Birla Institute of Technology and Science, Pilani

K K Birla Goa Campus

CS F342 Computer Architecture

Semester 2, Year 2021-22

Project Report

Design of Five Stage Pipelined MIPS Processor with Hazard Detection and Mitigation Abilities

Group 8

Rohit Rahul Mundada	2019AAPS0343G
Sparsh Kanubhai Kachhadiya	2019A8PS0491G
Vedant Sachin Bang	2019AAPS0251G

Data Hazards

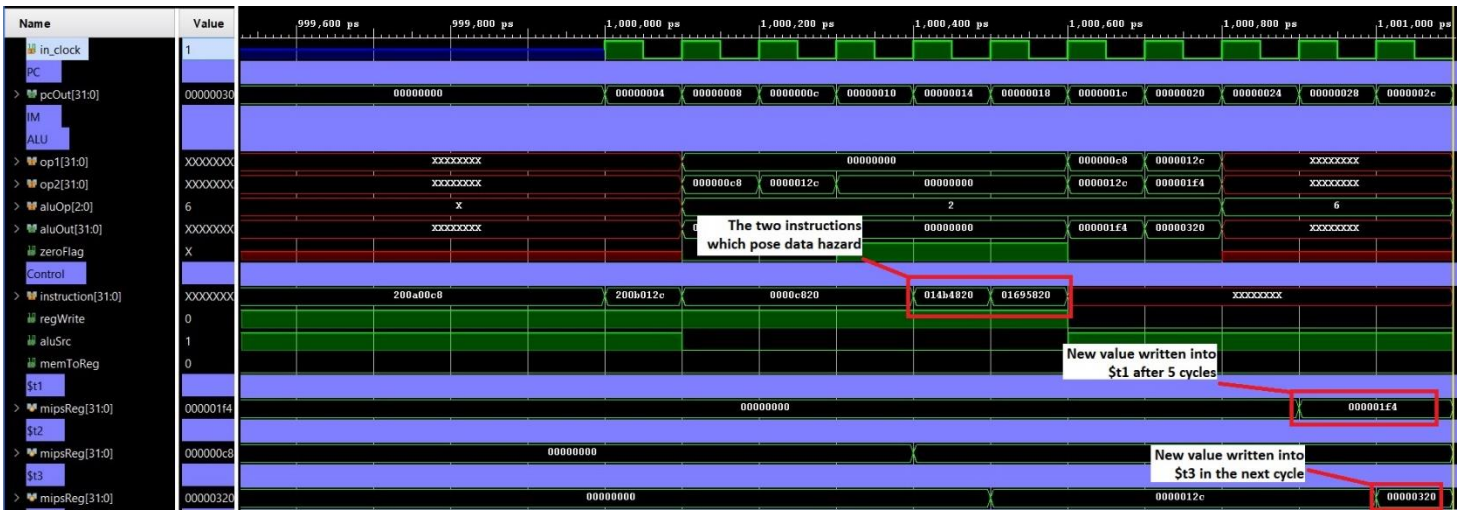
There are certain patterns of instructions which require access to the Register File or the Data Memory in succession to one another. Without hazard mitigation, the instructions will end up operating on older data values from the registers or the memory. We have classified the hazards based on these patterns.

1. Hazards of the type where successive instructions write and read from the same register in the register file:

Consider the example:

```
addi $t2,$zero,200
addi $t3,$zero,300
add $t9,$zero,$zero  // in place of nop
add $t9,$zero,$zero  // in place of nop
add $t9,$zero,$zero  // in place of nop
add $t1,$t2,$t3
add $t3,$t3,$t1
```

The last two instructions pose a data hazard. We have solved this by forwarding the value of the ALU result to the ALU inputs in the previous stage, behind the pipe.

