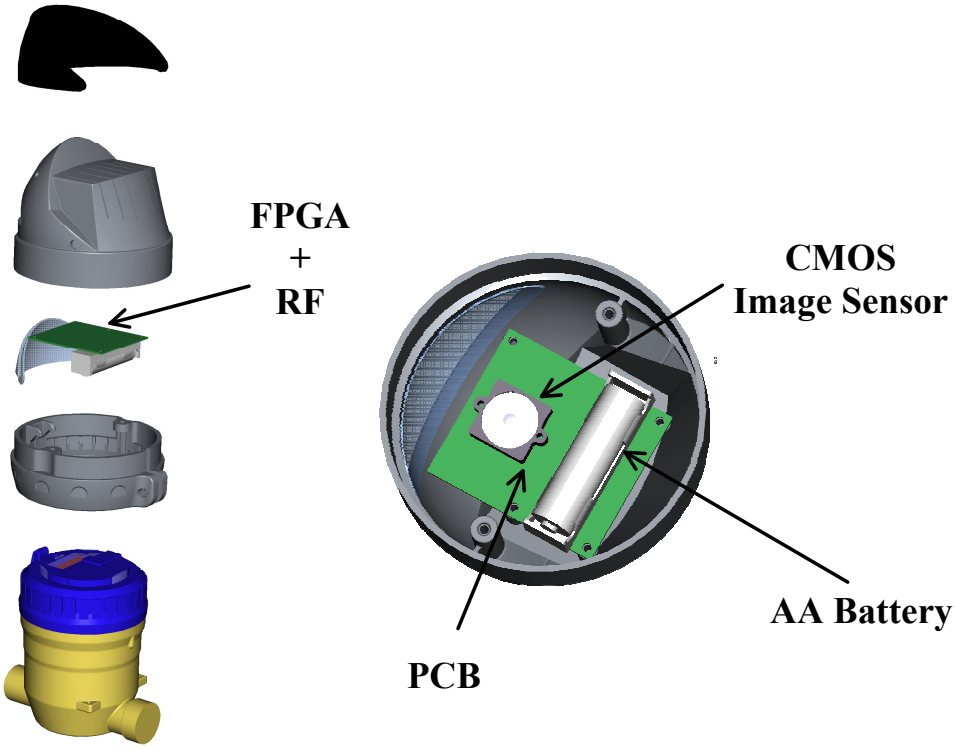

Figure 51 Mirakonta automated meter reading system prototype

In a first version, the automated meter reading (AMR) device was only taking the picture and transmitting it to the remote system. But in order to extend battery life by reducing transmission time it was necessary to perform optical digit recognition inside the device.

The challenge was to use the unused resources of the existing low cost FPGA, which was mainly used for the control of the radio link and ultra low power management. This is a tough goal since good OCR algorithms rely on performing several complex analysis steps on the images and need a non-negligible amount of memory and a microprocessor indeed.

Figure 52 Sensor capture of the meter device

The result was a novel optical digit algorithm very well suited for our specific problem that takes ideas from cross-crossing OCR algorithms to produce a symbol string and use sequence alignment algorithms, often used in genomics, to identify the best matching sequence with a given set of predefined digit patterns.

The idea is quite simple. An image sensor produces pixels in a row-scan fashion. Segmentation and binarization can be performed by some simple data flow processing circuits. Once we have segmented and binarized a character, we process each row of the character and produce a symbol. The symbol basically classifies the row into a set of observed row patterns (Figure 53 a). The symbol generation does not require intermediate memory since a simple FSM that analyzes the occurrence of white pixels depending on the pixel position can be used (Figure 53 b).

a)

Row Characteristic Pattern	Symbol	Binary Value
	·	000
	R	001
	S	010
	L	100
	D	101
	W	111

b)

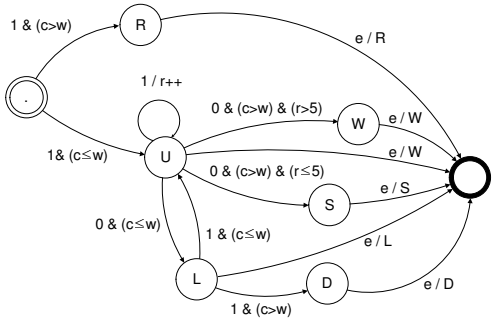

Figure 53 a) Row Patterns and their associated symbol. b) FSM to produce each row symbol

As a result, four sequences of symbols are created, one for each counter digit. When the sequence generation is complete a custom algorithmic machine is used to compute the maximum alignment of the sequences with a set of test patterns that describe the

ideal sequence of each digit. The alignment is computed using the Smith-Watterman algorithm (24).

$$H_{i,j} = \max \begin{cases} 0 \\ H_{i-1,j-1} + S_{i,j} \\ H_{i-1,j} - d \\ H_{i,j-1} - d \end{cases} \tag{24}$$

The digit reference pattern that obtains the higher score is chosen as the recognized digit. This process is repeated for each digit of the AMR.

The overall system's block diagram is depicted in Figure 54.

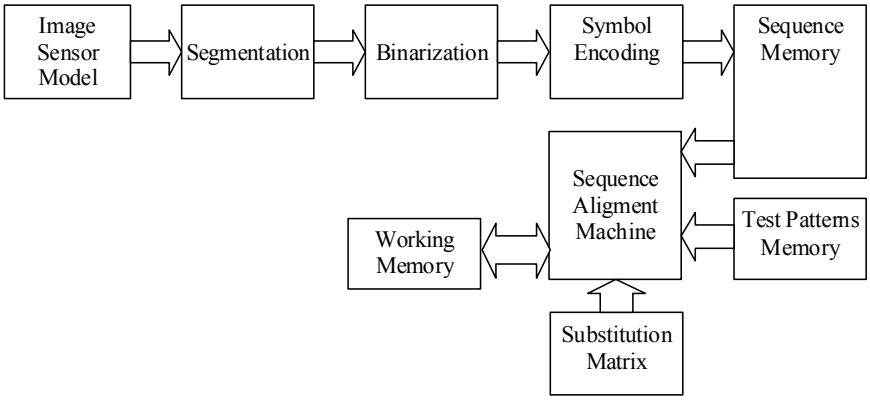

Figure 54 Digit Recognition System Block Diagram

Most of the design units are easily implemented using an structural design style with only an exception: the `EditDistanceProcessor` which is implemented following a RTL design style since it has greater behavioral complexity. Figure 55 shows an schematic view of the implemented system. The module `RowColDeriver` processes the synchronicity signals from the image sensor to create row and column coordinates for each pixel. `Downsampler` module takes the red pixels from the Bayer pattern produced by the sensor, resulting in a downsampled monochrome (red channel) image. Since the digit positions are fixed in relation with a configurable point, `LocationGenerator` module takes this point to derive all the digit positions, which is then used by `LocationMatch` module to determine when the sensor is producing a pixel which is part of a digit. Since the image sensor produces a noisy image the `MedianFilter` module filters the image that is later binarized by the `Threshold` module, which uses the max and min values obtained by the `WindowMean` module in the previous frame. The binary pixels identified as part of a digit are processed by the `SymbolEncoder`, and the resulting symbol is stored in a sequence memory by the `SymbolWriter`. Finally, the `EditDistanceProcessor` computes the local alignment algorithm producing the recognized digit values.

