

INDIAN INSTITUTE OF TECHNOLOGY, ROPAR

CS204 Course Project - Phase III

Team Members:

1. Aditya Agarwal - 2019CSB1064
2. Aneeket Mangal - 2019CSB1071
3. Fadia Het Rakeshkumar - 2019CSB1084
4. Shikhar Soni - 2019CSB1119
5. Tanmay Aeron - 2019CSB1124



Project Description

The aim of this project is to simulate the machine level execution of RISC V 32-bit instructions using a high level language.

The Project also aims to give updates to the user regarding each step of the execution of the program. It also returns the final status of the memory and registers as output for the user to analyse the working of their programs thoroughly.

The Project currently allows the user to use 29 different instructions and can be extended to allow the use of any number of instructions by editing the .csv files as long as the instructions are supported by 32-bit RISC V ISA.

The program supports the use of 'knobs' which will enable the user to select whether or not they want pipelining, data forwarding, and other details to be printed at the end of running the program.

The program executes each instruction using five stages as described in the RISC V architecture.

There are 5 separate knobs, all with a different purpose.

1. **Knob1:** Used to enable/disable pipelining during runtime.
2. **Knob2:** Used to enable/disable data forwarding
3. **Knob3:** Used to enable/disable printing values in the register file at the end of each cycle.
4. **Knob4:** Used to enable/disable printing information in the pipeline registers at the end of each cycle, along with the cycle number.
5. **Knob5:** This is like enabling knob4 for a specific instruction. We will be able to see pipeline register for that particular instruction only.

The program also allows for the capability of a simple cache integrated with the previous phases of the project.

The cache is implemented using the concept of LRU(Least Recently Used) in which the data that was used least recently is deemed the victim block, and evicted from the cache.

There are two different caches:

1. Instruction Cache (I\$)
2. Data Cache (D\$)

All data during the Fetch stage is handled by the I\$ and all data during Memory Stage is handled by the D\$.

Technologies employed

Back-end - Python3

1. **pandas** for reading **.csv** files.
2. **os** for getting and adding path to certain file locations.
3. **collections.defaultdict** to make a hash map for memory.
4. **sys** for reading and editing files with ease.
5. **json** for crunching data
6. **glob** for file management

Front-end - Python3

1. **PyQt5** for the Graphic User Interface.
2. **qdarkstyle** for dark theme in the GUI.

How to run the code:

Instructions to run GUI version:

python main.py -g

After running this command in the terminal, the GUI will pop up.

You can then enable each knob by ticking the checkbox for each. For the fifth knob you can add the instruction number by adding the instruction in the box.

Information about the Fetch, Decode, etc. can be seen in the info tab on the right side when you click on the respective buttons in the DataPath tab

Instructions to run Non GUI version:

add '-k1' to enable knob1

add '-k2' to enable knob2

add '-k3' to enable knob3

add '-k4' to enable knob4

add '-k5 instruction number' to enable for that particular instruction

add '-ICache cachesize blocksize ways' for setting instruction cache details

add '-DCache cachesize blocksize ways' for setting data cache details

If you want to run a particular file, add that file in test folder and add '-f filename', otherwise main.mc is run

For example, if you want to enable knob1, knob2 , knob3, knob4 and knob5 with instruction 3, and make a data cache of size 512, block size 8 and ways to be equal to 2, and run fib.mc, write

python main.py -k1 -k2 -k3 -k4 -k5 3 -DCache 512 8 2 -f fib.mc

Other features:

Arguments: [-h] [-g] [-f F] [-k1] [-k2] [-k3] [-k4] [-k5 INS] [-ICache cacheSize blockSize noOfWays] [-DCache cacheSize blockSize noOfWays]

Description:

1. [-h, -help] show help message and exit
2. [-g, -gui] enable GUI
3. [-f F, -filename F] specify file which is to be run, defaults to main.mc
4. [-k1, -knob1] enable Pipelining
5. [-k2, -knob2] enable Data Forwarding
6. [-k3, -knob3] show value in registerFile at end of each cycle
7. [-k4, -knob4] show value in Pipeline Registers at end of each cycle
8. [-k5 INS, -knob5 INS] show value in Pipeline Registers at end of each cycle for particular instruction
9. [ICache cacheSize blockSize noOfWays] configure input cache in format cache size, block size and number of ways in Instruction Cache.
10. [DCache cacheSize blockSize noOfWays] configure data cache in format cache size, block size and number of ways in Data Cache. If not specified both of the above default to 64 bytes (cache size), 4 bytes (block size), 2 (way set associative).
11. [-h] for help on running the code

Implementation Details

We have implemented the project using the following method:

First it takes the input from a **.mc** file that contains the input in the required format, the content of this file can be modified using the GUI editor or by changing the main.mc file.

