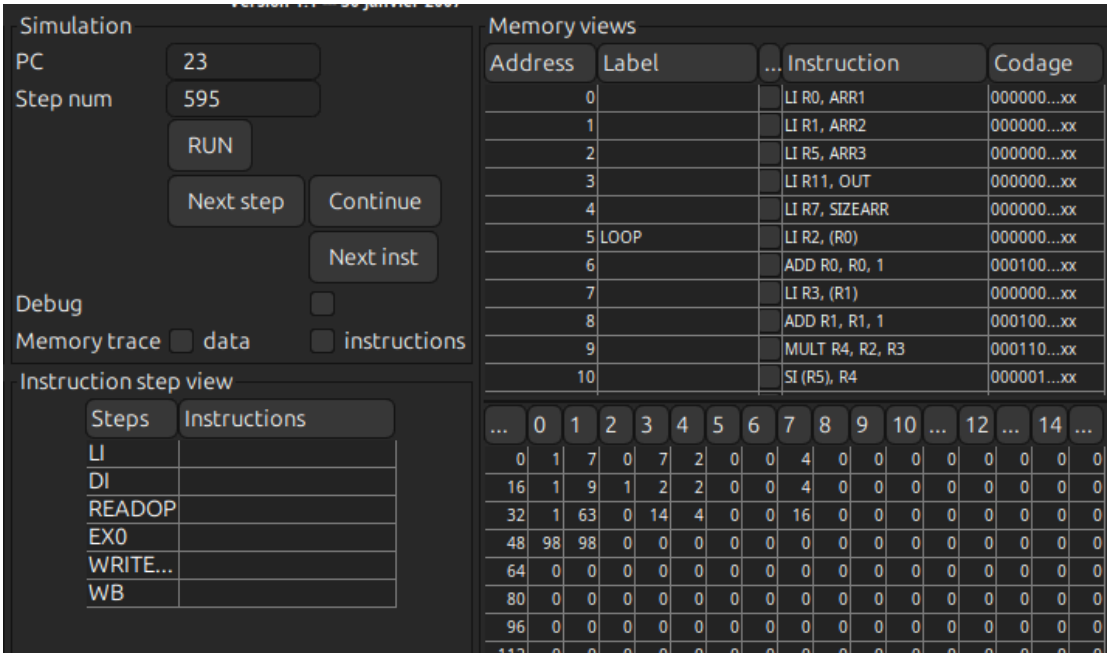
Explanation of my code:

- Implementation



- Input and Output mechanism

```
.data 0
.global ARR1
1 7 0 7 2 0 0 4          -- Input

.data 16
.global ARR2
1 9 1 2 2 0 0 4          -- Weight

.data 32
.global ARR3
0 0 0 0 0 0 0 0

.data 48
.global OUT              -- Output (out1 / out2)
0 0
```

  - Inputs: 1 7 0 7 2 0 0 4 start from address 0

- Weights: 1 9 1 2 2 0 0 4 start from address 16
- Outputs of the weight sum will be stored start from address 32
- Outputs out1 and out2 will be calculated and stored in address 48 and 49
- Each element of the input array will be multiplied to each element in the weights array respectively. After that, all the results will be added together to get the result out1. Finally, check if out1 is negative or bigger/equal to 0 to get the result out2 (if out1 >= 0 → out2 = out1, if out1 < 0 → out2 = 0).

