ECE 369A – Fall 2023

HW3

20 Points

Group # ?????			
Name	**Last Name**	**% Effort**	**Lab Section** **A** - 2:00-3:15pm / **B**- 3:30:4:45pm / **C**- 5:00-6:15pm
?	**?**	**?**	**?**
?	**?**	**?**	**?**
?	**?**	**?**	**?**

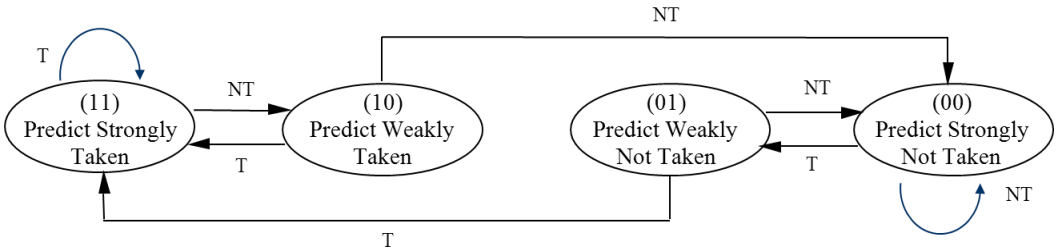
-2 pt per missing "Group #", "Name", "Last Name", "% Effort", "Lab Section"

Grading:

- Will randomly pick a subset of the questions and scale the overall score to 30 points.
- 5pts per day late penalty
- 4pts penalty per skipped question

Show your work in order to receive full credit. If you write the final result only (without showing how you derived it), then you will receive partial credit even if the answer is correct

Problem 1 Figure illustrates the Automaton-A3 branch predictor, which predicts taken when in states 11 and 10.



a) What will be the prediction ratio (accuracy) with the 2-bit branch prediction scheme for **only** the branch predictor of the "**if" statement** in the given code below? Assume that a branch is taken when its condition statement is satisfied, and the predictor is initially in "**Strongly Not Taken**" state. **No justification no credit!**

```
for (i=0;i<1000;i++){
  for (j=0;j<3;j++){
    if (((i*j)+1) % 2) == 0 ){
        c++;
    } } }
```

Fill in the table below for some number of iterations of the nested for loops that is sufficient to reach a conclusion and based on your analysis calculate the ratio of the number of correct predictions for the "if" statement with respect to the total number of times that "if" statement is executed in the nested for loop. A branch outcome is taken when the condition for that instruction is satisfied.

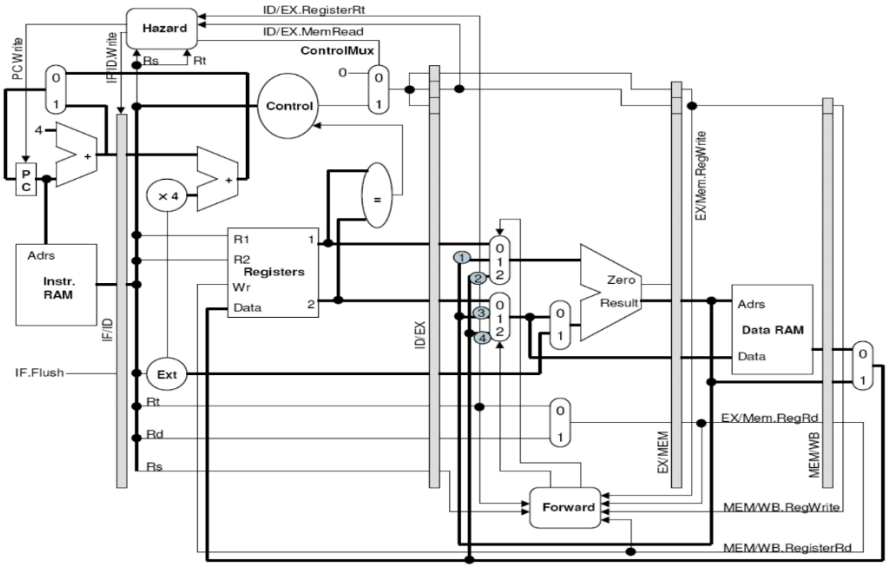
T: Taken, N: Not taken, C: Correct prediction, I: Incorrect prediction, Current state: 00, 01, 10, or 11

