EECS 645 Computer Architecture

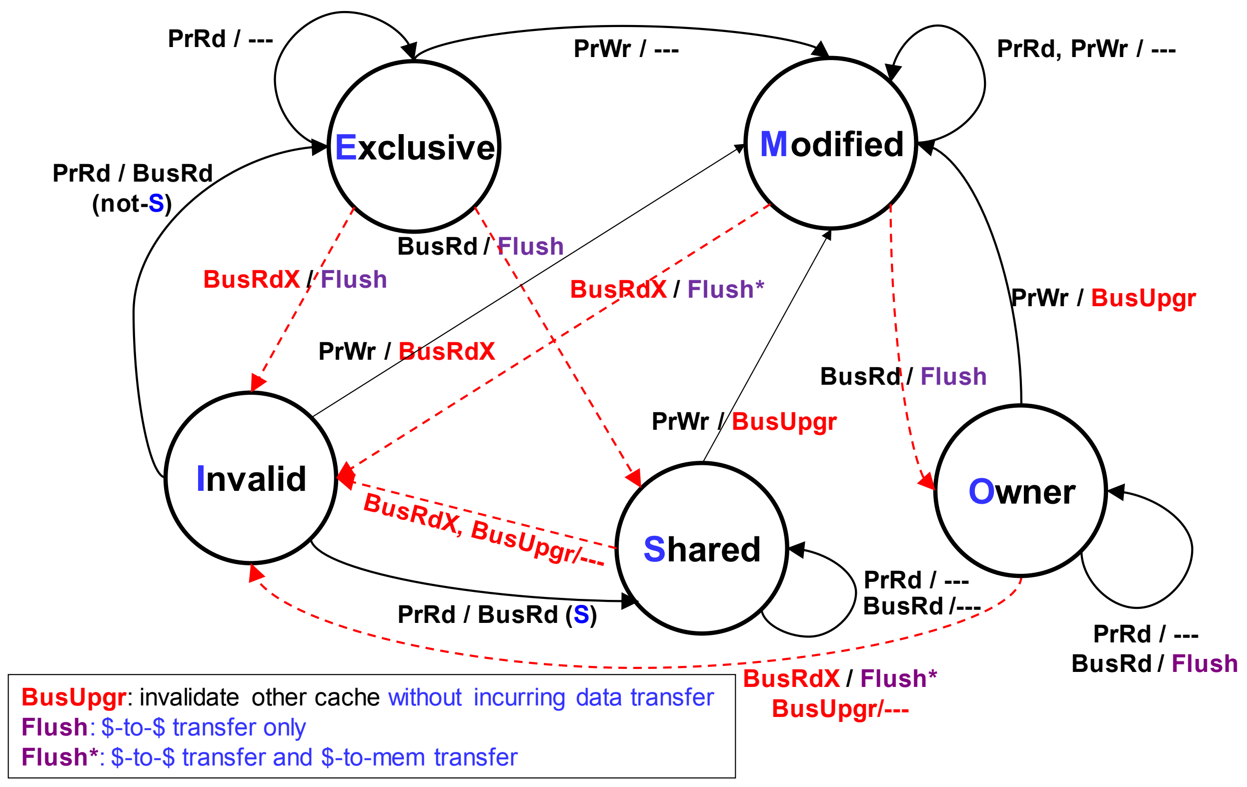
Final Project

Due by 11:30pm on May 12th 2019
Submit to Blackboard

In this project, you will implement a **MOESI invalidation-based coherence protocol for a shared-memory multiprocessor (4 processors)** using any High-level programming languages that you are comfortable with (such as C/C++, Java, python, etc.). A maximum of two students are allowed to collaborate on the project. If you decide to do it by yourself, a 10% bonus point will be added.

Problem Statement

MOESI contains one additional state "**O**wner" on top of the MESI protocol. The **O** state is implemented to facilitate cache-to-cache transfer when the cache line is modified by a processor P (in **M** state) and can directly be supplied by P. Similar to the **S** state, a line found in the **O** state in one processor will be present in at least one other processor's cache. Note that the memory will have a stale version of this line. In the MOESI state diagram below, we also introduce a new transaction called ***BusUpgr*** (Bus upgrade) to substitute *BusRdX* in some transitions where we can simply invalidate others' cache lines without incurring an actual data exclusive read. **Flush** transactions will generate cache-to-cache transfer to the requester and update memory if the cache line is dirty. When all lines are in **S** state and a **BusRd** is put out by a processor, the line will be supplied from the memory.



