UNIVERSITY OF AMSTERDAM
Institute of Informatics
Computer Systems Architecture

Simon Polstra
Dr. Andy Pimentel
January 2, 2020

Parallel System Architectures 2019/2020

— Lab Assignment: Cache Coherency —

Introduction

In this assignment you will use **SystemC** to build a simulator of a **Level-1 Data-cache** and various implementations of a **cache coherency protocol** and evaluate their performance. The simulator will be driven using trace files which will be provided. All documents and files for this assignment can be found on Canvas. For information about cache coherencies and memory, see Appendix A and B.
This framework contains all documentation, the helper library with supporting functions for managing trace files and statistics, the trace files, and a Makefile to automatically compile your assignments. Using the Makefile, the contents of each directory under `src/` is compiled automatically as a separate target. It already includes directories and the code of the Tutorial, as well as a piece of example code for **Assignment 1**.

Trace files are loaded through the functions provided by the helper library (`psa.h` and `psa.cpp`), which is in the provided framework or can be downloaded separately from the resources mentioned above. Detailed documentation for these functions and classes is provided in Appendix D, but the example code for **Assignment 1** also contains all these functions and should make it self-explanatory.

Note that, although later on simulations with up to eight processors with caches are required to be built, it is easier if you start with a single CPU organization. Then, extend it to support multiple CPUs, but make sure that the number of CPUs and caches in your simulation is not statically defined in your code, but uses the number of CPUs read from the trace file.

You should print important events, such as cache actions and cache state transitions to the console. This will help to evaluate the correctness of each action in your cache system. Part of the evaluation of your work is also based on this log information. If you want to get fancy look into using the logging functions provided by SystemC such as `SC_REPORT_INFO()`.

Submission & Assessment

For assessment you must:

1. Demonstrate your solutions and code to one of the lab assistants during one of the lab sessions before each deadline.
2. Submit your source code for each sub-assignment (packed in `tar/gz` format) incrementally on Canvas before each deadline.
3. Write a report with your results, explanation of the results and a comparison between different cache coherency protocols.

Submission Requirements:

1. Your source code should compile cleanly and run on a recent Linux distribution with the command `make` without any modifications to the submitted code, either using the provided Makefile or your own, which has to be included.

2. The first command line argument your program accepts is the name of the trace file. This happens automatically when the `init_tracefile` function is used.

3. Cache actions and cache state transitions should be printed to the console.

4. Hit/Miss rate statistics should be gathered with the provided functions, and should be printed with the `stats_print` function after the simulation ends.

5. Your simulations should always terminate and exit without any errors.

6. For each deadline, write a report (separate or incremental) in which you describe the problem, your solution to the problem, discuss your findings and results, and analyse why you observe these results.

7. **Assignment 2 and 3**: Supporting a different number of CPUs in your simulation should **not** require recompilation.

8. A report should be submitted in `pdf` format only.

Additional Information: Even though attending the lab sessions is not compulsory, we would suggest you to come in regularly. This is because the lab assistants will have the most time for answering your questions during the sessions.
Questions: You can of course ask questions during the lab sessions. You can also ask questions in the discussions section on Canvas. This way other students can benefit from the discussion as well. You are encouraged to participate in these discussions. If you have attended the lab sessions and have additional questions that you don't want share with other students you can ask them via email (`s.polstra@uva.nl`).

Trace files

In order to drive the simulation, there are 3 kinds of trace files available to you: *random*, *debug* and *FFT*. All versions exist for 1, 2, 4 and 8 processors. The *random* trace files (`rnd_pX.trf`) have the processors read or write memory randomly in a window in memory that is moving. The *debug* trace files (`dbg_pX.trf`) are a short version of the random trace files, suited for debugging your simulation. The *FFT* trace files (`fft_16_pX.trf`) are the result of the execution of a FFT on a 64K array and can be used, along with the random trace files, to generate results for your reports. All reads and writes are byte-addressed, word-aligned accesses.

Assignment 0: Getting started with SystemC

Read and follow the **SystemC** tutorial. Complete this before the second lab session. There is no submission required for this assignment.

Assignment 1: Implement a single cache

In this assignment you must build a single 32kB 8-way set-associative L1 D-cache with a 32-Byte line size[1]. The simulator must be able to be driven with the single processor trace files. The addresses in the trace files are 32bits. Assume a Memory latency of 100 cycles, and single cycle cache latency. Use a least-recently-used, write-back replacement strategy. See the URL in the footnote and Figure 1 for more information.