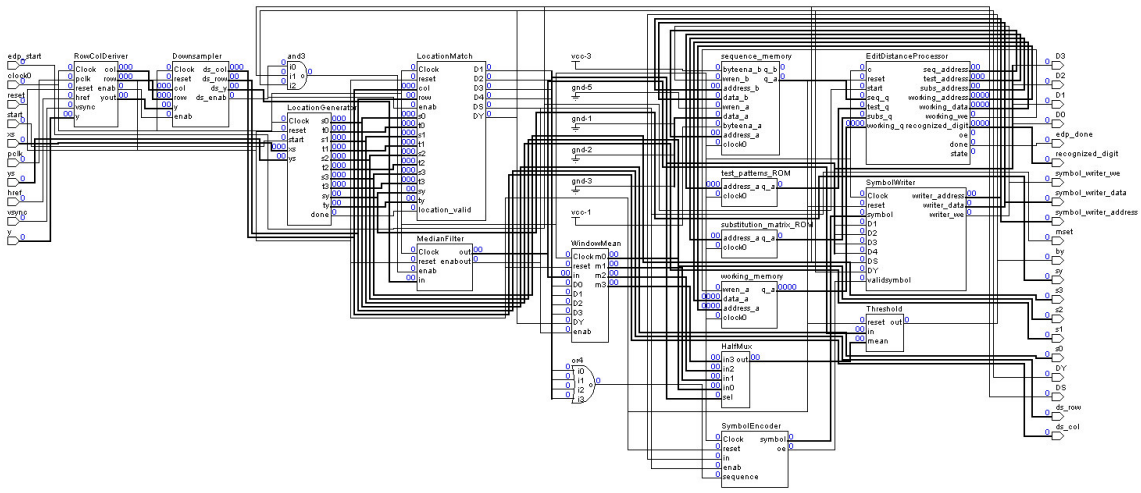

Figure 55 Schematic View of the OCR system

The testbench for such a system is not straightforward, first we need an accurate model of the sensor, and then we need to view the results of the most sensible parts of the process in order to easily identify the possible errors. Two issues have special interest: the resulting binary digits and the sequence patterns produced by the symbol encoder. By looking at the image of the binary digits we can immediately identify if an error occurred in the segmentation or the binarization phase. Errors at the sequence production phase are also evident if you can see the produced sequences in text form. Some custom schematic modules have been developed to allow such a rich interactive test environment (see Figure 56).

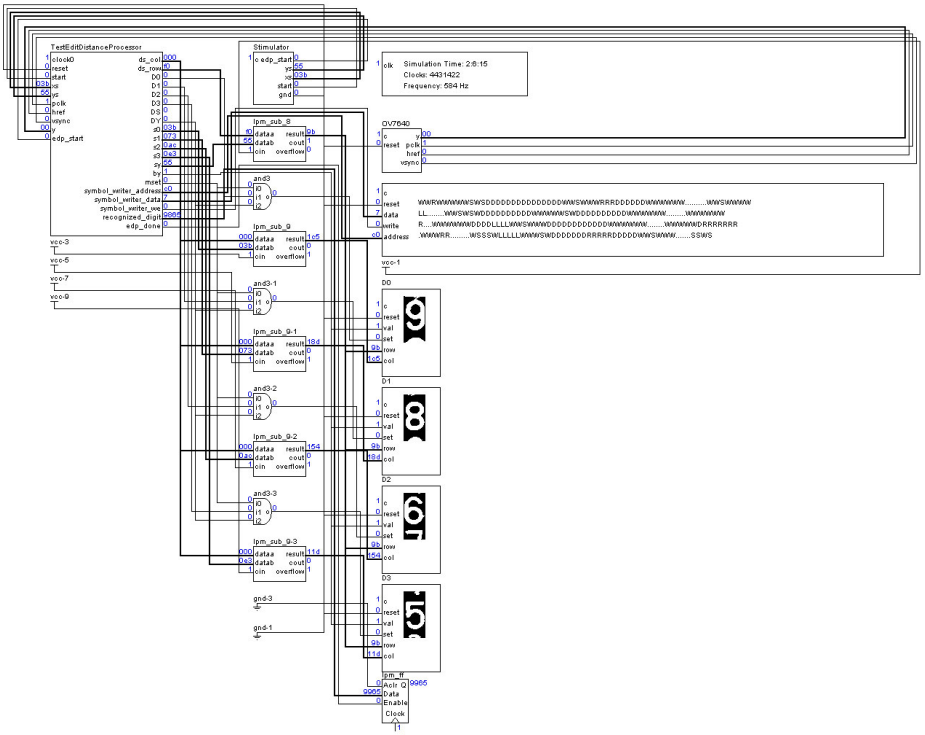

Figure 56 Testbench for OCR system

Additionally, unit tests for each module have been developed and simulated. Table 6 shows the simulation time for each unit test with and without using Jumble for the circuit under test. The maximum achieved speedup is 29.39, much lower than what was achieved with the previous Median Filter design. On one hand, as the complexity of the design are relatively low, the standard simulations run not so slowly. On the other hand, the redirected blocks have a relatively large interface to synchronize at each clock cycle of the simulation (e.g. 130 bits for the complete system) slowing down the Jumble simulation.

Table 6. 3M cycle simulation for different designs

Design	Std. Simulation	Jumble Simulation	Speedup Factor
RowColDeriver unit test	107 s	92 s	1.16
Downsampler unit test	190 s	87 s	2.18
LocationGenerator unit test	553 s	155 s	3.56
LocationMatch unit test	715 s	226 s	3.16
MedianFilter unit test	2,292 s	159 s	14.41
WindowMean unit test	3,318 s	286 s	11.60
Threshold unit test	3,662 s	244 s	15.00
SymbolWriter unit test	3,951 s	266 s	14.85
EditDistanceProcessor unit test	3,939 s	134 s	29.39
OCR System	3,426 s	227 s	15,09

The resource usage of the design, which can be seen in Table 7, is quite low as was intended due to the project requirements.

Table 7 Resource usage of the S30 device

Design Entity	LEs	Memory bits
RowColDeriver	34	0
Downsampler	9	0
LocationGenerator	111	0
LocationMatch	35	0
MedianFilter	263	3120
WindowMean	200	0
Threshold	9	0
SymbolWriter	40	0
SymbolWriter	20	0
EditDistanceProcessor	871	0
Complete System (includes instrumentation and memories)	4,321 (13% of device total)	139,440 (4% of device total)

MPEG Decoder

The third set of tests is performed on several IDCT designs. The different IDCT designs are tested from complex testbench consisting on a full Mpeg 1 [Mpeg1] decoder based on Java. The original code was using multithreading and synchronization between threads to perform the various tasks of the decoder. The code has been refactored to allow its integration into the JHDL simulation framework.

To perform this refactoring, first, a software implementation of the IDCT process was implemented and wrapped into a new JHDL circuit. This IDCT implementation was used to identify the circuit interface but had no time notion, which means that a valid result was produced immediately in a one clock cycle. Nevertheless, the circuit activation is based in the usage of the two signals *start* and *busy* (inspired in BlueSpec methodology [Arvind04]) to make the circuit independent of the number of cycles needed to complete the processing.

Next, the rest of the MPEG decoder code was wrapped in a new JHDL circuit, which includes all the necessary ports to communicate with the external IDCT modules.

While the IDCT block is designed in a RTL way, which in JHDL terminology is called behavioral model, this design style is not convenient for the rest of the circuit as we already have a sequential implementation of the circuit that we would like to reuse with minor modifications.

So, finally the Mpeg1Decoder uses a sequential behavioral design style and consists in `ThreadedLogic` derived class with a simple interface in which the original code main loop has been moved inside the `thread_run` method that calls the `sc_wait` function as needed when interfacing the external IDCT circuit. The sequential code does not directly access the wires values through *get* and *put* methods but modifies the values of interposed variables. A call to `sc_wait`, lead to an invocation of the `thread_clock` function that in turn uses the interposed variables to drive the wires.

```
public class MpeglDecoder extends ThreadedLogic
{
    public static CellInterface[] cell_interface=
    {
        in("reset", 1),
        out("x", 8),
        out("y", 8),
        out("rgb", 24),
        out("set", 1),
        out("dct_start", 1),
        in("dct_busy", 1),
    };

    ...

    MpeglDecoder(Node parent, Wire reset, Wire x, Wire y, Wire rgb, Wire set,
        Wire dct_start, Wire dct_busy,
        Wire[] dct_ins, Wire[] dct_outs, File file)
    {
        super(parent);

        // basic initialization
        ...
    }


    public void reset()
    {
        x.put(this, 0);
        y.put(this, 0);
        rgb.put(this, 0);
        set.put(this, 0);
        dct_start.putB(this, vdct_start = false);
    }

    public void thread_clock()
    {
        x.put(this, vx);
        y.put(this, vy);
        rgb.put(this, vrgb);
        set.putB(this, vset);
        dct_start.putB(this, vdct_start);

        for (int i=0; i<64;i++)
        {
            dct_ins[i].put(this, vdct_ins[i]);
        }
    }

    public void thread_run()
    {
        // Original code main loop
        ...
    }
}
```

To enhance the verification experience, a schematic image viewer is used to get an immediate feedback about the correctness of the system. So, instead of diving into huge waveforms or analyzing endless traces we can just look if the image resulting from the decoding process is the expected one in the circuit schematic view (Figure 57). Note again that this is interactively shown during simulation time, so that we can still use all the other standard features, like waveforms to detect a flaw in the design.

