TELKOMNIKA Telecommunication, Computing, Electronics and Control
Vol. 18, No. 3, June 2020, pp. 1229∼1236
ISSN: 1693-6930, accredited First Grade by Kemenristekdikti, No: 21/E/KPT/2018
DOI: 10.12928/TELKOMNIKA.v18i3.13174

On-chip debugging for microprocessor design

Fajar Suryawan, Bana Handaga, Abdul Basith
Department of Electrical Engineering, Universitas Muhammadiyah Surakarta, Indonesia

Article Info

Article history:

Received May 20, 2019
Revised Jan 22, 2020
Accepted Feb 21, 2020

Keywords:

Engineering education
Field programmable gate array
Microprocessor design
Post-silicon debug
Programmable logic

ABSTRACT

This article proposes a closer-to-metal approach of RTL inspection in microprocessor design for use in education, engineering, and research. Signals of interest are tapped throughout the microprocessor hierarchical design and are then output to the top-level entity and finally displayed to a VGA monitor. Input clock signal can be fed as slow as one wish to trace or debug the microprocessor being designed. An FPGA development board, along with its accompanying software package, is used as the design and test platform. The use of VHDL commands 'type' and 'record' in the hierarchy provides key ingredients in the overall design, since this allows simple, clean, and tractable code. The method is tested on MIPS single-cycle microprocessor blueprint. The result shows that the technique produces more consistent display of the true contents of registers, ALU input/output signals, and other wires – compared to the standard, widely-used simulation method. This approach is expected to increase confidence in students and designers since the reported signals' values are the true values. Its use is not limited to the development of microprocessors; every FPGA-based digital design can benefit from it.

Corresponding Author:

Fajar Suryawan,
Department of Electrical Engineering,
Universitas Muhammadiyah Surakarta,
Jl. A. Yani, Tromol Pos 1, Sukoharjo, Jawa Tengah, Indonesia.
Email: Fajar.Suryawan@ums.ac.id

1. INTRODUCTION

Digital device development depends greatly on precise understanding how data propagate between basic digital logic units, also called *register transfer level*. In design phase, designers often use simulation procedures to check whether their designs meet the logic requirements. An example of this is also encountered in senior level electrical/computer engineering bachelor-degree courses such as *Programmable Logic Design* or *Computer Architecture* [1–4]. In such courses, students are asked to design a micro-architecture of a microprocessor based on a given architecture (that is, the assembly language requirement). Students then write HDL code representing the micro-architecture and test the design against a set of instructions. Testing is generally done in simulation and, after a number of testing-coding iterations, hardware test is performed.

Software simulation is indispensable for its quick setting, fast compilation, and – provided the designer is experienced – accuracy. However, mismatches between simulation and synthesized hardware are not entirely unheard of, even for simple design. Mismatches also occur between pre-synthesis and post-synthesis simulations. To make matter worse, in post-synthesis (netlist) simulation one generally can only monitor the top-level ports; signals deeper in hierarchy are inaccessible. To address this problem, we propose a closer-to-metal approach for the register transfer level inspection. Effectively, this is an on-chip debugging technique

where signals of interest are brought up to the top level for output reading.

The term *on-chip debug* generally refers to a technique in microprocessor (or other digital device) design where a designer can inject a fault in the device under development to test its fault tolerant behavior [5–8]. For microprocessors, this is usually done using JTAG protocol based on NEXUS Consortium standard. In [9], on-chip debug is used in high level synthesis for FPGAs.

On-chip debug has been a concern from the beginning of computer era. FPGA has also taken part in this field. For example, work by Jamal et.al [10, 11] proposes better functional changes during on-chip debug, utilizing FPGA overlay architecture. Contemporary works in this field, particularly post-silicon debug, can be found in [12–15]. Indeed, post-silicon debug readiness needs to be prepared early in the design [16–18]. A number of authors extend the idea to other areas such as machine learning [19, 20].

In this article we describe *on-chip debug* more in its literal meaning. That is, the process of debugging a microprocessor in which the debug capability is embedded in the hardware design. Our contribution lies in the following aspects. First, we propose a simple hardware-oriented debugging method for use in any digital design. We hope this introductory notion will trigger students' creative faculty to solve some challenging problems that otherwise difficult to tackle. Second, we describe – in a tutorial way – the construction of a simple on-chip debugging feature in the design of a microprocessor using 'type' and 'record' in vhdl. We believe this will help students and designers easily duplicate our work.