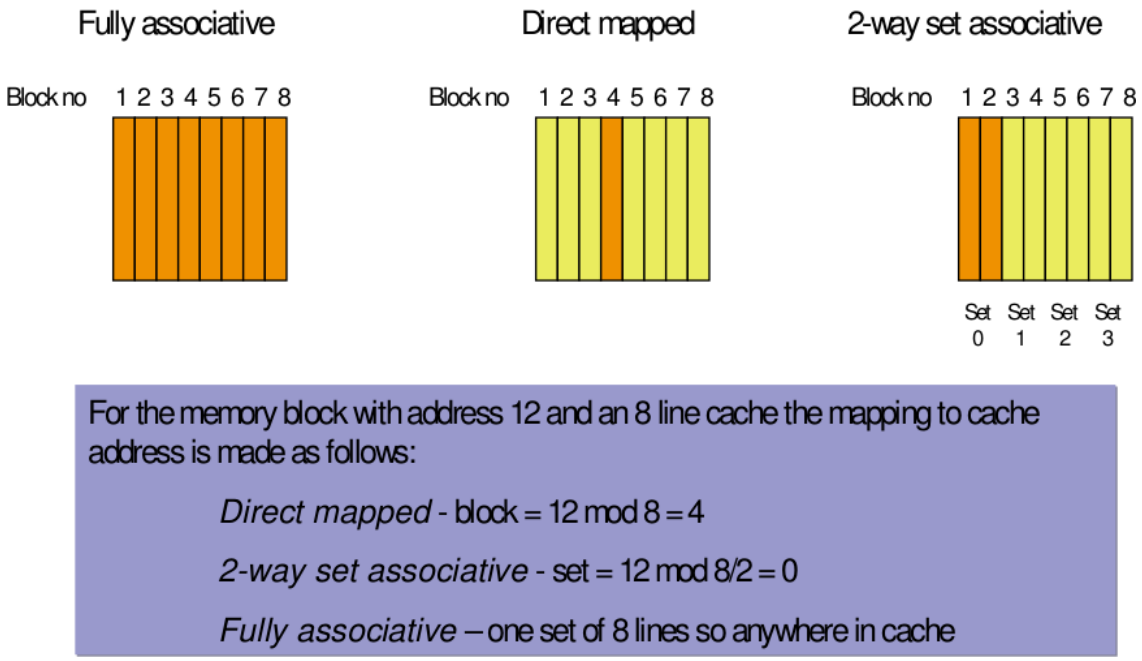

Figure 1: Cache Mapping.

Hints:

1. You can use the example code provided for **Assignment 1**, which is based on the tutorial, as a guide and modify the Memory module so that instead of RAM, it simulates a set-associative data cache.
2. As we only simulate accesses to memory, it is not necessary to simulate any actual contents inside the cache. Furthermore, you do not need to simulate the actual communications with the memory, just the delay.

Submission due date: to be announced on Canvas

[1] `http://en.wikipedia.org/wiki/CPU_cache#Associativity`

Assignment 2: Implement a multiprocessor coherent cache system

Extending the code from **Assignment 1**, you are to simulate a multiprocessing system with a shared memory architecture. Your simulation should be able to support multiple processors, all working in parallel. Each of the processors is associated with a local cache, and all cache modules are connected to a single bus as shown in Figure 7 in Appendix B. A main memory is also connected to the bus, where all the correct data is supposed to be stored. Again, it's your choice if you want to implement a separate memory component. Assume a pipelined memory which can handle one request per cycle and a latency of 100 cycles. In order to make cache data coherent within the system, implement a bus snooping protocol. This means that each cache controller connected to the bus receives every memory request on the bus made by other caches, and makes the proper modification on its local cache line state. In this assignment, you should implement the simplest VALID-INVALID protocol. The cache line state transition diagram for this protocol is shown in Figure 2. Using your simulator you must perform experiments with different numbers of processors (1 to 8 processors) using the different trace files for each case.

