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

This document describes the design aspect of myRICSVSim, a functional simulator for a subset of the 32-bit RISC-V instruction set.

# Input/Output

## Input

Input to the simulator is a .mc file that contains the encoded instructions and the corresponding addresses at which the instruction is supposed to be stored, separated by a space. For example:

0x0 0x003100B3

0x4 0x00A00113

0x8 0x00200193

It also contains the data to be pre-loaded into the memory in a similar format where the least significant digits contain the data for the smallest address. For example:

0x10000000 0x00000010

0x10000004 0x00000020

## Functional Behavior and output

The simulator reads the instructions from instruction memory, decodes the instructions, reads the registers, executes the operations, and writes back to the register file as well as memory. The instruction set supported is the same as the one taught in the lectures.

The execution of instructions continues till it reaches the instruction “subw x1, x1, x1”. In other words, as soon as the instruction reads “0x401080BB”, the simulator stops and writes the updated memory contents and register contents onto two different .mc files.

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

* Fetch
  + “FETCH: Fetch instruction 0x003100B3 from address 0x0”
* Decode
  + “DECODE: Operation is ADD, first operand is R2, second operand is 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, memory, intermediate outputs of each stage of instruction execution are declared as global. A category-wise explanation is given below:

**Registers** - Registers are implemented using a python list. The whole register file is taken as a list having 32 elements representing 32 registers. These 32 values are initialized to 0. As the flow of the program proceeds forwards, these values are updated as per the use. The general format of each value is a string of hexadecimal numbers.

**Memory** - Memory is implemented using a python dictionary. This dictionary stores data as key-value pairs. In this case, the memory address is the key, and the data stored at it is the value. These key-value pairs are updated as per the need while taking input or during store instructions.

We also have a **data\_out.mc** file, which gets created when the program is run. This contains the values (initially all 0x00000000) at memory addresses ranging from “0x10000000” to “0x10007ffc” which in numerical terms corresponds to 268435456 and 268468220 (with a gap of 4). This is basically the data memory.

Similarly, a **reg\_out.mc** file is also created when the program is run. This contains the values in the registers.

Apart from this, various control signals are also defined.

**Intermediate output for each stage**

* **Fetch** - Here, the global variable instruction\_word is updated. instruction\_word will now contain the hex code of instruction to be executed.
* **Decode** - Here, we update several global variables which are to be used in the upcoming instructions. They are alu\_control\_signal, operation, operand1, operand2, rd, offset, register\_data, write\_back\_signal, is\_mem, etc. The detailed use of each of these variables is explained in the implementation section.
* **Execute** - Here, we update register\_data, memory\_address, is\_mem, etc. register\_data variable after this step contains the data with which the destination register needs to be updated or memory addresses for instruction types like store, load, etc.
* **Memory** - Here, we update the register\_data. Here the memory address (if there is a need to update) is updated with the values. Parallelly, the PC is also updated as part of IAG.
* **Write-back** - Here, according to True or False values of write\_back\_signal, the destination register is updated with the data.

In all these stages, control signals are also updated as and when required.

## Simulator flow

There are two steps:

1. First, the memory is loaded with an input memory file.
2. Second, the simulator executes the instructions one by one.

For the second step, there is an infinite loop, which simulates all the instructions till the instruction sequence reads “subw x1, x1, x1”.

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

# Implementation

## Fetch

This is the first stage of instruction implementation. In this stage, the instruction is fetched from memory using the Program counter, generally referred to as PC, which essentially contains that instruction’s address. After that, the control signals for PC update are set to default.

## Decode

The decode step is the second step of the overall implementation. In this step, the hexadecimal value which was previously fetched from the PC is decoded. In other words, information about the opcode, function 3, and function 7 is extracted, and we finally get to know which operation we need to perform and the concerned registers and immediate of the operation. Furthermore, the values of the registers are read if required.

For this step’s smooth execution, we have created a .csv file, namely Instruction\_Set\_List.csv, which contains a list of all the instructions we need to execute as a part of this project along with their opcodes, function 3, function 7, and their types. Using this file, we sequentially match the fetched instruction to all the columns until we get a matching result, which indeed is the operation to be performed, and thus we extract the rs1, rs2, rd, func3, func7, opcode, and imm for the desired type of instruction.

Also, we set the write\_back\_signal in cases where we need to write the data back in the destination register and set it to true in corresponding cases like R instructions, I instructions, U and UJ type instructions.