Then, it stores the **.data** part into the data memory and the **.text** part in the text part of the memory. Once that is completed, we run the program using the following method:

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.

We have implemented data forwarding by passing the required values from the source buffer to the destination buffer without the need of stalling in most cases. This helps to reduce the number of stalls required. The types of data forwarding are:

1. Execute to Execute
2. Memory to Execute
3. Memory to Memory

The program detects hazards present which can cause an incorrect value to be register due to one instruction's stage being executed before another. This is fixed by either transferring data values directly with the help of data forwarding, or with the help of stalling which delays the next instruction so that the previous one is updated in time.

While data forwarding is enabled, there is no use of stalling except for one particular case where we use it along with Memory-to-Execute data forwarding.

Input/Output format details

Input Format:

```
0x0 0x00500513
0x4 0x008000EF
0x8 0x0440006F
$
0x10000000 0x64
```

The input is given through GUI if the GUI is run else it is taken from the input file.

The section given before the '\$' sign is the text segment of the code and all the lines after it signify the data segment of the code.

Output is generated in the 'generated' folder. It contains the following things:

1. 'registers.txt' contains the register values
2. 'memory.txt' contains the data part of the memory
3. 'stats.txt' contains the statistical details of the program
4. 'outputLog.txt' contains the buffer data of each cycle
5. 'buffer.txt' contains buffer data about the instruction given after k5 knob
6. 'CacheInfo.txt' contains information about the cache
7. 'DataCache.txt' contains information about the data cache
8. 'InstructionCache.txt' contains information about the instruction cache
9. 'forwarding.txt' contains information about forwarding used for front end information
10. Buffer Snapshot and Register Snapshots folders are used for front end information

Example output for Buffer.txt

Fetch: ('00000000', '00400513', '00000004')

Decode: (9, '00000000', '00000000', '00000000', '00000000', 10, 0, 4, '00000004', '00000004', [[False, 'NO', 0], [False, 'NO', 0]])

Execute: (9, '00000004', 10, '00000000', 0, 4, '00000004', '00000000')

MemoryAccess: (9, '00000004', 10, '00000000')

Example output for Forwarding.txt

"EE1": "F", "EE2": "F", "ME1": "F", "ME2": "F", "MM": "F"

Here EE1:F means forwarding of Execute to Execute between prev instruction(1 means previous instruction) and current instruction is False

Similarly ME2: T means forwarding from Memory to Execute from previous to previous instruction(2 means previous to previous instruction) is True

CacheInfo.txt

'F' stands for Fetch, 'L' stands for Load and 'S' stands for Store

In the Cache Info in Fetch the index, Set, isMiss and Victim block is printed

Here the isMiss = T means isMiss is True and so we found a miss

Similarly if Fetch or Load or Store is -1 means the corresponding instruction is not executed.

The user can specify input configurations for the Instruction Cache and Data Cache separately. Outputs for the two will also be displayed separately.

Input parameters that can be set by the user include:

1. Cache size
2. Cache Block size
3. Number of ways for Set Associative

Output parameters include:

1. Number of accesses
2. Number of hits
3. Number of misses

In *stats.txt* we have printed the following:

1. Total number of cycles
2. Total number of instructions executed
3. CPI
4. Data Transfer instructions executed
5. ALU instructions executed
6. Control instructions executed
7. Total Stall Count
8. Number of Data Hazards
9. Total Control Hazards
10. Total Branch misprediction
11. Number of stalls due to data hazards
12. Number of stalls due to control hazards

