CS 433 Final Exam – May 14, 2014

Professor Sarita Adve

Time: 3 Hours

Please print your name and NetID and circle the appropriate category in the space provided below.

<table>
<thead>
<tr>
<th>Name</th>
<th>NetID</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Undergraduate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Graduate</td>
</tr>
</tbody>
</table>

Instructions

1. No books, papers, notes, or any other typed or written materials are allowed. No calculators or other electronic materials are allowed.

2. Please do not turn in loose scrap paper. Limit your answers to the space provided if possible. If this is not possible, please write on the back of the same sheet. You may use the back of each sheet for scratch work.

3. In all cases, show your work. No credit will be given if there is no indication of how the answer was derived. Partial credit will be given even if your final solution is incorrect, provided you show the intermediate steps in reaching the final solution.

4. If you believe a problem is incorrectly or incompletely specified, make a reasonable assumption and solve the problem. The assumption should not result in a trivial solution. In all cases, clearly state any assumptions that you make in your answers.

5. This exam has 7 problems and 14 pages (including this one). All students should solve problems 1 through 6. Only graduate students should solve problem 7. Please budget your time appropriately. Good luck!

<table>
<thead>
<tr>
<th>Problem</th>
<th>Maximum Points</th>
<th>Received Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8 Graduate Problem</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>Undergraduates (56 points), Graduates (64 points)</td>
<td></td>
</tr>
</tbody>
</table>
Problem 1 [10 points]:
Consider a computer with a memory system with 16 bits for physical address, 32 bits for virtual address, page size of 4KB, 16 bit words, and word (16 bit) addressable memory. The computer system contains a 32KB data cache that is virtually indexed and physically tagged. The data cache is 8-way set associative and the cache line size is 16 words.

Part A [2 points]:
Specify what bit ranges of the address (virtual or physical) comprise the virtual page number and the physical page frame number. How many physical and virtual pages does this system have?

Solution:
11 bits are offset within a page, so bits 11:31 of the virtual address are the virtual page number, and bits 11:15 of a physical address are the physical page number. There are $2^{21}$ virtual pages, and $2^5$ physical pages.

Grading:
0.5 point for each of the following: the bit ranges of the page number in the virtual address, the bit ranges of the page number in the physical address, the number of physical pages, and the number of virtual pages.

Part B [2 points]:
Assume each Page Table Entry (PTE) has 5 state bits (dirty bit, protection bits, etc.) in addition to the address translation. How many bits does each PTE consume? How much memory, in bytes, does the bookkeeping for all the virtual pages consume? Assume only one level of page translation (as discussed in class) and assume each page table entry is word aligned.

Solution:
A PTE has 5 bits of the physical page number in addition to the 5 state bits. This makes a total of 10 bits which fits in one 16 bit word. Each mapping therefore takes one word. There are $2^{21}$ mappings. Therefore, the page tables take up 4MB.

Grading:
1 point for the size in bits of a PTE. 0.5 point for the total number of PTEs. 0.5 point for rounding up the PTE size to a multiple of word size and for expressing the final answer as the product of the number and size of PTEs.

Part C [3 points]:
Specify what bit ranges within the virtual or physical address (whichever is appropriate) comprise the cache index and cache tag bits. State explicitly whether the bits used are from the physical or virtual address.

Solutions:
There are 32KB / 2 bytes per word / 16 words per line / 8 lines per set = $2^7$ sets. The index needs 7 bits. The cache is virtually indexed, so the bits are taken from the virtual address. The offset within a cache line is 4 bits, so bits 4:10 are used for the index. However, the page size is $2^{11}$ words, so these bits will be the same in the physical address as well. The tag is the remaining bits of the physical address, bits 12:15.

Grading:
Part D [3 points]
If the architect were to decrease the cache associativity to 4-way, what would the cache index bits be then? Would this create a problem for the Virtually Indexed/Physically Tagged caching scheme? If so, describe the problem and a hardware-only way of solving it which does not require waiting for address translation.

Solution:
If the cache were only 4-way associative, there would be 256 sets. Then the cache index would be 8 bits long, and use bits 4:11 of an address. If these bits are taken from the virtual address, bit 11 is part of the virtual page number. If two virtual pages map to the same physical page, then two equivalent virtual addresses could have different index bits. One solution is to ignore bit 11 of the virtual address, and look for the data in both of the possible ways.

