CS 433 Final Exam: Dec 13, 2022

Professor Sarita Adve

Time: 3 Hours

Please print your Name and NetID and circle your course section below.

Name:		
NetID:		
Section:	T3 (Undergraduate)	T4 (Graduate)

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 **6 problems** and **15 pages** (including this one). **All students** should solve **problems 1, 2A-E, 3 through 6**. Only **graduate students should solve problem 2F-G**. Please budget your time appropriately. Good luck!

Problem	**1**	**2**	**3**	**4**	**5**	**6**	**Total**
Points T3	13	15	14	6	8	5	61 (undergrads)
T4	13	21	14	6	8	5	67 (graduates)
Score							

Problem 1 [13 points]

Consider a virtual memory system with the following parameters.

- 64-bit virtual addresses
- 48-bit physical addresses
- 4KB pages
- 16B cache blocks
- Byte-addressing

Furthermore, main memory is interleaved on a word (32-bit) basis with four banks and a new bank access can be started every cycle. It takes 10 processor clock cycles to send an address from the processor to main memory; 50 cycles for memory to access a word; and 20 cycles to send one word of data back from memory to the processor. The memory bus width is 1 word.

Part A [3 points]

How many bits are needed for the page offset, virtual page number, and physical page number?

Solution:

2^12 byte page size implies 12 bits for the page offset.

64 – 12 = 52 bits for the virtual page number, 48 – 12 = 36 bits for the physical page number.

Grading: 1 point for each set of bits.

Part B [4 points]

What is the minimum size of a page table entry? Ignore bits to guide the replacement policy, but ensure you account for all other necessary bits. Assume a valid page can have every combination of read, write, and execute permissions. Any extraneous bits mentioned will fetch negative points.

Solution: The PTE needs to store 36 bits of physical address. There are also 3 permission bits, 1 valid bit, and 1 dirty bit for a minimum of 41 bits.

Grading: 1 point each for the correct number of physical address, permission, valid, and dirty bits. -1 for any extraneous bits. 4 points maximum. 0 points minimum.

Part C [2 points]

What is the minimum size of an entry in a fully-associative TLB? Again, ignore bits for the replacement policy. Any extraneous bits mentioned will fetch negative points.

Solution:

In a fully associative TLB, every entry needs the entire virtual page number as a tag, in addition to the information the PTE requires. The size of a TLB entry therefore is 52 + 41 = 93 bits.

Grading: 1 point for the correct size of the tag and 1 point for the correct size of the “data” bits. -1 point for extraneous bits. Maximum 2 points. Minimum points is 0. No penalty for cascading errors; i.e., if the data bits are the same as calculated in part B, 1 point will be given.

Part D [2 points]

If the processor did not have a TLB, how long would each address translation take? Assume that the complete page table resides in main memory and there are no page faults.

Solution:

A PTE needs to be fetched from the page table (which resides in main memory) to perform address translation. Each PTE is 41 bits, therefore two bus transfers are required to send the data back.

The time required would be 10 cycles to send the address + 50 cycles to access the first word + 2*20 cycles to send the data back = 100 cycles.

Grading: 1/2 point for realizing that two bus transfers would be required to send a PTE from memory to the processor. 1.5 points for stating the correct memory access time formula (½ point partial credit for each component – cycles to send the address, access the first word, and cycles to send the data back).

If you clearly stated that you assumed a multilevel page table and multiple memory accesses for a page table entry, we did not deduct points.

Part E [2 points]

Now assume that the processor has a fully-associative TLB with a hit rate of 97% and a hit time of 1 cycle. How long do address translations take now, on average? What is the speedup over the system without a TLB? Again, assume that all pages reside in memory and there are no page faults.

Solution:

Average address translation time = TLB hit time + miss rate x miss penalty = 1 + 0.03*100 = 4 cycles.

Speedup = 100/4 = 25x.

Grading: 1.5 points for the correct formula for address translation time. 1/2 point for speedup.

Problem 2 [15 points for undergraduates, 21 points for graduates]

ALL STUDENTS SHOULD SOLVE PARTS A TO E. ONLY GRADUATE STUDENTS SHOULD SOLVE PARTS F AND G.

