Cache Coherence Tutorial

The cache coherence protocol described in the book is not really all that difficult and yet a lot of people seem to have troubles when it comes to using it or answering an assignment question. I think that it's just because you are given the complete picture all at once and it is discussed and everything seems to be understandable. Then when the question comes you're not ready for it. So what we are going to do here is start with the specification table of what happens for each situation and we are going to build the diagram from scratch.

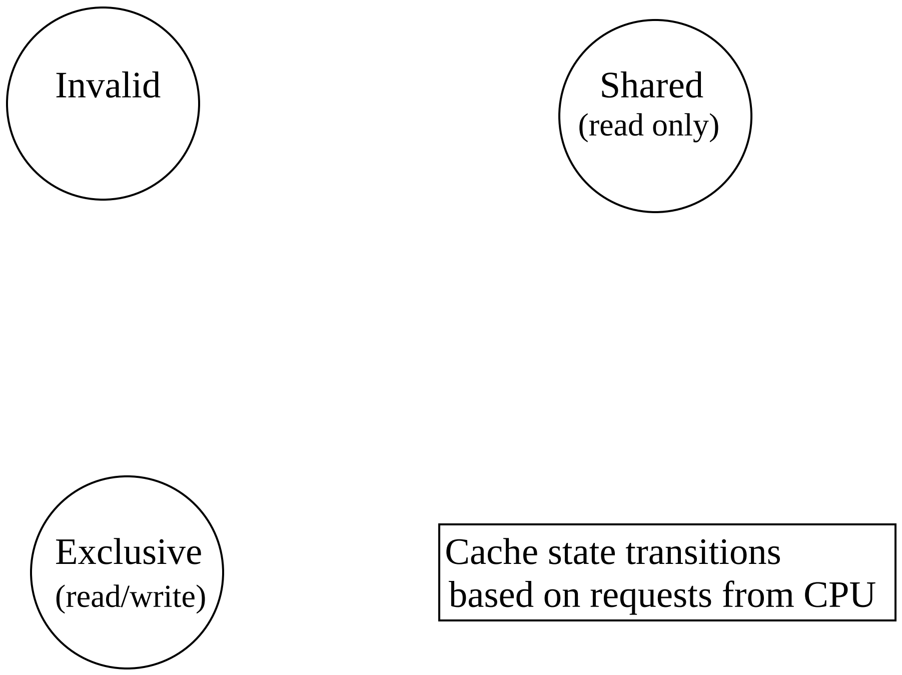
The scenario is the snooping bus protocol. This is basically a shared memory multiprocessor environment. Whether it is truly a shared memory or a distributed memory environment is immaterial to the concept, the memory is considered to be just one large memory area and addressing is used to differentiate various separations of the memory. Each processor includes its own cache which will contain at times copies of the data that is or should be in real memory (by "should be" I mean that a write to a memory location could be designed to just put the new data into a cache location and invalidate all other copies without actually updating the real memory location, on the assumption that it will be updated later.) So after the system has been working for awhile we could have the situation of the real memory having numerous locations with valid data in them with multiple copies spread out over the various caches of the processors in the system and also certain cache locations in various processors that have the only valid data for a certain memory location and not even the real memory location has a valid copy. This does not present a validity problem because each memory address sent out on the bus is received by every single cache in the system and the real memory as well. Either a cache will respond to the request for a memory location or the real memory will be used to satisfy the request.

The cache will be set up in a standard memory hierarchy such as is described in chapter 5 of the textbook. Whether it is directly mapped or set-associative is not important to this discussion so we will treat it as direct-mapped. Initially consider the size of the cache block (also called line) as being one memory address wide so you don't have to deal with multiple write situations for a cache line in different processors *(think about this one)*. The state protocol of the book, figure 6.11, can actually be represented in the cache by adding two bits to each cache line and these two bits will be manipulated according to the rules set out in the table of figure 6.10. Note that there are no extra bits attached to the real memory, only the caches. This diagram would apply mainly to a data cache, an instruction cache would be similar except that there would be no write operations involved.

