Architecture Report

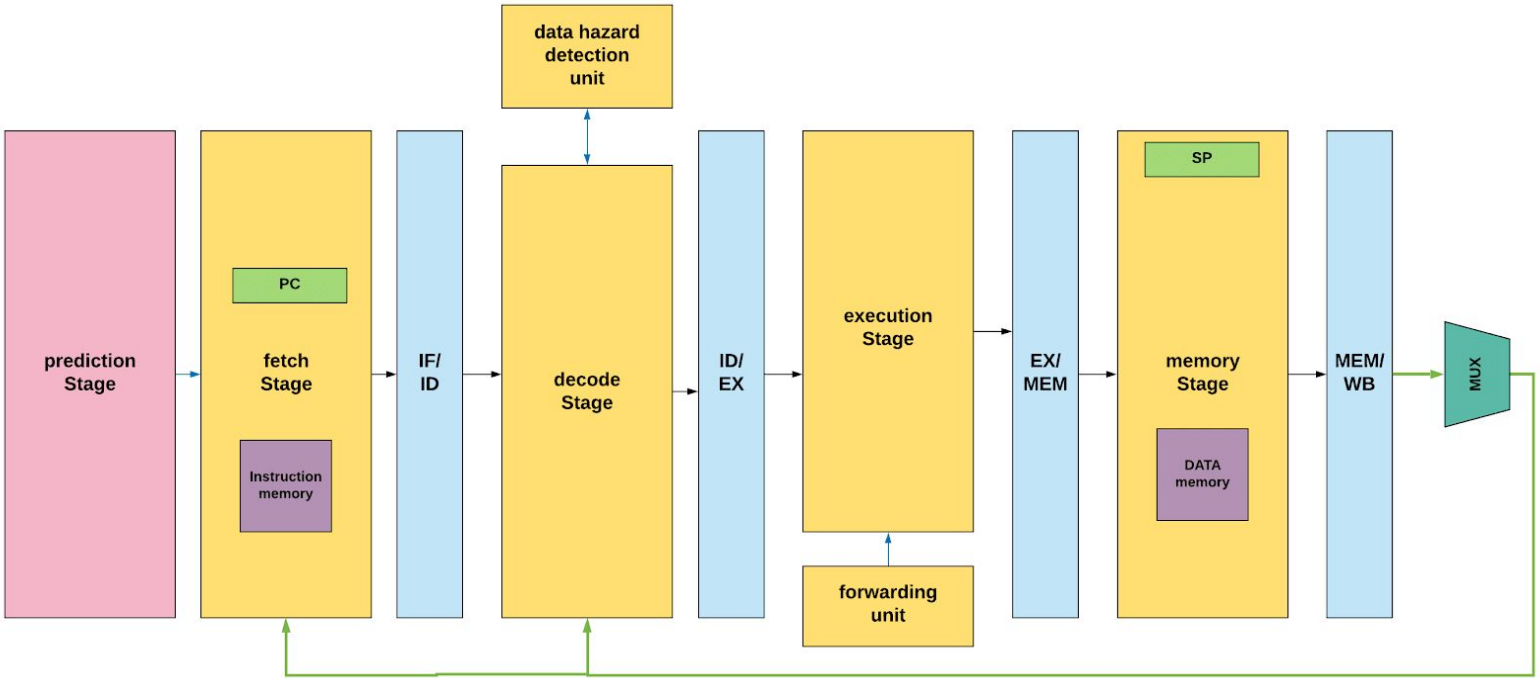
Team (6)

Team members:

Ahmed Waleed Ahmed	Sec: 1	BN: 8
Ali Khaled Ali Mohamed	Sec: 1	BN: 35
Omar Ahmed Desoky	Sec: 1	BN: 37
Mostafa Waleed	Sec: 2	BN: 24

Full Design:

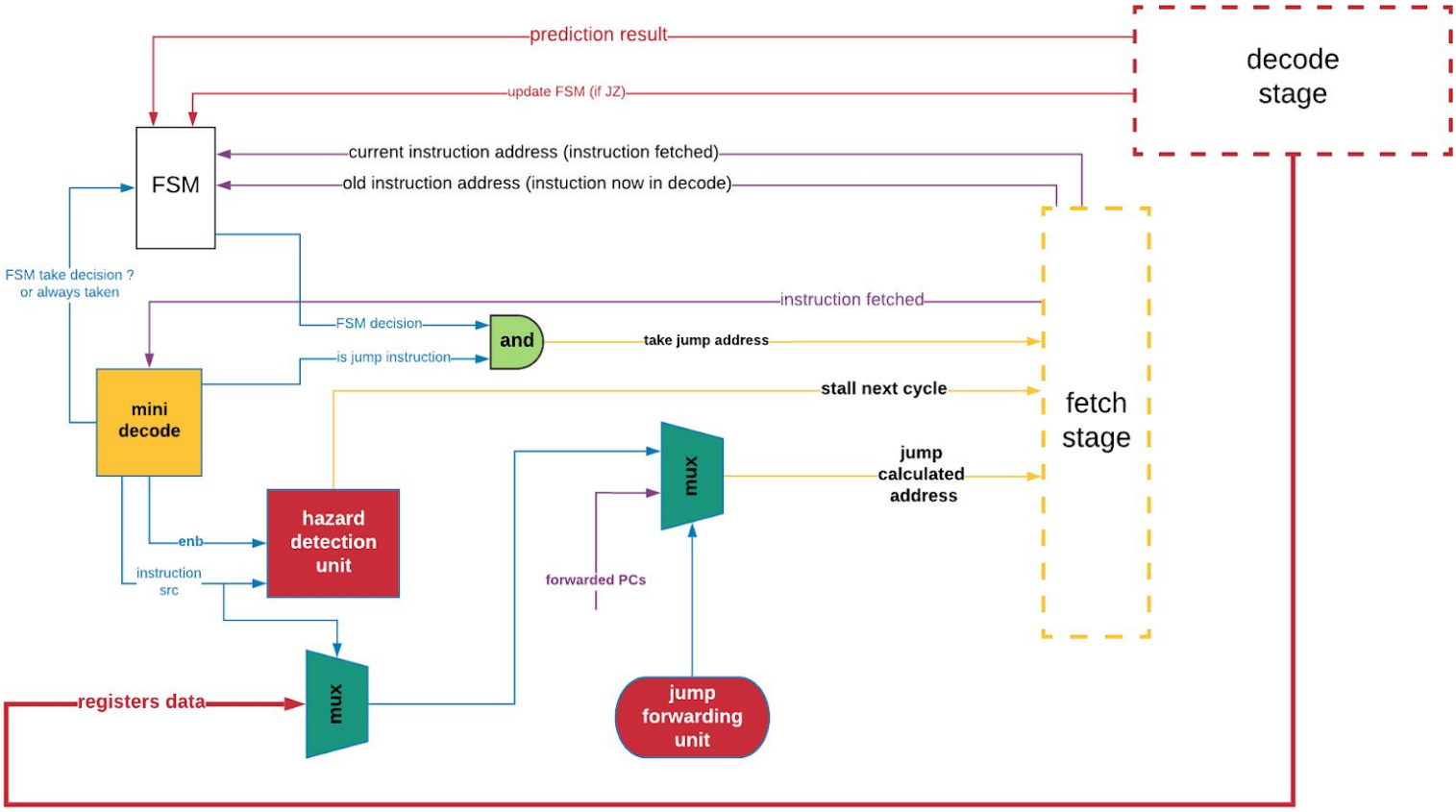
Design overview :



1. **Prediction stage :** responsible for prefetching jump address to update PC in fetch stage with and taking prediction decision (taken/not taken) according to FSM
2. **Fetch stage :** responsible for fetching an instruction from instruction memory and write it in the buffer [IF/ID] .. also responsible for selecting which pc to read
3. **Decode stage :** responsible for decoding instruction fetched and producing corresponding control signals to later stages .. also responsible for reading from decoded registers and writing to register specified at last buffer [MEM/WB]
4. **Data hazard detection unit :** responsible for stalling the pipe in load use case data hazard also used to stall pipe in case of RET/RTI/INT/two-words instructions as LDM...
5. **Execution stage :** contains ALU and responsible for the main operation of the instruction according to control signals from decode
6. **Forwarding unit :** responsible for choosing ALU inputs (data in register directly or some data forwarded from a further stage) .. providing full forwarding to prevent normal data hazards
7. **Memory stage :** responsible for reading or writing to data memory and controlling stack pointer “SP” according to control signals of the current instruction
8. **WB stage :** is just a mux to choose between memory output and ALU output to write into the required register or PC

Now we will explain each stage in detail...

1. Prediction Stage :



Contains 4 internal blocks :

1.1 Mini decode :

Decodes the instruction to 3 decisions :

1. Not jump instruction ➔ jump address will not be taken

2. “Jmp” instruction ➔ jump address will be taken always

3. “Jz” instruction ➔ jump address will be taken or not taken depending on FSM decision

Also responsible for enabling hazard detection unit in case of jump instructions and passing the code of src register to register selector and hazard unit

1.2 Hazard detection unit :

Stalls the pipe in case of not ready data “will be available after execution or after memory” to take the jump as predicted with data consistency correct

1.3 Jump forwarding unit :

Forwards data in src1 of jump instruction if it is available at further stage “not ready in register file to take”

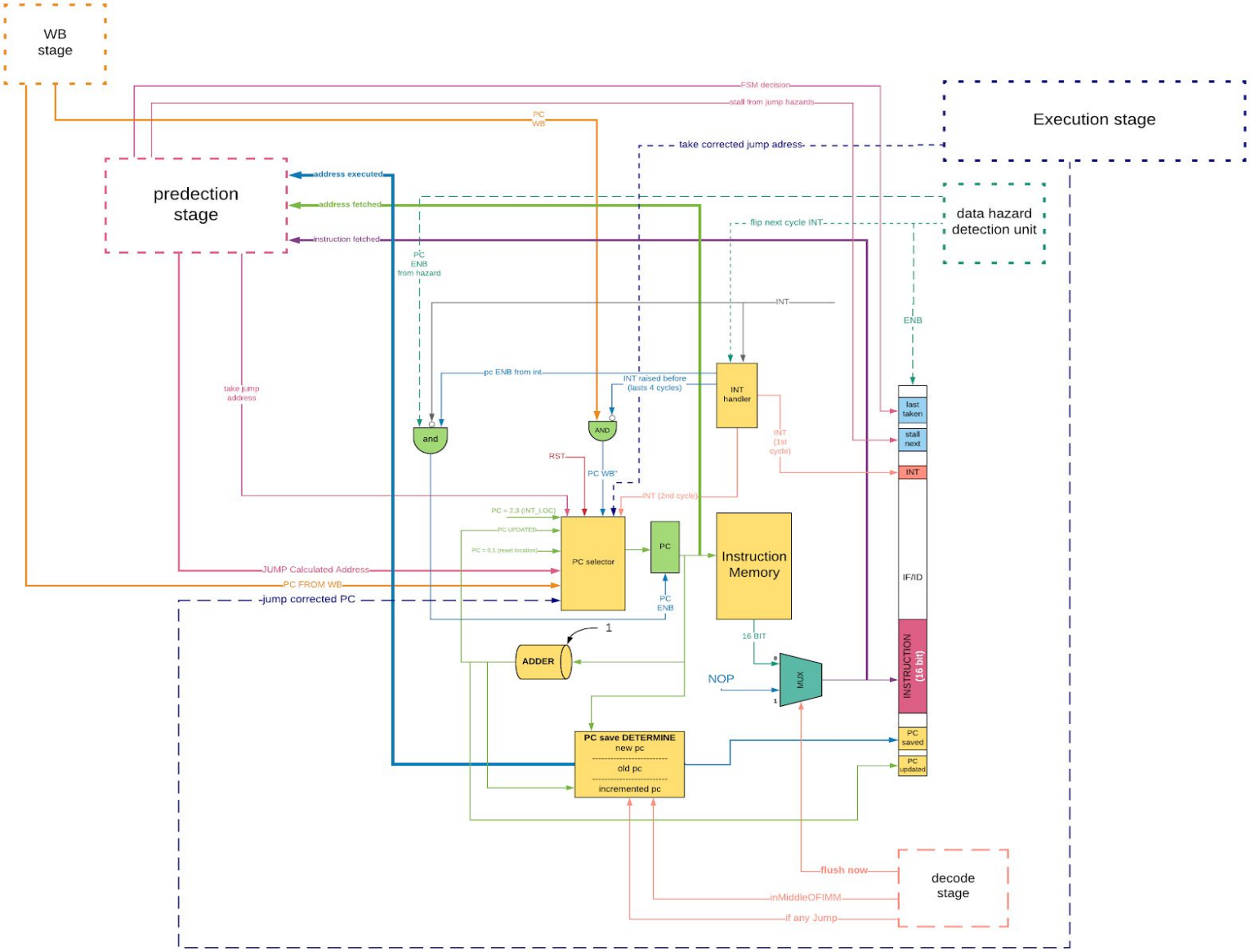
Forwards data in case of RTI/Call/Ret directly from memory to PC

1.4 FSM block :

Contains a standard 2 bits FSM with update input coming from decode state (comparison result) and input as “taken/not taken” decision

The state of FSM “the 2 bits” are distinct for every jump instruction and defined by the address of the instruction (address fetched & address in decode) and by having the address of the instruction FSM can recall the instruction 2 bits to make a decision or update its state

2. Fetch Stage :



Contains 4 internal blocks :

2.1 Instruction memory :

Contains compiled instructions as specified by assembler .. has 16-bit width and is implemented as an asynchronous read-only memory

2.2 PC selector :

Selects PC to enter the PC according to the following 1 -to- 1 mapping :

RST ➜ memory location {0,1}

INT (2nd cycle) ➜ memory location {2,3}

Take jump address ➜ Calculated jump address

Take corrected jump address ➜ jump corrected PC

PC wb ➜ PC from WB

Otherwise ➜ updated PC “from adder”

2.3 INT handler :

Responsible for handling interrupt correctly in the pipe with two cycles...

1st cycle ➜

- pass INT signal to decode to create control signals that let the memory save current pc and flags
- Stop pc from fetching new instruction (PC enable =0)

2nd cycle ➜

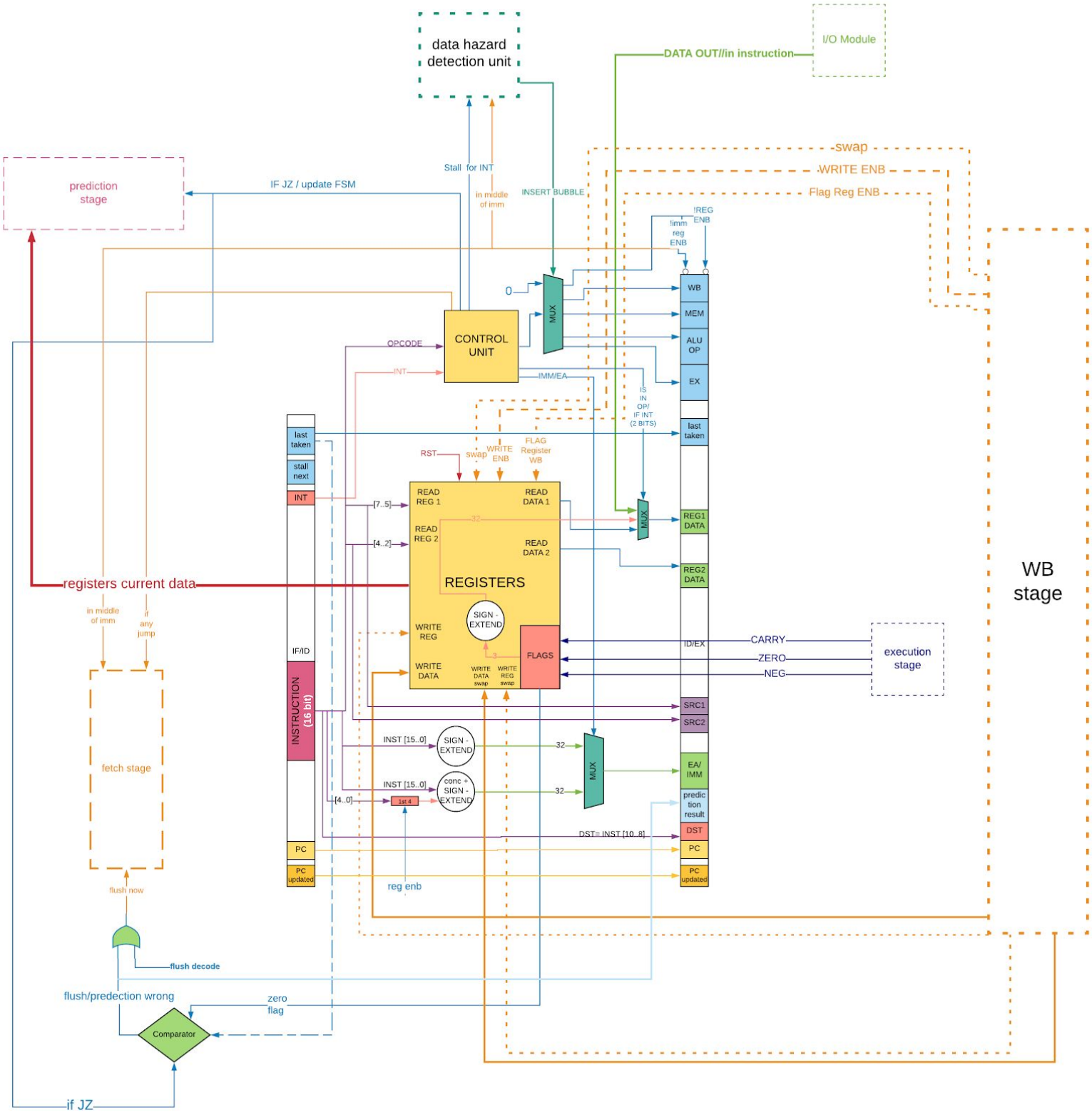
- PC will take memLocation (2,3} and fetch it
- PC wb will be disabled and keep like that for two more cycles preventing previous instructions before INT from writing in PC

2.4 PC save determine :

Contains a 64-bit shift register that keeps track for current PC alongside with the former one giving us the ability to choose which PC to be saved “propagate in the pipe till it reaches memory” .. in all cases PC saved should be the following PC to the current instruction “PC updated” except :

1. if we are in the middle of a two-words instruction ➜ second half is useless without the first one .. so I will save the old PC “the first part of the instruction” to refetch again at RTI
2. If we are in the middle of any jump instruction ➜ because we can’t risk saving wrongly-predicted PC before its correction

3. Decode Stage :



Contains 3 internal blocks :

3.1 Register file :

Contains 8 general-purpose 32-bit registers + 1 flag 3-bit register

Responsible for writing to correct register at rising edge of the clock due to a swap or write instruction

Responsible for reading the correct register at the falling edge of the clock “after being written to” according to src 1,2 in the IR of the instruction decoded

3.2 Control unit :

The core of the decode stage and the mastermind of the whole processor...

Responsible for decoding the instruction to a bunch of control signals used in this stage and propagates to following stages as well

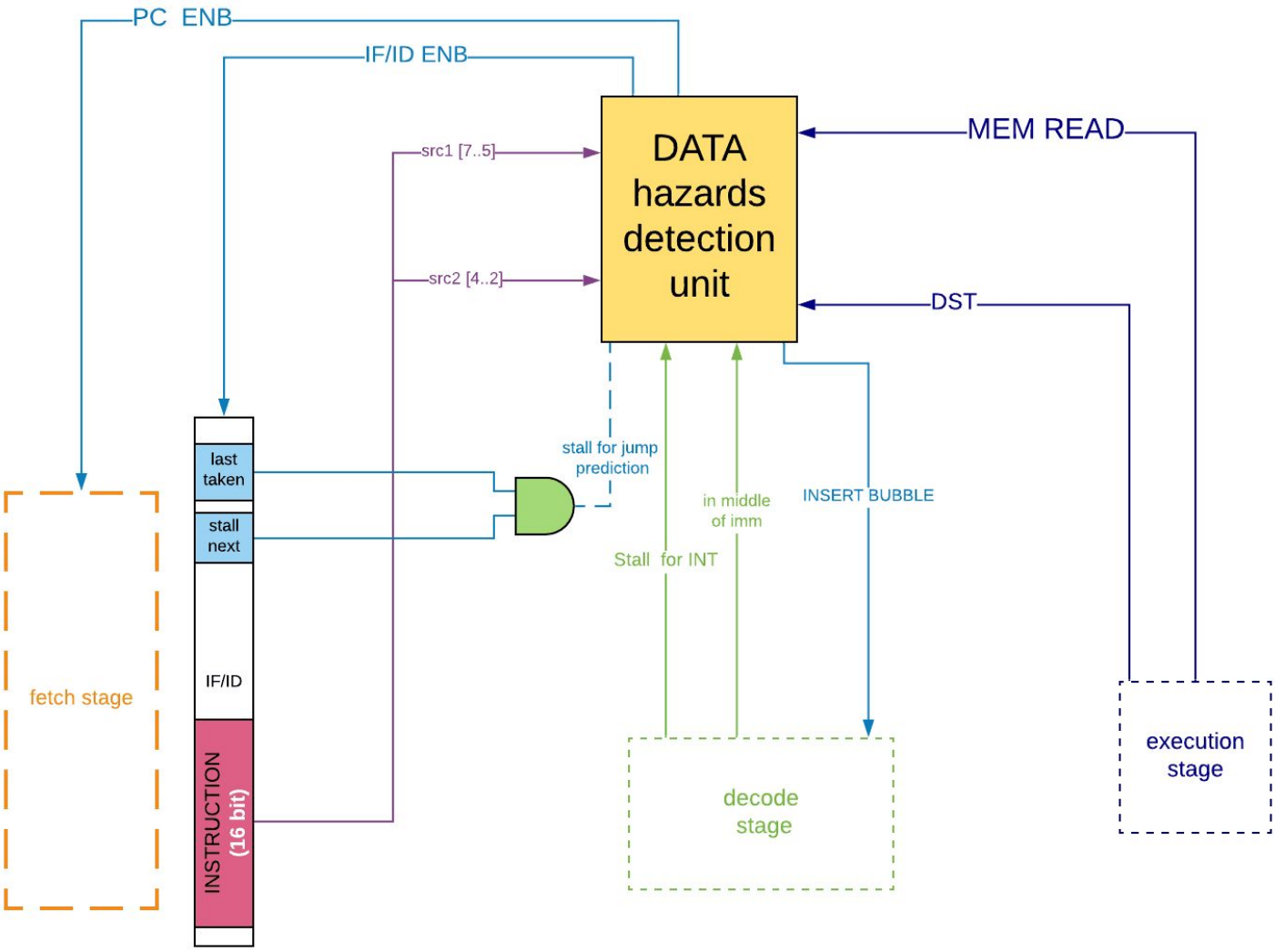
3.3 Comparator :

Compares between the zero flag and the prediction result in case of Jz instruction

If they are equal ➔ prediction is correct and nothing to do

If one is different than the other ➔ flush next instruction “wrongly fetched” and fetch stage take the corrected address in the next cycle

4. Data hazard detection unit :



Stalls the pipe through 3 tools and for 3 reasons :

Tools :

1. PC enb : freezes PC ➔ don't take a new value .. refetch the same PC again
2. IF/ID enb : freeze buffer and don't take the new instruction ➔ same instruction is kept in the buffer
3. Insert bubble : vanishes the current decoded instruction and flushes all control unit signals with zeros instead ➔ same as NOP

Reasons :

1. Stall for INT is risen ➔ used to stall the pipe to let INT and RTI uses the memory for two cycles without losing an instruction (for PC and flags) .. also used in cases of Call/Ret to stall the pipe until data is ready to be written back in pc
2. Stall from jump prediction is raised and the jump was taken ➔ data to jump into was not ready so we need to stall
3. Load use case in any instruction (except two-words instructions)➔ DST of previous instruction equals any of the sources of the current one + instruction executed require memory read .. so the data won't be available until next cycle

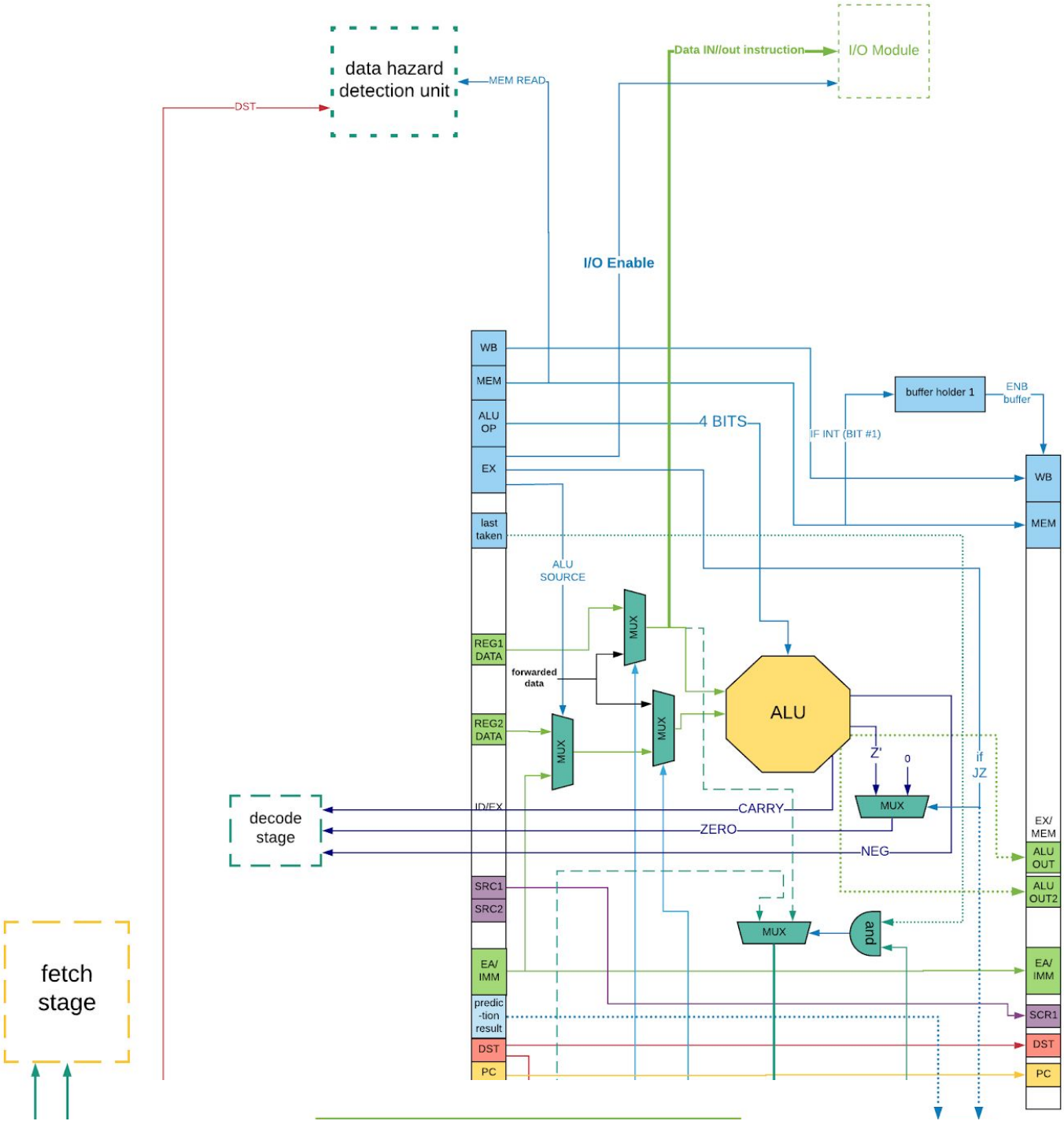
N.B. : two-words instructions don't need stalling as they are already done in two cycles if it has load use or hasn't

e.g. : pop R1 "memory operation"

LDI R1,5 "load use case"

↦ Don't cause hazards

5. Execution stage :



Contains 2 internal blocks :

5.1 ALU :

Do the main functionality of the operation through its 11 operations :

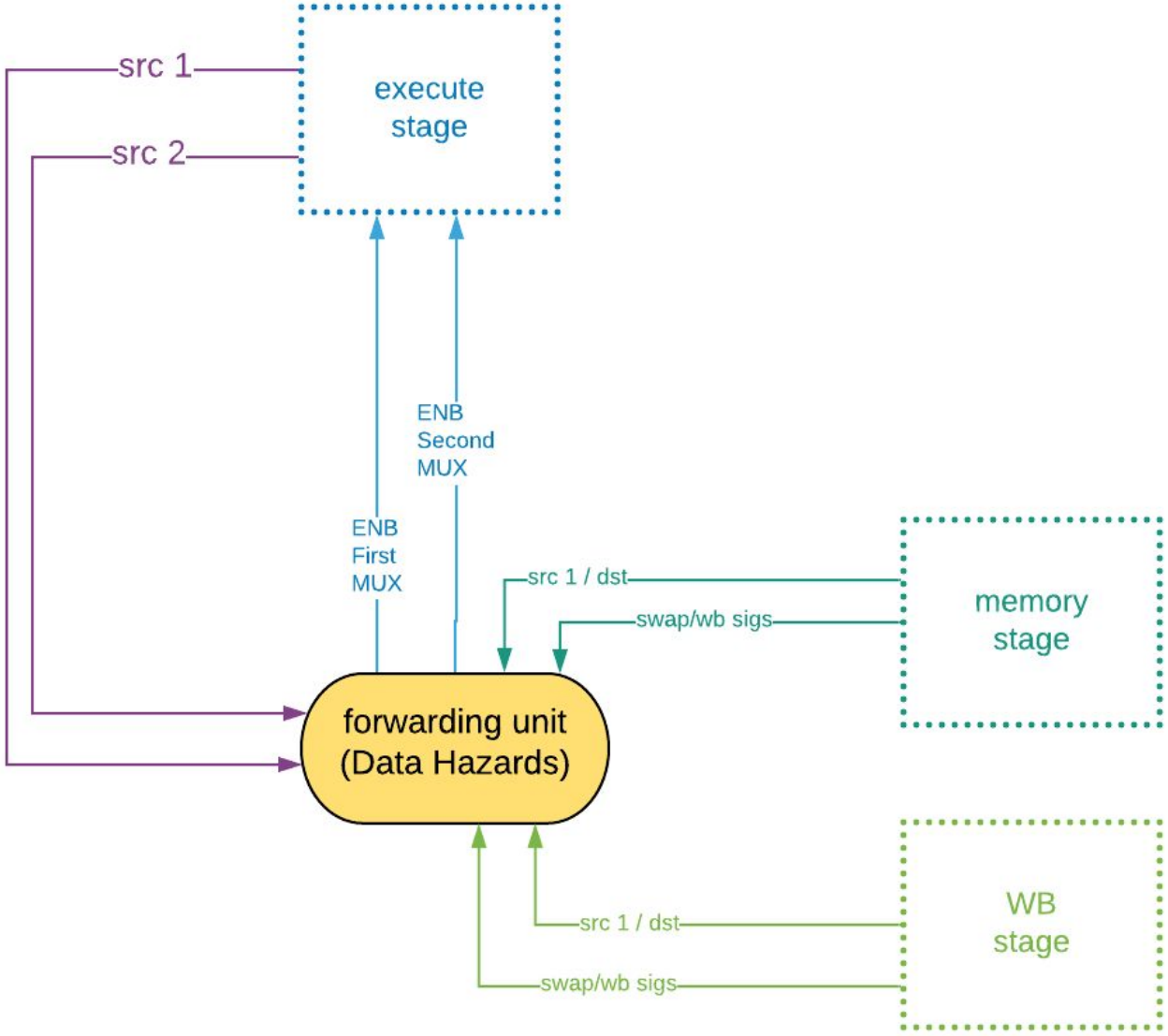
Add	Sub	And	OR
Swap	Inc	Dec	SHL
SHR	Not	Do nothing	

