Jordan Wu

U0900517

Homework Assignment 8

1. **Find the data dependencies in the following sequence of instructions:**

```
and  $t1, $t2, $t1
lw   $t2, 0($t1)
lw   $t1, 4($t1)
or   $t3, $t1, $t2
```

Draw a diagram similar to the diagrams from class and from figure 4.52 (on page 305), and show each data dependency with a line segment from the WB stage to the REG stage.

Single-cycle design will work correctly but the performance is inefficient to be used in modern designs. The reason why this design is not desirable is that the clock cycle is set by the highest latency instruction. Implementing any complex instruction will cause the clock cycle to be longer and will decrease performance of the processor. Pipelining is a technique that is used to increase the throughput of the processor by executing multiple instructions simultaneously. Using this technique involves dividing an instruction into five stages which will each execute during a single clock cycle. Here are the pipeline stages:

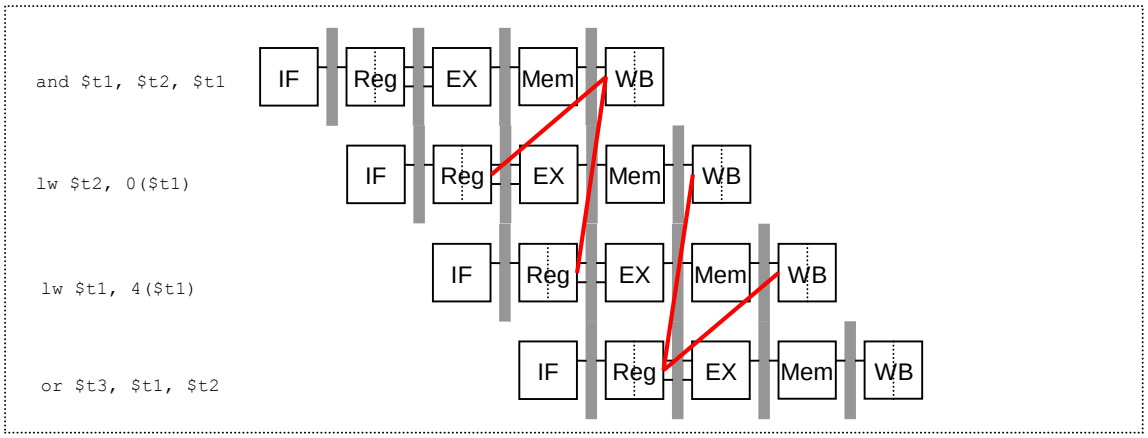
1. IF: Instruction fetch
2. ID: Instruction decode and register file read
3. EX: Execution or address calculation
4. MEM: Data memory access
5. WB: Write back

Note: Pipelining will be five times faster than the single-cycle design (N times faster where N is the number of stages).

Implementing Pipelining will cause some events called hazards, situations when the next instruction cannot execute in the following clock cycle. One hazard is called Data Hazards, this occurs when the later instruction needs the results of the earlier instruction to execute. This means that the pipeline must stall before executing the instruction because of data dependences. The primary solution to Data Hazards is by forwarding the data to the dependent instruction. The following sequence of instruction has a few dependences, shown in color:

```
and  $t1, $t2, $t1
lw   $t2, 0($t1)
lw   $t1, 4($t1)
or   $t3, $t1, $t2
```

Pipelined Dependences Diagram:



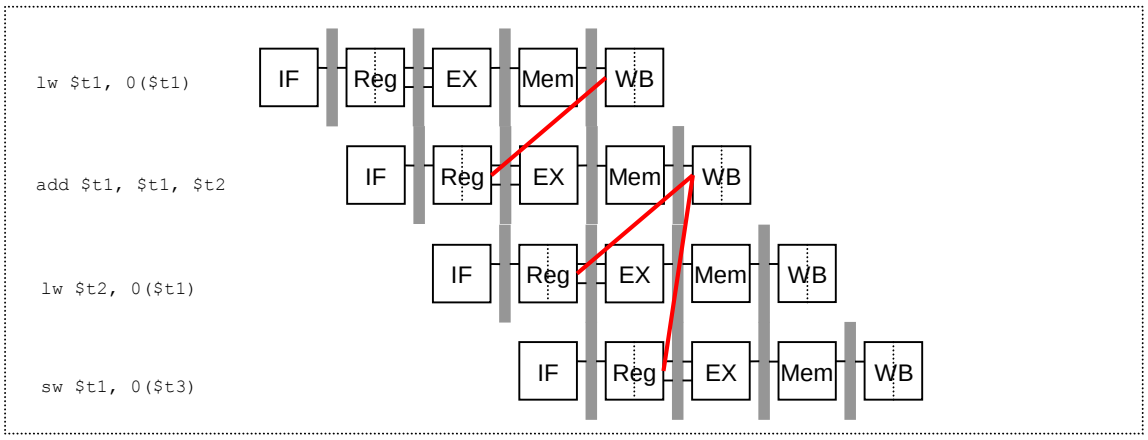
2. Repeat problem #1 on this sequence of instructions:

```
lw   $t1, 0($t1)
add  $t1, $t1,  $t2
lw   $t2, 0($t1)
sw   $t1, 0($t3)
```

