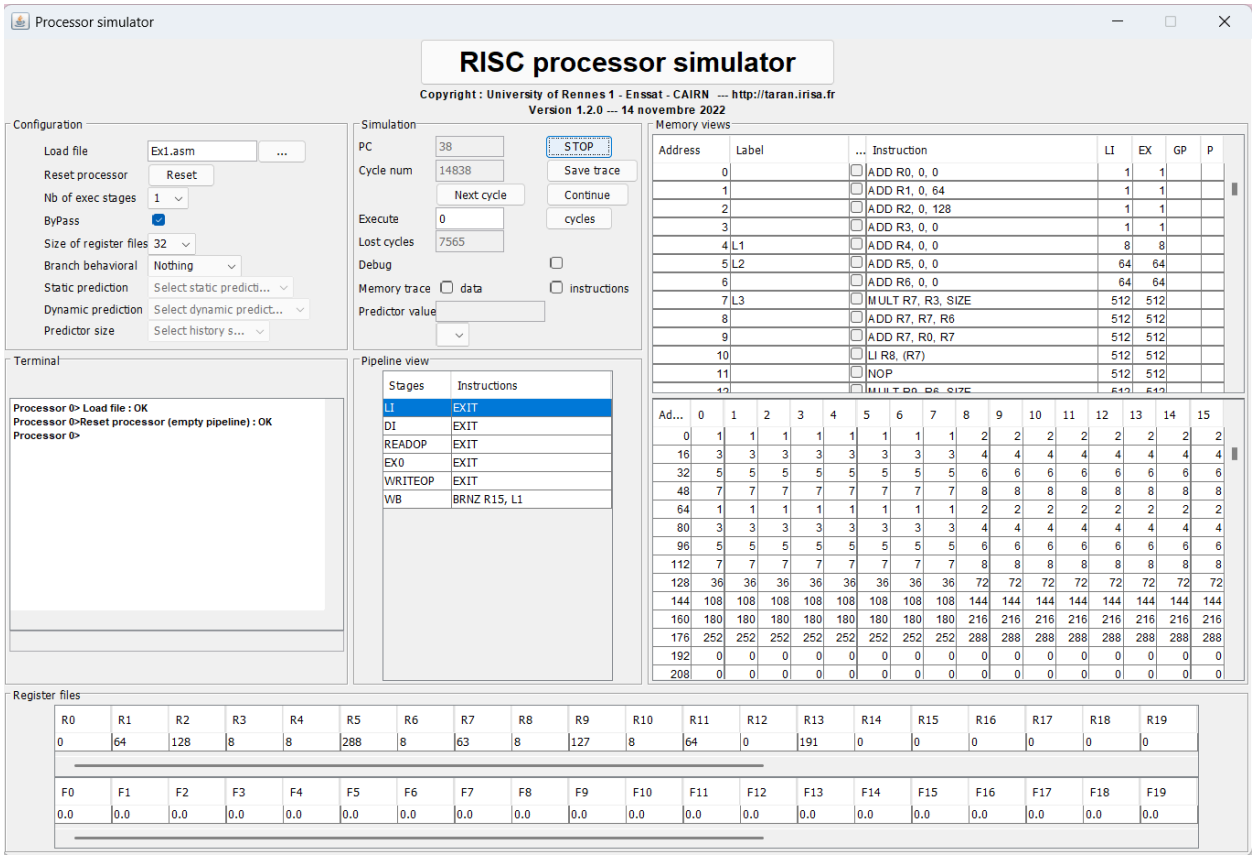
Practical 3: Pipeline hazard

1. Analyze Bypass technique:

1.1 With Bypass technique:

a) 1 execution stage:



- Execution Overview:
  - + Program Counter (PC): 38
  - + Cycle number: 14,838
  - + Lost cycles: 7,565 → nearly 50.97% of cycles are lost (likely due to stalls or pipeline hazards).
  - + Register Values:
    - ○ Registers like R0 to R8 hold initialized values that suggest matrix operations (likely a matrix multiplication task).
    - ○ R7 = 127 suggests result accumulation or indexing.
    - ○ Memory values indicate results written into the data section at regular intervals, consistent with matrix storage.

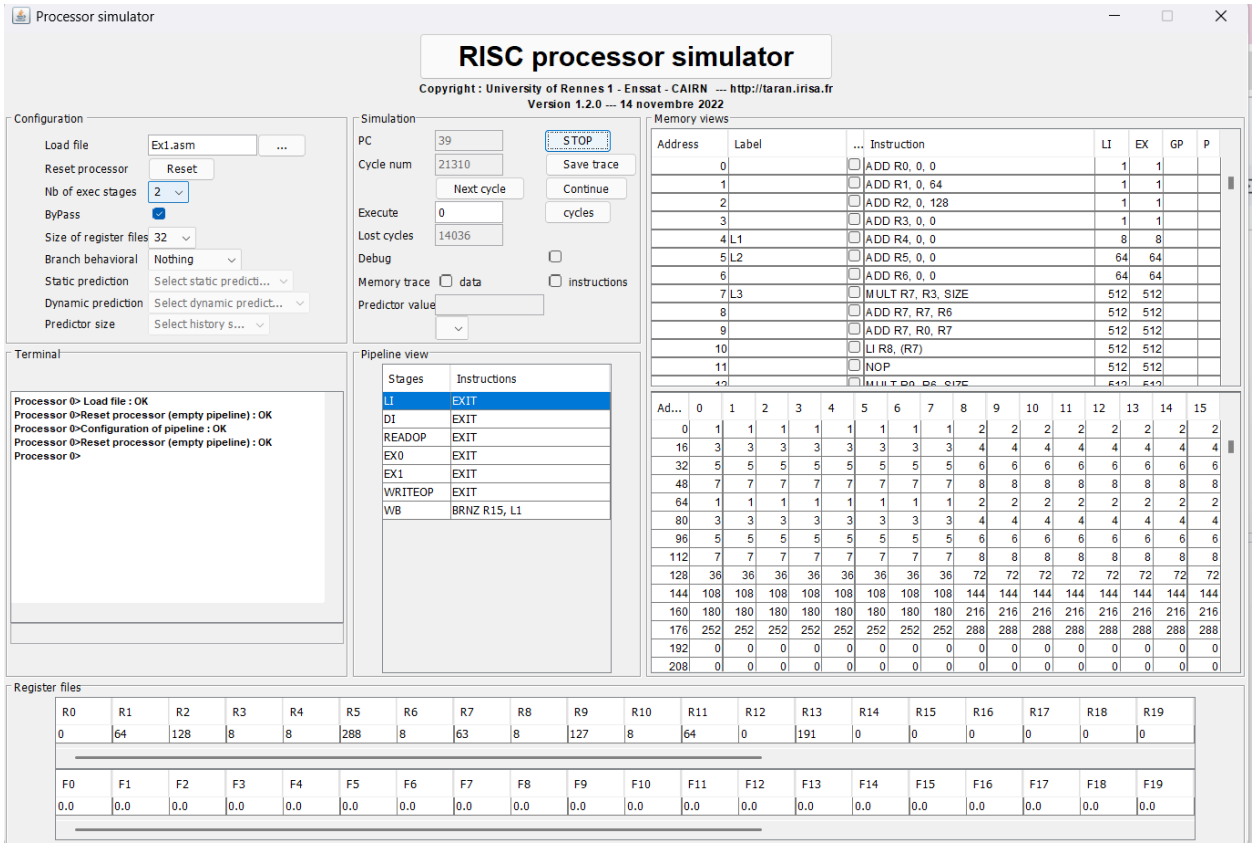
➔ The simulation completed successfully, and memory contents confirm that the expected results of the matrix multiplication were correctly stored. The register

and memory states are consistent with an 8×8 matrix result, which validates the functional correctness of the program.

