Lincoln Sand

1. Consider an in-order 5-stage pipeline similar to the one discussed in class, e.g., see slides 3-9 of lecture 19. First assume that the pipeline does not support bypassing (forwarding). What are the stall cycles introduced between the following pairs of back-to-back instructions? Then, solve the same problem while assuming support for bypassing. Clearly show your work, i.e., show how each instruction goes through the 5 stages, indicate the point of production and point of consumption, show how the consuming instruction is held back in the D/R stage when there are stalls (similar to the example on slide 3 of lecture 19). Recall that a register read is performed in the second half of the D/R stage and a register write is performed in the first half of the RW stage. (60 points)

a.

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

Without Bypassing (Forwarding):

```
Cycle 1:  IF  -    -   -   -
Cycle 2:  DR  -    -   -   -
Cycle 3:  EX  -    -   -   -
Cycle 4: MEM  -    -   -   -
Cycle 5:  RW  -    -   -   -
Cycle 6:     IF  -   -   -   -
Cycle 7:     DR  DR  -   -   -  (Stall)
Cycle 8:         DR  EX  -   -
Cycle 9:             MEM RW  -
Cycle 10:               RW  -
```

The second add has to wait for the first add to complete the RW stage before it can read the updated value of $1 in the D/R stage. Thus, there are 2 stall cycles.

With Bypassing (Forwarding):

```
Cycle 1:  IF  -    -   -   -
Cycle 2:  DR  -    -   -   -
Cycle 3:  EX  -    -   -   -
Cycle 4: MEM  -    -   -   -
Cycle 5:  RW  -    -   -   -
Cycle 6:     IF  -   -   -   -
Cycle 7:     DR  EX  -   -   -
Cycle 8:         MEM RW  -
Cycle 9:             RW  -
```

With bypassing, the second add can use the result of the first add immediately after it is produced in the EX stage. Thus, there are no stall cycles.

b.

```
lw $1, 8($2)
add $4, $1, $3
```

Without Bypassing (Forwarding):

```
Cycle 1:  IF  -    -   -   -
Cycle 2:  DR  -    -   -   -
Cycle 3:  EX  -    -   -   -
Cycle 4: MEM  -    -   -   -
Cycle 5:  RW  -    -   -   -
Cycle 6:     IF  -   -   -   -
Cycle 7:     DR  DR  -   -   -  (Stall)
Cycle 8:         DR  EX  -   -
```

```
Cycle 9:              MEM RW  -
Cycle 10:                RW  -
```

The add instruction needs to wait for the lw instruction to complete the RW stage before it can read the updated value of $1. There are 2 stall cycles here.

With Bypassing (Forwarding):

```
Cycle 1:  IF  -    -   -   -
Cycle 2:  DR  -    -   -   -
Cycle 3:  EX  -    -   -   -
Cycle 4: MEM  -    -   -   -
Cycle 5:  RW  -    -   -   -
Cycle 6:     IF  -   -   -   -
Cycle 7:     DR  DR  -   -   -  (Stall)
Cycle 8:         EX  MEM -   -
Cycle 9:             RW  RW  -
```

The add instruction can get $1's value directly after the lw instruction's MEM stage. Therefore, only one stall cycle is needed.

c.

```
lw $1, 8($2)
sw $3, 8($1)
```

Without Bypassing (Forwarding):

```
Cycle 1:  IF  -    -   -   -
Cycle 2:  DR  -    -   -   -
Cycle 3:  EX  -    -   -   -
Cycle 4: MEM  -    -   -   -
Cycle 5:  RW  -    -   -   -
Cycle 6:     IF  -   -   -   -
```

```
Cycle 7:     DR  DR  -   -   -  (Stall)
Cycle 8:         DR  EX  -   -
Cycle 9:             MEM RW  -
Cycle 10:               RW  -
```

The sw instruction needs the address in $1 from the lw instruction. Hence, there are 2 stall cycles.

