Parallel Computer Architecture
Prof. Hemangee K. Kapoor
Department of Computer Science and Engineering
Indian Institute of Technology Guwahati
Week - 04
Lecture - 25

Lec 25: 3 state: MSI Protocol

Hello everyone. We are doing module 3 on Cache Coherence. Today's lecture is on the MSI protocol. This is the list of the four cache coherence protocols which we are going to study. Today's protocol is the three-state MSI protocol. So as the three states are M stands for modified, S for shared and I for invalid.

This is an invalidation based protocol similar to the VI protocol, but it is a write back cache based protocol instead of a write through. VI protocol was using write through caches and MSI protocol is going to use write back caches. So that's the difference between the two and because of the write through versus the write back difference, we have a new state here called the modified state. The types of messages, they are very similar to the VI.

From the processor side nothing changes. We normally have a read from the processor which is termed as processor read, processor write and when the processor sends a read or write if it is a cache hit no problem, but if it is a cache miss in case of a read the snoop controller has to go on to the bus to bring the block. Hence it sends a bus read transaction. In case it is a write request from the processor and the block is absent in the cache, then we have to go out to bring the block either from the memory or from some other cache who might have modified the block. So for this instead of just sending a bus read we are going to send a something, some variant of a bus write.

In the VI we simply sent a bus write because it was a write through protocol. In the MSI protocol, we don't send a bus write, but we send a BusEdX. So what does this BusRdX mean? In case of a processor write, we send not the BusRd but a variant of that called the BusRdX. This X means for exclusive. We want an exclusive access to this cache block because we are going to modify the data inside this block.

So we are asking essentially permission to write this block. So when a BusRdX transaction goes on to the interconnect if there is any other cache which is holding this block it has to relinquish that block and give full permission to this particular cache to modify it. Okay. So that is the new transaction which was not there in the VI protocol. Another variant of the bus write is the bus write back. So what is BusWB? This happens

when the particular cache removes the block from its cache that is happening during block evictions due to cache replacement policies this might take place or if a new reader comes into the system this particular cache suppose we are talking of P1 has this block dirty and suppose P2 wants to read this cache. Right.

So if it wants to read this block then the data has to be provided by P1 to P2 even in that case it has to do something similar to a write back. Okay. Then we look at the different states of the MSI protocol. So the first state is invalid. Invalid essentially means the block is not present in this cache. Then the second state is shared. Now shared means this block is present or may be present in multiple caches.

So current cache has it in the shared mode and maybe other caches also have it. Now when other caches can have the block and this is an invalidation based protocol, what do we imply? We imply that all these other copies are also of the same kind. So they have the same value and indirectly it means it is a read only copy. So when I have the block in the shared mode it is a read only copy. The third state is modified.

Modified means dirty copy. This particular cache has changed the cache block. It has the dirty copy. It is in some way owner of this data item because in future if there is a new reader into the system who wants to read or write to this block, the particular cache has to provide the data. So I for invalid, S for shared which is nothing but only read only copies.

So these are normally read only copies across different caches. Right. So other caches also have this block and in case of modified the block is dirty. This is the only cache having this block because the most up to date copy is with this cache and not even with the memory. And why is it not with the memory? Because this is a write back protocol. In VI protocol, whenever a write happened the memory was also updated hence the memory always had up to date copy.

But in MSI protocol, write back protocol, only this particular cache has the up to date copy. Memory is out of date. Memory has the stale data. Okay. This being a invalidation based protocol, so whenever the cache goes to state M it has to invalidate all the other copies. So these, this is the meaning of the three states. Okay, okay.

So let us do the FSM interactions as a sample example. We have three processors P1, P2, P3 and the blue is the cache block in this case. So P1 has this cache block so it is not invalid. So in P1, the block could be either in S state or the M state. In P2 also it is not invalid so it could be probably here.

And because it is both in P1 and P2, we can infer that it could be in shared state in these two whereas in P3 it is in invalid state because P3 does not have this block. Right. So this is the global three FSMs which I have. Now, here if P1 reads it has the block so it can simply do a hit into the cache and finish its work. Similarly P2 also can finish reading because both have the copies with them. P3 has nothing to do with this transaction.

Next if P2 tries to write to this block, if P2 wants to write, presently it is in the shared state. It is in the shared state and when in shared state it knows that there could be probably other copies into the system and hence it has to invalidate others. So it sends this some message over the bus and that message deletes the block inside P1. So P1 moves to this when I do the second example. Right. During the first one both were in one, when I do step number two P1 moves to two P3 is already invalidated.