Grading:
½ point for giving the range of index bits. 1 point for noticing this makes aliasing problematic. 1.5 points for a correct hardware only solution. (software-only solutions like prohibiting aliasing are not accepted, nor doing address translation before indexing).

Problem 2 [5 Points]
Consider a set associative cache with the following parameters: the miss rate is 0.05, the cycle time is 1ns, a hit in the cache takes 1 cycle, and a miss takes an additional 99 cycles (total of 100 cycles for a miss).

Now assume that we employ way prediction to the same cache with the following parameters. The cycle time is 0.95 ns. For accesses that eventually hit in the cache, the correct way prediction rate is 90%. For an access that hits with the way correctly predicted, it still takes 1 cycle. For accesses that eventually hit in the cache but with the initial way mispredicted, it takes an additional one cycle. An access that misses in the cache still takes an additional 99 cycles after all the cache accesses.

What is the AMAT (average memory access time) of the original cache without way prediction and of the cache with way prediction measured in absolute time in ns? Assume a simple, single-issue, 5-stage pipeline, in-order processor that blocks on every read and write until it completes.

Solution:
AMAT = hit time + miss rate * miss penalty

Original system:
AMAT₁ = 1 cycle * 1ns + 0.05 * 99 cycles * 1ns = 5.95 ns

Way Prediction:
AMAT₂ = (hit time with correct way prediction*correct way prediction accuracy) + (hit time with incorrect way prediction*(1 - correct way prediction accuracy)) + (miss rate * miss penalty)

AMAT₂ = (1 cycle * 0.95ns * 0.90) + (2 cycles * 0.95ns * 0.10) + (0.05 * 99 cycles *0.95ns) = 0.855 ns + 0.19 ns + 4.70ns = 5.74ns
Grading scheme:
1 point for knowing AMAT equation. 1 point for correctly setting up the AMAT equation for the original system and 3 points for correctly setting up the AMAT equation for the system with way prediction. For the latter, 1 point partial credit for each of the three components of the equation above.

Problem 3 [14 points]:

Consider the following code which sums the elements of a product of two matrices:

```c
int i, j, k; // i, j, k are allocated in processor registers
float sum; // sum is allocated in a processor register
float a[8][8], b[8][8];

for (i = 0; i < 8; i++) { // (1)
    for (j = 0; j < 8; j++) { // (2)
        for (k = 0; k < 8; k++) { // (3)
            sum += a[i][k] * b[k][j]; // (4)
        }
    }
}
```

Assume the following:
- There is a perfect instruction cache; i.e., do not worry about the time for instruction accesses.
- Both int and float are of size 4 bytes.
- Assume that only the accesses to the arrays a and b generate accesses to the data cache. The rest of the variables are all allocated in registers.
- Assume a fully associative, LRU data cache with 8 lines, where each line is 32 bytes.
- Initially, the data cache is empty.
- The arrays a and b are stored in row major form.
- To keep things simple, assume that statements in the above code are executed sequentially. Statements (1), (2), and (3) take 10 cycles for each invocation. Statement (4) takes 10 cycles plus an additional 20 cycles per data cache miss to wait for the data. That is, if both array accesses in statement (4) miss, then it takes a total of 50 cycles.
- Assume that the arrays a and b both start at cache line boundaries.

Part A [3 points]:
How many accesses to arrays a and b will result in cache misses? Explain your answer.

Solution:
A full row of each array fits in one cache line. There are 8 total lines in the cache. Each access to a within the outermost loop is to the same cache line, so it is never evicted from the LRU cache until the next iteration of the outermost loop. For b, however, the accesses iterate through the 8 rows, and each line is evicted from the cache before it is accessed again. Thus, there are 8 misses for a and 512 misses (8 * 8 * 8) for b. This gives a total of 520 misses.

Grading:
1.5 points for calculating and explaining the misses for each array correctly.
Part B [8 points]:
Now assume there is a data prefetch instruction with the format `prefetch(array[index1][index2])`. This prefetches the entire cache line containing the word `array[index1][index2]` into the data cache. It takes 1 cycle for the processor to execute this instruction and send it to the data cache. The processor can then go ahead and execute subsequent instructions. If the prefetched data is not in the cache, it takes 20 cycles for the data to get loaded into the cache.

Add prefetch instructions to the code so as to minimize the execution time. Do not transform the code in any other way. How many cache misses for each of arrays a and b will occur at line (4) in your modified code?