Request	**Source**	**State of addressed cache block**	**Function and explanation**
Read hit	processor	Shared or exclusive	Read data in cache.
Read miss	processor	invalid	Place read miss on bus.
Read miss	processor	shared	Address conflict miss: place read miss on bus.
Read miss	processor	exclusive	Address conflict miss: write back block, then place read miss on bus.
Write hit	processor	exclusive	Write data in cache.
Write hit	processor	shared	Place write miss on bus.
Write miss	processor	invalid	Place write miss on bus.
Write miss	processor	shared	Address conflict miss: place write miss on bus.
Write miss	processor	exclusive	Address conflict miss: write back block, then place write miss on bus.

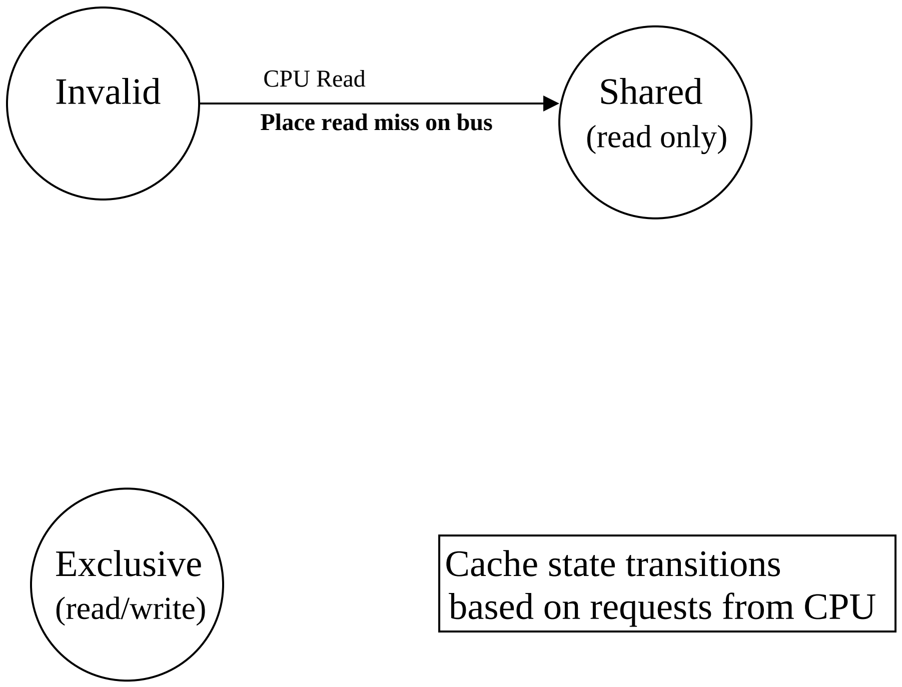
Partial copy of figure 6.10

We will create the diagram for the processor modifications to the cache states first and then deal with what happens at the other end, when requests come from the bus which actually means what other processors have sent out from their cache control to our cache control. We start with just the three states, without any transitions.

