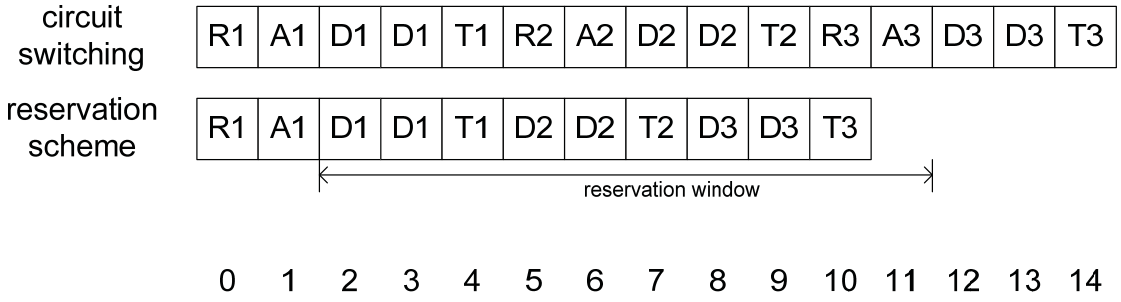
Problem M4.1: Networks-on-Chip

Problem M4.1.A

Consider a flow control method similar to circuit switching but where the request message 'reserves' each channel for a fixed period of time in the future (for example, for 10 cycles since a reservation is made). At each router along the path, a reservation is made if a request from a neighbor can be accommodated. If the request cannot be accommodated a NACK is sent that cancels all previous recommendations for the connection, and the request is retired. If a request reaches the destination, an acknowledgement is sent back to the source, confirming all reservations.

Draw a time-space diagram of a situation that demonstrates the advantage of reservation circuit switching over conventional circuit switching.

Clearly, this scheme eliminates the overhead of establishing connections for every packet. For example, if a source is sending out short packets (two data flits per packet) and the reservation window is 10 cycles, the time-space diagram looks like the following:

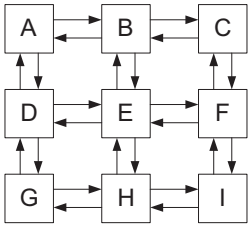


Note that tail flits may be able to get eliminated in this new scheme if they are used only to indicate when channels can be deallocated.

Problem M4.1.B

(a) Randomized dimension-order: All packets are routed minimally. Half of the packets are routed completely in the X dimension before the Y dimension and the other packets are routed Y before X.

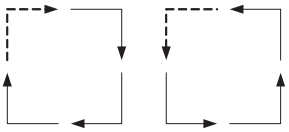
No, this generates a cycle in CDG.



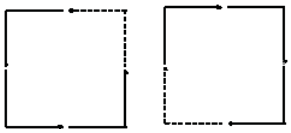
In the CDG corresponding to the mesh network above, for example, EF→FC→CB→BE→EF generates a cycle (Flow E-F-C, flow F-C-B, flow C-B-E, and flow B-E-F will generate a deadlock).

(b) Less randomized dimension-order: All packets are routed minimally. Packets whose minimal direction is increasing in both X and Y, always route X before Y. Packets whose minimal direction is decreasing in both X and Y, always route Y before X. All other packets randomly choose between X before Y and vice versa.

Yes. This effectively eliminates the following two turns.



(c) All packets are prohibited to take the two turns in dash:



No. In the 3-by-3 mesh network in part (a), EB->BC->CF->FE->ED->DG->GH->GE->EB generates a cycle.

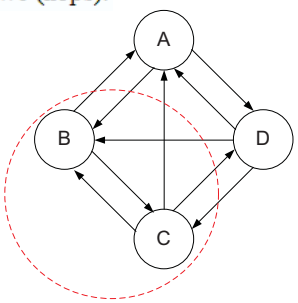
Problem M4.2: Non-mesh Networks

Problem M4.2.A

Fill in the following table of the properties of this network.

