XDebugger: An Elastic Solution to Dynamic Batch In-Circuit Debugging on FPGA

Fulong Chen[1,*], Yunxiang Sun[2], Xuemei Qi[2], Jie Yang[2], Heping Ye[2], Junru Zhu[2] and Qimei Tang[2]

[1]*Department of Computer Science & Technology, Anhui Normal University, Wuhu, Anhui 241002, China*

[2]*Network and Information Security Engineering Research Center, Anhui Normal University, Wuhu, Anhui 241002, China*

Abstract-In digital system design, Intellectual Property (IP) reuse technology reduces the complexity of System on a Chip (SoC) design, and improves its design efficiency. However it also brings some testing or verification difficulties. Aiming at the low efficiency of testing mechanism, the difficulty of real-time signal monitoring and non-reusability of module-level debugging platform for IP design, we propose an elastic solution to dynamic batch in-circuit emulating on Field Programmable Gate Array (FPGA) so as to optimize the testing process. Through the simulation debugging kit working in the computer side, the downloading pathway of stimulus signals and configuration data is created; through the simulation monitoring kit working in the FPGA side, the uploading pathway of simulation data and feedback signals is created; and with Universal Asynchronous Receiver/Transmitter (UART), the stimulus signals, configuration data, simulation data and feedback signals are transmitted between the computer side and the FPGA side. After testing of multiple IP instances, the results show that the method has strong universality and can improve the efficiency of IP verification.

Keywords: IP testing, In-circuit debugging, Elastic design, FPGA.

1. INTRODUCTION

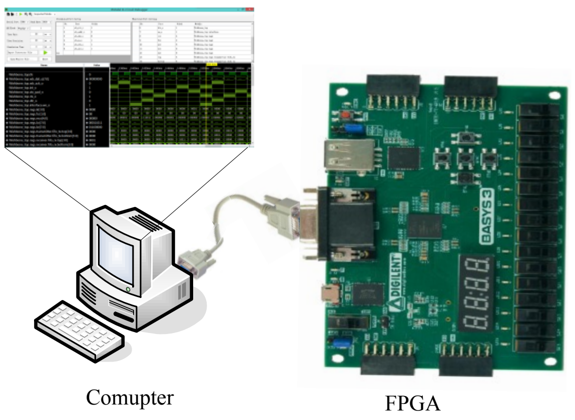
With the continuous innovation of integrated circuit design methods and the continuous improvement of chip manufacturing technologies, the functions of complex digital systems made of integrated circuits are becoming more and more powerful, and the difficulty and complexity in design, especially with regards to testing, are increasingly significant. The integrated circuit design has transformed from Application Specific Integrated Circuit (ASIC) to SoC, and SoC design method based on IP reuse has become the mainstream of the digital system design [1].

1.1. Related Works

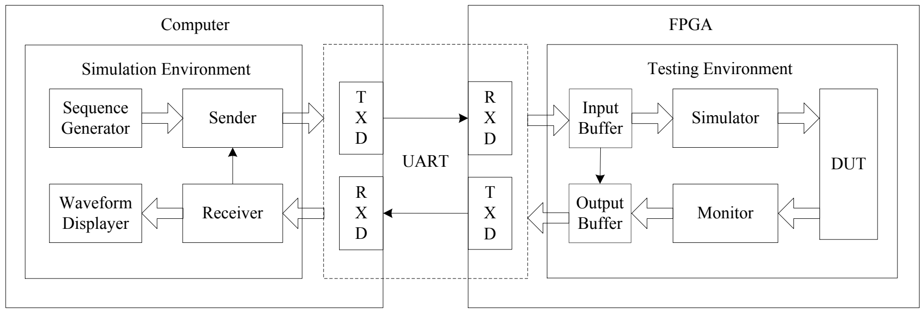
IP reuse technology effectively shortens the design cycle, improves the design efficiency, and reduces the design complexity, and the functions of real chips can be simulated with high-level hardware models of IP cores in FPGA. The FPGA testing not only has higher accuracy, but also the running speed will not be reduced with the increase of design complexity. However, the FPGA testing need the complete support of testing platforms to generate stimulating signals and monitor the port signals. Since the testing is still the most usefull approach to ensure the reliability and availability of systems, the stimulating mechanism and the non-reusability of testing platforms have become the major bottleneck of improving the efficiency of IP testing [2-4].

In recent years, researchers have carried out extensive researches on software simulation and hardware emulating platform of IP testing. UC Berkeley developed a software system named Ptolemy that provided heterogeneous simulation environment [5]. This system can be used in the modeling and simulation of embedded system, and customized IP cores can be added for simulation models [6]. However it is mainly used for modeling and simulation, effective solution to FPGA in-circuit verification is not given to improve the testing efficiency. Chung put forward PROToFLEX [7] which provided a novel multiprocessor emulation approach in which the execution of many processor contexts is interleaved onto a shared emulation engine [8]. The point is that not all systems are composed by processors, especially the semi-customized digital systems. The Research Accelerator for Multiple Processors (RAMP) project, which was co-founded by many colleges and universities in 2005 ISCA conference, developed a testing platform that supports multi-processor design, and its goal is to ramp up the rate of innovation in hardware and software multiprocessor research [9-10]. Like PROToFLEX, the complexity of the RAMP verification platform is higher, and cannot verify a variety of IP cores according to flexible and effective ways. Xilinx Company also provided an in-circuit logic analyzer ChipScope [11] which can implant Integrated Logic Analyzer (ILA) and Integrated Controller (ICON) into FPGA, and the Joint Test Action Group (JTAG) cable will be used to collect and observe signal transformation in the chip. The logic analyzers

*Address correspondence to this author at the Department of Computer Science and Technology, Anhui Normal University, 189 Jiuhua East Rd., Wuhu, Anhui Province 241002, P. R. China; Tel: (+86-553) 5910757; E-mail:long005@mail.ahnu.edu.cn



(a) Platform



(**b**) Structure

Fig. (1). Architecture of XDebugger.

need a special JTAG adapter to work, and have no versatility. Therefore, more universal communication technology and computer with monitoring functions [12] helps to simplify in-circuit debugging and speed the test schedule of IP design.