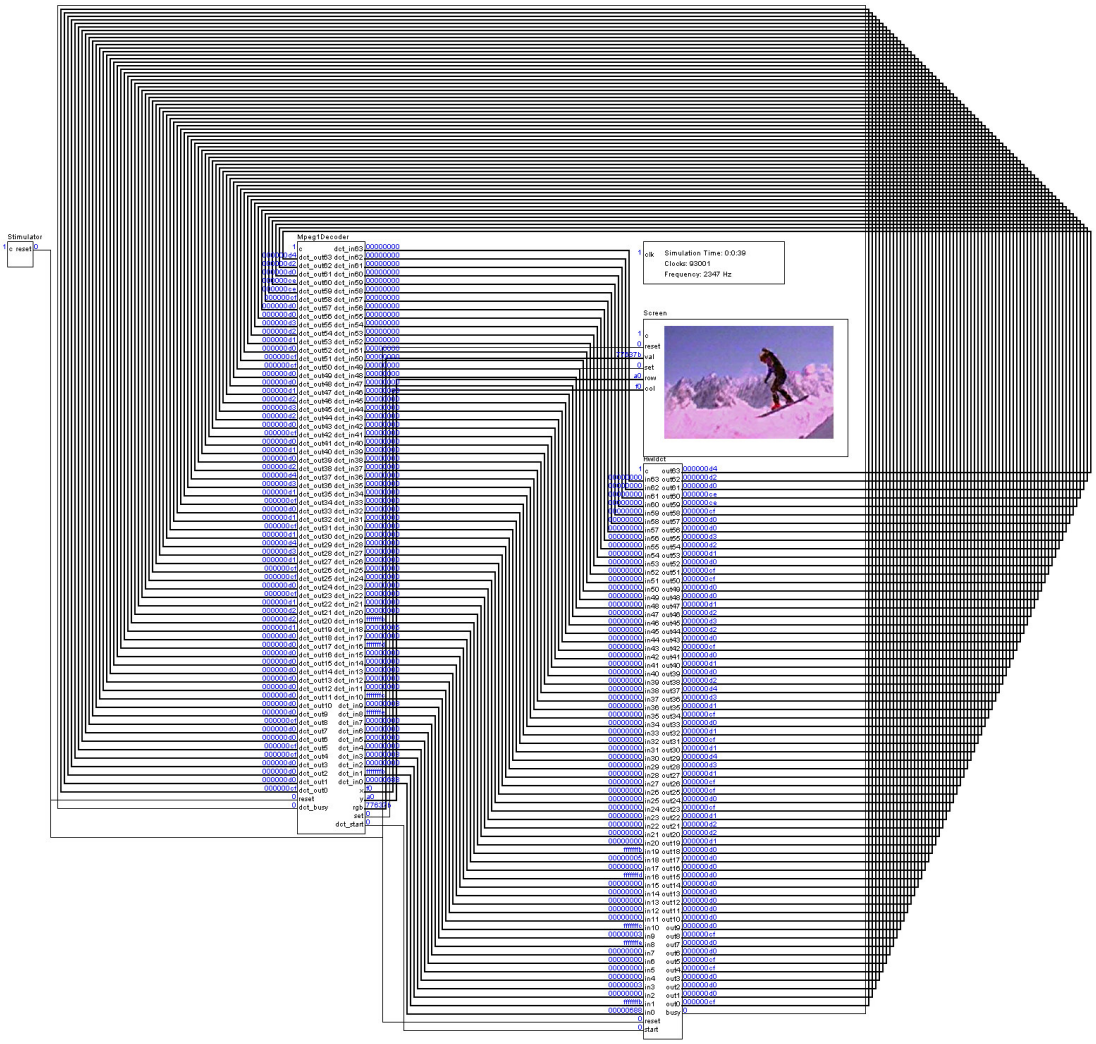

Figure 57 SchematicView of the complex testbench for an Mpeg decoder

The Discrete Cosine Transform

The DCT consists on a transformation of an image block of NxN from space to frequency domain. This transformation of data gives no compression by itself.

MPEG standard uses a value of 8 for N. In this way, simple implementations can be designed, both in Hardware and in Software, with reasonable requirements of memory and computational load.

The mathematical expression of the DCT is (25)

$$Y(u,v) = \frac{C(u)C(v)}{4} \sum_{x=0}^{7} \sum_{y=0}^{7} X(x,y) \cos\frac{(2x+1)u\pi}{16} \cos\frac{(2y+1)v\pi}{16} \quad (25)$$

where

$$C(u), C(v) = \begin{cases} 1/\sqrt{2} & u,v = 0 \\ 1 & otherwise \end{cases} \quad (26)$$

If we define the matrix T as the DCT transform of the identity matrix, we can rewrite the DCT expression in matricial format as

$$Y = TXT^t \quad (27)$$

Using the orthogonal property, the inverse transform (iDCT) can be written as

$$X = T^t YT \quad (28)$$

A typical approach to simplify the computation of the iDCT transform (and the DCT as well) is to separate it using a row column decomposition method. Using the matrix expression, this can be done by using a new Z variable. Finally, we end up with an expression of X consisting of two multiplications by the same constant matrix T (29).

$$\begin{aligned} X &= ZT \\ Z &= T^t Y \\ Z^t &= Y^t T \\ X &= (Y^t T)^t T \end{aligned} \quad (29)$$

Generic multiplier implementations

There are lots of possible designs to implement the iDCT operation. The designs can be classified by the method they used. There are some that use the *Row Column decomposition Method* (RCM) and others that are *Not based on the Row Column decomposition Method* (NRCM).

The designs can also be classified by the input/output interface that can be either serial or parallel.

Table 8. DCT designs

Property	Cho[]	Chang[]	Gong[]
No. of multipliers	$\frac{N^2 \log_2 N}{2}$	N^2	N

No. of adders	$\frac{5N^2 \log_2 N - 2N}{2} + 2$	$N^2 + 3N$	$2N$
Latency	not reported	$2N+1$	1
Cycles/block	1	N	N^2
Speed (pixels/cycle)	N^2	N	1

Constant multipliers proposed designs

If we look at the resulting form of the T matrix after computing (25), we get a matrix as shown below

$$T = \begin{bmatrix} h & h & h & h & h & h & h & h \\ k & i & g & e & -e & -g & -i & -k \\ j & f & -f & -j & -j & -f & f & j \\ i & e & -k & -g & g & k & -e & -i \\ h & -h & -h & h & h & -h & -h & h \\ g & -k & e & i & -i & -e & k & -g \\ f & -j & j & -f & -f & j & -j & f \\ e & -g & i & -k & k & -i & g & -e \end{bmatrix}$$

It is interesting to note that all the coefficients are constant values. The cost of a constant multiplier in hardware is much lower than the cost of a generic multiplier, so my proposal is to use constant multipliers instead of generic ones to compute the iDCT transform using a large combinational circuit when possible.

However some variations can be made in order to decrement the number of used functional units (multipliers, adders, or subtractors) while introducing some sequencing.

The following subsections analyze some of these design variations.

Brute force approach

We can simply implement all the operations of the matrix multiplication. For each cell, we have N multiplications and N-1 adders. As we compute N^2 cells, the number of multipliers is N^3 and number of adders is N^3-N^2. As we are working with N=8, this gives 512 multipliers and 348 adders.

Figure 58 CDFG for constant matrix multiplication

Product term reuse

If we look at the formulas that produce each result matrix cell, we realize that some product terms appear in multiple occasions. For instance if we take $R_{i,3}$ (30) and $R_{i,4}$ (31) we can see that all product terms in $R_{i,3}$ appear in $R_{i,4}$, the only difference is how the product terms are added or subtracted.

$$R_{i,3} = I_{i,0}h + I_{i,1}e - I_{i,2}j - I_{i,3}g + I_{i,4}h + I_{i,l}i - I_{i,6}f - I_{i,7}k \quad (30)$$

$$R_{i,4} = I_{i,0}h - I_{i,1}e - I_{i,2}j + I_{i,3}g + I_{i,4}h - I_{i,l}i - I_{i,6}f + I_{i,7}k \quad (31)$$

If we reuse the terms we end up using 176 terms.

