CS 152
Computer Systems Architecture

Summer Session II 2018

Homework 3

Due August 30th, 2018, EEE 11:59PM in PDF form

Name	Student ID
Cristian Gonzales	**45349991**

IMPORTANT NOTES:

1. **No late submissions.**
2. **Please show your work. Remember that bottom line answers without proper explanations are worth ZERO points.**
3. **Even if you consult with your peers, do make sure the answers herein are yours and that you do not copy or cut-and-paste from other people's work.**

For Grading Purposes Only:

Q1	Q2	Q3	Q4	Q5	Q6
10	10	10	10	10	10

Q7	Q8	Q9	Q10
10	10	10	10

Total Score
100

Problem 1 (10 points)

True and False Questions. Give reasons for your answers.

(a) Allowing jumps, branches, and ALU instructions to take fewer clock cycles than the five required by the load instruction will increase pipeline performance under all circumstances.

This is false. Improving pipeline performance implies that data will be written faster, so just because the clock cycles of these instruction types are less than the five required by the load, this doesn't necessarily mean that the execution will become faster. For instance, just because jumps and branches take 3 cycles, at the third stage, we still have to flush the pipe if the branch is taken. For flushing in the ALU instruction, we have to wait for the ALU to finish, put the result in the third pipeline register, and from there, straight to the fourth pipeline register, and finally take it back to the register file. If a load takes 5 clock cycles to execute, the length of each clock cycle will still remain the same.

(b) Trying to allow some instructions to take fewer cycles does not help, since the throughput is determined by the clock cycle. (It is true that the number of pipe stages per instruction affects latency, not throughput.)

This is true. The throughput, informally, can be characterized as one instruction finishing every single clock cycle. Once the pipeline is full, we get 1 instruction per clock cycle as throughput. The number of pipe stages are affects only the latency, or the time it takes to fill the pipe. The pipes are designed such that after you fill the pipe, there's one instruction finishing every clock cycle.

(c) Allowing jumps, branches, and ALU operations to take fewer cycles only helps when no loads or stores are in the pipeline, so the benefits are small.

This is false. Even if loads and stores are not in the pipeline, each instruction would still have to go through each of the five cycles even though the amount of cycles was reduced for specific instructions. For instance, for an ALU operation, even though nothing is being done in the MEM stage of the pipeline, it still has to go through that stage and at the rising edge of the clock, write to the MEM/WB pipeline register and perform the write back to the register file. Also, for jumps and branches, we have branch predictions, so this entails more cycles in any case.

(d) Since branches and jumps can take fewer cycles, there is some opportunity for improvement in pipelining performance. (You cannot make ALU instructions take fewer cycles because of the write-back of the result.)

This is false. Throughput is defined by one instruction finishing every single clock cycle and the number of pipe stages per instruction affects latency. Thus, just because branches and jumps can take fewer cycles, we still have to account for branch predictions, so this only affects the latency. This doesn't define the execution time of one instruction finishing every clock cycle, and because performance is directly correlated with throughput, this statement is false.

(e) We could improve performance if instead of trying to make instructions take fewer cycles, we explore making the pipeline longer, so that instructions take more cycles, but the cycles are shorter.

This is true. In principle, you can increase the latency if you make the pipe longer. If the stages are of finite length, that means they are faster, so you can reduce the clock frequency.

Problem 2 (10 points)

Consider the pipelined implementation (without forwarding and/or stalling) of the MIPS microprocessor.

(a) Explain how this pipelined implementation deals with I-type conditional branch instructions in the case the branch is not taken and in the case the branch is taken.
(b) Explain why data dependency hazards may occur in this implementation.
(c) List all possible instruction sequences that may exhibit data dependency hazards.
(d) For each sequence in (c), give a software equivalent that does not suffer from the data dependency limitation.

(a) If the branch is not taken, the instruction that came after the branch are fed into the pipe and everything continues as if nothing happened. If the branch is taken, there is a flush (clearing the whole pipe and start from scratch—the hiccup would cost 4 clock cycles). What this entails is that there are three instructions that come after the branch instruction, and one instruction preceding the branch instruction. Thus, when we flush, we must allow the preceding instruction to terminate and take the destination address, and flush the rest. At the third rising edge of the clock, when we are setting the program counter to the destination address of the branch, we may clean the control lines for the previous registers. At the following edge of the clock, stalls will ensue. The destination will be at the program counter, and during the clock cycle, the instruction at the destination address will be fetched, and so on. Pipelines are designed to flush any time the branch is taken.

