Name _____

Computer Architecture and Engineering
CS152 Quiz #5
April 27th, 2012
Professor Krste Asanović

Name:_____

This is a closed book, closed notes exam.
80 Minutes
22 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	_____	1 Points
Question 1	_____	12 Points
Question 2	_____	21 Points
Question 3	_____	25 Points
Question 4	_____	21 Points
TOTAL	**_____**	**80 Points**

Name _____

Question 1: The Fourth C of Cache Misses (12 points)

In Unit 2 we talked about the three C's of caches misses: *capacity misses*, *conflict misses*, and *compulsory misses*. Now in multiprocessors, we can add a fourth C: *coherence misses*. A coherence miss is a cache miss due to another core having invalidated the data in your cache.

Mark whether the following modifications to the cache parameters will cause an **increase, decrease**, or whether the modification will have **no effect** on the number of *coherence misses*. You can assume the baseline cache is set associative. **Explain your reasoning** to receive credit.

Assume that in each case the other cache parameters (number of sets, number of ways, number of bytes/line) and the rest of the machine design remain the same.

	coherence misses
increasing number of bytes per line	More false sharing, greater # of coherence misses. (Theoretically, could be less if sharing large objects - fewer misses needed to transfer data - but in practice false sharing misses are the dominating factor)
increasing number of sets	No effect to a first order However can increase slightly as greater probability of holding on to data that causes a coherence miss.
increasing number of ways	Same as above

Name _____

Question 2: Memory Fences (21 points)

We are interested in implementing a parallel "branch and bound" algorithm, in which each core attempts to find the shortest path between two nodes in a graph. A critical component of the algorithm is the *bound variable*, aka, the "cost" of the best path found so far by any core. If the current path a thread is exploring costs more than the bound, the current thread knows that the current path can not be the shortest path, and aborts exploring the path further and instead tries exploring a new path.

The bound variable is globally visible to all threads. It is read by all threads to compare their current paths to the bound, but it is only written to when a thread finds a newer, better path.

Conceptually, the code for updating the bound variable is as follows:

```
int volatile bound = 0;
int volatile lock = 0;

// updating the bound variable
acquire_lock(&lock);

if (new_bound < bound)
   bound = new_bound;

release_lock(&lock);
```

Here is an example piece of code that reads the bound variable:

```
if (my_current_bound > bound)
  abort_and_try_a_new_path(); // current_path is too long!
else
  continue_extending_current_path();
```

Name _____

Q2.A: Shared Variable, Using Atomic Ops (10 points)

Assembly-code implementations of these accesses to the bound variable are shown below. These run correctly on a processor with sequential consistency, however, they may not be correct when running on a processor with a fully relaxed memory model, such as the RISC-V Rocket core from Lab 5.

Insert the appropriate memory fence(s) below to insure correctness on a relaxed memory model ($MEMBAR_{LL}$, $MEMBAR_{LS}$, $MEMBAR_{SL}$, $MEMBAR_{SS}$). For example, $MEMBAR_{SL}$ forces all stores before the memory barrier to complete and be visible to all cores in the system before allowing any new loads to be issued. You will be graded both on correctness and efficiency. *You must also note if no memory fences are required in the checkbox.*

LOCK and BOUND are memory addresses that point to the respective variables' locations in memory. For this part, we are using *fetch_and_or* (abbreviated as FAO) to access the LOCK variable.

```
# Update bound.
try:

  ADDI  x1, x0, 1

  FAO x2, LOCK, x1

  BNEZ x2, try
    MEMBAR_SL
  LD x3, BOUND

  BGTE new_bound, x3, done

  ST new_bound, BOUND

done:
   MEMBAR_SS
  ST zero, LOCK
```

☐ no memory fences required

```
# Read bound.

  LD x1, MY_BOUND

  LD x2, BOUND

  BGT x1, x2, return

  JMP CONTINUE_PATH_FUNC

return:

  # start a new path
```

☐ √ no memory fences required

The first fence is to prevent reading a stale copy of bound (due to the branch predictor running ahead). The second memory fence is to prevent an other core seeing the lock getting freed before we have made visible to them the new value of the bound variable.

Q2.B: Shared Variable, Using Dekker's Algorithm (11 points)

Now, let us remove the atomic *fetch_and_or* instruction, and instead use Dekker's Algorithm to implement the lock using regular loads and stores (we are only considering a dual-core system).

