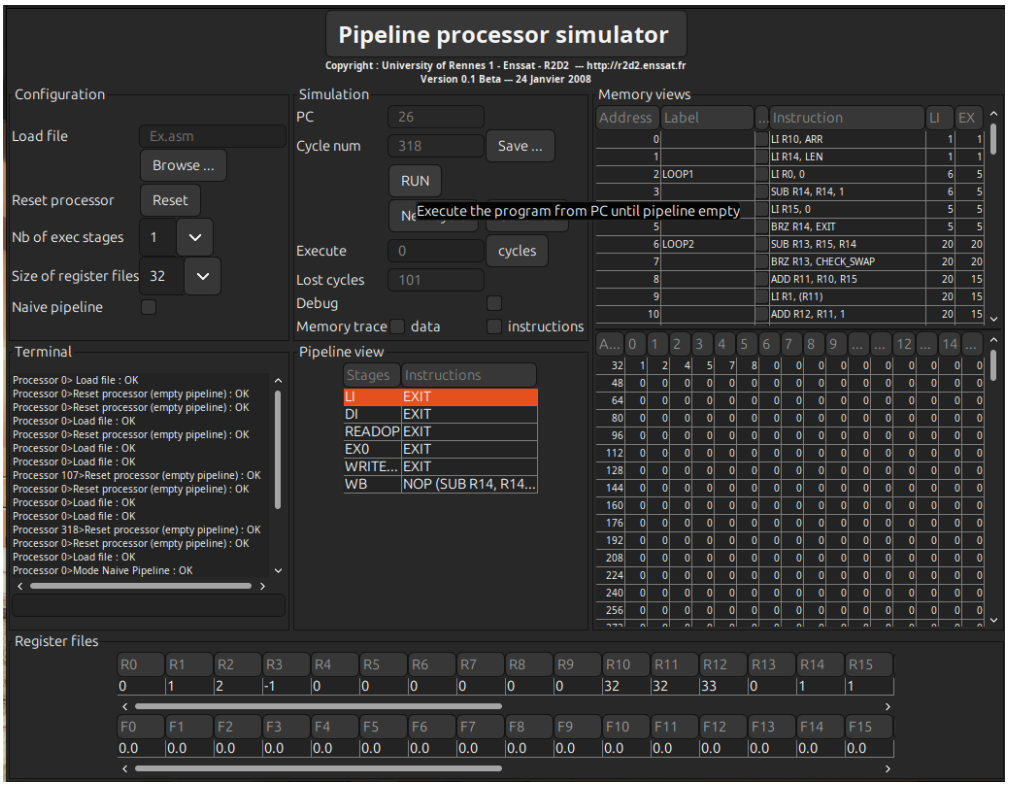
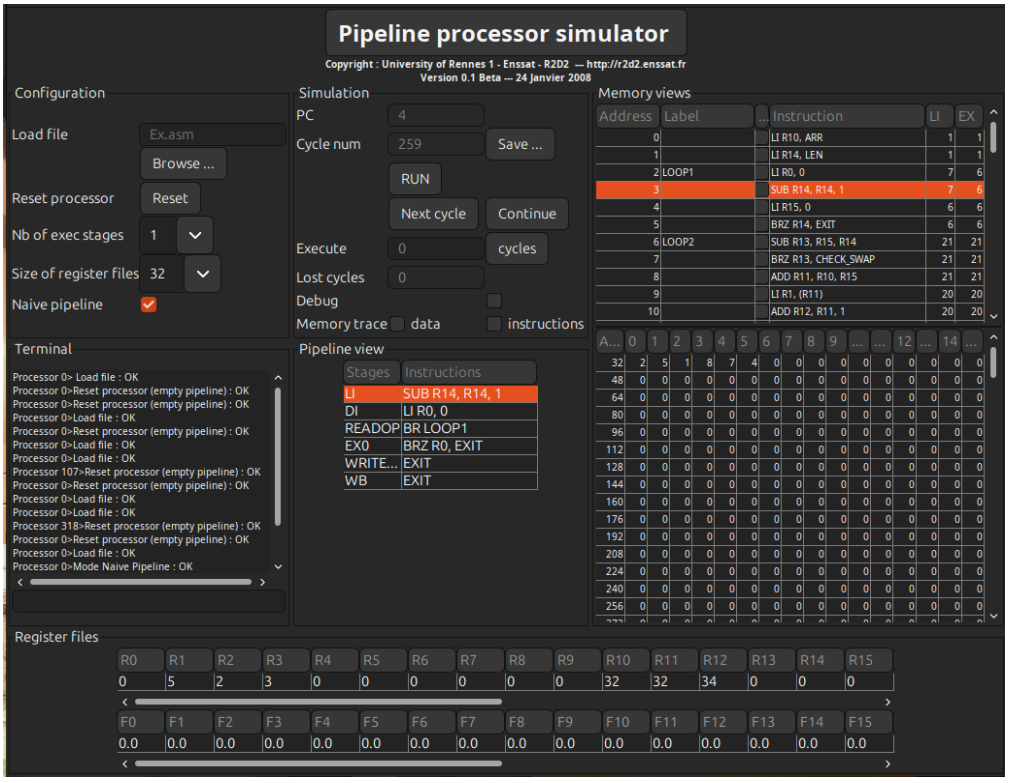
# Nguyen Tai Anh- 22BI13028- Practical 2- Advanced CA and x86 ISA

