Please enter your name, Athena login name, and recitation section above. Enter your answers in the spaces provided below.

The last page of the quiz has scratch copies of some diagrams. This quiz has a separate appendix that includes the RISC-V calling convention and the 6.004 RISC-V reference.

Problem 1. Operating Systems (11 points)

(A) (3 points) You are given a RISC-V processor that does not have hardware support for multiplications, but you still need to do multiply operations in your programs. Which of the following are valid approaches to perform multiplications? (Select all that apply)
   a. Use the normal `mul` RISC-V instruction in your programs, and use exceptions to emulate them in software.
   b. Write a `multiply` function in assembly, and use it as a normal function every time you want to perform a multiplication.
   c. Write a `multiply` function and replace every `mul` instruction that would be in the code by `jal multiply`.

(B) (2 points) Addresses that are used for memory-mapped I/O (MMIO):
   a. Are special, and most of the time they cannot be cached.
   b. Should always be cached, this way we can perform MMIO faster.

(C) (2 points) An unsupported instruction at pc X causes an exception, and the kernel emulates this instruction in software. Where should the kernel return control to?
   a. Address X
   b. Address X + 4

(D) (2 points) An interrupt always returns control to the process that was interrupted.
   a. True
   b. False

(E) (2 points) System calls (invoked using `ecall` in RISC-V):
   a. Are a type of exception.
   b. Are a type of interrupt.
   c. Are the same as a function and can be used interchangeably.
Problem 2. Virtual Memory (26 points)

For the following questions, assume a RISC-V processor with 32-bit virtual addresses, 24-bit physical addresses and page size of 256 \( (2^8) \) bytes per page. The Page Table of this processor uses an LRU replacement strategy, and handles missing pages using a page fault handler.

(A) (4 points) What is the size of the page table? Assume that each page table entry includes a dirty bit and a resident bit. Specify the number of page table entries and the width of each entry.

Number of entries in the page table: \( 2^{24} \)

Width of each page table entry (bits): 18

(B) (4 points) If we double the page size to 512 \( (2^9) \) bytes but keep the same virtual and physical address lengths, what effect will the change have on the size of a page table entry and on the number of entries in the page table? Use a letter “a” through “e” to indicate how the new value of the parameter compares to the old value of the parameter:

(a) doubled      (b) increased by 1 (c) stays the same      (d) decreased by 1 (e) halved

Number of entries in the page table: 4

Width of each page table entry in bits: 12

You now run a test program on the original processor that has 256-byte pages.

Execution of this test program is halted just before executing the following two instructions. The first 8 locations of the page table at the time execution was halted are shown to the right; the least recently used page (“LRU”) and next least recently used page (“next LRU”) are as indicated. Execution resumes and the following two instructions at locations 0x1FC and 0x200 are executed:

```
lw x11, 0x34C(x0)  |  PC = 0x1FC
sw x11, 0x604(x0)  |  PC = 0x200
```

(C) (8 points) Assume that all the code and data for handling page faults is located on physical page 0. When the processor executes the two instructions above it will access five different physical pages. Please list the five physical page numbers in hex, in the order they are first accessed.

Hint: All the page table information you need is shown in the table to the right. Note that if a page fault occurs, the processor accesses physical page 0 to handle the fault.

<table>
<thead>
<tr>
<th>User-mode</th>
<th>D</th>
<th>R</th>
<th>PPN</th>
</tr>
</thead>
<tbody>
<tr>
<td>page table</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LRU→2</td>
<td>0</td>
<td>1</td>
<td>0x602</td>
</tr>
<tr>
<td>next LRU→5</td>
<td>0</td>
<td>1</td>
<td>0x097</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>1</td>
<td>0x790</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Physical page numbers (hex): 0x007, 0x097, 0x602, 0x097, 0x790
(D) (2 points) Our processor has a **direct-mapped, 4-entry TLB**. Which bits of the VPN should the TLB use as index bits? Which VPN bits should the TLB use for its tag field?

