EE2003 Assignment 4

Students are required to write a single report with PART A report and PART B report

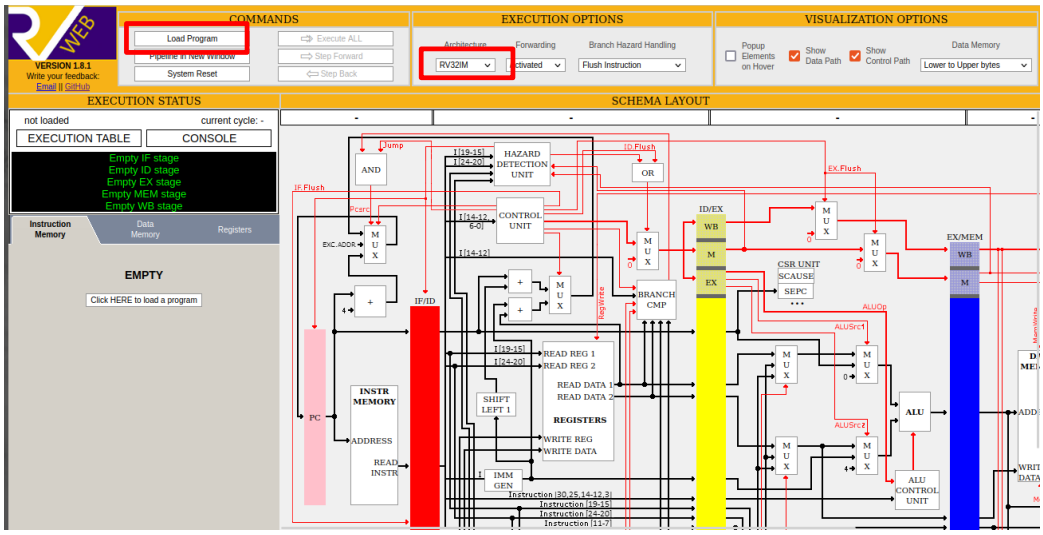
PART A : HAZARDS

In this part of assignment students are required to implement the given program using the base RISCV 32I instructions and load it into the Web RISCV simulator, set up the different configurations and interpret the results. More instructions about writing the code using base instructions and writing the report on the interpretation the results will be provided in this document

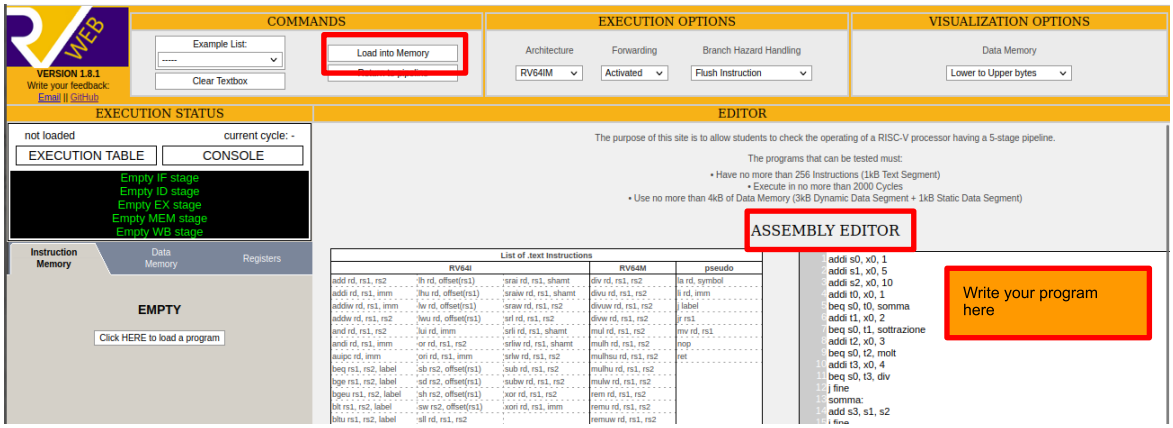
Simulator: https://webriscv.dii.unisi.it
RISCV 32I base instructions : Screenshot from 2022-10-15 23-37-01.png
RISCV ISA Manual : RISCV ISA specification
Please read the ISA Manual to understand the function of each base instruction

Using Web RISCV simulator : (follow the highlighted red boxes)