Diameter	**2**
Average Distance	**7/6**
Bisection Bandwidth	**6 flit/cycle**

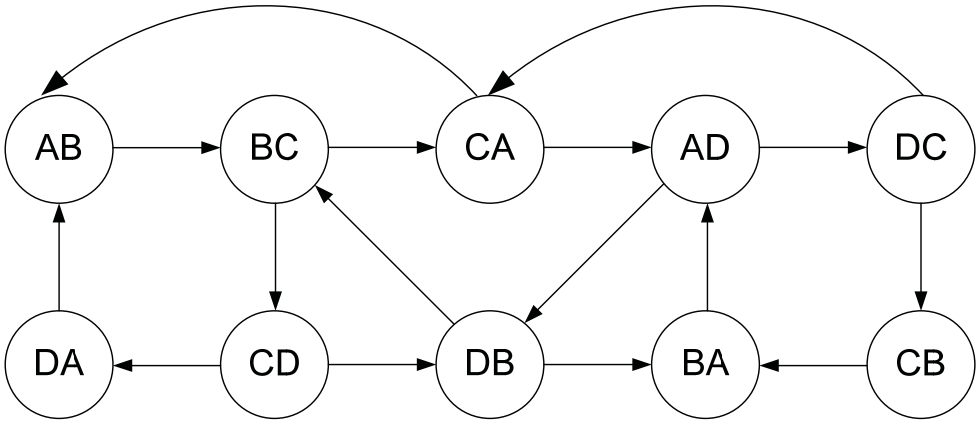
There are 12 source-destination pairs. Among them, only the routing distance of B-to-D and A-to-C is 2 hops (which is the diameter of this network), while others are 1. Therefore, the average distance is (1x10 + 2x2)/12 = 7/6 (hops).



And the bisection bandwidth is 6 flit/cycle.

Problem M4.2.B

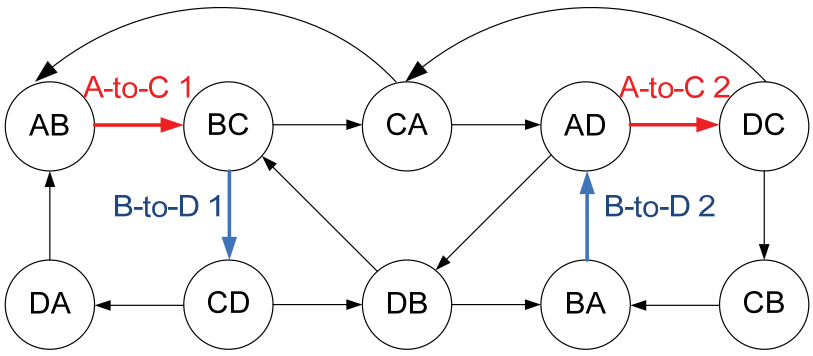
Draw the channel dependency graph of this network.



Problem M4.2.C

Is a **minimal** routing on this network deadlock-free? Show your reasoning and give a deadlock scenario if it is not deadlock -free.

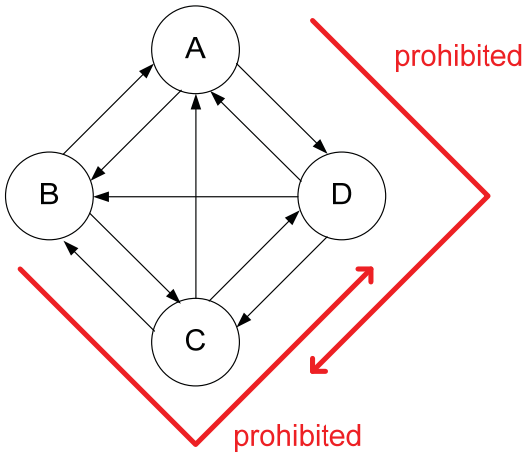
Yes. In minimal routing, all flows except for the ones from B to D and from A to C have 1-hop distance, which is represented by a single node in CDG; they are not holding resources while waiting for another because they need only one resource. The dependencies of flow from B to D and from A to C is represented in the CFG as following (note that each flow can take two possible minimal routes):



There are no cycles in the CDG, thus the routing is deadlock-free.

Problem M4.2.D

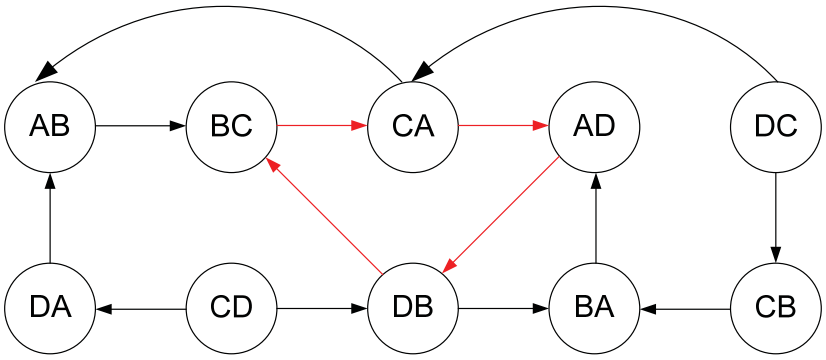
Now, we use a possibly **non-minimal** routing on this network. Plus, we prohibited the following two movements on the non-minimal routing: 1) A to D then D to C and 2) B to C then C to D.