2. RESEARCH METHOD

This research starts with a list of design requirements for an on-chip debugging feature in a microprocessor:

(a) non-intrusiveness: the debugging feature should be as discreet as possible so as not to obstruct the main design
(b) meaningful message: the interface to human reader should be immediately readable
(c) easy to modify: when the designer wants to tap other signals in the microprocessor design, it should be straightforward to do so

The main idea of this chip debugging feature is shown in Figure 1.

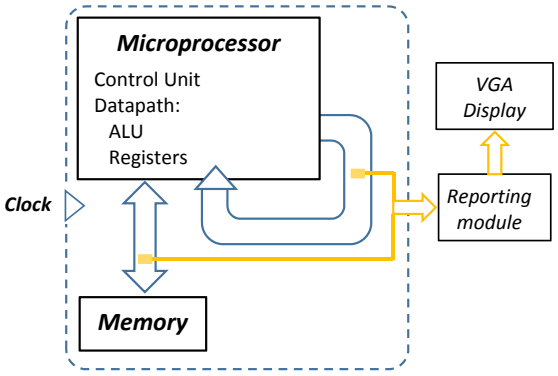


Figure 1. On-chip debugging principles in this paper.

The next step is building a model microprocessor from scratch using VHDL in an FPGA chip (an Altera DE2-115 development board was used). Here we use a scaled-down version of the MIPS microprocessor architecture [21, 22] as our proof of concept. MIPS architecture has the advantages of, among others, being simple and consistent for students to follow. MIPS is of RISC-architecture and originated as a pedagogical model at Stanford University. We developed the chip-debugging feature based on a MIPS implementation described by Harris and Harris [23]. In this stage a VGA display module were also built, consisting of a vga sync module, two character memory modules, and a font ROM. The experimental hardware setup is shown in Figure 2. We extend the single-cycle MIPS instruction set found in the main text of [23] by constructing a number of new instructions and 'rewire' the datapath as needed. We then verify – using the proposed on-chip debugging technique – that the final result works as expected. In the next section we will discuss the engineering design in more detail.

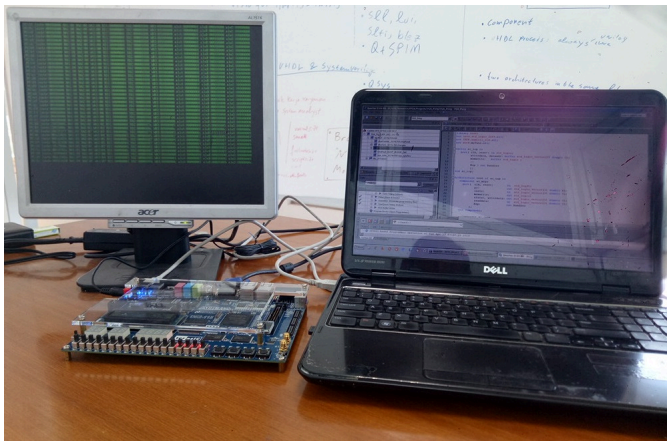


Figure 2. Hardware setup. Altera DE2-115 development board is used to test the proposed technique, together with a 1024x768 resolution monitor.

3. PROTOTYPE DESIGN

The design requires several aspects to work seamlessly together. These are: processor design with its signal inspection, information display, and experiment design.

3.1. Processor design and signal inspection

As mentioned before, the approach works by sensing internal microprocessor signals (including memory access ones) and sending them up through the design hierarchy. Using hierarchical design implies that many entities and files are used, which poses a new challenge on how to tap signals from different entities in a straightforward and unobtrusive way. The signal tapping as shown in Figure 1 is implemented using a shared bus that is available across the hierarchy. Figure 3 shows the organization of modules that make up the entire microprocessor.