Data cache performance can be improved if a processor loads data into the cache before the program requests it. This prefetching requires predicting which data will soon be accessed. A simple strategy only prefetches on a cache miss, and loads the requested block as well as the following block. Consider the effect of this strategy on the memory accesses of the following programs, under the following assumptions:

- The cache block size is 16 bytes
- Array entries are 4 bytes
- Arrays are aligned so the first element is at the start of a cache block
- The cache is initially empty
- Local variables are stored in registers, not memory
- The cache is sufficiently large that there will be no capacity misses
- The cache is sufficiently large/associative that there will be no conflict misses
- If a processor issues a load to an address that was previously prefetched, assume the prefetch already returned the block in the cache and the load will be a hit.

For each program, give the number of data cache misses that will occur with and without next-line prefetching, and how much data will be loaded into the cache with and without next-line prefetching.

Part A [3 points]

```
for (int i = 0; i < 128; ++i) {
   a[i] = sin(a[i]);
}
```

Solution:
128 words / 4 words/cache line = 32 cache lines. Without prefetching, there will be 32 misses. With prefetching, the misses will be halved; i.e., 16 misses. Either way, 128*4 = 512 bytes of data will be read into the cache.

Grading:
1 point for number of misses without prefetching. 1 point for number of misses with prefetching. 1 point for amount of loaded data.

Part B [3 points]

```
for (int i = 127; i >= 0; --i) {
        a[i] = sin(a[i]);
}
```

Solution:
Prefetching goes in the wrong order, so both strategies incur 128/4 = 32 cache misses. The first prefetch on the array reads one unnecessary block, so with prefetching 128*4+16 = 528 bytes will be loaded. Without prefetching only the blocks covering the array will be loaded, or 128*4 = 512 bytes.

Grading:
1 point for number of misses. 1 point for amount of loaded data without prefetching. 1 point for amount of loaded data with prefetching.

Part C [3 points]

```
for (int i = 0; i < 128; ++i) {
        a[4*i]= sin(a[4*i]);
}
```

Solution:
Here each access to a occurs to successive cache lines, in order. Without prefetching there will be 128 cache misses. With next-line prefetching there will be half as many misses, 128/2 = 64 cache misses. Either way, all cache blocks covering an accessed element of A will be loaded, and no unnecessary blocks will be prefetched, so a total of 128*16 = 2048 bytes of data will be loaded.

Grading:
1 point for number of misses without prefetching. 1 point for number of misses with prefetching. 1 point for amount of loaded data.

Part D [3 points]

```
for (int i = 0; i < 128; ++i) {
        a[8*i] = sin(a[8*i]);
}
```

Solution:
Now each access to a is sufficiently far apart that the prefetched lines will not even be used. With or without prefetching there will be 128 misses. Without prefetching only necessary blocks will be loaded, for a total of 128*16 = 2048 bytes of data loaded. With prefetching twice as many blocks will be loaded, for a total of 2*128*16 = 4096 bytes of data loaded.

Grading:
1 point for number of misses. 1 point for amount of loaded data without prefetching. 1 point for amount of loaded data with prefetching.

Part E [3 points]

Now consider software prefetching for the code in part D. Make the following additional assumptions:

- Statements in the code are executed sequentially. The loop test takes 4 cycles per invocation. The assignment statement takes 20 cycles if there is no cache miss (those 20 cycles include multiplication to find the index, the load, the sin computation, and the store), and an additional 40 cycles if there is a data cache miss.
- There is a data prefetch instruction with the format prefetch(array[index]). This prefetches the single block containing the word array[index] 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 data to be prefetched is not already in the cache, then it takes 35 cycles for the data to get loaded into the cache.
- Assume the memory system can handle an infinite number of concurrent prefetches; e.g., the cache has infinite MSHRs.
- The instruction cache is perfect; i.e., the hit rate is 100% and it can be ignored for this problem.

Modify the code in part D to use software prefetching. Do not add startup or cleanup code. Given that restriction, avoid as many cache misses as possible. Additionally, write code that issues as few prefetches as possible, given the number of misses remaining. As always, be sure to explain your answer.

Solution:
Since the prefetch latency is 35 cycles and the computation takes 20 cycles in the best case, we need to prefetch two iterations ahead to completely hide the prefetch latency.

```
for (int i = 0; i < 128; i++) {
        prefetch(a[8*i+16]);
        a[8*i] = sin(a[8*i]);
}
```

Grading: 1 point for having just one prefetch. 2 points for the correct address. 3 total points.