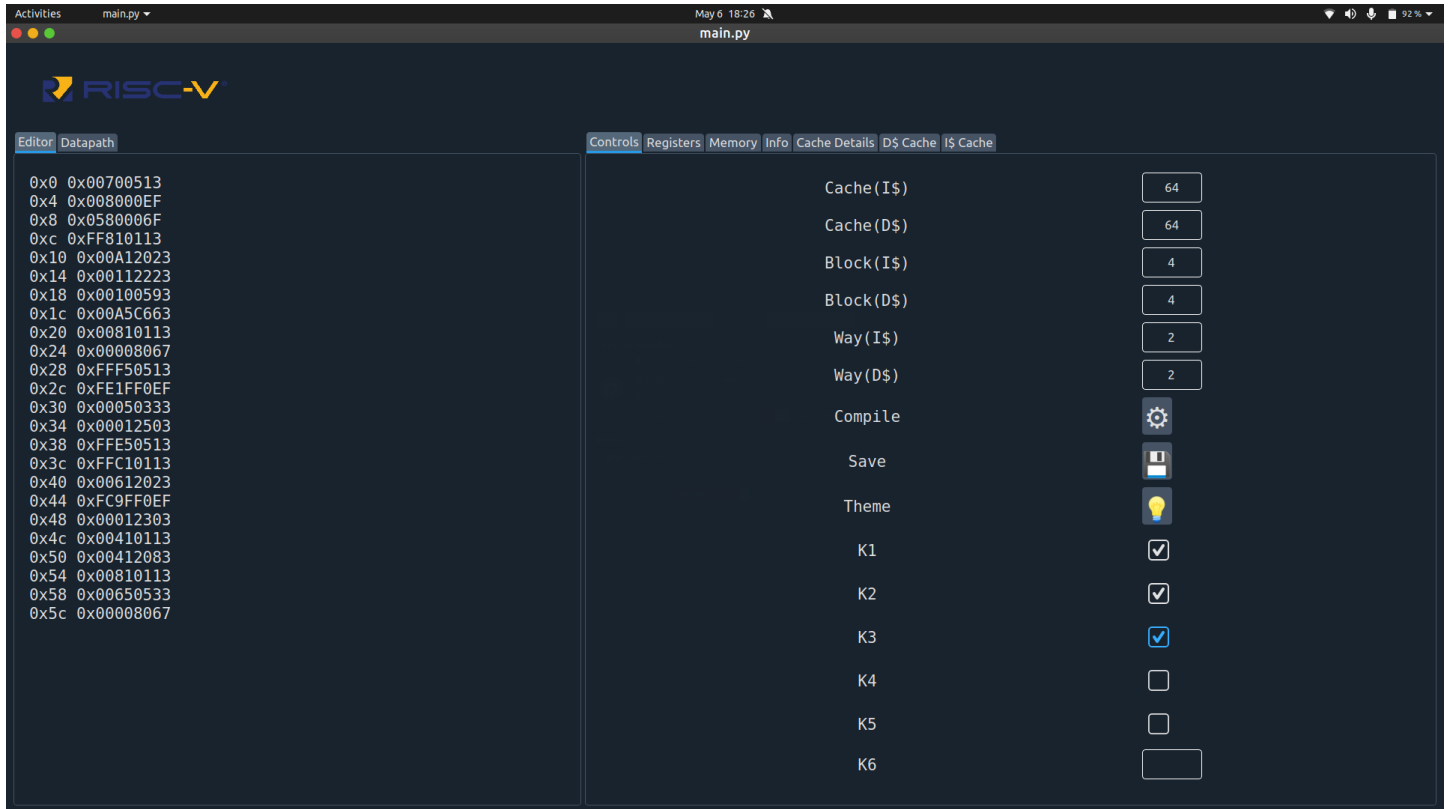
An example output for *stats.txt* is :

Total number of Cycles in the program :14
Total number of Instructions executed :4
CPI is :3.5
Data Transfer Instructions Executed :0
ALU instructions are :4
Control instructions :0
Total Stall Count are :6
Number of Data Hazards are :3
Total Control Hazards are :0
Total Branch mispredictions are :0
Number of stalls due to data hazards are :6
Number of stalls due to control hazards are :0

The instruction fed was:

0x0 0x00A50513
0x4 0x00A50513
0x8 0x00A50513
0xC 0x00A50513

Graphical User Interface



Above is the GUI image, the tab on the right is the 'controls tab' and the user can change the knobs and cache configurations accordingly (cache size must be greater than the block size and the cache size should be a power of 2 along with the block size being a multiple of 4 to ensure word alignment).

We can input the Cache Size, Block Size (in bytes) and Ways on the right side of the page.

We can enable the knobs by ticking the corresponding boxes, for knob5 one needs to enter the instruction number on the 'k6' block (if no number is entered then it defaults to 1). We can compile the code by pressing the 'Settings/Compile' icon corresponding to compile, there's also the save icon below it represented by a 'floppy disk'.

We can also change the theme by clicking the 'bulb button' to change from light to dark mode and vice versa. When we press the compile button the code in the editor gets saved in main.mc and the code will be executed.