Later if P3 tries to read, if P3 tries to read the block during the third example, P3 will actually go into the shared state because it is wanting to read but before it goes to the shared state it puts the bus read transaction and who will provide the data? In this case P2 because P2 has done the most recent write, so P2 gives the data and P2 moves to the shared state in this case because after giving the data to P3, P2 is not permitted to modify because there is another sharer into the system and hence P2 becomes shared, P1 state does not change. So P1 is equal to invalid in this case. So P1 is invalid P3 reads and its states becomes S. Okay. So this is an informal discussion of how these FSM interact with each other through this MSI protocol. Right. So we have seen these three things 1, 2 and 3 and in the FSM I have put these states which they go through during these three examples.

I will just put them here and during 2 also P3 remains I. So for these three steps or examples which I have taken, I will list them inside this FSM. So for P1 we have listed that during step 1 it is shared, step 2 it becomes invalid and even in step 3 it is invalid. For P2, initially it is S then during 2 when it writes, it becomes modified, during 3, it becomes shared and for P3, it is invalid during 1 and 2 and it becomes shared during step number 3. Okay. So that was a sample introduction of how these FSMs will interact with each other.

Now we will look at actual transactions which happen in the MSI protocol. Right. So the popular ones as we did with VI also is related to the bus, the bus read and instead of the bus write we have a different transaction called the BusRdX or the BusWB. Now what is bus read? Where the processor read comes it misses into the cache, so the cache controller has to send a bus read transaction on the interconnect, so that whoever is the owner of this particular block will give you the data. Now the data will come either

from the memory or from another cache who has recently modified the block.

That is the bus read. Bus read exclusive. Now when a processor write happens and that block is not present in the cache, then you have to bring the block. Even if the block is present in the cache but it is in the shared state that is you have brought the block only for reading. Now you wish to write to that block so you have to inform others because others need to invalidate their copies. So in this case whenever there is a processor write, we have to send a BusRdX transaction if we do not have the modification writes on this block. Okay.

So during processor write send a BusRdX transaction, so that others will invalidate their copies. With this BusRdX the particular cache gets an exclusive permission to write. So this exclusive word is important. This gets exclusive permission so that it can keep on modifying this block without informing others in future. Okay. As long as it gets this permission. The third type of transaction is the bus write back.

Now bus write back happens when the cache has to remove the block. In this case, the processor is not involved at all. Why is the processor not involved? Because processor is not sending a read or write to this cache. But the cache block may be evicted because some new block wants to come and sit into its place or there is a new reader into the system who wants to read or write to this block. Right. So there is another processor who wants this block so we have to give this block. So this is called a bus write back transaction.

Another word for that is called a flush. So you have to flush the data block onto the bus. So this is the third transaction. A small note here which is a variant which we will discuss later is that.

Now pay attention. Here the block is in state S and we get a write miss. So what this means is when the processor wants to write you already have the block. Now if you already have the block, why send a BusRdX? Because if you send a BusRdX, you are going to get the data and other copies will get invalidated. Well okay, you can always ignore this data which you got but can we do something better so that this data transaction is not sent unnecessarily. Okay. So alternative is instead of sending a BusRdX, I would send a bus upgrade transaction.

What does this mean? That this cache wants to move from S to M. So it wants to get promoted to state M. If we are in I and we want to go to M then definitely we have to send the BusRdX because we do not have the block and we want the data. But if we are in S we already have the data we only want to invalidate others and hence we can simply

say bus upgrade. So that is a variant we will discuss about this later.

But right now these three transactions come onto the bus. Now we will discuss the MSI FSM. So I will draw the states in one color and then we will discuss what requests come from the processor to this cache. What actions the cache takes. So that is one side of the FSM which I will draw using blue color.

And this cache when it is sitting idle if some transactions come on the bus that is some other cache wants to read or write it is going to snoop on the bus and take appropriate action. So these I will draw in red color. So I would also ask you that you pause the video at appropriate times and try to think and draw it yourself. So I will draw the first three states. Initially we have the state I where the block is absent then we have the state S and then we have the state M for modified.

