Name: ______________________

Computer System Architecture
6.823 Quiz #4

November 23th, 2005

Professor Arvind

Dr. Joel Emer

Name:___________________

This is a closed book, closed notes exam.

80 Minutes

15 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.

Writing name on each sheet	________	2	Points
Part A	________	14	Points
Part B	________	12	Points
Part C	________	10	Points
Part D	________	12	Points
Part E	________	18	Points
Part F	________	12	Points
TOTAL	________	**80 Points**	

Name: ______________________

Part A: Directory-based Cache Coherence Protocol (14 Points)

In this part, questions are about the directory-based cache coherence protocol presented in Handout #12 (2005). We have provided a copy of this handout in Appendix A after omitting the operations in T_R(dir) states.

Question 1. (6 Points)

Fill in the following table:

- If the condition indicated in a row is possible, circle Yes (Y) in "Possible?" column and fill out the next three columns (Next State, Dequeue Message and Action) appropriately,
- otherwise, circle No (N) and explain briefly (in one sentence).

Current State	Message Received	Possible? (Y or N)	Next State	Dequeue Message?	Action
T_R(dir) & (id $\in$ dir)	ShReq(a)	Y			
		N			
	ExReq(a)	Y			
		N			
	InvRep(a)	Y			
		N			
	WbRep(a)	Y			
		N			
	FlushRep(a)	Y			
		N			
	WbReq(a)	Y			
		N			

Name: ______________________

Question 2. (8 Points)

In Table A-1 row 7 specifies the PP behavior when the current cache is C-nothing (not C-pending) and an ExRep(a) message is received. Give a simple scenario that causes this situation.

Part B: Silent Drops (12 Points)

It is generally desirable to minimize generation of bookkeeping messages in a cache coherence protocol. Therefore, it would be good to have "silent drops" when a cache invalidates a shared cache line (C-shared).

Question 3. (6 Points)

In the ***directory-based cache coherence protocol*** in Appendix A, when a cache line wants to voluntarily invalidate a shared cache line, the PP informs the memory of this operation (see row 13 in Table A-1). Can we just drop the line silently? That is, can we voluntarily invalidate a shared cache line without sending InvRep message to the home node? If we can, explain why. If we cannot, describe a simple scenario where the system would in an erroneous state or deadlocked.

Question 4. (6 Points)

Can we silently drop a shared cache line in the ***snoopy cache coherence protocol*** described in Appendix B? As before, if we can, explain why. If we cannot, describe a simple scenario where there would be an error.

Part C: Synchronization (10 Points)

Question 5. (10 Points)

Ben Bitdiddle modifies Dekker's algorithm by inverting the predicate in the value of turn variable (named Bitdiddle's algorithm). He claims that the new algorithm is still correct.

Process 1

```
        ...
        c1=1;
        turn = 1;
L:      if c2=1 & turn=1
                then go to L
          <critical section>
          c1=0;
```

Process 2

```
        ...
        c2=1;
        turn = 2;
L:      if c1=1 & turn=2
                then go to L
          <critical section>
        c2=0;
```

Dekker's Algorithm

Process 1

```
        ...
        c1=1;
        turn = 1;
L:      if c2=1 & turn=2
                then go to L
          <critical section>
          c1=0;
```

Process 2

```
        ...
        c2=1;
        turn = 2;
L:      if c1=1 & turn=1
                then go to L
          <critical section>
        c2=0;
```

Bitdiddle's Algorithm

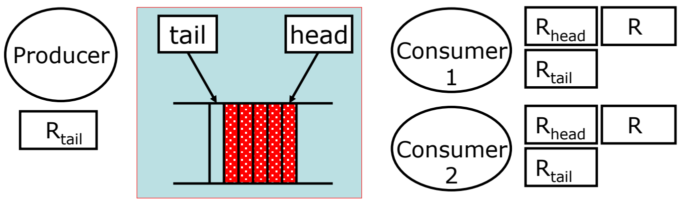
Do you agree with him? Discuss the correctness of the new algorithm from the point of view of (1) the enforcement of synchronization (i.e., only one process can enter into the critical section at any given time), (2) no deadlock, and (3) fairness.

