FPGA INTEGRATED LOGIC ANALYZER WITH TESTING AUTOMATION FACILITIES

Laurențiu-Cristian DUCA[1]

În acest articol se prezintă un analizor logic integrat pentru FPGA. Analizorul este dedicat pentru depanarea şi verificarea în timp real, direct pe placa de dezvoltare a aplicațiilor bazate pe FPGA. Deține facilități pentru testarea automată a aplicatiilor şi pentru simplificarea interfațării cu analizoarele de protocoale de comunicații.

This paper presents an FPGA integrated logic analyzer. The logic analyzer is designed for in-circuit debug and verification of FPGA based applications. It has facilities for testing automation of the FPGA based applications and easy interfacing with third party protocol analyzers.

Keywords: FPGA, integrated logic analyzer, in-circuit debugging, testing automation, co-simulation, Verilog simulator

1. Introduction

In the field of debugging FPGA based applications, cost of traditionally external logic analyzers can be greatly decreased and interfacing to protocol analyzers can be significantly improved.

When using traditional external logic analyzers, the pins of the ASIC chip are connected to a special dedicated socket from where the signals on specific pins can be grabbed in real-time and viewed on the screen of the dedicated external logic analyzer. Usability and scope of logic analyzers are well presented in [1]. In industry, representative products of the external logic analyzers market are the Agilent 16800/16900 Series Logic Analyzer and Tektroniks TLA 5000 Series Logic Analyzer.

In the FPGA field, the intelligence of the external analyzer can be split in two sides. The first side is represented by the FPGA integrated modules which are responsible to real-time data capturing of the monitorized signals. The other part of the logic analyzer is a computer application which graphically displayes the captured data. Xilinx Chipscope and Altera Signal Tap are two of the most representative integrated logic analyzers of the market products. Also is the SUMP logic analyzer which is an open source project and available for download [2] on the Internet.

[1] Assist., Dept. of Computer Science, University "Politehnica" of Bucharest, Romania

In the logic analyzer presented in this paper has been implemented a method for testing automation and easy interfacing with third party protocol analyzers. This logic analyzer represents an original and personal work of the author of this paper.

2. Architecture and functionality

The logic analyzer that will be presented is named openverifla. The project is publicly available for download [3] on the Internet under the terms of the GNU GPL open source license. The accent of the presentation is put on the original features of the logic analyzer and the standard part is just sketched as a main view.

The main architecture of the openverifla logic analyzer is shown in Fig. 1. The logic analyzer has two sides, the FPGA part and the PC one. These communicates via a PC-FPGA interface.

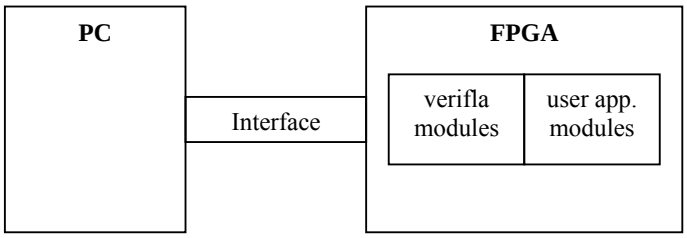

Fig. 1 Main architecture of the logic analyzer

The openverifla FPGA modules are implemented in Verilog HDL. In order to use the logic analyzer, these modules must be implemented in the FPGA chip along with the user application. The modules capture the signal transitions of the monitorized lines and send the data capture to the PC for graphical visualization and future analyze.

The PC part of the application is implemented in the Java language. The Java application receives the captured data and saves it on the disk in a file named *capture.v*. This file is a behavioural Verilog HDL file. An Verilog HDL simulator with a graphical viewer for the signals is necessary in order to simulate *capture.v* and view the captured data.

In the current version, the interface between the FPGA and PC is made via the RS232 standard serial interface. The micro-UART Verilog drivers written by Jeung Joon Lee [4] were adapted in a version usable by openverifla FPGA modules. The *rxtx* library [5] is used to gain access to the PC serial port. It is distributed under the GNU GPL license and supports many operating systems such as Linux, Windows. The library is wrote in the C/C++ language and the Java port is made through the JNI interface.

The original features of this logic analyzer are shown through an illustrative example. In this example, a keyboard driver implementation is verified. The keyboard protocol is presented in Fig. 2. When no event occures, the clock and data lines are idle – being hold as high. When a key is pressed or released, the key-code is sent on the data line, bit by bit. It is preceded by the start bit and followed by the parity and stop bits. This is done synchronously with the clock signal. When the CLOCK signal is low (0), DATA have a stable value. When the CLOCK signal is high (1), on the DATA line it may be a signal transition. The key-codes are different by the ASCII codes. For example, 'a' has 0x1C and its bits will be sent as 0011 1000.

