Single Cycle RISC-V Design Project
(Due: November 1, 2024)

In this project, you are required to design a single-cycle RISC-V processor. To make this project manageable, it will be split into smaller components, where you implement increasing sets of RISC-V instructions by certain due dates. You will need to submit only one project report during the last week of the semester. However, you will need to demonstrate progress by the check point.

- Make sure to go through all the guidelines at the end of this assignment before working on it.
- **Software:** You will use the FPGA design, simulation and synthesis software in this project. This project will be simulation based. There are two software options: (1) If you are familiar with Xilinx Vivado software, it is available in all ECE lab computers. You can also download the Vivado ML Standard Edition for free. Note that the software installation may take a while and require a large disk space. (2) If you are familiar with the Altera Quartus software, you can download and install Questa*-Intel® FPGA Starter Edition which is previously called ModelSim.
- **Simulation**: You are not required to implement the designed processor on an actual FPGA board, but it is required to pass the testbench that simulates the execution of the RISC-V instruction.

Part (a):

Implement processor with the following instructions: **`ADD, ADDI, SW, LW, ADD, SUB, AND, OR, MUL, SLLI.`** Validate the design by simulating the processor to execute the following two programs. Note that you can use the first program to test your processor after implementing the first three instructions (ADDI, ADD, SW). It is a good idea to create ModelSim testbenches and verify your code is working. You can then continue expanding the instruction set and eventually test with the second program. Read the memory contents after execution is finished. Contrast the memory contents with the results you would get from the RISC-V simulator. (Let's use the opcode 11111111 for HALT and you can get the machine code of these programs using the Venus tool: https://venus.kvakil.me/)

Program 1:

```
addi   x1, x0, 0
addi   x2, x0, 16
addi   x3, x0, 100
addi   x4, x0, 8
add    x5, x1, x2
add    x6, x3, x4
sw     x5, 0(x1)
sw     x6, 4(x2)
HALT
```

Program 2:

```
addi   t0, x0, 8
addi   t1, x0, 15
```

```
        sw      t1, 0(t0)
        add     t2, t1, t0
        sub     t3, t1, t0
        mul     s1, t2, t3
        addi    t0, t0, 4
        lw      s2, -4(t0)
        sub     s2, s1, s2
        slli    s2, s2, 2
        sw      s2, 0(t0)
        halt
```

Part (b):

Implement processor with the following instructions: **`add, addi, sw, lw, add, sub, and, or, slli, mul, beq, bne, jal, jalr`**

Validate the design by booting the processor and running the following code which calculates the factorial code. Make sure to initialize the SP to the end of your memory. Read the memory contents after execution is finished. Contrast the memory contents with the results you get from the RISC-V simulator.

```
        addi    a0, x0, 12
        jal     ra, fact
        sw      a0, 0(x0)
        halt
fact:
        addi    sp, sp, -8
        sw      ra, 4(sp)
        sw      a0, 0(sp)
        addi    a0, a0, -1
        bne     a0, x0, else
        addi    a0, x0, 1
        addi    sp, sp, 8
        jalr    x0, 0(ra)
else:
        jal     ra, fact
        addi    t0, a0,0
        lw      a0, 0(sp)
        lw      ra, 4(sp)
        addi    sp, sp, 8
        mul     a0, a0, t0
        jalr    x0, 0(ra)
```

In addition to the factorial program, please validate your design using another meaningful piece of code that include the direct use of a conditional branch (e.g., BNE or BEQ).

Your final design of the single-cycle processor and the project report should be uploaded to Canvas by November 1, 2024.

Write a final project report using part (b) as the reference for the following requirements:

For your final working version of your design, please make sure to remove any debugging circuitry (e.g., for memory-content editor or signal tap) to reduce your footprint and improve your timing. You might also like to explore the Quartus tool settings, which can adjust the strength of circuit optimization and trade-off design area with timing.

1. Include the assembly and machine code of the factorial program and your program of choice in the documentation.
2. Your Vivado/ModelSim testbenches and their outputs.
3. Print screenshots that show the memory contents after executing the factorial program and your program of choice.
4. Synthesize your design using a target FPGA. Find the actual maximum frequency. If the resulted frequency is less than 20MHz, there will be 10% penalty on project grading. Note that your critical path will likely involve the multiply instruction, so to crank your frequency up, you want to streamline the data path components involved in the multiply.
5. Report the FPGA resources being used by your processor: CLBs or LEs, embedded multipliers, memory blocks, and register resources. Make sure to remove any debugging logic if you used it.

Guidelines for Creation of Instruction and Data Memories:

Ideally the instruction memory should be built out of the ROM component. Unfortunately, the ROM component in the FPGA cannot be read combinationally; i.e., the output will not update its value until the positive edge of the clock comes in. Because the ROM address is supplied directly from the PC register, it will not be possible to update the PC and fetch the instruction in the same cycle. To avoid this problem, there are three possibilities:

- create a ROM directly using an array of 32-bit registers in your code and initialize the registers within your code using the `initial` statement; or
- use a ROM component and introduce a phase shift in the clock between the PC and the ROM. This shift should be ideally small; or
- Use the ROM component and use the register inside of it as your PC.

For the data memory, you should initialize the RAM blocks using the IP Catalog. Make your data memory size 1024 bytes or 256 words. Make sure the output port is not registered; however, there is no way to avoid that the inputs are not registered (same problem as in ROM). To fix the situation in this case, I suggest clocking the data RAM with the falling edge of the clock. This will give the processor half a cycle (or more) to fetch and execute the instruction, and another half a cycle (or less) to access the data memory and update its contents into register. That can allow you to bump up your design frequency sometimes.