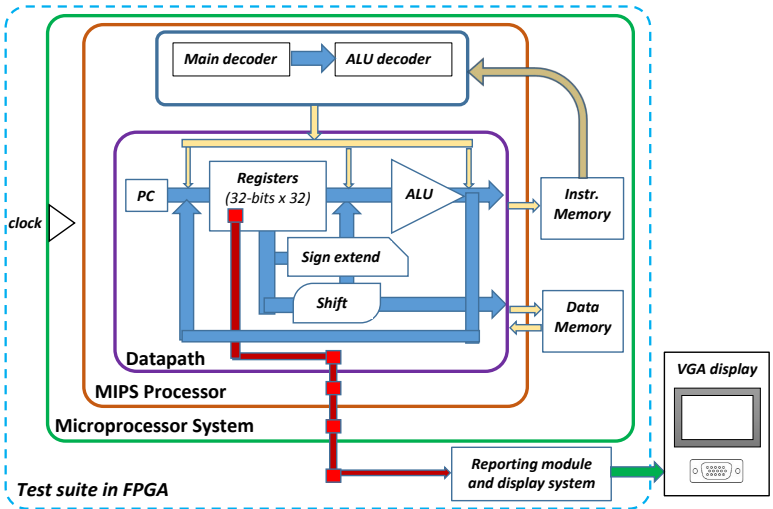


Figure 3. Module organization and hierarchy. The small red blocks are instantiations of the record entity that encapsulates the tapped signals' information. The red 'cable' then acts as a debug-bus across the hierarchy.

The *arithmetic logic unit* (ALU) does the arithmetic computation with the help of a set of registers (32-bits $\times$ 32) that functions as a scratchpad for the ALU. *Program counter* (PC) acts as a pointer to instruction. Sign-extend module extends less-than-32-bit-wide numbers (for example in immediate-type instructions) to its 32-bit representation. The shift module functions as bit-wise shifter. All these are in the "datapath" module

which also hosts a number of multiplexers controlling which way data would flow into. Controlling is done by the "controller" module outside the datapath, which decode 32-bit instructions from the instructions memory. Datapath and Controller forms the "MIPS processor" module. Together with *Instruction Memory* and *Data Memory* modules, they make up the complete microprocessor system.

The signal tapping, shown as red blocks in Figure 3, is a record-type entity instantiated at every modules of interest. Acting as a "debugging bus", this record is ready to accept the value of any signal of interest in every level in the hierarchy. Since the bus is logically encapsulated, it does not obstruct the main design. Modifying the bus' content is straightforward and can be done once in the definition, without the need to change any code in the instantiation part.

```
-- File name: myfuns.vhd
-- ... Other statements ...

-- 'Regsbundler' below is for collecting
--     contents of registers.
-- There are 32 registers, each 32 bit wide.
type regsbundler is record
   R00,R01,R02,R03,R04,R05,R06,R07,
   R08,R09,R10,R11,R12,R13,R14,R15,
   R16,R17,R18,R19,R20,R21,R22,R23,
   R24,R25,R26,R27,R28,R29,R30,R31:
         std_logic_vector(31 downto 0);
end record;

-- 'Bundler' below is for collecting
--     signals in datapath.
-- 'Regsbundler' above is also included.
type bundler is record
   pc : std_logic_vector(31 downto 0);
   instr : std_logic_vector(31 downto 0);
   RA1,RA2,WA3:std_logic_vector(4 downto 0);
   alua,alub:std_logic_vector(31 downto 0);
   aluout:std_logic_vector(31 downto 0);
   RD1,RD2,WD3:std_logic_vector(31 downto 0);
   regs:regsbundler; -- registers' contents
end record;
-- ...
```

```
-- File name: sc_datapath.vhd
-- ... Other library declarations ...
use work.myfuns.all;

Entity sc_datapath is
port( -- Other port definitions
      -- ...
      Reporter : out bundler
      );
end sc_datapath;

Architecture struct of sc_datapath is
-- ... Other statements ...
----- Reporter collects ------------
Reporter.pc <= pc;
Reporter.instr <= instr;
Reporter.RA1 <= instr(25 downto 21);
Reporter.RA2 <= instr(20 downto 16);
Reporter.WA3 <= writereg;
Reporter.RD1 <= srca  ;
Reporter.RD2 <= writedata ;
Reporter.WD3 <= result ;
Reporter.alua   <= srca;
Reporter.alub   <= srcb;
Reporter.aluout <= aluout;
-----------------------------------
-- ...
```

Figure 4. 'Record' type for the construction of debugging bus. On the left: definition. On the right: example instantiation and usage in the datapath module.