And outputs 2 outputs depending on the operation and 3 flags (neg, zero, carry) where they are forwarded directly to the register file from the ALU

5.2 Buffer Holder 1 :

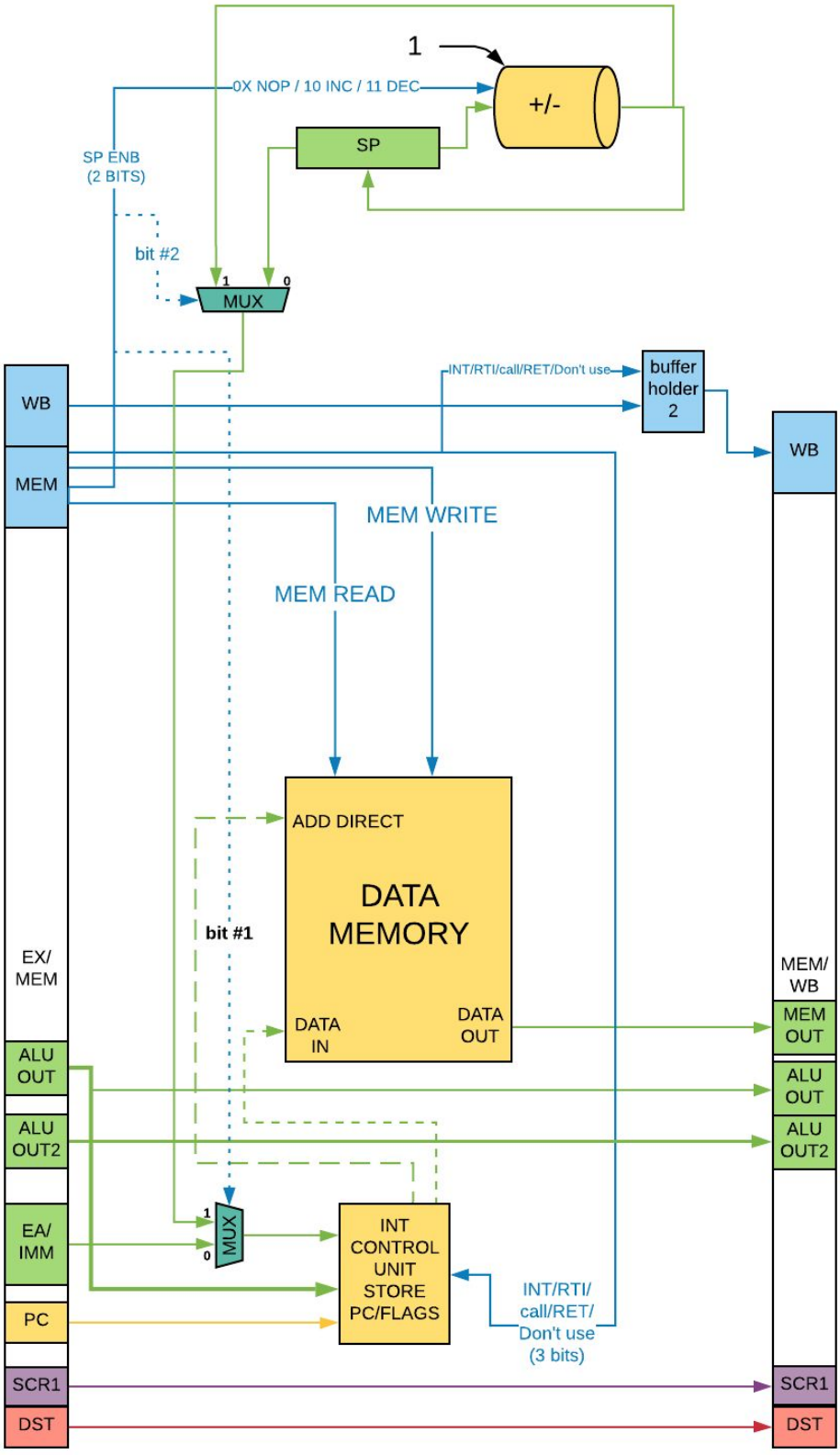
Used in case of INT/RTI to make the memory stage hold the control signals of the operation for two cycles .. enough to push/pop both PC and flags to/from memory

6. Forwarding unit :



Provides full forwarding for all ordinary and swap operations .. block provides forwarding from memory or ALU through two multiplexers in the execution stage to determine ALU source 1 and 2 locations and forward them or take the usual data from the register file

7. Memory stage :



Contains 4 internal blocks :

7.1 INT controller :

The gateway for memory to decide address and data

For Address : “ decided by a mux outside the block”

INT/RTI/Call/Ret ➜ always SP

Normal instructions ➜ EA/IMM

For Data :

INT ➜ ALU output “flags” then PC

Call ➜ PC

Normal instructions ➜ ALU output

Otherwise ➜ data is redundant as memory doesn’t write

7.2 Data memory :

The core of the memory stage ..

Has length 32-bit and writes on the rising edge of the clock and is read asynchronously .. stack part starts at address `X"00000FA0"` in memory

7.3 Buffer holder 2 :

Used in case of RTI to manipulate WB signals to write in PC in the first cycle and write to flags in the second cycle but doing nothing to the signals otherwise

7.4 SP incrementer/decrementer :

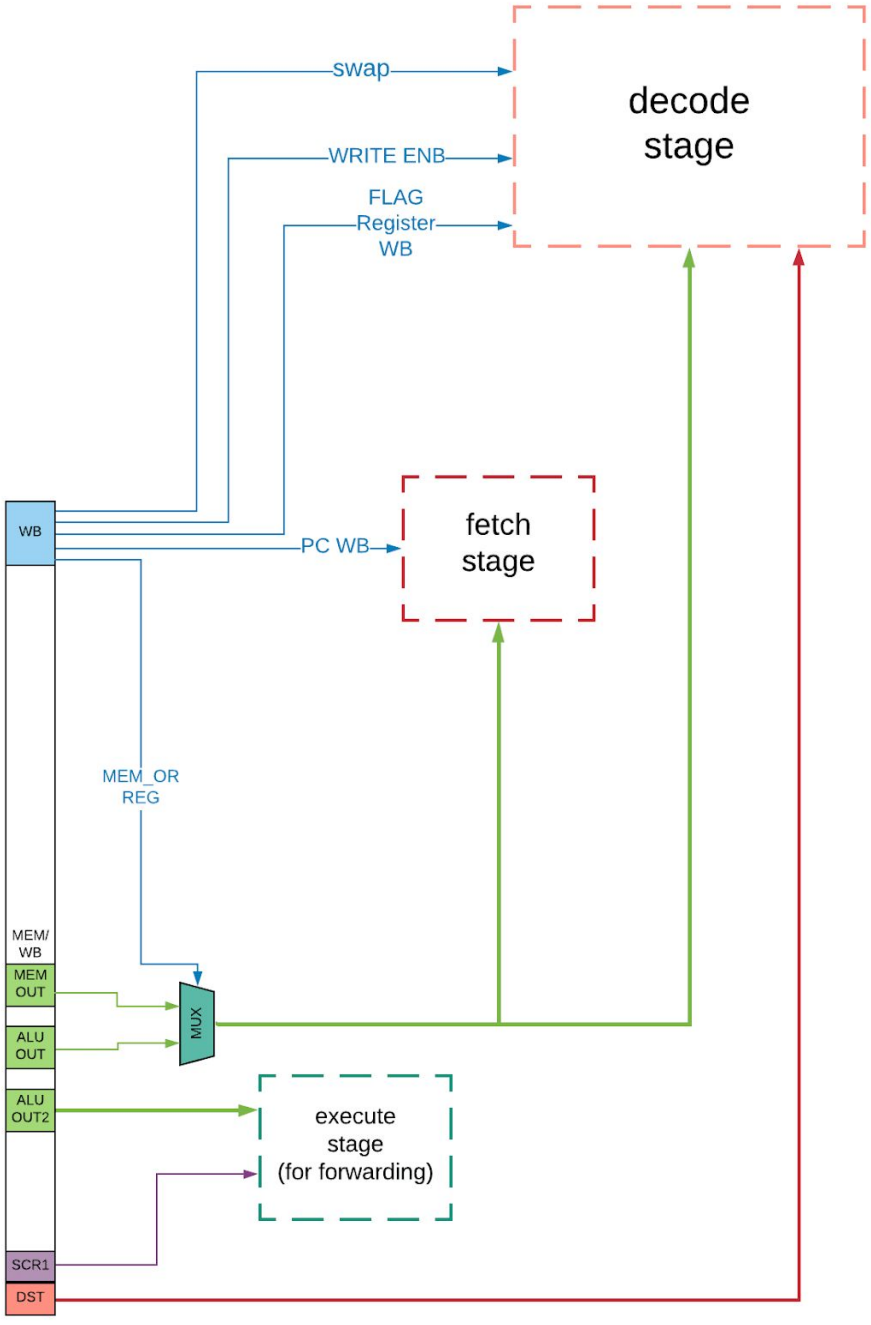
The register has its own incrementer/decrementer that works according to 2bit signal SP enb which also controls which result to take

11 ➜ as in POP/RTI/Ret .. decrement SP, take SP value after decrement, neglect EA/IMM as input for INT controller

10 ➜ as in Push/INT/Call .. increment SP, take SP value before increment, neglect EA/IMM as input for INT controller

0X ➜ for other instructions .. incrementer/decrementer do nothing, SP value is neglected as address memory operations

8. WB stage :



A combinational stage “not a real stage” .. passes output data and signals to all other stages and stages handle them by themselves

Output data to be written - in PC or register file - is either ALU output or memory output chosen according to WB signal “MEM OR REG” from the control unit

Not Implemented:

- Memory Cache system

Not Working properly:

- Data hazards in branch operations

- branch prediction

Test Cases Analysis:

In General, in order to get the right value, The value must be in WB stage,while the current instruction is at most in the decode stage**.**

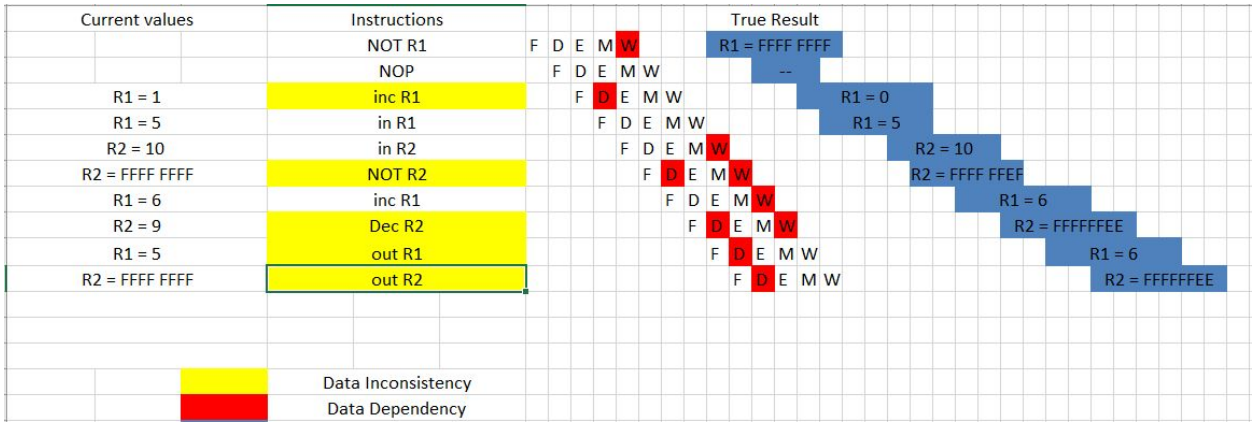
Our Design fetches any immediate or effective values in **2 clocks**, the first clock we fetch (half instruction [fetching Registers]), and the 2nd clock we forward the value (2nd half instruction) from fetch to decode.

Our Data Memory, Its width is 32bit so it takes **1 clock** to save in Memory, While Instruction Memory is still 16bit takes **2 clocks** to fetch any Imm or effective address.

Stack pointers, Starts at 0000FA0.

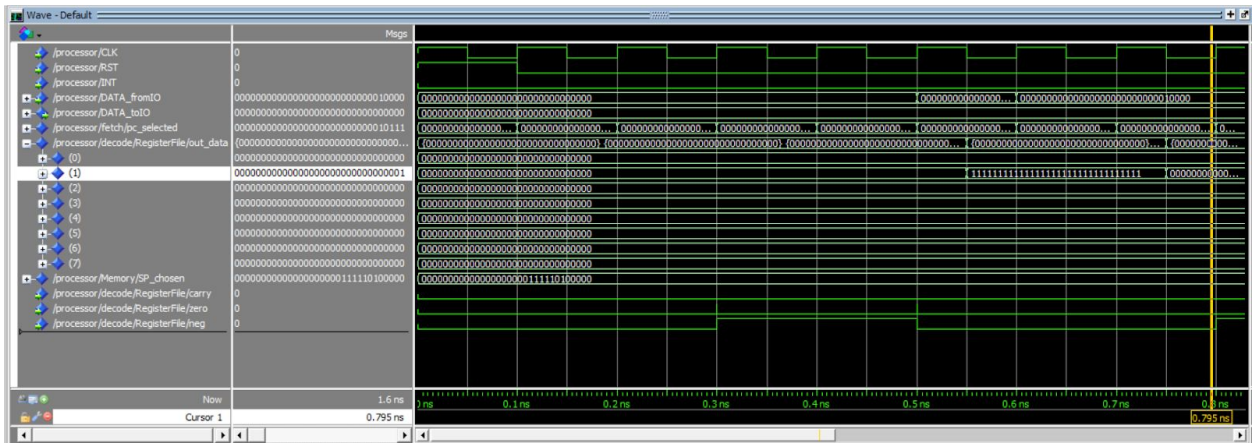
To make it easier to understand the solution easily, we made an excel sheet which provides steps for each clock, so by running the do file along with the excel sheet you get the expected value cycle by cycle.

1. **One Operand:**
    - **V1 (No Forwarding/Hazard Detection/Flushing):**

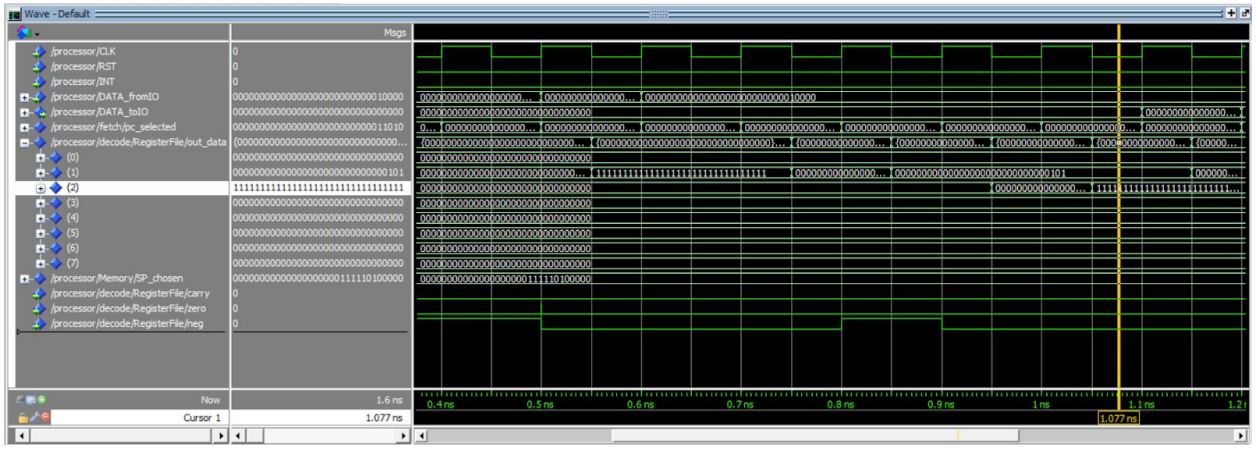


Hazards:

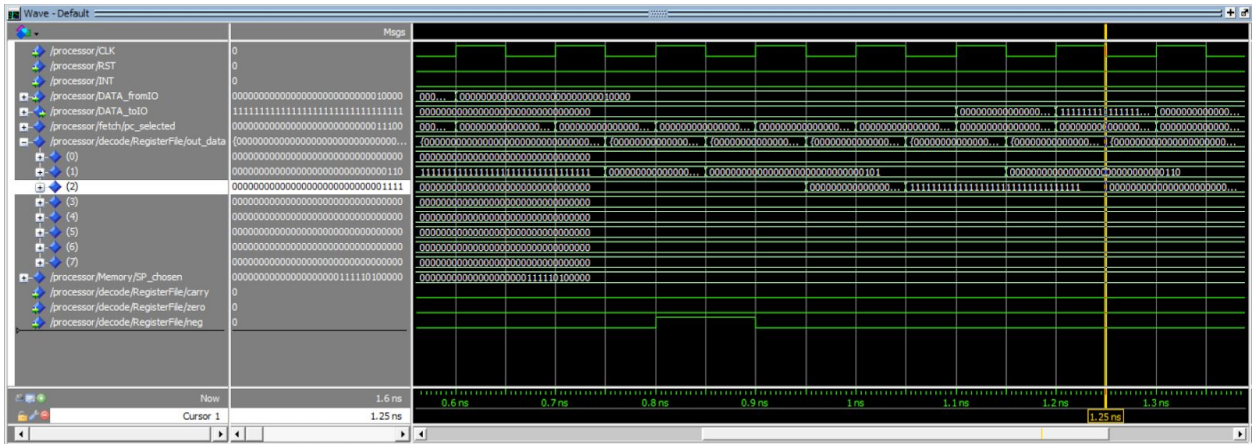
1. Inc R1: R1 takes the initial 0 value instead of FFFFF FFFFF as NOT R1 still didn't write back its value to make it 1 which is not right.



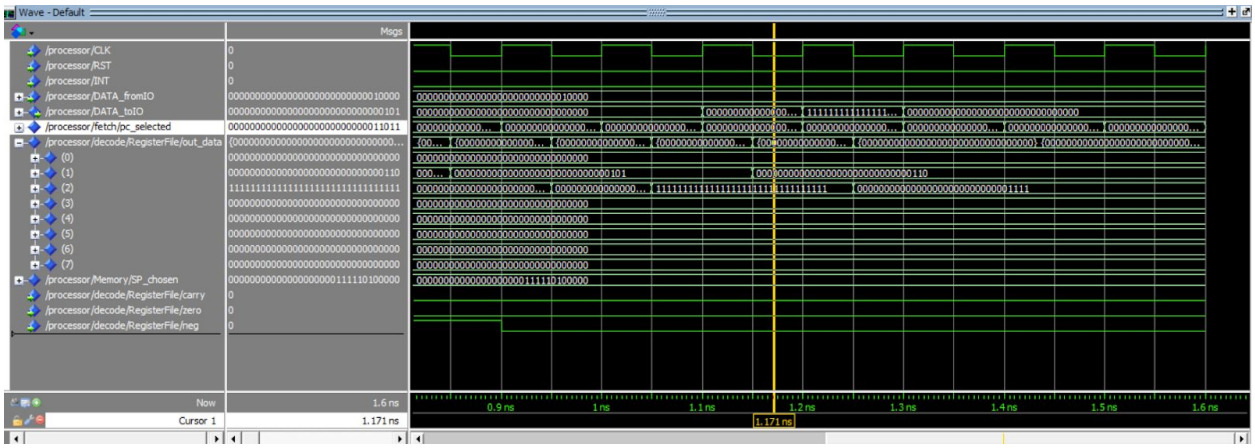
2. NOT R2: R2 takes the initial 0 value instead of 10 as in R2 still didn't write back its value to make it FFFF FFFF which is not right.



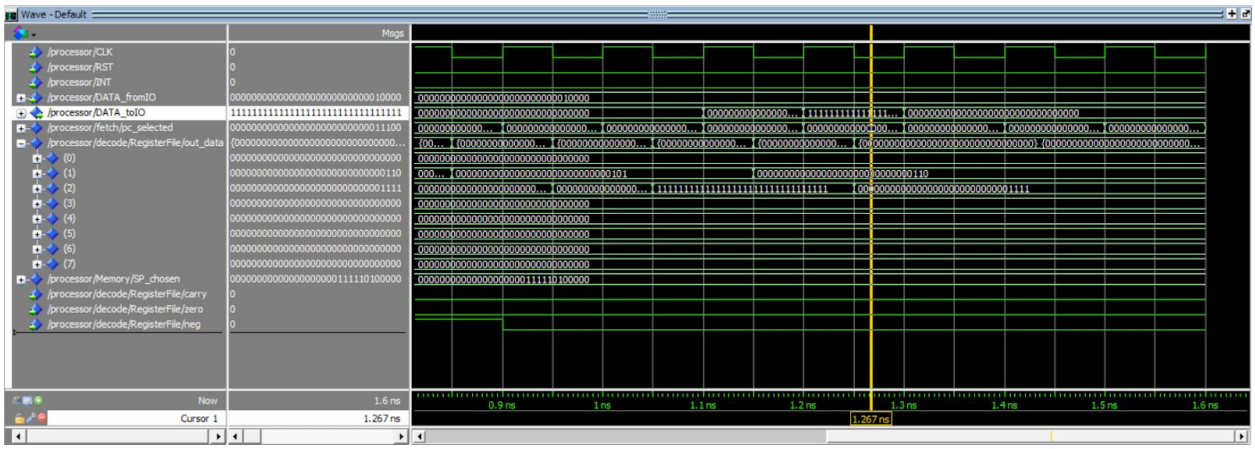
3. Dec R2: R2 takes the value of 10 which is not the correct output of NOT R2 to make it 9 which is not right.



4. Out R1: R1 takes the value of 5 which is before Incrementing it (6)

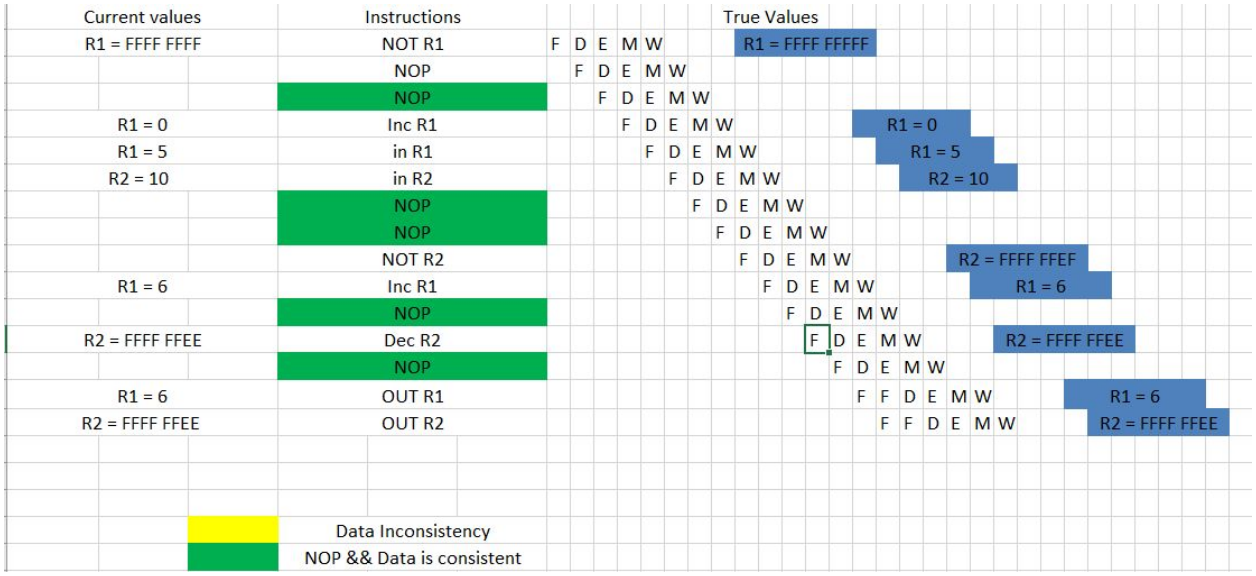


5. Out R2: R2 takes the value of 5 which is before Incrementing it (6)

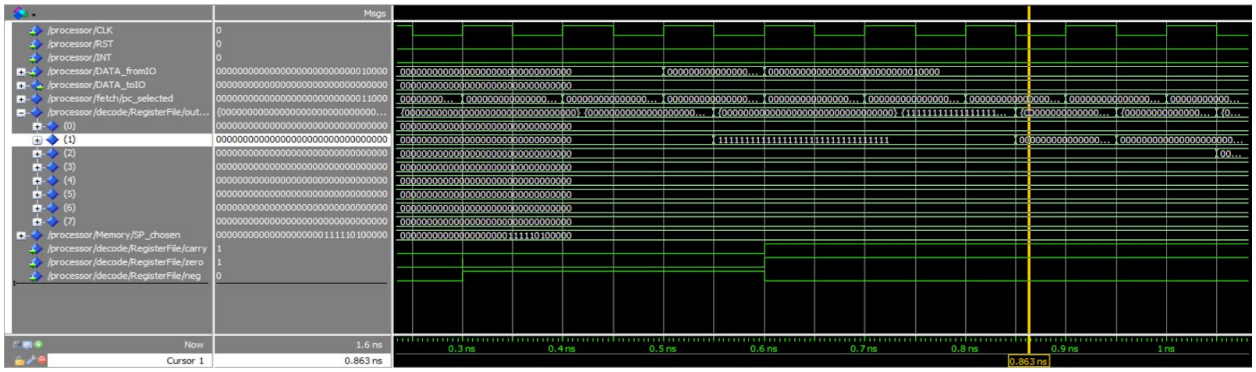


These Hazards can be solved by Adding No-operation before each Data Hazard to give it time to write back and then take the correct result in the hazardous operation.

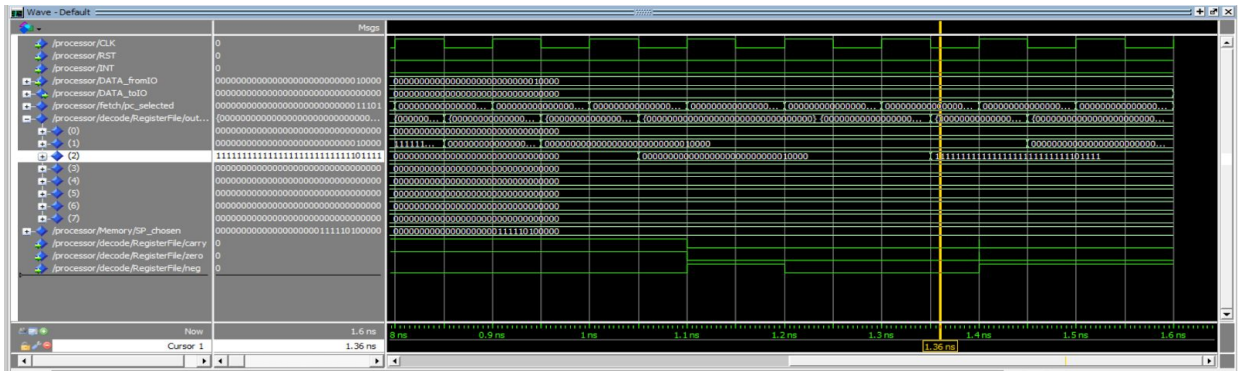
Solution:



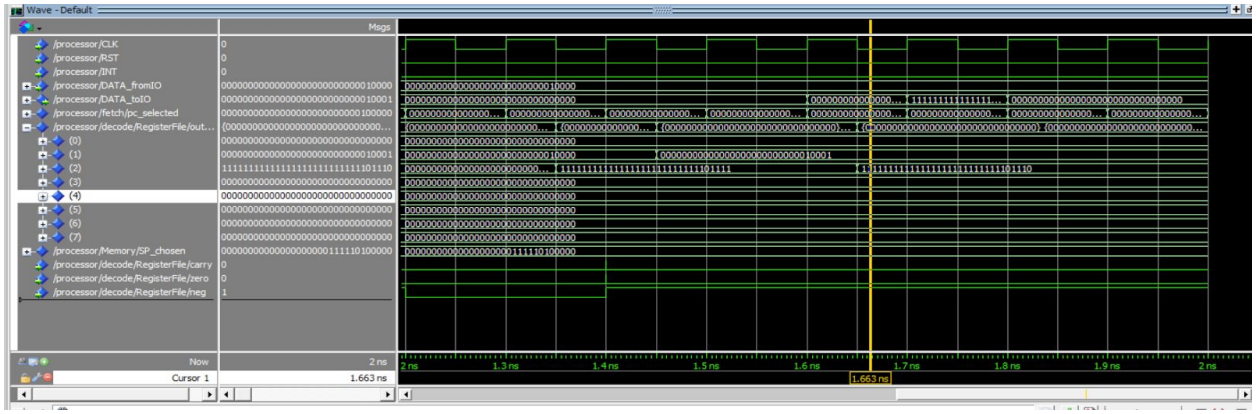
1. Inc R1:



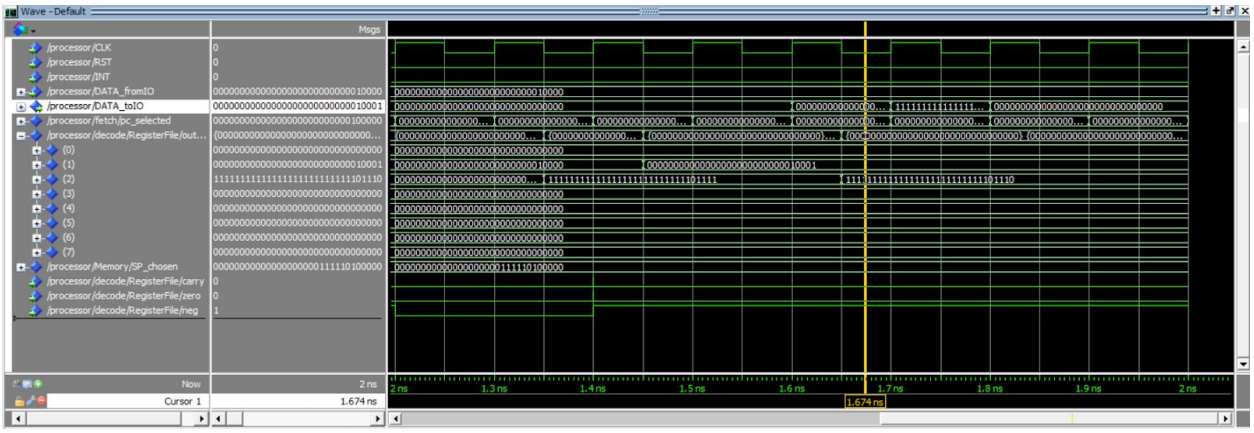
2. NOT R2:



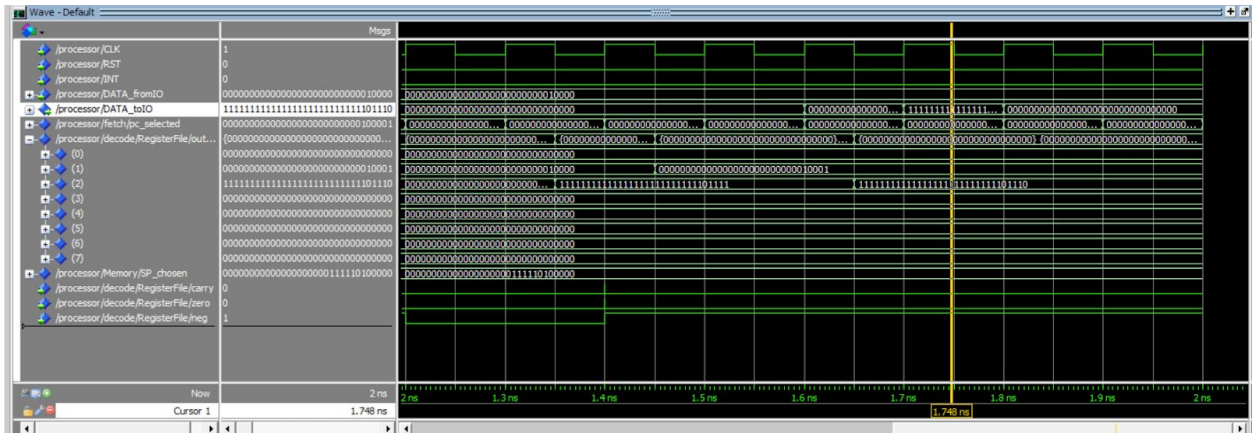
3. Dec R2:



4. Out R1:

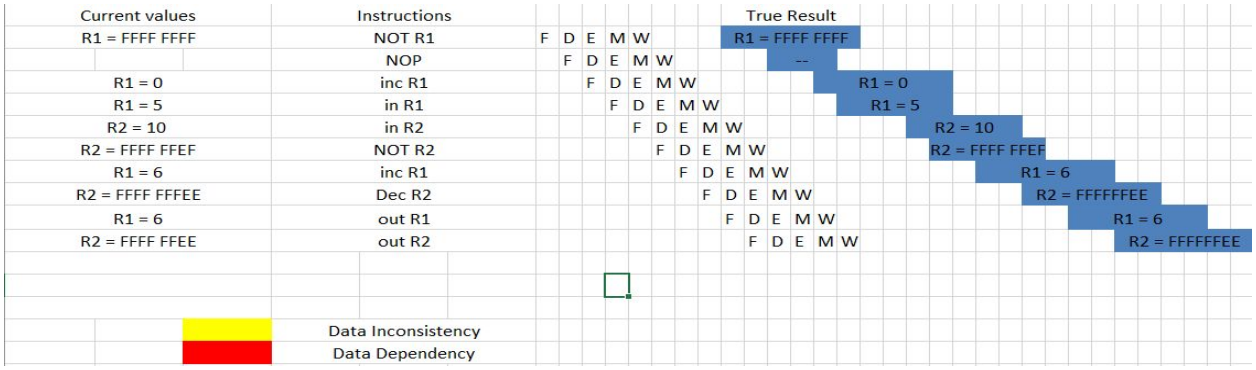


5. Out R2:

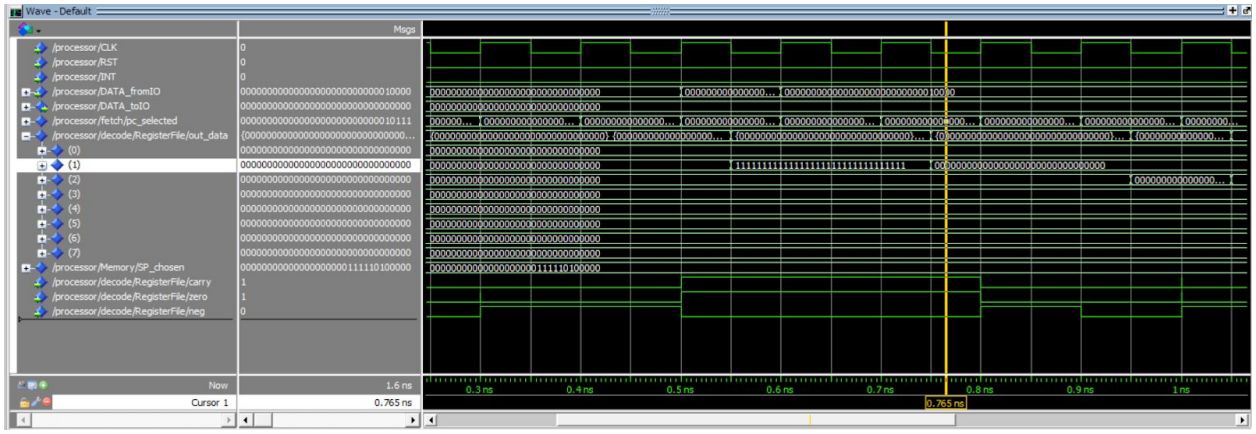


- **V2 (With Forwarding):**

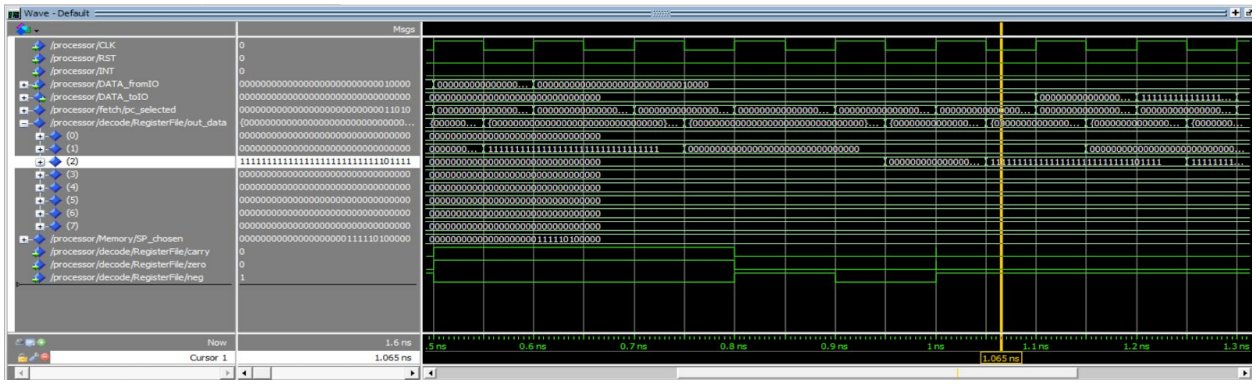
No Hazards appeared in the Test Case, and all outputs are right.



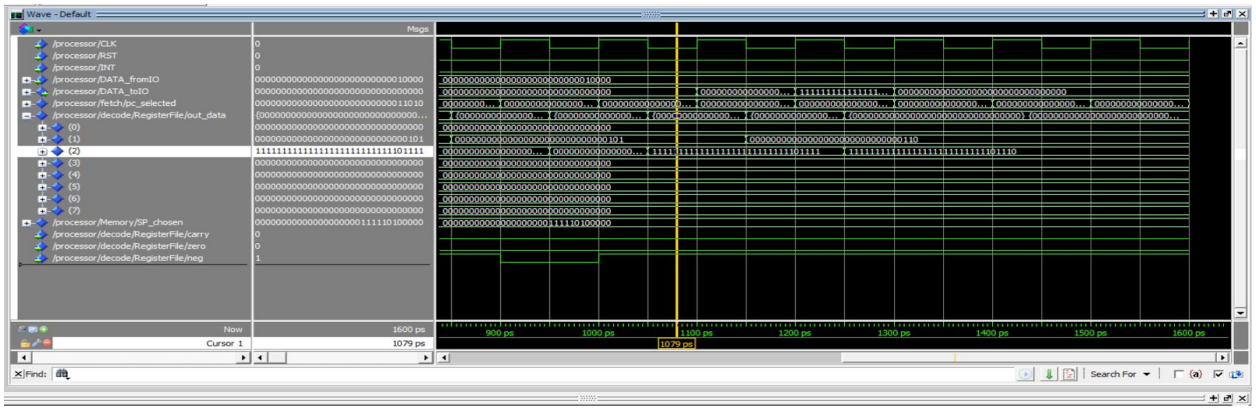
1. Inc R1:



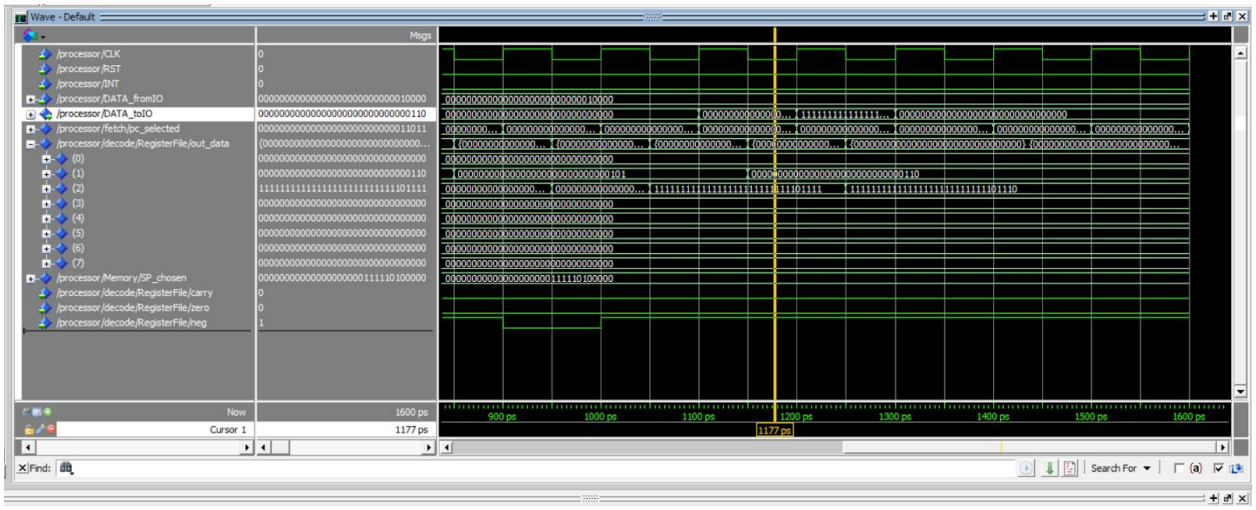
2. NOT R2:



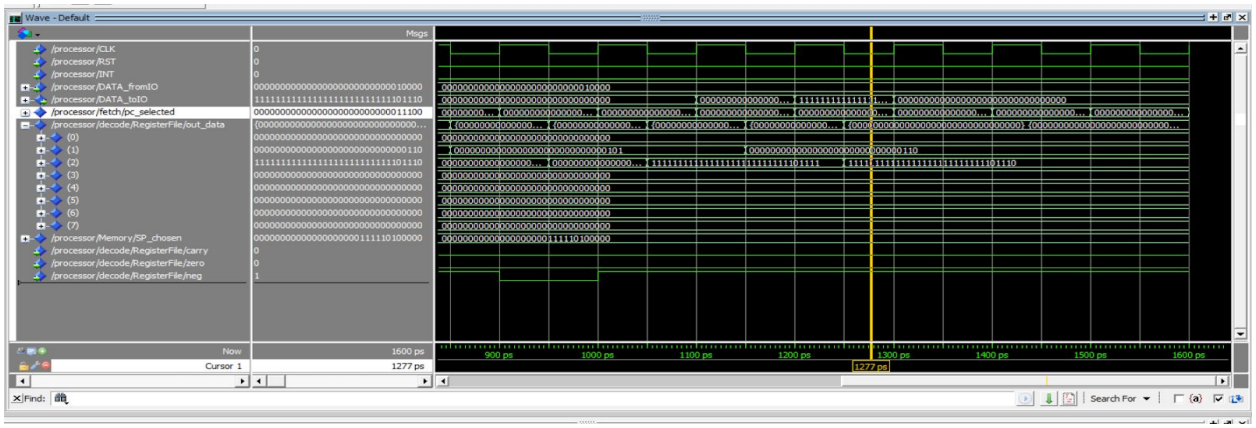
3. DEC R2:



4. OUT R1:



5. OUT R2:



- **V3 (With Forwarding/Hazard Detection):**

Similar results as V2

- **V4 (With Forwarding/Hazard Detection/Flushing):**

Similar results as V2

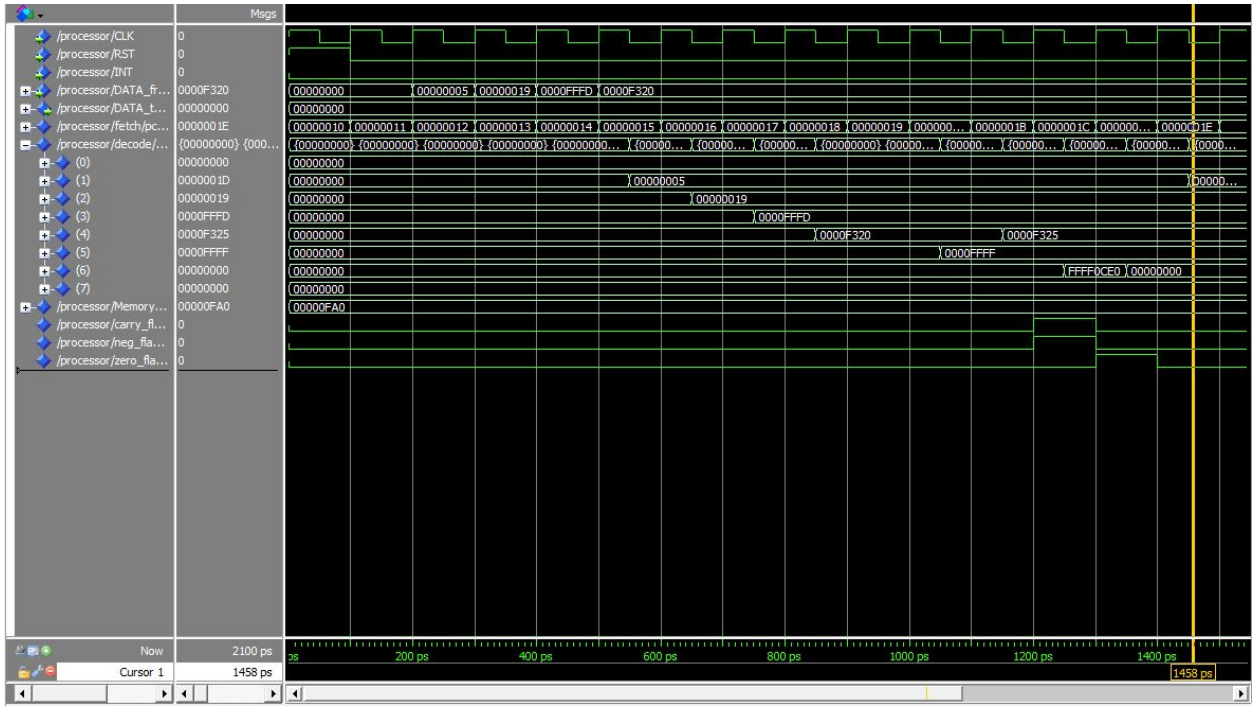
2. **Two Operands:**
    - **V1 (No Forwarding/Hazard Detection/Flushing):**

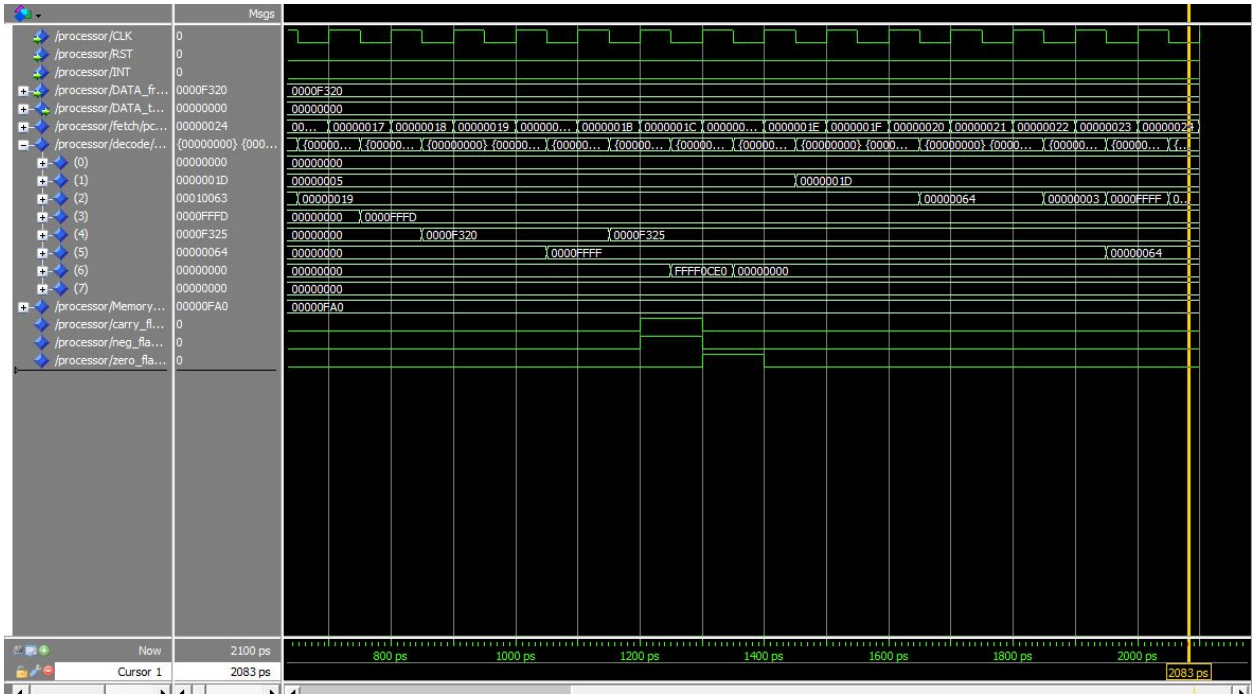
Hazards:

1. As we can see in the screen shot we found that the instruction SUB R5,R4,R6 , we need to get the right value of R4 and in order to do that Add R1,R4,R4 must be in WB stage, so we read a wrong value
2. Instruction SHL R2,3, data is read wrong as we can see in the screenshot, the decode comes earlier than WB
3. The last 2 instructions (SWAP R2,R5 and ADD R5,R2,R2) we also read wrong values and both happen as decode comes earlier than WB.

The Corresponding VHDL Sim:

- Executed Instructions: (in R1 -> OR R2,R1,R1)
- Executed Instructions: (SHL R2,2 -> end of instructions)

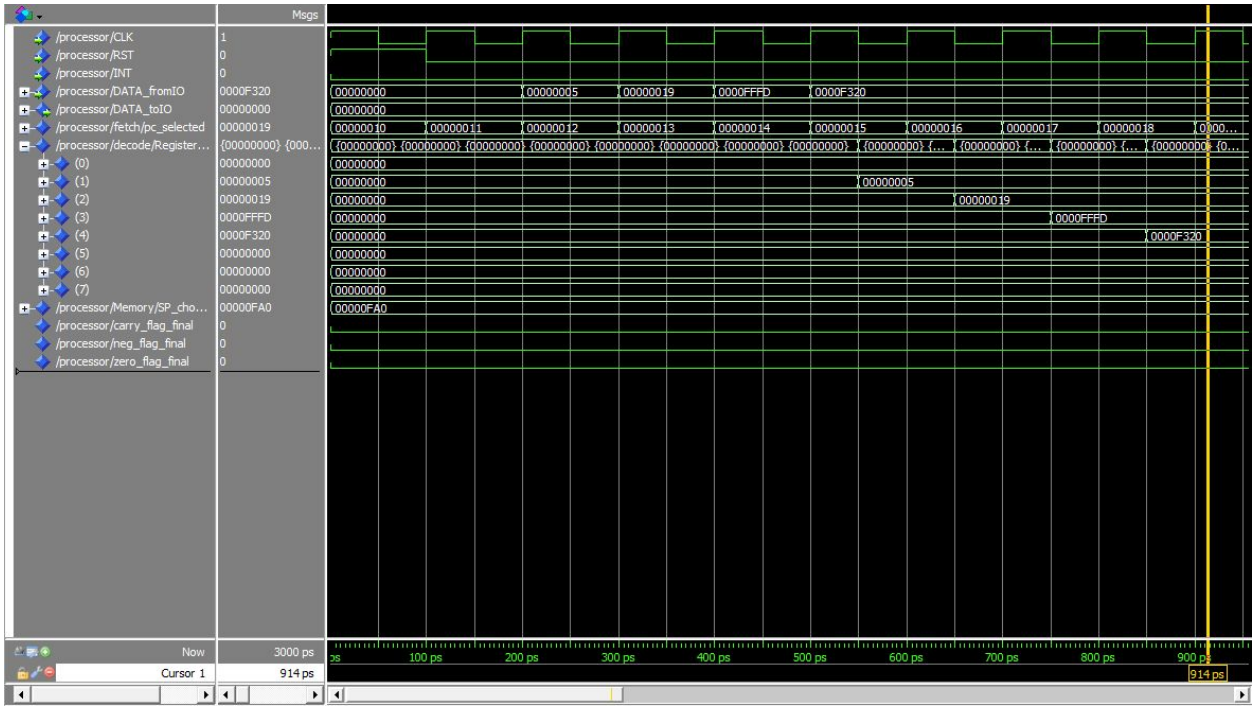

Msgs
/processor/CLK
/processor/RST
/processor/INT
/processor/DATA_fr...
/processor/DATA_t...
/processor/fetch/pc...
/processor/decode/...
(0)
(1)
(2)
(3)
(4)
(5)
(6)
(7)
/processor/Memory...
/processor/carry_fl...
/processor/neg_fla...
/processor/zero_fla...
Now
2100 ps
Cursor 1
1458 ps
200 ps
400 ps
600 ps
800 ps
1000 ps
1200 ps
1400 ps

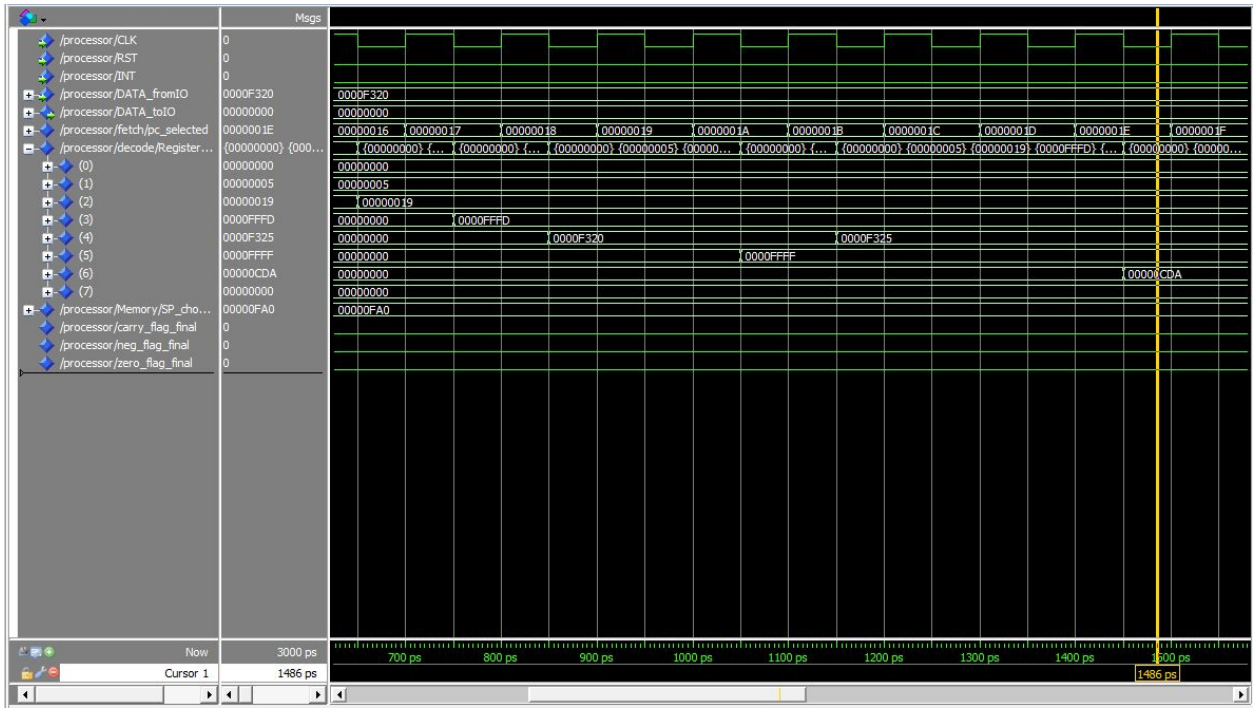

Msgs
/processor/CLK
/processor/RST
/processor/INT
/processor/DATA_fr...
/processor/DATA_t...
/processor/fetch/pc...
/processor/decode/...
(0)
(1)
(2)
(3)
(4)
(5)
(6)
(7)
/processor/Memory...
/processor/carry_fl...
/processor/neg_fla...
/processor/zero_fla...
Now
2100 ps
Cursor 1
2083 ps
800 ps
1000 ps
1200 ps
1400 ps
1600 ps
1800 ps
2000 ps

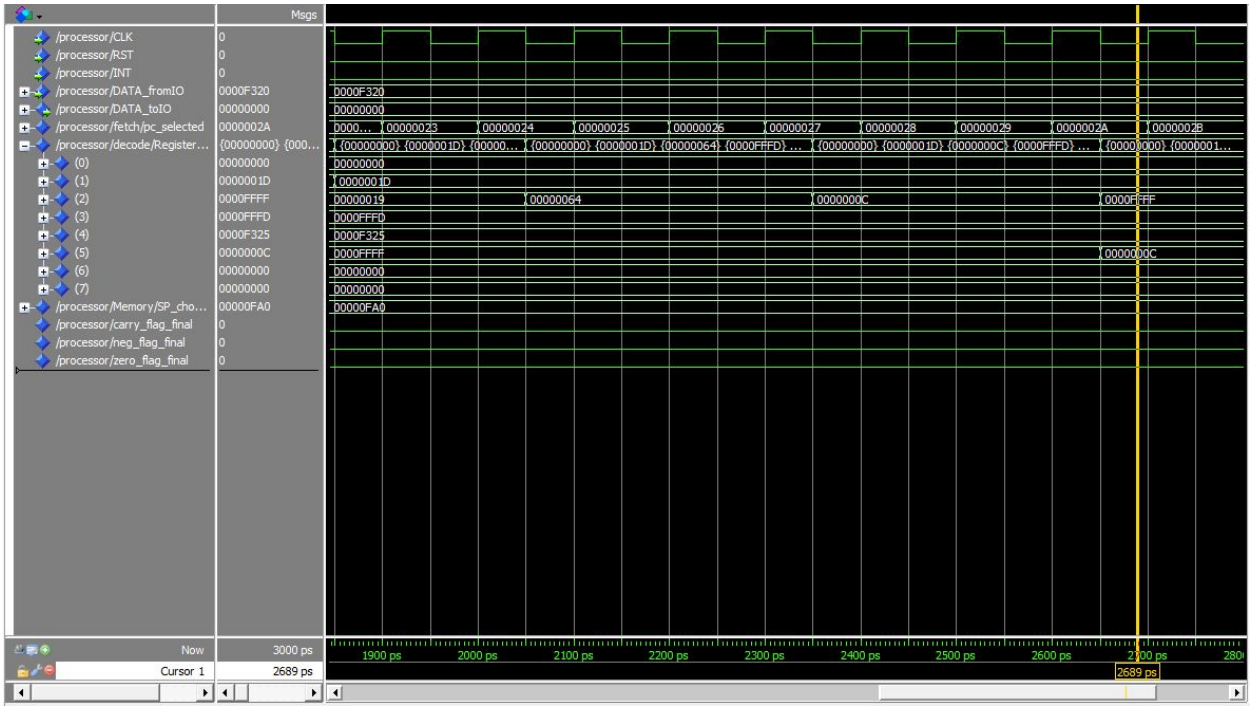
Solution:

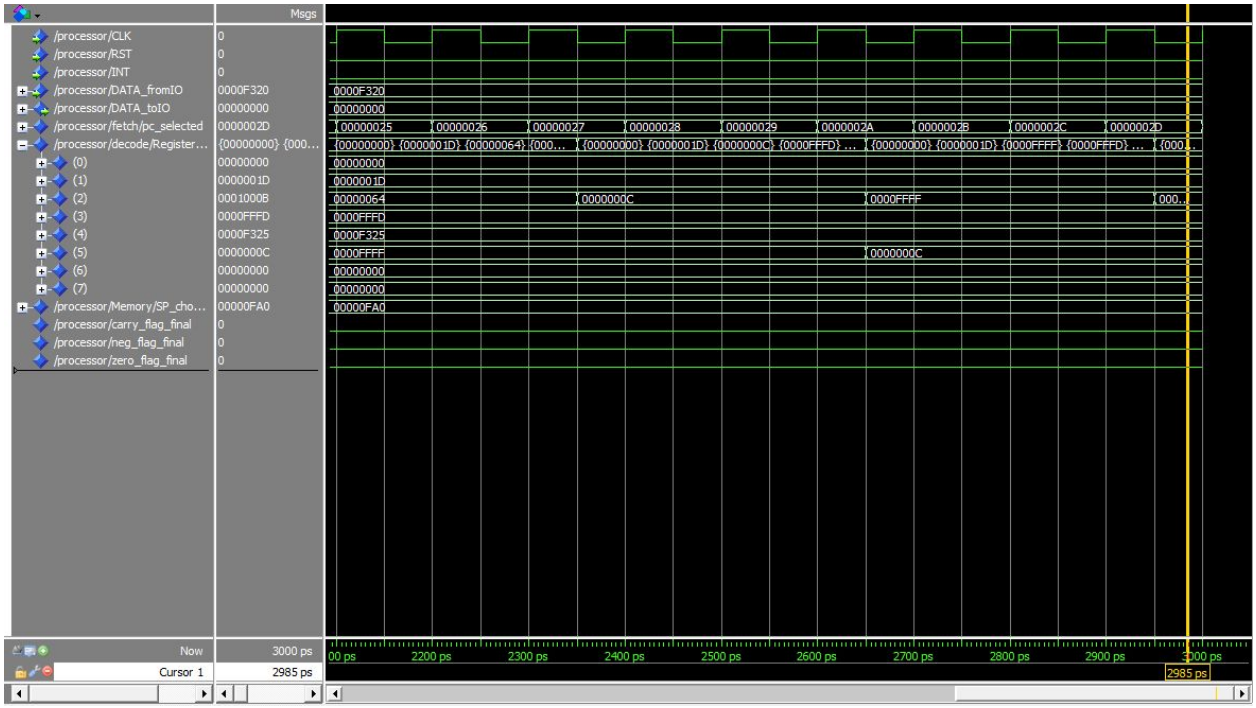
The Corresponding VHDL Sim:

- Executed Instructions: (in R1 -> in R4)
- Executed Instructions: (Iadd R3,R5,2 -> SUB R5,R4,R6)
- Executed Instructions: (NOP -> SWAP R2,R5)
- Executed Instructions: (NOP -> end of instructions)


Msgs
/processor/CLK
/processor/RST
/processor/INT
/processor/DATA_fromIO
/processor/DATA_toIO
/processor/fetch/pc_selected
/processor/decode/Register...
(0)
(1)
(2)
(3)
(4)
(5)
(6)
(7)
/processor/Memory/SP_cho...
/processor/carry_flag_final
/processor/neg_flag_final
/processor/zero_flag_final
Now
3000 ps
Cursor 1
914 ps
100 ps
200 ps
300 ps
400 ps
500 ps
600 ps
700 ps
800 ps
900 ps


Msgs
/processor/CLK
/processor/RST
/processor/INT
/processor/DATA_fromIO
/processor/DATA_toIO
/processor/fetch/pc_selected
/processor/decode/Register...
(0)
(1)
(2)
(3)
(4)
(5)
(6)
(7)
/processor/Memory/SP_cho...
/processor/carry_flag_final
/processor/neg_flag_final
/processor/zero_flag_final
Now
3000 ps
Cursor 1
1486 ps
700 ps
800 ps
900 ps
1000 ps
1100 ps
1200 ps
1300 ps
1400 ps
1486 ps