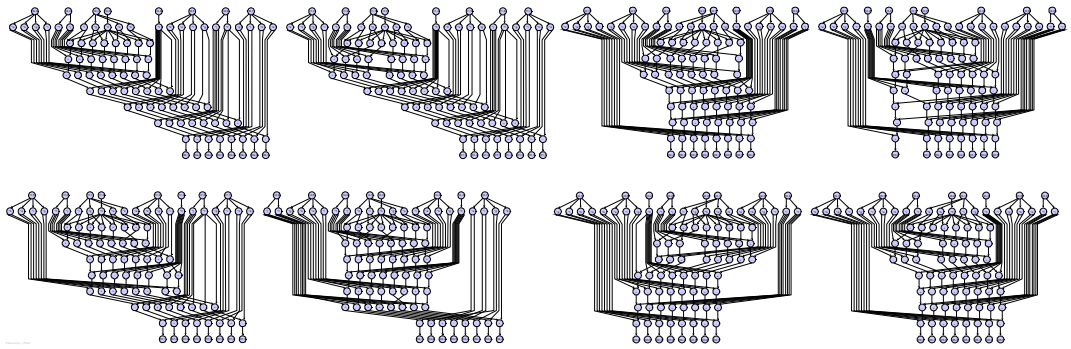

Figure 59 CDFG for constant matrix multiplication

Multipliers row sequencing

In the previous solutions, we compute all product terms in parallel using a big combinational circuit. Each R row is computed using a single I row. As seen in (30) and (31) each R row is computed using the same formulas.

We can see that for each row the constant coefficients are only multiplied by a few input matrix values:

$$
\begin{aligned}
e &\leftarrow I_{i,1}, I_{i,3}, I_{i,5}, I_{i,7} \\
f &\leftarrow I_{i,2}, I_{i,6} \\
g &\leftarrow I_{i,1}, I_{i,3}, I_{i,5}, I_{i,7} \\
h &\leftarrow I_{i,0}, I_{i,4} \\
i &\leftarrow I_{i,1}, I_{i,3}, I_{i,5}, I_{i,7} \\
j &\leftarrow I_{i,2}, I_{i,6} \\
k &\leftarrow I_{i,1}, I_{i,3}, I_{i,5}, I_{i,7}
\end{aligned}
\tag{32}
$$

This gives 22 multipliers per row. And considering all rows (8) gives 176, which is the previous result.

If we multiplex the rows in 8 cycles we can produce all the product terms in 8 cycles using only 22 multipliers.

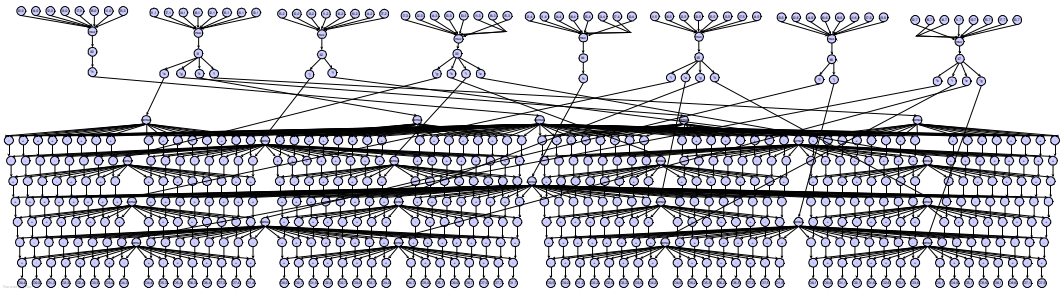

Figure 60 CDFG for constant matrix multiplier

Adders row sequencing

We do the same for the adders' network. We have 56 adders for each matrix row. If we compute the whole matrix at once this means 56*8 adders.

Using only 56 can reduce the area usage.

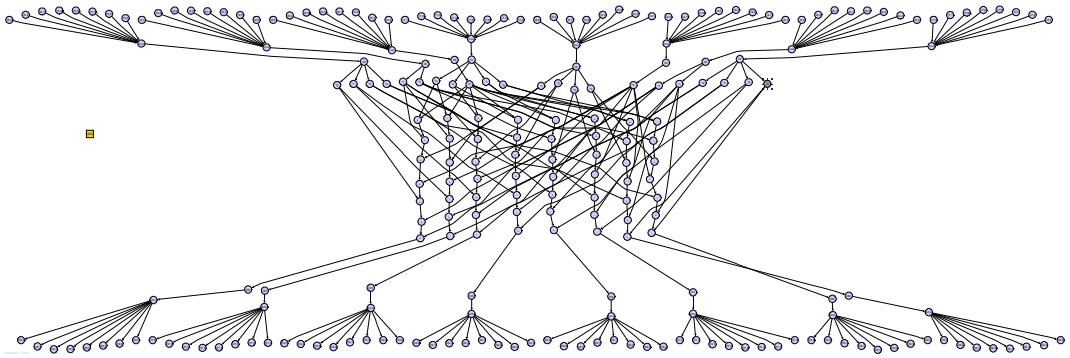

Figure 61 CDFG from row computation using 22 multipliers

Multipliers column sequencing

From (32) we can see that each multiplier takes 4 different columns at most. So we can share the constant multipliers among the different terms by serializing the inputs and deserializing the outputs.

As a result we use 7 constant multipliers, which is the minimum possible number of constant multipliers.

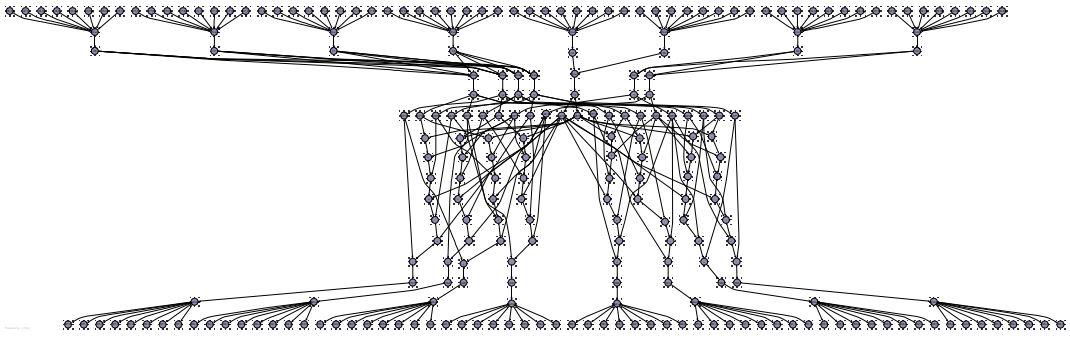

Figure 62 CDFG from row computation using 7 multiplexed multipliers

Constant Multiplier Implementation

A constant multiplier can be implemented as a sequence of add and shift operations. I denote this type of constant multiplier as AS (add shift) multiplier.

For instance $Y = X \cdot 15$, is expressed as $Y = X \cdot 1111$ in binary notation and can be decomposed as follows as $Y = X \cdot 2^{3+} X \cdot 2^2 + X \cdot 2^1 + X \cdot 2^0$

As the shift operation comes for free in digital logic, in this example we only need 3 adders to compute the product. In fact, the number of used adders depends on the active bits in the constant multiplier. The more “ones” in the constant value, the more adders we use.

However, this can be further optimized if we notice that 15 = 16-1. So we can rewrite the previous example as Y=X·(16-1), which in binary notation becomes Y= X·(10000 - 1) = X·10000 – X. Finally we get $Y=X\cdot2^4-X\cdot2^0$.

In essence, we introduced the subtract operation to obtain a much more compact expression that allows to save computing resources. In this case we only need 1 subtractor to compute the product, compared with the previous 3 adders. I denote this type of constant multiplier as ASS (add subtract shift) multiplier.

Design Verification

The design variations have been tested in the complex Mpeg1Decoder testbench. The brute force approach has been ignored, as it does not fit into the available FPGA. The first test consists on implementing the product term reuse with AS type constant multipliers and replace the whole `ConstantMatrixMultiplier` module for FPGA execution. As it is a pure combinational circuit, it is possible to reuse its hardware implementation when using jumble simulation. As simulator will evaluate the propagate function in different times, we can redirect the two different constant matrix multipliers towards the single hardware implementation. The second test follows this approach to execute the two `ConstantMatrixMultiplier` modules present on the iDCT design. The third test introduces the use of ASS type of constant matrix multipliers. About an 11% of the area of the original module is saved in this step (see first and second row of Table 10). The fourth test repeats the trick of reusing the hardware implementation but now with the ASS based constant matrix multiplier. The fifth test uses the design with 22 multipliers and 56 adders, but as the design is smaller, we replace the entire `HwIdct` block for FPGA execution. Finally, the sixth test uses the design with 7 multipliers and 56 adders.

