Name _____(answer key)_____

Computer Architecture and Engineering
CS152 Quiz #5
April 27th, 2011
Professor Krste Asanović

Name:___<ANSWER KEY>__

This is a closed book, closed notes exam.
80 Minutes
16 Pages

Notes:

- Not all questions are of equal difficulty, so look over the entire exam and budget your time carefully.
- Please carefully state any assumptions you make.
- Please write your name on every page in the quiz.
- You must not discuss a quiz's contents with other students who have not yet taken the quiz. If you have inadvertently been exposed to a quiz prior to taking it, you must tell the instructor or TA.
- You will get **no** credit for selecting multiple-choice answers without giving explanations if the instruction ask you to explain your choice.

Writing name on each sheet	_____	2 Points
Question 1	_____	30 Points
Question 2	_____	20 Points
Question 3	_____	28 Points
TOTAL	**_____**	**80 Points**

Name _____(answer key)_____

Question 1: Locking Performance (30 points)

While analyzing some code, you find that a big performance bottleneck involves many threads trying to acquire a single lock.

Conceptually, the code is as follows:

```
int mutex = 0;

while( true )
{
  noncritical_code( );

  lock( &mutex );
  critical_code( );
  unlock( &mutex );
}
```

Assume for all questions that our processor is using a directory protocol, as described in Homework #5 (also found in **Appendix A**).

Name _____(answer key)_____

Test&Set Implementation

First, we will use the atomic instruction `test&set` to implement the *lock(mutex)* and *unlock (mutex)* functions.

In C, the instruction has the following function prototype:

```
int return_value = test&set(int* maddr);
```

Recall that `test&set` atomically reads the memory address *maddr* and writes a 1 to the location, returning the original value.

Using `test&set`, we arrive at the following first-draft implementation for the *lock()* and *unlock ()* functions:

```
void inline lock(int* mutex_ptr)
{
   while(test&set(mutex_ptr) == 1);
}

void inline unlock(int* mutex_ptr)
{
   *mutex_ptr = 0;
}
```

Q1.A: Test&Set, The Initial Acquire (5 points)

Let us analyze the behavior of `Test&Set` while running 1,000 threads on a 1,000 cores.

Consider the following scenario: At the start of the program, the lock is invalid in all caches. Then, every thread executes `Test&Set` once. The first thread wins the lock, while the other threads will find that the lock is taken. How many invalidation messages must be sent when all 1,000 threads execute `Test&Set` once?

1,000 Test&Sets are performed in the above scenario.

Test&Set is an atomic read-write operation and requires exclusive access to the lock's address. Therefore, each Test&Set invalidates the previous core who performed Test&Set. However, the first core had no one to invalidate, because the lock was initially uncached. Therefore, 999 invalidation messages were sent.

-1 points for off-by-one errors.

Invalidations ___**999**__

Name _____(answer key)_____

Q1.B: Test&Set, Spinning (5 points)

While the first thread is in the critical section (the "winning thread"), the remaining threads continue to execute `Test&Set`, attempting to acquire the lock. Each waiting thread is able to execute `Test&Set` five times before the winning thread frees the lock. How many invalidation messages must be sent while the winning thread was executing the critical section?

999 cores are spinning, each executes T&S five times for a total of 4995 Test&Sets performed.

Each Test&Set invalidates the previous core who performed Test&Set. Therefore, 4995 invalidation messages were sent.

(This assumes that every thread is interleaved).

-1 point for calculating for all 1000 threads spinning.

Invalidations ___**4995**__

Q1.C: Test&Set, Freeing the Lock (5 points)

How many invalidation messages must be sent when the winning thread frees the lock? Assume the critical section is very long, and all 999 other threads have been waiting to acquire the lock.

Freeing the lock involves writing to the lock's address which requires invalidating anybody else who has cached that address. Because all of the other cores are spinning on Test&Set, and only one core will have the lock address at a time, the winning lock will invalidate only the last core to perform a Test&Set.

Invalidations ___**1**__

Test&Test&Set Implementation

Since our analysis from the previous parts show that a lot of invalidation messages must be sent while waiting for the lock to be freed, let us instead use the atomic instruction `test&set` to implement Test&Test&Set.

```
void inline lock(int* mutex_ptr)
{
   while( (*mutex_ptr == 1) || test&set(mutex_ptr) == 1);
}

void inline unlock(int* mutex_ptr)
{
   *mutex_ptr = 0;
}
```

(*Note:* the loop evaluation is short-circuited if the first part is true; thus, `test&set` is only executed if (*mutex_ptr) does not equal 1).