## Execute

This is the third step of instruction execution. It can be considered the main step of the overall execution because we now know what ALU operations are to perform. A detailed explanation for each type of instruction is given below:

**R type Instructions:** In this case, the hex values of rs1 and rs2 (denoting register1 and register2) are converted to integers, then they are operated upon (added in case of add, subtracted in case of ‘sub’, and similarly others), and the hex value of the result is stored in the ‘register\_data’ variable for the write-back procedure.

**I type Instructions:** In this case, the hex value of ‘rs1’(register1) is converted to an integer, and the binary value of the ‘immediate’ field is converted to integer as well and operated upon accordingly (added in case of addi, etc.), and the hex value of the result is stored in the ‘register\_data’ variable for the write-back procedure in case of ‘addi’, ‘andi’ or ‘ori’.

In case of instructions such as ‘lb’, ‘lw’, ‘lh’, the result is stored in the ‘memory\_address’ variable, and the flag ‘is\_mem’ is set accordingly for memory access and writeback procedure.

In the case of ‘jalr’ instruction, the control signals for PC update are set. The return address value is calculated and stored in the ‘register\_data’ variable for the write-back procedure.

**S type Instructions:** In this case, the hex value of rs1(source register) is converted to an integer, and the binary value of immediate value(offset) is also converted to an integer; they are added in order to get the final address of memory location(Base + Offset) and then the result is stored in ‘memory\_address’ variable. ‘is\_mem’ control variable list is also updated which here we are using as a flag for memory and write-back procedure. (e.g. ‘sw’, ‘sh’, ‘sb’)

**SB type Instructions:** In this case, we compare the values of targeted source registers. For this to happen, we converted the hex values of rs1 and rs2 to integer, and a comparison is made accordingly. And as per the comparison result, we update the control signals for the ‘PC’ update and set pc\_offset.

**U type Instructions:** In this case, the value of immediate is being assigned to the ‘register\_data’, and then the value is shifted left by 12 bits since auipc and lui both load just the upper 20 bits and make the lower 12 bits zero.

Here in the case of ‘auipc’ instruction, the integer value of PC is also extracted along with the integer value of register\_data, and they are being added upon, and the result is again stored in register\_data in hex format.

**UJ type Instructions:** This case involves only jal instruction. Here, the return address is stored in the ‘register\_data’ variable, and the PC control signals are set.

Now, to overcome overflow, only the last eight nibbles of the ‘register\_data’ are taken.

Finally, to fit the format, the extra 0’s are added to the ‘register\_data’ variable if required.

## Memory

This is the fourth step of the overall process. In this step, we write/load the result back to/from the specified location of the memory. In most of the instructions like add, sub, and, or, div, rem, addi, andi, and many others, nothing is done in this step as there is no need to write/load any data back to/from memory. In such cases, the flow of execution simply moves onto the next step while subsequently printing the message “No memory operation”. But in the case of other instructions like sb, lb, lh, lw, sh, and sw, the concerned memory operations are performed.

For proper execution of this, we initialize a list ‘is\_mem’ as [-1,-1]. This is the default value which indicates that no memory operation is to be performed. Further, the array is\_mem is modified in the execute stage so as to make the further classification of whether to read or write in memory easily. It is to be noted that such modifications are performed only in those types of functions, which need a memory operation. Other functions continue with the default value.

In load type instructions, the first value of this list is kept 0, and for store type instructions, it is kept 1. Further, for indicating byte, the value at the 2nd index is kept 0; for indicating halfword, it is kept 1, and finally 3 for indicating a word. Further, the register\_data is sign-extended in case data is loaded from memory.

Parallelly, the execution steps of the Instruction address generator(IAG), i.e., the PC update, are also performed in this step using the control signals updated in the previous steps.

## Write-Back

This is the final step in the instruction implementation. In this step, registers are updated if the current operation being executed requires a register update. So as to do this conveniently, we have taken a boolean global variable, namely **write\_back\_signal**. The value of this variable is being assigned in the decode stage, where if the instruction is of type R, U, UJ, or I, it is kept as **True,** and if the instruction type is S or SB, its value is kept **False**, indicating if or not the current instruction demands a write-back operation.

Finally, when the control of execution reaches the write\_back() function, if the value of write\_back\_signal is equal to false, simply a message “No write-back operation” is printed, whereas if the value is True, the destination register is updated with register data (which contains the result of instruction execution that is obtained in the executing stage or obtained during the memory operation). This completes the implementation of an instruction.

# Test plan

We test the simulator with the following assembly programs:

* Fibonacci Program
* Factorial Program
* Bubble Sort Program