

CS 433 Final Exam – May 11, 2006

Professor Sarita Adve

Time: 7:00-10:00pm, 3 hours

Please *clearly* print your **full name**, **NetID** and **circle the appropriate category** in the space provided below. **Failure to completely fill out this table will result in a ZERO grade.**

Name	SOLUTIONS		
NetID			
Category (circle one)	3 Credit Hours	4 Credit Hours	
	UG	Grad(On-Campus)	Grad(I2CS)

Instructions

1. You may only use class handouts from this semester's offering, the course text (*Computer Architecture: A Quantitative Approach* - 3rd Edition - by Hennessy and Patterson), your own homework submissions for this course, and notes written or typed by yourself. No other materials are allowed, including other books, notes prepared by others, or materials from previous offerings of this course or from other universities.
2. Calculators are allowed. You may not use any other electronic devices.
3. Please do not turn in your loose scrap paper. Limit your answers to the space provided, if possible. If not, write on the back of the same sheet. You may use the back of each sheet for scratch work.
4. In all cases, show your work. *No credit will be given for numeric answers 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 getting the final solution.
5. **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 you make in your answers.**
6. This exam has **5** problems and **13** pages (including this one). All students should attempt all problems. Please budget your time appropriately. Good luck!

Problem	Maximum Points	Received Points
1	14	
2	20	
3	5	
4	8	
5	18	
Total	65	

Problem 1 [14 points]

Consider a single processor system with the following specification:

- Data cache size is 1 KB (for data) and is 4-way set-associative. Its block size is 16 bytes. It is physically indexed and physically tagged. It uses LRU for replacement within a set and a write-allocate write-back policy for writes.
- 2-way set-associative TLB with 32 total entries and an LRU replacement policy.
- Physical addresses of 24 bits.
- Virtual addresses of 32 bits.
- Byte addressable memory.
- Page size is 64 KB.

Part A [4 points]

For each field listed below, indicate the bits of the virtual address that correspond to it. Show your work.

The virtual page offset:

The virtual page number:

The TLB index:

The TLB tag:

Solution:

We can think of a virtual address as (page number, page offset).

The virtual page offset:

The page size is 64 KB, so $64 \text{ KB} = 2^{16}$. 16 bits are needed to indicate the page offset. The last 16 bits of the virtual address (virtual address bits 15:0) correspond to the page offset.

The virtual page number:

Since the last 16 bits are for page offset, we have $32 - 16 = 16$ bits for virtual page number. Virtual address bits 31:16 correspond to the virtual page number.

We can also think of the virtual address as (tag, index, page offset).

The TLB index:

The TLB has 32 entries total and is 2-set way associative, so we will need $32/2 = 16 = 2^4$ sets. We need 4 bits to indicate the TLB index. We know the page offset takes the last 16 bits of the virtual address and the index field is before the offset field, so 19:16 correspond to the index.

The TLB tag:

$32 - 4 \text{ index bits} - 16 \text{ offset bits} = 12$ bits are left for TLB tag, which are the first 12 bits of the virtual address. 31:20 correspond to the tag field.

Grading:

1 point for each field. No point is given if no work is shown in deriving the answer.

Part B [5 points]

For each field listed below, indicate the bits of the physical address that correspond to it. Show your work.

The physical page offset:

The physical page number:

The cache block offset:

The cache index:

The cache tag:

Solution:

We can think of a physical address as (page frame number, page offset)

The physical page offset:

Same as virtual page offset, which is 16 bits. The last 16 bits of the physical address (15:0) correspond to this field.

The physical page number:

The remaining bits of physical address are $24 - 16 = 8$ bits. The first 8 bits of the physical address (23:16) correspond to this field.

We can also think of the physical address as (tag, index, block offset).

The cache block offset:

The block size is 16 bytes, so $16 = 2^4$. 4 bits are used for this field, so the last 4 bits of the physical address (3:0) correspond to this field.

The cache index:

The cache is 4-way set-associative and the cache size is 1 KB, so we have $1 \text{ KB} / (4 * 16) = 2^4$ sets. We will need 4 bits to indicate this field, so 7:4 bits of the physical address correspond to this field.

The cache tag:

$24 - 4 \text{ index bits} - 4 \text{ block offset bits} = 16$, so the first 16 bits of the physical address correspond to tag.

Grading:

1 point for each field. No point is given if no work is shown in deriving the answer.

