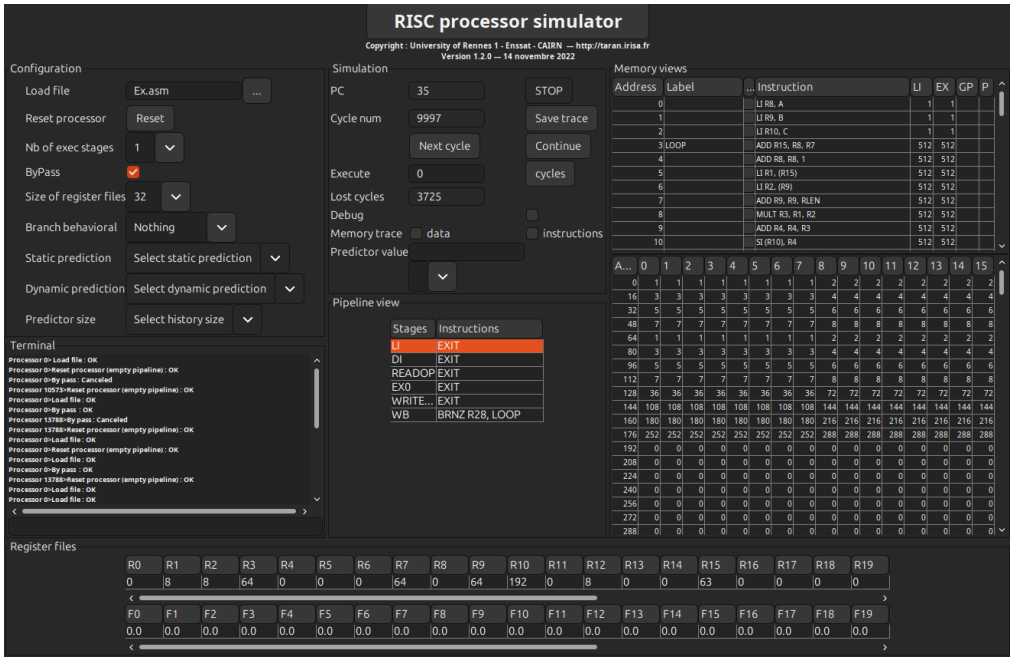
Nguyen Tai Anh- 22BI13028- Practical 3- Advanced CA and x86 ISA

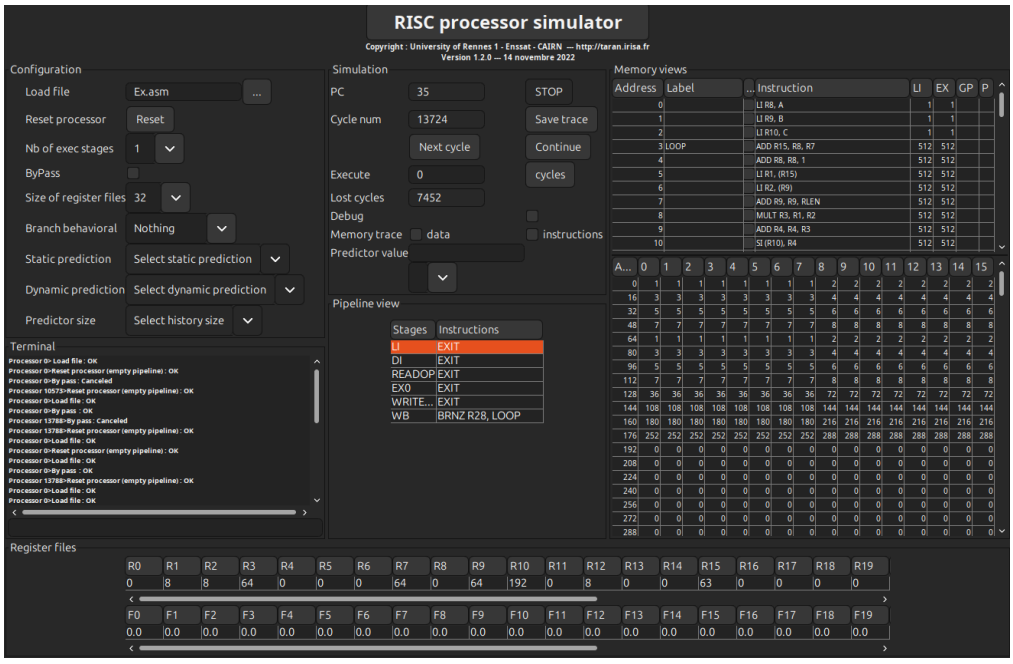
Step 1: Analyze Bypass technique

- Execute code without and with bypass

With ByPass:



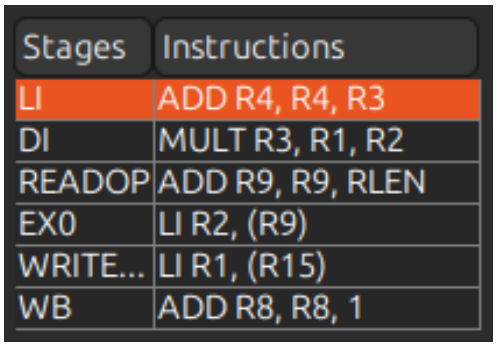
Without ByPass:



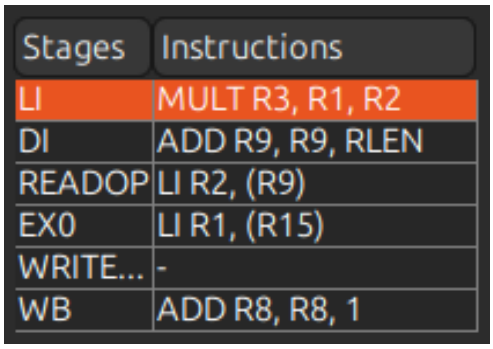
What is ByPass technique:

+ 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.



Stages	Instructions
LI	ADD R4, R4, R3
DI	MULT R3, R1, R2
READOP	ADD R9, R9, RLEN
EX0	LI R2, (R9)
WRITE...	LI R1, (R15)
WB	ADD R8, R8, 1



Stages	Instructions
LI	MULT R3, R1, R2
DI	ADD R9, R9, RLEN
READOP	LI R2, (R9)
EX0	LI R1, (R15)
WRITE...	-
WB	ADD R8, R8, 1

As can be seen from the two images above, on the left, ByPass is enabled and on the right, ByPass is disabled, we can see the main difference between them. After calculating R15 from R8, the result of R15 immediately directs to Li R1, (R15) and executes the instruction when ByPass is on. But when ByPass is disable, there will be a lost cycle between ADD R8, R8, 1 and Li R1, (R15), means that after calculating R15 from R8, the result of R15 must be write to the register first and then next direct to Li R1, (R15) later (or Li R1, (R15) have to wait for the previous instruction to be written first).

