CprE 381 Homework 7

[Note: The first couple of questions are intended to help you familiarize yourself with pipelining. The third question will help you understand data hazards and how to mitigate them. The final question will help you prepare for the exam by looking through lecture notes and online quizzes and then thinking which questions I might ask.]

1. Pipeline Simulation
   For each of the modules from Figure 4.51 (ZyBooks Figure 4.7.16) that are listed in the table below, specify what the inputs and outputs are in each cycle for the following code. You do not need to specify values for those modules, inputs, or outputs not listed in the table. Manually simulate (i.e., fill out the table) until the **`sw`** has completed (i.e., left the write-back stage). *[Hint: the table already has the first couple of cycles filled out. If a value depends on an instruction before or after the code below, report it as X.]*

Cycle	Instruction Memory		Register File				ALU				MemtoReg MUX			PCSrc MUX		
	Addr	Instr	Read reg 1	Read data 1	Write reg	Write data	A	B	Op (e.g, add, sub)	ALU result	1	0	s	1	0	s
1	0x00000010	addi …	X	X	X	X	X	X	X	X	X	X	X	X	0x00000014	0
2	0x00000014	lui …	0x00	0x00000000	X	X	X	X	X	X	X	X	X	X	0x00000018	0
3	0x00000018	xor…	0x00	0x00000000	X	X	0x00000000	0x0000003F	addi	0x0000003F	X	X	X	X	0x0000001C	0
4	0x0000001C	sub …	0x02	0x00000000	X	X	0x00000000	0x00001010	lui	0x10100000	X	X	X	X	0x00000020	0
5	0x00000020	sll …	0x04	0x00000003	0x0F	0x0000003F	0x000003FF	0xFFFFFFFF	xor	0xFFFFFC00	X	0x0000003F	0	X	0x00000024	0
6	0x00000024	ori …	0x00	0x00000000	0x17	0x10100000	0x00000003	0x00000400	sub	0xFFFFFC03	X	0x10100000	0	X	0x00000028	0
7	0x00000028	beq …	0x17	0x10100000	0x0D	0xFFFFFC00	0x00000000	0x00000000	sll	0x00000000	X	0xFFFFFC00	0	X	0x0000002C	0
8	0x0000002C	sll …	0x07	0xFFFFFFFF	0x0E	0xFFFFFC03	0x10100000	0x00000040	ori	0x00001050	X	0xFFFFFC03	0	X	0x00000030	0
9	0x00000030	sll …	0x00	0x00000000	0x00	0x00000000	0x0000003F	0xFFFFFFFF	sub	0x00000040	X	0x00000000	0	X	0x00000034	0
10	0x00000034	sll …	0x00	0x00000000	0x17	0x10100040	0x00000000	0x00000000	sll	0x00000000	X	0x00001050	0	X	0x00000038	0
11	0x00000038	sw	0x00	0x00000000	X	X	0x00000000	0x00000000	sll	0x00000000	X	0x00000040	0	X	0x0000003C	0
12	0x0000003C	X	0x17	0x10100040	0x00	0x00000000	0x00000000	0x00000000	sll	0x00000000	X	0x00000000	0	X	0x00000040	0
13	0x00000040	X	X	X	0x00	0x00000000	0x10100040	0x00000000	add	0x10100040	X	0x00000000	0	X	0x00000044	0
14	0x00000044	X	X	X	0x00	0x00000000	X	X	X	X	X	0x00000000	0	X	0x00000048	0
15	0x00000048	X	X	X	0x0F	0x10100040	X	X	X	X	X	0x10100040	0	X	0x0000004C	X