- Using a single execution stage simplifies the pipeline structure but introduces significant performance penalties. The most notable consequence is the **substantial number of lost cycles** (7,565 out of 14,839), accounting for over half of the processor's operational time. This inefficiency is primarily caused by:
    + **Pipeline stalls** due to unresolved data hazards.
    + **Lack of parallelism**, as only one instruction can be processed through the execution unit at a time.
    + **Increased serialization** of operations, especially during compute-intensive tasks like matrix multiplication.

➔ Although the bypass mechanism is enabled, its impact is limited in this configuration due to the constrained execution bandwidth.

b) 2 execution stages:



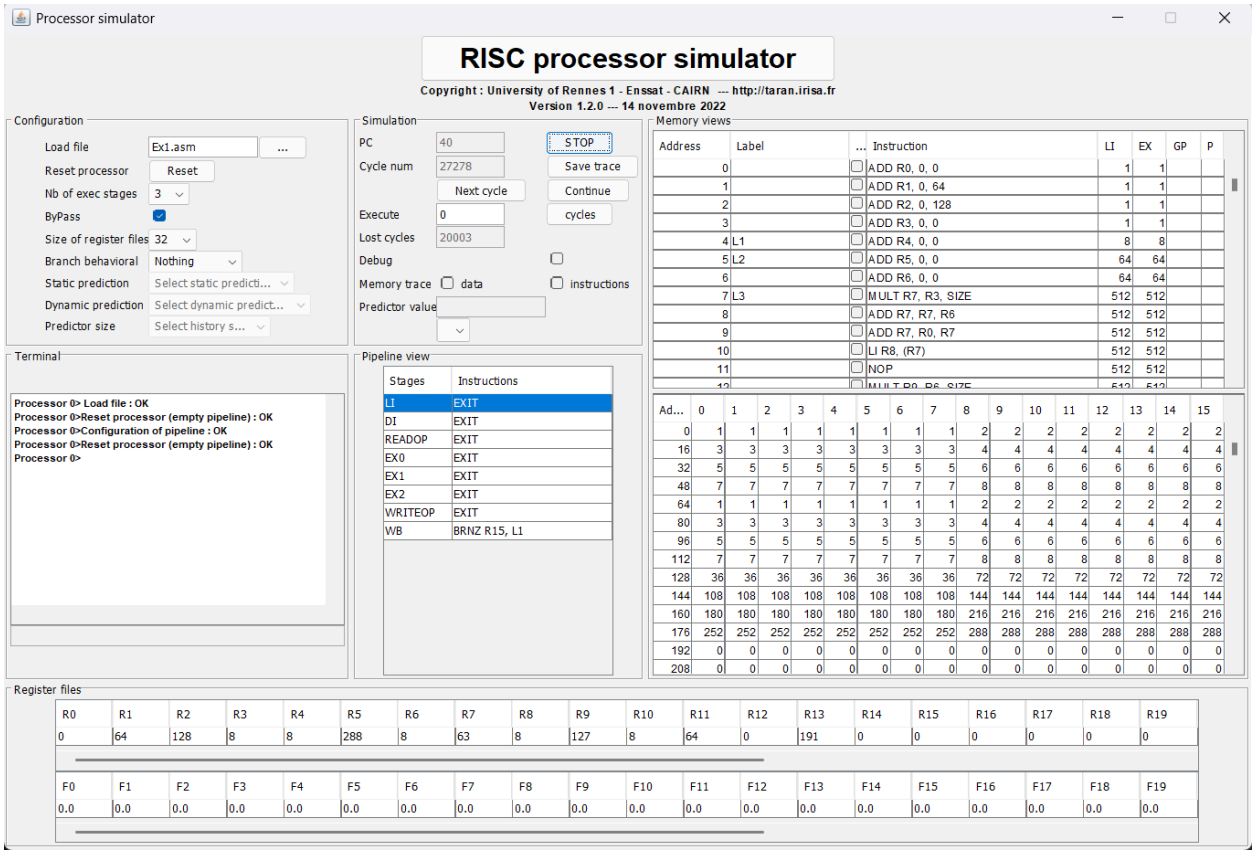
- Execution overview for 2 execution stages:

    + Program Counter (PC): 39
    + Total Number of Cycles: 21,310
    + Lost Cycles: 14,036
    + Percentage of Lost Cycles: Approximately 65.9%
    + Bypass Mechanism: Enabled
- The register and memory contents confirm that the matrix multiplication computation completed successfully, indicating that program correctness was not compromised by the increased pipeline depth.
- Contrary to the expectation that additional execution stages would reduce total execution time through increased instruction-level parallelism, the results showed a deterioration in performance compared to the one-stage configuration:
- The total cycle count increased by 6,471 cycles (from 14,839 to 21,310).
- The number of lost cycles rose dramatically from 7,565 to 14,036.
- Despite additional pipeline stages, the processor exhibited lower throughput and greater idle time.

This decline in performance is primarily attributed to:

- Pipeline hazards, such as data and control dependencies, which could not be resolved efficiently due to the absence of branch prediction and dynamic scheduling.
- Pipeline imbalances, where increased execution capacity was not matched by enhancements in instruction issue or hazard detection mechanisms.
- Increased pipeline depth, which introduced more inter-stage dependencies and stall conditions, reducing overall efficiency.

c) 3 execution stages:



- Execution overview for three-stage execution:
  + Program Counter (PC): 40
  + Total Number of Cycles: 27,278
  + Lost Cycles: 20,003
  + Percentage of Lost Cycles: Approximately 73.3%
  + Bypass Mechanism: Enabled
- The register file and memory content confirm that the matrix multiplication algorithm completed accurately, indicating functional correctness was maintained.
- The transition from two to three execution stages yielded further degradation in performance:
  + Total cycle count increased by 5,968 cycles (from 21,310 to 27,278).
  + Lost cycles increased by 5,967 (from 14,036 to 20,003).
  + The lost cycle ratio increased by approximately 7.4%, indicating a growing inefficiency in pipeline utilization.
- Several factors contribute to this regression:
  + Deeper execution pipelines introduce additional hazards, especially read-after-write (RAW) and write-after-read (WAR) dependencies, which the current processor configuration cannot effectively mitigate.

+ The absence of dynamic hazard detection and branch prediction increases the occurrence of pipeline stalls, particularly within loop structures.
+ The pipeline becomes increasingly underutilized as more stages are introduced without concurrent improvements in instruction dispatch or reordering mechanisms.

