1 Precheck

|1.1| Each hardware thread in the CPU uses a shared cache.

|1.2| The dirty bit signifies an address line in cache that has up-to-date memory, but main memory at this address is out-of-date.

|1.3| Given a multi-level cache setup that has n bytes worth of storage, we're always able to utilize all n bytes for distinct data.

|1.4| Cache state information, like cache line tags, are stored independently from the data cache.

2 Writes

|2.1| When it comes to writing data to cache memory, there are multiple write policies to consider that offer different options when building our system. Some of them you might encounter are:

1. **Write-through**: Write through: In this policy, when we have a write we write to both the cache and the memory. This is the case for every write, so the main memory always has the updated data. This is simple to implement, but writing to main memory every single time is slow.

2. **Write-back**: On a write, the data is only updated/written in the cache. The main memory only receives the data upon eviction. This means the cache has more up to date data most of the time. While this is faster as there is less accesses to main memory, it is harder to implement as we have to include more overhead, such as dirty bits and so on.

3. **Write-around**: Data is only written to main memory, and whenever we do so we invalidate the old data in the cache.

Another thing to consider is what we do when we have a write miss. For that, we have 2 possible policies:

1. **Write-allocate**: On a write miss, we pull the block you missed on into the cache

2. **No write-allocate**: On a write miss, you do not pull the block you missed on into the cache. Only memory is updated. On a read miss, we still pull the data into the cache.

Considering the above information, lets consider a direct mapped, no write-allocate write-through cache with a capacity of 8B and a block size of 4B. Lets also assume that the memory addresses are 8 bits each. Assuming the cache is completely empty in the beginning, we make memory accesses to the following locations:

- 0x6A, Write

- 0x85, Read

- 0x6B, Read

- 0x87, Read

- 0x68, Write

With the above memory access pattern and the given cache configuration, how many times do we access the main memory?
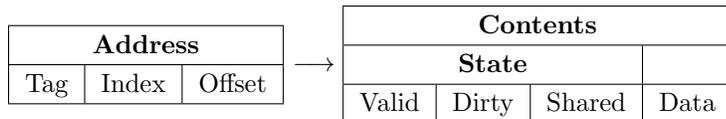
|2.2| Lets say for the same cache size, memory accesses but the only difference is that we have a no write-allocate write-back cache instead. How many memory accesses to the main memory do we have in this case?

|2.3| For one last optimization, we decide to use a write-allocate cache instead. So, now we have a write-allocate, write-back cache. How many times do we access the memory now?

3 States

3.1 Parallel processing allows individual cores of a CPU to operate as independent units with their own caches. However, for this to be the case, the machine must be able to coordinate the information flow of all cores and all caches so that this information is reliable to some degree. Therefore, we impose **cache states**, composing of the **valid**, **dirty** and **shared** bits, to denote status of the cache data at a specific cache block. These cache states are used when there is a cache **miss** or **write** to a certain core's cache so that if the information is modified in one place, the other caches are informed. In summary, we don't want two caches with different data both saying that they have the most up-to-date data, because that simply can't be true. In other words, from the perspective of the **host processor**, their cache line states may be update due to actions taken by **proxy processor** execution.

Consider this visual representation of the addressing of a cache block and the updated construction of the block itself:

Address		
Address		
Tag	Index	Offset

$\longrightarrow$

Contents		
Contents		
State		
Valid	Dirty	Shared	Data

Each state describes a specific set of conditions, on a **single cache block**, in respect to the overall memory system(all caches and main memory). These conditions are listed below, your job is to pair all appropriate conditions with their corresponding state.

Note: these conditions can apply to multiple states, therefore pick all that apply.

