
# CS 240 – Computer Architecture

## Fall 2024

## Lab 5:

## RISC-V Core

## Scope of the Lab

In this assignment, you will implement the behavioral logic of a RISC-V core module using Verilog. The RISC-V core will fetch RISC-V instructions from memory, decode them, execute them, and then write the calculated values to the memory. This design is a single-issue non-pipelined core, meaning only one instruction is processed at a given time.

To decode the instructions, you can use the RISC-V decoder you implemented in the previous lab.

The given memory module has only one read/write port, meaning you can read or write only one address at any given time. The memory also includes a mask signal to write certain bits into memory.

## Resources

For this lab, we will continue to use EDA Playground. You can follow the instructions in the previous lab document if you haven't registered to EDA Playground yet. You are given the skeleton of the module implementation. You can reach the task from the following link.

Task: https://www.edaplayground.com/x/js_7

There are four design files included in this lab.

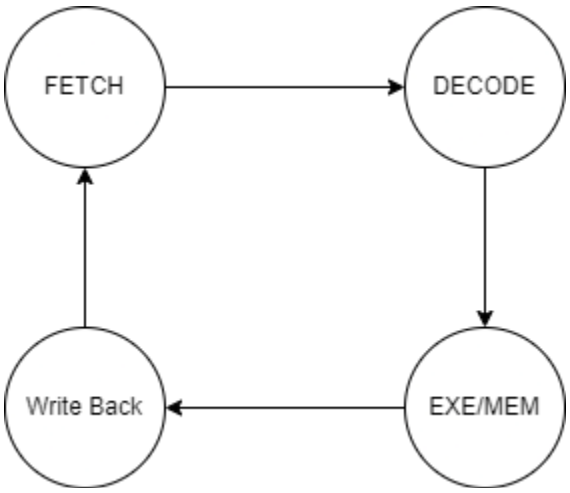
**design.sv:** Implements the top module that contains the RISC-V core and RAM modules.
**core.v:** Implements a basic RISC-V core. You will be working on this file.
**blram.v:** Implements a byte-indexed, single port block ram.
**program.v:** Includes the RISC-V assembly code encoded as hex and mapped to the RAM cells that will be loaded to the memory at the start of the simulation. The program loaded to the memory is Bitmap assembly code from https://eseo-tech.github.io/emulsiV/. You can use the online simulator to compare your core.

**You also need to add a decoder.v file** to your playground to decode the instructions. You can copy the decoder module that you implemented in the previous lab in this file.

## Exercise

In this lab, you will implement the behavior of a RISC-V Core module. Your module will have 4 states; FETCH_STATE, DECODE_STATE, EXMEM_STATE, and WB_STATE.

You will implement the following instructions,
**ALU Instructions:** ADD, ADDI, SUB, LUI, AND, ANDI, OR, ORI, XOR, XORI, SLL, SLLI, SRL, SRLI, SRA, SRAI, SLT, SLTI, MUL
**Load/Store Instructions:** LW, LB, LH, SW, SB, SH
**Control Flow Instructions:** BEQ, BNE, BLT, BGE, JAL, JALR




In the **FETCH_STATE**, you will fetch the instruction at the program counter from memory.

In the **DECODE_STATE**, you will decode the instruction from hex to find the opcode, registers, funct3, and immediate values of the given instruction. For this state, you can use the RISC-V decoder that you implemented in the previous lab.

In the **EXMEM_STATE**, you will implement the execution logic of the instruction that you decoded. You will use the result_next signal to pass the ALU result into the related register. You should also set regfileWr_next signal to 1 in order to enable write to register. For writing to memory you should set wrEn signal to 1 . This state is a merged version of EXECUTE and MEMORY stages. You should also start handling any LOAD and STORE instructions here.

In the **WB_STATE**, for the purposes of this design, this state is only required to write the value that is coming from the memory for LOAD instructions.