Thomas Boston

Robert Koenig

Professor Li

CPEG324

05/14/2025

Lab 4 - VHDL Calculator Implementation

Abstract:

In this lab, we will be using the 8-bit ISA we developed in the first lab, along with the calculator created in lab 3, to implement a calculator on an FPGA. The calculator utilized 8 input switches for instruction entry and push buttons for clocking operations and reset, supporting key instructions such as *load*, *add*, *swap*, and *print*. Output was displayed in decimal format on a 2-digit seven-segment display, enabling real-time visualization of computed values. The final system was tested on a Zybo Z7 board using Vivado, and demonstrated in a live presentation, highlighting both design methodology and functional performance.

Division of Labor:

Robert completed the VHDL code for the project, including debugging and ensuring its functionality on the ZYBO Z7 board. He also created the demo for the in class presentation. Thomas created the lab report and completed the appropriate work to insure the report was completed.

Detailed Strategy:

Our strategy for this project was to take the work completed throughout the previous labs and apply it in a practical sense so it would function properly on the ZYBO Z7 board. First, we made sure that the SSD schema code worked. We were able to source code from labs previously completed in CPEG202, with some modifications to include a minus sign to represent negative numbers. Here is our representation of the SSD standard logic:

```
with seg_in select seg <=
    "0111111" when "0000",
    "0000110" when "0001",
    "1011011" when "0010",
    "1001111" when "0011",
    "1100110" when "0100",
    "1101101" when "0101",
    "1111101" when "0110",
    "0000111" when "0111",
    "1111111" when "1000",
    "1101111" when "1001",
    "1000000" when "1111",
    "0000000" when others;
```

We can see the binary values of numbers 0 through 9, then “1111” to represent a negative sign, and anything else will represent a blank value. We then systematically went through the OP code truth table and translated the code from the logical representation into VHDL code, updating bits within the code to represent various bits from the RTL design, for example, we can see in the following screenshot that the opcode for the add command requires the write register to be “1”, the destination register to be “1”, the alu_op code to be “11”, the alu_src to be “0”, and the print_en to be “0”.