These are the three states and now we look at interactions with respect to the processor. Now we are in state I. From state I, if we get a read from the processor, what are we going to do? We do not have the block so send a request on to the bus to acquire the block. So on a processor read request that is the input. If you put a forward slash and then we are going to write what is to be done.

So we have to send a bus read transaction on to the bus. Now what happens because of this we will get the data block and we will move to the shared state because it is only for reading, we can be in shared state and this also implies that there could be other caches in the system in the shared state. From I, if you receive a processor write request what will you do? You need the block plus you also need permission to write to the block. So what is the output? You are going to send a BusRdX transaction. So this is different than the previous one and because of this you would move to the M state.

So once you send the BusRdX other blocks will get invalidated and you will go to M. All right. Now from the S state within the S state, if you receive a read what will you do? You will have a self loop here on processor read ok and nothing has to be done. So the output is a hyphen that is nothing goes on to the bus you can continue to read the block. Within S if you receive a processor write what are we going to do? We are going to send the BusRdX. Right now I am not using the optimization in this, I will just say BusRdX transaction goes and we move to the state M. Within state M all the reads and writes will hit and we do not need to inform on the bus.

So processor read nothing to do, sorry processor write again nothing to do okay. With this we have completed all the processor read write interactions on this particular FSM. Now in the red color we will write what should this particular cache do if it receives a

BusRd or a BusRdX, okay. If we are in the state I, if we receive a BusRd or a BusRdX on a particular block, we do not have the block so we have to do nothing. So I am not going to draw those arcs. So nothing to do from state I.

In state S, if you receive a bus read what will we do? Nothing, but I will just note it here for completeness for I. I would not write these unnecessary transactions but here if we receive a bus read we are not going to change state and do nothing because no action has to be taken on a bus read. Then if you receive a BusRdX in state S that signal means there is a cache which wishes to write to this block, presently we have this block in read only mode, invalidation based protocol, so we have to remove this block, ok ay. So when we receive a BusRdX when in state S, we have to invalidate our copy, go to state I, any action to do ? Nothing, because the data block will be provided by the memory in this case. So this cache need not provide the data to the new requester. From state M, if you receive a bus read, state M, bus read says there is another cache wants to read this block they want to read we can also continue to read, okay.

So we can safely move to state S on a bus read, bus read comes, you can move to state S but we have more responsibility here that the latest copy is with us and so we need to send that copy on to the bus by the flush transaction. So on a bus read you still have to give the data. What happens if you receive a BusRdX while in M? That would say that I have a BusRdX comes when we are in state M. Here again we need to give the data because we have the recent copy. What should be our next state? Somebody wants to write to this block, so we cannot hold it anymore and we have to go to state I. So that is how the FSM gets built and here I have given a neater picture of the same thing which we worked out, okay. Similar exercise to the VI protocol, we will try to fill this MSI state transition table. Again I request you to pause the video, look at the FSM and try to fill it yourself.

So we will do it slowly during this lecture. Okay. So blue is this processor and yellow is the other processors in the system. If this processor sends a load that is a processor read request and we do not have the block, so we are in valid state what should we do? This is going to result into a cache miss, the block will get loaded and what is the next state? The next state will be S. You will get the block and move to the state S when there is a store from this particular processor. Even this results into a cache miss but here once you get the data you will move to the state M because we are going to modify the content. Here this is a cache hit and we are reading the block, we already have it in the S state, so nothing to do. Here this is a hit but the hit happens only on the block but it is in the read mode, so actually this is a read hit but a write miss. So when it is a write miss, we need to send the BusRdX or the BusUpgr transaction and then shift to the state M.

If in the modified state, nothing to do, both turn out to be hits. Okay. In the yellow processor that is other processors sending a load request, so another processor wants to load, it will send a bus read, we do not have the block in invalid, so do not do anything here. Another processor wants to read. We have the block in read only but we do not need to take any action here. Another processor wants to write, we have the block in shared so what will we do here? We need to invalidate our block, move to the state ,I no data to be provided, we only have the read only copy. Last row, another processor wants to read, we have the block in modified state, so here we need to send the data.

We have to send the data to the processor and change to state S because we can continue to read and the last row again you have to send the data but here in the last cell the other processor wants to write to this block so we need to move to the state I, right. So that is how we will fill this table and the neater version is given in this slide. Okay. One more example we have processor A and B and that is the order of the time. So this is the order in which things happen, so we need to list actions taken by the processor, what are the different states of the cache and so on when we try to run this example, okay. So let us start with load x. So when processor says, processor A says load x, what are we going to do? we do not have the block, so we are going to send A bus read, get the data and move to state S. On the store x we are going to send a BusRdX and then move to state M. Here on the blue load x what is going to happen? Blue load x will send a bus read transaction once it sends the bus read transaction it is going to get the data from whom in this case A will give the data and we will move to the state S.