Is this routing deadlock-free? Show your reasoning and give a deadlock scenario if it is not deadlock -free.

No.

Prohibiting those movements, the CDG becomes:



However, there are still cycles in this CDG. For example, if flow 1 from B to D is routed through B-C-A-D, flow 2 from C to B is routed through C-A-D-B, and flow 3 from A to C is routed through A-D-B-C, there can be deadlock by these three flows.

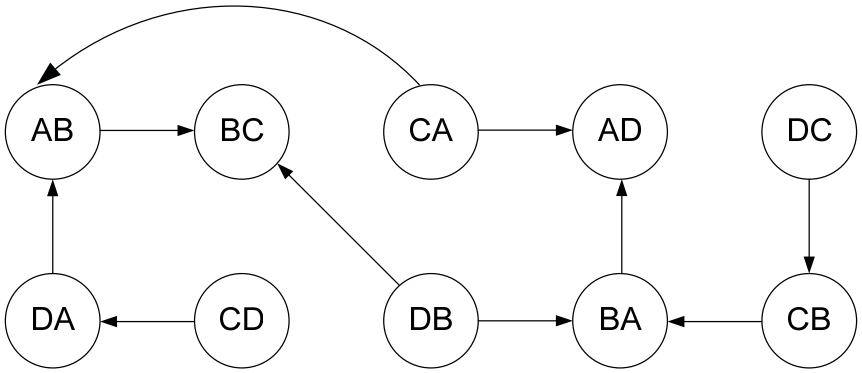
Problem M4.2.D

Still having the two paths in M4.2.D prohibited, we added another restriction in routing: the link from C to A can be used only by packets generated at C, before the packets are transferred to any other nodes (it should be the first link those packets ever take). Also, the link from D to B can be used only by packets generated at D with the same condition (however, routes may be non-minimal).

Is this routing deadlock-free? Show your reasoning and give a deadlock scenario if it is not deadlock -free.

Yes.

These conditions effectively eliminate any turns cornered at C and moving out to A, and any turns cornered at D and moving out to B. Then the CDG becomes:



This CDG is acyclic, thus this routing is deadlock-free.

Problem M4.3: Sequential Consistency [? Hours]

Problem M4.3.A

Can X hold value of 4 after all three threads have completed? Please explain briefly.

(Yes) / No

C1-C4, B1-B3, A1-A4, B4- B6

Problem M4.3.B

Can X hold value of 5 after all three threads have completed?

Yes / (No)

All results must be even!

Problem M4.3.C

Can X hold value of 6 after all three threads have completed?

(Yes) / No

All of C, All of A, All of B

Problem M4.3.D

For this particular program, can a processor that reorders instructions but follows local dependencies produce an answer that cannot be produced under the SC model?

Yes / (No)

All stores/loads must be done in order because they're to the same address, so no new results are possible.

Problem M4.4: Synchronization Primitives [? Hours]

The mechanism here is as follows: LdR requests READ access to the address, StC requests WRITE access to the address. Many students suggested that LdR can request WRITE access to the address right away, which could lead to live lock.

Problem M4.4.A

Describe under what events the local reservation for an address is cleared.

If another processor requests Write access to the same cache line.

Problem M4.4.B

Is it possible to implement LdR/StC pair in such a way that the memory bus is not affected, i.e., unaware of the addition of these new instructions? Explain

Yes. Writeback [P2C_Req(a) S] and [C2P_Req(a) S] are sent normally. The "reservation" is local (probably in the snooper or in the cache, though that might take too much resources – there are very few reservations needed at the same time for any processor).

Problem M4.4.C

Give two reasons why the LdR/StC pair of instructions is preferable over atomic read-test-modify instructions such as the TEST&SET instruction.

1. Bus doesn't need to be aware of them.
2. Everything is local.
3. No ping-pong.
4. No extra hardware (tied to 1)

Problem M4.4.D

LdR/StC pair of instructions were conceived in the context of snoopy busses. Do these instructions make sense in our directory-based system in Handout #12? Do they still offer an advantage over atomic read-test-modify instructions in a directory-based system? Please explain.

No – our bus invalidates before transitioning from S to M. In general, maybe.

Problem M4.5: Implementing Directories

Problem M4.5.A

Overhead for a 4-processor system: 4 bits / 32 bytes = 4 / (32 * 8) = 1/64

Overhead for a 64-processor system: 64 bits / 32 bytes = 64 / (32 * 8) = 1/4

Problem M4.5.B

Sequence 1	bit-vector scheme # of invalidate-requests	single-sharer scheme # of invalidate-requests
Processor #0 reads **B**	0	0
Processor #1 reads **B**	**0**	**1**
Processor #0 reads **B**	**0**	**1**

For the bit-vector scheme: No invalidate-requests are sent.

