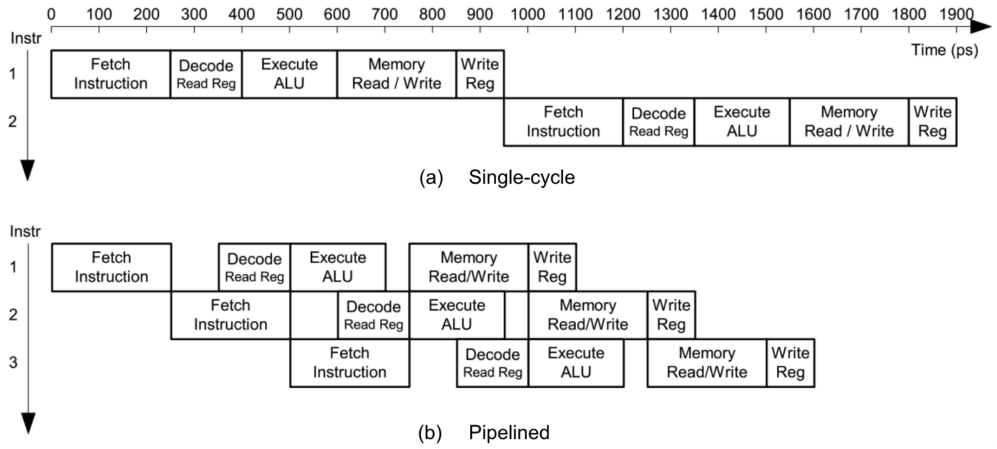
EECS 151/251A Homework 5

Due Friday, October 7th, 2022 11:59PM

Problem 1: Pipelined Design

Here is a diagram that shows timing of datapath stages for both single-cycle processor design, and 5-stage pipelined processor design.



(a) Answer the following two-choice questions about pipelined processor design.

- [① | ②] The main goal of pipelined processor design is to increase ① clock frequency | ② executed instructions per cycle.
- [T | F] Deeper pipelines with more stages can always yield higher performance, because we can execute more instructions in parallel.
- [T | F] The area and power cost is higher for pipelined design than single-cycle design.
- [T | F] Pipelined design generally has lower CPI (cycles per instruction) than single-cycle design.
- [T | F] Pipelined design has lower hardware utilization than single-cycle design, because each instruction can occupy only one stage at a time.
- [T | F] In a single-cycle design, none of the hazards – structural, data and control – can occur.

Solution:

- ①.
- **F**. Pipelines too deep can degrade performance due to higher chance of hazards and pipeline flushes.
- **T**.
- **F**. Pipelined design always has higher CPI because of hazards.
- **F**. Pipelined design has higher hardware utilization because all stages are occupied by different instructions.
- **T**. Hazards don't occur in single-cycle design because all instructions start execution after the previous one completely retires.

(b) Assume that the critical path in *Fetch* stage takes 150ps, *Decode* takes 200ps, *Execute* takes 50ps, *Memory* takes 150ps, and *Writeback* takes 100ps. What would be the minimum clock period achievable for this pipelined design? How much speedup do you achieve over the single-cycle design?

Note: The diagram is not drawn to scale. Also, ignore register clk-q delays or setup/hold time requirements.

Solution:

200ps. Single-cycle design would have clock period $150 + 200 + 50 + 150 + 100 = 650ps$, so there will be $650/200 = 3.25$ speedup in terms of clock frequency.

(c) In your course project, you will be designing a 3-stage pipelined processor instead of a 5-stage one. Which stages would you merge together among the five F/D/X/M/W stages to achieve best performance? Express using formats like FDX/M/W, F/DXM/W, etc.

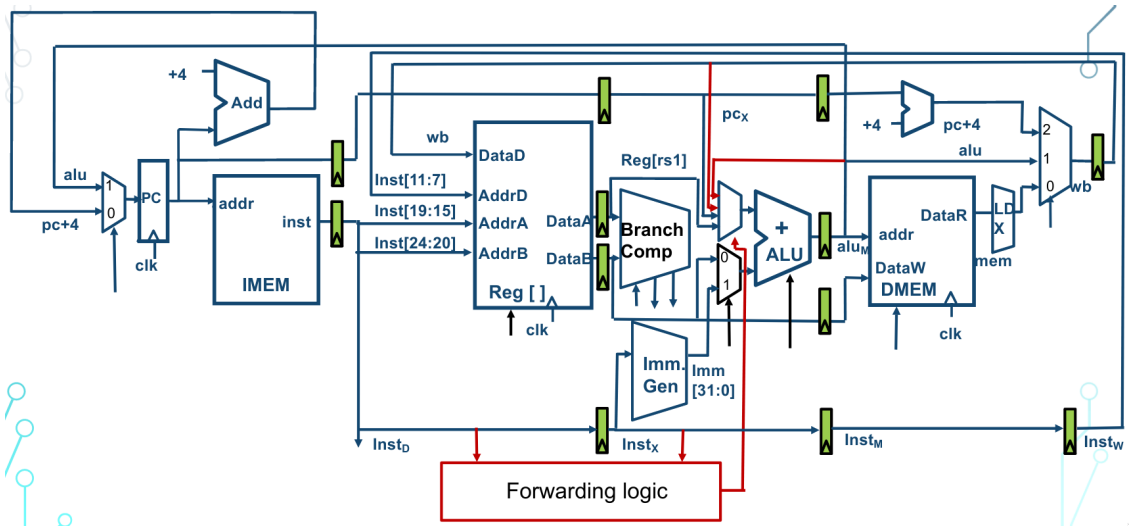
Solution:

F/DX/MW, because that way yields the shortest critical path in the stages (250ps).

