University of Science and Technology of Hanoi

Advanced Computer Architecture and x86 ISA

Lecturer: Ms. Le Nhu Chu Hiep

MIDTERM PROJECT

Implementation of a perceptron in machine learning context

Student: Luong Quynh Nhi - 23BI14356

Jun, 2025

Implementation Overview

- Goal: Implement the Perceptron algorithm on RISC simulator (JSimRISC), simulating its operations via a pipelined processor
- Algorithm Summary: The perceptron performs a weighted sum of inputs, adds a bias, and outputs 1 if the result > 0, otherwise 0.
- Key Formula: Output = 1 if (∑(input[i] * weight[i]) + bias) > 0 else 0
- Architecture**:**
    - ISA: RISC-style with instructions like LI, SI, MULT, ADD, MOVE, BRZ, etc.
    - Pipeline: 6 or 8 pipeline stages
    - Bypass: Enabled/Disabled for hazard mitigation
    - Branch Prediction: Static or Dynamic (as configured)

Input Mechanism

- Where Inputs Are Stored: Inputs are defined in the .data section and loaded using li and pointer-based li R4, (R1) syntax.
- Input vectors and weights are manually inserted at memory addresses.
- Sample input:

```
.data 0

.global inputs

1 0 1

.data 32

.global weights

2 -1 3

.data 80

.global bias

-2
```

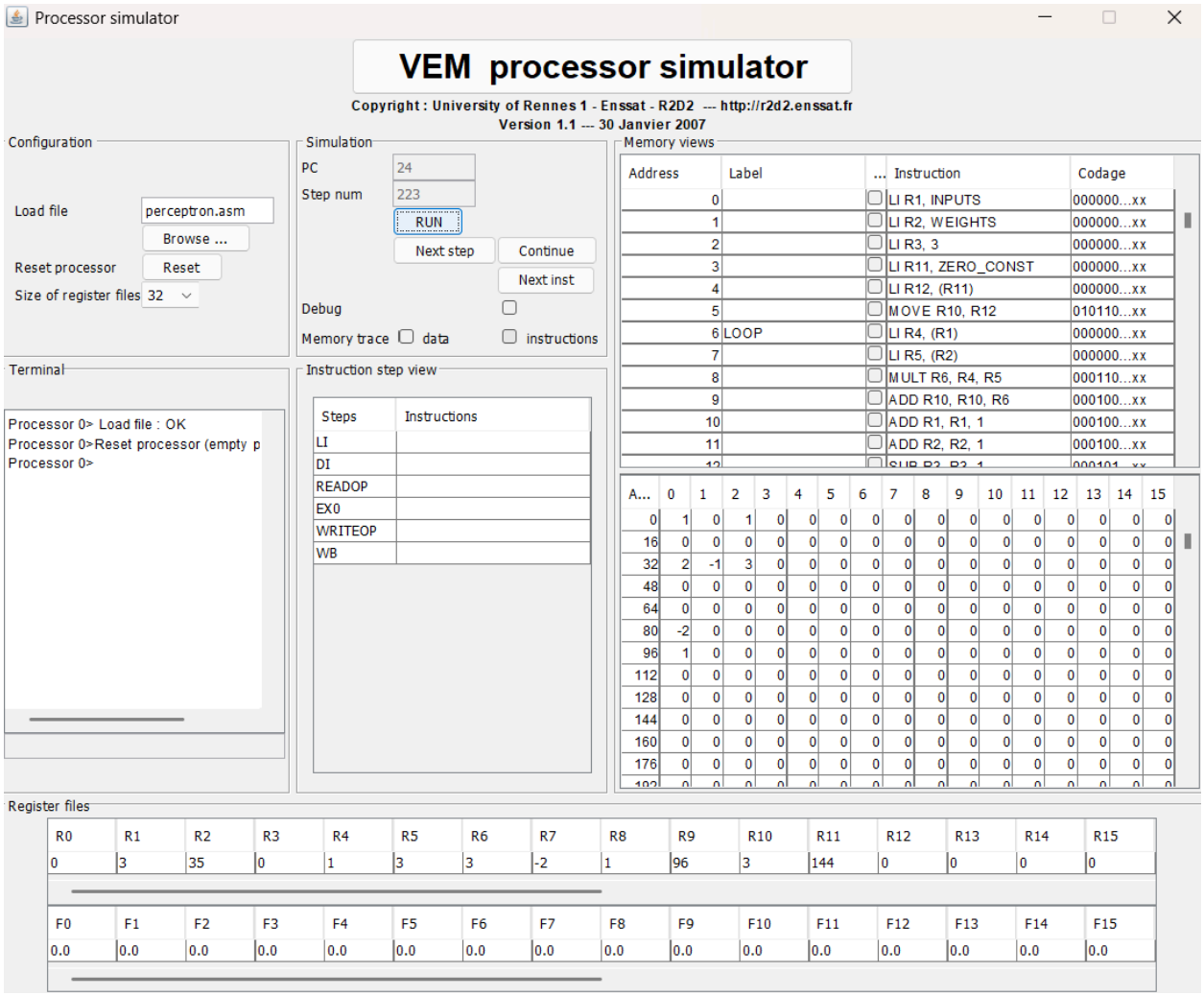
Output Mechanism

- Where Output Is Stored: Output is stored at memory address defined in the .data section under label result.
- How Output Is Written: Output value (0 or 1) is stored using si:

```
.data 96
.global result
0
```

```
.store_result
li r9, result
si (r9), r8
```

1. Step 1:



Program Execution Flow:

a) Initialization (PC = 0-3):

Instruction	Effect
LI R1, INPUTS	R1 ← 0 (input array starts at addr 0)
LI R2, WEIGHTS	R2 ← 16 (weights start at addr 16)
LI R3, 3	R3 ← 3 (loop count = 3 inputs)

LI R11, ZERO_CONST	R11 ← 80 (address of 0 constant)
LI R12, (R11)	R12 ← 0 (from memory[80])

b) Bias Addition (PC = 13-15):
- Loads bias (-2) → R7
- Adds to accumulator: R10=5 + (-2) = 3

c) Activation Function (PC=16-22):
- Checks R10 = 3 (positive)
- Sets R8 = 1 (positive classification)
- Stores result at address 96

Register States Post-Execution:

Register	Value	Purpose
R1	3	Points to end of inputs array
R2	35	Points to end of weights array
R3	0	Loop counter (completed)
R4	1	Last input value loaded
R5	3	Last weight value loaded
R6	3	Last multiplication result
R7	-2	Bias Value
R8	1	Final Output
R9	96	Result memory address