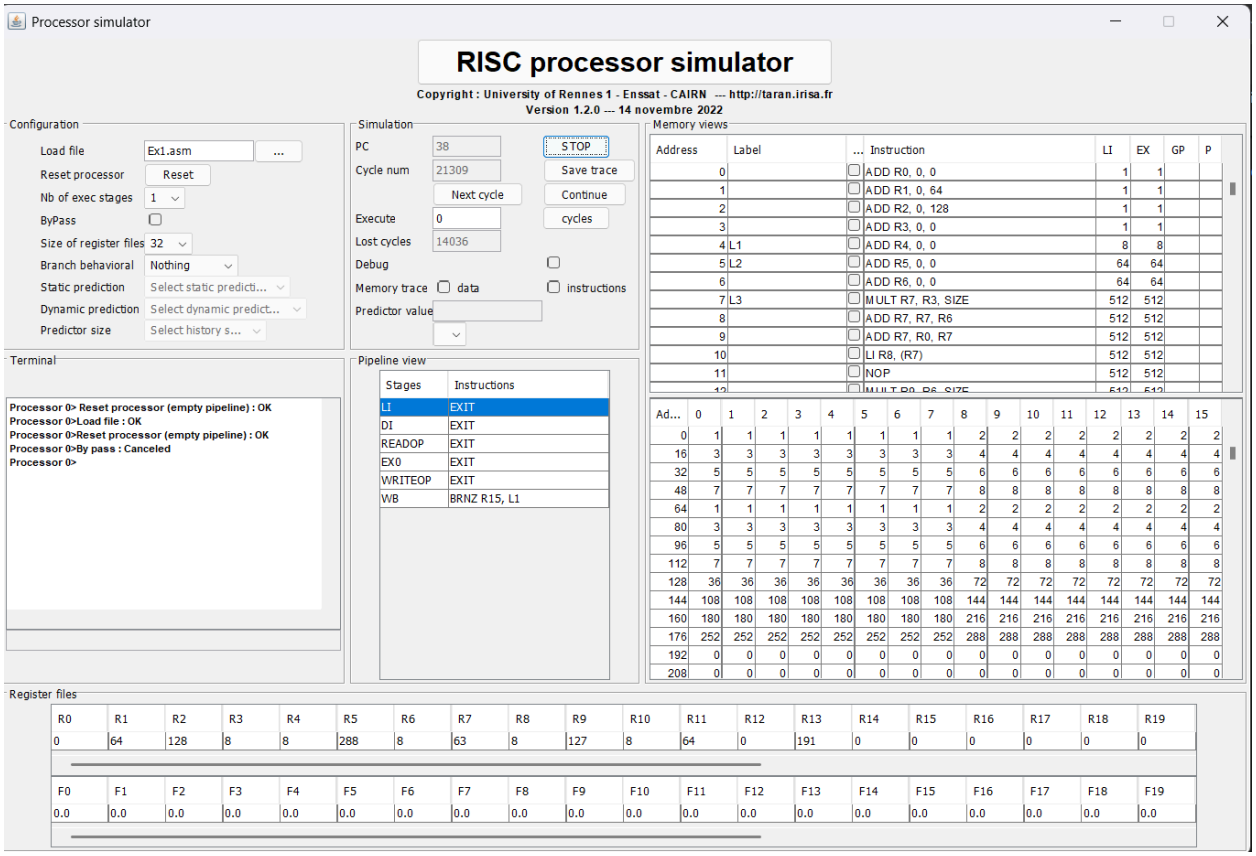
d) Predicted Effects of Adding More Stages (4, 5, 6,...):

If the number of execution pipeline stages continues to increase without architectural improvements (e.g., hazard prediction, reordering, speculation), we can predict the following:

- Pipeline stalls will increase exponentially, since deeper pipelines delay result propagation, extending the duration for which dependent instructions must wait.
- Lost cycles will dominate execution, potentially exceeding 80% or more, especially for loop-based computations like matrix multiplication.
- Instruction throughput will decrease, even though more instructions can be "in-flight", because they will frequently pause at execution stages waiting for dependencies to resolve.
- Overall processor efficiency will decline, as the ratio of useful work (effective cycles) to total cycles continues to shrink.

1.2 Without Bypass technique:

a) 1 execution stage:



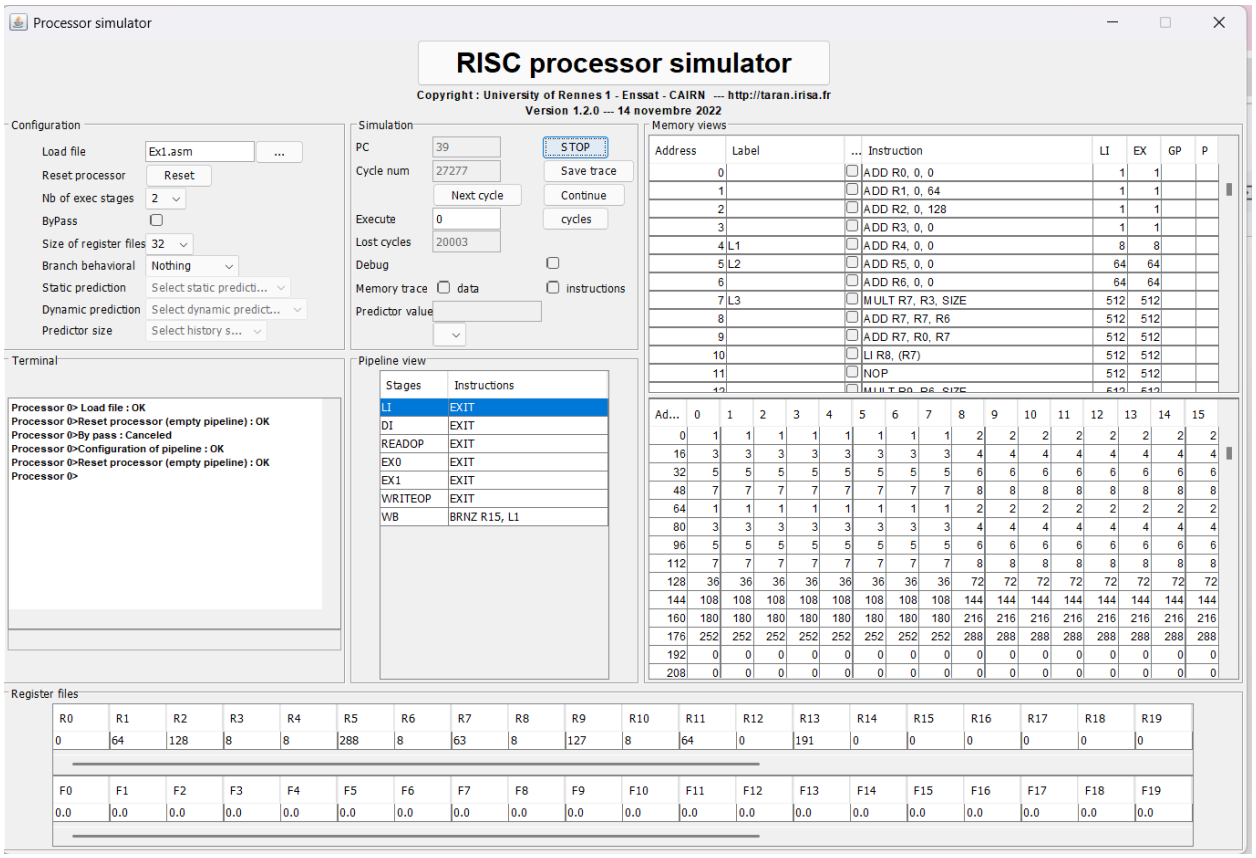
- Execution Overview:
    + Program Counter (PC): 38
    + Cycle number: 21309
    + Lost cycles: 14036 → approximately 65.88% of cycles are lost (likely due to stalls or pipeline hazards).