Activities main.py May 6 18:26 main.py

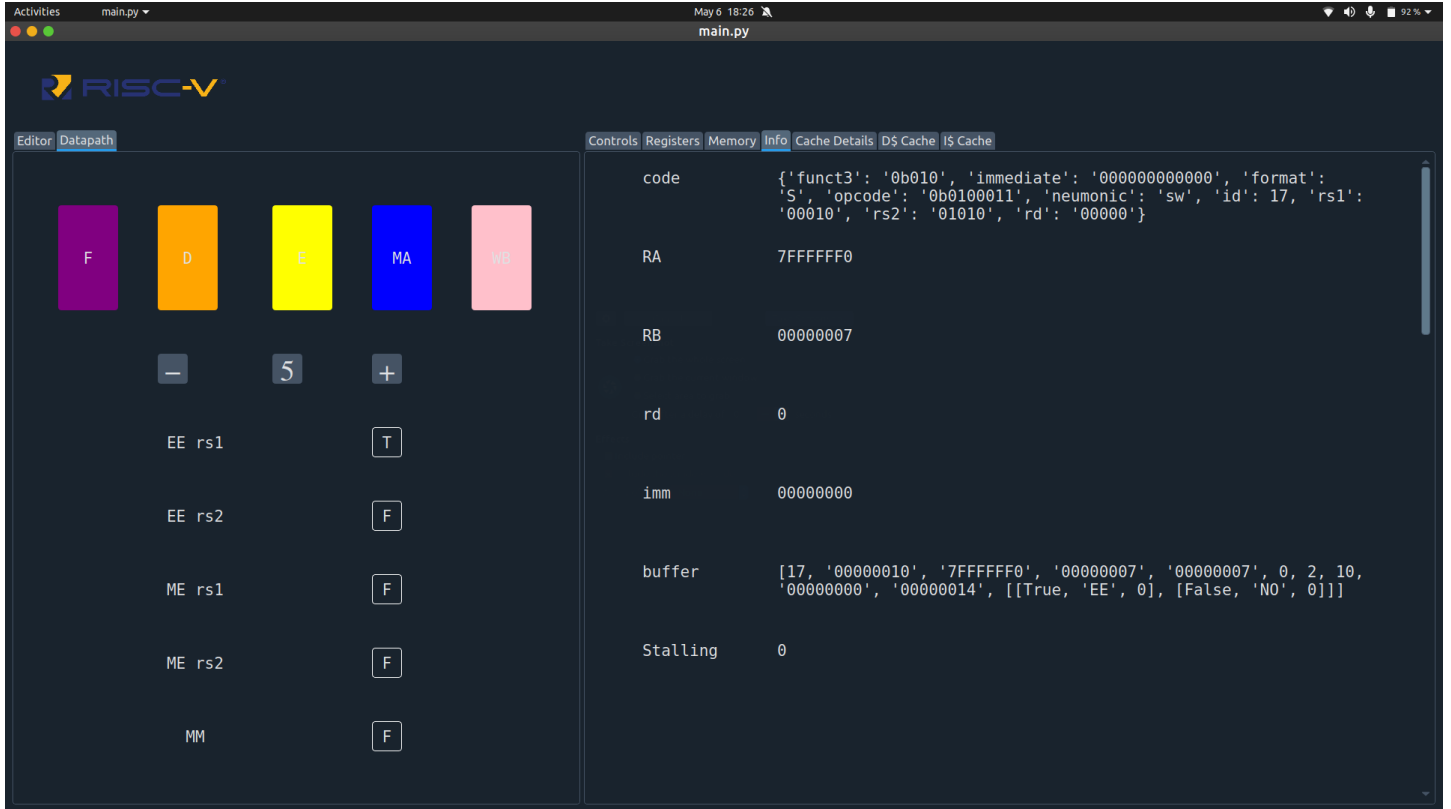
RISC-V

Editor Datapath Controls Registers Memory Info Cache Details D\$ Cache I\$ Cache

0x0 0x00700513	x0	zero	0x00000000
0x4 0x008000EF	x1	ra	0x00000008
0x8 0x0580006F	x2	sp	0x7ffffff0
0xc 0xFF810113	x3	gp	0x10000000
0x10 0x00A12023	x4	tp	0x00000000
0x14 0x00112223	x5	t0	0x00000000
0x18 0x00100593	x6	t1	0x00000008
0x1c 0x00A5C663	x7	t2	0x00000000
0x20 0x00810113	x8	s0/fp	0x00000000
0x24 0x00008067	x9	s1	0x00000000
0x28 0xFF50513	x10	a0	0x0000000d
0x2c 0xFE1FF0EF	x11	a1	0x00000001
0x30 0x0050333	x12	a2	0x00000000
0x34 0x00012503	x13	a3	0x00000000
0x38 0xFFE50513	x14	a4	0x00000000
0x3c 0xFFC10113	x15	a5	0x00000000
0x40 0x00612023	x16	a6	0x00000000
0x44 0xFC9FF0EF	x17	a7	0x00000000
0x48 0x00012303	x18	s2	0x00000000
0x4c 0x00410113			
0x50 0x00412083			
0x54 0x00810113			
0x58 0x00650533			
0x5c 0x00008067			

We can see the values stored in all the Registers here. Values are stored in Hexadecimal format.

User can scroll down to observe the registers that are not immediately visible on the screen.