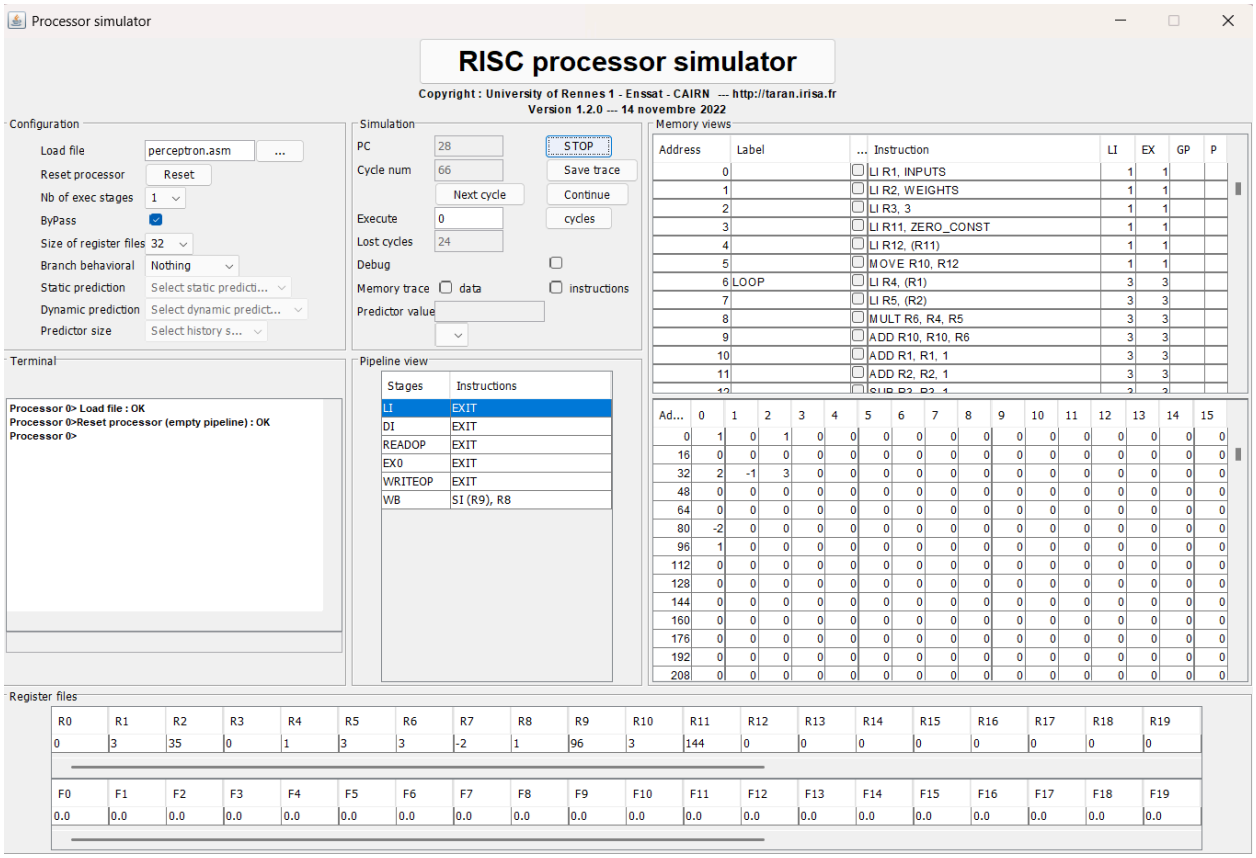
R10	3	Final weighted sum (5 + (-2)) = 3
R11	144	zero_const address
R12	0	Temporary zero value

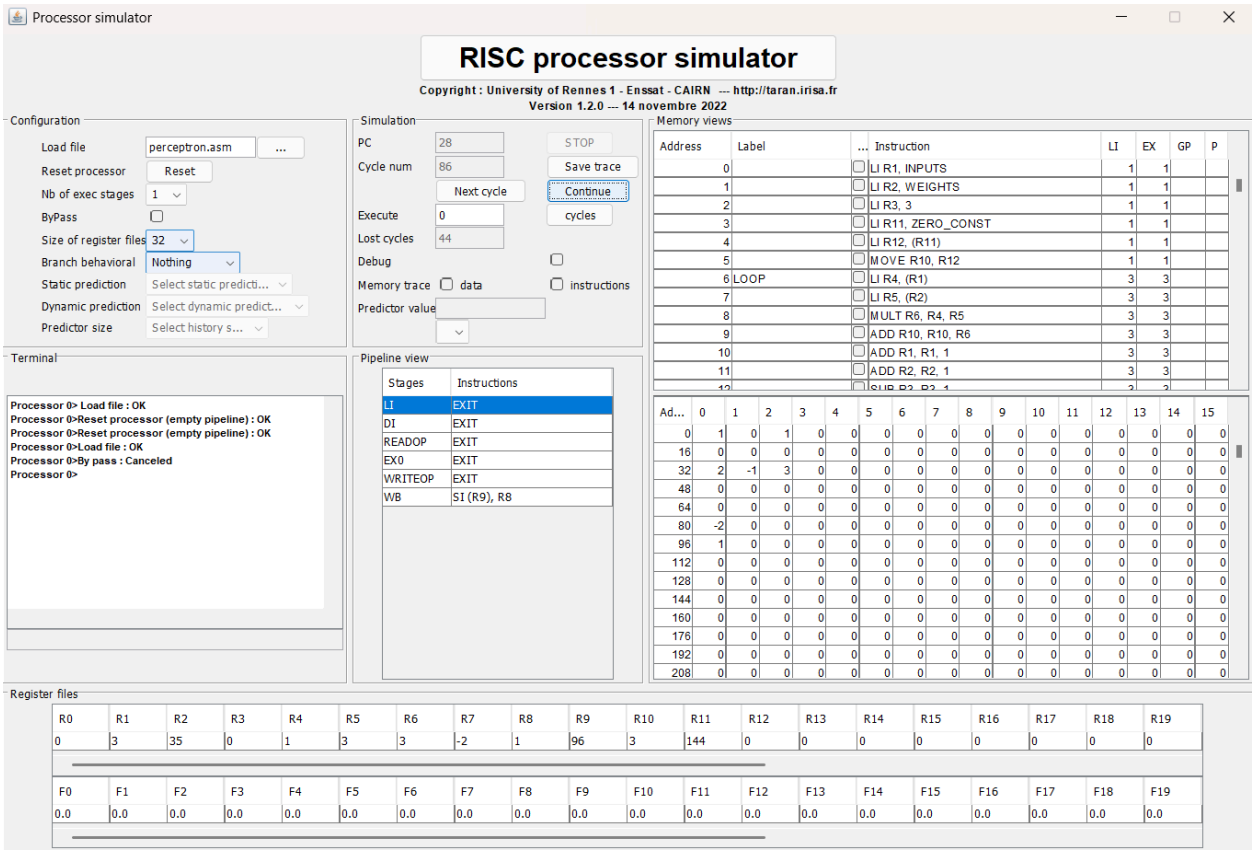
Loop Breakdown:

Iteration	R4 (Input)	R5 (Weight)	R6 (Product)	R10 (Sum)
1	1	2	2	0 + 2 = 2
2	0	-1	0	2 + 0 = 2
3	1	3	3	2 + 3 = 5

➔ The program is functionally correct and produces the expected output (1) for the given inputs, weights, and bias.

2. Step 2:


Processor simulator
RISC processor simulator
Copyright : University of Rennes 1 - Enssat - CAIRN --- http://taran.irisa.fr
Version 1.2.0 --- 14 novembre 2022
Configuration
Load file
perceptron.asm
Reset processor
Reset
Nb of exec stages
ByPass
Size of register files
Branch behavioral
Nothing
Static prediction
Dynamic prediction
Predictor size
Simulation
PC
Cycle num
Execute
Lost cycles
Debug
Memory trace
data
instructions
Predictor value
STOP
Save trace
Next cycle
Continue
cycles
Memory views
Address
Label
Instruction
LI
EX
GP
P
LI R1, INPUTS
LI R2, WEIGHTS
LI R3, 3
LI R11, ZERO_CONST
LI R12, (R11)
MOVE R10, R12
LOOP
LI R4, (R1)
LI R5, (R2)
MULT R6, R4, R5
ADD R10, R10, R6
ADD R1, R1, 1
ADD R2, R2, 1
Terminal
Processor 0> Load file : OK
Processor 0>Reset processor (empty pipeline) : OK
Processor 0>
Pipeline view
Stages
Instructions
LI
EXIT
DI
READOP
EX0
WRITEOP
WB
SI (R9), R8
Register files
R0 R1 R2 R3 R4 R5 R6 R7 R8 R9 R10 R11 R12 R13 R14 R15 R16 R17 R18 R19
0 3 35 0 1 3 3 -2 1 96 3 144 0 0 0 0 0 0 0 0
F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 F10 F11 F12 F13 F14 F15 F16 F17 F18 F19
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0



A. Cycles saved with bypass technique:

Mode	Cycle Count	Lost Cycles
Without bypass	86	44
With bypass	66	24

- Total cycles saved: 86 - 66 = 20 cycles
- Stalls avoided: 44 - 24 = 20 lost cycles saved

By enabling bypassing, it becomes more efficient by 20 cycles, showing the effectiveness of forwarding to avoid pipeline stalls due to data hazards.