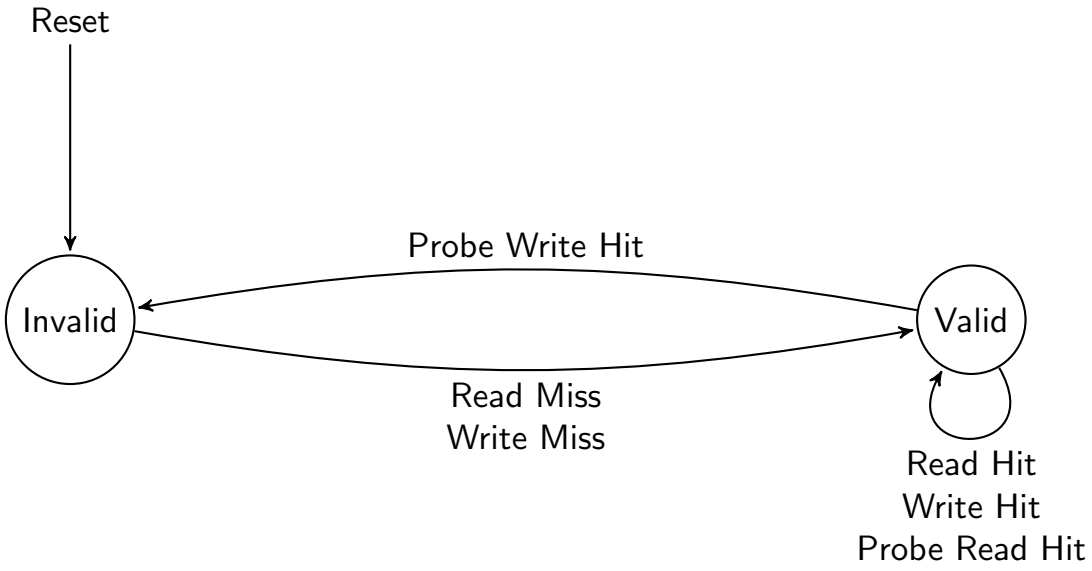

Figure 2: VALID INVALID Cache Line State Transition Diagram.

For each experiment you should record (but not limited to) the following data for your report:

1. Cache hit and miss rates (use the framework functions)
2. Main memory access rates
3. Average time for bus acquisition (as a measure for bus contention)
4. Total simulated execution time

You should also investigate what happens to cache hit and miss rates when snooping is deactivated.

Hints:

1. Some help on implementing a Bus is provided in Appendix C. You could use or modify these implementations for your simulation.
2. The bus can only serve one request at any certain time. Thus if one cache occupies the bus, the other cache has to wait for the current operation to complete before it can utilize the bus. The

cache does not occupy the bus when it waits for the memory to respond. This means that you will need to implement a split-transaction bus. The responses from memory should have priority on the bus.

3. The cache must not only be able to serve the request from the processor, it also has to observe all the transactions happening on the shared bus, and make proper adjustment on its own corresponding cache line states as described in Figure 2.

Submission due date: to be announced on Canvas

Assignment 3: Implement the MOESI protocol

In this assignment, you are to simulate a version of the *MOESI* protocol. Create a copy of your **Assignment 2** directory and modify this to simulate the *MOESI* protocol (i.e: alse keep your original **Assignment 2** data!). This protocol is implemented in AMD64 caches. The cache line states of the *MOESI* protocol are:

- **Invalid**: A cache line in the invalid state does not hold a valid copy of the data. Valid copies of the data can be either in main memory or another processor cache.
- **Exclusive**: A cache line in the exclusive state holds the most recent, correct copy of the data. The copy in main memory is also the most recent, correct copy of the data. No other processor holds a copy of the data.
- **Shared**: A cache line in the shared state holds the most recent, correct copy of the data. Other processors in the system may hold copies of the data in the shared state, as well. If no other processor holds it in the owned state, then the copy in main memory is also the most recent.
- **Modified**: A cache line in the modified state holds the most recent, correct copy of the data. The copy in main memory is stale (incorrect), and no other processor holds a copy.
- **Owned**: A cache line in the owned state holds the most recent, correct copy of the data. The owned state is similar to the shared state in that other processors can hold a copy of the most recent, correct data. Unlike the shared state, however, the copy in main memory can be stale (incorrect). Only one processor can hold the data in the owned state, all other processors must hold the data in the shared state.

The state transition diagram is depicted in Figure 3. Cache lines are initialized to the *Invalid* state, where they hold no data. A cache **read-miss** or **write-miss** causes the cache to retrieve the most recent copy of the data from either another cache or main memory. On a **read-miss**, if another cache has the data, the cache gets a copy (a **probe-read**) and the line becomes *Shared*. Otherwise, the cache gets the data from memory and the line becomes *Exclusive*. On a **write-miss**, the cache gets the data from another cache (a **probe-write**) or from main memory and the line becomes *Modified*. On a **read-hit**, the cache line remains in its current state. On a **write-hit**, the cache line becomes *Modified* and a **probe-write** is sent out to the other caches, causing their copies to become *Invalid*.

For more information about the *MOESI* protocol, see section 7.3 in the AMD64 Architecture Programmers Manual (`https://www.amd.com/system/files/TechDocs/24593.pdf`)

You perform the same experiments with different numbers of processors as in **Assignment 2** and record the same data for your report.