In C, the code for updating the bound variable for Core 0 is as follows:

```
// On Core 0
lock0 = 1;
turn = core_id;

while (lock1 & (turn == core_id))
{
   ;//do nothing
}

if (new_bound < bound)
   bound = new_bound;

lock0 = 0;
```

"core_id" is a register that holds the core's ID number (either 0 or 1).

Name _____

Below we show code for both core 0 and core 1 when updating the *bound variable* (the code for each differs slightly). However, we only show one copy of the code for reading the *bound variable*. Again, add in the appropriate memory fence(s) ($MEMBAR_{LL}$, $MEMBAR_{LS}$, $MEMBAR_{SL}$, $MEMBAR_{SS}$) to ensure correct performance on a processor using a relaxed memory model. You will be graded both on correctness and efficiency.

```
Core 0
# updating the bounds
# variable

  ADDI x1, x0, 1

  ST x1, LOCK0
  MEMBAR_SS
  ST core_id, TURN

try:

  LD x2, LOCK1

  LD x3, TURN

  BNE x3, core_id, try

  BNEZ x2, try
  MEMBAR_LL
  LD x3, BOUND

  BGTE new_bound, x3, done

  ST new_bound, BOUND

done:
  MEMBAR_SS
  ST zero, LOCK0
```

☐ no memory fences required

```
Core 1
# updating the bounds
# variable

  ADDI x1, x0, 1

  ST x1, LOCK1
  MEMBAR_SS
  ST core_id, TURN

try:

  LD x2, LOCK0

  LD x3, TURN

  BNE x3, core_id, try

  BNEZ x2, try
  MEMBAR_LL
  LD x3, BOUND

  BGTE new_bound, x3, done

  ST new_bound, BOUND

done:
  MEMBAR_SS
  st zero, LOCK1
```

☐ no memory fences required

Name _____

Insert the appropriate memory fence(s) below for when either core reads the bound variable.

```
# Read bound.

  LD x1, MY_BOUND

  LD x2, BOUND

  BGT x1, x2, abort

  JMP CONTINUE_PATH_FUNC

abort:

  # start a new path
  JMP START_NEW_PATH_FUNC
```

☑ no memory fences required

Reading the bound doesn't need any fences (again).

Once again, we need a fence before freeing the lock (so the updated value is made visible to everybody first), and we need a fence after the spin/try loop to make sure we don't get a stale copy.

There is a new fence needed: a MEMBAR_SS to enforce the ordering of writing to the lock and turn variables. If the turn update is made visible before lock* update, then it is possible for both cores to enter the critical section!

Also, there is an unfortunate bug in the assembly for the while loop: as written, it behaves as an OR condition that short circuits, instead of an AND condition.

Name _____

Question 3: Scaling Directory Protocols (25 points)

In this question we will discuss implementing cache-coherence protocols that scale well to thousands of cores.

Q3.A: Motivation (3 points)

As discussed in class, directory protocols scale to higher core counts better than snoopy protocols. Why?

Snoopy protocols rely on cheap, global broadcasts to all cores.

Q3.B: Full-map Directory Overhead (4 points)

The directory protocol discussed in Lecture 19 (and found in Appendix A) describes a "full-map" directory protocol, in which the directory contains a pointer to every cache. For a 1024-core processor, and where each cache line is 64 bytes, how many directory state bits are required to track which caches are sharing a given line of memory? What is the ratio of directory state bits to data bits?

A bit-vector will do for tracking which cores are in the sharers list (only one bit required for "does this cache have it or not?").

1024 bits -> 128 bytes of overhead
128 B : 64 B, or 2:1

Directory Bits: __1024__
Overhead Ratio: __ 2:1__

Name _____

Q3.C: Limited-map Directory Overhead (4 points)

As you have shown in the previous question Q3.B, a full-map directory can require a large amount of state to track every potential sharer. Instead, let us consider using a "limited-map" directory, in which only a limited number *n* cores may cache a given memory line. Our proposal is to allow up to 8 cores to cache a given line. If a 9th core wants to read the line, the directory must first invalidate one of the 8 sharers to make room for the new request.

How many bits are required for the directory to track up to 8 sharers in a 1024-core processor? What is the ratio of directory bits to memory bits when the memory line is 64 bytes in size?

