Computer Organization Project

Professor: Hadi Shahriar Shahhoseini, Ph.D.

TAs: Faraz Ghoreishy and Arvin Delavari

School of Electrical Engineering, Iran University of Science and Technology

December, 2023

Introduction

The goal of this project is to study multi-cycle implementation of a RISC-V processor. This processor is going to execute a simple RISC-V assembly code which is the subject of the first part in the first assignment. Your processor will support instructions that are used in this code, which means you will have to consider multiply instructions as is needed in one of the calculation steps of this code.

This project has two sections each with their own separate deadlines. **There are directed and open-ended portions in this assignment and there will be a presentation, in which you will explain your different design methodologies. In the presentation you will face some technical challenges which will gain you additional bonus points.** You are free to work in groups of two or three. Each student will turn in their group's single report but you will show your design as a group at the presentation date.

Multi-Cycle Implementation

As you know we can break down each instruction into a series of steps corresponding to the functional unit operations that are needed. We can use these steps to create a multi-cycle implementation. In this architecture, each step will take 1 clock cycle. This allows that components in the design and different functional units to be used more than once per instruction, as long as it is used on different clock cycles. This sharing of resources can help reduce the amount of hardware required. This classic view of CPU design partitions the design of a processor into data path design and control design. Data path design focuses on the design of ALU and other functional units as well as accessing the registers and memory. Control path design focuses on the design of the state machines to decode instructions and generate the sequence of control signals necessary to appropriately manipulate the data path.