The following sequence of instruction has the following dependences, shown in color:

```
lw   $t1, 0($t1)
add  $t1, $t1,  $t2
lw   $t2, 0($t1)
sw   $t1, 0($t3)
```

Pipelined Dependences Diagram:

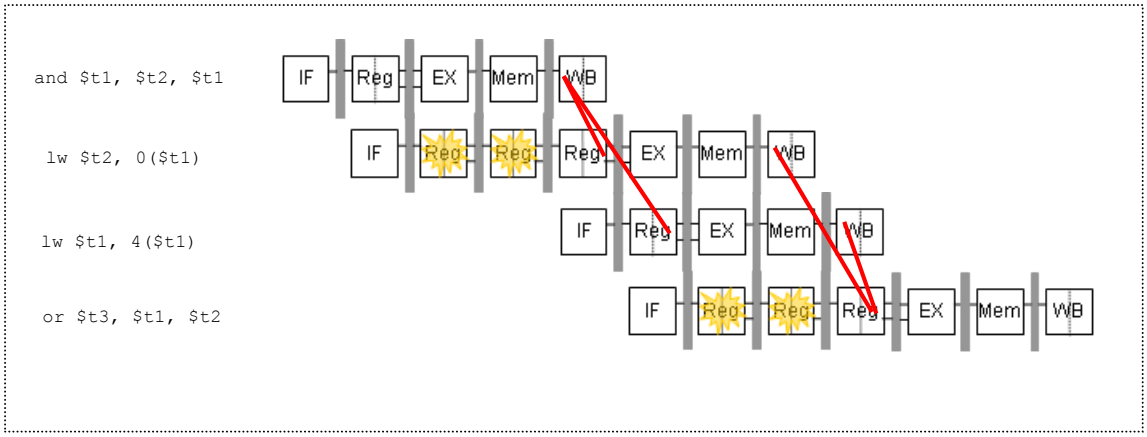


Problem 1 and Problem 2 show that there are data hazards if there are data dependences between the instructions in the sequence. Data hazards happens when the sequence instructions need data from the WB stages of an earlier instruction.

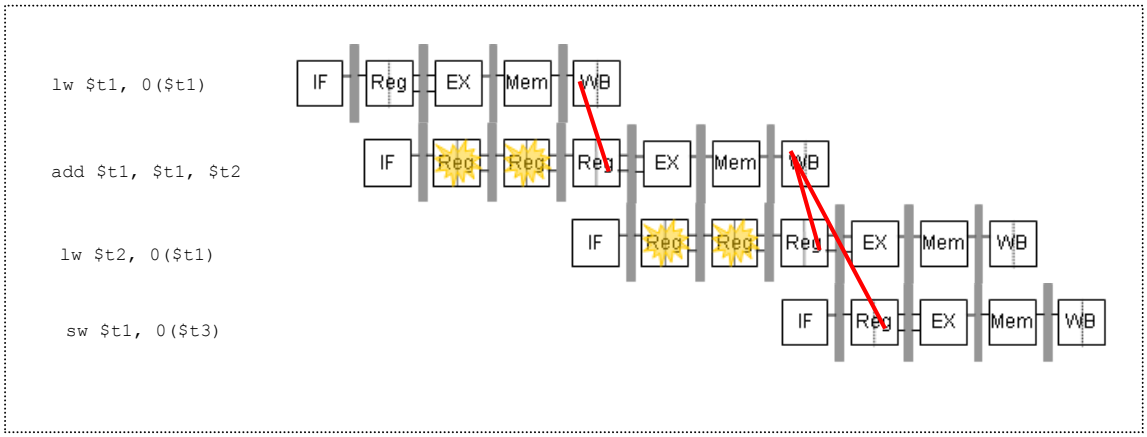
3. **Your solutions from questions #1 and #2 above demonstrate that there are data hazards in the instruction sequences. Redraw the diagrams with both data dependencies and with stalls (bubbles) added to eliminate data hazards. For each stall, just draw either a repeated stage, or a 'bubble' to indicate that a stage did not complete.**

   You should still show each data dependency with a line segment from the WB stage to the REG stage. You must assume that a register may be written and then read within the same clock cycle.

Pipelined Dependences with Stalls Diagram (Problem 1):



Pipelined Dependences with Stalls Diagram (Problem 2):



Observe that stalling is not a very desirable strategy when dealing with Data hazards.

4. Repeat problem #3 again, but this time assume that you can use forwarding to eliminate some stalls. Instead of drawing data dependencies from the WB stage to the REG stage, draw the forwarding of data values. Draw lines from some logic block at the start of a clock cycle to the logic block where the data is needed. (See page 308.)

If there is a stall you cannot eliminate, leave it in.