3.2. Information display

To show the debugging steps, the values of signals of interest are displayed in VGA monitor, one row per clock. The Altera DE2-115 board is equipped with a VGA port, but users must themselves program the VGA synchronization and character generation. Here we adopt the method described in [24], and the arrangement is depicted if Figure 5. In this layout, *phase locked loop* (PLL) is used step up the clock frequency. It feeds the VGA sync module, which produces the horizontal and vertical synchronization signals, to be used by the VGA display. The vga sync module also generates information of current pixel's x and y position. This information is used by the character generation and Font-ROM modules to render appropriate character at a given time. Again, the interested readers are referred to [24] for further technical details regarding the display arrangement. The microprocessor system, shown in Figure 5, also in Figure 3 as green-outlined box, transmits the debugging signals off to the report compiler where all tapped signals are lined up and sent off to the character generation circuit.

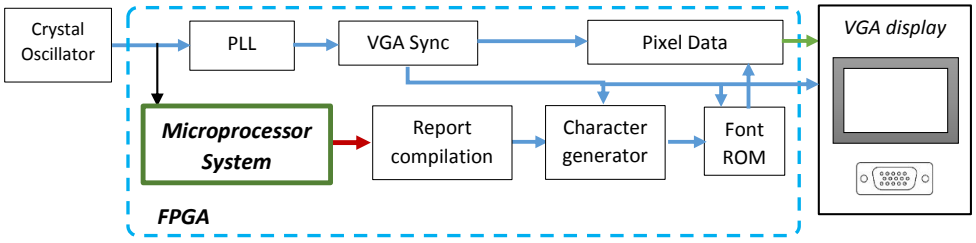


Figure 5. Display arrangement, including reporting module.

3.3. Test design

The next step in this work is to design a test which will confirm the functionality of of the overall setup, which will serve as a proof of concept for our proposed method. A microprocessor architecture expansion task is chosen as the test. That is, new instructions are to be introduced to set. This will require a modification in the micro-architecture of our microprocessor and testing the functionalities of the new instructions. As the base architecture, we adopt [23], which in turn was inspired by earlier editions of [25]. Specifically, the reader are referred to Chapter 6 (Architecture) and Chapter 7 (Microarchitecture) of [23].

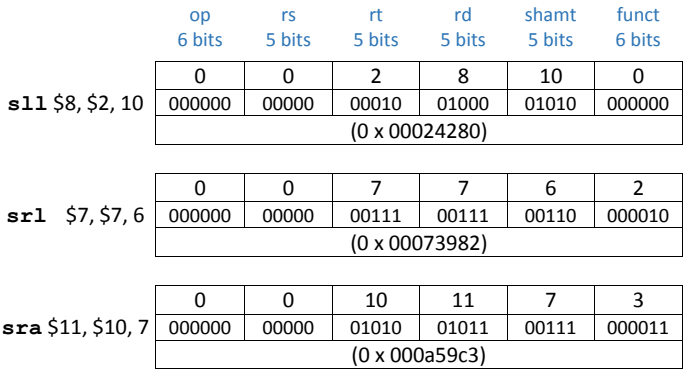
These added instructions are *shift left logical* (sll), *shift right logical* (srl), and *shift right arithmetic* (sra). The three instructions are of R-type instruction and have the same invocation form. For instance, the format for sll is:

sll rd,rt,shamt

where *rd* is the destination register, *rt* is the source register. The four-bytes data is stored in *rt* is shifted left by *shamt* amount, and then stored in *rd*. Similar form holds for *srl* and *sra*.

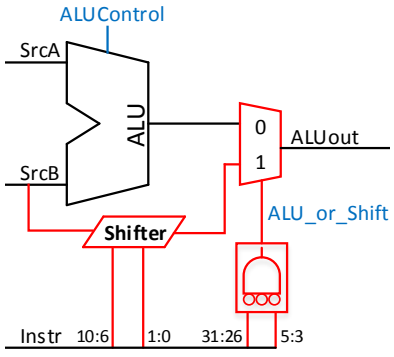
The architecture for these three register-type instructions is shown in Figure 6(a). Following the convention, the six most significant bits (instr[31:26], the op field) are 0, indicating R-type instructions. The 6 least significant bits (instr[5:0], the funct field) indicate which R-type instruction is operative. (And only last two bit indicate which of the three shifts will be operative). The shift amount is placed in the shamt field (instr[10:6]). The source and destination register are in the rt and rd fields, respectively.