(b) By definition, a data dependency hazard is one that occurs when the value read by one instruction is not the correct one because the previous instruction has not finished updating. In the most general sense, a data hazard may be defined as a dependency, of an instruction, on data that is being accessed by a prior instruction. A data dependency may happen in this implementation, because, for example, given the following instructions

```
add $3, $1, $2
add $5, $3, $4
```

it can be seen in the pipelined datapath that, without pipeline forwarding, the ALU will be performing the addition operation between registers $1 and $2 for the first instruction at the same time that the operand fetch is being performed for the second instruction (thus, the operand that will be fetched for the second instruction will be a stale value because the first instruction has not written back the result of the ALU operation back to the register file). Thus, it cannot be the case that this implementation does not have data hazards (to conclude an informal proof by contradiction).

(c) For this implementation, the exact number of instructions in a sequence that should be considered is four instructions ("a window of four") for register operations because at the rising edge of the clock where the first instruction writes back, the fourth/last instruction in the "window of four instructions" will be reading at the same time that the first instruction is writing back. Thus, the instruction following the fourth instruction will be "safe" from any data hazards. We have to look on the left hand side of the assignment, and see if that register appears on the right hand side of the next three instructions (thus, the "window" of four instructions). For memory operations, we should look at "windows" of two instructions where a load uses a base register (rs) that is being updated in the instruction behind it; otherwise, loads before stores are not a problem because a store writes at the rising edge of the clock that reads the load behind it.

(d) Nops for memory operations and out-of-order executions for register operations are both viable software solutions to instructions that are vulnerable to data dependencies.

Problem 3 (10 points)

Identify all of the data dependencies in the following code. Which dependencies are data hazards that will be resolved via forwarding?

```
add   $2, $5, $4
add   $5, $2, $4
sw    $5, 100($2)
lw    $4, 0($5)
add   $3, $2, $4
```

Evaluating in sequences of 4 (since there are 4 instructions that can affect the preceding instructions in a waterfall fashion), the data dependencies include registers $2, $5, and $4. They are circled below:

All the dependencies except for the immediate instruction that follows the load instruction (more specifically, register $4) may be resolved via forwarding; that is to say, dependencies from registers $2 and $5 may be resolved via forwarding.

Problem 4 (10 points)

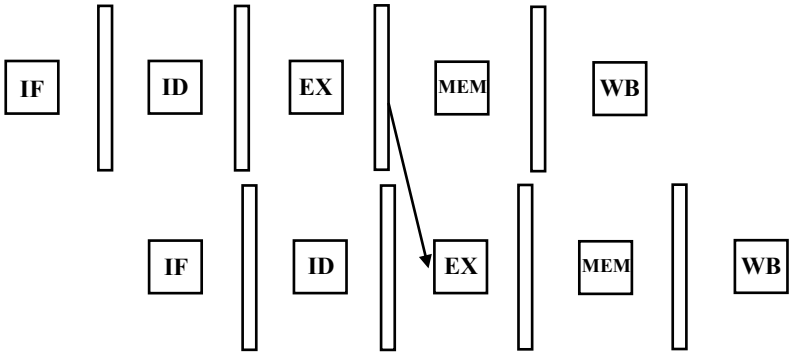
Assume a 5 stage pipelined MIPS processor with stages IF, ID, EX, MEM and WB. LOAD and STORE are the only instructions accessing memory. Branches are resolved at ID stage.

(a) Give a code sequence that has data hazard which can be solved by forwarding.
(b) Give a code sequence that has data hazard that cannot be solved by forwarding. Indicate stall cycles required.
(c) Explain branch hazards. Why do branch hazards degrade the performance?

