PROJECT PHASE 1: Single Cycle Design

Design Document

Group No. 18

Group Members:

Rohit Madan – 2020EEB1202

Omkar Prashant More – 2020EEB1188

Harsh Gupta – 2019MED1008

The document describes the Design of our RISC-V simulator, which currently acts as a simulator the Single Cycle Executions. Our Simulator is built completely using Python Programming Language.

Input:

For the input, we provide a .mc file which contains:
Instruction Memory (Instruction Address and Instruction Encoding)

Data Segment (Address and Data which is stored)

We are using the Encoding 0xFFFFFFFF as our termination code.

We have the Code/Instruction Segment before the Termination Code and after it Starts the Data segment.

Functionality:

The simulator reads the instruction from instruction memory (FETCH), decodes the instruction (DECODE), execute the operation (EXECUTE), access and update the memory (MEMORY ACCESS) and write back to the register file (WRITE BACK).

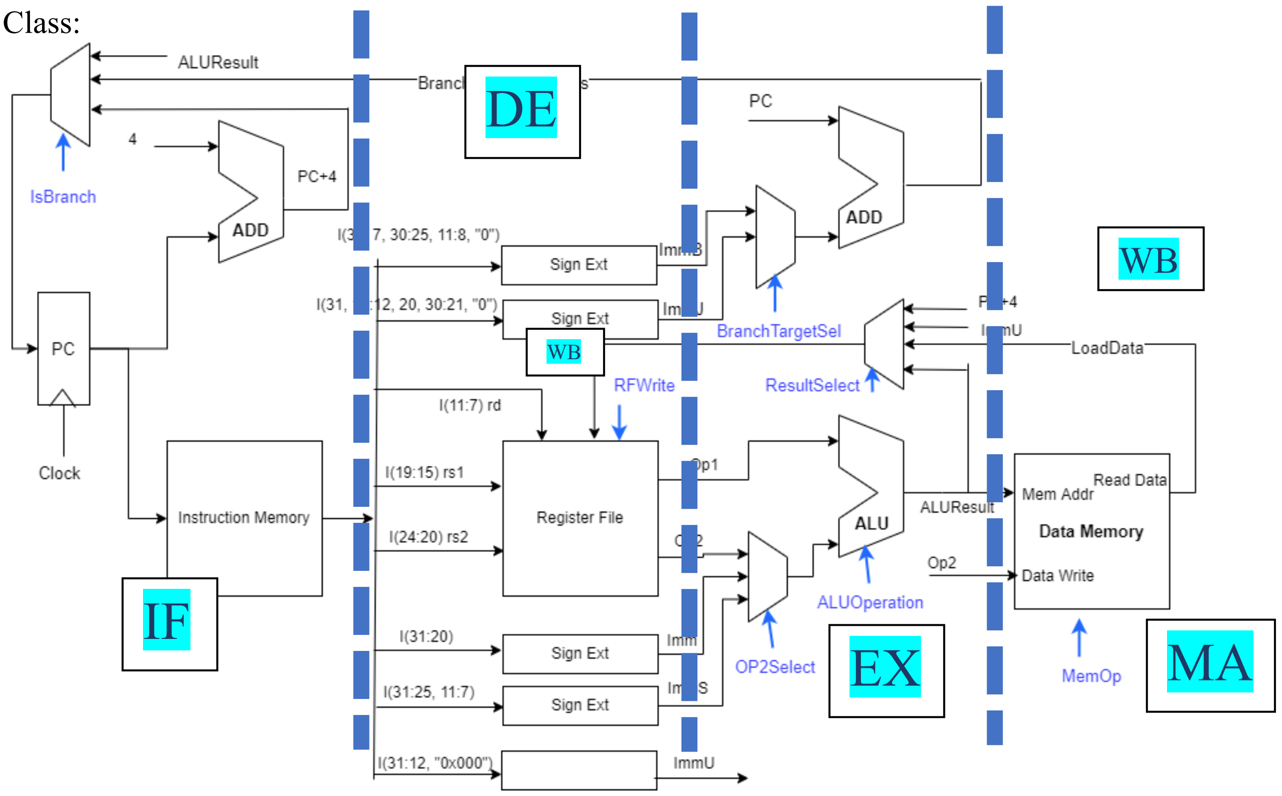
We have included Print statements at the end of each of the 5 Steps to help the user in understanding the process, and assist us in debugging the code:

The print statements are as follows:

1. **Fetch():** “FETCH: Fetch Instruction (Encoding) from address (Instruction Address)”
2. **Decode():** “DECODE: Read Register: rs1 = , rs2 = “
3. **Execute():** “EXECUTE: Operation on rs1 and rs2/immediate”
4. **Memory_access()**: “MEMORY: Storing at (address)/ No Memory Operation”
5. **Write_back():** “WRITEBACK: Value in register (rd) is (memory)”

Design:

Our Design outline was completely based on the following Single Cycle Design taught in the Class:



The variable names were also kept similar to those in above diagram, for convenience.