Part F [3 points] – ONLY GRADUATE STUDENTS SHOULD SOLVE THIS PROBLEM

Describe the design for a hardware prefetcher that can handle all the cases from parts A – D. Your prefetcher must minimize both the number of misses and the amount of useless data that is prefetched. Assume the prefetcher is invoked only on loads. You have to explain the design only at a conceptual level (e.g., analogous to the level at which we explained branch predictors in the lecture); i.e., you do not need to show the actual circuitry.

Solution:
We need to add a predictor to the prefetcher that learns the stride of the misses and uses that stride to compute the prefetch address. This involves the following steps:

(1) We need to keep track of the address of the last miss in an "address" register.
(2) At the current miss, a simple subtractor can determine the difference between the address of the current miss and that of the last miss (stored in the address register). This difference (the stride) is stored in a "stride" register. Note that the stride can be positive or negative.
(3) This stride value is then added to the address of the current miss and a prefetch to the computed address is issued.

Grading: 1 point for each of the three steps above.

Part G [3 points] – ONLY GRADUATE STUDENTS SHOULD SOLVE THIS PROBLEM

Now assume that the loops in parts A to D are modified so that they additionally traverse (with some constant stride) an array that is disjoint from array "a." Does your prefetcher design for part F still work as well? If yes, explain why. If not, explain how you will modify it to make it work efficiently for the new loops. Credit will be given for this part only if a reasonable solution is provided for part F.

Solution:
The previous solution does not work well because the histories of the two array accesses interfere with each other. The predictor should be modified so that the address and stride are now stored in a table that is indexed by the program counter.

Grading: 1 point for an explanation of whether the prefetcher from part F will or will not work. If the prefetcher from Part F already works for this case, then two additional points. If the prefetcher from part F does not work for this case, then 2 points for a modification to make it work correctly.

Problem 3 [14 points]

This question concerns a snooping *update* (as opposed to invalidate) cache coherence protocol. Consider a system where the processors are connected by a bus and cache coherence is maintained through a snooping update protocol. In such a protocol, when a cache modifies its data, it broadcasts the updated data on a bus using a *bus update* transaction, if necessary. Memory and all caches that have a copy of that data then update their own copies. This is in contrast to the invalidation protocol discussed in class where a cache invalidates its copy in response to another processor's write request to a block.

Our update protocol has three states – CE, CS and DE:

- CE (Clean Exclusive): The block is present *only in this cache (exclusively)* and memory also has the same (clean) copy.
- CS (Clean Shared): The block is present in *several caches (shared)* and memory and all those caches have the same (clean) copy.
- DE (Dirty Exclusive): The block is present *only in this cache (exclusively)* and the data in the cache is updated or dirty (i.e., a more recent version than the copy in memory).

All caches are write-allocate. A write-back policy is used if the line is in DE or CE state. For a line in CS state or a line not present in the cache, a write-through policy is used. The bus has a special line called Shared Line (SL) whose state is usually 0. *When cache i performs a bus transaction for a specific cache line*, all the caches that have the same line pull up the Shared Line (SL) to 1. If no other cache has the line, the Shared Line (SL) remains at 0. When cache i performs a bus transaction, it uses the state of the Shared Line (SL) to determine whether to change to an exclusive state or the shared state.

Assume that if a request is made to a block for which memory has a clean copy, memory will service that request. If the memory does not have a clean copy, the cache with the updated block will service the request and memory will also get updated.

For the question below, consider the following bus transactions:

- BR: Bus Read – Request to get the cache line (on a cache miss).
- BU: Bus update – Request to update copies of the cache line in memory and other caches with the new value of a word in the block.
- BRU: Bus read and update – A combination of BR and BU.

Note: you are not required to consider Bus Writeback, which may take place on a replacement.

Part A [8 points]

Fill out the following state transition table for the processor i performing a memory instruction. Show the next state for a block in the cache of processor i and any bus transaction performed by processor i. Each entry should be filled out as:

Next State/Bus Transaction (e.g. CS/BR), where

Next State = CS, CE, DE or NIC (Not in Cache; i.e., a cache miss)

Bus Transaction = BR, BU, BRU, or NT (No transaction)

Note: If an entry is not possible (i.e., the system cannot be in such a state) write "Not Possible" in that entry.