Using a single execution stage (Nb of exec stages = 1) simplifies the pipeline but leads to significant performance issues. The high number of lost cycles (14036 out of 21309) highlights inefficiencies caused by:

- Pipeline stalls due to data hazards, as seen in the pipeline view (e.g., repeated EXIT instructions indicating delays).
- Lack of parallelism, with only one instruction processed at a time in the execution stage.
- Serialization of operations, particularly in compute-heavy tasks like the MULT and ADD sequences observed.
    ➔ With bypass not enabled, data forwarding is unavailable, exacerbating stalls and further increasing the number of lost cycles. This lack of bypass

support, combined with the single execution stage, severely limits instruction throughput, resulting in nearly two-thirds of cycles being lost.

b) 2 execution stages:



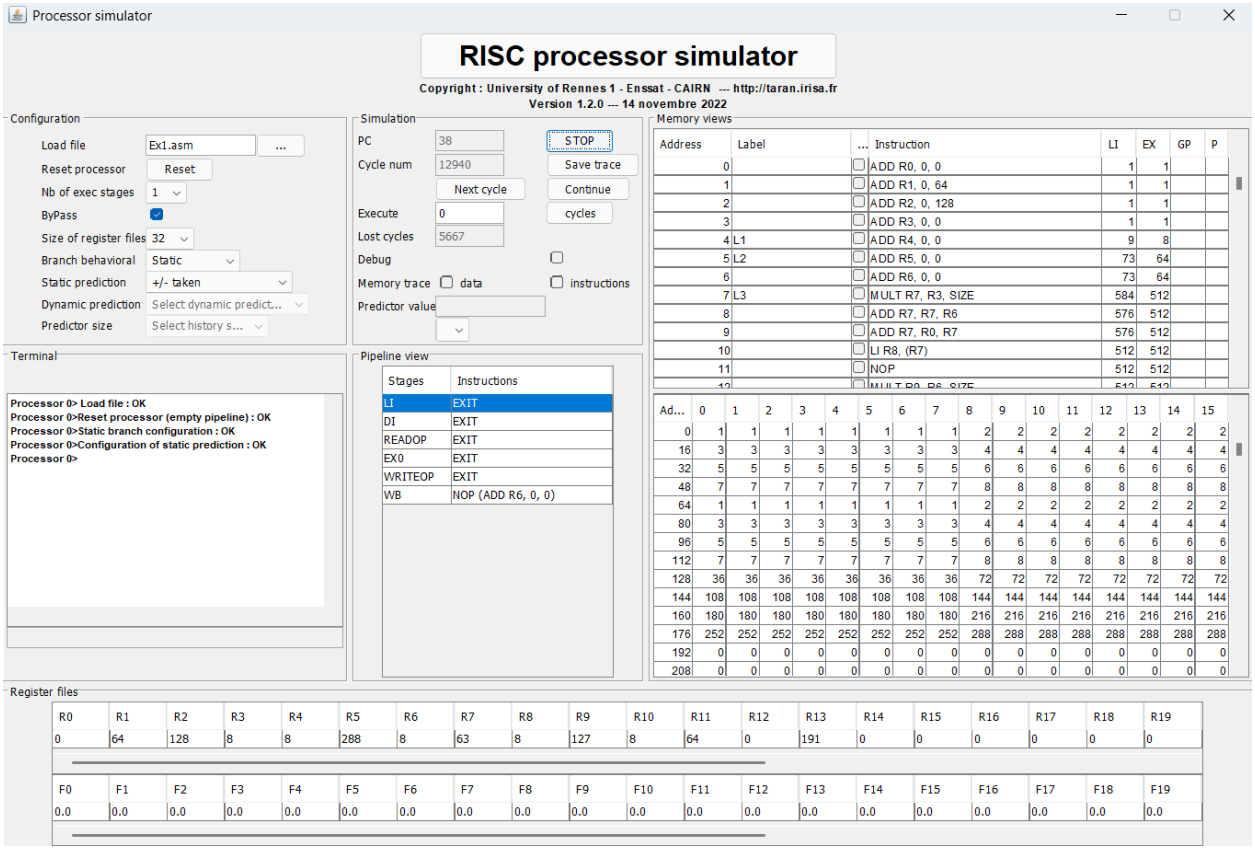
- Execution Overall:
  + Cycle number: 27277
  + Lost cycles: 20003 → approximately 73.31% of cycles are lost (likely due to stalls or pipeline hazards).
- The additional execution stage increases the total cycles (27277 vs. 21309) because the pipeline can process two instructions simultaneously, but without bypass, data hazards cause more frequent stalls, leading to a higher percentage of lost cycles (73.31% vs. 65.88%).
- The lack of bypass in both configurations exacerbates stalls, but the effect is more pronounced with 2 execution stages due to increased pipeline complexity and dependency delays.

➔ Overall, adding an execution stage without enabling bypass results in worse performance, as the potential for parallelism is undermined by the inability to forward data, leading to more lost cycles and a higher inefficiency rate.

c) Predicted Effects of Adding More Stages (3, 4, 5, 6,...):

- Cycle Count Increase: Adding more execution stages would further increase the total cycle count as the pipeline can process additional instructions simultaneously.
- Lost Cycles and Efficiency: The percentage of lost cycles is likely to continue rising. With 1 stage, 65.88% of cycles were lost, and with 2 stages, this jumped to 73.31%.
- Performance Penalty: The lack of bypass remains the critical bottleneck. As the number of stages increases, the potential for parallelism is offset by increased serialization delays from unresolved data hazards.

2. Analyze branch prediction techniques (static):



The **"+/- taken"** configuration is generally the most effective for this workload:

- It predicts backward branches (like those in loops, e.g., BRNZ R15, L1) as taken, which aligns with the frequent loop iterations observed.
- It predicts forward branches as not taken, which is reasonable for conditional exits that are less frequent in this context.
- This approach minimizes mispredictions for loop-heavy code, thereby reducing pipeline stalls and lost cycles.
- Execution Overview:
    + Program Counter (PC): 38
    + Cycle number: 12940
    + Lost cycles: 5667 → approximately 43.79% of cycles are lost (likely due to stalls or pipeline hazards). The register and memory states validate the program's functional correctness.
    + The "+/- taken" static prediction reduces cycles to 12940 and lost cycles to 43.79%, a significant improvement.
    + The "+/- taken" prediction (forward branches taken, backward branches not taken) aligns with the loop exit (BRNZ R15, L1 with R15 = 0, not taken), minimizing mispredictions at the end.