Part C [5 points]

The following table gives the state of the TLB, providing the address translations for the relevant virtual addresses (using hexadecimal notation). Valid bit of 1 implies the entry is valid.

TLB							
Index	Tag	Physical page number	Valid	Index	Tag	Physical page number	Valid
F	FFF	EF	1	7	560	6F	1
	FFC	71	1		8F1	A9	1
E	FF9	28	1	6	A31	31	1
	FF4	AB	1		3B0	49	1
D	F55	CD	1	5	9E1	55	1
	FF6	34	1		041	00	1
C	446	1F	1	4	151	48	1
	00A	BB	0		1D0	32	1
B	6AC	D5	1	3	0A0	6A	1
	715	8E	1		9D5	56	1
A	31A	D4	1	2	68E	60	1
	51D	64	1		E71	78	1
9	39E	5A	1	1	A34	EE	1
	43B	F8	1		8BB	AA	1
8	732	DE	1	0	7A0	99	0
	48F	B2	0		3F0	94	1

The following table lists a stream of virtual address accesses by the processor (all addresses are hexadecimal). Complete the rest of the entries in the table using the above information and your solutions to parts A and B. For the TLB hit and Cache hit, specify “yes” or “no.” Assume initially the cache is empty.

Virtual address	Corresponding physical address	Part of physical address that indexes the cache	TLB hit?	Cache hit?
FFFF ABCD				
446C CEBA				
48F8 ABCD				
446C CEAB				

Solution:

Virtual Address	Corresponding Physical Address	Part of Physical Address that indexes cache	TLB hit?	Cache hit?
FFFF ABCD	EF ABCD	C	Yes	No
446C CEBA	1F CEBA	B	Yes	No
48F8 ABCD	B2 ABCD	C	No	No
446C CEAB	1F CEAB	A	Yes	No

Grading:

0.25 point per entry. 1 point for all correct entries.

Problem 2 [20 points]

Consider the following program:

```
int i, int j, double result[4][100], double a[101][4]

for (i=0; i<4; i++)
{
    for (j=0; j<100; j=j++)
        result[i][j] = a[j][0]*a[j+1][0] + 0.5;
}
```

Arrays **result** and **a** contain 8 byte double precision floating point elements.

Assume the following:

- The program is running on a machine with an L1 data cache.
- The cache is fully associative with 100 blocks and an LRU replacement policy. The block size is 16 bytes. It is write-through and no write-allocate.
- Assume that only the accesses to the array locations generate loads to the data cache. The rest of the variables are all allocated in registers.
- The arrays are stored in row major form.
- The arrays start at cache line boundaries.
- Initially, the data cache is empty.

Part A [4 points]

Explain which loads to the L1 data cache result in misses for the above program. Give the total number of such misses and indicate which are capacity, conflict, and cold misses. Assume that the processor issues loads in the order in which they appear in the program.

Solution:

When $j=0$, $a[0][0]$ and $a[1][0]$ are accessed. Their misses bring in blocks with $a[0][0]$ and $a[0][1]$ (not used); and $a[1][0]$ and $a[1][1]$ (not used). When $j=1$, $a[1][0]$ and $a[2][0]$ are accessed. $a[1][0]$ hits and $a[2][0]$ misses.

In the loop where $i=0$, there will be 101 misses for $a[0][0] \dots a[100][0]$.

Since the cache size is only large enough to store 100 blocks of a , when $a[100][0]$ is accessed, the cache will bring in $a[100][0]$ and $a[100][1]$, and replace $a[0][0]$ and $a[0][1]$ (since it is LRU). When $a[0][0]$ is accessed for $i=1$ and $j=0$, the cache will not find $a[0][0]$ in the cache anymore, so we need to bring in $a[0][0]$ and $a[0][1]$, replacing $a[1][0]$ and $a[1][1]$. When $j=1$, $a[1][0]$ and $a[2][0]$ are accessed, but $a[1][0]$ is not in the cache anymore, so we need to bring $a[1][0]$ and $a[1][1]$, replacing $a[2][0]$ and $a[2][1]$ in the cache, and so on. In other words, there will be 101 misses for each i .

Given there are 4 iterations of i , $4 \cdot 101 = 404$ misses for array a .
Total = 404 misses

The first 101 misses are cold misses while the rest are capacity misses.

Grading:

- 1 point for identifying the correct number of cold misses.
- 1 point for identifying the correct number of capacity misses.
- 2 points for reasonable explanations.

