Design Document: Functional Simulator for Subset of RISC-V Instruction Set

Phase 3

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 and cache memory (both instruction cache and data cache).

# 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 even if it is set. Also, it is assured that the cycle number is always printed when any one of the register file data or pipeline registers(for one or all instructions) is printed.

## Cache Inputs

The simulator also takes the following cache parameters as input:

1. **Cache Size**: The size of the cache
2. **Cache Block Size**: The size of the blocks in the cache
3. **Type of Cache**: Whether the cache is Direct Mapped or Fully Associative or Set Associative.
4. **Ways**: Number of ways in case of Set Associative Cache.

## 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. Now updated for Phase 3, the simulator now uses cache memory implementations - data cache (D$) and instruction cache (I$).

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 20 lines containing the stats and their values. The following are the 12 stats printed in stats.txt along with the 8 stats for cache implementation (4 stats each for data and instruction cache):

* Stat 1: Total number of cycles
* Stat 2: Total instructions executed
* Stat 3: CPI
* Stat 4: Number of Data-transfer (load and store) instructions executed
* Stat 5: Number of ALU instructions executed
* Stat 6: Number of Control instructions executed
* Stat 7: Number of stalls/bubbles in the pipeline
* Stat 8: Number of data hazards
* Stat 9: Number of control hazards
* Stat 10: Number of branch mispredictions
* Stat 11: Number of stalls due to data hazards
* Stat 12: Number of stalls due to control hazards

**Instruction Cache:**

* Stat 13: Number of read accesses
* Stat 14: Number of read hits
* Stat 15: Number of read misses
* Stat 16: Number of write-through no-write allocates

**Data Cache:**

* Stat 17: Number of read accesses
* Stat 18: Number of read hits
* Stat 19: Number of read misses
* Stat 20: Number of write-through no-write allocates

# Design of Simulator

## Data structure

Registers, memory, and some intermediate 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** - We have two types of memory in our Risc-V Simulator. One is the main memory, and the other is the cache memory.

For the purpose of Implementing the main memory (called “Mem”), we are 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.

Use of the cache memory is elaborated in the “Implementation of the cache” section, which describes the use of the Memory class and the various features it provides. The main memory is primarily initialized in the load\_program\_memory function.

**Pipelining -** Pipelining is implemented by keeping an array of five stages of instructions called “pipeline\_instructions''. Here, pipeline\_instructions[4] represent fetch() stage, pipeline\_instructions[3] represent decode() stage, pipeline\_instructions[2] represent execute() stage, pipeline\_instructions[1] represent mem() stage, and pipeline\_instructions[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, 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/data for instruction types like stores, loads, etc.
* **Memory** - Here, we update the register\_data of the State class instance for some instructions. 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.

## Implementation of Cache

We have made a “memory” class that simulates the working of cache memory. In order to make our work more organized, we have implemented a class and then initialized two instances of the object of memory class, one for data cache and one for instruction cache, respectively, since the two cache memory behaves identically.

We are using an array of Dictionaries in python to help us simulate a cache memory where each dictionary contains a “tag address” as key and (block, recency) as its value. So, for each index in the array named “cache”, we are maintaining dictionaries where { tag : ( block, recency ) } is the { key : value } pair.

**Policies Used in Cache Implementation**

1. **Type of Cache** - The simulator supports Direct Mapped cache, Fully Associative cache, and Set Associative cache.
2. **LRU Policy** - In cache implementation, we are using LRU policy (Least Recently Used). We are keeping track of the least recently used block by maintaining a recency list.
3. **Write Through Policy** - For writing to the cache, we are using a write-through technique which means we also write to the lower level after the cache, which in our case is the main memory.
4. **No-Write Allocate Policy** - For writing to the cache, if the address is found in the cache, we write to it along with the main memory. However, if the address is not found in the cache, we simply write to the main memory directly.

**Data Cache -** We created an instance of the memory class called data\_cache in main.py. This simulates the data cache memory (D$).

**Instruction Cache -** We created an instance of the memory class called instruction\_cache in main.py. This simulates the instruction cache memory (I$).

## Classes Implemented

In this phase of the project, we have implemented classes to make our program more readable and organized.

## State

The **State** class creates instances of the current state of instruction. This helps us in organizing and keeping track of the progress made by a single instruction. We have created all required variables inside this class that help us in identifying an instruction, accessing all the interstate registers of that class, ALU controls, and Memory access and writeback controls for that instance of an instruction.

## 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.

We have implemented BTB as BTFNT static prediction. Further, jumps are always considered as 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 next instruction is directly added to the instructions 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.

## Memory

The memory class present in the memory.py file is made specifically to assist the cache implementation. We use this to implement both the data cache and the instruction cache. Using this class, we make an instance of the cache memory of the desired cache size, block size, and associativity. These are present as class variables in the class. In this class, we have used several functions for performing different tasks and for keeping things systematic.