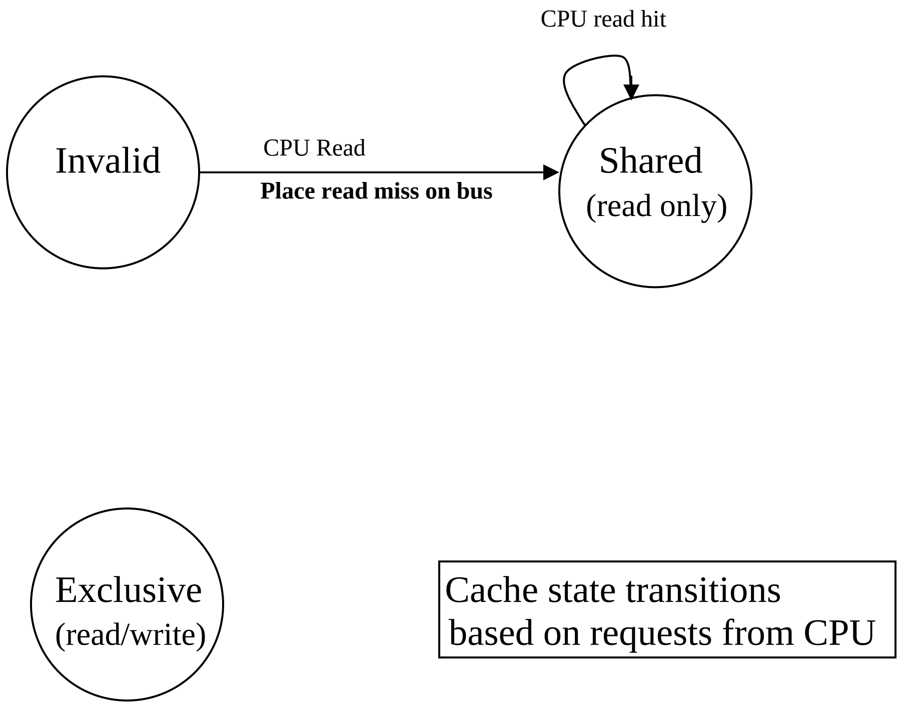


This can be represented by two bits attached to the cache line. The actual value of the bits, one or zero, is unimportant since the transitions will know them and change them properly. We will now use the entries of the table to build the transitions onto the diagram.

First assume that we start with empty caches on all machines, the real memory has a program in it that will be executed by the various processors. So the very first request for our processor, assuming our processor will start first and will cause the other processors to start up later, will find the cache for the memory location to be **invalid** and hence will generate a Read miss that will cause a ***place read miss*** to go out ***on*** the ***bus***. All the other caches will be in the **invalid** state so the real memory will respond with the data and return it to this cache putting it into the cache location and the state of this cache line will now be **shared**. Our diagram now looks like this:

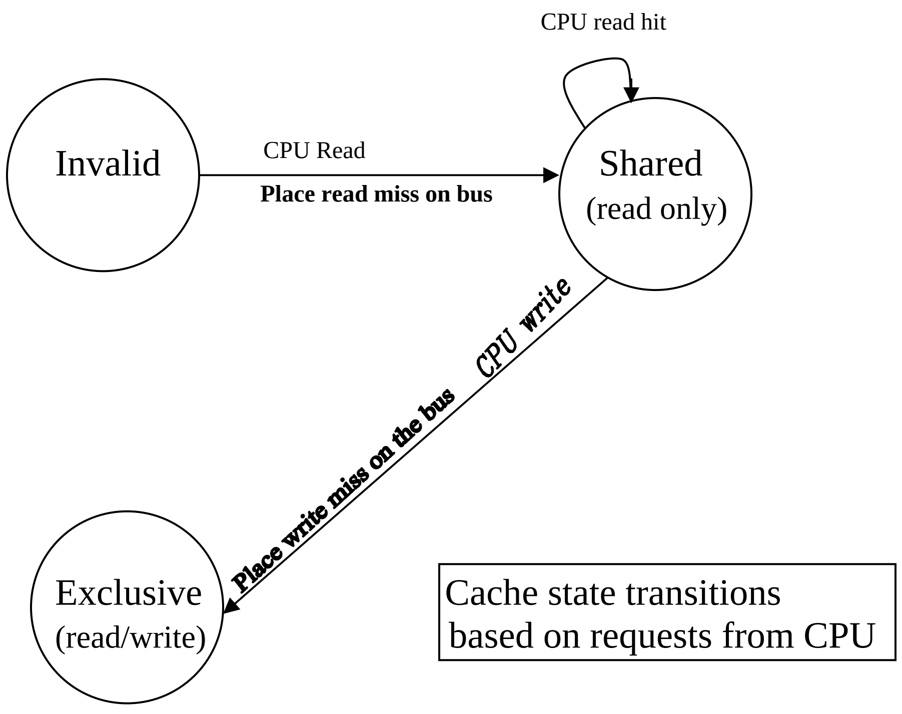


This will happen for a few locations as the program starts to execute and continues. At some point the same data location may be requested again. Now we left it in **shared** and no other processors have started running yet. So now we will have the situation of a *read hit*. The request from the processor will find the data in its own cache and it will use it directly, no access to the bus is involved.



So now we've done two reads to the particular location, the first time it had to get the data from real memory, the second time it found it in the cache. Now we will see what happens when we write to that location. Remember the cache line presently has the location in the state of **shared**. The write means that there will now be a new data value for that data location. This means that the state of this cache line will now change from **shared** to **exclusive** and the data will be put into the cache line. It also means that any other cache that also had a copy of the old data will no longer be valid so the ***place write miss*** signal goes out ***on*** the ***bus*** to inform them of this fact.