It takes log(1024) bits, or 10 bits, to hold the ID number of a given cache, and an 11th bit to mark whether or not that cache is a sharer (if the directory state is in the "shared" state, how do you know which subset of the cache pointers are actually valid?).

11 bits * 8 sharers = 88 bits.

-1/2 point for "80 bits" and "5:32", since one could argue over the ambiguity of "directory bits" only referring to the bits required to track which caches are sharers, but in reality, you need the valid bits to distinguish who really is a sharer.

88 bits is 11 bytes, so 11:64 ratio.

(it is also possible to come up with schemes that use less than 1 valid bit per sharer, but the answer is strictly greater than 80 bits).

Directory Bits: __88__
Overhead Ratio: __11:64__

Name _____

Q3.D: Performance: Many Readers (4 points)

Now we will compare the performance of the different directory schemes.
How many invalidations must a full-map and a limited-map (with 8 max sharers) perform after *all 1024 cores* have read the same shared variable into their cache? The corresponding memory line is initially in R(ε) state in the directory.

```
# each core
LD x1, SHARED_VARIABLE
```

Put your answers in Table 3-1 below, and explain your reasoning.

protocol	Number of Invalidations
full-map	**0**
limited-map (8 max)	**1016**

Table 3-1: Many Readers

Name _____

Q3.E: Performance: Many Readers, One Writer (4 points)

How many invalidations must the two protocols perform when *all 1024 cores* load the same shared variable into their cache, and then *one* core writes to the shared variable?

```
# each core
LD x1, SHARED_VARIABLE

... passage of time ...

# one core
BNE zero, core_id, done
ST x2, SHARED_VARIABLE
```

Put your answers in Table 3-2 below, and explain your reasoning.

protocol	Number of Invalidations
full-map	**1023**
limited-map (8 max)	**1016 + 7 = 1023**

Table 3-2: Many Readers, One Writer

Full credit given for 'full-map: 1024 invalidations", since one could argue based on our broken protocol in Appendix A that the requesting core (which is already in the shared state) must send out his own invalidation, to then move up to c-none to c-exclusive (since there is no c-shared -> c-exclusive line in the protocol!). In a more perfect world, only 1023 invalidations are required.

Name _____

A Third Protocol

In your answers to the previous questions, you have shown that full-map protocols may be infeasible due to their large overhead, but that limited directories may require too many invalidations for certain use-cases.

We will now describe a **third** directory protocol: a limited-map directory with a "globally shared" bit, called the *limited-map broadcast* protocol. If more than 8 sharers are sharing a memory line, the "globally shared" bit is set. If the directory needs to send an invalidate to the sharers, it must broadcast invalidations to *every core in the system* if the "globally shared" bit is set (as there is no way to know exactly which cores have the data). If the "globally shared" bit is not set, then it needs to only send invalidations to the sharers in its directory. Thus, we gain the benefit of having a low overhead protocol *and* can allow many caches to share a single cache line simultaneously.

Q3.F: Choose the Best Protocol: Part 1 (3 points)

Given the option of selecting between a full-map directory, the limited-map directory, and the limited-map broadcast directory, which protocol would you recommend for the following use cases? *Circle your answer below, and explain your choices.*

Your TA Chris wants to run Blackscholes, a financial derivatives modeling algorithm. Each core is given a different data point to calculate, independent of all other data (i.e., it is embarrassingly parallel). There are no explicitly shared data structures. *However*, the model involves calls to math functions `exp()` and `pow()`, which are implemented using look-up tables. Thus, a core executing a `pow()` function must read a table for the result. These tables are located in a statically allocated memory data structure that is visible to all cores.

There will be many readers (everybody) but no writers, so we want a protocol that allows lots of sharers, and we do not care about the overhead of invalidations caused by write requests.

0 points for limited-map
1 point for full-map, which is correct, but hugely inefficient.

Full-map Limited-map (Limited-map Broadcast)

Name _____

Q3.G: Choose the Best Protocol: Part 2 (3 points)

Your TA Chris now wants to compute the shortest distance between two nodes on a map (e.g., computing driving directions). This is accomplished through an algorithm in which each core picks its own path to try out. Work is put into a central "work queue" data structure (in this case, paths that haven't been completed yet). When a core is not busy, it reads the work queue and pulls a task off of it. When it is finished with its task, it will write more work to the work queue. Cores are constantly adding new work to the work queue, and removing work from the work queue (aka, lots of reads and writes to the same memory locations). Circle which protocol you think best matches this application, and explain your answer.