[Pipelined MIPS – Simulation Table]

```
# Assume that $a0 = 3, $a1 = 1024, $a2 = 1023, $a3 = -1
# at the start of your manual simulation.
# Assume that lui is supported by the lui operation in
# the ALU and that the value shifted for lui is the B
```

```
# input of the ALU (note that this is likely different
# than your project implementation and that's OK).
# The following instructions start at address 0x00000010:
addi $t7, $zero, 63
lui  $s7, 0x1010
xor  $t5, $a2, $a3
sub  $t6, $a0, $a1
sll  $zero, $zero, 0
ori  $s7, $s7, 0x0040
beq  $t7, $a3, Exit
sll  $zero, $zero, 0
sll  $zero, $zero, 0
sll  $zero, $zero, 0
sw   $t7, 0($s7)
...
Exit: # This label resolves to address 0x00000100.
```

2. Data Dependencies, Hazard Detection, and Hazard Avoidance
   *[**WARNING**: The datapath described and used here is intentionally different from your project's datapath. The goal is for you to be able to understand the relationship between datapath design and hazards.]*

   a. Identify all the read after write data dependencies in the following code. Use the format (reading instruction address, register read/written, writing instruction addr).

```
# The following code starts at address 0x00400010
addiu $t0, $zero, 0     # clear i ($t0)
j     cond
loop:
addu  $t1, $a0, $t0
lb    $t1, 0($t1)       # load value to histogram
addu  $t1, $a3, $t1     # calculate histogram bin
lb    $t2, 0($t1)       # load bin value
addiu $t2, $t2, 1       # increment bin value
sb    $t2, 0($t1)       # store bin value
addiu $t0, $t0, 1       # increment i
cond:
sltu  $t1, $t0, $a1     # loop condition check (N in $a1)
bne   $t1, $zero, loop
check:
addu  $t1, $a3, $a2     # check bin value at input ($a2)
location
lb    $t1, 0($t1)
sltiu $t1, $t1, 100
bne   $t1, $zero, Exit
jal   detection         # jump to function (not shown)
Exit:
```

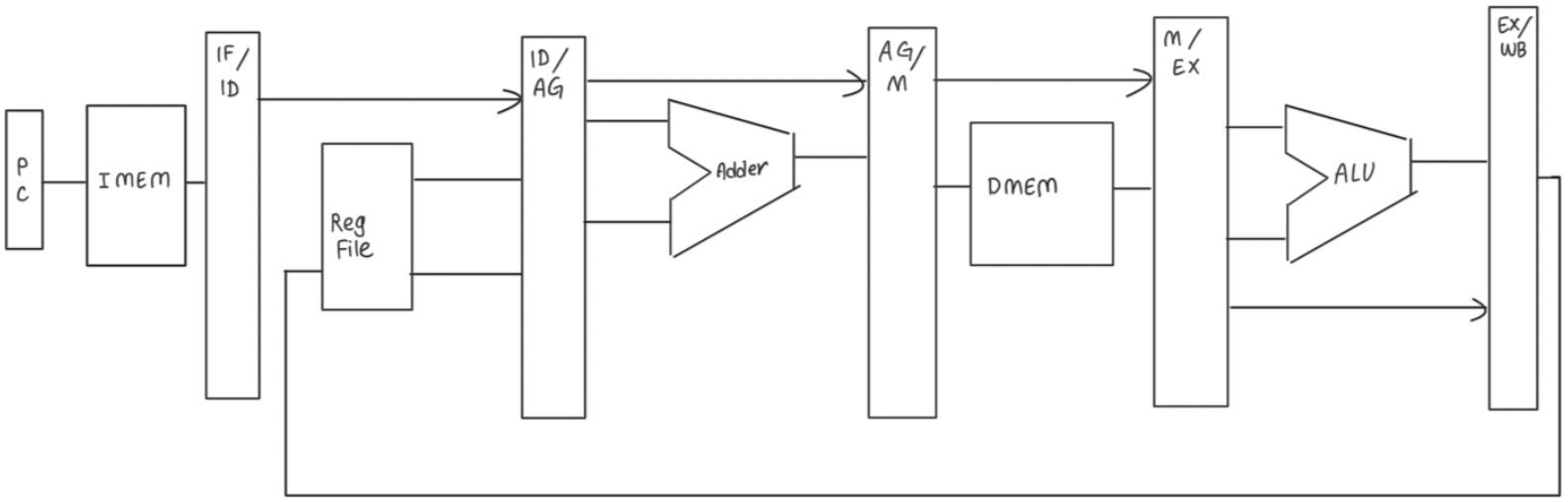
Reading Inst Addr	Reg Read/Written	Writing Inst Addr
0x00400034	$t0	0x00400010
0x00400038	$t1	0x00400034
0x00400018	$t0	0x00400010
0x0040001C	$t1	0x00400018
0x00400020	$t1	0x0040001C
0x00400024	$t1	0x00400020
0x00400028	$t2	0x00400024
0x0040002C	$t1	0x00400020
0x00400030	$t0	0x00400010
0x00400034	$t0	0x00400030
0x00400040	$t1	0x0040003C
0x00400044	$t1	0x00400040
0x00400048	$t1	0x00400044
0x0040001C	$t1	0x0040001C
0x00400030	$t0	0x00400030
0x0040002C	$t2	0x00400028
0x00400018	$t0	0x00400030