1.2. Contributions

In this paper, we propose an elastic solution to dynamic batch in-circuit debugging on FPGA. Using general UART without special JTAG, the testing process is optimized. It consists of two parts- one is the FPGA side and another is the computer side, which communicate with UART. According to different modules under testing with simple configuration, we can effectively save time of testing model reconfiguration in this way, and the hardware overhead is small. It also can take full advantage of the early accumulation to accelerate the design process and reduce the risk of product development such as image processing [13], artificial intelligence simulation [14-16], data processing [17], and optimization [18]. Multiple IP testing instances shows that the method can effectively improve the efficiency of IP testing and reduce the working strength of designers.

2. DESIGN SCHEME OF XDEBUGGER

A complete in-circuit emulating platform is mainly composed of a hardware testing environment and a software emulating environment, in which the verification process is controlled by test vectors and debugging commands. XDebugger is shown in Fig. (**1a**), where ***X*** means that we can debug different IP modules including combinational logic, sequential logic and hierarchical logic.

The software emulating environment is a emulation module running in the computer side to produce testing sequences, send testing sequences, receive feedback signals and display waveforms. The hardware testing environment is a testing module working in the FPGA side to stimulate the Device Under Testing (DUT) with stimulus signals and capture the stimulus signals. They ensure the completeness and orderliness of debugging effectively.

Fig. (**1b**) shows the structure of XDebugger. Firstly, in the computer side, the generator produces stimulus signals, and the latter are transmitted to the sequence sender for serial sending. Secondly, in the FPGA side, the stimulator loads stimulus signals from the input buffer, and throws them to the DUT as input signals. Then the monitor captures the monitoring signals and sends them to the output buffer which transmits them to the computer side through serial transmission. Finally, in the computer side, the receiver receives the monitoring signals, and the waveforms with the displayer.

3. IMPLEMENTATIONS OF SOFTWARE EMULATING ENVIRONMENT

In the traditional ASIC chip design process, software simulators, e.g. modelsim, Xilinx ISim, etc., are effective means to verify design results. However, with the increasing scale of SoC design, the time spent on simulation is also exponentially increasing. Meanwhile, although software simulators can debug DUTs well, it cannot truly reflect the working hardware condition of and cannot be connected to real external devices. FPGA is a very good solution to this problem. Utilizing the parallel processing ability of FPGA, XDebugger can better debug DUTs.

We design a software emulating environment in the computer side. It provides an interface for transmitting the testing data. We can organize stimulus signals and monitor feedback signals in computer side. It automatically generate the Verilog HDL codes of the FPGA side according to the stimulated port settings, monitored port settings and other parameter configurations. The stimulus vectors are read into the sequence generator from the corresponding test vector file. And the monitored port settings are read into the waveform displayer. Then, stimulus vectors are sent to the FPGA side. The latter will start debugging, and send back the feedback vectors to the computer side. Finally the displayer can draw the waveform for each monitored ports according to he received feedback vectors.

3.1. Generator

(1) Timer setting

When the testing module in the FPGA side works, it needs to be triggered by a clock. Therefore we define some parameters which formats are as follows:

#time_precision = Value Unit; representing the time precision which specifies the sampling cycle of monitoring signals.

#time_unit = Value Unit; representing the time unit.

#clock_cycle = Value; representing the clock cycle of the DUT.

Where **Unit** may be **s, ms, us, ns**.

(2) Stimulated port setting

The generator need provide a configuration method so that designer can set which ports will be stimulated. Each stimulated port is defined as the following format:

@PortName,Width;

Where **PortName** gives the name of the stimulated port and **Width** gives its width and it must be the port of the top module. Assuming that we have n stimulated ports, and for the stimulated port $p_i,\ i=1,\ 2,\ ...,\ n$, its width is w_i, we can get the total width

$$w=\sum_{i=1}^{n} w_i \tag{1}$$

(3) Monitored port setting

Finally, the sequence generator also need provide a configuration method so that designer can set which signals will be monitored. Each monitored port is defined as the following format:

[InstanceName[.InstanceName[...]].]PortName,Width;

Where **PortName** gives the name of the monitored port and **Width** gives its width. If **PortName** is not the port of the top module but one port of one instance, **InstanceName** must be explicitly indicated. Assuming that we have N monitored ports, and for the stimulated port $P_i,\ i=1,\ 2,\ ...,\ N$, its width is W_i, we can get the total width

$$W=\sum_{i=1}^{N} W_i \tag{2}$$

(4) Stimulus vector

Stimulus vectors are generated by the sequence generator. Each stimulus vector's format is defined as follows:

#Delay VectorValue;

Where **Delay** is an integer for representing that the stimulus vector is updated with the new value **VectorValue** after **Delay** time units. The sequence generator reads an ASCII file including a series of stimulus vectors in the above format and send them to the sender. Besides the above format, there is another particular format **#Delay Stop;**, which also be encapsulated for **Stop** command representing that the monitoring stops after **Delay** time units.

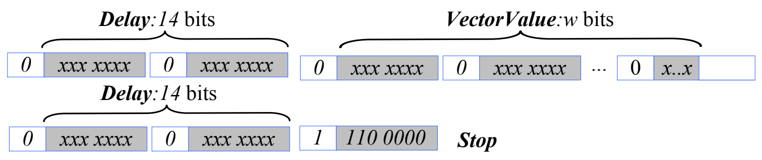
3.2. Sender

The Sender is used to converted stimulus vectors from the generator into serial bits for serial sending with UART. In this process, the sender is not only to ensure that stimulus vectors will not be lost, but also to guarantee the transmission keeping the time sequence on track.

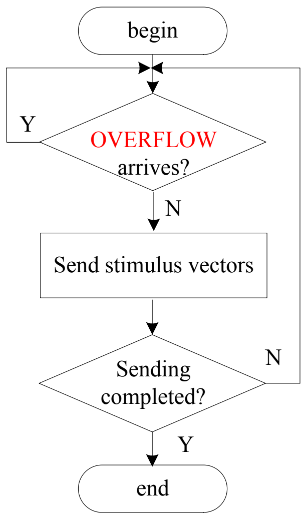
All the stimulus vectors are decomposed into bytes. In each byte, if the most significant bit (MSB) is 0, the other 7 bits will be used to store bits of stimulus vectors. Assuming that in each stimulus vector, **Delay** has 14 bits and **VectorValue** has w bits, they are encoded as shown in Fig. (**2a**). All stimulus vectors are generated by the generator and sent by the sender as shown in Fig. (**2b**).