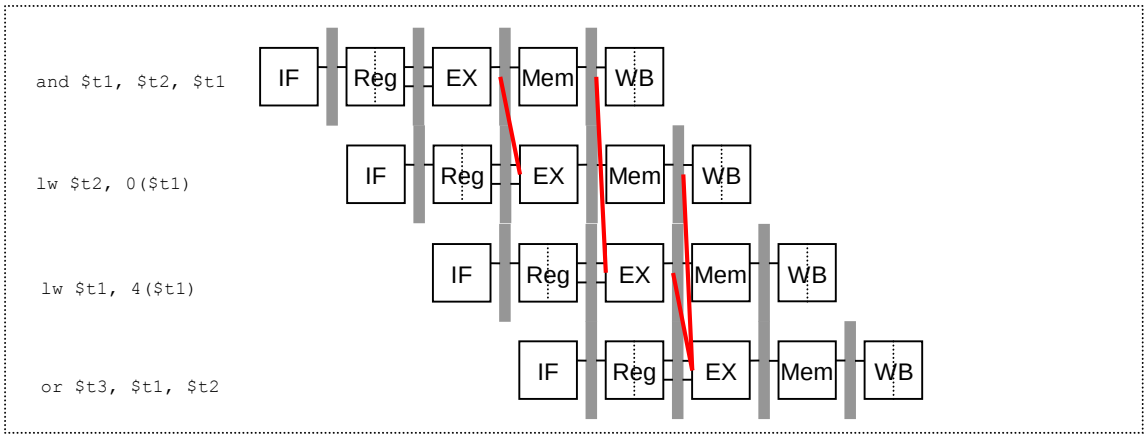
Two pairs of hazard conditions:

1a. EX/MEM.RegisterRd = REG/EX.RegisterRs
1b. EX/MEM.RegisterRd = REG/EX.RegisterRt
2a. MEM/WB.RegisterRd = REG/EX.RegisterRs
2b. MEM/WB.RegisterRd = REG/EX.RegisterRt

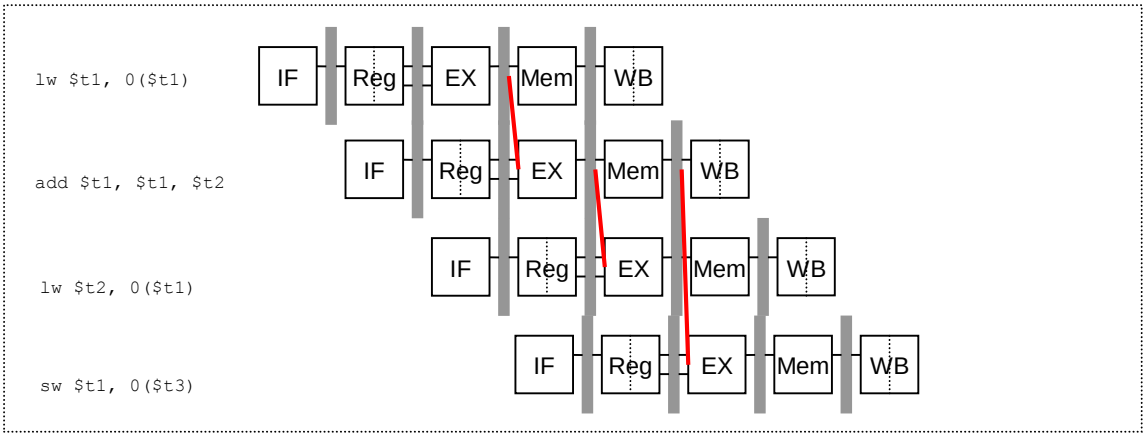
Note: EX/MEM is the latch between EX and MEM stage, MEM/WE is the latch between MEM and WB stage, and last REG/EX is the latch between the REG and EX stage. On the left hand side will be the destination register and on the right hand side will be either the Rs or Rt register depending on the specific instruction that is executing.

FORWARDING CAN ONLY BE USED IN THE SAME CLOCK CYCLE!!!

Pipelined Dependences Diagram (Problem 1):



Pipelined Dependences Diagram (Problem 2):



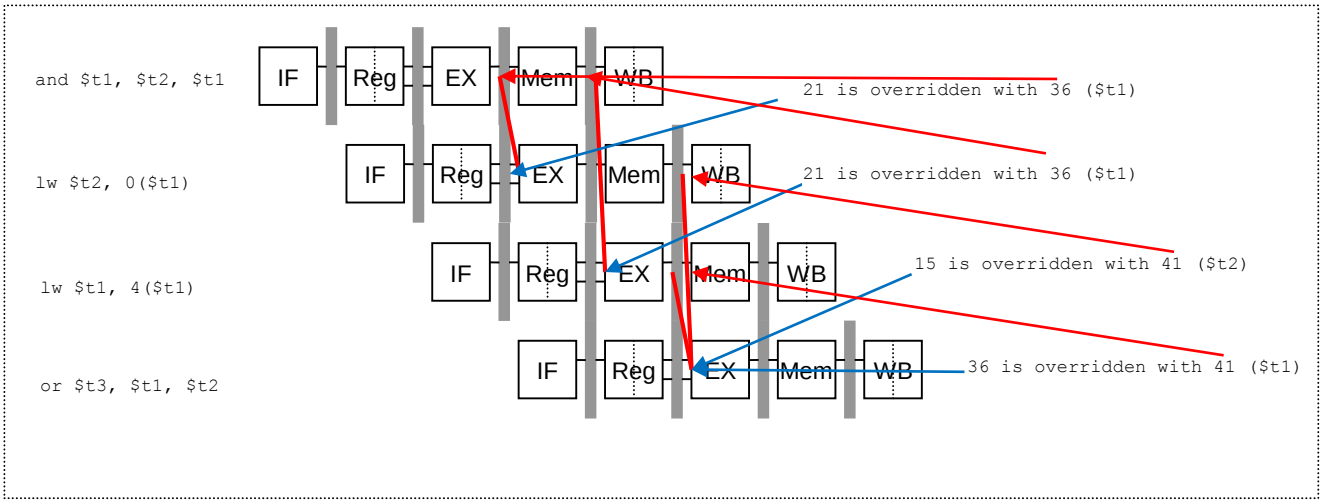
5. Consider the results of problem #4. Forwarding prevents incorrect values from being used in a computation. For this problem, I want you to show *actual* values that are forwarded. Assume registers contained these values (decimal numbers):

