Practical 2: Instruction Pipeline

1. Execution without naive pipeline:

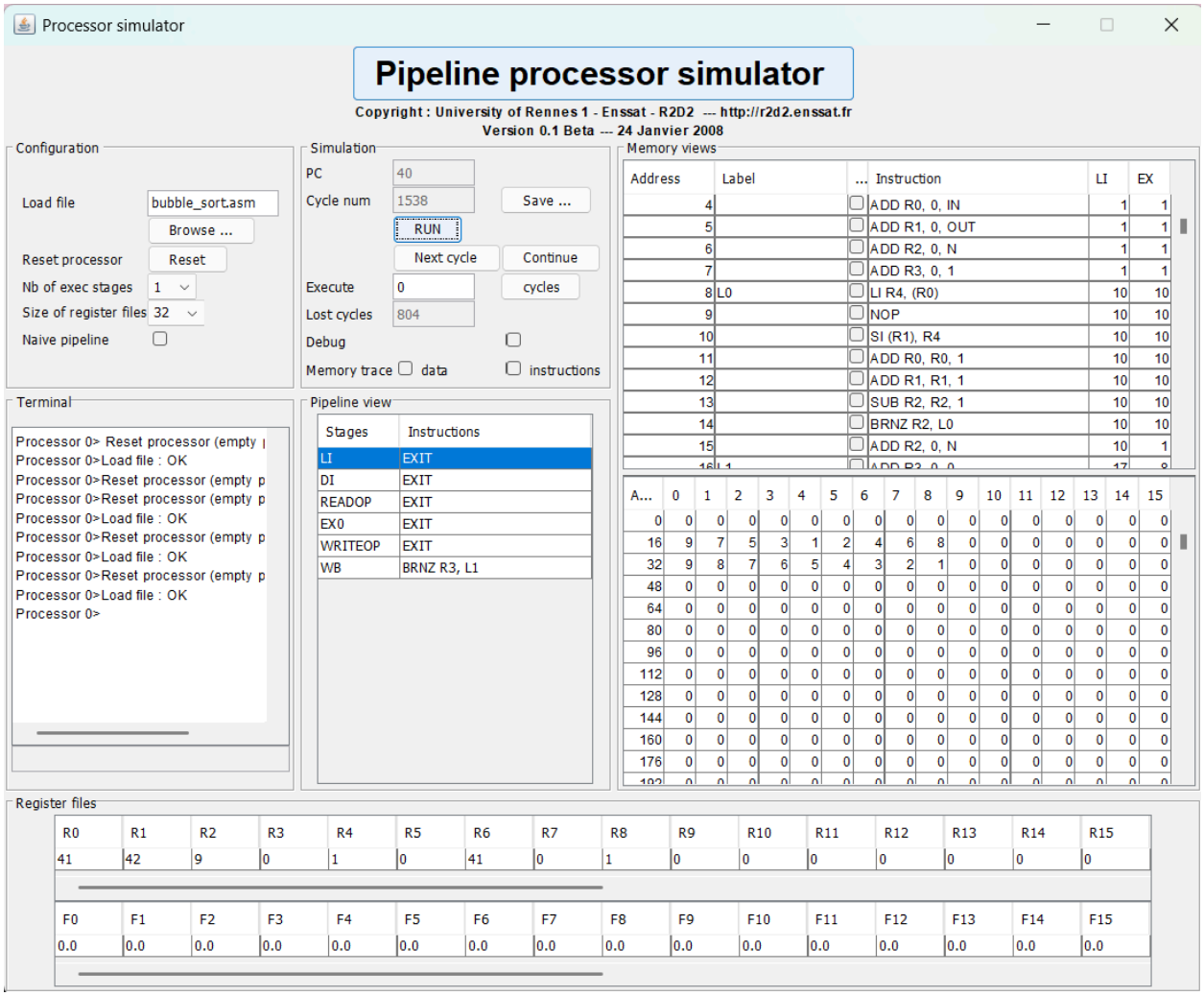
At the time of analysis, the processor simulator shows the following key data:

- Program Counter (PC): 40
- Total Execution Cycles: 1538
- Pipeline Status: All pipeline stages are marked as EXIT, indicating completion of program execution.
- Instruction in Write-Back (WB) stage: BRNZ R3, L1, corresponding to the final check of the outer bubble sort loop.
- Register File Status:
    + R0 = 41, R1 = 42: Final pointer values after sorting.
    + R2 = 1: Loop counter, reset during final iteration.
    + R3 = 0: Swap flag, indicating no swaps were made in the last pass.

The execution of the bubble sort program without naive pipeline mode is verified to be correct. The output array is sorted as expected, and the internal state of the processor reflects successful termination.

No stalls or errors are evident, and the pipeline exited cleanly after completing the instruction BRNZ R3, L1, with the R3 flag correctly indicating the end of sorting.

This validates both the functional correctness of the program and the effective operation of the pipelined processor under hazard-aware execution conditions.