Hints:

1. You will need to take care of the situation where a CPU reads from or writes to a location for which another cache holds data which was not written back to memory yet. You can solve this by implementing cache to cache transfers or you can invalidate the read or write bus request and issue a writeback to update the memory with correct data if needed. The other processor will resubmit it's bus request and it will then receive the correct data.
2. The bus implementation might have to be modified to provide additional information. For instance, when a cache is suffering a read miss, and fetching the data from main memory. It has to check whether the data is incorrect in the main memory (when a cache has a piece of data stored at Modified or Owned state, the value is incorrect in the main memory).
3. In case the cache is running out of space, any data in the Modified and Owned state has to be saved to main memory first, before it can be replaced.

Submission due date: to be announced on Canvas

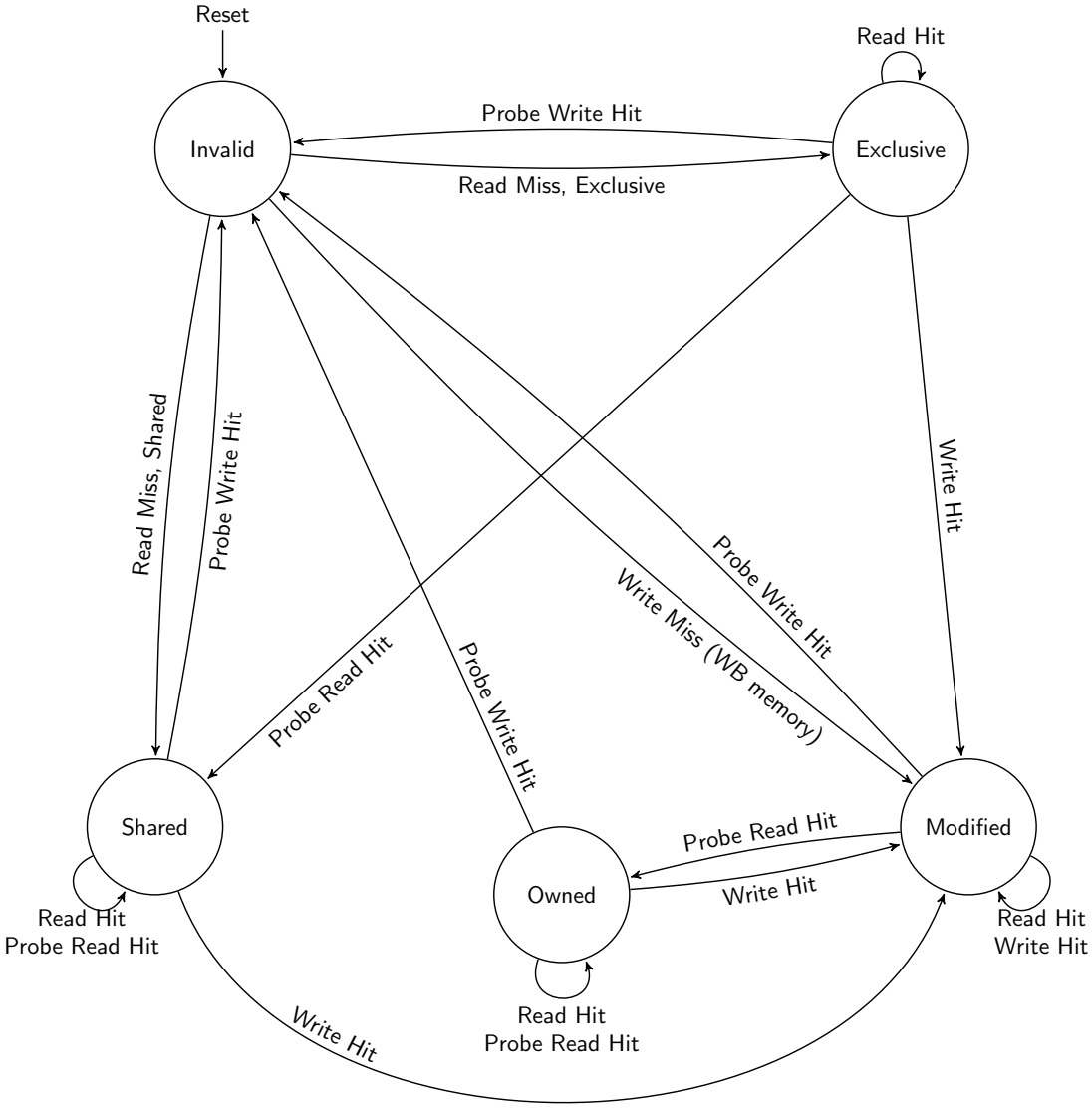

Figure 3: MOESI Cache Line State Transition Diagram.