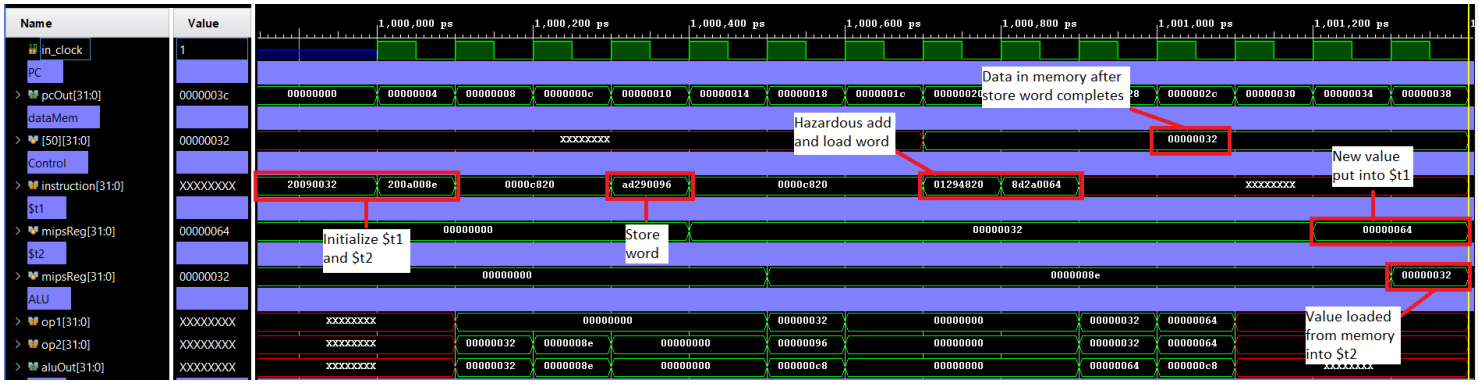


Another example of the hazard where register file is successively written and read:

```
addi $t1,$zero,50
addi $t2,$zero,142
add $t9,$zero,$zero     // in place of nop
add $t9,$zero,$zero     // in place of nop
sw $t1,150($t1)
add $t9,$zero,$zero     // in place of nop
add $t9,$zero,$zero     // in place of nop
add $t9,$zero,$zero     // in place of nop
```

```
add $t1,$t1,$t1
lw $t2,100($t1)
```

$t1 and $t2 are initialized with certain values. The value of $t1 is stored at memory location 200 ($t2+150). Then $t1 is modified and subsequently used for load word. The last 2 instructions pose a data hazard. It is solved by taking the ALU result passed through the EX-pipeline and forwarding it back to inputs of ALU. The input MUXs of ALU are appropriately controlled by a forwarding unit.

