CS 433 – Final Exam – Dec 19, 2018

Professor Sarita Adve

Time: 3 Hours

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

Name	**Solutions**
NetID	
Category	Undergraduate 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 solution has ***6 problems*** and ***14 pages*** (including this one). **Only graduate students should solve problems 2F, 2G, and 5B. All students should solve the rest of the problems.** Please budget your time appropriately. Good luck!

Problem	Maximum Points	Received Points
1	11	
2	14 (undergrads), 20 (grads)	
3	8	
4	8	
5	6 (undergrads), 11 (grads)	
6	5	
Total	52 (undergrads), 63 (grads)	

Problem 1 [11 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:

Since 4KB is 2**11 words (for 16 bit words), 11 bits are needed as the 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 [3 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. 1 point for the total number of PTEs. 0.5 point for rounding up the PTE size to a multiple of word size and 0.5 point 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, bits11:15.

Grading:
1.5 points for each answer. ½ point for correct number of bits, ½ point for correct range of bits, and ½ point for specifying virtual or physical address.

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 that does not require waiting for address translation before indexing the cache.

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 arbitrary virtual pages are allowed to map to the same physical page, then these two virtual addresses could have different index bits. One solution is that while indexing, the cache should assume bit 11 could be 0 or 1 and look for the data in the two corresponding sets in parallel (i.e., in 8 total ways) Assuming the translation completes in the time that these ways are accessed, the correct way can be determined once the data is accessed.

Grading:
½ point for giving the range of index bits. 1 point for noticing and explaining how 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 [14 points for undergraduates, 20 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 (in address order). Consider the effect of this strategy on the memory accesses of the following programs, under the following assumptions for the data cache:

- 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 each 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 [2 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 40 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 40 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. 1 point for the correct address. 2 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 [8 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, the caches are write-back and write-allocate and cache coherence is maintained through a snooping update protocol called protocol X. In a snooping update protocol, when a cache modifies its data, it broadcasts the updated data bytes 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.

Protocol X 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 the line with 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 – assume all writes to A are word writes. 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.

Event	MSI state for block A	Protocol X state for block A	External actions by Processor 1 for MSI (say none if no action)	External actions by Processor 1 for X (say none if no action)
Local read A	**S**	**V-E**	**Read A on Bus**	**Read A on Bus**
Local write A	M	D	Invalidate	None
Bus read A	S	S	Send line	Send line
Bus write A	I	S	None	None
Local read A	S	S	Read A on Bus	None
Local write A	M	S	Invalidate	Update word
Bus read A	S	S	Send line	None
Local write A	M	S	Invalidate	Update word
Local write A	M	S	None	Update word

Grading: 8 points. ¼ point each entry. No penalty for cascading errors

Problem 4 [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 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 5 [6 points for undergraduates, 11 points for graduates]
ALL STUDENTS SHOULD SOLVE PART A. ONLY GRADUATE STUDENTS SHOULD SOLVE PART B.

Part A [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.

```
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).

Part B [5 points] – ONLY GRADUATE STUDENTS SHOULD SOLVE THIS PROBLEM

Now assume that the system *T* relaxes sequential consistency by not enforcing any program order constraints for a given thread. The system provides the memory barrier instruction, MB, in Problem 4. Insert the minimum number of MB instructions in your code for sem_post and sem_wait in Part A so that the code will give sequentially consistent results for system *T*. Write or show clearly where these instructions would be inserted. Credit will be given for this part only if the solution in Part A is mostly correct.

Solution:

The MB instruction must be placed after every test of the lock variable (1 instance each in post and wait) and before every reset of the lock variable (one instance for post and two for wait).

Grading: 1 point for each correct placement. -0.5 for each extra MB.

Problem 6 [5 points]
This question concerns the mini-project presentations in class. Circle the most appropriate choices for each question below. Points will be given for a question only if all appropriate choices for that question are circled and no incorrect choice is circled. Note: Points were also given if the majority of the correct choices were circled and no incorrect choice was circled.

Part A [1 point]

Which of the following is true of the AMD Naples EPYC family processor?
a) It uses multiple chips in one package instead of a monolithic die
b) It uses a perceptron (neural network) based branch predictor
c) It is designed through a historic joint initiative between AMD and IBM
d) It is built with 0.001 nm technology

Solution: a, b

Part B [1 point]

Which of the following is true of the ARM A55 Cortex processor?
a) It implements the Neon SIMD instruction extension
b) It was designed to be implanted in the human cerebral cortex
c) It uses the Big.Little architecture
d) It has private L1 and L2 caches with an optional shared L3 cache

Solution: a, c, d

Part C [1 point]

Which of the following is true of the Tensor Processing Unit (TPU) accelerator?
a) It is targeted specifically for neural networks
b) It consists of a significant matrix multiply unit
c) It consists of a large "weight" memory
d) It has a large out-of-order core on die to help with workloads that cannot be accelerated

Solution: a, b, c

Part D [1 point]

Which of the following is true of the IBM Power 9 processor?
a) It allows implementing multiple loads and stores as atomic transactions in hardware
b) It is designed through a historic joint initiative between AMD and IBM
c) It supports simultaneous multithreading
d) It supports heterogeneous computing through the Coherent Accelerator Processor Interface (CAPI)

Solution: a, c, d

Part E [1 point]

Compared to all the processors and accelerators covered in the project presentations, the following is/are true for the NVIDIA Tesla V100 GPU:
a) It supports the largest number of threads
b) It has the most sophisticated branch predictor
c) It has the most restrictive programming model
d) It supports the largest number of registers

Solution: a, d