b. The 5-stage MIPS pipeline described in lecture (and used for your term project) is not the only possible pipeline design. Consider the six stages described in the table below (this ordering is actually similar to the original MIPS pipeline described in a 1981 research paper). Using your answer to part 3a., please identify all possible data hazards occurring in the eight instructions following the loop label assuming the register file reads the new value being written into a specific register in a given cycle. Justify your answer with a pipeline diagram that shows possible backwards dependencies.

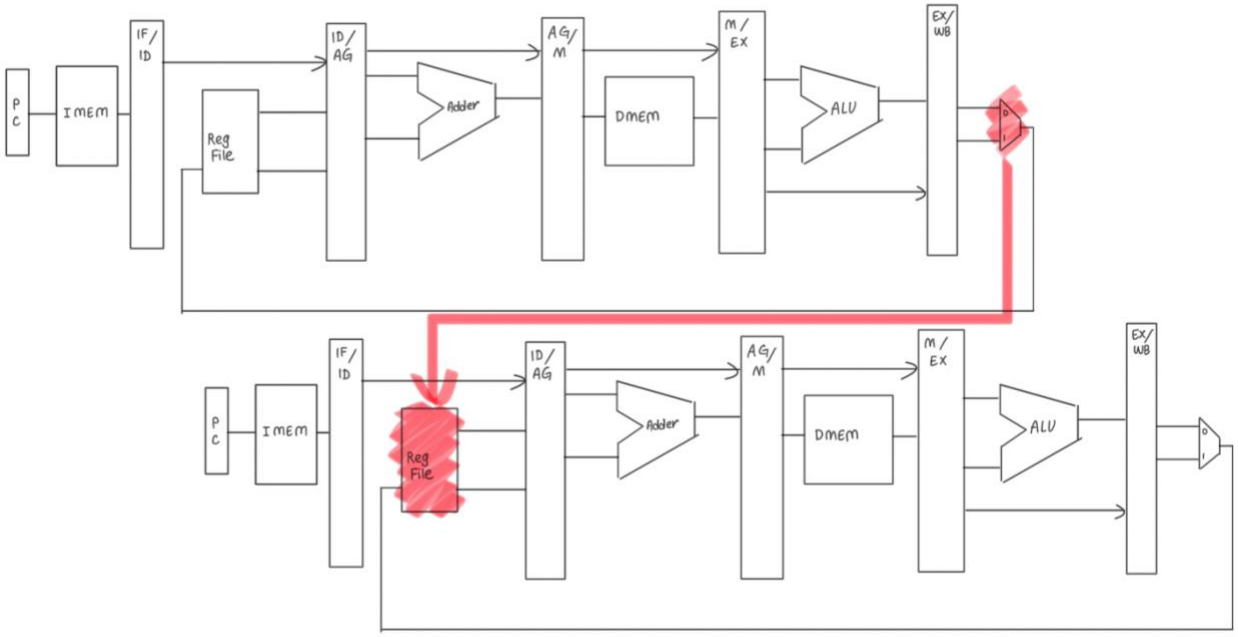
[Note: Only compute any data hazards that occur from the instruction at address 0x00400018 through address 0x00400034.]

