Design Document: Functional Simulator for Subset of RISC V instruction set

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

# Input/Output

## Input

Input to the simulator is .MC file contains the encoded instruction and the corresponding address at which the 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.

The execution of instruction continues till it reaches instruction “0xFFFFFFFF”; 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 third instruction above following messages are printed.

* Fetch prints:
  + “FETCH: Fetch instruction 0xE3A0200A from address 0xc”
* Decode
  + “DECODE: DECODING INSTRUCTION AT PC 0x8 of type I”
* Execute
  + “EXECUTE: EXECUTING ADDI INSTRUCTION AT PC 0X4”
* Memory
  + “MEMORY: No memory operation.”
* Writeback
  + “WRITEBACK: WRITING BACK TO MEMORY”

# Design of Simulator

## Data structure

Registers, memories, and intermediate output for each stage of instruction execution are declared as global. The variables are not visible outside the file, thus, making the data encapsulated in the myRISCSim.py.

## Simulator flow:

There are two steps:

1. First, memory is loaded with an input memory file.
2. Simulator executes instructions in a pipeline.
3. Registers are written in reg\_out.mc file and data memory is written in data\_out.mc file.

For the second step, there is an infinite loop, which simulates all the stages in a pipeline till the instruction sequence reads “0xFFFFFFFF”.

After it is read, only the instructions remaining in the pipeline are run.

We have used 2D arrays for pipeline registers.

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

## Fetch:

The implementation of instruction begins with this phase. Using the Program counter, PC, which effectively carries that instruction's address, the instruction is now retrieved from memory.

Here, the instruction\_word[0][0] of the instruction is updated. Instruction\_word will now contain the hex code of the instruction to be executed.

If there is a stall, PC [0][0] is not updated, otherwise is updated.

It also checks the presence of pc in Branch Table Buffer. If it is present, next is taken from it otherwise pc is incremented by 4.

## Decode:

The second stage of the total implementation is the decoding process. The hexadecimal value is retrieved from the PC[0][1] and decoded in this phase. To put it another way, information about the opcode, function 3, and function 7 and immediate is retrieved. As a result, we can determine the operation that must be carried out as well as the relevant registers and immediately after the operation. Moreover, if necessary, the register values are read. The value of operand1 and operand2 is updated on which the operations would be performed in the execute stage.

These values are passed to [2][0] index of 2D arrays to be used in next Cycle.

In addition, we set the RF\_Write signal to true when we need to write the data back into the destination register, such as with R, I, U, and J-type instructions.

In this stage, we are also checking whether rs1 and rs2 of this stage is matching with rd of any of the further stages. In case Yes, stalls are provided accordingly.

## Execute:

The third stage of an instruction's execution is this. Since we now know what ALU operations need to be performed from the opcode got from DECODE stage of previous cycle, it may be regarded as the primary phase in the total execution. This is a description of each sort of instruction in further detail:

**R type Instructions:** In this case, the ALU result is calculated using operand1 and operand2 by operating them with add, subtract, shift etc. and the result is stored in the ALUResult variable for the write-back and memory access procedure. In case of SLT like instructions ALU result is set to 1 or zero according the give conditions.

**I type Instructions** In this case, the ALU result is calculated using operand1 and operand2 by operating them with add in case of addi, and operator in case of andi , or in case of ori operation. Load operation’s ALUResult is also calculated in this stage and stored in ALUResult.

**S type Instructions:** In this case, the ALU Result gives the memory address by addition of operand1 and operand2.

**B type Instructions:** In this case, we subtract operand1 and operand2 and store in ALU Result and then determine if the branch needs to be taken or not according to the instructions.

**U type Instructions:** In this case, the ALU Result is calculated either by shifting operand2 by 12 bits or adding the shift to PC accordingly if the instruction is lui or auipc.

**J type Instructions:** This case involves only jal instruction. Here, the ALU result will always be zero.

In all these isBranch control signal is set on the basis that if the branch is taken or not. Result\_select control signal is also used and it takes value 0, 1, 2. Result\_select= 0 defines that we have to write ALU result to the destination register, result\_select=1 defines that we have to write PC+ imm to the destination register, and result\_select=1 defines that we have to write rs1+ immediate.

## Memory Access:

The process's fourth step is this one. At this stage, the result is read or loaded from the memory location that has been provided as incase of load and store instructions. As there is no need to write or load any data back to or from memory, nothing is done in this stage for the majority of instructions, including add, sub, and, or, div, rem, addi, andi, and many more. In such circumstances, the execution flow merely advances to the next stage. The relevant memory operations are carried out in the case of other instructions like lb, lh, and lw.

We have implemented Data Memory as a dictionary to read and write values . In lb, lh, lw the data is loaded from Data memory accordingly. For S type instructions, values are written in the data memory.