	SL is 0 **if** proc i does a bus transaction		SL is 1 **if** proc i does a bus transaction	
Current State in processor i	**Read by proc i**	**Write by proc i**	**Read by proc i**	**Write by proc i**
CE	CE/NT	DE/NT	CE/NT	DE/NT
CS	CS/NT	CE/BU	CS/NT	CS/BU
DE	DE/NT	DE/NT	DE/NT	DE/NT
NIC	CE/BR	CE/BRU	CS/BR	CS/BRU

Grading: ¼ point for each Next State or Bus Transaction value that is correct. That is, each entry is worth ½ point.

Part B [6 points]

Fill out the following state transition table for the cache of processor i. Show the next state for a block in the cache of processor i and any action(s) taken by the cache when a bus transaction is initiated by another processor j. Each entry should be filled as:

Next State/Action (e.g. CS/UPDL)

Where

Next State = CS, CE, DE or NIC (Not in Cache)

Action = PULLSL1: Pull SL to 1

UPDL: Update block in cache i (i.e., one's own cache)

PROVL: Provide block in response to BR or BRU (main memory is also updated)

NA: No Action

Note: If an entry is not possible (i.e., the system cannot be in such a state) write "Not Possible" in that entry.

State in proc i	BR by proc j	BU by proc j	BRU by proc j
CE	CS/PULLSL1	Not possible	CS/PULLSL1, UPDL
CS	CS/PULLSL1	CS/UPDL, PULLSL1	CS/UPDL, PULLSL1
DE	CS/PROVL,PULLSL1	Not possible	CS/PROVL, UPDL, PULLSL1
NIC	NIC/NA	NIC/NA	NIC/NA

Grading: ¼ point for each correct Next State, 1/4 point for each correct component of each Action. Each "Not Possible" carries ½ point.

Problem 4 [6 points]

You are to implement a simple counting semaphore using `test_and_set`. Each semaphore contains an integer value. You must write two functions, each of which must perform atomically:

- `sem_post`: increment the value
- `sem_wait`: wait for the value to be positive, then decrement the value

For `test_and_set`, use the following prototype to atomically set `lock_var` to 1 and return its previous value: `int test_and_set (Lock lock_var);`

Add C-like pseudocode to the stub below. Ignore initialization. Assume sequential consistency.

```
typedef struct {
      int value;
      Lock lock_var;
} sem_t;

sem_post (sem_t *s)
{
       //  your code below


      (1) while (test_and_set(s->lock_var)!=0);
      (2) s->value++;
      (3) s->lock_var = 0;




}
```

Grading: 3 points. 1 point for each of (1), (2) and (3) instructions.

```
// repeated struct from prior page for convenience
// typedef struct {
//    int value;
//    Lock lock_var;
// } sem_t;

sem_wait (sem_t *s)
{
      // your code below
(1)   while(1) {
(2)         while(test_and_set(s->lock_var)!=0);
(3)         if(s->value > 0){
                  // decrement value
                  s->value--;
                  s->lock_var = 0;
                  break;
            }
(4)         else {
                  //release lock, try again
                  s->lock_var = 0;
                  continue;
            }
      }
}
```

Grading: 3 points. 0.5 point for each of (1), (2). 1 point for block at (3) and 1 point for block at (4).

Problem 5 [8 points]

Consider the following system S:

- The system S contains a cache per processor that can contain shared read/write data. Cache coherence is maintained through an invalidation snooping protocol.
- When a processor's cache controller sees an invalidate on the bus for a line present in its cache, it buffers this invalidate in a local buffer. The bus is free for the next transaction as soon as such buffering is done. The cache controller will apply the invalidate to its cache line some time later (e.g., when the cache is not being used by the processor). It is also possible that the buffered invalidates are applied to a cache in an order different from the order in which they were received from the bus.
- A processor is allowed to have multiple outstanding memory accesses and these accesses could occur out of program order.
- An instruction called memory_barrier, denoted by MB, is provided with the following specification.
- An MB by processor P is not issued until the following is true of all operations op of processor P that are before MB by program order:
    - if op is a read, then it has returned its value, and
    - if op is a write, then the invalidate for that write has been applied to all the cache lines with an older value.
- Further, processor P does not issue any memory operation until all preceding MB instructions (by program order) have been issued.