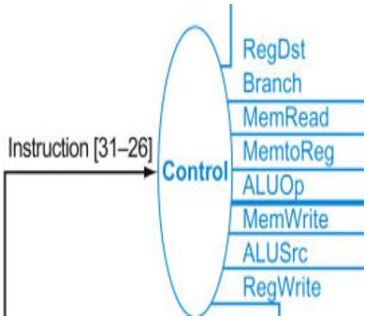
i	0	0	0	1	1	1	2	2	2	3	3	3	4	4	4	5
j	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0
Branch Outcome (T/N)	N	N	N	N	T	N	N	N	N	N	T	N	N	N	N	N
Current State (00/01/10/11)	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0
Prediction (T/N)	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
Next State (00/01/10/11)	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0
Correct/Incorrect (C/I)	C	C	C	C	I	**C**	**C**	**C**	**C**	**C**	**I**	C	C	C	C	C

Pattern is ICCCCC, 1 out of 6 is a misprediction ignoring the first 4 correct predictions that have negligible impact on overall accuracy.

Problem 2 Assume that the datapath is the standard 5-stage pipeline where conditional and unconditional branches are resolved in the decode stage and all register updates are completed during the write back stage. You should use the datapth given below as a starting point. This datapath supports forwarding to EX stage only and supports hazards for lw followed by a dependent instruction. In this exercise you will be given various dependency scenarios. You will answer the following questions:

1. Is the current forwarding logic sufficient to resolve the dependency? If so, write the forwarding logic.
2. If the answer to "1" is no, will the dependency be resolved with another forwarding unit in another stage of the pipeline? If so, then indicate the pipeline stage where forwarding is needed, show **the block diagram with all inputs and outputs just like the forwarding unit that is in the EX stage.** Give proper names to each input and output. **Finally write the forwarding logic (**the expressions) that will detect and resolve the dependency case.
3. If the dependency can't be resolved with forwarding only, then
    a. **design the hazard detection with conditions to capture the dependency and the outputs to control the datapath** and indicate the pipeline stage where hazard detection is needed. You need to cover hazard conditions for all pipeline stages where a hazard occurs.
    b. **After implementing the hazard detection unit and assuming that it is in place, will this dependency benefit from forwarding? If yes,** then indicate the pipeline stage where forwarding is needed, show **the block diagram with all inputs and outputs just like the forwarding unit that is in the EX stage.** Give proper names to each input and output. **Finally write the forwarding logic (**the expressions) that will detect and resolve the dependency case





Naming Convention: For each signal, labeling starts with the source stage IF/ID, ID/EX, EX/MEM, or MEM/WB indicating Decode, Execute, Memory Access and Write Back stages respectively. Controller module and signals shown above (right side) are in the decode stage. Controller generates the following signals to configure the datapath: RegDst, Branch, MemRead, MemtoReg, ALUOp, MEMWrite, ALUSrc, RegWrite.

Example1: The RegWrite signal should be named as IF/ID.RegWrite, ID/EX.RegWrite, EX/MEM.RegWrite, MEM/WB.RegWrite, if the source of signal is from the Decode, Execute, Memory Access and Write Back stages respectively.
Example2: if you need to use the "Opcode" field (Instruction[31:26]), signal should be named as IF/ID.Opcode, ID/EX.Opcode, EX/MEM.Opcode, MEM/WB.Opcode if the source of signal is from the Decode, Execute, Memory Access and Write Back stages respectively.