Problem 2: Data Hazard and Forwarding

This diagram shows the datapath for a 5-stage pipeline with forwarding logic implemented.

Assume that the write and the read to the same register can happen in a single cycle. Also, assume each stage has the same critical path as Problem 1: *Fetch* 150ps, *Decode* 200ps, *Execute* 50ps, *Memory* 150ps, and *Writeback* 100ps.



(a) First, assume no forwarding is implemented. How many cycles will the following RISC-V code take to execute? What is the CPI of this code? Show how you derived your result.

Note: Count all the cycles starting from when the first instruction enters the fetch stage, to when the last instruction leaves the writeback stage.

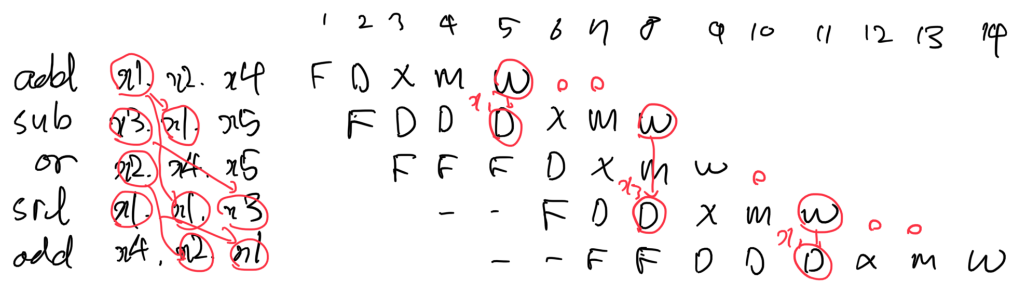
```
and x1, x2, x4
sub x3, x1, x5
or  x2, x4, x5
srl x1, x1, x3
add x4, x2, x1
```

Solution:

14 cycles. Without hazards, it will take $5 + 4 = 9$ cycles; but the data hazard on x1 between 1st-2nd and 4th-5th adds two stall cycles each, and data hazard on x3 between 2rd-4th adds one more stall cycle, resulting in $9+2+2+1 = 14$. CPI becomes $14/5 = \mathbf{2.8}$.

You can also draw the full F/D/X/M/W cycle table, in which case if there are stall cycles identified before 2nd, 4th and 5th instructions, we should give partial points.

Pipeline diagram:



(b) Now, all data forwarding logic including ALU-to-ALU and Mem-to-ALU are implemented, as shown in the datapath diagram. How do the answers of (a) change?

Solution:

This is simply **9 cycles** which includes the unavoidable 4 cycles of pipelining latency. CPI is $9/5 = \mathbf{1.8}$.

(c) Lastly, suppose we are employing a higher-frequency design where the register file is no longer able to write to and read from the same register at a single cycle. However, this faster register file reduces the critical path of the *Decode* (D) stage from 200 down to 100ps.

What are the number of cycles and CPI with this design? Also, taking into account the new critical path of the design, how does the actual time taken to execute this code change from (b)? Assume that we don't add any new forwarding logic to the datapath.

Solution:

Total **11 cycles**, because the `srl` instruction now has to stall at its D stage for two cycles so that it can read register `x3` after the W stage of the `sub` instruction has finished. CPI is **2.2**.

In terms of actual time, the clock period went down from 200ps (the answer for 1-(b)) to 150ps because the D stage is no longer the critical path. So the total time goes down from $9\text{cycles} \times 200ps = 1800ps$ to $11\text{cycles} \times 150ps = 1650ps$, even though the number of cycles has increased.

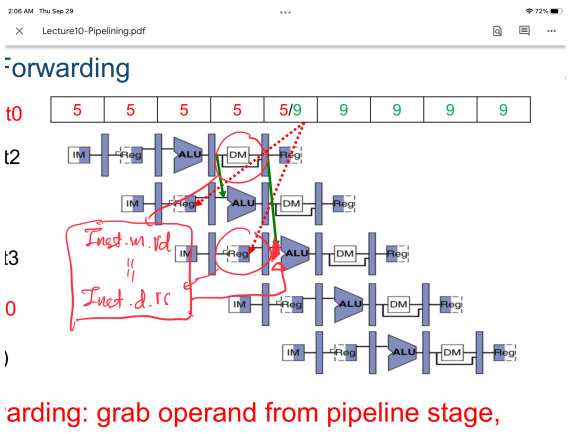
If cycles and CPI are answered correct but the total time is slightly different (but decreased), give **full points**.