Because we expect writes to occur often, we can expect that few readers will be required in between writes (actually, only one person will probably read the data structure before writing to it!). Thus, we do not mind having a protocol that can only handle having a few sharers.

From this point of view, the intended answer is "limited-map", but limited-map broadcast is also valid, as it will share the same performance characteristics as limited-map when there are very few readers.

1 point for full-map, because it would be hugely inefficient.

Full-map

Limited-map

Limited-map Broadcast

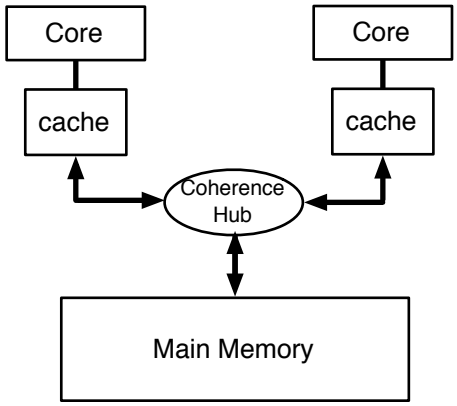
Name _____

Question 4: Implementing Load-reserve, Store-conditional In a Snoopy Processor (21 points)

For this question, we will try to implement the instructions *load-reserve* and *store-conditional*. Described in Lecture 17, the load-reserve loads a value from memory into the cache, and also holds a reservation flag. When performing the store-conditional, the store completes successfully if the core still holds the reservation flag. The store-conditional then invalidates all other reservation flags in the system. If the store-conditional is executed and the core no longer holds its reservation, the store is not performed and a status flag is set noting *failure*. At least, that is the **conceptual** description. But for this question, we will propose an actual implementation!

For this entire question, the base-line processor is using the MOESI cache coherence protocol, as shown in Appendix B (a copy of the Handout #7 used for PSet #5). Also, the cache is direct mapped.

The processor in this question uses a *logical bus*, meaning that while the cores and memory are connected together and all actions are broadcast to all agents on the bus, it is *physically* implemented a bit differently.



(This is the same set-up as the dual-core Rocket processor from Lab 5).

On a store memory operation, the core first checks the cache to see if it has the line in the OE or CE state. If not, the cache must then send out a CRI or CI message to request exclusive access. The coherence hub will broadcast the CRI or CI message to all of the other caches (called a "probe"). The caches will acknowledge back that they either do not have the line or that they have now invalidated the line. The coherence hub will now send the request to memory. Some time later, memory will respond with the data and give the requesting core the exclusive access it had requested.

This behaves exactly like the bus as described in lecture: the only difference is that, unlike the bus, operations through the hub are *not* atomic.

Q4:A: The First Proposal (5 points)

Our first proposal is the following:

> **Load-reserve** is performed just like a regular load, except that it demands *exclusive* access instead of *shared* access. If the load misses in the cache, the cache issues a *coherent read & invalidate (CRI)* of the line (requests write permission), and brings the data into its cache in the c*lean exclusive (CE)* state (or if the load hits in the cache, but only has *shared* access, it sends out a *coherent invalidate (CI)* to get *exclusive* access).
>
> **Store-conditional** first checks the cache to see if we have the data in the *clean exclusive (CE)* state or the *owned exclusive (OE)* state. If we do, the store is performed and returns a *success* status. If the data is in any other state, then the store is not performed and a *failure* status is returned.

Note that we are not explicitly storing any reservation flags anywhere under this implementation; we are relying on the existing cache coherence infrastructure to help us detect if a store-conditional has occurred somewhere else in the system between our own execution of load-reserve and store-conditional.

Given this proposal, what problem(s) might we see in a system experiencing high contention for a critical section protected with these instructions?

Live lock.

We cant make forward progress unless our data line is in the exclusive state: but if somebody else tries to load-reserve the same data line we lose the line, and our store-conditional will fail. This cycle can continue under high contention, where everybody is trying to load-reserve the line, but before they perform the store-conditional, somebody else has already stepped in and stolen the line away.

Ideally, store-conditional should let *one* cache go ahead, and invalidate the others. The problem here is that we are literally removing a previous core's reservation when a new core comes in and performs its own load-reserve.