Based on the above architecture, a number of modifications are implemented in the microarchitecture. Figure 6(b) shows part of the new microarchitecture design. The shifter module receives the instr[10:6] as the amount of shift and receives instr[1:0] as shift mode chooser (which of the three shift commands is operative). The 32 bit data (to be shifted) comes from the register and is output to a multiplexer, which will choose between two signals: output from the original ALU or output from the shifter.



	op 6 bits	rs 5 bits	rt 5 bits	rd 5 bits	shamt 5 bits	funct 6 bits
sll $8, $2, 10	0	0	2	8	10	0
	000000	00000	00010	01000	01010	000000
	(0 x 00024280)					
srl $7, $7, 6	0	0	7	7	6	2
	000000	00000	00111	00111	00110	000010
	(0 x 00073982)					
sra $11, $10, 7	0	0	10	11	7	3
	000000	00000	01010	01011	00111	000011
	(0 x 000a59c3)					

(a)



(b)

Figure 6. New instructions for the microprocessor: sll, srl, and sra. (a) Bit arrangement within 32-bit wide instruction, conforming MIPS ISA in [23]. (b) Hardware implementation in the micro-architecture (only modified part is shown).

After modifying the microarchitecture, a set of code will be used to confirm the validity. The test code formulated is a continuation of the test code in [23], page 437. It consists of computations involving all instruction by which a specific state is targeted ("value of 7 is stored in memory address 84"). A fault in the implementation of any instruction will render the target state not achieved. It is very unlikely to produce the expected result under faulty condition.

Figure 7 shows the overall instruction test, written in MIPS assembly language. The blue lines (address 0 to 40 and address 64) are the original test from [23], and the green ones (address 44 to 60) are the new code. In the end of the test, the computed value (happens to be 126 = 0x7e) is stored in the memory address 84 = 0x54.

```
#         Assembly             Description               Address   Machine
#-------------------------------------------------------------------------
main:     addi $2, $0, 5       # initialize $2 = 5          0       20020005
          addi $3, $0, 12      # initialize $3 = 12         4       2003000c
          addi $7, $3, -9      # initialize $7 = 3          8       2067fff7
          or   $4, $7, $2      # $4 = (3 OR 5) = 7          c       00e22025
          and  $5, $3, $4      # $5 = (12 AND 7) = 4        10      00642824
          add  $5, $5, $4      # $5 = 4 + 7 = 11            14      00a42820
          beq  $5, $7, end     # shouldn't be taken         18      10a7000a
          slt  $4, $3, $4      # $4 = 12 < 7 = 0            1c      0064202a
          beq  $4, $0, around  # should be taken            20      10800001
          addi $5, $0, 0       # shouldn't happen           24      20050000
around:   slt  $4, $7, $2      # $4 = 3 < 5 = 1             28      00e2202a
          add  $7, $4, $5      # $7 = 1 + 11 = 12           2c      00853820
          sub  $7, $7, $2      # $7 = 12 - 5 = 7            30      00e23822
          sw   $7, 68($3)      # [80] = 7                   34      ac670044
          lw   $2, 80($0)      # $2 = [80] = 7              38      8c020050
          j    sh_test         # should be taken            3c      08000011
          addi $2, $0, 1       # shouldn't happen           40      20020001
sh_test:  sll  $8, $2, 10      # $8 = 7*2^10= 7168          44      00024280
          addi $7, $8, 256     # $7 = 7168+256 = 7424       48      21070100
          srl  $7, $7, 6       # $7 = 7424/2^6 = 116        4c      00073982
          addi $9, $0, 1280    # initialize $9 = 1280       50      20090500
          sub  $10, $0, $9     # $10 = -1280                54      00095022
          sra  $11, $10, 7     # $11 = -1280 SRA 7 = -10    58      000a59c3
          sub  $12, $0, $11    # $12 = 0 - (-10) = 10       5c      000b6022
          add  $2, $7, $12     # $2 = 116 + 10 = 126        60      00ec1020
end:      sw   $2, 84($0)      # write mem[84] = 126        64      ac020054
```

Figure 7. Testing the new microarchitecture. Blue lines (address 0 to 40 and address 64) are test code from [23]. Green lines (address 44 to 60) are inserted to test new instructions implemented in the microarchitecture.