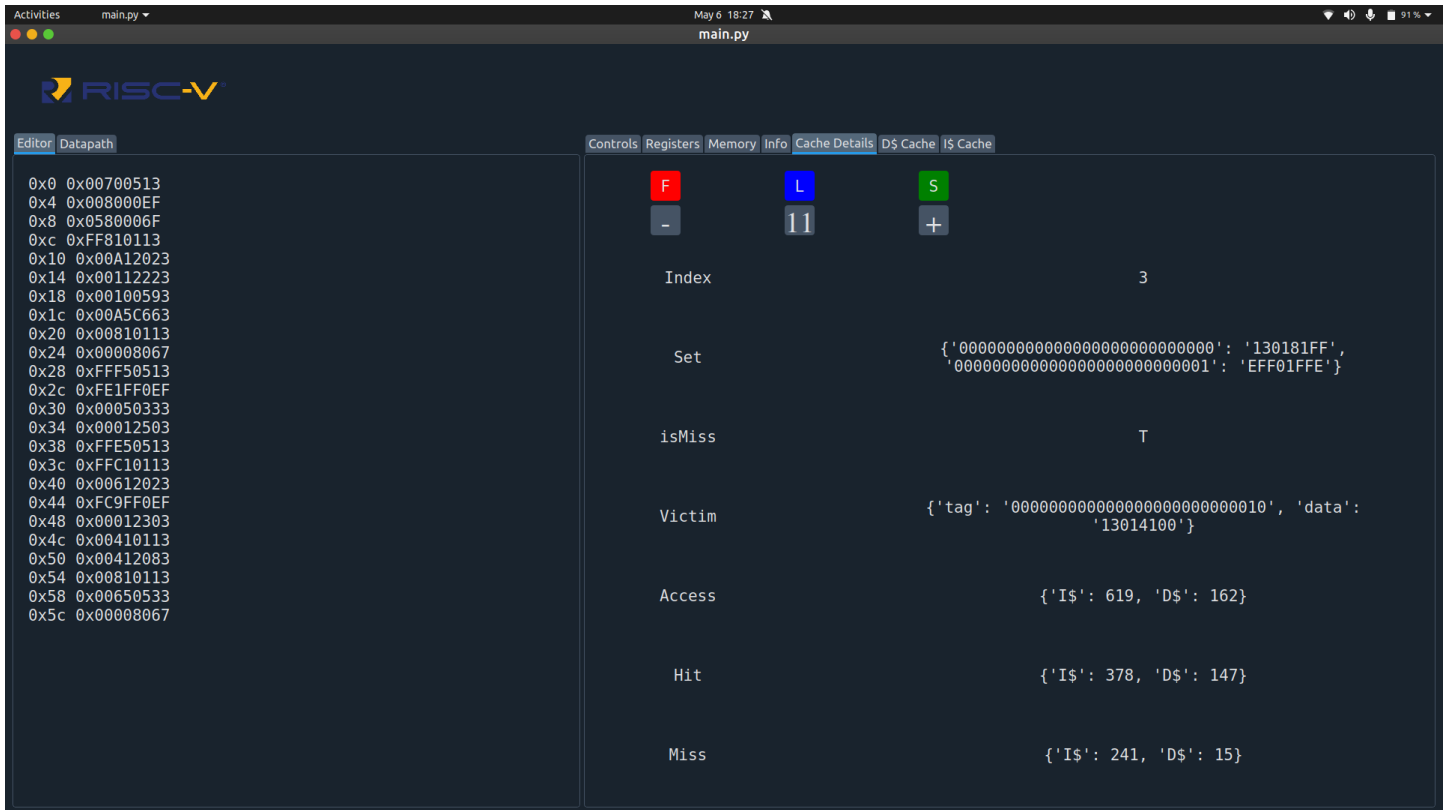
The 'info tab' contains all the necessary information about the pipelined execution of each stage. It should only be used when the code is run in pipelined mode.

We can see the info about each cycle and each stage. The cycle for which info is seen is seen in the box. In the image, we are checking details for the 5th cycle.

One can increase or decrease the total cycles by clicking '+' or '-' respectively. One can't change cycles beyond the total number of cycles of the program or less than 0 either.

We can see the info about the stage by clicking the 'F' 'D' 'E' 'MA' or 'WB' boxes accordingly.

EE rs1 'T' implies that there is Execute to Execute forwarding going on because of a data hazard in rs1.



We can see the Cache Details corresponding to the selected cycle.