$t1	$t2	$t3
21	15	12

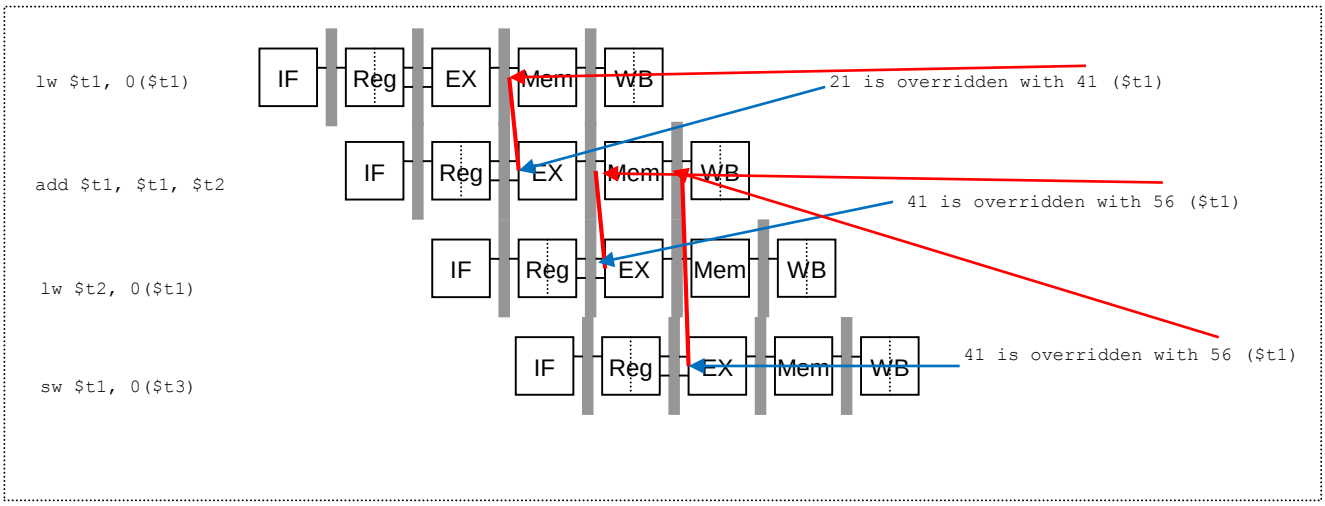
Also, assume that any memory reads will result in the value 41 (or 0x00000029 in hex) being read from memory.

Copy your diagrams from problem #4, locate the first forwarded value in each diagram, and then indicate on the diagram the *source* of the forwarded data values and the overridden data values. Indicate both exact values, and indicate which register is being forwarded.

Pipelined Dependences Diagram (Problem 1):



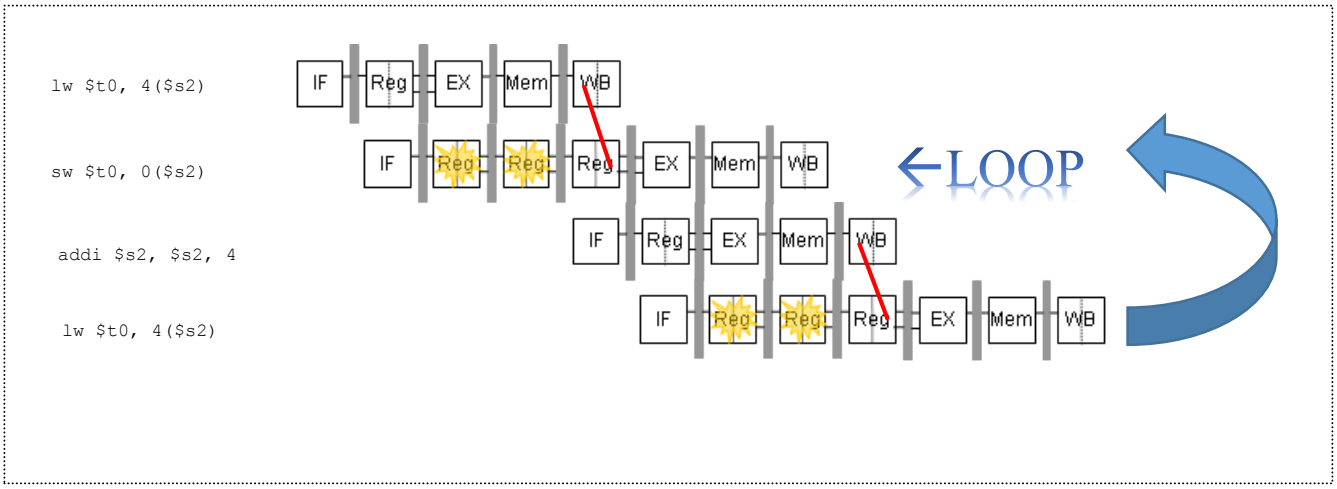
Pipelined Dependences Diagram (Problem 2):