4. RESULT AND ANALYSIS

Two scenarios are administered to assess the proposed method. First one involves simulation of the microprocessor system using ModelSim. The second one is to run the microprocessor in the Altera FPGA chip. In both scenarios, the same microprocessor module is used (green box in Figure 3). In the first scenario –the ModelSim simulation– a SystemVerilog testbench file is built as wrapper for the microprocessor. The second scenario synthesizes the microprocessor and programmed the netlist into the FPGA chip.

In both scenarios, the test code in its machine form as shown in Figure 7, rightmost column, is embedded to the chip in the Instruction Memory. The instruction is executed sequentially, with jumps at specific moments. Debugging in microprocessor design most of the times involves probing values of internal signals such as program counter, instruction, register address input, register data input, memory address, ALU inputs, ALU output. Indeed, these are the signals we will display in both scenarios.

Figure 8 shows the result for the first scenario. It can be seen that a number of signal values are not resolved. Instead of showing values of signals, 'xxxxxxxx' are shown. In the second scenario, where on-chip debugging features are employed in the FPGA-based microprocessor chip, all signal values are delineated correctly. The result is depicted in Figure 9. The difference in the signal display (though the final result is the same, implying correct model/design) might come from overly tight timing specification, coarse timing resolution, multiply driven signals, different initial states, or simply a bug in the simulation software. The on-chip debugger, on the other hand, shows real data from the hardware (though still limited by the display system's speed capability).