Name _____

Q4:B: A Fix for the First Proposal (4 points)

Because the first proposal from Q4.A may not work in a high-contention environment, here is a second proposal:

> **Load-reserve** is performed just like a regular load. If the load misses in the cache, the cache issues a *coherent read (CR)* of the line (requests *read* permission), and brings the data into its cache in either the c*lean exclusive (CE)* state or the *clean shared (CS)* state.
>
> **Store-conditional** first checks the cache to see if we still have the data. 3 possible scenarios may occur:
>
> - 1) if the data is in *clean exclusive (CE)* or *owned exclusive (OE)* state, the store completes successfully, and a *success* status is returned (just like in Q4.A).
>
> - 2) if the data is in the *clean shared (CS)* or *owned shared (OS)* state, the core must *first* broadcast on the bus a ***CI*** message to request exclusive access to the line. The core will *eventually* be given a copy of the data with exclusive access (CE or OE), at which point the store may proceed and a *success* status is returned.
>
> - 3) If the data is in the *invalid* state, the store is not performed and a *failure* status is returned (our reservation must have been cleared by another store).

Unfortunately, there is a new bug that we have introduced! *How can this go wrong?* (Hint: make sure you understand that in this system actions on the bus are not atomic, as described on Page 14).

Race.

The problem is *two* cores can hold the line in the shared state, and thus think their store-conditional has succeeded! Sure, they sent out a CI message on the bus, and will *eventually* receive exclusive access on which they can perform their store, but the point of store-conditional is that *only one* successfully performs the store-conditional.
In lecture, we often discuss snoopy protocols in systems that are connected to a physical bus, in which access to the bus is atomic and only one core can talk to the bus at a time. In such a system this race would probably not occur (depending on how you implement the retry mechanism), since the CI message can't be sent out by two cores simultaneously. But in this system described for this quiz (and matching Lab 5's dual-core rocket system), two CI messages can be sent out to the "coherence hub" simultaneously, and so this race does exist.

(note: I'm being a bit loose when I said "successfully performs the STC". STC is always performed and committed as an instruction, but I mean to imply "success" in that the actual store was performed and committed to memory and that a *success* flag was set).

Q4:C: A Fix for the Second Proposal (4 points)

Describe how you can fix the broken proposal in Q4.B so it works correctly.

A number of viable solutions were proposed by students. Credit was given for providing a working solution (a point was taken off if it was unwieldy or very low performance).

One possible solution for when multiple caches share a line is to only allow the OS cache to succeed when performing the STC.

Q4:D: Multi-programming Issues (4 points)

Your solution you described in Q4.C will work when a thread has full control of the processor. However, what can go wrong if we allow multiple threads to be multiplexed onto a single core? (i.e., the operating system can swap out threads).

Multiple threads multiplexed onto a single core will share the same cache!

Image 4 cores are running on a single core, and each thread performs a load-reserve. The first LDR requires fetching the data from memory, but the subsequent 3 LDRs will hit in the cache! Then, each of the 4 threads performs the STC: it will appear that no other cache has performed the LDR or a STC because it is designed with other caches in mind, and using the coherence protocol to change the state of the line when other threads touch it.

Name _____

Q4:E: Even More Issues! (4 points)

Below is code for atomically reading a shared variable in memory, protected by a lock implemented using load-reserve (LDR) and store-conditional (STC).

The "status" variable is a register that holds 0 when the STC completes with a *success* code, and 1 otherwise. Remember, each core is using a direct-mapped cache.

```
try:
  LDR  x1, LOCK
  LD   x2, SHARED_VARIABLE
  STC  x0, LOCK
  BNEZ status, try
```

When testing out the above code, you find that it gets stuck in an infinite loop! What happened?

If the LOCK and the SHARED_VARIABLE alias to the same set in the DM cache, the SHARED_VARIABLE could evict the LOCK, and thus the STC will fail every time!

END OF QUIZ

Name _____

Appendix A

Directory-based Cache Coherence Protocol *4/27/2011*

Before introducing a directory-based cache coherence protocol, we make the following assumptions about the interconnection network:

- Message passing is reliable, and free from deadlock, livelock and starvation. In other words, the transfer latency of any protocol message is finite.
- Message passing is FIFO. That is, protocol messages with the same source and destination sites are always received in the same order as that in which they were issued.