A Memory Architectures in Multiprocessing Systems

The most straightforward way to accelerate the execution of a sequential program is to extract multiple tasks from the program and run them simultaneously on multiple processors, which is called parallel computing. From the memory perspective, three major groups of memory architectures associated with multiprocessing systems can be identified: *Shared Memory Architecture*, *Distributed Memory Architecture*, and *Distributed Shared Memory Architecture*.

Generally a **Shared Memory Architecture** (Figure 4) refers to a large bulk of memory that can be accessed by multiple different processors in a multiprocessing computer system. In this category all the processors share a unique memory addressing space. All the memory accesses are directed to the large bulk of shared memory.

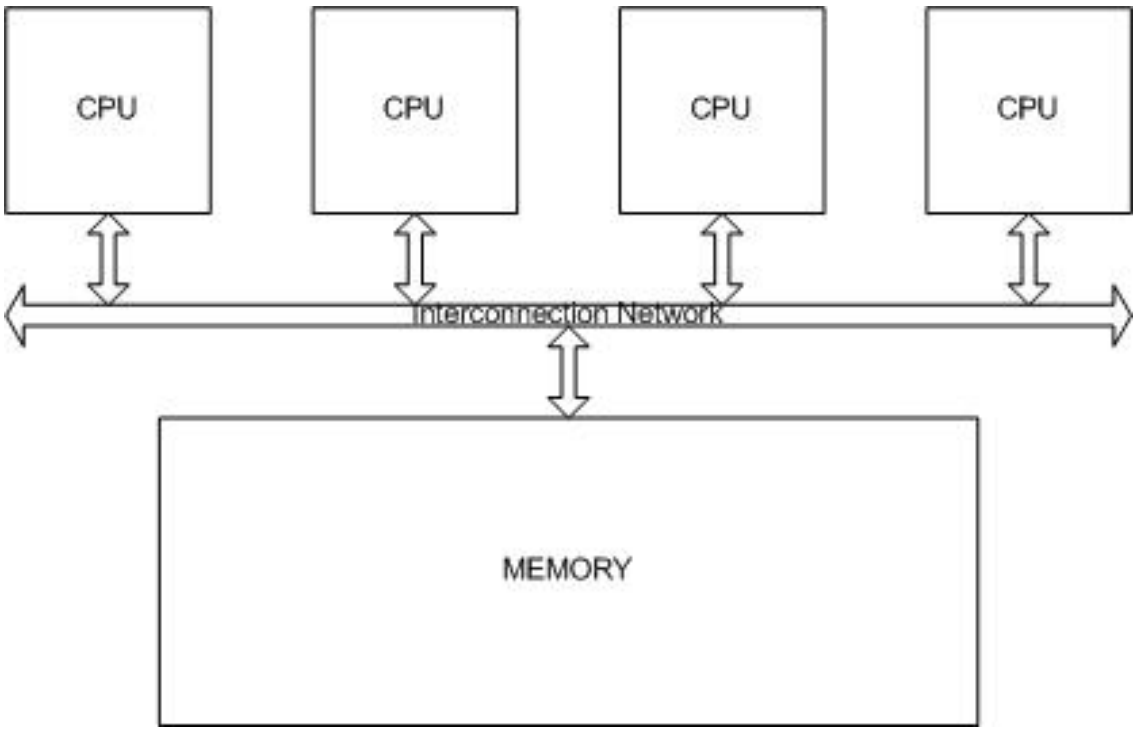

Figure 4: Shared Memory Architecture

In a **Distributed Memory Architecture** (Figure 5), each processor has its own memory. The communication between different processors is carried out by message passing across the interconnection network.

The third group of memory organization is a **Distributed Shared Memory Architecture** (Figure 6). In this group of multiprocessing system, all the processors still hold a common addressing space. However, the physical memory is distributed across the network, and each processor is associated with a bulk of local memory, which is associated with a part of the memory addressing space. Each processor could access all the memory modules across the network, meanwhile, the access to the local memory is much faster than the access to the remote memory module.