Store x will send a BusRdX transaction. Once you send a BusRdX, the data is already with this processor but this processor moves to M and here the processor A goes to state I, right. So the same thing is listed here. So as the transactions happen or as the instructions get executed in every processor in the order of time we have to understand how the protocol will behave, okay. The same cache coherence example which we are familiar with first P1 reads u, P3 reads u, then P3 modifies u, P1 and P2 try to read u. So we are going to solve this coherence problem using MSI protocol. For this we are going to use this table and see if we can have a coherent behavior of the system, okay. So in the second column here we have written what the processors are trying to do and we will fill the remaining table, okay.

So when P1 reads u, what should happen? P1 reads u, none of the processors have the block, so state in P1, here what am I going to write here, what becomes the state. The state will become S. What is the bus action? We are going to send a BusRd and who gives the data ?memory. Similarly, when P3 reads u, memory gives the data because we send a bus read transaction and the state of P3 becomes S. P1 is already in S. When P3 writes u, it is going to send a BusRdX, we do not need the data, but memory will provide,

we will ignore the data. So I will just put a symbol here so that data is not important. We already have the data and the state of P3 will become M and the state of P1 will become I. When P1 tries to read u again, it will send a bus read. Who gives the data in this case? P3. So P3 is cache will give you the data. Once P3 is cache gives the data it moves from M to S and this one moves to S and similarly when P2 reads the data, it will send a bus read, who will give the data? In this case we can safely assume that when P3 sent the data on the broadcast medium, the memory was updated and hence memory can also give the data here and the state of P2 will become S. Okay. So that is how we can see that every processor is going to get the correct value of u.

Okay. So we have seen the MSI protocols FSM and informally seen how it works. Now we need to look at the correctness aspect of this protocol for correctness. As we have already discussed, we need to prove two properties, one is the write propagation and one is write serialization. Write propagation means that when a cache writes to the block every cache in the whole system is going to see this block, okay, see this write happening that is write propagation. Write serialization means that when the processor writes to the location or when two processors write to the same location in a given order, everybody in the system sees the writes happening in the same order. So if you write A then B, you should, everybody should see the values in that particular order, okay. So that is about proving the two properties of write propagation and write serialization. Okay. So first we will prove write propagation. Okay. Before I go into the detail proof an overall idea is that for write propagation that is for every cache to be able to see that this write has happened. Every write should go onto the bus because bus is the medium which will order everything. So a new writer will send a BusRdX, when this BusRdX goes, others will see that so and so processor is modifying the block, so they will invalidate their copies. So this BusRdX will essentially tell them that a write is going to happen. The writer will eventually finish writing into the block and may continue to write because it is a write back protocol. Later on a new cache wants to read this block, so what will it do? It will send a bus read in that case the latest values will be sent onto the bus, right. So this is the overall idea of write propagation. So we will discuss it properly here, okay. When there is a write miss happening into a processor, write miss means I am going to write to this block. So how do others know about it, because the cache miss happened in this processor, how should others know about it, right? So we could be either in state S or in state I and we will send the BusRdX transaction. So when this BusRdX goes onto the bus, the other caches will invalidate their block. So other processors will invalidate the block on seeing the BusRdX. When will they see the value of this? The current processor that is the writer will see the value after it finishes this transaction, that is it has got the block, it has finished the write that is when the actual value will get updated. And if I ask you, when will the other caches see this value? The other caches will be able to see this newly updated value only in future if they incur a cache miss, okay. So that is

about how the write miss will be made visible. What about write hits. Write hits is the several writes which will happen in this current processor. This particular processor is definitely going to see them in program order and the other processors will see them whenever they incur a cache miss again in the future, okay.