6. Consider a small fragment of code that when repeated, moves elements in an array back one position:

```
lw   $t0, 4($s2)
sw   $t0, 0($s2)
addi $s2, $s2, 4
... keep repeating these statements over and over ....
```

If stalls are used to resolve data hazards, what is the average CPI of a very long sequence that repeats these three instructions?

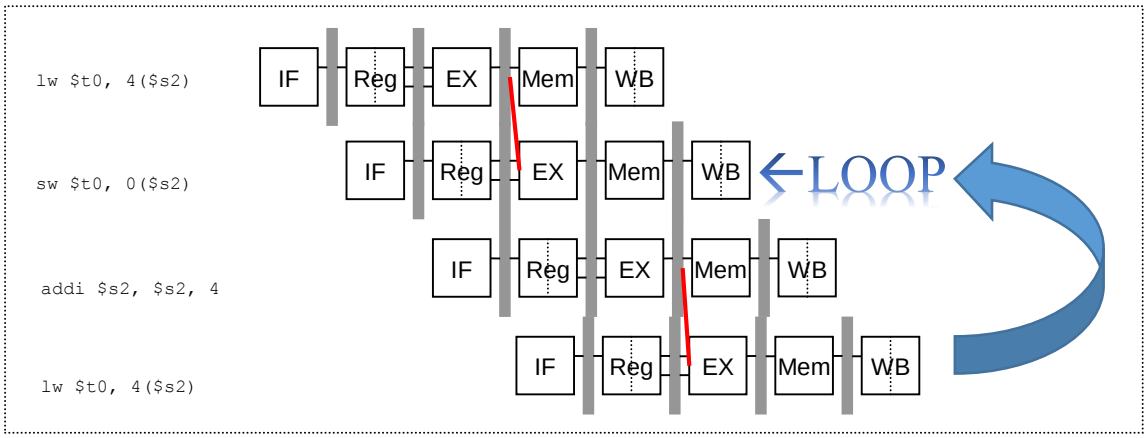


If stalls were used for this sequence of instruction that is repeated over and over again. The second instruction store word will need to stall two clock cycle for the load word to write back into the destination register ($t0). The second dependence is when load word is executing again, this time we need to stall two clock cycle for the addi instruction to write back into the destination register ($s2). Lastly this will be repeated (infinite loop). The average CPI will be the sum of the latency of each instruction inside the loop to execute divided by the number of instruction.

$$average\ CPI = \frac{5(clock\ cycles)}{3\ instruction} = 1.667(\frac{cycles}{instruction})$$

Seem right should be between the highest and lowest latency instruction (between 7 and 5).

If forwarding is used to resolve data hazards, what is the average CPI of a very long sequence that repeats these three instructions?



If forwarding is used the data can be retrieved from internal buffers and will not require any bubble/delays in the clock cycle. The data can be retrieved from EX/MEM or MEM/WB (determine by which stage is in the same clock cycle as the EX stage of the instruction that needs the data) and will be used in the EX stage of the dependent instruction. The average CPI will be the sum of the latency of each instruction inside the loop to execute divided by the number of instruction.

$$average\ CPI = \frac{3(clock\ cycles)}{3\ instruction} = 1(\frac{cycles}{instruction})$$

Comparing the average CPI of stalling to forwarding, forwarding is the better technique when dealing with Data hazards.

7. **Do problem 4.15, parts 1, 2, 3, 4, and 5 from the end of Chapter 4. To clarify, assume our pipelined design, assume no data hazards, and assume branch outcomes are determined at the end of the EX stage. (Count the stalls appropriately.) To solve the problem, you will need to do the following:**
    - **Determine the relevant information.**
    - **Determine the original overall CPI without any hazards (the problem assumptions make this easy).**
    - **Determine how to compute how the overall average CPI will change when branch stalls are added. Coming up with these formulas is most of the work for this problem.**
    - **Show how you solve each part.**

4.15

The importance of having a good branch predictor depends on how often conditional branches are executed. Together with branch predictor accuracy, this will determine how much time is spent stalling due to mispredicted branches. In this exercise, assume that the breakdown of dynamic instructions into various instruction categories is as follows:

R-Type	BEQ	JMP	LW	SW
40%	25%	5%	25%	5%

Also, assume the following branch predictor accuracies.