1. Set the architecture and Load the program:



2. Write your assembly code in the Assembly editor and Load it into the memory

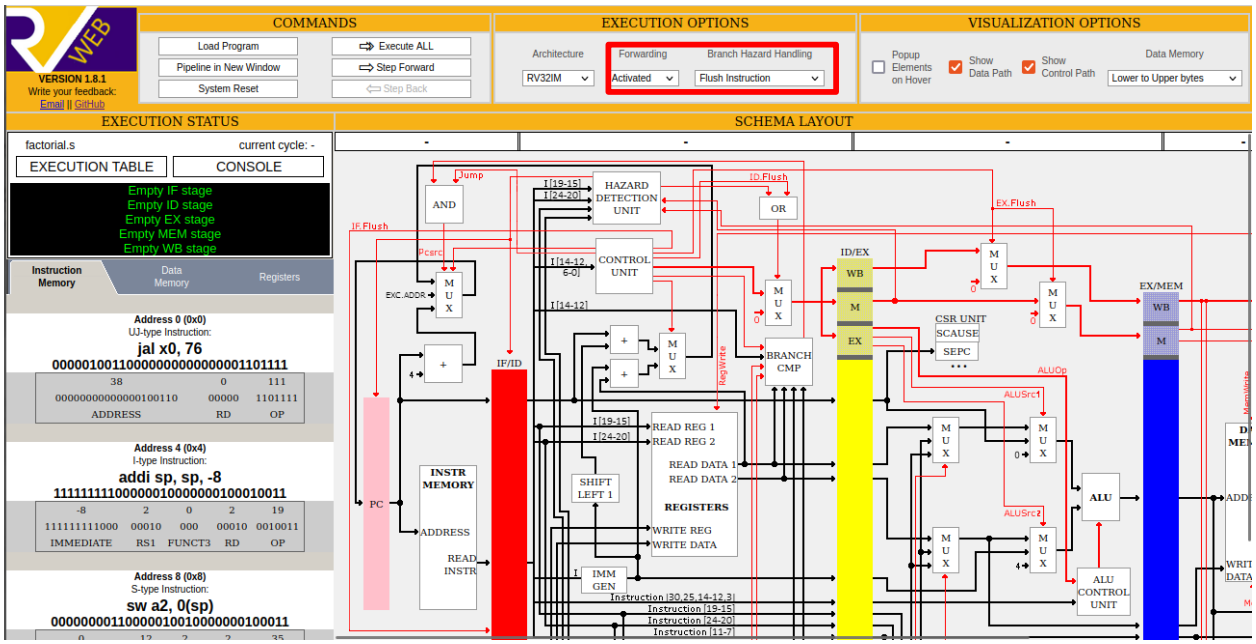


3. Set the Configurations :

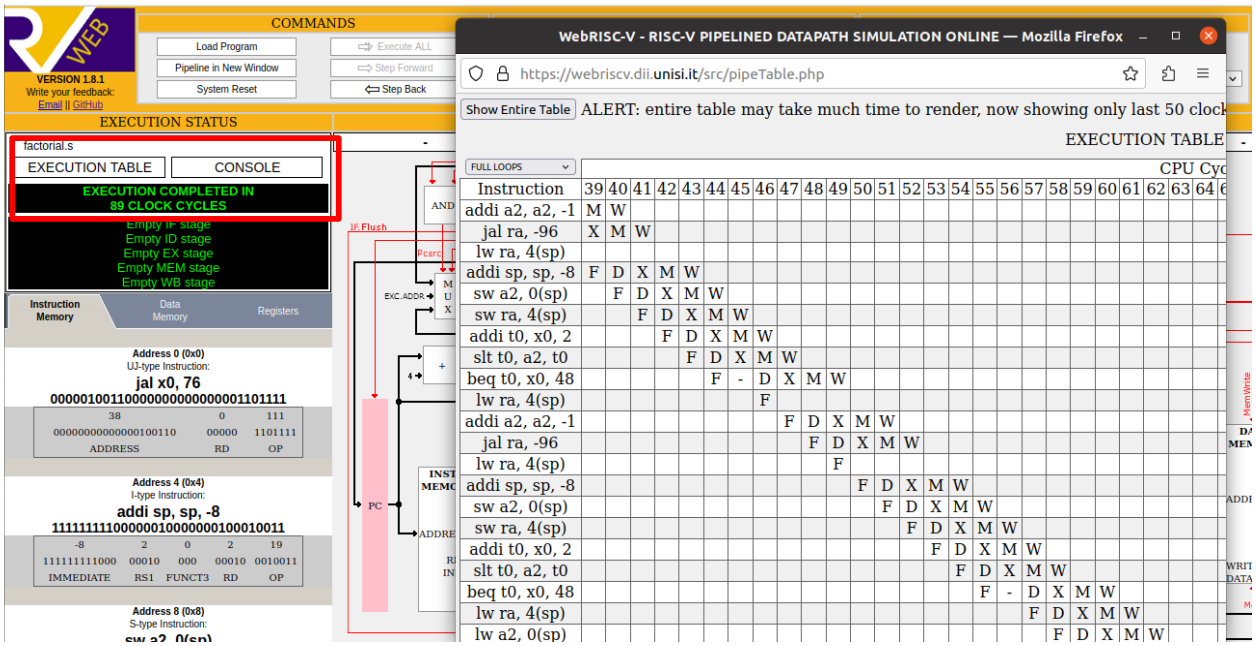
Set the configurations before executing, to see the simulation results

Forwarding : Activated or Deactivated

Branch Hazard Handling



4. Execute the program and observe the number of clock cycles in Execution table



Problems:

Write RISC-V assembly code using the base instructions set for the following:

1. Factorial: Find the factorial of a number greater than 5 using iteration method
2. Fibonacci Series: Find the n^{th} (n > 10) element of the fibonacci series.
3. GCD: Greatest common factor of two numbers of your choice
4. Dot product: Load two vectors of size 10 from the data memory and find the dot product.
5. Bubble sort: Sort a vector of 10 elements and store it into the data memory
6. Counter: Create a counter which counts till n (n > 20).
7. Complex Multiplier: Take 2 complex numbers as input and store the resultant product into data memory
8. Binary Search: Search an element n (input) from a vector of size 10 in the memory
9. Fixed point addition: Add two floating fixed-point numbers of 32 bit each (16+16). You may store integer part and decimal part in separate or same register
10. Modulo Operator: Find the modulus using repeated subtraction method
11. Palindrome Checker: Check whether the binary representation of the integer stored in the input register is a palindrome or not. You can output 1 in a separate register if it is palindrome.
12. Sum of integer array: Find the sum of the integer array of length 10. The integers in the array can range from -8 to 7. Store them wisely using the least number of registers or memory locations possible and find their sum.

Instructions about input and output presentation:

1. The input to the program can be manually added to a register or to the data memory at the beginning of the program.
   For example:

Storing in registers : addi x1, x0, 5 will set x1=5
Storing in memory : sw x1, 1024(x0)
The offset must be greater than 1024!

2. When storing vectors into data memory, store them in consecutive addresses so that you can load them back using a loop.
3. While writing the program, make sure that you create data hazards explicitly if not present already, so that you can see the difference when using the different configurations in the simulation results.

Information about configurations in the simulator to code work correctly:

1.When forwarding is disabled , when there is a dependency between 2 consecutive instructions(RAW hazard) , the simulator stalls for 2 clock cycles and then lets the 2nd instruction go to the decode stage. So either you can leave the code as it is , or you can insert 2 'nop' instructions between the 2 instructions, so the total number of clock cycles to finish execution remains same.

Instruction	1	2	3	4	5	6
addi ra, x0, 0	F	D	X	M	W	
addi sp, x0, 1		F	-	-	D	X
addi gp, x0, 9					F	D
addi tp, x0, 1						F

Bonus points : Extra points will be given to those students who will reorder the instructions so that actual valid instructions in the program(which have no dependencies from the first instruction) get executed in the pipeline instead of 'stalls' or 'nop' instructions, thus reducing the number of clock cycles required to finish the execution

2.When forwarding is enabled:

In this configuration , when there is a dependency between 2 consecutive instructions the pipeline is not stalled as the values get forwarded to the execute stage of the second instruction from the memory stage of the first instruction.

Important: The above scenario of forwarding is only possible when the first instruction is an ALU instruction, You will observe that when the first instruction(in RAW hazard) is a LOAD instruction, you will still observe a stall even when the forwarding configuration is enabled, you must explain the reason for such stall in your report.

3. When flushing is enabled :

In this configuration , it is predicted that the branch is ALWAYS NOT TAKEN and when the branch is taken the instruction after the branch instruction in the pipeline is flushed thus wasting the resources of the pipeline for one clock cycle

4. When Executing branch delay slot is enabled :

In this configuration the instruction after the branch instruction is executed but NOT FLUSHED, so students must take care in inserting the right instruction after the branch instruction so that the program correctness is preserved

Expected Results:

Submit a report containing the following:

1. RISC-V Assembly code
2. Identify all the data and control hazards. Mention what kind of dependency in the case of data hazards
3. Simulate & calculate the throughput w & w/o forwarding
4. Simulate & calculate the throughput w & w/o flushing
5. Compare all the results

You should be able to explain the results of all the 4 possible configurations during the evaluation (i.e. reasons for the stalls ,reason behind reduced number of clock cycles when forwarding is enabled and when branch delay slot is executed etc)

PART B : BRANCH PREDICTION

In this part of assignment students should interpret the results of the execution of a program when run on the RISCV processor simulator (will be given below) using different branch prediction strategies

Simulator : https://github.com/hehao98/RISCV-Simulator

Instructions:

1.Please follow the instructions provided in the git repo to build the simulator.

2.Run the elf file of the program you are assigned with this command with different branch prediction strategies :

```
./Simulator riscv-elf-file-name [-v] [-s] [-d] [-b strategy]
```

The students can find the program's elf file and disassembled file in the riscv-elf folder.

Branch prediction strategies:

- AT: Always Taken
- NT: Always Not Taken
- BTFNT: Back Taken Forward Not Taken
- BPB: Branch Prediction Buffer (2 bit history information)

Problems:

Each student will be assigned a program from the following list:

1. ackermann.riscv
2. matrixmulti.riscv
3. test_arithmetic.riscv
4. helloworld.riscv
5. quicksort.riscv
6. test_branch.riscv

You can find these files in the riscv-elf/ directory of the repository.

Report making:

Students must run the program on the simulator with the 4 branch prediction strategies and make a table showing the results of overall performance of the program when run with different strategies.

Students must be able to explain the 4 branch strategies used and how they affected the overall performance of the execution of the program during the evaluation.