Phase 2 Documentation

PIPELINED IMPLEMENTATION OF 32-BIT RISC-V INSTRUCTION EXECUTION

Indian Institute of Technology, Ropar

CS204, Computer Architecture and Design

Course Project

Submitted by:

Jaglike Makkar, Apurv Rathore, Vasu Bansal, Anant Dhakad, Aman Kumar Chourasiya

Submitted to:

Dr. T.V. Kalyan

Acknowledgment

We would like to thank Dr. T.V. Kalyan for providing us such a wonderful opportunity to work on this project. With the help of this project, we are able to practice and actually implement what we learnt in theory. It was really fun simulating the actual hardware of the computer. Through this project, we were able to learn many new interesting aspects, which would have been not possible, if we would have solely relied upon theoretical knowledge. We would also like to thank for extending cooperation in clearing all the doubts that arose from time to time and also cooperating in the aspects relating to the deadline/submissions of this phase of the project.

Table of Contents

Introduction

This document describes the design aspect of RISC-V Simulator, a GUI-based functional simulator for a subset of RISC-V instruction set. This code has been written in **Python** programming language.

Target Audience

Any one, who wants to see and discover how the computer is able to execute the codes that we write so easily in high level languages and also how this black box converts a series of zeros and ones into useful information upon which we are highly reliable. A negligible amount of knowledge of python is required to understand the code for our project. So if you want to decode this black box called computer, then go and try our RISC-V simulator.

Definitions, Acronyms and Abbreviations used

GUI - Graphical User Interface
PyQt5 - Python library used to built gui

Overview of the Project and Key Features

The following set of instruction are fully supported with both pipelined and non pipelined approaches

R format - add, and, or, sll, slt, sra, srl, sub, xor, mul, div, rem
I format - addi, andi, ori, lb, ld, lh, lw, jalr
S format - sb, sw, sd, sh
SB format - beq, bne, bge, blt
U format - auipc, lui
UJ format - jal

The gui design looks as follow:

Screenshot of graphical user interface

Requirements for running the project

Python and PyQt5

How to install and run the project

Download the zip file and extract it. Make sure the requirements mentioned in requirements.txt file are installed in your computer. Run the main file with the help of python using the command

➜ **python gui.py**

Code Assumptions/System Requirements/ Constraints

The instruction set provided as input, must be a subset of those that our code supports.

Design Specification

A) Directory Structure

Phase 2 -

| - hdu_class.py
| - input.mc
| - output.txt
| - state_class.py
| - gui.py
| - main.py

B) Input/Output Mechanism

Input Mechanism

Input to the RISC-V Simulator is a Mem file that contains the contents of Instruction memory and the Data memory separated by a dollar sign.
It contains the machine code of the instructions and their corresponding address separated by space.
The Data Memory is stored as the data and its corresponding address separated by space.

Example :

0x0 0xE3A0200A
0x4 0xE3A0300A
0x8 0xE0821003
0xc $

0x10000000 0x2
0x10000004 0x8

Note : 0xc $ is used to distinguish between Instruction Memory and Data Memory.

Output Mechanism

Output.txt - This file shows the values in Registers, Memory and PC after each instruction execution.
Output.mc - This file shows the instruction memory and data memory in the same format as input.mc
Simulator - It shows the type of instruction(Add, sub, xor, etc) and values stored in registers after each instruction

C) Code WalkThrough

Data Structures and Variables Declaration

The registers, instruction memory, data memory are declared as variables of the **class CPU** as opposed to the phase 1 of the project, where these variables were declared as global variables. Similarly the control signals which are used to define the control circuitry are declared as part of the **class State**. In this phase we have adopted a coding strategy which is majorly based on using classes. By the use of classes, we are easily able to follow very important and basic principles like **encapsulation** and **abstraction**. A class based approach helps significantly in the implementation of pipelined instruction execution. Registers are maintained in an array where each element of the array is the simulation of the actual register at the hardware level. The Instruction Memory and the Data Memory are maintained in python dictionaries where the key is the address of the instruction and the value is the instruction in case of instruction memory and the data in case of data memory. Each element of these dictionaries is a list of size 4 (i.e. containing 4 elements) and the default value for each of these elements is zero. This is ensured by the use of **defaultdict** function imported from the **collections** package inside the

state_class file. Some global variables are also declared like the ones used to check data forwarding, stalling, knobs pressed or not, and many more. There are many other variables and data structures used in the entire code, so the ones stated here are just a brief overview.