So we will explain this more with a diagram. So this is about write propagation correctness. P1 incurs a write miss. It incurs a write miss, so it goes there and then it goes onto the bus, sends a BusRdX transaction. What are we looking at? How is the write miss seen by others? So when a BusRdX goes, these are the three processors which will see this transaction happening, okay. So when they see this transaction, what will they do? They will invalidate their states, right? The states will get invalidated and in the future because the state is invalidated, in future they will incur a read miss and due to this read miss, it will eventually fetch the newly updated data by the processor P1, okay. So in case of a write miss, BusRdX transaction and the subsequent invalidations are going to prove the write propagation property. How do I prove for write hits? right. So P1 has a write hit, so this is not going to go on the bus. So it is happily changing the block locally, so the block keeps on changing here. P2, 3 and 4 have the block in invalid state. So they do not have the block at all because they have recently given it to P1 for writing. Now P4 wants to read this block, so it incurs a read miss and it sends a bus read transaction. What will happen? P2 and P3 will ignore it but P1 sees this transaction, it gives the data and then this data will be picked up by P4 and doing. And when it gives the data it changes its state to S, okay.

So all the several writes which P1 has performed, the final write will be put on the bus and P4 will be able to read that write, okay. So that was write propagation. Now write serialization, write serialization is, when I, suppose I do W1 and then I do W2 in this order. So everybody in the system should see the same order, okay. Nobody should see first W2 and then W1. So that is the write serialization property. So for write serialization, we are going to imagine a scenario that the cache has performed a write and when it performs this write it would have already sent the BusRdX on to the interconnect. Everybody has invalidated their copies and a write is happening in this cache. And eventually several such writes will happen locally and these writes will occur in the program order of this particular processor. So the write serialization with respect to this processor is guaranteed because it is the responsibility of the processor to do the writes in the program order, okay. Now all these several writes which we did in this local processor, they are not visible to the others but the last write, when it goes on to the bus due to cache misses of others will be serialized on the bus.

So what is the scenario? Initially the cache sent a BusRdX, acquired the block, did several updates and in future, another cache sent a bus read, so this cache flushed the

data on to the bus. So there were two bus transactions between the several write hits which happened and these two bus transactions are going to order the writes and serialize it globally for everybody, okay. So that is the overall picture. So all the writes appear as BusRdX. Exactly one valid copy is there and there are no others who have this copy anymore. The writes are performed one after the other, within this particular writers cache and after these writes are performed, this processor will know that they are completed.

So multiple writes can take place to the same location within the writer. So writer is going to do these multiple copies, multiple updates. And if you see this was the first write which went on to the bus and later there were several updates on this block. Subsequently, the final value say it was w3. This will go on to the bus, right. So these points where the bus transaction actually happens is the serialization point. So this processor P, who has done this write, will do these writes in the program order. So they are serialized and for other processors the new bus transaction will serialize it on the bus.

So we will also understand this with a diagram. Again we have these four processors and it is the same cache block the colors do not represent different blocks but colors say, it is the same cache block, every color match to a different processor. So initially R1, these three reads have already taken place on that block and now processor P3 sends a write. Now when this write goes using a BusRdX transaction, it is going to invalidate the green and the blue. It is going to invalidate these two. So w1 transaction is propagated because invalidation has happened. Then there will be local hits into the cache. These all are cache hits but these are all writes.

So I am going to do several writes to this cache block locally within P3. So P3 is going to do all these writes. Okay. And how will I say what is the serialization order? It is w1 then 2 then 3 then 4, because of the program order of P3. Later a w5 request, that is a write request comes from P4. So P4 wants to write, so in this case the pink processor has to give the data and it is going to invalidate the pink processor. Because others anyway do not have the cache block. Okay. So if you see these two writes which went on to the bus, they have been serialized. So first w1 went on to the bus and then w5 directly went on to the bus, and between these two bus transactions the write 2, 3 and 4 happened in the program order.

So this way we can infer that write 1 to write 5, all of them have been serialized with respect to the bus. Okay. Next is write serialized with respect to reads. This example showed that how the writes get serialized with respect to the writes and the local write hits.

Now writes serialized with respect to reads by other processors, right. So there is a processor P which issued the BusRdX request. It got the data it kept on modifying it in the program order. So within the processor both the reads and writes are serialized properly, because of the program order of P. Now another processor P2 wants to read this block and this block is with processor P. So to read this, there has to be at least one bus transaction which P2 has to do before it gets the data from the processor P. So the bus transaction, which P2 sends, will ensure that all the writes which are done by P will be now visible or available to processor P2, okay.