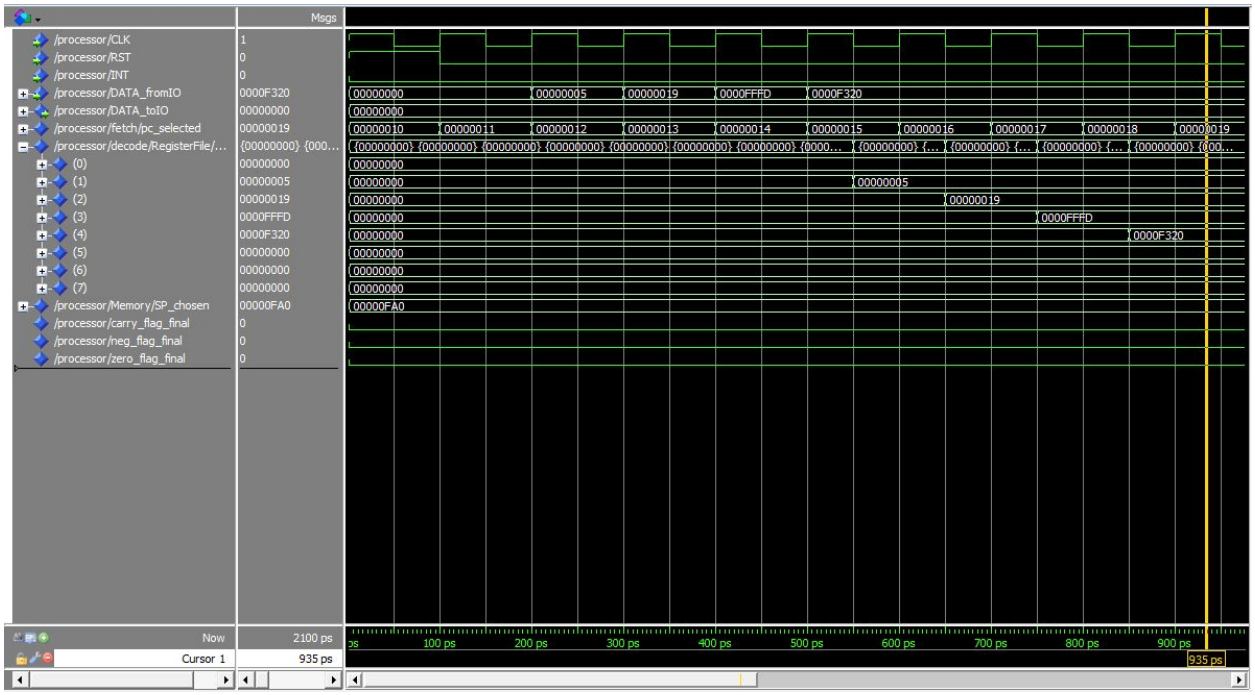
- **V2 (With Forwarding):**

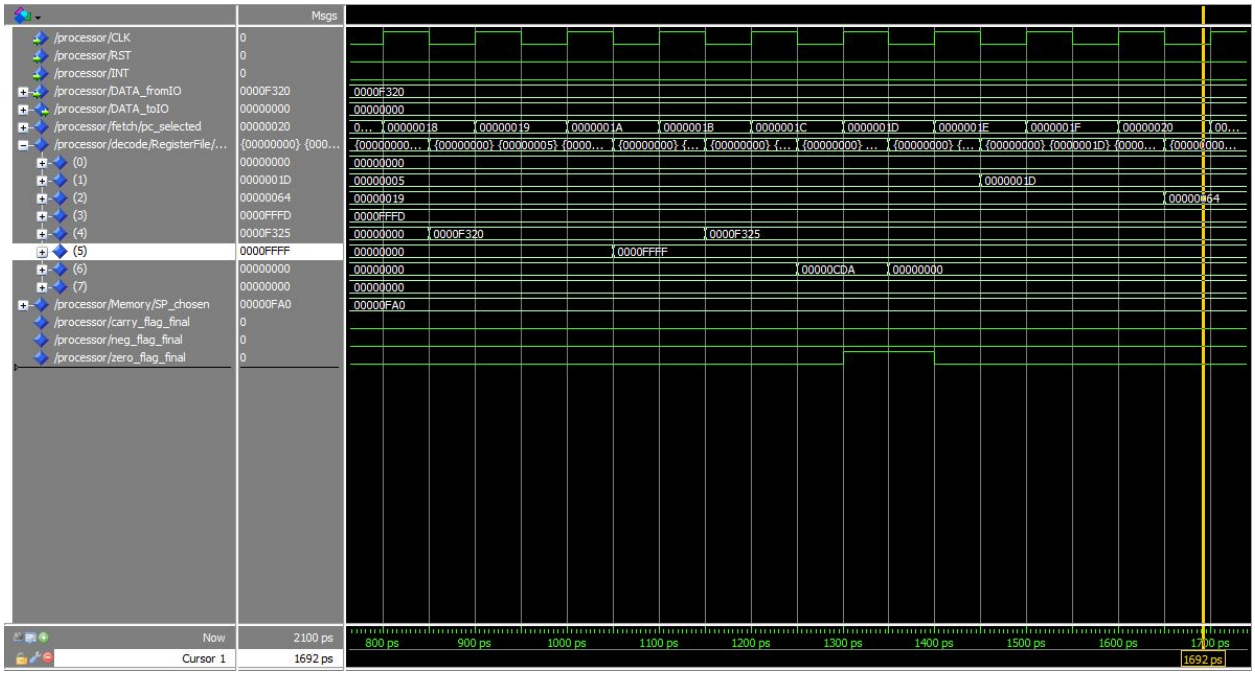
In the given test case, there are no Load use cases, so we will never need to stall as our forwarding unit forward from both ALU and Memory.

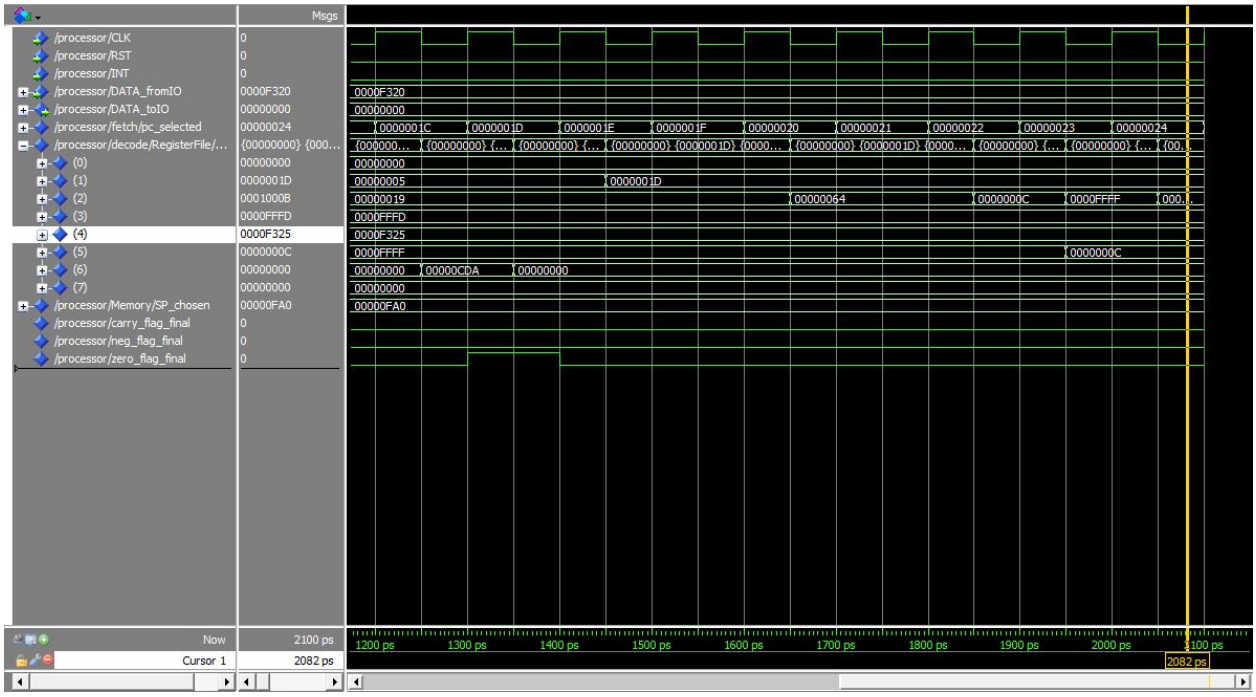
So this test case will work without any problem.

The Corresponding VHDL Sim:

- Executed Instructions: (in R1 -> in R4)
- Executed Instructions: (IADD R3,R5,2 -> SHL R2,2)
- Executed Instructions: (SHR R2,3 -> end of instructions)


Msgs
/processor/CLK
/processor/RST
/processor/INT
/processor/DATA_fromIO
/processor/DATA_toIO
/processor/fetch/pc_selected
/processor/decode/RegisterFile/...
(0)
(1)
(2)
(3)
(4)
(5)
(6)
(7)
/processor/Memory/SP_chosen
/processor/carry_flag_final
/processor/neg_flag_final
/processor/zero_flag_final
Now
2100 ps
Cursor 1
935 ps
100 ps
200 ps
300 ps
400 ps
500 ps
600 ps
700 ps
800 ps
900 ps


Msgs
/processor/CLK
/processor/RST
/processor/INT
/processor/DATA_fromIO
/processor/DATA_toIO
/processor/fetch/pc_selected
/processor/decode/RegisterFile/...
(0)
(1)
(2)
(3)
(4)
(5)
(6)
(7)
/processor/Memory/SP_chosen
/processor/carry_flag_final
/processor/neg_flag_final
/processor/zero_flag_final
Now
2100 ps
Cursor 1
1692 ps
800 ps
900 ps
1000 ps
1100 ps
1200 ps
1300 ps
1400 ps
1500 ps
1600 ps
1700 ps



- **V3 (With Forwarding/Hazard Detection):**

Similar results as V2

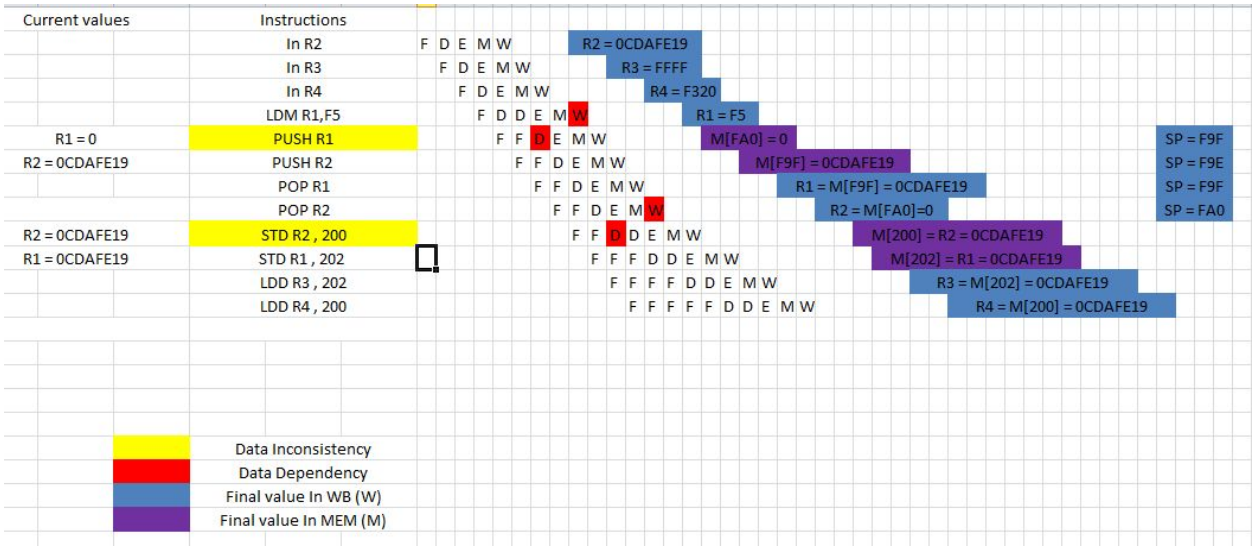
- **V4 (With Forwarding/Hazard Detection/Flushing):**

Similar results as V2

3. **Memory:**
    - **V1 (No Forwarding/Hazard Detection/Flushing):**

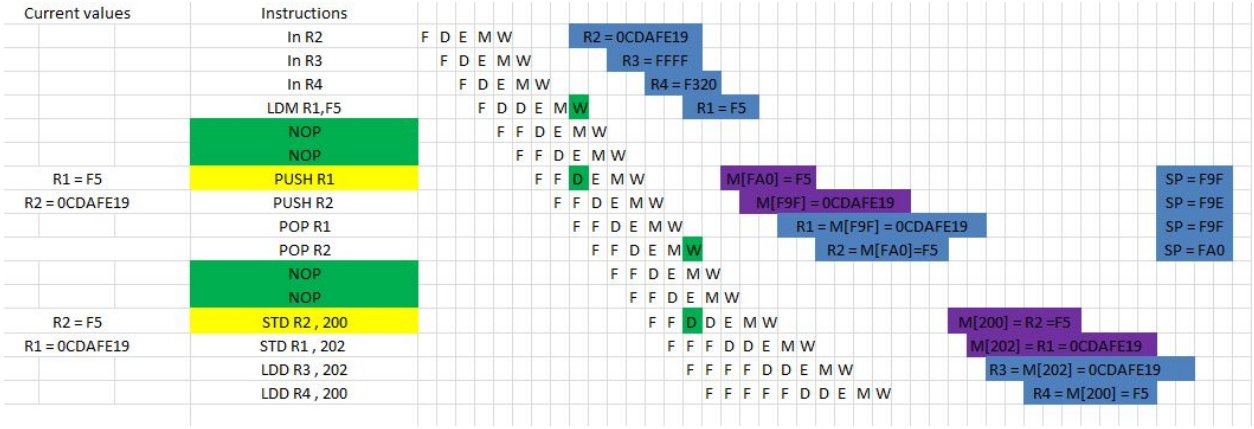
As we can see in the screen shot we found that, PUSH R1 instruction needs value of R1 to be F5 which is loaded in R1 from LDM R1,F5 . and this happens as decode comes earlier than WB.

Also STD R2, 200 instruction needs to get value from POP R2.

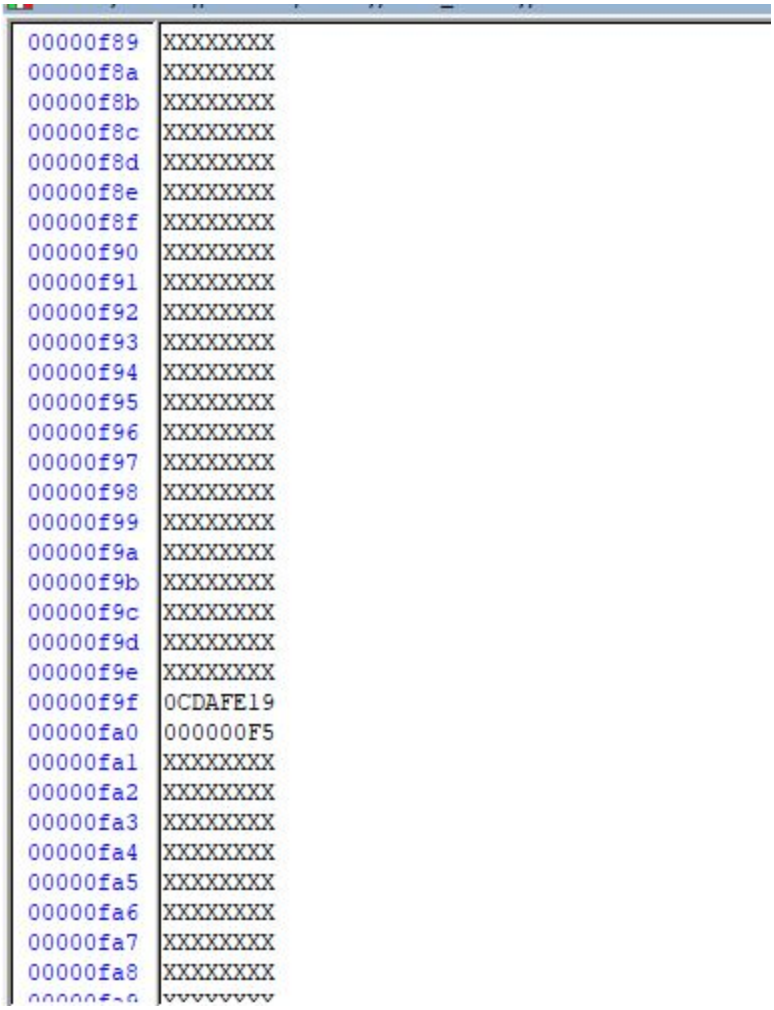


Memory View:

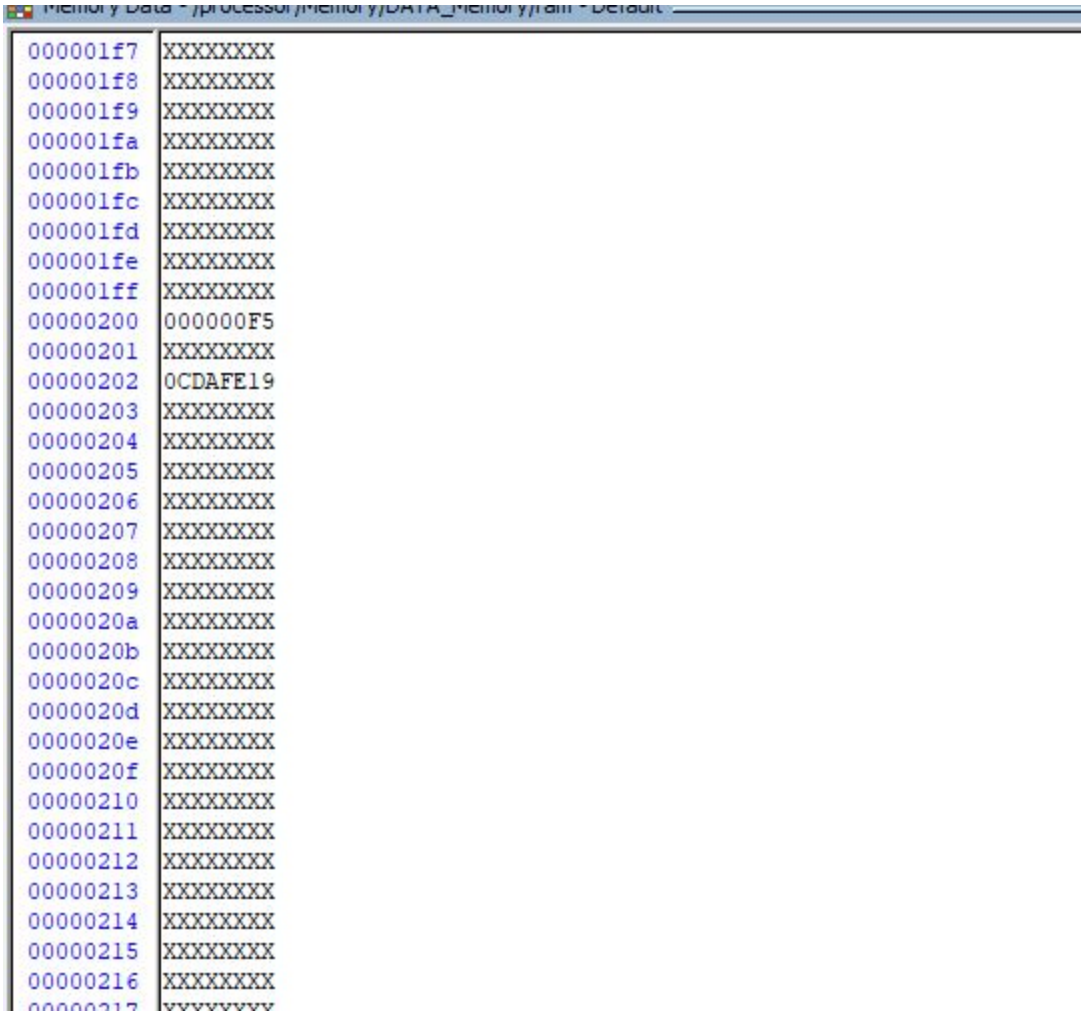
Solution:



Memory View:



00000f89	XXXXXXXX
00000f8a	XXXXXXXX
00000f8b	XXXXXXXX
00000f8c	XXXXXXXX
00000f8d	XXXXXXXX
00000f8e	XXXXXXXX
00000f8f	XXXXXXXX
00000f90	XXXXXXXX
00000f91	XXXXXXXX
00000f92	XXXXXXXX
00000f93	XXXXXXXX
00000f94	XXXXXXXX
00000f95	XXXXXXXX
00000f96	XXXXXXXX
00000f97	XXXXXXXX
00000f98	XXXXXXXX
00000f99	XXXXXXXX
00000f9a	XXXXXXXX
00000f9b	XXXXXXXX
00000f9c	XXXXXXXX
00000f9d	XXXXXXXX
00000f9e	XXXXXXXX
00000f9f	0CDAFE19
00000fa0	000000F5
00000fa1	XXXXXXXX
00000fa2	XXXXXXXX
00000fa3	XXXXXXXX
00000fa4	XXXXXXXX
00000fa5	XXXXXXXX
00000fa6	XXXXXXXX
00000fa7	XXXXXXXX
00000fa8	XXXXXXXX
00000fa9	XXXXXXXX



000001f7	XXXXXXXX
000001f8	XXXXXXXX
000001f9	XXXXXXXX
000001fa	XXXXXXXX
000001fb	XXXXXXXX
000001fc	XXXXXXXX
000001fd	XXXXXXXX
000001fe	XXXXXXXX
000001ff	XXXXXXXX
00000200	000000F5
00000201	XXXXXXXX
00000202	0CDAFE19
00000203	XXXXXXXX
00000204	XXXXXXXX
00000205	XXXXXXXX
00000206	XXXXXXXX
00000207	XXXXXXXX
00000208	XXXXXXXX
00000209	XXXXXXXX
0000020a	XXXXXXXX
0000020b	XXXXXXXX
0000020c	XXXXXXXX
0000020d	XXXXXXXX
0000020e	XXXXXXXX
0000020f	XXXXXXXX
00000210	XXXXXXXX
00000211	XXXXXXXX
00000212	XXXXXXXX
00000213	XXXXXXXX
00000214	XXXXXXXX
00000215	XXXXXXXX
00000216	XXXXXXXX

- **V2 (With Forwarding):**

For this Test case No hazard occurs as for Pop R2 and STD R2 , 200 because we fetch instruction in 2 cycles so we stall for 1 cycle in decode, and hazard detection unit is not tested but if the test case was POP R2 then Inc R2, then hazard detection will work, you can strong test cases in test case folder.

So this test case will work without any problem.

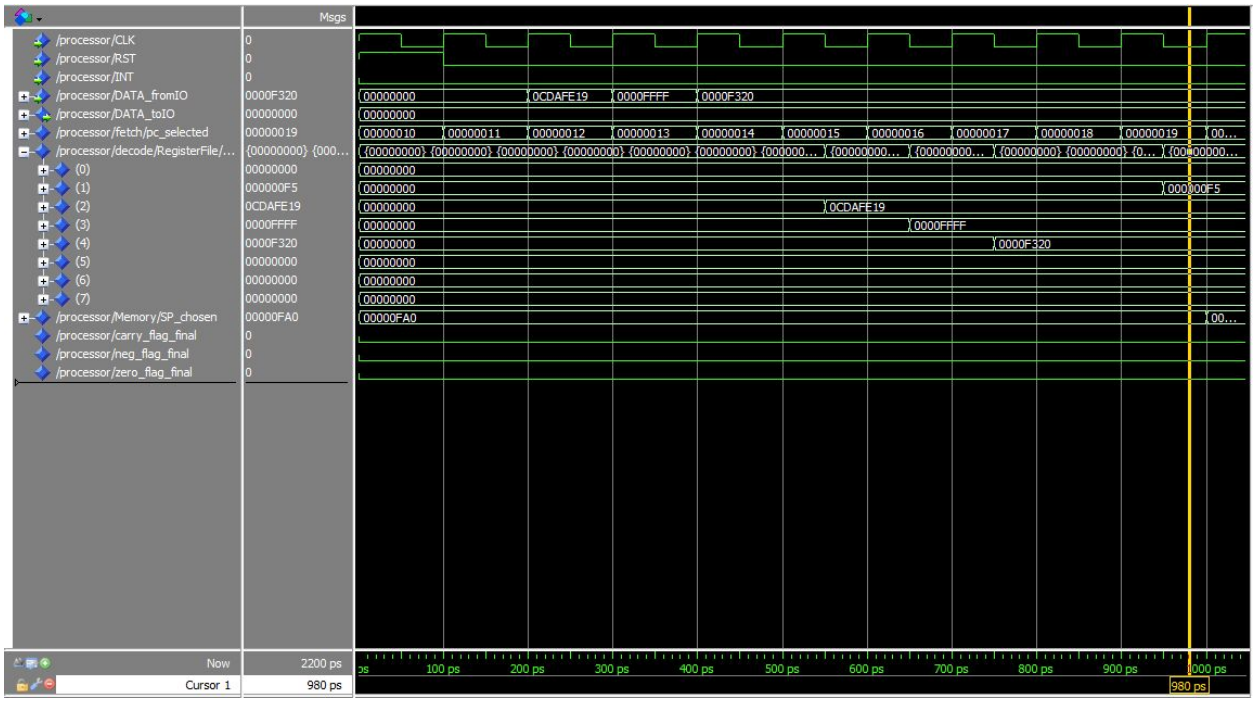
Memory View:

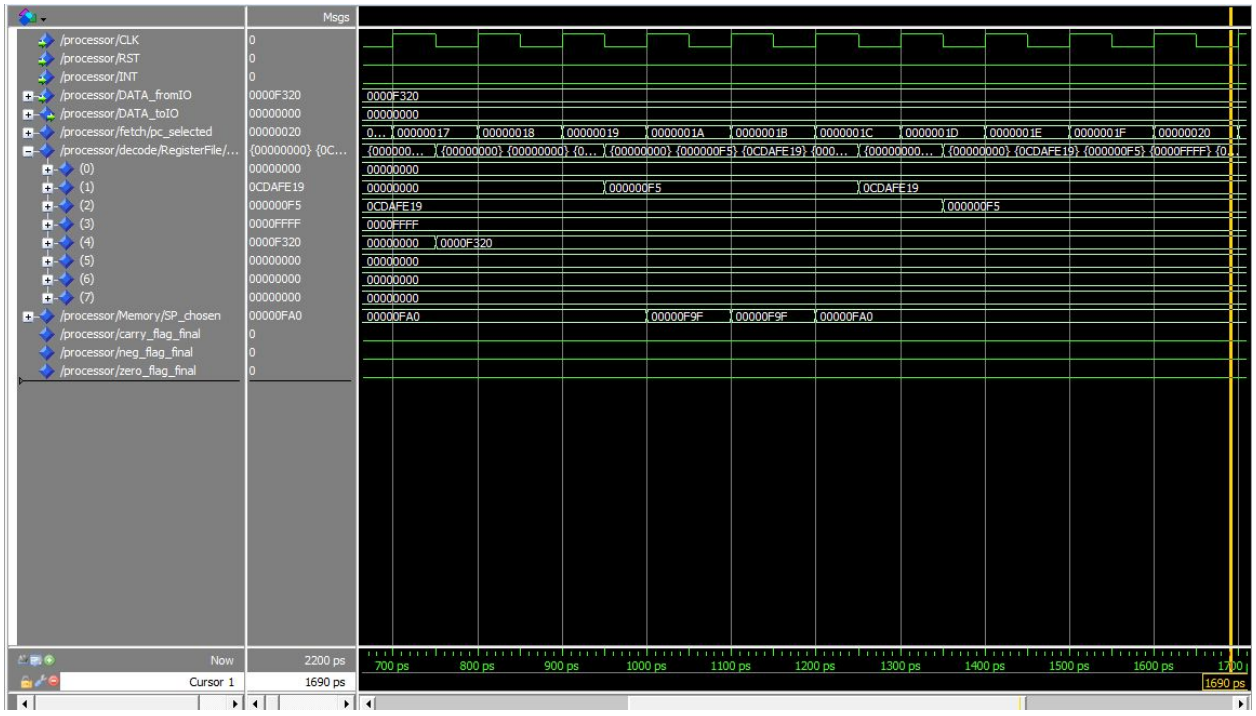
00000f93	XXXXXXXX
00000f94	XXXXXXXX
00000f95	XXXXXXXX
00000f96	XXXXXXXX
00000f97	XXXXXXXX
00000f98	XXXXXXXX
00000f99	XXXXXXXX
00000f9a	XXXXXXXX
00000f9b	XXXXXXXX
00000f9c	XXXXXXXX
00000f9d	XXXXXXXX
00000f9e	XXXXXXXX
00000f9f	0CDAFE19
00000fa0	000000F5
00000fa1	XXXXXXXX
00000fa2	XXXXXXXX
00000fa3	XXXXXXXX
00000fa4	XXXXXXXX
00000fa5	XXXXXXXX
00000fa6	XXXXXXXX
00000fa7	XXXXXXXX
00000fa8	XXXXXXXX
00000fa9	XXXXXXXX
00000faa	XXXXXXXX
00000fab	XXXXXXXX
00000fac	XXXXXXXX
00000fad	XXXXXXXX
00000fae	XXXXXXXX
00000faf	XXXXXXXX
00000fb0	XXXXXXXX
00000fb1	XXXXXXXX
00000fb2	XXXXXXXX

