NAME - Athena login name

Recitation section

WF 10, 34-301 (Kathy)  □ WF 12, 35-308 (Brian) □ WF 2, 34-303 (Quan)
WF 10, 13-4101 (Ileana) □ WF 12, 36-155 (Jason) □ WF 2, 13-5101 (Domenic)
WF 11, 34-301 (Kathy)  □ WF 1, 35-308 (Brian)   □ WF 3, 34-303 (Quan)
WF 11, 13-4101 (Ileana) □ WF 1, 36-155 (Jason) □ WF 3, 13-5101 (Domenic)

Please enter your name, Athena login name, and recitation section above. Enter your answers in the spaces provided below. Show your work for partial credit. You can use the extra white space and the backs of the pages for scratch work.

Problem 1. OS & Virtual Memory (16 points)

(A) (6 points) A RISC-V system with segmentation-based virtual memory is currently running two processes, A and B, with segment base and bound registers listed below:

Process A: base register = \( \text{0x1000} \)  bound register = \( \text{0x4000} \)
Process B: base register = \( \text{0x10000} \)  bound register = \( \text{0x40000} \)

The table below lists three virtual addresses. Fill the table by translating the addresses for processes A and B. In each cell, write either the physical address that corresponds to the virtual address, or write SEGFAULT if this virtual address is outside the process's address space (which would cause a segmentation fault, i.e., an out-of-bounds violation).

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

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Physical address for process A (or SEGFAULT)</th>
<th>Physical address for process B (or SEGFAULT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{0x500} )</td>
<td>0x1500</td>
<td>0x10500</td>
</tr>
<tr>
<td>( \text{0x3500} )</td>
<td>0x4500</td>
<td>0x13500</td>
</tr>
<tr>
<td>( \text{0x10000} )</td>
<td>Segfault</td>
<td>0x20000</td>
</tr>
</tbody>
</table>
(B) (10 points) Each of these instruction sequences experiences an interrupt or an exception that causes the processor to enter privileged mode (i.e., the operating system).

<table>
<thead>
<tr>
<th>Sequence A</th>
<th>Sequence B</th>
</tr>
</thead>
<tbody>
<tr>
<td>// address 0x0 not resident in memory lw x1, 0x0(x0)</td>
<td>li x1, 0x1234 li x2, 0x7 // this instruction is not in RV32I div x3, x1, x2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sequence C</th>
<th>Sequence D</th>
</tr>
</thead>
<tbody>
<tr>
<td>li x1, 0x10000 li x2, 0 // this loop gets // a timer interrupt loop: addi x2, x2, 1 bne x1, x2, loop</td>
<td>endless_loop: la a0, string // loads address li a7, 0x13 // print_string syscall number ecall j endless_loop string: .ascii &quot;Hello from Quiz 3!\n\0&quot;</td>
</tr>
</tbody>
</table>

Assume that the OS may only switch processes while servicing timer interrupts. Identify the instruction sequence(s) for which the statement applies. If no sequence applies, indicate NONE.

Which sequences...

(i) enter privileged mode: ___A, B, C, D___

(ii) save x1 through x31 and exception pc for the currently executing process: ___A, B, C, D___

(iii) handle an exception by emulating the instruction at the saved pc of the current process: ___B_________

(iv) handle an exception by loading a page from disk: ___A_________

(v) handle a system call: ___D_________

(vi) add 4 to the saved pc of the current process: ___B, D_________

(vii) subtract 4 from the saved pc of the current process: ___NONE_____

(viii) choose a new current process (which may be the same as the current process): ___C_________

(ix) load registers for the current process, then exit privileged mode and return to the saved pc of the current process: ___A, B, C, D___
Problem 2. Virtual Memory (16 Points)

Consider a RISC-V processor that includes a 38-bit virtual address, an MMU (memory management unit) that supports 4096 \(2^{12}\) bytes per page, \(2^{28}\) bytes of physical memory, and a large Flash memory that serves as a disk. The MMU and the page fault handler implement an LRU replacement strategy.