Q1.D: Test&Test&Set, The Initial Acquire (5 points)

Let us analyze the behavior of `Test&Test&Set` while running 1,000 threads on a 1,000 cores.

Consider the following scenario: At the start of the program, the lock is invalid in all caches. Then every thread performs the first `Test` (reading *mutex_ptr*) once. After every thread has performed the first `Test` (which evaluates to *False*, because *mutex* == 0), each thread then executes the atomic `Test&Set` once. Naturally, only one thread wins the lock. How many invalidation messages must be sent in this scenario?

1,000 cores perform the first Test. That requires read permissions and invalidates nobody (since the lock is initially invalid). All 1,000 cores end up with a copy of the lock.

Then, all cores execute T&S. The *first* T&S will invalidate the other 999 cores' copy, for 999 invalidations.

The other 999 T&S's will invalidate the previous core to perform T&S, for 999 more invalidations. In total 999+999 invalidations occur.

Invalidations ___**1998**__

Q1.E: Test&Test&Set, Spinning (5 points)

While the first thread is in the critical section, the remaining threads continue to execute `Test&Test&Set`. Each waiting thread is able to execute `Test&Test&Set` five times before the winning thread frees the lock. How many invalidation messages must be sent while the winning thread was executing the critical section?

Once the lock has been grabbed by the winning core, the other 999 threads will only see *mutex == 1*, and not execute the Test&Set. Therefore, executing the 4995 Test&Test&Sets while waiting for the lock to be freed only requires read permissions.

However, the very *first* Test&Test&Set will require downgrading the last core who performed a Test&Set operation to the *Shared* state, so it could be argued that 1 invalidation message was sent (technically, a *WriteBackRequest* message). So 0 invalidations occurred and 1 downgrade occurred. Either 0 or 1 would be acceptable answers.

Invalidations ___**0/1**__

Q1.F: Test&Test&Set, Freeing the Lock (5 points)

How many invalidation messages must be sent when the winning thread frees the lock for the `Test&Test&Set` implementation? Assume the critical section is very long, and all 999 other threads have been waiting to acquire the lock.

Freeing the lock will require invalidating the 999 shared copies held by the spinning threads.

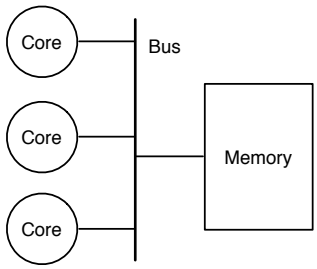
-1 for off-by-one errors.

Invalidations ___**999**__

Name _____(answer key)_____

Question 2: Sequential Consistency (20 points)

For this question, we consider the implementation of sequential consistency (SC) in a multi-processor system. Each processor is a simple single-issue, in-order core. Every core is connected to a bus which is directly connected to memory (as shown below).



Assume that the bus can only service a single memory operation every cycle. If multiple cores contend for the bus, the memory operation from a randomly selected requesting core is chosen to proceed. All remaining cores trying to make a bus request will stall their pipeline until the bus can service their memory request.

Q2.A: Baseline (5 points)

Our baseline memory system has no caches; instead, every memory operation is sent directly to main memory, where it is visible to all processors simultaneously.

Does this system provide sequential consistency? Explain why or why not. If it does *not* provide SC, provide a simple code segment that shows how to break SC.

Yes, this is SC.

This exactly fits the definitive example of SC. All cores execute memory operations in-order, and memory randomly picks one memory operation to execute atomically. Once the memory operation is chosen by the bus/memory, it is now globally visible to all other cores. Therefore, a valid in-order interleaving will be visible, and the *same* interleaving will be visible to all cores.

Q2.B: Adding Write Buffers (5 points)

We find that we are losing a lot of performance waiting for the bus, so it is proposed that we add a write buffer (WB) to each processor. When a store occurs, it is inserted into the write buffer and the CPU can immediately continue.

Stores still occur in-order out of a given write buffer. Also, loads must wait for the WB to drain before executing (*note:* stores are not globally visible until they *leave* the WB and are accepted by the memory system).