Stage Order	**Stage Abbreviation**	**Stage Function**
1	IF (Instruction Fetch)	Load Instruction from M[PC]; increment PC
2	ID (Instruction Decode)	Generate control signals for instruction; read register operands; branch and jump Next PCs determined
3	AG (Address Generation)	Calculate addresses for memory operations
4	M (Memory Access)	Access memory
5	EX (Execution)	Perform any ALU operations

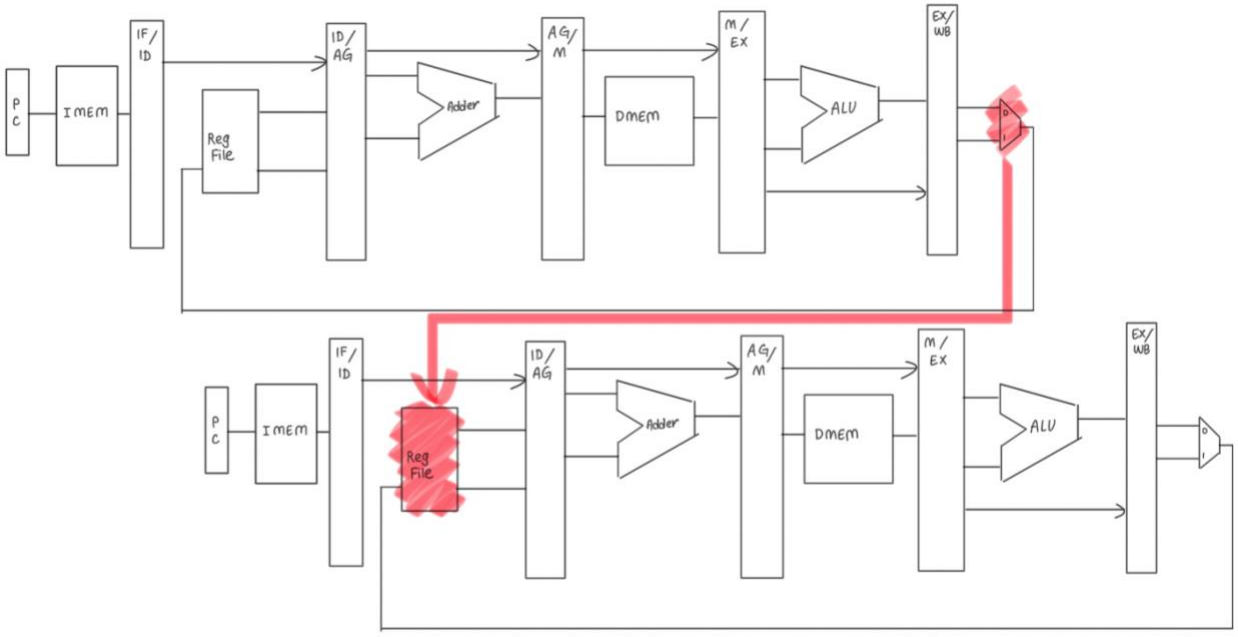
6	WB (Writeback)	Write results (ALU or memory operations) back to registers