Simulator Flow

There are two broad steps in the flow of the simulator:

1. First is that the input file is loaded/fed to the python code and using this file Instruction Memory and Data Memory are filled up with the instructions and data provided respectively.
2. Secondly, the simulator executes the instructions one by one or all of them together depending upon whether the step/run button is pressed. In case the step button is pressed, at each stage the instruction in execution is being shown. Additionally the **number of clock cycles** are also displayed at the end of each step.
3. In case there is a case of data forwarding between two instructions, then it is clearly shown in the GUI as well.

Major functions/classes that are used with brief explanation

1. class BTB

a) The Object of this class contains a table which stores the target address associated with an instruction if the instruction is a branch or jal or jalr instruction.
b) The target address associated with jalr instruction is updated each time it is encountered.
c) When a branch instruction or jal or jalr instruction is seen for the first time, this creates a stall in the pipeline and the data associated with that instruction in the BTB table is updated in the decode stage.

2. function evaluate inside class HDU

a) This function explores the data hazard possibility associated between instructions in two stages of the pipeline.
b) It accepts two integers values which denotes the stage in which those instructions are present in the pipeline.
c) Now it checks whether there is a data hazard among these two instructions.
d) If so then it forwards the data if the knob for Data Forwarding is turned on.
e) This function is called only if the data dependency can be resolved using Data Forwarding and stalling is not required.

3. function isDataHazard inside class HDU

This function takes the pipeline stages as input parameters and examines all the situations where there is data hazard between any two instructions in the pipeline and decides whether stalling is needed or data forwarding is enough for resolving the problem.

4. IAG (Instruction Address Generator) module

This module is used to update the value of the PC.

If **MuxPC_Select** is off, then the PC is updated to store the value RA. Else if **MuxINC_Select** is on, then the PC is incremented by immediate value and if **MuxINC_Select** is off then the PC is incremented by four.

5. function ProcessorMemoryInterface inside class CPU

a) This function helps in interfacing with the Data Memory.
b) This function decides if there is a need to do Memory Read operation or Memory Write operation using the **MuxMA_Select**.
c) If it is so, then using the values of MDR and MAR it does the operation required.

The 5 basic stages/ Building Blocks of the code

These functions are enclosed within the class CPU along with other helper functions.

1. Fetch

In this stage, the PC is converted into its hexadecimal equivalent and stored in a variable. Using this variable as key, Instruction is fetched from the Instruction Memory and the PC_Temp is updated to store PC + 4.

2. Decode

Here the opcode and the func3 fields of the instruction is extracted using bit masking technique. **Error handling** is also done here. Even if there is no func3 field in the instruction, extracting those bits does not hinder the process. Also in case the opcode is invalid

1. Extraction

There are five different types of instructions in the instruction set that we are asked to support. So based on the opcode, the instruction is classified into one of those categories i.e. **R** type, **I** type, **S** type, **SB** type, **U** type and **UJ** type.

After identifying the type of the instruction, the values of each of its fields are extracted and stored in desired variables.

There are multiple instructions in these categories as well. So to identify the instruction particularly, the values of funct3 and funct7 fields are used if necessary.

Based on the type of instruction, the values of **rs1** and **rs2** registers are stored in **RA** and **RB** respectively.

If it is a store instruction then **RM** is updated to store **RB**.