Name: ______________________

Part D: Load-reserve/Store-conditional (12 Points)

Question 6. (12 Points)

Here are two implementations of one producer-multiple consumer example discussed in Lecture 16 (2005). One is based on an atomic read-modify-write (Test&Set) and the other on Load-reserve/Store-conditional. Assume that we implement both in a system having a snoopy bus with caches.



```
Test&Set(m, R):
        R ← M[m];
        if  R==0 then
              M[m] ←1;
```

```
Load-reserve(R, m):
     <flag, adr>(<1, m>;
     R ( M[m];

Store-conditional(m, R):
     if <flag, adr> == <1, m>
     then cancel other proc' reservation on m;
              M[m] ((R;
              status ((succeed;
     else  status ((fail;
```

```
P:       Test&Set(mutex,Rtemp)
         if (Rtemp!=0) goto P
         Load(Rhead, head)
spin:    Load(Rtail, tail)
         if Rhead==Rtail goto spin
         Load(R, Rhead)
         Rhead=Rhead+1
         Store(head, Rhead)
V:       Store(mutex,0)
         process(R)
```

```
try:     Load-reserve(Rhead, head)
spin:    Load (Rtail, tail)
         if Rhead==Rtail goto spin
         Load(R, Rhead)
         Rhead = Rhead + 1
         Store-conditional(head, Rhead)
         if (status==fail) goto try
         process(R)
```

Which implementation do you prefer for performance? Argue in terms of bus occupancy.

Part E: Memory Models (18 Points)

Assume that the initial value of r1, r2, M[a] and M[flag] are 0 upon entry.

```
Process 1           Process 2

Store(a,1);         L:    r1 := Load(flag);
Store(flag,1);            BEQz(r1,L);
                          r2 := Load(a);
```

Code Example E-1 (for Questions 7 and 8)

Question 7. (4 Points)

Assuming sequential consistency (SC) enumerate all possible value pairs of (r1, r2) after executing Code Example E-1.

Question 8. (4 Points)

What if we allow reordering of loads but not stores? Enumerate all possible value pairs after executing Code Example E-1. Does speculative execution make any difference to your answer? Explain.

Question 9. (4 Points)

Modern microprocessors have store buffers to reduce the latency of store instructions which are often not on the critical path. In these processors, a load instruction first looks up the store buffer to find a match. If there is a match, the value is forwarded from the store buffer; otherwise, the load instruction goes to the data cache. We call this mechanism *load-store short-circuiting.*

Assume (1) **no** reordering of load and store instructions within a processor and (2) zero initial values for M[flag1] and M[flag2].

```
Process 1                  Process 2

Store(flag1, 1);           Store(flag2, 1);
r1 := Load(flag2);         r2 := Load(flag1);
```

Code Example E-2 (for Question 9)

Ben Bitdiddle has implemented a processor with load-store short-circuiting and found out that all four combinations of (r1, r2) pair—i.e., (0, 0), (0, 1), (1, 0) and (1, 1)—are possible after execution of Code Example E-2.

Explain briefly how each combination can occur (e.g., order of instruction execution, operation of store buffer, etc.).

(0, 0):

(0, 1):

(1, 0):

(1, 1):

Question 10. (6 Points)

Some of IBM 370 families implement load-store short-circuiting. In their implementation, if we insert load instructions (shaded in gray) into the code in Question 9, one combination gets eliminated from the possible final states of (r1, r2). (Again, assume (1) **no** reordering of load and store instructions within a processor and (2) zero initial values for M[flag1] and M[flag2].)

```
Process 1             Process 2

Store(flag1, 1);      Store(flag2, 1);
r3 := Load(flag1);    r4 := Load(flag2);
r1 := Load(flag2);    r2 := Load(flag1);
```

Code Example E-3 (for Question 10)

Which combination can be eliminated? From this information, what can you infer about the implementation (or behavior) of the store buffer and short circuits in IBM 370? Explain briefly.

Name: ________________________

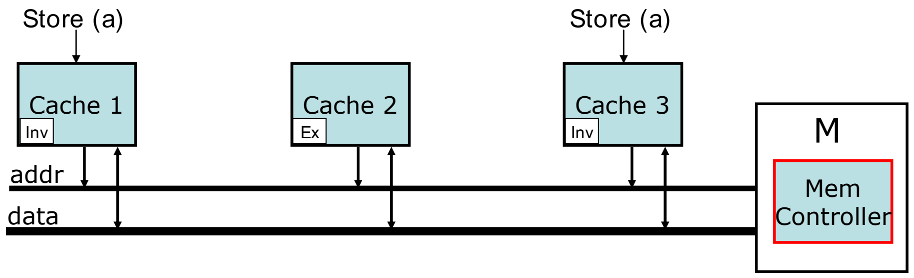
Part F: Snoopy Cache Coherence Protocol (12 Points)

Given the snoopy cache coherence protocol presented in Lecture 19 (2005), which is summarized in Appendix B, answer the following questions.

