Please write your name and Kerberos on your solutions.

The only material you may access for this exam is this pdf, the RISC-V ISA handout (https://6004.mit.edu/web/static/fall20/resources/references/6004_isa_reference.pdf) and the quiz 3 answer template if you would like to use it (https://6004.mit.edu/web/static/fall20/resources/quizzes/q3_answer_template.pdf).

Each subproblem is labelled with (Label: X), (e.g., Label: 2A_1). If you are writing your solutions on blank paper, please make sure you write down the label of each subproblem together with its solution and circle each answer. So, for example, if your computed value for problem 2A_1 is 5, you would write:

2A_1: 5

NOTE: For the quiz, you must show your work to get credit for your answers.

Problem 1. Honor Code Agreement (1 points)

We are using the honor system during this exam, and ask that you accept the following terms of this honor system:

1. You will not share the exam with anyone
2. You will not discuss the material on the exam with anyone until after solutions are released
3. You understand that the exam is closed book and that you may only use reference material provided with the exam
4. You understand that you must show your work to get credit for your answers.

(label: 1) I will abide by the above terms (circle one): Agree / Do Not Agree
Problem 2. Operating Systems and Virtual Addresses (16 points)

Consider the following two processes running RISC-V programs labeled with virtual addresses. All pseudoinstructions in these programs translate into a single RISC-V instruction.

program for process A:
. = 0x50
   li a0, 0x250    // load address of string to be printed
   li a7, 0x13     // set system call number for print_string
   ecall
   mv a0, a7
   ret

. = 0x250
string:
   .ascii "printing from program A"

program for process B:
. = 0x550
   li a0, 13
   li a1, 7
   rem a0, a0, a1  //rem rd, rs1, rs2 stores rs1 modulo rs2 into rd
   li a2, 0x700
   mv a0, a2
   sw a0, 0(a2)
   ret

For each of the following questions (A-F), either provide the requested answer, or if there is not enough information to determine the answer, then write CAN’T TELL. You must also provide an explanation for your answers. Assume all memory accesses are legal for parts (A) through (D). Also, assume that neither process A nor B has run yet.

(A) (3 points) The OS schedules process B first, but the processor does not support the rem instruction, so the OS emulates it in software. What are the values of a0 and pc (in virtual address) when the common handler returns to process B after emulating the rem instruction?

(label: 2A_1) a0: _6________

(label: 2A_2) pc: _0x55C____

(label: 2A_3) Explanation:

After the OS emulates rem and returns control, a0 stores the result of rem, 13 mod 7 = 6

The OS returns control to the instruction following rem, at pc = 0x550 + 3*4 = 0x55C
(B) (3 points) Just prior to process B executing `li a2, 0x700`, a timer interrupt occurs and the OS switches to process A. What are the values of a0 and pc (in virtual address) when the common handler returns to process A?

(label: 2B_1) a0: _Can’t tell_

(label: 2B_2) pc: _0x50_____

(label: 2B_3) Explanation:

Since A hasn’t run yet, the pc is the pc of the first instruction, and a0 is at its initial value, which is unknown (a0 = 0 is also correct if you explain that you assume all registers start initialized to 0).

(C) (3 points) After process A executes ecall, what is the value of pc (in virtual address) when the common handler returns to process A?

(label: 2C_1) pc: _0x5C_____

(label: 2C_2) Explanation:

The OS returns control to the instruction following ecall, at pc = 0x50 + 3*4 = 0x5C
(D) (3 points) Just prior to process A executing the `mv` instruction, a second timer interrupt occurs and the OS switches to process B. What are the values of a0 and pc (in virtual address) when the common handler returns to process B?

(label: 2D_1) a0: ___6_______

(label: 2D_2) pc: ___0x55C____

(label: 2D_3) Explanation:

The OS returns control to the `li a2, 0x700` instruction, where the interrupt happened. This is at `pc = 0x550 + 4*3 = 0x55C`, same as (A); a0 is also the same as in (A), since process B has not modified it.

For part (E), assume virtual addresses are translated with the following base and bound registers:

- process A: base register = 0x100, bound register = 0x300;
- process B: base register = 0x500, bound register = 0x600.