input →

weight →

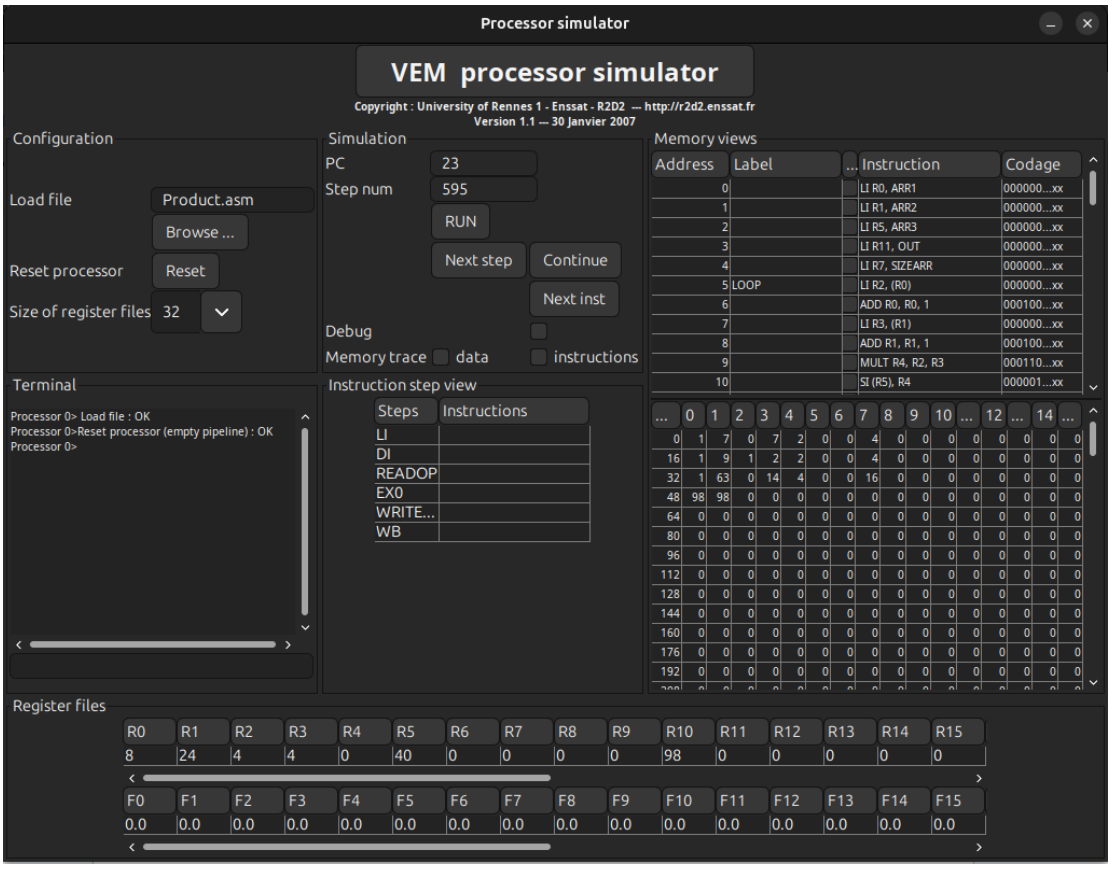
input * weight

out1 / out 2

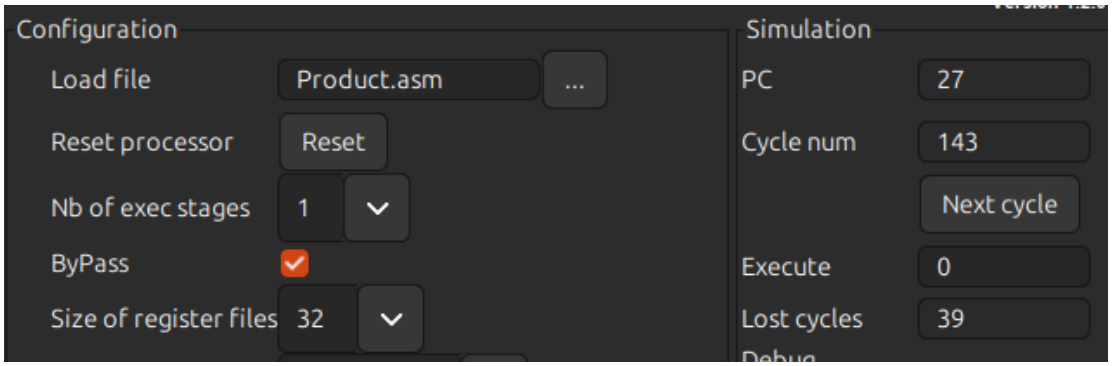
...	0	1	2	3	4	5	6	7
0	1	7	0	7	2	0	0	4
16	1	9	1	2	2	0	0	4
32	1	63	0	14	4	0	0	16
48	98	98	0	0	0	0	0	0

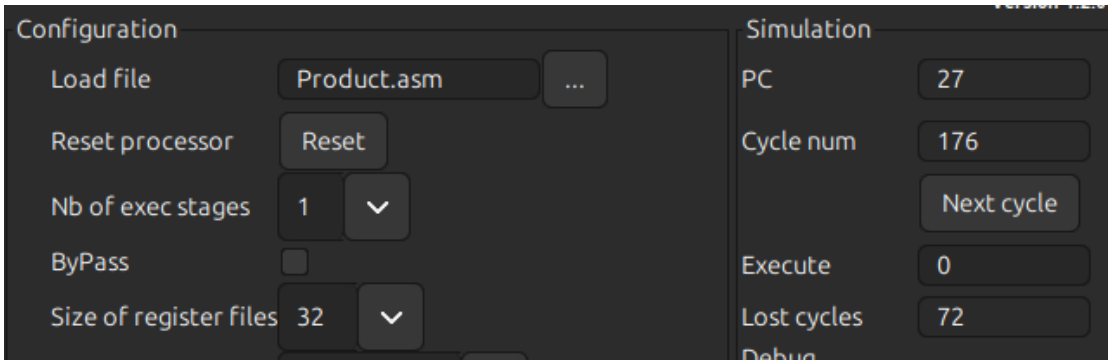
Todo work:

- Step 1:



    - Size of register files: 32
    - PC: 23
    - Step num: 595
- Step 2:





    - Bypass enabled
        - Size of register files: 32
        - Number of execution stages: 1
        - PC: 32
        - Cycle numbers: 143
        - Lost cycles: 39
    - Bypass disabled
        - Size of register files: 32
        - Number of execution stages: 1
        - PC: 32
        - Cycle numbers: 176
        - Lost cycles: 72

      → There are 33 cycles are saved with bypass technique

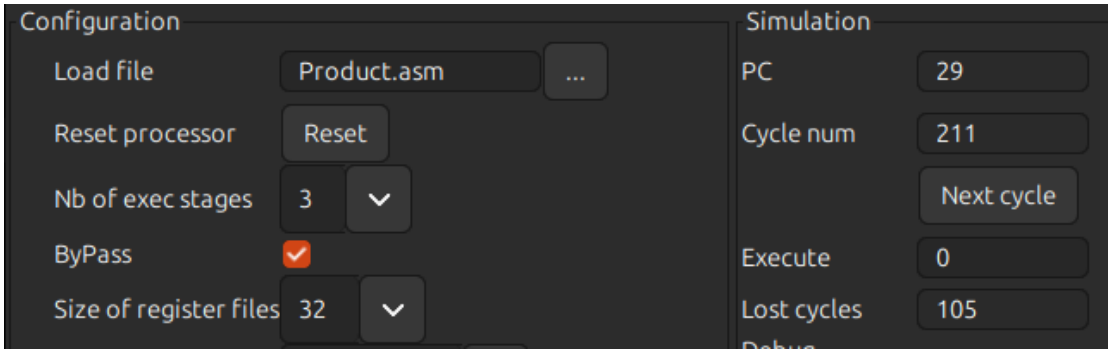
    - Analyze the main problem of the code
        - The problem of the code might come from data dependencies where they will bring data hazard to the code and cause stalls.