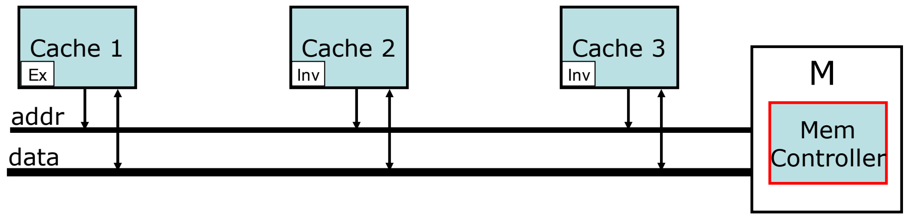
Question 11. (6 Points)

Assume a three-processor snoopy-bus configuration. In the initial state shown below, Cache 2 has an exclusive (dirty) copy of address (a), and Cache 1 and 3 are about to execute a store instruction on address (a).

In the final state, Cache 1 holds an exclusive copy and the other two are in invalid state. Starting from the initial state, describe a possible scenario which leads to the final state. You must provide (1) a sequence of bus transactions, (2) memory controller's response to each bus request, and (3) cache states in each step.



Initial state of the system



Final state of the system

Name: ______________________

Question 12. (6 Points)

Is it possible for two caches to be in the exclusive state at the same time? Explain why or why not.

Appendix A. Directory-based cache coherence protocol

This directory-based cache coherence protocol is consistent with the one described in Lecture 17 (2005). We provide this description to help you remember the protocol.

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 A-1: Cache State Transitions **(complete)**

Name: ______________________

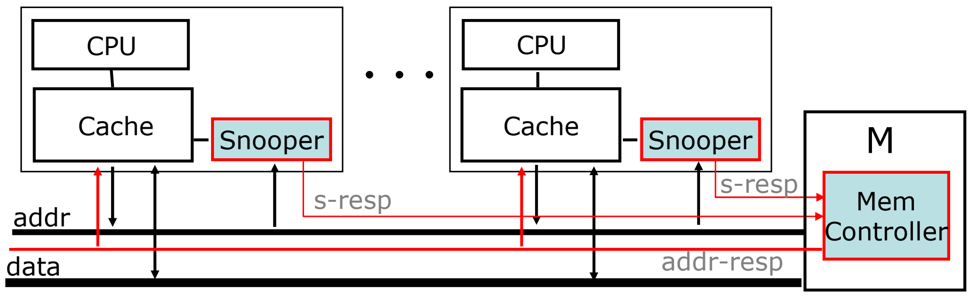
No.	Current State	Message Received	Next State	Dequeue Message?	Action
1	R(dir) & (dir = ε)	ShReq(a)	R({id})	Yes	ShRep(Home, id, data(a))
2	R(dir) & (dir = ε)	ExReq(a)	W(id)	Yes	ExRep(Home, id, data(a))
3	R(dir) & (dir = ε)	(Voluntary Prefetch)	R({id})	N/A	ShRep(Home, id, data(a))
4	R(dir) & (id ∉ dir) & (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	Tw(id)	WbRep(a)	R({id})	Yes	data-> memory
19	Tw(id)	FlushRep(a)	R(ε)	Yes	data-> memory
N/A	Tr(dir) …	**The row(s) for Tr(dir) are not given for questions.**			

Table A-2: Home Directory State Transitions, Messages sent from site **id (incomplete)**

Name: ________________________

Appendix B. Snoopy cache coherence protocol

This snoopy cache coherence protocol is consistent with the one described in Lecture 19 (2005) (without intervention). We provide this description to help you remember the protocol.



There are three possible states of a data block in cache: Invalid (Inv), Shared (Sh) and Exclusive (Ex).

There are three bus transactions (only address bus requests are shown):

- <ShReq, a>: issued by a cache on a read miss to load a cache line.
- <ExReq, a>: issued by a cache on a write miss (Sh or Inv state).
- <WbResp, a>: issued by a cache to write back its copy to memory. This transaction only affects the memory but not other snoopers.

Here is a summary of memory controller responses.

Address-Request	**Address-Response**	**Data**
<btag, ShReq, a>	retry	-
	unanimous-ok	<btag, M[a]>
<btag, ExReq, a>	retry	-
	unanimous-ok	<btag, M[a]>
<btag, Wb, a>	unanimous-ok	<btag, Wb, a, data> (Data to be written in memory)

An address response makes the following effects on the bus master (who initiated the address request):

Address Bus transaction <btag, a>	**Data Bus Transaction <btag, v>**

Unanimous-ok:
<a, type>==c2m.first
→ c2m.deq
obt.enq (tag, type, a)
 	
Retry:
<a, type>==c2m.first
→ c2m.deq
c2m.enq (type, a)
	<tag, type, a >==obt.first
	→ cache.setState(a,type);
	cache.setData(a,v);
	obt.deq