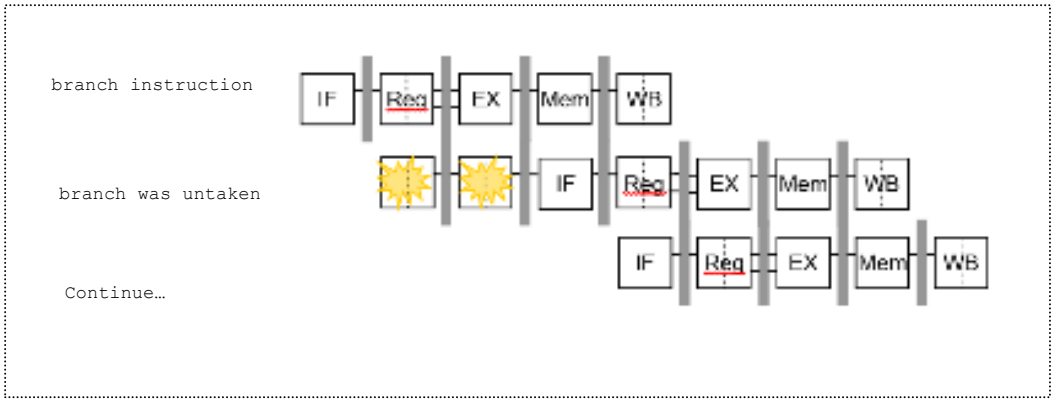
Always-Taken	Always-Not-Taken	2-bit
45%	55%	85%

4.15.1

Stall cycles due to mispredicted branches increase the CPI. What is the extra CPI due to mispredicted branches with the always-taken predictor? Assume that branch outcomes are determined in the EX stage, that there are no data hazards, and that no delay slots are used.

Control hazard happens when there is a need to make a decision based on the results of one instruction while others are executing (branch instructions). Computers uses prediction to handle branches, always-taken predictor means that we will always assume that a branch has been taken. Only when branches are untaken will the pipeline stall.

The figure below will show what happens when the branch was untaken for an always-taken predictor.



The extra CPI is due to the stall required when the branch is untaken and that our predicting was wrong will cause three stall cycles.

$$Extra\ CPI = P[not\ BEQ] \cdot 1\ cycle + P[BEQ] \cdot (P[Alway - Taken](1\ cycle) + P[Always - not - Taken] \cdot (3\ cycle))$$

$$= 0.75 + 0.25 \cdot (0.45 \cdot 1 + 0.55 \cdot 3) = 1.275 = 0.275$$

4.15.2

Repeat 4.15.1 for the "always-not-taken" predictor

This time our predictor is always-not-taken which is the vice versa of the always-taken predictor. This will have the same figure as the previous problem except this time, the stall will occur when the branch is taken and causes three stall cycles.

$$Extra\ CPI = P[not\ BEQ] \cdot 1\ cycle + P[BEQ] \cdot (P[Alway - not - Taken](1\ cycle) + P[Always - Taken] \cdot (3\ cycle))$$

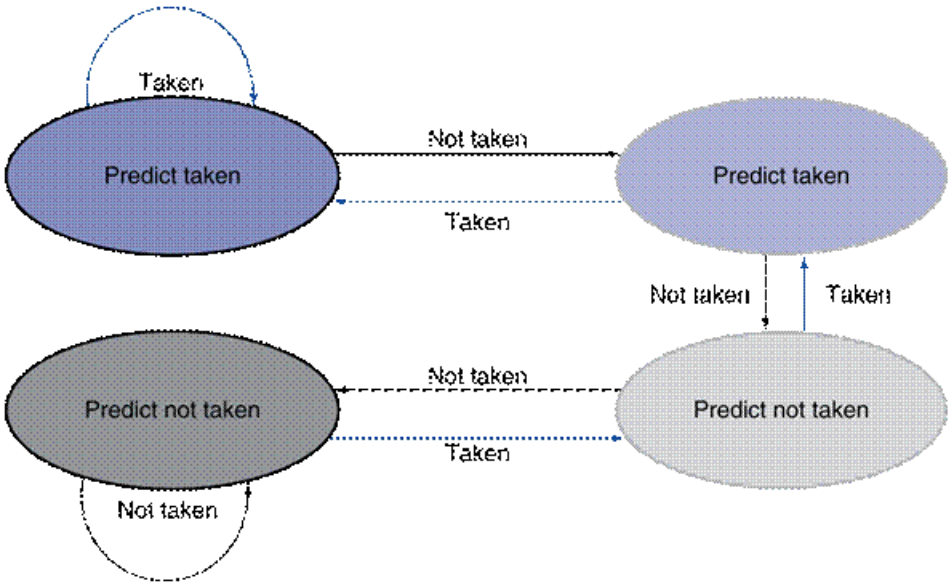
$$= 0.75 + 0.25 \cdot (0.55 + 0.45 \cdot 3) = 1.225 = 0.225$$

Since the P[error] for always-not-taken is lower than that of the always-taken. The extra CPI should be lower than the previous problem.

4.15.3

Repeat 4.15.1 for the 2 bit predictor.

In a 2-bit predictor, the prediction must be wrong twice before it is changed. The figure below will show the finite-state machine for a 2-bit predictor.



Since both the always-taken and always-not-taken have the same stall cycles when misprediction happens, the stall cycles will be three.