For the single-sharer scheme:
1 invalidate-request is sent to P0 when P1 reads B.
1 invalidate-request is sent to P1 when P0 reads B the second time.

Sequence 2	bit-vector scheme # of invalidate-requests	single-sharer scheme # of invalidate-requests
Processor #0 reads **B**	0	0
Processor #1 reads **B**	**0**	**1**
Processor #2 writes **B**	**2**	**1**

For the bit-vector scheme:
1 invalidate-request is sent to each shared processor (P0 and P1) when P2 writes B.
-> 2 invalidate-requests are sent.

For the single-sharer scheme:
1 invalidate-request is sent to P0 when P1 reads B.
1 invalidate-request is sent to the only sharer (P1) when P2 writes B.

Problem M4.5.C

Sequence 1	global-bit scheme # of invalidate-requests
Processor #0 reads **B**	0
Processor #1 reads **B**	**0**
Processor #0 reads **B**	**0**

For the global-bit scheme: No invalidate-requests are sent.

Sequence 2	global-bit scheme # of invalidate-requests
Processor #0 reads **B**	0
Processor #1 reads **B**	**0**
Processor #2 writes **B**	**64**

For the global-bit scheme:
1 invalidate-request is sent to each of the 64 processors because the global bit is set when P2 writes B. -> 64 invalidate-requests are sent.

Note: If the protocol is optimized, no invalidate-request would be sent to P2 and the number of invalidate-requests would be 63 instead of 64.

Problem M4.6: Tracing the Directory-based Protocol [? Hours]

Processor A	Processor B	Processor C
A1: ST X, 1	B1: R := LD X	C1: ST X, 6
A2: R := LD X	B2: R := ADD R, 1	C2: R := LD X
A3: R := ADD R, R	B3: ST X, R	C3: R := ADD R, R
A4: ST X, R	B4: R:= LD X	C4: ST X, R
	B5: R := ADD R, R	
	B6: ST X, R	

Problem M4.6.A

Processor A			Processor B			Processor C		
Ins	EO	Messages	Ins	EO	Messages	Ins	EO	Messages
A1	1	<M,A,Req,x,M> <A,M,Rep,x,I,M,0>	B1	4	**<M,B,Req,x,S> <A,M,Req,x,S> <M,A,Rep,x,M,S,2> <B,M,Rep,x,I,S,2>**	C1	8	**<M,C,Req,x,M> <B,M,Req,x,I> <M,B,Rep,x,M,I,6> <C,M,Rep,x,I,M,6>**
A2	2		B3	5	**<M,B,Req,x,M> <A,M,Req,x,I> <M,A,Rep,x,S,I,-> <B,M,Rep,x,S,M,->**	C2	9	
A4	3		B4	6		C4	10	
			B6	7				

How many messages are generated? **14**

Problem M4.6.B

Processor A			Processor B			Processor C		
Ins	EO	Messages	Ins	EO	Messages	Ins	EO	Messages
A1	**5**	**<M,A,Req,x,M> <B,M,Req,x,I> <M,B,Rep,x,M,I,2> <A,M,Rep,x,I,M,2>**	B1	**1**	**<M,B,Req,x,S> <B,M,Rep,x,I,S,0>**	C1	**8**	**<M,C,Req,x,M> <A,M,Req,x,I> <M,A,Rep,x,M,I,2> <C,M,Rep,x,I,M,2>**
A2	**6**		B3	**2**	**<M,B,Req,x,M> <B,M,Rep,x,S,M,->**	C2	**9**	
A4	**7**		B4	**3**		C4	**10**	
			B6	**4**				

How many messages are generated? **12**

Problem M4.6.C

Can the number of messages in Problem M4.6.B be decreased *by using voluntary responses*? Explain.

Yes – all the requests can be eliminated using voluntary rules. Total number of messages would be 6 instead of 12.