(a) data in host cache up-to-date

(b) data in main memory out-of-date

(c) data in main memory up-to-date

(d) dirty bit = 1 in host cache's line copy

(e) no copies exist in other (proxy) caches

(f) copies may exist in other (proxy) caches

(g) write will not change cache line state in host processor

(h) access from processor will result in a miss

1. **Modified**(M)

2. **Owned**(O)

3. **Exclusive**(E)

4. **Shared**(S)

5. **Invalid**(I)

4   Coherency

To be able to utilize multiple threads simultaneously, the hardware of your machine must be able to keep multiple caches in check so that there is no conflicting data. The diagram below represents the transitioning between states based on the reads and writes to that particular block of data.
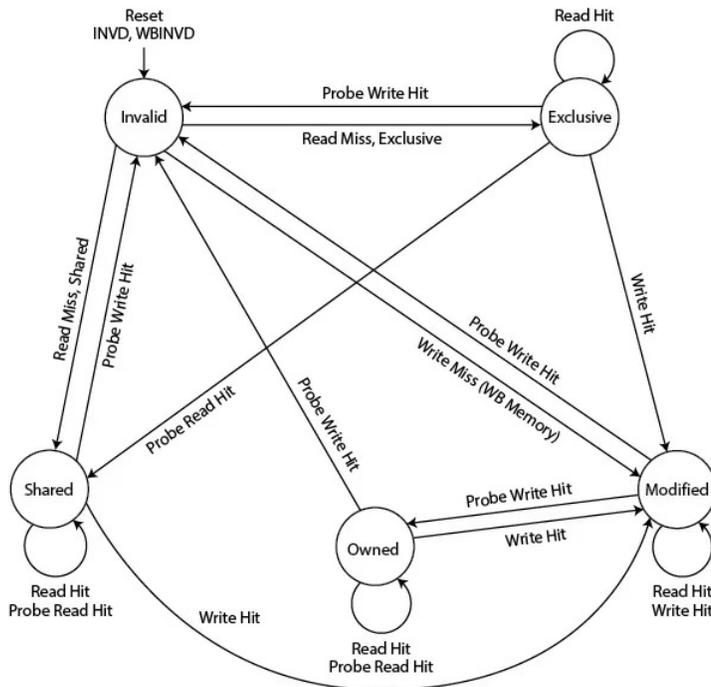
Note: A "probe" read or write indicates an action taken by a proxy processor; i.e. the action is taken by a processor other than the one we're examining.

In the following problems you will assume that the system is implemented with a **write-back** policy and you will be given a specific coherency protocol. Use the provided tables to fill out the processor states for each cache write or cache read.

1. Memory Address 0xDEADBEEF in Processor 1 is read. Hit or miss?
2. Memory Address 0xDEADBEEF in Processor 1 is written to with integer 10.
3. Memory Address 0xDEADBEEF in Processor 3 is written to with integer 50.
4. Memory Address 0xDEADBEEF in Processor 4 is read. Hit or miss?

4.1   Fill in the following table with the corresponding processor states according to the **MSI** protocol for each step below.

Processor 1				Processor 2				Processor 3				Processor 4			
Valid	Dirty	Shared	State	Valid	Dirty	Shared	State	Valid	Dirty	Shared	State	Valid	Dirty	Shared	State
1				1				1				1			
2				2				2				2			
3				3				3				3			
4				4				4				4			

Cache 1   Cache 2   Cache 3   Cache 4

Interconnection Network

Memory   I/O
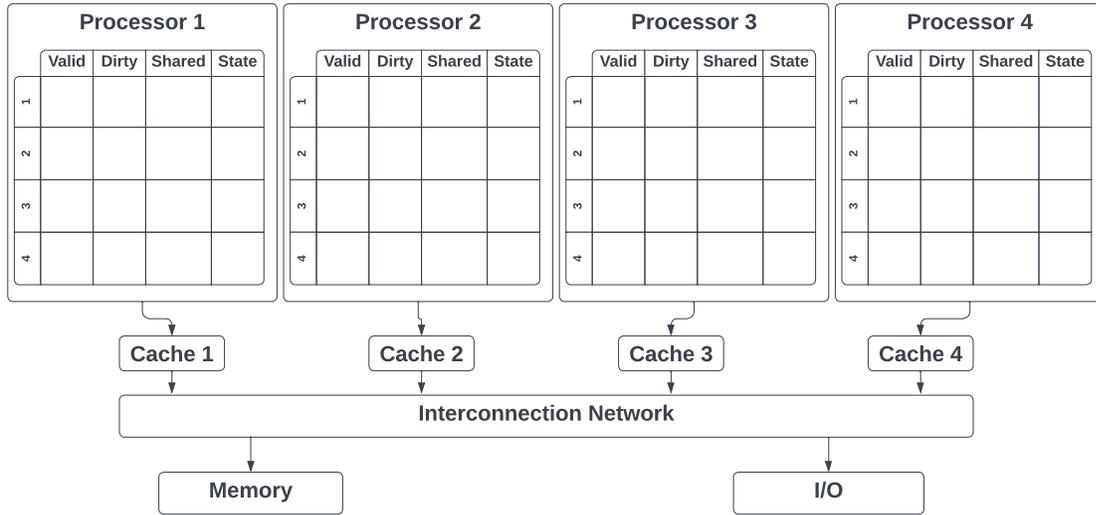
4.2   We now modify the MSI protocol by adding an **Exclusive(E)** state that represents the state where a cache is the **only** cache that has seen this data. Fill in the following table with the corresponding processor states according to the **MESI** protocol for each step below.