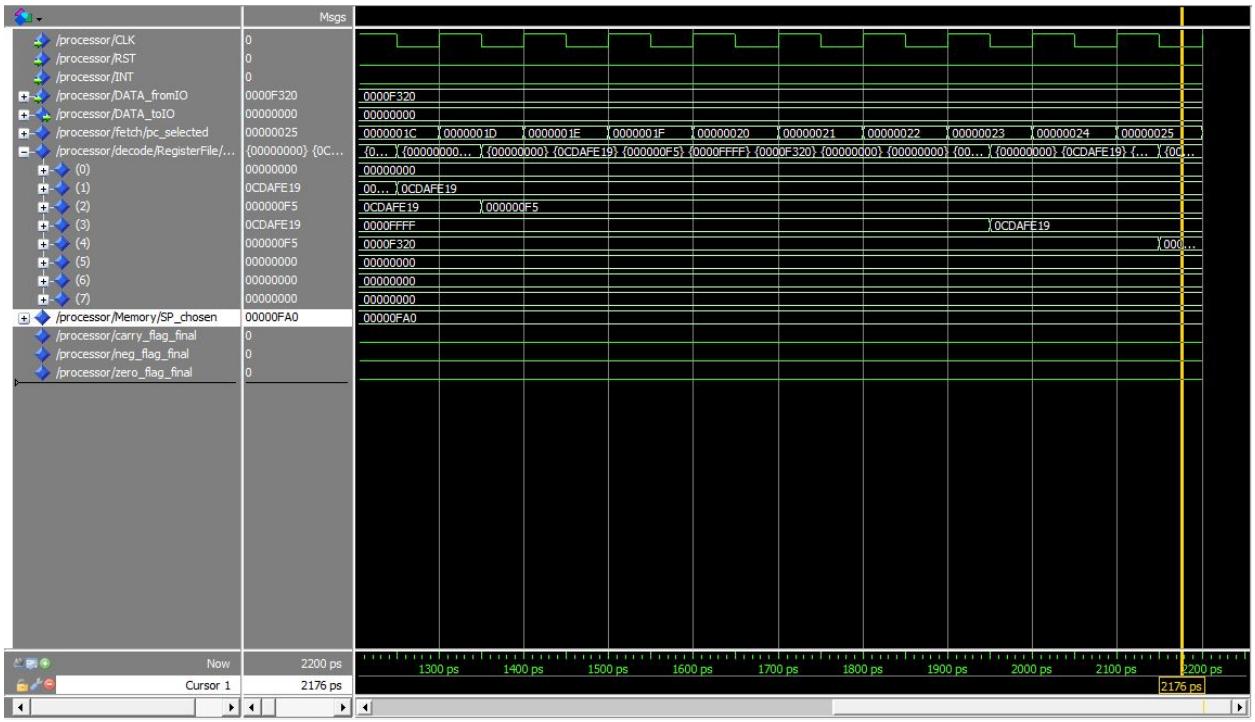
000001f3	XXXXXXXX
000001f4	XXXXXXXX
000001f5	XXXXXXXX
000001f6	XXXXXXXX
000001f7	XXXXXXXX
000001f8	XXXXXXXX
000001f9	XXXXXXXX
000001fa	XXXXXXXX
000001fb	XXXXXXXX
000001fc	XXXXXXXX
000001fd	XXXXXXXX
000001fe	XXXXXXXX
000001ff	XXXXXXXX
00000200	000000F5
00000201	XXXXXXXX
00000202	0CDAFE19
00000203	XXXXXXXX
00000204	XXXXXXXX
00000205	XXXXXXXX
00000206	XXXXXXXX
00000207	XXXXXXXX
00000208	XXXXXXXX
00000209	XXXXXXXX
0000020a	XXXXXXXX
0000020b	XXXXXXXX
0000020c	XXXXXXXX
0000020d	XXXXXXXX
0000020e	XXXXXXXX
0000020f	XXXXXXXX
00000210	XXXXXXXX
00000211	XXXXXXXX
00000212	XXXXXXXX
00000213	XXXXXXXX

The Corresponding VHDL Sim:

- Executed Instructions: (in R2 -> LDM R1,F5)
- Executed Instructions: (PUSH R1 -> STD R1,202)
- Executed Instructions: (LDD R3,202 -> end of instructions)


Msgs
/processor/CLK
/processor/RST
/processor/INT
/processor/DATA_fromIO
/processor/DATA_toIO
/processor/fetch/pc_selected
/processor/decode/RegisterFile/...
(0)
(1)
(2)
(3)
(4)
(5)
(6)
(7)
/processor/Memory/SP_chosen
/processor/carry_flag_final
/processor/neg_flag_final
/processor/zero_flag_final
0CDAFE19
0000FFFF
0000F320
0CDAFE19
0000FFFF
0000F320
000000F5
00000FA0
Now
2200 ps
Cursor 1
980 ps
100 ps
200 ps
300 ps
400 ps
500 ps
600 ps
700 ps
800 ps
900 ps
980 ps


Msgs
/processor/CLK
/processor/RST
/processor/INT
/processor/DATA_fromIO
/processor/DATA_toIO
/processor/fetch/pc_selected
/processor/decode/RegisterFile/...
(0)
(1)
(2)
(3)
(4)
(5)
(6)
(7)
/processor/Memory/SP_chosen
/processor/carry_flag_final
/processor/neg_flag_final
/processor/zero_flag_final
0000F320
00000000
00000017
00000018
00000019
0000001A
0000001B
0000001C
0000001D
0000001E
0000001F
00000020
000000F5
0CDAFE19
0000FFFF
0000F320
00000FA0
00000F9F
00000F9F
00000FA0
Now
2200 ps
Cursor 1
1690 ps
700 ps
800 ps
900 ps
1000 ps
1100 ps
1200 ps
1300 ps
1400 ps
1500 ps
1600 ps
1690 ps



- **V3 (With Forwarding/Hazard Detection):**

Similar results as V2

- **V4 (With Forwarding/Hazard Detection/Flushing):**

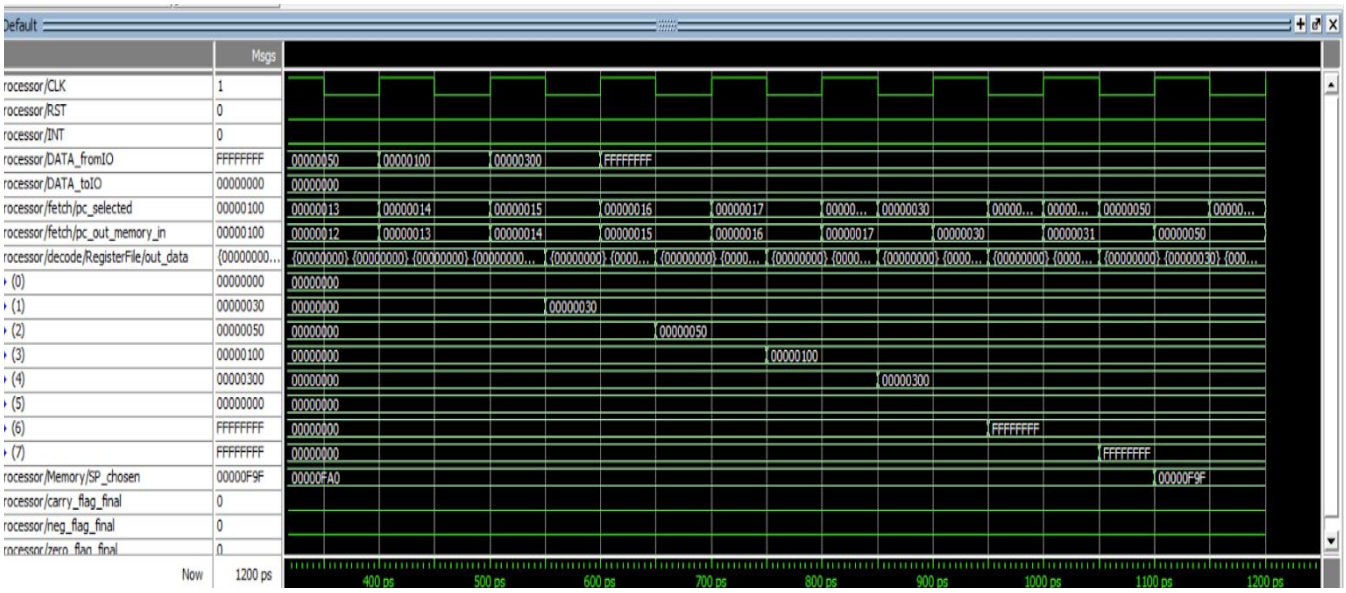
Similar results as V2

4. Branching:

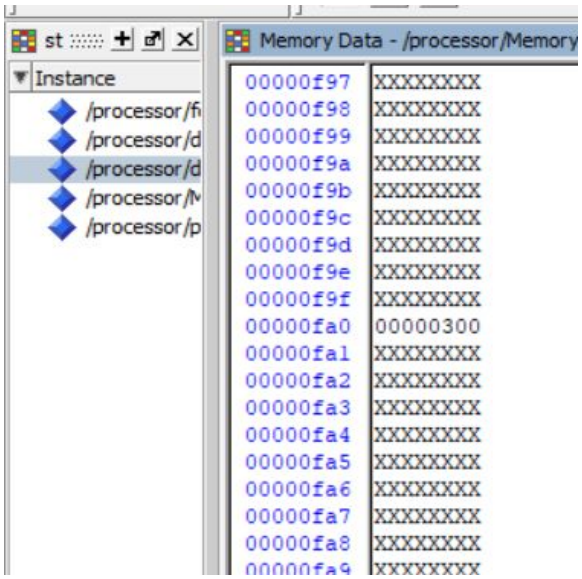
It has been tested on the final Project (Project-V4 Folder) and here is the Results:

I will make the test in several ways and will try to analyze the behavior as much as possible.

- Without trying any interrupt inside the code.

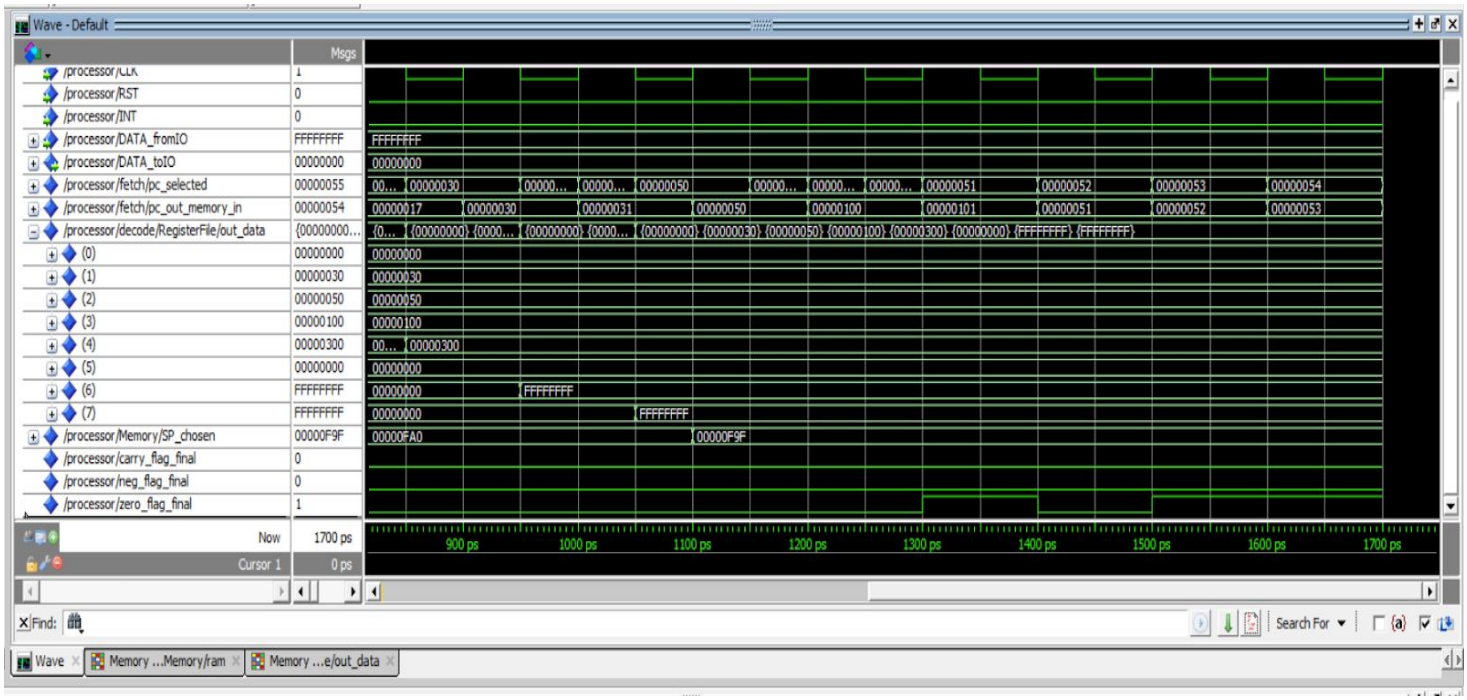


Here we can see inputs saved in the registers and the push instruction is executed also and the results will be given in the next image

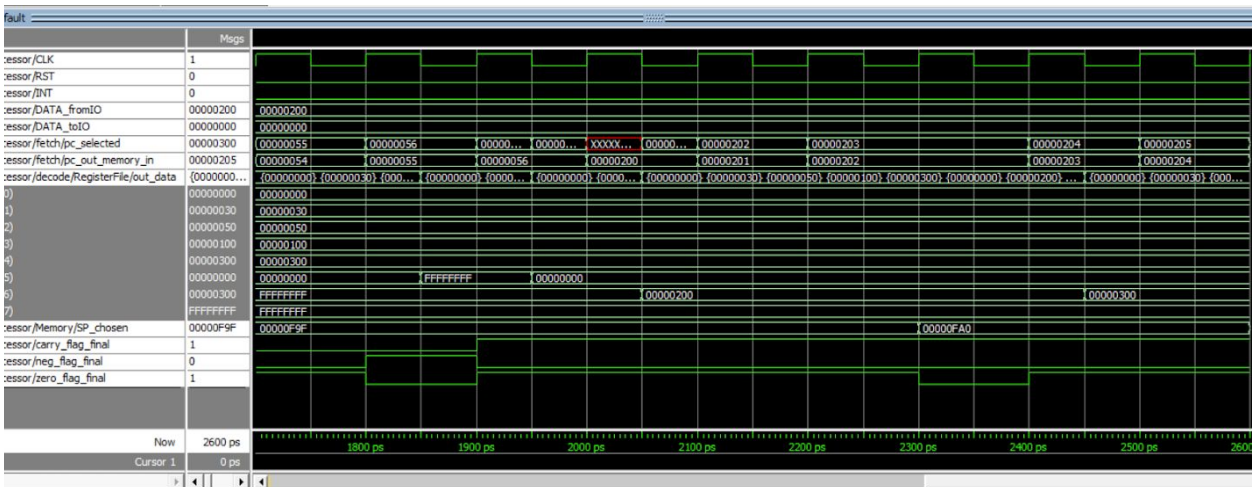


We start our stack from FA0

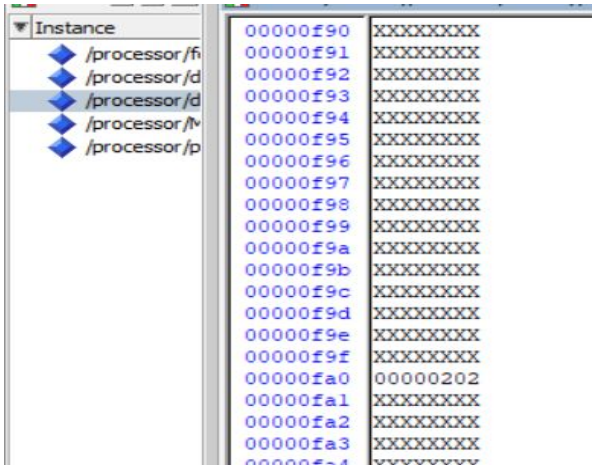
Then we will jump to R1 = 30

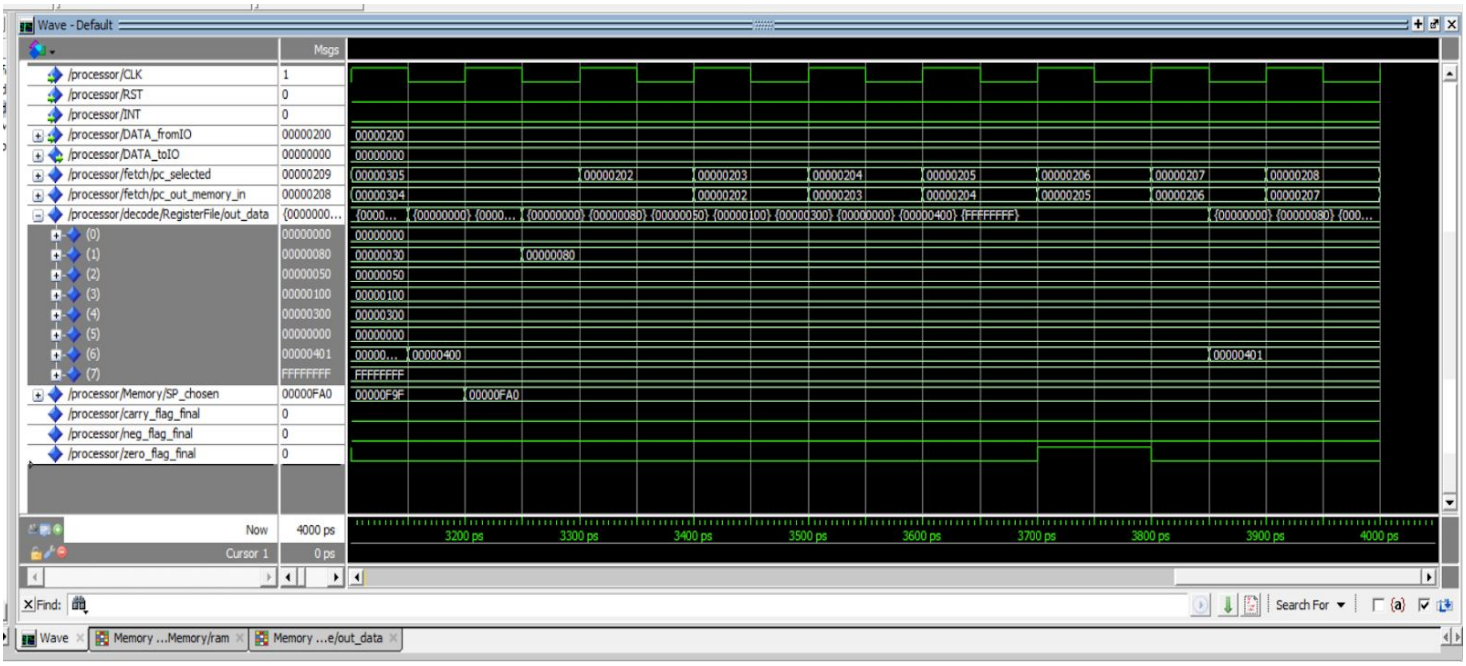


In this image we can see that pc_out_memory_in (executed in fetch stage) = 30 right now which means jmp R1 executed then we will make add to set Zero flag =1. And pc will 100 as a wrong prediction then will be 101 but all these instructions are flushed and then will get the right one in the execution stage 51 to continue work



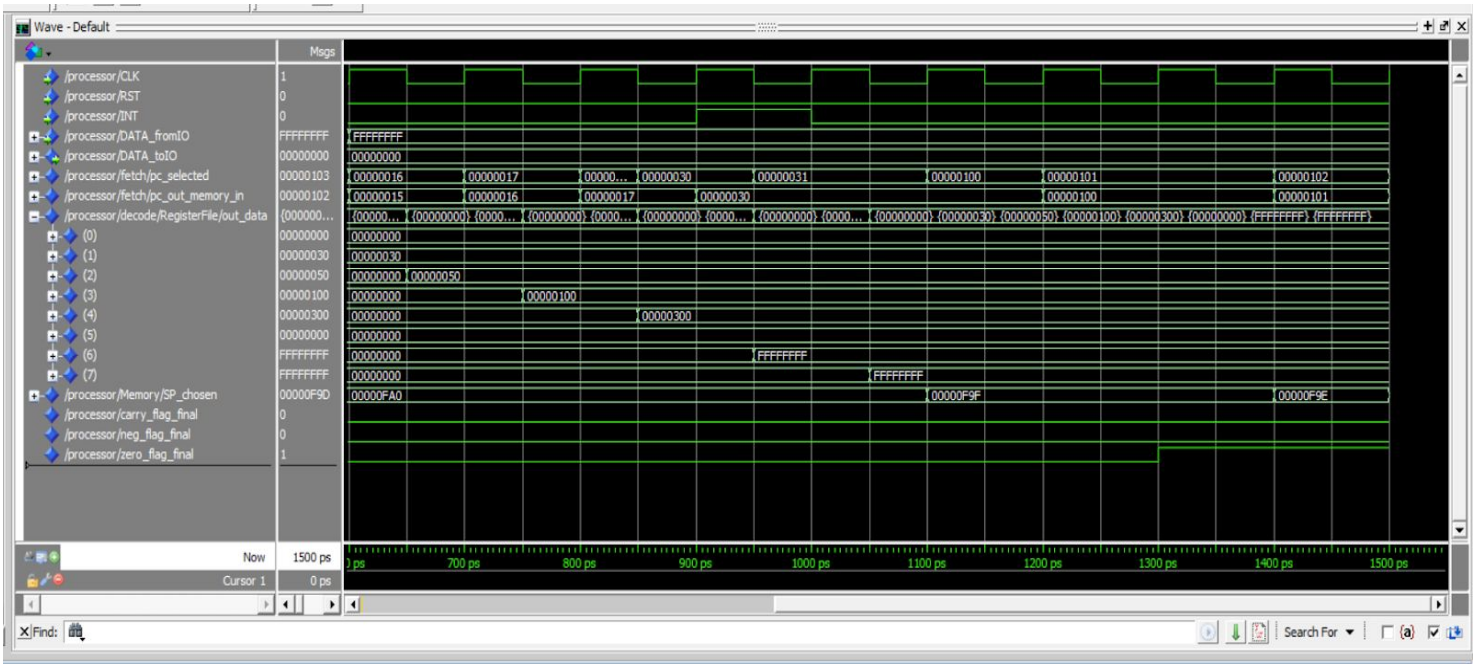
In the previous images we can see NOT R5 and INC R5 and then we received the value of R6 from in port then we used software solution with the data hazards detection unit because this unit in the decode we make all jz related work in fetch so willn't work but we implement an another in the fetch but we notice that it can't cover all cases so we add the software solution. Jump to 200 where we will pop R6 as we can see then call wil store pc in stack in FA0.



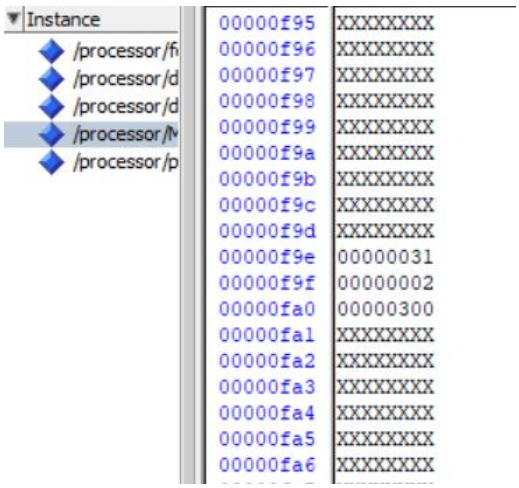


We can see that when we RET from .ORG 300 we make a pop and all the operations in the return is down and R6 increment.

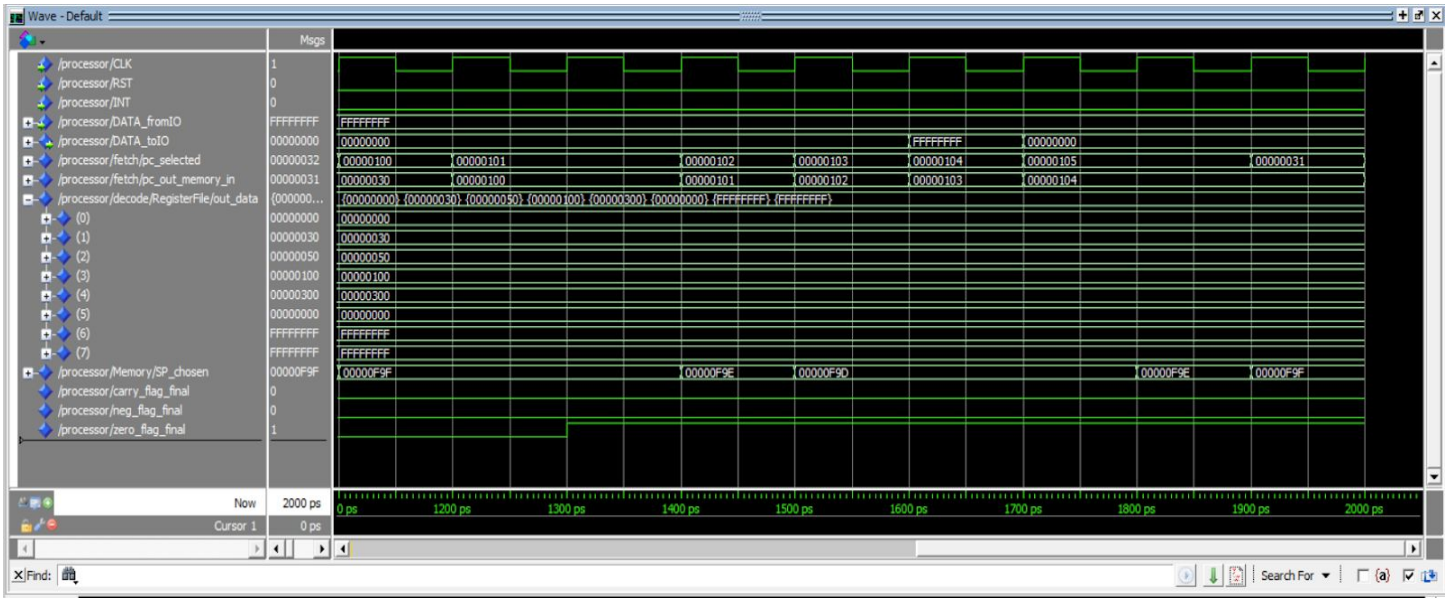
- This time we will try the 1_st INT only.



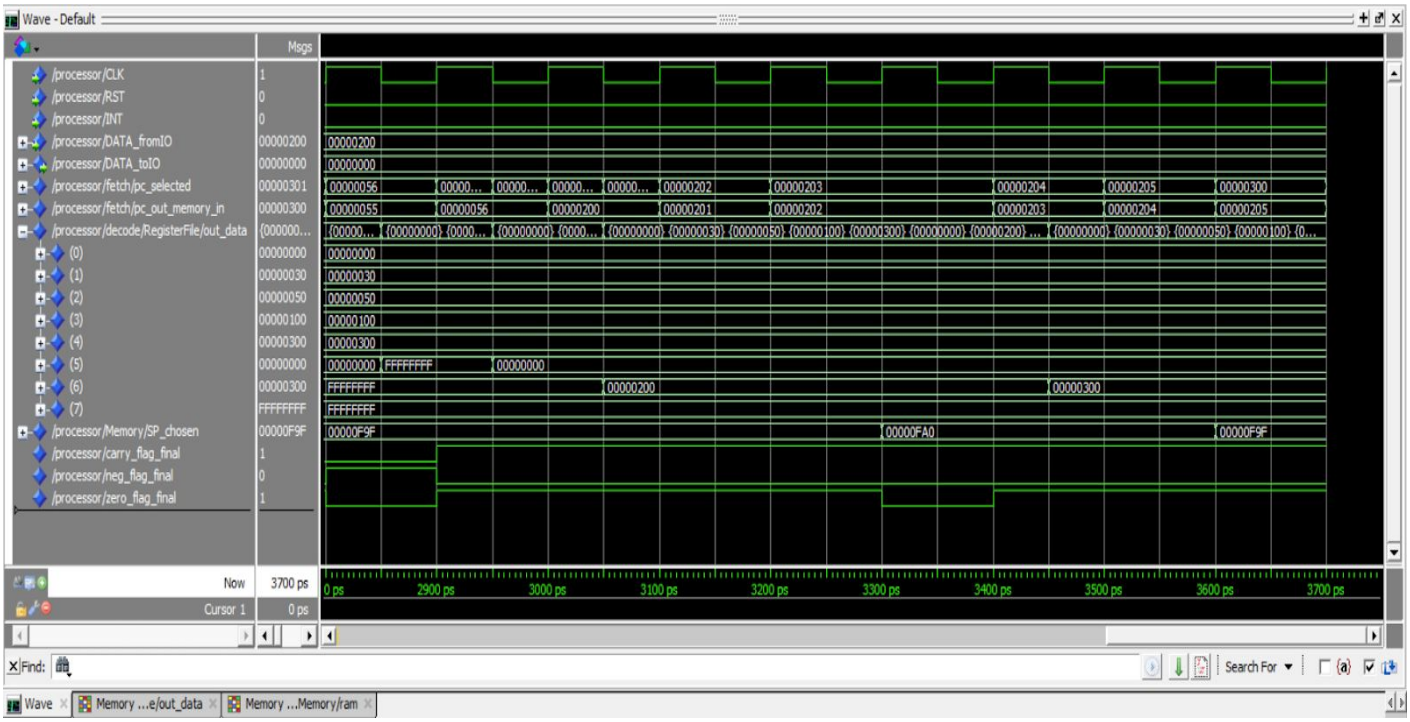
We can see that we INT is raised we start from next cycle to fetch him (1000 -> 1100) and then we make the value M[3,2] enter the pipe as PC to got executed then we make stall one cycle to be able to push pc and flags in the stack

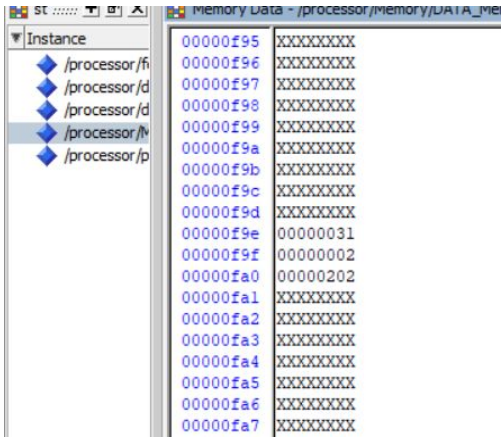


Note bit 1 = zero flag.

