Computer System Architecture
6.823 Quiz #1
March 6th, 2020

Name: __________________________

This is a closed book, closed notes exam.
80 Minutes
16 Pages (+2 Scratch)

Notes:
• Not all questions are of equal difficulty, so look over the entire exam and budget your time carefully.
• Please carefully state any assumptions you make.
• Show your work to receive full credit.
• Please write your name on every page in the quiz.
• You must not discuss a quiz's contents with other students who have not yet taken the quiz.
• Pages 17 and 18 are scratch pages. Use them if you need more space to answer one of the questions, or for rough work.

Part A ________ 15 Points
Part B ________ 25 Points
Part C ________ 25 Points
Part D ________ 35 Points

TOTAL ________ 100 Points
Part A: Self-Modifying Code (15 Points)

In this part, we will use the EDSACjr instruction set from Handout 1.

Question 1 (8 points)

Consider the program shown below, which consists of a loop that runs once. The memory map shows the memory contents before the program starts. An array A is stored in memory in a contiguous manner, starting at location A. Memory locations N, I, COUNT, and ONE are shown with their initial values of n (=1024), 0, 0, and 1, respectively. The output of the program is stored in COUNT.

Memory:                                       Program:

<table>
<thead>
<tr>
<th>A:</th>
<th>Program:</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[0]</td>
<td>loop:</td>
</tr>
<tr>
<td>...</td>
<td>SUB N</td>
</tr>
<tr>
<td>A[n-1]</td>
<td>BGE done</td>
</tr>
<tr>
<td>...</td>
<td>CLEAR</td>
</tr>
<tr>
<td>ONE:</td>
<td>arrayRd:</td>
</tr>
<tr>
<td>1</td>
<td>ADD A</td>
</tr>
<tr>
<td>N: 1024</td>
<td>BGE cont</td>
</tr>
<tr>
<td>I: 0</td>
<td>LOAD COUNT</td>
</tr>
<tr>
<td>COUNT: 0</td>
<td>ADD ONE</td>
</tr>
<tr>
<td></td>
<td>STORE COUNT</td>
</tr>
<tr>
<td></td>
<td>cont:</td>
</tr>
<tr>
<td></td>
<td>LOAD arrayRd</td>
</tr>
<tr>
<td></td>
<td>ADD ONE</td>
</tr>
<tr>
<td></td>
<td>STORE arrayRd</td>
</tr>
<tr>
<td></td>
<td>LOAD I</td>
</tr>
<tr>
<td></td>
<td>ADD ONE</td>
</tr>
<tr>
<td></td>
<td>STORE I</td>
</tr>
<tr>
<td></td>
<td>BGE loop</td>
</tr>
<tr>
<td></td>
<td>done:</td>
</tr>
<tr>
<td></td>
<td>END</td>
</tr>
</tbody>
</table>

Write C-like pseudocode that shows the computation this program performs. Your code should operate on array A and the output in a variable named count.
Question 2 (7 points)

Alyssa P. Hacker points out that the loop can be written with fewer branches by using bit manipulation instructions. Note that, on EDSACjr, the most significant bit of each 16-bit integer is a sign bit, which is set to 1 if the integer is negative. Shift instructions perform arithmetic shifts.

Follow Alyssa’s suggestion and rewrite the loop with fewer branches. Part of the program is already written for you. If you need to, you can use additional memory locations for your own variables. You should label each variable and show its initial value in memory.

Memory:                                       Program:

<table>
<thead>
<tr>
<th>A:</th>
<th>loop:</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>LOAD I</td>
</tr>
<tr>
<td>A[0]</td>
<td>SUB N</td>
</tr>
<tr>
<td>A[1]</td>
<td>BGE done</td>
</tr>
<tr>
<td>...</td>
<td>CLEAR</td>
</tr>
</tbody>
</table>

| ONE: | LOAD I                |
| 1    | SUB ONE               |
| N:   | STORE I               |
| n    | I                     |
| I:   | BGE loop              |
| 0    | END                   |
| COUNT: | done:                |
| 0    |                      |

...
Part B: Cache Replacement Policy (25 Points)

Consider a 4-way set-associative cache that is accessed by the following sequence of memory addresses:

A, B, A, B, C, D, E, F, A, B

Each letter represents a unique address of a line, NOT a byte or a word. Assume that all these line addresses map to the same set of the cache. The set starts empty.