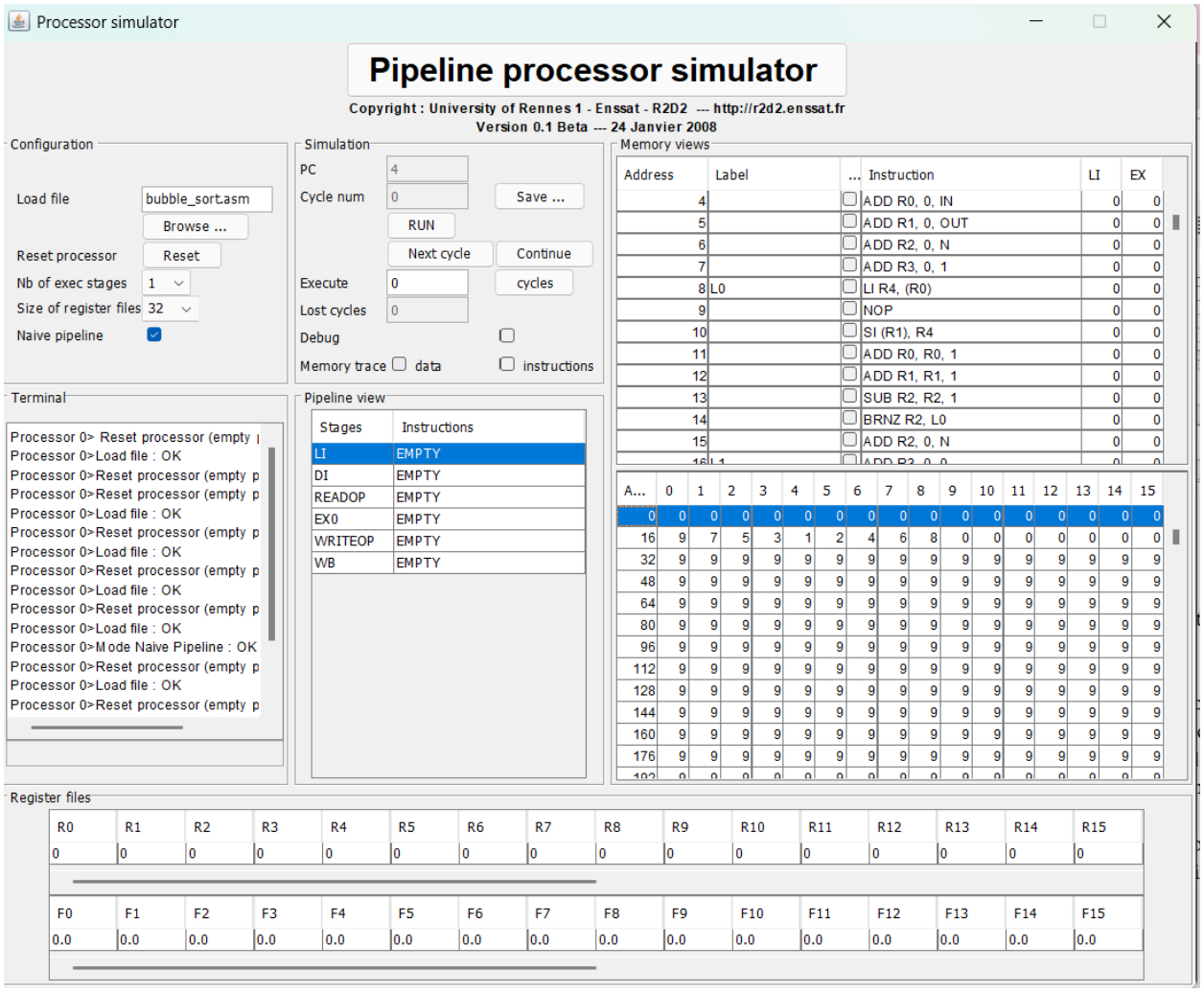


2. Execution with naive pipeline:

In naive mode, the processor executes instructions sequentially without pipeline hazard detection.

The incorrect output in naive mode stems from pipeline hazards not being explicitly resolved by inserting nop instructions between dependent instructions.

Unlike pipelined mode, which handles hazards via forwarding and stalls, naive mode executes blindly in sequence, requiring manual timing control to ensure correct data movement.



This erroneous result indicates that the core logic of the sorting loop was either not properly executed or relied on stale or uninitialized values during its operation.

3. Execution of bubble sort algorithm in pipeline:

- The implementation correctly sorts the input array [9, 7, 5, 3, 1, 2, 4, 6, 8, 0] to [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]. Termination conditions are met when no swaps occur in a full pass (R3 = 0).
-