* The “set” function calculates the number of sets and the number of index bits based on associativity, block size, and cache size. Further, it initializes the cache array.
* The “get\_index” function extracts and returns the index corresponding to a memory address.
* The “get\_tag” function extracts and returns the tag corresponding to a memory address.
* The “get\_block\_offset” extracts and returns the block offset corresponding to a memory address.
* We have a function “replace\_block”, to assist block replacement based on the LRU policy.
* Each time cache is accessed, there is a change in the recency values of elements present in the cache. This process of updating is done in the “update\_recency” function.
* The “add\_block” function adds the new block to the cache if they are already not present in there.
* The “make\_table” function packs the whole cache into a nested list format to pass it to gui.py for showing cache in GUI.
* Finally, we have the “read” and “write” functions that help us perform the read and write operations on the cache.

# Simulator flow

It consists of the following steps:

1. The memory is loaded with an input memory file. The input knobs and the input cache parameters 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 (**HDU**) 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 added to BTB on its first occurrence and predicted if reencountered using BTFNT static branch prediction model.

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 functions.

# Implementation

In phase 3, changes have only been made to fetch and memory functions, as they are the one which requires access to instruction cache and data cache respectively.

## 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.

The above is done similar to as it was done in Phase 1. For Phase 2, we also need to maintain Branch Target Buffer (BTB) (a BTB class has already been made specifically for this purpose, and its functioning has been explained above in the 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.

Unlike the previous stages, in phase 3, the fetch stage fetches the instruction from the instruction cache and stores it in a data variable. Other things like handling of BTB, updating the next\_pc variable are done as they were done previously.

## 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:

1. If the instruction is a control instruction, the execute stage must be done in this step itself.
2. Moreover, if the given instruction has already been encountered, i.e., it is present in BTB, we check if our prediction is correct or not and accordingly modify the target associated with the current instruction in PC. If it is not present in BTB, we execute it, get the target address, and add it to the BTB also.

## 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. 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 instructions, which need a memory operation. Other instructions 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 Instruction address generator(IAG) is also called to get the next PC.

Additionally, in phase 3, if the instruction is a Load instruction, then the data corresponding to the required memory location is fetched from the cache and stored in a data variable. If the instruction is a store instruction, the cache and the memory are updated corresponding to write-through no-write allocate policies. The rest of the tasks continue as previously described.

## 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 State class boolean 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, nothing is done. In contrast, if the value is True, the destination register is updated with register data (which contains the result of instruction execution obtained in the executing stage or obtained during the memory operation). This completes the implementation of an instruction.

Additionally, if the instruction is a dummy instruction(which can be marked by a State class variable), we simply skip to the next step. This instruction is added to handle hazards and not do anything specific.

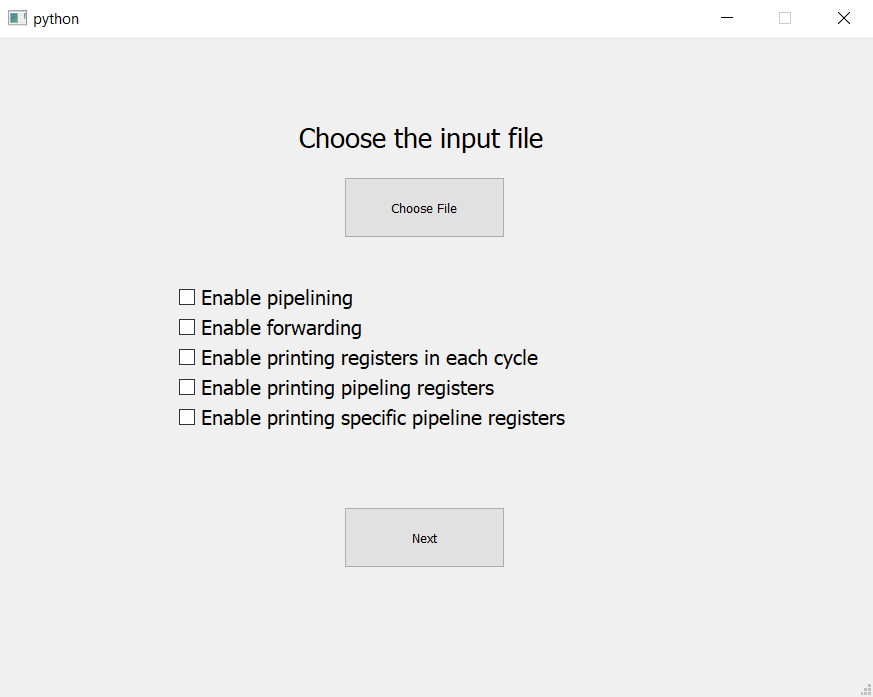
# Test plan

We test the simulator with the machine code of the following assembly programs:

* Fibonacci Program
* Factorial Program
* Bubble Sort Program

# GUI

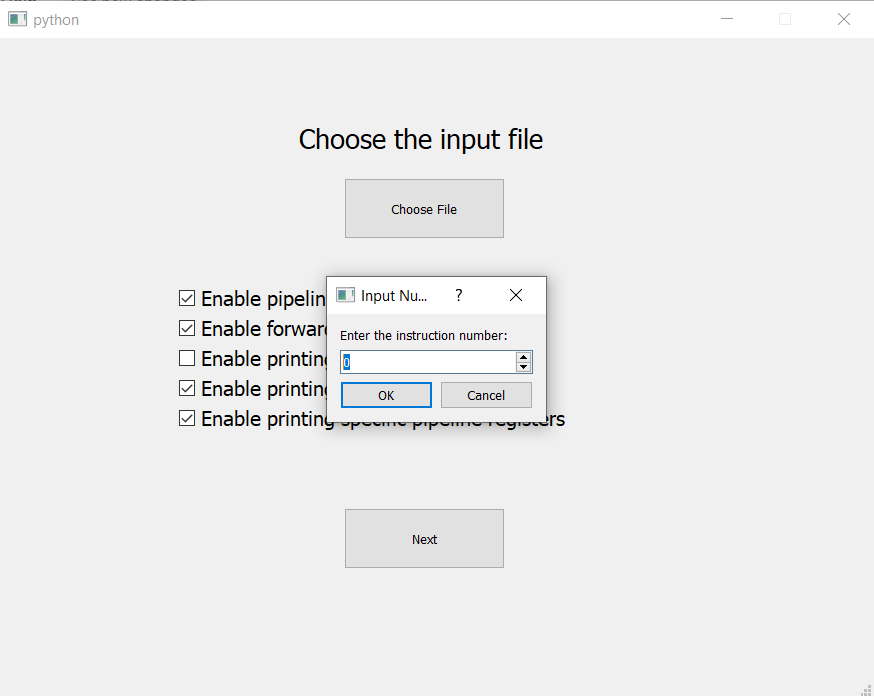
1. The first GUI window allows the user to select the input file present in the test directory.



On selecting “Choose File”, all the files in the test directory will be displayed, out of which the user can select anyone.

1. The user can also select various knobs. By default, the simulator executes multi-cycle implementation.

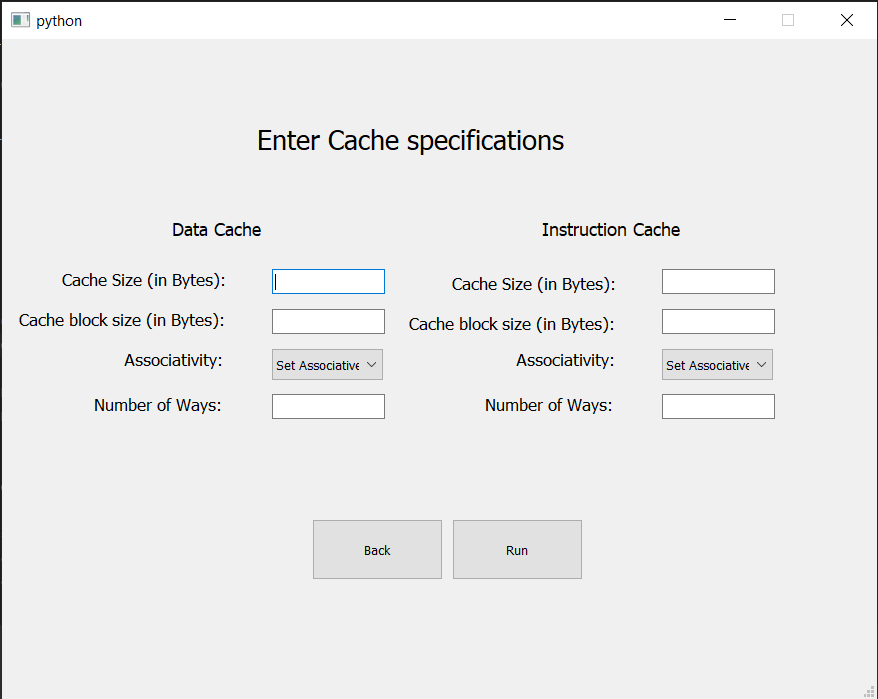
On selecting the last knob, i.e., “Enable printing specific pipeline registers”, a dialog box will be displayed. The user has to enter the instruction number for which the pipeline registers need to be printed. The instruction should be one of the instructions from the input file counted as 0, 1, 2, …