(A) (2 points) Assuming the page table includes the standard dirty and resident bits, specify the width of each page table entry in bits, and the number of entries in the page table.

Size of page table entry in bits: \(18\)

Number of entries in the page table: \(2^{26}\)

(B) (1 point) What is the maximum fraction of virtual memory that can be resident in physical memory at any given time?

Max fraction of virtual memory resident in physical memory: \(1/2^{10}\)

(C) (3 points) How do your answers to parts (A) and (B) change if the page size is doubled while all other parameters stay the same? 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

Size of page table entry in bits: \(d\)

Number of entries in the page table: \(e\)

Max fraction of virtual memory resident in physical memory: \(c\)
Going back to the original design with \(2^{12}\) bytes per page, the contents of the first 8 entries of the page table are shown below together with the contents of a 4 entry fully associative TLB (translation lookaside buffer) which uses an LRU replacement policy:

<table>
<thead>
<tr>
<th>VPN</th>
<th>V</th>
<th>D</th>
<th>PPN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>1</td>
<td>0</td>
<td>0x100</td>
</tr>
<tr>
<td>0x4</td>
<td>1</td>
<td>0</td>
<td>0x5</td>
</tr>
<tr>
<td>0x3</td>
<td>1</td>
<td>1</td>
<td>0x40</td>
</tr>
<tr>
<td>0x7</td>
<td>0</td>
<td>0</td>
<td>--</td>
</tr>
</tbody>
</table>

**TLB**

<table>
<thead>
<tr>
<th>VPN</th>
<th>V</th>
<th>D</th>
<th>PPN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x2</td>
<td>1</td>
<td>1</td>
<td>0x6</td>
</tr>
<tr>
<td>0x3</td>
<td>1</td>
<td>1</td>
<td>??</td>
</tr>
<tr>
<td>0x5</td>
<td>0</td>
<td>0</td>
<td>--</td>
</tr>
<tr>
<td>0x6</td>
<td>1</td>
<td>0</td>
<td>0x34</td>
</tr>
<tr>
<td>0x7</td>
<td>1</td>
<td>0</td>
<td>??</td>
</tr>
</tbody>
</table>

(D) (2 points) Note that two of the entries in the page table above are missing (labelled “??”). Below, specify what the missing values are if they can be inferred from the information available, or answer “Can’t Tell” otherwise.

**PPN for VPN 0x3 or Can’t Tell:** _______0x40_______

**PPN for VPN 0x7 or Can’t Tell:** ____Can’t Tell_____

(E) (8 points) For each of the following memory accesses, compute its corresponding physical address and indicate whether the access causes a TLB miss and/or a page fault. Assume each access is independent, i.e., each access starts with the TLB and Page Table state above.

<table>
<thead>
<tr>
<th>Virt Address</th>
<th>VPN</th>
<th>PPN</th>
<th>Phys Address</th>
<th>TLB Miss?</th>
<th>Page Fault</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xABC</td>
<td>0x0</td>
<td>0x100</td>
<td>0x100ABC</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>0x6004</td>
<td>0x6</td>
<td>0x34</td>
<td>0x34004</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>0x1234</td>
<td>0x1</td>
<td>0x6</td>
<td>0x6234</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Problem 3. Pipelined Processors (19 Points)

Consider the execution of the following code sequence on a 5-stage pipelined RISC-V processor, which is fully bypassed. The code below iterates over the array shown below to find the index of the element whose value is 0x104. Thus, on the second iteration of the loop, it branches to label done. The instruction unimp signals the end of the program.

This processor:
- Always speculates that the next instruction should come from address PC+4.
- Computes branch decisions and jump targets in the EXE stage.
- Annuls incorrectly fetched instructions resulting from branches and jump instructions.
- Receives the results of load instructions (from the data memory) in the WB stage.

. = 0x0
// x10 holds the value we are looking for
// x11 holds the address of A[0]
// x13 holds the address of the current data access
// Assume x10 = 0x104; x11 = 0x500; and x13 = 0x500 before executing the loop
loop:
    lw x14, 0(x13)
    beq x14, x10, done
    addi x13, x13, 4
    j loop
done:
    sub x13, x13, x11
    srli x13, x13, 2
    unimp

. = 0x500
// Array data
.word 0x0000100  // A[0]
.word 0x0000104  // A[1]
.word 0x0000108  // A[2]
.word 0x000010c  // A[3]
...