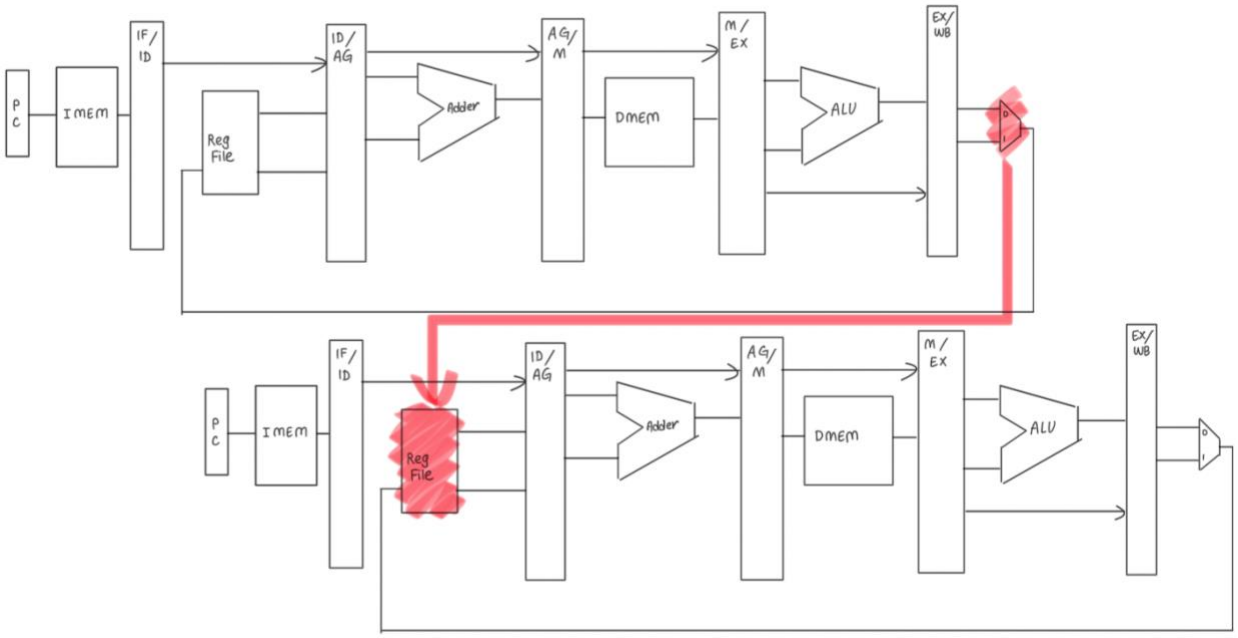

```
1. addu   $t1, $a0, $t0    # $t1 is a data hazard
   lb     $t1, 0($t1)
```



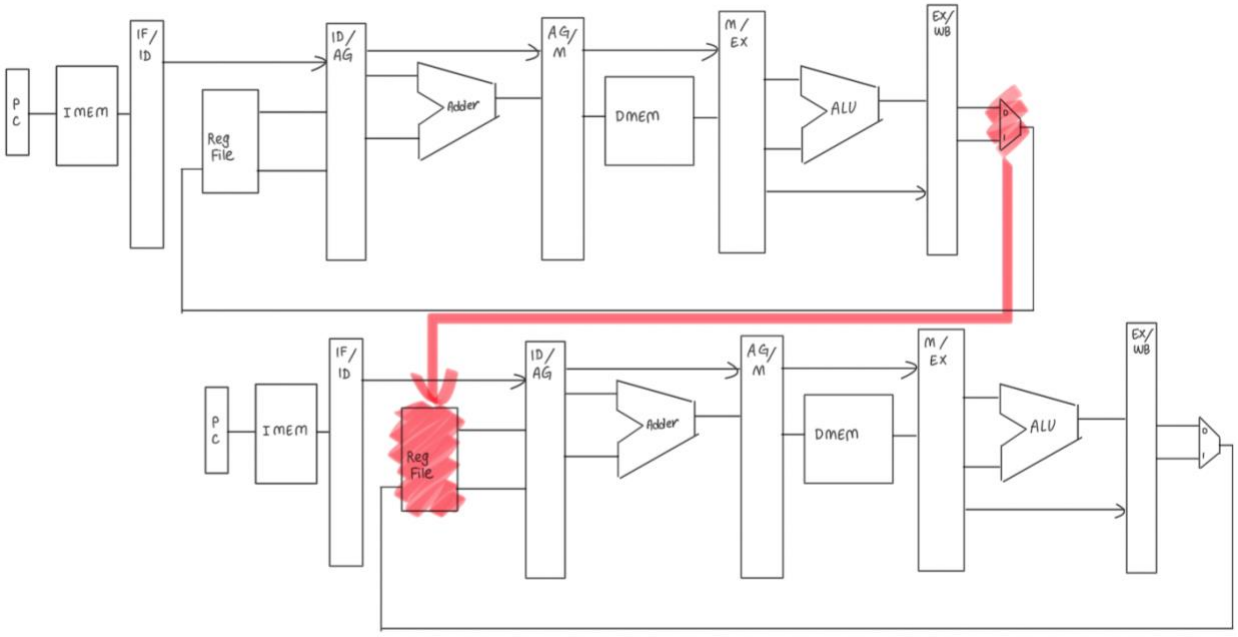

```
2. lb     $t1, 0($t1)      # $t1 is a data hazard
   addu   $t1, $a3, $t1
```



