Design Document: RISCV Simulator

The document describes the design aspect of RISCV Simulator, which simulates the machine level execution of RISC V 32-bit instructions using a high level language.

# Input/Output

## Input

Input to the simulator is {**FILE NAME**} data.mc file that contains the encoded instruction and the corresponding address at which instruction is supposed to be stored, separated by space as a separator.

{**INSTRUCTIONS**}

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 CS204 and as per the lab assignments we did.

The execution of instruction continues till it reaches instruction {**EXIT CODE INST**} “swi 0x11”. In other words, as soon as the instruction reads {**EXIT CODE**} “0xEF000011”, the simulator stops and writes the updated memory contents onto a memory text file.

{**EXAMPLE OF O/P + DESC OF GUI**}

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

* Fetch prints:
  + “FETCH:Fetch instruction 0xE3A0200A from address 0x0”
* Decode
  + “DECODE: Operation is ADD, first operand R2, Second operand R3, destination register R1”
  + “DECODE: Read registers R2 = 10, R3 = 2”
* Execute
  + “EXECUTE: ADD 10 and 2”
* Memory
  + “MEMORY: No memory operation”
* Writeback
  + “WRITEBACK: write 12 to R1”

# Design of Simulator

## Data structure

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

For the implementation of registers and memories, two separate arrays are used, while for storing the instructions {**INSTR ARRAY**} instruction\_dict and for storing data in memory {**MEM ARRAY**} data\_dict is used. Our memory is Byte Addressable.

## Simulator flow:

There are two steps:

1. First memory is loaded with an input memory array and each instruction of the input file is stored in {**INSTR ARRAY**} instruction\_dict and memory data in {**MEM ARRAY**} data\_dict.
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 {**EXIT**} “0xffffc”.

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

{**CHANGE THESE ONCE**}

1. **Instruction Fetch**: PC is incremented by 4, and the instruction is loaded from the memory and stored in the Instruction Register.
2. **Instruction Decode**: We are identifying the instruction using the .csv file and pandas library and returning all the required fields. The registers RA and RB are also set during this stage.
3. **Execute**: The ALU executes the instruction by computing the desired output using values stored in registers in RA, RB, imm, etc. The required type of ALU instruction is determined by another .csv file. We are updating the output in the RZ register.
4. **Memory Access**: Memory is read and written in this stage. This stage is used only for load and store instructions. For the remaining instructions, this stage is redundant. The value of the RY register is also consequently updated.
5. **Register Write-back**: Here we are storing the value of the temporary register RY in the destination register, resulting in the register files being updated. The register update is extremely fast compared to memory read and update..

# Test plan

We test the simulator with following assembly programs:

* Fibonacci Program
* 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].