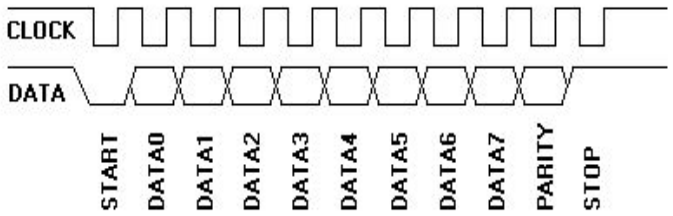

Fig. 2 Keyboard protocol

A keyboard driver implementation is shown in Table 1. The module top_of_verifla is instantiated in the KEYBOARD module. This way, it will be implemented in the FPGA chip along with the user application. The signals to be captured are DATA, KBD_CLK_EF and MASTER. When a key is pressed, its keyboard-code bits are sent on the data line, conforming to the keyboard protocol shown above. The implementation of the KEYBOARD module de-serializes the data bits and then places the keyboard-code in the MASTER octet register. The signal transitions are captured on-the-fly by the openverifla modules and then will be sent to the PC, where will be graphically displayed.

Table 1

An implementation of the keyboard driver

```
module KEYBOARD(DATA, KBD_CLK_REAL,RESET,MASTER,
        //top_of_verifla
                                , clk,
                                // Transceiver
                                uart_XMIT_dataH, uart_REC_dataH
);

//top_of_verifla
input clk;
        // Transceiver
```

```
input uart_REC_dataH;
output uart_XMIT_dataH;

// App. specific
input DATA,KBD_CLK_REAL,RESET;

wire KBD_CLK_EF;
output [7:0] MASTER;     // master register for storing keyboard data
reg [7:0] MASTER;
reg [3:0] i;          // initial value needs to be not equal to 0 through 7. set initial to 10.


//assign KBD_CLK_EF=KBD_CLK_REAL;
IBUF b1(KBD_CLK_EF, KBD_CLK_REAL);

always @ (negedge KBD_CLK_EF or posedge RESET)
begin
        if(RESET)
        begin
                i=10;
                MASTER=0;
        end
        else begin
                if ((i >= 0) && (i <= 7))
                // data bit.
                begin
                        MASTER = {DATA, MASTER[7:1]};
                        i = i + 1;
                end
                else if ((i == 8) || (i == 9))
                // parity bit or stop bit.
                begin
                        i = i + 1;
                end
                else // start bit
                begin
                        i = 0;
                end
        end
end

// VeriFLA
top_of_verifla verifla (clk, !RESET, 0,
                                        {22'h000000, DATA, KBD_CLK_EF, MASTER},
                                        // Transceiver
                                        uart_XMIT_dataH, uart_REC_dataH
                                        );

endmodule
```

After the application is implemented in the FPGA chip, the Java application is run on the PC. This way, the openverifla modules are instructed to start a new capture and after the capture is finished, to send the capture to the PC.

Now, these modules wait for signal events on the monitorized lines. If a key is pressed - for example 'a', the keyboard module implementation does its job and the openverifla modules makes the capture on-the-fly and sends it to the PC.

The Java application gets the capture and builds the *capture.v* Verilog file. After this, the *capture.v* can be added and simulated in a Xilinx ISE normal project. The result is shown in Fig. 3.