3.3. Signal Receiver

As shown in Fig. (**3**), the receiver is used to get the response (feedback) vectors which are transferred from UART,



(**a**) Encoding



(**b**) Work flow

Fig. (2). Sender.

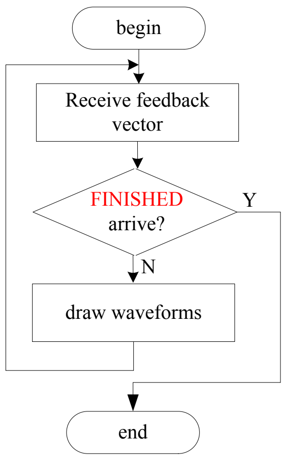


Fig. (3). Work flow of Receiver.

and then store them in the current emulating environment so that the displayer can draw the correspond waveforms and designers can verify their correctness. During debugging process, the receiver may not receive or wait for the response data regularly; therefore it needs to control the time sequence as well as the sender.

3.4. Displayer

Finally, the displayer draws the response vectors in waveform, and designers can conveniently and visually observe the debugging results.

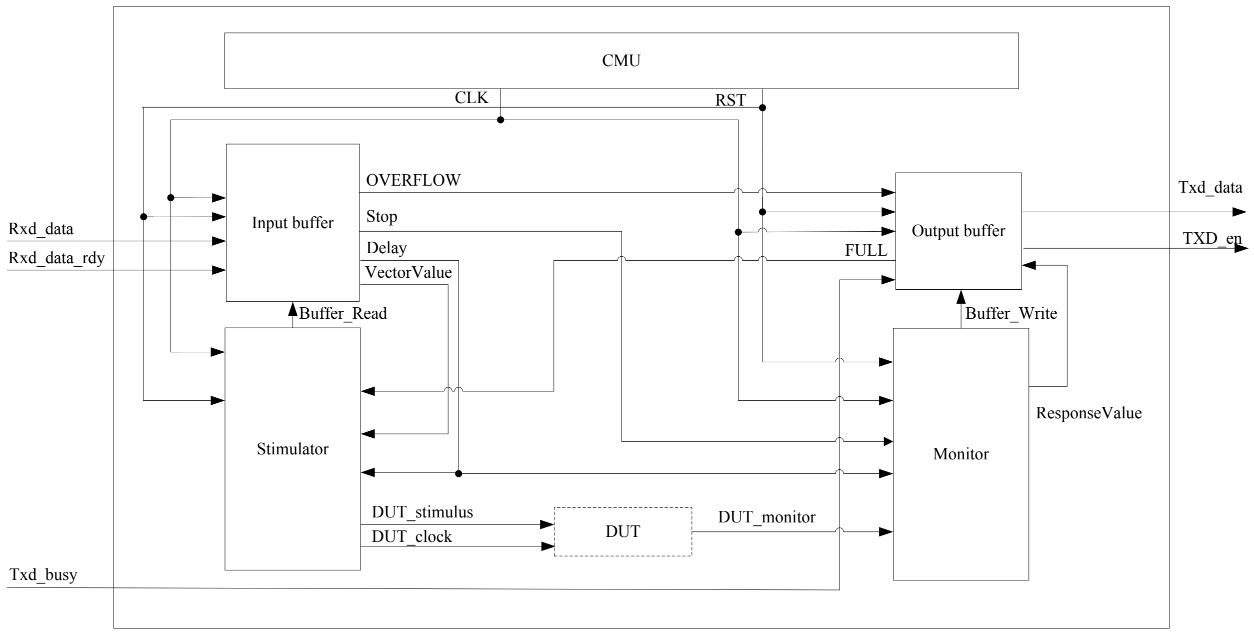


Fig. (4). Testing Environment.

4. IMPLEMENTATIONS OF HARDWARE TESTING ENVIRONMENT

The core work of the entire in-circuit debugging is the FPGA testing. It has two major functions. One is to control the ports of the DUT, so that the stimulus vectors are pushed to the fitting input ports in the given time according to the timing requirements, and the feedback vectors are sampled from the monitored ports. The second is to manage data transmission.

To improve the elasticity of in-circuit debugging, shorten debugging time, and save system resources, we design the testing environment on FPGA. Its structure is shown in Fig. (**4**). It mainly consists of a CMU (Clock Management Unit), an input buffer, an output buffer, a stimulator, a monitor and a DUT. Since the elastic debugging platform will be applied in different designs, it can be put into use after making some appropriate adjustments. It need have a certain configurability so that designers only modify one part of HDL codes, and the other part codes are generated automatically by the generator. For different design, the buffer depth and width are also different. In addition, buffers extremely consume FPGA resources. It will waste too much FPGA resources if the buffer storage is fixed. Therefore, the buffer is elastic and has the expandable feature to meet large or small design.

4.1. Input Buffer

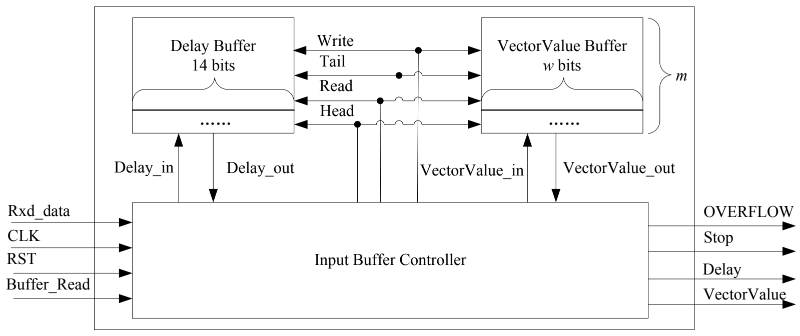
The input buffer is composed of a **Delay** buffer, a **VectorValue** buffer and an input buffer controller as shown in Fig. (**5**). The **Delay** buffer is a table in which each line can store one **Delay** in the format of 14 bits. 14 bits indicate that **Delay** spans $\left[0,\ 2^{14}-1\right]$. In fact, only [0,999] are used due to the time unit **s, ms, us, ns**. The **VectorValue** buffer is also a table in which each line can store one **VectorValue** in the format of *w* bits. Both of them are round-robin queues, which have the maximum lines (=*m*) depending on the initial configuration, and are controlled by the input buffer controller. The input buffer controller mainly completes the following two operations:

(1) Buffer writing

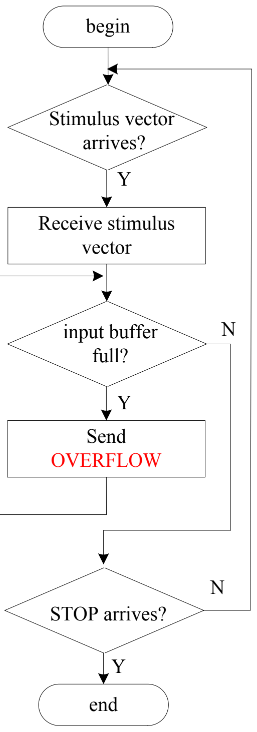
Before the input buffer starts to work and **RST=1**, it sets the initial state for the **Delay** buffer and the **VectorValue** buffer. Then it starts to receive stimulus vectors. Once it receives a byte **Rxd_data**, it will check the MSB of **Rxd_data** and take the appropriate action. If it is a stimulus vector, i.e., **Rxd_data[7]=0**, it will initialize **Delay_width=0** and **VectorValue_width=0** so as to start to receive a full stimulus vector and store it in the **Tail** pointed line of the **Delay** buffer and the **VectorValue** buffer. Otherwise, i.e., **Rxd_data[7]=1**, it will start to receive a particular stimulus vector **Stop** and send it to the stimulator. In that situation, its **Delay** is stored into the **Delay** buffer and a random **VectorValue** is stored into the **VectorValue** buffer. Before the input buffer controller stores the received stimulus vector into the input buffer, it needs to make sure whether the buffer is full. If it is full, it must stop receiving and send a particular signal to the output buffer.

(2) Buffer reading

When the buffer is triggered (**Buffer_Read=1**) by the stimulator, it will read a test vector from the **Head** pointed line of the **Delay** buffer and the **VectorValue** buffer, and send the test vector to the stimulator. Once the last test vector is read, the buffer will generate a special signal **Stop=1** for the monitor so that the latter stops monitoring the response signals.

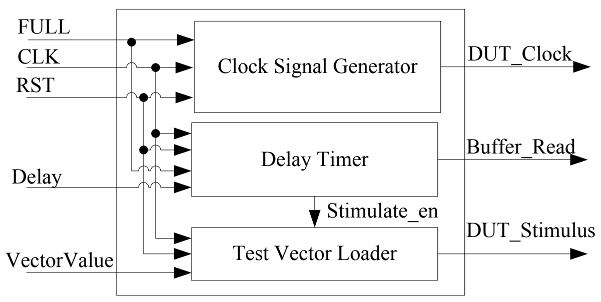


(**a**) Structure



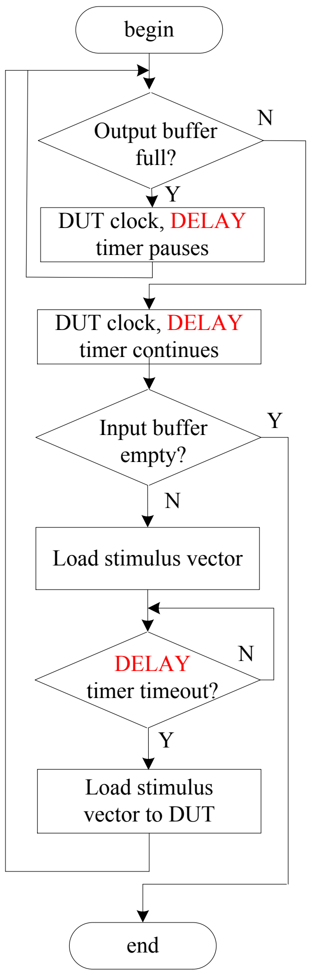
(**b**) Work flow

Fig. (5). Input buffer.



(**a**) Structure

Fig. (**6**). Contd…



(**b**) Work flow

Fig. (6). Stimulator.

4.2. Stimulator

The stimulator provides the DUT with the clock signal and stimulus signals related to **Delay**. As shown in Fig. (**6**), it is composed of the following three modules:

(1) Clock signal generator

It generates the clock signal for the DUT. The frequency $f = \dfrac{1}{ClockCycle \times TimeUnit}$ if **FULL** is not true.

(2) Delay timer

It is a timer which works when **FULL** is false. The timing cycle is $Delay \times TimeUnit$. When it is initialized, it makes **Buffer_Read=0** and **Stimulate_en=0** so as to disable the **Buffer_Read** signal of the input buffer. After that, it makes **Buffer_Read=1**, fetches a test vector from the input buffer. Then, it makes **Buffer_Read=0** again and starts timing. Once the timer expires, it will make **Stimulate_en=1**. This process will continue to be repeated until the input buffer is empty.

(3) Test vector loader

This module is used to assign the signal values of the current test vector to the corresponding DUT ports when **Stimulate_en=1**.

4.3. Monitor

The monitor is a signal detector used to monitor the response signals of the monitored ports of the DUT. As shown in Fig. (**7**), it is composed of the following three modules:

(1) Stop timer

It is a timer. In order to limit the size of the output buffer and prevent buffer overflow, the monitor must stop sampling the response signals at some time point and disable the

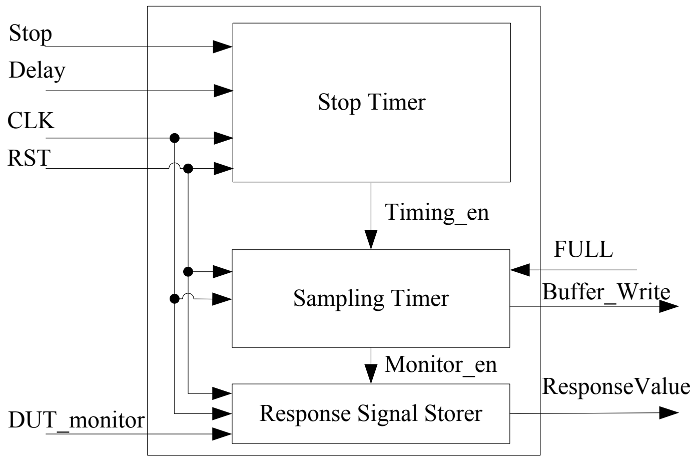


Fig. (7). Monitor.