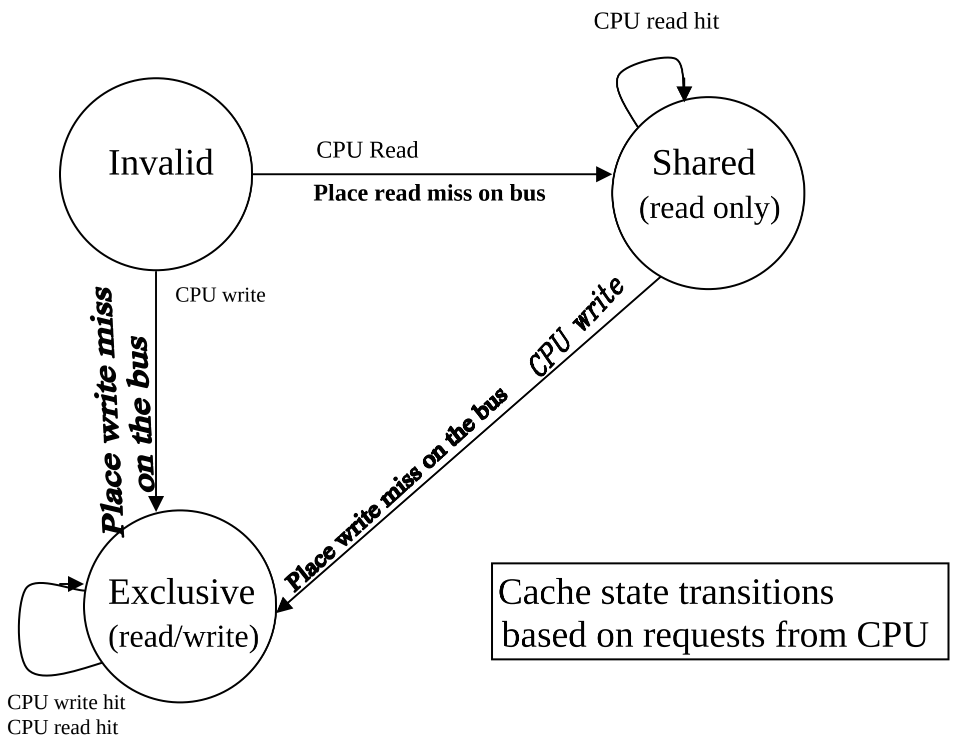
There's a complication that has to be mentioned at this point, when the cache line is actually larger than one memory address which would normally be realistic, then another cache in another processor may have modified another part of that cache line. This means that the data we are writing has to be merged with the data written from the other processor. This means that the data from the other cache has to be read into our cache before we update our part of the line. Putting the signal ***place write miss on bus*** will cause this to happen and then the data that we are writing will be put into its part of the cache line and our cache becomes the **exclusive** owner of the memory location. The real memory may or may not have been updated during this process but the end result will be sure; all the other caches will have invalidated any copies of the location and we will now have the only good copy in our cache.



If the state on our cache for the memory location had been **invalid** this does not means that other caches were **invalid** also. It is possible that other caches could have been in the **exclusive** or **shared** state for the memory location. This means that the same effect would have had to be initiated. The ***place write miss on bus*** signal would have had to be sent out, any **exclusive** copies would have had to be transferred to our cache and all **shared** copies would have to have been invalidated and then our new copy of the cache line would be updated with the new value and the state would become **exclusive**.