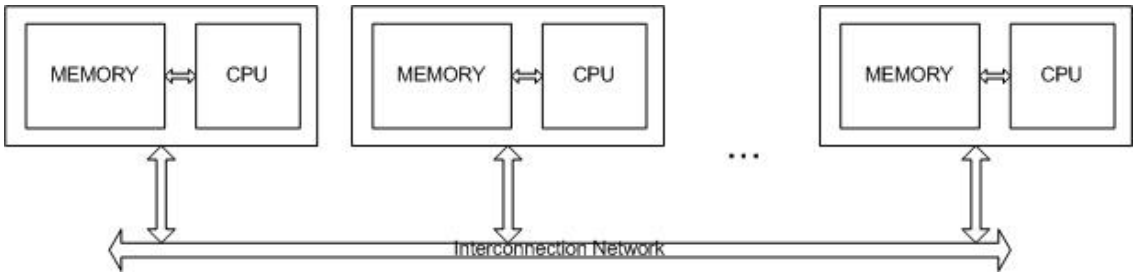

Figure 5: Distributed Memory Architecture

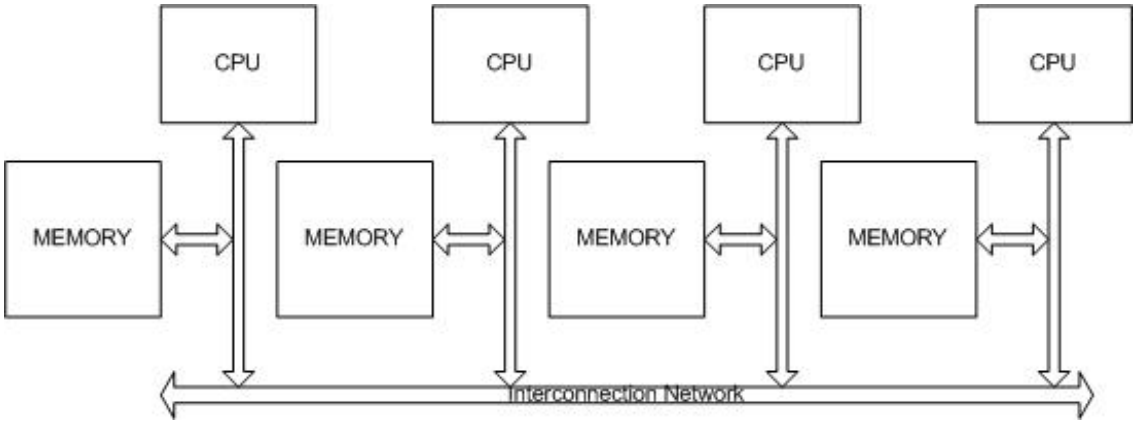

Figure 6: Distributed Shared Memory Architecture

B Cache Coherence

In this assignment we focus on the Shared Memory Architecture. As in an uniprocessor system, caches can help reducing the memory access delay. However, now each processor can read and write directly to local cache, as shown in Figure 7. Thus, when one processor writes to a memory location and its cache just stores it locally, a read of the same memory location on another processor can read a different value from its cache. Subsequently, this could lead to the wrong execution outcome of the program. To avoid this problem, cache coherence among different cache modules has to be implemented within the cache system.

Two major coherence protocols are the *snooping* protocol and *directory-based* protocol. In the *snooping* technique each cache monitors every request on the bus and changes the state in its cache lines accordingly. This protocol requires every transaction in the network to be visible to all caches. Thus it is most commonly used with a bus implementation.

The other coherence protocol is a *directory-based* protocol. In this protocol, directories are utilized to track data and memory requests in the memory system. The directories can be implemented distributed as well as centralized. In comparison with the snooping protocol, the directory based protocol is easier to use in network topologies other than a bus topology. Those other network implementations are more scalable than normal bus. This protocol is generally used in distributed shared memory architectures.