P[error] = 1 – P[2-bit] = 15%

$$Extra\ CPI = P[not\ BEQ] \cdot 1\ cycle + P[BEQ] \cdot (P[2-bit](1\ cycle) + P[error] \cdot (3\ cycle))$$

$$= 0.75 + 0.25 \cdot (0.85 + 0.15 \cdot 3) = 1.075 = 0.075$$

MISPREDICTION WILL INCREASE OUR CPI WHICH WILL THEN INCREASE OUR EXECUTION TIME AND LOWER OUR PERFORMANCE. HAVING AN ACCURTE PREDICTOR WILL OPTIMIZE OUR PERFORMANCE BY USING THE 2-BIT PREDICTOR.

4.15.4

With the 2-bit predictor, what speedup would be achieved if we could convert half of the branch instructions in a way that replaces a branch instruction with an ALU instruction? Assume that correctly and incorrectly predicted instructions have the same chance of being replaced.

Let's first look at the assumption that correctly and incorrectly predicted instruction have the same chance of being replaced. So we can assume 50%. If we didn't not convert the branch instruction we will have P[BEQ]=25% since the probability of BEQ instruction has not changed. P[error] = 1 – P[2-bit] = 15%. ALU instructions can be executed in one cycle, CPI for ALU will be 1. Let compute the CPI without replacing the branch instruction.

$$CPI = P[not\ BEQ] \cdot 1\ cycle + P[BEQ] \cdot (P[2-bit](1\ cycle) + P[error] \cdot (3\ cycle))$$

$$= 0.75 + 0.25 \cdot (0.85 + 0.15 \cdot 3) = 1.075$$

Now let's replace half our BEQ instruction, this time our P[error] is multiplied by P[replaced].

$$CPI = P[not\ BEQ] \cdot 1\ cycle + P[BEQ] \cdot (P[not\ replaced] \cdot \big(P[2-bit]P(1\ cycle) + P[error] \cdot (3\ cycle)\big) + P[replaced] \cdot (1\ ALU\ cycle))$$

$$= 0.85 + 0.25 \cdot (0.5 \cdot (0.85 + 0.15 \cdot 3) + 0.5) = 1.1375$$

Speedup is a metric for relative performance improvement when executing a task. The following equation is from Wikipedia for calculating speedup (using cycles per Instruction).

$$S = \frac{CPI\ for\ standard\ branch\ predictor}{CPI\ for\ modified\ branch\ predictor} = \frac{1.075}{1.1375} = 0.945\ speedup$$

This was a minor speedup.

4.15.4

With the 2-bit predictor, what speedup would be achieved if we could convert half of the branch instructions in a way that replaces a branch instruction with two ALU instruction? Assume that correctly and incorrectly predicted instructions have the same chance of being replaced.

The CPI for standard branch predictor is the same since we are not modifying anything to it. The only difference is there is two ALU instruction which will take 2 CPI to execute now. Now changing the modified branch predictor.

$$CPI = P[not\ BEQ] \cdot 1\ cycle + P[BEQ] \cdot (P[not\ replaced]$$

$$\cdot \big(P[2-bit]P(1\ cycle) + P[error] \cdot (3\ cycle)\big) + P[replaced] \cdot (2\ ALU\ cycle))$$

$$= 0.85 + 0.25 \cdot (0.5 \cdot (0.85 + 0.15 \cdot 3) + 0.5 \cdot (2)) = 1.2625$$

Now calculate the speedup.

$$S = \frac{CPI\ for\ standard\ branch\ predictor}{CPI\ for\ modified\ branch\ predictor} = \frac{1.075}{1.2625} = 0.8514\ speedup$$

Adding another ALU instruction will decrease our speedup and is not a desirable outcome.

8. **Below is a small loop (inside a function) written in MIPS. We don't care what it does, but we do care how long it takes to execute:**

```
Function:
        add  $t0, $zero, $a0    # Start here
Loop:
        lw   $t1, 0($a1)
        lw   $t2, 4($t1)
        add  $t3, $t0, $a2
        bne  $t2, $t3, Pass
        nop
        add  $t4, $t2, $t4
Pass:
        addi $t0, $t0, -1
        addi $a1, $a1, 4
```

```
bne  $zero, $t0, Loop
nop

and  $v0, $t4, $t4      # End here
jr   $ra
```

Assume this is executed on a five-stage pipelined processor that uses full data forwarding (and stalls when forwarding is impossible) to resolve data hazards. Assume the processor uses a single branch delay slot to resolve control hazards (no stalling or predictions - the branch delay slot is always executed even if the prior instruction branches).

Also assume that the first branch (that branches to 'Pass') branches the first time, and every other time (alternating) thereafter. (It will take the branch three times if the loop repeats five times.)

(a): If at the start $a0 is 5, how many cycles will this code take to complete? (Hint: Search for data hazards that require stall(s). Then, count the cycles from the start of the first instruction to the completion of the last add instruction.)

Must of the instruction's data dependencies can be fixed by forwarding, except for the load-use data hazard which occur in the instruction sequence after load word (lw). This require us to stall one before forwarding is possible. The value 5 in $a0 is stored into $t0 where it used to decrement the loop until it is 0. This mean the Loop is ran five times before it will stop looping. Using the assumption of **(that branches to 'Pass') the** 1st branches taken → 2nd branch not taken → 3rd branch taken → 4th branch not taken → 5th branch taken. This mean if it was taken then some instruction are skipped otherwise there will be a few extra instruction. It will take 47 instructions to complete this code. The loop takes 5 instruction if it branches to Pass, otherwise it takes 7 instruction. The Pass instruction takes 3 instruction and will branch back to Loop five times. The beginning of the code takes one instruction and to end the code it will take another 2 instructions.