Question 1: Using LRU (4 points)

Suppose that the cache uses LRU replacement policy. Is each access a hit or a miss? Fill the table below, writing H for a hit and an M for a miss.

<table>
<thead>
<tr>
<th>Line Address</th>
<th>A</th>
<th>B</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>H / M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Question 2: Using SRRIP (4 points)

Suppose that the cache uses a 2-bit SRRIP replacement policy. Check out the SRRIP handout for more details. Is each access a hit or a miss? Fill the table below, writing H for a hit and an M for a miss.

<table>
<thead>
<tr>
<th>Line Address</th>
<th>A</th>
<th>B</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>H / M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Question 3: Comparing SRRIP and LRU (6 points)**

Is SRRIP always better than LRU? If it is, state why. If it is not, construct a sequence of line addresses on which LRU outperforms SRRIP on the 4-way cache from Questions 1 and 2.

**Question 4: 2-bit SRRIP scan sequence length (6 points)**

Consider another sequence of line addresses. All other conditions do not change.

\[ A_1, A_2, A_1, A_2, B_1, B_2, B_3, \ldots B_n, A_1, A_2 \]

If the cache uses 2-bit SRRIP, and we want to guarantee that the last two accesses (to \( A_1 \) and \( A_2 \)) both hit in the cache, what is the maximum possible value of \( n \)?

**Question 5: M-bit SRRIP scan sequence length (5 points)**

Consider an M-bit (instead of 2-bit) SRRIP. Other conditions remain the same as in Question 4. What is the maximum possible value of \( n \) for which the last two accesses are hits, expressed as a function of \( M \)?
Part C: Segmented Page Table (25 Points)

Though hierarchical page tables are widely adopted now, they are not the only option. Back in 1960s, Multics, a time-sharing operating system, proposed using segmentation to address the efficiency issues of a linear page table. The observation behind this idea is that, for a typical program, the pages in its virtual address space are clustered as shown in the following figure:

![Virtual address space of a typical program]

Typical programs use both low and high virtual addresses, with code and heap located in low addresses and the stack in high addresses (this lets the stack and heap grow as needed). However, as shown in the figure, the code, heap, and stack each consists of a contiguous chunk of pages. This clustering can be exploited to reduce the amount of memory allocated for a flat page table.

Specifically, instead of having a single page table for a program, a machine can divide a program’s virtual address space into 3 segments: code, stack, and heap, based on the highest two bits of the virtual address. Each segment has its own page table, sized to track only the translations of its corresponding segment.

Each segment page table is allocated in a consecutive chunk of physical memory. The PTEs are stored consecutively and are directly indexed by the virtual page number. The base address and the size (in bytes) of the page table are stored in six registers. The base address is the physical address of the first PTE in the page table.
Ben Bitdiddle has designed a machine based on this idea. The machine uses 26-bit virtual addresses, 32-bit physical addresses, 64KB pages, and 32-bit PTEs. All PTEs in a page table are valid. Please read their formats below carefully. Assume there is no TLB.

### Virtual address:

<table>
<thead>
<tr>
<th>Segment Number</th>
<th>Virtual page number</th>
<th>Page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>25 24 23 16 15 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Physical address:

<table>
<thead>
<tr>
<th>Physical page number</th>
<th>Page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 16 15 0</td>
<td></td>
</tr>
</tbody>
</table>

### PTE:

<table>
<thead>
<tr>
<th>PPN/DPN</th>
<th>Irrelevant status bits</th>
<th>Resident bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 16 15 1 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Below is a snapshot of the six page-table-related registers. On the right is a snapshot of the physical memory.

