For all problems, assume we are starting with the 5-stage pipeline discussed in lecture, unless otherwise stated

Problem 1: Benchmarking *[32 points]*

Consider a 10000 instruction LC-2K benchmark with:

25% **lw**,	30% of which are followed by a dependent instruction
15% **sw**	
30% **add**/**nor**	0% of which are immediately followed by a dependent instruction, and 50% of which are followed by a dependent instruction with a non-dependent instruction in between
20% **beq**,	60% of which are taken
10% **noop/halt**	

Each use of the ALU costs **10ns**, reading memory costs **50ns**, writing memory costs **60ns**, register file reads and writes cost **5ns**, and everything else costs **0ns**. The pipelined design has an additional **10ns** delay due to pipeline registers and control logic.

a. What is the optimal achievable serial runtime of this benchmark? *[Sol'n given in lab slides]*

add / nor	(50 + 5 + 10 + 5)ns = 70ns	**beq**	(50 + 5 + 10)ns = 65ns
lw	(50 + 5 + 10 + 50 + 5)ns = 120ns	**noop / halt**	50ns
sw	(50 + 5 + 10 + 60)ns = 125ns		

Total = 10,000 (0.25 $\times$ 120ns (lw) + 0.15 $\times$ 125ns (sw) + 0.3 $\times$ 70ns $\times$ (add / nor) + 0.2 $\times$ 65ns (beq) + 0.1 $\times$ 50ns (noop / halt)) = 877,500ns

b. What is the benchmark runtime on the single cycle design?
125ns (sw is longest) * 10,000 = 1,250,000 ns

c. How many cycles are needed to complete this benchmark on the multi-cycle design?
10,000 $\times$ (0.15 (sw) + 0.3 (add / nor) + 0.2 (beq)) $\times$ **4 cycles / inst**. + 10,000 $\times$ 0.25 (lw) $\times$ **5 cycles / inst**. + 10,000 $\times$ 0.1 (noop / halt) $\times$ 2 cycles / inst. = 40,500 cycles

d. What is the total execution time of this benchmark on the multi-cycle?
40,500 cycles $\times$ 60ns / cycle = 2,430,000 ns

e. What is the average CPI (Cycles per Instruction) on the multi-cycle?
40,500 cycles / 10,000 instructions = 4.05 cycles / instruction

f. Assuming we fix the code by inserting noops, what is the runtime and CPI on the pipelined design?

0.25 lws × 0.3 imm. dep. × 10000 instructions need 2 noops (Accept 3 noops if the below line has 2 noops to account for the project version of the pipeline)
0.3 adds/nors × 0.5 once removed dep. × 10000 instructions need 1 noop (Accept 2 noop
0.2 beqs × 10000 instructions need 3 noop
Thus, we have 1500 + 1500 + 6000 = 9000 noops
CPI = (10000 + 9000 + 4) cycles / 19000 instructions = 1.00021
Clock period = 60 ns (sw) + 10ns (pipeline registers latency) = 70 ns
Runtime = 19,004 cycles * 70ns = 1,330,280ns

g. Assuming we use Detect-and-Stall to deal with hazards, what is the runtime and CPI on the pipeline?
Runtime is same as avoidance - 1,330,280ns
CPI = (10000 + 9000 + 4) cycles / 10000 instructions = 1.9004

h. Assuming we use Detect-and-Forward for data hazards and Detect-and-stall for control hazards, what is the runtime CPI on the pipeline?
0.25 lws × 0.3 imm. dep. × 10000 instructions need 1 noops
0.2 beqs × 10000 instructions need 3 noops
Thus, we have (750 + 6000) = 6750 noops
CPI = (10000 + 6750 + 4) cycles / 10000 instructions = 1.6754
Runtime = 16754 cycles * 70ns = 1,172,780ns

i. Assuming we use Detect-and-Forward and Speculate-and-squash predicting not taken instead, what is the runtime and CPI on the pipeline?
lw noops are the same = 750
0.2 beqs × 0.6 incorrectly predicted × 10000 instructions need 3 noops
Thus, we have (750 + 3600) = 4350 noops
CPI = (10000 + 4350 + 4) cycles / 10000 instructions = 1.4354
Runtime = 14354 cycles * 70ns = 1,004,780ns

j. Assuming we use Detect-and-Forward and Speculate-and-squash predicting taken, what is the runtime and CPI on the pipeline?
lw noops are the same = 750
0.2 beqs × 0.4 incorrectly predicted × 10000 instructions need 3 noops
Thus, we have (750 + 2400) = 3150 noops
CPI = (10000 + 3150 + 4) cycles / 10000 instructions = 1.3154
Runtime = 13154 cycles * 70ns = 920,780ns

k. Assuming we use Detect-and-Forward and predict every branch correctly, what is the runtime and CPI on the pipeline?
lw noops are the same = 750
beq noops are 0!
Thus, we have 750 noops

CPI = (10000 + 750 + 4) cycles / 10000 instructions = 1.0754
Runtime = 10754 cycles * 70ns = 752,780ns

I. Compare all of the above results. In a few sentences, identify some comparisons with results that might be surprising. Why would these results be surprising, and what does that say about advancements in computer architecture as Moore's law is ending? *[10 points]*

The multi-cycle takes almost double the time as the single cycle. This shows that this architecture is a good attempt to split things up, but doesn't always work if the different cycles take uneven parts of the latency (memory is overpowering in this case). Even the pipeline with naive hazard resolution is slower than the single cycle in this case (but wouldn't be if memory wasn't overpowering or stage split up was better). So we had to add in some forwarding logic to increase parallelism. Both of these suggest that improving the memory system would fix a lot (hmm, what if we developed these things called caches?? Another idea would be to split the mem stages/cycles like in HW4). Finally, with really good branch prediction, we can get a pipeline better than the optimal serial - in effect, beating out what we thought was theoretically possible. So is there something we think is impossible now that we could beat out in the future? (i.e. OoO processing, superscalar, etc.)

Problem 2: Branch Prediction *[18 points]*

Consider the sample assembly code from the project 1 spec. What is the mispredict ratio for each of the following schemes?

- Predicted Not Taken
- Predicted Taken
- 2-bit predictor initialized to strongly not taken, with a separate predictor for each branch
- Forward branches predicted Not Taken, Backward branches predicted Taken
- Predict Taken for unconditional branches, 1-bit per branch for others, initialized to Taken

```
        lw      0       1       five            load reg1 with 5 (symbolic address)
        lw      1       2       3               load reg2 with -1 (numeric address)
start   add     1       2       1               decrement reg1
        beq     0       1       2               goto end of program when reg1==0
        beq     0       0       start           go back to the beginning of the loop
        noop
done    halt                            end of program
five    .fill   5text
neg1    .fill   -1
stAddr .fill    start                   will contain the address of start (2)
```

Contents of reg1 each iteration of the loop: 4, 3, 2, 1, 0

reg1==0? : F, F, F, F, T
→ T/N? : N, N, N, N, T

Unconditional beq 0 0: T, T, T, T

This gives us a 5/9 mispredict rate for predicted not taken, and 4/9 mispredict for predicted taken.

2-bit for the conditional will only mispredict at the end. For the unconditional, the predictor will predict Not Taken for the first 2 times, then will predict taken. So 3/9 mispredict ratio.

For the forward NT / backward T predictor: The unconditional is backwards, and the conditional is forwards. So the conditional would only be mispredicted the last time, and we get 1/9 mispredict ratio.

The unconditional won’t be mispredicted for the last scheme, and the 1-bit for the conditional will mispredict once at the beginning, and at the end. So 2/9 mispredict ratio.