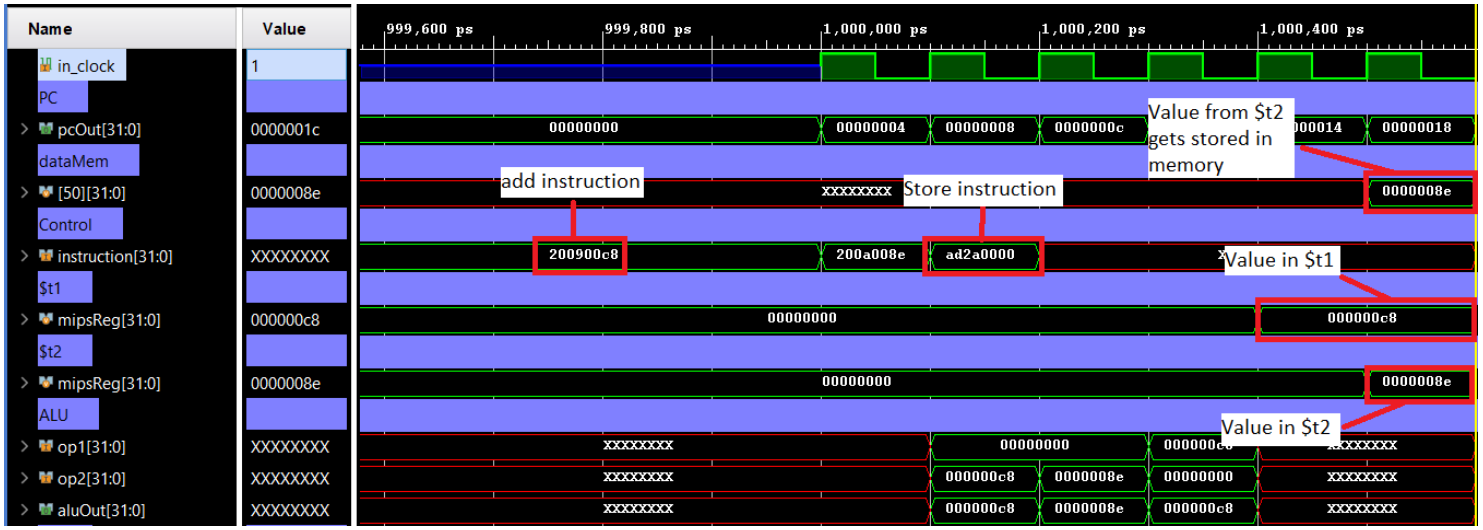


---

2. Hazards where an instruction attempts to read from a register, one instruction after it has been written into:

```
addi $t1,$zero,200
addi $t2,$zero,142
sw $t2,($t1)
```

The first and third instructions pose a hazard. (The second and third also pose a hazard but that has been resolved in the previous examples). To solve this hazard, we have forwarded the value of the ALU result from the last stage (writeback) to the input of the ALU. The input MUXs of ALU are appropriately controlled by forwarding unit.



Consider another example:

```
addi $t2,$zero,150
addi $t1,$zero,50
addi $t3,$zero,173
sw $t3,150($t1)
add $t2,$t2,$t1
add $t9,$zero,$zero // in place of nop
lw $t1,($t2)
```

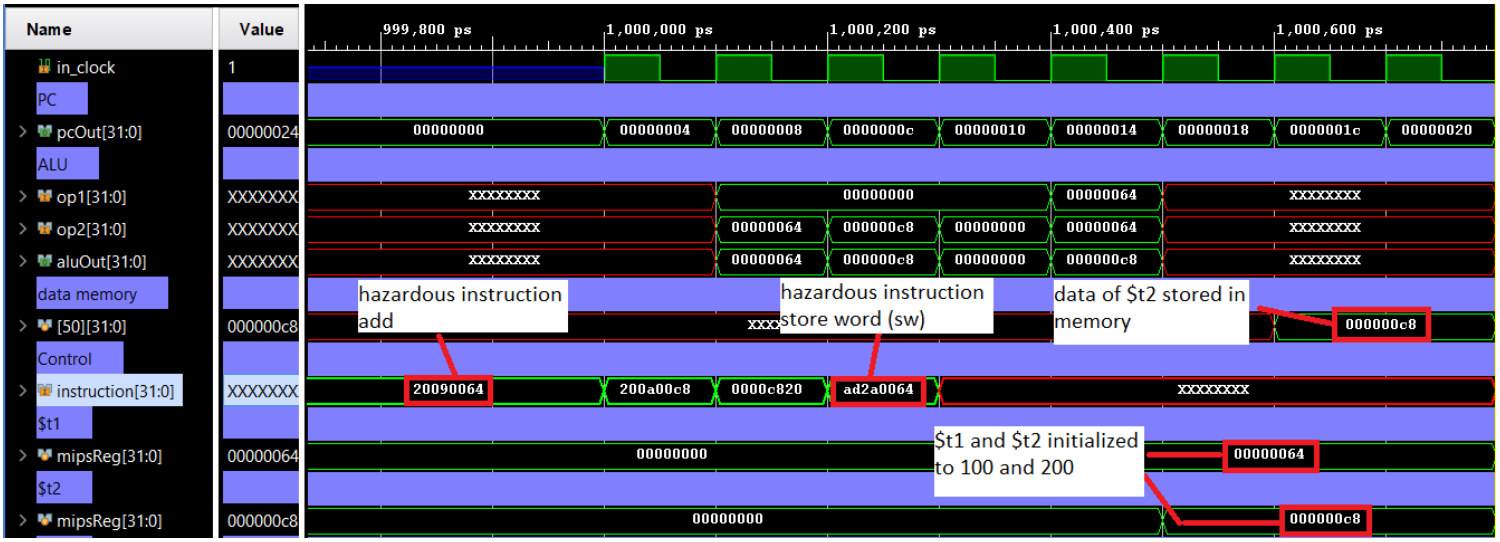
This program contains multiple hazards. Focusing on the fifth and seventh instructions, the hazard is solved by forwarding the value of ALU result from the writeback stage to the input of the ALU. The input MUXs of ALU are appropriately controlled by forwarding unit.