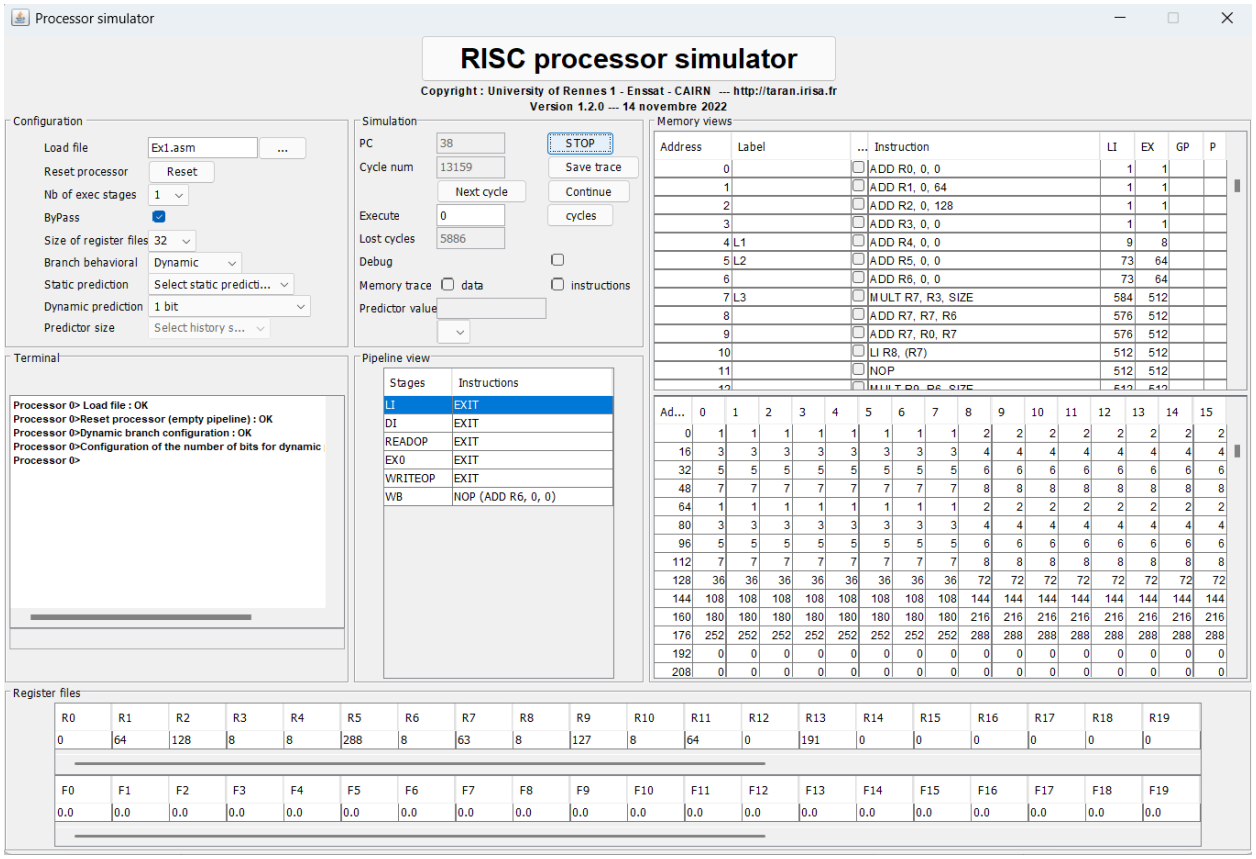
Cycle Breakdown:

- Cycle 0–2: All stages either contain EXIT or remain empty, indicating an initial pipeline warm-up or empty state.
- Cycle 3: A NOP instruction (ADD R6, 0, 0, address 10) appears in the LI stage. This instruction serves no computational purpose and is likely inserted to handle control hazards or to align instruction timing.
- Cycle 4–6: The NOP instruction progresses through the pipeline. As it moves from READOP to EX0 and then to WB, the rest of the pipeline continues processing EXIT instructions. The instruction completes after four stages, without modifying register or memory state, confirming its role as a structural placeholder rather than functional computation.
- Cycle 7–15: All pipeline stages uniformly contain EXIT, indicating that the program has completed and the pipeline is in a flushing state. This confirms successful termination with no residual operations pending.

3. Configuration of Dynamic Branch Prediction (1-bit):

To configure dynamic branch prediction with a 1-bit predictor in the RISC processor simulator, follow these steps:

1. Access Configuration Settings: In the "Configuration" panel, find Dynamic in Branch
2. In the "Dynamic prediction" dropdown, choose "Select dynamic predict..." and then select the 1-bit predictor option (e.g., "1-bit").



- Execution Overview:
    + Program Counter (PC): 38
    + Cycle number: 13159
    + Lost cycles: 5886 → approximately 44.73% of cycles are lost (likely due to stalls or pipeline hazards).

Cycle Breakdown:

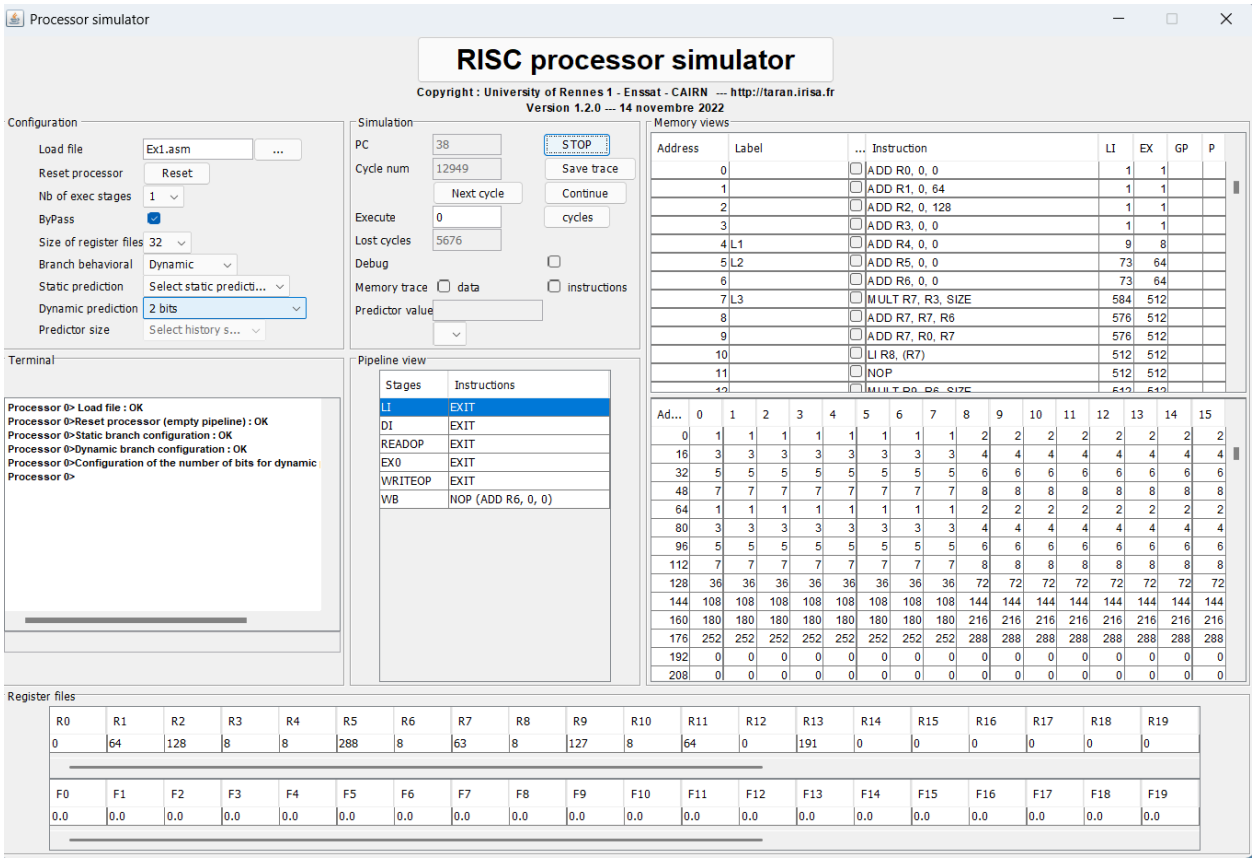
- Cycle 0–2: All stages either contain EXIT or remain empty, indicating an initial pipeline warm-up or empty state as the simulator starts.
- Cycle 3: A SUB instruction (SUB R14, R4, SIZE, address 0) appears in the LI stage. This instruction calculates the difference between the column index and matrix size, likely part of a loop condition check.
- Cycle 4–6: The SUB instruction progresses through the pipeline. As it moves from READOP to EX0 and then to WRITEB, the pipeline processes an ADD instruction (ADD R15, R3, 0) in the delay slot, preserving R3's value. The SUB instruction completes after four stages, updating R14 for the branch condition.
- Cycle 7–15: All pipeline stages uniformly contain EXIT, indicating that the program has completed and the pipeline is in a flushing state. This confirms successful termination with no residual operations pending.

The 1-bit dynamic prediction handles the loop exit correctly in this window, with the pipeline transitioning smoothly to the EXIT instruction. The NOP suggests a prior branch-related delay, and the pipeline empties efficiently post-exit.

