Parallel Computer Architecture
Prof. Hemangee K. Kapoor
Department of Computer Science and Engineering
Indian Institute of Technology Guwahati
Week - 07
Lecture - 37

Lec 37: Request table and Organization

Hello everyone. We are doing module 4 on snoop based multiprocessor design. This is lecture number 8, where we are going to discuss the request table and the organization of the bus. Okay. We are continuing the topic of split transaction bus and in the split transaction bus there will be several requests which are outstanding. In any design there could be 8 to 16 requests which are pending at a time and hence we need to store them some somewhere. We also need a data structure to identify, what all things we need with related to a req, request.

So, these requests are kept in a table called the request table and this request table is with every processor or every bus controller associated with every node. Okay. So, what does this table contain or rather what should every request contain? So, every request has the address for which that request is going. What is the type of request that is BusRd, BusRdX, upgrade, okay, the type of request, a tag assigned to it when it gets the tag, again that tag information, so 3 things. Then the state of the block in the local cache that is not the requester, but all the processor nodes whenever they see this request they will keep a track of whether they have this block or not, so that information.

And of course the sender who is the originator of this request. Okay. So, all this information is stored in one location incorporating one request and we have probably 8 such pending requests in the request table. So, this request table which stores so much information in one entry. Okay. So, I have one entry in which all this information is kept and I have 8 such entries in the table. Anytime request has to be sent you have to check whether there are conflicts.

Whenever a response comes you have to compare the tag. Okay. So, we might want to compare the tag and the address every time. Okay. And how are you going to search this table? So, searching this table has to be done exhaustively because they are not indexed on the address or the tag. Hence, we treat it as a fully associative data structure. Okay. This is not a set associative, it is not direct map, it is a fully associative storage element.

And when you have new requests which come onto the bus, we have to make sure that

that particular request with that ID gets added at the same index across all the request tables. Okay. So, if I have these several processor nodes each of them has a request table. When the new request comes it gets added at exactly same location in all the request tables and it has to be because the tag index everything should match. Okay. We will look at an example. So, here all entries are to be examined for the match.

So, I am going to compare all these tables which you can see are the request tables with three different processors, the blue, the orange and the purple processor. So, you can see here, this is the tag. Suppose I use t1 is the tag, b is the address and this is the state state of the block in the local processor okay, and some more information will be there, but I am interested in the tag and the address. So, whenever a request comes it gets a tag, it is associated with an address. So, I have two requests already entered here and you can see t1 is occupying the first row in every table, t2 is occupying the second row in every table.

So, when a request comes, a request comes that request comes with an address it does not come with the tag. Okay. So, it comes with that address and so you have to compare here, the second field. When a response comes you have to compare the tag field because the response comes with the tag. Okay and if this table gets full what to do? It will not because we are only allowing 8 entries the 9th entry is anyway not permitted because only 8 outstanding requests will be granted, 9th request will not be granted that is how the table size is limited and if a response comes for a request, the pink line comes, that request is already handled, the response is done and so we can free up that entry. Right. For example, a response for t2 arrives, so the entry of t2 is deleted from all these tables. Okay.

So, you have one more row available to put future request. So, the t2 tag id will be reused by future requests. So, next we are going to see how does the bus interface and the request table get integrated. So, we are going to see the design of this. Before that a quick recap of how our organization was earlier.

If you recollect this figure of a single level cache with an atomic bus, at the bottom we have the atomic bus, at the top you have the cache, we had the bus side controller, the processor side controller, request from processors went as address and command, data came in the data buffer and snooping requests came in the address and the command, the snoop state was handled here. Okay, and there were comparators for the right buffer and for the cache. Okay. So, this was the design. Now inside this design we have to integrate the request table. So, the request table will come here and apart from this, the bus, the system bus which was one bundle of wires is now two bundles or three bundles of wire because we have split it into data bus and address bus. So, accordingly we have to add

more buffers to keep track of what is happening. Okay.

So, we will now try to make overall picture and then I will give you the detailed design in a slide. Okay. So, we have the request table which has got all these entries which we were discussing. There are seven entries, so the seven rows row number 0 to row number 7, inside this, okay, this can be treated as the tag, then you have the address and other entries. So, other information you have seven rows inside this request table. Then you have the write back buffer.