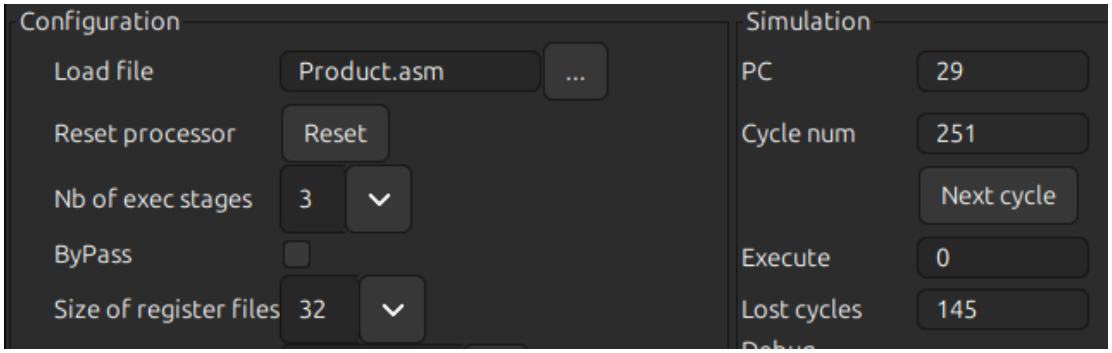
- Data dependencies

```
mult R4, R2, R3
si (R5), R4
```

R4 = R2 + R3
The result of R4 after that will be use to store in the address R5

- Step 3
  In order to get 8 pipeline stages, I will set the Number of execution stages to 3





  - Bypass technique enabled
    - PC: 29
    - Number of execution stages: 3
    - Cycle number: 211
    - Lost cycles: 105
  - Bypass technique disabled
    - PC: 29
    - Number of execution stages: 3
    - Cycle number: 251
    - Lost cycles; 145

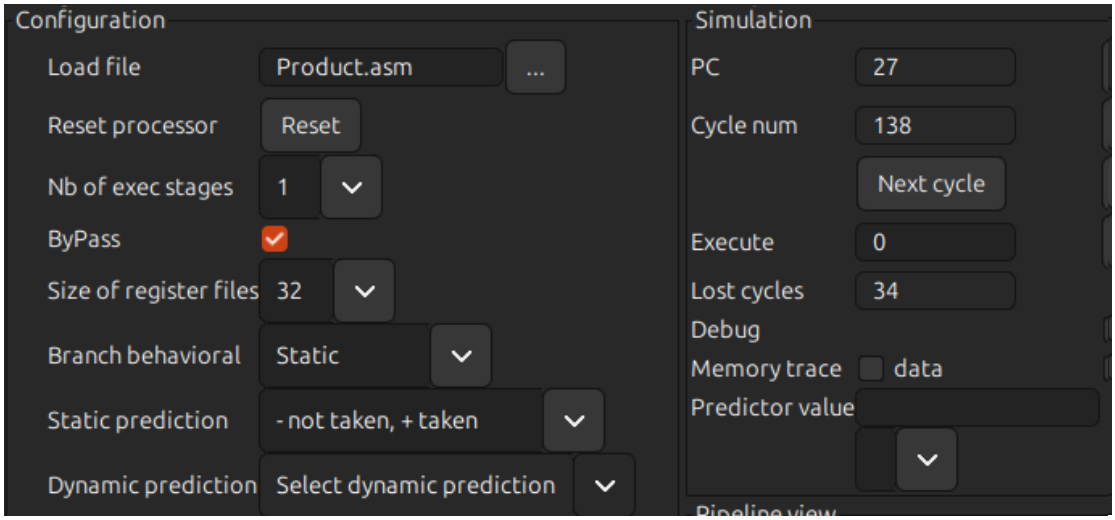
    → There are 40 cycles are save with bypass technique enabled
  - Conclusion about ByPass technique
    - Bypassing, also known as "data forwarding," is a technique used in modern CPU architectures, including x86, to improve the efficiency of pipelined processors. It helps to reduce the delay caused by data

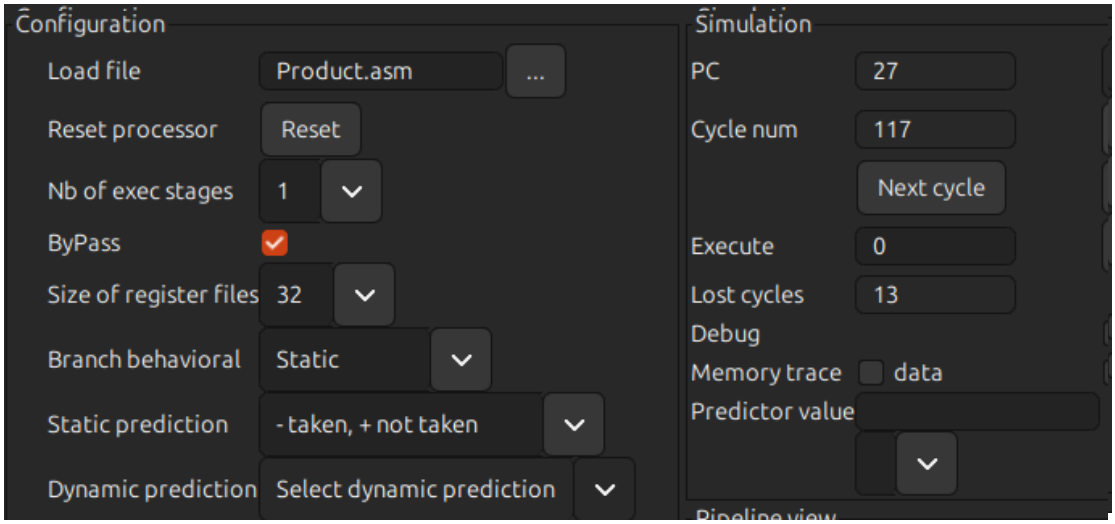
hazards, specifically read-after-write (RAW) hazards, in the instruction pipeline.