Case1:

```
sub $s3, $t1, $a0
xor $s1, $a1, $a2
beq $s3, $a3, loop

forwarding from Mem to Id stage (sub-beq dependence)
If (
       ( ExMem.RegWrite == 1) &&
       ( ExMem.Rd == IfId.Rs ) &&
       ( IfId.Opcode == beq ) \\ IFID.branch==1
   )
        C = 1
```

Forwarding can resolve the dependency without having to stall (hazard detection unit is not needed).

Case2:

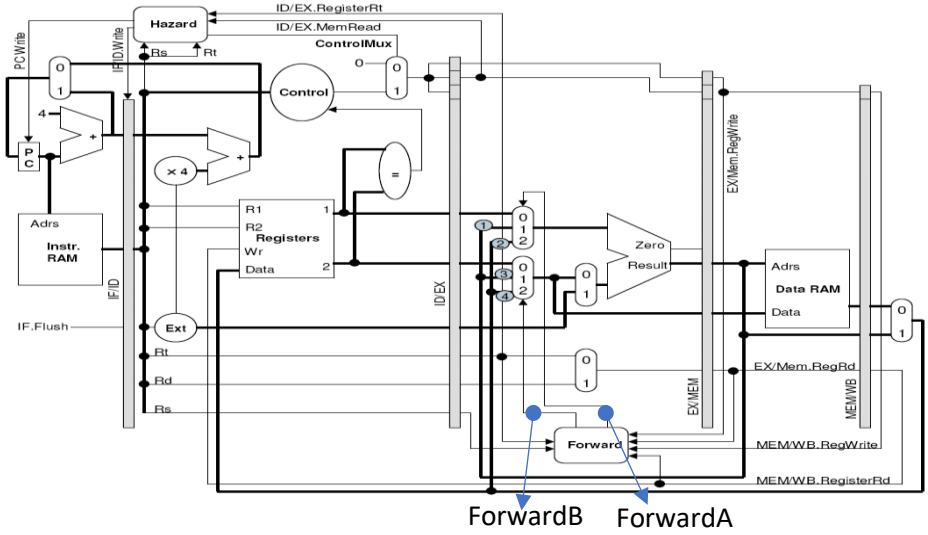
```
lw $s3, 4($a0)
xor $t0, $a1, $a0
beq $a3, $s3, loop

Stalls when lw in Mem and beq in ID stage

Hazard detection in ID stage

If (
     (ExMem.MemRead == 1) &&
     ( ExMem.Rt == IfId.Rt) &&
     ( IfId.Opcode == beq )   \\ or (IFID.branch == 1)
    )
        PCWrite = 0;
        IfIdWrite = 0;
        ControlMux=0;
```

Branch check is needed. Otherwise, all dependencies for instructions other than branch type in the decode stage will result with a stall.

Problem 3: Pipelining and forwarding

The pipelined datapath below has a number of wires/buses annotated.

- Branches are assumed to be not taken.
- A hazard unit inserts stalls for lw instructions




```
if ((EX/MEM.RegWrite == 1) and
    (EX/MEM.RegisterRd == ID/EX.RegisterRs))
        then ForwardA = 1
else if ((MEM/WB.RegWrite == 1) and
        (MEM/WB.RegisterRd == ID/EX.RegisterRs))
        then ForwardA = 2
```

This datapath is currently supporting hazard detection for "lw" followed by a dependent instruction. Datapath is not completely supporting hazards related to branch instruction (beq). Implement the **complete** hazard detection on the provided datapath to support the following cases:

(i) lw instruction followed by dependent beq instruction

```
//When lw is in EX stage and Branch in ID stage
If ID/EX.MemRead == 1 && (IF/ID.opcode == branch) &&
( (ID/EX.Rt == IF/ID.Rs) or (ID/Ex.Rt = IF/ID.Rt) ) {
  PCWrite = 0;
  IF/ID.Write = 0;
  ControlMux = 0;
}

//When lw is in MEM stage and Branch in ID stage
If EX/Mem.MemRead == 1 && (IF/ID.opcode == branch) &&
( (EX/Mem.Rt == IF/ID.Rs) or (EX/Mem.Rt = IF/ID.Rt) ) {
  PCWrite = 0;
  IF/ID.Write = 0;
  ControlMux = 0;
}
```

