HW09: Out of Order Execution
CSE 30321 Computer Architecture
Names: Aaron Wang and Ethan Little

Preamble:

1. **This assignment is only 2 problems, but there are many details and instructions to follow. Read the instructions carefully and ask questions *before* you start (so you don't end up having to redo the pipe traces, etc.)**

1. ***Copy this assignment from our Google Drive directory, not from Gradescope. Downloading the PDF from Gradescope may not preserve all answer boxes.***

2. ***Reminder:*** point totals differ across assignments mainly as a way to allow for partial credit, *but all assignments carry the same weight* (see the syllabus for details).

3. 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.

4. You are encouraged to work in groups of 2. Please type your names at the top of this document.

Problem 1: Dynamic Scheduling
Question 1

Given the assumptions/setup below, fill the pipe trace table, show the final state of the Free List, and show the history of register mappings in the Register Map table. Your ROB and pipe trace should both indicate how registers were renamed, i.e., use t registers and not x registers in both. Do not erase anything from the ROB, Free List, or Map Table – instead, use ~~strikethrough~~ to indicate changes to help us award partial credit as needed. In addition, show the state of the ROB after the **second** `lw` Commits[1].

-Initial register mappings for x1 - x14, and Free List are shown below
-The pipeline can issue 2 instructions per cycle of any type (and F 2, D 2, …, C 2)
-All destination registers are renamed in decode/dispatch; renames are returned to free list in C
-Each instruction spends at least 1 cycle in a reservation station (Issue stage)
-Every ALU operation takes 1 CC in Execute
-Instructions wait in reservation stations until the producer's WriteBack stage, and can execute in the next clock cycle; indicate reservation station stalls with IR as we did in class

[1] You will have *two* copies of the ROB, one that shows the entire history (with strikethrough), and one where you show the state in the cycle when the second lw commits, including the effects of that lw committing.

-There are enough reservation station entries, bus resources for WriteBack, and ALUs to eliminate structural hazards for those resources
-Assume all loads hit in the cache and spend 3 clock cycles in Execute (for address generation + L1 cache access)
-This trace includes a `sw`; stores spend 2 clock cycles in Execute (for address generation + writing the store data to the ROB). Remember, stores have no RF destination, and they can **skip WriteBack**. Stores must still wait to Commit until they are no longer speculative; assume the store data is stashed in the ROB, written during the 2nd cycle of E.

Pipe stages:

Fetch (F), Decode/Dispatch (D), Issue (I), Execute (E), WriteBack (W), Commit (C)

Instructions:
```
lw x4, 0(x14)
add x8, x4, x7
sw x8, 0(x14)
addi x7, x7, -1
addi x14, x14, 4
lw x5, 0(x14)
add x5, x5, x7
sw x5, 0(x14)
```

Pipe Trace (add columns and rows as needed)

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
lw t55, 0(t40)	F	D	I	E	E	E	W	C								
add t3, t55, t30	F	D	I	IR	IR	IR	IR	E	W	C						
sw t3, 0(t40)		F	D	I	IR	IR	IR	IR	IR	E	E	C				
addi t27, t30, -1		F	D	I	E	W	W	W	W	W	W	C				
addi t2, t40, 4			F	D	I	E	W	W	W	W	W	W	C			
lw t8, 0(t2)			F	D	I	IR	IR	E	E	E	W	W	C			
add t14, t8, t27				F	D	I	IR	IR	IR	IR	IR	E	W	C		
sw t14, 0(t2)				F	D	I	IR	IR	IR	IR	IR	IR	IR	E	E	C

Free list: ~~t55~~, ~~t3~~, ~~t27~~, ~~t2~~, ~~t8~~, ~~t14~~, t88, t42, t98, returned: t45, t28, t30, t40, t87, t8

Reorder Buffer (entire state/history)

~~lw t55, 0(t40)~~
~~add t3, t55, t30~~
~~sw t3, 0(t40)~~
~~addi t27, t30, -1~~
~~addi t2, t40, 4~~
~~lw t8, 0(t2)~~
~~add t14, t8, t27~~
~~sw t14, 0(t2)~~

Reorder Buffer (when second lw commits)

add t14, t8, t27
sw t14, 0(t2)

Register Map

x1	t9
x2	t16
x3	t22
x4	~~t45~~, t55
x5	~~t87~~, ~~t8~~, t14
x6	t19
x7	~~t30~~, t27

x8	~~t28~~, t3
x9	t34
x10	t77
x11	t13
x12	t35
x13	t65
x14	~~t40,~~ t2

Question 2

Given the assumptions/setup below, fill the pipe trace table, show the final state of the Free List, and show the history of register mappings in the Register Map table. In addition, show the state of the ROB when the `beq` is in Execute[2]. Your ROB and pipe trace should both indicate how registers were renamed, i.e., use t registers and not x registers in both. Do not erase anything from the ROB, Free List, or Map Table – instead, use ~~strikethrough~~ to indicate changes to help us award partial credit as needed.