- Analyze the results (nb of cycles), comments, conclusion

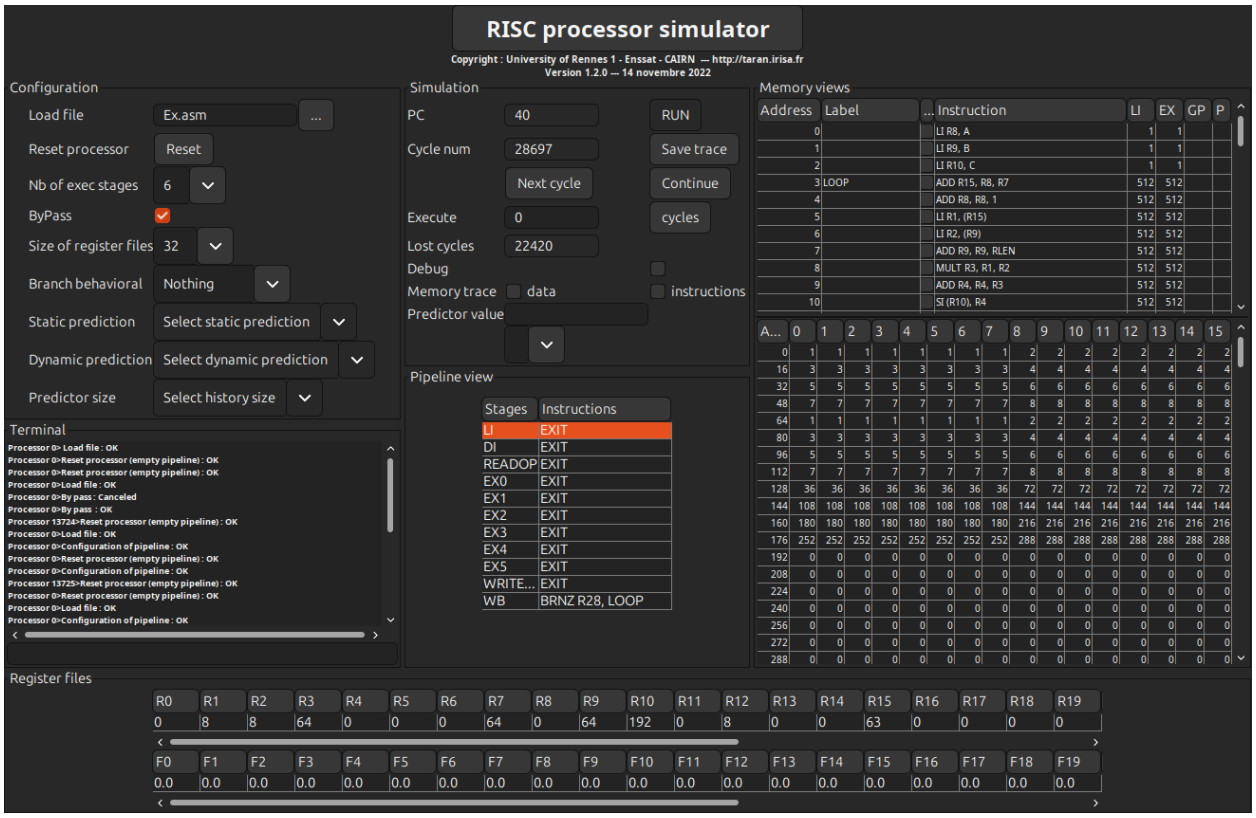
With ByPass: There are 9997 cycle numbers, and 3725 of them are lost during the execution.

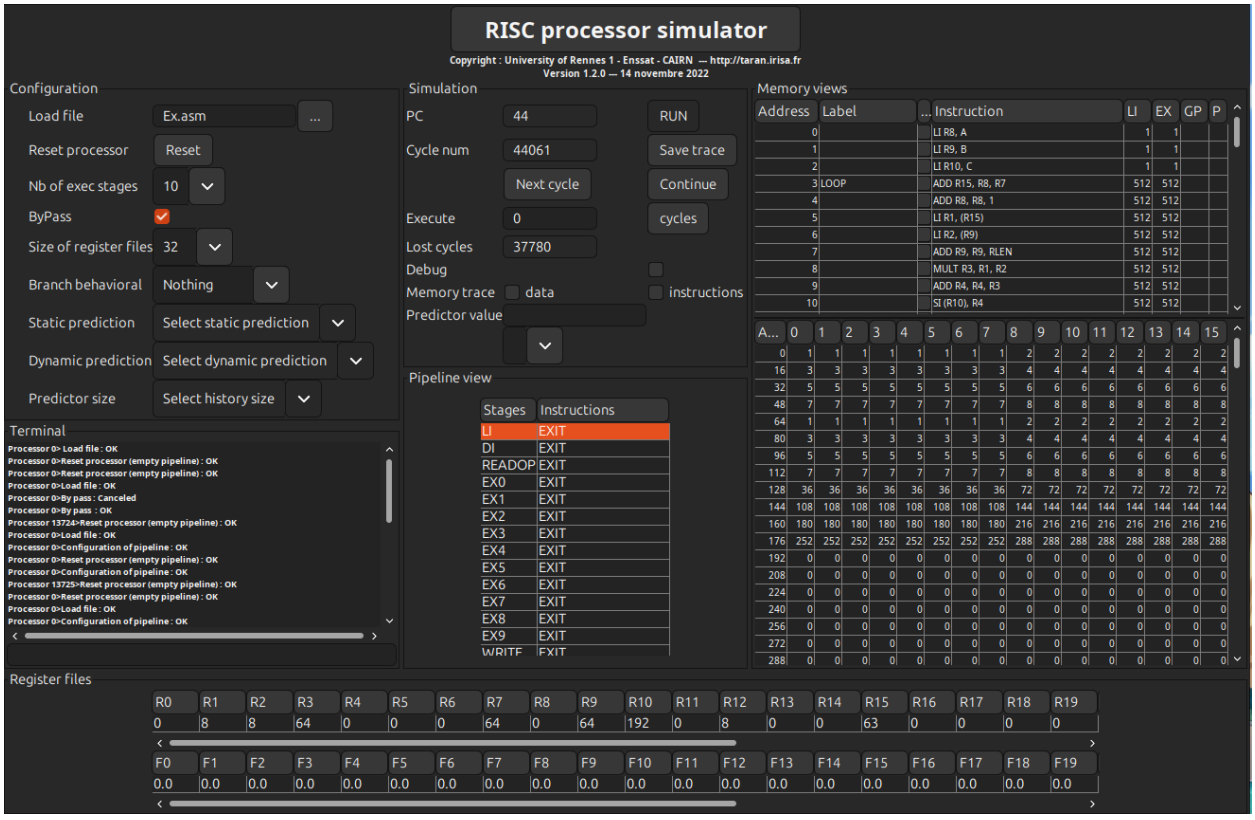
Without ByPass: There are 13724 cycle numbers, and 7452 of them are lost during the execution.

→ As can be seen, without ByPass, there are more cycles as well as lost cycles during execution.

→ With bypass, the pipeline can avoid the stalls forward the result of the first instruction to the second instruction (The result of the first instruction will directly transfer to the second instruction)→ no waiting for the result written to register file and read back in → instructions are executed more quickly and efficiently.