(ii) R-type of instruction followed by a dependent beq instruction

You need to consider all scenarios such as lw-beq, lw-indepdent instruction-beq flows. You need to introduce new inputs to the hazard detection unit. Show all necessary inputs and label each input wire clearly. Then you need to implement hazard detection logic for multiple scenarios across the pipeline stages to stall the pipeline. For all applicable scenarios you need to write the full expression covering dependencies on Rs and Rt registers and generate proper output signal values.

```
//When R-type is in EX stage and Branch in ID stage
If ID/EX.RegWrite == 1 && (IF/ID.opcode == branch) &&
( (ID/EX.Rd == IF/ID.Rs) or (ID/Ex.Rd = IF/ID.Rt) ) {
  PCWrite = 0;
  IF/ID.Write = 0;
  ControlMux = 0;
}

//When R-type is in MEM stage and Branch in ID stage
If EX/Mem.RegWrite == 1 && (IF/ID.opcode == branch) &&
( (EX/Mem.Rd == IF/ID.Rs) or (EX/Mem.Rd = IF/ID.Rt) ) {
  PCWrite = 0;
  IF/ID.Write = 0;
  ControlMux = 0;
}
```

Problem 4 (Datapath for this problem is given in the next page)

Part (a)

This pipelined datapath shown in the next page *has several differences* in terms of branch resolution stage and forwarding paths. In what pipeline stage are branches resolved? How many cycles would we have to stall after a branch, if branches are not predicted?

Branches are resolved in the Execute stage (cycle 3). As soon as the branch is decoded (cycle 2), datapath stalls till the resolution of the branch, therefore 1 stall cycle is observed.

Part (b) Based on the forwarding paths that are present, fill in the table for the given cases (stalls can be marked with an **S).** You are not allowed to change the datapath, modify the instructions, order of operands.

- Register file: writes during first half, reads during second half of the clock.
- Control logic stalls in decode stage if a forwarding path is not available.