*Hint:* Recall that the bound check and the translation from virtual to physical address happen in parallel.

(E) (4 points) Which instructions (if any) involve illegal memory accesses which cause a segmentation fault? Explain why the instructions in your list result in segmentation faults, and explain why all other instructions do not.

(label: 2E_1) List of instructions that result in segmentation faults:

```
sw a0, 0(a2) from process B
```

(label: 2E_2) Explanation:

All instructions are located within each process’s bound, so instruction fetches do not cause segfaults.

The only data access is from the sw instruction, and this segfaults because the virtual address, 0x700, is over process B's bound, 0x600.
Problem 3. Virtual Memory (23 points)

Markus is designing a RISC-V processor with a 32-bit virtual address space, a physical address space of $2^{28}$ bytes, and a page size of 4096 ($2^{12}$) bytes per page. Markus plans to use a page table with an LRU replacement strategy, a 4-entry direct mapped TLB, and a page fault handler to handle missing pages. Assume each page table entry will include a dirty bit and a resident bit.

(A) (4 points) Specify the number of page table entries and the width of each entry. Show your work for full credit.

(label: 3A_1) Number of entries in the page table: $2^{20}$

(label: 3A_2) Width of each page table entry (bits): 18 (28-12+1+1)

(B) (3 points) Markus will be using his processor to run a large number of identical, independent processes that need to store less than 100 bytes each. He noticed while performing a stress test of the processor that he was getting a lot of page faults, which were hurting the system’s performance. He would like to decrease the number of page faults by modifying the page size. Should Markus increase or decrease the page size to improve performance? Explain why. Ignore effects introduced by changing the size of the page table.

(label: 3B)

The page size should be decreased. 100 bytes is a fraction of a 4KB page, so most of the space allocated to each process is unused. Having a larger number of smaller pages can accommodate more processes without paging.
(C) Finally, Markus decides to test his own understanding of virtual memory. To do this he copies down the current state of the 4-entry direct mapped TLB and the first few entries of the page table including which page is the LRU page. They contain the values shown to the right. He then performs the two memory accesses below.

**Note that the last 2 bits of the VPN are used as the line index in the TLB. Also, assume that updates to the TLB caused by TLB hits are only written back to the page table upon eviction of a TLB line.**

1. (6 points) For each of these accesses, compute its corresponding physical address and indicate whether the access causes a TLB miss and/or a page fault in the table below.

**(label: 3C_1)**

<table>
<thead>
<tr>
<th>Virtual Addr</th>
<th>Write / Read</th>
<th>Phys Addr (in hex)</th>
<th>TLB Miss?</th>
<th>Page Fault?</th>
</tr>
</thead>
<tbody>
<tr>
<td>a. 0x4006</td>
<td>Write</td>
<td><em>0xE3006</em></td>
<td>Y / N</td>
<td>Y / N</td>
</tr>
<tr>
<td>b. 0x5776</td>
<td>Read</td>
<td><em>0xFF776</em></td>
<td>Y / N</td>
<td>Y / N</td>
</tr>
</tbody>
</table>

VA = 0x4006 -> VPN = 0x4 -> TLB miss @ index 0 -> Page table walk, resident @ PPN = 0xE3 -> fill TLB & mark dirty (write)

VA = 0x5776 -> VPN = 0x5 -> TLB miss @ index 1 (V=0) -> Page table walk, not resident -> Page fault. Page out LRU VPN. 0x6, use free PPN 0xFF to page in VPN=0x5. Update page table and fill TLB.

2. (8 points) Show any expected changes to the contents of the page table and TLB after the two memory accesses by crossing out any changed values and writing the updated value underneath each modified cell of the TLB and page table.

**(label: 3C_2)**

3. (2 points) Which physical pages (if any) are written back to disk?

**(label: 3C_3) Physical page(s) (or NONE) written back to disk: __0xFF__________________________
Problem 4. Pipelined Processors (20 points)

Val wants to have a Thanksgiving party, but there’s a nasty virus going around. Instead, she will buy the most expensive vegan turkey, a tomato, and a celery. She is at Whole Foods. She already knows that she can buy a vegan turkey for $10 at Wal-Mart, so she will be looking for something more expensive. Val wants to figure out how much her vegan meal will cost so she writes some C code, before converting it into assembly (shown on the right). Assume the registers are initialized to the values specified in the assembly code comments.