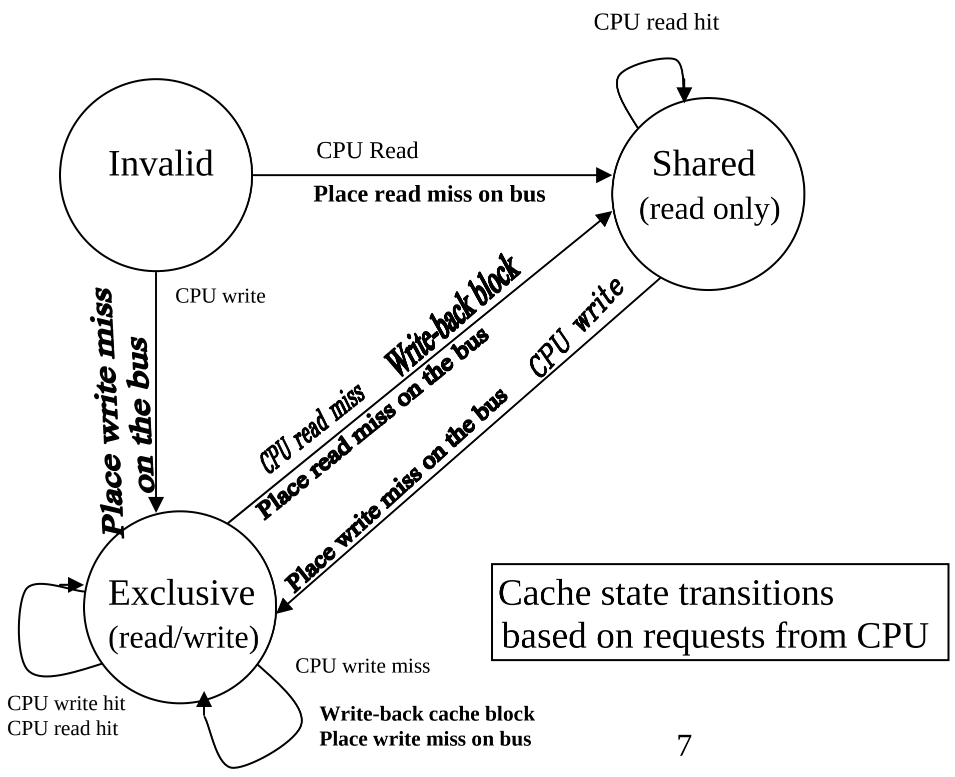
If the state on our cache for the memory location had been **exclusive**, then it would be easy. We would have a cache *write hit* on the location we are writing and the data would just need to be replaced with the new data from the processor. We are already in the **exclusive** state so we know that the cache line that we have has the most valid data of all the caches and memory in the system.

The same thing is true if we have a *read hit* when the cache state is **exclusive** for the particular memory location requested. We will stay in the **exclusive** state for this cache line and just use the data from the hit. Now the state diagram looks like this:



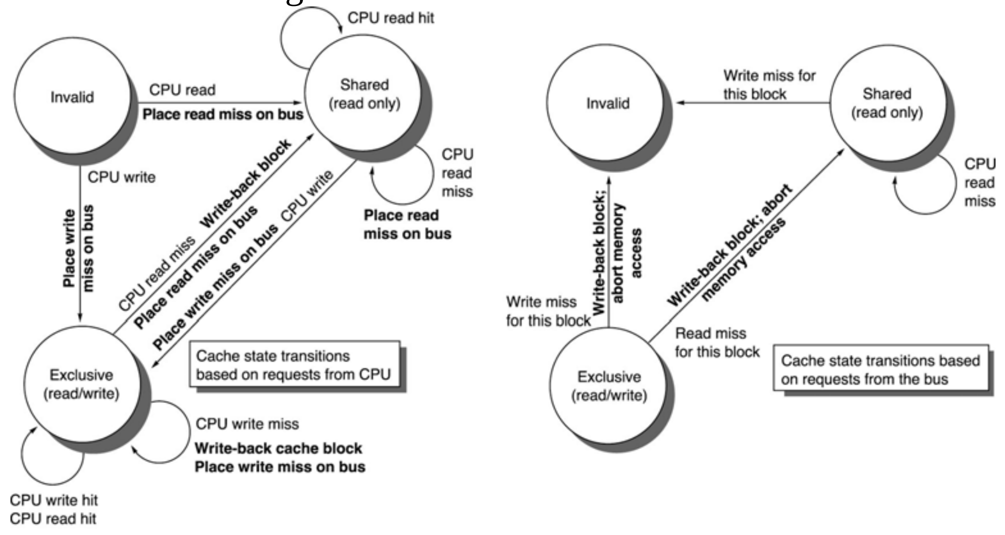
So far we have done two reads and one write to the same memory location and possibly we have done other reads and writes to other memory locations also. We have also looked at other situations involving the exclusive state that were fairly simple. Now we have to consider the problem of what happens when the line that is in the cache is not the one that the processor wants. This raises two possibilities, either we have to save the old contents before we use the cache location or else we just throw the old data away.

