CDA 4205L Lab #8: Single-Cycle CPU Implementation

University of South Florida

Lab Date: Mar. 5

Report due: One week after the lab section, see Canvas

***This is a group lab**. One submission per group is sufficient.*

This lab uses Vivado!

Welcome to CDA 4205L Lab #8! The goal of this lab is to demonstrate an example implementation of a single-cycle CPU in Verilog HDL.

Prelab

This lab builds on the ALU and Control Unit design from Lab #7. You will be provided a near-complete single-cycle RISC-V CPU implemented in Verilog HDL. This implementation is *almost* capable of running code generated by your assembler from Lab #3 and #4 – but one critical component is missing – the instruction memory.

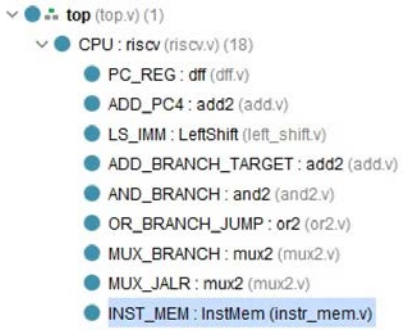
Creating an instruction memory In Verilog module is done by first declaring a generic memory variable and loading the assembler output (machine code) into it. The Verilog command `$readmemb(<file_name>, <memory_name>)` will read the data from `<file_name>` and load it into `<memory_name>`. This is for simulation purposes only, so you can simply place the command in an initial block. If your instructions are written in hex, the command `$readmemh()` will work the same way.

Lab

Setup

1. Download the Vivado project files from Canvas.

2. Open the file `instr_mem.v`. This file contains a template for a memory module.

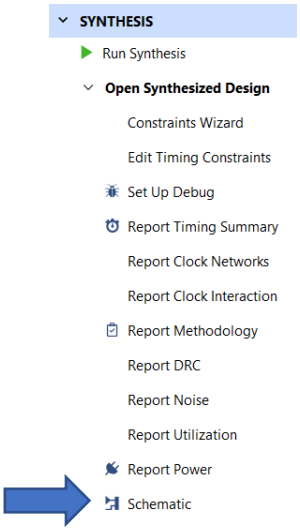


3. Declaring a memory is very similar to declaring a register (`reg`) variable. The main difference is that you must specify the *depth* as well as the *width*. In this case, there are already *parameters* as part of the module definition (`DATA_WIDTH`, `ADDRESS_WIDTH`, `NUM_MEM_CELLS`) that are pre-initialized. Note that, while these values are *defaults*, when you instantiate a new `InstMem` module in the top-level, you can override them to be whatever size you need. This lets you re-use the module in different designs without extra work. Declare a memory using the syntax `reg [DATA_WIDTH - 1 : 0] inst_mem[0 : NUM_MEM_CELLS - 1]`
4. Next, initialize the memory in an `initial` block. Use the `$readmemb()` directive. This command takes two parameters, the memory file name, and the memory variable you created in step 3. Note that referencing a macro definition requires prepending with `` ` ``.

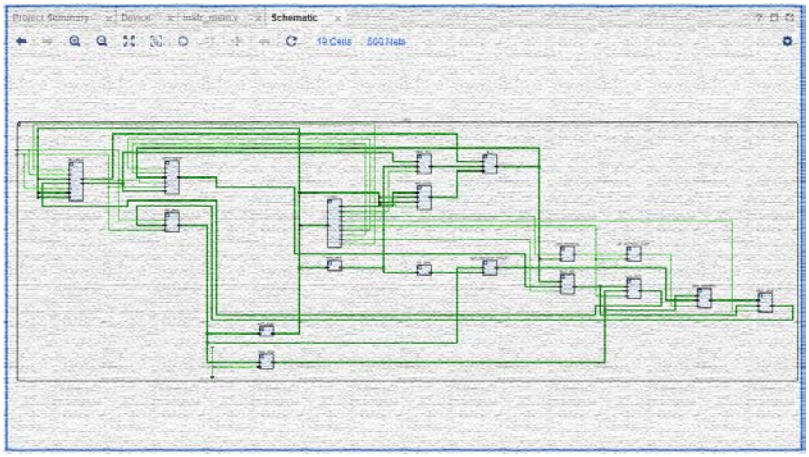
5. Finally, because this memory is part of a single-cycle implementation, data is **not** read *synchronously*. Hence, we do **not** use a clock; instead, it is a *continuous assignment* statement in the form `assign data = inst_mem[<address>]`, where *<address>* is an *n*-bit vector specifying where it should load the next instruction from.
   Note that we do not actually need the lower **two** bits of the address; hence, we can read `addr[ADDRESS_WIDTH - 1 : 2]` to give us *<address>*.