C Code

```c
int price[6] = {7, 5, 8, 10, 15, 7};
int maximum = 10;
int celery = 3;
int tomato = 5;
for (int i = 0; i < 6; i++) {
    if (price[i] > maximum)
        maximum = price[i];
}
int total_cost = maximum + celery + tomato;
```

Assembly Code

```assembly
// x4 = 0x24 - length of price in bytes
// x5 = 0x3 - celery
// x6 = 0x5 - tomato
// x7 = 0xA - maximum
// x1 = 0x400 - address of price[0]

start:  addi x2, x0, 0
        slli x2, x2, 2
loop:   add  x8, x2, x1
        lw x3, 0(x8)
        bge x7, x3, skip
        mv x7, x3
        ori x7, x7, 0
skip:   addi x2, x2, 4
        blt x2, x4, loop
        add x7, x7, x5
        add x7, x7, x6
```

In the following five-stage pipelined RISC-V processor (IF, DEC, EXE, MEM, WB):

- All branches are predicted not-taken. (Always fetch from PC + 4).
- Branch decisions are made in the EXE stage.
- The pipeline has full bypassing.
- The processor annuls instructions following taken branches.
- Assume that in the first iteration of the loop both branches are taken.

(A) (label: 4A) (6 points) Her program just started running “start”. Fill in the pipeline diagram for the first 14 cycles. Show all bypass passed used in each cycle.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>addi</td>
<td>slli</td>
<td>add</td>
<td>lw</td>
<td>bge</td>
<td>mv</td>
<td>mv</td>
<td>mv</td>
<td>ori</td>
<td>addi</td>
<td>blt</td>
<td>add</td>
<td>add</td>
<td>add</td>
</tr>
<tr>
<td>DEC</td>
<td>addi</td>
<td>slli</td>
<td>add</td>
<td>lw</td>
<td>bge</td>
<td>bge</td>
<td>bge</td>
<td>mv</td>
<td>NOP</td>
<td>addi</td>
<td>blt</td>
<td>add</td>
<td>NOP</td>
<td></td>
</tr>
<tr>
<td>EXE</td>
<td>addi</td>
<td>slli</td>
<td>add</td>
<td>lw</td>
<td>NOP</td>
<td>NOP</td>
<td>bge</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>addi</td>
<td>blt</td>
<td>NOP</td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>addi</td>
<td>slli</td>
<td>add</td>
<td>lw</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>bge</td>
<td>NOP</td>
<td>NOP</td>
<td>addi</td>
<td>blt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>addi</td>
<td>slli</td>
<td>add</td>
<td>lw</td>
<td>NOP</td>
<td>NOP</td>
<td>bge</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>addi</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6.004 Fall 2020 - 7 of 16 - Quiz #3
(B) (2 points) How many cycles did it take to execute the first loop iteration on this processor? Make sure not to include the first two instructions at label start in your cycle count.

\[ \text{(label: 4B) Cycles to execute first iteration of the loop on this processor: } 11] \]

(C) (4 points) If you could modify your fetch stage to always fetch the correct next instruction instead of predicting all branches not taken, how many cycles will it now take to execute the first iteration of the loop on this modified processor? Explain your answer.

\[ \text{(label: 4C) Removes both branch annulments, so saves 4 cycles, making the cycles per iteration 11-4 = 7.} \]

(D) (4 points) Val spent so much money on vegan turkey that she couldn’t afford a processor with bypassing. In the cheapo processor she bought, all data hazards are resolved by stalling. Also, once again, all branches are predicted not taken.