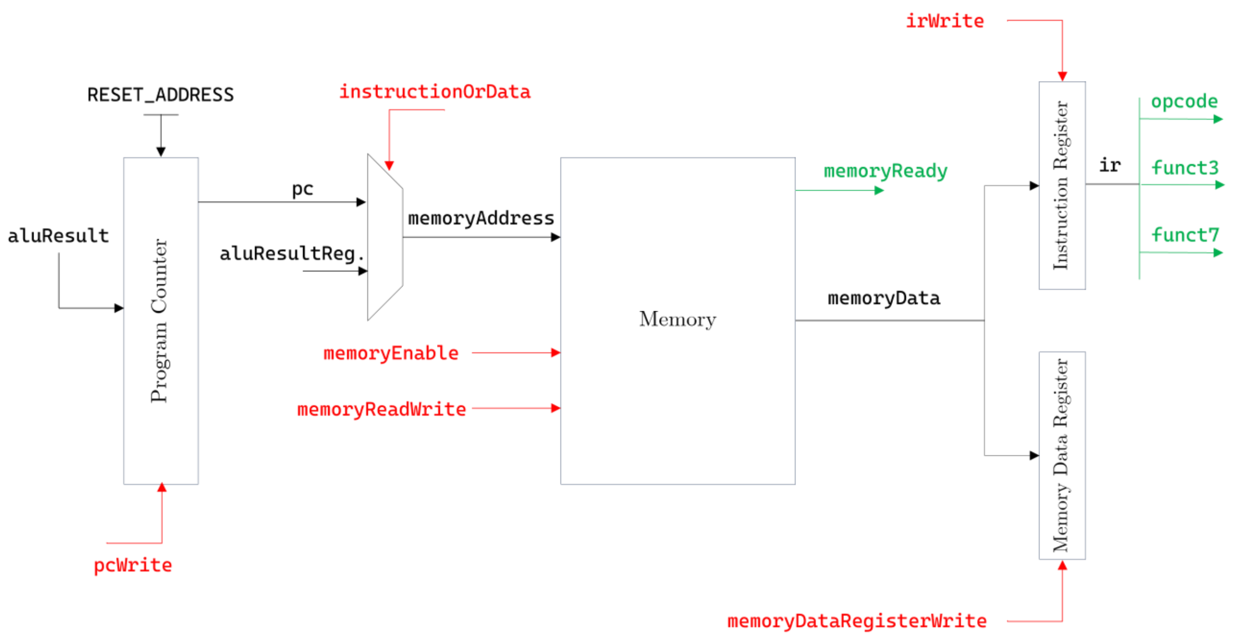

Figure 1 - First part of data path, fetch and decode

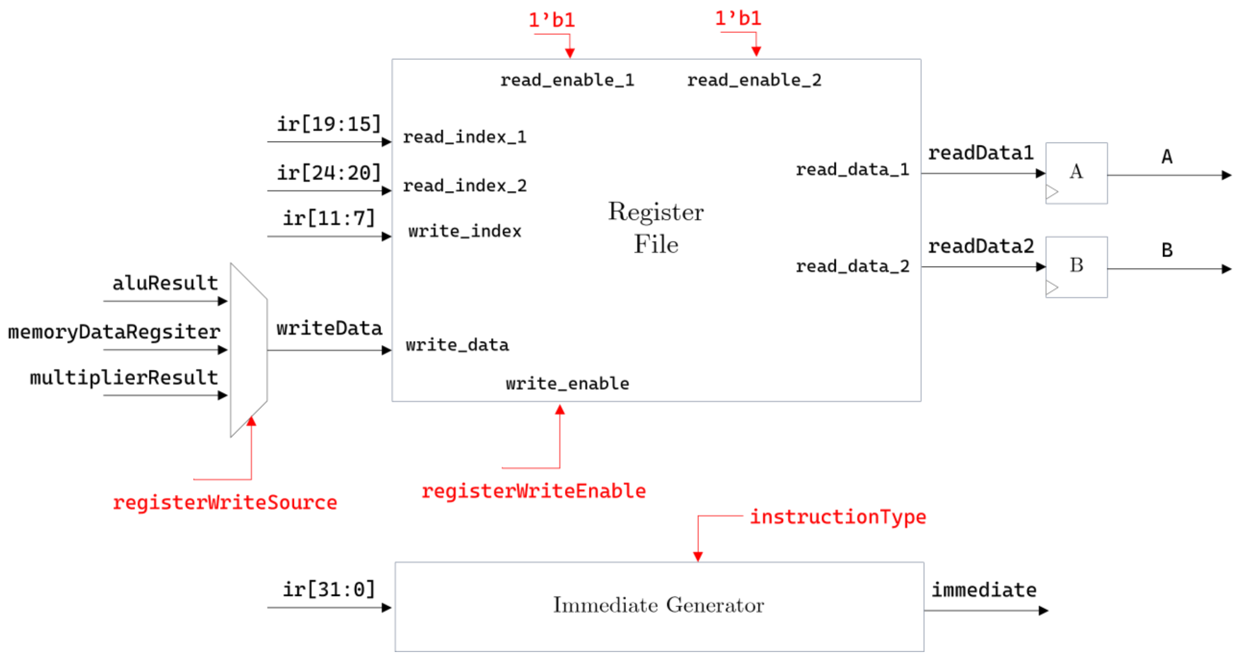

Figure 2 - Second part of data path, operand fetch and immediate generation

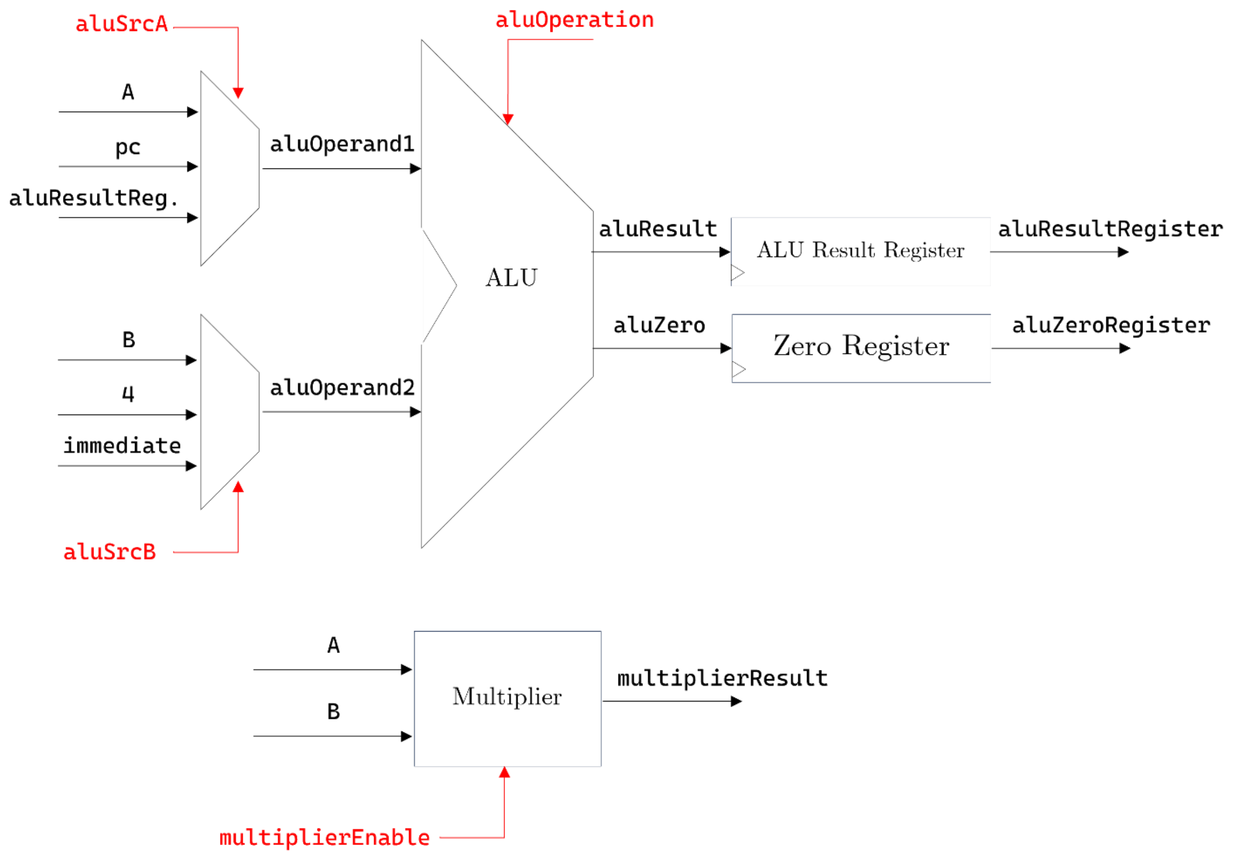

Figure 3 - Final part of data path, execution stage

The data path file written in Verilog is provided in RISCV_Core.v. The first few lines include the components of data path in the main code. The first section of **`RISCV_Core`** module is the controller FSM which we will discuss in the second assignment of this project. The data path provided in figures 1-3 acts as follows for completely executing an instruction; assuming that program counter register contains a valid value, first by setting a correct value to **`instructionOrData`** we will assure that we are reading from the instruction section of the memory. After a brief delay contributed to the memory access time, the signal **`memoryReady`** will rise to indicate that data from memory is ready to be read by the processor. At this moment **`irWrite`** will assert the value on data bus to be set on the instruction register(**`ir`**).

As the processor we will be studying in this project implements the RISC-V ISA, the **`opcode`**, **`funct3`**, **`funct7`** and other fields of the instruction will be decoded accordingly. The RISC-V ISA manual can be found at https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf. Your processors in this project implement the 32-bit variant, known as RV32.

In the operand fetch stage, the controller sets the correct instruction type for generating the 32-bit immediate value. The operand fetch stage always reads values from the register file and assigns them to registers A and B. In the execution stage the controller selects the first operand of the ALU between three signals: register A, program counter register and feedback line from ALU output (`aluResult`). The controller also selects the second operand of the ALU between three signals: register B, constant value of 4, immediate generated in the operand fetch stage. Controller unit signals the operation to be taken by the ALU. In case of instructions that can't be executed by the ALU such as multiply instructions, controller enables a separate multiplier circuit. The multiplier execution unit also resides in this stage. The inputs to this unit come from register A and register B and the unit writes to a specific signal that can be the input for writing to the register file.