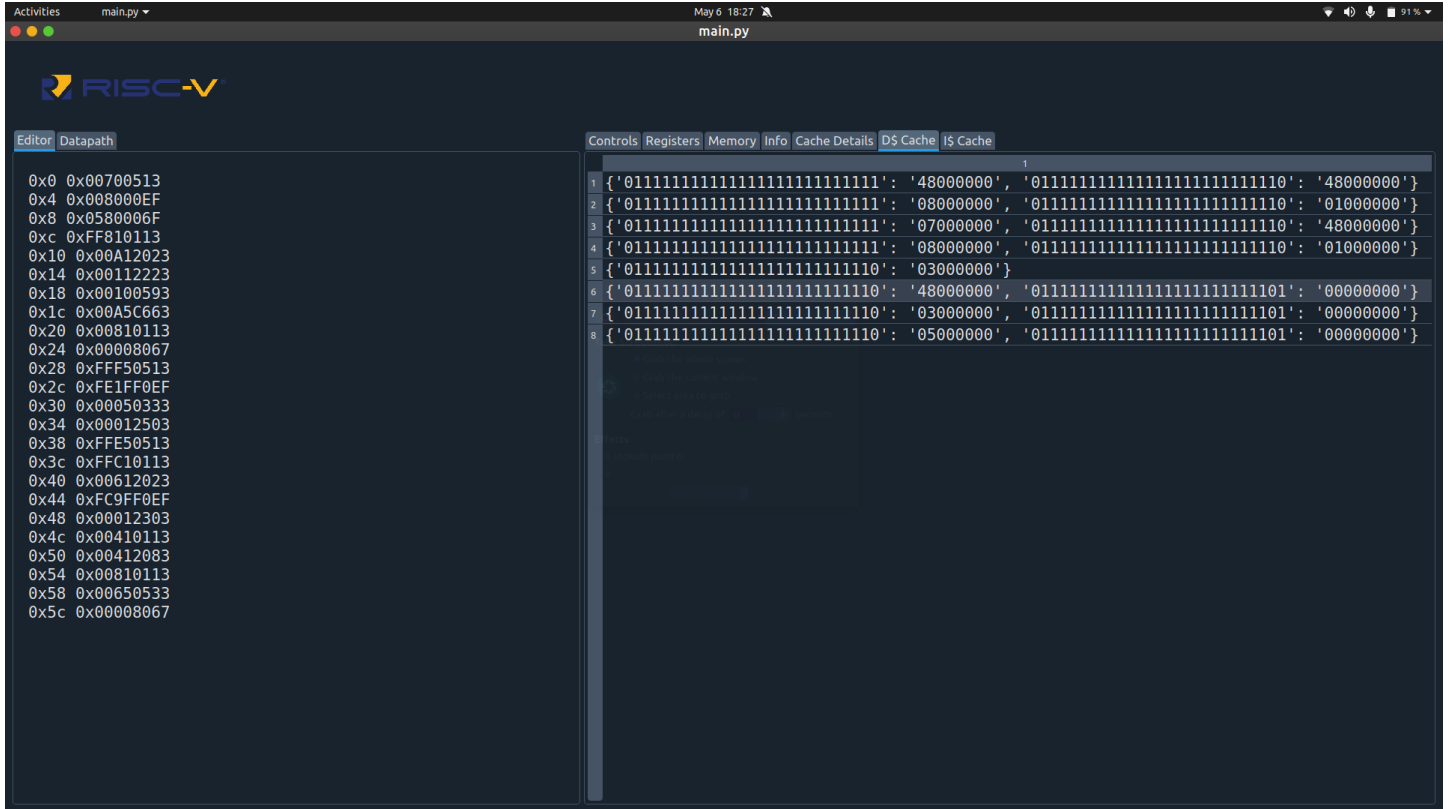
One can increase or decrease the total cycles by clicking '+' or '-' respectively. One can't change cycles beyond the total number of cycles of the program or less than 0 either.

We can see the details about Fetch, Load, and Store accordingly by clicking on that block.

'isMiss' is T(meaning True) when Miss occurs else it is F(meaning False). Here we can see the Total number of Accesses, Hits, Misses for Instruction and Data respectively, they are independent of the cycle number and are the net stats after the execution of the whole program.

The tag and data of the Victim is shown in the Victim block. If no data is present then it will be -1.

For example if you want to see the Load Cache Details for the 11th cycle first go to the 11th cycle and then click on the Load button.

