Introduction to Processor Architecture

Project Phase I: Sequential RISC-V Processor Design

Spring 2026

Due Date: 21 Feb

1 Abstract

Congratulations on successfully implementing the ALU. This document specifies the design, implementation, and verification plan for a sequential (non-pipelined) RISC-V processor implemented in Verilog and simulated with iVerilog. The processor implements a chosen subset of the **RV64I** base integer instruction set and executes instructions in a single instruction-cycle flow. Pipelining, hazard detection/forwarding, and related topics will be implemented in the next stage of the project. Figure 1 outlines the sequential datapath/processor architecture. Please refer to the textbook for a detailed explanation.

The project submission must include the following:

- A report describing the design details of the various stages of the processor architecture, the supported features (including simulation snapshots of the features supported) and the challenges encountered.
- Verilog design and testbench codes for the processor.

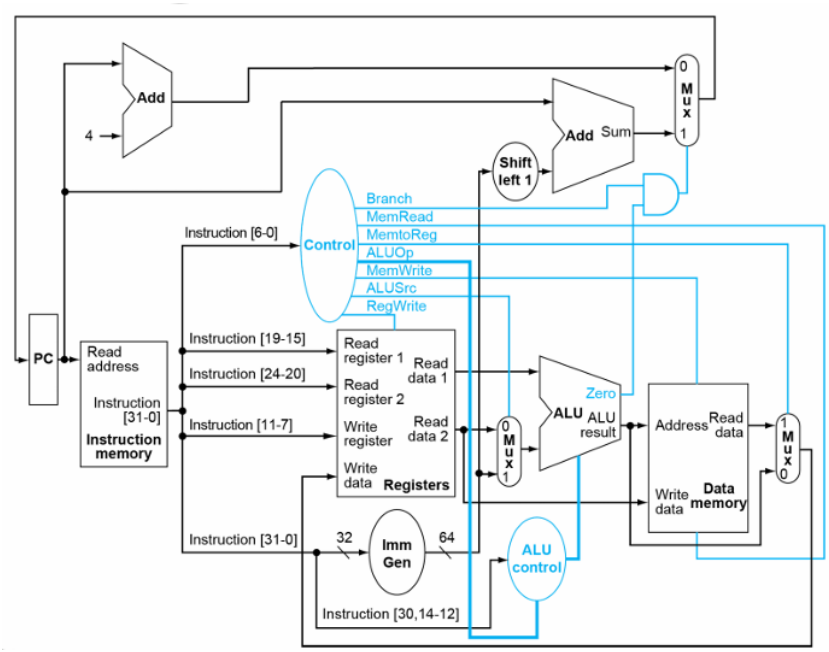

Figure 1: Sequential datapath. *From Ch.4, Computer Organization and Design by D. Patterson and J. Hennessey*

2 Specifications

The required specifications in the processor design are as follows:

- **PC** :

  INPUTS: `clk, reset, pc_in`

  OUTPUTS: `pc_out`

  The PC is a 64-bit register that stores the address of the current instruction. The PC is incremented by 4 (32-bit instructions) after every instruction, unless there is a branch, in which it is updated according to the branch target address. The PC is updated with the value of `pc_in` every clock cycle, so it does not need an exclusive write signal.

- **Register File** :

  INPUTS: `clk, reset, read_reg1, read_reg2, write_reg, write_data, reg_write_en`

  OUTPUTS: `read_data1, read_data2`

  Declare **32 registers, each of them being 64-bit wide**. Also, make sure that the register contents are printed out into a `registers.txt` file which is formatted exactly as the sample file. This must be strictly followed as automated tests will be ran.

- **Instruction Memory** :

  INPUTS: `clk, reset, addr`

  OUTPUTS: `instr`

  Have a file for e.g. `instructions.txt` in the same directory, which contains byte addressed hexadecimal instructions (1 byte of instruction per line). Thus, every four lines together constitute one complete instruction. Include this in the verilog design file (for e.g. `instruction_mem.v`) for the instruction memory block, parse the `instructions.txt` file and fill the instruction-memory array accordingly. The 32-bit instruction can then be passed as the output. Use a macro, such as `IMEM_SIZE` to set the size of the instruction memory block. A value of `4096 bytes` or higher is sufficient.