While it is relatively straightforward (but not necessarily easy) to fix bugs in the simulation side, hardware reporting is precious and some times the only choice a designer has. The MIPS example shown here serves as a demonstration of this on-chip debugging technique.

```
VSIM 2> run
#    pc=00, instr=20020005, A1=00, A2=02, A3=02, RD1=00000000, RD2=00000005, WD3=00000005, alua=00000000, alub=00000005, aluout=00000005
#    pc=04, instr=2003000c, A1=00, A2=03, A3=03, RD1=00000000, RD2=xxxxxxxx, WD3=0000000c, alua=00000000, alub=0000000c, aluout=0000000c
#    pc=08, instr=2067fff7, A1=03, A2=07, A3=07, RD1=0000000c, RD2=xxxxxxxx, WD3=00000003, alua=0000000c, alub=fffffff7, aluout=00000003
#    pc=0c, instr=00e22025, A1=07, A2=02, A3=04, RD1=00000003, RD2=00000005, WD3=00000007, alua=00000003, alub=00000005, aluout=00000007
#    pc=10, instr=00642824, A1=03, A2=04, A3=05, RD1=0000000c, RD2=00000007, WD3=00000004, alua=0000000c, alub=00000007, aluout=00000004
#    pc=14, instr=00a42820, A1=05, A2=04, A3=05, RD1=00000004, RD2=00000007, WD3=0000000b, alua=00000004, alub=00000007, aluout=0000000b
#    pc=18, instr=10a7000a, A1=05, A2=07, A3=07, RD1=0000000b, RD2=00000003, WD3=00000008, alua=0000000b, alub=00000003, aluout=00000008
#    pc=1c, instr=0064202a, A1=03, A2=04, A3=04, RD1=0000000c, RD2=00000007, WD3=00000000, alua=0000000c, alub=00000007, aluout=00000000
#    pc=20, instr=10800001, A1=04, A2=00, A3=00, RD1=00000000, RD2=00000000, WD3=00000000, alua=00000000, alub=00000000, aluout=00000000
#    pc=28, instr=00e2202a, A1=07, A2=02, A3=04, RD1=00000003, RD2=00000005, WD3=00000001, alua=00000003, alub=00000005, aluout=00000001
#    pc=2c, instr=00853820, A1=04, A2=05, A3=07, RD1=00000001, RD2=0000000b, WD3=0000000c, alua=00000001, alub=0000000b, aluout=0000000c
#    pc=30, instr=00e23822, A1=07, A2=02, A3=07, RD1=0000000c, RD2=00000005, WD3=00000007, alua=0000000c, alub=00000005, aluout=00000007
#    pc=34, instr=ac670044, A1=03, A2=07, A3=07, RD1=0000000c, RD2=00000007, WD3=00000050, alua=0000000c, alub=00000044, aluout=00000050
#    pc=38, instr=8c020050, A1=00, A2=02, A3=02, RD1=00000000, RD2=00000005, WD3=00000007, alua=00000000, alub=00000050, aluout=00000050
#    pc=3c, instr=08000011, A1=00, A2=00, A3=00, RD1=00000000, RD2=00000000, WD3=00000000, alua=00000000, alub=00000000, aluout=00000000
#    pc=44, instr=00024280, A1=00, A2=02, A3=08, RD1=00000000, RD2=00000007, WD3=00001c00, alua=00000000, alub=00000007, aluout=00001c00
#    pc=48, instr=21070100, A1=08, A2=07, A3=07, RD1=00001c00, RD2=00000007, WD3=00001d00, alua=00001c00, alub=00000100, aluout=00001d00
#    pc=4c, instr=00073982, A1=00, A2=07, A3=07, RD1=00000000, RD2=00001d00, WD3=00000074, alua=00000000, alub=00001d00, aluout=00000074
#    pc=50, instr=20090500, A1=00, A2=09, A3=09, RD1=00000000, RD2=xxxxxxxx, WD3=00000500, alua=00000000, alub=00000500, aluout=00000500
#    pc=54, instr=00095022, A1=00, A2=09, A3=0a, RD1=00000000, RD2=xxxxxxxx, WD3=xxxxxxxx, alua=00000000, alub=xxxxxxxx, aluout=xxxxxxxx
#    pc=58, instr=000a59c3, A1=00, A2=0a, A3=0b, RD1=00000000, RD2=fffffb00, WD3=fffffff6, alua=00000000, alub=fffffb00, aluout=fffffff6
#    pc=5c, instr=000b6022, A1=00, A2=0b, A3=0c, RD1=00000000, RD2=fffffff6, WD3=0000000a, alua=00000000, alub=fffffff6, aluout=0000000a
#    pc=60, instr=00ec1020, A1=07, A2=0c, A3=02, RD1=00000074, RD2=0000000a, WD3=0000007e, alua=00000074, alub=0000000a, aluout=0000007e
#    pc=64, instr=ac020054, A1=00, A2=02, A3=02, RD1=00000000, RD2=0000007e, WD3=00000054, alua=00000000, alub=00000054, aluout=00000054
# Simulation succeeded
# Break in Module testbench_verilog at D:/UMS/the_mic/testbench_verilog.sv line 88
VSIM 3>
```

Figure 8. ModelSim output. Unresolved signals are indicated.

Figure 9. Output of the on-chip debugging technique.

5. CONCLUSION

This paper proposed a hardware-oriented approach of debugging a chip design in FPGA, by way of tapping signals from any level in the hierarchy up to the top level. These signals of interest were then displayed using VGA module provided by the board. The tapping was done using a the VHDL 'record' type as a bus with which signals are bundled and transported up the hierarchy.

A microprocessor design challenge was used as the test case. The proposed method correctly displayed the internal signals. The approach naturally showed higher fidelity compared to simulation. While software simulation of hardware design is indispensable and will continue to get better, hardware-level debugger and reporting module is invaluable and some times the only option. This is even more true when the chip is already in the deployment stage. We hope that this paper will inspire other researchers and students alike to employ the same technique in their designs.

ACKNOWLEDGMENT

This research is partly funded by Universitas Muhammadiyah Surakarta's Doctoral Research Grant. The author would like to thank his students in *Programmable Logic Design* and *Computer Architecture* classes.

REFERENCES

[1] C. Kellett, "A project-based learning approach to programmable logic design and computer architecture," *Education, IEEE Transactions on*, vol. 55, no. 3, pp. 378–383, 2012.

[2] J. H. Lee, S. E. Lee, H.-C. Yu, and T. Suh, "Pipelined CPU design with FPGA in teaching computer architecture," *Education, IEEE Transactions on*, vol. 55, no. 3, pp. 341–348, 2012.

[3] W. Richard, D. Taylor, and D. Zar, "A capstone computer engineering design course," *Education, IEEE Transactions on*, vol. 42, no. 4, pp. 288–294, 1999.