Table 9. 3M cycle simulation for different designs

Design	Std. Simulation	Jumble Simulation	Speedup Factor
ConstantMatrixMultiplier	183,186 s	100,489 s	1.82
2 ConstantMatrixMultiplier	183,186 s	11,619 s	15.76
ConstantMatrixMultiplier using ASS multipliers	143,957 s	76,654 s	1.87
2 ASS ConstantMatrixMultiplier	143,957 s	11,384 s	12.64
HwIdct (IDCT64_22cm_56ad)	36,133 s	3,569 s	10.12
HwIdct (IDCT64_7cm_56ad)	27,372 s	3,558 s	7.69

As shown in Table 9, simulation speedups go from 2 to 12 approx. It is very interesting to observe that reusing combinational blocks have a great impact in the simulation speedup (goes from 1.82 to 15.76). Also noticeable is that the speedup achieved when replacing the whole `HwIdct` is lower than the previous one (achieved when replacing some of its parts). This is due to the width of the interface that must be synchronized in both cases; the interface of the `HwIdct` has 4097 bits while the `ConstantMatrixMultiplier` has an interface of 2049 bits. The greater the interface, the more time must the simulator dedicate to transfer data to the real hardware.

Table 10. Resource usage of the S30 device

Block of the design	LEs Complete Design	LEs Constant Matrix Multiplier	Computing Elements	Memory bits Complete Design
ConstantMatrixMultiplier	26,800 (82% of device total)	20,576 (63% of device total)	176 M 448 A/S	1,152 (<1% of device total)
ConstantMatrixMultiplier using ASS multipliers	24,592 (75% of device total)	18,304 (56% of device total)	176 M 448 A/S	1,152 (<1% of device total)
CMatrixMultiplier22	20,879 (64% of device total)	14,716 (45% of device total)	22 M 448 A/S	1,152 (<1% of device total)
CMatrixMultiplier56	9,546 (29% of device total)	3,906 (12% of device total)	22 M 56 A/S	1,152 (<1% of device total)
Hwldct (IDCT64_22cm_56ad)	14,150 (43% of device total)	3,616 & 3,906 (12% of device total)	22 M 56 A/S	2,097 (<1% of device total)
Hwldct (IDCT64_7cm_56ad)	13,680 (42% of device total)	3,330 & 3,733 (11% of device total)	7 M 56 A/S	2,097 (<1% of device total)

Chapter 7

Conclusions and Future Work

I have presented a method to interactively select a part of a design during a simulation session and download it into a supported hardware platform for hardware execution. The system could reduce simulation time by some orders of magnitude providing a convenient system for HIL verification, but this would be achieved only if Amdahl's α is very close to 1. However, in more realistic experiments, I have got a speedup factor of 10 to 30. This could be optimized by implementing faster methods to transfer the data to the interface.

In future work, I will try to improve the obtained speedups through the use of larger blocks instead of bit-by-bit scan chain. Another interesting idea is to directly map the input and output interface to the host memory and avoid the use of the shift registers. The actual T_{Jumble} is proportional to the width of the interface as formulated in (8). This approach would significantly reduce by a factor of 32 (33) the time spent in inputs and outputs transfers as would benefit from burst PCI transfers, since each PCI bus read/write operation would be enough to place inputs and outputs.

$$T_{Jumble} = \left\lceil \frac{W_{\text{int}\,erface}}{32} \right\rceil \cdot T_{PCI} \qquad (33)$$

The current system is limited to download a single block at a time. Future work will address the need to download multiple independent blocks.

Another pitfall of the system is that it does not provide a method to verify that the replaced block corresponds to the block that is currently programmed into the hardware platform. This problem will be addressed by adding some metadata information into the synthesized design so it can be compared with the object instructed to be replaced.

The four-command process could be merged into a single command that offers a simple single push button solution for hardware emulation of selected blocks.

Jumble is practical for design verification but could also be useful in the deployment of the final designs, especially if the designs that we are verifying are though to be PCI coprocessors. With some more work, and following a similar process that have been used, we could create an automated way of wrapping the design and create an API so that hardware functions are directly usable from end user applications. This API could be a Windows DLL, a COM object or a Java class.

As has been stressed through this work, JHDL has the ability to manipulate the circuit hierarchy during simulation sessions. Here this is used to substitute a circuit and redirect its interface to its hardware implementation. By doing this, we can compare different simulation sessions and verify that the system is equivalent. However, it can be difficult to ensure the equivalence if we do not keep track of all the signals that get in an out of the circuit, and this is obviously prohibitive for most designs. As a possible improvement, we could avoid removing the software model of the hardware circuit and maintain both: the software implementation and its hardware implementation through

the redirector together with an additional module aimed to verify its equivalence at each clock cycle. This would eliminate any speedup buy could offer a safe intermediate step before going to the pure hardware implementation.

Finally, Jumble can be very useful for complex designs in which simulation costs are considerable. In our research group, we are currently working on projects where this is the case, like NoC and Soft Core simulations.

Apendix A

Coding effort

In order to clarify the extension of this work we detail the various contributions.

Table 11 Coding effort of the various contributions to this work

Module	Language	Authors	Source Code Files	Source Code Lines
Altera support	JHDL	Jordi Farré, David Castells	125	29,491
Median Filter Test Case	JHDL	David Castells	21	1,679
Mpeg Test Case	JHDL/Java	David Castells	126	39,236
OCR Test Case	JHDL	David Castells	108	19,154
PCI renenumerator	C++	David Castells	20	4,339
VHDL netlister	VHDL/JHDL/ Java	Alexis Morugó, David Castells	3	2,116
PLD Platform Model	JHDL	David Castells	49	5,808
JNI native interface to PLD board	C++/Java	David Castells	4	1,961
Wrapping infrastrucure	JHDL/Java	David Castells	20	3,159
Quartus Automation	Java	David Castells	5	528
Threaded Logic	JHDL/Java	David Castells	1	287
Common utility logic	JHDL	David Castells	62	7,567
TOTAL			**544**	**115,325**

References

[Aldec] Aldec, Inc., http://www.aldec.com

[Amdahl67] G.M. Amdahl. Validity of single-processor approach to achieve large-scale computing capability. Proceedings of AFIPS, 483-485, 1967.

[Altera96] Altera Corporation, “LPM (Library of Parametised Modules) Quick Reference Guide”, December 1996. http://www.altera.com/literature/catalogs/lpm.pdf

[Altera00] Altera Corporation, “Instatiating LPM in EDIF”

[Altera01] Altera, San Jose CA. SignalTap Embedded Logic Analyzer Megafunction, April 2001 ver.2.0

[Altera05] Altera Corporation, Altera DSP Builder Reference Manual, January 2005, version 2.1.3, http://www.altera.com

[Alpha] Alpha Data Systems, Simulink Board Support Blockset, http://www.alpha-data.com/simulink_bsb dsheet.html

[Annapolis04] Annapolis Micro Systems, “Wildcard Reference Manual Rev 3.4,” Annapolis Micro Systems, Inc, Annapolis, MD, 2004, (http://www.annapmicro.com/)

[Arnold92] J. Arnold, D. Buell and E. Davis, "Splash II", 4th ACM Symposium on Parallel Algorithms and Architectures, San Diego, CA, USA, pp. 316-322, 1992.

[Arnold93] J. M. Arnold, "The Splash 2 software environment", in *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, D. A. Buell and K. L. Pocek, Eds., Napa, CA, Apr. 1993, pp. 88-93.

[Arvind04] Arvind, Rishiyur S. Nikhil, Daniel L. Rosenband, and Nirav Dave. “Highlevel Synthesis: An Essential Ingredient for Designing Complex ASICs.” in *Proceedings of ICCAD'04*, San Diego, CA, 2004.

[Ayuso06] N. Ayuso, J. Pico, N. Benitez, J. Carrabina, E. Pons, B. Martinez, D. Castells-Rufas, M. Monton, L. Terés, J. Merino, E. Gonzalez, A. Guerendiain, G. Alvarez, C. Amuchastegui. UNIVERSAL RECONFIGURABLE SYSTEM AND METHOD FOR THE REMOTE READING OF COUNTERS OR EQUIPMENT COMPRISING VISUAL INDICATORS, European patent number 11446, 2006.

[Axis] Axis Systems, Inc., http://www.axiscorp.com

[Babb97] J. Babb, R. Tessier, M. Dahl, S. Hanono, D. Hoki, and A. Agarwal. “Logic emulation with virtual wires”. *IEEE Transactions on CAD*, 16(6):609–626, Jun. 1997

[Ballagh01] J. Ballagh, P. Athanas, and E. Keller, “Java Debug Hardware Models using JBits,” *8th Reconfigurable Architectures Workshop*, San Francisco, CA, April 27, 2001.