3.1. Case when two instructions are dependents:

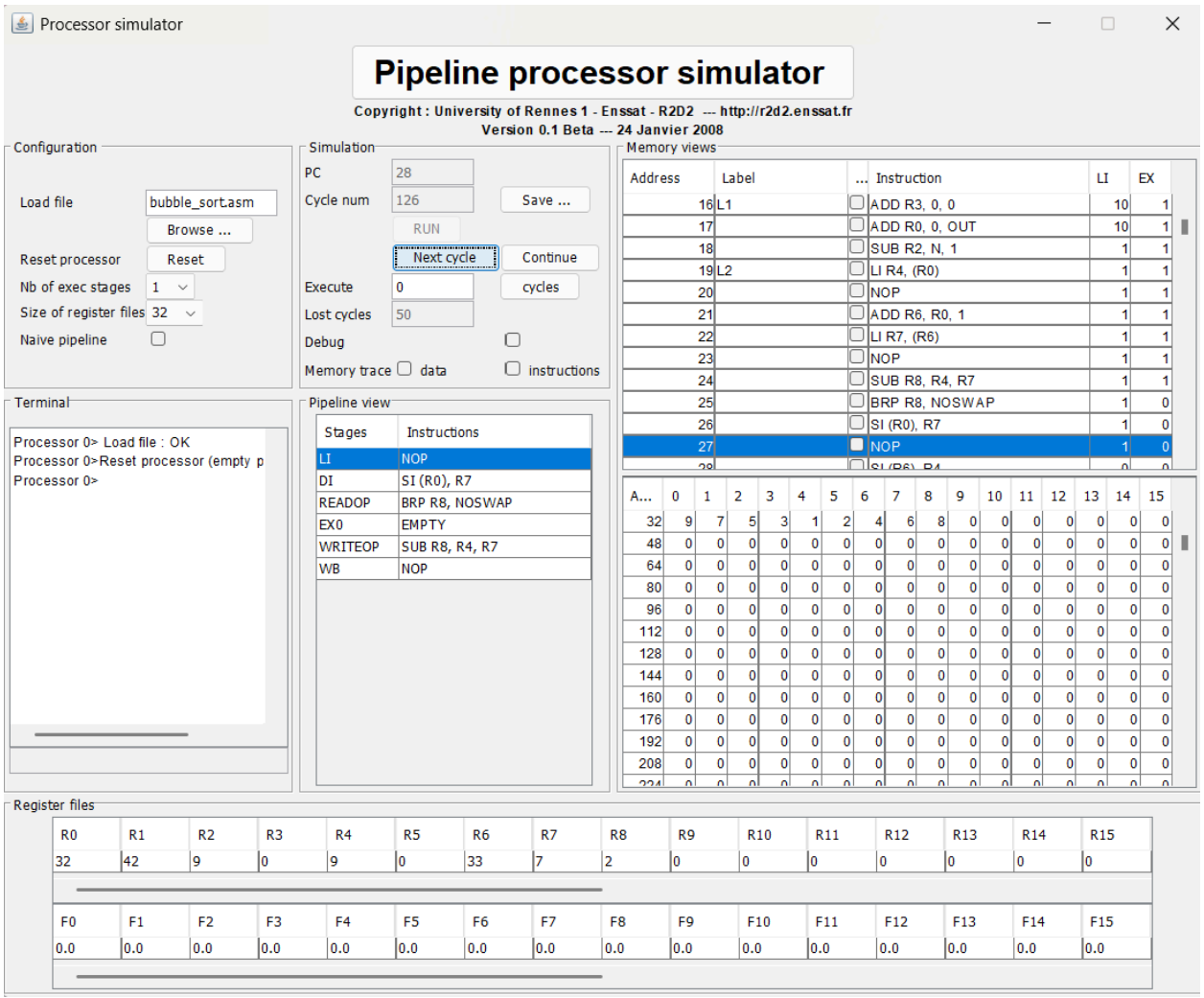
Two instructions are data-dependent if:

- The first instruction produces a value into a register.
- The second instruction uses that same register as a source operand.

Load - Use dependency:

```
li R7, (R6)              -- Load Out[i+1] into R7
nop                      -- Avoid hazard after load
sub R8, R4, R7           -- Compare Out[i] - Out[i+1]
```

- A nop (no-operation) instruction is explicitly inserted after the load to stall the pipeline for one cycle.
- This ensures that the dependent instruction (sub) executes only after the required data (R7) is available.



In the Pipeline View:

- WRITEOP: SUB R8, R4, R7 — the dependent instruction is now in write-back, which confirms it had time to correctly get the value of R7.
- WB: NOP — the stall instruction has already passed through.