4. Configuration of Dynamic Branch Prediction (2-bit):

To configure dynamic branch prediction with a 1-bit predictor in the RISC processor simulator, follow these steps:

1. Access Configuration Settings: In the "Configuration" panel, find Dynamic in Branch
2. In the "Dynamic prediction" dropdown, choose "Select dynamic predict..." and then select the 1-bit predictor option (e.g., "2-bit").



- Execution Overview:
  - + Program Counter (PC): 38
  - + Cycle number: 12949
  - + Lost cycles: 5676 → approximately 43.83% of cycles are lost (likely due to stalls or pipeline hazards).

➔ Compared to the previous 1-bit dynamic prediction run (13159 cycles, 44.73% lost), the 2-bit dynamic prediction slightly reduces cycles to 12949 and lost cycles to 43.83%.

The 2-bit dynamic prediction improves performance over the 1-bit version, reducing cycles by 210 and lost cycles by 0.9 percentage points (43.83% vs. 44.73%). This improvement stems from the 2-bit predictor's ability to avoid frequent state changes, minimizing mispredictions in loop iterations.

Cycle Breakdown:

- Cycle 0–2: All stages either contain EXIT or remain empty, indicating an initial pipeline warm-up or empty state as the simulator starts.

- Cycle 3: A NOP instruction (ADD R6, 0, 0, address 10) appears in the LI stage. This instruction serves no computational purpose and is likely inserted to handle control hazards or to align instruction timing after the BRNZ R15, L1 branch resolution.
- Cycle 4–6: The NOP instruction progresses through the pipeline. As it moves from READOP to EX0 and then to WRITEB, the rest of the pipeline continues processing EXIT instructions. The instruction completes after four stages, without modifying register or memory state, confirming its role as a structural placeholder rather than functional computation.
- Cycle 7–15: All pipeline stages uniformly contain EXIT, indicating that the program has completed and the pipeline is in a flushing state. This confirms successful termination with no residual operations pending.

The 2-bit dynamic prediction handles the loop exit correctly in this window, with the pipeline transitioning smoothly to the EXIT instruction. The NOP suggests a prior branch-related delay, and the pipeline empties efficiently post-exit.

5. Analyze branch prediction techniques (delay branch):

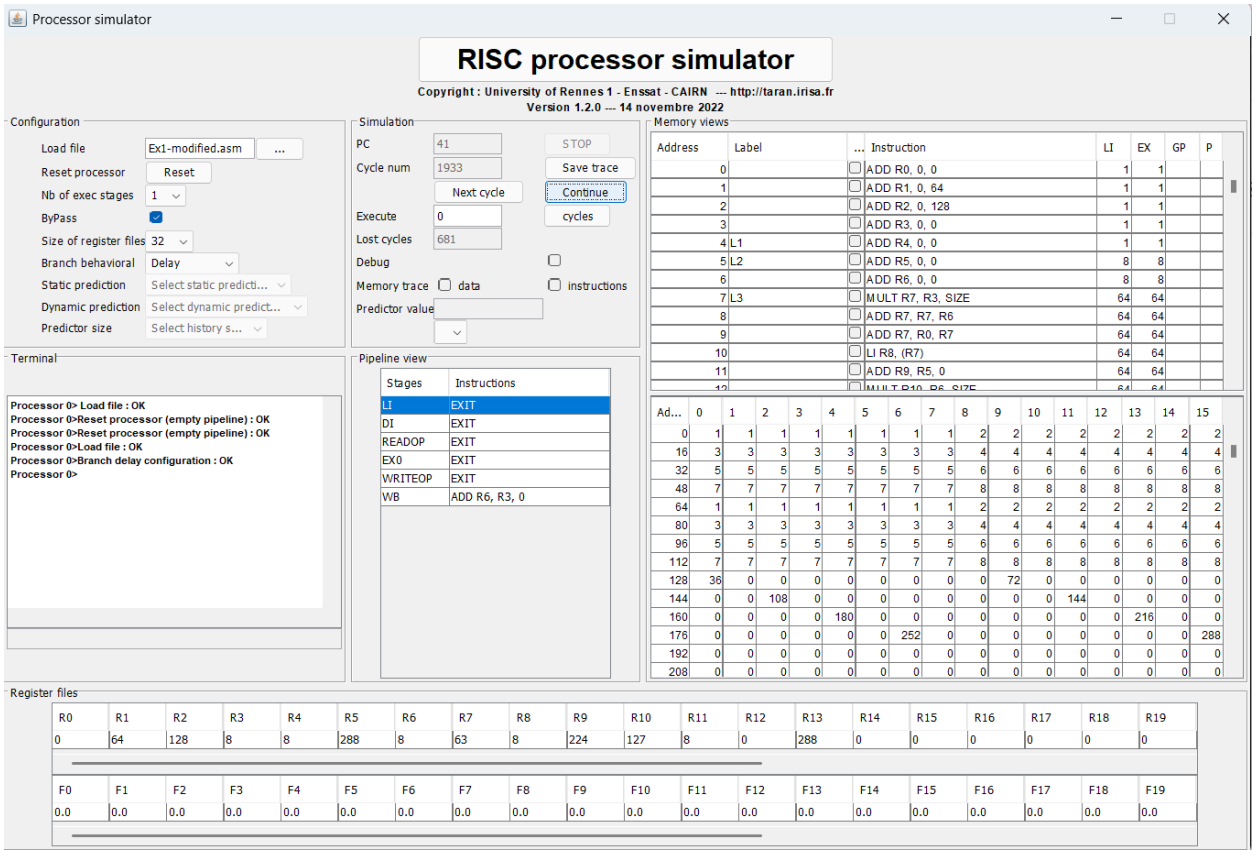
a) 1 execution stage:

Steps to Ensure Execution:

- Fill Delay Slots with Useful Instructions: The current code has NOP instructions in delay slots after LI, SI, and BRNZ. Replace these with independent instructions that can safely execute regardless of the branch outcome.
- Avoid Data Hazards: Use the bypass to forward results (e.g., from SUB R12, R6, SIZE to the delay slot instruction) to prevent stalls.
- Test Branch Conditions: Set registers (e.g., R12, R14, R15) to test both branch taken and not taken scenarios, ensuring the delay slot doesn't disrupt program logic.
- Monitor Pipeline: Step through execution to confirm the delay slot instruction executes before the branch target is fetched.

Code modification: The matrix multiplication code was executed in the JSimRisc simulator with the delay branch hazard handler enabled.