If we have the **exclusive** copy of the old data then it must be put back into real memory before we can use the cache line for the new data. This is done by doing a ***write-back block*** operation and then doing a ***place write miss on bus*** operation or a ***place read miss on bus*** operation depending on whether the data is being written or read. This will cause the old data to be put into real memory in both cases, read or write. Then the new data has to be obtained and put into the cache line and finally the state of the cache line has to be set according to whether the operation was a *write* (**exclusive**) or a *read* (**shared**). When the cache line desired is already occupied by another memory location data value then this is what is considered an *address conflict miss*. The old data has to go (without losing it) and the new data has to come in. The diagram now looks like this:



Remember that ***place write miss on bus*** and ***place read miss on bus*** cause different things to happen. The read miss will just cause the contents of real memory or the contents of some cache somewhere that has the **exclusive** state set for the particular address to be returned back to our cache. The ***place write miss on bus*** will not cause the old data to be written back, this is done by the ***write-back block*** operation beforehand. The ***place write miss on bus*** will cause the same thing as mentioned previously for the *write miss* operation on the shared or invalid states. It will go search for an **exclusive** copy in another cache and retrieve that or it will retrieve the block from memory and put it into our cache and it will invalidate any other cache lines that have that memory location marked as **shared**.

There is only one transition left that we have not mentioned. This is when the cache line is in the **shared** state but the address that we want needs to go into the same cache location because the memory addressing overlaps. In a direct mapped cache this always causes a replacement, in a set-associative cache there is an algorithm that determines which line in the set will be replaced. The old data does not need to be retained because it was already **shared** in other caches (by virtue of this cache having it in the **shared** state) and real memory. So all that is required is that the new data value be obtained from real memory or another cache that has that memory location marked as **exclusive** (remember that the old cache line was **shared** but the new cache line we have to snoop to find out where it is. But the state stays **shared** because the new line will become **shared** everywhere by the nature of the fact that we did a read of the memory address. The final diagram is the same as figure 6.11 in the textbook.

