**CS204**

**Design Document: RISC-V32I Simulator**

**Phase -1**

**Group No. 15**

*Vasu Bansal 2020EEB1217*

*Tiya Jain 2020EEB1213*

*Keshav Arora 2020EEB1177*

*Akash Garg 2020MEB1262*

**Supported Instructions:**

R format - add, and, or, sll, slt, sra, srl, sub, xor

I format - addi, andi, ori, lb, lh, lw, jalr

S format - sb, sw, sh

SB format - beq, bne, bge, blt

U format - auipc, lui

J format - jal

**Variables Defined:**

* pc: Program Counter
* clk: clock
* instruction\_memory: default dictionary with key-value pair as address-instruction
* data\_memory: default dictionary with key-value pair as address-data
* address: stores the key (address) value for both instruction\_memory and data\_memory
* instruction: stores the value (instruction for instruction\_memory and data for data\_memory)
* mem\_location: stores the address in decimal form
* tempI, tempS, tempB, tempU, tempJ: it contains the arranged form of the immediate from the machine code in binary format for sign extension
* control\_file: contains the pointer to control.csv file
* pc\_new: stores the updated (pc+4) value of pc
* offset: provides the address for branch target
* rm: buffer register to store the ALUresult
* ma: variable to store the data value loaded from memory
* result\_write: variable to write the result in the register

**Contents of the Repository:**

**myRiscSim.py**

This is the main code of the simulator, which takes input from the ‘input. mc’ file and performs sequential execution of the machine code (contained in the ‘input. mc’ file). Every instruction has five stages: fetch, decode, execute, memory access and write back. The simulator provides the status of each of the five stages while processing the stages and displays the total cycle count for the entire program. The file also creates a file containing the modified data values (for the data segment) and instructions (for the code segment) after the processing of all five stages is completed.

The significant segments of this file are briefed as follows:

**Reading from file:**

‘*input. mc*’ file is read using the ‘*def read\_from\_file (file\_name)*’ function, which creates a pointer to the file and an iterator to read the contents of the file line by line.

Instruction and Data Memory are treated as separate entities and structured into a dictionary with a key as the memory address and the corresponding values as instruction (for code segment) or data (for data segment).

So, this function maps the key with the corresponding instruction for the instruction memory and the corresponding data for the data memory.

Our memory is byte-addressable. Each key (or index) contains a byte of information.

**Control Signal Generation:**

‘*def control\_signal (arg)*’ defines control signals that are decided in the decode stage of the execution. The control signals are uniquely associated with the value of the opcode, func3, func7- which forms the type of instruction performed at every sequence.

**Fetch:**

This function takes the machine code from the instruction memory and converts it to a binary string value stored in the variable called binary\_instruction.

**Sign Extension:**

This function extends the representative bits of the value without affecting the magnitude of the value. It extends the number of bits used for the immediate, considering the instruction format determined by the opcode.

It assembles the bits from the *binary\_instruction* to form the immediate and then copies the most significant bit to fill in the remaining bits.

For B-Type and J-Type instruction, the number of bits after sign extending is 13 and 21, respectively. The extra one bit is added to maintain consistency across the formats since the least significant bit is always 0 and is therefore omitted.

**Decode:**

This function initially initialises every format’s section and sign extended immediate values. It then decides the immediate type based on the control signal OP2\_Select.

control.csv file is read as a data frame using pandas which contains a list of all instruction types and related control signals juxtaposed with the instruction name. Using this file, control signals are written in the myRISCSim.py file. The conditional if-else statements contain varying values of opcode, func3, and func7 as conditions to generate these control signals.

**Execute:**

The function starts with incrementing the value of pc to the next address and executing the ALU operation to be performed while also selecting the value of op2. For B-Type and J-Type instructions, the value of control signals like ‘*is\_branch’*, and ‘*Branch\_trg\_sel*’ according to the ALU result.

**Memory Access:**

This part of the code handles the read-and-write transactions in the data memory for loading and storing instructions from and to the memory. *Mem\_read/ mem\_write* control signals govern the conditional statements. A ‘*num\_b’* named control signal decides the type of instruction (byte, half word, word) under both the load, and store instructions. For byte and half-word type instructions, the value is sign extended.

**Write Back:**

The last stage of the execution updates the value of pc, dependent on the control signal *is\_branch*. It also writes the value in the destination register rd (result\_write) (if required by the instruction)

**Terminate:**

This function terminates the file when any empty line is encountered in the input.mc file.

**Output Files**

* OutputFile\_txt():
  + This function outputs the register and data memory
* OutputFile\_txt():
  + This function output the instruction and data memory

**Main Function**

* This is the primary function of the file that calls in all the five stages after reading from the file. It also updates the clock cycle and prints it at the end.

**Control.csv**

This file is the supporting file for the myRISCsim.py file that contains the instruction name and the values of control signals for every instruction in each line. It designates a unique number for every ALUop, unique to every instruction type.

**How to run the program?**

* Clone the git repository to your local machine
* Install python3
* Open VS CODE or any other IDE and run the ‘myRISCsim.py file’ in the src folder

**Test Files for the Simulator**

* *fibonacci. mc*: calculate the fibonacci series till the 7th number and store the number in the memory
* *nsum.mc*: calculates the sum of whole numbers till ‘10’ and stores the result in register
* *bubblesort.mc*: sort the array using the bubble sort technique

**\*\*\*IMPORTANT: THE CODE IS EXECUTED IN REFERENCE TO THE STEPS AND FLOW OF THE GIVEN DIAGRAM\*\*\***