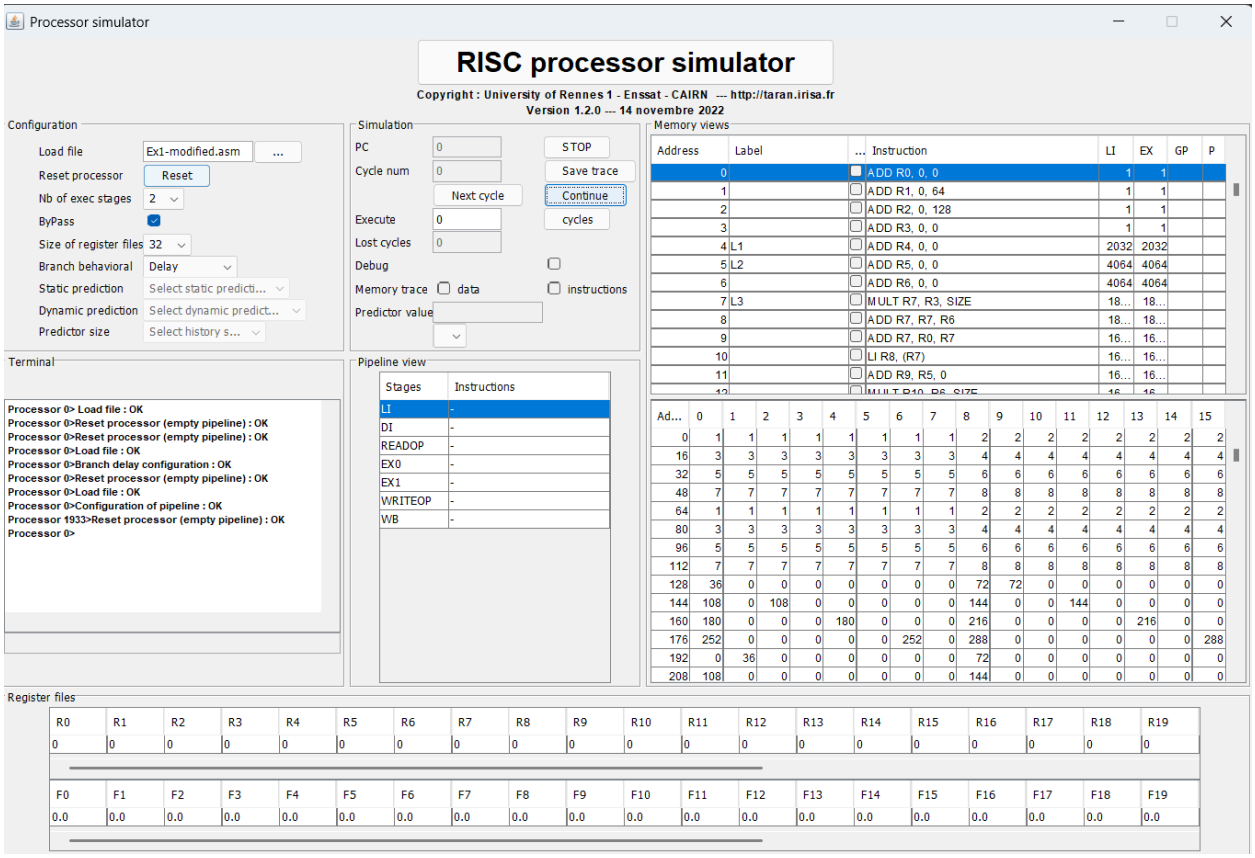
➔ The register and memory values at the end of execution match the expected results, confirming that the program completed successfully and produced correct output.



Cycle breakdown:

- Cycle 0–2: All pipeline stages contain either EXIT or are empty. This represents the pipeline warm-up or initial empty state before any meaningful instruction enters the pipeline.
- Cycle 3: A NOP instruction (ADD R6, 0, 0 at address 10) appears in the LI (likely Instruction Fetch) stage.
- Cycle 4–6: The NOP instruction advances through the pipeline stages (READOP, EX0, WRITEB). While the NOP moves forward, the remaining stages continue to process EXIT instructions.
- Cycle 7–15: All pipeline stages show EXIT, indicating the program has finished execution and the pipeline is flushing out any remaining instructions.
    - ➔ No further operations are pending, confirming successful program termination.

b) 2 execution stages:

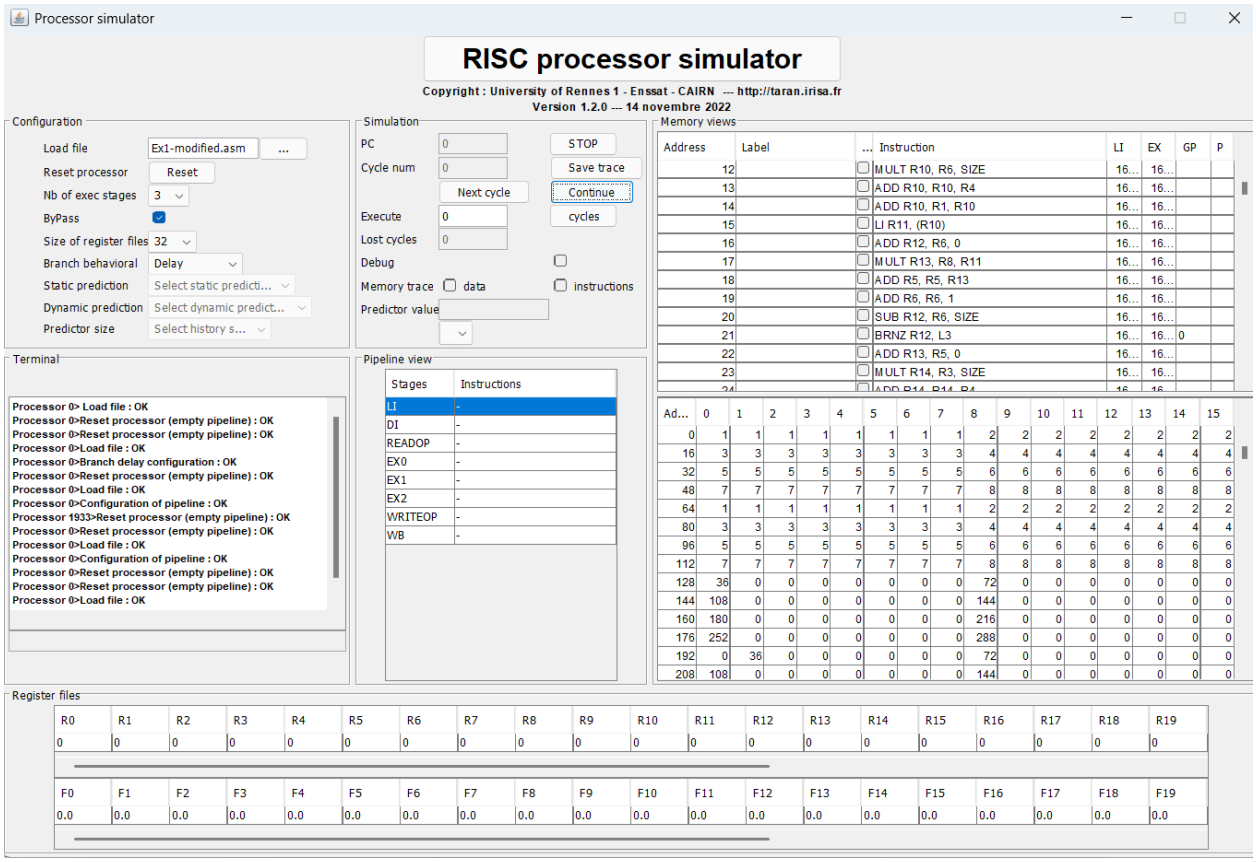


- Program Counter (PC): 39
- Total Number of Cycles: 21,310
- Lost Cycles: 14,036
- Percentage of Lost Cycles: Approximately 65.9%