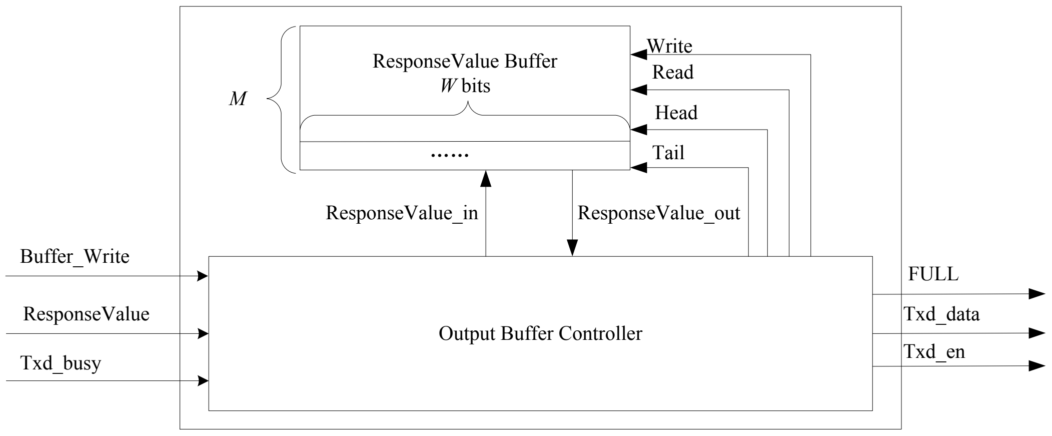


Fig. (8). Output buffer.

sampling timer. In the input buffer, the last test vector is the **Stop**. When The stop timer gets **Stop=0**, it makes **Timing_en=1**. Otherwise, when it gets **Stop=1**, it starts timing. The timing cycle is **Delay**. Once the timing ends, it will make **Timing_en=0**.

(2) Sampling timer

It is also a timer which works when **FULL** is not true. The timing cycle is the sampling cycle **TimePrecision**. When it is initialized, it makes **Buffer_Write=0** and **Monitor_en=0** so as to disable the **Buffer_Write** signal of the output buffer. After that, it makes **Buffer_Write=1**, stores the values of the response signals to the output buffer. Then, it makes **Buffer_Write=0** again and starts timing. Once the timer expires, it will make **Monitor_en=1**. This process will continue to be repeated until the output buffer is full or the monitoring is finished.

(3) Response signal storer

This module is used to sample the signal values of the monitored ports of the DUT and generate the corresponding response values when **Monitor_en** is effective, i. e., **Stimulate_en=1**.

4.4. Output Buffer

The output buffer is composed of a **ResponseValue** buffer and an output buffer controller as shown in Fig. (**8**). The **ResponseValue** buffer is a table in which each line can store one response vector in the format of *W* bits. Similar to the input buffer, the **ResponseValue** buffer is also a round-robin queue, which has the elastic maximum lines (=*M*) depending on **time_precision** and **Stop**, and is controlled by the output buffer controller. The output buffer controller also has the following two operations:

(1) Buffer writing- to store the values **ResponseValue** of the current monitored response signals into the **Tail** pointed line of the output buffer.

(2) Buffer reading- to load the values **ResponseValue** of the current monitored response signals from the **Head** pointed line of the output buffer so as to send them to the computer side through the serial transmitter.

5. EXPERIMENTS AND ANALYSIS

5.1. Experiment Environment

According to the proposed in-circuit debugging kit for IP core design, we build the in-circuit debugging platform. The software emulating environment is implemented in C# language which supports graphical user interface helpful to draw waveforms easily, and the hardware testing environment is constructed in Verilog HDL which supports IP modular design helpful to generate codes automatically. The target device is Xilinx Artix-7 XC7A35T FPGA.

In the software emulating environment, users need carry out the following operations:

- Select the serial port and its baud rate;
- Set the trigger mechanism(**Posedge**, **Negedge**, **High**, **Low**) and the half cycle of the clock of the DUT;
- Set the time unit;
- Set the time precision;
- Set the simulation time;
- Set the simulated ports and test vectors;
- Set the monitored ports.

Both custom mode and imported mode are supported for setting simulated ports and test vectors. In the custom mode, users need set **Delay**, all port names (**Port**) and their values (**Value**) for each test vector as shown in Fig. (**9a**). This mode is suitable for less test vectors. If there are more test vectors, the imported mode is recommended. In the imported mode, users only need set each simulated port name (**Port**) and its width (**Width**) for the test vector, and import the test vector values from a test vector file which is defined according to the formats as shown in Fig. (**9b**) in Section 3.1.

After completing the above operations, users can click the running button to start sending the stimulus vectors to the FPGA side so that latter can receive them, assign them to the DUT and generate the expected monitored signals, and send those monitored signals to the computer side. Then users can observe the waveforms of all monitored signals as shown in Fig. (**9c**), or open the monitor file and view the details.

5.2. Performance Analysis

In-circuit debugging platform is aimed to improve the simulation accuracy of the DUT. However, owing to the serial communication, the acceleration ability of the debugging platform determines the simulation efficiency and effect. According to the debugging process, from sending of stimulus vectors to receiving of monitored response signals, the total spent time includes the following three parts:

(1) Time of sending of stimulus vectors

Assumed that there are *m* stimulus vectors, and each stimulus vector has 14+*w* bits, the total number of sent bytes is $m\times\left(2+\left\lceil\frac{w}{7}\right\rceil\right)$. Besides, the **Stop** command has 2 bytes for **Delay** and for 1 byte for **Stop**. Therefore, in all, there are $m\times\left(2+\left\lceil\frac{w}{7}\right\rceil\right)+3$ bytes which need to be sent through the serial communication. If a frame of serial data has 1 start bit, 8 data bits for one byte, 1 parity check bit and 1 stop bit, and the baud rate is λ, then the total sending time is:

$$T_{send}=\frac{\left(m\times(2+\left\lceil\frac{w}{7}\right\rceil)+3\right)\times 11}{\lambda} \tag{3}$$