        - Bypassing addresses this issue by allowing an instruction to use the result of a previous instruction as soon as it is computed, without waiting for it to be written back to the register file. This is achieved by forwarding the result directly from one stage of the pipeline to another. Typically, the result is forwarded from the output of the execution stage (EX) to the input of the execution stage of the dependent instruction.
        - As can be seen from step2a and step3a above, with ByPass technique enabled, the number of cycles and lost cycles are significantly less than when ByPass is disabled.
- Step 4
    - Technique for branch instruction is interesting is
        - Branch Prediction
            - Modern processors use branch prediction to guess the outcome of a branch instruction before it is known for certain. Accurate branch prediction can significantly reduce the performance penalty associated with branch instructions.
            - Static Branch Prediction: This technique involves a fixed strategy, such as always predicting that forward branches will not be taken and backward branches will be taken.
            - Dynamic Branch Prediction: This involves using runtime information to make more accurate predictions. Techniques like two-level adaptive prediction and branch history tables fall under this category.
    - Other techniques for branch instruction are
        - Branch Delay Slots
            - Some architectures, like MIPS, use branch delay slots where the instruction immediately following a branch is always executed, regardless of whether the branch is taken. This technique can improve performance by ensuring the pipeline is kept busy.
        - Branch Target Buffer (BTB)
            - A Branch Target Buffer is a specialized cache that stores the destination addresses of previously executed branch instructions. When a branch instruction is encountered, the BTB is checked to quickly determine the target address,

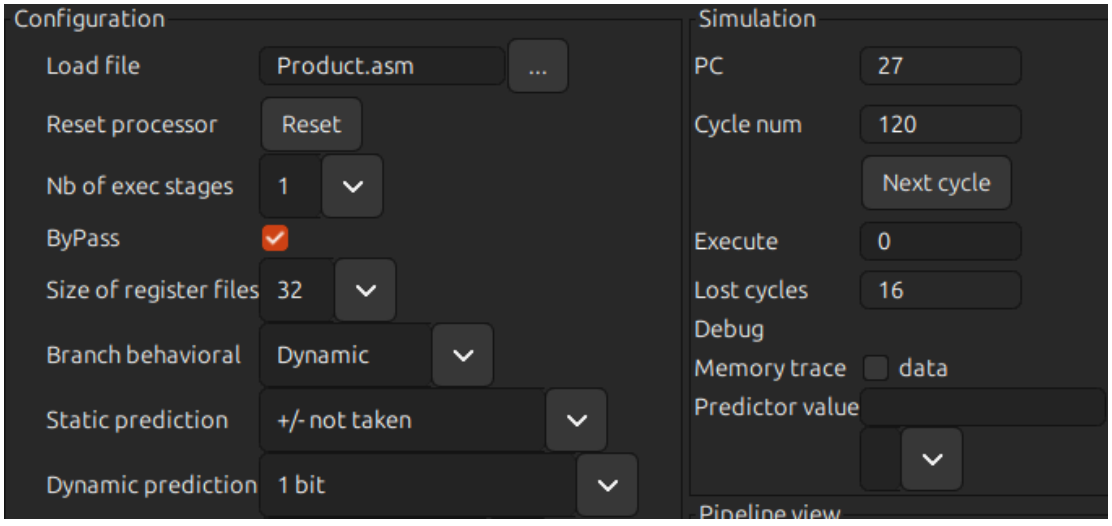
allowing the instruction pipeline to fetch the next instruction without stalling.

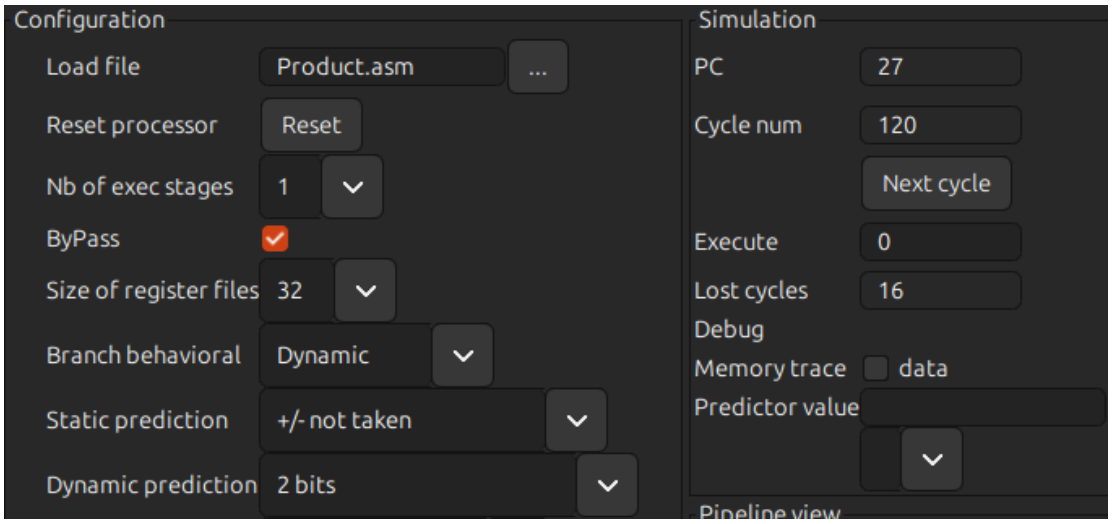
- Test static and dynamic prediction
  - Static