In the write-back stage, three different set of instructions are capable of writing to register file. When executing load and store instructions the controller sets the alu inputs and operations to calculate target address and sets the `memoryAddress` accordingly. In case of load instructions, the result will be written to `memoryDataRegister`. The controller then assigns the right values to input multiplexor at the write port of register file. This multiplexor can select between `memoryDataRegister`, `aluResult` and `multiplierResultRegister` at the write-back stage. In general design of the processor the multiplexor at the input of write port of register file should be able to select between different sources that provide instruction results.

Assignment 1 – Deadline: January 9th, 2024

The Verilog modules and main core file along with a testbench is provided in the zip file CompOrg_FinalProject.zip. This also includes a /Firmware directory that contains a [.s] assembly code which is the program to be executed on your CPU and a python script that generates a suitable memory firmware file from assembly code. You can also access all the files related to the class and final project by cloning the associated Git repository:

```
git clone https://github.com/IUST-Computer-Organization/Fall-2023.git
```

Problem 1) We are going to write a RISC-V assembly code that calculates the factorial of the input given in register `a0`. The template body of this program is provided in factorial.s given in /Firmware directory. The first instruction in this code initiates a value in the register `a0`. Under the label `factorial` write codes you want to run once in order to set up the calculation. Under the label `loop` put your main calculation code, to run repeatedly. You'll need to use a branch instruction using the label `loop` as branch target to create a "for loop" for factorial calculation. The last line in the code is the `ebreak` instruction that indicates the end of program execution, do not remove this line.

```
# Initializing the value of n
    addi    a0, zero, 5

factorial:
    # Setup code, to be executed once
loop:
    # Main code implementing the factorial calculation
    ebreak
```

After completing this code, you'll need to run it to ensure correct execution. For this purpose, we are going to use RISC-V Venus simulator extension on VSCode. The ISA simulator serves as our reference for the ISA. It is not cycle-accurate, but it quickly executes assembly code and provides results. In VSCode press Ctrl+Shift+X to open Extensions panel. Search for RISC-V Venus Simulator in the search bar and install the extension. After complete installation of the extension open your assembly code factorial.s and press Ctrl+F5 to run and debug it.

After making sure that the program runs correctly, we'll need to get an assembly text file output. From run and debug panel, choose VENUS OPTIONS go to Views and select Assembly. Save the generated file as factorialAssembly.txt.

A python file is provided in /Firmware directory that transform the assembly text file to a suitable hex file that will be read by Verilog testbench as the instruction section of memory. The processor accesses this section during fetch stage of each instruction. Run the following command from your terminal in the correct directory:

```
python hexConverter.py factorialAssembly.txt
```

This will create the firmware.hex file that will be loaded as the memory array in RISCV_Core_Testbench.v to be executed on your processor.

Problem 2) The data path module instantiations and component connections shown in figures 1-3 are written in Verilog and are available in RISCV_Core.v. In this problem and the next one, your job is to describe the internal hardware of some the modules. The file Arithmetic_Logic_Unit.v contains a template body of an ALU suitable for integration in this CPU. In this problem your is to design an ALU and complete this file. Input and output ports of this module match their instantiation in the main core (if you change any of the ports, be aware to also change the module instantiation in the core). The behavioral description of the zero output of ALU is also provided at the end of the module. Using the `ALU_OPERATIONS` defines in Defines.v create the required internal circuit that supports the provided operations. Some of basic operations your design needs to support are: Addition, Subtraction, bitwise logical operations such as And, Or and Xor.

Problem 3) One of the major execution units present in many CPU cores are multipliers. In this problem you are going to learn about one way of implementing them and after designing a multiplier circuit you need to integrate it with the main core. The **`Multiplier`** module instantiation is already done in RISCV_Core.v. This instantiation references the module in Multiplier.v. As a sample the multiplication in this module is done using Verilog's `*` operator. You need to replace this code with your own design. Using the following paper and the proposed design, implement an 8-bit multiplier with Verilog and complete the Multiplier.v. As the registers in RISC-V are 32-bits using an 8-bit multiplier will cause us many limitations which we will neglect for now and multiplier circuit takes only the low byte of the operands.

Also, find the maximum n for which it is possible to calculate $n!$ with this multiplier.