Part B [5 points]

Suppose the processor has a data prefetch instruction with the format `prefetch(array[index])`. This prefetches the entire cache block containing the word `array[index]` into the data cache.

Insert prefetch instructions in the above code to minimize the number of cache load misses. Do not change the order of the original load instructions. Assume that 7 iterations of the inner loop are required to cover the latency of an L1 cache miss. Do not worry about the epilog or prolog for the loop in which you are inserting prefetches (i.e., do not worry about prefetching for the first few iterations of this loop and do not worry about the extra prefetches in the last few iterations of this loop). Other than prolog and epilog related issues, your code should minimize any unnecessary prefetches. Write your code below.

Solution:

Prefetches need to be added only for array a.

```
for (i=0; i<4; i++) {  
    for (j=0; j<100; j++)  
    {  
        prefetch(a[j+8][0]);           /* a[j+1][0] for 7 iterations later */  
        result[0][j]=a[j][0]*a[j+1][0] + 0.5;  
    }  
}
```

Grading:

- 1 point for recognizing only array a needs prefetching.
- 1 point for correct placement and form of the prefetch instruction, even if the offset for the j index is wrong.
- 1 point for using the correct prefetching offset, `a[j+8][0]`.
- 2 points for code that minimizes any unnecessary prefetches. 1 point if the code does not minimize unnecessary prefetches.

Part C [7 points]

Assume now we have a cache of infinite size. Repeat Part B. Explain the difference in the solutions for parts B and C.

Solution:

```
for (j=0;j<100;j++)
{
    prefetch(a[j+8][0]);          /* a[j+1][0] for 7 iterations later */
    result[0][j]=a[j][0]*a[j+1][0] + 0.5;
}

for (i=1; i<4; i++)
{
    for (j=0; j<100; j++)
    {
        result[i][j]=a[j][0]*a[j+1][0] + 0.5;
    }
}
```

In part C, prefetches are required for the capacity misses in the loops with $i=1, 2, 3$. In part D, no such prefetches are required since there are no capacity misses. The iteration for $i=0$ therefore needs to be peeled out from the rest of the loop.

Grading:

1 point for reasonable explanations on the difference in the solutions for part C and D.

1 point for recognizing only array a needs prefetching.

1 point for recognizing iteration $i=0$ needs to be separated from the rest of the loop.

1 point for correct placement and form of the prefetch instruction, even if the offset for the j index is wrong.

1 point for using the correct prefetching offset, $a[j+8][0]$

2 points for code that minimizes any unnecessary prefetches. 1 point if the code does not minimize unnecessary prefetches.

Part D [4 points]

Now consider again the original code (i.e., without prefetches) with the original cache hierarchy (as in Part A). Besides prefetching, what other software-only technique can you use to avoid the capacity misses in the original code? Rewrite the original code to include this technique below.

Solution:

We can use loop interchange.

```
for (j=0; j<100; j++)
{
    for (i=0; i<4; i++)
        result[i][j]=a[j][0]*a[j+1][0] + 0.5;
}
```

Grading:

1 point for recognizing that the accesses in array a don't change for each iteration of i.

1 point for applying software-only technique to the code.

2 points for code that avoids capacity misses.

Problem 3 [5 points]

Consider a cache-based system with a conventional directory-based cache coherence protocol such as that discussed in class. Suppose each processor in this system issues memory accesses to its cache in program order. Further, a processor issues a memory access to its cache only after all its previous reads in program order have returned their values and all its previous writes in program order have reached their corresponding directory.

Is this system guaranteed to be sequentially consistent? If yes, why? If not, show and explain an example code and execution where sequential consistency is violated and explain what additional condition the system should satisfy to obtain sequential consistency for your example.

Solution:

The system is not sequentially consistent. Consider the following code:

Initially A=B=0

P1 P2

Write, A, 1 Write, B, 1

Read, B Read, A

Assume P1 and P2 have both A and B in their caches with the value 0. P1 could issue its read of B after its write of A reaches the directory but before the invalidation for A reaches P2. Similarly, P2 could issue its read of A after its write of B reaches the directory but before the invalidation for B reaches P1. Then both P1 and P2 could return the value 0 for their reads, which is a violation of sequential consistency.

To achieve sequential consistency for the above example, a processor should not issue its request until invalidations corresponding to its previous writes have been received by all other processor caches.

