HETEROGENEOUS HARDWARE-SOFTWARE PROTOTYPING SYSTEM FOR PC-CONTROLLED FPGA-BASED DESIGNS

Ernest Jamro, Kazimierz Wiatr

AGH University of Science and Technology
al. Mickiewicza 30, 30-059 Kraków, Poland
(jamro,wiatr)@agh.edu.pl

Abstract. This paper describes the Advanced Programmable System Interface (APSI), dedicated for an FPGA-based board controlled by a PC. The APSI includes: the dedicated script language interpreter to efficiently communicate with a FPGA-based board; heterogeneous hardware-software co-simulation to simulate either PC or hardware (FPGA-based board) sides; and internal logic state analyzer. The whole APSI system has been design by the authors and significantly seeds up development cycle for the FPGA-based designs. The proposed system contains several novel ideas, e.g. the concept hardware-software co-simulation, internal logic state analyzer with data compression, clock enable and VHDL-based interface.

Keywords: hardware-software co-design, design systems, simulators, prototyping, testability, modelling.

1. INTRODUCTION

A great number of prototyping systems e.g. [1, 2] and tools for hardware-software designs [3, 4] have been designed. These systems however cannot be used as a complete system which include: a PC communication port, dedicated easy-to-use interface, heterogeneous hardware-software co-simulation, modular design and internal logic state analyzer.

The APSI employs new script command language which makes data transfer and other basic communication with FPGA-based board much easier. For example a single instruction is employed to transfer a file to a specified address range on the FPGA-based board.

The APSI uses modular hardware design. For example, for an FPGA board connected with the PC by the Parallel Port (PP), a special module (bridge) is used to transmit PC-controlled transfers to the On-Chip Peripheral Bus (OPB) [5] or the Wishbone bus [6], two separate versions are supported. The bridge module can be straightforward instantiated in a system employing the Xilinx Embedded Development Kit (EDK) which main feature is fast and easy core connection.

The APSI includes heterogeneous hardware-software co-simulation, which allows the PC activity to be smoothly simulated. The user can easily switch between hardware-execution mode, when the script interpreter communicates with the FPGA-based board, and hardware-simulation mode, when the script interpreter communicates with a VHDL simulator. Consequently the user need not write long stimulus to simulate the PC-side activity. This is a very important feature as the PC often plays a key role in the whole system.

Similar systems have been built but neither of them is a complete system. Most systems concentrate on hardware-software co-simulation [2, 7, 8] but they assume that hardware and software is on the same platform. The presented system assumes that some procedures are processed on the PC and the result of the PC-side activity is generated directly in the PC (in the original platform). Other procedures are processed on the FPGA-based board and therefore hardware description language, e.g. VHDL simulator is used. The key issue of the proposed system is the communication between the script interpreter and the VHDL simulator.

The simulation is a very important part of design process. Nevertheless often the simulation process takes too much time, the hardware model is not available or not complete, timing conditions are not met (or even not simulated) or great many other unpredictable effects occur, which cause that the real system does not function properly. In this case a logic state analyzer is often the only solution to trace the error. An external one is often the only option e.g. was used in [1]. The external logic state analyzer has several disadvantages over the internal one. It can trace only external signals, consequently the internal signals are often probed (driven the a external pin) and this caused that only a limited number (e.g. only 10-20 [1]) of signals can be probed simultaneously. Consequently to view different signals, the

incremental probe routing had to be repeated and this was time consuming [1].
Xilinx Inc. also provides the internal logic state analyzer denoted as ChipScope [9]. ChipScope is a commercial tool which uses its own interface and communication ports. This experiences several drawbacks: the communication port may not be available in the tested board and the capture moment cannot be easily synchronized with the system activity, as it is the case for the proposed Logic Analyser for Reconfigurable Computing Systems (LA_RCS). The LA_RCS has several additional advantages over the ChipScope especially for the system prototyping, e.g. uses a VHDL simulator to view the captured signals and therefore can be easily integrated with the simulator, incorporates Run-Length Coding (RLC) capture data compression, which significantly increase the number of recorded samples (the internal memory size is very limited), uses advanced clock enable logic, which allows to capture only selected samples, e.g. active bus cycles, or bus cycles that address a specified device.