---

3. Hazards where an instruction attempts to read from a register, two instructions after it is written into:

```
addi $t1,$zero,100
addi $t2,$zero,200
add $t9,$zero,$zero // in place of nop
sw $t2,100($t1)
```

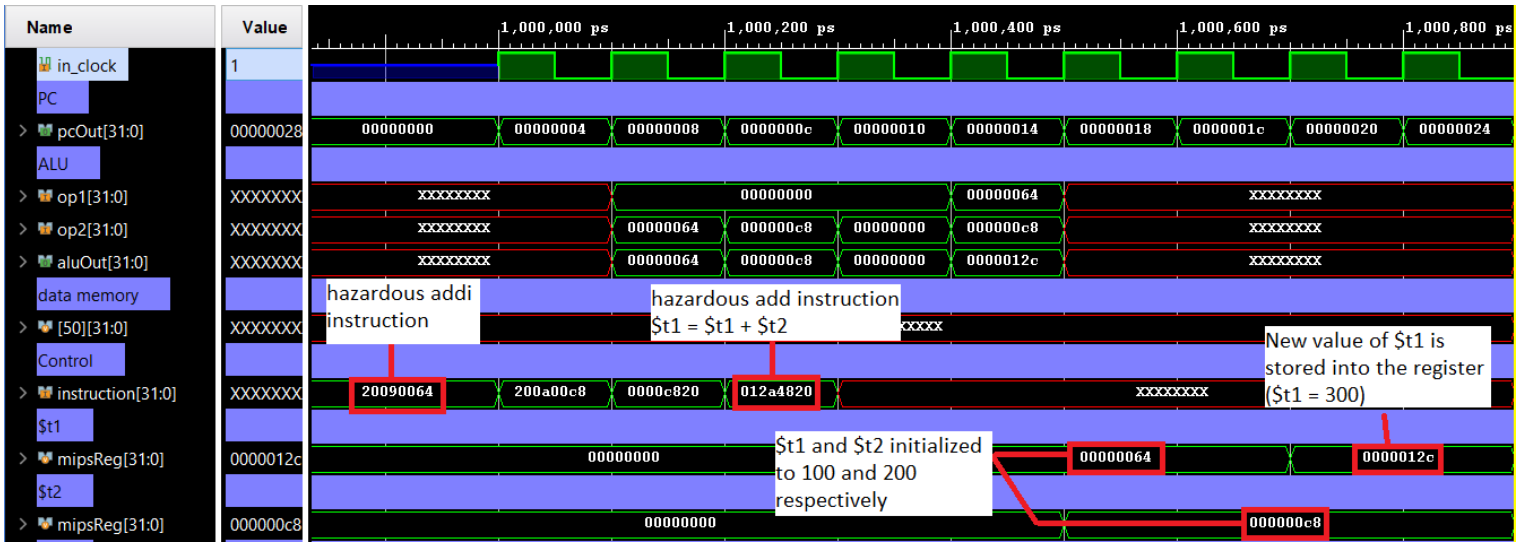
The hazard posed by first and fourth instruction is solved by forwarding the ALU or Data Memory result from the writeback stage into the MUXs connected at the outputs of the Register File. These MUXs give their output to the next pipeline. The data is thus directly taken from the writeback stage, since an additional clock cycle will be required by the Register File to write new values into its registers.