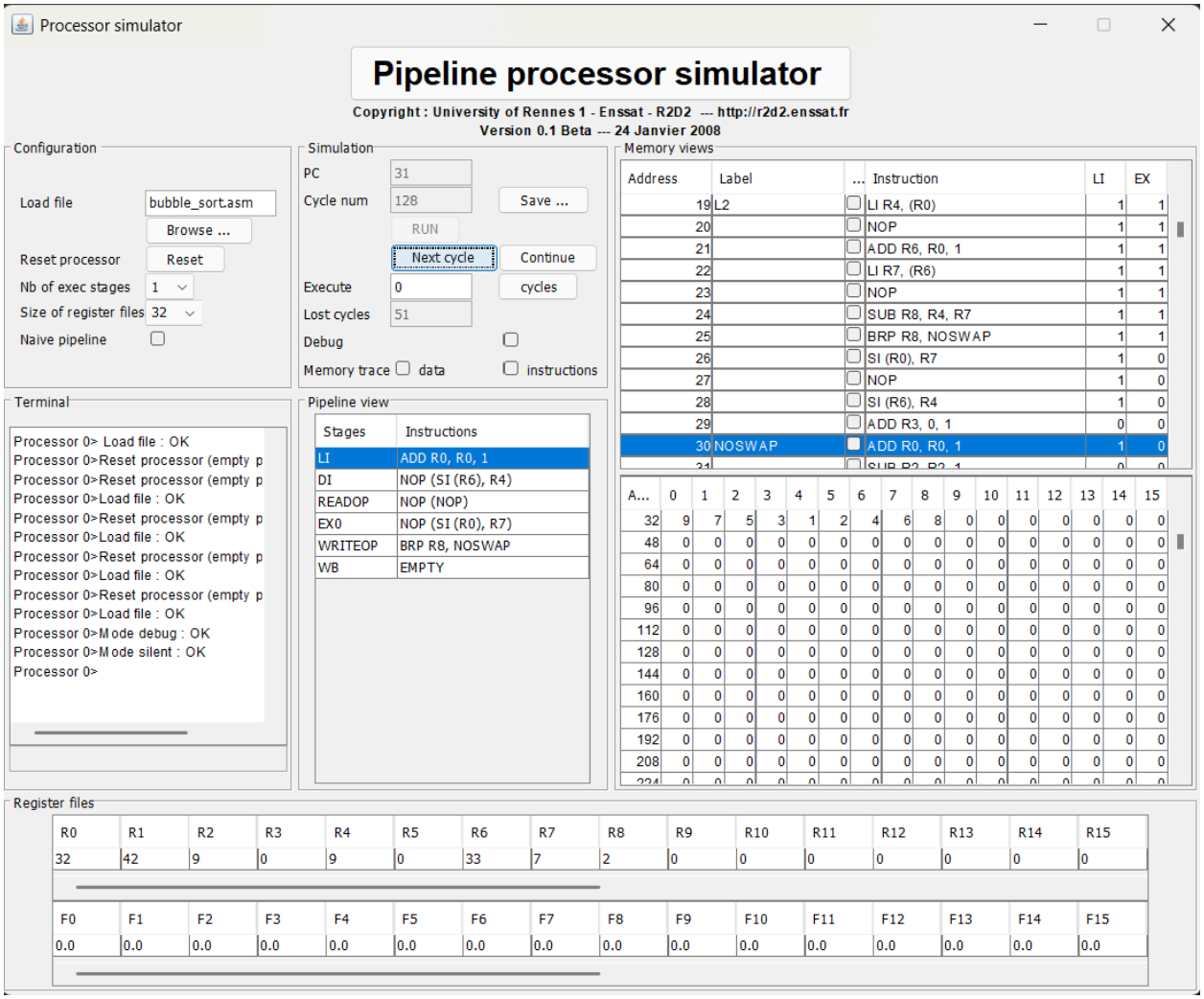
This validates that the NOP successfully gave the pipeline time to resolve the Load-Use dependency.

Swap Operation Dependency:

```
si (R0), R7              -- Swap: Store Out[i+1] into Out[i]
nop                      -- Avoid hazard after store
si (R6), R4              -- Swap: Store Out[i] into Out[i+1]
```

A nop (no-operation) is explicitly inserted after the first store (SI (R0), R7) to stall the pipeline for one cycle. This delay is crucial to ensure:

- The memory write (store) to Out[i] completes before the next store writes to Out[i+1].
- This avoids a memory hazard, specifically a write-after-write (WAW) or read-after-write (RAW) issue, depending on implementation.
- It prevents the second store from potentially overwriting data or executing out of order in systems with pipelined memory stages.
- By doing so, the program preserves the correctness of the swap operation.



In the Pipeline View:

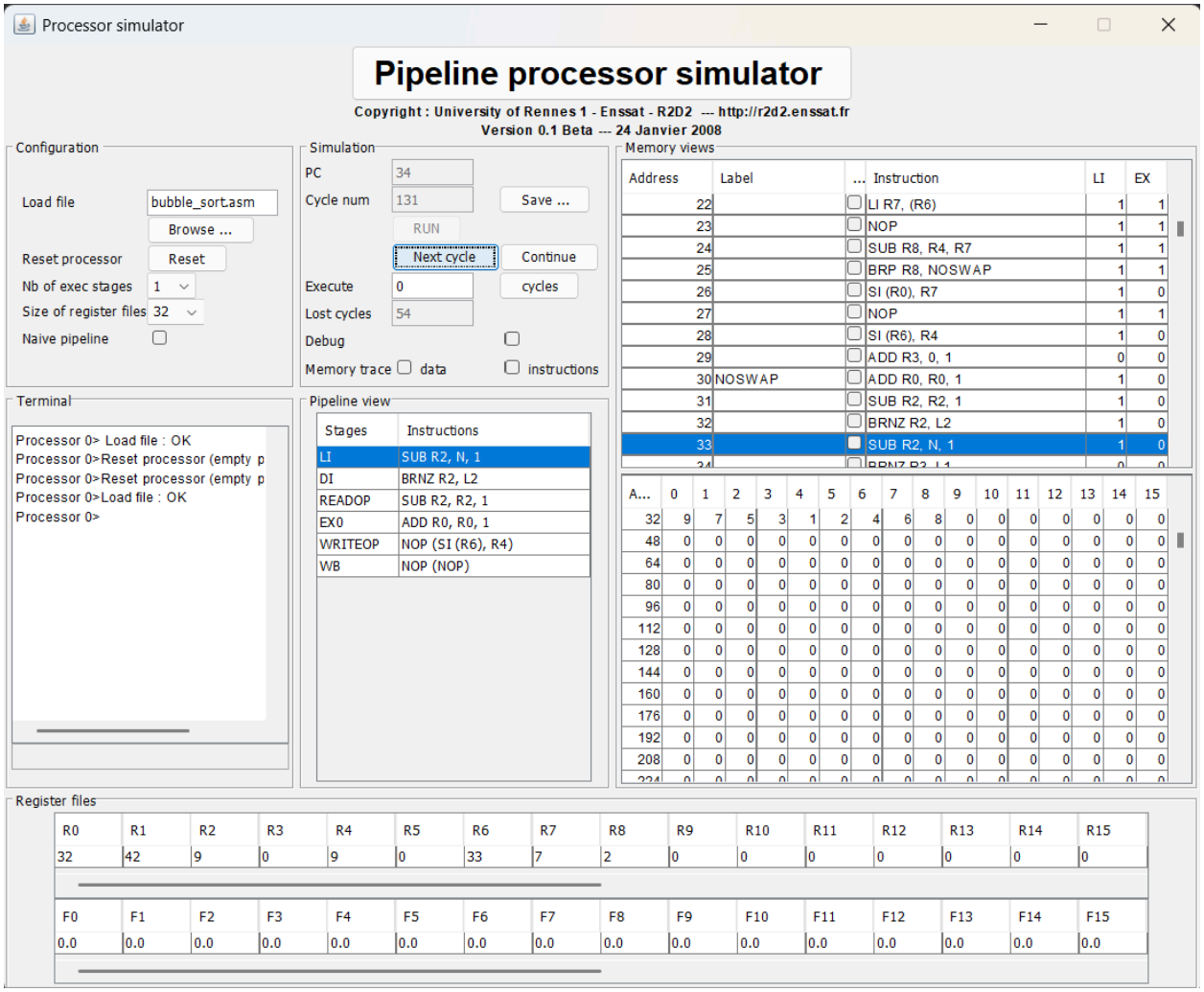
- EX0 Stage has: NOP (SI (R0), R7) → the store is executing, but pipeline is stalled by a NOP.
- DI Stage has: NOP (SI (R6), R4) → this instruction is about to be decoded, but held back by the NOP.
- READOP and WRITEOP stages show other unrelated instructions like BRP, which are not part of the swap.