## I. Validate the correctness of program with and without naive mode

1. **Without** naive pipeline mode, the program runs normally as in Labwork 1:




2. **With** naive pipeline mode, the program is not running correctly:

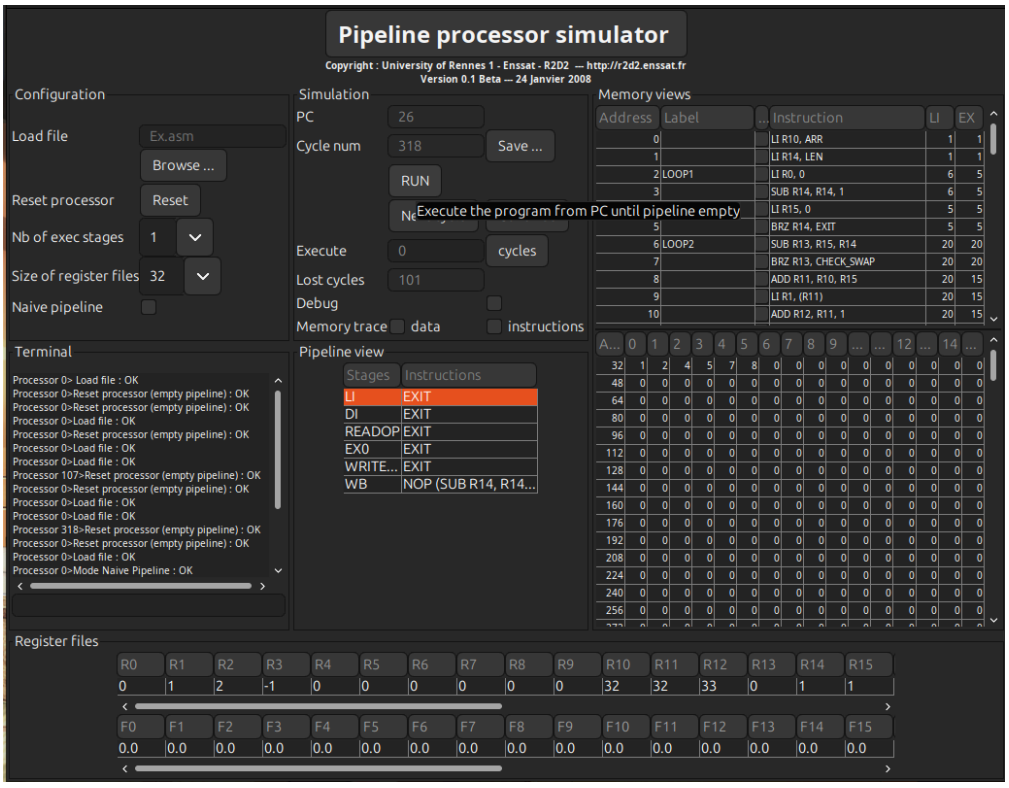


3. **Explanation:**

- When executing in normal pipeline mode, results become more accurate because each instruction is executed in distinct stages of the pipeline. This ensures that the outcome of each instruction is correctly processed before continuing to the next stage. The value from the previous instruction must reach the Write Back (WB) stage and be updated before the next dependent instruction is executed. For example, “add R13, R14, R15”, the result of R12 = R14 + R15 will be saved first and then move to the next instruction such as li R0, R13.
- The reason why naive pipeline mode will run incorrectly is because in naive pipeline mode, instructions dependent on the results of previous instructions may not wait for those results. For example, if R14 and R15 are loaded from memory and immediately compared or swapped/add/subtract, the compare or swap might execute before the load completes or before moving to the next instruction related to these parameters, leading to the fact that the result will be failed and making the execution incorrect.