## Write Back:

The implementation of the instruction ends with this step. If the operation being carried out at the time calls for a register update, register updates are made in this phase. We have used the RF\_Write signal to control it. If the instruction is of type R, U, J, or I, the value of this variable is maintained as True; otherwise, if the instruction is of type S or B, the value is maintained as False. This variable indicates whether the current instruction requires a write-back operation.

Lastly, if the value of write back signal is equal to false when execution control reaches the write back() method, nothing is done. On the other hand, if the value is True, register data is updated in the target register (which contains the result of instruction execution obtained in the executing stage or obtained during the memory operation). The execution of an instruction is finished at this point.

Result\_select control unit is used to decide what needs to be written back in the registers.

Control signals are updated as and when required in all these stages.

Incase of stall in any stage, pipeline register values are passed to next stage and the function is returned.

When a stage is being executed, values are written to [i][0] index of that stage’s pipeline register and when the next stage executes, values from [i][0] are shifted to [i][1].

# Memory System – Main Memory and L1 Cache:

We have implemented three mapping: Direct mapping, Fully Associative mapping and Set-Associative mapping.

Our interface initially gives choice of mapping and takes input of Cache Size and Block Size.

We have created a separate memory.py file dealing with the memory system.

Initially, we are calculating all parameters of physical address i.e. tag bits, index/set bits and block offset.

Our two main functions are read and write which are checking for the presence of hit and miss, and accordingly taking further actions.

Incase of read miss, we are checking whether cache has empty space or not and thus accordingly adding or replacing block.

Incase of write, incase of a hit, we have implemented Write through and incase of write miss, we have implemented No Write Allocate.

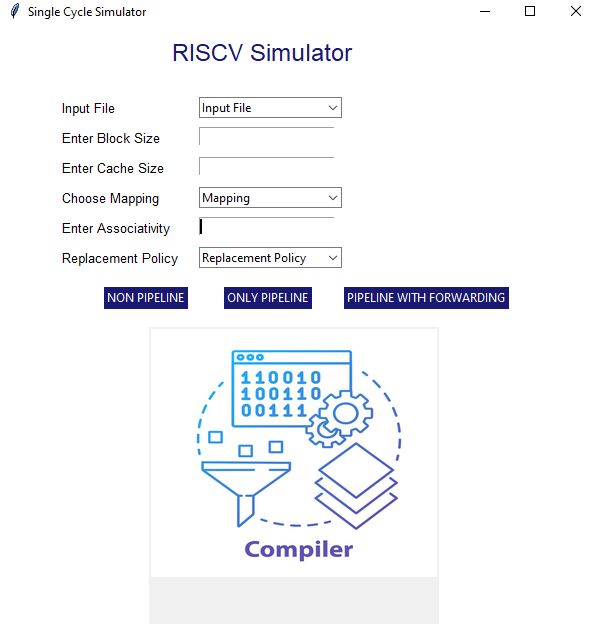
# Test plan

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].
* Bubble Sort

# GUI

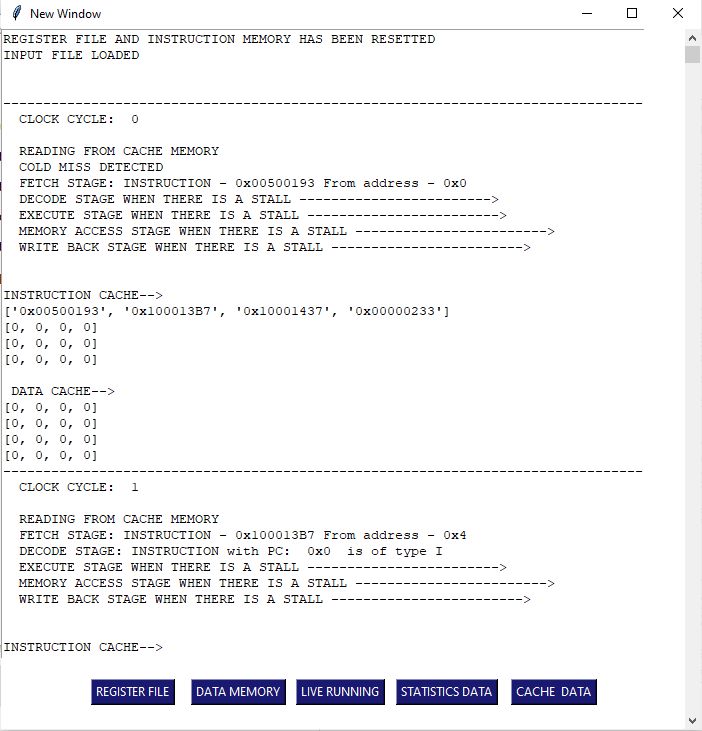
1. The first GUI window takes input of Test file, Cache size, block size, mapping, associativity and Replacement policy from the user.