Consider another example with similar hazard, but with an add instruction:

```
addi $t1,$zero,100
addi $t2,$zero,200
add $t9,$zero,$zero // in place of nop
sw $t2,100($t1)
```

Here also the logic applied is similar to the previous case. The value from the writeback stage is forwarded back to the output of the Register file as the Register file takes an extra clock cycle to actually store the data into the registers.



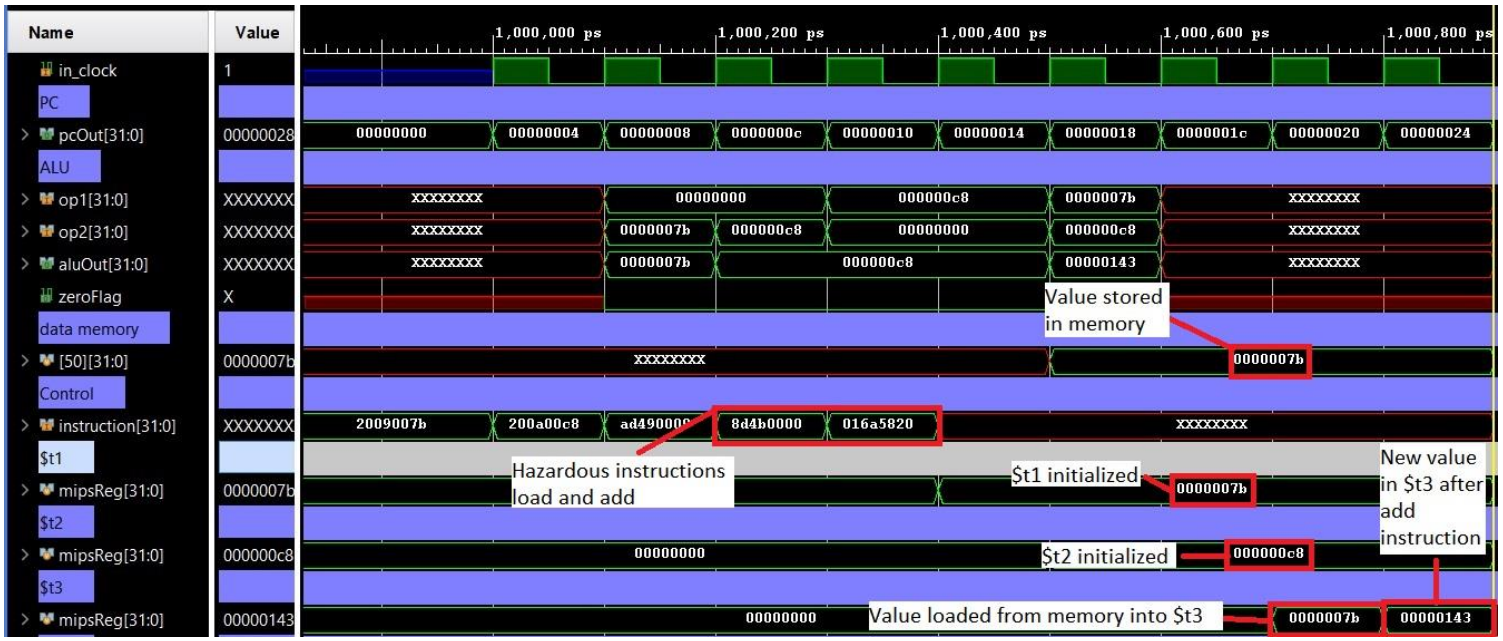
---

4. Another hazard occurs when certain data is loaded from memory into register and immediately required in the next instruction:

```
addi $t1,$zero,123
addi $t2,$zero,200
```

```
sw $t1,($t2)
lw $t3,($t2)
add $t3,$t3,$t2
```