Start of code: 1 instruction

Loop (taken to Pass): 5 instructions x 3 = 15 instructions

Loop (not taken to Pass): 7 instructions x 2 = 14 instructions

Pass: 3 instructions x 5 = 15 instructions

End of code: 2 instructions

SUM = 47 cycles

(b): Currently, the branch delay slots are filled with nop instructions, but this is inefficient. Rearrange the instructions so that the branch delay slots are filled with useful work and that data hazard stalls are eliminated (if possible). (You may duplicate and rearrange instructions, and remove the nop instructions, but do not unroll the loop or change the instructions.) The functionality of the code must be preserved. Explain in comments why your arrangement works.

Your finished code should eliminate stalls due to data hazards and fill branch delay slots. Your goal is to make your answer for part (c), below, as small as possible.

The first nop in the Loop: can be used for one of the load word instruction that is an independent instruction from before the branch. This will eliminate the load-use data hazard that requires one stall to allow forwarding. Doing this will allow useful work for this nop instruction without affecting the code. **REPLACE THE NOP INSTRUCTION IN THE LOOP: WITH THE LOAD WORD LW $t1, 0($a1) INSTRUCTION.** The last nop in the Pass: should be replace with the addi instruction that is an independent instruction from before the branch. **REPLACE THE NOP INSTRUCTION IN THE LOOP: WITH THE addi $a1, $a1, 4 INSTRUCTION.** Replacing the nop instruction with the following instruction will eliminate stall and will help decrease the number of cycles to complete this code. **BRANCH DELAY SLOTS ALLOWS EXECUTE AFTER THE BRANCH INSTRUCTION!!**

Following revised code with branch delay slots:

```
Function:
        add  $t0, $zero, $a0    # Start here
Loop:
        lw   $t2, 4($t1)
        add  $t3, $t0, $a2
        bne  $t2, $t3, Pass
        lw   $t1, 0($a1)
        add  $t4, $t2, $t4
Pass:
        addi $t0, $t0, -1
        bne  $zero, $t0, Loop
        addi $a1, $a1, 4

        and  $v0, $t4, $t4      # End here
        jr   $ra
```

(c): If at the start $a0 is 5, how many cycles will your revised code take to complete? Count the cycles from the fetching of the first instruction through the write back of the last instruction.

With the following changes, the load-use data hazard are eliminated and the nop instruction are doing useful work. This time the Loop: will have 4 instructions if it branches to Pass, otherwise 5 instructions. The Pass: will have 3 instructions and will run five times.

Start of code:	1 instruction
Loop (taken to Pass):	4 instructions x 3 = 12 instructions
Loop (not taken to Pass):	5 instructions x 2 = 10 instructions
Pass:	3 instructions x 5 = 15 instructions
End of code:	1 instructions
SUM =	39 cycles

(d): Now, change the assumption about branch delay slots. Assume the processor does not use branch delay slots, that control hazards are resolved by stalling one cycle when needed, and that the nop instructions are removed from the original code above.

Also assume a two-bit branch predictor is used to minimize stalls due to branch hazards, and that it's first prediction for each branch will be correct. Again revise the code to minimize data and control hazards. How many cycles will the code take to complete under these revised assumptions?

Revising the code to eliminate the load-use data hazard:

```
Function:
        add  $t0, $zero, $a0    # Start here
Loop:
        lw   $t2, 4($t1)
        add  $t3, $t0, $a2
        lw   $t1, 0($a1)
        bne  $t2, $t3, Pass
        add  $t4, $t2, $t4
Pass:
        addi $t0, $t0, -1
        addi $a1, $a1, 4
        bne  $zero, $t0, Loop

        and  $v0, $t4, $t4      # End here
        jr   $ra
```

The following code using two-bit branch predictor will minimize stalls.

This will stall whenever branch is not taken which in this case will be

two for the Loop: and one for the Pass: the total of 3 stalls.

Start of code:	1 instruction
Loop (taken to Pass):	4 instructions x 3 = 12 instructions
Loop (not taken to Pass):	5 instructions x 2 = 10 instructions
Pass:	3 instructions x 5 = 15 instructions
Stall:	3 instruction
End of code:	1 instructions
SUM =	42 cycles

(e): Which is better: Using branch delay slots and requiring the programmer to resolve control hazards, or using branch prediction and stalling to resolve control hazards? Define 'better' in this context, and make a logical argument in support of your position.

The best solution would be the branch delay slots which will optimize the performance of this code but requires software to rearrange the instructions. The problem with this is that the compiler must recompile every time a change is made. The better solution would be to have a global branch predictor that can guess whether or not to branch. This allow the predictor to be right or wrong, only when it is wrong a stall occurs. If this prediction is good (is right more than it is wrong) it will increase the performance of the code. Having a branch predictor is the better solution but the best solution will be to use branch delay slots.