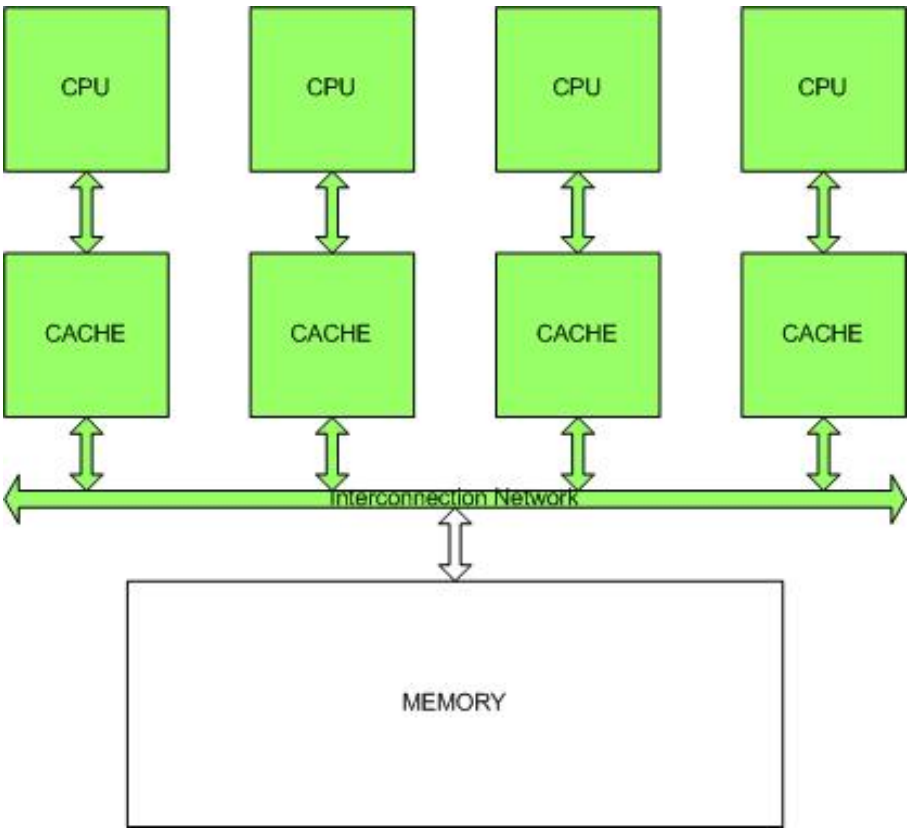

Figure 7: Cache associated with different processors in Shared Memory system

From the cache line state transition perspective, most of the cache coherence protocol consists of a portion of five states, which are *Modifed*, *Owned*, *Exclusive*, *Shared*, and *Invalid* (MOESI).

- **Invalid state** means that the current cache line does not hold valid data.
- **Shared state** represents the data is shared across the network and the value is the same as the

value in the main memory.

- **Exclusive state** means that the data is the only copy in the cache system, the value is the same as the value in the main memory.
- **Owned state** means that the cache line holds the correct data and it could be shared, however at the same time, the value could be different from the value in the main memory. Only one cache can hold the data in Owned state.
- **Modified state** means that the cache line holds the most recent and correct data, which is different from the value in main memory.

One of the most commonly used cache coherence protocols are different versions of the MESI protocol. In this version, the arcs in the transition diagrams are slightly different.

C Implementing a Bus in SystemC

This Appendix provides you with some sample code that may help you to implement your bus.

1. A bus could simply be represented by a group of signals. For instance, the address bus could be represented with `sc_signal_rv<32> BusAddress;` The corresponding port in cache module can be constructed as `sc_inout_rv<32> Port_BusAddress;`

   By connecting all the `Port_BusAddress` in a cache with `BusAddress signal`, all the cache modules could read from or write to the address bus. It is important to notice that the main memory delay has to be handled at some places, probably in cache.

2. A bus can also be packed into a bus class `Bus` which is implementing a bus interface `Bus_if`.

```
class Bus_if : public virtual sc_interface
{
public:
    virtual bool read(int addr) = 0;
    virtual bool write(int addr, int data) = 0;
};

class Bus : public Bus_if, public sc_module
{
public:
    // ports
    sc_in<bool>       Port_CLK;
    sc_signal_rv<32> Port_BusAddr;

public:
    SC_CTOR(Bus)
    {
        // Handle Port_CLK to simulate delay
        // Initialize some bus properties
    }

    virtual bool read(int addr)
    {
        // Bus might be in contention
        Port_BusAddr.write(addr);
        return true;
    };

    virtual bool write(int addr, int data)
    {
        // Handle contention if any
           Port_BusAddr.write(addr);
        // Data does not have to be handled in the simulation
        return true;
    }
}
```

Listing 1: A simple bus interface and class

   Within the cache, a bus port with bus interface can be represented with `sc_port<Bus_if> Port_Bus;`. The statement `Port_Bus->read(address);` in a cache can put a read request on bus.

   The code in Listing 1 is not complete, so please add more code to simulate your bus. You will need to implement a split-transaction bus for the assignment. You can add as many signals and ports as needed. Please notice that in the **MOESI** assignment, the bus might also need to transfer some information about cache line state information, because if cache A wants to read data which is at *Modified* state in cache B, cache B has the obligation to write the data back to main memory before cache A can read from it.

D Helper Functions

D.1 Trace files

The TraceFile class is a data type that represents a file that contains traces of memory requests from a programs execution. Instances of this class can be used to read those traces from trace files in order to drive a memory simulation. A trace file might contain traces from multiple processors if it has been created from a multi-processor system. The file psa.h contains the declaration of the TraceFile class while the file psa.cpp contains its implementation. These two files must be part of programs that use the class. We also provide a structure to collect statistics from the simulation. Below is brief explanation of using the trace file and how to store the statistics to be displayed at the end of simulation.