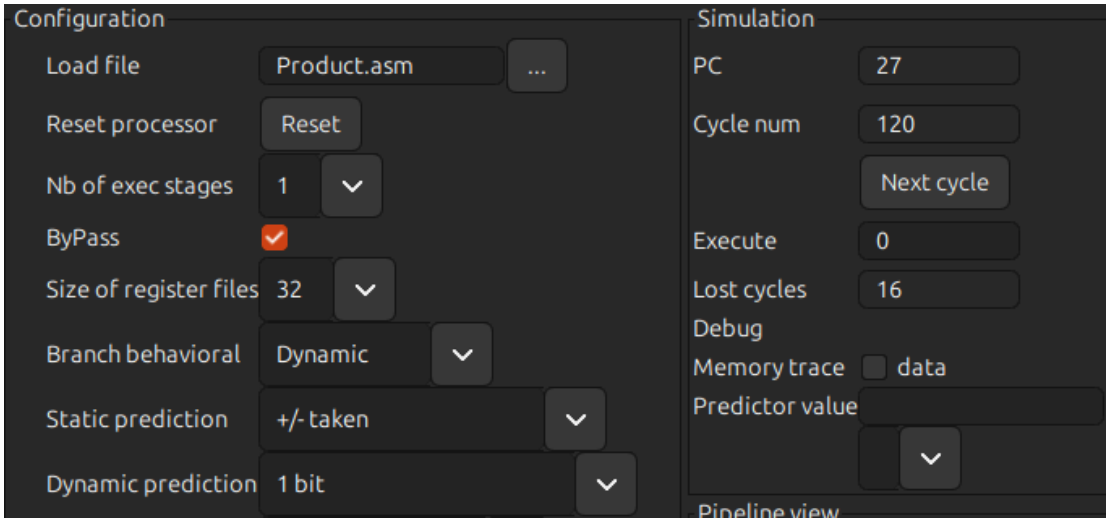


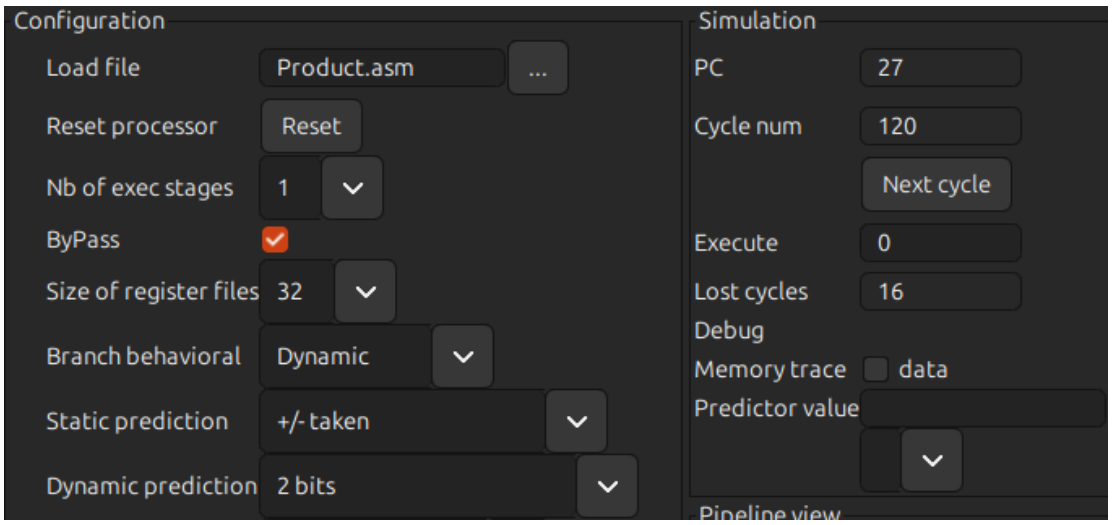


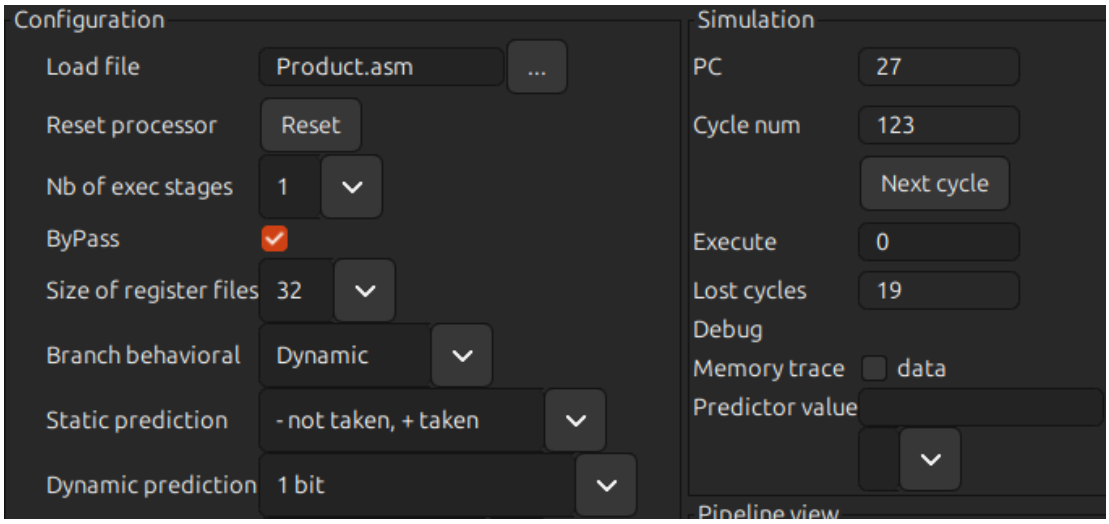
  - Dynamic

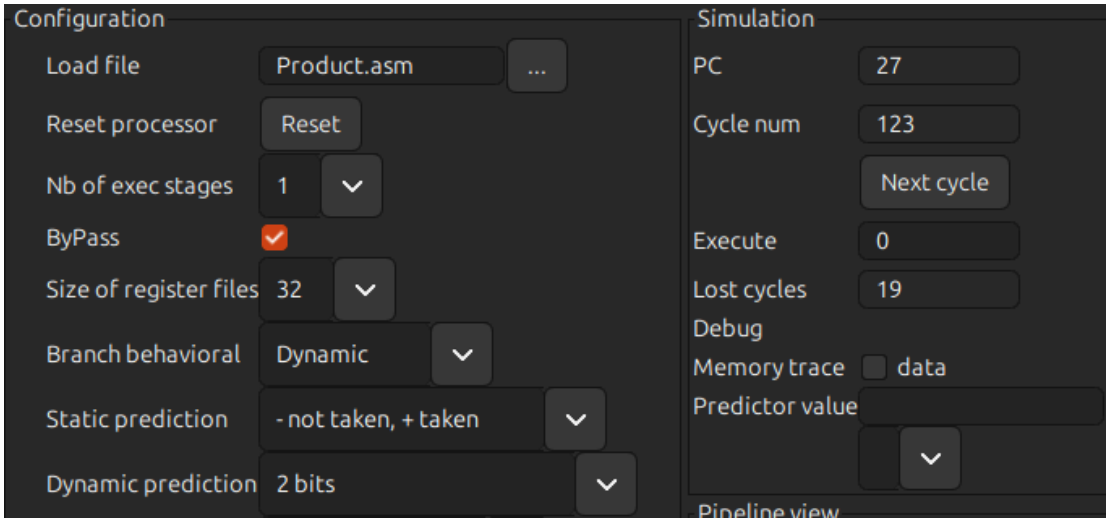


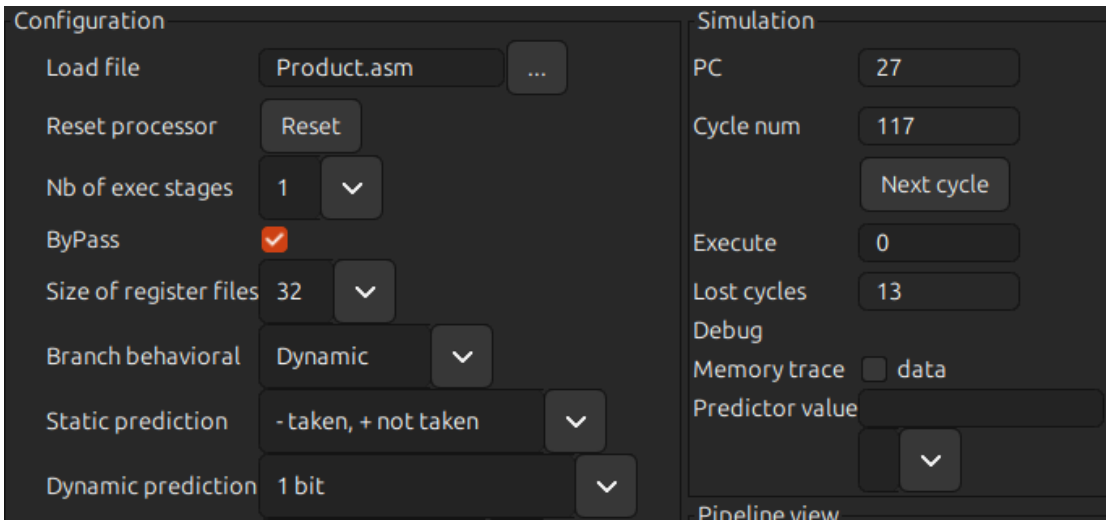

Configuration
Load file
Product.asm
...
Reset processor
Reset
Nb of exec stages
1
ByPass
Size of register files
32
Branch behavioral
Dynamic
Static prediction
+/- not taken
Dynamic prediction
2 bits
Simulation
PC
27
Cycle num