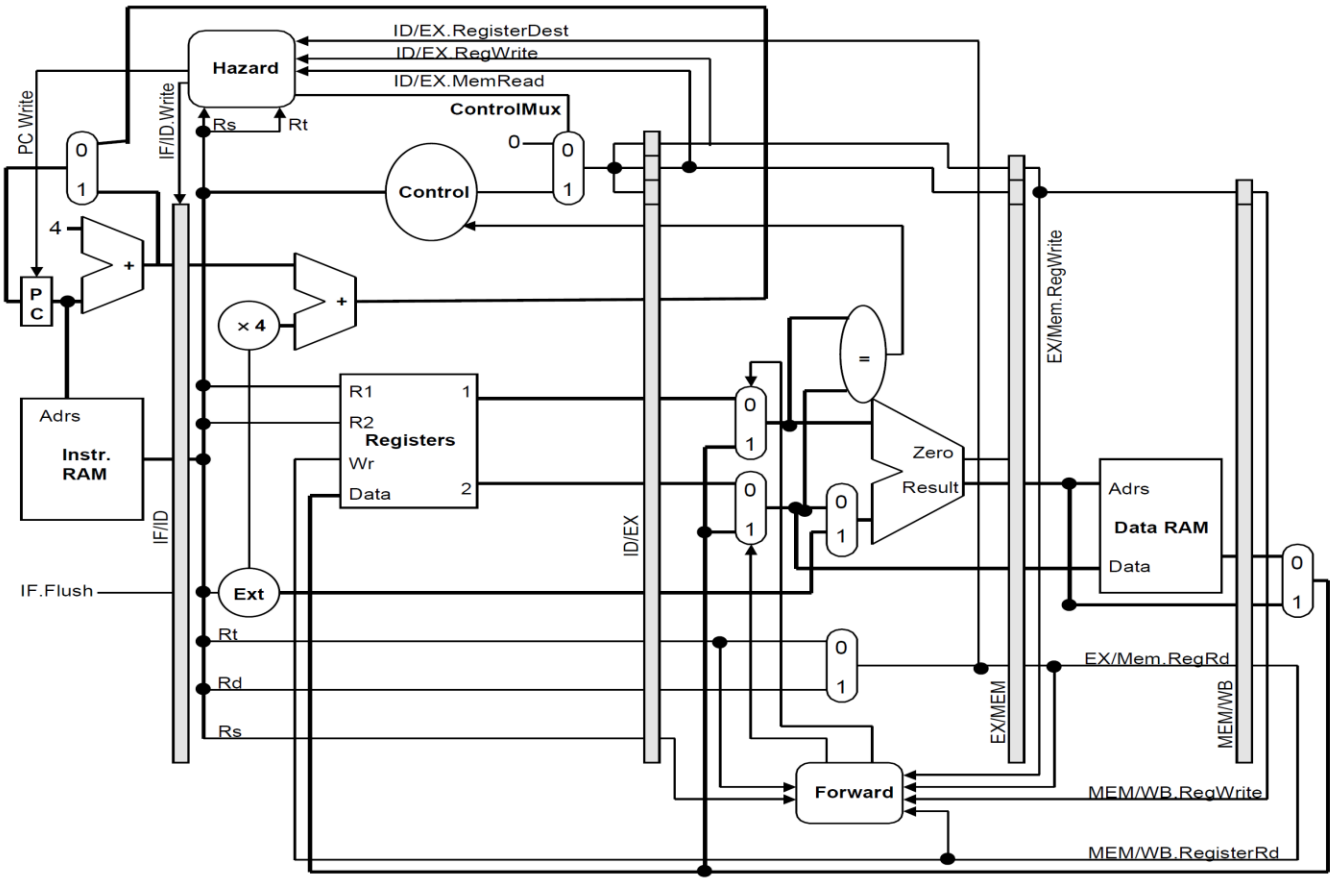
Case-1	1	2	3	4	5	6	7	8	9	10	11	12	13
lw $1, 8($3)	IF	ID	EX	ME	WB								
sub $4, $2, $1		IF	ID	S	E	M	W						

Case-2	1	2	3	4	5	6	7	8	9	10	11	12	13
add $1, $2, $3	IF	ID	EX	ME	WB								
sub $4, $2, $1		IF	ID	S	EX	ME	WB						

Case-3	1	2	3	4	5	6	7	8	9	10	11	12	13
add $1, $2, $3	IF	ID	EX	ME	WB								
sub $4, $2, $1		IF	ID	S	EX	ME	WB						
or $1, $4, $1			IF	S	ID	S	E	ME	WB				

Case-4	1	2	3	4	5	6	7	8	9	10	11	12	13
add $1, $3, $5	IF	ID	EX	ME	WB								
beq $1, $2, skip		IF	ID	S	EX	ME	WB						

Case-5	1	2	3	4	5	6	7	8	9	10	11	12	13
Loop: addi $a0, $a0, 4	IF	ID	EX	ME	WB								
lw $t0, -4($a0)		IF	ID	S	EX	ME	WB						
bne $a1, $t0, loop			IF	S	ID	S	EX	ME	WB				