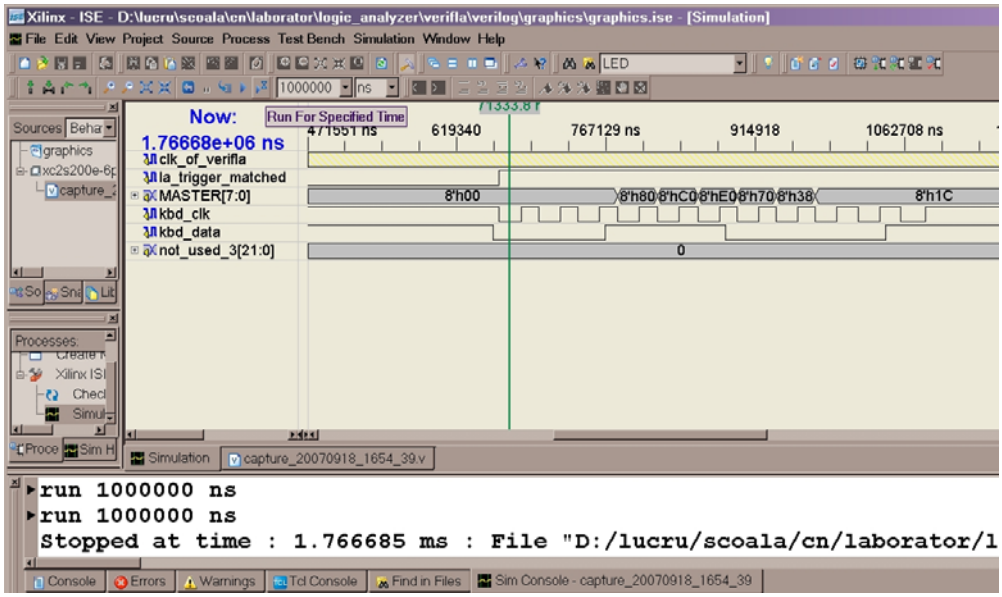

Fig. 3 Simulation of *capture.v*

The development-board clock is used as the openverifla modules clock and has a freqvency of 50 MHz. The frequency of the keyboard clock is about few Khz. So, in the Xilinx simulation, *run 1000000 ns* commands were necessary, to reach the *$stop* instruction of the *capture.v*.

3. Testing automation

The *capture.v* file is built such that it contains signal transitions for all monitorized lines. Having both the input stimulus and output signals captured, testing automation can be done.

In the above test case we wanted to verify if the driver receives corectly a key that was pressed. Consider more that this key is pressed immediately after the

solution was implemented on the FPGA and the reset button associated to the application was pressed.

The main structure of the *capture.v* file is shown in Table 2. The *kbd_data*, *kbd_clk* lines and the *MASTER* register were monitorized. The clock of the openverifla modules is *clk_of_verifla*.

Table 2

Code structure of the *capture.v* file

```
`timescale 1ns / 10ps

module capture_20070921_1522_28(clk_of_verifla, la_trigger_matched,
MASTER, kbd_clk, kbd_data, not_used_3);

output clk_of_verifla;
output la_trigger_matched;
output [7:0] MASTER;
output kbd_clk;
output kbd_data;
output [21:0] not_used_3;
reg [7:0] MASTER;
reg kbd_clk;
reg kbd_data;
reg [21:0] not_used_3;
reg la_trigger_matched;
reg clk_of_verifla;

parameter PERIOD = 10;
initial // Clock process for clk_of_verifla
begin
forever
begin
clk_of_verifla = 1'b0;
#(5); clk_of_verifla = 1'b1;
#(5);
end
end

initial begin
#(5);
la_trigger_matched = 0;
{not_used_3,kbd_data,kbd_clk,MASTER}= 32'b00000000000000000000001100000000;
#10;
// ------------- Current Time: 10*(1ns)
#655350;
{not_used_3,kbd_data,kbd_clk,MASTER}= 32'b00000000000000000000000100000000;
#10;
// ------------- Current Time: 655370*(1ns)
#5750;
```

```
{not_used_3,kbd_data,kbd_clk,MASTER}= 32'b00000000000000000000000000000000;
la_trigger_matched = 1;
#10;
// ------------- Current Time: 661130*(1ns)
#22030;
{not_used_3,kbd_data,kbd_clk,MASTER}= 32'b00000000000000000000000100000000;
#10;
... // here code similar to the above lines was stripped out.
$stop;
end
endmodule
```

By using the co-simulation method, *capture.v* can be simulated along with a specification only Verilog source that is supposed to be correctly or is already simulation verified. This may be provided by a third party protocol analyzer. Co-simulation can be done by instantiating both modules in a single source and then simulating this one. For the keyboard driver example, the source for co-simulation is shown in Table 3. The inputs of the supposed correct model are driven from the captured ones in the real test – which are now taken from the *capture.v*. These are the *kbd_clk* and *kbd_data* lines. All the co-simulation time, the *MASTER* and *km_MASTER* registers should coincide. If they does not, there is a difference of the two common simulated solutions. So one of the two is affected by a user design logical error. The number of errors are kept in the *errors_nr* variable. With the presence of this variable in simulation, it can be located the moment in time where an error appeared and the appropriate state of the user application.