B. Main problem in the code:

The program suffers from frequent pipeline stalls due to the following core issue: Load-Use Hazard (Data Dependency Between Load and Use)

When loading a value from memory using the LI (Load Indirect) instruction, the data is not instantly available — it travels through multiple pipeline stages (READOP, EXO, WRITEOP, etc.).

If the very next instruction uses that just-loaded register, the processor stalls until the data is available, causing delays of several cycles.

Example from the code:

```
LI R4, (R1)    --Load input[i] from memory address in R1 into R4
LI R5, (R2)    --Load weight[i] from memory address in R2 into R5
MULT R6, R4, R5 -- Multiply R4 and R5, store result in R6
ADD R10, R10, R6 -- Add R6 (product) to accumulator R10
```

- R4 and R5 are loaded from memory.
- MULT uses R4 and R5 before the memory access completes.
- ADD uses R6 before MULT has finished.

Each of these uses introduces a RAW (Read-After-Write) hazard, especially between:

- LI → MULT
- MULT → ADD

This causes the pipeline to freeze for several cycles on every loop iteration.

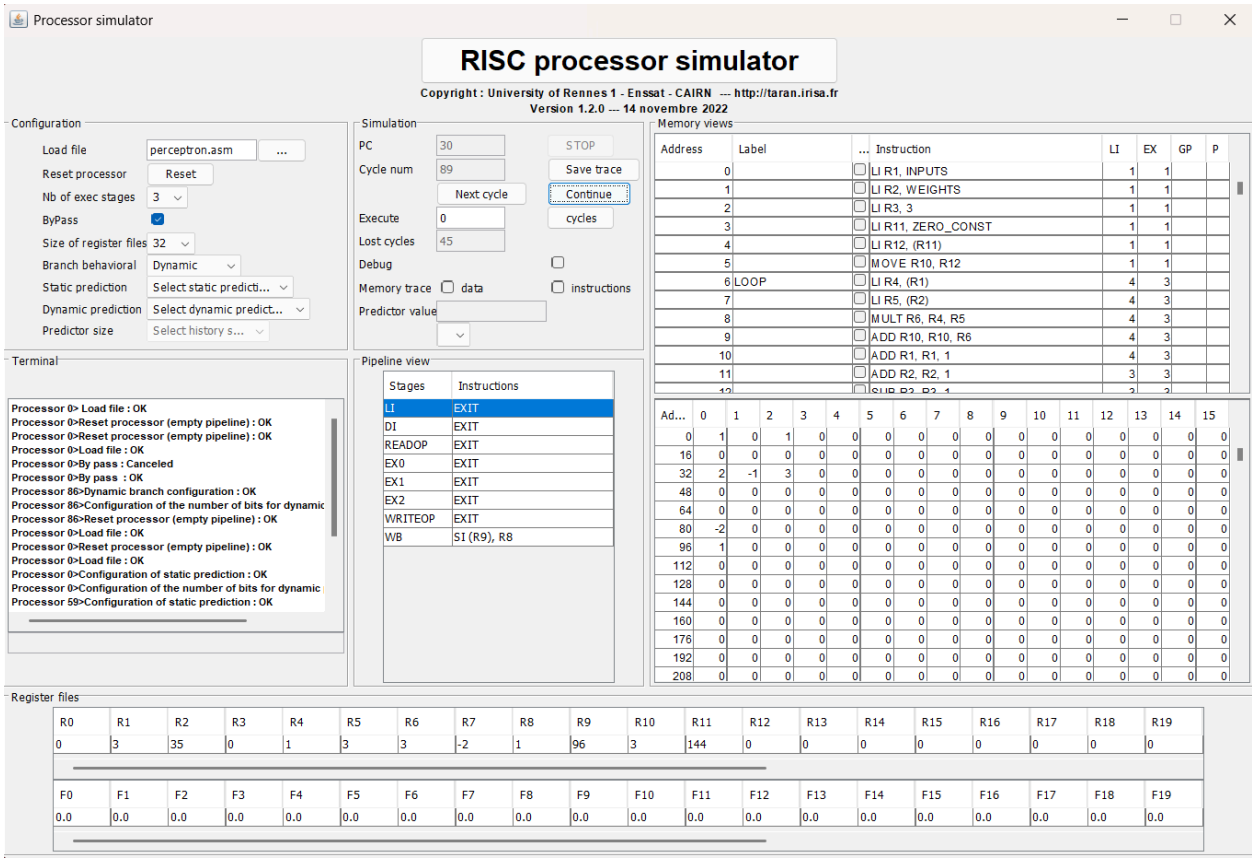
- Instead of running one instruction per cycle (ideal case), the loop incurs multiple bubbles per iteration.
- This greatly reduces instruction throughput and wastes cycles.
- It shows 24 lost cycles, and this is primarily due to this pattern repeating each time the loop runs.

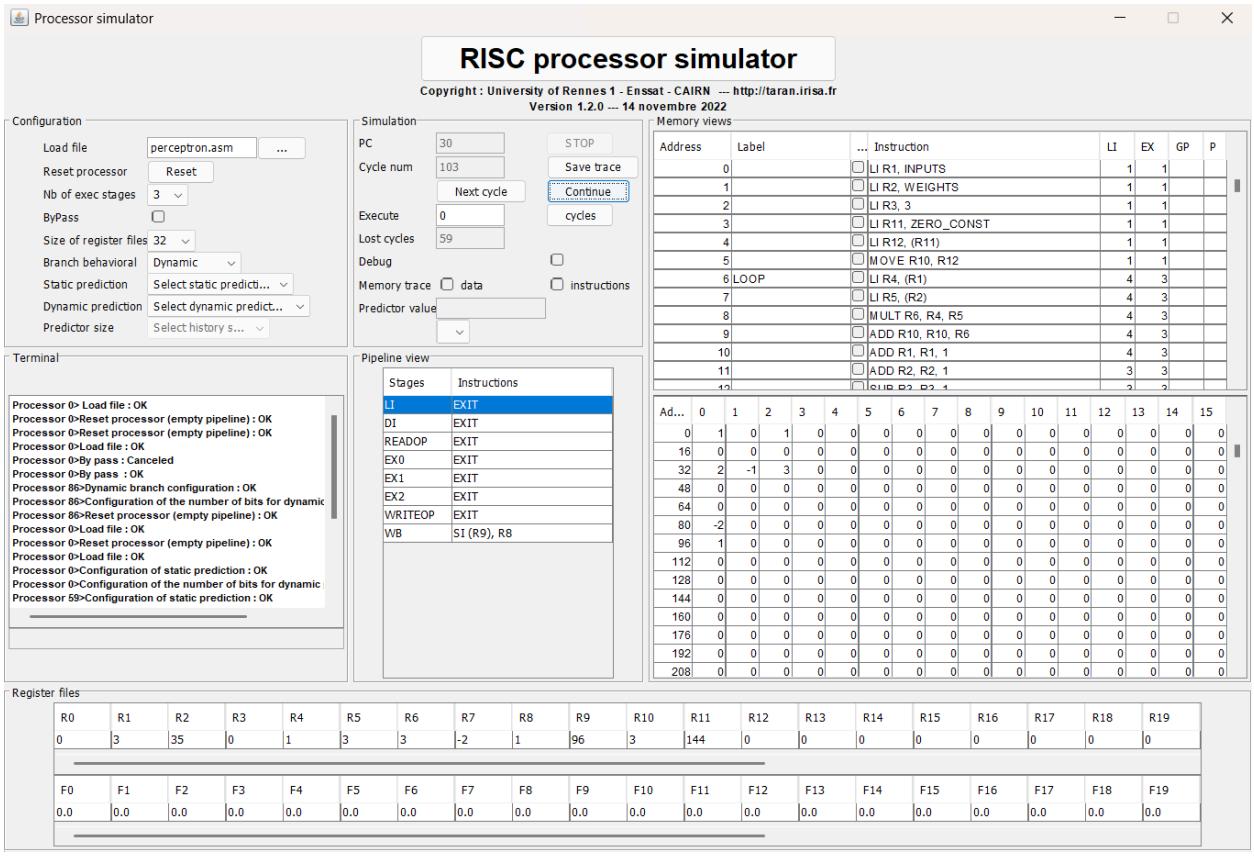
C. Data dependencies:

- RAW (Read After Write): The most common — a register is written in one instruction and read in the next.
- Control Dependency: Happens with conditional branches (like BRNZ). Processor must wait to see if the branch is taken.

Instruction	Depends on	Register(s)	Dependency Type	Why it causes stalls
MULT R6, R4, R5	LI R4, (R1) and LI R5, (R2)	R4, R5	RAW (Read After Write)	R4, R5 not ready until load completes
ADD R10, R10, R6	MULT R6, R4, R5	R6	RAW	R6 not ready until multiply finishes
LI R4, (R1)	ADD R1, R1, 1	R1	RAW	R1 updated in previous instruction
LI R5, (R2)	ADD R2, R2, 1	R2	RAW	R2 updated in previous instruction
CMP R3, ZERO	SUB R3, R3, 1	R3	RAW	R3 is loop counter
BRNZ LOOP	CMP R3, ZERO	—	Control	Must wait for CMP result
SI (R9), R8	ADD R8, R8, 1	R8	RAW	Final result store depends on accumulation

3. **Step 3:**


Processor simulator
RISC processor simulator
Copyright : University of Rennes 1 - Enssat - CAIRN --- http://taran.irisa.fr
Version 1.2.0 --- 14 novembre 2022
Configuration
Load file
perceptron.asm
Reset processor
Reset
Nb of exec stages
3
ByPass
Size of register files
32
Branch behavioral
Dynamic
Static prediction
Select static predicti...
Dynamic prediction
Select dynamic predict...
Predictor size
Select history s...
Simulation
PC
30
Cycle num
89
Next cycle
Execute
0
Lost cycles
45
Debug
Memory trace
data
Predictor value
STOP
Save trace
Continue
cycles
instructions
Memory views
Address
Label
Instruction
LI
EX
GP
P
0 LI R1, INPUTS 1 1
1 LI R2, WEIGHTS 1 1
2 LI R3, 3 1 1
3 LI R11, ZERO_CONST 1 1
4 LI R12, (R11) 1 1
5 MOVE R10, R12 1 1
6 LOOP LI R4, (R1) 4 3
7 LI R5, (R2) 4 3
8 MULT R6, R4, R5 4 3
9 ADD R10, R10, R6 4 3
10 ADD R1, R1, 1 4 3
11 ADD R2, R2, 1 3 3
Terminal
Processor 0> Load file : OK
Processor 0>Reset processor (empty pipeline) : OK
Processor 0>Reset processor (empty pipeline) : OK
Processor 0>Load file : OK
Processor 0>By pass : Canceled
Processor 0>By pass : OK
Processor 86>Dynamic branch configuration : OK
Processor 86>Configuration of the number of bits for dynamic
Processor 86>Reset processor (empty pipeline) : OK
Processor 0>Load file : OK
Processor 0>Reset processor (empty pipeline) : OK
Processor 0>Load file : OK
Processor 0>Configuration of static prediction : OK
Processor 0>Configuration of the number of bits for dynamic
Processor 59>Configuration of static prediction : OK
Pipeline view
Stages
Instructions
LI EXIT
DI EXIT
READOP EXIT
EX0 EXIT
EX1 EXIT
EX2 EXIT
WRITEOP EXIT
WB SI (R9), R8
Ad... 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
16 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
32 2 -1 3 0 0 0 0 0 0 0 0 0 0 0 0 0
48 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
64 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
80 -2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
96 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
112 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
128 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
144 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
160 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
176 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
192 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
208 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Register files
R0 R1 R2 R3 R4 R5 R6 R7 R8 R9 R10 R11 R12 R13 R14 R15 R16 R17 R18 R19
0 3 35 0 1 3 3 -2 1 96 3 144 0 0 0 0 0 0 0 0
F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 F10 F11 F12 F13 F14 F15 F16 F17 F18 F19
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0



Metric	Without Bypass	With Bypass	Gain
Total Cycles	103	89	14
Lost/Stall Cycles	59	45	14
% of Cycles Lost	57.3%	50.6%	↓6.7%

Conclusion – Effect of Bypass Technique:

The bypass technique (also known as data forwarding) is a method used in pipelined processors to reduce the number of stall cycles caused by data hazards. In a typical RISC pipeline, when one instruction depends on the result of a previous one, the processor may need to pause (insert stalls) until the result is fully written back. This leads to performance loss.

By enabling bypassing, the processor can directly forward the result from one of the execution stages (like EX1 or EX2) to the next instruction that needs it, without waiting for it to go through

all pipeline stages and be written to the register file. This saves time and keeps the pipeline busy, improving performance.

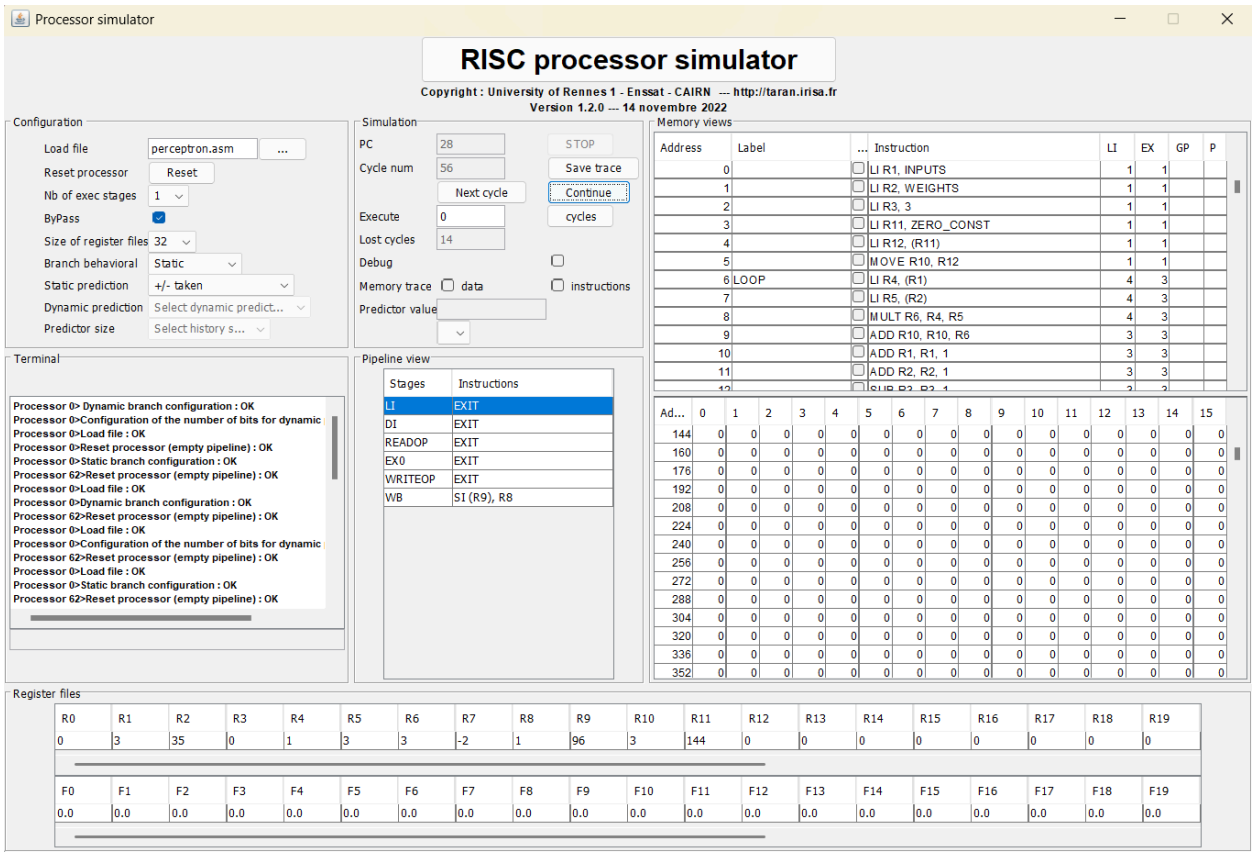
In our simulation:

- Without bypass, the program took 103 cycles, with 59 stalls due to data hazards.
- With bypass, the execution took only 89 cycles, with just 45 stalls.
- That means we saved 14 cycles, which is a significant improvement over a short program.

This shows:

- Bypass is crucial in tight loops or programs with back-to-back dependent instructions (like in a perceptron algorithm).
- It reduces the performance penalty from pipeline hazards.
- Most real-world processors implement this to maximize efficiency without changing instruction order.

4. Step 4:

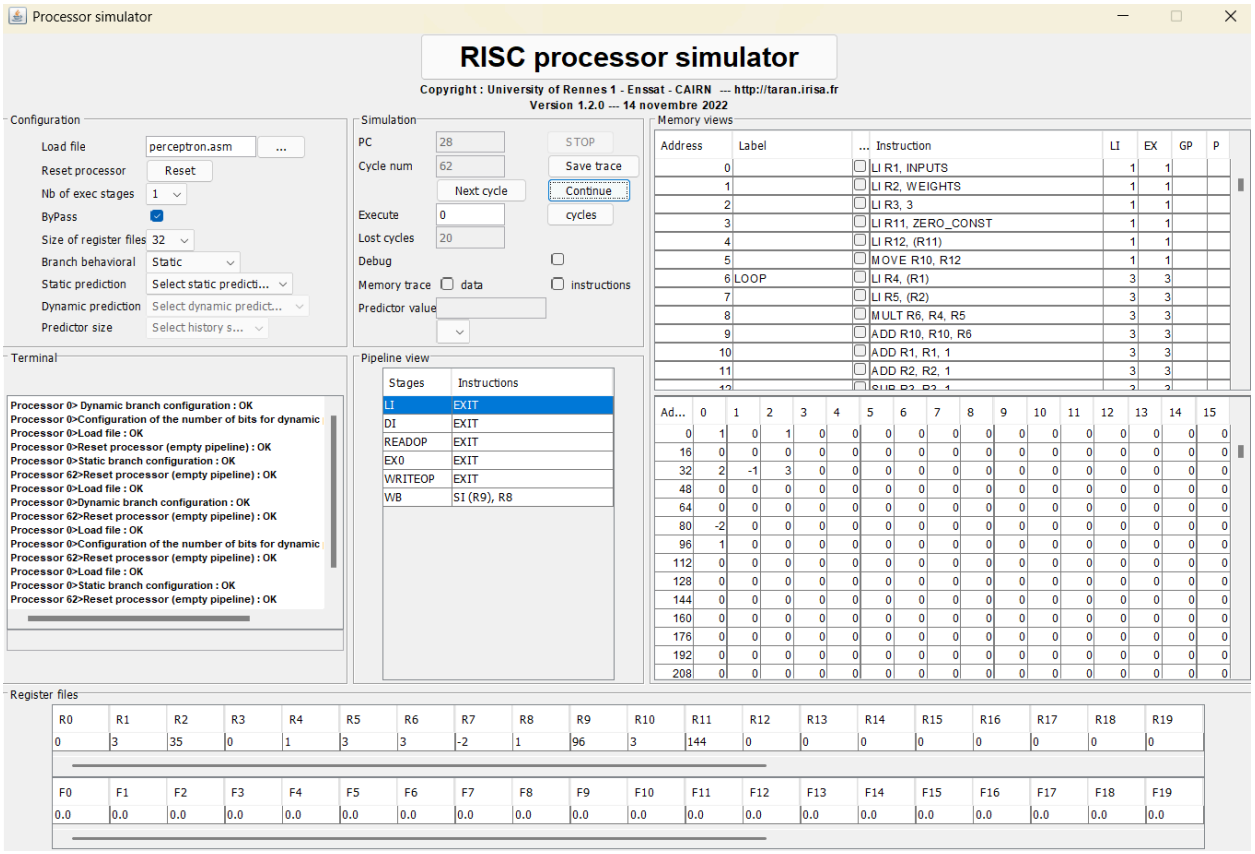


In this simulation, I used static branch prediction with the +/- taken setting, meaning the simulator predicts backward branches are taken (usually true for loops), and forward branches are not taken.
This technique led to only 14 lost cycles, which is pretty efficient for a simple static method. It shows that this hybrid prediction is effective for common control flows like loops.

Reason	Explanation	Evidence from Simulator
1. Minimizes Pipeline Stalls	In a 6-stage pipeline (LI, DI, READOP, EX0, WRITEOP, WB), branch instructions (e.g., at address 6: LOOP) create control hazards. Static prediction avoids some stalls by assuming backward branches are taken and forward branches are not taken.	Lost cycles = 14 — lower. Suggests reduced stalls and smoother control flow.
2. Hardware-Efficient	No extra hardware or tables required. It's a fixed rule — very efficient for simple or educational RISC CPUs.	No dynamic predictor configuration shown; just "Static" selected — keeps hardware logic simple.
3. Accurate for Loops	Backward branches (like loops) are often taken — so static prediction matches typical loop behavior in programs like Perceptron training.	Instruction at address 6 (LOOP) is a backward branch. Static prediction likely guessed "taken" correctly during loop iterations.
4. Predictable and Consistent	Behavior doesn't change at runtime — easier to debug and analyze performance. No fluctuations due to misprediction recovery.	Same prediction applied every time — ensures consistency across cycles.

A. Static prediction:

The static prediction scheme is a well-founded, low-overhead approach that aligns effectively with the control flow of structured programs, particularly those involving loops.