[Banarjee99] P. Banarjee et al, "MATCH: A MATLAB Compiler for Configurable Computing Systems”. Technical Report, Center for Parallel and Distributed Computing, Northwestem University, Aug. 1999, CPDC-TR-9908-013.

[Banarjee00] P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck, M. Haldar, P. Joisha, A. Jones, A. Kanhare, A. Nayak, S. Periyacheri, M. Walkden, and D. Zaretsky, “A MATLAB Compiler for Distributed Heterogeneous Reconfigurable Computing Systems”, *International Symposium on FPGA Custom Computing Machines (FCCM'00)* IEEE Computer Society Press, Los Alamitos, Calif., 2000.

[Basu98] A. Basu, R. S. Mitra, and P. Marwedel. “Interface synthesis for embedded applications in a co-design environment”. In *11th IEEE International conference on VLSI design*, pages 85{90, C, 1998.

[Bauer94] T. J. Bauer. “The design of an efficient hardware subroutine protocol for FPGAs”. Master’s thesis, MIT, 1994.

[Bauer98] J. Bauer, M. Bershteyn, I. Kaplan, and P. Vyedin. “A reconfigurable logic machine for fast event-driven simulation”. Design Automation Conference, .June 1998.

[Bazeghi05] C. Bazeghi, F. J. Mesa-Martinez, J. Renau: “µComplexity: Estimating Processor Design Effort”. *Proceedings of MICRO 2005*: pp 209-218

[Bellows98] P. Bellows and B. L. Hutchings. “JHDL - an HDL for reconfigurable systems”. In J. M. Arnold and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 175-184, Napa, CA, April 1998.

[Bellows04] Peter Bellows. ”High-Visibility Debug-by-Design for FPGA Platforms”. ERSA 2004: 247-258

[Benitez04] Domingo Benitez, “Análisis de Prestaciones de Coprocesadores Reconfigurables”, in *Proceedings of IV Jornadas de Computación Reconfigurable y Aplicaciones JCRA*. Barcelona, September, 2004.

[Bershad90] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. "Lightweight Remote Procedure Call". ACM Trans. on Computer Systems, 8(l), February 1990.

[Birkner98] J. Birkner, “From Simple PALs to High-Speed, High-Density Leading Edge FPGAs, Their Technologies and Applications” MAPLD 98 Proceedings, 1998.

[Bishop97] W.D. Bishop, W.M. Loucks, “A Heterogeneous Environment for Hardware/Software Cosimulation,” *Proceedings of the IEEE Annual Simulation Symposium*, 1997, pp. 14-22.

[Borgatti96] M. Borgatti, R. Rambaldi, G. Gori, R. Guerrieri, “A Smoothly Upgradable Approach to Virtual Emulation of HW/SW Systems,” *Proceedings of the International Workshop on Rapid System Prototyping*, 1996, pp. 83-88.

[Borgatti97] M. Borgatti, E. Cevenini, R. Rambaldi, M. Felici, A. Ferrari, R. Guerrieri, "Fast board-level prototyping of a speech recognition system using virtual emulation" *Proceedings of the 8th IEEE International Workshop on Rapid System Prototyping*, 1997.

[Budiu02] M. Budiu, M. Mishra, A. Bharambe, and S. C. Goldstein, "Peer-to-Peer Hardware-Software Interfaces for Reconfigurable Fabrics," in FCCM, 2002.

[Butts92] M. Butts, J. Batcheller, and J. Varghese, “An efficient logic emulation system”, in *IEEE 1992 International Conference on Computer Design: VLSI in Computers and Processors*, 1992, pp. 138–141.

[Canellas00] Canellas, N., Moreno, J. M., “Speeding up hardware prototyping by incremental Simulation/Emulation”, in *Proceedings of 11th International Workshop on Rapid System Prototyping*, 2000.

[Cardoso98] J. M. P. Cardoso, H. C. Neto, “Towards an Automatic Path from JavaTM Bytecodes to Hardware Through High-Level Synthesis,” In Proc. of the 5th IEEE International Conference on Electronics, Circuits and Systems, Lisbon, Portugal, September 7-10, 1998, pp. 85-88.

[Cardoso99] J. M. P. Cardoso and H. C. Neto, “Macro-based hardware compilation of java bytecodes into a dynamic reconfigurable computing system,” in Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines (K. L. Pocek and J. M. Arnold, eds.), (Napa, CA), p. n/a, IEEE, 1999.

[Carloni02] L.P. Carloni, F. De Bernardinis, A. Sangiovanni-Vincentelli, and M. Sgroi, “The art and science of integrated systems design”, in *Proc. of 28th European Solid-State Circuits Conference (ESSCIRC 2002)*, 2002,Florence, Italy, 25-36.

[Cho01] 조영철 ; 최기영, "An approach to combining emulation and simulation for efficient debugging of system-on-chip design", *CAD 및 VLSI 설계 연구회 학술 발표회 논문집*, pp.210-214, 2001. 5

[Chou92] P. Chou, R. Ortega, and G. Borriello. “Synthesis of the hardware/software interface in microcontroller-based systems”. In *Proceedings of ICCAD*, pp.488–495, Nov. 1992.

[Cong94] Cong, J., and Ding, Y. “On Area/Depth Trade-off in LUT-Based FPGA Technology Mapping”, *IEEE Transactions on VLSI Systems*, vol. 2, no. 2, 137-148, June 1994.

[Cosic92] K. Cosic, I. Kopriva, I. Miller, “Workstation for Integrated System Design and Development,” Simulation, vol 58, n 3, Mar. 1992, pp. 152-162.

[Çakır01] M. Çakır, E. Grimpe, “ProtoEnvGen: Rapid ProtoTyping Environment Generator”, in *Proceedings of VLSI-SOC 2001.*

[Çakir03] M. Çakir, E. Grimpe, “HW-Driven Emulation with Automatic Interface Generation”, in *Proceedings of FPL 2003*, pp. 627-637, 2003.

[DeHon04] A. DeHon, J. Adams, M. DeLorimier, N. Kapre, Y. Matsuda, H. Naeimi, M. Vanier, M. Wrighton. “Design Patterns for Reconfigurable Computing”. FCCM 2004: 13-23.

[Dick01] C. H. Dick and H. M. Pedersen, “Design and Implementation of High-Performance FPGA Signal Processing Datapaths for Software Defined Radios”, *Embedded Systems Conference* Apr. 2001.

[Dozza98] D.Dozza, R.Rambaldi, M.Borgatti and R.Guerrieri , “OMI-Compliant Model for Virtual Emulation” in *Proceedings of the Ninth International Workshop on Rapid System Prototyping*, 1998.

[Edenfeld03] D. Edenfeld, A. B. Kahng, M. Rodgers, and Y. Zorian. “2003 Technology Roadmap for Semiconductors”. *IEEE Computer*, 37(1):47-56, 2004.

[Edif] http://www.edif.org

[Edwards97] M. Edwards: “Software Acceleration Using Coprocessors: Is it Worth the Effort ?” *Proceedings of 5th International Workshop on Hardware/Software Codesign* (Codes/CASHE‘97), pp. 135-139, Braunschweig 1997

[Fischer98] F. Fischer, A. Muth, G. Fiirber “Towards interprocess communication and interface synthesis for a heterogeneous real-time rapid prototyping envimnment”. *6th International Workshop on Hardware/Software CO-Design* (Codes/CASHE ’98). Seattle, USA, 1998.

[Fritsch99] Ch. Fritsch, J. Haufe, Th. Berndt: Speeding Up Simulation by Emulation - A Case Study. in Design, Automation and Test in Europe Conference, Munich 1999, User Forum, 127-134

[George99] George, Alan D., Ryan B. Fogarty, Jeff S. Markwell, and Michael D. Miars, “An Integrated Simulation Environment for Parallel and Distributed System Prototyping,” Simulation, Vol. 75, No. 5, May 1999, pp. 283-294.

[Graham00] P. Graham, B. Hutchings, and B. Nelson, “Improving the fpga design process through determining and applying logical-to-physical design mappings”, Technical Report CCL-2000-GHN-1, Brigham Young University, Provo, UT, April 2000.

[Graham01] P. S. Graham. “Logical Hardware Debuggers for FPGA-based Systems”. PhD thesis, Brigham Young University, Provo, UT, USA, December 2001.