The memory consistency model of system S is not sequential consistency and does not impose any constraints on the ordering of loads and stores other than that due to the MB instruction.
Answer the following two parts for the above system.

Part A [4 points]

Consider the following program. *(Note: Operations in a vertical column are issued by the same processor, and appear in program order.)*

Initially X = Y = 0

P1	P2	P3
X = 1	tmp1 = X	tmp2 = Y
	Y = 1	tmp3 = X

Suppose processor P2's read of X and processor P3's read of Y both return the value 1. Then what values could processor P3's read of X return on a sequentially consistent system? What values could it return on system S?

Solution:

Under sequential consistency, memory operations appear to execute atomically and in program order. Since we are told that P2's read of X and P3's read of Y both return the value 1, we can conclude that the order of memory operations as seen by the system is:
(X = 1) → (tmp1 = X) → (Y = 1) → (tmp2 = Y) → (tmp3 = X).

Therefore, the read of X by P3 returns a 1. The above order is correct because of the following reasons: We are given that the read of X at P2 returns a 1. This implies that the write of 1 to X by P1 happened before it. The store to Y by P2 comes after reading X in program order. The same store happens before the read of Y on P3 since we are told that the read of Y at P3 returns a 1. Finally, the read of X by P3 occurs after the read of Y in program order and therefore appears after it in the system.

On system **S**, since there are no guarantees on the order in which loads and stores are executed by each processor, P3 could either read a 0 or a 1 for the value of X.

Grading:
2 points for the answer about the sequentially consistent system and 2 points for the answer for the hypothetical system.

Part B [4 points]

The designers of system S claim that S is simple to program because *programmers who want sequential consistency can simply put MB instructions before and after every memory operation to get sequential consistency*. Is the italicized statement true? If not, why not and how would you modify system S to make the statement true?

Solution:

The use of the **MB** instruction can ensure that memory instructions are executed in program order by the processors. However, the semantics of **MB** do not guarantee the atomicity of memory operations in the system.
To make system S appear to be sequentially consistent, atomicity has to be guaranteed. Atomicity can be guaranteed if on a write, the cache-coherence protocol guarantees that all processors in the system "appear" to see the modification due to the write at the same time. This can be accomplished by not allowing reads to return the value of a write until the invalidates for the write reach all caches with a copy of the line.

Grading:
2 points for explaining why using MB is not enough to ensure sequential consistency and 2 points for saying how to modify S to ensure it.

Problem 6 [5 points]

This question concerns the mini-project presentations in class. Circle the most appropriate choices for each question below. Some questions have multiple correct choices. Points will be given for a question only if **all appropriate** choices for that question are circled and **no incorrect** choice is circled. You only **need to answer five of the six questions correctly for full credit** (if you answer all six, we will give full credit if five are correct).

Part A

What is true about AMD's recent Zen processors?

a) They use the RISC V ISA to achieve much higher IPC than previous AMD processors.
b) There are several features for better security.
c) The L1 data cache uses a linear address utag/way-predictor.
d) There are no bank conflicts anywhere in the memory hierarchy.

Part B

What is true about the Fujitsu A64FX?

a) It targets low cost smartphones.
b) It supports a Torus interconnect.
c) All memory resides on the processor chip die for fast access.
d) It uses an Arm ISA.

Part C

What is true about the Google TPU?

a) There are several generations of the TPU.
b) Newer generations avoid the expensive High Bandwidth Memory (HBM) used by GPUs.
c) The design employs concepts from systolic array architectures.
d) It is designed primarily for Neural Networks.

Part D

What is true about the IBM Power 9?

a) It employs a hardware cache coherence protocol.
b) It supports several on-chip and off-chip accelerators.
c) It is a single-issue in-order processor.
d) All of the above.

Part E

What is true about the Intel Alder Lake architecture?

a) It supports Advanced Vector Extension instructions (AVX).
b) The processor has two types of cores.
c) It is targeted specifically for supercomputers.
d) It includes a thread director for the purpose of avoiding deadlocks in the coherence protocol.

Part F

What is true about the Nvidia Ampere GA102 GPU?

a) The license agreement forbids its use for crypto currency mining.
b) It supports a hardware ray tracing engine.
c) It supports tensor cores for tensor processing.
d) None of the above.