Processor 1				Processor 2				Processor 3				Processor 4			
Valid	Dirty	Shared	State	Valid	Dirty	Shared	State	Valid	Dirty	Shared	State	Valid	Dirty	Shared	State
1				1				1				1			
2				2				2				2			
3				3				3				3			
4				4				4				4			

Cache 1   Cache 2   Cache 3   Cache 4

Interconnection Network

Memory   I/O
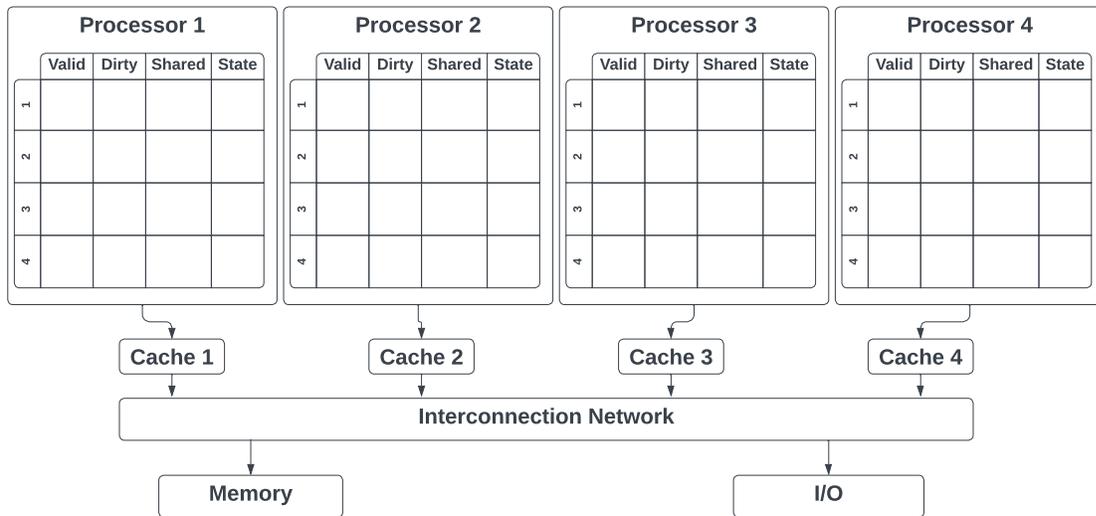
4.3   We will now introduce a fifth state, the **Owned(O)** state, that represents the instance where a cache has exclusive ownership of a cache line and other caches can read from it. Fill in the following table with the corresponding processor states according to the **MOESI** protocol for each step below.

Processor 1				
	Valid	Dirty	Shared	State
1				
2				
3				
4				

Processor 2				
	Valid	Dirty	Shared	State
1				
2				
3				
4				

Processor 3				
	Valid	Dirty	Shared	State
1				
2				
3				
4				

Processor 4				
	Valid	Dirty	Shared	State
1				
2				
3				
4				

Cache 1   Cache 2   Cache 3   Cache 4

Interconnection Network

Memory   **I/O**