[Graham01b] Paul Graham, Brent Nelson, and Brad Hutchings, “Instrumenting Bitstreams for Debugging FPGA Circuits,” 2001 *IEEE Symposium on Field-Programmable Custom Computing Machines*, Rohnert Park, California April 29 - May 2, 2001.

[Guerendiain05] A. Guerendiain, G. Alvarez, C. Amuchastegui, N. Ayuso, J. Pico, N. Benitez, J. Carabina, E. Pons, B. Martinez, D. Castells, M. Monton, L. Teres, J.L. Merino. ”Método y Sistema Universal y reconfigurable de lectura remota de contadores o equipos provistos de indicadores visuales”. Patent Number: P200500991, Spain, April, 2005

[Hanono95] S. Z. Hanono, "Innerview hardware debugger: A logic analysis tool for the virtual wires emulation system," M.S. Thesis, Massachusetts Univ. Technol., 1995.

[Haufe98] J. Haufe, P. Schwarz, T. Berndt, J. Große, “Accelerated Logic Simulation by Using Prototype Boards”. In Proceedings of Design Automation and Test in Europe, Paris 1998, pages 183-189.

[Hemani04] A. Hemani, “Charting the EDA Roadmap”, *The Chip, IEEE Circuits & Devices Magazine*, November-December, 2004.

[Hoenicke01] J. Hoenicke, “Java Optimize and Decompile Environment (JODE) program”. http://jode.sourceforge.net/ , 2001

[Hunt02] W. Hunt, "Introduction: Special issue on microprocessor verification", in *Formal Methods in System Design*, Kluwer Academic Publishers, 2002.

[Hutchings99] B. Hutchings, P. Bellows, J. Hawkins, S . Hemmert, B. Nelson, and M. Rytting, "A cad suite for high-performance fpga design," in *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines* (K. L. Pocek and J. M. Arnold, eds.), (Napa, CA), p. n/a, IEEE Computer Society, IEEE, April 1999.

[Hutchings00] B. L. Hutchings, B. E. Nelson, "Using general-purpose programming languages for FPGA design," in *Proc. 37th Design Automation Conf.*, Los Angeles, CA, June 2000, pp. 561-566.

[Hutchings00b] B. Hutchings, B. Nelson, M. Whirthlin, “Designing and Debugging Custom Computing Applications”, *IEEE Design & Test of Computers*, January-March 2000.

[Hutchings01] B. L. Hutchings and B. E. Nelson. “Unifying simulation and execution in a design environment for FPGA systems”. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9:201-205, February 2001.

[Hutchings04] B. Hutchings and B. Nelson. “Giga Op DSP On FPGA”. *In Proceedings of ICASSP 2001*, May 2001.

[Hwang98] S. Hwang, T. Blank, K. Choi, “Fast Functional Simulation : An Incremental Approach,” in *IEEE Transactions*. CAD, 1988.

[Hwang01] J. Hwang, B. Milne, N. Shirazi, J. D. Stroomer, “System Level Tools for DSP in FPGA”, in *Proceeding of FPL2001*, pp 534-543, 2001.

[Indrusiak03] L. S. Indrusiak, F. Lubitz, M. Glesner, R. A. L. Reis, “Ubiquitous Access to Reconfigurable Hardware: Application Scenarios And Implementation Issues”. *Proceedings of DATE ‘03*, Munich, 2003. p.940 – 945.

[Indrusiak05] Indrusiak, L.S.; Prudêncio, R. B.; Glesner, M. Modeling and Prototyping of Communication Systems using Java: a Case Study. In: *Proceedings of 16th IEEE International Workshop on Rapid System Prototyping (RSP)*, 2005, Montreal,Canada.

[Jimenez05] D. F. Jiménez, L. S. Indrusiak, M. Glesner, “Proxy-based Integration of Reconfigurable Hardware within Simulation Environments: Improving E-Learning Experience in Microelectronics”, in *Proceedings of IEEE International Conference on Microelectronic Systems Education* (MSE’05), 2005.

[Kahng00] A. B. Kahng, "Futures for DSM Physical Implementation: Where is the Value, and Who Will Pay?", 12th DA Show keynote, July 14, 2000.

[Kim96] K. Kim, Y. Kim, Y. Shin, K. Choi, “An Integrated Hardware-Software Cosimulation Environment with Automated Interface Generation,” *Proceedings of the International Workshop on Rapid System Prototyping*, 1996, pp. 66-71.

[Kim04] Y. Kim, W. Yang, Y. Kwon, C. Kyung, “Communication-Efficient Hardware Acceleration for Fast Functional Simulation”, *Proceedings of DAC 2004*, June, 2004, pp 293-298.

[Krukowski99] A. Krukowski and I. Kale. “Simulink/matlab-to-vhdl route for full custom/FPGA rapid prototyping of DSP algorithms”. In *Matlab DSP Conference (DSP99)*, Tampere, Finland, November 16-17 1999.

[Krupnova00] H. Krupnova, G. Saucier, “FPGA-Based Emulation: Industrial and Custom Prototyping Solutions”, in *Proceedings of FPL 2000*.

[Kudlugi01] M. Kudlugi, S. Hassoun, C. Selvidge, and D. Pryor. “A Transaction-Based Unified Simulation/Emulation Architecture for Functional Verification”. In *ACM/IEEE Design Automation Conference (DAC)*, June 2001.

[Kulmala] A. Kulmala, “HDL Verification – Simulation Engines”, course materials of System Design I (TK2400) of Tampere University of Technology.

[Le97] T. Le, F.-M. Renner, M. Glesner, “Hardware in-the-loop Simulation - a Rapid Prototyping Approach for Designing Mechatronics Systems,” *Proceedings of the International Workshop on Rapid System Prototyping*, 1997, pp. 116-121.

[Lee01] Edward A. Lee. “Overview of the Ptolemy Project. Technical Memorandum UCB/ERL M01/11”. March 6, 2001.

[Lee01b] Edward A. Lee, "Design Methodology for DSP" Final Report 2000-01, University of California at Berkeley, 2001.

[Lee03] Edward A. Lee, Stephan Neuendorfer, and Michael J. Wirthlin. “Actor-oriented design of embedded hardware and software systems”. *Journal of Circuits, Systems, and Computers*, 12(3):231 – 260, 2003.

[Lehmann02] T. Lehmann, “Towards Device Driver Synthesis,” PhD. Thesis, University of Paderborn, 2002.

[Liu03] Jie Liu and Edward A. Lee, “Timed Multitasking for Real-Time Embedded Software,” *IEEE Control Systems, special issue on Software-Enabled Control*, vol. 23, no. 1, January, 2003, pp 65-75.

[Liu04] Jie Liu, Johan Eker, Jorn W. Janneck, Xiaojun Liu, and Edward A. Lee, “Actor-Oriented Control System Design: A Responsible Framework Perspective” *IEEE Trans. on Control System Technology*, vol. 12, No. 2, March 2004, pp. 250-262.

[Lyr] Lyr Signal Processing, DSP Link, FPGA Link: DSP + FPGA co-design, hardware-in-the-loop co-simulation, http://www.signal-lsp.com/

[Ma03] J. Ma, "Incremental Design Techniques with Non-Preemptive Refinement for Million-Gate FPGAs", PhD Thesis, Virginia Tech, January 2003.

[Maheshwari97] R. Maheshwari, S. S. S. P. Rao and P.G. Poonacha, “FPGA implementation of median filter”, *10th International Conference on VLSI Design*, Jan’97, pp-523-524.

[Mentor] http://www.mentor.com/products/fv/emulation/

[Model] http://www.model.com/

[Molina07] A. Molina, O. Cadenas, “Functional verification: approaches and challenges”. *Latin American Applied Research*, January 2007, vol.37, no.1, p.65-69. ISSN 0327-0793.

[Mpeg1] MPEG1, ISO/IEC 11172-2:1993 “Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s, Part 2: Video”

[Mueller01] W. Mueller, J. Ruf, D. Hoffmann, J. Gerlach, T. Kropf, W. Rosenstiehl, "The Simulation Semantics of SystemC", DATE 2001, Munich, 2001

[Muller97] Pierre-Alain Muller, “Modelado de objectos con UML”. Ed. Gestión 2000, Barcelona, 1997. ISBN 84-8088-226-3.

