HW05: Pipeline Control Hazards
CSE 30321 Computer Architecture
Name(s): Aaron Wang and Ethan Little

Preamble:

1. Copy this assignment to your Google Drive folder and enter your answers in the boxes provided; **each empty box should be filled in with an answer**. You can either type your answers directly or insert images as long as they are legible. For solutions that require code, use a fixed-width font (Consolas preferred) no smaller than 10 points with proper indentation. Save your solutions as a single PDF file and upload them to Gradescope.
2. You are encouraged to work in groups of 2. Please type your names at the top of this document.

Problem 1

Consider the following C (left) and equivalent pseudo-C (right) where a loop compares data elements from two arrays. Register names in the pseudo-C are used as variable names to tie to the RISC-V code in the pipe traces (below) that implements this loop. Assume N (x20) is initialized prior to this code.

```
while (i < N) {
            tmp1 = arr1[i];
            tmp2 = arr2[i + 1];
            if (tmp1 < tmp2)
              arr2[i + 1] = tmp1;
            else
              arr2[i + 1] = 0;
            i = i + 1;
      }
```

```
while (x3 < x20) {
            x6 = x4[x3];
            x7 = x5[x3 + 1];
            if (x6 < x7)
              x5[x3 + 1] = x6;
            else
              x5[x3 + 1] = 0;
            x3 = x3 + 1;
      }
```

Question 1

Complete the pipe trace for a 5-stage RISC-V pipeline. You should assume (1) the first **bge** instruction is not taken and predicted correctly, and that all correct predictions incur no delay (thus, assume the `slli` is fetched in cycle 2)**;** (2) x6 is *greater than* x7 and the second **bge** instruction is predicted incorrectly; (3) both unconditional branch instructions (i.e., jump instructions) are predicted correctly and their target is known early enough that no delay/stall cycle(s) are needed; (4) on branch mispredictions, the outcome of a branch (conditional or unconditional) is definitively known after it *finishes* the M stage, and there is no additional flush latency - in other words, on a branch misprediction the first instruction on the correct path will be fetched 1 cycle after the branch's Memory stage (this matches the behavior in QtRvSim); (5) there is full forwarding to the extent possible in the datapath, as we have discussed in class.

In all parts of this problem, you may assume that a register can be written and read from the register file in the same clock cycle. You only need to fill this pipe trace for 1 iteration (i.e., the instructions below). Finally, note there may be more columns in the pipe trace than needed.

Question 1 Pipe Trace

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
loop:	bge x3, x20, out	F	D	E	M	W																							
	slli x10, x3, 2		F	D	E	M	W																						
	add x11, x4, x10			F	D	E	M	W																					
	lw x6, 0(x11)				F	D	E	M	W																				
	add x11, x5, x10					F	D	E	M	W																			
	lw x7, 4(x11)						F	D	E	M	W																		
	bge x6, x7, else							F	D	D	E	M	W																
	sw x6, 4(x11)								F	F	D	E																	
	j end										F	D																	
else:	sw x0, 4(x11)												F	D	E	M	W												
end:	addi x3, x3, 1											F		F	D	E	M	W											
	j loop														F	D	E	M	W										
out:																													

Question 2

This is the same code and setup as before but with one change (highlighted) Again, complete the pipe trace. You should assume (1) the first **bge** instruction is not taken and predicted correctly, and that all correct predictions incur no delay (thus, assume the `slli` is fetched in cycle 2)**;** (2) x6 is *greater than* x7 and the second **bge** instruction is predicted incorrectly; (3) the first j instruction is predicted incorrectly, but the second j instruction is again predicted correctly and its target is known early enough that no delay/stall cycle(s) are needed; (4) on branch mispredictions, the outcome of a branch (conditional or unconditional) is definitively known after it *finishes* the M stage, and there is no additional flush latency - in other words, on a branch misprediction the first instruction on the correct path will be fetched 1 cycle after the branch's Memory stage (this matches the behavior in QtRvSim); (5) there is full forwarding to the extent possible in the datapath, as we have discussed in class.

In all parts of this problem, you may assume that a register can be written and read from the register file in the same clock cycle. You only need to fill this pipe trace for 1 iteration (i.e., the instructions below). Finally, note there may be more columns in the pipe trace than needed.

Question 2 Pipe Trace

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
loop:	bge x3, x20, out	F	D	E	M	W																							
	slli x10, x3, 2		F	D	E	M	W																						
	add x11, x4, x10			F	D	E	M	W																					
	lw x6, 0(x11)				F	D	E	M	W																				
	add x11, x5, x10					F	D	E	M	W																			
	lw x7, 4(x11)						F	D	E	M	W																		
	bge x6, x7, else							F	D	D	E	M	W																
	sw x6, 4(x11)								F	F	D	E																	
	j end										F	D																	
else:	sw x0, 4(x11)											F	F	D	E	M	W												
end:	addi x3, x3, 1													F	D	E	M	W											
	j loop														F	D	E	M	W										
out:																													

Question 3

Consider this statement from Q1:
“(3) both unconditional branch instructions (i.e., jump instructions) are predicted correctly **and their target is known early enough that no delay/stall cycle(s) are needed**…”

By the end of which stage (F, D, E, M, or W) must a jump instruction’s target be computed for this to work? How does this compare qualitatively to the assumption from our initial class examples, i.e., up to and including “Option 3: Branch Prediction” in the L12 notes? Justify your answers.