- Modify the nb of pipeline execution stages, and analyze what happen


RISC processor simulator
Copyright : University of Rennes 1 - Enssat - CAIRN — http://taran.irisa.fr
Version 1.2.0 — 14 novembre 2022
Configuration
Load file
Ex.asm
Reset processor
Reset
Nb of exec stages
6
ByPass
Size of register files
32
Branch behavioral
Nothing
Static prediction
Select static prediction
Dynamic prediction
Select dynamic prediction
Predictor size
Select history size
Simulation
PC
40
RUN
Cycle num
28697
Save trace
Next cycle
Continue
Execute
0
cycles
Lost cycles
22420
Debug
Memory trace
data
instructions
Predictor value
Pipeline view
Stages
Instructions
LI EXIT
DI EXIT
READOP EXIT
EX0 EXIT
EX1 EXIT
EX2 EXIT
EX3 EXIT
EX4 EXIT
EX5 EXIT
WRITE... EXIT
WB BRNZ R28, LOOP
Memory views
Address
Label
Instruction
LI
EX
GP
P
0 LI R8, A 1 1
1 LI R9, B 1 1
2 LI R10, C 1 1
3 LOOP ADD R15, R8, R7 512 512
4 ADD R8, R8, 1 512 512
5 LI R1, (R15) 512 512
6 LI R2, (R9) 512 512
7 ADD R9, R9, RLEN 512 512
8 MULT R3, R1, R2 512 512
9 ADD R4, R4, R3 512 512
10 SI (R10), R4 512 512
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 13724>Reset processor (empty pipeline) : OK
Processor 0>Load file : OK
Processor 0>Configuration of pipeline : OK
Processor 0>Reset processor (empty pipeline) : OK
Processor 0>Configuration of pipeline : OK
Processor 13725>Reset processor (empty pipeline) : OK
Processor 0>Reset processor (empty pipeline) : OK
Processor 0>Load file : OK
Processor 0>Configuration of pipeline : OK
Register files
R0 R1 R2 R3 R4 R5 R6 R7 R8 R9 R10 R11 R12 R13 R14 R15 R16 R17 R18 R19
0 8 8 64 0 0 0 64 0 64 192 0 8 0 0 63 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


RISC processor simulator
Copyright : University of Rennes 1 - Enssat - CAIRN — http://taran.irisa.fr
Version 1.2.0 — 14 novembre 2022
Configuration
Load file
Ex.asm
Reset processor
Reset
Nb of exec stages
10
ByPass
Size of register files
32
Branch behavioral
Nothing
Static prediction
Select static prediction
Dynamic prediction
Select dynamic prediction
Predictor size
Select history size
Simulation
PC
44
RUN
Cycle num
44061
Save trace
Next cycle
Continue
Execute
0
cycles
Lost cycles
37780
Debug
Memory trace
data
instructions
Predictor value
Pipeline view
Stages
Instructions
LI EXIT
DI EXIT
READOP EXIT
EX0 EXIT
EX1 EXIT
EX2 EXIT
EX3 EXIT
EX4 EXIT
EX5 EXIT
EX6 EXIT
EX7 EXIT
EX8 EXIT
EX9 EXIT
WRITE EXIT
Memory views
Address
Label
Instruction
LI
EX
GP
P
0 LI R8, A 1 1
1 LI R9, B 1 1
2 LI R10, C 1 1
3 LOOP ADD R15, R8, R7 512 512
4 ADD R8, R8, 1 512 512
5 LI R1, (R15) 512 512
6 LI R2, (R9) 512 512
7 ADD R9, R9, RLEN 512 512
8 MULT R3, R1, R2 512 512
9 ADD R4, R4, R3 512 512
10 SI (R10), R4 512 512
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 13724>Reset processor (empty pipeline) : OK
Processor 0>Load file : OK
Processor 0>Configuration of pipeline : OK
Processor 0>Reset processor (empty pipeline) : OK
Processor 0>Configuration of pipeline : OK
Processor 13725>Reset processor (empty pipeline) : OK
Processor 0>Reset processor (empty pipeline) : OK
Processor 0>Load file : OK
Processor 0>Configuration of pipeline : OK
Register files
R0 R1 R2 R3 R4 R5 R6 R7 R8 R9 R10 R11 R12 R13 R14 R15 R16 R17 R18 R19
0 8 8 64 0 0 0 64 0 64 192 0 8 0 0 63 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

Note: I only show the ByPass enabled case when changing the nb of exec stages!!!

Conclusion: As can be seen from these pictures above, I set the number of execution stages from 1, 2, 4, 6, 10 → when the number of execution stages increases, the cycle numbers and lost cycles also increase. But it seems like the lost cycles increase faster and more compared to the cycle numbers.

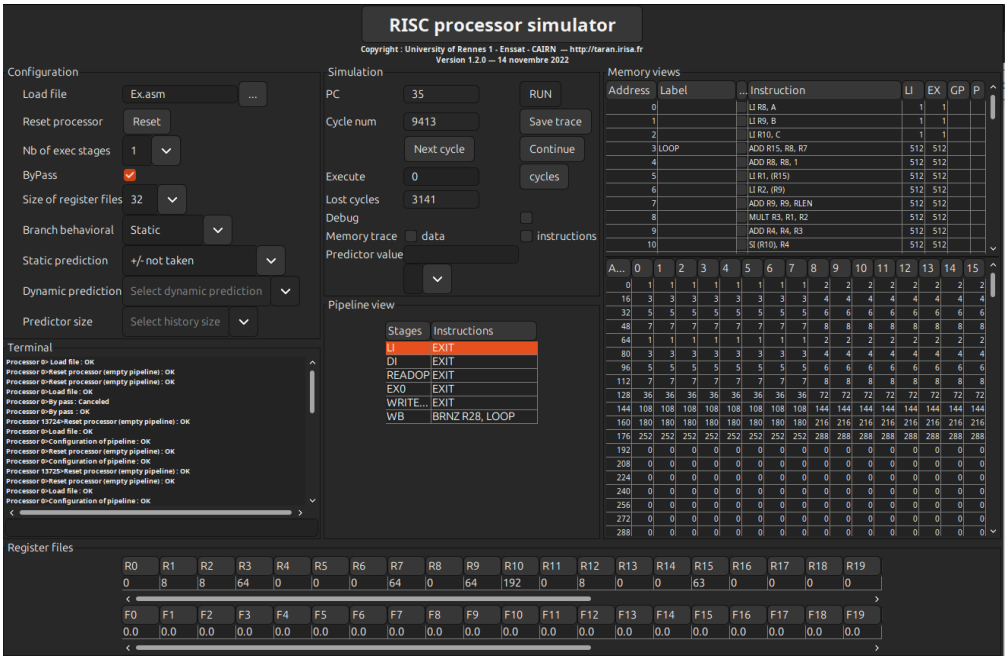
Number of exec stage(s)	ByPass enabled		ByPass disabled	
	Cycle numbers	Lost cycles	Cycle numbers	Lost cycles
1	9997	3725	13742	7452
2	13725	7452	17453	11180
4	21183	14908	24911	18636
6	28697	22420	32425	26148
10	44061	37780	47789	41508

Explanation: Increasing the number of execution stages in a pipeline can improve instruction throughput by allowing for higher clock speeds and more parallel instruction processing (means that the processor will work more and reduce the execution time). However, this also introduces greater complexity in managing pipeline hazards, increases the branch misprediction penalty, and adds overhead for pipeline control. These factors contribute to a higher number of total cycles and lost cycles, as the processor works to handle the additional challenges introduced by the deeper pipeline.