Her program just started running “start”. Fill in the pipeline diagram for the first 14 cycles:

\[ \text{(label: 4D)} \]

<table>
<thead>
<tr>
<th>Cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>addi</td>
<td>slli</td>
<td>add</td>
<td>add</td>
<td>add</td>
<td>add</td>
<td>lw</td>
<td>lw</td>
<td>lw</td>
<td>bge</td>
<td>bge</td>
<td>bge</td>
<td>bge</td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td>addi</td>
<td>slli</td>
<td>slli</td>
<td>slli</td>
<td>add</td>
<td>add</td>
<td>add</td>
<td>add</td>
<td>lw</td>
<td>lw</td>
<td>lw</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXE</td>
<td>addi</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>slli</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>add</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>addi</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>slli</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>add</td>
<td>NOP</td>
<td>NOP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>addi</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>slli</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>add</td>
<td>NOP</td>
<td>NOP</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(E) (4 points) How many cycles did it take to execute the first loop on this processor? Hint: To answer this question use the bypass and stall information you determined in parts (A) and (D) to calculate how many cycles the first iteration of the loop would take on the cheapo processor. Explain how you arrived at your solution.

\[ \text{(label: 4E_1) Cycles per iteration on this processor: } 24] \]

\[ \text{(label: 4E_2) Explanation: Without bypasses, there is a 3-cycle penalty for each EXE \( \rightarrow \) DEC bypass used in the bypassed pipeline, and a 1-cycle penalty for every WB \( \rightarrow \) DEC bypass used. In part B there were 4 EXE \( \rightarrow \) DEC bypasses and 1 WB \( \rightarrow \) DEC bypass, so cycles = 11 + 4*3 + 1*1 = 24.} \]
Extra pipeline diagrams (for parts A and D):

<table>
<thead>
<tr>
<th>Cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>addi</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>addi</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Problem 5. Synchronization (18 points)

Suppose you have the following three processes that are run on a shared processor. They can coordinate their execution via shared semaphores, using standard signal and wait procedures. Assume that execution may switch between any of the three processes at any point in time.

(label: 5B)

<table>
<thead>
<tr>
<th>Semaphore initial values: <strong>E=1, XorCorStop=0, UT=0</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Process 1</strong></td>
</tr>
<tr>
<td>Loop1:</td>
</tr>
<tr>
<td>wait(E)</td>
</tr>
<tr>
<td>print(“E”)</td>
</tr>
<tr>
<td>signal(XorCorStop)</td>
</tr>
<tr>
<td>wait(XorCorStop)</td>
</tr>
<tr>
<td>print(“C”)</td>
</tr>
<tr>
<td>signal(UT)</td>
</tr>
<tr>
<td>wait(XorCorStop)</td>
</tr>
<tr>
<td>Goto Loop1</td>
</tr>
</tbody>
</table>

(A) (2 points) Could the string “XTEUC” be printed without any synchronization? Explain why or why not.

(label: 5A)

No, process 3 will always print U before T.

Yes is a correct answer is you explain that this string may be appear as a substring of a longer string produced over multiple loop iterations (e.g., UXTEUC).

(B) (12 points) In the above diagram, add calls to signal and wait to ensure the three processes print “EXECUTE” **exactly once**. You should specify a semaphore in each call to signal or wait. Add the minimum number of semaphores and signal and wait calls possible. Specify the initial values for the semaphore(s).

An alternative solution to the one above is to initialize P2’s semaphore (XorCorStop) to 1, and to have two waits at the beginning of the loop (instead of one) and no wait at the end.
(C) (4 points) Explain how you ensured that the processes stop printing after they have printed “EXECUTE” once.

(label: 5C)

P2 calls wait(XorCorStop) after printing C. P1 will unblock P2 when it prints the last E and calls signal(XorCorStop). Then, P2 will block indefinitely on the wait(XorCorStop) call at the beginning of its second loop iteration. At this point, all processes are blocked indefinitely (P1 on the E semaphore, P3 on the UT semaphore), producing the desired behavior.
Problem 6. Pipelined Processor Performance (22 points)

Piper Twain is building a RISC-V processor to accelerate machine learning applications in low-power systems. These applications execute mostly ALU operations, so Piper is building the Twofer, a new processor that can perform two ALU operations in a single instruction. She is imposing some restrictions on these operations to make the Twofer’s implementation very cheap: the processor requires two ALUs but avoids using register files with many ports.

