LAB 4: Debugging with VHDL

ECE 368

Dr. Paul J. Fortier

Team 5

This lab is verified to be the work of

Robert Mushrall

_____

Timothy Doucette Jr.

_____

Christopher Parks

_____

Date Submitted: 11 March 2016

Date Corrected:

Corrected by:

Table of Contents

List of Figures

Abstract

This lab served as an introduction to various applications of the VHSIC Hardware Description Language (VHDL) for hardware synthesis using a Nexys 2 FPGA. Code for a debug unit, ALU, and VGA display module were provided in order to develop a better understanding of debugging concepts using VHDL. There were a few errors in the debug unit that required the executors of the lab to hunt out and fix before flashing the project to the Nexys board. After that was finished, the code for a properly working debug unit was flashed to the Nexys board and demonstrated to the TA.

Introduction

This lab served as a deeper introduction of the VHDL hardware synthesis language by exploring the concepts and applications of debugging and debug units, as well as the use of VGA as a method of debugging. This was done by displaying debug data on a VGA enabled monitor. Students executing the lab were required to modify the VHDL code to make the debug data readable to the user.

Building a Debug: Methods

This portion of the experiment was dedicated to the construction of a debug unit for the ALU used in a previous lab. Once the example code was added to the project it was required that code used in previous labs needed to be added to the project as well, which can be observed in Figure 1.

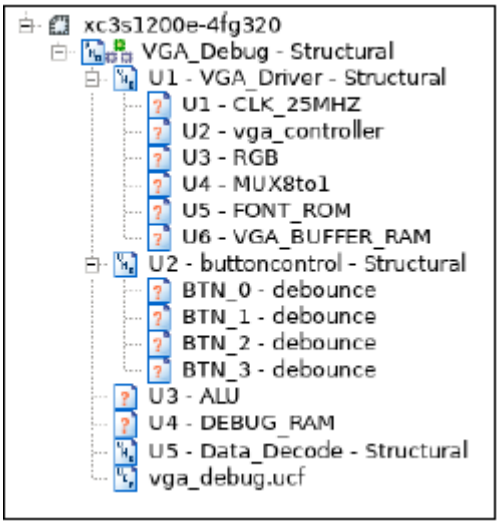

Figure 1: Required Source Code

The source code used to resolve this issue included the following files:

- alu_arithmetic_unit.vhd
- alu_logic_unit.vhd
- alu_mux.vhd
- alu_shift_unit.vhd
- alu_tb.vhd
- alu_toplevel.vhd
- front_rom_ascii.vhd
- load_store_unit.vhd
- mux8to1.vhd
- rgb.vhd
- clk25MHz.vhd
- debounce.vhd

RAM also needed to be generated for the VGA buffer and for the debugger. The Xilinx Core Generator tool was used to generate RAM for the VGA buffer and the debugger similar to how it was generated in the previous lab. A block diagram was then created to develop a better understanding of the functionality of the debug unit and can be observed in the appendix.

Once the required files were added to the project the Nexsys was then flashed and tested to test proper functionality. The switches were used as an input for a hexadecimal value that would then be translated into an ascii character that would be outputted to the screen. The buttons were used to trigger flags in the code to activate various functions.

- **RUN_FLAG**: This flag shows that the Debug process is running, creating the debug data that will be outputted to the monitor.
- **DD_FLAG**: This is the Data Dump flag. The purpose of this flag is to output the values being stored in the ALU to the screen for debugging purposes. These values include RA, RB, the Opcode, the CCR, and the ALU_OUT.
- **VGACLR_FLAG**: This flag clears the data in the VGA buffer and also takes the ascii value, translated from the hexadecimal value given by the switches, and fills the monitor screen with the character correlating to that value.
- **RST**: This flag triggers a reset. Once BTN3 is pressed this flag is set to a ‘1’ which restores all values to their initial states. Typically this means everything is set back to ‘0’.

```
RUN_FLAG <= DBTN(0);
DD_FLAG <= DBTN(1);
VGACLR_FLAG <= DBTN(2);
RST <= DBTN(3);
```

Figure 2: Flags triggered by buttons

The initial state of the output can be seen in Figure 3 below. The results of the tests of the unedited code can be observed in the results section below.