For your convenience, a colon (“:”) separates the lower 16 bits from the upper bits of each address and data value.

### Snapshot of page table base and size registers

<table>
<thead>
<tr>
<th>Segment Number (in binary)</th>
<th>Segment</th>
<th>Page table Base</th>
<th>Page table Size (in bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Unused</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>Code</td>
<td>0x100:0000</td>
<td>0x40</td>
</tr>
<tr>
<td>10</td>
<td>Heap</td>
<td>0x100:0404</td>
<td>0x40</td>
</tr>
<tr>
<td>11</td>
<td>Stack</td>
<td>0x104:6134</td>
<td>0x40</td>
</tr>
</tbody>
</table>

### Physical memory snapshot

- 0x201:91A0 0xD:2E1C
- 0x201:919C 0x3:AA00
- 0x201:9198 0x6:010C
- 0x201:9194 0xA:74CC
- 0x201:9190 0x7:30B1
- ...还有很多其他地址和数据值...
**Question 1: Address translation (15 points)**

The table below lists four virtual addresses. Assume the program issues an access with each of these addresses. For each access, answer the following questions:

A) Does the access trigger a segmentation fault, i.e., use an illegal memory location?

B) Does the access trigger a page fault?

C) What is the result of the address translation? This can be physical (byte) address in the absence of faults, a disk page number in the case of a page fault, or “None” in all other cases.

<table>
<thead>
<tr>
<th>Virtual address</th>
<th>A) Segmentation fault? (Yes/No)</th>
<th>B) Page fault? (Yes/No)</th>
<th>C) Translation result (specific PA or DPN, or “None”)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x104:6140</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x201:9190</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x1FF:AC5D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x000:5123</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Question 2: Segmented vs. flat page table (4 points)**

Compared to a single flat page table, does this segmented page table design:

A) Reduce the amount of allocated memory for page tables?

B) Reduce the number of PTEs accessed by a specific program?

**Question 3: Segmented vs. hierarchical page table (6 points)**

Does each of the following programs prefer a segmented or a hierarchical page table? Why?

A) A program that uses almost all the virtual address space as its heap?

B) A program that frequently accesses a small chunk of virtual memory, on a system with no TLB.
Part D: Instruction Pipelining (35 Points)

Throughout this part of the quiz, the questions will refer to executing the computation shown in the following C-like pseudocode segment, which computes the product of 4096 integers held in an array:

```c
int A[4096];
...
int product = 1;
for (int i = 0; i < 4096; i++) {
    product = product * A[i];
}
...
```

We will consider performing this computation using the following MIPS code segment:

```
0x0FC:  ... 
0x100:  ADDI r1, r0, 4096
0x104:  ADDI r2, r0, 1
0x108:  loop: LW r4, 0(r3)
0x10C:  MUL r2, r2, r4
0x110:  ADDI r3, r3, 4
0x114:  ADDI r1, r1, -1
0x118:  BNEZ r1, loop
0x11C:  SUB ...
```

(Refer to the handout on the MIPS ISA.) Assume register r3 initially points to the first element of array A. Register r2 is used to hold the product of the array elements.

**Note:** Some of the questions use pipeline diagrams, and in others, you may find it helpful to draw them. *Page 17 includes additional pipeline diagrams in case you need them.*
**Question 1: Pipelined Princeton architecture (8 points)**

Consider the 2-stage pipelined Princeton architecture implementation discussed in lecture, where instruction fetches and data accesses both use one single-port unified memory. This memory can perform only one access on each cycle.

(a) Fill in the following resource-use diagram to show the steady-state execution of the loop on the pipelined architecture. You must show instructions from at least one full loop iteration. (5 points)

(b) During the loop’s steady-state execution, what is the average number of clock cycles per loop iteration? (3 points)
**Question 2: Using interleaved memory (15 points)**

Ben Bitdiddle is unhappy with the frequency of stalls, so he decides to improve the Princeton architecture by replacing the single-port unified memory with an interleaved memory. (Described in the “Interleaved Memory” handout.)