-Initial register mappings for x1 - x14, and Free List are shown below
-The pipeline can issue 2 instructions per cycle of any type (and F 2, D 2, …, C 2)
-All destination registers are renamed in decode/dispatch; renames are returned to free list in C
-Each instruction spends at least 1 cycle in a reservation station (Issue stage)
-Every ALU operation takes 1 CC (clock cycle) in Execute
-Branches resolve in Execute and skip WriteBack
-Instructions wait in reservation stations until the producer's WriteBack stage, and can execute in the next clock cycle; indicate reservation station stalls with IR as we did in class
-There are enough reservation station entries, bus resources for WriteBack, and ALUs to eliminate structural hazards for those resources
-Assume all loads hit in the cache and spend 3 clock cycles in Execute (for address generation + L1 cache access)
Branch assumptions:

- -The operands for the `beq` are not ready until cycle 7, so the `beq` cannot enter Execute until cycle 8.
- -The branch is predicted taken, and this is a *misprediction*.
- -The branch prediction hardware is designed such that the instruction after a branch can be fetched in the same cycle, as long as only 2 total instructions are fetched that cycle.
- -On a misprediction:
  - -The first correct path instruction is fetched on the CC after the branch's C
  - -Flushed renames are returned to the free list immediately (in same cycle as branch's C)

Pipe stages:

Fetch (F), Decode/Dispatch (D), Issue (I), Execute (E), WriteBack (W), Commit (C)

Instructions:

```
            beq x12, x14, done
            add x9, x13, x4
            sub x10, x1, x9
done:       lw   x10, 0(x2)
            or x6, x14, x10
            addi x3, x2, 4
```

[2] You will have *two* copies of the ROB, one that shows the entire history (with strikethrough), and one where you show the state in the cycle when the beq is in Execute, including the effects of any instructions in Commit in that cycle (if any).

Pipe Trace (add columns and rows as needed)

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
beq t4 t2	F	D	I	IR	IR	IR	IR	E	C											
lw t16 0(t14)	F	D	I	E	E	E	W	W	-											
or t17 t2 t16		F	D	I	IR	IR	IR	E	-											
addi t18 t14 4		F	D	I	E	W	W	W	-											
add t19 t3 t12										F	D	I	E	W	C					
sub t20 t15 t19										F	D	I	IR	IR	E	W	C			
lw t21 0(t14)											F	D	I	E	E	E	W	C		
or t22 t2 t21											F	D	I	IR	IR	IR	IR	E	W	C
addi t23 t14 4												F	D	I	E	W	W	W	W	C

Free list: ~~t16~~, ~~t17~~, ~~t18~~, ~~t19~~, ~~t20~~, ~~t21~~, ~~t22~~, ~~t23~~, returned: t16, t17, t18 t7, t6, t20, t10, t13

Reorder Buffer (entire state/history; add rows as needed)

~~beq t4 t2~~
~~lw t16 0(t14)~~
~~or t17 t2 t16~~
~~addi t18 t14 4~~
~~add t19 t3 t12~~
~~sub t20 t15 t19~~
~~lw t21 0(t14)~~
~~or t22 t2 t21~~
~~addi t23 t14 4~~

Reorder Buffer (when beq is in Execute; add rows as needed)

beq t4 t2
lw t16 0(t14)

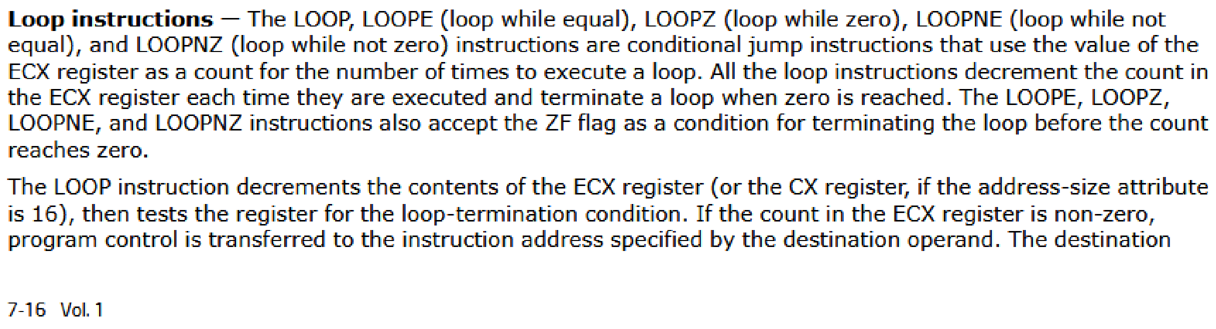
or t17 t2 t16
addi t18 t14 4

Register Map

x1	t15
x2	t14
x3	~~t13~~ ~~t18~~ ~~t13~~ t23
x4	t12
x5	t11
x6	~~t10~~ ~~t17~~ ~~t10~~ t22
x7	t9
x8	t8
x9	~~t7~~ t19
x10	~~t6~~ ~~t16~~ ~~t6~~ ~~t20~~ t21
x11	t5
x12	t4
x13	t3
x14	t2

Extra Credit - 2 points*

I have been careful all semester (and actually, back in Logic Design) to say things like "branches/stores do not write any registers (in the RISC-V ISA considered in this class)." As a counter-example (no pun intended), see the LOOP instruction family in x86:



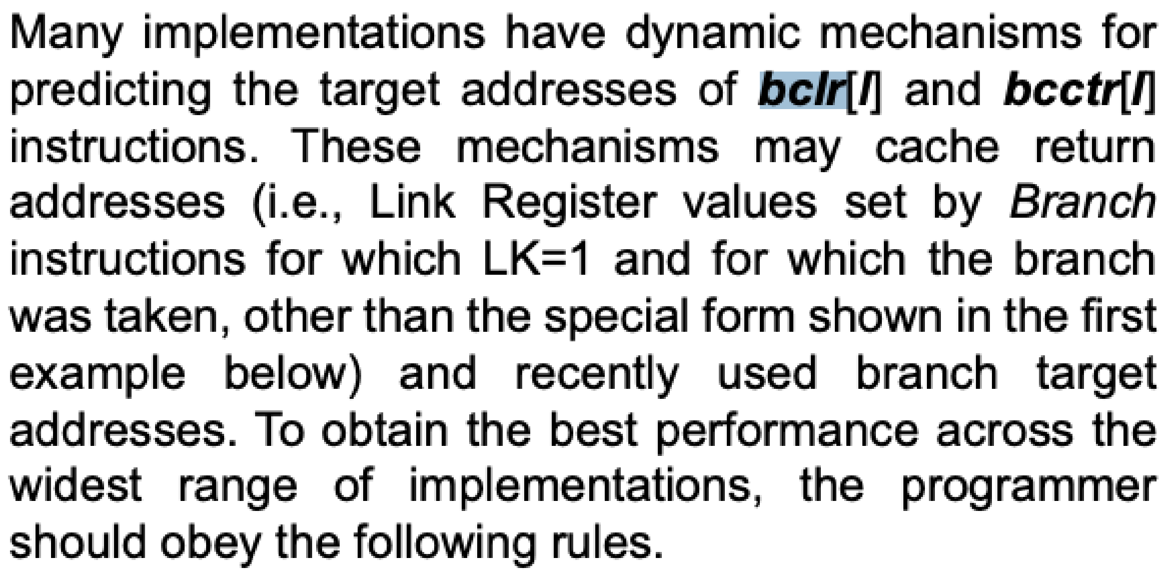
Loop instructions — The LOOP, LOOPE (loop while equal), LOOPZ (loop while zero), LOOPNE (loop while not equal), and LOOPNZ (loop while not zero) instructions are conditional jump instructions that use the value of the ECX register as a count for the number of times to execute a loop. All the loop instructions decrement the count in the ECX register each time they are executed and terminate a loop when zero is reached. The LOOPE, LOOPZ, LOOPNE, and LOOPNZ instructions also accept the ZF flag as a condition for terminating the loop before the count reaches zero.

The LOOP instruction decrements the contents of the ECX register (or the CX register, if the address-size attribute is 16), then tests the register for the loop-termination condition. If the count in the ECX register is non-zero, program control is transferred to the instruction address specified by the destination operand. The destination

7-16 Vol. 1

(source: *Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4,* https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html).

Find one additional example (i.e., not in the x86 LOOP family) of either a store or branch in any ISA that updates a register. The updated register can be a general purpose register (e.g., like x0-x31 in our RISC-V) or a special purpose or condition register. Cite your source, which must be an ISA programmer's manual or equivalent. (While Google searches or "AI" are a way to track down example instructions, they are not sufficient as a source and won't count for full credit.) The example given above is all I am looking for: instruction name, copy/paste the description, and cite your source.


Many implementations have dynamic mechanisms for predicting the target addresses of ***bclr***[*l*] and ***bcctr***[*l*] instructions. These mechanisms may cache return addresses (i.e., Link Register values set by *Branch* instructions for which LK=1 and for which the branch was taken, other than the special form shown in the first example below) and recently used branch target addresses. To obtain the best performance across the widest range of implementations, the programmer should obey the following rules.

page 63 of https://wiki.raptorcs.com/w/images/f/f5/PowerISA_public.v3.1.pdf

bclr stands for Branch Conditional to Link Register, which writes to the link register l which is mainly used for predictions and returning from functions.

In addition to the x86 manual linked above, here are example programmers manuals that would be valid sources:
IBM PowerISA v3.1 pages 37-38 (link goes to large PDF):
https://wiki.raptorcs.com/w/images/f/f5/PowerISA_public.v3.1.pdf

RISC-V specs, specifically, the The RISC-V Instruction Set Manual Volume I: Unprivileged ISA:
https://lf-riscv.atlassian.net/wiki/spaces/HOME/pages/16154769/RISC-V+Technical+Specifications

*This is worth 2 points on this assignment (so max score is 62/60). Remember, all assignments are weighted equally. With 10 assignments, 35% of the grade → 2/60 * (35/10) = 0.117% of final/total grade. There will be another extra credit released at a later date that offers an additional 0.35%.

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 (60 points)

	Deliverable	Points
1.	Question 1	30

2.	Question 2	30

Extra Credit (2 points)

	Deliverable	Points
1.	Instruction name, description, source	2