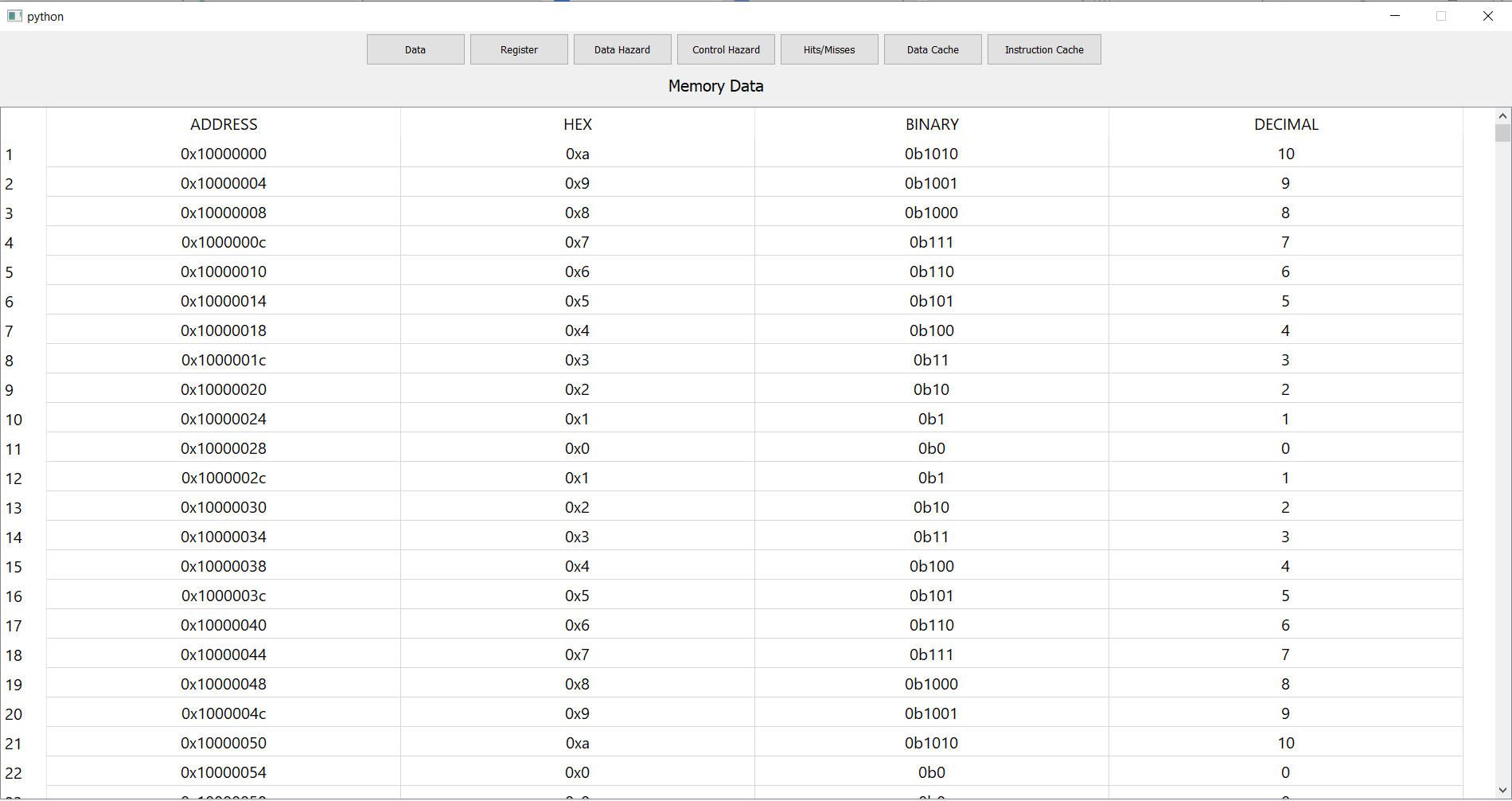
After selecting the input file and the input knobs, the user needs to select “Next” to go to the next window.

Here, it should be noted that knob 4 will work if and only if knob 5 is not set and knob 4 is set.