To this end, Piper designs a new RISC-V instruction type, D-type (for double-op), that encodes two ALU operations. In assembly, a D-type instruction is written as two operations separated by a semicolon:

\[ \text{aop ard, ars1, ars2; bop brd, brs1, brs2} \]

For instance, the following two examples are valid D-type instructions:

1. \[ \text{add x3, x2, x1; sub x27, x18, x25} \]
2. \[ \text{and x18, x2, x1; xor x7, x18, x25} \]

These fields have the following semantics and restrictions:

- The operations (aop, bop) can be any of the 10 ALU operations for R-type instructions (add, sub, sll, srl, sra, slt, sltu, and, or, xor).
- The source registers for operation A (ars1, ars2) must be within the top (lower index) 16 registers (x0-x15); and the source registers for operation B (brs1, brs2) must be within the bottom (higher index) 16 registers (x16-x31).
- The destination registers for operations A and B must be in separate halves of the registers: one operation must write to the top half of the registers (x0-x15), and the other must write to the bottom half (x16-x31), but ard and brd must never be in the same half.
- Within the same instruction, both operations have serial semantics. That is, if operation A writes to a register that operation B reads, operation B uses the value written by operation A (this may only happen if A writes to a bottom-half register and B writes to a top-half register). For example, in example (2) above, A (and) writes to x18, and B (xor) reads x18. The xor operation should use the value of x18 produced by the and operation.

D-type instructions can be encoded in 32 bits and added to a RISC-V processor, but their specific encoding is not needed for this problem.

Piper has designed the following 6-stage pipeline for the Twofer:
This pipeline uses two separate register files, RegFileA and RegFileB, which hold the top and bottom halves of the RISC-V registers. Each of these register files has only 2 read ports and 1 write port, and 16 registers, so together they are as cheap as the standard 32-register RISC-V register file.

The IF and MEM stages are like those in the standard 5-stage pipeline. The DEC stage reads the source registers for operation A (from RegFileA). The EXA stage executes operation A and, in parallel, reads the source registers for operation B (from RegFileB). The EXB stage executes operation B. Both results are sent down to the MEM stage. Finally, the WB stage writes to both RegFileA and RegFileB; because operations A and B cannot both write to the same register file, a single write port suffices for each.

Having operations A and B happen in separate stages makes implementing serial semantics easy: if operation A writes to one of the source registers of operation B, the muxes at the outputs of RegFileB (colored in light blue in the figure) pass this updated value to EXB.

Beyond D-type instructions, this pipeline supports all other RISC-V instruction types. Other instructions execute on EXA or EXB depending on the registers they use. Every instruction performs its operation in EXA if its source register(s) are in the top half, and in EXB if in the bottom half. For instructions with two source registers in separate halves (this includes register-register ALU and branch instructions), one of the source registers is read in DEC, the other in EXA, and execution happens in EXB.

You don’t need to draw pipeline diagrams to answer the following questions, and we recommend you don’t. But if you find them useful, the last page has some blank Twofer pipeline diagrams.

(A) (10 points) For each of the following code fragments, find out how many cycles are lost to stalls due to data hazards. Assume full bypassing, and for each example, list all bypass paths are used, if any. Note that bypassing here is more involved than in the 5-stage pipeline: bypassed data is not only routed to DEC. For each bypass path used, fill the given table to list the register whose value is bypassed and the stages it is bypassed from and to.

Note: Bypassing always happens from a later to an earlier pipeline stage, so the light-blue muxes in EXA used to implement serial semantics on operations A and B are not a bypass.

Example:
```
lw x20, 0(x2)  Cycles lost to stalls: 2
add x16, x20, x22  Bypass paths:
```

<table>
<thead>
<tr>
<th>Value of reg</th>
<th>Bypassed from</th>
<th>Bypassed to</th>
</tr>
</thead>
<tbody>
<tr>
<td>x20</td>
<td>WB</td>
<td>EXA</td>
</tr>
</tbody>
</table>

(since add runs on EXB)
1. lw x2, 0(x2)
   add x2, x2, x2
   (label: 6A_1_1) Cycles lost to stalls: _3________
   (label: 6A_1_2) Bypass paths:
   add runs on EXA in this case