The value of the **immediate field** is sign extended.

Note : In case of UJ and SB format instruction, immediate value is doubled after sign extension.

Note : It is essential to identify the particular instruction to figure out the type of operation to be performed.

2. Operation Identification

An array **ALUOp** of size 15 is created. The size 15 is chosen as there are 15 distinct operations possible for the ALU unit to perform. Each index of this array represents a unique operation. The below table maps each index of this ALUOp array to its corresponding operation.

Index	Operator/function used for this operation	Name of the operation
0	+	Addition
1	-	Subtraction
2	/	Divide
3	*	Multiply
4	%	Remainder
5	^	XOR
6	<<	Shift left
7	sra	Shift Right Arithmetic
8	>>	Shift Right Logical
9	|	Or
10	&	And
11	<	Less than
12	==	Equal to

13	!=	Not equal to
14	>=	Greater than equal to

In the Execute stage, the type of operation to be performed is determined by checking, at which index in the ALUOp array the stored number is one.

Example : Suppose index 0 has value 1, then addition will be performed in the ALU.

After the instruction is particularly identified, corresponding to the type of operation which should be performed in this instruction, the particular index in the array ALUOp is turned on and the rest are turned off.

3. Control Signals Generation

The necessary control signals needed for the smooth operation of this instruction is generated in the Decode Stage itself. After identifying the instruction, the values of each of these control signals are properly set.

3. Execute

In this stage, two variables are created namely **InA** and **InB** to store the final operands. InA is set to RA. Based on the value of **MuxB_Select**, InB stores either RB or immediate value calculated in Decode Stage.

Now, the index of the array ALUOp which is turned on is found. Based on the type of operation it represents, that operation is performed between InA and InB and the result is stored in **RZ**.

In case of comparison operation, RZ stores 1 if the type of comparison performed is true otherwise 0.

4. Memory Access

In this stage, based on the value of **MuxY_Select**, three different operations can be performed.

Case 1 : MuxY_Select == 0

In this case, the result of Execute Stage i.e. **RZ** is stored in **RY**.

Case 2 : MuxY_Select == 1 (in base 10)

The hexadecimal equivalent of **RZ** stored in **MAR** (Memory Address Register). Now if **Mem_Read** is on, then using MAR as key, data is fetched from the Data Memory and stored in **RY.**

Otherwise if **Mem_Write** is on, **MDR** is updated to store **RM** and this value is updated in the desired address of Data Memory.

Case 3 : MuxY_Select == 2 (in base 10)