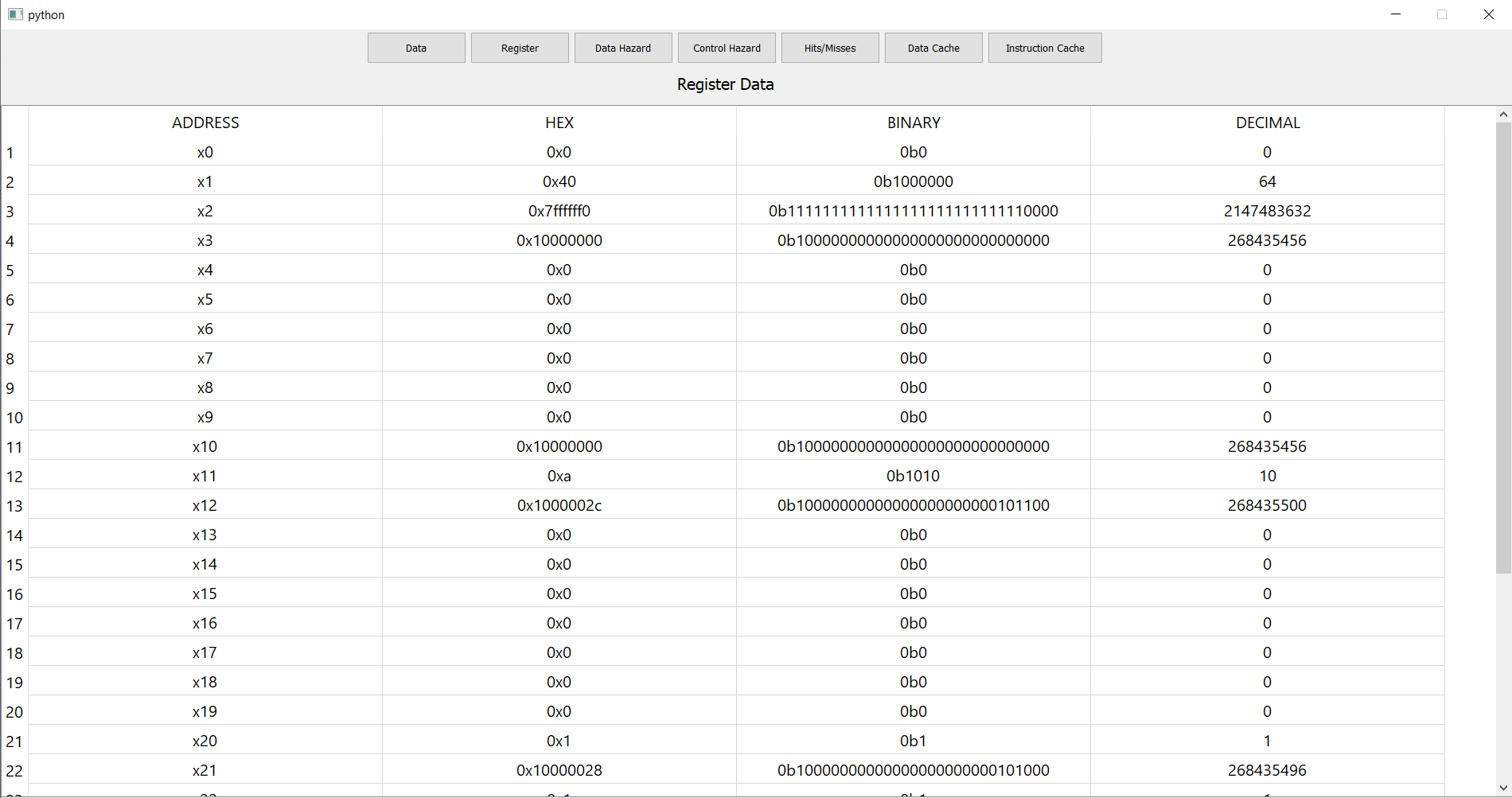
1. The next GUI window asks to enter cache specifications in bytes. Specifications of I$ and D$ can be different. If you have not set the values and pressed the “run” button, the Simulator will assume default settings. The default value of cache size, cache block size, associativity, and the number of ways is 128B, 4B, set-associative, and 2, respectively.