(a) The following code sequence has a data hazard:

```
add $3, $1, $2
add $5, $3, $4
```

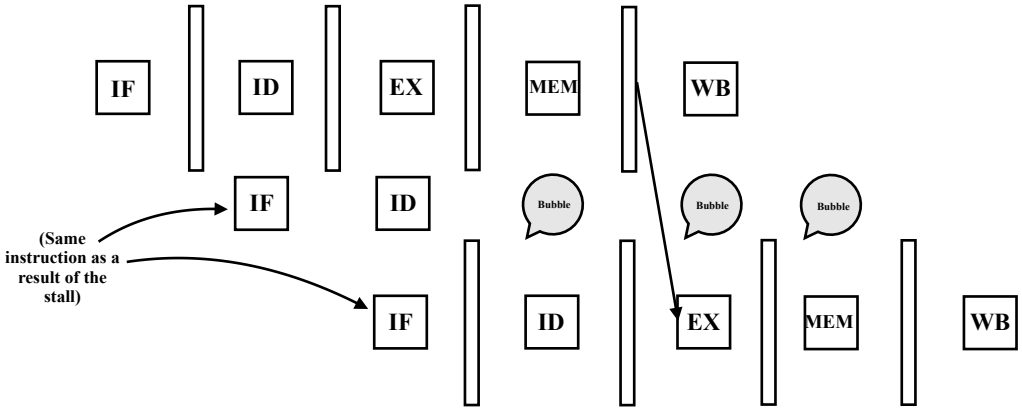
This can be solved via forwarding because the pipeline register EX/IM, upon calculation of the value of register $3 in the first instruction, may feed the appropriate operand/value for $3 into the ALU for the EX stage of the second instruction. This can be illustrated, roughly, as follows:



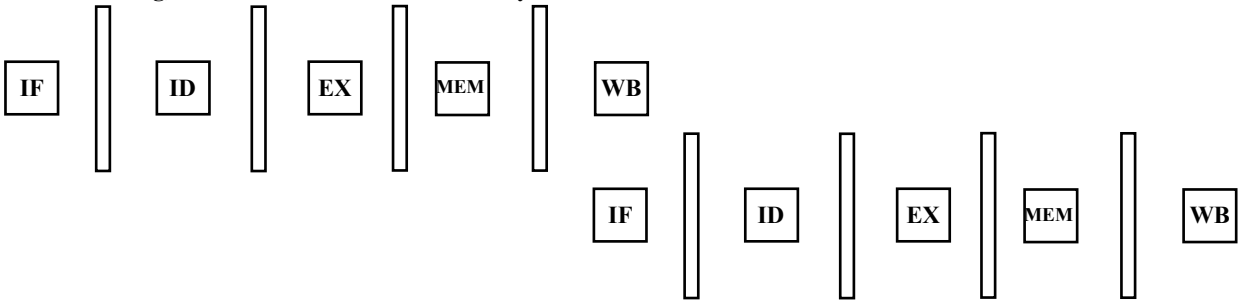
(b) This code sequence has a data hazard:

```
lw   $2, 0($1)
add $4, $3, $2
```

This cannot be solved via forwarding because if an instruction followed a load instruction immediately after in the pipeline and had a data dependency in said load instruction (as shown above), the load instruction would have written the value of register $2 into the MEM/WB register at the rising edge of the clock, at which point the subsequent instruction would have written the value of the ALU operation (in this case, $3 and $2) into the EX/MEM register—it would be too late to provide the register $2 data dependency, as we cannot "go back in time" to do pipeline forwarding. If we use forwarding, we would only need to stall for one cycle, as follows:



Without forwarding, we would need to stall for three cycles:



(c) Informally, branch hazards are control hazards where instructions that should not be executed “sneak in” after the branch. Branch hazards degrade the performance because we have to pay the latency every time a branch is taken, so we have a few clock cycles where we are doing nothing useful (no instructions are being executed).

Problem 5 (10 Points)

Consider the following MIPS code sequence.

```
        add $2, $2, $2
        add $5, $5, $5
L:      lw $8, 1000 ($5)
        sub $5, $2, $8
        addi $2, $2, -4
        beq $2, $0, L
        sw $5, 500 ($2)
```

Assume that there is **no forwarding unit** (including register file forwarding) but instead there is a data hazard detection unit that introduces the stalls needed to avoid data hazards. Suppose the processor uses **Assume Branch Not Taken** strategy and branches are resolved in the ID stage. Illustrate the execution of the given code.