> T1: Why do we not need the lower two bits of the address when reading from the instruction memory?

6. Once you are ready, ensure **top.v** is set as the top-level module (it should appear in **bold** in the Sources window) and click *Run Synthesis*. Open the Synthesized design. Generate the Schematic from the synthesized design:

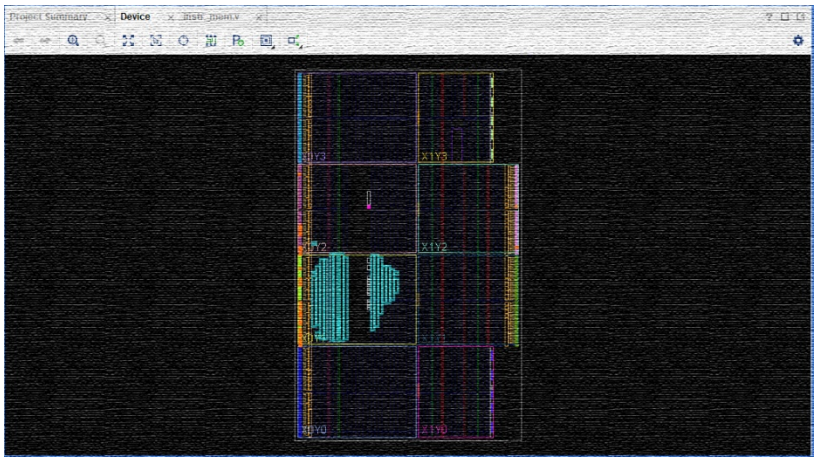


> T2: Take a screenshot of the block diagram generated by Vivado. Remember to click into the CPU block before taking the screenshot so it shows the actual design.



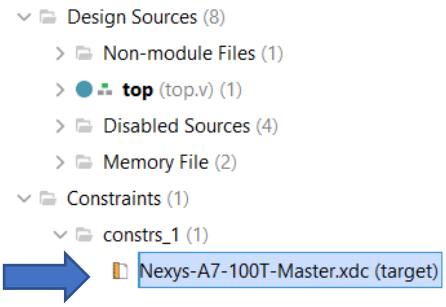
7. Click *Run Implementation*. Once complete, open the Implemented design. Once the design is open, click on the "Device" tab. You should see a high-level view of the FPGA. All the filled-in boxes are regions where the logic resources have been used to implement the design.

T3: Take a screenshot of the FPGA view in the Device tab.



T4: Report the utilization of logic resources (LUTs, FFs, DSPs) used by the design.