If cycles and CPI are right but wall time is not decreased, **-1 points**.
If cycles and CPI are wrong but the answer identifies new stall cycles from the absense of RF forwarding, **half points** should be given.

(d) In the datapath diagram, there is actually a missing input wire to the forwarding logic block that is needed to implement all data forwarding. Which signal should this wire be coming from?

Solution:

There should be a wire coming from $Inst_M$ as well, not just $Inst_D$ and $Inst_X$. This will be used for Mem-to-ALU forwarding, where we check $Inst_D.rs == Inst_M.rd$, and if they match we assert the MUX selection logic for the ALU accordingly.



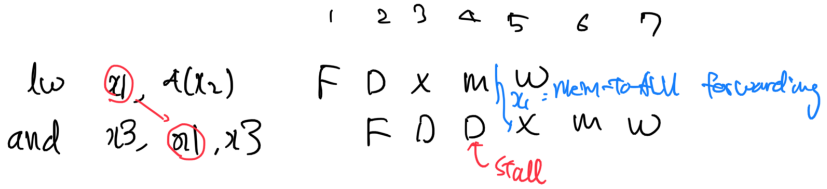
(e) (251A Only) Calculate the number of cycles the following code takes to execute, assuming all aforementioned forwarding logic is implemented as in the diagram. Is the forwarding implementation in the diagram sufficient to eliminate all stalls? If not, how would you change the forwarding datapath to achieve this? Would there be any disadvantage in such implementation?

```
lw  x1, 4(x2)
and x3, x1, x3
```

Solution:

Takes **7** cycles. There is 1 stall cycle in between `lw` and `and`.

This is because `lw` produces x1 at the M stage, whereas `and` uses x1 at the X stage. While we can forward things from MEM-to-ALU, that only works for instructions that are at least two inst apart. These consecutive load-to-use instructions cannot avoid a stall cycle.



One way to solve this is to forward from the output port of the DMEM block to the input of ALU, rather than from the pipeline register that stages the DMEM output port. This allows us to do DMEM read and ALU compute at the same cycle. This will create a very long critical path though, and the decreased clock frequency will likely overshadow the benefit from removing a stall.

If answer gets total cycle wrong (Ex. answered 3 cycles because it didn't include the whole F-D-X-M-W latency) but got 1 stall cycle right, **full** points;
if answer doesn't specify stall cycle between *lw* and *and*, **0** points;
if gets stall correct but does not identify forwarding solutions, **half** points;
if gets forwarding correct but does not identify disadvantages, **-1** points.

Problem 3: Control Hazard

Refer back to the datapath diagram shown in Problem 2.

Assume that all data forwarding logic are implemented so that only control hazards are causing stalls in the pipeline.

(a) Here is a simple loop code in RISC-V.

```
    li   x1, 0
    li   x4, 0
    li   x2, 5
    blez x2, exit
loop:
    lw   x3, 0(x1)
    add  x4, x4, x3
    addi x1, x1, 4
    addi x2, x2, -1
    bnez x2, loop
exit:
    add  x4, x4, x4
```

i. If branches are always predicted not taken, how many branches are predicted correctly? How many are not? What is the total **stall cycles** introduced by branch mispredictions?

Solution:

2 correctly predicted branches, **4 mispredicted branches**, **8 or 12 stall cycles**. The loop executes for 5 times, and there are 30 instructions executed in total, 6 of them being branches. The first branch *blez* is not taken. The second *bnez* is taken 4 times, and then not taken at the end. If we predict-not-taken, there are 4 mispredictions, so we have $4 \times mispredict\_cost = 8$ or 12.

Note: There may be two valid answers for the stall cycles. The stall cycle cost of a branch misprediction can be reduced to 2 cycles (branch comparison happens in X stage, which is two stages after the F stage), however the lecture material states 3 dead cycles. So we should take both answers as correct.

ii. Repeat i. for when the branches are always predicted taken.

Solution:

4 correctly predicted branches, **2 mispredicted branches**, **4 or 6 stall cycles**. Same as above, except that the misprediction count went down from 4 to 2, so we have $2 \times mispredict\_cost = 4$ or 6.

(b) You wrote a branch predictor that can predict branches at 99% accuracy. It is a simple block whose inputs are the PC and branch outcome of the current branch instruction, and whose output is the predicted next PC. Which stage in the datapath would you place this block to be most effective? You don't need to specify how to wire things up correctly, just choose a stage among F/D/X/M/W.

Solution:

F stage. Full points only for the correct answer.

(c) Remember that you have to design a 3-stage pipeline for your project. Could you merge the 5 stages into 3 in such a way that minimizes cycle cost of control hazards? Thinking back to what we did in Problem 1-(b), would this design yield performance improvement for the predict-all-taken case in (a)?

Note: Assume we don't move any components around the datapath, but simply remove pipeline registers to merge stages.

Solution:

We can merge **FDX together** to completely eliminate control hazards. Goal of this problem is to make students understand the fundamental cause for control hazard, which is the timing difference in the pipeline between where branch comparison and PC use happens. However, using numbers from 1-(b), this results in 400ps clock period which is half the clock frequency. Although we can save 4/6 stall cycles out of 34/36, the drop in clock frequency makes it not worth it.