Portland State University

Electrical and Computer Engineering

**ECE 585 Final Project Report**

**Split L1 cache design and simulation**

Josiah Sweeney

Nguyen Pham

Neima Kashani

Melinda Van

Table of contents

**Introduction**

Due to the great disparity between microprocessors and memory performance, caches are used to as a buffer between RAM and CPU to alleviate this gap, therefore, they are normally places nearby processor cores. A cache is a block of memory that stores copies of most frequently used information from main memory location. Whenever an application makes a request, the processor can fetch or store data in the caches instead of going all the way to main memory to retrieve the data needed. Most processors have a hierarchy of multiple cache levels: L1, L2, and L3. L1 caches are normally implemented as a split cache, with separate instruction and data caches. Data is transferred between memory and caches in a block of fixed size, called a cache line. A cache line is made up of a copy of memory data and its memory address, or a tag. The tag is used to compare with the requested memory address by the processor. If the processor finds the requested memory location in the cache, a cache hit occurs. Otherwise, it is a cache miss.

There are several placement policies to map data from memory to the cache. The simplest approach is to assign each memory block to one specific cache line. This is called direct mapping method. The best and most flexible policy is fully associative mapping in which any block of memory can be placed in any cache line. Set associative mapping is an enhanced form of direct mapping. A memory block can be mapped to any line of a specific set. When the cache is full, it follows replacement policies to evict existing data. One of the most popular policies is least recently used (LRU) to replace the least recently accessed data. Another aspect to consider when designing the cache is cache coherence to synchronize shared data stored in multiples cache levels and main memory. Snooping is one of the cache coherency schemes used to keep track of the sharing status of the copies of data across the memory system. The state of the copy of data in the cache can be noted as modified, exclusive, shared, and invalid. This snoopy scheme is knowns as MESI protocol.

To further reinforce our understanding about caches, we were assigned to design and simulate a split L1 cache using C programming language for a new 32-bit processor (acting processor) and can be used with up to three processors (shared processors) in the shared memory system. Both caches implement MESI protocol to maintain coherency and inclusivity by communicating with a shared L2 cache whose implementation and simulation is not in the scope of the project. However, we are required to model the interaction between L1 and L2 caches by displaying the appropriate messages for the cache operations. The L1 caches also employ LRU with counter as replacement policy.

**Design Specifications**

The L1 instruction cache consists of 16K sets of 4-way set associative 64-byte lines. The L1 data cache consists of 16K sets of 8-way set associative 64-byte lines. The data cache implements write allocate policy, except for the first write from memory to the cache, which is write-through. Both caches employ LRU using a counter and MESI protocols and are backed up by a shared L2 cache to maintain cache coherency and inclusivity. This can be accomplished during simulation by displaying the appropriate messages to show the communication between L1 and L2 caches.

* Return data to L2 <address> : in response to a 4 in the trace file your cache should signal that it’s returning the data for that line (if present and modified)
* Write to L2 <address>: this operation is used to write back a modified line to L2 upon eviction from the L1 cache. It is also used for an initial write through when a cache line is written for the first time so that the L2 knows it’s been modified and has the correct data
* Read from L2 <address>: this operation is used to obtain the data from L2 on an L1 cache miss
* Read for Ownership from L2 <address>: this operation is used to obtain the data from L2 on an L1 cache write miss

In addition to those responses, the program is required to keep track and display all the key statistics of both L1 caches:

* + Number of cache reads
  + Number of cache writes
  + Number of cache hits
  + Number of cache misses
  + Cache hit ratio

The simulation also supports three modes without the need of recompilation:

* Mode 0: Summary of usage statistics and print commands only
* Mode 1: Information from mode 0 and display additional messages from L2
* Mode 2: Information from previous modes and a message for every cache hit

The simulation reads cache access caches/events from a trace text file, and it can support any trace file provided without recompiling the program so long as the provided file has the correct format