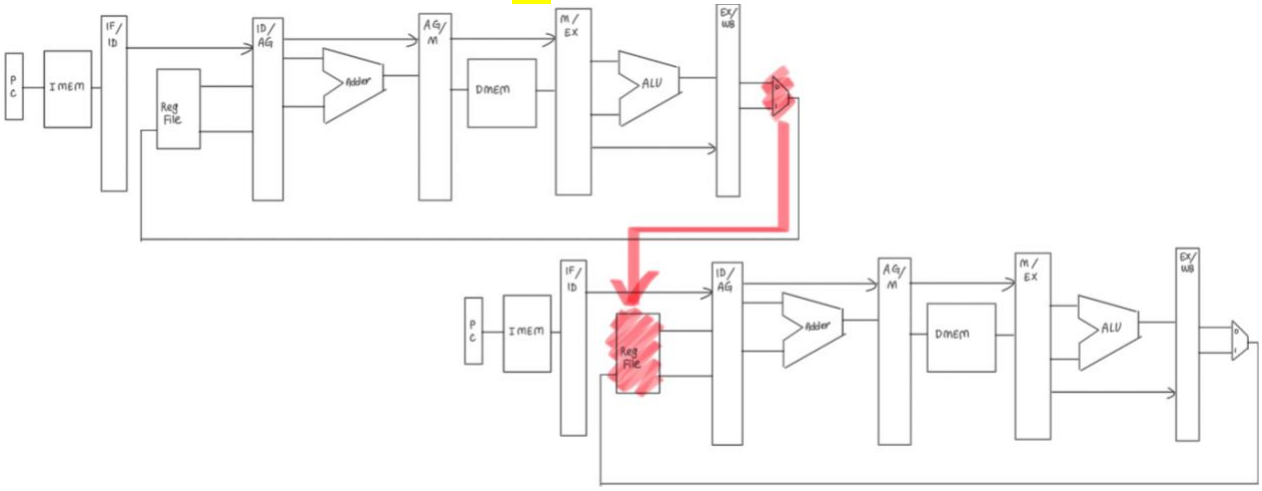
3. **addu** $t1, $a3, $t1 # $t1 is a data hazard
   lb $t2, 0($t1)