2. THE APSI INTERFACE

The main part of the described system is the script interpreter: the program *apsi.exe* which acts according to a script command file. The APSI system and its documentation are freely available at [10], therefore hereby only a few script commands will be enumerated, in order to illustrate capabilities of the APSI:

config *file_name*- configure the FPGA with the configuration file *file_name*.
readblock *file_name adr_start adr_stop*- read block from the address *adr_start* to *adr_stop.* The read data are written to the file *file_name.*
readbyte *address* – read a single byte from the address *address.* The read value updates a status registers and is displayed in the program console.
sleep *integer* – suspend the program execution for *integer* milliseconds.
run *command* – run the DOS command *command*, e.g. execute another script file when the command is: *apsi.exe script_file*.
goto *label* – go to *label* (change program flow).
loop *integer label* – go to *label (integer-1)* times.
waitbit0 *bit_mask address* – wait until the data read from *address* satisfies the *bit_mask* condition.
gobit1 *bit_mask label* – go to *label* if the status register satisfies the *bit_mask* condition.
go> *integer label* – go to *label* when the status register is greater than *integer.*
stat+= *integer* – add *integer* to the status register.
stat=>m *integer* – write the status register to the internal program memory at address *integer.*
filecomp *file1 file2* – compare the contents of two files.

The script commands allow advance communication between the PC and FPGA-based boards. It is relatively easy to configure FPGA and write/read internal/external memory. Nevertheless the program *apsi.exe* allows to execute much more sophisticated commands e.g. to control a loop, conditional branch or postpone program activity (*sleep*). One of the most important commands is *waitbit0* (*waitbit1*) which reads the memory until the bit-mask condition is met. This command can be used e.g. to wait until a receive / transmit control register satisfies a specified bit condition (the write buffer is full), etc. Another very useful command is *filecomp* which compares two files and can be used e.g. to check if transmitted and received data are the same, e.g. to check whether the transfer between PC and FPGA-based board is correct.
It should be noted that the proposed script language has limited resources in comparison to e.g. C++ language. Therefore some designs may require a more sophisticated language to be used. Fortunately, the program *apsi.exe* was written in C++ and therefore new-sophisticated functions can be added directly in the C++. The C++ source code of the program a*psi.exe* consists of two classes: the first one, denoted as *LowLevel*, communicates with the FPAG-based board using very basic commands: *readbyte, readblock*, etc. The second class reads the script commands and translates them into the basic commands used in the class *LowLevel*. Consequently a user can easily refer to the *LowLewel* functions to write own sophisticated commands directly in C++.

2.1. Modular Design

Complex designs require a bus standard to be established in order to easily connect separate modules. Two different bus standards are available in the APSI: Wishbone [11] and On-chip Peripheral Bus (OPB) [12]. The OPB is preferred as designed modules can be directly added to the Xilinx Embedded Development Kit (EDK) system, which was developed to speed-up modular designs. Another advantage of OPB and EDK designs is the soft-processor MicroBlaze (provided by Xilinx Inc.) and PowerPC processor (PowerPC hard-macro is incorporated only in the Xilinx Virtex II Pro) which are provided together with the EDK and are OPB-compatible.

The program *apsi.exe* communicates with a hardware module incorporated inside the FPGA. This module, denoted as *opb_epp* for the OPB or *epp* for the Wishbone bus, is a bridge between the Parallel Port (PP) mode EPP and OPB (Wishbone). The module *opb_epp (epp)* is a master device which makes core connection much easier and does not requires a microprocessor to master data transfers. Besides the module *opb_epp* can be used instead of

the microprocessor. For example, the UART module *opb_uartlite* can be tested by *opb_epp* instead of the microprocessor. This kind of test may give better result as commands are written directly in the PC (APSI script file or in the C++ compiler) and every OPB communication is recorded in the PC. The drawback of this method is a relatively slow transfer rate on the Parallel Port (PP) in comparison to the system bus and therefore fast cores cannot be tested in this way. It should be noted that the PP is only an example of the communication port and another interface e.g. UART, USB, PCI can be adopted in a similar way.