Does this system provide sequential consistency? Explain why or why not. If it does *not* provide SC, provide a simple code segment that shows how to break SC.

Yes, this is SC.

Memory operations are performed in-order, and the results of the stores are not visible to anyone until they leave the write buffer (even to the core that issued the store).

Q2.C: Optimizing Loads (5 points)

Waiting for the store buffer to drain before executing loads is terrible for performance. Instead, we propose that loads are allowed to execute even if the WB still has stores in it. However, if the load reads the *same* address as a store in the WB, then the load (and all following instructions) stall until the WB drains.

Does this system provide sequential consistency? Explain why or why not. If it does *not* provide SC, provide a simple code segment that shows how to break SC.

Yes, this is SC.

Only loads that do not match stores in the WB can run ahead. Therefore, even though they are running ahead of the completion of some stores, they do not return a value that is not already visible to other cores.

Also, one could argue that without caches, loads will still be performed in-order, behind the stores coming out of the WB. In this case, loads can't really "run ahead" if the WB has stores still in it, so this question degrades to Q2.B. However, it would be higher performance to let loads get priority over the stores in the WB, because loads are on the critical path in the code.

Q2.D: Bypassing Loads (5 points)

Let us now examine an alternative proposal for optimizing load performance. For a lot of the code we are interested in, many of the loads match with the stores in the WB.

Therefore, we propose to not allow loads to leave the processor until the WB has drained, but we will allow loads to be bypassed from the WB if the addresses match.

Does this system provide sequential consistency? Explain why or why not. If it does *not* provide SC, provide a simple code segment that shows how to break SC.

No, this is *not* SC.

Stores still in the WB are not globally visible to the other cores. Therefore, bypassing loads out of this WB to the issuing processor will mean that different cores see a different record of the interleaving of memory operations.

initially X=0, Y=0

Core0	Core1
ST X=1	ST Y=1
LD Y	LD X
LD X	LD Y

Core 0 will get x,y = (1,0), because it bypasses the X=1 store out of the write buffer, but core 1's ST Y isn't yet visible to it.

Likewise, Core 1 will get x,y=(0,1), because it bypasses the Y=1 store out of its write buffer, but core 0's ST X isn't yet visible to it!

So Core 0 thinks that Store X occurred before Store Y, but Core 1 thinks Store Y occurred before Store X.

Thus, not SC.

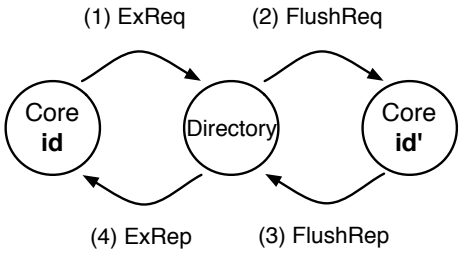
Name _____(answer key)_____

Question 3: Directory Protocols (28 points)

For this question we will optimize the directory protocol used in Homework #5 (found in **Appendix A**).

Directory protocol performance can suffer from the latency of requesting and obtaining permission. For the directory protocol used in Homework #5, an **exclusive request** (write operation) for a memory line already in the exclusive state **W**(id') requires **four** messages:

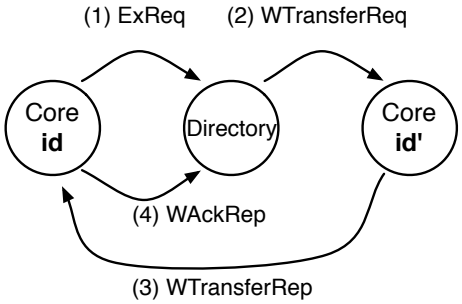
1) An *exclusive request (ExReq)* is sent to the directory from the requesting cache ***id***.
2) A *flush request (FlushReq)* is sent to the current user ***id'*** of the cache line.
3) A *flush response (FlushRep)* is sent back to the directory from the (former) user ***id'***.
4) An *exclusive response (ExRep)* is sent from the directory to the requesting cache ***id***.



In this problem, we will look at adding a new **three message** system to optimize the above situation, ***an exclusive request to a memory line that is in the W(id') state.*** *Note*: we are ***not*** optimizing share requests, and we are ***not*** optimizing exclusive requests to shared memory lines.