Solution:
```java
for (i = 0; i < 8; i++) {    // (1)
    prefetch(a[i][0]);      // p1
    prefetch(b[0][0]);      // p2
    for (j = 0; j < 8; j++) {    // (2)
        for (k = 0; k < 8; k++) {   // (3)
            prefetch(b[k+1][0]);   // p3
            sum += a[i][k] * b[k][j];  // (4)
        }
    }
    prefetch(b[0][0]);   // p4
}
```
For each iteration of the outermost loop, we can insert one prefetch to avoid the compulsory miss for array a (line p1). Every access to b would be a miss, so we need to insert a prefetch for each of these accesses. In this solution, we prefetch the first row of b before each invocation of the innermost loop (lines p2 and p4). For subsequent rows of b, we insert a prefetch in the innermost loop for the next iteration (line p3). The execution of lines (4) and (3) provide the 20 necessary cycles to transfer the data to the cache in time in response to p3.

For the above code, there are no cache misses at line 4.

Grading:
2 points for eliminating all misses for a. 2 points for eliminating the miss for b[0][0]. 2 points for eliminating the other misses for b. 2 points for giving the correct number of cache misses for the transformed code.

Part C [3 points]:
Now assume that the prefetch instruction is no longer available. Transform the code above so that it reduces the total number of cache misses for the data reads in part A to less than one fifth the original. You may not add any additional memory accesses or other variables to the original code. How many total read misses does your new code have?

Solution:
Interchanging the order of the j and k for loops reduces the total read misses. With the new code, a still misses 8 times but b now only misses 64 times, for a total of 72 misses.
Grading:
2 points for effectively transforming the code. 1 point for correctly calculating the misses. Alternative techniques will also be considered and graded based on their effectiveness.

Problem 4 [8 points]:

Consider the following snooping update-based protocol for a bus-based shared-memory system, called Protocol X. The protocol has three states – Valid-Exclusive (V-E), Dirty (D), and Shared (S). (Technically, there is also an Invalid state for an empty block, but since the protocol never sends an invalidation, this may be ignored.)

The V-E state implies that this is the only cache to hold a copy of the block and that it is clean; i.e., main memory has an up to date copy of the block (for this problem, you do not have to worry about the mechanism used to determine if this cache has the only copy). V-E transitions to Shared when another processor performs a read. Shared implies that one or more caches contain a copy, and that all copies are clean. The Dirty state is analogous to the Modified state in the MSI invalidate protocol studied in class, in that this is the only copy cached, and that main memory is out of date. If memory has a clean copy of a line, then it will service any request for that line.

Complete the following table, filling in the state transitions specifically for Processor 1’s cache and address A. Assume writeback caches with only one entry which begins empty. Include the new state of address A in Processor 1’s cache for the MSI protocol studied in class and protocol X after each event shown, and note any bus traffic generated by Processor 1’s cache for MSI and X (distinguish between writing a full line or just a word). Local access on A implies the access is initiated by processor 1 and bus access implies the access is initiated by another processor on the bus. Assume that these accesses occur in the order below, and that no other memory accesses / traffic occur. The first row is filled out.

<table>
<thead>
<tr>
<th>Event</th>
<th>MSI state for block A</th>
<th>Protocol X state for block A</th>
<th>External actions by Processor 1 for MSI (say none if no action)</th>
<th>External actions by Processor 1 for X (say none if no action)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local read A</td>
<td>S</td>
<td>V-E</td>
<td>Read A on Bus</td>
<td>Read A on Bus</td>
</tr>
<tr>
<td>Local write A</td>
<td>M</td>
<td>D</td>
<td>Invalidate</td>
<td>None</td>
</tr>
<tr>
<td>Bus read A</td>
<td>S</td>
<td>S</td>
<td>Send line</td>
<td>Send word</td>
</tr>
<tr>
<td>Bus write A</td>
<td>I</td>
<td>S</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Local read A</td>
<td>S</td>
<td>S</td>
<td>Read A on Bus</td>
<td>None</td>
</tr>
<tr>
<td>Local write A</td>
<td>M</td>
<td>S</td>
<td>Invalidate</td>
<td>Update</td>
</tr>
<tr>
<td>Bus read A</td>
<td>S</td>
<td>S</td>
<td>Send line</td>
<td>None</td>
</tr>
<tr>
<td>Local write A</td>
<td>M</td>
<td>S</td>
<td>Invalidate</td>
<td>Update</td>
</tr>
<tr>
<td>Local write A</td>
<td>M</td>
<td>S</td>
<td>None</td>
<td>Update</td>
</tr>
</tbody>
</table>

Grading: 8 points. ¼ point each entry. No penalty for cascading errors.
Problem 5 [10 points]:

You are to implement a queue using an array in a multiprocessor system. The elements of the array can be accessed in parallel by multiple processors. You are to write two functions:

- **enqueue**, which will add an element to the tail of the queue, and
- **dequeue**, which will remove an element from the head of the queue.