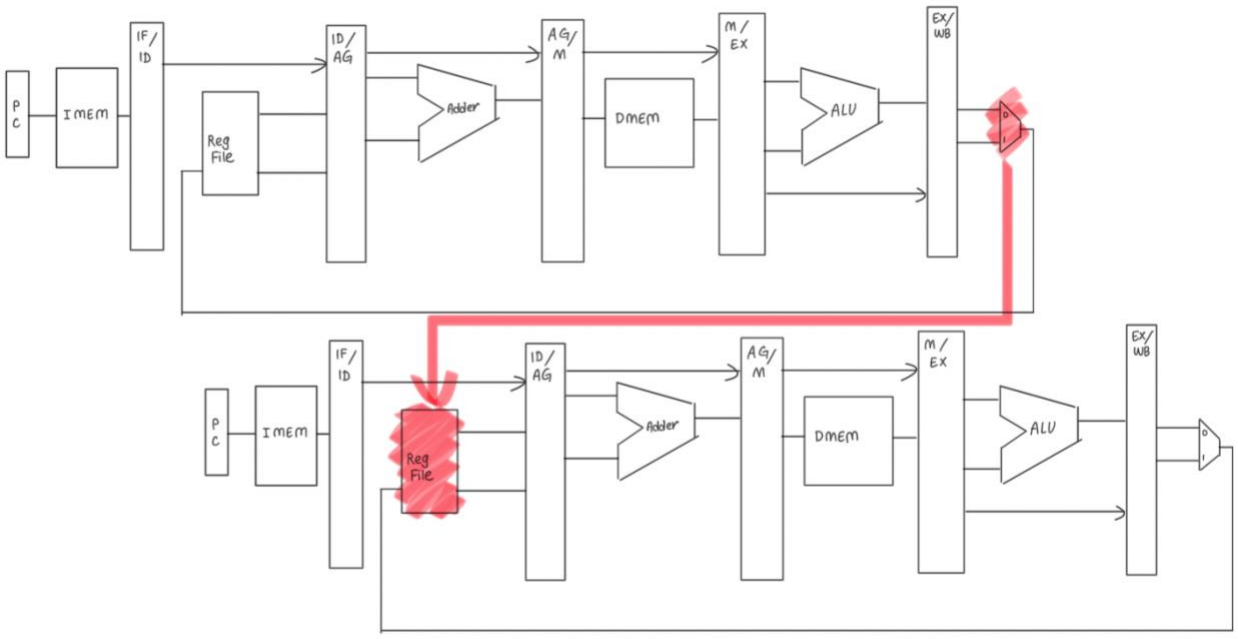
4. **lb** $t2, 0($t1) # $t2 is a data hazard
   addiu $t2, $t2, 1



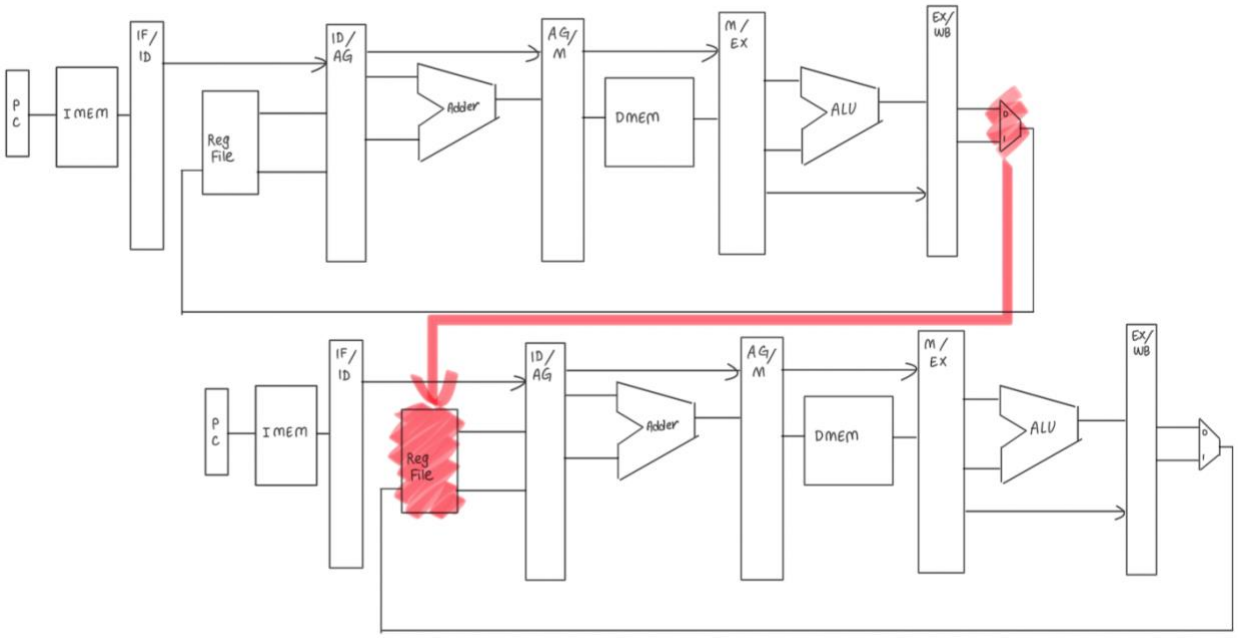

```
5. addu  $t1, $a3, $t1    # $t1 is a data hazard
   sb    $t2, 0($t1)
```




```
6. addiu $t0, $t0, 1      # $t0 is a data hazard
   sltu  $t1, $t0, $a1
```