I am not drawing the cache which is at the top of this figure because we do not have space and you can recollect that the cache sends information downwards. So, the cache sends request, when the cache will send a request, it should go out onto the bus and a response will come from the bus. So, you need to store the response also. Now we have multiple request pending, hence we need these buffers. So, when this request comes, this is a new request generated by a particular node, it has to check whether such an address is already in this queue.

If it is then you are not permitted to send because it will conflict depending on the type of request. Okay. So, we need to check this before we can proceed. So, if you are permitted to proceed, you will go and put the request into the address and the command buses. So, you will put this, eventually you will get a tag assigned and then that tag will come on the bus and this tag gets stored into this table. Results of data come onto the data buffer, this data buffer is then connected to the cache. Okay.

Let us first draw the buses. So, we have this as the address bus and this as the data bus. So, now we have two buses address and data. The data bus, when it sends the response it will put the tag into this box and the snoop states because response and snoop happen together. So, snoop is related to the data bus and so it goes up here.

Then any request and response coming from the bus, from here. So, anything which comes from the data bus goes as a response to the processor. So, this goes as a response. This is a request and pardon me this arrow is not correct, I am going to erase this. This is a response coming from the cache.

So, let me redo this portion. This red one, this is the request coming from the cache and this is a response coming from the cache. So, your cache is going to send or write back a data block or give some response. It is coming in the downward direction. So, the red is coming in the downward direction, the blue is the data bus which takes data inside the processor. So, here the, this is where the data will enter the processor. Okay and then we have associated other buffers and comparators and so on.

So, overall the picture is that we have to add the request table, more comparators here because you have to compare addresses, tags with the write buffer with the cache and split the bus into address and data and add associated further buffers to the system. Okay. So, these extra things will get added to the base organization to make it a split transaction bus. Okay. So, now this slide is showing you the neater diagram. Even here the cache is not shown, cache is still above the contents of the slide. So, cache is coming from top here. Okay.

So, you can see the request table in the center, the write buffer. At the bottom you can see the two buses, address bus and the data bus. Appropriate arrows are showing you what bus is carrying what type of an information. On the right hand side you can see processor side requests coming, processor side responses coming. On the left hand side you can see that the data bus is sending the tag and the snoop information which goes up towards the processor on the left and in the middle we have the request table.

Request table is connected to the processor requests because to check whether we are allowed to issue the request or can we merge the request and so on. Okay. So, this is how the bus interface is adapted to handle split transactions by adding a request table. Right. So, with that organization now we will do the other discussion about when do I present the snoop results. Okay. So, we have been discussing that let us do it at the response phase and what are the snoop results. The three wired-OR signals shared, dirty and the third one is inhibit line which says that do not read these two until the inhibitors enabled.

So, the next question to answer is when do I present the snoop results. Because we are going to follow variable delay snooping and the request has already gone. Once the request goes every processor has already decided whether it should take action or not take action, it has determined its local state of the cache block and so on. Only thing the answer has not come. Okay. So, the three wired-OR signals shared, dirty, inhibit so we know about them but should we send them right now as part of the request or not. So, we decide that let us not send it now but send it at the time of response.

So, if you are doing it that time but your computation of the snoop has happened during the request. So, the processor has seen the request, computed the snoop answers but it is not allowed to send them. So, what should it do? It has to store these answers inside the request table. So, the snoop answers are kept in the request table with the appropriate tag. In future whenever a response for that tag passes onto the bus,oaky, so what should you do? You have to go check the tag and whatever are your snoop answers they have to be put onto the bus on these three wired-OR signals depending on the matching of the

request and the response IDs. Okay.

So, for snoop results we will do it during the response because it is going to take many cycles before you respond. So, you have to store that information inside the request table. Okay. So, once you decide you keep that in the correct entry, you send it later whenever there is a matching snoop happening on a particular message ID, at that time you have to send the snoops on the bus. Okay. So, snoop results are computed during request, but sent during the response. In the meantime they are stored in the request table.

Second is how to handle conflicting request. Conflicting request is you have one write and probably others reads and writes. It is very easy to do this with respect to the request table because you know what all requests are pending, you also know their addresses. So, whenever a new request comes for another request which is already in the pending queue, you can compare that the address is same and the type of request is conflicting. So, you can simply avoid conflicting request. Okay.