**VPN bits used for TLB index:** VPN[\_1\_ : \_0\_

**VPN bits used for TLB tag field:** VPN[\_23\_ : \_2\_]

(E) (8 points) The figure below shows the contents of the TLB prior to the execution of the two instructions in the previous question. For simplicity, we show the tag and index bits of the VPN, even though the TLB needs to store the tag bits only. Fill in the contents of the TLB after the execution of each of the two instructions. If an entry has not changed, you may write NO CHANGE over it instead of copying it.

**Note:** The TLB caches page table entries, which the page fault handler may modify. To ensure correct behavior, most processors include a privileged instruction to invalidate a TLB entry. Assume that the page fault handler uses that instruction. Also assume that all page fault handler accesses happen through physical memory and do not go through the TLB.

**Initial TLB contents:**

<table>
<thead>
<tr>
<th>Index</th>
<th>VPN (Tag+Index)</th>
<th>V</th>
<th>D</th>
<th>PPN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x0</td>
<td>1</td>
<td>0</td>
<td>0x123</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>0x2</td>
<td>1</td>
<td>0</td>
<td>0x602</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Fill in TLB contents after execution of** \( \text{l}w \ x11, \ 0x34C(x0) \ | \text{PC} = 0x1FC \)

<table>
<thead>
<tr>
<th>Index</th>
<th>VPN (Tag+Index)</th>
<th>V</th>
<th>D</th>
<th>PPN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><strong>NO CHANGE</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x1</td>
<td>1</td>
<td>0</td>
<td>0x007</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>0x3</td>
<td>1</td>
<td>0</td>
<td>0x602</td>
</tr>
</tbody>
</table>

* D = 1 is also correct
* Old values in fields other than V are correct

**Fill in TLB contents after execution of** \( \text{s}w \ x11, \ 0x604(x0) \ | \text{PC} = 0x200 \)

<table>
<thead>
<tr>
<th>Index</th>
<th>VPN (Tag+Index)</th>
<th>V</th>
<th>D</th>
<th>PPN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><strong>NO CHANGE</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td><strong>NO CHANGE</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0x6</td>
<td>1</td>
<td>1</td>
<td>0x790</td>
</tr>
<tr>
<td>3</td>
<td><strong>NO CHANGE</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Problem 3. Pipelined Processors (20 points)

(A) (8 points) Consider a five-stage pipelined RISC-V processor (IF, DEC, EXE, MEM, WB), where:
- All branches are predicted not-taken.
- Branch decisions are made in the EXE stage.
- The pipeline has full bypassing.

Assume that the loop shown to the right has been running for a while and will be re-executed after this iteration. Complete the next 10 cycles of the pipeline diagram below. **Draw an arrow showing any use of bypass paths from the EXE, MEM, or WB stages to the DEC stage.** Denote annulled instructions and stalls using NOPs.

If you need them, additional pipeline diagrams are available at the end of the quiz.

<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>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>add</td>
<td>slli</td>
<td>xor</td>
<td>bnez</td>
<td>and</td>
<td>or</td>
<td>add</td>
<td>slli</td>
<td>xor</td>
<td>bnez</td>
</tr>
<tr>
<td>DEC</td>
<td>add</td>
<td>slli</td>
<td>xor</td>
<td>bnez</td>
<td>and</td>
<td>NOP</td>
<td>add</td>
<td>slli</td>
<td>xor</td>
<td></td>
</tr>
<tr>
<td>EXE</td>
<td>add</td>
<td>slli</td>
<td>xor</td>
<td>bnez</td>
<td>NOP</td>
<td>NOP</td>
<td>add</td>
<td>slli</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>add</td>
<td>slli</td>
<td>xor</td>
<td>bnez</td>
<td>NOP</td>
<td>NOP</td>
<td>add</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>add</td>
<td>slli</td>
<td>xor</td>
<td>bnez</td>
<td>NOP</td>
<td>NOP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(Full credit if bypass arrow in cycle 6 is not shown)

(B) (2 points) How many cycles does it take to execute one iteration of the loop on this processor?