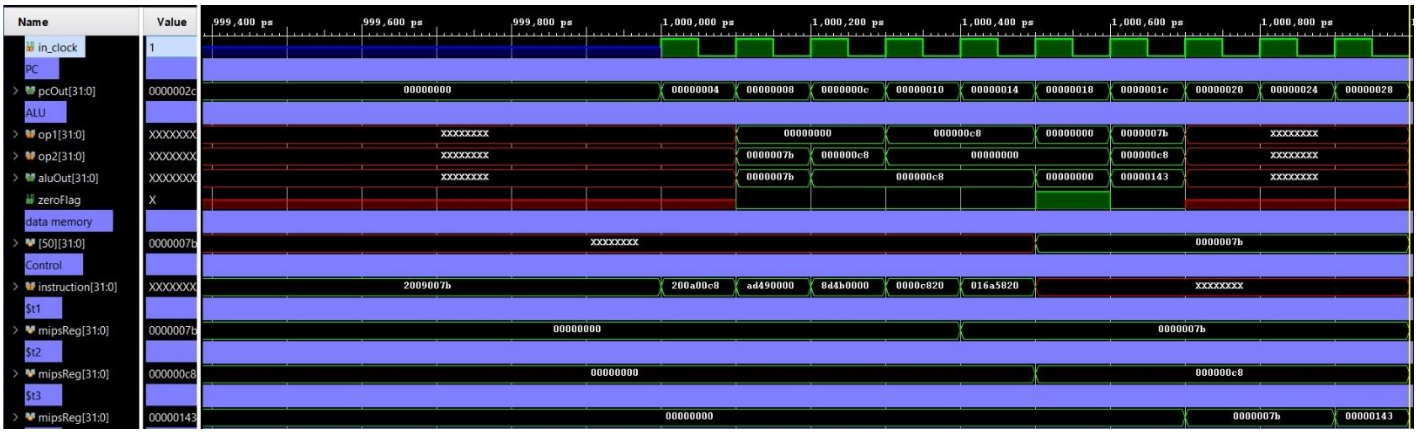
The last two instructions pose a hazard. We have solved this by forwarding the output from the data memory from the memory stage, back to the input of the ALU. The input MUXs are appropriately controlled by forwarding unit.



A variation of the previous hazard is loading data from the memory and accessing the register one instruction after:

```
addi $t1,$zero,123
addi $t2,$zero,200
sw $t1,($t2)
lw $t3,($t2)
add $t9,$zero,$zero // in place of nop
add $t3,$t3,$t2
```

This hazard is solved by forwarding the output from data memory, taken from the writeback stage, into the inputs of the ALU. The input MUXs are appropriately controlled by forwarding units.