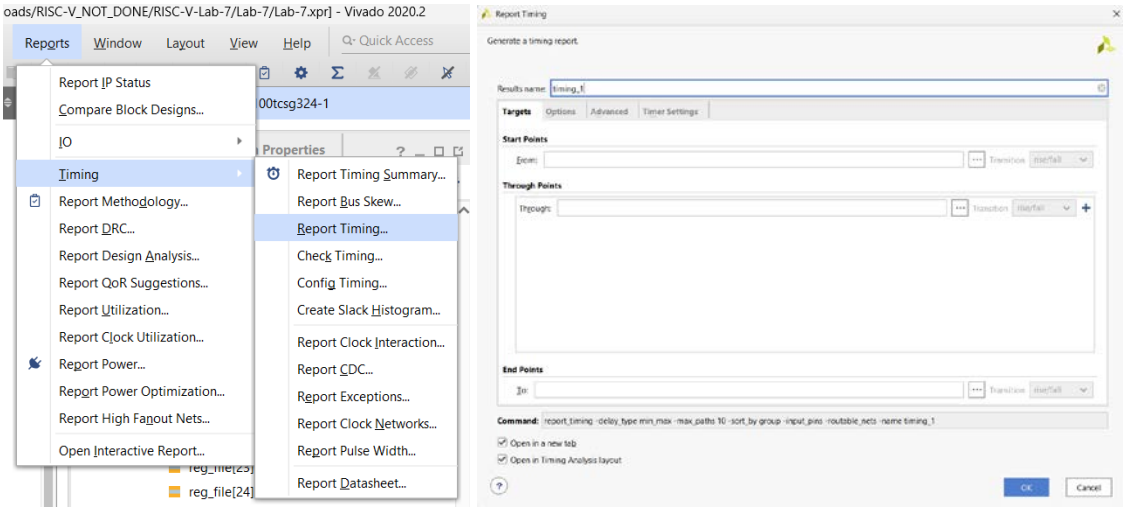
8. The constraints file defines certain properties of the FPGA and how physical connections/pins get mapped to the logic in the design. The file can be found here:

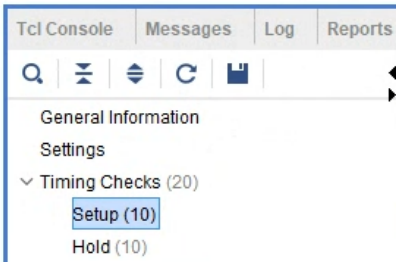


One important pin is the *clock* input. A clock is "created" or defined with the command `create_clock`. This defines several properties of the clock.

T5: What is the clock period/frequency for this design according to the `create_clock` command?

9. Vivado can generate a *timing report* which can be used to find the critical path in the design. Ensure the Implemented Design is open, then click Reports, Timing, Report Timing… Click OK on the window the appears.

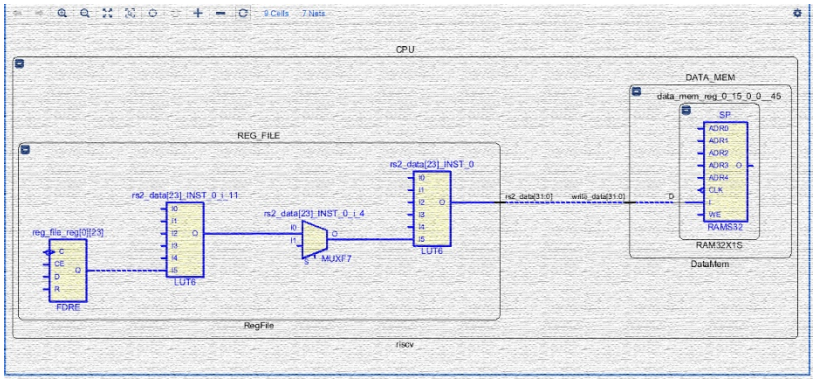




T6: What is the length (in nanoseconds) of the critical path? This will be the path with the highest Total Delay. Based on the clock period, how much "slack" is there in the timing? (That is, how much time is "leftover".)

10. Right-click the critical path entry, and select Schematic. This will bring up another window that highlights the path (in blue) and shows all the functional units through which the signal passes.

T7: Take a screenshot of the schematic showing the critical path. What instruction(s) would you expect to use this path?



The selected path schematic in this screenshot is intentionally inaccurate.

T8: The given implementation only supports full 32-bit instructions, but the RISC-V spec calls for "compressed instructions" as well. Compressed instructions provide a different encoding that stores common instructions in 16 bits instead of 32. How would our `InstMem` `module` change if we wanted to support compressed instructions?

T9: Why is a continuous assignment statement used for reading from the instruction memory? How would this differ if we were implementing a pipelined datapath?

In-class work - Complete **T1, T2, T3, T4**, **T5, T6, and T7,** show it to the TA for review and submit it on Canvas before the lab session ends. Submit your in-class work as a Word document (.docx) format.

Final Report - Submit your report pdf with answers to all questions and screenshots. Submit the instr_mem.v file as well, for a total of two files in a single .zip file which you submit. One submission per group.