## II. Analyze the execution of bubble algorithm in pipeline:

1. **Note if the execution is correct or not**: The execution without naive pipeline mode is correct.



2. **Capture case when two instructions are dependents and understand what happens**

```
sub R13, R15, R14
brz R13, .CHECK
```

R13 = R15 - R14, and loop until R13 = 0, continue at CHECK Instruction.

```
add R11, R10, R15
li R1, (R11)
```

R11 = R10 - R15, after that the value of R11 will be load to R1

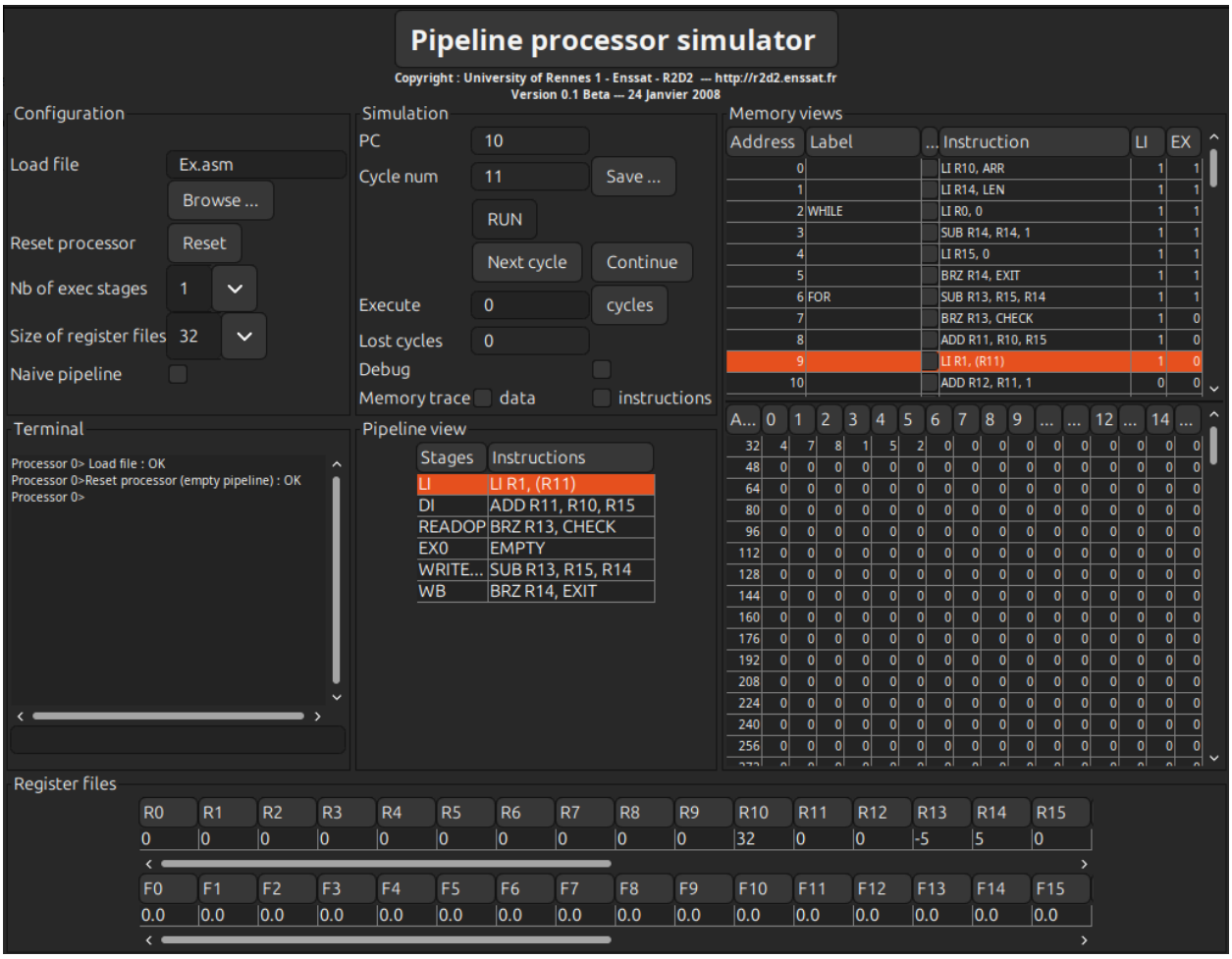
```
add R12, R11, 1
li R2, (R12)
```

R12 = R11 - R1, after that the value of R12 will be load to R2

3. **Capture case when a branch is executed and understand what happens**

When a branch is executed, the instructions inside the branch will be executed in the new flow, and the instructions after the branch will not be executed until the branch has been resolved.

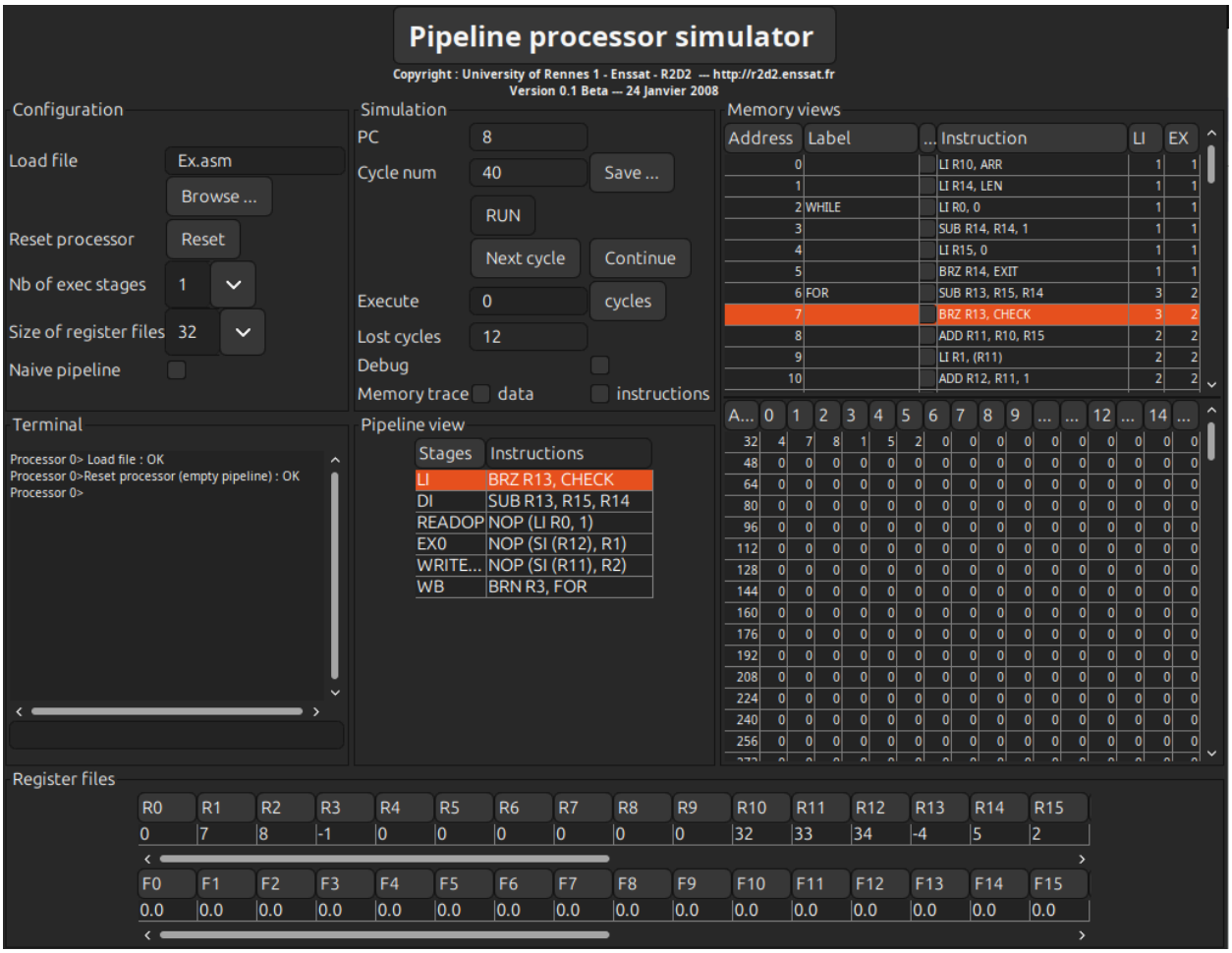
```
brz R14, EXIT       -- if length is zero, exit
```