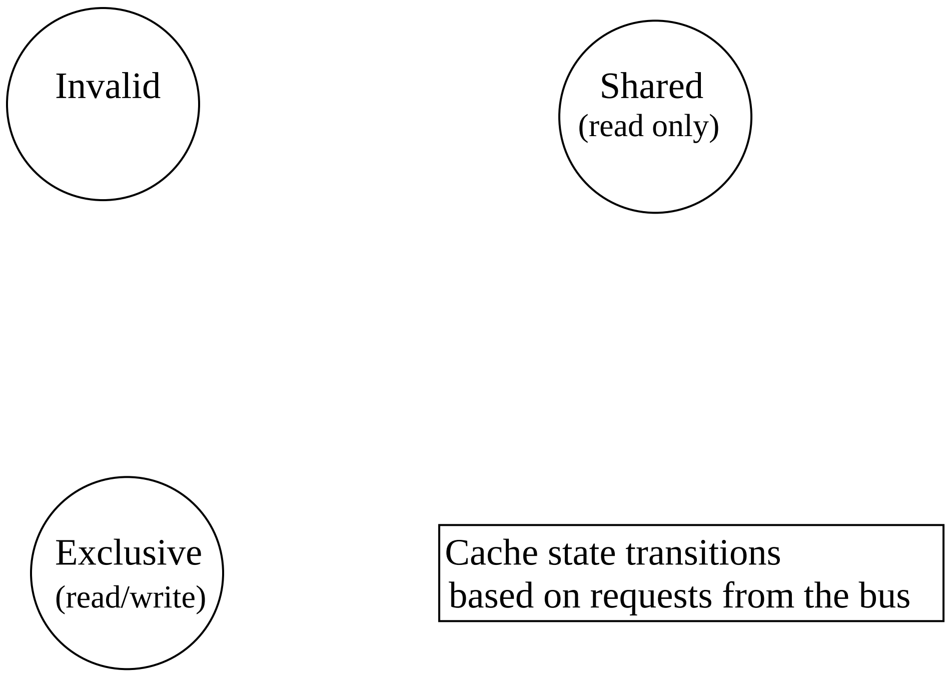


The diagram on the left is the part that we have done. This is the transitions that occurred in the cache line states from the requests of the processor. We took the viewpoint of our processor and cache and all the other ones were outside on the bus somewhere. Now we are going to take the viewpoint as one of the caches out there on the bus. Now our cache, we're still the same cache just on the other end of the requests, will receive a request from the bus and do the transitions based on these requests.

I suggest as an exercise that the student tries to build the right hand diagram from the table before continuing. Then use the rest of this tutorial to confirm your results.

Request	**Source**	**State of addressed cache block**	**Function and explanation**
Read miss	processor	invalid	Place read miss on bus.
Read miss	processor	shared	Address conflict miss: place read miss on bus.
Write miss	processor	invalid	Place write miss on bus.
Write miss	processor	shared	Address conflict miss: place write miss on bus.

The rest of figure 6.10 for requests originating from the bus

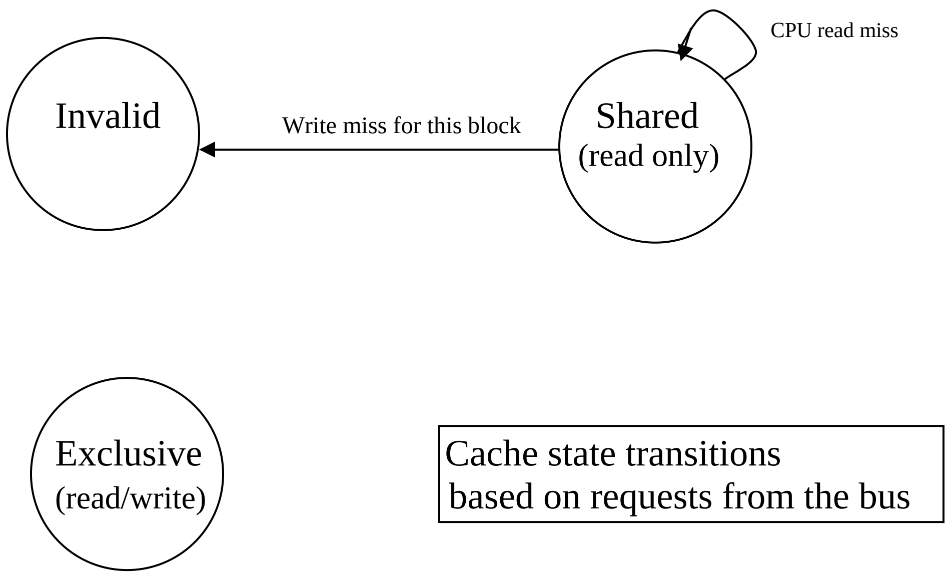


Note that we are still dealing with the same two bits that are attached to each cache line in our processor’s cache. Now we only deal with three types of requests occurring from the bus, a read miss, a write miss and a write-back block. Be aware also that when these signals appear on the bus there is also the address that exists on the address lines of the bus. This address is compared to the address identified in the cache line to see if this cache is actively concerned with the present transaction.

The first situation is if the cache line concerned with the address presently on the bus is invalid. This is easy, because you will never get a match and nothing will happen in our cache. The second situation, similar to this, is if the cache line concerned with the address presently on the bus is in the shared or exclusive state but the address does not match the address of the cache line presently resident. This will also not have anything happen.