Control Hazards

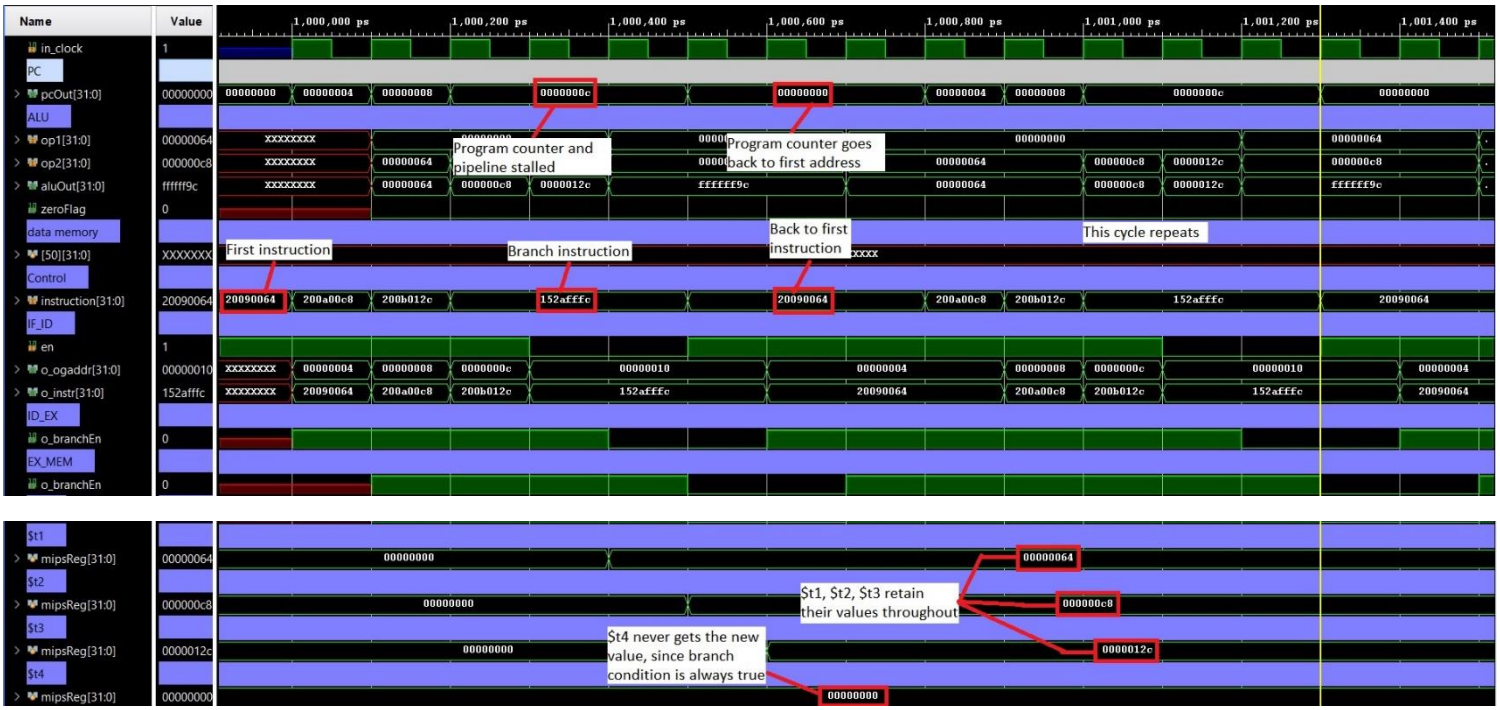
In our MIPS processor, the only kind of control hazard that can come is when a branch instruction is used. In our implementation, we are stalling the IF/ID (fetch-decode) pipeline as well as the program counter when we see that the instruction is a branch instruction. The enable signal of the fetch pipeline is kept low for 2 clock cycles because the branch instruction and whether to branch or not is known at the EX/MEM (execute-memory) stage where the branch address is calculated and value of zero flag is known. Based on the value of zero flag it is decided whether to take the branch or not. Following are two scenarios during a branch instruction:

1. Branch is taken:

Consider the example:

```
begin:addi $t1, $zero, 100
addi $t2, $zero, 200
addi $t3, $zero, 300
bne $t1, $t2, begin
add $t4, $t3, $t2
```

The branch condition is false hence program keeps looping.