Assume the queue always has at least one element; i.e., a **dequeue** is never called on an empty queue. Also assume that the queue never gets full; i.e., the array is infinitely long and you don’t have to worry about calling an **enqueue** on a full queue.

**Part A [6 points]:**
Write the **enqueue** and **dequeue** functions using an atomic test&set instruction to achieve synchronization. Don’t worry about using test&test&set, but otherwise, write the most efficient code possible. Add C or C++ like pseudo-code to the stub below. **Assume local-index below is local to each processor** and is not part of shared memory; i.e., each processor has its own copy of this variable (e.g., in its own register). Assume the system implements sequential consistency.

```c
int head;   /* index for the head of the queue */
int tail;    /* index for the tail of the queue */
int local-index;   /* current index for enqueuing or dequeuing, each processor has its own copy */

/* Assume the queue is never full and always has at least one element */

enqueue(item) {

    queue[index] = item;

}

dequeue() {

    item = queue[index];

    return item

}

Solution:
enqueue(item) {

    while (test&set(qlock)); /* Spin until lock is obtained */
    local-index = tail; /* index for tail of queue */
```
```c
    tail++;     
    qlock = 0; /* unlock */       
    queue[local-index] = item;
}

dequeue() {
    while (test&set(dqlock)); /* Spin until lock is obtained */
    local-index = head;
    head++;
    dqlock = 0; /* unlock */
    item = queue[local-index];
    return item;
}

Grading:
2 points for a correct queue function. 2 points for a correct dequeue function. 1 additional point for realizing that different locks can be used for these functions. 1 additional point for updating the queue outside the critical section in both the queue and dequeue functions. 6 total points.

Part B [4 points]:
Repeat part (A) using the fetch&increment instruction instead of the test&set for synchronization.

```c
int head;         /* index for the head of the queue */
int tail;         /* index for the tail of the queue */
int local-index;  /* current index for enqueuing or dequeuing; each processor has its own copy */

/* Assume the queue is never full and always has at least one element */

enqueue(item) {

    queue[local-index] = item;

}

dequeue() {

    item = queue[local-index];

    return item;

}

Solution:

enqueue(item) {
    index = fetch&increment(tail);
    queue[local-index] = item;
```
return;
}

dequeue() {
index = fetch&increment(head);
item = queue[local-index];
return item;
}

**Grading:**
2 points for each function. A solution that implements test&set using fetch&increment gets graded out of 2 points.

**Problem 6 [9 points]:**

Systems with a relaxed consistency model often offer a *memory fence* instruction to allow the programmer to enforce correct behavior. A full fence prevents any reordering of memory operations, requiring all previous (by program order) reads and writes to complete before subsequent (by program order) accesses begin.

Consider a machine with a very relaxed memory model, where any program ordered pair of accesses to different variables may be reordered. Program order between accesses to the same variable, however, must be enforced. Consider the code segments below – the left segment executes on processor 0 while the right segment executes on processor 1.

All variables are initially 0.

**Processor 0**
- \( A = 1; \)
- \( B = 1; \)
- \( \text{Flag} = 1; \)
- while (Flag == 1) {}  \( \text{reg1} = B; \)
- \( \text{reg2} = C; \)

**Processor 1**
- while (Flag == 0) {}  \( \text{reg3} = A; \)
- \( \text{reg4} = B; \)
- \( C = 1; \)
- Flag = 0;

**Part A [3 points]:**
What are the different combinations of values that the four reads on A, B, and C can return (i.e., write the possible combinations of final values of reg1, reg2, reg3, and reg4)?

**Solution:**
reg 1’s final value is always 1 because Processor 0’s write of B must occur before its read of B and there is no other write to B. reg2, reg3, and reg4 could have either 0 or 1 as their final values since there is no ordering constraint between the writes of A, B, C and the reads for those register values.

**Grading:**
1 point for recognizing that reg 1 should always return 1. 2 points for recognizing that without fences, the relaxed machine could return 0 or 1 for the other registers.
Part B [2 points]:
Which of the results identified in part A are possible under Sequential Consistency?

Solution:
With sequential consistency, the only allowed result is: reg1=reg2=reg3=reg4=1.

Grading:
2 points for the correct sequentially consistent result.

Part C [4 points]:
The code is repeated below. Insert the minimal number of fence instructions in this code to limit the results on our very relaxed machine to only those possible under Sequential Consistency.

All variables are initially 0.

<table>
<thead>
<tr>
<th>Processor 0</th>
<th>Processor 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A = 1;</td>
<td>while (Flag == 0) {;}</td>
</tr>
<tr>
<td>B = 1;</td>
<td>reg3 = A;</td>
</tr>
<tr>
<td>Flag = 1;</td>
<td>reg4 = B;</td>
</tr>
<tr>
<td>while (Flag == 1) {;}</td>
<td>C = 1;</td>
</tr>
<tr>
<td>reg1 = B;</td>
<td>Flag = 0;</td>
</tr>
<tr>
<td>reg2 = C;</td>
<td></td>
</tr>
</tbody>
</table>

Solution:
To achieve this result on our relaxed machine, we need a fence before each write to flag and after each read of flag since these are the synchronizing or racing instructions.

Grading:
1 point each for correct placement of each fence instruction. -1/2 for each extra fence (except that a fence before a read to Flag or after a write to Flag are ok since we discussed in class that putting fences around synchronization or races gave SC). 4 points maximum, 0 points minimum.
ONLY GRADUATE STUDENTS SHOULD SOLVE THIS PROBLEM

Problem 7 [8 points]:

A ticket lock is yet another scheme for multiprocessor synchronization. The ticket lock is modeled after the way customers are often served in a deli, where the customers all take a uniquely numbered ticket as they arrive, and there is a display showing the “now serving number.” When your number appears, you are served.

In a multiprocessor system, a ticket lock is implemented with two counters - one for the “take a unique number” counter and one for the “now serving” counter. As each processor arrives at the lock point, it reads the first counter to get its unique number and increments the counter. It then waits (spins) until its number appears on the 2nd counter. It then proceeds into its critical section. At the end of the critical section, it increments the 2nd counter so another processor can proceed.

Assume the system implements a fetch&increment instruction as discussed in class. Assume a sequentially consistent memory model.

Part A [2 points]:
Would you use the fetch&increment instruction to implement the first counter, “take a unique number,” or would you use ordinary loads/stores? Explain your answer.

Solution: The fetch&increment instruction is best. This location must return unique values and is updated by every processor, so simply using one load and one store will not work.

Grading: No partial credit.

Part B [2 points]:
Would you use the fetch&increment instruction to implement the second counter, the “now serving number,” or would you use ordinary loads/stores? Explain your answer.

Solution: This location is only written to by the processor in the critical section, and in the critical section the value of the second counter is the value the lock holder saw when it read the “take a number” counter. The lock holder can use a simple store to increment the “now serving” counter.

Grading: No partial credit.

Part C [4 points]:
For the system described, write assembly code to implement the ticket lock sequence of taking a unique number, waiting for your turn, and updating the counters as needed. Do not worry about the initial value of either counter or about overflow. Indicate where the critical section is executed with a comment line such as /*Critical Section*/. Use names such as “Count1” for the first counter and “Count2” for the 2nd counter - make sure it is obvious what you are referring to. For the fetch&increment instruction, use the format “FETCH&INC Rn, ADDR” where ADDR is the memory address that is fetched and incremented and Rn is the register that stores the result of the fetch&increment.
Solution:

FETCH&INC R1, Count1  /* take your number and increment */
WAIT:  LD  R2, Count2  /* poll now serving counter */
       SUB R2, R2, R1  /* test for match, spin if not equal */
       BNEQZ R2, WAIT

       /* Critical Section */

       ADD R1, R1, #1  /* increment now serving count */
       ST R1, Count2  /* and update the memory location */

Grading:
1 point for proper atomic Fetch&Inc to get the ticket number – no points if not Fetch&Inc
1 point for proper poll and spin on compare to second counter
1 point for proper location of critical section
1 point for proper update of Count2 – ½ point if answered part B as yes and did a proper atomic update here (if implementation is consistent with answer of Part B)