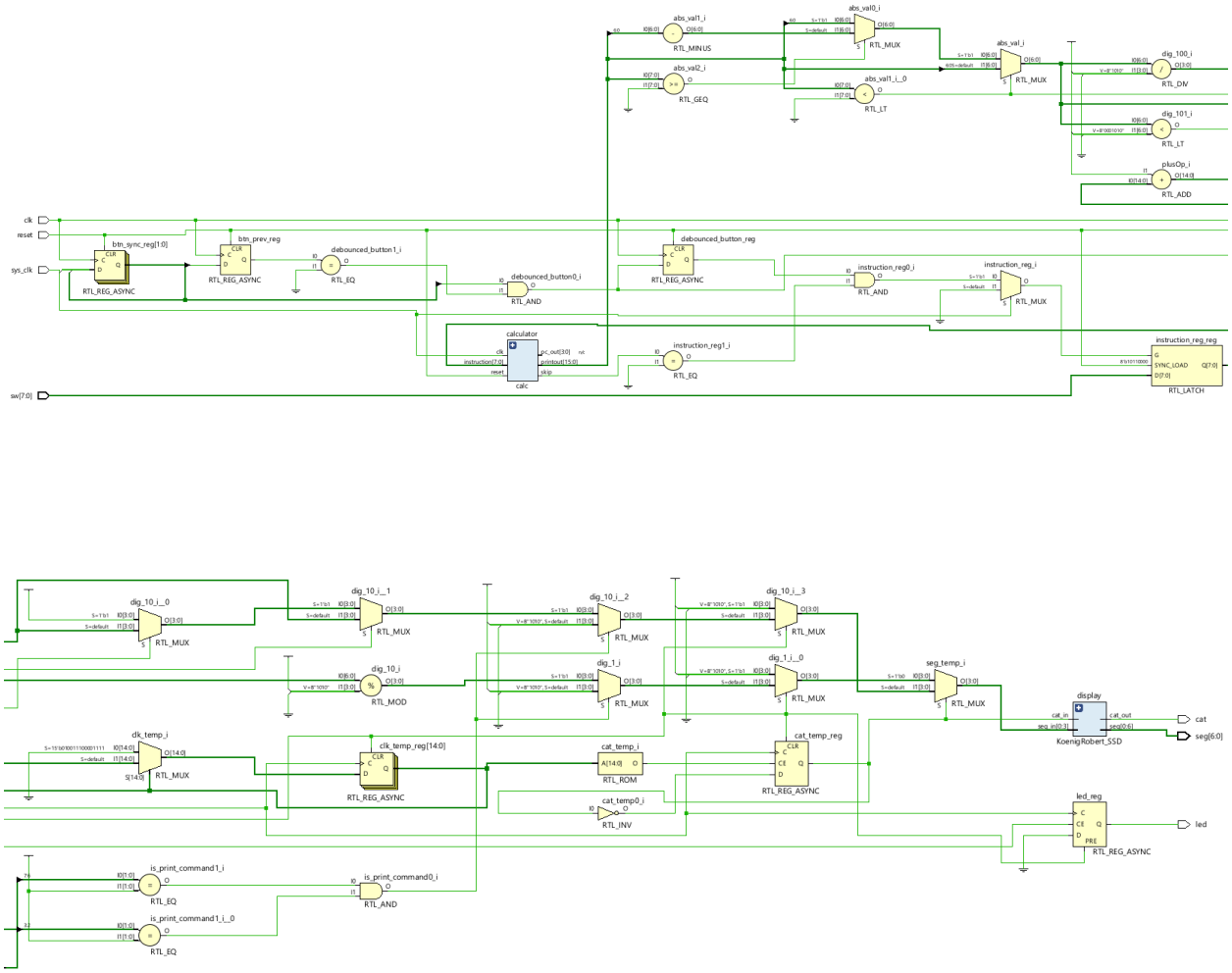
```
case opcode is
    when "00" => --add
        reg_wr <= '1';
        reg_dst <= '1';
        alu_op <= "11";
        alu_src <= '0';
        equ <= alu_equal;
        print_en <= '0';
```

This was repeated for every operation that the calculator must complete. We also had to include a section to sign extend the immediate value so that we can include appropriate 2s complement negative values:

```
--sign extention
sign_ext_imm(3 downto 0) <= imm;
sign_ext_gen: for i in 4 to 15 generate
    sign_ext_imm(i) <= imm(3);
end generate;
```

Given we only had a 2 digit SSD, we could not represent the entire output, so we must display the lower printable portion of our register output.

The next part of our strategy was to take the RTL level design that we completed in draw.io for the previous lab and implement that into VHDL. This involved ensuring all MUX, display ports, register components, calc components, ALU components, and any other components functions as they should and then using their logic in a practical sense in the translation into VHDL RTL design, which can be seen below.



Overall Implementation

We also had to implement the calculator component RTL level design into VHDL RTL level design.

Results:

The results gathered from this lab were successful. The RTL Level Design, which can be seen below, functioned correctly and was able to perform all operations correctly. The VHDL code works and the test bench created serves as a performance metric to ensure the code's correctness. The test bench checks for several edge cases that may occur and ensures that each outcome is appropriate. The only limitation we came across is the limited range of numbers we can load into the registers. Given we need a certain number of bits for OPcode and destinations, the load values are limited to numbers from -8 to 7. The Calculator component of the RTL level design functions correctly, as well as the overall design. Previous labs testbenches did not account for the use of the FPGA, switches, or SSD PMODs, so the testbench had to be revamped to include all of those additions.

Conclusion:

During the completion of this lab, we faced several challenges that made both the hardware and software uncooperative that caused us to have to redesign the hardware level FPGA several times. The issue of only having 2 digits to display was also quite annoying as it would have been easier and more usable if the entire register was able to be printed each time. A future iteration of this design would include a second SSD PMOD, or the addition of a new 4 digit PMOD so that the entire register could be displayed each time, rather than just the lower half. However, in spite of these issues the final code works as instructed and functions how the calculator is intended to.