**Design Document: RISC-V SIMULATOR**

The document describes the design aspect of our RISC-V simulator made using Python as a programming language.

# **Input/Output**

## **Input**

The input to the program is a .mc file that contains the encoded instructions and the corresponding address at which instruction is supposed to be stored, separated by space. For example:

0x0 0xE3A0200A

0x4 0xE3A03002

0x8 0xE0821003

## **Functional Behavior and output**

The simulator reads the instruction from instruction memory, 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 of CS-204.

The execution of instruction and fetching continues till it reaches instruction “text\_end” in the input file. In other words as soon as the instruction reads “text\_end”, the simulator stops and writes the updated memory contents to a memory dictionary.

The simulator also prints messages for each stage, for example:

We have added **GUI** for this simulator :

1. A window will appear

2. If you want to run the code step by step:-

>>>>Click on Step button on the gui

-There is a Registers Button and a Memory Button.

-The Registers Button shows all the registers and the values in them

-The Memory Button shows all the data stored at different addresses in the memory

-There are three more buttons which show the Heap and Stack Memory.

3. If you want to run the whole code together:- Click on the Run Button

A separate box is shown where the clock cycle number is displayed. Steps followed in each instruction are also shown as follows:

- Fetch prints:

o “FETCH:

PC\_temp -> PC+4

Fetched instruction - instruction”

- Decode

o “DECODE:

Instruction Type - instr\_type

Operation - operation +

Register values are read.”

- Execute

o “EXECUTE:

PC -> hex(PC)

Returned value - calculated value”

- Memory

o “MEMORY ACCESS:

Memory at 'address' is updated.”

- Writeback

o “REGISTER UPDATE:

Register xi is updated.”

# **Design of Simulator**

## **Data structure**

Registers, memories, Instruction register , clock used for each stage of instruction execution are declared as global variables.

For the implementation of registers and memories two separate dictionaries are used, while for storing the instructions ‘instructions’ and for storing data in memory ‘memory’’ is used.

## **Simulator flow:**

There are two steps:

1. First memory is loaded with an input memory dictionary and each instruction of the input file is stored in instructions and memory data in memory.

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 “text\_end”.

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

### **FETCH:**

In this step after the input file is passed to temp3.py, current instruction is returned by this function which is stored in ‘instructions’ .

### **DECODE:**

As for each instruction stored in ‘instructions’ dictionary, each instruction is decoded according to the opcode it has. Further according to the type instruction necessary details are stored and returned in reg\_list which is further used in the Execution step. All sign extensions required are taken care of for the execution step in this stage.

### **EXECUTE:**

The execute step uses the already calculated values of the register addresses and the instruction to be performed to calculate a value (var) which is then used to write to registers or memory depending upon the type of instruction.The value of PC is also updated in this step, as PC\_temp is calculated.

### **MEMORY:**

In this step the memory function is called with the address of memory to be updated or accessed as a parameter and according to the operation which requires the use of memory such as lb,lh,ld,jal,jalr the values in the ‘memory’ dictionary are read or updated. When memory is accessed, the read value is returned.

### **WRITEBACK:**

If any operation requires updating the values of registers, they are updated by calling the registerUpdate function.

# **Test plan**

We test the simulator with following assembly programs:

1. input\_fib11.mc
2. input\_fact10.mc
3. input\_bubble.mc
4. inp.mc