Step 2: Analyze branch prediction techniques (static)

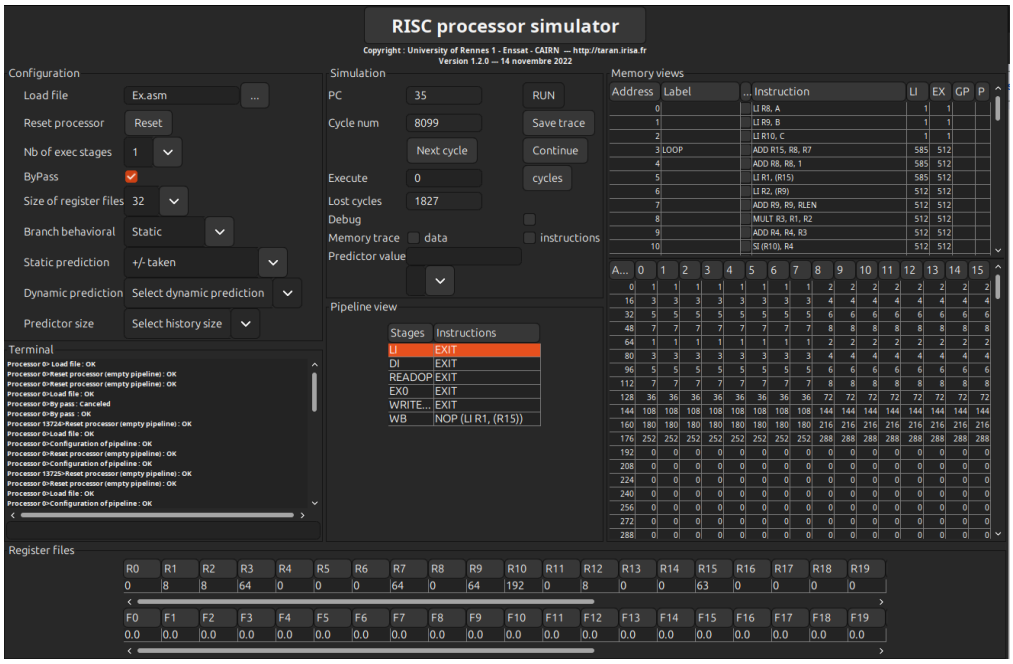
- Configure static prediction + Analyze the nb of cycles, comments, conclusion

+, **“+/- not taken”:**



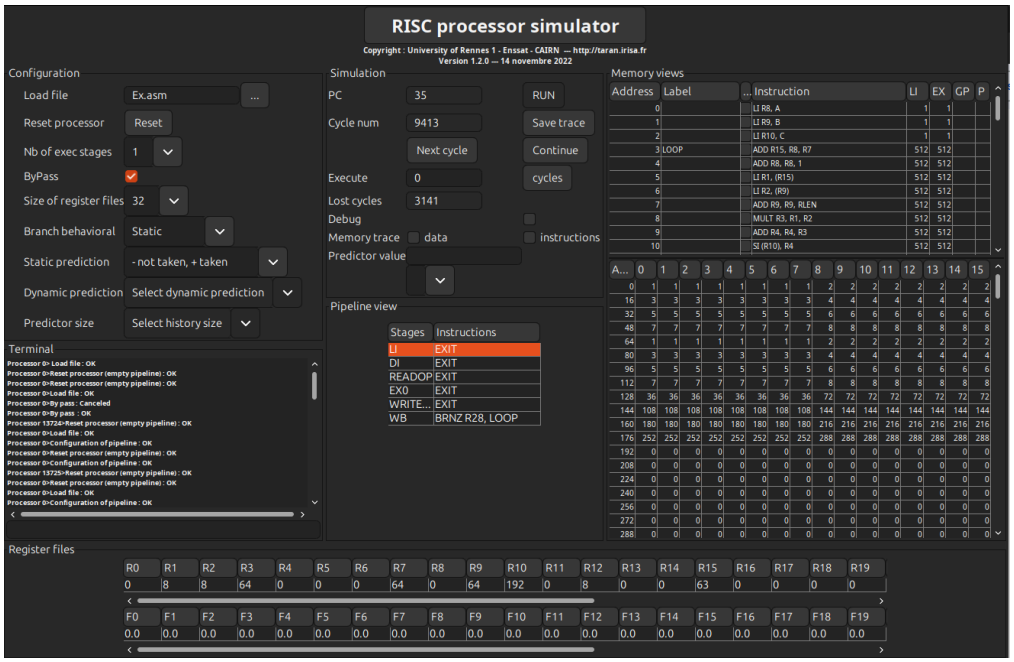
The “+/- not taken” means that for backward and forward conditional branch instructions, the processor will try to NOT jump. As can be seen from the image, when I enable Static branch behavior and set the Static prediction to “+/- not taken”, the Cycle num and Lost cycles slightly decreases to 9413 and 3141

+, “**+/- taken”:**



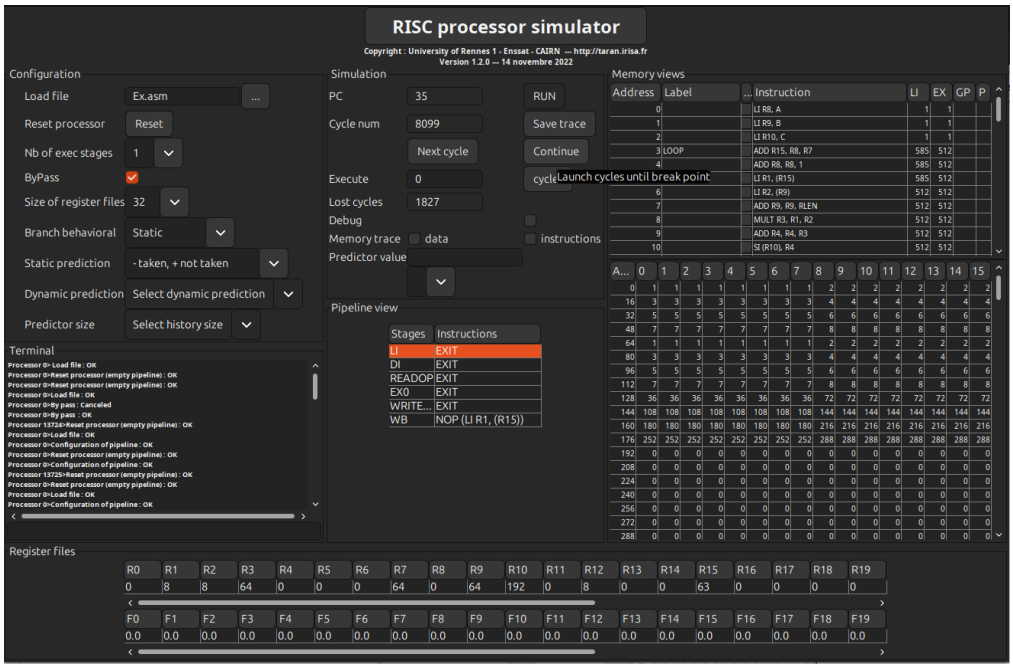
"+/- taken means" that for backward and forward conditional branch instructions, the processor will try to jump. As can be seen from the image above, after choosing Static prediction to "+/- taken", the cycle num and lost cycles have a significant decrease from the original no branch behavior execution with 8099 cycle num and 1827 lost cycles.