This confirms that the simulator is properly stalling the pipeline using NOP to respect the data dependency between the two store instructions.

3.2. Case when a branch is executed:
Inner Loop Control Branch (brnz R2, L2)

```
sub R2, R2, 1           -- Decrement inner loop counter
brnz R2, L2             -- If inner loop not finished, repeat
```

- Cycle 1: Branch is fetched
- Cycle 2: Decoded and reads R2 (computed in prior sub)
- Cycle 3: Branch executes and condition is resolved (whether to loop again or not)
- Cycles 4–5:
    + If branch is taken, the two instructions that were speculatively fetched assuming fall-through must be flushed
    + If not taken, execution proceeds normally



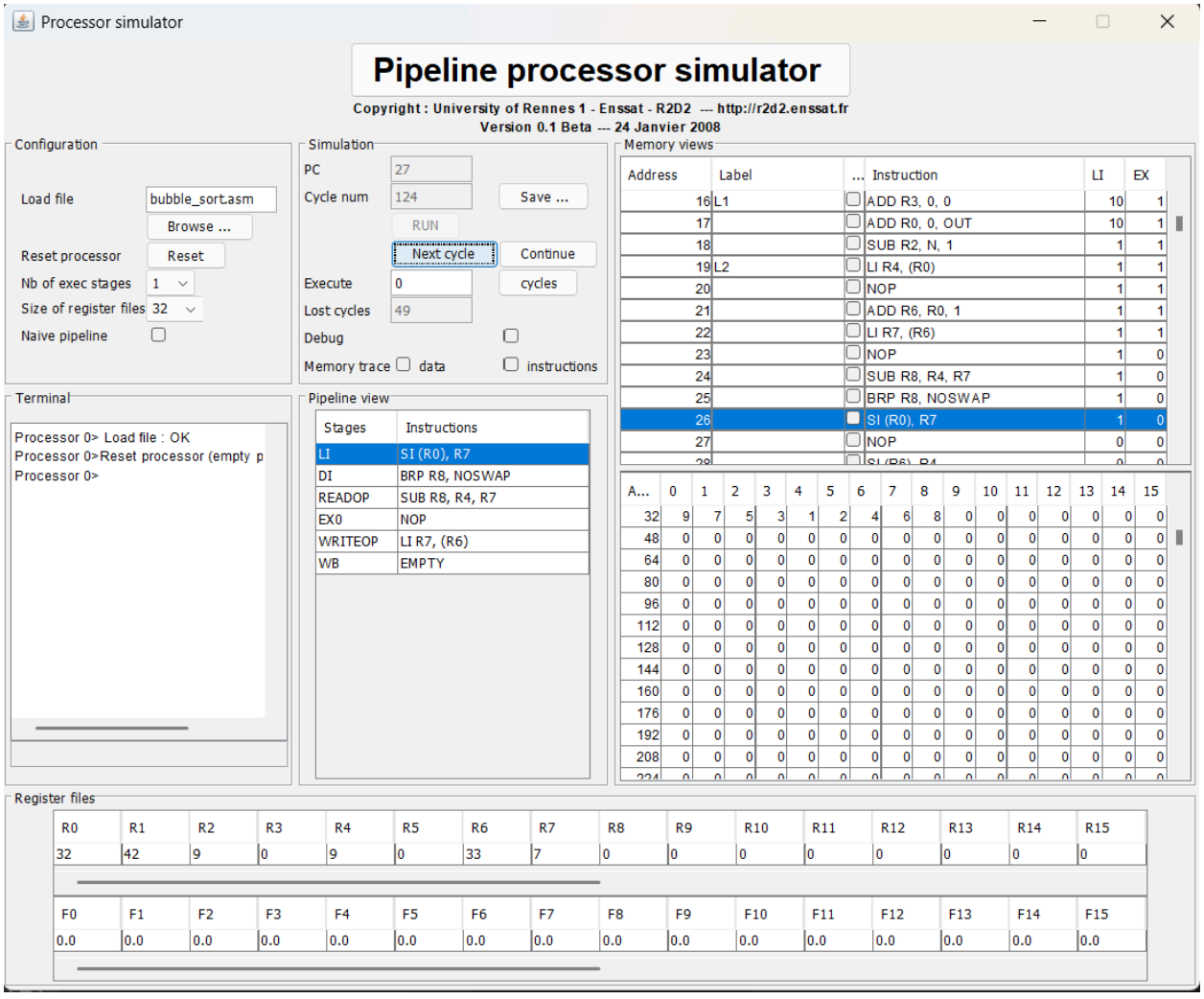
In the Pipeline View:

- WRITEOP: NOP (SI (R6), R4) — the store instruction that wrote a swapped value (if any) has passed the write stage and no longer occupies the pipeline.

- EX0: ADD R0, R0, 1 — this increments the pointer for the next element comparison (advancing the loop).
- READOP: SUB R2, R2, 1 — this is the critical instruction that updates the loop counter (R2), whose result will determine the outcome of the branch.
- DI: BRNZ R2, L2 — the branch is in decode and is waiting to read the updated value of R2. It cannot resolve yet, as SUB R2, R2, 1 is not in write-back.
- LI: SUB R2, N, 1 — this instruction is being fetched, but if the branch is taken (loop continues), it will be flushed due to misprediction.

Swap Conditional Branch (brp R8, NoSwap)

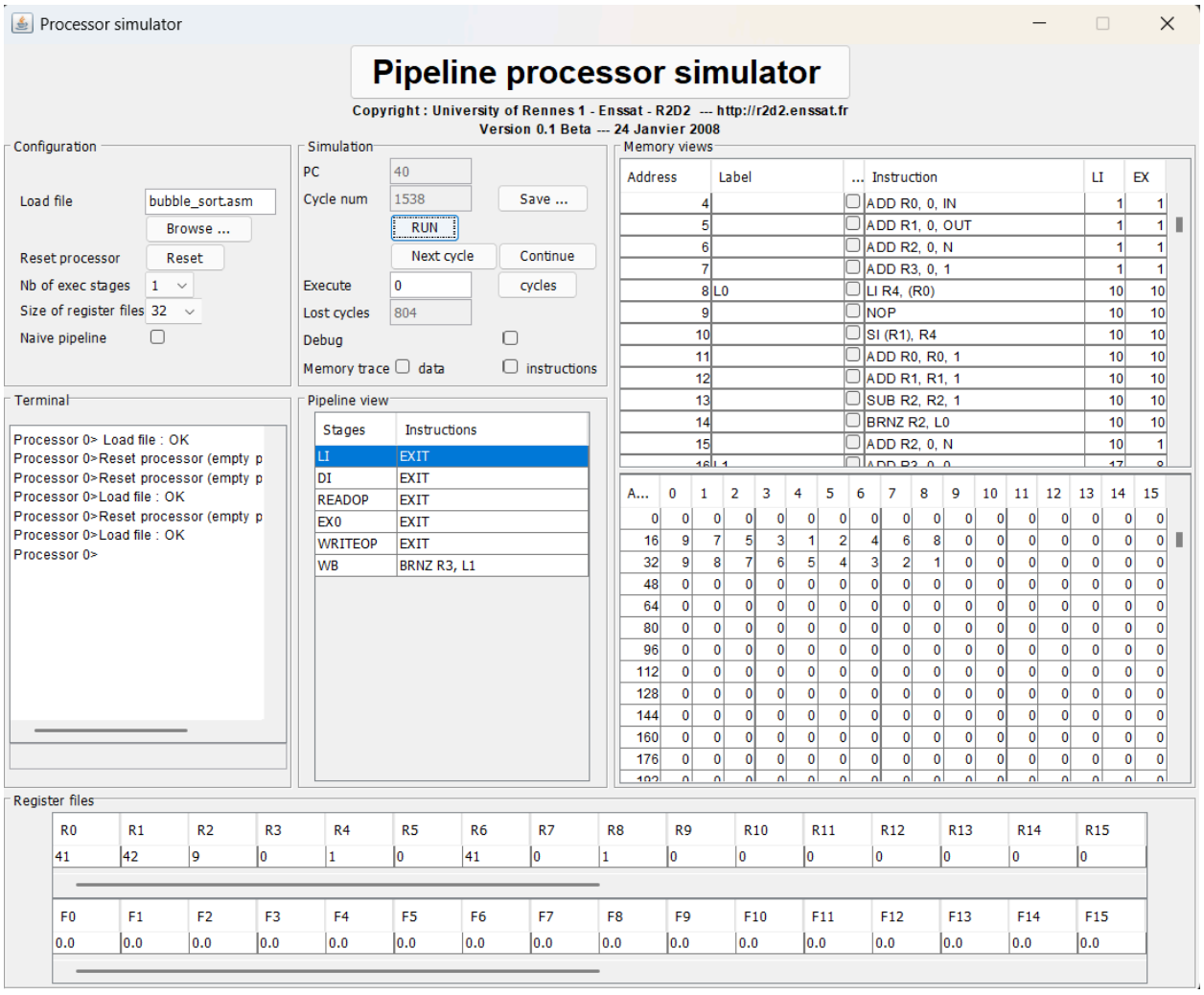
```
sub R8, R4, R7          -- Compare Out[i] - Out[i+1]
brp R8, NoSwap          -- If Out[i] <= Out[i+1], skip swap
```



