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

Phase 2

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

# Input/Output

## Input file

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

## Input Knobs

The simulator contains five input knobs that work as per the following description:

**pipelining\_enabled**: This is a boolean knob that, if set, will consider pipelined execution; otherwise, it will sequentially execute the instructions.

**forwarding\_enabled**: This is a boolean knob which, if set, will work with data forwarding (If pipelineing\_enabled is set); else, it will consider stalling.

**print\_registers\_each\_cycle**: This is a boolean knob that, if set, will print the register values after each cycle in the terminal.

**print\_pipeline\_registers**: This is a boolean Knob that will print pipeline registers of each cycle along with cycle number if set.

**print\_specific\_pipeline\_registers:** This takes a list of 2 elements; the first element is boolean, which, if true, sets this knob. If set, the second element is an integer that tells us which instruction details to print. For example, if print\_specific\_pipeline\_register has value [True, 10], The details of the 10th instruction in the input file will be printed.

Here, if print\_specific\_pipeline\_registers is set, then print\_pipeline\_registers will not work. Also, it is assured that the cycle number is always printed when any one of the register file data or pipeline register(for one or all) is printed.

## Functional Behavior and output

The simulator reads the instructions from instruction memory, follows the knobs commands, decodes the instructions, reads the registers, executes the operations, and writes back to the register file and memory. The instruction set supported is the same as the one taught in the lectures. The execution supports pipelining, stalling, and forwarding, depending on the knob configurations.

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, registers contents onto two different .mc files and logs stats in stats.txt file.

## Output file

The simulator writes in three different output files:

**reg\_out.mc:** This file contains 32 lines storing registers(x0 - x31) and their values in hexadecimal format, space-separated. For example:

x0 0x00000000

x1 0x0ff2ac36

...

similarly up to x31

**data\_out.mc:** This file contains 8191 lines storing data memory from 0x10000000 to 0x10007ffc (both inclusive) and their values in hexadecimal format, space-separated just like reg\_out.mc.

**stats.txt:** It contains 12 lines containing the stats and their values. The following are the 12 stats printed in stats.txt:

* Stat1: Total number of cycles
* Stat2: Total instructions executed
* Stat3: CPI
* Stat4: Number of Data-transfer (load and store) instructions executed
* Stat5: Number of ALU instructions executed
* Stat6: Number of Control instructions executed
* Stat7: Number of stalls/bubbles in the pipeline
* Stat8: Number of data hazards
* Stat9: Number of control hazards
* Stat10: Number of branch mispredictions
* Stat11: Number of stalls due to data hazards
* Stat12: Number of stalls due to control hazards

# Design of Simulator

## Data structure

Registers, memory, some intermediate control signals are declared as class variables of the class **Processor**, which constitutes the whole Processing unit except for the Hazard detection Unit (**HDU**) and Branch Table Buffer (**BTB**). Further, each instruction is considered an instance of **State** class and contains various data associated with it and control signals.

**Registers** - Registers are implemented using a python list. The whole register file is taken as a list having 32 elements representing 32 registers. 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.

**Pipelining -** Pipelining is implemented by keeping an array of five stages of instructions called “states''. Here, states[4] represent fetch() stage, states[3] represent decode() stage, states[2] represent execute() stage, states[1] represent mem() stage, states[0] represent write\_back() stage. Each stage represents a different instruction and that particular stage for that particular instruction. For example if we have 5 instructions say i1, i2, i3, i4, i5, then, our pipelining list will contain write\_back() for i1, mem() for i2, execute() for i3, decode() for i4, and fetch() for i5.

**Intermediate output for each stage**

* **Fetch** - Here, the class variable instruction\_word of State class instance is updated. instruction\_word will now contain the hex code of instruction to be executed.
* **Decode** - Here, we update several class variables of State class instance, which are to be used in the upcoming stages. 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., of the State class instance. 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 stores, loads, etc.
* **Memory** - Here, we update the register\_data of the State class instance. 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.

## Branch Table Buffer (BTB)

A **BTB** class is defined to perform operations related to the branch table buffer. We are using a dictionary to simulate the branch table buffer. Here for every key-value pair, we are using Program Counter, PC as key, and a boolean value along with target address as value for that key.

For example, PC -> [boolean, target\_address] where PC is key and [boolean, target\_address] is the value. Here, we used boolean to track if the target\_address is Taken or Not Taken. If boolean is True, then target\_address will be Taken otherwise Not Taken.

## Hazard Detection Unit (HDU)

* **Data Hazards -** To detect data hazards and control hazards during the implementation of instructions, we have incorporated necessary changes in our main.py and myRISCvSim.py file. An **HDU** class is defined, which contains the functions for handling data hazards. We have two functions for handling data hazards. The argument passed to these functions is the set of instructions currently under operation (stored in variable pipeline\_instruction).

If forwarding is disabled (**data\_hazard\_stalling** function is used in this case to detect data hazards), and if the new instruction can not be added, a stall is added, else the new instruction is added to the set of instructions under implementation. On the other hand, if forwarding is enabled (**data\_hazard\_forwarding** function is used in this case to detect data hazards and do forwarding), we check for all the possible data hazards that could take place and the data forwarding that would solve them, namely E to E, M to E, E to D, M to D, and M to M. If there is a data hazard that can not be solved, we add it a stall to the pipeline. If there is no data hazard, the instruction is added directly to the instructions to be implemented list. The PC is updated accordingly.

* **Control Hazards -** There is no separate function for handling control hazards; instead, they are managed within the fetch and decode stage using the BTB class. In case, control hazards are there, appropriate stalls are added.

## Simulator flow

It consists of the following steps:

1. The memory is loaded with an input memory file, and the knobs are set using the GUI.
2. If pipelining is not enabled, instructions are executed one by one.
3. If pipelining is enabled
   1. Each instruction is added to the pipeline queue one by one for fetch. The queue has a length of 5, and every instruction is at a different stage at a time.
   2. Before each cycle, this queue is sent to the Hazard detection unit to check if it has a Data Hazard. If forwarding is not enabled, dummy instructions are added in between (stalling) to tackle the hazard; if forwarding is enabled, data is forwarded from produced instruction to consumer instruction, and a stall is added if the data hazard cannot be resolved.
   3. Each control/jump instruction is first added to BTB and predicted if reencountered using BTFNT static branch prediction model. Further, jumps are predicted as always Taken.

There is an infinite loop that simulates all the instructions until 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.

The above is done similar to as it was done in Phase 1 . For Phase 2 the additional things to be taken into consideration are the following . If the instruction is a dummy instruction , that is an instruction added just to handle hazards and not doing anything as specific , nothing is done . We simply skip to the next step . Secondly we also need to maintain a Branch Target Buffer (BTB) (a btb class has already been made specifically for this purpose and its functioning has been explained already above in BTB section). If the source instruction PC is there in BTB we predict accordingly whether the target branch will be taken or not and accordingly modify the next\_pc variable .

## 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.

In continuation of the above tasks performed in the Decode step in accordance with Phase 1 , we have the following additional things to take into account in Phase 2 . We simply skip to the next step if it's a dummy instruction . Secondly if the pipelined is enabled , and the instructions under being processed is decode instruction , we update the next pc . Moreover if the given instruction has already been encountered , i.e. its present in BTB , we update the next\_pc , check if our prediction is correct or not and accordingly modify the target associated with the current instruction in PC .

## 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”. However, 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 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. 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