+, "**- not taken, + taken":**



“- not taken, + taken” means that for backward conditional branch instructions, the processor will try to NOT jump and for forward conditional branch instructions, the processor will try to jump. As can be seen from the image, the result after execution is exactly the same as when executed with prediction “+/- not taken”. There are 9413 cycle num and 3141 lost cycles.

+, “**- taken, + not taken”:**



“- taken, + not taken” means that for backward conditional branch instructions, the processor will try to jump and for forward conditional branch instructions, the processor will try to NOT jump. As can be seen from the image, the result is the same as the situation when “+/- taken” with 8099 cycle num and 1827 lost cycles.

- The good configuration among these 4 predictions that can be seen is “+/- not taken” and “- taken, + not taken” as they are the clearest and have the least number of cycles as well as lost cycles. Reason why they have the same result is because in my code, there are only backward (or “-”) branch instructions.

- Analyze cycle by cycle (- taken, + not taken)

In all my loops, there are only backward branch instructions and no forward branch instructions.

For loop1, loop2 and loop3:

```
.Loop
    add R15, R8, R7
    add R8, R8, 1
    li R1, (R15)

    li R2, (R9)
    add R9, R9, RLEN

    mult R3, R1, R2
    add R4, R4, R3
    si (R10), R4

    sub R5, R5, 1
    add R6, R5, RLEN
    brnz R6, Loop

    add R10, R10, 1
```

Stages	Instructions
LI	ADD R15, R8, R7
DI	NOP (LI R5, 0)
READOP	NOP (LI R4, 0)
EX0	NOP (ADD R10, R1...
WRITE...	BRNZ R6, LOOP
WB	-

Not using Static branch prediction

Stages	Instructions
LI	LI R2, (R9)
DI	LI R1, (R15)
READOP	ADD R8, R8, 1
EX0	ADD R15, R8, R7
WRITE...	BRNZ R6, LOOP
WB	ADD R6, R5, RLEN

Using Static branch prediction

When configuring Static branch behavior and enabling Static mode to " - taken, + not taken", the backward branch instruction will be predicted and remove all the NOP instructions that come after these backward branches while forward branch instructions will not be predicted. In this case, my code only has backward branch instructions.

As can be seen, in loop1, when Static branch prediction "- taken, + not taken" is disabled, after the "BRNZ R6, LOOP" instruction is written, there are 3 NOP instructions from loop2 are executed before instructions in loop1 are executed again.

But when Static branch prediction "- taken, + not taken" is enabled, after the instruction "BRNZ R6, LOOP" is written, instruction from loop1 is executed again immediately rather than 3 NOP instructions from loop2. → Reduce the cycle numbers and lost cycles during execution of the whole program.

For loop2 and loop3, there are also two BRNZ instructions → The result will be the same as in loop1. These branches will also skip the NOP instructions from their next loop and immediately come back to instructions in their loop.

```
.Loop2
    li R4, 0
    li R5, 0
    li R8, 0

    sub R9, R9, 63

    add R11, R11, 1
    sub R29, R11, RLEN
    brnz R29, Loop

    add R7, R7, RLEN
```

```
.Loop3
    li R11, 0
    li R9, SIZE

    add R12, R12, 1
    sub R28, R12, RLEN

    brnz R28, Loop
```

Step 3: Configure dynamic branch prediction, 1 bit (explain how you configure)

- **Configure dynamic branch prediction, 1-bit (explain how you configure)**

  I change the Branch behavior to Dynamic and choose Dynamic prediction to 1-bit.

- Analyze the nb of cycles, comments, conclusion

As can be seen from the image, when I enable Dynamic branch behavior technique, the cycle number decreases to 8309 compared with more than 9900 originally. The lost cycles decrease significantly to 2037.

Explanation: Dynamic branch prediction is a technique used in modern processors to guess the outcome of a branch instruction (whether it will be taken or not) at runtime, rather than relying on a fixed strategy (static prediction). This technique uses historical information to make more accurate predictions, improving instruction pipeline efficiency and overall processor performance. That is why the lost cycles decrease so much as the cycle number decreases slightly.

- Analyze cycle by cycle

In all my loops, there are only backward branch instructions and no forward branch instructions.

For loop1, loop2 and loop3:

```
.Loop
    add R15, R8, R7
    add R8, R8, 1
    li R1, (R15)

    li R2, (R9)
    add R9, R9, RLEN

    mult R3, R1, R2
    add R4, R4, R3
    si (R10), R4

    sub R5, R5, 1
    add R6, R5, RLEN
    brnz R6, Loop

    add R10, R10, 1
```

Stages	Instructions
LI	ADD R15, R8, R7
DI	NOP (LI R5, 0)
READOP	NOP (LI R4, 0)
EX0	NOP (ADD R10, R1...
WRITE...	BRNZ R6, LOOP
WB	-

Not using Dynamic branch prediction: 1-bit mode

Stages	Instructions
LI	LI R2, (R9)
DI	LI R1, (R15)
READOP	ADD R8, R8, 1
EX0	ADD R15, R8, R7
WRITE...	BRNZ R6, LOOP
WB	ADD R6, R5, RLEN

Using Dynamic branch prediction: 1-bit mode

A 1-bit branch predictor utilizes a single bit to remember the outcome of the last branch instruction. This bit, often referred to as a prediction bit, can be in one of two states: "taken" or "not taken." In the given scenario, the first bit would be set to 1 because when the instruction "BRNZ R6, LOOP" is executed, there are no intervening instructions such as "NOP (ADD R10, R10, 1)" or "NOP (LI R4, 0)" following it before reaching instructions within the Loop1 ("ADD R15, R8, R7"). Consequently, based on

this history, the predictor would predict that the branch will be taken in subsequent executions. During program execution, this prediction bit would be updated dynamically based on the actual outcomes of branch instructions, thereby refining the predictor's accuracy over time.