(a) Help Ben complete his datapath by drawing wires to connect the three unconnected address and data inputs of the interleaved memory below. (4 points)

(b) What type of hazard is a bank conflict an example of: a structural, control, or data hazard? (2 points)
(c) Consider again the code from page 10. During the steady-state execution of the loop, what is the average number of bank conflicts that would occur per iteration if we failed to stall conflicting instructions? Does it depend upon the initial value of r3, the address of the first element of the array in memory? (3 points)

(d) If we stall to deal with bank conflicts in the memory, what is the average number of clock cycles needed per iteration? (2 points)

(e) In lecture, we derived the following stall signal for the 2-staged pipelined Princeton architecture shown in Question 1.

\[
\text{Stall} = (\text{OpCode is LW, SW, or jump}) \\
+ ((\text{OpCode is branch}) \cdot (\text{branch is taken}))
\]

If we now implement stalling to deal with bank conflicts in Ben’s new machine, what will be the new stall signal? You may write an expression for the new signal in terms of the old stall signal together with other values available in Ben’s datapath from Question 2 part (a). (4 points)

\[
\text{NewStall} = \text{______________________________}
\]
**Question 3: Unrolling the loop (5 points)**

Alyssa P. Hacker suggests that Ben can reduce bank conflicts by carefully controlling data layout in memory and rewriting the MIPS code as follows:

0x0FC: ...  
0x100: ADDI r1, r0, 4096  
0x104: ADDI r2, r0, 1  
0x108: loop: LW r6, 0(r3)  
0x10C: LW r7, 4(r3)  
0x110: MUL r2, r2, r6  
0x114: MUL r2, r2, r7  
0x118: ADDI r3, r3, 8  
0x11C: ADDI r1, r1, -2  
0x11E: BNEZ r1, loop  
0x124: SUB ...  

During the loop’s steady-state execution, what is the average number of clock cycles needed per element of the array that is accessed? Does it depend upon the initial value of r3, the address of the first element of the array in memory?
Question 4: Five-stage pipeline (7 points)

To increase clock frequency, we divide the datapath into the five standard pipeline stages. As in Ben’s design, we still use a Princeton architecture design with a single memory. The pipeline is shown below in a stylized fashion, with the same memory drawn once in the I-Fetch stage and once in the Memory stage. Both stages use the same two-bank interleaved memory described in the handout, but each stage is connected to a different port of that memory. Assume the pipeline is fully bypassed as shown in lecture, and that branches are resolved in the decode stage.

Consider again the first MIPS code segment, which performs one load per iteration:

```
0x0FC: ...  
0x100: ADDI r1, r0, 4096  
0x104: ADDI r2, r0, 1  
0x108: loop: LW r4, 0(r3)  
0x10C: MUL r2, r2, r4  
0x110: ADDI r3, r3, 4  
0x114: ADDI r1, r1, -1  
0x118: BNEZ r1, loop  
0x11C: SUB ...  
```

(a) Assuming we stall for bank conflicts, what is the average number of clock cycles needed per iteration during the loop’s steady-state execution? Does it depend upon the initial value of r3, the address of the first element of the array in memory? (4 points)
(b) Reorder the instructions in the code segment to reduce stalls without changing the functionality of the code, while still performing only one load per iteration. Indicate how many fewer stalls per iteration your revised loop incurs on the 5-stage pipeline. (3 points)
**Space for pipeline diagrams**

We will not grade anything from this page unless you 1) clearly label a section of this page as answering a particular question AND 2) you put an indication near the question that your answer can be found on this page.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>0</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>
</tr>
</thead>
<tbody>
<tr>
<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></td>
</tr>
</tbody>
</table>
Scratch space

Use these extra pages if you run out of space or for your own personal notes. We will not grade this unless you tell us explicitly in the earlier pages.