Problem M4.6.D

Processor A			Processor B			Processor C		
Ins	EO	Messages	Ins	EO	Messages	Ins	EO	Messages
A1	**1**	**<M,A,Req,x,M> <A,M,Rep,x,I,M,0>**	B1	**2**	**<M,B,Req,x,S> <A,M,Req,x,S> <M,A,Rep,x,M,S,1> <B,M,Rep,x,I,S,1>**	C1	**3**	**<M,C,Req,x,M> <A,M,Req,x,I> <B,M,Req,x,I> <M,A,Rep,x,S,I> <M,B,Rep,x,S,I> <C,M,Rep,x,I,M,1>**
A2	**4**	**<M,A,Req,x,S> <C,M,Req,x,S> <M,C,Rep,x,M,S,6> <A,M,Rep,x,S,6>**	B3	**5**	**<M,B,Req,x,M> <A,M,Req,x,I> <C,M,Req,x,I> <M,A,Rep,x,S,I> <M,C,Rep,x,S,I> <B,M,Rep,x,I,M,6>**	C2	**6**	**<M,C,Req,x,S> <B,M,Req,x,S> <M,B,Rep,x,M,S,2> <C,M,Rep,x,I,S,2>**
A4	**7**	**<M,A,Req,x,M> <B,M,Req,x,I> <C,M,Req,x,I> <M,B,Rep,x,S,I> <M,C,Rep,x,S,I> <A,M,Rep,x,I,M,2>**	B4	**8**	**<M,B,Req,x,S> <A,M,Req,x,S> <M,A,Rep,x,M,S,12> <B,M,Rep,x,S,12>**	C4	**9**	**<M,C,Req,x,M> <A,M,Req,x,I> <B,M,Req,x,I> <M,A,Rep,x,S,I> <M,B,Rep,x,S,I> <C,M,Rep,x,I,M,12>**
			B6	**10**	**<M,B,Req,x,M> <C,M,Req,x,I> <M,C,Rep,x,M,I,4> <B,M,Rep,x,I,M,4>**			

How many messages are generated? **46**

Problem M4.7: Snoopy Cache Coherent Shared Memory [? Hours]

Problem M4.7.A Where in the Memory System is the Current Value

See Table M4.7-1, M4.7-2 and M4.7-3.

Problem M4.7.B MBus Cache Block State Transition Table

See Table M4.7-1, M4.7-2 and M4.7-3.

Problem M4.7.C Adding atomic memory operations to MBus

Imagine a dual processor machine with CPUs A and B. Explain the difficulty of CPU A performing fetch-and-increment(x) when the most recent copy of x is cleanExclusive in CPU B's cache. You may wish to illustrate the problem with a short sequence of events at processor A and B.

The problem is that CPU B can read the value in location x while CPU A is performing the fetch-and-increment operation—which violates the idea of fetch-and-increment being atomic. For example, consider the following sequence of events and corresponding state transitions and operations:

Event	**CPU A**	**CPU B**
1	Read(x); I->CS; send CR	
2		Snoop CR; CE->CS
3		Read(x)
4	Write(x); CS->OE; send CI	
5		Snoop CI; CS->I

Fill in the rest of the table below as before, indicating state, next state, where the block in question may reside, and the CPU A and MBus transactions that would need to occur atomically to implement a fetch-and-increment on processor A.

State	other cached	ops	actions by this cache	next state	this cache	other caches	mem
Invalid	yes	read	**CR**	**CS**	√	√	√
cleanShared	yes	write	**CI**	**OE**	√		

initial state	other cached	ops	actions by this cache	final state	this cache	other caches	mem
Invalid	no	none	none	**I**			√
		CPU read	**CR**	**CE**	√		√
		CPU write	**CRI**	**OE**	√		
		replace	none	*Impossible*			
		CR	none	**I**		√	√
		CRI	none	**I**		√	
		CI	none	*Impossible*			
		WR	none	*Impossible*			
		CWI	none	**I**			√
Invalid	yes	none	same as above	**I**		√	√
		CPU read		**CS**	√	√	√
		CPU write		**OE**	√		
		replace		*Impossible*			
		CR		**I**		√	√
		CRI		**I**		√	
		CI		**I**		√	
		WR		**I**		√	√
		CWI		**I**			√

initial state	other cached	ops	Actions by this cache	final state	this cache	other caches	mem
cleanExclusive	no	none	none	**CE**	√		√
		CPU read	none	CE	√		√
		CPU write	none	OE	√		
		replace	none	I			√
		CR	none or CCI[1]	**CS**	√	√	√
		CRI	none or CCI[1]	I		√	
		CI	none	*Impossible*			
		WR	none	*Impossible*			
		CWI	none	I			√

Table M4.7-1

[1] Some Sun MBus implementations perform CCI from the cleanExclusive state, while others do not. We accept both answers.