7. **addiu** $t2, $t2, 1 # $t2 is a data hazard
   sb $t2, 0($t1)



c. Pipeline Hazard Avoidance (Software)
Insert the minimum number of NOP instructions in order to allow the instruction sequence used in 3b to run correctly on the 6-stage pipeline described above without any hardware hazard detection or forwarding logic. NOTE: the instruction sequence used does not have any control flow instructions!

```
addu  $t1, $a0, $t0
NOP
NOP
NOP
NOP
lb    $t1, 0($t1)
NOP
NOP
NOP
NOP
addu  $t1, $a3, $t1
NOP
NOP
NOP
NOP
lb    $t2, 0($t1)
NOP
NOP
NOP
NOP
addiu $t2, $t2, 1
NOP
NOP
NOP
NOP
sb    $t2, 0($t1)
addiu $t0, $t0, 1
NOP
NOP
NOP
NOP
cond:
sltu  $t1, $t0, $a1
```

d. Pipeline Hazard Avoidance (Forwarding Logic)
Now we want to implement forwarding logic in HW to improve the CPI and # of instructions executed in our implementation. Specify the forwarding condition for the address generation module's input A (assume this is the base register input for the base offset address mode). Use the same format as we did in lecture (see Lec10.1) where the ForwardA select signal is 00 for the ID/AG pipeline register, 01 for the M/EX pipeline register, and 10 for the EX/WB pipeline register. Demonstrate that your forwarding condition holds true by drawing and annotating a pipeline diagram for the execution of the first six instructions after the `loop` label (**`addu`**, **`lb`**, **`addu`**, **`lb`**, **`addiu`**, **`sb`**).

I don’t know how to do this. Sorry.

e. Why such a pipeline?
What would be a possible advantage of having the above six-stage pipeline (think about this pipeline implementing a more CISC-like ISA)? What would your average CPI be? (answer qualitatively rather than quantitatively)

A six-stage pipeline could have several advantages, especially for a more CISC-like ISA. One of these advantages is the ability to handle complex instructions more efficiently. By dividing execution into six stages, the pipeline allows for better workload distribution, reducing bottlenecks that shorter pipelines might experience. The extra stage can also enable higher clock speeds, as each stage performs less work (due to having fewer components) per cycle compared to a shorter pipeline (with more components in a single stage).

The average cycle per instruction (CPI) is 1. The CPI wouldn’t change because the instructions go through 6 stages, completed in 6 cycles. This is also true for the 5-stage pipeline that completes each instruction in 5 cycles. Therefore, this indicates that even though there are six stages in the above pipeline, the average CPI would not be impacted and would remain at 1.

3. Exam Question
Develop your own exam question (roughly 10-15 points) from MIPS arithmetic, single-cycle processor design, performance analysis, pipelining, or data hazards. Your question shouldn’t simply ask students to recall information, but should ask for an application of a concept or require understanding of a concept or need analysis of a processor/application. You must include a correct and complete solution to your question. This question should be your own work and not copied from a book or an old exam. ***Post your question AND solution to the Exam 2 channel for others to use.***

Posted on Teams channel.