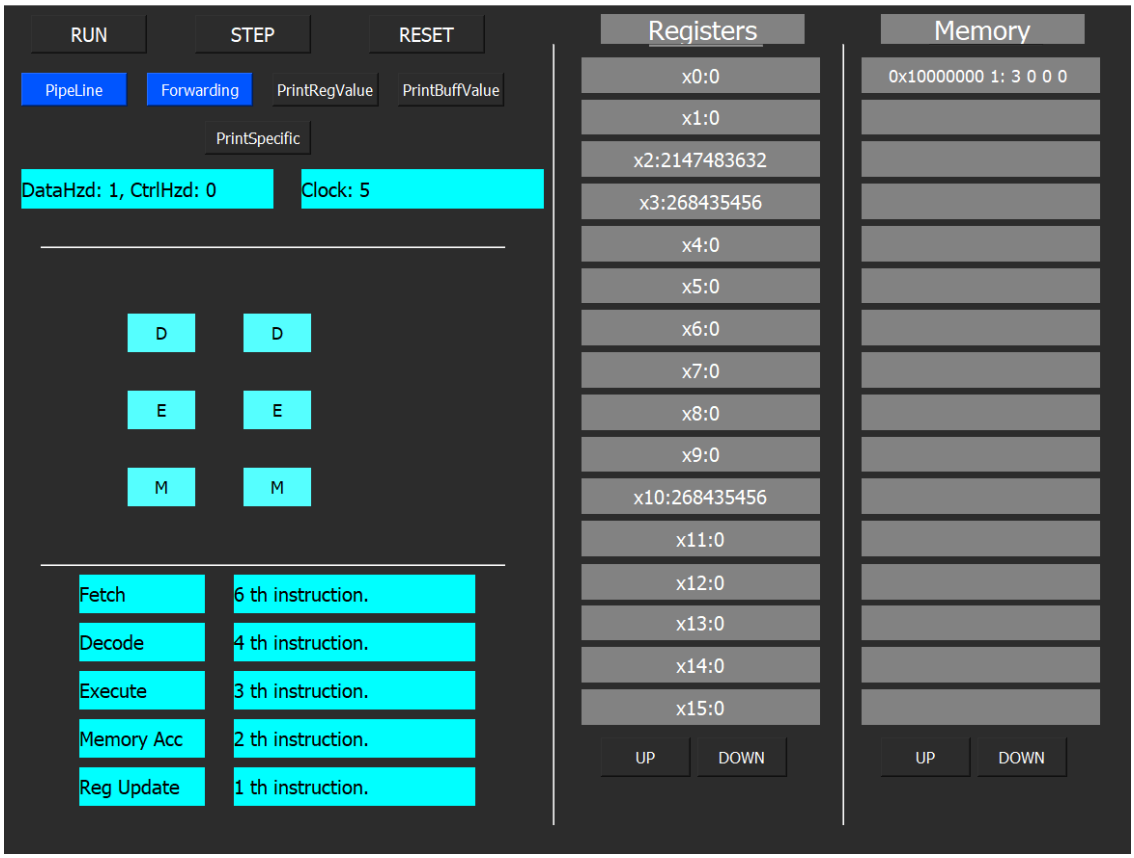
RY is updated to store **PC_Temp**.

5. Write Back

If **RF_Write** is on and RD is not equal to zero then the value of **RD** register is updated to **RY**.

Gui Design and Working

PyQt5 designer was used to make the GUI design.



Future Contingencies

In the upcoming phase of the project, we would extend the memory module of this code and simulate it in the way it actually works in the hardware i.e. we would be implementing the hierarchical memory structure for the memory and also introduce faster memory segments like the Cache memory.

Software/ Technologies used for project creation

1. VS Code was used as the ide to write and test our code
2. Git/GitHub was also used for the development of the project
3. Google docs was used to create the documentation for the project
4. Google meet was used for conducting virtual meetings and discussing various things about the project.

Credits/Contributions :

Aman Kumar Chourasiya:

Extended the phase1 code, debugged the code to successfully run fibonacci and factorial programs, adding branch prediction logic, added data forwarding paths and hazard detection logic, added 5 basic units/functions explanation to the documentation, added clocks, reset and error box in gui.
Added the knobs and the logic associated with them. Played a significant role in designing the overall concept of designing and implementing the pipeline logic.

Apurv Rathore:

Extended the phase1 code, added the gui to the code, pipeline flushing logic, hazard detection unit, knobs for gui, created the register file for gui, created the UI for GUI, writing the output file, created the BTB class, integrating the memory to GUI, debugging the code.

Anant Dhakad:

Debugging the code, Debugging the code, linking the gui, added color coding to the gui, added logic for stalling, worked on README, knobs for gui, design data memory in the gui, created the UI for GUI, wrote the logic for printing the output file.

Jaglike Makkar:

Creating the branch target buffer, added new control signals, implemented and integrated data hazard detection, adding branch prediction logic, documenting

some new functions, implemented buttons like run, step, reset and 5 knobs, integrated GUI.

Vasu Bansal:

Adding/linking gui in the design file, extended the phase1 code, debugging the code, pipeline flushing logic, writing the output file, hazard detection, adding documentation and formatting, knobs for pipelining, design data memory in the gui, branch prediction, added some knobs logic.