initial state	other cached	ops	Actions by this cache	final state	this cache	other caches	mem
ownedExclusive	no	none	none	**OE**	√		
		CPU read	none	**OE**	√		
		CPU write	none	**OE**	√		
		replace	WR	**I**			√
		CR	CCI	**OS**	√	√	
		CRI	CCI	**I**		√	
		CI	none	*Impossible*			
		WR	none	*Impossible*			
		CWI	none	**I**			√

initial state	other cached	ops	actions by this cache	final state	this cache	other caches	mem
cleanShared	no	none	none	**CS**	√		√
		CPU read	none	**CS**	√		√
		CPU write	CI	**OE**	√		
		replace	none	**I**			√
		CR	none[2]	**CS**	√	√	√
		CRI	none	**I**		√	
		CI	none	*Impossible*			
		WR	none	*Impossible*			
		CWI	none	**I**			√
cleanShared	yes	none	same as above	**CS**	√	√	√
		CPU read		**CS**	√	√	√
		CPU write		**OE**	√		
		replace		**I**		√	√
		CR		**CS**	√	√	√
		CRI		**I**		√	
		CI		**I**		√	
		WR		**CS**	√	√	√
		CWI		**I**			√

Table M4.7-2

[2] Some Sun MBus implementations perform CCI from the cleanShared state. However, in these implementations, requests are not broadcast on a bus, but are handled by a central system controller. The system controller arbitrates which cache with a cleanShared copy provides the data. Unless an explanation is provided, CCI is not a valid response from this state.

initial state	other cached	ops	actions by this cache	final state	this cache	other caches	mem
ownedShared	no	none	none	**OS**	√		
		CPU read	none	**OS**	√		
		CPU write	CI	**OE**	√		
		replace	WR	**I**			√
		CR	CCI	**OS**	√	√	
		CRI	CCI	**I**		√	
		CI	none	*Impossible*			
		WR	none	*Impossible*			
		CWI	none	**I**			√
ownedShared	yes	none	same as above	**OS**	√	√	
		CPU read		**OS**	√	√	
		CPU write		**OE**	√		
		replace		**I**		√	√
		CR		**OS**	√	√	
		CRI		**I**		√	
		CI		**I**		√	
		WR		*Impossible*			
		CWI		**I**			√

Table M4.7-3

Problem M4.8: Snoopy Cache Coherent Shared Memory [? Hours]

Problem M4.8.A

Fill out the state transition table for the new COS state:

initial state	other cached	ops	actions by this cache	final state
COS	yes	none	none	**COS**
		CPU read	**none**	**COS**
		CPU write	**CI**	**OE**
		replace	**none**	**I**
		CR	**CCI**	**COS**
		CRI	**CCI**	**I**
		CI	**none**	**I**
		WR	**Impossible**	
		Or:	**none**	**COS**
		CWI	**none**	**I**

Note that WR is not necessary during replace because the line is clean.
Also, an incoming WR operations is Impossible because other caches can only have the block in the CS state, but (none, COS) was also accepted as a correct answer.

Problem M4.8.B

cache transaction	source for data	state for data block B			
		cache 1	cache 2	cache 3	cache 4
0. *initial state*	—	**I**	**I**	**I**	**I**
1. cache 1 reads data block B	**memory**	**CE**	**I**	**I**	**I**
2. cache 2 reads data block B	**CCI**	**COS**	**CS**	**I**	**I**
3. cache 3 reads data block B	**CCI**	**COS**	**CS**	**CS**	**I**
4. cache 1 replaces block B	**-**	**I**	**CS**	**CS**	**I**
5.cache 4 reads data block B	**memory**	**I**	**CS**	**CS**	**CS**

Problem M4.8.C

When the CPU does a write, it can change a cache block from CE to OE with no bus operation, but to transition from COS to OE it must first broadcast a CI on the bus to invalidate any shared (CS) copies of the block.

Problem M4.9: Snoopy Caches [? Hours]

Problem M4.9.A

Hint: Consider how much processing can be performed safely on the following sequences after an invalidation request for x has been received

Ld x; Ld y; Ld x

Ld x; St y; Ld x

The snooper can allow the CPU to continue executing normally, but cannot allow any new messages from the outside to enter the caches until AFTER the caches cleared their content.

Problem M4.9.B

Consider a situation when L2 has a cache line marked Ex and a ShReq comes on the bus for this cache line. What should the snooper do in this case, and why?

Here the snooper MUST respond RETRY and get the cache to write back the value.

Problem M4.9.C

When an ExReq message is seen by the snooper and there is a Wb message in the C2M queue waiting to be sent, the snooper replies ***retry***. If the cache line is about to be modified by another processor, why is it important to first write back the already modified cache line? Does your answer change if cache lines are restricted to be one word? Explain.

Because otherwise the Wb can happen out of order with some other memory operation and SC could be broken.