120
Next cycle
Execute
0
Lost cycles
16
Debug
Memory trace
data
Predictor value
Pipeline view

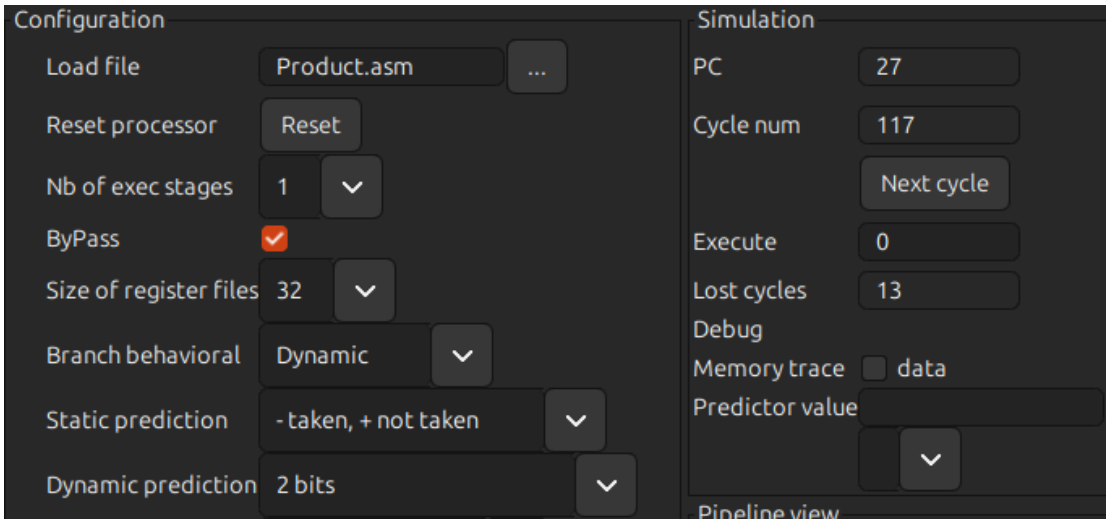

Configuration
Load file
Product.asm
...
Reset processor
Reset
Nb of exec stages
1
ByPass
Size of register files
32
Branch behavioral
Dynamic
Static prediction
+/- taken
Dynamic prediction
1 bit
Simulation
PC
27
Cycle num
120
Next cycle
Execute
0
Lost cycles
16
Debug
Memory trace
data
Predictor value
Pipeline view