With Bypassing (Forwarding):

```
Cycle 1:  IF  -    -   -   -
Cycle 2:  DR  -    -   -   -
Cycle 3:  EX  -    -   -   -
Cycle 4: MEM  -    -   -   -
Cycle 5:  RW  -    -   -   -
Cycle 6:     IF  -   -   -   -
Cycle 7:     DR  DR  -   -   -  (Stall)
Cycle 8:         EX  MEM -   -
Cycle 9:             RW  RW  -
```

The sw instruction needs the address in $1, which is available after the lw's MEM stage. Only one stall cycle is required for the address to be available.

d.

```
lw $1, 8($2)
sw $1, 8($4)
```

Without Bypassing (Forwarding):

```
Cycle 1:  IF  -    -   -   -
Cycle 2:  DR  -    -   -   -
Cycle 3:  EX  -    -   -   -
Cycle 4: MEM  -    -   -   -
Cycle 5:  RW  -    -   -   -
```

```
Cycle 6:     IF  -   -   -   -
Cycle 7:     DR  DR  -   -   -  (Stall)
Cycle 8:         DR  EX  -   -
Cycle 9:             MEM RW  -
Cycle 10:               RW  -
```

In this case, sw needs the data from $1 to store, which is only available after the lw instruction completes its RW stage. Thus, there are 2 stall cycles.

With Bypassing (Forwarding):

```
Cycle 1:  IF  -    -   -   -
Cycle 2:  DR  -    -   -   -
Cycle 3:  EX  -    -   -   -
Cycle 4: MEM  -    -   -   -
Cycle 5:  RW  -    -   -   -
Cycle 6:     IF  -   -   -   -
Cycle 7:     DR  DR  -   -   -  (Stall)
Cycle 8:         EX  MEM -   -
Cycle 9:             RW  RW  -
```

Here, sw can use the data from $1 immediately after the lw instruction's MEM stage. There is one stall cycle for the data to be ready.

2. Consider a program that executes a large number of instructions. Assume that the program does not suffer from stalls from data hazards or structural hazards. Assume that 16% of all instructions are branch instructions, and 20% of these branch instructions are Taken. What is the average CPI for this program when it executes on each of the processors listed below? All of these processors implement a 5-stage in-order pipeline and resolve a branch outcome at the end of the 2nd stage (similar to the 5-stage pipeline discussed in class). If it helps, assume that the program has 100 total instructions and would finish in 100 cycles (CPI = 1.0) if it encountered zero stall cycles. Then, figure out the stall cycles for each of the cases below, so for example, 10 stall cycles would equate to an execution time of 110 cycles and a CPI of 1.1. (40 points)

a. The processor pauses instruction fetch as soon as it fetches a branch. Instruction fetch is resumed after the branch outcome has been resolved.

Branches: 16% of 100 = 16 branches

Total stall cycles = 16 branches $\cdot$ 2 cycles = 32 cycles

Total execution time = 100 instructions + 32 stall cycles = 132 cycles

Average CPI = $\frac{\text{Total Cycles}}{\text{Total Instructions}} = \frac{132}{100} = 1.32$.

b. The processor always fetches instructions sequentially, i.e., it predicts every branch as being Not-Taken. If a branch is resolved as Taken, the incorrectly fetched instructions after the branch are squashed.

Taken branches = 20% of 16 = 3.2 (round to 3 for simplicity)

Total stall cycles = 3 taken branches (rounded from

3.2) $\cdot$ 2 cycles = 6 cycles

Total execution time = 100 instructions + 6 stall cycles = 106 cycles

Average CPI = $\frac{106}{100}$ = 1.06.

c. The processor implements a branch delay slot. The compiler is able to fill the branch delay slot with an instruction that comes before the branch in the original code.

The compiler can place an instruction into the branch delay slot which will always be executed whether the branch is taken or not. This means there are no stall cycles due to branches since the "stall" is effectively used to do useful work.

Total stall cycles = 0 cycles

Total execution time = 100 instructions + 0 stall cycles = 100 cycles

Average CPI = $\frac{100}{100}$ = 1.0.

d. The processor does not implement branch delay slots. Instead, it implements a hardware branch predictor that makes correct predictions for 96% of all branches. When an incorrect prediction is discovered, the incorrectly fetched instructions after the branch are squashed.

With a 96% correct branch prediction rate, only 4% of the branches are mispredicted. Given that 16% of instructions are branches, this means 0.64% of all instructions will suffer from a misprediction (16% $\cdot$ 4%).

Branch instructions = 16

Mispredicted branches = 16 $\cdot$ 4% = 0.64 (rounded to 1 for simplicity)

Total stall cycles = 1 mispredicted branches (rounded from 0.64) $\cdot$ 2 cycles = 2 cycles

Total execution time = 100 instructions + 2 stall cycles = 102 cycles

Average CPI = $\frac{102}{100}$ = 1.02.