In this case, firstly, I already have instructions to calculate the length of the initialize array to implement the loop, then, I set this value to R14 and decrease by 1 when one loop is finished. After this value of R14 become 0, the loop will end and exit the program to avoid the infinite loop

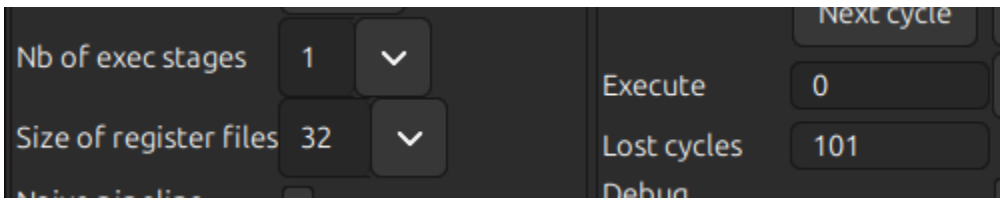
Another example is:

```
brn R3, FOR          -- if current element <= next element, continue loop
```

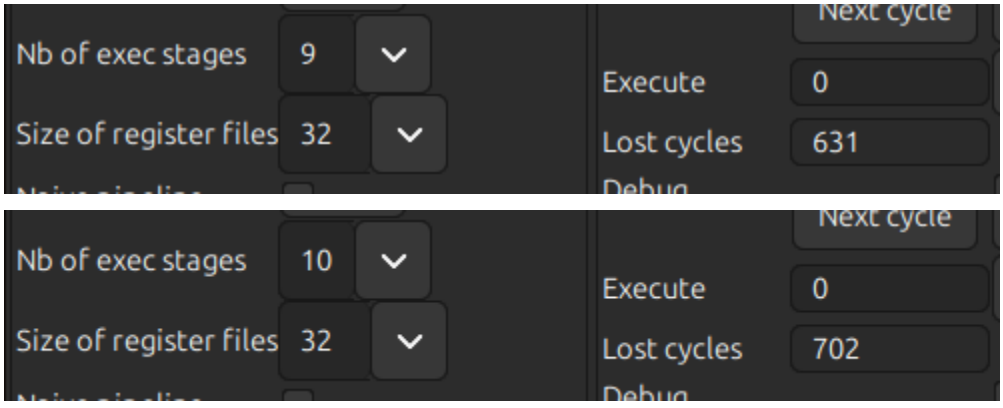



In this case, the value of R3 is R1 - R2 which R1 is the value of one element in the array and R2 is the next element of R1 in the array. If R3 is negative (brn R3), that means that R1 < R2 → R1 and R2 will not swap due to the bubble sort algorithm.

4. **Note the number of cycles which are lost during the execution with different number of pipeline stages**




Nb of exec stages 2
Size of register files 32
Next cycle
Execute 0
Lost cycles 204
Debug
Nb of exec stages 3
Size of register files 32
Next cycle
Execute 0
Lost cycles 296
Debug
Nb of exec stages 4
Size of register files 32
Next cycle
Execute 0
Lost cycles 388
Debug
Nb of exec stages 5
Size of register files 32
Next cycle
Execute 0
Lost cycles 347
Debug
Nb of exec stages 6
Size of register files 32
Next cycle
Execute 0
Execute on
Lost cycles 418
Debug
Nb of exec stages 7
Size of register files 32
Next cycle
Execute 0
Lost cycles 489
Debug
Nb of exec stages 8
Size of register files 32
Next cycle
Execute 0
Lost cycles 560
Debug




→ **Explanation:** When the instruction is executed in different stages, there are different numbers of instructions executed at the same time, so that there will be different amounts of cycles lost during the execution normally, the bigger the number of stages, the bigger the number of lost cycles. When the number of execution stages is increased, each instruction must pass through more stages before it is completed, which might lead to a higher number of cycles in the pipeline (in some cases, this number of cycles might reduce rather than increase). Even though the total number of cycles and potential lost cycles may rise, the crucial point is that the final output remains correct. Adding more execution stages boosts pipeline performance by enabling concurrent processing of multiple instructions and increasing parallelism.