Configuration
Load file
Product.asm
...
Reset processor
Reset
Nb of exec stages
1
ByPass
Size of register files
32
Branch behavioral
Dynamic
Static prediction
+/- taken
Dynamic prediction
2 bits
Simulation
PC
27
Cycle num
120
Next cycle
Execute
0
Lost cycles
16
Debug
Memory trace
data
Predictor value
Pipeline view

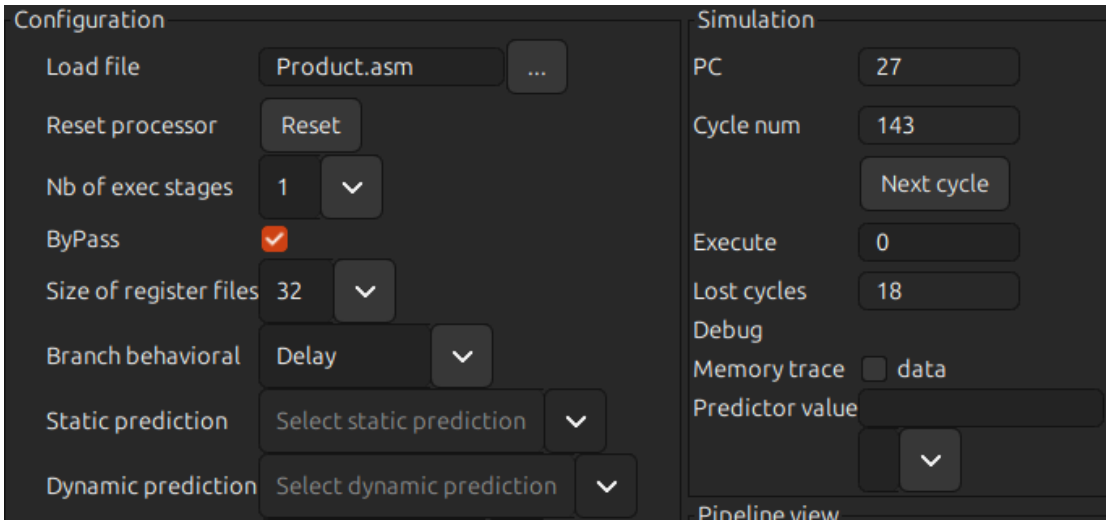

Configuration
Load file
Product.asm
...
Reset processor
Reset
Nb of exec stages
1
ByPass
Size of register files
32
Branch behavioral
Dynamic
Static prediction
- not taken, + taken
Dynamic prediction
1 bit
Simulation
PC
27
Cycle num
123
Next cycle
Execute
0
Lost cycles
19
Debug
Memory trace
data
Predictor value
Pipeline view


Configuration
Load file
Product.asm
...
Reset processor
Reset
Nb of exec stages
1
ByPass
Size of register files
32
Branch behavioral
Dynamic
Static prediction
- not taken, + taken
Dynamic prediction
2 bits
Simulation
PC
27
Cycle num
123
Next cycle
Execute
0
Lost cycles
19
Debug
Memory trace
data
Predictor value
Pipeline view


Configuration
Load file
Product.asm
...
Reset processor
Reset
Nb of exec stages
1
ByPass
Size of register files
32
Branch behavioral
Dynamic
Static prediction
- taken, + not taken
Dynamic prediction
1 bit
Simulation
PC
27
Cycle num
117
Next cycle
Execute
0
Lost cycles
13
Debug
Memory trace
data
Predictor value
Pipeline view



- Step 5
  - If the delayed branch is selected, we should modify the code before executing it.
  - Explanation: When implementing the delayed branch technique, the processor executes the conditional branch instruction in the usual manner but postpones the calculation of the branch target address. Instead, it proceeds to execute the subsequent instructions following the branch instruction until the branch target address is determined. Upon computing the branch target address, the processor retrieves the instruction located at that address and initiates its execution.
  - Without optimization and correction

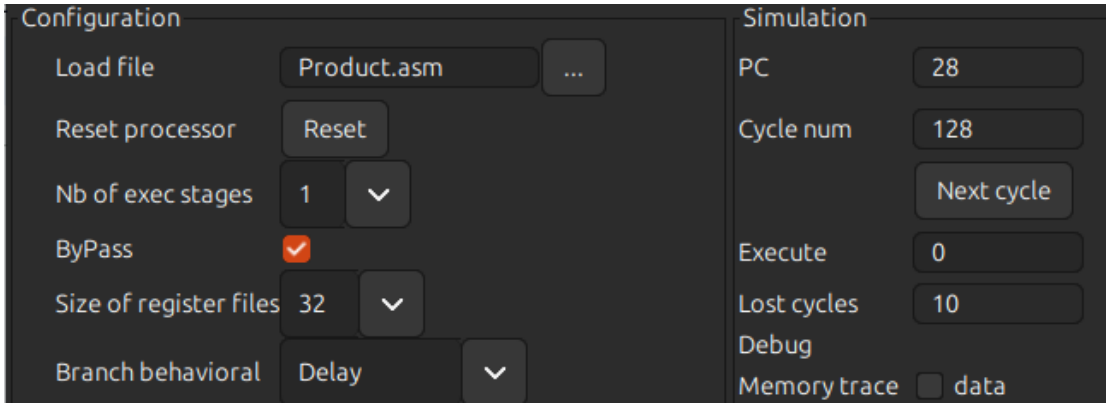


    - As can be seen, when I select Delay Branch Behavioral, the cycle it takes to execute the code is 143, the same with normal execution with nothing enabled and less lost cycles with only 18