(2) Time of testing of the DUT

Assumed that there are *m* stimulus vectors which have **Delay**s $D_1, D_2, ..., D_m$, and the **Stop** command has the **Delay** D_{Stop}, we have the total delay

$$D=\sum_{i=1}^{m}D_i+D_{Stop} \tag{4}$$

Therefore the testing time is

$$T_{Test}=D\times TimeUnit \tag{5}$$

(3) Time of receiving of monitored response signals

During the working period of the DUT, the total number of sampling is

$$M=\frac{T_{Test}}{TimePrecision} \tag{6}$$

Therefore the total number of bytes which need to be transmitted is $\left\lceil\frac{W\times M}{8}\right\rceil$, and the spent receiving time in the computer side is

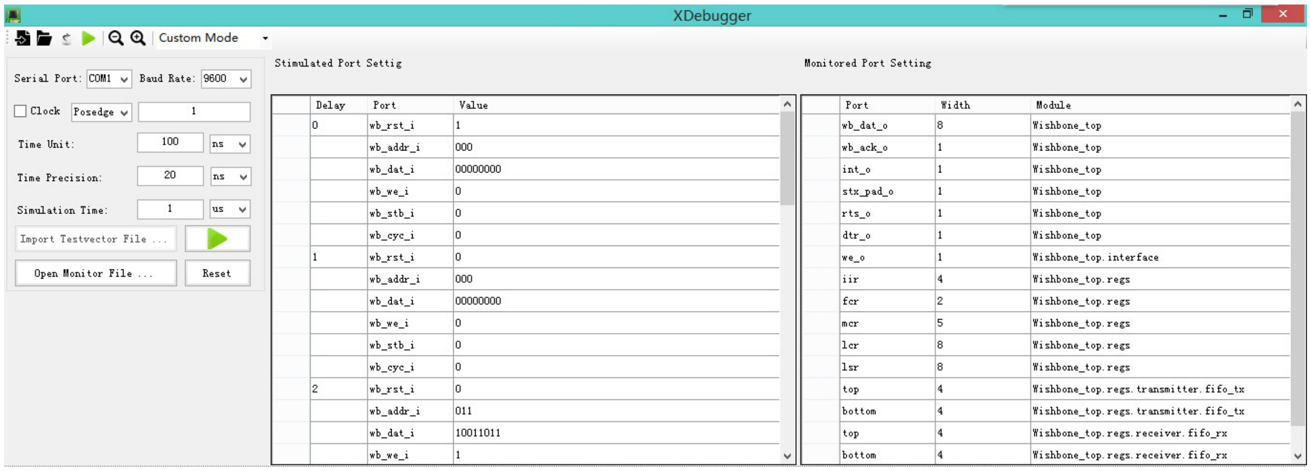
$$T_{\mathrm{Re}ceive}=\frac{\left\lceil\frac{W\times M}{8}\right\rceil\times 11}{\lambda} \tag{7}$$

During the working of the DUT, the generated monitored response signals are also being sent to the computer side. Therefore the total debugging time is

$$T<T_{Send}+T_{Test}+T_{\mathrm{Re}ceive} \tag{8}$$

5.3. Resource Analysis

In the in-circuit debugging platform, only the DUT is our design module. In order to verify its correctness and feasibility, we need design some auxiliary modules such as the input buffer, stimulator, monitor, output buffer and UART in the FPGA side. All these modules consume FPGA resources. Table **1** gives the resource consumption statistics of some test instances in the case of *time_precision=20ns, time_unit=100ns, clock_cycle =2, m=100*. Among them,



Serial Port: COM1 Baud Rate: 9600
Clock Posedge 1
Time Unit: 100 ns
Time Precision: 20 ns
Simulation Time: 1 us

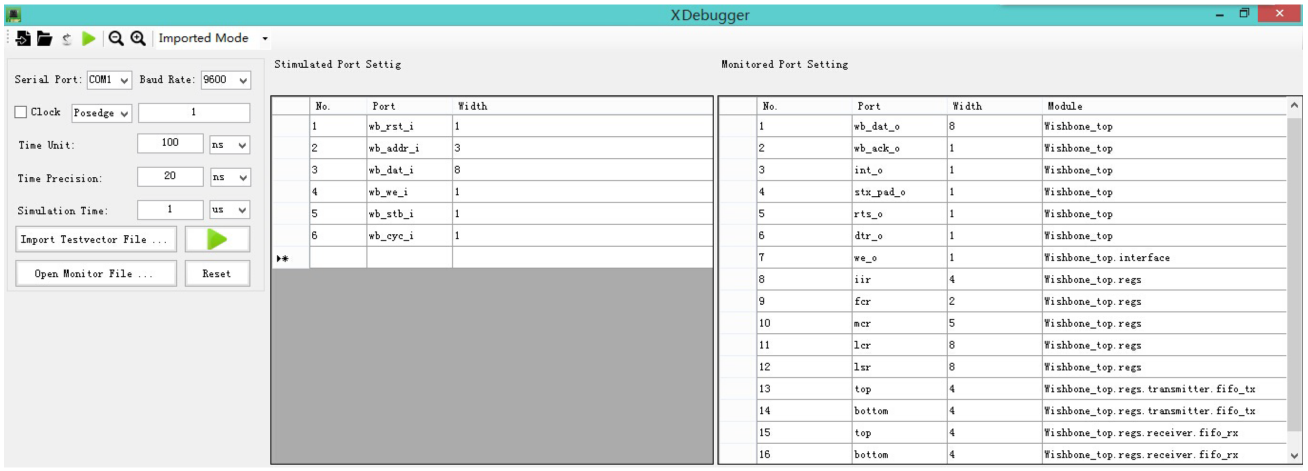
Stimulated Port Settig

Delay	Port	Value
0	wb_rst_i	1
	wb_addr_i	000
	wb_dat_i	00000000
	wb_we_i	0
	wb_stb_i	0
	wb_cyc_i	0
1	wb_rst_i	0
	wb_addr_i	000
	wb_dat_i	00000000
	wb_we_i	0
	wb_stb_i	0
	wb_cyc_i	0
2	wb_rst_i	0
	wb_addr_i	011
	wb_dat_i	10011011
	wb_we_i	1