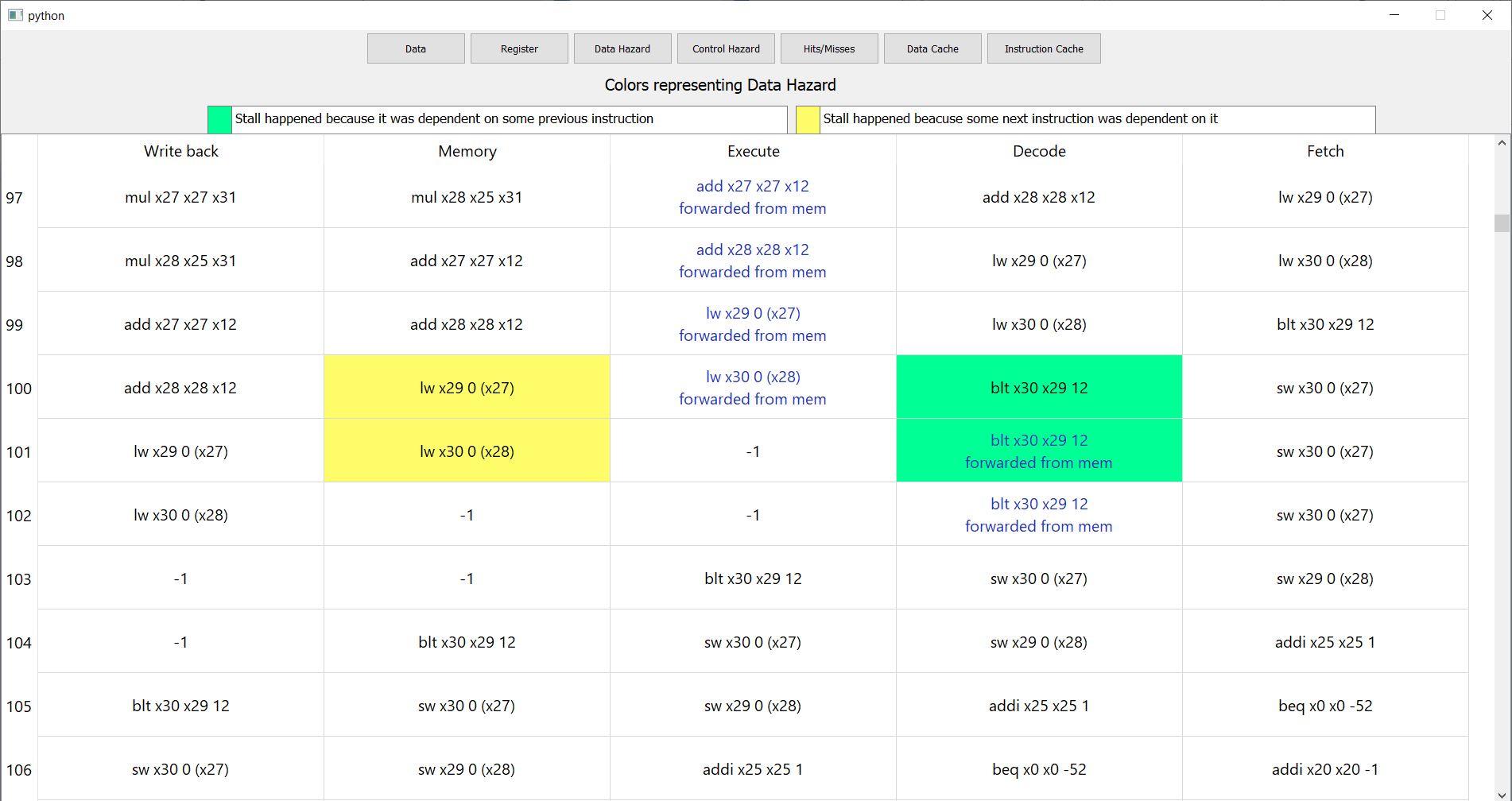
1. After the execution, a GUI window opens again, showing different data corresponding to input program execution like updated data memory, updated register file, data hazards, forwarding, control hazards, branch predictions, I$ cache, D$cache, cache update of I$ in fetch stage, cache update of D$ in mem stage and hits and miss in every clock cycle. For this purpose, it has seven buttons at the top.
2. The first button(“Data”) shows the updated data memory contents in binary, hexadecimal, and decimal format along with the memory address. In each address, a word is displayed with the content of the smallest address in the least significant position and so on.



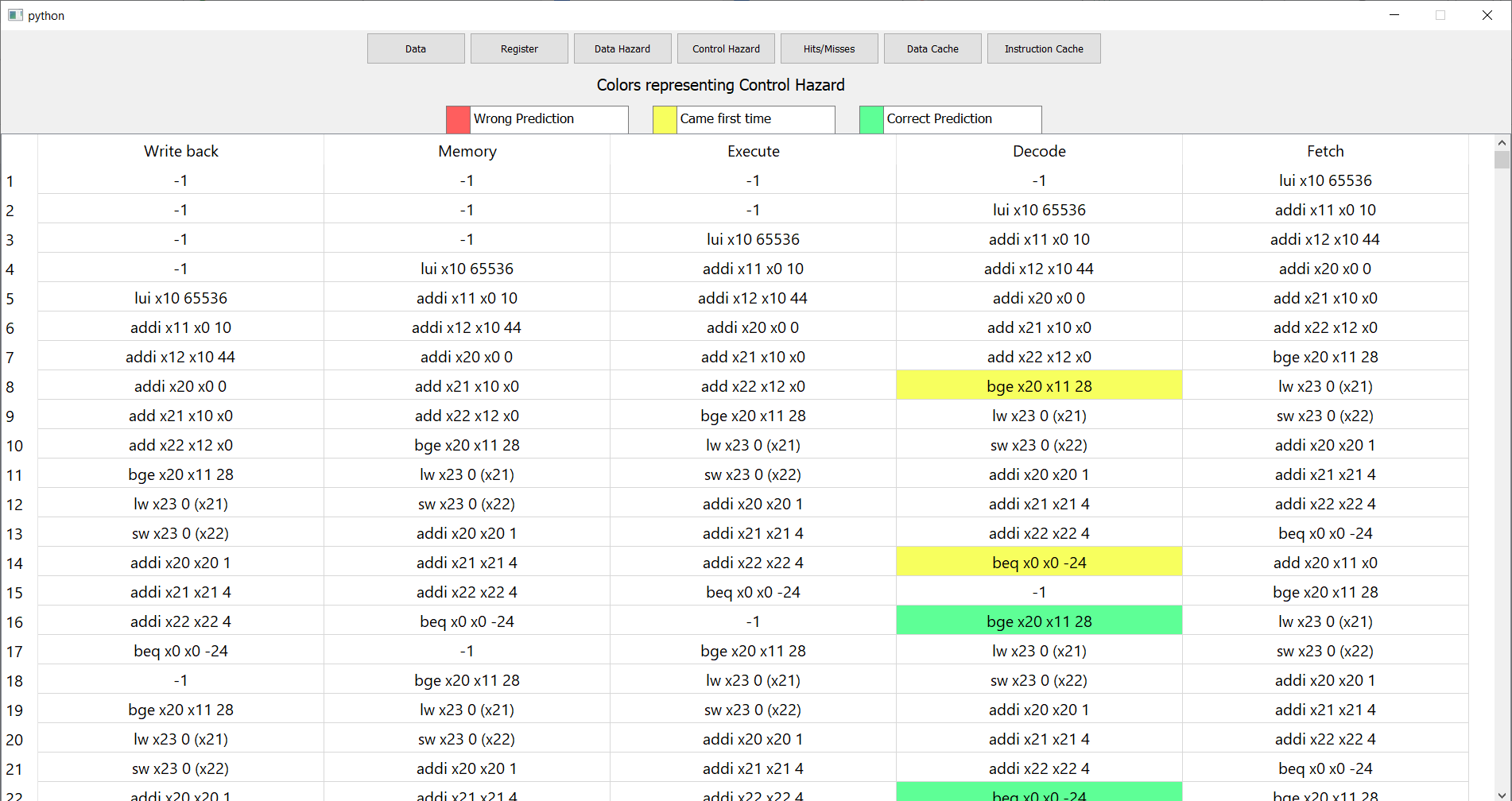
1. The second button(“Register”) shows the updated register file contents in binary, hexadecimal, and decimal format along with the register number. The data is from MSB to LSB from left to right.