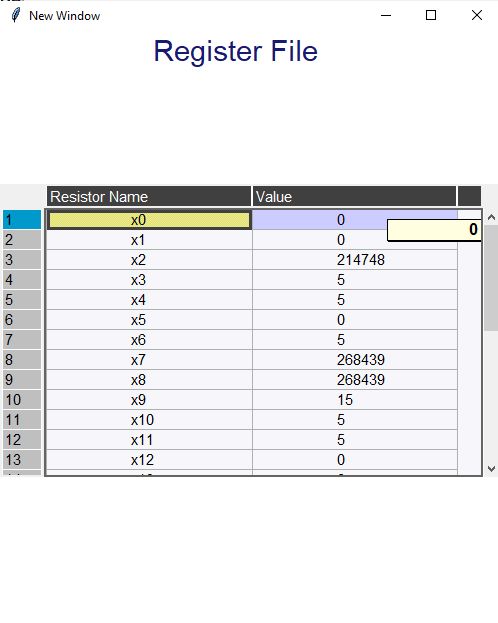
It also allows user to view the processing from Single-Cycle, Pipeline as well as Forwarding-enable Pipeline processing.

1. The instruction text Box appears after selecting the input file that contains all the stages the instruction is going through, along with the clock cycle and PC values.

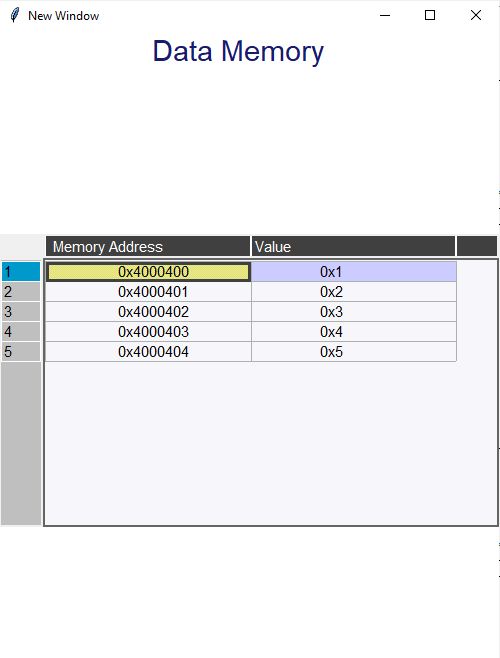
It also shows the presence of stall where ever present.



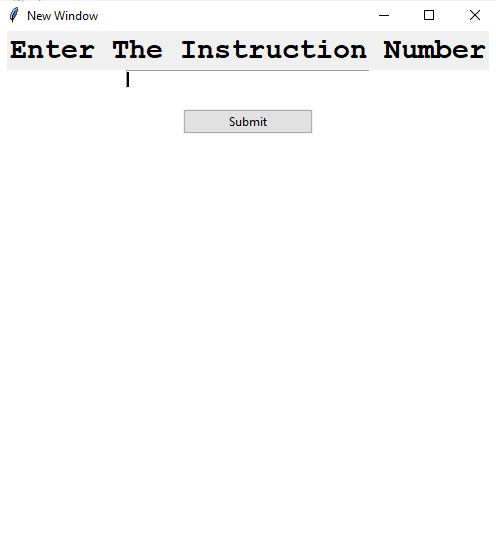
1. The Instruction text Box has five buttons at the end, for Register File, Data Memory, Live running of instructions, Cache data and Data statistics.
2. The Register File button opens the register File window containing the registers and their value.



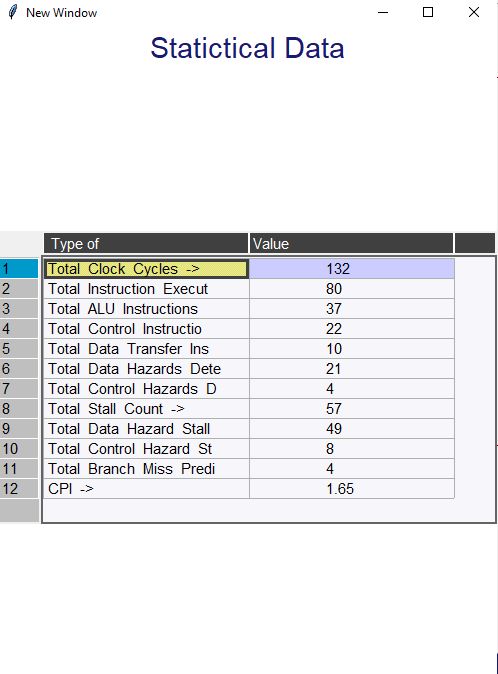
1. The Data memory contains the Memory registers along with their value.



1. The Live running button open the following window where you can the instruction number of the instruction from which you want to view the processing. By clicking on next and previous button, you can go to next and previous clock cycle respectively.



1. The Statistics data button opens up a window where you can view all the statistics of the processing like total clock cycles, total ALU instructions, total control instructions, etc.



1. The Cache data button opens up a window where you can view information related to Cache like total read misses, write misses, conflict misses, cold misses, etc.