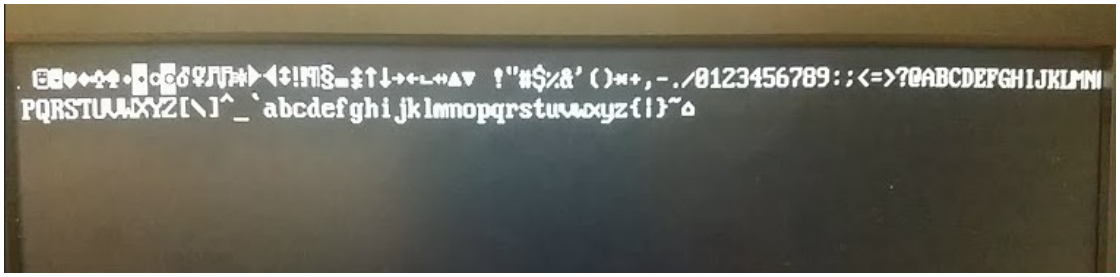

Figure 3: Initial output to monitor upon connecting VGA

When testing the data dump the data values appeared to be clumped together, as seen in Figure 4. In order to neaten up the data output it was required edit the source code to incorporate spaces to the output in order to distinguish which values were related to which debug data.

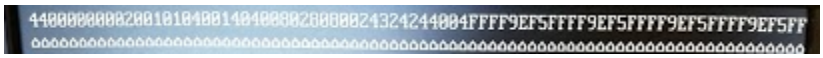

Figure 4: Data Dump (untidy)

In order to generate the hardware needed to include the spaces a multiplexer needed to be added to the RTL and a new SPACE state needed to be added to the debug process finite state machine. Figure 5 and Figure 6 below highlight the changes made to the code in order to incorporate the space multiplexer and the SPACE state to the finite state machine. The additions made to the code are highlighted.

```
--Dump Data from debug buffer
DATADUMP: PROCESS(DD_FLAG,CLK)
BEGIN
    IF(RST = '1') THEN
        DD_STATE <= INIT;
    ELSIF(RISING_EDGE(CLK)) THEN
        CASE DD_STATE IS
            WHEN INIT =>
                DD_ADR <= (OTHERS => '0');
                DD_WE <= '0';
                DD_STATE <= READY;
                DD_SPACE_COMPLETE <= '0';

            WHEN READY =>
                IF(DD_FLAG = '1') THEN
                    DD_WE <= '1';
                    DD_STATE <= RUN;
                END IF;

            WHEN RUN =>
                if (DD_ADR = x"4F") then --4F = 128 => limit of DEBUG
                    DD_ADR <= DD_ADR + 1;
                    DD_WE <= '0';
                    DD_STATE <= COMPLETE;
                else
                    if(DD_ADR_8 = "111") THEN
                        if(DD_SPACE_COMPLETE = '1') THEN
                            DD_ADR <= DD_ADR + 1;
                            DD_SPACE_COMPLETE <= '0';
                        else
                            DD_SPACE_COMPLETE <= '1';
                            DD_SPACE_MUX <= '1';
                            DD_STATE <= SPACE;
                        end if;
                    else
                        DD_ADR <= DD_ADR + 1;
                    end if;
                end if;
```

Figure 5: (PART 1) FSM for Data dump debug buffer and Space MUX.

```
            WHEN SPACE =>
                    DD_SPACE_MUX <= '0';
                    DD_STATE <= RUN;
            WHEN COMPLETE =>
                IF(DD_FLAG = '0') THEN
                    DD_COMPLETE <= '0';
                    DD_STATE <= INIT;
                ELSE
                    DD_COMPLETE <= '1';
                END IF;
            WHEN OTHERS =>
                DD_STATE <= INIT;
        END CASE;
    END IF;
END PROCESS DATADUMP;

DD_DATA <= DB_DATA;
WITH DD_SPACE_MUX SELECT
    DD_DATA  <= DB_DATA  WHEN '0',
                VGACLR_DATA   WHEN '1',
                DB_DATA  WHEN OTHERS;
```

Figure 6: (PART 2) FSM for Data dump debug buffer and Space MUX.

Further edits were made to the code because the original incrimination of the Data Dump Address (DD_ADR) didn't account for the hold caused by the SPACE state. In order to rectify this DD_ADR needed to increment more in order to compensate for the hold. After altering the increment of DD_ADR it was discovered that the data in the signal included the data from both VGACLR and the Debug data (DB_DATA). This resulted in a loss of data when outputting to the monitor. A new signal was created in order to separate VGACLR data and DB_DATA so that there would be no loss. The final results can be seen in the results section below.