Cache states: For each cache line, there are 4 possible states:

- C-invalid (= `Nothing`): The accessed data is not resident in the cache.
- C-shared (= `Sh`): The accessed data is resident in the cache, and possibly also cached at other sites. The data in memory is valid.
- C-modified (= `Ex`): The accessed data is exclusively resident in this cache, and has been modified. Memory does not have the most up-to-date data.
- C-transient (= `Pending`): The accessed data is in a *transient* state (for example, the site has just issued a protocol request, but has not received the corresponding protocol reply).

Home directory states: For each memory block, there are 4 possible states:

- R(*dir*): The memory block is shared by the sites specified in *dir* (*dir* is a set of sites). The data in memory is valid in this state. If *dir* is empty (i.e., *dir* = ε), the memory block is not cached by any site.
- W(*id*): The memory block is exclusively cached at site *id*, and has been modified at that site. Memory does not have the most up-to-date data.
- T_R(*dir*): The memory block is in a transient state waiting for the acknowledgements to the invalidation requests that the home site has issued.
- T_W(*id*): The memory block is in a transient state waiting for a block exclusively cached at site *id* (i.e., in C-modified state) to make the memory block at the home site up-to-date.

Protocol messages: There are 10 different protocol messages, which are summarized in the following table (their meaning will become clear later).

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

Name _____

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
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)
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))
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

Table H12-1: Cache State Transitions

Name _____

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(dir)	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')	No	FlushReq(Home, id', a)
15	W(id)	ExReq(a)	W(id)	Yes	None
16	W(id)	WbRep(a)	R({id})	Yes	data -> memory
17	W(id)	FlushRep(a)	R(ε)	Yes	data -> memory
18	Tr(dir) & (id ∈ dir)	InvRep(a)	Tr(dir - {id})	Yes	None
19	Tr(dir) & (id ∉ dir)	InvRep(a)	Tr(dir)	Yes	None
20	Tw(id)	WbRep(a)	R({id})	Yes	data-> memory
21	Tw(id)	FlushRep(a)	R(ε)	Yes	data-> memory

Table H12-2: Home Directory State Transitions, Messages sent from site **id**

Name _____

Appendix B

Snoopy Cache Coherence Protocol *4/27/2011*

We introduce an invalidation coherence protocol for write-back caches similar to those employed by the SUN MBus. As in most invalidation protocols, only a single cache may *own* a modified copy of a cache line at any one time. However, in addition to allowing multiple shared copies of clean data, multiple shared copies of modified data may also exist. (Here, modified data refers to data different from memory. When multiple shared copies of modified data exist, one of the caches *owns* the current copy of the data instead of the memory.) All shared copies are invalidated any time a new modified (write) copy is created.

The MBus transactions with which we are concerned are:

- Coherent Read (**CR**): issued by a cache on a read miss to load a cache line.
- Coherent Read and Invalidate (**CRI**): issued by a cache on a write-allocate after a write miss.
- Coherent Invalidate (**CI**): issued by a cache on a write hit to a block that is in one of the shared states.
- Block Write (**WR**): issued by a cache on the write-back of a cache block.
- Coherent Write and Invalidate (**CWI**): issued by an I/O processor (DMA) on a block write (a full block at a time).

In addition to these primary bus transactions, there is:

- Cache to Cache Intervention (**CCI**): used by a cache to satisfy other caches' read transactions when appropriate. A **CCI** intervenes and overrides the answers normally supplied by memory. Data should be supplied using **CCI** whenever possible for faster response relative to the memory. However, only the cache that ***owns*** the data can respond by **CCI.**

The five possible states of a data block are:

- Invalid (**I**): Block is not present in the cache.
- Clean exclusive (**CE**): The cached data is consistent with memory, and no other cache has it.
- Owned exclusive (**OE**): The cached data is different from memory, and no other cache has it. This cache is responsible for supplying this data instead of memory when other caches request copies of this data.
- Clean shared (**CS**): The data has not been modified by the corresponding CPU since cached. Multiple **CS** copies and at most one **OS** copy of the same data could exist.
- Owned shared (**OS**): The data is different from memory. Other **CS** copies of the same data could exist. This cache is responsible for supplying this data instead of memory when other caches request copies of this data. (Note, this state can only be entered from the **OE** state.)