Recall, the j instruction – e.g., `j end`, and `j loop` in this example – is implemented with a `jal` in (this version of) RISC-V.

Note: this question is not asking *how* this can be done, assume it is possible.

Fetch stage. It needs it by the next instructions fetch stage, which will be the same as this decode stage. Therefore, it needs it before this decode stage and needs it done by the end of the fetch stage.

Problem 2
Implement the program from Problem 1 in QtRVSim. Initialize x20 to 5, x4 to 0x600 and x5 to 0x700 all prior to the loop. Put an `ebreak` at `out:`.

Tip: you can copy/paste both columns with the label and code directly from this Google Doc. You may need first to paste it somewhere without the table formatting (e.g., into a blank Google Doc using Shift+Ctrl+v).

Start a simulation (File → New Simulation) with the following *Custom* setup:
- Core ISA and Hazards: pipelined with hazard unit that stalls or forward when hazard is detected
- Branch predictor: ensure the “Branch Predictor” checkbox is **not selected**
- L1 Program Cache, L1 Data Cache, and L2 Cache: ensure the “Enable cache” checkboxes are **not selected**
- All other settings can be left at the defaults

Click “Start Empty” and then compile the assembly program.

Now, **step** through the code, watch the instructions flow through the pipeline on the “Core” visualization, and answer the following questions:

Question 1

Q1a
How are conditional *and* unconditional branches handled? Specifically, how does the implementation decide what to fetch next on a branch, and what is the recovery behavior when it mispredicts (fetches down the wrong path)?

It fetches what is sequentially next in the code, always predicting that the branch isn't taken. On mispredictions, the branch instruction continues to writeback, while flushing the rest of the pipeline and beginning to fetch the correct path after the memory stage.

Q1b
How many cycles does it take to execute this code from start to finish in this pipelined configuration?

It takes 96 cycles total from start to finish. It is 92 excluding the code we added (addi at the beginning and ebreak at the end).

Q1c
Start a simulation with a "No pipeline no cache" configuration. How many cycles does it take to execute this code from start to finish in the non-pipelined configuration?

It takes 55 cycles to execute from start to finish. 51 cycles excluding the code we added.

Question 2

You will now compare the pipelined performance to the non-pipelined performance as the number of loop iterations double (x20 will range from 1000 to 16000). Record the number of cycles for the non-pipelined and pipelined implementations, ***ignoring the cycles for instructions outside the loop***. Fill in the table below.

In addition, do the following:

a. Determine the non-pipelined and pipelined CPI *for the code in the loop only*.
   i. The non-pipelined implementation is a **single-cycle** design,
   ii. For the pipelined design, use our "Ideal + Stall Cycles" method to calculate CPI

b. We know how many instructions are executed in each loop iteration, so check if your CPI is correct by multiplying CPI*Instructions and comparing to the results in your table for the 8000 iterations case.

c. Using your CPI, calculate CPU Time for each implementation for the 8000 iteration case. Finally, calculate the speedup of the pipelined implementation based on these CPU Times.

   Show your work for all calculations.

A few assumptions and tips for this question:

- Click **Max** in QtRVSim so it runs as quickly as it possibly can (no animations, etc.).
  - If working on a laptop, it helps to plug your charger in so the CPU is not throttled.
- Since we never initialized either array, all memory outside the text segment is set to 0. This means x6 and x7 *will always be equal*.

- Note QtRVSim only counts stalls due to load-use hazards in the “Stalls” statistic.
    - Do *not* use this for your calculations below, it won’t count stalls due to branches.
- Assume the clock rate for the non-pipelined design is 1GHz.
- Further, assume the pipelined implementation evenly splits the clock period of the single-cycle design among the 5 stages (this ignores any overhead for writing pipe stage registers, etc.).

Q2a

x20 (loop iterations)	Cycles Non-pipelined	Cycles Pipelined
1000	10000	17000
2000	20000	34000
4000	40000	68000
8000	80000	136000
16000	160000	272000

Q2b
CPI Non-pipelined:
CPI Non-pipelined = 1

CPI Non-Pipelined sanity check for 8000 loop iteration case:
1 CPI*10IPL*8000L = 80000 cycles

CPI Pipelined:
CPI Pipelined = 1.7

CPI Pipelined sanity check for 8000 loop iteration case:
1.7 CPI*10IPL*8000L = 136000 cycles

Q2c
CPU Time non-pipelined for 8000 loop iteration case:
80000*1GHz = $8*10^{-5}$ seconds

CPU Time pipelined for 8000 loop iteration case:
136000*0.2GHz = $2.72*10^{-5}$ seconds

Speedup pipelined:**8/2.72=2.94x**

What to Turn In
Fill in the boxes on the previous pages with answers to all the questions. Save your Google Doc as a PDF and upload it to **Gradescope** for grading. Note Gradescope can be accessed via our Canvas site, or by visiting gradescope.com. Below is a checklist for this assignment:

Problem 1 (50 points)

	Deliverable	Points
1.	Q1 filled pipe trace	20
2.	Q2 filled pipe trace	20
3.	Q3 when must j target be computed	10

Problem 2 (50 points)

	Deliverable	Points
1.	Q1a branch handling explanation	10
2.	Q1b total cycles pipelined	5
3.	Q1c total cycles non-pipelined	5
4.	Q2a table	10
5.	Q2b CPI calculations	10
6.	Q2c CPU Time and speedup calculations	10