A Simple High-Speed Multiplier Design, Jung-Yup Kang et al.

IEEE Transactions on computers, 2006

Assignment 2 – Deadline: Your Presentation Date

In this assignment, you are going to design the controller for your processor and after finalizing your design, perform testing and verification. As shown in the figures 1-3, the green signals are fed as inputs to the controller and the controller generates the red signals as is needed for each stage of execution. The multi-cycle controller in this processor is implemented as an FSM that determines the control signals based on the present state and the next state is determined by the inputs to the controller. Pseudocode describing the controller FSM for some states is as follows:

State	Control Signals
fetch_begin	Set `instructionOrData` to `INSTRUCTION` Set `memoryReadWrite` to `READ` Go to **fetch_wait**
fetch_wait	While memory is not ready stay in **fetch_wait** If (`memoryReady`) go to **fetch_done**
fetch_done	Write to instruction register. Set `irWrite` to `ENABLE` Set first ALU operand to `PC` Set second ALU operand to `4` Set ALU operation to `Add` Write the result to PC. Set `pcWrite` to `ENABLE` Go to **decode**
decode	If opcode field is `OP` then `instructionType` is R-type If opcode field is `OP_IMM` then `instructionType` is I-type If opcode filed is `LOAD` then `instructionType` is I-type If opcode field is `BRANCH` then `instructionType` is B-type … Go to **execute**
execute	…

In this assignment your job is to complete the table above which describes the processor states. Different instructions require different control signals when they get to execution stage. The Fetch and Decode stages occur for all instructions regardless of what they are. This means that the program counter register increments by four at the fetch stage. You need to be aware of this calculation when considering control transfer instructions. The RISC-V ISA provides two types of control transfer instructions: unconditional jumps and conditional branches. All branch instructions use the B-type instruction format. The immediate value extracted from **`ir[31:0]`** encodes signed offsets. The offset is added to the address of the branch instruction to give the target address. Branch instructions compare two registers. BEQ and BNE take the branch if registers *rs1* and *rs2* are equal or unequal respectively. BLT and BLTU take the branch if *rs1* is less than *rs2*, using signed and unsigned comparison respectively.

When calculating the branch target address, you need to be aware that the program counter register is already increment by four in Fetch stage. You'll need to decrement the calculated target address by four if the branch is to be taken.

For your convenience the control signals required are defined in RISCV_Core.v and the values that can be assigned by the controller are provided in Defines.v. You can use these definitions for a simpler and more readable implementation of your controller FSM.

After the completion of your controller its time to test and verify your processor. The behavior of a RISC-V program depends on the execution environment in which it runs. A RISC-V execution environment interface (EEI) defines the initial state of the program, the accessibility and attributes of memory and I/O regions, the behavior of all legal instructions executed on the core and the handling of any interrupts or exceptions raised during execution including environment calls. The RISCV_Core_Testbench.v testbench provides a suitable environment for this processor to run. It includes a memory module that is loaded by the firmware.hex file containing instructions for the program you wrote in the first problem of assignment 1. The memory interface behavior is described at the end of the testbench. After making sure that your code factorial.s runs correctly in Venus simulator, use the Python script provided in /Firmware directory to generate firmware.hex and then simulate the environment test bench with iVerilog.

You can follow the correct execution of the program on your CPU using the waveforms in RISCV_Core.gtkw. You can open this file using GTKWave and trace the control signals and debug wires connected to the register file. Ensure the correct execution of your program by inspecting the values of registers at the end of execution.

Submission

Send all your assignment related files and results of your simulations along with a detailed report in [.pdf] format describing your design choices all in one zip file. There will a presentation session for all groups in which you will present your CPU and show your results. For additional bonus points you will face technical challenges and questions related to your design.

References

A. Waterman and K. Asanović, Eds., The RISC-V instruction set manual, volume I: User-level ISA, Version 20191213, RISC-V Foundation, Dec. 2019. [Online]. Available: https://riscv.org/specifications/

Patterson, D. A., Hennessy, J. L. (2020). Computer Organization and Design RISC-V Edition: The Hardware Software Interface. Netherlands: Elsevier Science.