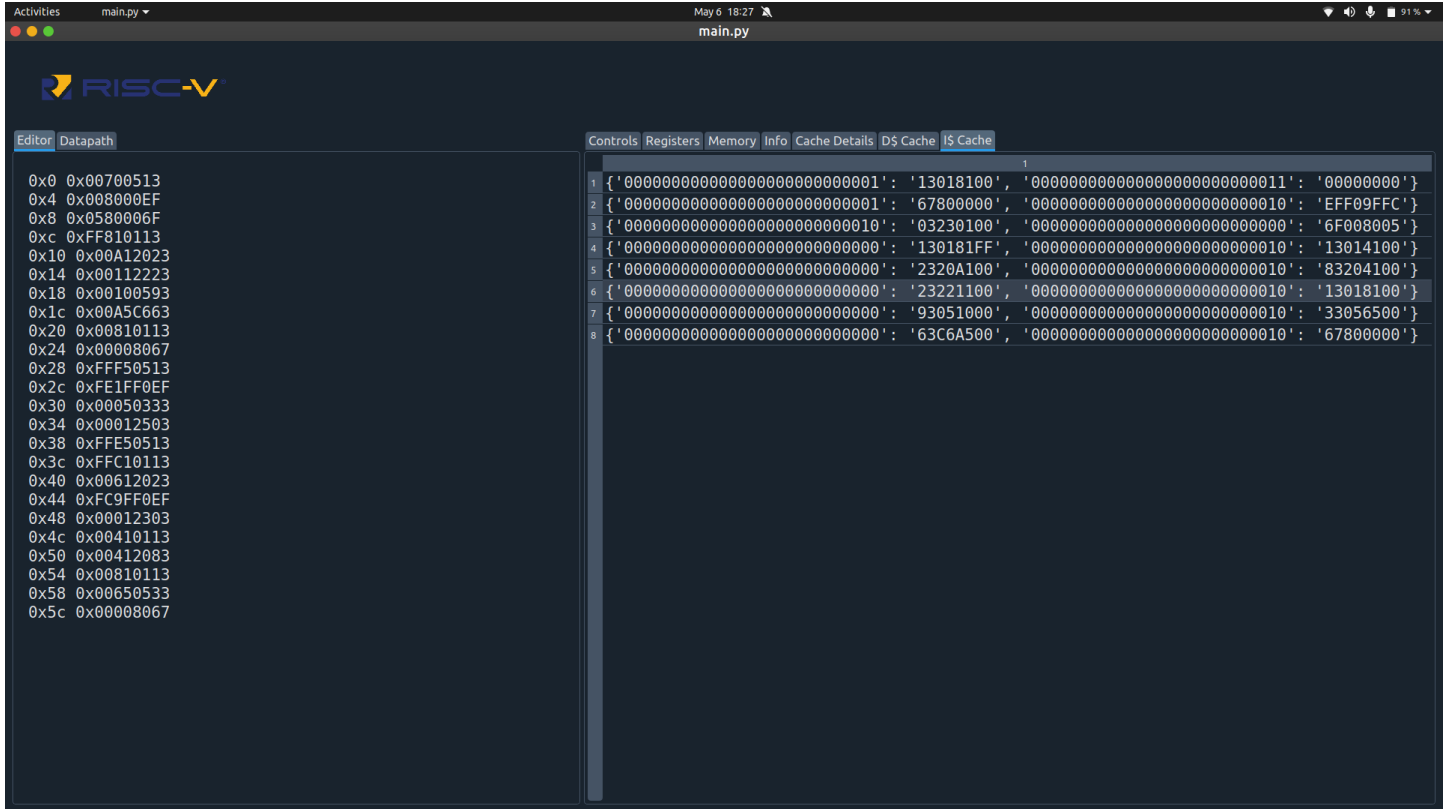


The information about the Data Cache is seen here

We can see the whole info by double clicking on the corresponding box. We can then scroll thorough the info using the right or left arrow key.

One set is represented by the line number and the total blocks in it are given in a line.