Under the new three message proposal, an *exclusive request* to a memory line already in the **W(**id') state behaves as follows:

1) An *exclusive request (ExReq)* message is sent to the directory from cache ***id***.
2) A *write-transfer request (WTransferReq)* is sent from the directory to the current exclusive owner of the memory line (***id'***).
3) A *write-transfer response (WTransferRep)*, holding the data payload, is given to the requesting core (***id***) by the now former exclusive owner (***id'***) of the memory line (this is a cache-to-cache message).
4) The new owner (***id***) sends a *write-transfer acknowledgement reply (WAckRep)* to the directory once it receives the requested memory line from ***id'***.



(*Note*: yes, four messages are still being sent, but the messaging critical path between a cache requesting write permission and receiving the write permission is now only three messages. The fourth message is just for the directory to know the handoff between caches completed).

We will add three new types of messages, highlighted in the following table:

Category	Messages
Cache to Memory Requests	`ShReq, ExReq`
Memory to Cache Requests	`WbReq, InvReq, FlushReq,` **`WTransferReq`**
Cache to Memory Responses	`WbRep(v), InvRep, FlushRep(v),` **`WAckRep`**
Memory to Cache Responses	`ShRep(v), ExRep(v)`
Cache to Cache Responses	**`WTransferRep`**

- memory to cache requests: **WTransferReq**
    - The receiving core (***id'***) will be invalidated, and send its copy of the data to another core (using a *WTransferRep* message).
- cache to cache responses: **WTransferRep**
    - The owner (***id'***) will send its copy of the memory line directly to another cache using this message.
- cache to memory responses: **WAckRep**
    - The new owner (***id***) who requested exclusive access will acknowledge that he has received the data from core ***id'*** by sending a *WAckRep* back to the directory.

Name _____(answer key)_____

Q3.A: Cache State Transitions (16 points)

Fill in the following table regarding cache state transitions to express this new protocol. Notice that no new cache states are required to handle this new optimization. However, the cache state protocol must now handle the new message types.

No.	Current State	Handling Message	Next State	Dequeue Message?	Action
1	C-nothing	Load	C-pending	No	ShReq(id,Home,a)
2	C-nothing	Store	C-pending	No	ExReq(id,Home,a)
3	C-nothing	WbReq(a)	C-nothing	Yes	None
4	C-nothing	FlushReq(a)	C-nothing	Yes	None
5	C-nothing	InvReq(a)	C-nothing	Yes	None
6	C-nothing	ShRep (a)	C-shared	Yes	updates cache with prefetch data
7	C-nothing	ExRep (a)	C-exclusive	Yes	updates cache with data
7b	**C-nothing**	**WTransferReq**	**C-nothing**	**Yes**	**None**
7c	**C-nothing**	**WTransferRep**	**C-exclusive**	**Yes**	**WAckRep(id, Home, a)**
8	C-shared	Load	C-shared	Yes	Reads cache
9	C-shared	WbReq(a)	C-shared	Yes	None
10	C-shared	FlushReq(a)	C-nothing	Yes	InvRep(id, Home, a)
11	C-shared	InvReq(a)	C-nothing	Yes	InvRep(id, Home, a)
12	C-shared	ExRep(a)	C-exclusive	Yes	None
13	C-shared	(Voluntary Invalidate)	C-nothing	N/A	InvRep(id, Home, a)
13b	**C-shared**	**WTransferReq**	**C-nothing**	**Yes**	**InvRep(id, Home, a)**

voluntarily flushed this line previously, but directory doesn't know yet

Prefetcher demanded exclusive permission for this line

voluntarily wrote back this line previously, but directory doesn't know yet....

Table 3.A.I: Cache State Transitions for Cache Part I

Q3.A: Continued...

