The Modified-Shared-Invalid (MSI) protocol maintains the following invariant:

Each block of memory is always in exactly one of the following states:

1. It is not valid in any caches (no caches have it in shared or modified)
2. It is writable in exactly one cache (in the modified state) and potentially inconsistent with main memory.
3. It is read-only in one or more caches (in the shared state) and all copies are consistent with memory.

To maintain this invariant the MSI protocol forces state transitions as dictated by the following state machine, which shows the state of the memory block with respect to a single processor.

![Figure 1: The MSI protocol. All edges are labeled with the activity that causes the transition; any value after the / represents an action place on the bus. All edges not shown are self edges that perform no actions (or are actions that are not possible).](image)

The local processor is capable of performing the following actions:

- **load**: attempting to load data in the cache line
- **store**: attempting to store data into the cache line
- **eviction**: kicking out the block of memory to put a different block of memory into that block of the cache.

These actions can result in the following bus actions, which in turn can cause state transitions in the caches of other processors on the bus.

- **GETS**: request a copy of the memory block on the bus in Shared state
- **GETX**: request a copy of the memory block on the bus in eXclusive state (a.k.a. Modified)
- **UPGRADE**: request write permission for a block that you already have (so that can change your local copy to Modified state)
- **SOURCE**: provide (or “source”) a copy of the block to another processor’s cache, because your copy is more recent than the copy in memory.
- **WRITEBACK**: write the block back to main memory, so that main memory will have a copy of the most recent data for the block.
Example

Given the following state of a multi-core computer, write the series of bus actions and the final states of the caches and memory for the following sequence of actions:

1. processor 1: X = X + 7
2. processor 2: Y = 12
3. processor 1: X = X + Y

<table>
<thead>
<tr>
<th>Processor 1 Cache</th>
<th>Processor 2 Cache</th>
<th>Main memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>X: 10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Y: 40</td>
</tr>
</tbody>
</table>
Problem 1

Given the following state of a multi-core computer, write the series of bus actions and the final states of the caches and memory for the following sequence of actions:

1. processor 1: A = B
2. processor 2: A = 12
3. processor 1: B = 3

![Diagram of processor caches and memory states]

Problem 2

The following code sums the contents of an array of integers of length “length”.

```c
int sum_all(int *myarray, int length) {
    int sum = 0;
    for(int i = 0; i < length; ++i) {
        sum += myarray[i];
    }
    return sum;
}
```

This code has been parallelized to run on a multi-core processor. Below is shown the code that runs on each processor. In addition to the array base pointer and length, each thread receives its thread id and the number of threads and sums its portion of the array into an entry of an array of partial sums.

```c
int partial_sums[NUM_THREADS] = {0};

void sum_all_thread(int *myarray, int length, int my_thread_id, int num_threads) {
    for(int i = my_thread_id ; i < length ; i += num_threads) {
        partial_sum[my_thread_id] += myarray[i];
    }
}
```

What are the problems with this code?
Problem 3

What are the possible final values of A, B, C for the following pair of thread programs executing on a multiprocessor? Initially $A = B = C = 0$.

\[
\begin{align*}
&\text{P0} & & \text{P1} \\
&A = B + 7; & & C = A; \\
&B = 2C; & & \text{} \\
\end{align*}
\]

Problem 4

What is the final state of the caches for the following multiprocessor with an MSI cache coherence protocol? Assume the following instructions are executed in the indicated order.

1. processor 1: li $t1, 11$
2. processor 1: sw $t1, 1000($0)
3. processor 2: lw $t1, 1000($0)
4. processor 2: lw $t2, 2000($0)
5. processor 2: add $t1, $t1, $t2
6. processor 2: sw $t1, 2000($0)
7. processor 1: lw $t3, 2000($0)