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Add	F	D	E	M	W															
Add		F	D	E	M	W														
Lw			**F**	—	—	—	**D**	**E**	**M**	**W**										
Sub							**F**	—	—	—	**D**	**E**	**M**	**W**						
Addi											**F**	**D**	**E**	**M**	**W**					
Beq												**F**	—	—	—	**D**	**E**	**M**	**W**	
Sw																**F**	**D**	**E**	**M**	**W**

[Note] F: Fetch, D: Decode, E: Execution, M: Memory Access, W: Register Write Back, —: **Stall**

Problem 6 (10 points)

Using the same sequence of instructions as in Problem 5, now suppose the situation **beq** will be taken and branches are resolved in the ID stage (Processor uses **Assume Branch Not Taken** strategy). Show the execution of the given code around the loop, starting with the execution of a **beq** instruction and ending with the next execution of a **beq**. **(10 points)**

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Beq	F	D	E	M	W															
Sw		F	D	**F**	**F**	**F**														
Lw			**F**	**D**	**E**	**M**	**W**													
Sub				**F**	—	—	—	**D**	**E**	**M**	**W**									
Addi								**F**	**D**	**E**	**M**	**W**								
Beq									**F**	—	—	—	**D**	**E**	**M**	**W**				

[Note] F: Fetch, D: Decode, E: Execution, M: Memory Access, W: Register Write Back, **F**: **Flush,** —: **Stall**

Problem 7 (10 points)

Using the same instruction sequence as in Problem 5, redraw a diagram that describes the execution of the given code assuming that the processor has a **forwarding unit**.

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Add	F	D	E	M	W															
Add		F	D	E	M	W														
Lw			**F**	**D**	**E**	**M**	**W**													
Sub				**F**	—	—	**D**	**E**	**M**	**W**										
Addi							**F**	**D**	**E**	**M**	**W**									
Beq								**F**	**D**	**E**	**M**	**W**								
Sw									**F**	**D**	**E**	**M**	**W**							

[Note] F: Fetch, D: Decode, E: Execution, M: Memory Access, W: Register Write Back, —: **Stall,** →: **Data forwarding**

Problem 8 (10 points)

An unpipelined processor has a cycle time of 30ns. What is the cycle time of a pipelined version of the processor with 5 evenly divided pipeline stages if each pipeline latch has a latency of 1ns? What if the processor is divided into 50 pipeline stages? What is the minimum cycle time we can hope to get if we could add as many pipeline stages as desired?

For a function F(x) representing the unpipelined processor with a cycle time of 30ns, if we evenly divide the processor into 5 pipeline stages the cycle time will be 7ns (a pipeline stage of 6ns plus a pipeline latch latency of 1ns). If we evenly divide the processor into 50 pipeline stages, the cycle time will be 1.6ns (a pipeline stage of 0.6ns plus a pipeline latch latency of 1ns). The minimum cycle time we can hope to get if we could add as many pipeline stages as desired is 1ns, because, in theory, if you can add an infinite amount of pipeline stages to the processor, the function F(x) = (30 / x)ns + 1ns for x pipeline stages (where the unpipelined processor has a cycle time of 30ns and each pipeline latch has a fixed latency of 1ns), the limit of F(x) as x approaches infinity is simply 1ns.

Problem 9 (10 points)

Consider the following code segment within a loop body:

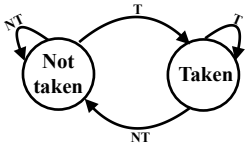
```
if ( n % 2 == 0 )  // branch 1
        a++;
if ( n % 10 == 0 )// branch 2
        b++;
```

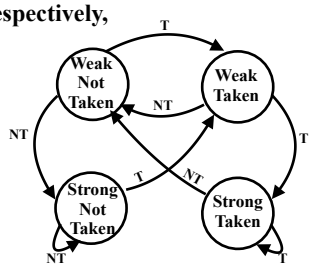
Assume that the following list of 10 values of n is to be processed by 10 iterations of this loop:
18, 29, 30, 41, 52, 60, 79, 80, 91, 100

List the predictions for the following branch prediction schemes and calculate the prediction accuracies for each scheme.

(a) Always taken.
(b) Always not taken.
(a) 1-bit predictor, initialized to predict taken.
(b) 2-bit predictor, initialized to weakly predict taken.

