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: (e)

Width of each page table entry in bits: (d)

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>Physical page numbers (hex):</th>
<th>0x007</th>
<th>____</th>
<th>0x602</th>
<th>____</th>
<th>0x097</th>
<th>____</th>
<th>0x790</th>
</tr>
</thead>
<tbody>
<tr>
<td>User-mode page table</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>R</td>
<td>PPN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0x123</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0x007</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LRU→2</td>
<td>0</td>
<td>1</td>
<td>0x602</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>–</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>0xACE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>next LRU→5</td>
<td>0</td>
<td>1</td>
<td>0x097</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>0x790</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>–</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
(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: \(\text{VPN}[\_1\_ : \_0\_]\)

VPN bits used for TLB tag field: \(\text{VPN}[\_23\_ : \_2\_]\)

(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></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></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fill in TLB contents after execution of \(\text{lw} \ x11, \ 0x34C(x0) \mid \text{PC} = 0x1FC\):

<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>NO CHANGE</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></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 \(\text{sw} \ x11, \ 0x604(x0) \mid \text{PC} = 0x200\):

<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>NO CHANGE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>NO CHANGE</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>NO CHANGE</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Problem 3. Pipelining Combinational Circuits (20 points)

For each of the questions below, please create a valid K-stage pipeline of the given circuit. Each component in the circuit is annotated with its propagation delay in nanoseconds. **Show your pipelining contours** and place large black circles (●) on the signal arrows to indicate the placement of pipeline registers. Give the latency and throughput of each design, assuming ideal registers (t_{PD}=0, t_{SETUP}=0). Remember that our convention is to place a pipeline register on each output.

**If you need them, additional copies of these circuits are available at the end of the quiz.**

(A) (2 points) Show the maximum-throughput 1-stage pipeline.

(B) (4 points) Show a maximum-throughput 2-stage pipeline.
(C) (6 points) Show a **maximum-throughput pipeline** that uses the smallest possible number of pipeline stages.

![Diagram](image)

Latency (ns): __12_____

Throughput (ns\(^{-1}\)): _1/4_____

(D) (8 points) You reimplement the 4ns combinational component in the previous circuit using two faster components connected in series, shown in grey below. You can choose the propagation delays of these two components, as long as their delays add to 4ns (e.g., they could be 3ns and 1ns, 2ns and 2ns, etc.)

Choose the propagation delays of both components in a way that lets you pipeline the circuit for maximum throughput while minimizing the number of pipeline stages. Then, find the **maximum-throughput pipeline**. For full credit, your solution should use the minimum possible number of pipeline stages.

![Diagram](image)

Latency (ns): __9_____

Throughput (ns\(^{-1}\)): _1/3_____

---

6.004 Fall 2018 - 5 of 11 - Practice Quiz #3A
Problem 4. 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 non-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 killed instructions and stalls using NOPs.

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

(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_____

<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>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</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>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</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>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</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>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
</tr>
</tbody>
</table>

(Full credit if bypass arrow in cycle 6 is not shown)
(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 non-taken.
- Branch decisions are made in the EXE stage.
- The pipeline has **no bypass paths**.

This processor uses a scoreboard to keep track of data hazards. The scoreboard consists of a FIFO that keeps track of the destination register of each instruction in the pipeline. The scoreboard also has search, insert, and remove methods.

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:

- Specify the value of the epoch register for each cycle.
- Fill out the pipeline diagram, specifying the instruction at each stage for each cycle.
- Denote killed 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>Epoch</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<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>and</td>
<td>NOP</td>
<td>add</td>
</tr>
<tr>
<td>EXE</td>
<td>add</td>
<td>slli</td>
<td>NOP</td>
<td>NOP</td>
<td>xor</td>
<td>bnez</td>
<td>NOP</td>
<td>NOP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>add</td>
<td>slli</td>
<td>NOP</td>
<td>NOP</td>
<td>xor</td>
<td>bnez</td>
<td>NOP</td>
<td>NOP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>add</td>
<td>slli</td>
<td>NOP</td>
<td>NOP</td>
<td>xor</td>
<td>bnez</td>
<td></td>
<td></td>
<td></td>
<td></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 5. Processor Pipeline Performance (14 points)

You are designing a four stage RISC-V processor (IF, DEC, EX, WB) that:

- Uses a BTB for next address prediction and resolves branches in the EX stage.
- Uses a scoreboard to 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.

The scoreboard for these processors ensures 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.

(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

Though not needed, you may find the following pipeline diagrams 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>slli</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>slli</td>
<td>blt</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>
The two processors have 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 6. 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):

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

and the following code runs in the Filter Process:

```c
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:

\[
\text{out} = (\text{out}+1) \mod 100; \quad \text{in process (circle one): } \quad \text{P} \quad \text{C} \quad \text{Neither}
\]

\[
\text{signal(}\text{charsA}\text{);} \quad \text{in process (circle one): } \quad \text{P} \quad \text{C} \quad \text{Neither}
\]

\[
\text{signal(}\text{indata}\text{);} \quad \text{in process (circle one): } \quad \text{P} \quad \text{C} \quad \text{Neither}
\]

\[
\text{wait(}\text{charsB}\text{);} \quad \text{in process (circle one): } \quad \text{P} \quad \text{C} \quad \text{Neither}
\]

(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): \quad \text{YES} \quad \text{NO}

Works with multiple Producers (circle one): \quad \text{YES} \quad \text{NO}

Works with multiple Filters (circle one): \quad \text{YES} \quad \text{NO}

END OF QUIZ 3!
Have a great summer!