3. APSI SIMULATION MODE

The APSI plays a key role in the system therefore has to be taken into account while the whole system is simulated.
The novel heterogeneous simulation system included in the APSI consists of two parts:
1) The APSI script language interpreter (*apsi.exe*) which has been modified for the heterogeneous simulation.
2) The VHDL testbench module: *epp_model* which emulates the PP mode EPP and the APSI system.

Adding the new script command: *vhdlsim* changes the APSI script interpreter mode. After the command *vhdlsim* is executed, the script interpreter does not refer to the hardware any longer, but writes every transfer to a special binary file denoted as *apsi.tst*. This file format consists essentially of tree basic instructions:
1) write a single address byte
2) write a single data byte
3) read a single data byte.

It should be noted that adapting the APSI to incorporate heterogeneous simulation was rather simple, only the above three basic instructions has been modified to write data to file *apsi.tst* rather than to send them to the PP.
After the script interpreter generates the file *apsi.tst*, the next step is to simulate the whole system in a VHDL simulator. Before the simulation is started the VHDL module *epp_model* should be instantiated into the top-level simulation file. The *epp_model* should be instantiated at the place where the real PP is connected. This is illustrated in Fig 1.

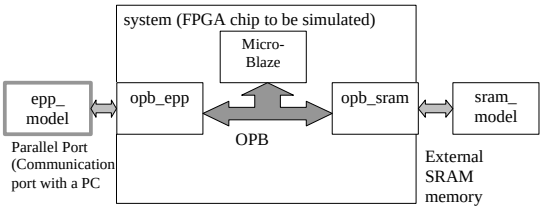


Fig. 1. An example of simulation block diagram

During the VHDL-simulation, all signals can be traced, as it is the case for a standard simulation, the *epp_model* drives the PP signals similarly as if the real PP was connected to the system. Besides during simulation, the *epp_model* writes the data read from the PP to the file *apsi.out*.
The simulation results can be analyzed directly by tracing simulation signals or by *apsi.exe* which should be re-executed. This time, however, *apsi.exe* reads the file *apsi.out* and behaves as data incorporated inside this file are read directly from the PP. Consequently, each time command: *read data* is executed from the script file, *apsi.exe* reads data from the file *apsi.out* rather than from the PP. Summing up, data read from the file *apsi.out* are treated as the real PP data, and the data are e.g. displayed on the monitor for the command *readbyte* or written to the separate file *file_name* for the command: *readblock file_name*.
The whole heterogeneous simulation process is summarized in Fig. 2.

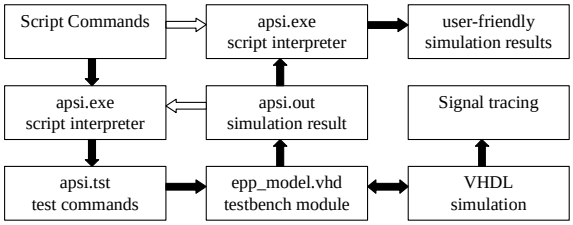


Fig. 2. Heterogeneous simulation scheme

Example 1
Below a simple project example is given. This example tests communication between the PC and on-board SRAM memory (e.g. for the system given in Fig. 1), i.e. compares transmitted (file *test.hex)* and received data (file *read.hex*). The following APSI script is proposed:

Listing 2. An example of a simple test procedure

vhdlsim // the vhdl-simulation mode
writeblock test.hex 1 400 //transfer the file test.hex to address: 1 to 400
readblock read.hex 1 400 // read data from address: 1 to 400 and write them to file read.hex
filecomp read.hex test.hex // compare the files contents

It should be noted that if the command *vhdlsim* is removed, *apsi.exe* will communicated with the PP and the same procedure will be tested for real. A similar simulation without the proposed APSI system would take hours to write the stimulus, trace and compare the transmitted and received data. In the case when large amount of data are transmitted, the standard simulation method cannot be practically employed.