So, no new requests are issued for a block that has an outstanding transaction which is conflicting and this is easily possible by checking the entry of the request table. Right. You can easily check the request table. Is the address same? Yes, it is same. So, this is the type of request, if both are read, permitted, if one is read the second is write, not permitted. So, this is how we will handle conflicting requests.

So, even though the bus is still pipelined, what we are saying is operations on the same location will be still serialized by the bus similar to the an atomic bus. Okay. So, we are treating the split transaction bus as an atomic bus with respect to a given location, but in parallel many different locations can still continue their transactions. Serialization is guaranteed by commitment aspect, that is once the transaction gets on to the bus, we can say that the transaction is guaranteed to finish. Okay. So, the next aspect is flow control. So, what is this? I have shown on the slide two processors and connected to single level caches and the bus and processor sends request to the cache, cache sends response to the processor in the form of data.

So, responses are mainly in the form of data and this whole block of data 32 bytes, 64 bytes, 128 bytes all of this has to be stored into the buffer before the processor picks it up. And if I permit multiple such requests pending, you will have multiple responses going up and hence the buffer size will be bigger. So, you need to worry about how big a buffer can you afford. Okay. So, because it is going to occupy power and area on the chip.

So, we need to handle this. Another issue to handle is, if the buffer becomes full, what to do. So, that is the aspect of flow control. We have a similar problem in the case of the cache talking to the bus that is the bus snooper snooping on to the request going on to the bus. We have eight pending requests. So, all of them are buffered, probably responses are going down.

So again, lot of buffer between the cache and the bus has to be handled. Another problem is at the memory. So, memory is also sending requests, it is also sending and receiving responses. Okay. Send and receive is happening and now imagine a scenario ,I have eight pending requests total in the split transaction scenario. All the processors, suppose the all eight requests are for write back. Okay.

So, P0 is doing write back, P1 is doing write back. Every processor is sending the data and if you accept all these eight requests, their requests are coming with the data block and they will all go together to the memory. So, memory should have enough space to keep all these eight blocks and then write them to the memory banks in parallel or later on. Okay. So, in such a scenario you will have an overloaded memory if every processor tries to write to the memory.

So, we again need flow control at the memory. Right. So, we have two three scenarios to handle flow control. One is between processor cache, one is between cache and the bus and one is between the bus and the memory. So, at the cache control as we discussed you have incoming requests coming from the processor or from the bus. So, you have to manage the storage because every time the address as well as the complete block of the data has to be kept into the buffer, yet you want to have the buffer as small as possible. So, how do you manage this? So, I will manage the flow control at the cache by limiting the number of outstanding requests, that is how many requests can the cache handle.

If the buffer can handle four data items storage it will say I will only handle four requests. Okay. So, we are going to handle this by limiting the number of outstanding requests. Okay. So, what happens at the memory flow control? We have eight outstanding bus transactions are permitted and if each of them generates a write back then we need enough space at the memory to handle this. So, we will see how to do this. So there are two variants implemented by these two systems.

In the SGI challenge both the data and the address bus use the NACK line. So, they use negative acknowledgement to stop some action from happening. So, new transactions can be stopped by sending a NACK. So, if the buffers are full at the memory, the memory will raise the NACK signal. Memory raises the NAKC signal and hence the

transaction has to be canceled because there is a sender sending a block to the memory.

Memory says do not send me the block. I am, my buffer is full. So, the sender has to cancel its transaction. Okay. Transaction has to be canceled and later retry. So, sender has to again try sending the block in future.

Again you have problems of starvation. So, you can associate back off and priorities. So, that eventually the data block will be written to the memory. This is how SGI challenge solves it. So, what does Sun Enterprise do? It says that unlike SGI challenge where the sender retries, Sun Enterprise says the receiver will indicate whenever it is ready to receive the data. So, this way we are going to reduce the traffic and so the destination, that is the receiver will initiate the retry.

It will say that yes, now I have space you give me the data. So, what about the giver that is the source? The source will keep on watching when this retry request comes and when it comes it puts the data. What is the good point about this? That whenever a retry happens there is guaranteed space available because the receiver itself is saying give me the data and hence a single retry will make sure that the data transfer completes. Whereas in an SGI challenge the memory said do not give me the data. So, the sender keeps on trying in future until memory receives the data. So, this will end up having more retries whereas Sun Enterprise says in a single retry we can make this happen. Okay, right.

So, this is how flow control is managed. So, we have seen the implementation with respect to the request table and we stop this lecture here. Thank you so much.