Problem M4.10: Relaxed Memory Models [? Hours]

We will study the interaction between two processes on different processors on such a system:

P1	P2
P1.1: LW R2, 0(R8)	P2.1: LW R4, 0(R9)
P1.2: SW R2, 0(R9)	P2.2: SW R5, 0(R8)
P1.3: LW R3, 0(R8)	P2.3: SW R4, 0(R8)

Problem M4.10.A

Memory	contents
M[R8]	7
M[R9]	6

Yes **No**

P1.1 P2.1 P1.2 P1.3 P2.2 P2.3

Problem M4.10.B

memory	Contents
M[R8]	6
M[R9]	7

Yes **No**

The result would require that the memory contents don't change. Since each thread reads a data value and writes it to another address, this simply impossible here.

Problem M4.10.C

Is it possible for M[R8] to hold 0?

Yes **No**

The only way that M[R8] could end up with 0 is if P2.3 is completed before P2.1 and P2.2. This violates Weak Ordering, so it is not possible.

Now consider the same program, but with two **MEMBAR** instructions.

P1	P2
P1.1: LW R2, 0(R8)	P2.1: LW R4, 0(R9)
P1.2: SW R2, 0(R9)	$MEMBAR_{RW}$
$MEMBAR_{WR}$	P2.2: SW R5, 0(R8)
P1.3: LW R3, 0(R8)	P2.3: SW R4, 0(R8)

We want to compare execution of the two programs on our system.

Here the intention was to keep the starting conditions the same as in first three questions, and ask about the final conditions. This wasn't clear, so we accepted both solutions. The yes/no answers don't actually change, but Questions 11 for 12 become simpler.

Problem M4.10.D

If both M[R8] and M[R9] contain 6, is it possible for R3 to hold 8?

Without **MEMBAR** instructions? (**Yes**) **No**

With **MEMBAR** instructions? (**Yes**) **No**

Following sequence works with and without MEMBAR instructions:
P1.1 -> P1.2 -> P2.1 -> P2.2 -> P1.3 -> P2.3

Problem M4.10.E

If both M[R8] and M[R9] contain 7, is it possible for R3 to hold 6?

Without **MEMBAR** instructions? **Yes** (**No**)

With **MEMBAR** instructions? **Yes** (**No**)

If M[R8] and M[R9] are to end up with 7, we have to execute P2.3 before we execute P1.1 Since P1.3 has to come after P1.1 (Weak Ordering), R3, has to end up with 7 not 6.

Problem M4.10.F

Is it possible for both M[R8] and M[R9] to hold 8?

Without **MEMBAR** instructions? (**Yes**) **No**

P2.2 P1.1 P1.2 P2.1 P2.3 P1.3

With **MEMBAR** instructions? **Yes** (**No**)

The sequence above violates the MEMBAR in P2—P2.2 executes before P2.1. That is the only way to get 8 into both memory locations, thus the result is impossible with MEMBARs insterted.

Problem 4.11: Memory Models

Consider a system which uses **Sequential Consistency (SC)**. There are three processes, **P1**, **P2** and **P3**, on different processors on such a system (the values of R_A, R_B, R_C were all zeros before the execution):

P1	P2	P3
P1.1: ST (A), 1 P1.2: LD R_C, (C)	P2.1: ST (B), 1 P2.2: LD R_A, (A)	P3.1: ST (C), 1 P3.2: LD R_B, (B)

Problem 4.11.A

After all processes have executed, it is possible for the system to have multiple machine states. For example, $\{R_A, R_B, R_C\}= \{1,1,1\}$ is possible if the execution sequence of instructions is P1.1→P2.1→P3.1→P1.2→P2.2→P3.2. Also, $\{R_A, R_B, R_C\}= \{1,1,0\}$ is possible if the sequence is P1.1 → P1.2 → P2.1 → P3.1 → P2.2 → P3.2.

For each state of $\{R_A, R_B, R_C\}$ below, specify the execution sequence of instructions that results in the corresponding state. If the state is **NOT** possible with SC, just put X.

{0,0,0} : X

{0,1,0} : P2.1 P2.2 P1.1P1.2P3.1 P3.2

{1,0,0} : P1.1 P1.2 P3.1 P3.2 P2.1 P2.2

{0,0,1} : P3.1 P3.2 P2.1 P2.2 P1.1 P1.2

Problem 4.11.B

Now consider a system which uses **Weak Ordering(WO)**, meaning that a read or a write may complete before a read or a write that is earlier in program order if they are to different addresses and there are no data dependencies.

Does WO allow the machine state(s) that is not possible with SC? If yes, provide an execution sequence that will generate the machine states(s).

Yes. {0,0,0} by P1.2→P2.2→P3.2→P1.1→P2.1→P3.1

Problem 4.11.C

The WO system in Problem 4.11.B provides four fine-grained memory barrier instructions. Below is the description of these instructions.