(A) (3 points) Fill in the pipeline diagram below showing the execution the first iteration of the loop, where the beq branch is not taken. Begin with the fetch of the lw instruction and end with the fetch of the lw instruction on the next iteration of the loop. Include arrows showing the use of bypass paths.

<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>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>lw</td>
<td>beq</td>
<td>addi</td>
<td>addi</td>
<td>addi</td>
<td>j</td>
<td>sub</td>
<td>srli</td>
<td>lw</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td>lw</td>
<td>beq</td>
<td>beq</td>
<td>beq</td>
<td>addi</td>
<td>j</td>
<td>sub</td>
<td>NOP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXE</td>
<td>lw</td>
<td>NOP</td>
<td>NOP</td>
<td>beq</td>
<td>addi</td>
<td>j</td>
<td>NOP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>lw</td>
<td>NOP</td>
<td>NOP</td>
<td>beq</td>
<td>addi</td>
<td>j</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>lw</td>
<td>NOP</td>
<td>NOP</td>
<td>beq</td>
<td>addi</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
(B) (1 point) Are any instructions stalled in the decode (DEC) stage during execution of this loop? If so, list the opcode(s) here. If not, enter NONE.

**Instructions stalled in DEC stage:** __________ beq __________

(C) (1 point) How many NOPs, if any, are inserted into the pipeline to handle the stalls required for execution of one iteration of the loop?

**Number of NOPs added to handle stalls:** __________ 2 __________

(D) (1 point) Are any instructions annulled (killed) during execution of this loop? If so, list the opcode(s) here. If not, enter NONE.

**Annulled instructions:** __________ sub, srli __________

(E) (1 point) How many NOPs, if any, are inserted into the pipeline to handle all the required annulments during execution of this loop?

**Number of NOPs added to handle annulments:** __________ 2 __________

(F) (2 points) How many cycles does one iteration of the loop take?

**Number of cycles for one iteration of the loop:** __________ 8 __________

(G) (3 points) Fill in the pipeline diagram below starting at the second (and final) iteration of the loop, where the beq branch is taken, through to the end of the program. Begin with the fetch of the lw instruction and end with the fetch of the unimp instruction. **Include arrows showing the use of bypass paths.**

<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>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>lw</td>
<td>beq</td>
<td>addi</td>
<td>addi</td>
<td>addi</td>
<td>j</td>
<td>sub</td>
<td>srli</td>
<td>unimp</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td>lw</td>
<td>beq</td>
<td>beq</td>
<td>beq</td>
<td>addi</td>
<td>NOP</td>
<td>sub</td>
<td>srli</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXE</td>
<td>lw</td>
<td>NOP</td>
<td>NOP</td>
<td>beq</td>
<td>NOP</td>
<td>NOP</td>
<td>sub</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>lw</td>
<td>NOP</td>
<td>NOP</td>
<td>beq</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>lw</td>
<td>NOP</td>
<td>NOP</td>
<td>beq</td>
<td>NOP</td>
<td>NOP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(H) (1 point) Are any instructions stalled in the decode (DEC) stage in this pipeline diagram? If so, list the opcode(s) here. If not, enter NONE.

**Instructions stalled in DEC stage:** __________ beq __________
(I) (1 point) How many NOPs, if any, are inserted into the pipeline to handle the stalls required during execution of the final iteration of the loop through the end of the program?

Number of NOPs added to handle stalls: __________ 2 __________

(J) (1 point) Are any instructions annulled (killed) during execution of the final iteration of the loop through to the end of the program? If so, list the opcode(s) here. If not, enter NONE.

Annulled instructions: __________ addi, j __________

(K) (1 point) How many NOPs, if any, are inserted into the pipeline to handle all the required annulments during the last iteration of the loop through to the end of the program?