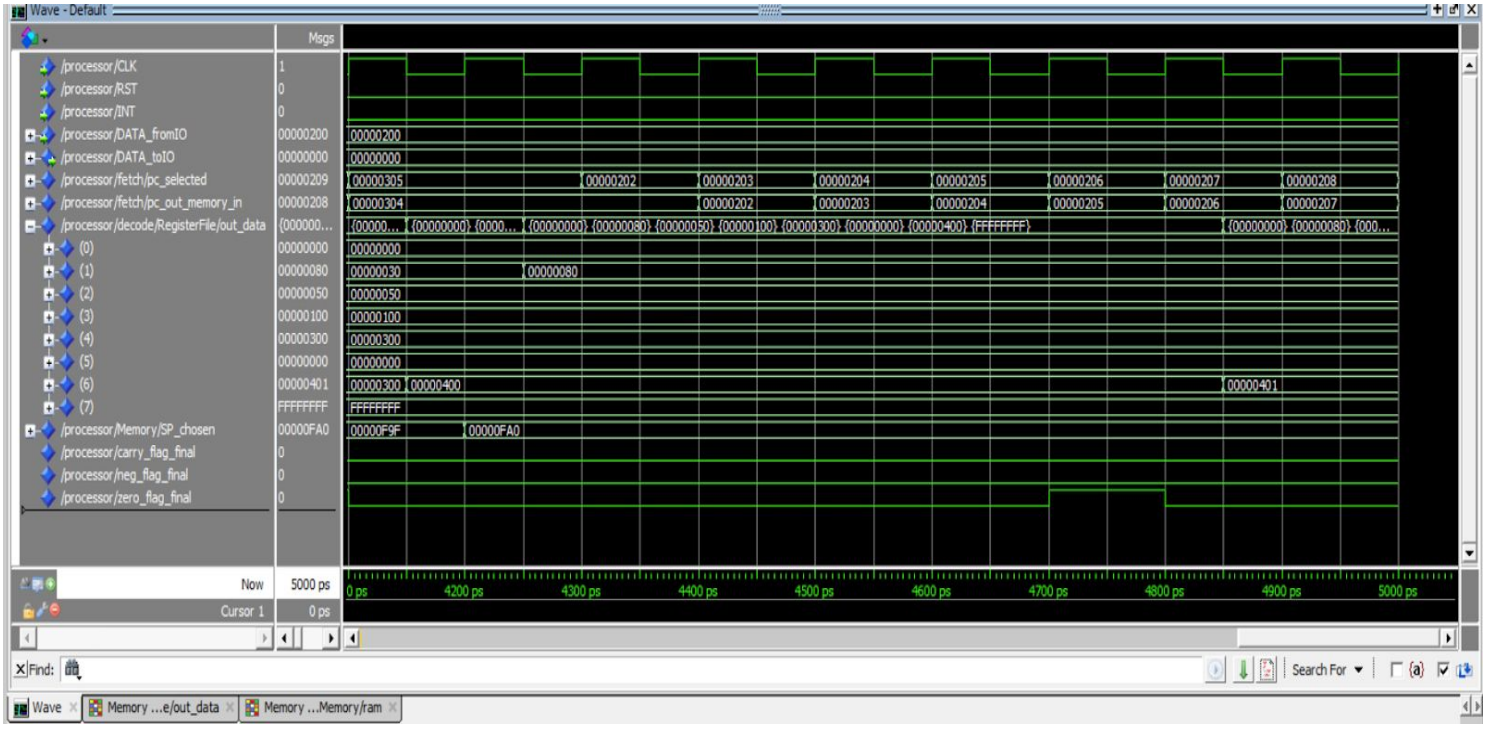


ADD, OUT, RTI are done and then we return back JZ R2 and will jump directly to 50 then JZ R3 will jump as taken but we will know in the execution stage that it's not and correct the prediction. Note we start with weak taken

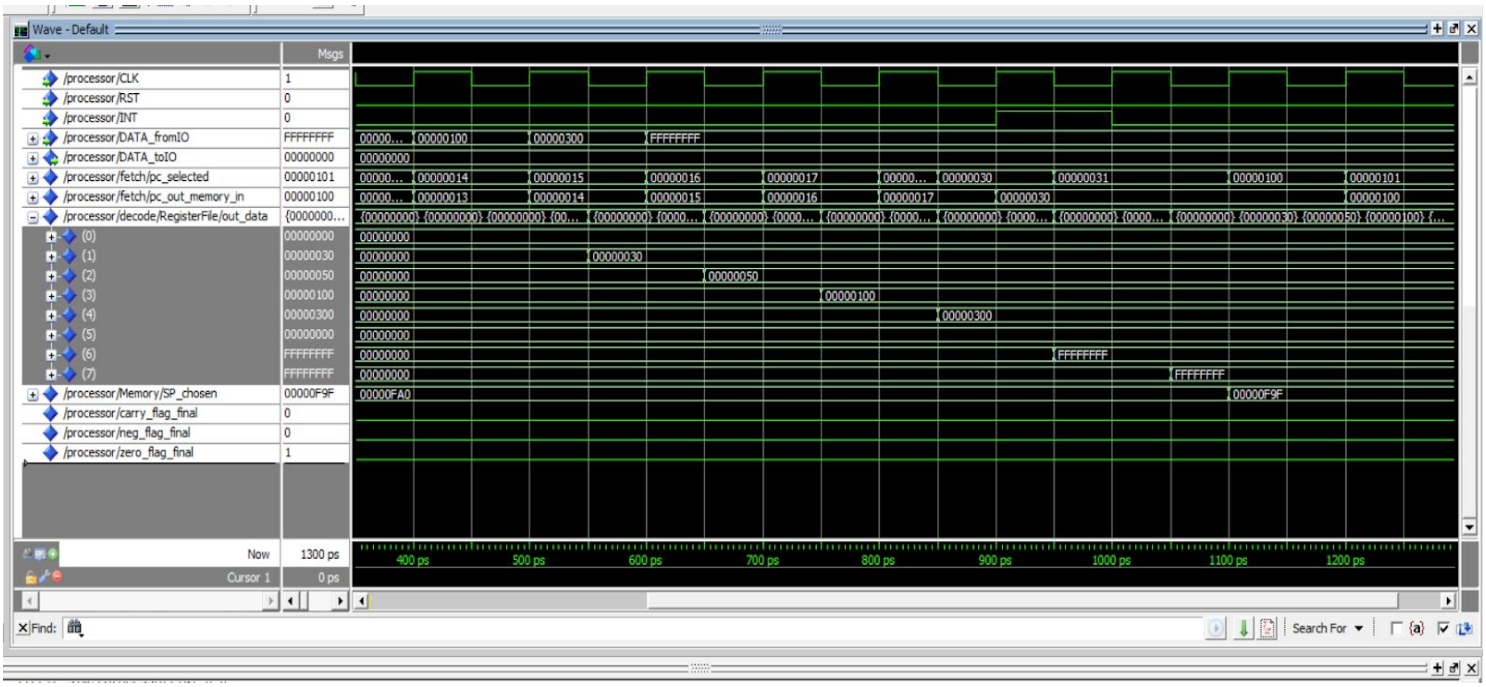


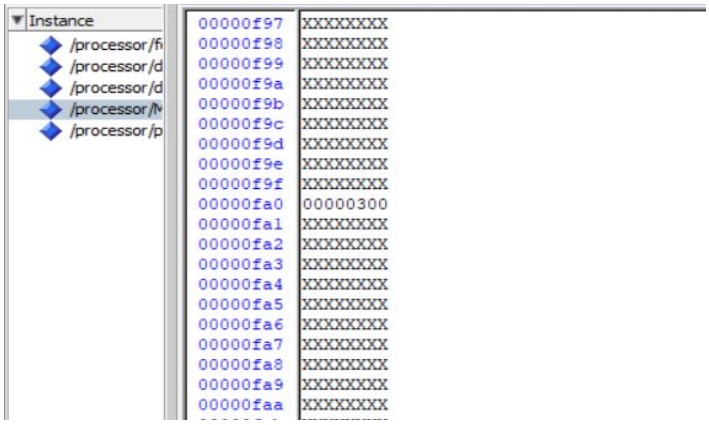


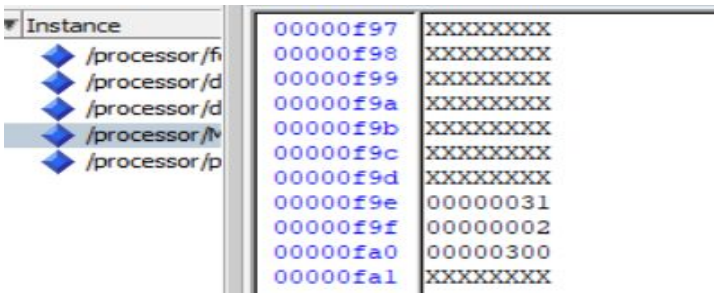
We leave old values as the sp pointer will make over write on them.

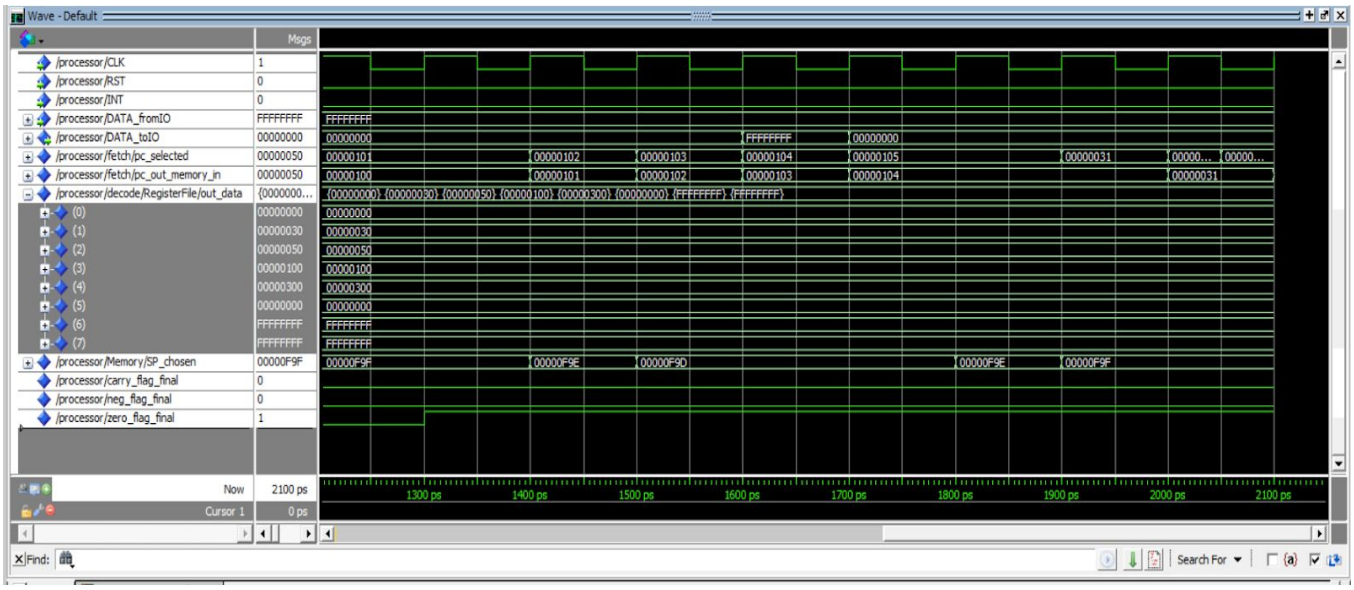


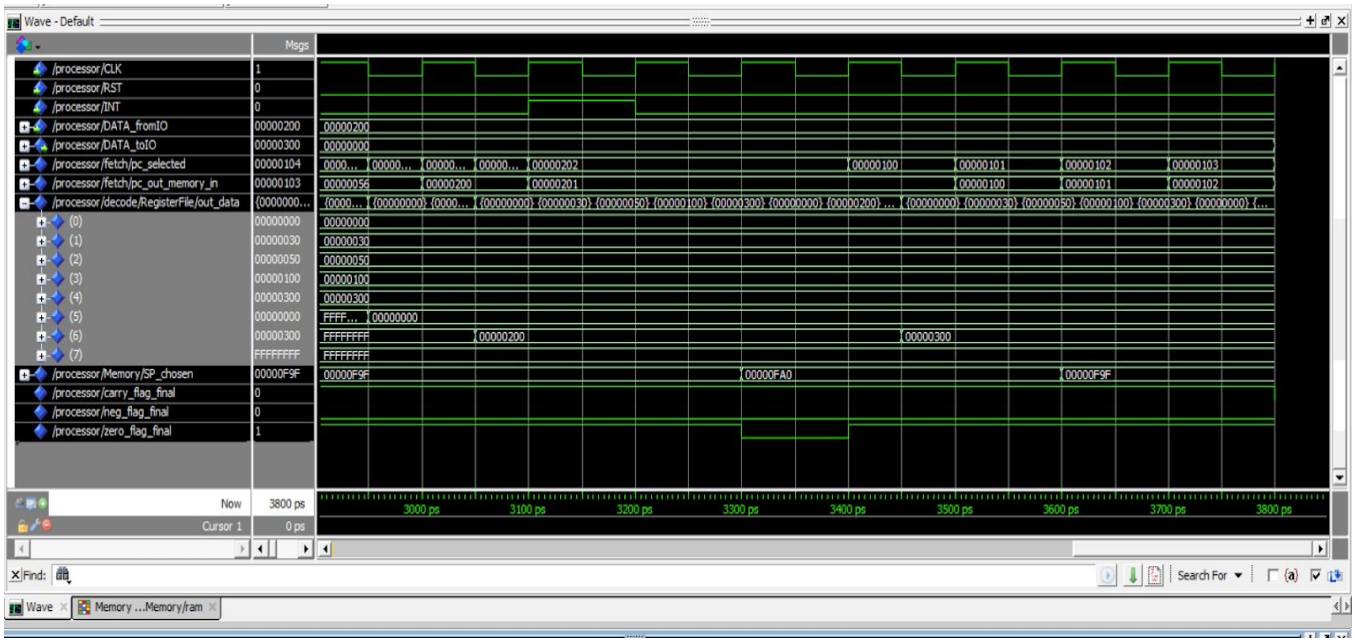
- With 2 INT in the same time.


Wave - Default
Msgs
/processor/CLK
/processor/RST
/processor/INT
/processor/DATA_fromIO
/processor/DATA_toIO
/processor/fetch/pc_selected
/processor/fetch/pc_out_memory_in
/processor/decode/RegisterFile/out_data
(0)
(1)
(2)
(3)
(4)
(5)
(6)
(7)
/processor/Memory/SP_chosen
/processor/carry_flag_final
/processor/neg_flag_final
/processor/zero_flag_final
Now
1300 ps
Cursor 1
0 ps
400 ps
500 ps
600 ps
700 ps
800 ps
900 ps
1000 ps
1100 ps
1200 ps
Find:
Search For

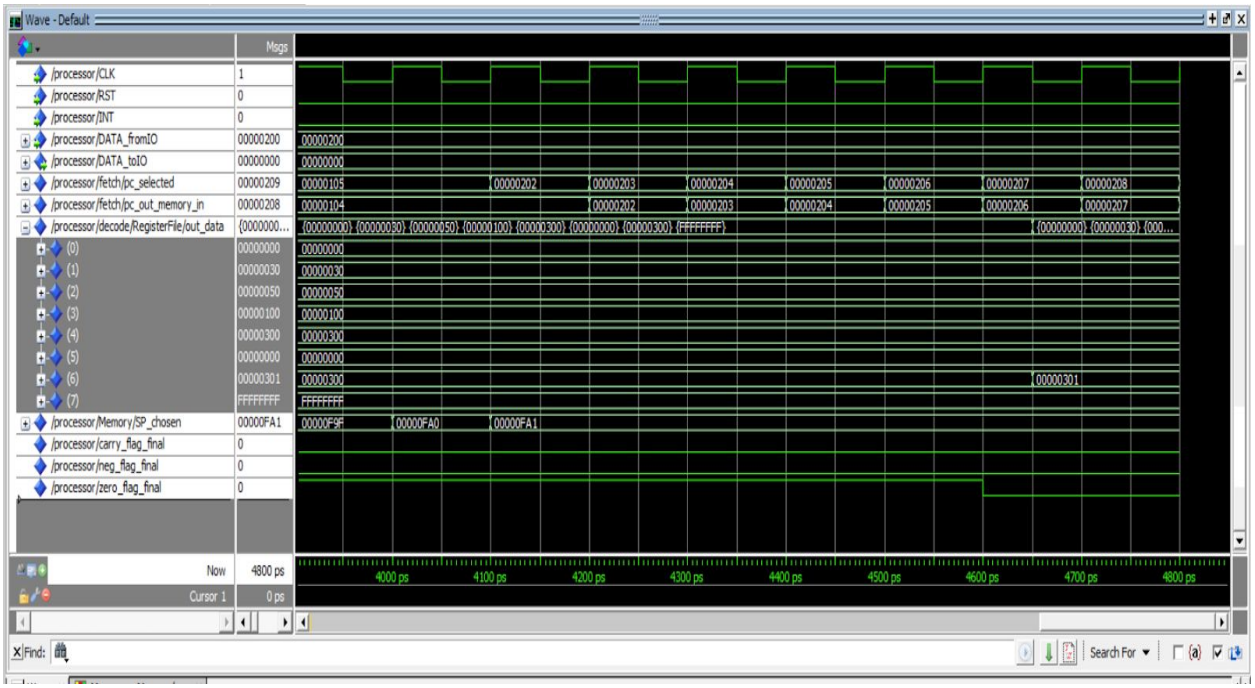

Instance
/processor/f
/processor/d
/processor/d
/processor/M
/processor/p
00000f97 XXXXXXXX
00000f98 XXXXXXXX
00000f99 XXXXXXXX
00000f9a XXXXXXXX
00000f9b XXXXXXXX
00000f9c XXXXXXXX
00000f9d XXXXXXXX
00000f9e XXXXXXXX
00000f9f XXXXXXXX
00000fa0 00000300
00000fa1 XXXXXXXX
00000fa2 XXXXXXXX
00000fa3 XXXXXXXX
00000fa4 XXXXXXXX
00000fa5 XXXXXXXX
00000fa6 XXXXXXXX
00000fa7 XXXXXXXX
00000fa8 XXXXXXXX
00000fa9 XXXXXXXX
00000faa XXXXXXXX


Instance
/processor/fi
/processor/d
/processor/d
/processor/M
/processor/p
00000f97 XXXXXXXX
00000f98 XXXXXXXX
00000f99 XXXXXXXX
00000f9a XXXXXXXX
00000f9b XXXXXXXX
00000f9c XXXXXXXX
00000f9d XXXXXXXX
00000f9e 00000031
00000f9f 00000002
00000fa0 00000300
00000fa1 XXXXXXXX


Wave - Default
Msgs
/processor/CLK
/processor/RST
/processor/INT
/processor/DATA_fromIO
/processor/DATA_toIO
/processor/fetch/pc_selected
/processor/fetch/pc_out_memory_in
/processor/decode/RegisterFile/out_data
/processor/Memory/SP_chosen
/processor/carry_flag_final
/processor/neg_flag_final
/processor/zero_flag_final
Now
Cursor 1
Find:
Search For


Wave - Default
Msgs
/processor/CLK
/processor/RST
/processor/INT
/processor/DATA_fromIO
/processor/DATA_toIO
/processor/fetch/pc_selected
/processor/fetch/pc_out_memory_in
/processor/decode/RegisterFile/out_data
/processor/Memory/SP_chosen
/processor/carry_flag_final
/processor/neg_flag_final
/processor/zero_flag_final
Now
Cursor 1
Find:
Search For
Wave
Memory ...Memory/ram

Here second INT start right after call so we ignore call and can't keep it because we keep the instruction called with the INT so CALL willn't be executed here and we will do the int and then rti and then inc R6 so it will be 301 instead of 401 and we can see that we didn't store any thing for call in stack to there no unneeded values here.



5. Branch Prediction:

We took the incremental way to build our pipe so we make the main pipe as a first step then add forwarding then data hazards detection units then we add the modules implemented to the Branch and Prediction at the same time so we add the branch on the pipe that contains the forwarding and data hazard detection unit so it's hard to merge these parts again with the main pipe and the pipe with forwarding and the pipe with data hazard detection unit as it will consume us a lot of time. So in this part, I'm gonna write a detailed analysis of the cases when we will need a forwarding and when we will need a data hazard detection unit as I can, and then I will show you the final solution with the prediction units integrated with the pipe.

- V1 (No Forwarding/Hazard Detection/Flushing):

  In this part =>

  .ORG 10

  LDM R2,0A #R2=0A

  LDM R0,0 #R0=0

  LDM R1,50 #R1=50

  LDM R3,20 #R3=20

  LDM R4,2 #R4=2

  JMP R3   #Jump to 20

  Every LDM will be separated into two instructions one for the instruction itself and the other one will be for the data and we execute this in code as we make the instruction enters to the decode stage and fetch the value then hold the pipe and make the ID/EX freeze some of its registers and let the others writable to update then with the write IMM/EA. Then we make the control unit out the proposed signal for LDM and continue in the pipe

  F D D E M WB

  So we can guarantee that we have the write data when we are entering the execution. And here we don't have any hazards here. So it will work properly.

  .ORG 20

  SUB R0,R2,R5 #check if R0 = R2

  JZ R1 #jump if R0=R2 to 50

  ADD R4,R4,R4 #R4 = R4*2

  OUT R4

  INC R0

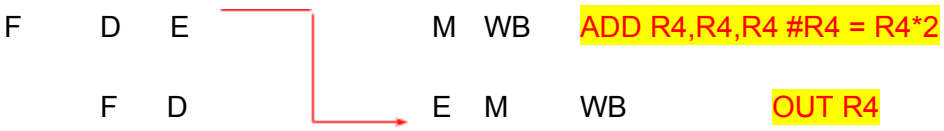
  JMP R3 #jump to 20

  In this part of the code, we will need a forwarding unit for OUT R4 because it is out DST of the previous instruction. So it will work properly by the forwarding unit or we can use NOP.

  The view in case of using NOP

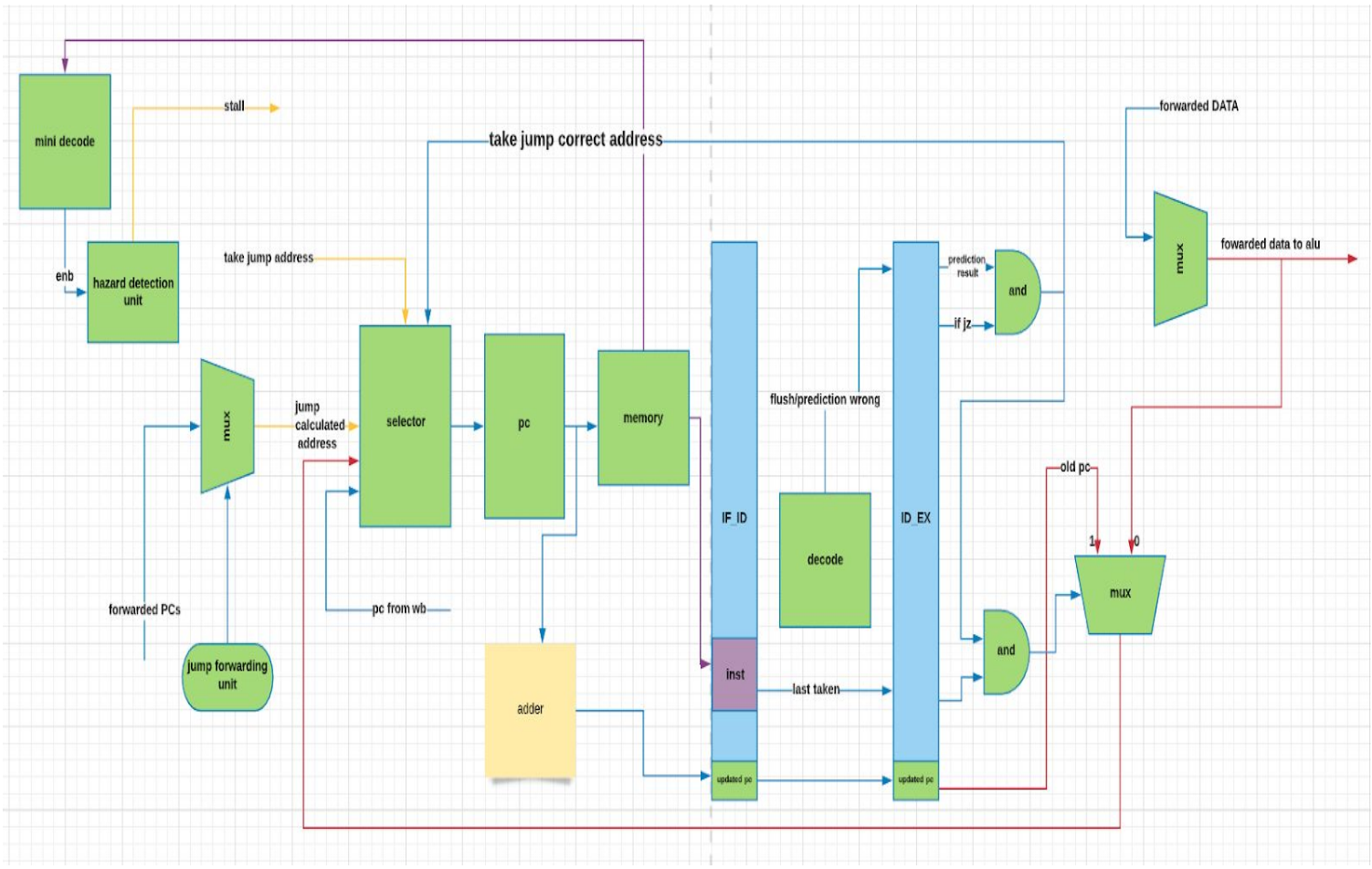
F	D	E	M	WB			ADD R4,R4,R4 #R4 = R4*2
	F	D	E	M	WB		NOP
		F	D	E	M	WB	NOP
			F	D	E	M	WB OUT R4

But in case of forwarding unit and this is the case that we will run on it the test file will be like



But before we go to the next part from the code I want to explain how JZ is working in our design to make the results more obvious.

As we can see in the following Figure after we fetch the JZ it will enter the decode stage only if it's SRC is available because of the hazard detection unit and we also implement another forwarding unit for the branch to get the data as fast as possible. When JZ enters the decode we will assume that this JZ weak taken so will fetch the instruction that is pointed to by the SRC and it will be PC = 50 and then this PC will enter decode and the PC = 51 will be fetched and now the JZ in executing and determine the PC will change to PC = 22 instruction after JZ or will continue to 52 if we will not continue we will flush the pipe.
In the next cycle (decode stage)the prediction will know that it will not be taken ,and start to move it from weak taken to weak not taken. After that loop will continue and make it move to String not taken then remain until loop end and prediction unit finds out that this will be taken in this time so will move it from strong not taken to weak not taken.



.ORG 50

LDM R0,0 #R0=0

LDM R2,8 #R2=8

LDM R3,60 #R3=60

LDM R4,3 #R4=3

JMP R3 #jump to 60

In this part of code, We have no hazards and the JMP will be executed properly without any problem.

.ORG 60

ADD R4,R4,R4 #R4 = R4*2

OUT R4

INC R0

AND R0,R2,R5 #when R0 < R2(8) answer will be zero

JZ R3 #jump if R0 < R2 to 60

INC R4

OUT R4

In this part, we will have 2 hazards

1- ADD R4,R4,R4 #R4 = R4*2

OUT R4

2- INC R4

OUT R4

In both, we will handle them as we discussed earlier, And JZ also will work as shown previously.

- **V2 (With Forwarding):**

  Already we illustrate this case in version 1.

- **V3 (With Forwarding/Hazard Detection):**

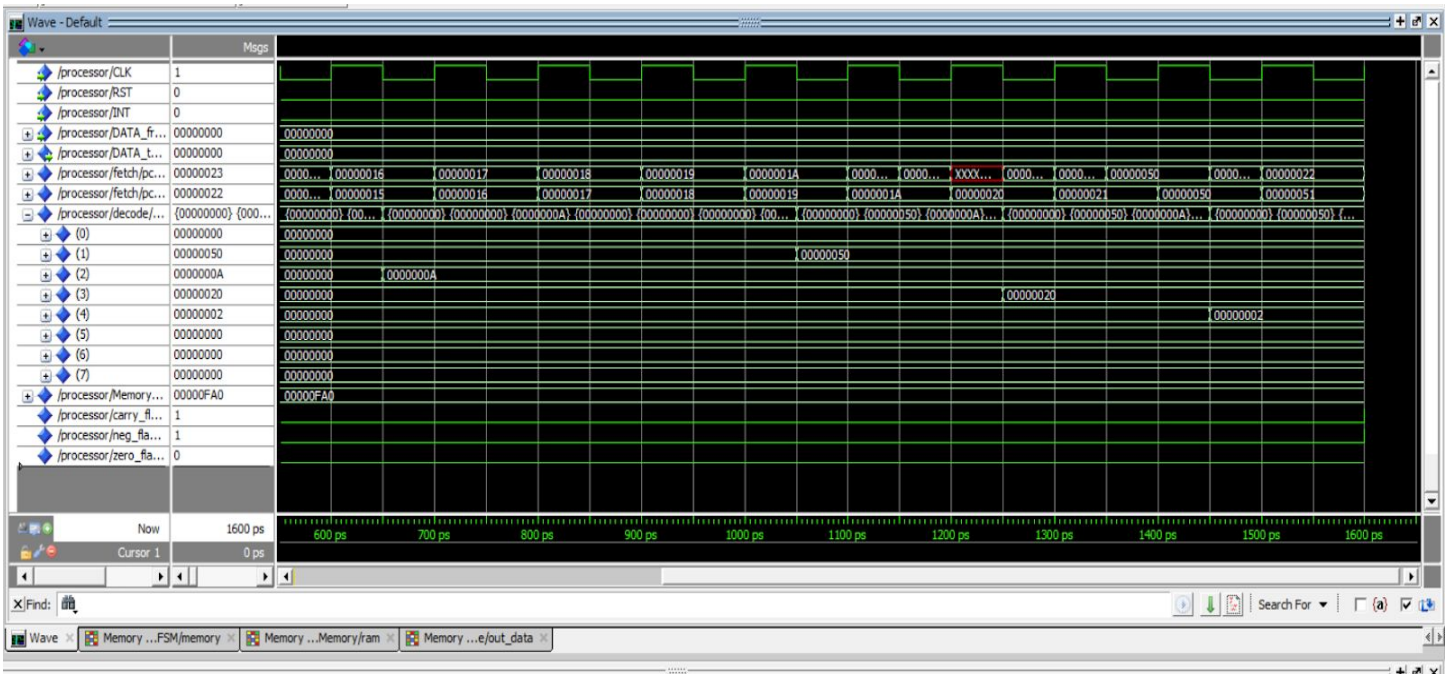
  No need for a data hazard detection unit here as there is no load use case in this test case.

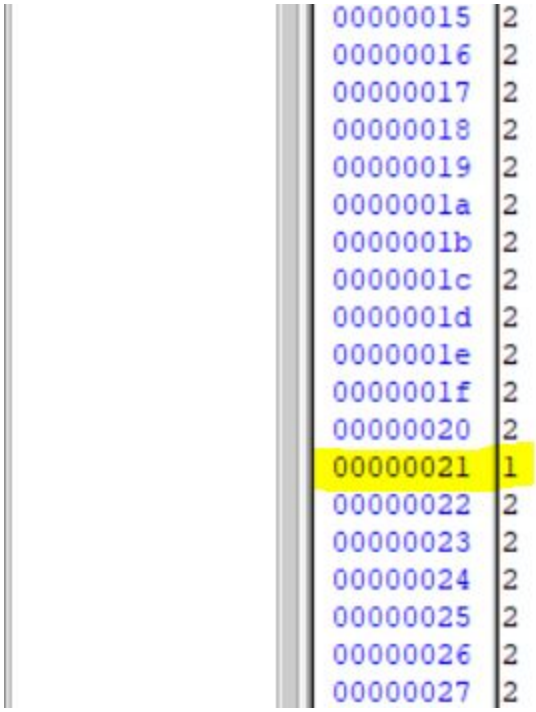
- **V4 (With Forwarding/Hazard Detection/Flushing):**

  In this case, we will be like V1 but we should illustrate that in v1 we still don't have a jump module but here is the current version that contains jump and prediction.