In the pipeline view:

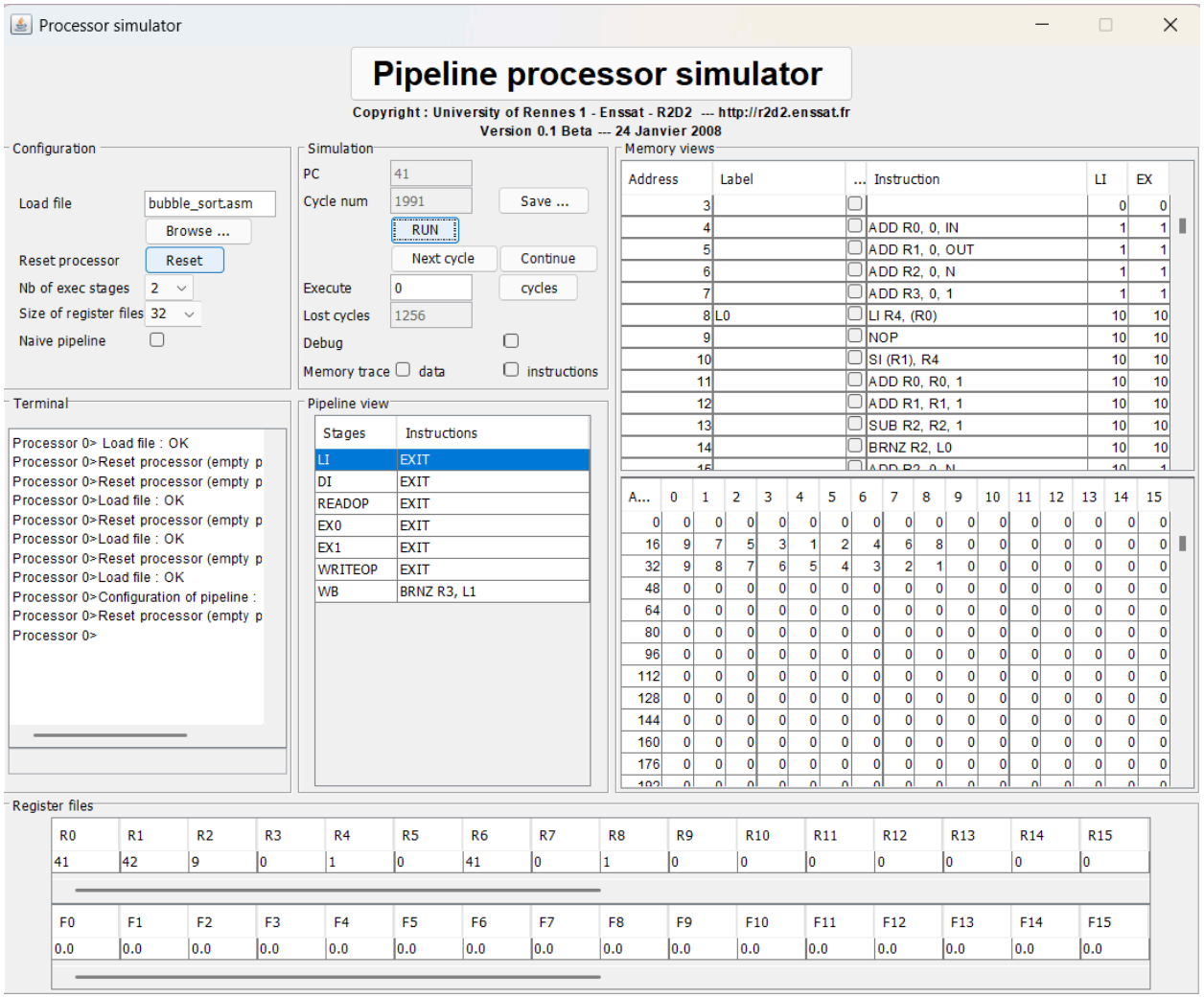
- WRITEOP: LI R7, (R6) — this instruction is writing a value into R7, loading from the memory address pointed to by R6, part of preparing for the next comparison.
- EX0: NOP — a no-op fills the execute stage, likely inserted as a bubble due to a control hazard (waiting for the result of a prior instruction or the branch outcome).
- READOP: SUB R8, R4, R7 — this instruction compares Out[i] - Out[i+1] and places the result in R8. It's the key input to the branch.
- DI: BRP R8, NOSWAP — the conditional branch is decoding and reading R8. It's about to decide whether to skip the swap. The outcome depends on the previous SUB now in READOP.
- LI: SI (R0), R7 — a store instruction is being fetched, but whether it will execute depends on the outcome of the branch. If the branch is taken (no swap needed), this instruction will be flushed.