n address

where n is

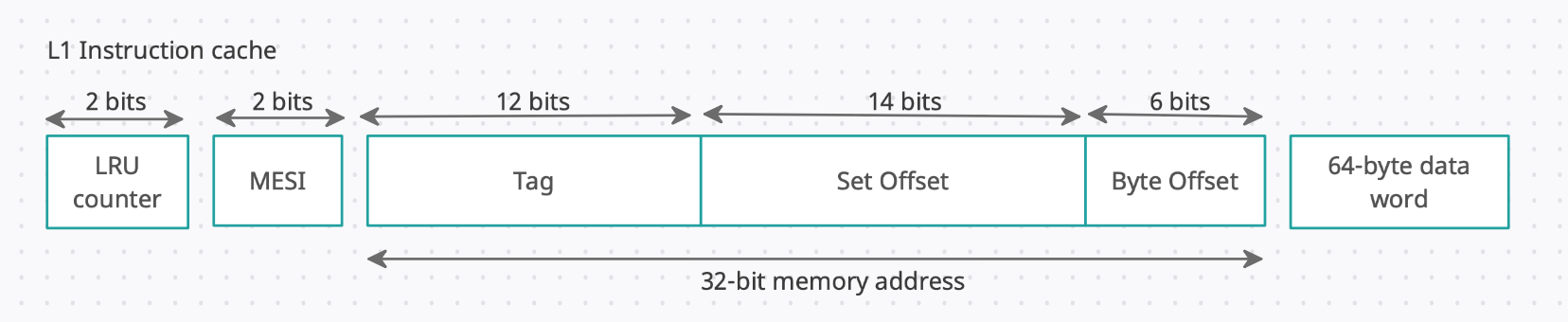
* 0 read data request to L1 data cache
* 1 write data request to L1 data cache
* 2 instruction fetch (a read request to L1 instruction cache)
* 3 invalidate command from L2
* 4 data request from L2 (in response to snoop)

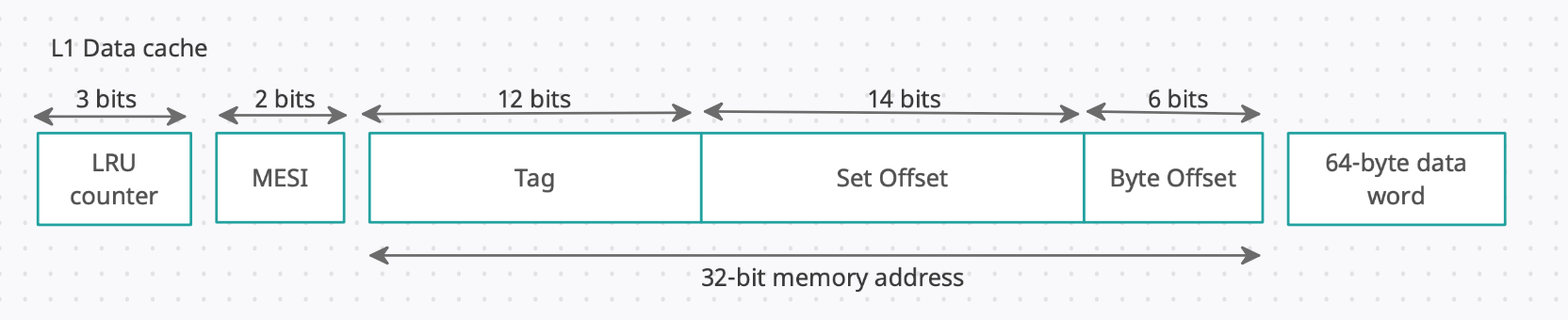
8 clear the cache and reset all state (and statistics)

9 print contents and state of the cache (allow subsequent trace activity)

**Assumption**

All cache lines in the caches are initialized to invalid state every time the program is simulated. For the first time the acting processor sends a read request to the caches, that cache line is updated to exclusive state since there are no other processors share the same copy. All requesting addresses are assumed to be in the main memory system. If there is a read hit on the cache line with an E state, it is assumed that this read is from another processor, so the cache line status is updated to S. However, if the read hit is on the cache line with a M state, it is assumed that the requesting device is the acting processor, and the cache line status remains unchanged. The caches themselves do not handle multiple requests when a transaction spans to multiple cache lines and involves other processors. Those dependent operations are assumed to be held in the shared processors, and the caches are only updated once all those transactions complete. The shared L2 cache is merely a stub in this simulation and is not a part of the project development.

**Cache Structure**



**Cache Operations**

The cache operations are handled by the commands extracted from the trace file. Those function

For command 0, it is a read request to L1 data cache. If the memory address is matched with the tag of one of the cache lines, it is a hit. The copy of data is delivered from the data cache to requesting device. If it is a miss, the requesting processor needs to fetch the data from L2 and write-back data to the data cache.

Command 1 is a write request to L1 data cache. If the requesting memory location exists in the data cache, data is write back to the matching tag line. If the status of that cache line is in modified, the processor will write the content of that modified line back to L2 cache before replacing it. If the requesting address does not exist in the cache, the processor will write through to L1 and L2 caches if it is an initial write. Otherwise, the processor will evict the LRU cache line and replace the new block of memory.

Command 2 is an instruction fetch in the instruction cache. The read operation is similar to the read operation described in the data catch. However, there is no write operation allowed in this cache to avoid overwriting instructions in the system. All instructions are fetched from the memory.