Conversely, functions *writeblock, readblock, filecomp* might be written directly in a VHDL testbench file. This would however require these commands to be implemented inside *epp_model* (in VHDL). Furthermore, every change made in the APSI script file would require a similar change to be made in the testbench file. It should be noted that the above functions are rather simple and if the more sophisticated APSI commands are taken into account the VHDL-testbench approach could not be adopted. Furthermore, the APSI can simulate sophisticated PC programs running directly on the PC, which speed-ups the simulation process and does not suffer from compatibility problem. The latest argument is very important as even if the C-language simulator (debugger) is included directly to the VHDL simulation program, C-language compilation may vary from the original compilation or the C source code may not be available.

4. INTERNAL LOGIC STATE ANALYZER (LA_RCS)

4.1. Hardware Interface

The integral part of the APSI system is the Logic Analyzer for Reconfigurable Computing Systems (LA_RCS). The LA_RCS is an internal logic state analyzer which at first captures the signals into FPGA internal BlockRAMs (BRAMs) and then off-line transfers them to the PC where they can further be analyzed.
The LA_RCS can be used as a stand-alone module, nevertheless the script interpreter supports commands which controls the LA_RCS. The LA_RCS can be activated (or even directly triggered) from the script and this allows for better processes synchronization, e.g. the LA_RCS is activated just before an observed data transfer (or other processes executed or triggered from the script). In this case previous data transfers wont trigger the LA_RCS. Furthermore, a single key-press will activate the whole system. For example it is a common rule to use three separate programs to configure the FPGA device, activate a logic state analyzer and to execute data transfers (or other commands). This is however time consuming and may not be acceptable in a real-time system. Another advantage of the described system is that the LA_RCS can be used several times while a single script file is executed. For example, for a system which checks communication between the PC and the FPGA-based board i.e. transmits and receives the same data, at first the LA_RCS captures the transmission process, then the captured data are read (written to a file on the PC hard disk) and then the LA_RCS again captures the receive process and again the captured data are read. The only constrain of the system is the additional time-slot required to transmit the captured data from the LA_RCS to the PC.
The full description of the LA_RCS is given in [10], here only the most significant features which are specific for this LA_RCS are enumerated. The LA_RCS is a VHDL module which should be instantiated in a design in the same way as other modules (the LA_RCS is a module available in the Xilinx EDK). Also the traced signals have to be defined during design cycle.
The LA_RCS includes two separate interfaces: data-capture and control. The control interface is used to set the trigger condition, activate the LA_RCS and to read the captured data. The control interface is OPB or Wishbone compatible and should be connected to *opb_epp* (OPB), *epp* (Wishbone) or another module for communication with the PC.

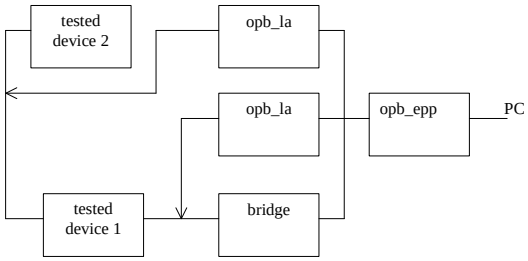


Fig 3. An example of a prototyping system with two LA_RCS

An example of a system with two independent LA_RCS is given in Fig. 3. It should be noted however that two (or more) independent LA_RCSs should be used when they are used simultaneously and independently. Otherwise a bus multiplexer (only one bus is traced at the time) or a LA_RCS with greater capture data width (two buses are traced synchronously) should be employed in order to save the area.