Memory views

Address	Label	...	Instruction	LI	EX	GP	P
9			ADD R4, R4, R3	4	4		
10			SI (R10), R4	4	3		
11			SUB R5, R5, 1	4	3		
12			ADD R6, R5, RLEN	4	3		
13			BRNZ R6, LOOP	3	3	3	T
14			ADD R10, R10, 1	0	0		
15			BR LOOP2	0	0		
16	LOOP2		LI R4, 0	0	0		
17			LI R5, 0	0	0		
18			LI R8, 0	0	0		
19			SUB R9, R9, 63	0	0		

As well as in Loop2 and Loop3, all the branches from these loops are backward branches, we can be sure that they are also note with bit number 1 (taken) because after the branches, instructions will instantly repeat from top of Loop2 and Loop3 rather than NOP of instruction following these branches first.

```
.Loop2
    li R4, 0
    li R5, 0
    li R8, 0

    sub R9, R9, 63

    add R11, R11, 1
    sub R29, R11, RLEN
    brnz R29, Loop

    add R7, R7, RLEN
```

```
.Loop3
    li R11, 0
    li R9, SIZE

    add R12, R12, 1
    sub R28, R12, RLEN

    brnz R28, Loop
```

Memory views

Address	Label	...	Instruction	LI	EX	GP	P
18			LI R8, 0	1	1		
19			SUB R9, R9, 63	1	1		
20			ADD R11, R11, 1	1	1		
21			SUB R29, R11, RLEN	1	1		
22			BRNZ R29, LOOP	1	1	1	T
23			ADD R7, R7, RLEN	0	0		
24			BR LOOP3	0	0		
25	LOOP3		LI R11, 0	0	0		
26			LI R9, SIZE	0	0		
27			ADD R12, R12, 1	0	0		
28			SUB R28, R12, RLEN	0	0		

Branch in Loop2

Memory views

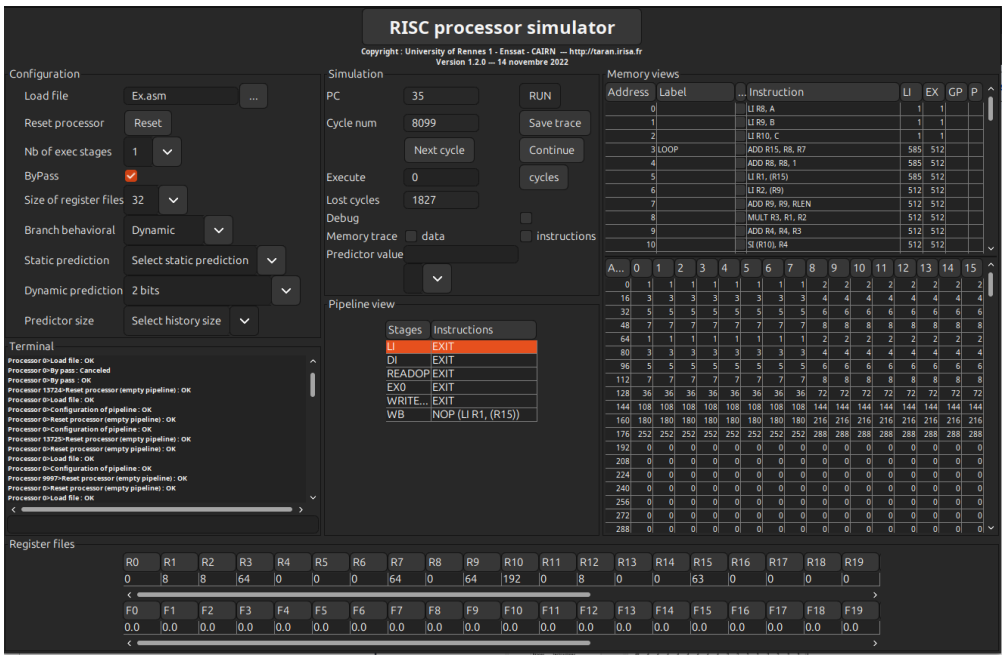
Address	Label	...	Instruction	LI	EX	GP	P
24			BR LOOP3	1	1		
25	LOOP3		LI R11, 0	1	1		
26			LI R9, SIZE	1	1		
27			ADD R12, R12, 1	1	1		
28			SUB R28, R12, RLEN	1	1		
29			BRNZ R28, LOOP	1	1	1	T
30	EXIT		EXIT	0	0		
31			EXIT	0	0		
32			EXIT	0	0		
33			EXIT	0	0		
34			EXIT	0	0		

Branch in Loop3

→ The 1-bit predictor provides a simple mechanism for improving the accuracy of branch prediction over static methods. By updating the prediction bit based on the most recent outcome, it can adapt to changing branch behaviors. However, due to its simplicity, it may not handle complex branching patterns efficiently. For example when both backward and forward branches appear in the code multiple times.

Step 4: Configure dynamic branch prediction, 2 bits (explain how you configure)

- Configure dynamic branch prediction, 2-bit (explain how you configure)



- Analyze the nb of cycles, comments, conclusion

As can be seen from the image, when I enable Dynamic branch behavior technique, the cycle number decreases to 8099 compared with more than 9900 originally. The lost cycles decreased significantly to 1827.

Explanation: A 2-bit saturating counter is used to predict whether a branch will be taken or not taken based on its recent history. Each branch has an associated 2-bit counter that can be in one of four states, representing a strong or weak prediction of either taken or not taken. The counter is updated based on the actual outcome of the branch → 2-bit saturating counters provide a robust and efficient means of dynamic branch prediction. By leveraging recent branch behavior to predict future actions, they significantly improve the accuracy of branch predictions, leading to better utilization of the processor pipeline and overall performance improvements.

- Analyze cycle by cycle

In all my loops, there are only backward branch instructions and no forward branch instructions.

For loop1, loop2 and loop3:

```
.Loop
    add R15, R8, R7
    add R8, R8, 1
    li R1, (R15)

    li R2, (R9)
    add R9, R9, RLEN

    mult R3, R1, R2
    add R4, R4, R3
    si (R10), R4

    sub R5, R5, 1
    add R6, R5, RLEN
    brnz R6, Loop

    add R10, R10, 1
```

Stages	Instructions
LI	ADD R15, R8, R7
DI	NOP (LI R5, 0)
READOP	NOP (LI R4, 0)
EX0	NOP (ADD R10, R1...
WRITE...	BRNZ R6, LOOP
WB	-

Not using Dynamic branch prediction: 2-bits mode

Stages	Instructions
LI	LI R2, (R9)
DI	LI R1, (R15)
READOP	ADD R8, R8, 1
EX0	ADD R15, R8, R7
WRITE...	BRNZ R6, LOOP
WB	ADD R6, R5, RLEN

Using Dynamic branch prediction: 2-bits mode

A 2-bit branch predictor employs a two-bit saturating counter to remember the outcome of the last branch instruction. This counter can be in one of four states: "strongly taken," "weakly taken," "weakly not taken," or "strongly not taken." In the given scenario, let's assume the initial state of the counter is "weakly not taken." When the instruction "BRNZ R6, LOOP" is executed, and there are no intervening instructions such as "NOP (ADD R10, R10, 1)" or "NOP (LI R4, 0)" following it before reaching instructions within the loop ("ADD R15, R8, R7"), the counter would transition to a more confident state of prediction, such as "weakly taken" or even "strongly taken" depending on the predictor's history. During subsequent executions, this 2-bit counter would be dynamically updated based on the actual outcomes of branch instructions, thereby refining the predictor's accuracy over time.