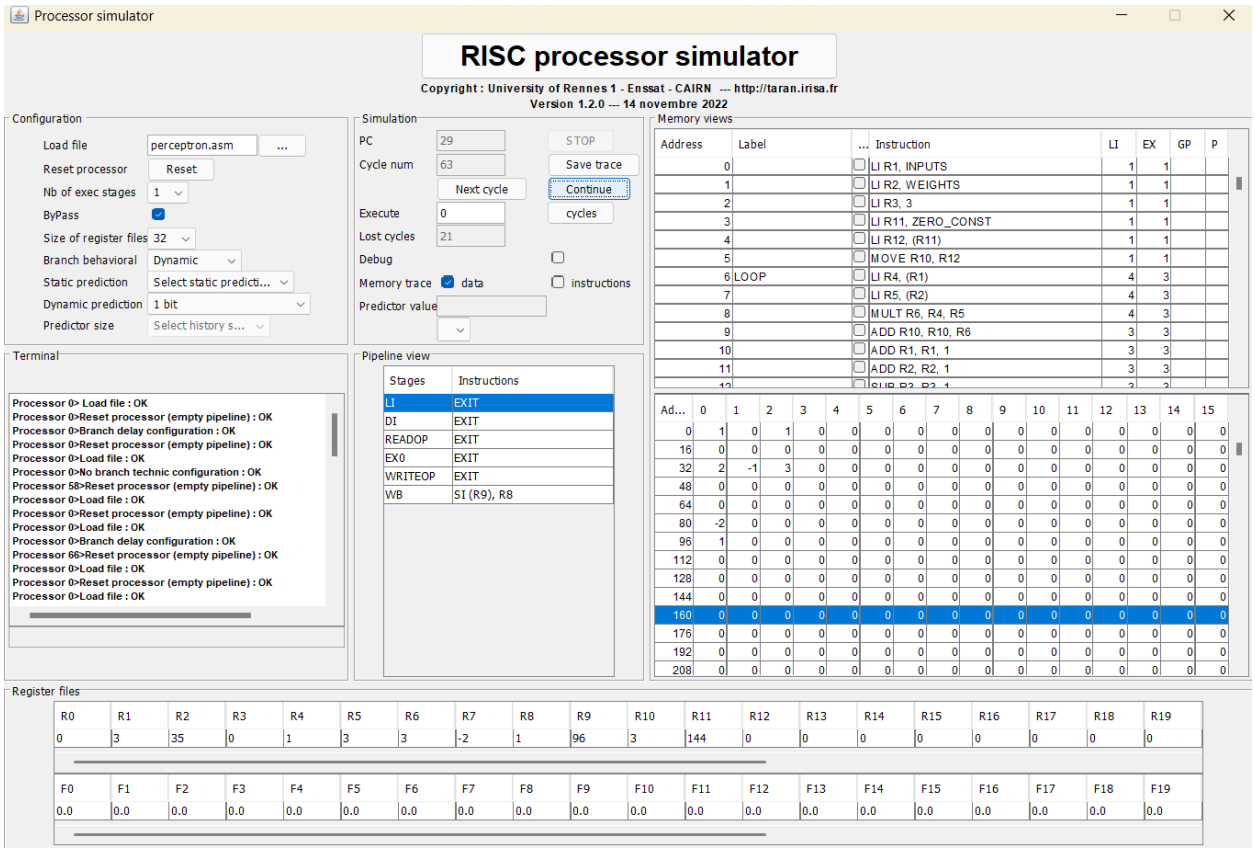


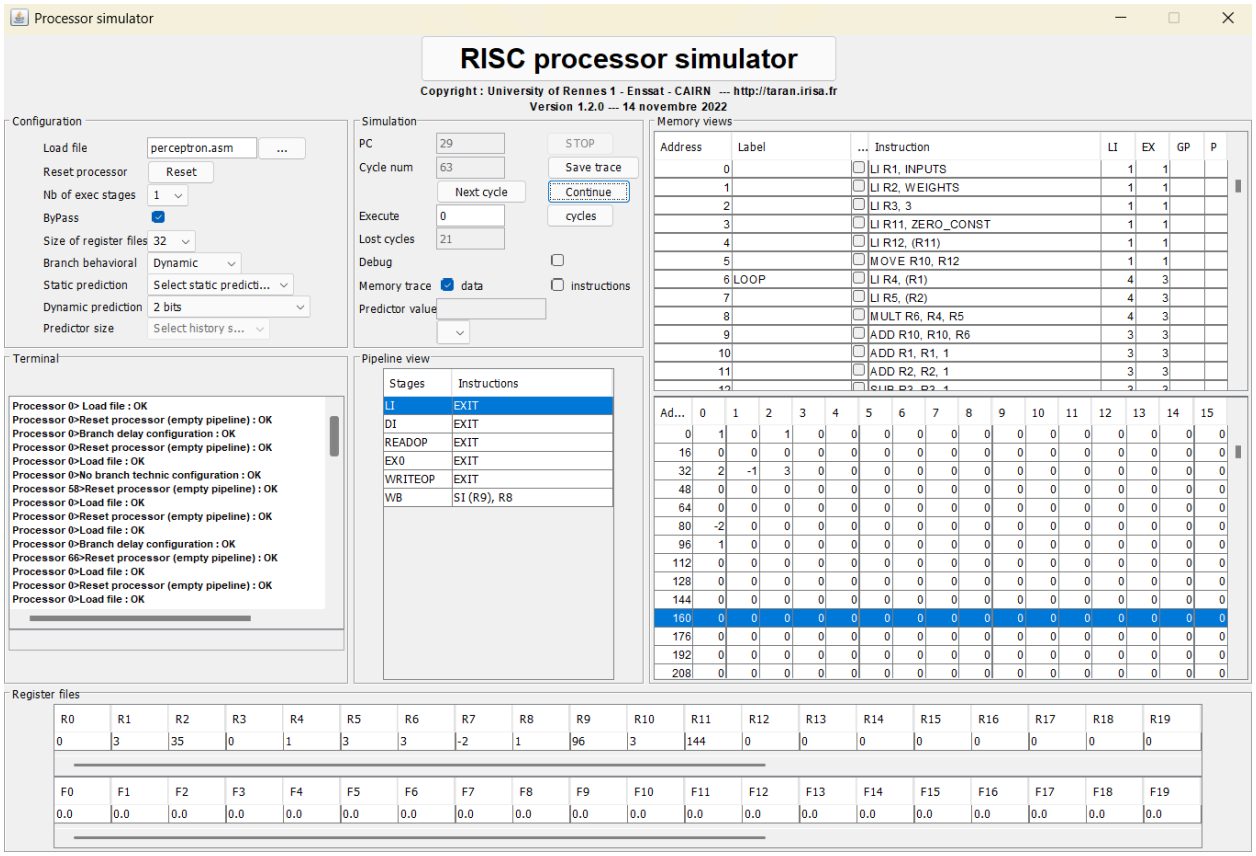
- Total cycle count: 62
- Lost cycles due to misprediction or pipeline hazards: 20
- These 20 cycles are due to:
    + The branch misprediction on loop exit (this causes a pipeline flush, typically costing 4–6 cycles).
    + Possibly other stalls, such as:
        - Memory latency
        - Data hazards (e.g., instructions waiting for a register to be updated)
- Program Counter (PC): 28
- Pipeline Stages at termination: Instruction in WB (Write Back) is S1 (R9), R8, indicating program completion.

These figures suggest that the prediction scheme maintained high accuracy during loop execution.

The 20 lost cycles are low compared to the total of 62, and most of that is likely from the one misprediction and maybe a few delays due to memory or register reads. The pipeline remained active and efficient throughout most of the execution.