- **MEMBAR$_{RR}$** guarantees that all read operations initiated before the MEMBAR$_{RR}$ will be seen before any read operation initiated after it.
- **MEMBAR$_{RW}$** guarantees that all read operations initiated before the MEMBAR$_{RW}$ will be seen before any write operation initiated after it.
- **MEMBAR$_{WR}$** guarantees that all write operations initiated before the MEMBAR$_{WR}$ will be seen before any read operation initiated after it.
- **MEMBAR$_{WW}$** guarantees that all write operations initiated before the MEMBAR$_{WW}$ will be seen before any write operation initiated after it.

Using the minimum number of memory barrier instructions, rewrite **P1**, **P2** and **P3** so the machine state(s) that is not possible with SC by the original programs is also not possible with WO by your programs.

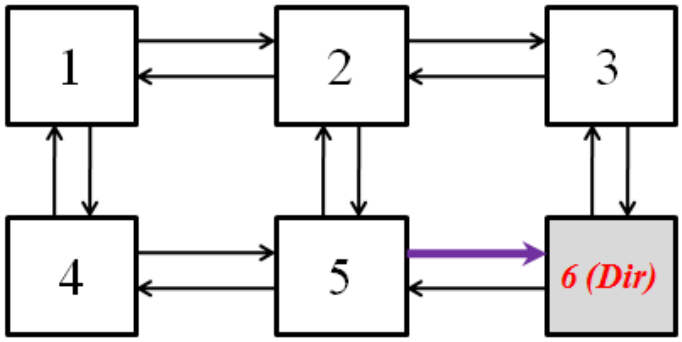
P1	P2	P3
P1.1: ST (A), 1 MEMBAR$_{WR}$ P1.2: LD R_C, (C)	P2.1: ST (B), 1 MEMBAR$_{WR}$ P2.2: LD R_A, (A)	P3.1: ST (C), 1 MEMBAR$_{WR}$ P3.2: LD R_B, (B)

Last updated:
11/28/2011

Problem M4.12: Directory-based Protocol

Problem 4.12.A

The following questions deal with the directory-based protocol discussed in class. Assume XY routing, and message passing is FIFO. (**XY routing algorithm** first routes packets horizontally, towards their X coordinates, and then vertically towards their Y coordinates.) Protocol messages with the same source and destination sites are always received in the same order as that in which they were sent. **For this question, assume that the cache coherence protocol is free from deadlock, livelock and starvation**.



Assume the node 6 serves as the home directory, where the states for memory blocks are stored. Assume all caches are initially empty and no responses are sent voluntarily (i.e. every response is caused by a request)

Processor 1	Processor 4	Processor 5
1.1: ST X, 10	4.1: LD R1, X	5.1: ST X, 20

Suppose the global execution order is as follows:

4.1 => 5.1 => 1.1

Assume that the next instruction will start its execution only when the previous instruction has completed. For each instruction, list all protocol messages that are sent over the link 5 -> 6 (the purple link in the above figure).

4.1: <6,4,C2M_Req,X,S> (4.1),

5.1: <6,5,C2M_Req,X,M>, <6,4,C2M_Rep,X,S,I> (5.1),

1.1: <6,5,C2M_Rep,X,M,I,20> (1.1)

Problem 4.12.B

For the directory protocol, we assume the message passing to be FIFO, meaning protocol messages with the same source and destination are always received in the same order as that in which they were sent. Now suppose messages can be delivered out-of-order for the same source and destination pairs. Describe one scenario that the cache coherence protocol will break due to this out-of-order delivery.

1. Core 1: <M,1,C2M_Req,a,S> => <1,M,M2C_Rep,a,I,S,data> (not yet reached)
2. Core 2: <M,2,C2M_Req,a,M> => <1,M,M2C_Req,a,I>

If <1,M,M2C_Req,a,I> arrives earlier than <1,M,M2C_Rep,a,I,S,data>, it will be ignored, and the core will not send any reply to home which is waiting. => Deadlock.

Problem 4.12.C

Under the 6823 directory-based protocol, a cache will receive a writeback request from the directory <M2C_Req, a, S> for address "a" when it is in state M and another cache wants a shared copy. Is it possible for a cache in the **S** state to receive <M2C_Req, a, S> ? Describe how this scenario can occur using the messages passed between the cache and the memory, and the state transitions.

Cache 1 in M, does voluntary writeback <M,1,M2C_Rep,a,M,S,data> and goes to S state. Now Cache 2 in I state does a <M,2,C2M_Req,a,S>. If the Mem hasn't received Cache 1's response yet, it will send a <1,M,P2C_Req,a,S> to Cache 1 which is in S.