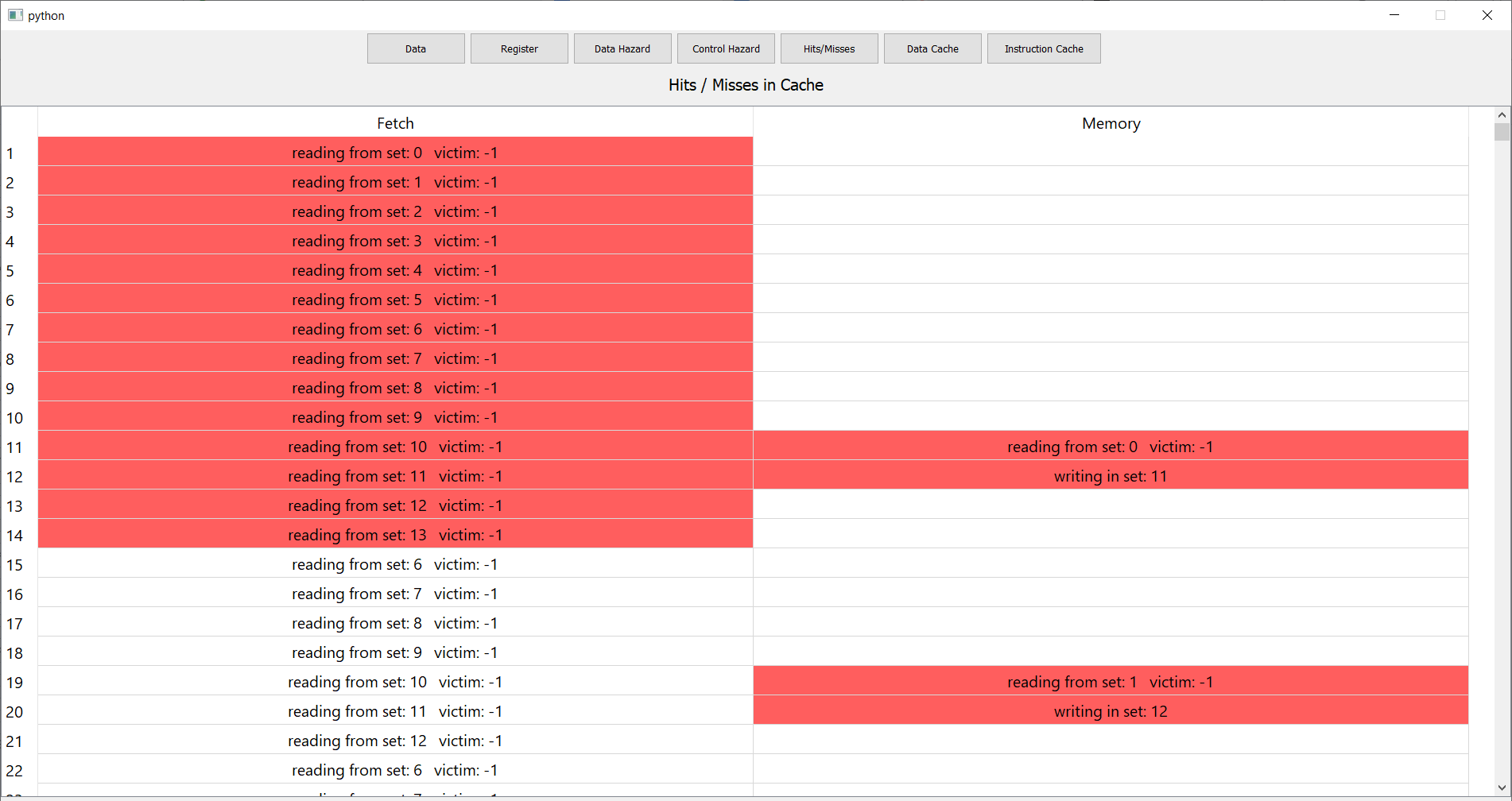
1. The third button(“Data Hazard”) shows the stage of the pipeline instructions, data hazards, and the forwarding paths.
2. Two colors show the instructions, which have a data hazard, and a stall will occur due to them.
3. An instruction in green color represents that the instruction was dependent on a previous/earlier instruction.
4. An instruction in yellow color indicates that some next instruction is dependent on it.
5. Further, below the instructions, a forwarding path is given if forwarding has happened for that instruction in that stage.