Simulator Flow:

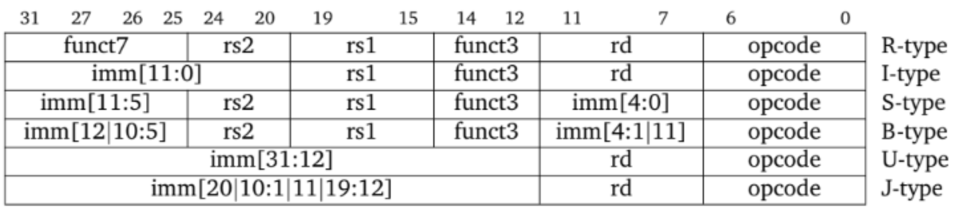
Steps:

1. Memory is Loaded from the input memory file.
2. Simulator executes the instructions one by one.

The second step is an infinite loop of executing instructions which terminates on detecting the termination code.

The implementation of fetch, decode, execute, memory and write-back.

1. From **fetch()**, we get the instruction memory and the data memory dictionaries.
2. In **decode()**, we detect the instruction format using the value of the opcode, and the instruction using the value of funct3. All these values including those of rs1,rs2,rd and imm are calculated by converting the hexadecimal instruction code into 32-bit Binary number and then determining the values using the following chart taught in class:



31 27 26 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]				rd	opcode	U-type
imm[20|10:1|11|19:12]				rd	opcode	J-type

InstructionPacket gets filled in the decode stage.

3. After getting to know the instruction (along with rs1, rs2, rd, imm) from the decode() function, we move on to the **execute()** function, where we perform arithmetic operation of Add & Subtract as required, ad then get our ALUResult value, along with isBranch condition (isBranch = 1 if branching else 0)
4. In **memory_access()**, we update the value of LoadData if need be. This function only carries out the operation in case of Load and Store operations.
5. In **write_back()**, we write the updated values (decided by our RFWrite Control) into the Register File (i.e. Destination Register). The value to be written is decided by our RFWrite Control.

Along with the following functions, we have the following supporting classes as well:

- **ALU**: Carries out Arithmetic Operations classified for each instruction format. (op1 and OP2Select as input pins and ALUResult as one output pin)
- **Control Unit:** Determines all the Control Variables like isBRanch, ResultSelect, RFWrite etc, which are pivotal in the processor stages (Execute and ahead). For each instruction type this class has functions that set the control signals requited for that type.
- **Memory:** The memory class consists of the read word, write word (in instruction as well as data memory), and load program memory functions which loads the input instructions into the Instruction Memory and loads the Initial Data Memory as well.
- **Instruction Packet:** Contains all the Global Variables required by the processor stages such as various immediates , op1,op2, PC, information about the instruction, ALUResult, Loaded Data etc.
- **Signed Extension Function:** We have this function, to extend out the MSB of the immediate value to 32bits. This is especially needed as we have signed numbers. We also have a function signed32, which gives us the negative decimal numbers when the MSB = 1 (2's complement Method).

Testing:

To check the correctness of our Simulator, we tested the simulator on the following programs:

1. Fibonacci Program
2. Sum of the array of N elements: Initialize an array in first loop with each element equal to its index. In second loop find the sum of this array, and store the result at Arr[N].
3. Bubble Sort Program

The codes for all the programs were written on the Venus RISC V Simulator, and then the Machine Code DUMP was obtained from there. Then we prepared our input .mc file, and ran the Simulator. We print the final Memory State at the end and the results were found to be accurate and matching with that of the Venus Simulator Output as well.

GUI Implementation:

GUI was implemented in the GUI.py file. The GUI Console shows us the Console Ouptut, the values in the individual 32 Regisers and the Final Memory Data.