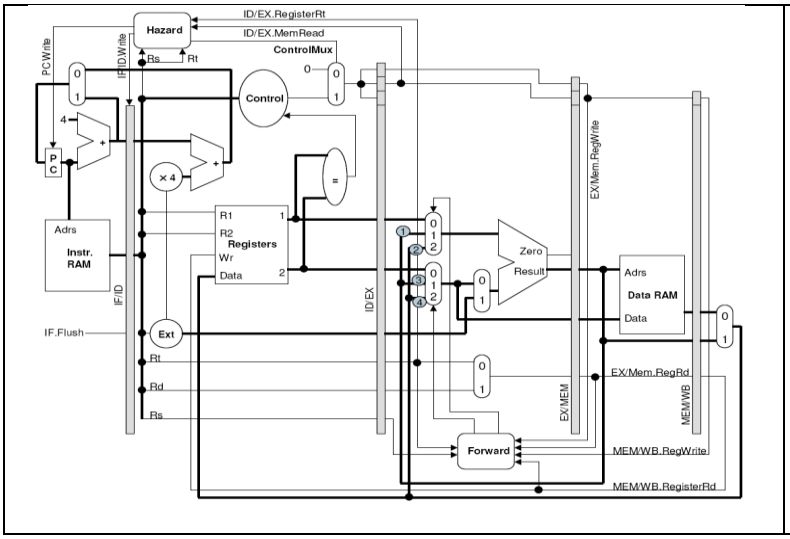


Part (c)

For the loop shown in case-5, in which cycle will the “addi” from the next iteration of the loop appear in the IF stage?

Since branches are resolved in the EX stage (cycle 7), instruction from next iteration of the loop will appear in the pipeline in cycle 8.

Problem 5.




```
apple:
      lw     $t0, 0($a1)
      mul    $t0, $t0, $a2
      lw     $t1, 0($a0)
      add    $t0, $t0, $t1
      sw     $t0, 0($a0)
      addi   $a3, $a3, -1
      addi   $a0, $a0, 4
      addi   $a1, $a1, 4
      bne    $a3, $zero, apple
```

Part (a) Assume that we run the "apple" code on the pipelined datapath shown above. Find the number of cycles needed to execute this code, accounting for all possible stalls and flushes. Assume that $a3 is initially set to 100. Show your work by filling the table below. Assume branches are executed as predicted not taken. That means, instruction(s) following the beq is/are flushed if the branch is taken (branch condition satisfied).

Inst	iter	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2
lw	N	F	D	E	M	W																	
mul	N		F	D	S	E	M	W															
lw	N			F	S	D	E	M	W														
add	N					F	D	S	E	M	W												
sw	N						F	S	D	E	M	W											
addi	N								F	D	E	M	W										
addi	N									F	D	E	M	W									
addi	N										F	D	E	M	W								
bne	N											F	D	E	M	W							
lw	N+1													F									

It takes 12 cycles before lw from N+1 cycle appears for loop iterations 1 through 99
Last iteration takes 15 cycles to complete
Therefore cycle count =99*12 + 15= 1203

First, you should see that the dependencies which may cause a problem are the ones between instructions 1 & 2, 3 & 4 and 4 & 5. Other dependencies, such as the ones between lines 2 & 4 or lines 6 & 9, are too far apart to be potential hazards. The two "lw" dependencies will force a stall, but the dependency in lines 4 & 5 will be resolved by forwarding from the EX/MEM register of the "add" to the EX stage of the "sw." Also, if we predict that branches are not taken, each execution of "bne" except for the last one will result in a misprediction and one cycle lost to a flush (branches are determined in the ID stage).

If you think about stalls and flushes in terms of "nop" instructions, then 12 instructions are executed for every loop iteration—there are nine instructions originally, plus two stalls and one flush. So running the loop 100 times involves executing 1199 instructions. (We subtract one since the last "bne"will be predicted correctly and does not result in a flush.) With a five-stage pipeline, 1199 instructions would execute in 1199 + 4 = 1203 clock cycles.

Part (b) Show how you can re-arrange the instructions in the code above to eliminate as many stall cycles as possible. You do not need to reduce the number of flushes.

Again, there are only two stalls in each loop iteration due to the "lw" instructions. The simplest solution is to just swap instructions 2 and 3 of the loop. This will separate the load and use of registers $t0 and $t1 by one extra cycle, which is enough to prevent the stalls.