14	C-exclusive	Load	C-exclusive	Yes	reads cache
15	C-exclusive	Store	C-exclusive	Yes	writes cache
16	C-exclusive	WbReq(a)	C-shared	Yes	WbRep(id, Home, data(a))
17	C-exclusive	FlushReq(a)	C-nothing	Yes	FlushRep(id, Home, data(a))
18	C-exclusive	(Voluntary Writeback)	C-shared	N/A	WbRep(id, Home, data(a))
19	C-exclusive	(Voluntary Flush)	C-nothing	N/A	FlushRep(id, Home, data(a))
19b	**C-exclusive**	**WTransferReq**	**C-nothing**	**Yes**	**WTransferReq(id, new_owner, data(a))**
20	C-pending	WbReq(a)	C-pending	Yes	None
21	C-pending	FlushReq(a)	C-pending	Yes	None
22	C-pending	InvReq(a)	C-pending	Yes	None
23	C-pending	ShRep(a)	C-shared	Yes	updates cache with data
24	C-pending	ExRep(a)	C-exclusive	Yes	update cache with data
24b	**C-pending**	**WTransferReq**	**C-pending**	**Yes**	**None**
24c	**C-pending**	**WTransferRep**	**C-exclusive**	**Yes**	**WAckRep(id, Home, a)**

Table 3.A.II: Cache State Transitions Part II

Q3.B: Home Directory State Transitions (12 points)

Now let us consider the transitions for the home directory. From the home directory's point of view, the following actions take place:

1) The home directory receives an *exclusive request (ExReq)*.
2) If the memory line is in the *W(id')* state, the home directory issues a *WTransferReq* to the current owner (***id'***).
3) The memory line is put in the *TW(id)* state.
4) The home directory then waits for a *WAckRep* from the requesting core (***id***) to confirm the transfer completed.

Fill in the entries in the following table to implement this behavior. The rest of the protocol will behave as before (*hint:* not all of the entries in the protocol will change).

Note: not all entries are shown that are required to handle all corner cases. Only focus on filling out the entries provided.

Name _____(answer key)_____

No.	Current State	Message Received	Next State	Dequeue Message?	Action
1	R(dir) & (dir = ε)	ShReq(a)	R({id})	Yes	ShRep(Home, id, data(a))
2	R(dir) & (dir = ε)	ExReq(a)	**W(id)**	**Yes**	**ExRep(Home,id,data(a))**
3	R(dir) & (dir = ε)	(Voluntary Prefetch)	R({id})	N/A	ShRep(Home, id, data(a))
4	R(dir) & (id ∉ dir) & (dir ≠ ε)	ShReq(a)	R(dir + {id})	Yes	ShRep(Home, id, data(a))
5	R(dir) & (id ∉ dir) & (dir ≠ ε)	ExReq(a)	**Tr(id)**	**No**	**InvReq(Home,dir,a)**
6	R(dir) & (id ∉ dir) & (dir ≠ ε)	(Voluntary Prefetch)	R(dir + {id})	N/A	ShRep(Home, id, data(a))
7	R(dir) & (dir = {id})	ShReq(a)	R(dir)	Yes	None
8	R(dir) & (dir = {id})	ExReq(a)	**W(id)**	**Yes**	**ExRep(Home,id,data(a))**
9	R(dir) & (dir = {id})	InvRep(a)	R(ε)	Yes	None
10	R(dir) & (id ∈ dir) & (dir ≠ {id})	ShReq(a)	R(dir)	Yes	None
11	R(dir) & (id ∈ dir) & (dir ≠ {id})	ExReq(a)	**Tr(dir -{id})**	**No**	**InvReq(Home,dir-{id},a)**
12	R(dir) & (id ∈ dir) & (dir ≠ {id})	InvRep(a)	R(dir - {id})	Yes	None
13	W(id')	ShReq(a)	Tw(id')	No	WbReq(Home, id', a)
14	W(id')	ExReq(a)	**Tw(id)**	**Yes**	**WTransferReq(home,id,a)**
15	W(id)	ExReq(a)	**W(id)**	**Yes**	**None**
16	W(id)	WbRep(a)	**R({id})**	**Yes**	**data->mem**
17	W(id)	FlushRep(a)	**R(e)**	**Yes**	**data->mem**
18	Tr(dir) & (id ∈ dir)	InvRep(a)	Tr(dir - {id})	Yes	None
19	Tr(dir) & (id ∉ dir)	InvRep(a)	Tr(dir)	Yes	None
22	**Tw(id)**	**WackRep(a)**	**W(id)**	**Yes**	**None**

The whole point of the optimization...the real magic here! Telling the past owner to transfer the data to the new guy "id".

This is the directory being told the transfer finished

Table 3.B: Home Directory State Transitions, Messages sent from site **id**

END OF QUIZ