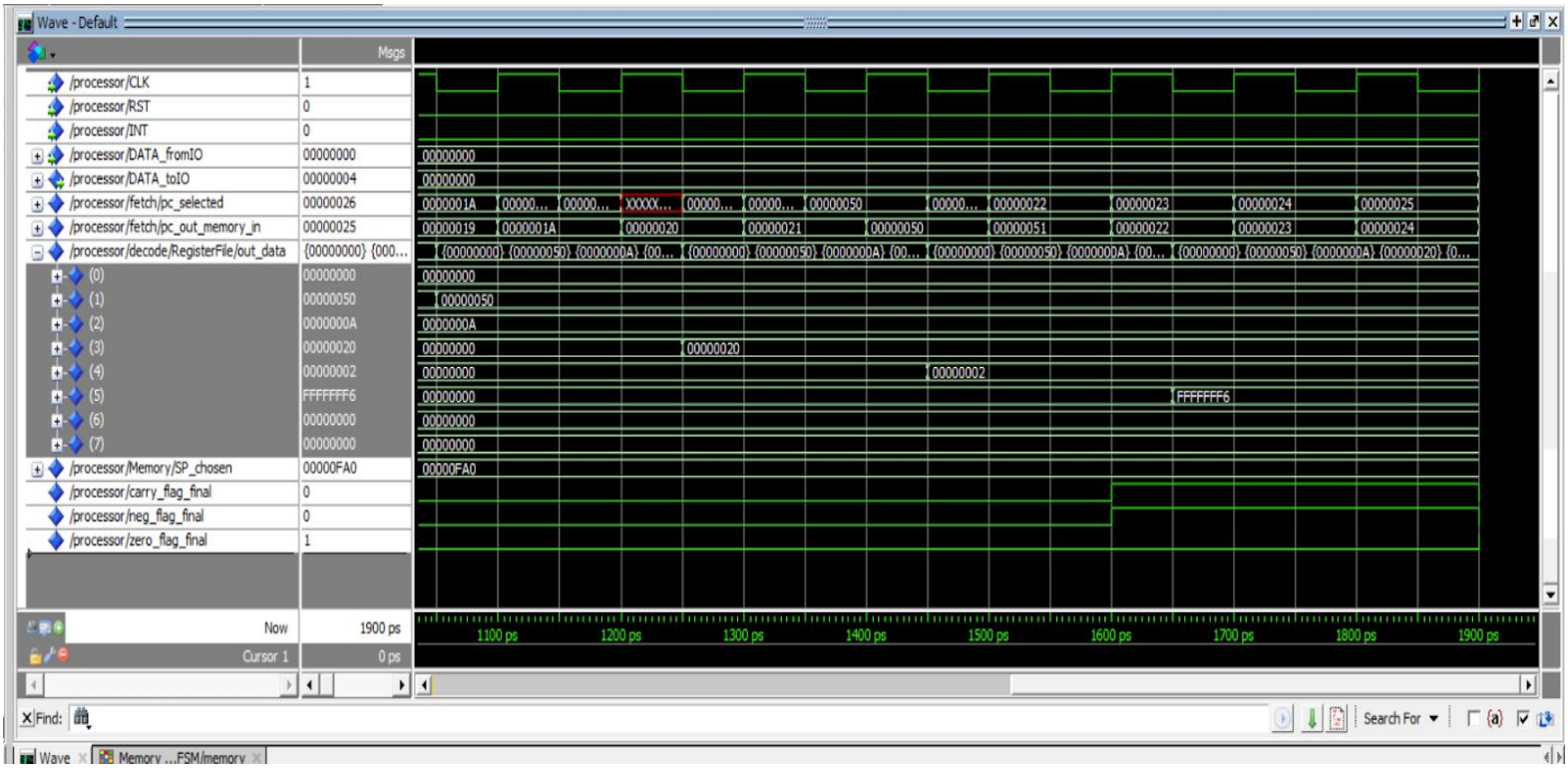
  In the following part, we will show values in the register file and the waveform.

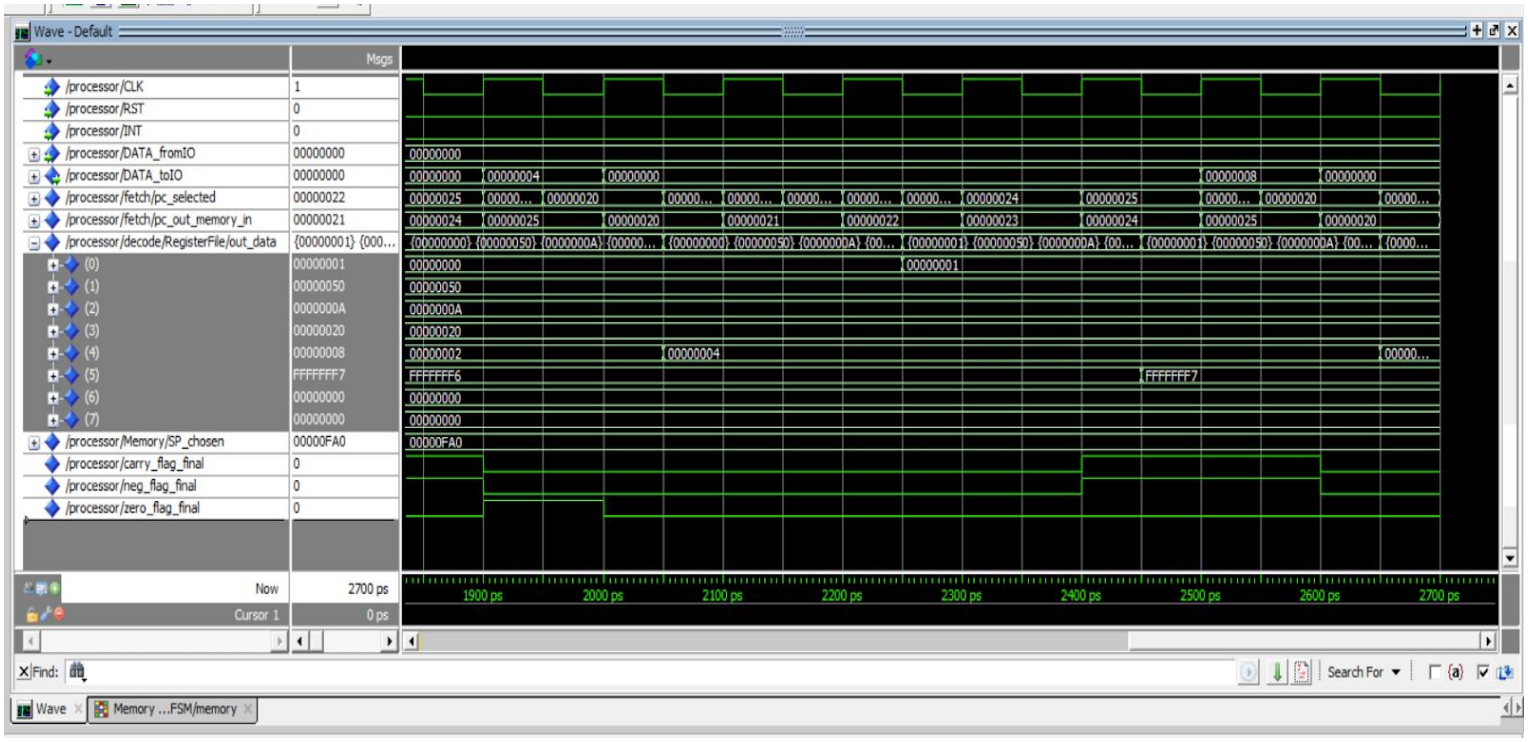
  In the following figure, we can see values in registers from LDM instructions and JMP R3 was done and we got the wrong decision to take it as taken JZ R1 but in the next figure you will see that PC will be corrected.



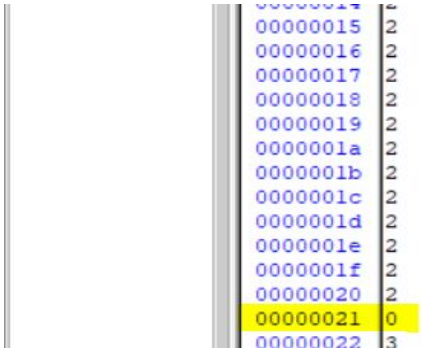


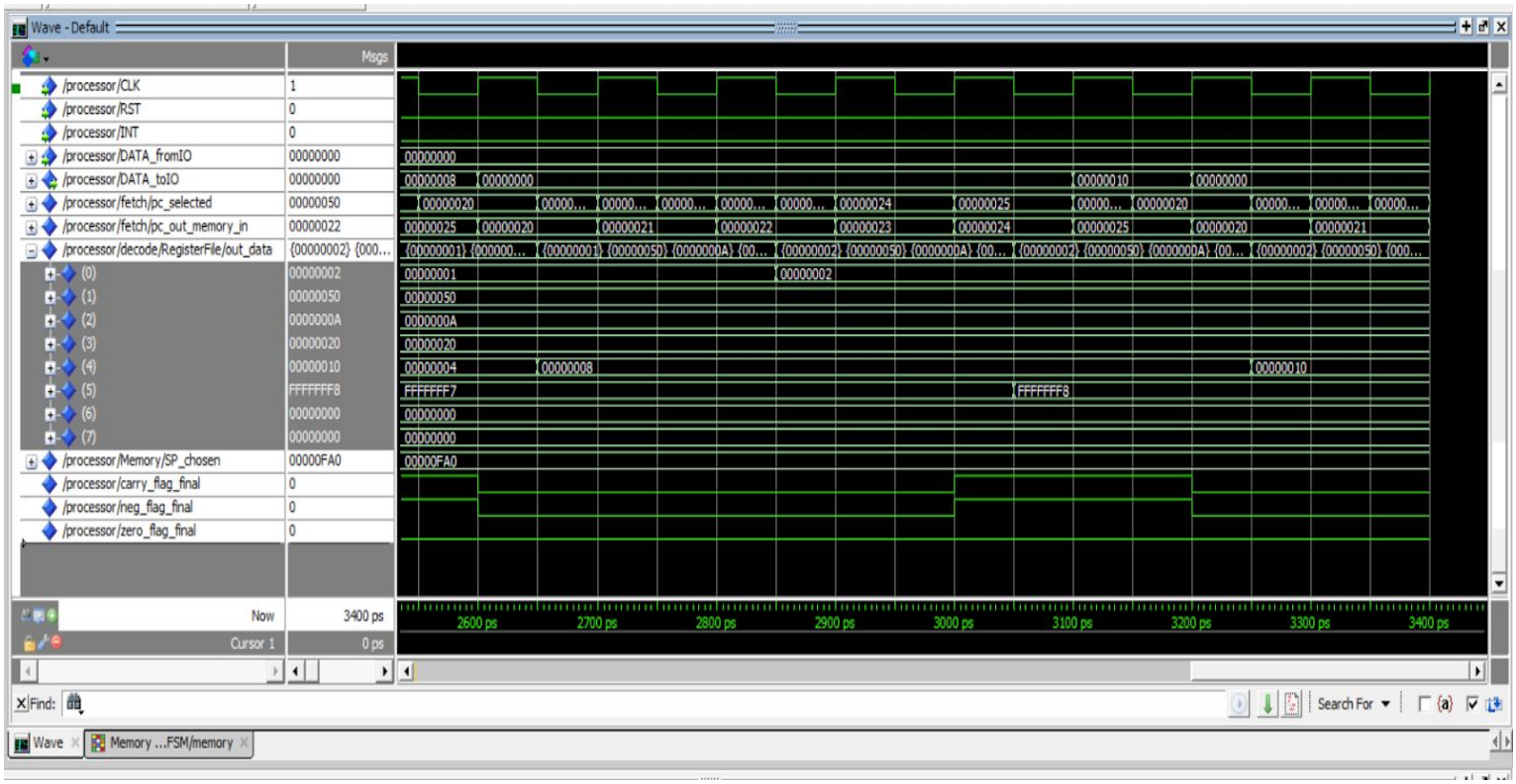
We initially make all instructions have a weak taken value. Here prediction detected that it will not taken so move it from 2 (weak taken) to 1 (weak not taken).

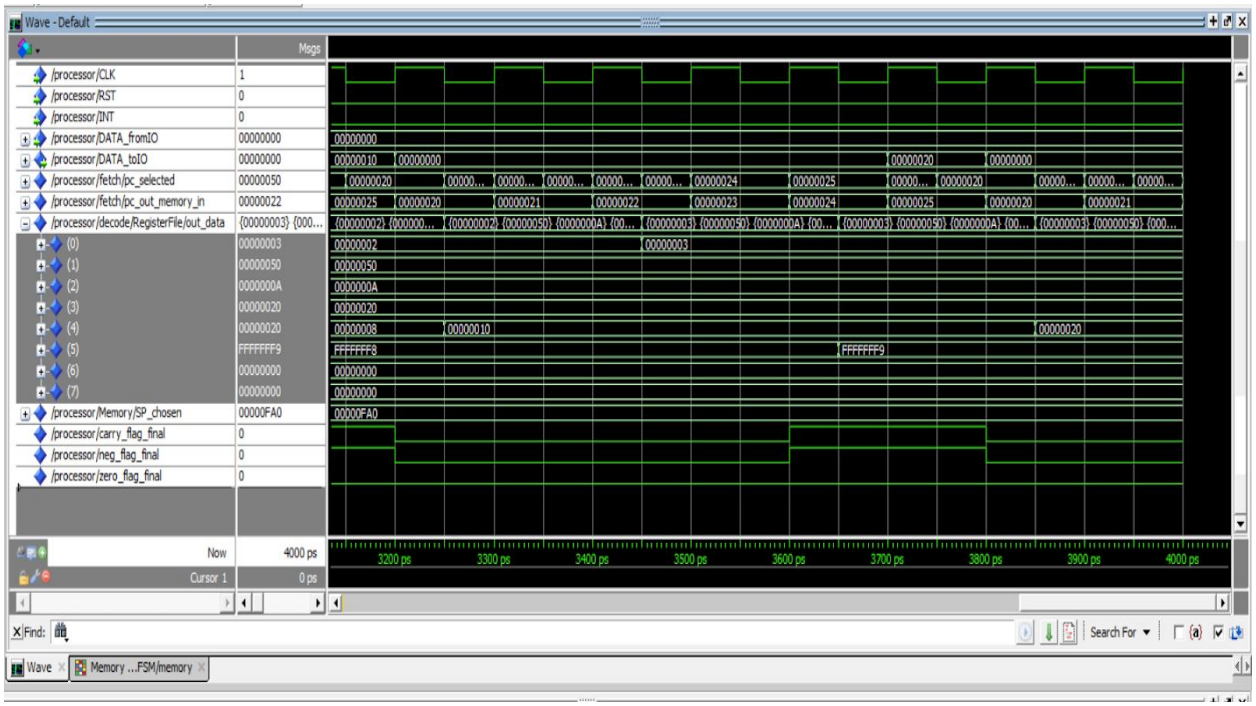

Wave - Default
Msgs
/processor/CLK
/processor/RST
/processor/INT
/processor/DATA_fromIO
/processor/DATA_toIO
/processor/fetch/pc_selected
/processor/fetch/pc_out_memory_in
/processor/decode/RegisterFile/out_data
(0)
(1)
(2)
(3)
(4)
(5)
(6)
(7)
/processor/Memory/SP_chosen
/processor/carry_flag_final
/processor/neg_flag_final
/processor/zero_flag_final
Now
1900 ps
Cursor 1
0 ps
1100 ps
1200 ps
1300 ps
1400 ps
1500 ps
1600 ps
1700 ps
1800 ps
1900 ps
Find:
Search For
Wave
Memory ...FSM/memory

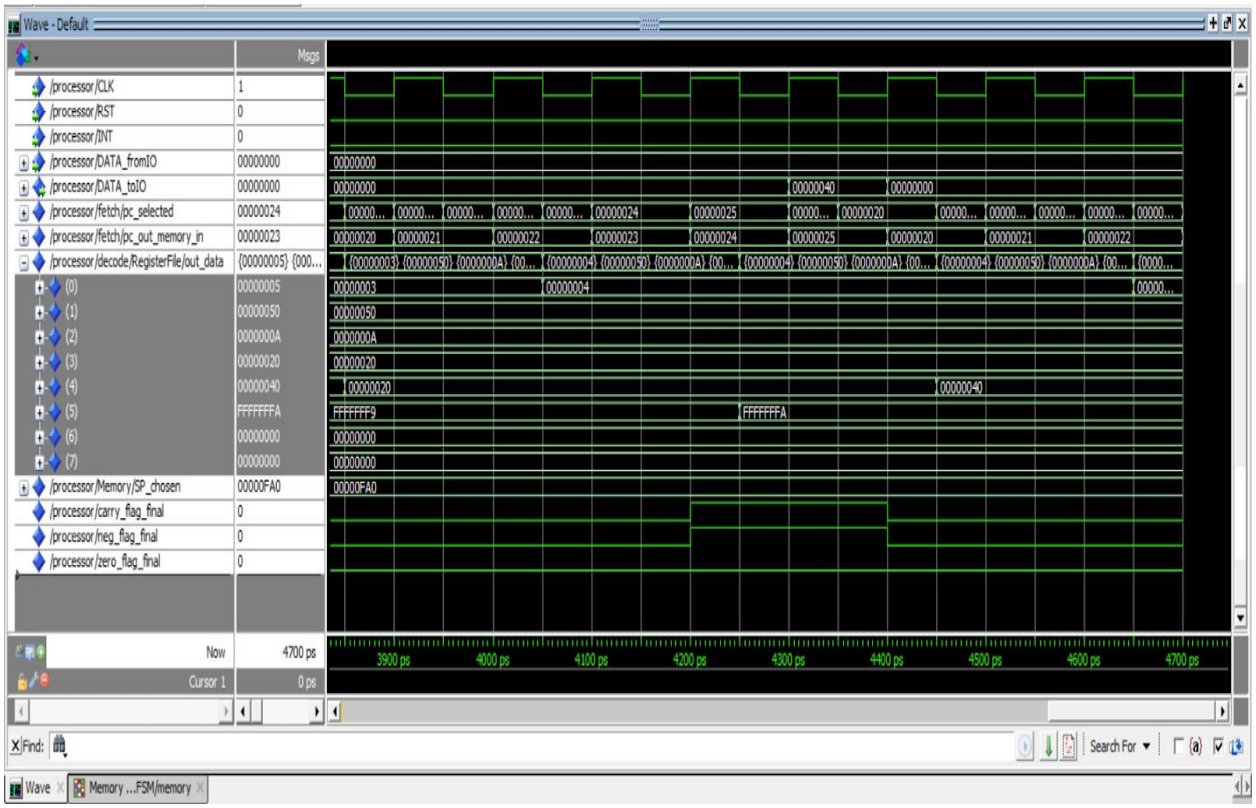

Wave - Default
Msgs
/processor/CLK
/processor/RST
/processor/INT
/processor/DATA_fromIO
/processor/DATA_toIO
/processor/fetch/pc_selected
/processor/fetch/pc_out_memory_in
/processor/decode/RegisterFile/out_data
(0)
(1)
(2)
(3)
(4)
(5)
(6)
(7)
/processor/Memory/SP_chosen
/processor/carry_flag_final
/processor/neg_flag_final
/processor/zero_flag_final
Now
2700 ps
Cursor 1
0 ps
1900 ps
2000 ps
2100 ps
2200 ps
2300 ps
2400 ps
2500 ps
2600 ps
2700 ps
Find:
Search For
Wave
Memory ...FSM/memory

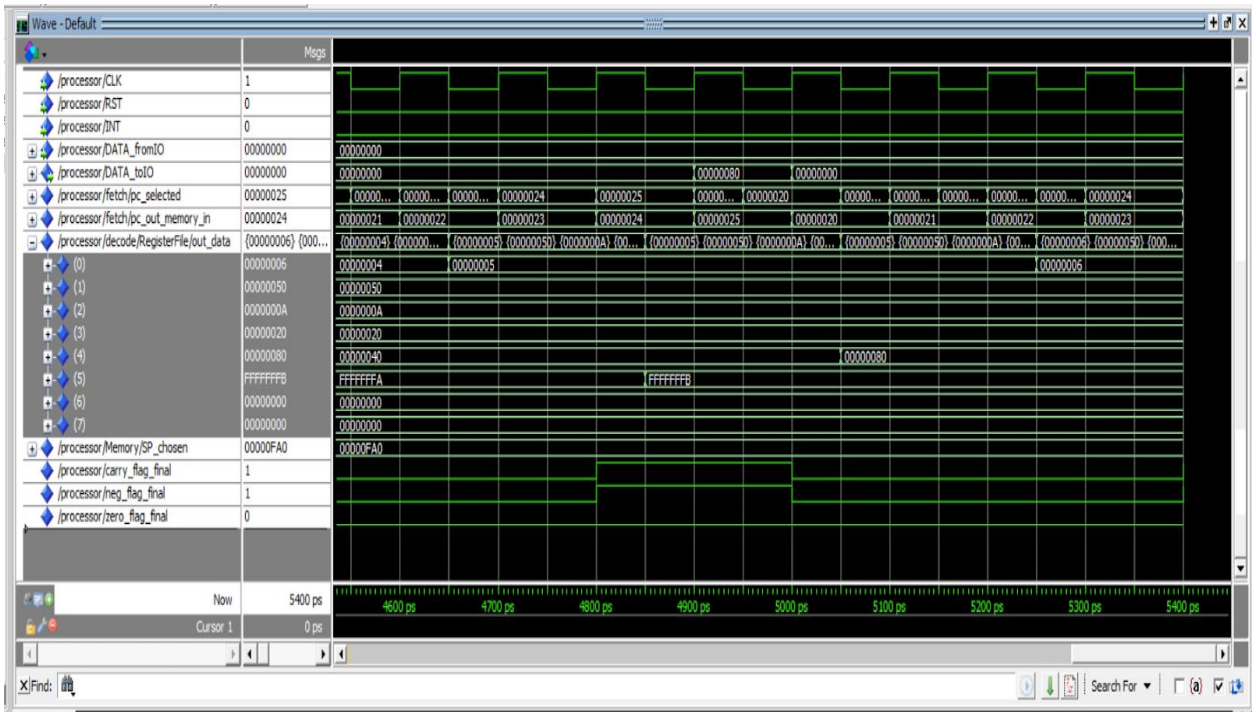
Here prediction predicts that it will not be taken so we go to 22 instead of going to 50 then 51 then return back to 22 and will move it from 1 (weak not taken) to 0 (strong not taken). And in the previous figure also SUB R0,R2,R5 is done and R4 also updated and out value in The Interval from 1900ps to 2000ps and R0 incremented and the loop will continue till R2 == R0 and then will jump to 50.

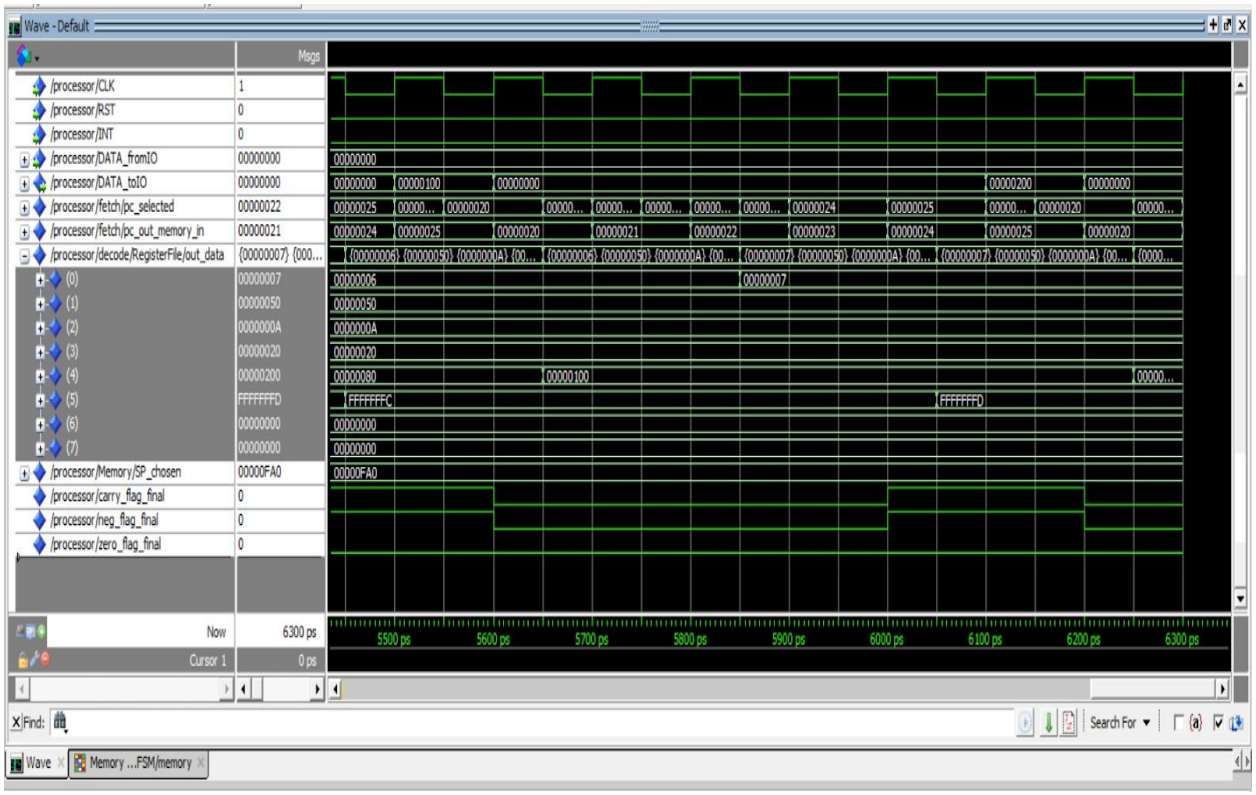


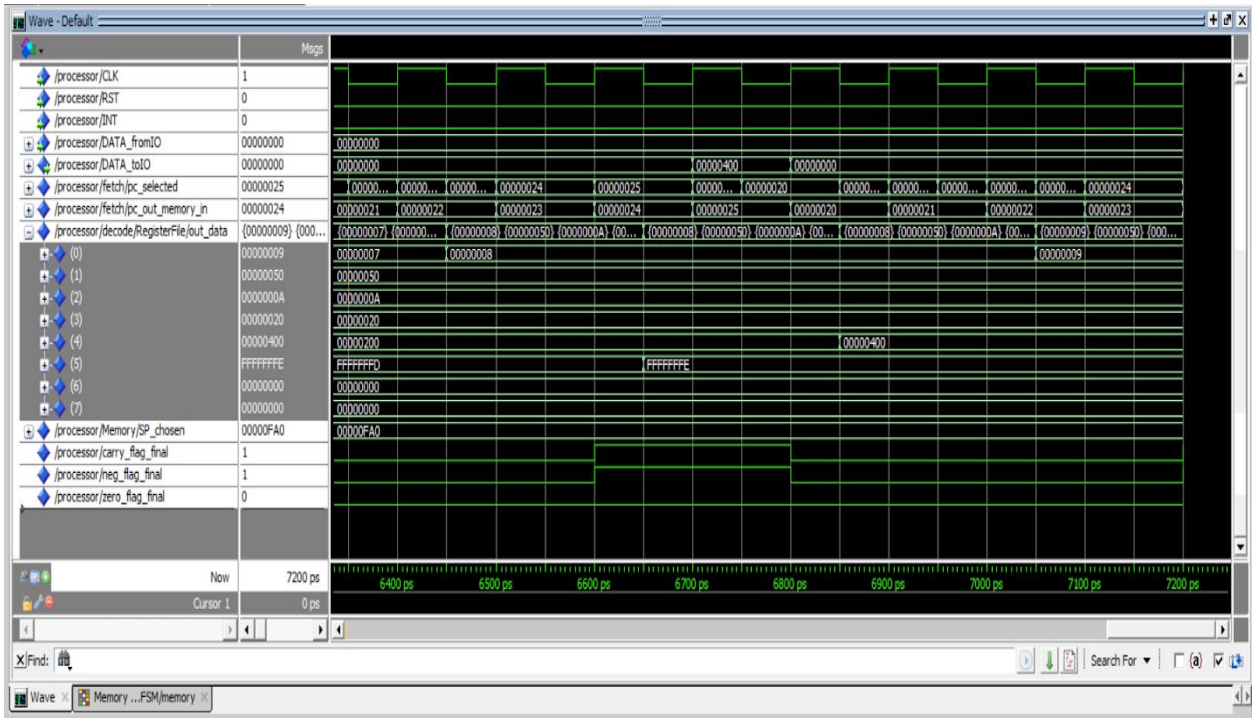


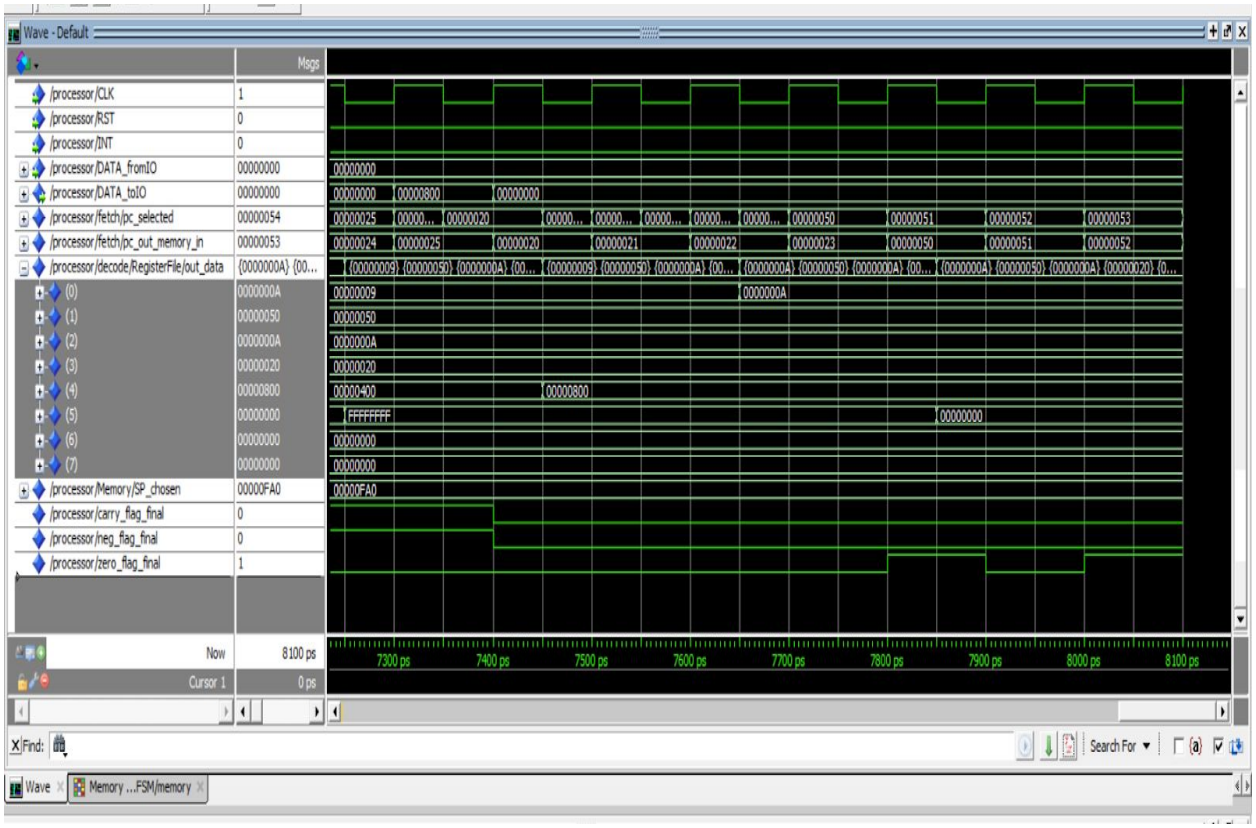

Wave - Default
Msgs
/processor/CLK
/processor/RST
/processor/INT
/processor/DATA_fromIO
/processor/DATA_toIO
/processor/fetch/pc_selected
/processor/fetch/pc_out_memory_in
/processor/decode/RegisterFile/out_data
(0)
(1)
(2)
(3)
(4)
(5)
(6)
(7)
/processor/Memory/SP_chosen
/processor/carry_flag_final
/processor/neg_flag_final
/processor/zero_flag_final
Now
4000 ps
Cursor 1
0 ps
3200 ps
3300 ps
3400 ps
3500 ps
3600 ps
3700 ps
3800 ps
3900 ps
4000 ps
Find:
Search For
Wave
Memory ...FSM/memory


Wave - Default
Msgs
/processor/CLK
/processor/RST
/processor/INT
/processor/DATA_fromIO
/processor/DATA_toIO
/processor/fetch/pc_selected
/processor/fetch/pc_out_memory_in
/processor/decode/RegisterFile/out_data
(0)
(1)
(2)
(3)
(4)
(5)
(6)
(7)
/processor/Memory/SP_chosen
/processor/carry_flag_final
/processor/neg_flag_final
/processor/zero_flag_final
Now
4700 ps
Cursor 1
0 ps
3900 ps
4000 ps
4100 ps
4200 ps
4300 ps
4400 ps
4500 ps
4600 ps
4700 ps
Find:
Search For
Wave
Memory ...FSM/memory