Grading:

1 point for correct answer.

2 points for showing a reasonable example.

2 points for explaining any additional conditions needed for the system to obtain sequential consistency.

Problem 4 [8 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 [4 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-like pseudo-code to this stub:

```
int head; /* index for the head of the queue */
int tail; /* index for the tail of the queue */
int index; /* current array index for enqueueing or dequeueing */

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

enqueue(item) {

    queue[index] = item;

}

dequeue() {

    item = queue[index];

    return item
}
```

Part B [4 points]

Repeat part (A) using the fetch&increment instruction instead of the test&set for synchronization.

```
int head; /* index for the head of the queue */
int tail; /* index for the tail of the queue */
int index; /* current array index for enqueueing or dequeuing */
```

```
/* 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:

[Part A]

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

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

Grading scheme: 2 points for each function. A solution that uses different locks for enqueue and dequeue is also correct.

[Part B]

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

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

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

Problem 5 [18 points]

This question concerns a cache coherence protocol implemented in the DEC Firefly machine. This is a snooping *update* (as opposed to invalidate) cache coherence protocol and is implemented for a system where the processors are connected by a bus. In a snooping update protocol, when a cache modifies its data, it broadcasts the updated data on the bus using a *bus update* transaction, if necessary. 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 line. In the Firefly protocol, the copy in memory is also updated on a bus update.

With the Firefly protocol, a cache line can be in two possible states (other than invalid):

- Dirty Exclusive (DE): The line is present *only (exclusively) in this cache*. The data in the cache is possibly *updated or dirty*, i.e., it may be a more recent version than the copy in memory.
- Clean Shared (CS): The line may be present in other caches (shared) too and memory and all those caches have the same (clean) copy.

All caches are write-allocate. A write-back policy is used if the line is in DE state. For lines in CS state, 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. Cache *i* uses the state of the Shared Line (SL) *when it performs a bus transaction* to determine whether to change to an exclusive state or the shared state.

If a request is made to a block for which memory knows it has a clean copy, then memory will service that request. Otherwise, the appropriate cache will service the request and memory will also get updated.

Consider the following bus transactions:

- BR: Bus Read – Place a read request on the bus.
- BU: Bus Update – Update copies in memory and other caches with the same cache block.

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

Part A [12 points]

Fill out the following state transition table for a processor i performing a memory access. 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 as:

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

Where *next state* = CS, DE or NIC (Not In Cache – i.e., a cache miss)

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

For Bus Reads, indicate who will provide the copy of the requested line.

	SL is 0 if processor i does a bus transaction		SL is 1 if processor i does a bus transaction	
Current state in processor i	Read by processor i	Write by processor i	Read by processor i	Write by processor i
DE				
CS				
NIC				

Part B [6 points]

Fill out the following state transition table for the cache of processor i showing the next state for a block in the cache of processor i and the action 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, DE or NIC (Not In Cache – i.e., a cache miss)

Action =
 PULLSLL1 : Pull SL to 1
 UPDL : Update block in cache i (i.e., one's own cache)
 PROVL : Provide an updated block in response to a BR (and main memory is also updated as part of this action)
 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 processor i	Bus Read by processor j	Bus Update by processor j
DE		
CS		
NIC		

Solution:
[Part A]

	SL is 0 if proc <i>i</i> does a bus trans.		SL is 1 if proc <i>i</i> does a bus trans.	
Current state in proc <i>i</i>	Processor Read by proc <i>i</i>	Processor Write by proc <i>i</i>	Processor Read by proc <i>i</i>	Processor Write by proc <i>i</i>
DE	DE/NT	DE/NT	DE/NT	DE/NT
CS	CS/NT	DE/BU (also accept NA)	CS/NT	CS/BU
NIC	DE/BR	DE/BR, BU	CS/BR	CS/BR, BU

Grading: 1 point for each entry. If only the next state or the bus transaction is right, 0.5 points.

[Part B]

State in proc <i>i</i>	Bus Read by proc <i>j</i>	Bus Update by proc <i>j</i>
DE	CS/PULLSL1, PROVL	Only possible with Bus Read by proc <i>j</i> , actions for Bus Read given in previous entry
CS	CS/PULLSL1	CS/PULLSL1, UPDL
NIC	NIC/NA	NIC/NA

Grading: 1 point for each entry. If only the next state or the bus transaction is right, 0.5 points.