So reads by all the processors will see the writes in the same order. Okay. Again picture. Three reads have taken place. Now this processor P3 does a write, you know which all gets invalidated. Some local reads and writes. Now I have also put some reads ,local reads are also happening and we need to see whether the reads and writes are serialized properly. And in this case definitely yes, because this is the order in which it is flowing in the program order, okay. So W1, R1, W2, R2 they all follow the program order and get serialized properly.

Later the green processor tries to read. Now how is this read getting serialized? When this read goes onto the bus. It is going to get the data from the pink processor because the data was modified by the ping processor. So this pink processor gets the value of W2 due to the bus transaction and if you see the W2 and this particular R, this path is also serialized, right. So if you see this one, so this W2 and the R also get serialized. Subsequently if there is another write, again this R and the W3 also get serialized.

Overall the bus is going to order the reads with respect to the writes, all right. Okay. So I hope with this we have proved the correctors that is serialization and write propagation. One small point left out of this MSI protocol is about, is there any small design choice which we want to make in the protocol. We discussed this small thing in the first few slides about the bus upgrade transaction. Okay. So recollect that I have the state M and the state S. If we are in state M and we get a bus read request onto the bus, what are we going to do? We flush the data because we have the data block with us. We are supposed to provide the data. So I gave the data onto the bus.

Now once I give the data, I have a choice that should I go to state S that is remain shared or should I go to state I. Okay we have two choices. But we preferred that I will go to state S because the assumption by the designer is that if we have this block, I will continue to read this block. Okay, I modified it, somebody wanted it, so I gave the block, but I will retain a copy because I want to continue reading to this block and why am I retaining ? Because the other cache is wanting only for reading, other cache is not asking for writing. So overall that wants to read I also can read, so overall reads being more

frequent, we assume that let us go from M to S. That was my design decision. Now this holds for many cases, right.This is good enough. So this assumption is good enough because the reads are more frequent.

However this does not hold in certain special situations. Now what are the special situations when I am modifying flag variables or I am modifying shared counters. Now what is a flag variable? This processor P or rather there is a global flag variable which everybody is accessing or a lock variable if you can imagine. So P1 goes sets the flag and so when it sets the flag it wants the block for writing. Later P2 wants to write to this flag, then again P1 or P3 will want to access. So there will be several processors who will try to read and write in sequence. So this will be the pair which will happen and not several reads in a sequence, right.

So our assumption was if we continue to read let us go from M to S. We went from M to S because our assumption was several reads will happen. But the scenario I am discussing is, there will not be several reads, there will be one read write followed by another read write by a different processor ,okay. So when the access flag variables, P1 is going to read modify then P2 is also going to modify it one after the other without having several reads in the middle, okay. So here what will happen? if I go from M to S in my protocol, when the other processor wants to write, we will also need a S to I transition, okay. So we will have to do this extra work every time because of the given scenario and I want to avoid this.

Same problem will occur when I am modifying shared counter. So there is a counter kept in the memory and every process is going to do a counter plus plus. So it is going to read the counter updated. So there are several read write sequences will happen here in this example, okay. So in the case of flag variables, if you can see here P1, P1 does the read of the flag, it does a write of the flag, okay. So here it is in state S it moves to state M. Then when P2 does the read of the flag, this one happens like this when P1 also goes to state S because of this read it is going to send the bus read transaction. We move from S to sorry M to S.

Here P has to go to I from S whereas this moves to M. So this particular one transaction. So you see this one, this is an extra work we have to do due to this protocol, okay. And later again the same thing P1 wants to read again, it has removed this block, so it has to go to S. Then it goes to M and then P2 has to relinquish the M state to P1. So what is the challenge here? This is a migratory behavior. What we saw that the block keeps on moving from one processor to another for writing and our assumption that I can retain the block for a long time for reading, no longer holds, because of this migratory behavior of this particular block. The challenge or the extra overhead. What was the overhead?

The overhead was, we went from M to S, then we went from S to I. So we had this extra transaction overhead which was unnecessary, okay.

So now the question is this good enough or should I change the protocol, okay. But mostly several reads are going to happen and these are special cases. So there are processors which modify the protocol at runtime. So they will see the behavior of the current set of processes and change themselves for from M instead of M to S, they will go from M to I. So at runtime they will update the protocol. However as you see the designer has got several choices and depending on what choice you do, it will going to affect the performance of your system, okay. So with this we have finished their discussion of the MSI protocol. We have seen examples, we have proved correctness and also seen several design options which are available to a user. Thank you. .