- **Control Unit** :

  INPUTS: `opcode`
  OUTPUTS: `Branch, MemRead, MemtoReg, ALUOp[1:0], MemWrite, ALUSrc, RegWrite`

  This block reads the opcode and generates the control signals for the rest of the processor. The `ALUOp` (2 bit signal) identifies the instruction type (arithmetic, logical or branch instruction) and is sent to the `ALU_Control` unit.

- **Immediate Generation**
  INPUTS: Instruction (32-bit)
  OUTPUTS: 64-bit signed extended immediate value

  The Immediate Generation block extracts the immediate field from the 32-bit instruction and sign-extends it to 64 bits. The immediate format is determined by the instruction type (I-type, S-type, or B-type), as specified in the RISC-V ISA. The generated immediate value is placed on a 64-bit bus and is used for ALU operations, branch target address computation, and memory address calculation.

- **ALU Control**
  `INPUTS:` 2-bit `ALUOp` signal
  `OUTPUTS:` 4-bit `ALUControl`

  The ALU Control Block generates a control signal which is required for selecting the operation performed by the ALU.

- **Arithmetic Logic Unit (ALU)**
  `INPUTS:` `input1`, `input2`, `control_signal`
  `OUTPUTS:` `result`, `zero_flag`

  The ALU performs arithmetic and logical operations on the two 64-bit input operands based on the control signal generated by the ALU Control block. You are expected to use the same implementation as done in the assignment, ignoring unnecessary flags and operations.

- **Data Memory**
  `INPUTS:` `clk`, `reset`, `address`, `write_data`, `MemRead`, `MemWrite`
  `OUTPUTS:` `read_data`

  - The data memory is used to perform read and write operation. Here the input `address` specifies the memory location.
  - Set the size of the Data memory to be exactly `1024 bytes`. Also, make sure that the data memory contents are printed out into a `data_mem.txt` file, which is formatted exactly as the sample file. It must have a 64-bit data bus for reading the stored value, and it is byte-addressed.
  - When `MemWrite` is asserted, the value in `write_data` is written to the memory.
  - When `MemRead` is asserted,the data stored at the specified address is read and provided on `read_data`.
  - For the sake of simplicity, the data memory block has a latency of one clock cycle. i.e., once the inputs are asserted, the outputs arrive at the posedge of the next clock cycle.

- **Multiplexers**

  Multiple multiplexers are used within the datapath to select appropriate inputs based on control signals.

- **Adder Blocks**

  Adders are used to compute `PC + 4` for sequential instruction execution and to calculate the branch target address.

Important points to notice:
RegMem can read two registers and write to a register in the same clock cycle. Data-Memory, on the other hand, can only read or write in the same clock cycle.
Please follow the **Big-Endian** approach for **both** the RegMem and DataMem. Although the textbook follows Little-Endian, for the sake of simplicity, we are going ahead with Big-Endian. For more details, refer to Pg. 178 in the course textbook.
The above implementations must execute the following instructions from RISC-V ISA: `add`, `sub`, `addi`, `and`, `or`, `ld`, `sd` **and** `beq`.

3 Design Approach

The design approach should be modular, i.e., each stage has to be coded as a separate module and tested independently in order to help the integration without too many issues. Please adhere to the following naming conventions for input and output files, as automated tests will be run on the codebase.

- In the `instruction_mem.v`, read the byte addressed (each line contains 1 byte) hexadecimal instruction in Big-Endian format from a text file named - `instructions.txt`
- After the execution of the instructions, the testbench must write the 32 64-bit register values (hexadecimal), followed by the number of clock cycles (decimal) to a text file named `register_file.txt`
- A sample input and output, along with an assembly-level explanation, has been provided. It is not necessary to submit an assembly-level explanation for any testcases implemented. However, it is recommended to have it for better readability.
- Name your testbench as - `seq_tb.v`. This must execute the instructions at `instructions.txt` and generate the `register_file.txt` as the output.
- In our testing, we will replace the `instructions.txt` file with our testcases and evaluate the generated `register_file.txt` file after we execute - `iverilog seq_tb.v`.

Please stick to the above naming formats to avoid any misevaluations.

4 Suggestions for Design Verification

Please adhere to the following verification approaches as much as possible.

- You can individually test each stage/module for its intended functionality with module specific test inputs.
- Please write an assembly program for any algorithm (e.g., Fibonacci algorithm) using RISC-V ISA and the corresponding encoded instructions, and use the encoded instructions to test your integrated design.

5 Evaluation

The marks will be assigned as follows:

- Report: 5 marks
- Sequential design automated scripts: 5 marks
- Sequential design viva: 10 marks