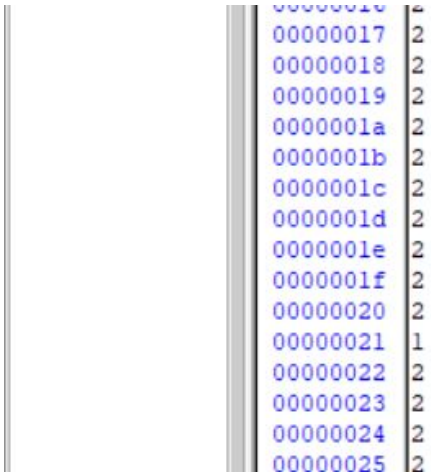

Wave - Default
Msgs
/processor/CLK
/processor/RST
/processor/INT
/processor/DATA_fromIO
/processor/DATA_toIO
/processor/fetch/pc_selected
/processor/fetch/pc_out_memory_in
/processor/decode/RegisterFile/out_data
(0)
(1)
(2)
(3)
(4)
(5)
(6)
(7)
/processor/Memory/SP_chosen
/processor/carry_flag_final
/processor/neg_flag_final
/processor/zero_flag_final
Now
5400 ps
Cursor 1
0 ps
4600 ps
4700 ps
4800 ps
4900 ps
5000 ps
5100 ps
5200 ps
5300 ps
5400 ps
Find:
Search For

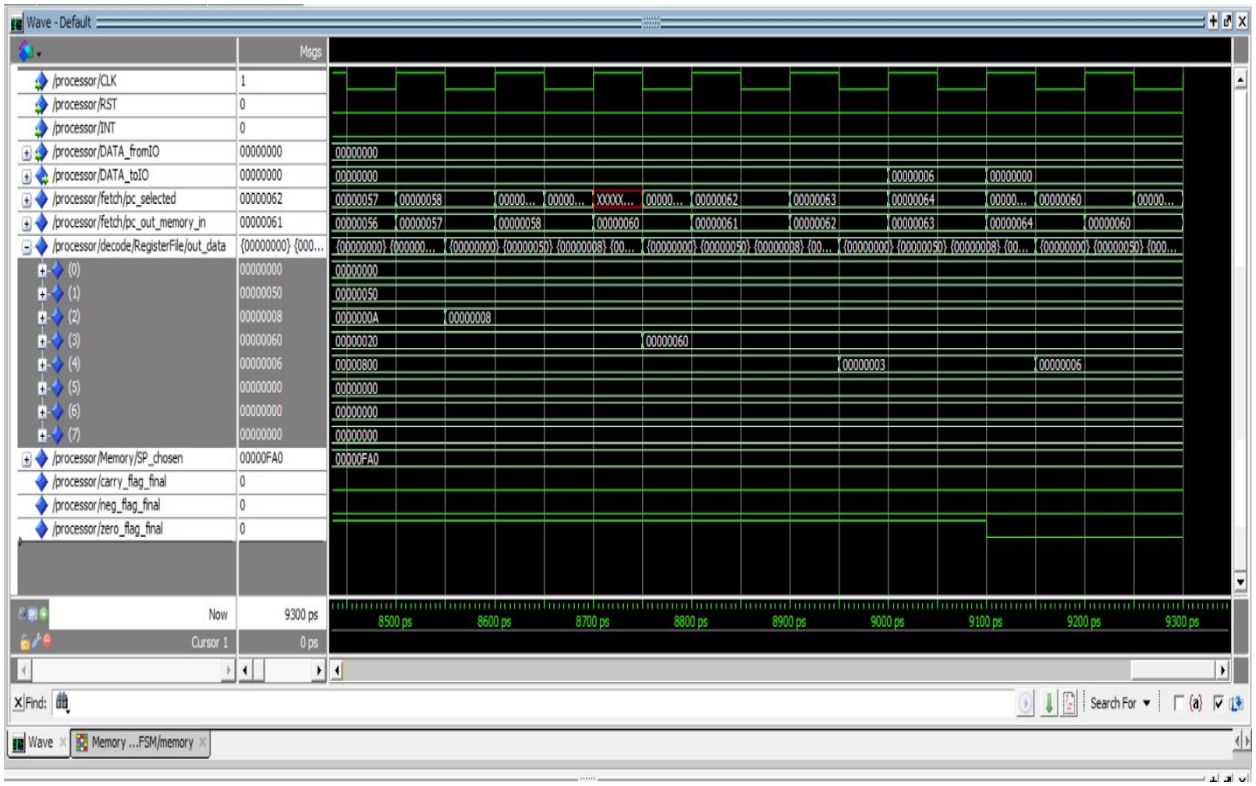

Wave - Default
Msgs
/processor/CLK
/processor/RST
/processor/INT
/processor/DATA_fromIO
/processor/DATA_toIO
/processor/fetch/pc_selected
/processor/fetch/pc_out_memory_in
/processor/decode/RegisterFile/out_data
(0)
(1)
(2)
(3)
(4)
(5)
(6)
(7)
/processor/Memory/SP_chosen
/processor/carry_flag_final
/processor/neg_flag_final
/processor/zero_flag_final
Now
6300 ps
Cursor 1
0 ps
5500 ps
5600 ps
5700 ps
5800 ps
5900 ps
6000 ps
6100 ps
6200 ps
6300 ps
Find:
Search For
Wave
Memory ...FSM/memory


Wave - Default
Msgs
/processor/CLK
/processor/RST
/processor/INT
/processor/DATA_fromIO
/processor/DATA_toIO
/processor/fetch/pc_selected
/processor/fetch/pc_out_memory_in
/processor/decode/RegisterFile/out_data
(0)
(1)
(2)
(3)
(4)
(5)
(6)
(7)
/processor/Memory/SP_chosen
/processor/carry_flag_final
/processor/neg_flag_final
/processor/zero_flag_final
Now
7200 ps
Cursor 1
0 ps
6400 ps
6500 ps
6600 ps
6700 ps
6800 ps
6900 ps
7000 ps
7100 ps
7200 ps
Find:
Search For
Wave
Memory ...FSM/memory

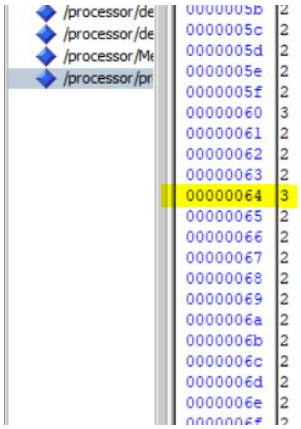

Wave - Default
Msgs
/processor/CLK
/processor/RST
/processor/INT
/processor/DATA_fromIO
/processor/DATA_toIO
/processor/fetch/pc_selected
/processor/fetch/pc_out_memory_in
/processor/decode/RegisterFile/out_data
(0)
(1)
(2)
(3)
(4)
(5)
(6)
(7)
/processor/Memory/SP_chosen
/processor/carry_flag_final
/processor/neg_flag_final
/processor/zero_flag_final
Now
8100 ps
Cursor 1
0 ps
7300 ps
7400 ps
7500 ps
7600 ps
7700 ps
7800 ps
7900 ps
8000 ps
8100 ps
Find:
Search For
Wave
Memory ...FSM/memory

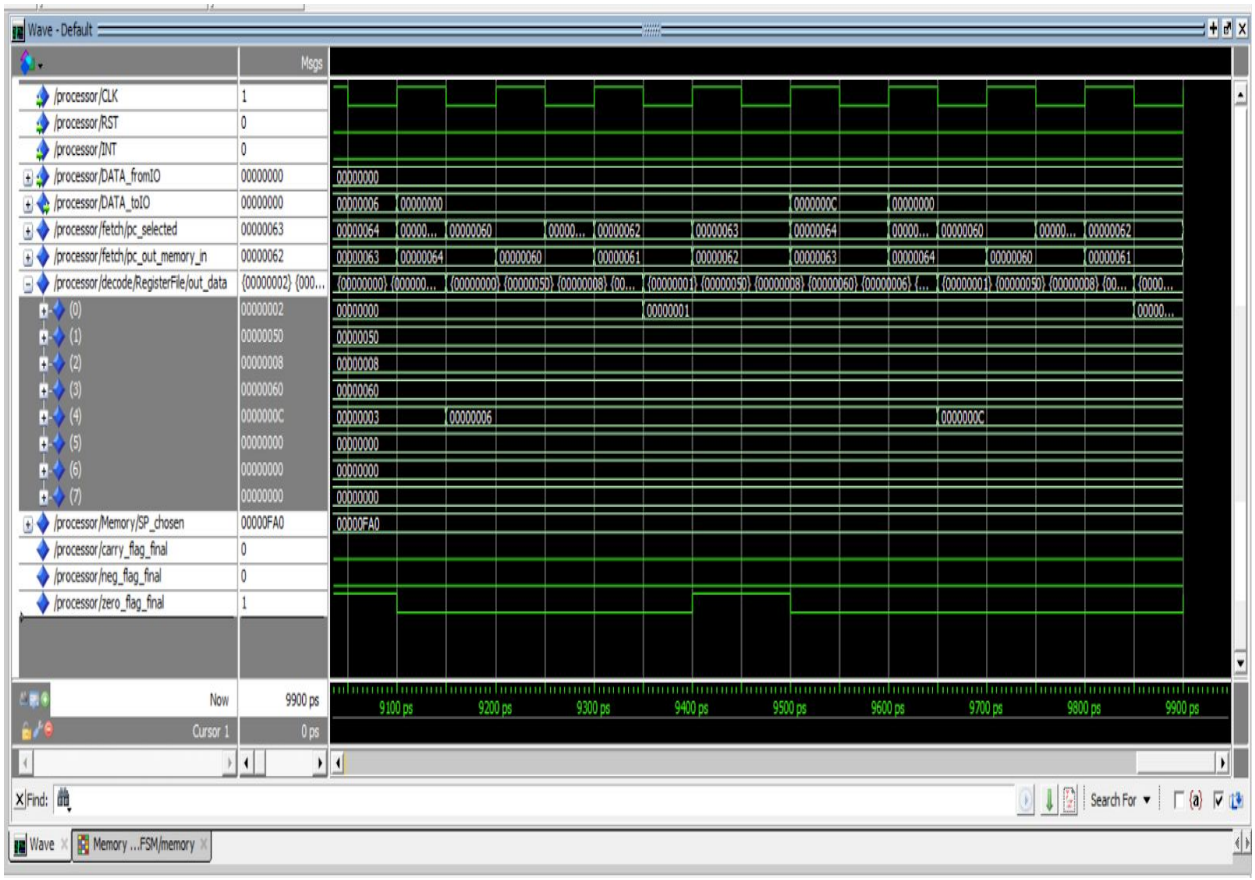
In the previous figure we can notice that we predict the JZ as not taken and then correct our prediction to go to 50 and move this jump from 0 (Strong not taken) to 1 (weak not taken)

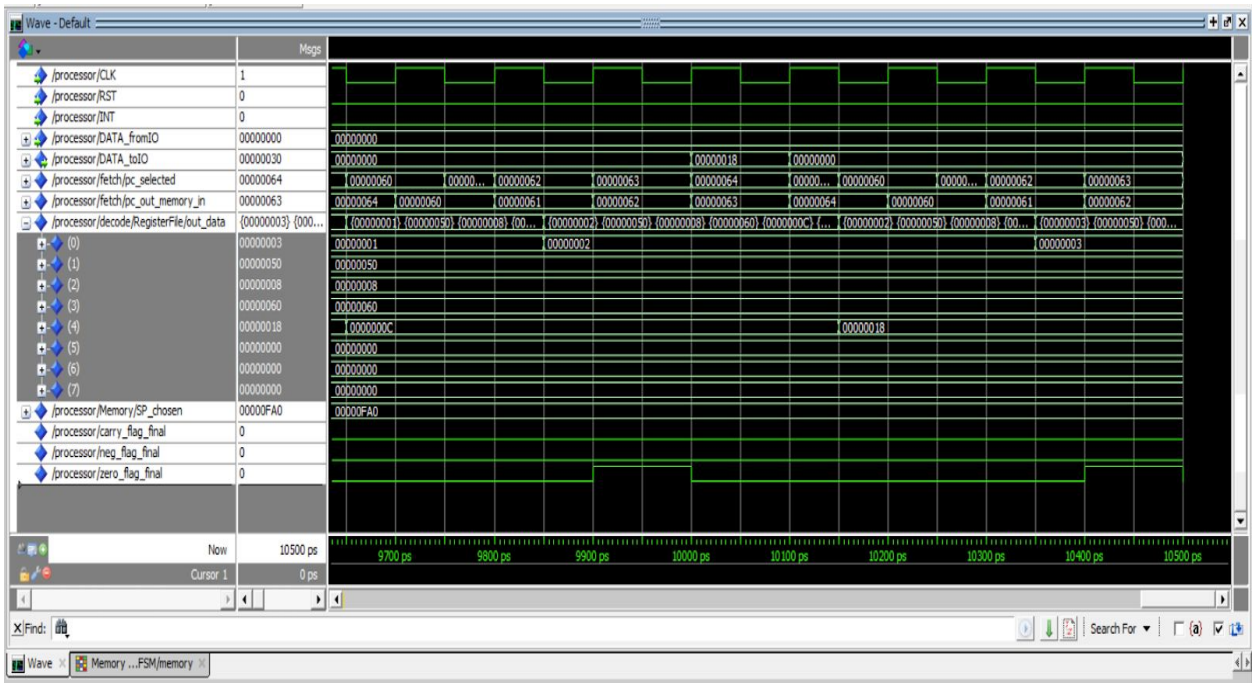


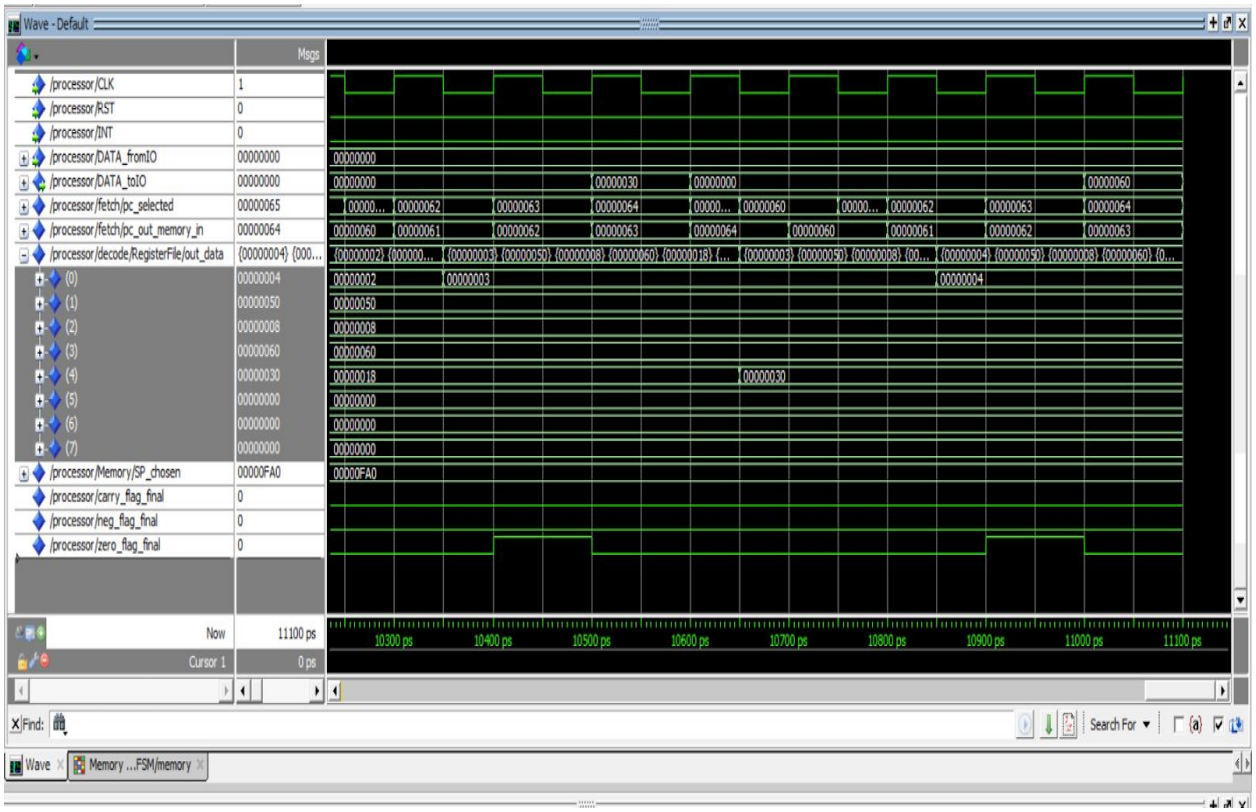


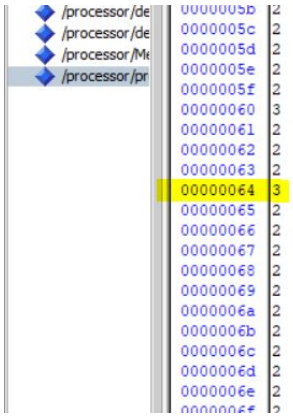
Here we predicted this jump as weak taken and our prediction is true so we update its state from 2 (weak taken) to 3 (strong taken).

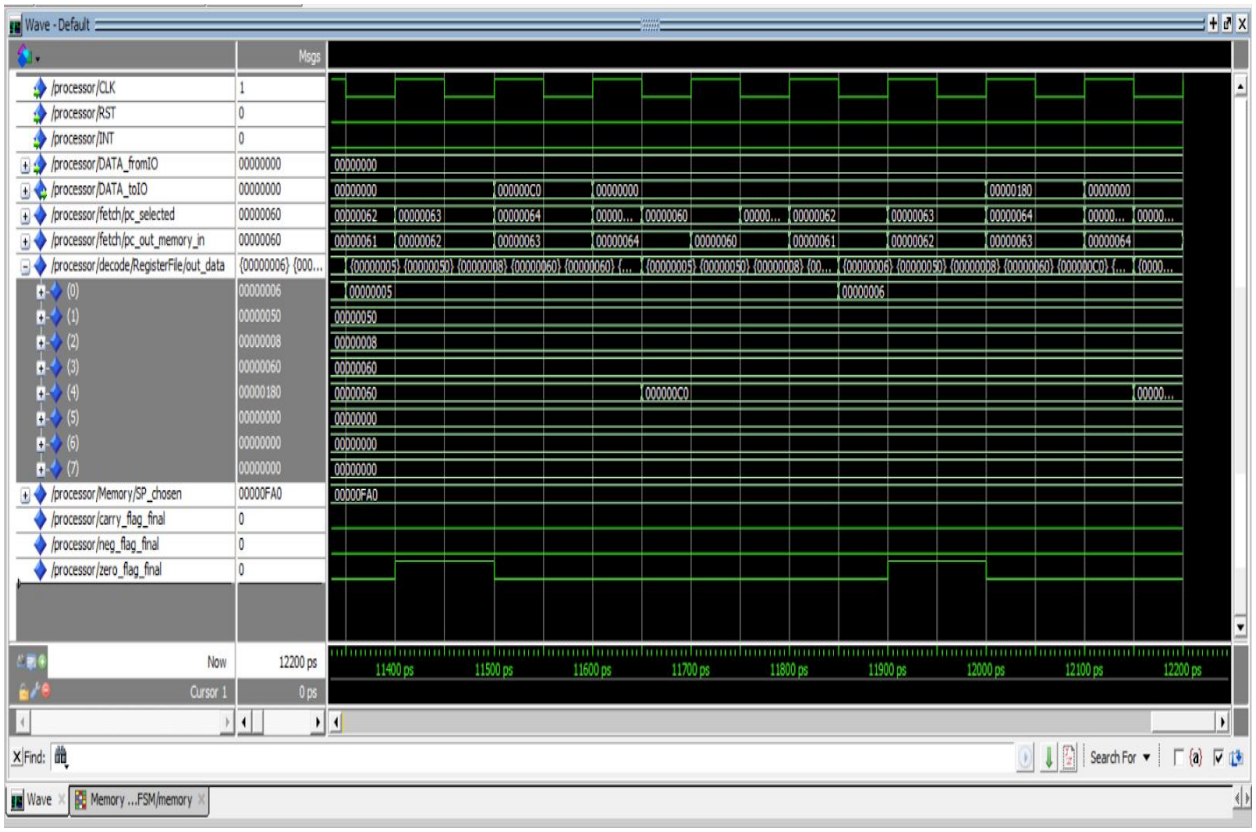




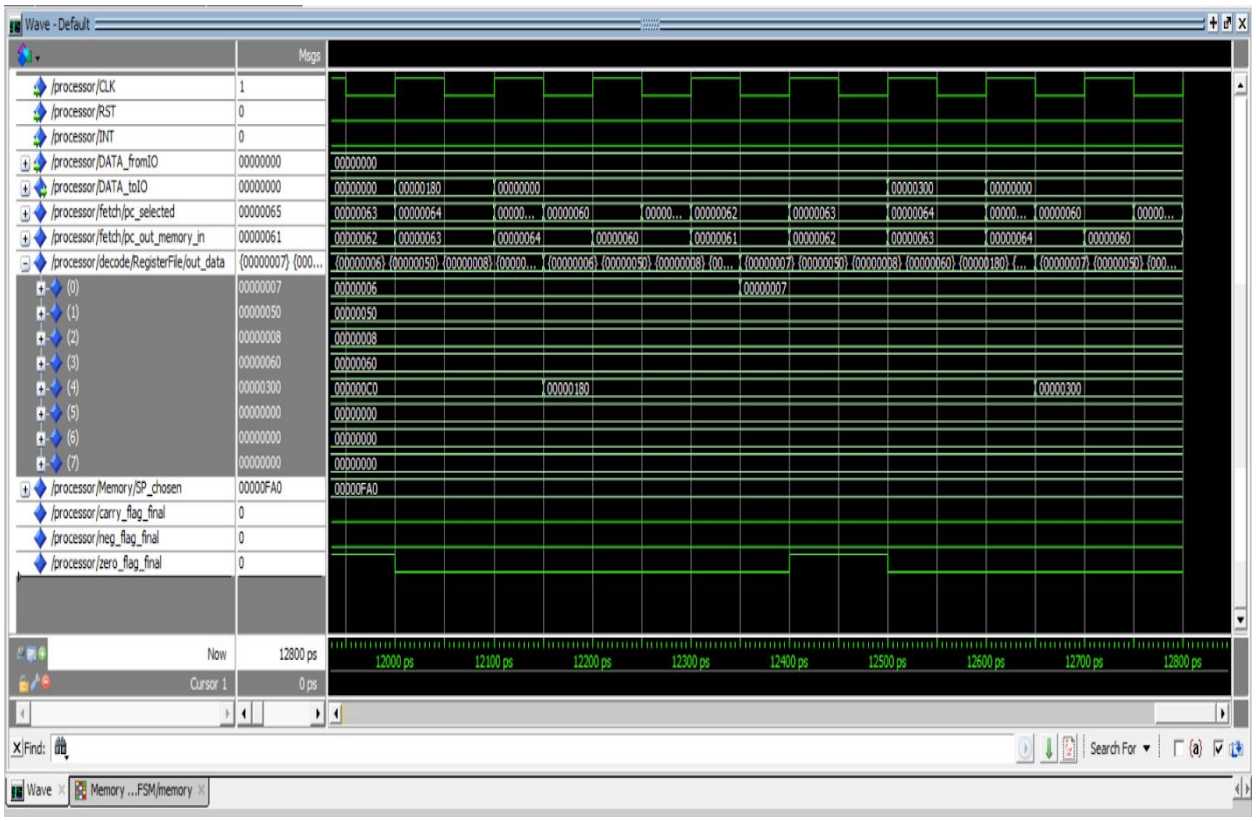

Wave - Default
Msgs
/processor/CLK
/processor/RST
/processor/INT
/processor/DATA_fromIO
/processor/DATA_toIO
/processor/fetch/pc_selected
/processor/fetch/pc_out_memory_in
/processor/decode/RegisterFile/out_data
(0)
(1)
(2)
(3)
(4)
(5)
(6)
(7)
/processor/Memory/SP_chosen
/processor/carry_flag_final
/processor/neg_flag_final
/processor/zero_flag_final
Now
10500 ps
Cursor 1
0 ps
9700 ps
9800 ps
9900 ps
10000 ps
10100 ps
10200 ps
10300 ps
10400 ps
10500 ps
Find:
Search For
Wave
Memory ...FSM/memory

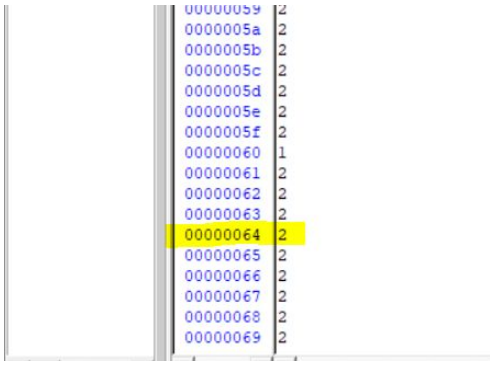

Wave - Default
Msgs
/processor/CLK
/processor/RST
/processor/INT
/processor/DATA_fromIO
/processor/DATA_toIO
/processor/fetch/pc_selected
/processor/fetch/pc_out_memory_in
/processor/decode/RegisterFile/out_data
(0)
(1)
(2)
(3)
(4)
(5)
(6)
(7)
/processor/Memory/SP_chosen
/processor/carry_flag_final
/processor/neg_flag_final
/processor/zero_flag_final
Now
11100 ps
Cursor 1
0 ps
10300 ps
10400 ps
10500 ps
10600 ps
10700 ps
10800 ps
10900 ps
11000 ps
11100 ps
Find:
Search For
Wave
Memory ...FSM/memory



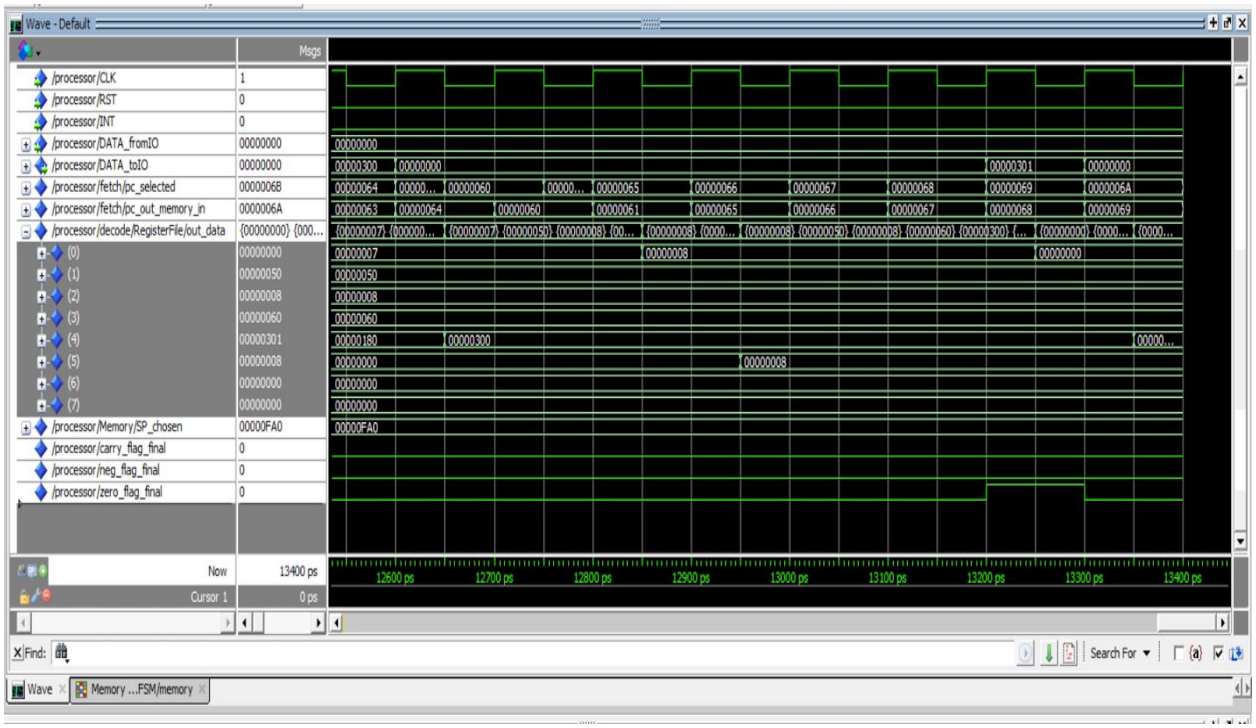


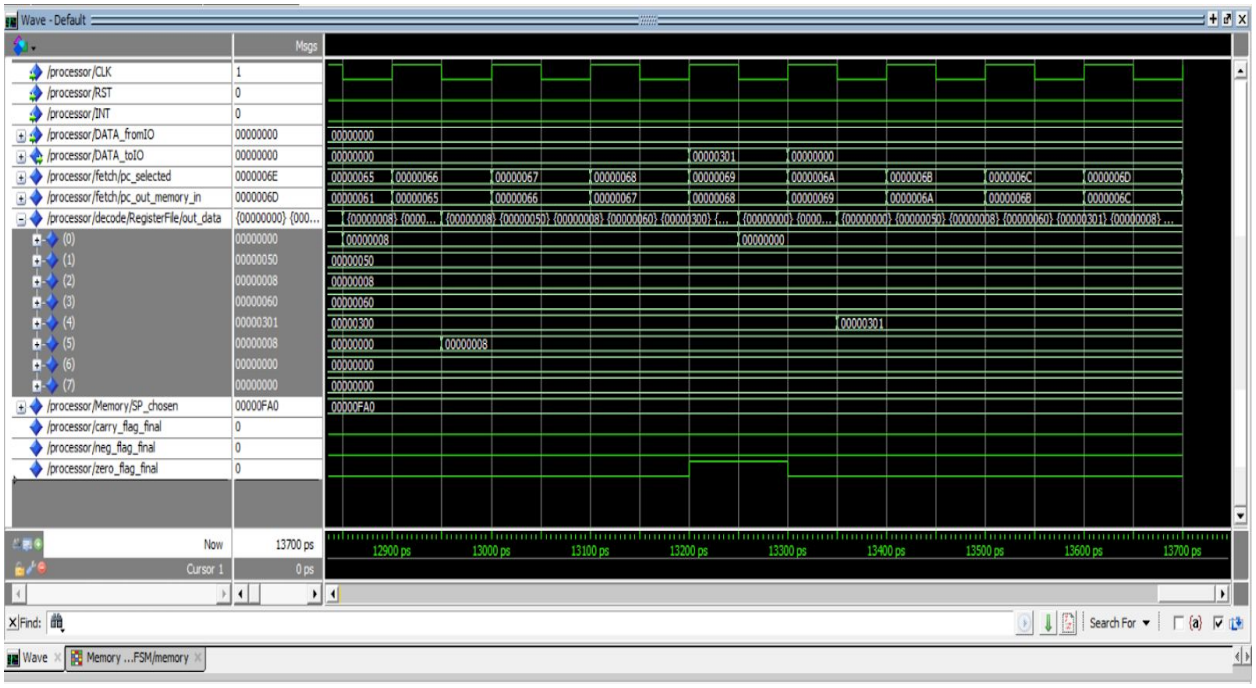
Here we can see out value before it be written in the register file which mean forwarding unit works properly and the loop will continue as we will see in the figures until R0 < R2(8)





Here we end the second loop and correct the last prediction from strong taken to weak taken





Here we are finishing our program as the data inside R4 incremented by 1 and appears in the out and data saved in the register file.