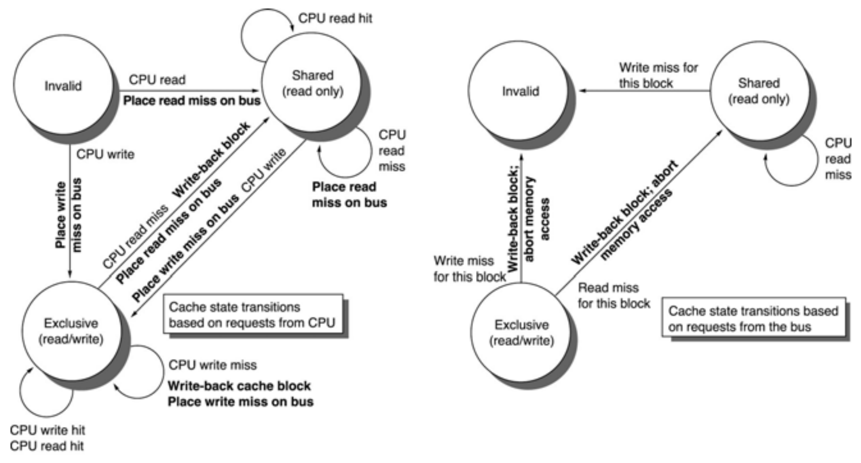
So the next situation is what will happen when our cache is in the shared state. If a read miss is requested and the address matches that which is on the bus, then not much happens. The data will be obtained from real memory for the other processor (unless some more elaborate data acquisition scheme is devised where it can obtain the data from a nearby processors’ cache, but we won’t consider that) so this cache will not have to provide anything. This cache will just remain in the shared state and that’s all. Same thing for a read miss on bus request but the address on the bus does not match the address in the cache line that it is associated with, the cache line just stays as it is and the state remains as shared.

For a write miss being on the bus with the address matching one of our cache lines then a very simple transition occurs. The signal means that another processor is doing a write to the memory location so the data in our cache line will no longer match the data for that address. The cache line will be changed to the invalid state and nothing more needs to be done. So the diagram now looks like this:

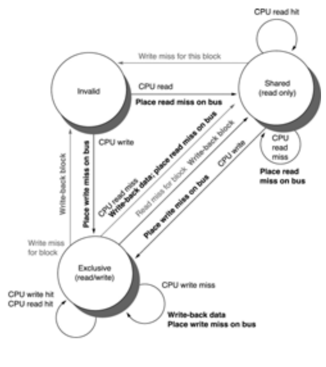


When the cache line state is exclusive and the address on the bus matches the address of the cache line then a little bit more happens. The exclusive state designation indicates that this cache line in this processor has the most up-to-date data in the cache line. This is only of concern if the address on the bus matches the cache line address because we need to put the data onto the bus. If the address is different then even if the cache line is where the address would go, we leave the cache line alone, because the address on the bus is being requested for the purposes of another processor and our processor doesn't need to get involved. But if the addresses do match then the data has to be put out on the bus and the state of this cache line has to be changed. A read miss indicates that another processor needs to use the data. This means to our cache; share it. So the data will be put out onto the bus and the real memory and the requesting processor will obtain a copy. Our cache will still have a valid copy but now it is shared so the transition will be to the shared state. Note in the diagram of figure 6.11 it states "Write-back block; abort memory access", the abort memory access refers to the fact that the data in memory is not valid because the data in this cache is the most up-to-date. What is aborted is the corresponding read of the memory location that now turns out to be useless. But the write-back block operation does put the valid data back into the real memory location.

On a write miss where the address on the bus matches the address of one of our cache lines then our cache has to go into action again. First it must put the valid data from the cache line on to the bus so that the other processor can get it and maybe also the real memory can make a copy although it is not really necessary since the other processor will have the data marked as exclusive. Then the state will transition to invalid which marks this cache line as invalid. The reason for this is that the write miss signal being on the bus means that another processor is doing a write to the memory location within the cache line range of addresses. We have to send all the data we have for the cache line over to the other processor so that it can add its portion and keep the new data as its own exclusive copy. Now the diagram of figure 6.11 is complete for both sides.



The two sides are joined together in figure 6.12 and it looks very complicated and easy to misinterpret. But having built the table into a diagram it should be an easy process to reconstruct it if necessary.



CPU read hit
Write miss for this block
Invalid
CPU read
Place read miss on bus
Shared (read only)
CPU read miss
Place read miss on bus
CPU write
Place write miss on bus
Write-back block
CPU read miss
Write-back data; place read miss on bus
Read miss for block
Write-back block
CPU write
Place write miss on bus
Write miss for block
Exclusive (read/write)
CPU write hit
CPU read hit
CPU write miss
Write-back data
Place write miss on bus