Memory views

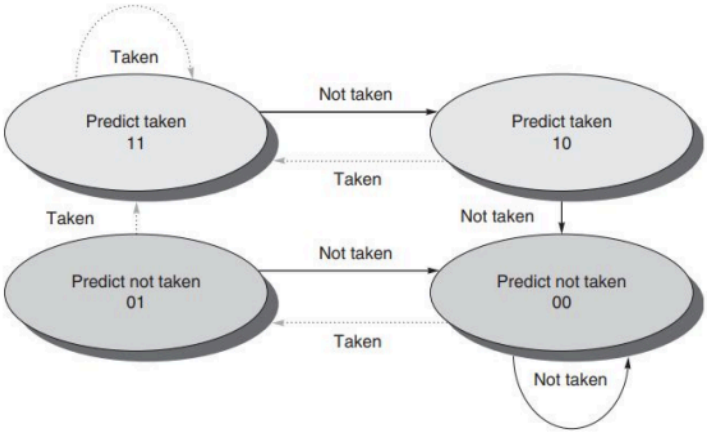
Address	Label	...	Instruction	LI	EX	GP	P
9			ADD R4, R4, R3	16	16		
10			SI (R10), R4	16	16		
11			SUB R5, R5, 1	16	16		
12			ADD R6, R5, RLEN	16	16		
13			BRNZ R6, LOOP	16	16	14	FT
14			ADD R10, R10, 1	2	1		
15			BR LOOP2	1	1		
16	LOOP2		LI R4, 0	1	1		
17			LI R5, 0	1	1		
18			LI R8, 0	1	1		
19			SUB R9, R9, 63	1	1		

As well as in Loop2 and Loop3, all the branches from these loops are backward branches, we can be sure that they are also noted with FT because after the branches, instructions will instantly repeat from top of Loop2 and Loop3 rather than NOP of instruction following these branches first.

Memory views

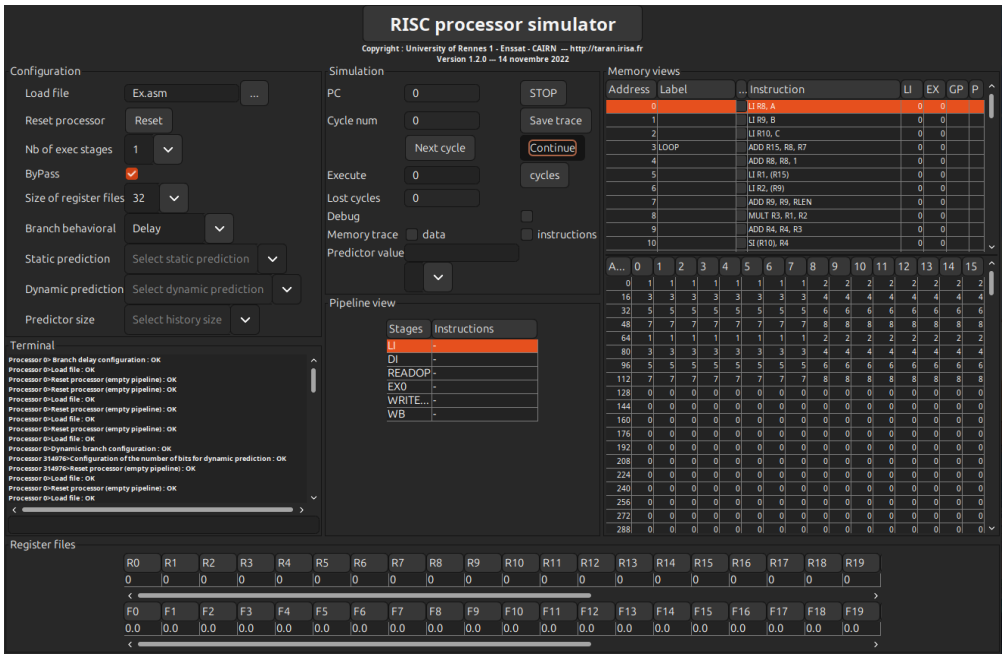
Address	Label	...	Instruction	LI	EX	GP	P
21			SUB R29, R11, RLEN	64	64		
22		✓	BRNZ R29, LOOP	64	64	56	FT
23			ADD R7, R7, RLEN	8	8		
24			BR LOOP3	8	8		
25	LOOP3		LI R11, 0	8	8		
26			LI R9, SIZE	8	8		
27			ADD R12, R12, 1	8	8		
28			SUB R28, R12, RLEN	8	8		
29		✓	BRNZ R28, LOOP	8	8	7	FT
30	EXIT		EXIT	1	0		
31			EXIT	1	0		

→ Each branch in the program is associated with a 2-bit saturating counter. This counter can take four states: strongly taken, weakly taken, weakly not taken, and strongly not taken. Initially, all counters are set to a default state, often "weakly not taken" or "weakly taken". It maintains a history of branch behavior using 2-bit saturating counters associated with each branch instruction. This history helps in making accurate predictions about whether a branch will be taken or not taken in the future. Overall, 2-bit dynamic branch prediction significantly enhances the performance of processors by reducing branch misprediction penalties and improving overall execution efficiency.



Step 5: Analyze branch prediction techniques (delay branch)

- Configure delay branch



As can be seen from the image, when I choose Delay branch behavior, the execution of the assembly code did not finish. The execution takes forever and there is no result is written back to register or memory.

- What do you have to do to ensure execution of the code

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.

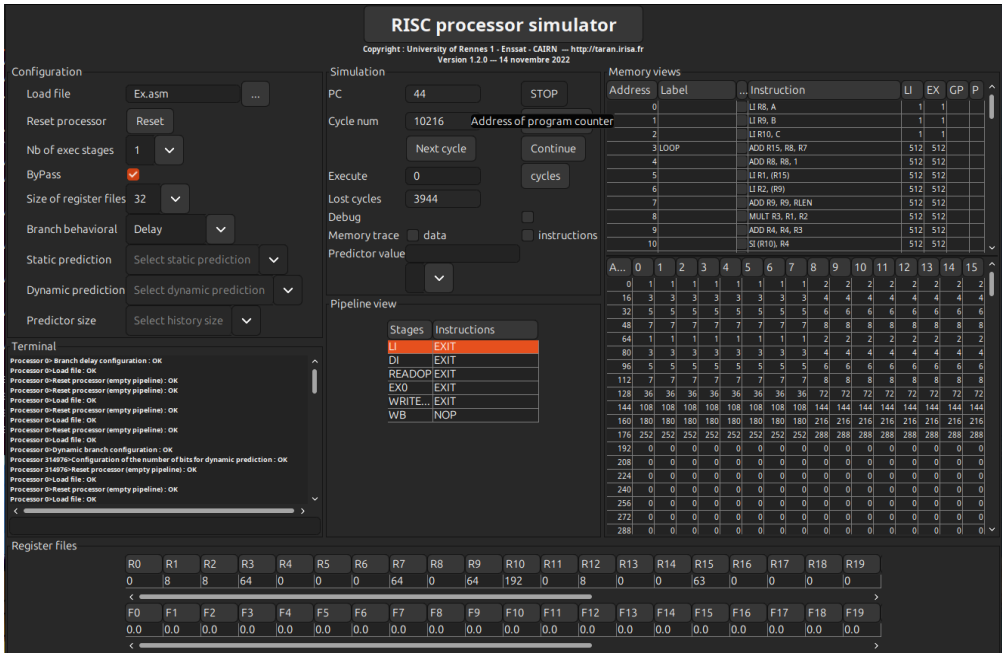
→ We have to modify the code (adding 3 NOPs following each branch “BRNZ” because there are 3 instructions are executed after them) in order to ensure that when the branch is executed, the following instruction is not executed (or stop executed) until the branch target address is determined.

