# ECE/CSC 406/506: Architecture of Parallel Computers
# Project 2. Coherence Protocols
# Version 1.0

**Due: Wednesday, November, 16th, 11:59 pm**

The purpose of this project is to give you an idea of how parallel architecture designs are evaluated, and how to interpret performance data.

In this project, you will create a simulator which will allow you to compare different coherence protocol optimizations. You will simulate a 4-processor system with bus-based coherence. Your job in this project is to instantiate these peer caches, and to *maintain coherence* across them by applying coherence protocols.

Your simulator should output various statistics, such as number of cache hits, misses, memory transactions, number of issued coherence signals, etc.

Simulator is functional, i.e. number of cycles, data transfer is not under consideration. It makes the project really easy to implement.

## Introduction

In this section provided infrastructure is described. Particular tasks are described in Parts 1-4.

### Organization

- src/ - starter source code
    - cache.cc
    - cache.h
    - main.cc
    - Makefile
- trace/ - memory access traces
- val/ - validation outputs for traces

The directory val contains validation outputs you are asked to match. You have to literally match the output files. If your output does not match the given validations in terms of results and formats, you will be deducted points. You can use the `make val` target to perform local check.

### Starter code

Class Cache simulates behavior of a single cache. It provides implementation of all basic cache methods and cache line states. Actions which should be performed during cache access are

described in the Access() method. Please study that class thoroughly. Note counter increment and LRU updates.

You can derive your coherent caches from that class, by providing a new Access() method (possible with new Snoop() method) and adding new cache line states. You are free to make **any changes** to the code, as long as it compiles into a single binary and has required command line arguments.

## Trace format

Trace reading routine is already provided in main.cc. If you want to create your own traces for debugging (highly recommended), please use the following format.

Each line in the trace file is one memory transaction by one of the processors. Each transaction consists of three elements: processor(0-3) operation(r,w) address(in hex).

For example, if you read the line `5 w 0xabcd` from the trace file, that means processor 5 is writing to the address "0xabcd" in its local cache. You need to propagate this request down to cache 5, and cache 5 should take care of that request (maintaining coherence at the same time).

## Self-grader and submission

Gradescope self-grader will be provided. To prepare your source code for submission, run `make pack`. You are allowed to add new source files. Make your first submission as early as possible to understand the requirements. Number of attempts is not limited. After you are done with your code and the report, attach the PDF file to your submission.

## Output of the simulator

1. Number of read transactions the cache has received
2. Number of read misses the cache has suffered
3. Number of write transactions the cache has received
4. Number of write misses the cache has suffered
5. Total miss rate (rounded to 2 decimals, should use percentage format)
6. Number of dirty blocks written back to the main memory (due to cache eviction and Flush)
7. Number of cache-to-cache transfers (from the requestor cache perspective, i.e. how many lines this cache has received from other peer caches)
8. Number of memory transactions
9. Number of interventions (refers to the total number of times that E/M transits to Shared states)
10. Number of invalidations (any state->Invalid)
11. Number of flushes to the main memory
12. Number of issued BusRdX transactions
13. Number of issued BusUpgr transactions

### Cache parameters
Size: 8192B, associativity: 8, block size: 64B

### Command line arguments
```
./smp_cache <cache_size> <assoc> <block_size> <num_processors> <protocol> <trace_file>
```

### Compiling and running simulator
Run all commands in src/ directory. Check targets and parameters in Makefile

To compile, run `make`. **Common problem:** not running `make clean` after modifying header files.

To make sure that your implementation doesn't fail, run `make run.`

To validate your implementation, run `make validate.` If the output looks strange, go and check that your simulator actually outputs something, with `make run.`

### Implementation suggestions
1. Read the given code carefully cache.cc, cache.h, and understand how a single cache works. Most of the code given to you is well encapsulated, so you do not have to modify most of the existing methods. You may need to add more functions as deemed necessary.
2. In cache.cc, there is a function called Cache::Access() , this function represents the entry point to the cache module, you might need to call this function from the main, and to pass any required parameters to it.
3. Recall that each processor has a separate cache and for each request in one cache (requestor), there should be some handling in other caches (snoopers). Think of method Snoop() which will handle signals and update the state of the cache line accordingly.
4. In cache.h, you might need to define new methods, counters, states, or any protocol-specific states and variables.
5. You might create an array of caches, based on the number of processors used in the system.
6. Feel free to create a class hierarchy and divide your code into multiple files.
7. Start early and do self-grading after each part.

## Part 1. MSI
(book p.215)

(50 pt) Implement basic MSI coherence protocol.

Implementation hint: add instance of cache per processor, add coherence state support to cache line, make sure that Access() in requestor emits coherence signals and that other snooping caches process those signals.

Statistics collected after executing memory trace should match provided validation outputs.

## Part 2. MSI vs MSI + BusUpgr

(book p. 219)

a) (10 pt) Implement MSI + BusUpgr protocol.

Implementation hint: this is a very simple extension to the previous Part. At this point your simulator should have coded state diagram transitions.

b) (5 pt) Compare basic MSI and MSI with BusUpgr optimization. Plot number of memory transactions for the long trace. Explain results briefly (in a couple of sentences).

## Part 3. MESI

(book p. 221)

a) (10 pt) Implement MESI protocol with BusUpgr and cache-to-cache transfers. These transfers happen on the cache miss. Other caches can provide blocks for both read and write misses. On the snooper's state diagram these situations are shown with FlushOpt, however we only record the number from the requestor 's perspective.

Implementation hint: in the simulator we are simulating caches functionally. We don't need to actually transfer data between caches. All we need is to detect situations where we can have cache-to-cache transfers and increment respective counter.

b) (5 pt) Compare MSI with BusUpgr optimization and MESI protocols. Plot number of memory transactions stacked with cache-to-cache transfers for the long trace. Explain results briefly (in a couple of sentences).

## Part 4. Snoop Filter

(book p. 239)

a) (5 pt) Collect and plot number of useful vs wasted cache lookups while snooping in MESI protocol for long trace. We call cache lookup *wasted* if a snoop request for a cache block was received, lookup in the tag array was performed, but that cache block was not valid.

b) (10 pt) Implement history filter. History filter should keep addresses of cache blocks which are definitely not in the cache. This is true for blocks which were recently invalidated due to coherence requests or for blocks which were looked up in the previous cycles, were not found and no read or write requests were performed to them by the cache since then. *For the sake of simplicity of implementation, the history filter shouldn't keep track of cache blocks evicted due to replacement policy.*

You should implement a history filter as a direct mapped tag array with 16 entries and block size the same as in the cache.

Implementation hint: class *Cache*, provided in the starter, provides all necessary methods to use it as a history filter. It has the tag array, replacement support, etc. If you kept the original class, you can just use an instance of it with parameters which you can calculate.

On Gradescope, only this configuration of the history filter will be checked.

**Note:** snoop filtering shouldn't change your coherence statistics. If you have different numbers, it means that you are filtering useful cache lookups.

c) (5 pt) Compare:

1) 16-entry direct mapped history filter with
2) 16-way fully associative and
3) 4-set 4-way associative filters.

Plot number of useful, filtered and wasted cache lookups. Explain results briefly (in a couple of sentences).