Number of NOPs added to handle annulments: __________ 2 __________

(L) (1 point) Which additional instructions, if any, would need to be stalled if the processor did not have a bypass path from the EXE stage but still had one from the MEM and WB stages?

Instructions stalled due to removal of EXE bypass path: __________ srli __________

(M)(2 points) How many additional NOPs, if any, are required to handle the removal of the EXE bypass path?

Number of NOPs required to handle removal of EXE bypass path: __________ 1 __________
Problem 4. Processor Pipeline Performance (19 points)

You are designing a five stage (IF, DEC, EXE, MEM, WB) RISC-V processor with the same functionality in each stage that we have seen in lecture:

- IF: Initiate instruction fetch
- DEC: Decode instruction and gather source operands (stall if not available)
- EXE: Perform ALU operations and resolve branches
- MEM: Initiate data memory accesses
- WB: Write results back to register file

You may assume:

- Both instruction and data memories respond to every request in one cycle
- The processor predicts that all branches are \textbf{TAKEN} and can start fetching the instruction at the target address on the cycle \textbf{immediately following} the branch.

This processor will spend most of its time executing the following loop:

\begin{verbatim}
L1:  lw   t1, 0(a0)
     addi a0, a0, 4
     sub  t3, t1, a0
     blt  a0, a1, L1
\end{verbatim}

(A) (3 points) You start with no bypass paths. For each of the following pairs of stages, indicate whether we should consider adding a bypass path:

\begin{center}
\begin{tabular}{c|c|c|c|c}
& From MEM to EXE & YES & NO \\
\hline
\textbf{Accepted either answer since its ambiguous.} & From EXE to DEC & YES & NO \\
\hline
\textbf{From DEC to EXE} & YES & NO
\end{tabular}
\end{center}

We are concerned with average performance and the loop runs for many iterations so you should ignore the first and last iterations. For the following questions, assume the loop has already been running for a while and that the branch will be taken again. Note that the processor predicts the branch to be taken. You may use the pipeline diagrams below to help you in answering the following questions.

\begin{center}
\begin{tabular}{c|c|c|c|c|c|c|c|c|c|c|c}
\textbf{A: Cycle} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\hline
IF & lw & addi & sub & blt & blt & blt & lw \\
DEC & lw & addi & sub & sub & sub & blt & lw \\
EXE & lw & addi & NOP & NOP & sub & blt & lw \\
MEM & lw & addi & NOP & NOP & sub & blt & lw \\
WB & lw & addi & NOP & NOP & sub & blt & blt
\end{tabular}
\end{center}
Before looking at any of the options in (A), you decide to compare two other options:

**Processor A:** Bypassing only from WB to DEC.

**Processor B:** Bypassing from both WB and MEM to DEC.

(B) (4 points) How many cycles per loop iteration does the decode stage stall due to data hazards on each of the processors?

Processor A decode stall cycles per iteration: _______2_______

Processor B decode stall cycles per iteration: _______1_______

(C) (2 points) How many cycles does this loop take to execute on each of the processors?

Processor A cycles per iteration: _______6_______

Processor B cycles per iteration: _______5_______
A bypass path can be viewed as an extra piece of logic (Y) in the decode stage connected as shown below for processor A:

For processor B, an additional Y unit is needed to handle the second bypass:

Assume that you have ideal registers ($t_{PD} = 0$, $t_{Setup} = 0$), and the following propagation delays for each logic block:

<table>
<thead>
<tr>
<th></th>
<th>IF</th>
<th>DEC</th>
<th>Y</th>
<th>EXE</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_{PD}$</td>
<td>2.5 ns</td>
<td>3 ns</td>
<td>2 ns</td>
<td>5 ns</td>
<td>4 ns</td>
<td>4 ns</td>
</tr>
</tbody>
</table>

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

Clock period for processor A: ______ 6 _______

Clock period for processor B: ______ 8 _______

(E) (2 points) For the loop shown above, what is the average CPI (cycles per instruction) for the two processors:

Average CPI for processor A: ______6/4_______

Average CPI for processor B: ______5/4_______

(F) (4 points) For the loop shown above, what is the average number of instructions per nanosecond for each of the two processors:

Average number of instructions per nanosecond for processor A: _____1/9_______

Average number of instructions per nanosecond for processor B: _____1/10_______

Instr/nsec = 1/(cycles/instr) x cycle/nsec = (1/CPI) x (1/tCLK)
Problem 5. Synchronization (18 points)

You have been hired to work for a juice company. They have a large supply of blueberry and cranberry juice and would like to automatically produce 2-liter bottles of juice blend, which consist of exactly 1 liter of blueberry juice and 1 liter of cranberry juice.