- Modify the code, and verify the execution

```
.Loop
    add R15, R8, R7     -- R15 = R8 + R7 (compute address offset for current element in a)
    add R8, R8, 1       -- Increment pointer for a
    li R1, (R15)        -- Load current element of a into R1

    li R2, (R9)         -- Load current element of b into R2
    add R9, R9, RLEN    -- Move to the next row in matrix b

    mult R3, R1, R2     -- Multiply R1 and R2, store result in R3
    add R4, R4, R3      -- Add result to R4 (accumulator)
    si (R10), R4        -- Store accumulator result in current position of c

    sub R5, R5, 1       -- Decrement R5 (loop counter)
    add R6, R5, RLEN    -- R6 = R5 + RLEN
    brnz R6, Loop       -- If R6 is not zero, continue Loop
    nop
    nop
    nop

    add R10, R10, 1     -- Increment pointer for c
    br Loop2

.Loop2
    li R4, 0            -- Set R4 to 0
    li R5, 0            -- Set R5 to 0
    li R8, 0            -- Set R8 to 0

    sub R9, R9, 63      -- R9 = R9 - (SIZE - 1) to move to next column of b

    add R11, R11, 1     -- Increment column counter
    sub R29, R11, RLEN  -- R29 = R11 - RLEN
    brnz R29, Loop      -- If R29 is not zero, continue Loop
    nop
    nop
    nop

    add R7, R7, RLEN    -- Move to next row of a
    br Loop3

.Loop3
    li R11, 0           -- Reset column counter
    li R9, SIZE         -- Reset pointer for b to start of the next column

    add R12, R12, 1     -- Increment row counter
    sub R28, R12, RLEN  -- R28 = R12 - RLEN

    brnz R28, Loop      -- If R28 is not zero, continue Loop
    nop
    nop
    nop
```

As can be seen from the image above, 3 NOP instructions have been added after each BRNZ instruction in order to skip the execution of 3 instructions after these branches.

Execution after modifying the code:



- Analyze cycle by cycle

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.

Through overlapping the execution of instructions that come after the branch instruction with the computation of the branch target address, the delayed branch technique aims to minimize the number of cycles lost due to branch mispredictions. Nevertheless, this approach can lead to an increase in the overall program execution time, as it introduces additional processing overhead for computing the branch target address and managing the coordination between instructions.

Stages	Instructions
LI	ADD R15, R8, R7
DI	NOP
READOP	NOP
EX0	NOP
WRITE...	BRNZ R6, LOOP
WB	-

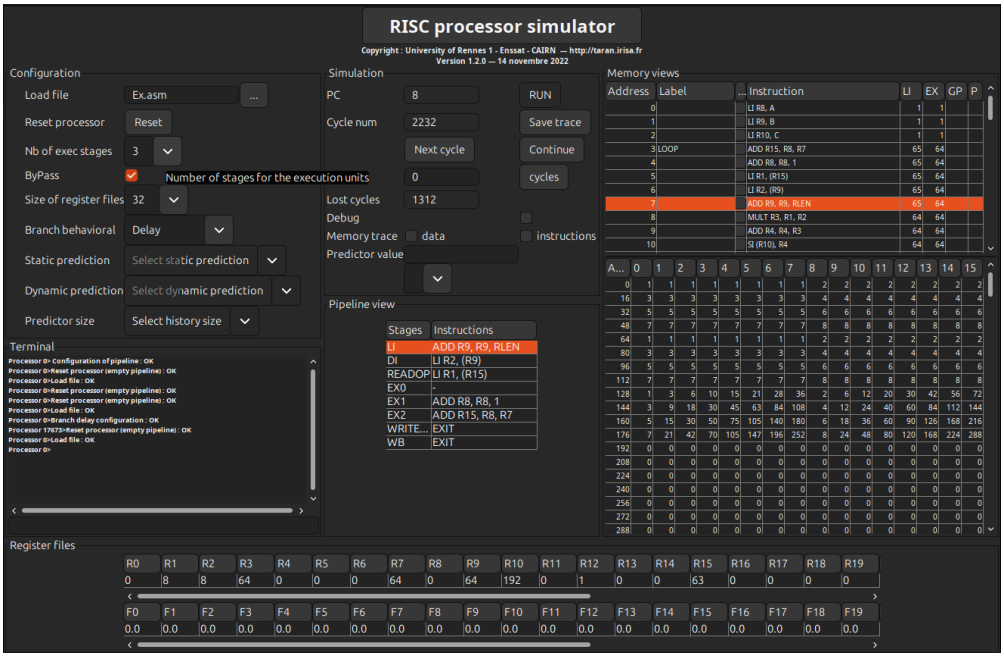
Stages	Instructions
LI	ADD R15, R8, R7
DI	LI R4, 0
READOP	BR LOOP2
EX0	ADD R10, R10, 1
WRITE...	BRNZ R6, LOOP
WB	-

In the scenario depicted in the images, when I introduced three NOP instructions after the "BRNZ R6, LOOP" instruction, the behavior changed. In the original code, following the execution of "BRNZ R6, LOOP", instructions beyond the branch, such as "ADD R10, R10, 1" and "BR Loop2", were executed. This led to an undesired outcome and an infinite loop due to the premature execution of instructions from the Loop1 block.

However, when the code was modified to include NOP instructions after the branch, the processor engaged in speculative execution, continuing with instructions beyond the branch while simultaneously initiating execution from the target of the branch. This speculative execution prevented the premature execution of instructions from the Loop1 block, potentially avoiding the infinite loop scenario observed in the original code. This behavior exemplifies how delayed branch execution can assist in maintaining program correctness and avoiding unintended outcomes by allowing the processor to defer the resolution of branches until after speculative execution has progressed.

- Modify the nb of execution stages

After I change the nb of execution stages to 3, the result is wrong again:



When I change the nb of execution stages to 5, the execution is not work due to infinite loop:

Step 6: Analyze branch prediction techniques

- Conditional move

Conditional move (CMOV) is a branch prediction technique that aims to mitigate the performance impact of conditional branches by replacing them with conditional move instructions where possible. Rather than branching based on a condition and executing different instructions depending on the outcome, conditional move instructions conditionally move data into a destination register based on a condition, typically without altering the control flow.

Syntax of Conditional move:

cmovz, cmovnz, cmovn, cmovp	Conditionnal copy from one register to another registres

The objective of this section is to attempt to replace certain conditional branch instructions with conditional moves. But JSimRISC Processor Simulation does not have conditional move instructions (means that syntax in the image above is not working in JSimRisc Processor Simulator), and even if it did, it is impossible to replace all conditional branch instructions by conditional move instructions because conditional moves can only work for complex branching logic.