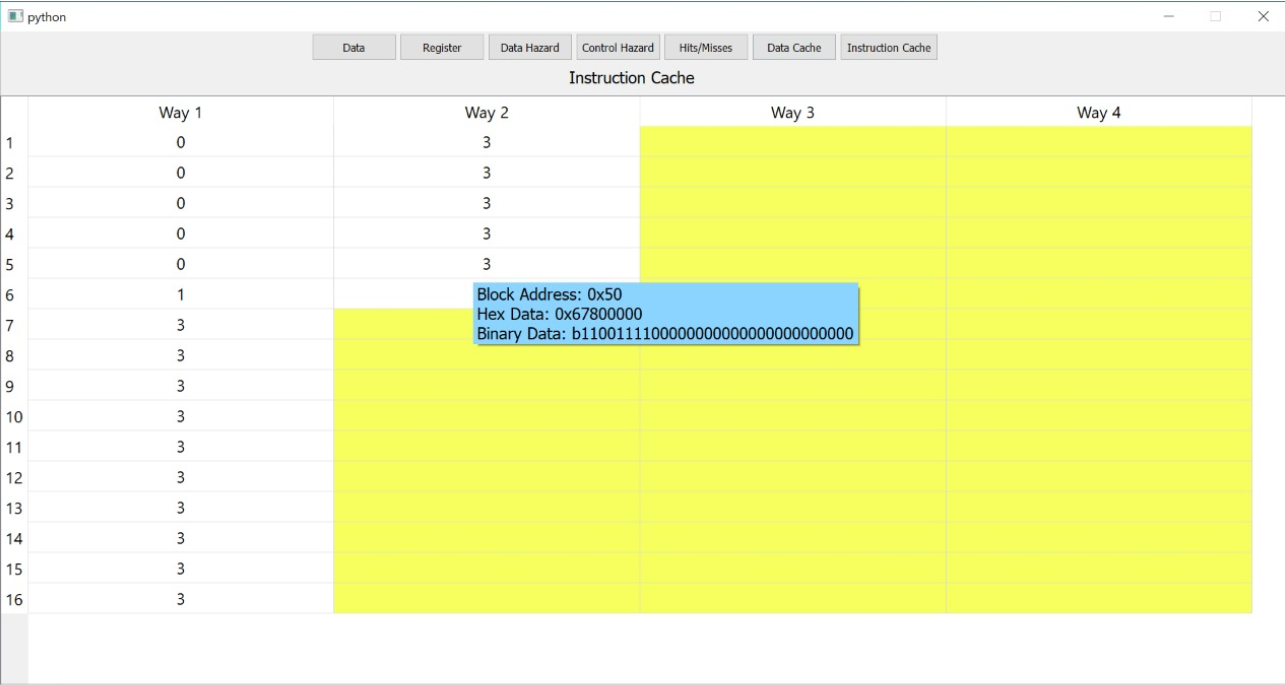
1. The fourth button(“Control Hazard”) shows the stage of the pipeline instructions, control hazards, branch predictions, and the corresponding stalls.
2. Three colors show the outcome of the branch prediction.
3. A green color indicates a correct prediction.
4. A yellow color indicates that the control instruction is encountered for the first time. A stall will follow it if the branch is Taken and the target address is not equal to the next address.
5. A red color indicates a branch misprediction. A stall will follow it.



9. The fifth button (Hit/Misses) shows the use of cache in every cycle by fetch (I$) and mem stages(D$). Red color indicates misses in both read and write. In case of a miss, if there is not a victim, victim: -1 is printed.



10. The sixth button (Data Cache) and Seventh button (Instruction Cache) prints the final face of the Data cache and Instruction Cache after the program’s execution. Rows represent the sets, and columns represent the ways. The numbers in the cells are the recency, a bigger number means a more recent block. If the user wants to know the data and address of any cell, hover the cursor over the cell to view its data and address. The yellow cells are unused cells whose dirty bit is still 0 after execution.



Further, the user can manually change the width of the rows and columns in the GUI.