Following the FSM diagrams for the one and two bit branch predictors, respectively,





(a) The predictions for the "Always taken" branch prediction scheme is as follows:

18, branch 1: Always taken
18, branch 2: Always taken
29, branch 1: Always taken
29, branch 2: Always taken
30, branch 1: Always taken
30, branch 2: Always taken
41, branch 1: Always taken
41, branch 2: Always taken
52, branch 1: Always taken
52, branch 2: Always taken
60, branch 1: Always taken
60, branch 2: Always taken
79, branch 1: Always taken
79, branch 2: Always taken
80, branch 1: Always taken
80, branch 2: Always taken
91, branch 1: Always taken
91, branch 2: Always taken
100, branch 1: Always taken
100, branch 2: Always taken

Prediction accuracy = 50% (10 out of 20)

(b) The predictions for the "Always not taken" branch prediction scheme is as follows:

18, branch 1: Always not taken
18, branch 2: Always not taken
29, branch 1: Always not taken
29, branch 2: Always not taken
30, branch 1: Always not taken
30, branch 2: Always not taken
41, branch 1: Always not taken
41, branch 2: Always not taken
52, branch 1: Always not taken
52, branch 2: Always not taken
60, branch 1: Always not taken
60, branch 2: Always not taken
79, branch 1: Always not taken
79, branch 2: Always not taken
80, branch 1: Always not taken
80, branch 2: Always not taken
91, branch 1: Always not taken
91, branch 2: Always not taken
100, branch 1: Always not taken
100, branch 2: Always not taken

Prediction accuracy = 50% (10 out of 20)

(c) The predictions for the 1-bit branch prediction scheme, initialized to "predict taken", is as follows:

18, branch 1: Predict taken
18, branch 2: Predict taken
29, branch 1: Predict not taken
29, branch 2: Predict not taken
30, branch 1: Predict not taken
30, branch 2: Predict taken
41, branch 1: Predict taken
41, branch 2: Predict not taken
52, branch 1: Predict not taken
52, branch 2: Predict taken
60, branch 1: Predict not taken
60, branch 2: Predict taken
79, branch 1: Predict taken
79, branch 2: Predict not taken
80, branch 1: Predict not taken
80, branch 2: Predict taken
91, branch 1: Predict taken
91, branch 2: Predict not taken
100, branch 1: Predict not taken
100, branch 2: Predict taken

Prediction accuracy = 50% (10 out of 20)

(d) The predictions for the 2-bit branch prediction scheme, initialized to "weakly predict taken", is as follows:

18, branch 1: Weakly taken
18, branch 2: Strongly taken
29, branch 1: Weakly not taken
29, branch 2: Strongly not taken
30, branch 1: Strongly not taken
30, branch 2: Weakly taken
41, branch 1: Strongly taken
41, branch 2: Weakly not taken
52, branch 1: Strongly not taken
52, branch 2: Weakly taken
60, branch 1: Weakly not taken
60, branch 2: Weakly taken
79, branch 1: Strongly taken
79, branch 2: Weakly not taken
80, branch 1: Strongly not taken
80, branch 2: Weakly taken
91, branch 1: Strongly taken
91, branch 2: Weakly not taken
100, branch 1: Strongly not taken
100, branch 2: Weakly taken

Prediction accuracy = 50% (10 out of 20)

Problem 10 (10 points)

Consider a processor with a delayed branch that has three delay slots. Three compilers compiler A, compiler B and compiler C, could run on this processor. Compiler A can fill the first delay slot 60% of the time and the second delay slot 40% of the time and the third delay slot 20% of the time (filling delay slot is independent). Compiler B can fully fill all the three delay slots. Compiler C leaves all the slots empty. Assuming that branches account for 20% of all instructions and arithmetic/logic operations for the remaining 80% of the instructions for any program, what is the improvement of CPI with compiler B compared to CPI with compiler A and compared to CPI with compiler C? Assume that CPI of arithmetic/logic operations is 1.

Assuming there are x instructions in the instruction mix, for compiler B, there are 0.2x branches, and no delays are suffered (because it fills all three delay slots), so it takes x units of time.

Compiler C leaves all slots empty so in addition to the x units of time, we have three times the number of branches the number of slots we're not filling and not executing any instructions, or 3(0.2x) = 0.6x, so this compiler takes 0.6x + x = 1.6x units of time.

The ratio of of CPI of compiler B compared to compiler C, is x/1.6x or 1/1.6, or a 60% improvement.

Comparing B to A, A can fill the first delay slot 60% of the time, which equates to 0.6(0.2x) = 0.12x units of time, the second delay slot 40% of the time or 0.4(0.2x) = 0.08x units of time, and the third delay slot 20% of the time or 0.2(0.2x) = 0.04x units of time. Thus, this compiler takes 0.12x + 0.08x + 0.04x + x = 1.24x units of time.

The ratio of of CPI of compiler B compared to compiler A, is x/1.24x or 1/1.24, or a 24% improvement.