Command 3 is a request sent by L2 processor to invalidate a cache line in L1 data cache. All this does is to set the requested cache line to invalid state in order to maintain coherency in the system

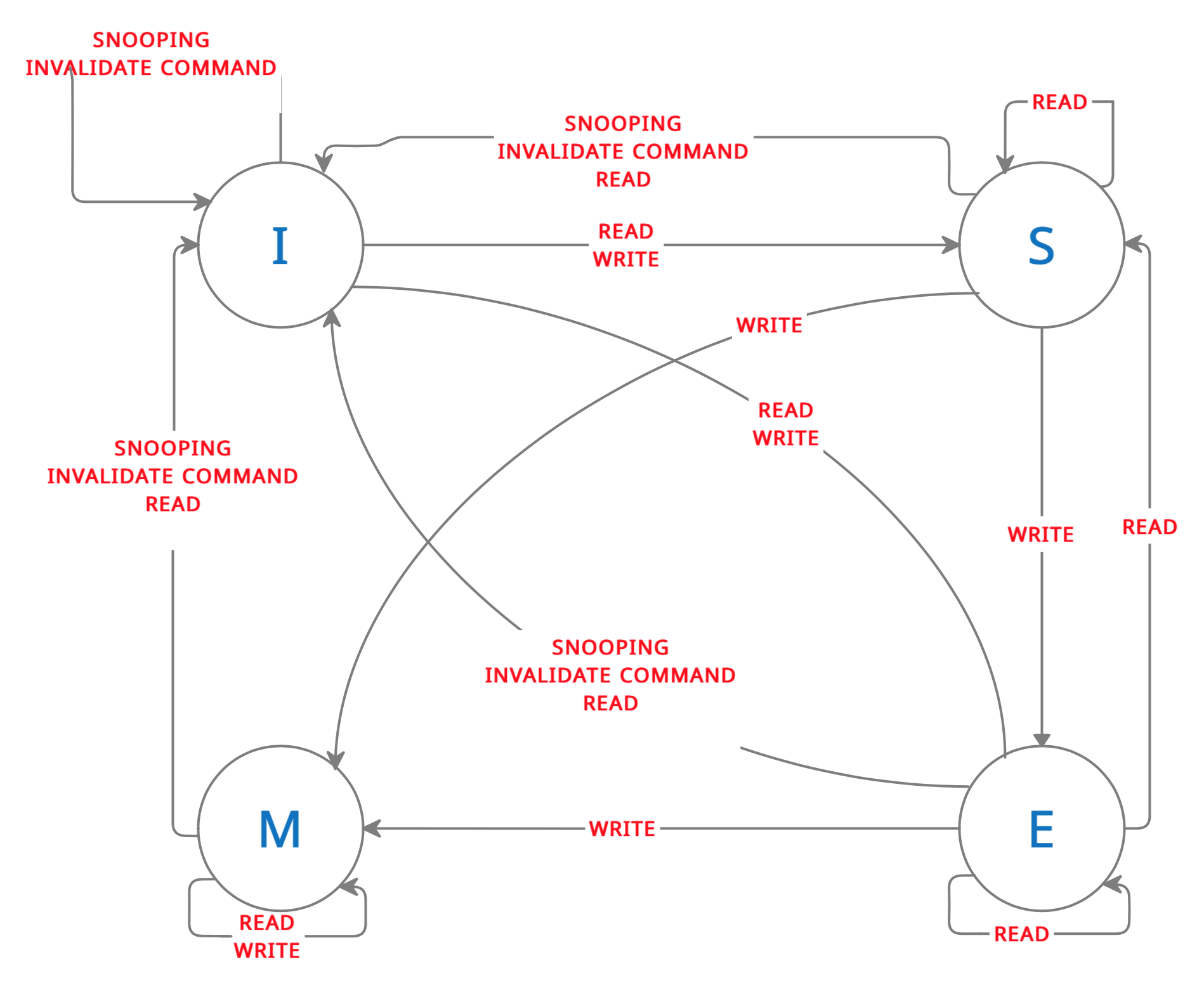
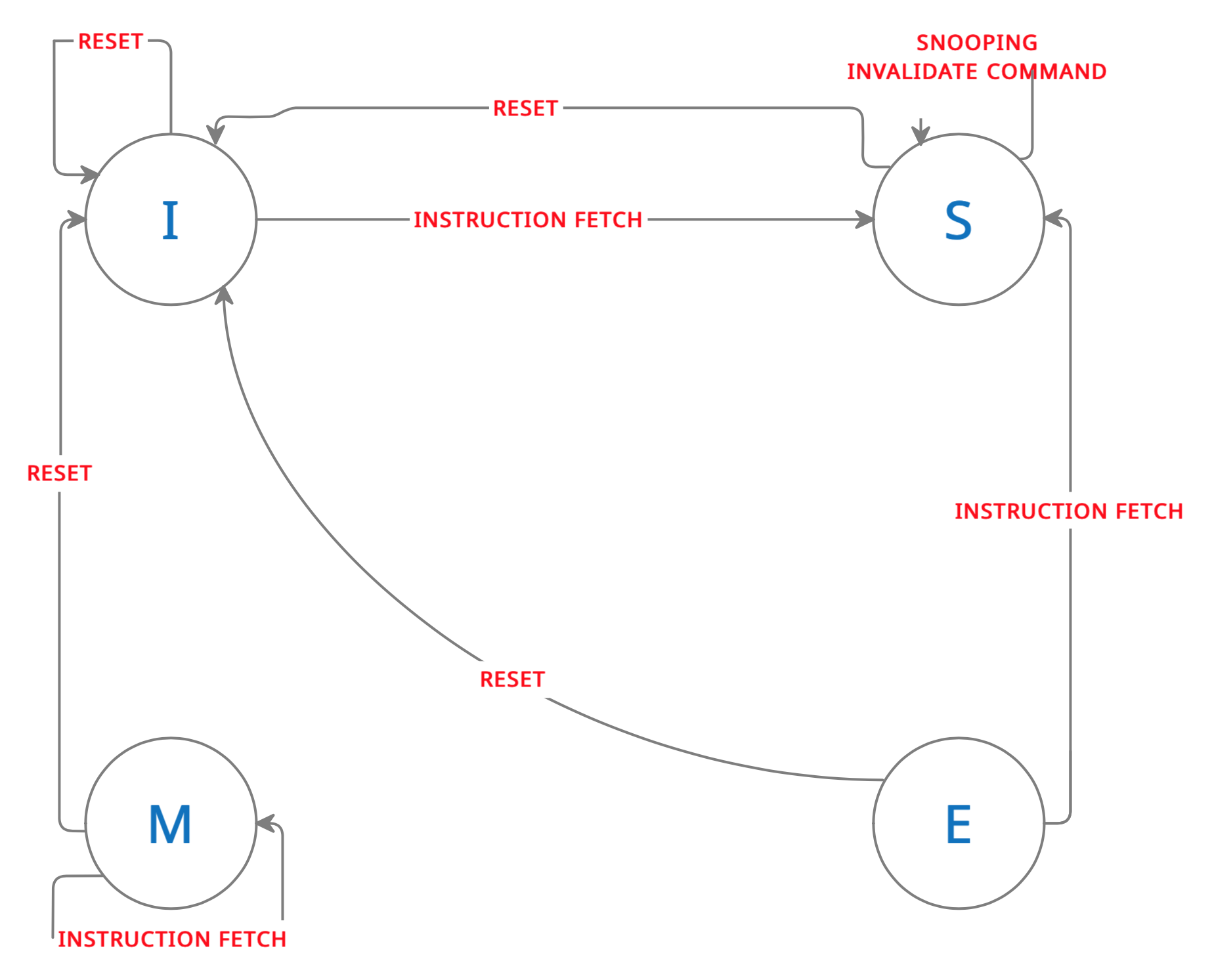
Command 4 is a read for ownership or a response from snooping. The L2 processor makes a read request of a copy of data for ownership. The copy is first written back to main memory and then delivered to the requested processor. The processor then updates the cache line in L2 cache to exclusive state and broadcasts the invalidate signal to other processors. Therefore, the cache line in L1 data cache is updated to the invalidate state. All of these transactions between L2 cache and memory system are assumed to be handle by other processors and not the acting processor. Our processor though needs to update the cache line in data cache to invalid state.

**LRU Replacement Policy**

For this project, the LRU protocol is implemented by adding an additional counter in front of each cache line in a set to indicate how recently an item is referenced. Whenever a copy of data in cache line is referenced, LRU counter is updated to the largest number. The LRU counters in other cache line in the set is decremented by 1 if they are greater than the previous LRU counter of the most recently used (MRU) cache line. When an eviction occurs, the data of LRU cache line is removed. The LRU counter for the data caches is 3-bit long, and the LRU counter for the instruction caches is 2-bit long. The MRU values for the data and instruction caches are 111 and 11 respectively.

**MESI Protocol**

The following diagrams are the state transitions described how we implemented MESI protocol in the cache design.



**Testing and Verifications**

The program was compiled, run, and tested in Dev C++ environment after the initial coding completed. Our test plan was designed to first checked for the cache functionalities separately by creating a dedicated trace file for each operation. Once we were able to ensure the accuracy of cache operations, additional function was added to the cache handle the communication between L1 and L2 caches. Beside from generating messages between L1 and L2 caches when L2 caches was involved, we checked for the status of MESI bits of requested cache line, which should be in invalidate states in all cases. The program itself does not handle any bus snooping protocol or L2 cache implementation since it is out of the scope of this project.

**Results**