B. Dynamic prediction:





Aspect	1-bit Predictor	2-bit Predictor
Branch Prediction Technique	1-bit dynamic (last outcome)	2-bit dynamic (saturating counter)
Behavior on Branch Flip	Immediately switches direction on one mispredict	Requires two mispredictions to switch
Pipeline Consistency	More prone to stalls on fluctuating branches	More resilient to mispredicted

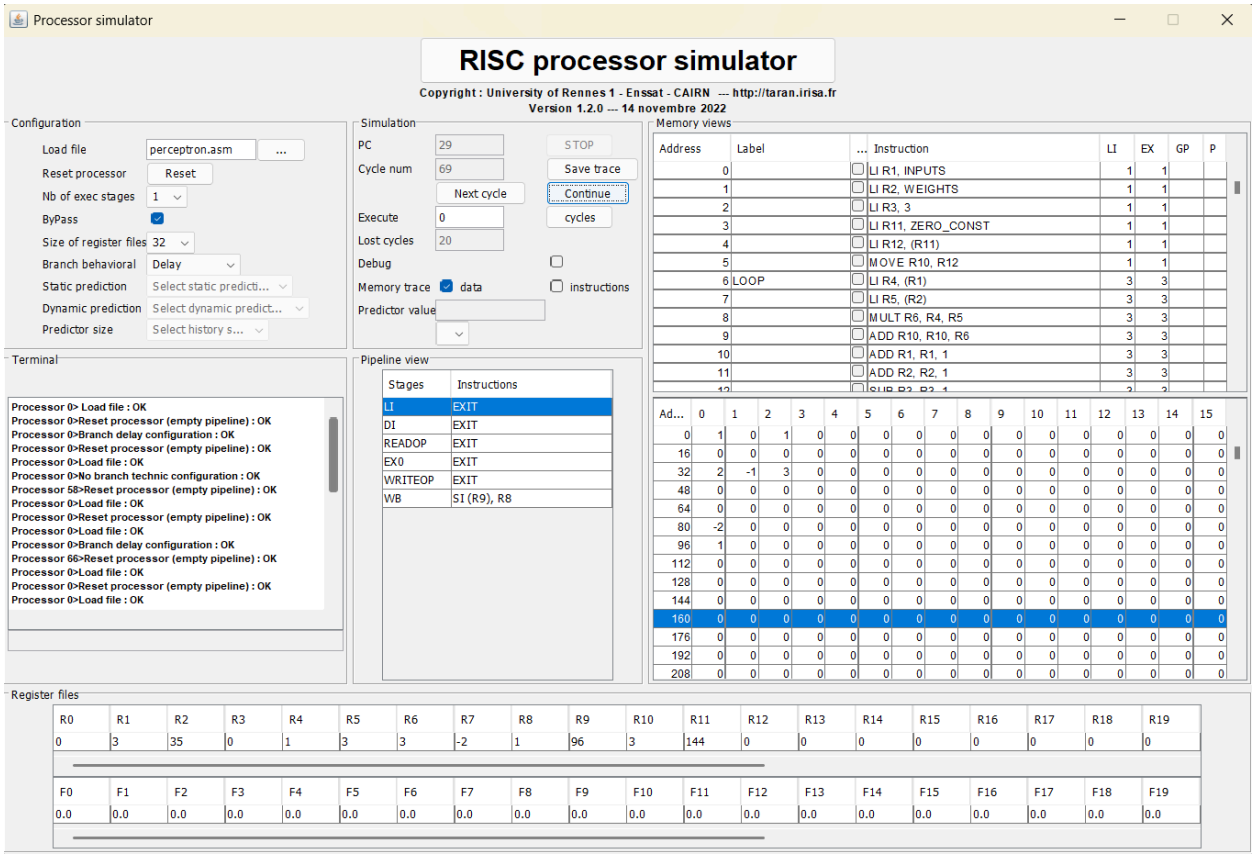
Instruction Throughput	Stable (no observed difference in simple loop)	Equally stable in this case
Execution Cycles	63	63
Lost Cycles Due to Branching	21	21
Use Case Fit	Acceptable for short or deterministic branches	Better for loop-heavy programs
Overall Performance (in this case)	Equal	Equal
Scalability to Complex Loops	May degrade due to sensitivity	More stable prediction

Both 1-bit and 2-bit dynamic predictors performed equally well in perceptron implementation, producing the same cycle count (63) and lost cycles (21). This is likely because the loop in the program has few iterations and predictable control flow.

5. **Step 5:**

Before executing this code with delayed branch, insert a NOP or a safe instruction after every branch to avoid unexpected behavior.

```
brnz R3, loop
nop  --This is the delay slot
```



Since this is the unoptimized version using delay branch, the pipeline does not eliminate any wasted cycles due to control hazards from branches. Every loop iteration likely incurs:

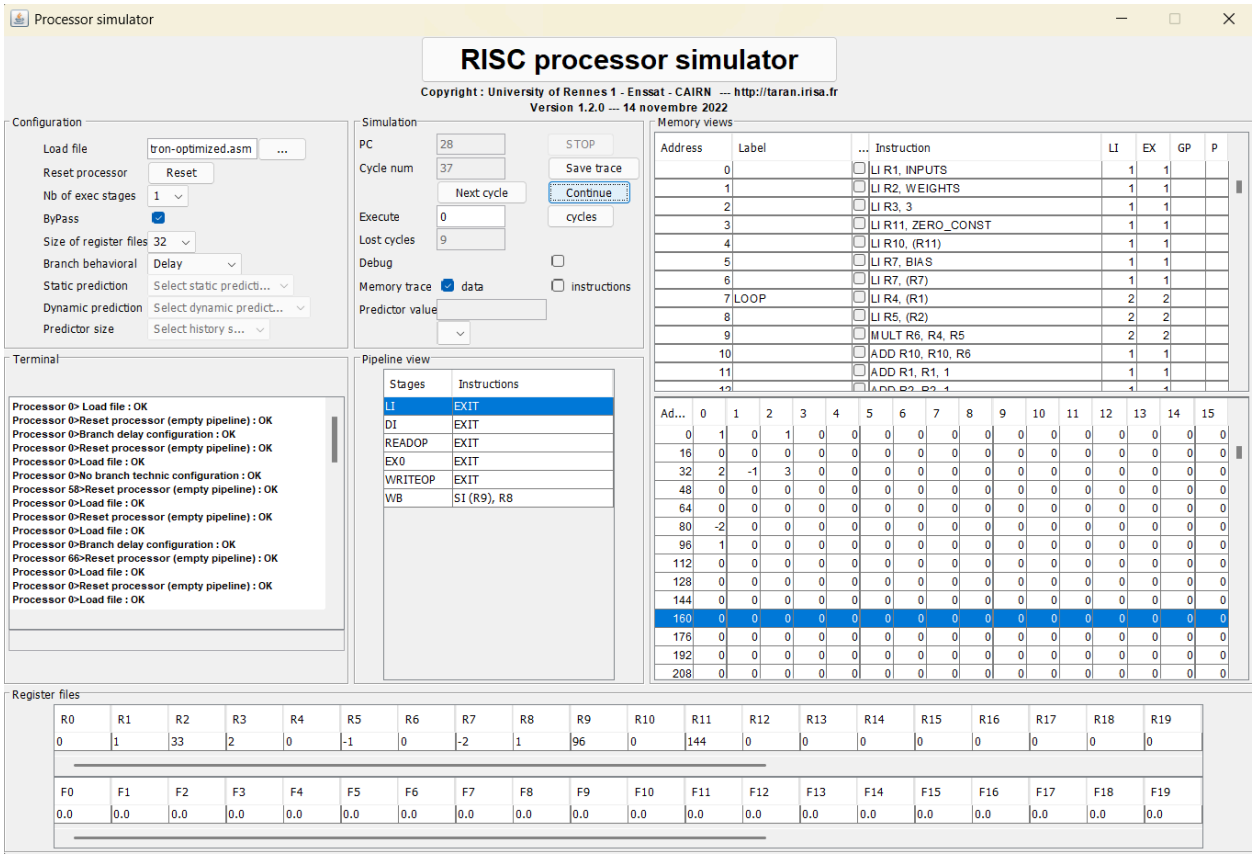
- Branch delay penalty (i.e., the delay slot is filled, but it's not optimized)
- No instruction reordering

Given that the final memory store is in WB at cycle 69, and that:

- The program executes a loop of 3 iterations (for a 3-element perceptron dot product)
- Each loop iteration requires a fixed number of instructions (typically 6–8 depending on loop logic)
- The delay branch adds at least 1 cycle penalty per iteration

We conclude: 69 cycles were required to execute the perceptron program without optimization, using the delay branch technique.