BusUpgr: invalidate other cache without incurring data transfer
Flush: $-to-$ transfer only
Flush*: $-to-$ transfer and $-to-mem transfer

Dealing with Dirty Writeback

What were not shown in the state diagram are the extra actions needed for dirty writebacks due to conflict misses. A cache line in the M or O state is inconsistent with the data in the main memory. Therefore, when such lines are being evicted (replaced), they need to update the memory to make it consistent upon the eviction.

Memory traces

There are four memory traces provided with this project. Each trace represents one memory access pattern of one processor in a multiprocessor system. Each line in the trace contains one particular memory access issued by the processor. The format of each trace is shown as follows:

<time stamp> <Read or Write> <physical memory address>

In the second field, "0" is a **Read** while "1" is a **Write**. If two memory accesses from different processors occur at the same cycle among traces (*i.e.,* simultaneous requests at the same cycle), then simply assume **the priority is always given to the one with the smallest processor ID**. In other words, the higher processor ID request will be pushed one cycle after its immediate lower processor ID request.

Simulation Configurations

You need to conduct this work by using the following cache configuration. Each processor only contains a "single level cache". All caches employ write-allocate policy.

1. 16KB, 32-byte cache line, Direct-mapped cache for all processors.
2. 32KB, 64-byte cache line, 2-way using least recently used (LRU) replacement policy for all processors. **(10% bonus points if you implement this configuration)**

Outputs from the design

You need to show the following statistics for the entire traces.

1. The total number of cache-to-cache transfers for each processor pair in the following format.

```
P0 cache transfers: <p0-p1> = xxx, <p0-p2> = xxx,  <p0-p3> = xxx
P1 cache transfers: <p1-p0> = xxx,  <p1-p2> = xxx,  <p1-p3> = xxx
P2 cache transfers: <p2-p0> = xxx,  <p2-p1> = xxx,  <p2-p3> = xxx
P3 cache transfers: <p3-p0> = xxx,  <p3-p1> = xxx,  <p3-p2> = xxx
```

2. The total number of invalidations due to coherence (i.e. not including line replacement) in each processor in the following format.

```
P0 Invalidation from: m = xxx,  o = xxx,   e = xxx,   s = xxx,   i = xxx
P1 Invalidation from: m = xxx,  o = xxx,   e = xxx,   s = xxx,   i = xxx
P2 Invalidation from: m = xxx,  o = xxx,   e = xxx,   s = xxx,   i = xxx
P3 Invalidation from: m = xxx,  o = xxx,   e = xxx,   s = xxx,   i = xxx
```

3. The number of dirty writebacks from each processor. Note that, in this report, please write back all the dirty lines at the end of your simulations.

```
P0 = xxx,  P1 = xxx, P2 = xxx, P3 = xxx
```

4. The number of lines in each state at the end the simulation for each processor.

   P0: m = xxx, o = xxx, e = xxx, s = xxx, i = xxx
   P1: m = xxx, o = xxx, e = xxx, s = xxx, i = xxx
   P2: m = xxx, o = xxx, e = xxx, s = xxx, i = xxx
   P3: m = xxx, o = xxx, e = xxx, s = xxx, i = xxx

Design Strategy

1. You need to implement a cache simulator. When you perform your MOESI, you have to conduct cycle-based simulation, in other words, maintain a global clock during your execution. If an event (from any processor) takes place in a particular cycle, then the state of the corresponding cache line needs to be updated according to MOESI. For example, you may create a tag array and a state array based on the cache size and line size to keep track of the tags and states of each cache line in each processor.
2. Deriving the state diagram once by yourself will substantially help you understand the protocol which is not really that complex as it seems.
3. A 2-way least recently used (LRU) simply tracks the latest used way to be the most recently used (MRU), the other line must be the LRU one. (Bonus)

How to receive the credit

Please upload one single zip file containing the following items:

- Source code with **detailed comments**
- Instructions for the grader to compile and run your program (e.g., a Makefile and READ.txt)
- A report of your implementation approach and results

Honor Code

You are allowed to discuss your design approaches with your peers. However, you are not allowed to copy others' codes and data. For those who violate the rule, both the originator and the copier will receive zero credit and will be reported to the Department and School of Engineering.