[4] F. Suryawan, "A project-based approach to fpga-aided teaching of digital systems," in *4th International Conference on Electrical Engineering, Computer Science and Informatics (EECSI)*. IEEE, 2017, pp. 590–595.

[5] A. Fidalgo, M. G. Gericota, G. R. Alves, and J. M. Ferreira, "Real-time fault injection using enhanced on-chip debug infrastructures," *Microprocessors and Microsystems*, vol. 35, no. 4, pp. 441–452, 2011.

[6] M. Portela-Garcia, C. Lopez-Ongil, M. García-Valderas, and L. Entrena, "Fault injection in modern microprocessors using on-chip debugging infrastructures," *IEEE Transactions on Dependable and Secure Computing*, vol. 8, no. 2, pp. 308–314, 2011.

[7] K. D. Maier, "On-chip debug support for embedded systems-on-chip," in *Circuits and Systems, 2003. ISCAS'03. Proceedings of the 2003 International Symposium on*, vol. 5. IEEE, 2003, pp. V–V.

[8] H. Park, J. Xu, J. Park, J.-H. Ji, and G. Woo, "Design of on-chip debug system for embedded processor," in *SoC Design Conference, 2008. ISOCC'08. International*, vol. 3. IEEE, 2008, pp. III–11 – III–12.

[9] P. Fezzardi, M. Lattuada, and F. Ferrandi, "Using efficient path profiling to optimize memory consumption of on-chip debugging for high-level synthesis," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 5s, p. 149, 2017.

[10] A.-S. Jamal, J. Goeders, and S. J. E. Wilton, "An FPGA overlay architecture supporting rapid implementation of functional changes during on-chip debug," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2018.

[11] A.-S. Jamal, "An FPGA overlay architecture supporting software-like compile times during on-chip debug of high-level synthesis designs," Ph.D. dissertation, University of British Columbia, 2018.

[12] P. Mishra and F. Farahmandi, *Post-Silicon Validation and Debug*. Cham, Switzerland: Springer, 2019.

[13] H. Oh, T. Han, I. Choi, and S. Kang, "An on-chip error detection method to reduce the post-silicon debug time," *IEEE Transactions on Computers*, vol. 66, no. 1, pp. 38–44, Jan 2017.

[14] H. Oh, I. Choi, and S. Kang, "DRAM-based error detection method to reduce the post-silicon debug time for multiple identical cores," *IEEE Transactions on Computers*, vol. 66, no. 9, pp. 1504–1517, Sep. 2017.

[15] Y. Cao, H. Palombo, S. Ray, and H. Zheng, "Enhancing observability for post-silicon debug with on-chip communication monitors," in *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, July 2018, pp. 602–607.

[16] S. Ray, "Soc instrumentations: Pre-silicon preparation for post-silicon readiness," in *Post-Silicon Validation and Debug*. Springer, 2019, pp. 19–32.

[17] R. Abdel-Khalek and V. Bertacco, "Post-silicon platform for the functional diagnosis and debug of networks-on-chip," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 3s, p. 112, 2014.

[18] M. Abramovici, "In-system silicon validation and debug," *IEEE Design Test of Computers*, vol. 25, no. 3, pp. 216–223, May 2008.

[19] D. Holanda Noronha, R. Zhao, J. Goeders, W. Luk, and S. J. E. Wilton, "On-chip fpga debug instrumentation for machine learning applications," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2019, pp. 110–115.

[20] K. Rahmani and P. Mishra, "Feature-based signal selection for post-silicon debug using machine learning," *IEEE Transactions on Emerging Topics in Computing*, pp. 1–1, 2017.

[21] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: the Hardware/Software Interface*, 5th ed. Morgan Kaufmann, 2014.

[22] J. L. Hennessy and D. A. Patterson, *Computer Architecture: a Quantitative Approach*, 5th ed. Morgan Kaufmann, 2012.

[23] D. M. Harris and S. L. Harris, *Digital Design and Computer Architecture*, 2nd ed. Morgan Kaufmann, 2013.

[24] P. P. Chu, *FPGA Prototyping by VHDL Examples*. Wiley, 2008.

[25] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: the Hardware/Software Interface*, 3rd ed. Morgan Kaufmann, 2005.