Monitored Port Setting

Port	Width	Module
wb_dat_o	8	Wishbone_top
wb_ack_o	1	Wishbone_top
int_o	1	Wishbone_top
stx_pad_o	1	Wishbone_top
rts_o	1	Wishbone_top
dtr_o	1	Wishbone_top
we_o	1	Wishbone_top.interface
iir	4	Wishbone_top.regs
fcr	2	Wishbone_top.regs
mcr	5	Wishbone_top.regs
lcr	8	Wishbone_top.regs
lsr	8	Wishbone_top.regs
top	4	Wishbone_top.regs.transmitter.fifo_tx
bottom	4	Wishbone_top.regs.transmitter.fifo_tx
top	4	Wishbone_top.regs.receiver.fifo_rx
bottom	4	Wishbone_top.regs.receiver.fifo_rx

(**a**) Custom mode



Serial Port: COM1 Baud Rate: 9600
Clock Posedge 1
Time Unit: 100 ns
Time Precision: 20 ns
Simulation Time: 1 us

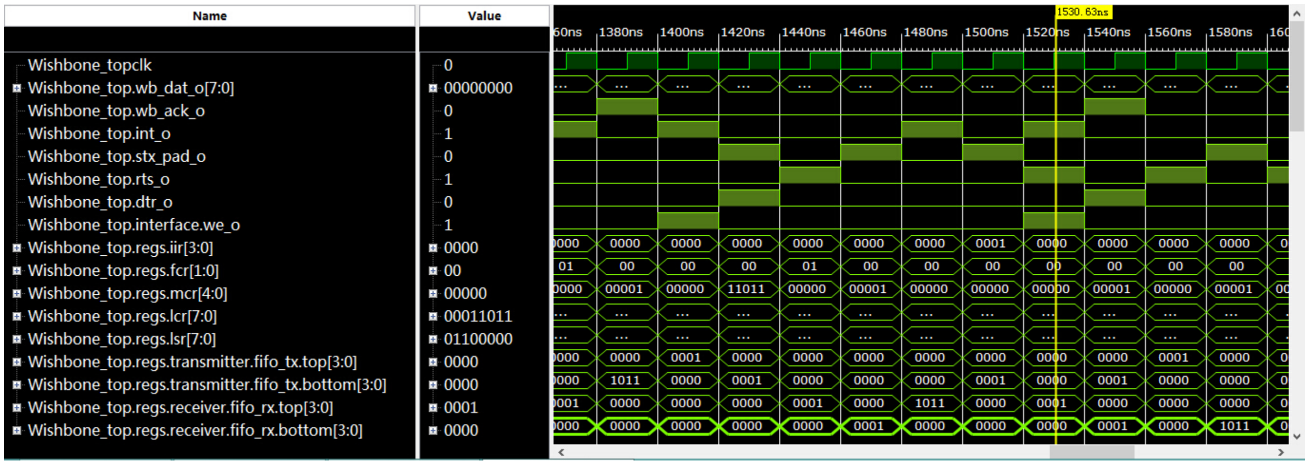
Stimulated Port Settig

No.	Port	Width
1	wb_rst_i	1
2	wb_addr_i	3
3	wb_dat_i	8
4	wb_we_i	1
5	wb_stb_i	1
6	wb_cyc_i	1

Monitored Port Setting

No.	Port	Width	Module
1	wb_dat_o	8	Wishbone_top
2	wb_ack_o	1	Wishbone_top
3	int_o	1	Wishbone_top
4	stx_pad_o	1	Wishbone_top
5	rts_o	1	Wishbone_top
6	dtr_o	1	Wishbone_top
7	we_o	1	Wishbone_top.interface
8	iir	4	Wishbone_top.regs
9	fcr	2	Wishbone_top.regs
10	mcr	5	Wishbone_top.regs
11	lcr	8	Wishbone_top.regs
12	lsr	8	Wishbone_top.regs
13	top	4	Wishbone_top.regs.transmitter.fifo_tx
14	bottom	4	Wishbone_top.regs.transmitter.fifo_tx
15	top	4	Wishbone_top.regs.receiver.fifo_rx
16	bottom	4	Wishbone_top.regs.receiver.fifo_rx

(**b**) Imported mode



Name	Value
Wishbone_topclk	0
Wishbone_top.wb_dat_o[7:0]	00000000
Wishbone_top.wb_ack_o	0
Wishbone_top.int_o	1
Wishbone_top.stx_pad_o	0
Wishbone_top.rts_o	1
Wishbone_top.dtr_o	0
Wishbone_top.interface.we_o	1
Wishbone_top.regs.iir[3:0]	0000
Wishbone_top.regs.fcr[1:0]	00
Wishbone_top.regs.mcr[4:0]	00000
Wishbone_top.regs.lcr[7:0]	00011011
Wishbone_top.regs.lsr[7:0]	01100000
Wishbone_top.regs.transmitter.fifo_tx.top[3:0]	0000
Wishbone_top.regs.transmitter.fifo_tx.bottom[3:0]	0000
Wishbone_top.regs.receiver.fifo_rx.top[3:0]	0001
Wishbone_top.regs.receiver.fifo_rx.bottom[3:0]	0000

(**c**) Waveform display

Fig. (9). Debugging modes.

Table 1. FPGA resources.

No.	DUT	Number of FPGA Slices/Slice Flip Flops/4 input LUTs						
		w	W	Input Buffer	Configurator	Monitor	Output Buffer	DUT
1	Decoder	3	8	890/1063/1793	104/98/105	121/63/219	673/521/1045	4/0/8
2	DIV	32	33	2912/1728/5528	108/105/187	141/94/255	2287/1981/3845	78/0/139
3	ALU	38	16	3455/2010/6424	110/107/203	135/80/242	1210/1012/1974	197/23/363
4	PWM	3	34	912/1093/1841	105/99/106	143/95/258	2358/2022/3879	29/41/57
5	EEPROM	24	11	2378/1673/4812	108/104/194	123/66/232	928/716/1438	109/67/210
6	FIFO	15	10	1368/1363/2483	106/103/115	123/65/226	835/653/1322	35/41/49
7	I2C	10	8	1355/1291/2305	105/99/195	121/63/219	673/521/1036	235/185/439
8	VGA	2	25	837/869/1573	102/98/103	141/90/258	2101/1811/3473	244/30/464
9	Wishbone	16	13	1626/1143/3290	109/105/192	127/70/235	1095/845/11696	507/488/961

Decoder, DIV and ALU are three combinational logic modules, PWM and EEPROM are two sequential logic modules, FIFO, I2C and VGA are three two-layer hierarchical logic modules, and Wishbone is a mutli-layer hierarchical logic module. As can be seen, a large number of resources are consumed by the input buffer and the output buffer.