[Nakamura04] Y. Nakamura, K. Hosokawa, I. Kuroda, K. Yoshikawa, T. Yoshimura, “A Fast Hardware/Software Co-Verification Method for System-On-a-Chip by Using a C/C++ Simulator and FPGA Emulator with Shared Register Communication”, *Proceedings of DAC 2004*, San Diego, CA, USA, June 2004.

[Ofner04] E. Ofner, J. Nurmi, J. Madsen, J. Isoaho, and H. Tenhunen, ”SoC-Mobinet – R&D and Education in System-on-Chip Design,” in *Proceedings of International Symposium on System-on-Chip*, November 2004, Tampere, Finland, pp. 77-80.

[Osiris] http://splish.ee.byu.edu/lab/osiris/osiris.html

[Poetter04] A. Poetter; J. Hunter; C. Patterson; P. Athanas; B. Nelson and N. Steiner: JHDLBits: The Merging of Two Worlds. Proceedings of the *14th Field-Programmable Logic and Applications* (FPL’04), Leuven, Belgium, Springer 2004 ISBN 3-540-22989-2, pp. 414 - 423.

[Price01] T. Price and C. Patterson, “Reconfigurable breakpoints for co-debug”, in *Field-Programmable Logic and Applications. Proceedings of the 11th International Workshop*, FPL 2001, G. Brebner and R. Woods, Eds., Belfast, Northern Ireland, August 2001, vol. 2147 of *Lecture Notes in Computer Science*, pp. 473–482, Springer-Verlag.

[Quickturn] Quickturn Home Page, //www.quickturn.com

[Ramaswamy02] Ramaswamy Ramaswamy, Russel Tessier: “The Integration of SystemC and Hardware-Assisted Verification”, in Proc. FPL 2002

[Ramon05] E. Ramon, J. Carrabina. "Using FPGAs for Software-Defined Radio Systems: a PHY layer for an 802.15.4 transceiver". *V Jornadas de Computación reconfigurable y Aplicaciones (JCRA).* Granada, 14-16 de September, 2005.

[Sangiovanni01] Alberto Sangiovanni-Vincentelli and Grant Martin, "Platform-Based Design and Software Design Methodology for Embedded Systems", *IEEE Design and Test of Computers*, Volume 18, Number 6, November-December 2001, pp. 23-33.

[Sarmadi02] S. B. Sarmadi, S. G. Miremadi, G. Asadi, A. R. Ejlali: “Fast prototyping with Co-operation of Simulation and Emulation”, in *Proceedings of FPL 2002*.

[Schumacher05] Schumacher, P.; Mattavelli, M.; Chirila-Rus, A. and Turney, R. “A software/hardware platform for rapid prototyping of video and multimedia designs”. In *Proceedings of the 5th International Workshop System-on-Chip for Real-Time Applications*. IEEE, 2005. pp.30-33; (20-24 July 2005; Banff, Canada.)

[Shirazi03] Shirazi, N., Ballagh, J., “Put Hardware in the Loop with Xilinx System Generator for DSP”, Xcell Journal, Issue 47, May 2003.

[Sima00] Sima, M., S. Vassiliadis, S. Cotofana, J.T.J. van Eijndhoven, and K. Vissers, "A Taxonomy of Custom Computing Machines," in *PROGRESS Workshop on Emheclcled Systems*, Utrecht, The Netherlands, 2000, pp. 87-93.

[Singh03] V. Singh, A. Root, E. Hemphill, N. Shirazi, J. Hwang. “Accelerating Bit Error Rate Testing Using a System Level Design Tool”. FCCM 2003: 62-68.

[Siripokarpirom04] R. Siripokarpirom and F. Mayer-Lindenberg, “Hardware-Assisted Simulation and Evaluation of IP Cores Using FPGA-based Rapid Prototyping Boards”, *International Workshop on Rapid System Prototyping*, Geneva, Switzerland, June 2004.

[Siripokarpirom06] R. Siripokarpirom, “Platform Development for Run-Time Reconfigurable Co-Emulation”, in *Proceedings of 17th IEEE International Workshop on Rapid System Prototyping* , Chania, Crete, June 14-16, 2006.

[Slaac] http://splish.ee.byu.edu/lab/jhdl/slaac1/index.html

[Slade03] A. Slade and B. Nelson. “Reconfigurable Computing Application Frameworks” . In Kenneth L. Pocek and Jeffrey M. Arnold, editors, *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines* (FCCM ’03). IEEE Computer Society, IEEE Computer Society Press, April 2003.

[Talavera03] G. Talavera “Hardware Software debugging techniques for Reconfigurable Systems on Chip”. M.S. Thesis, Universitat Autonoma de Barcelona, Sep. 2003.

[Tessier01] R. Tessier and W. Burleson, "Reconfigurable Computing and Digital Signal Processing: A Survey," J. VLSI Signal Processing, pp. 7-27, vol. 28, no. 3, May 2001.

[Tombs04] J. Tombs, M. Aguirre Echanóve, F. Muñoz, V. Baena, A. Torralba, A. Fernandez-León, F. Tortosa: “The Implementation of a FPGA Hardware Debugger System with Minimal System Overhead”. FPL 2004: 1062-1066

[Touhafi96] A. Touhafi, W. Brissninck, E.F. Dirkx: The Implementation of a Field Programmable Logic Based Co-Processor for Acceleration of Discrete Event Simulators. *Proc. 6th International Workshop on Field-Programmable Logic and Applications* (FPL‘96), pp. 415-424, Springer Verlag 1996

[Tripp02] J. L. Tripp, P. A. Jackson, and B. L. Hutchings. Sea Cucumber: a synthesizing compiler for FPGAs. In M. Glesner, P. Zipf, and M. Renovell, editors, Field-Programmable Logic and Applications, volume 2438 of Lecture Notes in Computer Science, pages 875–885, Montpellier, France, September 2002. Springer-Verlag.

[Turner99] R. Turner, "System-level verification -- a comparison of approaches," in *Proc. 10th International Workshop on Rapid System Prototyping* (RSP ’99), pp. 154-159, Clearwater, Fla, USA, June 1999.

[Valderas04] M. G. Valderas, Eduardo de la Torre, F. Ariza, Teresa Riesgo: “Hardware and Software Debugging of FPGA Based Microprocessor Systems Through Debug Logic Insertion”. FPL 2004: 1057-1061

[VHDL98] IEEE Standard VHDL Language Reference Manual, IEEE, Inc., NY, March 1988

[Vuillemin96] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard, "Programmable active memories: Reconfigurable systems come of age", IEEE Transactions on VLSI Systems, vol. 4, no. 1, pp. 56-69, 1996.

[Waterson01] Waterson, Mark F. “The hardware subroutine approach to developing custom co-processors”. M.S. thesis, University of Hawai'i at Mānoa, May 2001.

[Wildcard] http://splish.ee.byu.edu/lab/wildcard/index.html

[Wirthlin01] M. J. Wirthlin, B. L. Hutchings and C. Worth, “Synthesizing RTL Hardware from Java Byte Codes”, in Field Programmable Logic and Applications, G. Brebner and R. Woods (Eds), pp. 123 – 132, Belfast, Northern Ireland, UK, August 2001.

[Wisniewski01] R. Wisniewski, A. Bukowiec, M. Wegrzyn, “Benefits of Hardware Accelerated Simulation”, *International Workshop on Discrete-Event System Design*, (DESDes'01), June, 2001; Przytok, Poland

[Wheeler01] T. Wheeler, "Improving design observability and controllability for circuit debugging in FPGAs using design-level scan techniques," Master’s thesis, Brigham Young University, Provo, UT, 2001.

[Wheeler01b] T. Wheeler, P. Graham, B. Nelson, and B. Hutchings, ”Using design-level scan to improve FPGA design observability and controllability for functional verification”, in *Proceedings of the Eleventh International Workshop on Field Programmable Logic and Applications*, pp. TBA, Belfast, Northern Ireland, August 2001

[Xilinx00] Xilinx Corp., The MathWorks and Xilinx Strategic Alliance, http://www.xilinx.com/ipcenter/dsp/mathworks_xilinx_presentation.pdf

[XilinxSG] Xilinx Corp., Xilinx System Generator for DSP, Version 6.3i, http://www.xilinx.com

[Xilinx00b] Xilinx, San Jose CA. ChipScope software and ILA Cores User Manual, v. 1.1. June 2000

[Xilinx00b] UNIVERSAL RECONFIGURABLE SYSTEM AND METHOD FOR REMOTE READING OF COUNTERS OF EQUIPMENT COMPRISING VISUAL INDICATORS