4.2. Run-Length Coding (RLC) and Clock Enable Logic (CED)

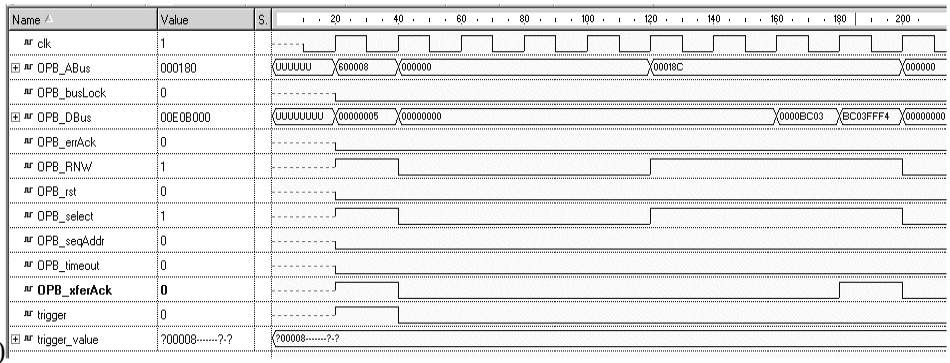
The LA_RCS has several distinctive features: Run-Length Coding (RLC) compression of the captured data and advanced Clock Enable logic for captured Data (CED). These features are principal as the size of the build-in BlockRAM is very limited. The RLC decrements the number of traced signals by 1 bit – the Most Significant Bit (MSB) is used to indicate whether the signal value or the signal count is coded.
For example the signal sequence: *1, 2, 2, 3, 3, 3, 4, 4, 4, 4* is encoded as: *01, 02, 80, 03, 81, 04, 82 (hex).* The RLC is a very efficient compression method for data captured by the logic analyzer as signals usually do not change on every clock cycles.

For example, in the case when the RLC is not implemented, tracing the UART module requires a very large number of samples as UART signals change infrequently. Alternatively the sampling frequency might be decreased, however in this case signal glitches may not be captured.
The RLC compresses the LA_RCS idle states very efficiently; the same state can be captured millions of times and this causes that the idle state may dominate the waveform, and therefore the user cannot efficiently watch the captures signals. Consequently in the LA_RCS, the user can adjust the maximum number of times the same state is repeated and therefore the waveform reading is much easier. For example watching UART activity might be difficult as a signal change is observed infrequently (e.g. every 1000 clocks). Scaling the watching time might cause that the signal glitches are invisible.

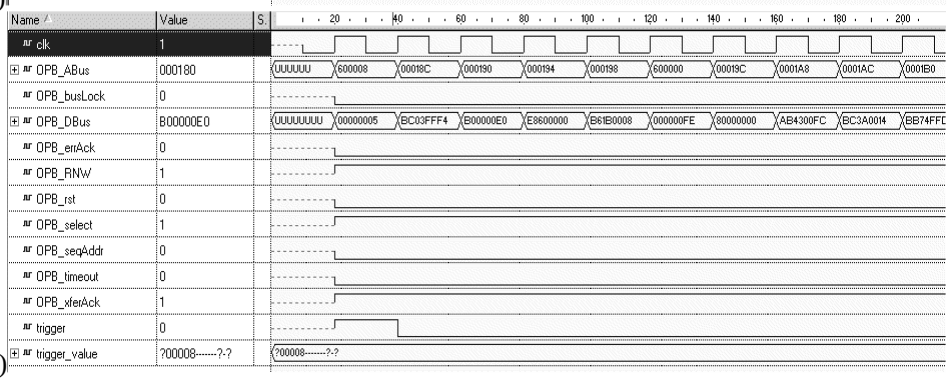
The LA_RCS incorporates also advanced clock enable logic (CED) for which the signals are captured only on selected clock edges. The CED logic is similar as for the standard trigger logic, the input signal is compared with a defined (by a control interface write) pattern and only for the match condition the input signals are sampled. This allows e.g. that only active bus cycles, or bus transactions that refer to a define address space (device) are captured. The RLC and CED can be combined and this forms a very powerful tool. The CED logic not only increases virtually the memory size but also limits the number of displayed samples. Consequently the user is not distracted by insignificant samples, e.g. need not search for samples which address only a specific device. The latest reason is essential and in the authors' opinion the CED-like logic should be adopted also for other (also external) logic analyzers and even VHDL simulators. The functionality of the CED logic is illustrated in Fig. 4.