Table 3

The co-simulation Verilog source for the keyboard driver

```
`timescale 1ns / 1ps
module kbd_drv_aut_test(clk_of_verifla, la_trigger_matched,
MASTER, kbd_clk, kbd_data, not_used_3,
errors_nr);

output errors_nr;
// same as kbd_capt
output clk_of_verifla;
output la_trigger_matched;
output [7:0] MASTER;
output kbd_clk;
output kbd_data;
output [21:0] not_used_3;

integer errors_nr;

// kbd_capt
wire clk_of_verifla;
```

```
wire la_trigger_matched;
wire [7:0] MASTER;
wire kbd_clk;
wire kbd_data;
wire [21:0] not_used_3;
capture_20070921_1522_28 kbd_capt (clk_of_verifla, la_trigger_matched,
   MASTER, kbd_clk, kbd_data, not_used_3);

// kbd_model
reg km_RESET;
wire [7:0] km_MASTER;
kbd_drv_behavioural kbd_model (kbd_data, kbd_clk, km_RESET, km_MASTER);

initial begin
   errors_nr=0;
   km_RESET=0;
   #1;
   km_RESET=1;
   #1;
   km_RESET=0;
end

always begin
   if(km_MASTER != MASTER)
   begin
      //$display("Error at time=%dns km_MASTER=%h, MASTER=%h",
      // $time, km_MASTER, MASTER);
      errors_nr = errors_nr + 1;
      //$stop;
   end
   #1;
end
endmodule
```

The co-simulation is shown in Fig. 4. The *errors_nr* variable is 0 at the end of the simulation. So, the co-simulation shows an identical behaviour for both solutions for the present test case.

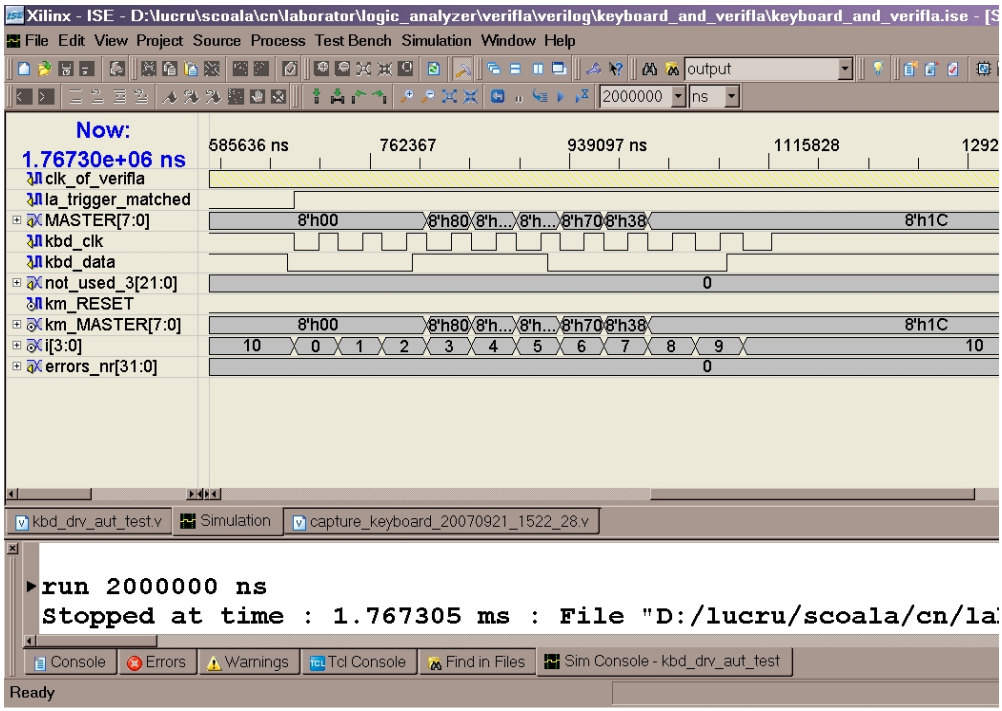

Fig. 4. Co-simulation for the keyboard driver

4. Conclusions

In this paper was presented an FPGA integrated logic analyzer and the accent put on its original features.

The logic analyzer FPGA modules can easily be attached to the user applications that are written in Verilog. Being implemented in Java, the PC part is platform independent.

The main ideea that make nice features to appear as possible, was to save the raw data capture in a behavioural Verilog file named *capture.v*. This file can be simulated in any Verilog simulator and signal transitions along with the time stamps are available for the user.

By using the co-simulation method, *capture.v* can be simulated along with a specification only Verilog source that is supposed to be correctly or is already simulation verified.

Interfacing with third party protocol analyzer programs can be done, by using the standard behavioural Verilog format of the *capture.v* file.

REFERENCES

[1] *A.L. Kuan*, A "How To" tutorial on Logic analyzer basics for digital design, http://www.pldesignline.com, 2007
[2] *M. Poppitz*, The SUMP logic analyzer, http://sump.org/projects/analyzer
[3] *L.C. Duca*, The openVeriFLA project, http://www.opencores.org/projects.cgi/web/openverifla
[4] *J.J. Lee*, micro-UART, http://www.cmod.com, 2001
[5] *K. Jarvi et all*, The rxtx library, http://www.rxtx.org