D.2 Opening a trace file

A function named init_tracefile is provided which takes the commandline arguments and creates a TraceFile object from the trace file specified in the first argument. The signature of this function is given as:

```
void init_tracefile(int* argc, char** argv[])
```

The created TraceFile object can then be accessed by the global pointer tracefile_ptr. This function also sets the global variable num_cpus to indicate how many CPU traces are present in the opened trace file. Furthermore, as argc and argv are passed by reference, the function removes the first argument so that the rest of the arguments can be parsed afterwards.

Alternatively, the tracefile can be opened manually by creating a TraceFile object. The TraceFile class has only one public constructor with the following signature:

```
TraceFile(const char* filename);
```

The parameter filename is a C-style string with the name of the file. On object creation (either on stack or dynamically using new) the file will be opened. If an error occurred when opening the file, a runtime_error is thrown.

On object destruction the file will be closed, if it wasnt already. You can explicitly close a file and free up resources before the object is destructed by using the close member function.

D.3 Reading from a trace file

Reading a trace for a processor from the file can be achieved using the next member function (after the file has been opened). The signature of this function is the following:

```
bool next(uint32_t pid, Entry& e);
```

The parameter pid is the id (index) of the processor for which the next trace-entry will be read and the e parameter is a reference to a structure of type TraceFile::Entry which will receive the entry read from the file. The function will return false if an error occurred. If there is no event for the current time step, the function will return true, and the entrys type will be ENTRY_TYPE_NOP. The Entry data type is a struct declared in the public interface of the class. Thus, when variables of this data type are declared, the name of the class must be prefixed with the name of the class (e.g. TraceFile::Entry).

If the file contains traces for more than one processor, then the reading of the traces does not need to be synchronized between processors. This means that each processor can call the next function individually. If a processor calls next when it already reached the end of its trace, it will keep reading ENTRY_TYPE_NOP.

```
// Open and set up the tracefile from cmdline arguments
init_tracefile(&argc, &argv);

TraceFile::Entry    tr_data;
Memory::Function    f;

// Loop until end of tracefile
while(!tracefile_ptr->eof())
{
  // Get the next action for the processor in the trace
  if(!tracefile_ptr->next(pid, tr_data))
  {
    cerr << "Error reading trace for CPU" << endl;
        break;
  }

  switch(tr_data.type)
  {
    case TraceFile::ENTRY_TYPE_READ:
      cout << "P" << pid << ": Read from " << tr_data.addr << endl;
      break;
    case TraceFile::ENTRY_TYPE_WRITE:
      cout << "P" << pid << ": Write to " << tr_data.addr << endl;
      break;
    case TraceFile::ENTRY_TYPE_NOP:
      break;
    default:
      cerr << "Error got invalid data from Trace" << endl;
      exit(0);
  }
}
```

Listing 2: An example of how the functions to handle trace files can be used

The code in Listing 2 will read the trace file provided at command line using the function `init_tracefile(&argc, &argv)` and will write all its trace entries to the standard output stream (`cout`). The `TraceFile` class also contains a number of member functions that return information about the file or its state and are shown in Table 1. How the functions in Table 1 can be used is also demonstrated in the example code in Listing 2.

Function Signature	Operation
`bool eof() const;`	Returns true when all processors reached the end of its trace.
`uint32_t get_proc_count() const;`	Returns the number of processors for which the file contains traces for.

Table 1: Table 1: Member Functions

D.4 Statistics

The `psa.cpp` file also comes with functions to gather and print statistics required for the simulator. These functions for the statistics are shown in Table 2. To explain the use of functions in Table 2, consider the code given in Listing 3. The cache hit or miss is generated randomly here.

```
case TraceFile::ENTRY_TYPE_READ:
    f = Memory::FUNC_READ;
    if(rand() % 2){
        stats_readhit(pid);
    }
    else{
        stats_readmiss(pid);
```

```
    }
    break;
```

Listing 3: Example code for reading tracefiles

Function Signature	**Operation**
`void stats_init();`	Initialize internal data for the number of processors required in the trace file. Requires `num_cpus` to be set.
`void stats_cleanup();`	Free the internal statistic data.
`void stats_print();`	Prints the statistics for each processor at the end of the simulation.
`void stats_writehit(uint32_t cpuid);`	Increments the writehit for the specified processor.
`void stats_writemiss(uint32_t cpuid);`	Increments the writemiss for the specified processor.
`void stats_readhit(uint32_t cpuid);`	Increments the readhit for the specified processor.
`void stats_readmiss(uint32_t cpuid);`	Increments the readmiss for the specified processor.

Table 2: Functions to calculate statistics