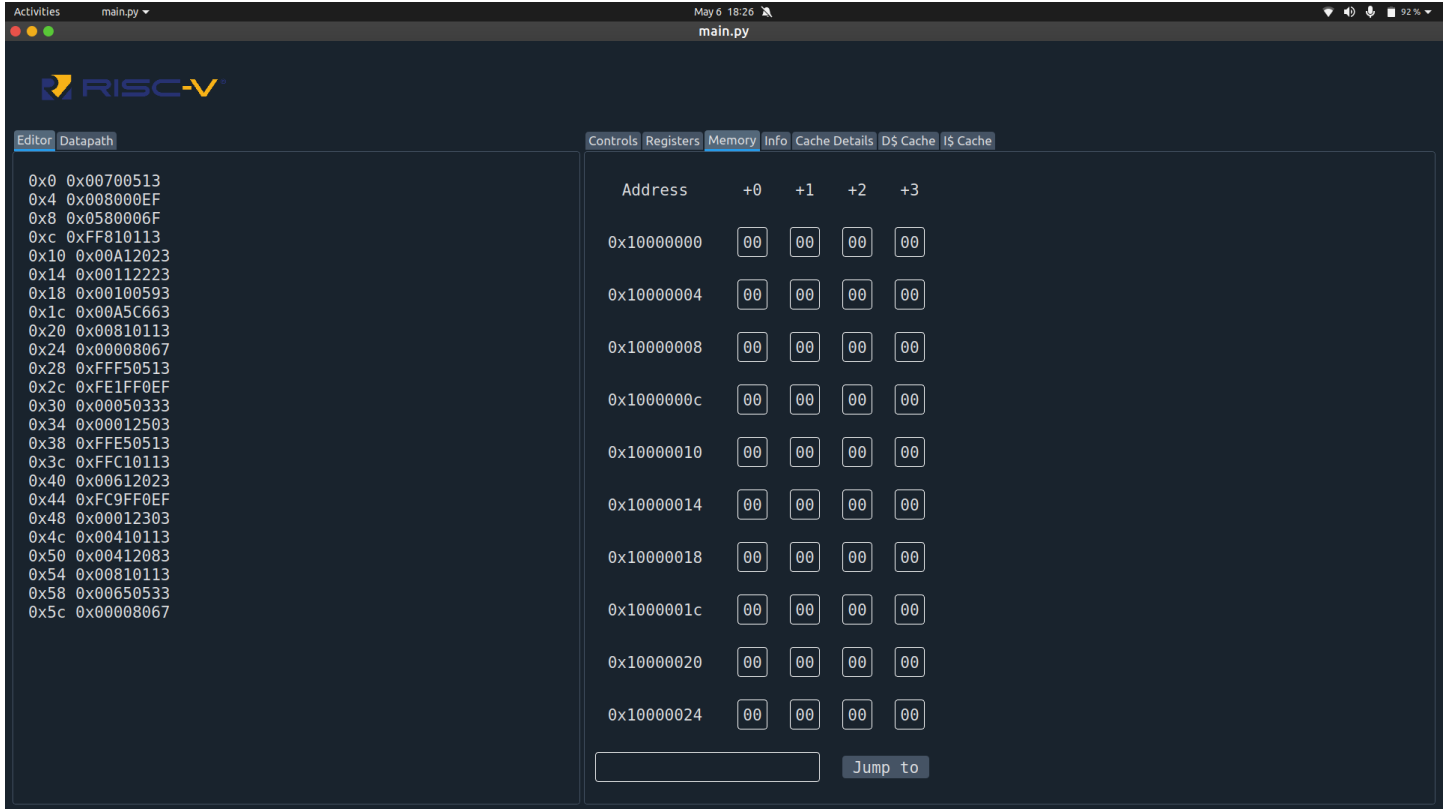
One block is of the format {tag:“data”}.



The information about the Data Cache is seen here

We can see the whole info by double clicking on the corresponding box. We can then scroll thorough the info using the right or left arrow key.

One set is represented by the line number and the total blocks in it are given in a line. One block is of the format `{tag:“data”}`.



In the memory we can see the address and the value at each address.

For example if the address is 0x1000000C then the address of each of the right box are 0x1000000C , 0x1000000D, 0x1000000E and 0x1000000F.

Here we can see the jump to register functionality, where we have stored a value in the register 0x10000500, and are jumping to it and seeing the value.

For example to jump on the location 0x10000500 write '10000500' in the Jump to box which is in the bottom right in the memory tab.

We have shown memory, registers and info on the right section of the GUI while on the left part there is Code Editor.

In the Code Editor you can edit your code or copy paste your code and also save it.

The info contains the details of the instruction being executed in each stage during a cycle.

First of all we will paste the code in the Editor on the left side and then save and compile the code by clicking the respective 'save' button which is the first button on the top right and 'compile' button. The code will get updated in **main.mc** file and then the code will be executed and a ✓ will appear. The memory and registers will be updated simultaneously.

Other details are present in their respective tabs as described earlier.

Refer to the next page for a more detailed explanation of each tab and its purpose.

Summary of the details of the layout of the GUI

- We can jump to the memory using the 'Jump to' box on the bottom right corner in the memory section.
- All the registers can be seen in the register file by scrolling in the register file section.
- Executing **main.py** always generates a GUI window where you can code in the Editor.
- We can also execute non GUI code by writing `python main.py`
- The DataPath explains how the data is flowing through each stage.
- Here the 5 stages can be seen in the DataPath tab.
- The number below it is the cycle number for which the DataPath is shown.
- You can see the DataPath in the respective cycle by going to that cycle.
- Click on + to go to the next cycle and on - to go to the previous cycle.
- We can observe all the required data in the cache by clicking on the required tabs
- One tab is meant to observe the Victim block in each cycle, where you can adjust the cycle number similar to the 'datapath tab' along with the details of individual sets during that cycle.
- Two more tabs are there for observing the final state of the instruction cache (I\$) or data cache (D\$) respectively after the program has run.