---

2. Branch is not taken:

Consider the example:

```
begin:addi $t1, $zero, 200
addi $t2, $zero, 200
addi $t3, $zero, 300
```

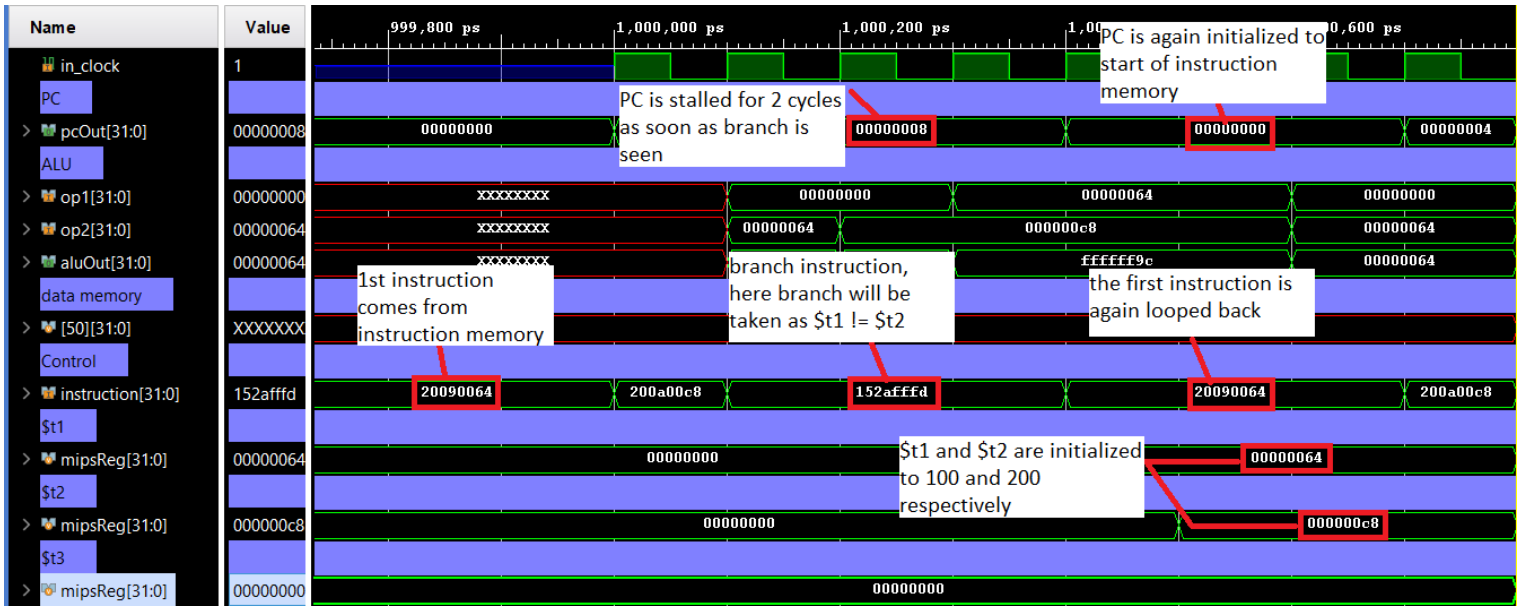
```
bne $t1, $t2, begin
add $t4, $t3, $t2
```

The branch condition is false hence $t4 gets a new value.

Consider another example which includes a branch as well as jump instruction:

```
begin:addi $t1,$zero,100
addi $t2,$zero,200
bne $t1, $t2, begin // code below this should not work
j end
add $t3, $t1, $t2
end:add $t2, $t1, $t1
```

Here first branch is taken so the jump instruction never comes from the instruction memory. The program keeps looping between the first three instructions.



Now we take the case when branch is not taken:

```
begin:addi $t1,$zero,200
addi $t2,$zero,200
bne $t1, $t2, begin
j end
add $t3, $t1, $t2 // this instruction is not to be executed
end:add $t2, $t1, $t1
```

The jump instruction is loaded from the instruction memory to the pipeline. The jump instruction then recalculates a new value for instruction address and sends to PC. The instruction at the new address is executed and the result is stored in the appropriate register. The instruction just after the jump is skipped.