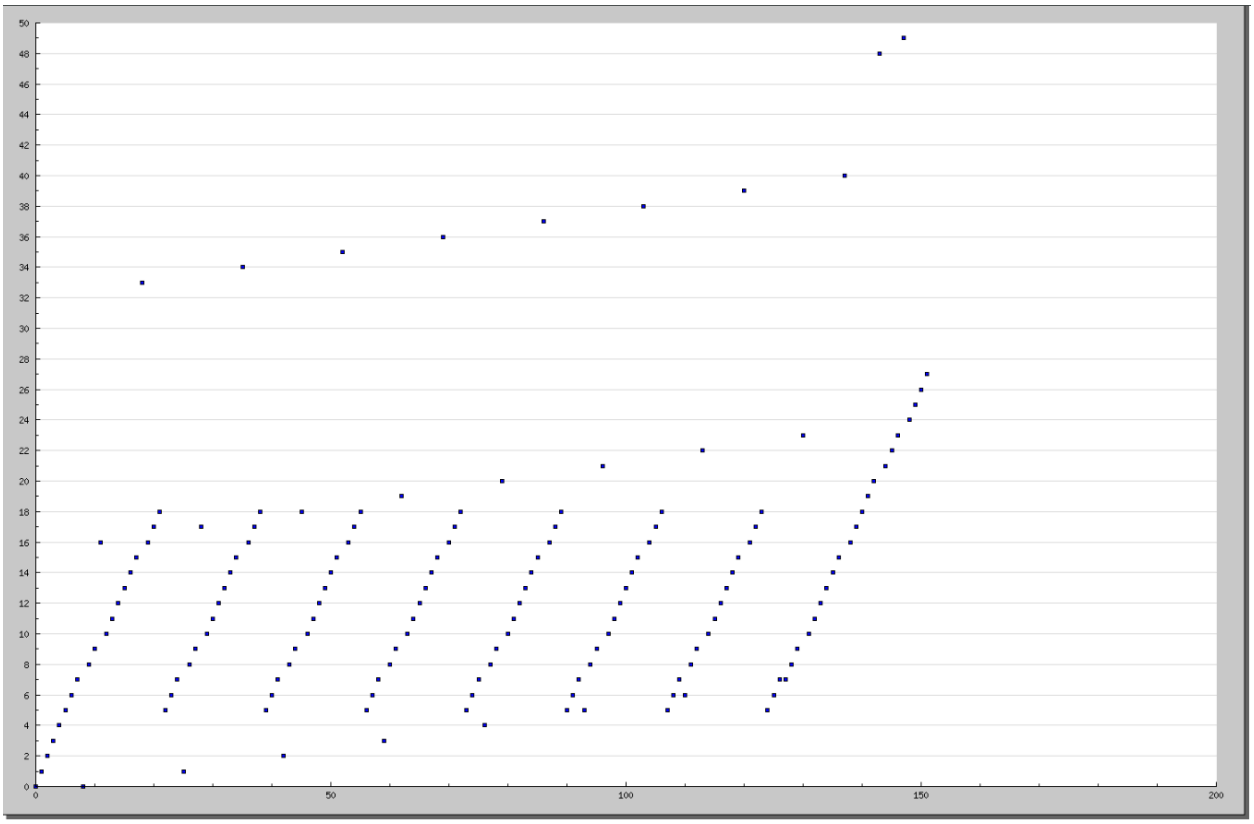
        - But the result is wrong

A...	0	1	2	3	4	5	6	7	8
0	1	7	0	7	2	0	0	4	0
16	1	9	1	2	2	0	0	4	0
32	1	63	0	14	4	0	0	16	0
48	1	64	64	78	82	82	82	98	98

    - Optimize the code
        - I have make some change to the code in order that reduce the most amount of dependent pair of instructions as well as add a NOP below the “BRNZ R7, LOOP”
          → New result is correct and has only 10 lost cycles:



- Step 6
    - Cache size: 64 mots
    - Block size: 4 mots
    - Associativity: Direct
    - Replacement policy: LRU
    - Total fetches: 152
    - Total misses: 11 → total hits: 141



- Other configuration with Cache size = 64 mots and

Associativity: Complete Replacement policy: LRU Block size = 4 mots	Total fetches: 152 Total misses: 11 → total hits: 141 → Hit ratio: 92%
Associativity: Direct Replacement policy: LRU Block size = 8 mots	Total fetches: 152 Total misses: 7 → total hits: 145 → Hit ratio: 95%
Associativity: Complete Replacement policy: LRU Block size = 8 mots	Total fetches: 152 Total misses: 7 → total hits: 145 → Hit ratio: 95%
Associativity: Direct Replacement policy: LRU Block size = 16 mots	Total fetches: 152 Total misses: 4 → total hits: 148 → Hit ratio: 97%

- As can be seen from the table above, when we change the associativity, the data will not change. But when we increase the number of Block size,

the Hit ratio will increase but it does not mean that it will be better than other configurations with less Block size.

- However, increasing cache size and block size can also increase access latency and reduce cache hit rates due to increased memory fragmentation and also time for memory and cache to access.
- The best configuration with Cache size of 64 mots could be Block size with 8 mots and any associativity in this case.