Three processes running on a shared processor control the juice-bottling machinery. Process A controls the juice bottles and can place them under a shared juice spout, label them, cap them, remove them, and ship them. Processes B and C control the output of blueberry and cranberry juice, respectively. Processes B and C output juice through a single shared spout. For the machine to work, the following constraints must be obeyed:

- Juice should only be output through the spout while there is a bottle under the spout that isn’t full and does not have the cap on.
- Machines B and C cannot both output juice through the spout at the same time.
- Every bottle should have exactly 1 liter of blueberry juice and 1 liter of cranberry juice.
- However, machine A can label a bottle at the same time that machine B or C is outputting juice into it.

Below is some starter code written for the processes, with no synchronization.

<table>
<thead>
<tr>
<th>A:</th>
<th>B:</th>
<th>C:</th>
</tr>
</thead>
<tbody>
<tr>
<td>put bottle under spout put label on bottle put cap on bottle remove and ship bottle goto A</td>
<td>output 1 liter blueberry juice through spout goto B</td>
<td>output 1 liter cranberry juice through spout goto C</td>
</tr>
</tbody>
</table>

(A) (3 points) Suppose the three processes run the above code directly, with no synchronization primitives inserted. For each of the following failure scenarios, circle whether it is possible for the scenario to occur.

1. Machine A ships a bottle containing 2 liters of cranberry juice. (Possible / Not Possible)
2. Machine B tries to put juice into a bottle that already has a cap on it. (Possible / Not Possible)
3. Machine A tries to cap a bottle that already has a cap on it. (Possible / Not Possible)
(B) (15 points) Please insert the appropriate semaphores, WAITs, and SIGNALs into the machine code below to ensure correct operation and to prevent deadlock. Be sure to indicate initial values for any semaphores you use. For full credit, use no more than 4 semaphores, and don’t introduce unnecessary precedence constraints.

Shared Memory (specify your semaphores and initial values)

Semaphore B = 0, C = 0, spout = 1, done = 0;

<table>
<thead>
<tr>
<th>A:</th>
<th>B:</th>
<th>C:</th>
</tr>
</thead>
<tbody>
<tr>
<td>put bottle under spout</td>
<td>wait(B)</td>
<td>wait(C)</td>
</tr>
<tr>
<td>signal(B)</td>
<td>wait(spout)</td>
<td>wait(spout)</td>
</tr>
<tr>
<td>signal(C)</td>
<td>output 1 liter blueberry juice through spout</td>
<td>output 1 liter cranberry juice through spout</td>
</tr>
<tr>
<td>put label on bottle</td>
<td>signal(spout)</td>
<td>signal(spout)</td>
</tr>
<tr>
<td>wait(done)</td>
<td>signal(done)</td>
<td>signal(done)</td>
</tr>
<tr>
<td>wait(done)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>put cap on bottle</td>
<td></td>
<td></td>
</tr>
<tr>
<td>remove and ship bottle</td>
<td></td>
<td></td>
</tr>
<tr>
<td>goto A</td>
<td>goto B</td>
<td>goto C</td>
</tr>
</tbody>
</table>

For ease of reference, the list of constraints that must be satisfied is copied below:

- Juice should only be output through the spout while there is a bottle under the spout that isn’t full and does not have the cap on.
- Machines B and C cannot both output juice through the spout at the same time.
- Every bottle should have exactly 1 liter of blueberry juice and 1 liter of cranberry juice.
- However, machine A can label a bottle at the same time that machine B or C is outputting juice into it.

END OF QUIZ 3!