If there are m stimulus vectors and each stimulus vector has w bits, the total number of bits for the **Delay** buffer and the **VectorValue** buffer is

$$b = m \times (14 + w) \tag{9}$$

If there are M times of sampling and each sampling generates W bits of response signals, the total number of bits for the **ResponseValue** buffer is

$$B = M \times W \tag{10}$$

All of them will consume lots of FPGA resources.

6. CONCLUSIONS AND FUTURE WORKS

In-circuit debugging is one of the key verification technologies in SoC design. With the development of SoC technology, IP core design helps to accelerate the product development. Unfortunately, the traditional verification methods are facing many problems. For example, the debugging platform is hard to reuse, and needs to be rebuilt when the DUT is only changed a little. It is also hard to push and capture the effective signals for observing to find errors.

By above knowledge, in this paper, an efficient elastic in-circuit debugging method is presented for verification of complex digital logic designs. This verification method provides a perfect platform for FPGA-based functional simulation. The successful IP verification instances have demonstrated that this debugging platform can decrease the complexity of verification by reusable elastic modules and provide a configurable solution to import test cases to the DUT. The in-circuit debugging platform makes the technology ubiquitous and low cost, and eliminates most differences between the development and runtime environments. Compared to Lauterbach in-circuit debugger which is a hardware assisted debug tool for embedded systems, especially some chip products, and exchanges debugging data between the computer side and the target chip using JTAG, our in-circuit debugging method is mainly used in the design and development stage of the modules and IP cores, and can be more flexible in making use of the advantages of the elasticity of FPGA devices to debug and verify different complex digital logic circuits.

Nevertheless, the current in-circuit debugging platform still has the following two problems which need to be solved in the future:

- Only the two valued logic is provided and the high impedance signal (z) is not supported.
- The low rate of serial port cannot meet the rapid data exchange between the computer side and the FPGA side.

CONFLICT OF INTERESTS

The authors declare that there is no conflict of interests regarding the publication of this paper.

ACKNOWLEDGEMENTS

This work is supported by the National Natural Science Foundation of China (No. 61370050), Anhui Provincial Natural Science Foundation (No. 1308085QF118), University Natural Science Research Project of Anhui Province (No. KJ2014A084), Wuhu City Science and Technology Key Project (No. 2014cxy04), and Anhui Normal University Postdoctoral Project (No.161-071214).

REFERENCES

[1] Z. Hu, A. Pierres, and S. Hu, "Practical and efficient SOC verification flow by reusing IP testcase and testbench", *Proceedings of IEEE International SoC Design Conference (ISOCC)*, pp. 175-178, 2012.

[2] T. Nahtigal, P. Puhar, and A. Zemva, "A systematic approach to configurable functional verification of HW IP blocks at transaction level", *Computers \& Electrical Engineering,* vol. 38, no. 6, pp.1513-1523, 2012.

[3] S. S. Shankar, and J. S. Shankar, "Synthesizable verification IP to stress test system-on-chip emulation and prototyping platforms", *Proceedings of 13th IEEE International Symposium on Integrated Circuits (ISIC)*, pp.609-612, 2011.

[4] J. N. Xu, F. F. Fu, and J. X. Wang, "Data Path design of FPGA-based HW/SW co-simulation platform", *Microelectronics & Computer*, vol. 31, no. 3, pp.107-114, 2014.

[5] Ptolemaeus C, System Design, Modeling, and Simulation: Using Ptolemy II, http://Ptolemy.org, 2014.

[6] J. Eker, J. W. Janneck, and E. A. Lee, "Taming heterogeneity-the Ptolemy approach", *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127-144, 2003.

[7] E. S. Chung, E. Nurvitadhi, and J. C. Hoe, "PROToFLEX: FPGA-accelerated hybrid functional simulator", P*roceedings of IEEE International Parallel and Distributed Processing Symposium*, pp.1-6, 2007.

[8] E. S. Chung, E. Nurvitadhi, and J. C. Hoe, "A complexity-effective architecture for accelerating full-system multiprocessor simulations using FPGAs", *Proceedings of the 16th ACM international ACM/SIGDA symposium on Field programmable gate array*s, pp.77-86, 2008.

[9] J. Wawrzynek, D. Patterson, and M. Oskin, "RAMP: Research accelerator for multiple processors", IEEE Micro, vol. 27, no. 2, pp.46-57, 2007.

[10] Z. Tan, A. Waterman, and R. Avizienis, "RAMP gold: an FPGA-based architecture simulator for multiprocessors", *Proceedings of the 47th ACM Design Automation Conference*, pp.463-468, 2010.

[11] ChipScope Pro Software and Cores User Guide, Xilinx UG029 (v14.3), http://www.xilinx.com, 2012.

[12] F. L. Chen, Z. X. Zhu, and X. Y. Fan, "FPGA-Based In-Circuit Verification of Digital Systems", *Advanced Materials Research*, vol. 187, pp.362-367, 2011.

[13] Z. K. Huang, , "A new image thresholding method based on gaussian mixture model", applied mathematics and computation, vol.205, no.2, pp. 899-907, 2008.

[14] R. Taormina, "Artificial Neural Network simulation of hourly groundwater levels in a coastal aquifer system of the Venice lagoon", *Engineering Applications of Artificial Intelligence*, vol.25, no.8, pp. 1670-1676, 2012.

[15] C.T. Cheng, "Long-Term prediction of discharges in manwan reservoir using artificial neural network models", L*ecture Notes in Computer Science,* vol.3498, pp.1040-1045, 2005.

[16] K. W. Chau, "Application of a PSO-based neural network in analysis of outcomes of construction claims", *Automation in Construction*, vol.16, no.5, pp.642-646, 2007.

[17] C. L. Wu, K. W. Chau, and Y. S. Li, "Predicting monthly streamflow using data-driven models coupled with data-preprocessing techniques", *Water Resources Research*, vol.45, no.8, pp.2263-2289, 2009.

[18] J. Z. K. Chau, "Multilayer Ensemble Pruning via Novel Multi-sub-swarm Particle Swarm Optimization", *Journal of Universal Computer Science*, vol.15, no.4, pp.840-858, 2009.

Received: May 07, 2015 Revised: May 19, 2015 Accepted: June 20, 2015