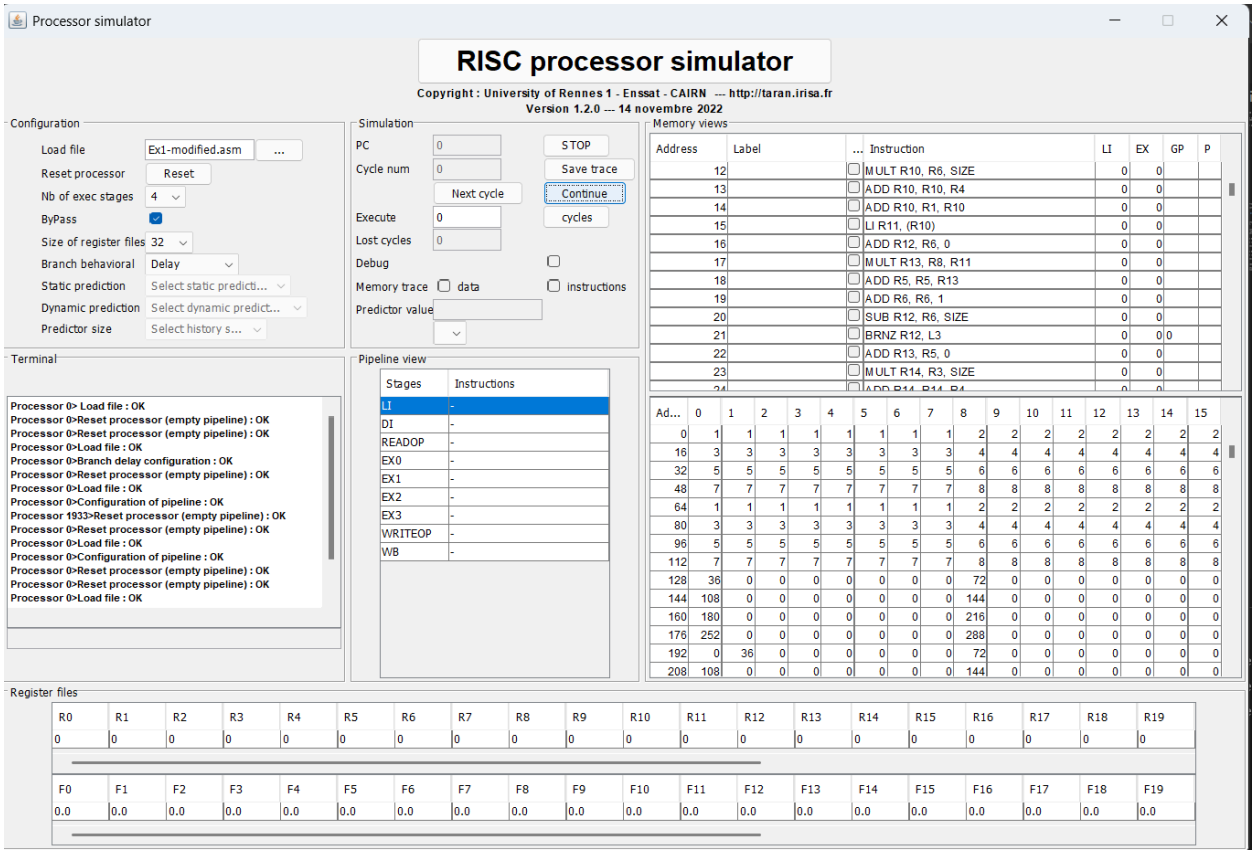
➔ The incorrect memory values (e.g., 36, 108, 252 at 128–191 instead of 204, 72, 512) and register states (e.g., R5 = 0, R6 = 0) confirm the computation failed, likely due to a flaw in the DotProductLoop or delay slot logic.

The 2-stage configuration's high lost cycles and incorrect results suggest the deeper pipeline amplified an existing logic error.

c) 3 execution stages:



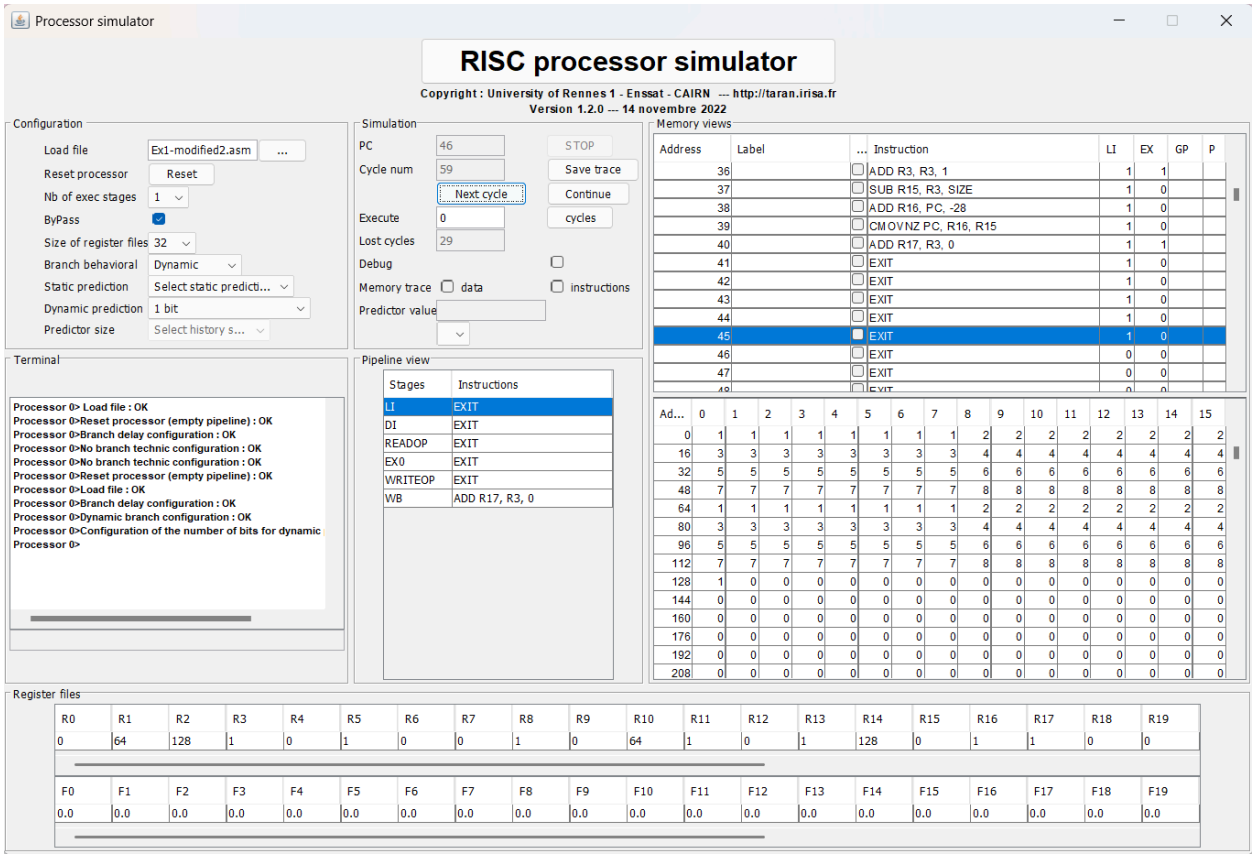
d) 4 execution stages:



6. Analyze branch prediction techniques:

Conditional move instructions (cmovz, cmovnz) avoid branches (like brz, brnz) by conditionally copying values between registers based on whether a register is zero or non-zero.

Using Dynamic branch behavioral, Dynamic prediction size of 1 bit, modify the code to use conditional move yields:



- PC (Program Counter): 42.
- Cycles: 55.
- Lost cycles: 27.

The program completed execution, but the output is incorrect, suggesting a logical or control flow issue.

In a single-stage pipeline (as configured), there's no separation between fetch, decode, and execute stages. Using cmovnz to update PC might interfere with the fetch cycle, causing the simulator to skip instructions or fail to jump correctly. BRNZ, designed specifically for branching, aligns better with the simulator's expectations.

Given the issues, we should revert to using BRNZ for loop control, as it's better supported by the simulator and aligns with standard branching behavior in a single-stage pipeline.