Building a Debug: Results

After flashing the FPGA with the provided code, teams were tasked with exploring what the board did with the machine code. The switches were set to "10111011" and button 2 was pressed to produce the result in Figure 7 below.

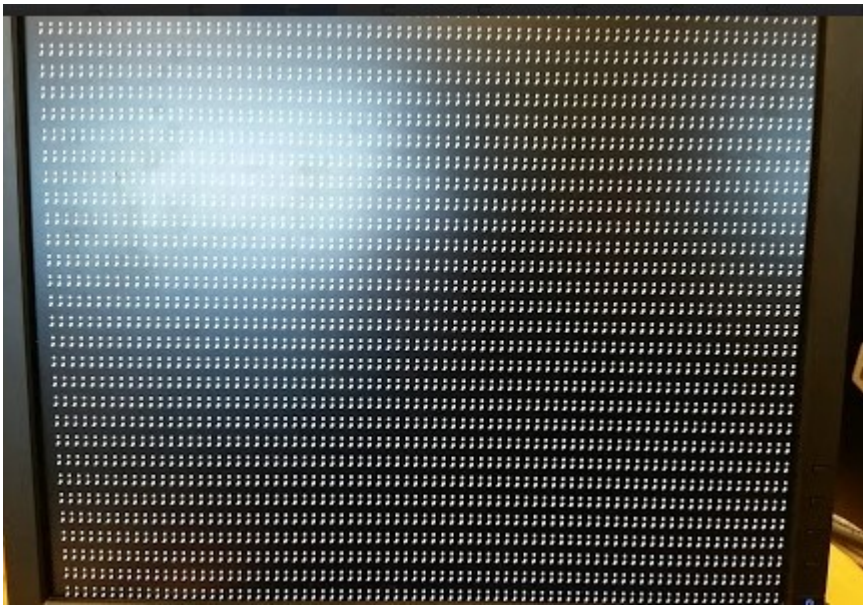

Figure 7: VGA Clear (Button 2) and Switches "10111011"

When the VGA was cleared with the switches set to "00000000" and button two was pressed, debug data was printed to the screen, as seen in Figure 8.

Figure 8: Initial Data Dump (Button 1)

After experimenting with the board more, other data was printed to the screen, with some value set as data for clearing the VGA, which can be seen in Figure 9.

Figure 9: Data Dump of Run flag after a VGA clear

After numerous failed attempts, the final result for the printing the debug data with spaces can be seen below in Figure 10. Notice that there are eight characters and then a space.

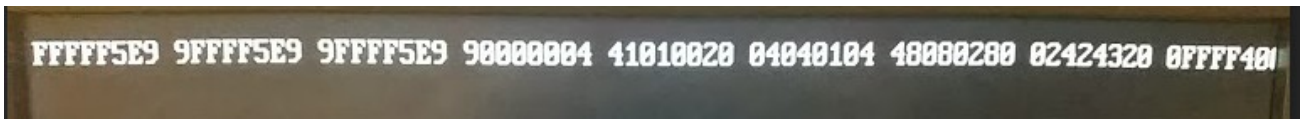

Figure 10: Cleaned up Data Dump

Discussion

This lab's goal was in part to provide a debugging system and in another part show how it works. By asking to insert spaces into the debug output, teams were tasked with picking apart the code, figuring out how the machine worked, then modifying it so that the debugging data was outputted with spaces inserted in order to clean up the data outputted. Multiple issues came from adding the spaces, such as the output signal being driven by multiple sources, issues with timing, and data being overwritten or skipped.

Conclusion

During lab, creating a block diagram of the machine proved to be extremely useful in figuring out how to add spaces to the output. Trying to modify a machine with no knowledge of how it worked proved to be a waste of time. A better approach is to try to learn how the machine works, making a block diagram, then try to modify the machine with knowledge of how it works. Using the option “Add Copy of Source” makes adding existing source documents incredibly easy and simple. In previous labs, problems came from trying to chase down all sources edited, as well as trying to update the source Github repository when sources were added from this repository and were edited. Currently, adding copy of source prevented future problems when moving the project and eliminated the possibility of attempting to edit outside sourced documents.

Laboratory Reflection

The biggest issue came from trying to get the lab done quickly, which caused the lab to take longer. Next time, the first step will be making sense of the machine and creating a block diagram. It is easiest to get an idea of the system by first looking looking at the components and how they are wired to various signals. After looking through the components and how they are wired, the complicated mess becomes simple devices wired together.

References

Lab 4 Handout (provided in lab)

Debug source code (provided in lab)

Appendix

See next pages