2. add x1, x2, x1; or x17, x18, x17
   sub x1, x2, x1; xor x17, x18, x17
   (label: 6A_2_1) Cycles lost to stalls: _0________
   (label: 6A_2_2) Bypass paths:
   Value of reg | Bypassed from | Bypassed to
   --- | --- | ---
   x2 | WB | DEC

3. add x1, x2, x1; or x19, x18, x17
   add x17, x1, x19
   (label: 6A_3_1) Cycles lost to stalls: _0________
   (label: 6A_3_2) Bypass paths:
   Value of reg | Bypassed from | Bypassed to
   --- | --- | ---
   x1 | EXA | DEC
   x17 | EXB | EXA

   add runs on EXB. Note bypasses happen on different cycles. It’s also correct to say \( x1 \) value is bypassed from EXB to EXA (then both bypasses would be in the same cycle)

4. add x18, x2, x1; or x1, x18, x17
   add x1, x2, x1
   (label: 6A_4_1) Cycles lost to stalls: _1________
   (label: 6A_4_2) Bypass paths:
   Value of reg | Bypassed from | Bypassed to
   --- | --- | ---
   x1 | EXB | DEC

   add runs on EXA
(B) (3 points) Assume that the pipeline always predicts branches not taken. What is the minimum and maximum number of cycles lost to taken branches in this pipeline? Briefly explain what causes this penalty. How would you write code to minimize this penalty?

(label: 6B_1) Minimum number of cycles lost: __2________

(label: 6B_2) Maximum number of cycles lost: __3________

(label: 6B_3) What causes the penalty, and how would you write code to minimize it?

This penalty depends on whether branches are resolved in EXA (2 cycles) or EXB (3 cycles). To minimize this penalty, we should use branches that run on EXA, by operating on the bottom half of registers (x0-x15).

(C) (6 points) Consider the following RISC-V code. Assume that the pipeline has full bypassing and that the branch at the end of the loop is predicted taken, so that in steady state there are no cycles lost to control hazards. Rewrite the four R-type instructions shaded in grey, using D-type instructions to maximize performance. In steady state, how many cycles does each loop iteration take, in both the original code and your code?

Notes:
• You may have to change the registers used by these instructions to maximize performance.
• Registers x4, x5, x6, and x7 are written and read only by these instructions.
• Registers x10, x11, and x12 hold constant values. These constants are replicated in x20, x21, and x22, so you can use x20 in place of x10, etc.
• You may not modify the code (instructions or registers) outside of the shaded block.

loop: lw x16, 0(x1)  
  addi x1, x1, 4  
  srl x4, x16, x10  
  sll x5, x7, x11  
  or x6, x5, x12  
  xor x7, x6, x4  
  blt x1, x13, loop

(label 6C_1) Rewrite shaded instructions:

  _sll x5, x7, x11; srl x17, x16, x20________
  _or x18, x5, x12; xor x7, x18, x17________
  _(x17 and x18 can be any bottom-half registers)________

(label: 6C_2) Cycles per loop iteration for original code: __8______________

(7 instructions + 1 stall cycle due to lw -> srl dependency, as srl runs on EXB)

(label: 6C_3) Cycles per loop iteration for your modified code: __6______________

(5 instructions + 1 stall cycle due to lw -> srl dependency)
(D) (3 points) Bypasses are expensive, so Piper is considering not implementing them. Are there any RISC-V instruction sequences that would incur some stalls in a conventional 5-stage pipeline without bypasses but that, when modified to use D-type instructions, would incur fewer stalls in the Twofer 6-stage pipeline without bypasses? If so, give such a sequence; in any case, briefly explain why or why not.

(label: 6D)

Yes. The following pair of instructions:
- \texttt{add x16, x1, x1}
- \texttt{add x1, x16, x16}

suffers a 3-cycle stall on the 5-stage pipeline, while the equivalent D-type instruction:
- \texttt{add x16, x1, x1; add x1, x16, x16}

has no stalls.

Blank Twofer pipeline diagrams (in case you find them useful)

<table>
<thead>
<tr>
<th>Cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

END OF QUIZ 3!