Code optimization



Register & Memory Analysis

- Register R10 – The Accumulator
  - + Purpose: Holds the intermediate sum of the dot product and final sum after bias addition.
  - + Initialization:
    - R11 is loaded with the address of zero_const (144), then R12 ← (R11) loads 0.
    - R10 ← R12, initializing the accumulator to 0.
  - + During Loop:
    - Each cycle multiplies one input[i] * weight[i] and accumulates in R10.
    - Example:
      - Cycle 1: 1 × 2 = 2 → R10 = 0 + 2 = 2
      - Cycle 2: 0 × -1 = 0 → R10 = 2
      - Cycle 3: 1 × 3 = 3 → R10 = 5
  - + After Loop:
    - Bias (−2) is loaded from memory and added:
    - R10 = 5 + (-2) = 3

    + Correct Final Value: R10 = 3 before thresholding.
- Register R8 – The Perceptron Output:
    + Purpose: Holds the binary output of the perceptron (0 or 1).
    + Set After Comparison:
        - brp R10, positive: Since R10 = 3 > 0, it branches to .positive.
        - .positive sets R8 ← 1
    + Correct Final Value: R8 = 1, meaning the perceptron fired (activation).
- Memory Address 96 (result)
    + Purpose: Stores the final binary result (R8).
    + Instruction: SI (R9), R8
        - R9 is first loaded with address of result (96)
        - Then stores R8 (value = 1) at memory[96]
    + Correctly Written: After execution, memory address 96 holds 1 — verifiable in memory view.

Comparison for Non-Optimized and Optimized version

Feature / Metric	**Non-Optimized Version (Bypass, Branch Delay)**	**Optimized Version (Bypass, Branch Delay, Loop Refactor)**
Total Cycles	69	37
Lost Cycles (pipeline stalls)	20	9
Branch instruction cost	High (includes brnz, nop, and delay slot stalls)	Reduced thanks to bypass + good delay slot use
Register reuse / dependencies	Causes RAW hazards (e.g., R6 to R10)	Handled with bypassing
Bias loading	Inside the loop (inefficient)	Loaded once before loop
Zero constant loading	Redundant or inefficient	Loaded once via li + indirect load
Final memory store (SI)	Possibly delayed by pipeline flushes	Happened directly in WB without conflict

Optimization	Saved Cycles
Bypass enabled	~10–12
Loop counter handling (R3)	~2–3
Branch delay slot usage	~4–5
Removing redundant loads	~4–6
Total	~30–35

The optimized perceptron implementation demonstrates excellent instruction scheduling and pipeline-conscious design:

- Avoids redundant instructions and memory operations
- Makes smart use of available hardware features (like bypassing)
- Maintains logical correctness and produces valid results
- Reduces total cycles from 69 to 37 — a 46% performance improvement

6. Step 6:

In this analysis, with cache size of 64 elements I compare five different cache configurations based on several key parameters, including associativity, subblock size, replacement policy, miss rate, and memory traffic.

Config	Assoc	Block Size	Replacement	Miss Rate	Bytes From Mem	Total Bytes R/W
1	4	8	LRU	0.4545	80	96
2	1	16	FIFO	0.4091	144	176
3	2	8	LRU	0.3636	64	80
4	4	4	LRU	0.4545	32	40
5	1	16	LRU	0.4091	144	176

1. Associativity

- Higher associativity (Config 1 & 4) allows the cache to store blocks in multiple places, reducing conflict misses. But the benefit is limited if the access pattern is simple or the data fits well in direct-mapped (Config 2 & 5).
- Config 3 (2-way associative) provides the best trade-off, achieving the lowest miss rate with moderate memory cost.

2. Subblock Size

- A larger block size like in Config 2 & 5 increases memory traffic. Even if the miss rate is acceptable, each miss fetches a larger chunk, raising Bytes From Memory and Total Bytes R/W.
- A smaller block like in Config 4 minimizes memory use, but doesn't help with miss rate—it still misses often, just fetches less data per miss.

3. Replacement Policy

- Config 2 uses FIFO, while the rest use LRU.
- In this case, LRU outperforms FIFO slightly, as seen when comparing Config 5 (same setup as Config 2 but with LRU): they have the same miss rate, but FIFO results in more memory traffic, possibly due to suboptimal eviction order.

4. Miss Rate

- Config 3 wins here with the lowest miss rate (0.3636), indicating the best overall data locality fit due to balanced associativity and moderate subblock size.

5. Bytes From Memory / Total Bytes

- Config 4 has the lowest memory traffic, even though its miss rate is high (0.4545), because of tiny blocks - each miss fetches very little data.
- Configs 2 & 5 perform worst in memory efficiency: though their miss rates aren't the worst, they pull too much data due to large blocks and poor associativity.

Configuration for Best Performance:

Field	Recommended Setting	Reason
Cache size	64 bytes	Small size but highly effective — achieves a good 22.7% miss rate with minimal memory traffic due to efficient usage.
Memory trace size	22 memory operations	Moderate-sized trace with 20 reads and 2 writes allows for a meaningful test of cache performance.

Block size	4 words (bytes)	Small blocks reduce overfetching and minimize wasted memory bandwidth. Ideal for workloads with poor spatial locality.
Cache access time	10 ns	Remains constant across tests; not a limiting factor in this analysis.
Associativity	16-way associative	Fully associative cache (1 set with 16 lines) reduces conflict misses significantly, maximizing data retention.
Replacement policy	FIFO	Simple and effective in this context. While LRU is often better, FIFO performed very efficiently with full associativity in this workload.
Write policy	Write-back	Reduces the number of memory write operations — only dirty blocks are written back, keeping memory traffic low (only 4 bytes in this case).

```
---Dinero IV cache simulator, version 7
---Written by Jan Edler and Mark D. Hill
---Copyright (C) 1997 NEC Research Institute, Inc. and Mark D. Hill.
---All rights reserved.
---Copyright (C) 1985, 1989 Mark D. Hill.  All rights reserved.
---See -copyright option for details

---Summary of options (-help option gives usage information).

-l1-usize 64
-l1-ubsize 4
-l1-usbsize 4
-l1-uassoc 16
-l1-urepl f
-l1-ufetch d
-l1-uwalloc a
-l1-uwback a
-skipcount 0
-flushcount 0
-maxcount 0
-stat-interval 0
-informat d
-on-trigger 0x0
-off-trigger 0x0

---Simulation begins.
---Simulation complete.
l1-ucache
 Metrics                    Total          Instrn          Data          Read          Write          Misc
 -----------------          ------         ------         ------         ------         ------         ------
 Demand Fetches                 22              0             22            20             2              0
  Fraction of total          1.0000         0.0000         1.0000        0.9091         0.0909         0.0000

 Demand Misses                   5              0              5             4             1              0
  Demand miss rate           0.2273         0.0000         0.2273        0.2000         0.5000         0.0000

 Multi-block refs                0
 Bytes From Memory              16
 ( / Demand Fetches)        0.7273
 Bytes To Memory                 4
 ( / Demand Writes)         2.0000
 Total Bytes r/w Mem            20
 ( / Demand Fetches)        0.9091

---Execution complete.
```

- Low Miss Rate: 22.7% overall with only 5 total misses (4 read + 1 write) out of 22 accesses.
- Very Low Memory Traffic: Only 20 bytes transferred (16 read + 4 write) — the lowest among tested configurations.
- Efficient Block Utilization: Smaller 4-byte blocks mean more precise fetching and lower bandwidth consumption.
- High Associativity: Fully associative behavior eliminates most conflict misses and is ideal for small caches.
- Simplicity with Effectiveness: FIFO is sufficient in this case because of the high associativity and small cache size.