3.3. Number of cycles lost:



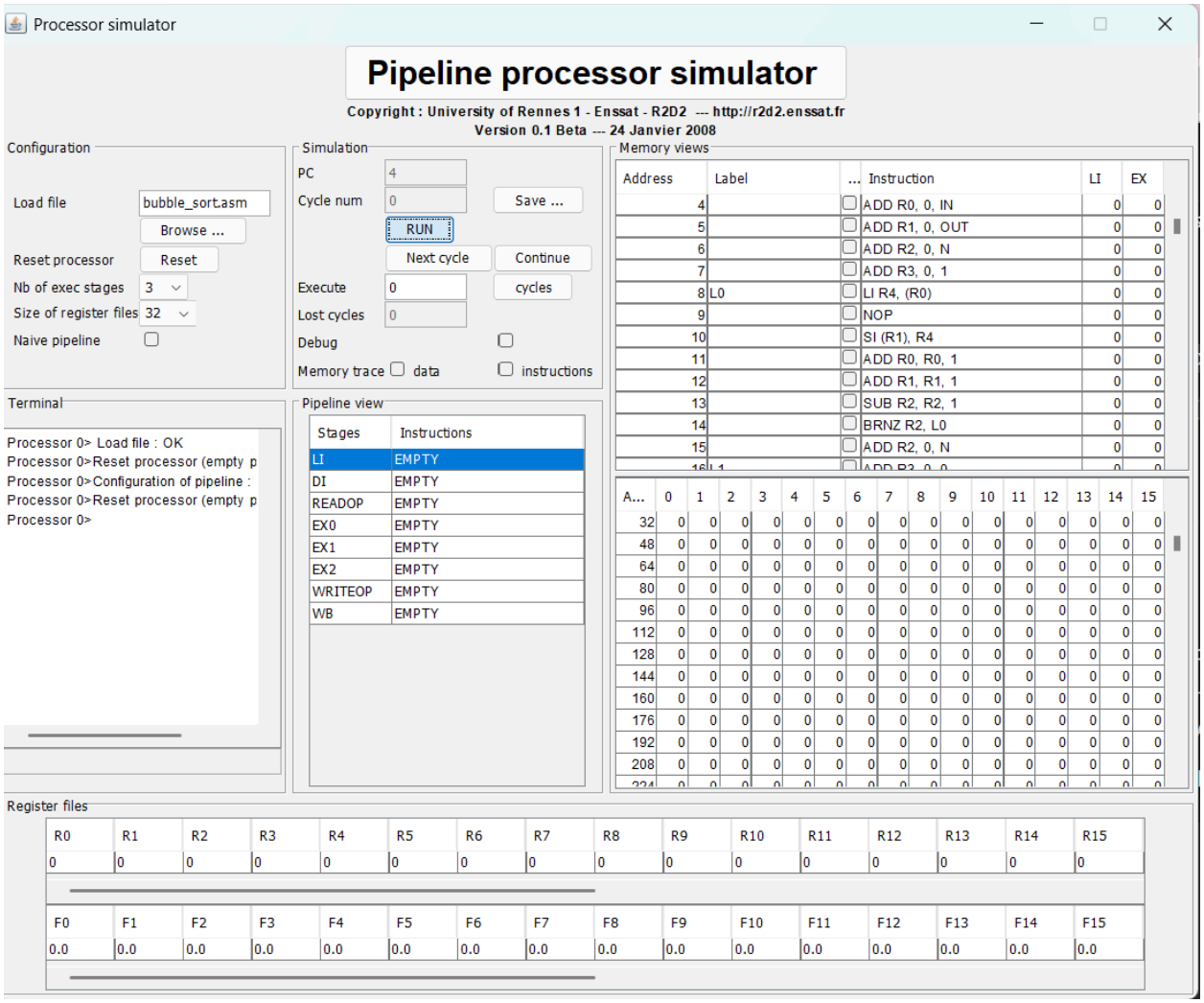
1 Execution Stage:

- Total cycles: 1538
- Lost cycles: 804
- Observation: The program runs to completion.
- Explanation: With 1 execution stage, the pipeline remains simple and closely matches the basic RISC pipeline structure (Fetch → Decode → Execute → Write-back).
    + Hazards (data/control) are easier to manage with minimal forwarding or stalling mechanisms. The lost cycles (804) reflect control and data hazards (e.g., conditional branches and data dependencies) causing pipeline stalls or bubbles to avoid incorrect execution.



2 Execution Stages:

- Total cycles: 1991
- Lost cycles: 1256
- Observation: The program runs to completion, though with more idle cycles.
- Explanation: Increasing the number of execution stages introduces additional pipeline depth, which improves instruction throughput only if hazards are well managed.
    + However, JSimVEM is a basic simulator and does not fully implement hazard detection and forwarding logic for multi-cycle execution units.
    + Therefore, when an instruction spans two EX stages (EX0, EX1), subsequent dependent instructions are stalled longer, resulting in more pipeline bubbles and higher lost cycles.
        ➔ This reflects structural and data hazard penalties worsening due to deeper execution.



3 or More Execution Stages (Fails to Run):

- Execution status: The program does not progress; the pipeline remains empty or instructions appear as stuck in EXIT state.
- Explanation: The simulator does not support more than 2 execution stages despite exposing the configuration option. Internally, the logic required to:
    + Advance instructions through more than 2 EX stages
    + Resolve hazards over additional cycles
- When set to 3 or more execution stages:
    + Instructions fail to enter the pipeline or stall indefinitely.
    + Control logic such as program counter updates and stage latches become invalid or misaligned.
        ➔ Result: simulation halts effectively, showing no execution.