Fig. 4 shows OPB cycles for the MicroBlaze software debugging - the UART is the debugging interface through which the microprocessor is controlled. Fig 4a shows a standard simulator and logic analyzer view for which the CED logic is not active. Consequently only a single bus transfer can be viewed at a time. Fig 4b shows only active OPB cycles – the LA_RCS captures data only when OPB_xferAck signal is active. This figure provides much more information than Fig 4a and the designer can trace the microprocessor state much easier. Fig 4c shows only active OPB cycles that refer to the UART. This figure allows the designer to concentrate only on the UART and this simplifies significantly the design verification.

a)



b)



c)

Fig. 2. The LA_RCS results for different Clock Enable Logic: a) no clock enable logic, b) capture only active OPB cycles, c) capture only UART data transfers

4.3. Software Interface

The LA_RCS does not incorporate its own GUI. The APSI script interpreter is only used to control the hardware and to transfer the captured data from the LA_RCS to the PC. In order to watch the captured signals a special VHDL module denoted as *la_view* was designed. This module reads the captured data from the hard disk and displays them using a standard VHDL simulator. This gives additional possibilities:

1) The captured signals can be further processed directly in the VHDL simulator.
2) The captured signals can be used as stimulus for simulation.
3) The simulation and captured signals can be automatically compared and differences quickly detected.

The above features are significant and may form a very powerful tool. For example, the authors often use the LA_RCS as a stimulus to check whether the real signals captured by the LA_RCS causes a designed module to work correctly during simulation.

5. CONCLUSIONS

This paper presents a complete prototyping system which includes PC interface, its simulation model and the internal logic state analyzer. The APSI system makes the design cycle much quicker, the simulation and testing time is significantly reduced. Let us give an example of practical use of the APSI system. Only 40% of students' designs functioned properly before the APSI system was introduced. Nowadays about 90% of projects are working properly, furthermore these projects are more complicated and much better tested.

The proposed system is under development, now only the PP interface is supported, however other communication ports can be adopted to the system. The system has been thoroughly tested on the XSV board [13] but other FPGA-based boards can be straightforward adopted to the APSI system.

REFERENCES

[1]. Krupnova, H., Meurou, V., Barnichon, C., Serra, C., Morsi, F., *How Fast Is Rapid FPGA-based Prototyping: Lessons and Challenges from the Digital TV Design Prototypes Project*, Proc. Field-Programmable Logic FPL 2002 Montpellier, France, Sep. 2-4, pp.26-35.

[2]. Gschwind, M., Salapura, V., Maurer, D., *FPGA Prototyping of a RISC Processor Core for Embedded Applications*, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 9, no. 2, April 2001, pp. 241-250.

[3]. Goering, R., *French EDK startup is fluent in co-design* EE Times, Oct. 31, 2003, www.eedesign.com.

[4]. Dolphin Integration, *SUCCESS™ Hardware / Software cosimulation* http://www.dolphin.fr /medal/success/success_overview.html

[5]. IBM, CoreConnect™ *bus architecture*, http://www-3.ibm.com/chips/products/coreconnect/

[6]. OpenCores Org. *WISHBONE SoC Interconnection* http://www.opencores.org/wishbone/

[7]. R. Goering *French EDK startup is fluent in co-design* EE Times, Oct. 31, 2003, www.eedesign.com.

[8]. Dolphin Integration, *SUCCESS™ Hardware / Software cosimulation* http://www.dolphin.fr /medal/success/success_overview.html

[9]. Xilinx Inc. *ChipScope Pro Software and Cores User Manual,v6.*1 August 29 2003.

[10]. Jamro, E. *Advanced Programable System Interface*, http://galaxy.uci.agh.edu.pl/~jamro/apsi

[11]. OpenCores Org. *WISHBONE SoC Interconnection* http://www.opencores.org/wishbone/

[12]. IBM, *CoreConnect™ bus architecture*, http://www-3.ibm.com/chips/products/coreconnect/

[13]. Xess Co. http://www.xess.com/manuals.html