**Design Document**

**Functional Simulator for Subset of RISC-V instruction set**

The document describes the design aspect of main.py, a functional simulator for a subset of the RISC-V instruction set.

# **Input/Output**

## **->Input**

Input to the simulator is a MEM file that contains the encoded instruction and the corresponding address at which instruction is supposed to be stored, separated by space. These machine instructions follow the Little Endian notation. For example:

* 0x0 0x0A20A0E3
* 0x4 0x0230A0E3
* 0x8 0x031082E0

Additionally, the MEM file also contains the data segment, which is distinguished from the instruction segment on the basis of the address values. Non-zero data values are stored bytewise preceded by the corresponding address at which they are supposed to be stored and separated by space.

## **->Functional Behavior and output**

The simulator reads the instructions from the MEM file, converts them into Big Endian(for ease of processing), and stores the **“address-instruction”** pair as a key-value pair in the **instructionSegment**. Similarly, the data is read from the MEM file and stored in **dataSegment** as key-value pairs.

For every instruction, the simulator then fetches an instruction, decodes the instruction, reads the register, executes the operation, and writes back to the register file. The instruction set supported is the same as given in the lecture notes.

The execution of instruction continues till it reaches instruction “**0x00000000**”. In other words, as soon as the instruction reads “**0x00000000**”, the simulator stops and writes the updated memory contents onto a memory text file.

The simulator also prints messages for each stage, for example for the first instruction above following messages are printed.

**1. Fetch prints:**

* **fetchInstruction():**

FETCH : Fetch the instruction **0xE3A0200A** from instructionSegment using the address **0x0.**

**2. Decode**

* **decodeInstruction():** Operation is ADD, destination register **x1,** First source register **x2** = 0x0000000a, Second source register **x3** = 0x00000002

**3. Execute**

* **executeInstruction():** ADD 0x0000000a and 0x00000002

**4. Memory**

* **memoryAccess():** No memory operation

**5. Writeback**

* **registerUpdate():** write 0x0000000c to **x1**

# **Design of Simulator**

## **->Data structure**

Registers, memories, intermediate output for each stage of instruction execution are declared globally. Efforts are made to make code equivalents of the physical hardware components.

## **->Simulator flow:**

There are two steps:

1. First memory is loaded with an input memory file.

2. Simulator executes instructions one by one.

For the second step, there is an infinite loop, which simulates all the instructions till the instruction sequence reads “**0x00000000**”.

Next, we describe the implementation of fetch, decode, execute, memory, and write-back function.

**-> Five Stages of Single Cycle Execution**

1. **Fetch**

In the **fetchInstruction(),** we fetch the instruction at the address pointed by the **PC** from the **instructionSegment and** store it in the instruction register **IR**. We also calculate the value of **PCTemp** for further use(if any).

1. **Decode**

First of all, we extract and match the opcode of the current instruction in the **IR** with the opcode provided in the datasheet. If the opcode of the IR matches any of the opcodes in the datasheet then we check the subsequent **func3** and **func7** codes to determine which operation needs to be performed. Depending on the identified instruction, control signals like **ALUop, MuxINCSelect, MuxASelect**, **MuxBSelect, isBranch,** etc are assigned to appropriate values. Also, the rest of the fields depending on the type of instruction - like **rs1, rs2, rd**, and immediate values are extracted.

1. **Execute**

According to the ALUop signal generated in the decode stage, we choose the operation to be performed. The output of the ALU is further stored in the **RZ** register. To implement the execute stage, we use **two2dec** to convert binary(2s complement) into decimal, and **dec2two** to convert decimal into binary(2s complement).

1. **Memory Access**

According to the control signals, **memRead**, and **memWrite** which were generated in the decode stage, we decide whether to read from the memory or write into the memory.

For load instruction, the address of the data to be loaded is kept in **MAR,** and the data after retrieval is sent to **MDR**. While, for the store instruction, the data present in **MDR** is stored at the memory location pointed by **MAR**. Based on the control signal **loadType**, the number of bytes to be loaded is decided. Similarly, the signal **storeType** is used to decide the number of bytes to be stored in the data segment.

1. **Register Update**

According to the control signal **writeRegisterFile**, we decide whether or not to update the registers. If writeRegisterFile is “**1**”, then we select the register according to the value of **rd**, and store the value of **RY** into the selected register accordingly.

# **Test plan**

We test the simulator with the following assembly programs:

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

**---------------------------------------------------------------END------------------------------------------------------**