Cycles per iteration on this processor: ___6_____

loop: add x3, x2, x1
      slli x2, x2, 1
      xor x4, x3, x4
      bnez x2, loop
      and x11, x2, x1
      or x12, x22, x7
      sra x13, x23, x8
      ...
(C) (8 points) Now consider a variant of the previous five-stage pipelined RISC-V processor (IF, DEC, EXE, MEM, WB), where:

- All branches are predicted not-taken.
- Branch decisions are made in the EXE stage.
- The pipeline has no bypass paths.

Assume that our example loop (same as the previous one, reshown to the right for convenience) has been running for a while and will be re-executed after this iteration. Complete the next 10 cycles of the diagrams below. Specifically:
  - Fill out the pipeline diagram, specifying the instruction at each stage for each cycle.
  - Denote annulled instructions and stalls using NOPs.

<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>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>add</td>
<td>slli</td>
<td>xor</td>
<td>bnez</td>
<td>bnez</td>
<td>bnez</td>
<td>and</td>
<td>or</td>
<td>add</td>
<td>slli</td>
</tr>
<tr>
<td>DEC</td>
<td>add</td>
<td>slli</td>
<td>xor</td>
<td>xor</td>
<td>xor</td>
<td>xor</td>
<td>bnez</td>
<td>NOP</td>
<td>add</td>
<td>slli</td>
</tr>
<tr>
<td>EXE</td>
<td>add</td>
<td>slli</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>xor</td>
<td>bnez</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
</tr>
<tr>
<td>MEM</td>
<td>add</td>
<td>slli</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>xor</td>
<td>bnez</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
</tr>
<tr>
<td>WB</td>
<td>add</td>
<td>slli</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>xor</td>
<td>bnez</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
</tr>
</tbody>
</table>

(D) (2 points) How many cycles does it take to execute one iteration of the loop on this processor?

Cycles per iteration on P2: ___8______
Problem 4. Processor Pipeline Performance (14 points)

You are designing a four stage RISC-V processor (IF, DEC, EX, WB) that:
- Resolves branches in the EX stage.
- Uses a scoreboard to track and stall on data hazards.
- Initiates accesses to data memory in the EX stage, but data memory responses are not available until the WB stage.

This processor has a tight area constraint, and you only have enough area for one bypass path for read-after-write hazards. You are trying to decide whether to put the bypass path from EX to DEC or from WB to DEC. As part of this evaluation, you construct two processors:

**Processor A:** Bypassing from WB to DEC.
**Processor B:** Bypassing from EX to DEC.

These processors ensure that if the instruction in the DEC stage does not have all of its operands ready to read from the register file or read from a bypass path, the instruction in the decode stage is stalled.

You are using the following loop of a program to evaluate the performance of the processor:

```
loop: lw t0, 0(a0)
    addi t1, t0, 1
    xor t3, t1, a0
    slli t2, t1, 1
    blt t0, a0, loop
```

For the following questions, assume this loop has been running for a long time, and that the processor assumes that the branch will be taken.

(A) (6 points) How many cycles does this loop take to execute in each of the processors?

**Processor A cycles per iteration:** ___7__________  
**Processor B cycles per iteration:** ___8__________

You are not required to fill these pipeline diagrams out, but you may 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>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>lw</td>
<td>addi</td>
<td>xor</td>
<td>xor</td>
<td>slli</td>
<td>slli</td>
<td>blt</td>
<td>lw</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td>lw</td>
<td>addi</td>
<td>addi</td>
<td>xor</td>
<td>xor</td>
<td>slli</td>
<td>blt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EX</td>
<td>lw</td>
<td>NOP</td>
<td>addi</td>
<td>NOP</td>
<td>xor</td>
<td>slli</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>lw</td>
<td>NOP</td>
<td>addi</td>
<td>NOP</td>
<td>xor</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>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>lw</td>
<td>addi</td>
<td>xor</td>
<td>xor</td>
<td>slli</td>
<td>blt</td>
<td>blt</td>
<td>lw</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td>lw</td>
<td>addi</td>
<td>addi</td>
<td>addi</td>
<td>xor</td>
<td>slli</td>
<td>blt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EX</td>
<td>lw</td>
<td>NOP</td>
<td>NOP</td>
<td>addi</td>
<td>xor</td>
<td>NOP</td>
<td>slli</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>lw</td>
<td>NOP</td>
<td>NOP</td>
<td>addi</td>
<td>xor</td>
<td>NOP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Assume the two processors have ideal registers ($t_{\text{setup}} = t_{\text{hold}} = t_{\text{pd}} = t_{\text{cd}} = 0 \text{ ns}$) and the following propagation delays for their pipeline stages:

- IF: 2 ns
- DEC: 3 ns
- EX: 4 ns
- WB: 2 ns

The logic for the bypass path of processor A can be viewed as taking the output from the DEC and WB stages and adding additional bypass logic (BYP) as shown in the picture below.

The logic for the bypass path of processor B can be viewed as taking the output from the DEC and EX stages and adding additional bypass logic (BYP) as shown in the picture below.

Assume that the BYP logic has a propagation delay of 1 ns.

(B) (4 points) What is the minimum clock period for each processor?

- Clock period for processor A (ns): ___4___________
- Clock period for processor B (ns): ___5___________

(C) (2 points) For the loop shown above, what is the average cycles per instruction for each processor?

- Average cycles per instruction for processor A: ___7/5___________
- Average cycles per instruction for processor B: ___8/5___________

(D) (2 points) For the loop shown above, what is the average number of instructions per nanosecond for each processor?

- Average number of instructions per nanosecond for processor A: ___5/28 = 0.179_____
- Average number of instructions per nanosecond for processor B: __5/40 = 1/8 = 0.125_
Problem 5. Synchronization (9 points)

In lecture, you saw the use of semaphores to mediate a communication stream between a
Producer and a Consumer process. In this problem, we assume the existence of three
asynchronous processes: a Producer process, producing a stream of characters; a Consumer
process, which consumes a stream of characters; and a Filter process spliced between the
Consumer and Producer processes. The Filter process takes characters from the producer,
processes them (via a translate function), and passes the result to the Consumer process. The
Producer and Consumer processes each communicate directly only with the Filter process.

The following is in Shared Memory (shared among Producer, Filter, and Consumer processes):

```
Semaphore charsA=???, spaceA=???, charsB=???, spaceB=???
char buf[100];
char indata;
int in=0, out=0;
```

and the following code runs in the Filter Process:

```
while (1) {                  /* loop forever… */
    char temp;                /* local variable */

    wait(charsA);
    temp = indata;
    signal(spaceA);
    temp = translate(temp);   /* do the actual translation */
    wait(spaceB);
    buf[in] = temp;
    in = (in+1)%100;          /* increment ‘in’ modulo 100 */
    signal(charsB);
}
```

(A) (1 point) What is the maximum number of characters that can be produced by the Producer
process but not yet processed by the Filter process?

Maximum unprocessed characters produced: _______1_________

(B) (2 points) What are appropriate initial values for each of the semaphores?

initial value for charsA: _______0_________
initial value for spaceA: _______1_________
initial value for charsB: _______0_________
initial value for spaceB: _______100_________
(C) (4 points) For each of the following lines of code, indicate whether you would expect to find them in the Producer process, the Consumer process, or Neither process:

\[
\text{out} = (\text{out}+1) \% 100; \quad \text{in process (circle one): P C Neither}
\]
\[
\text{signal}(\text{charsA}); \quad \text{in process (circle one): P C Neither}
\]
\[
\text{signal}(\text{indata}); \quad \text{in process (circle one): P C Neither}
\]
\[
\text{wait}(\text{charsB}); \quad \text{in process (circle one): P C Neither}
\]

(D) (2 points) Assuming that the only process synchronization appearing in the Producer and Consumer processes is the use of the four semaphores shown, will the above implementation work with multiple Consumer processes? Multiple Producer processes? Multiple Filter processes? “Work” means each character produced by a Producer is translated by exactly one Filter process and then consumed by exactly one Consumer process, i.e., no character is lost or processed twice.

Works with multiple Consumers (circle one): YES NO
Works with multiple Producers (circle one): YES NO
Works with multiple Filters (circle one): YES NO

END OF QUIZ 3!
Have a great summer!