SystemC to Simulate Cache Memory

Victoria Peterson - 15476758

***Repository**: https://github.com/VictorianHues/MultiCoreProcessorSystems*

I. Introduction

START of text.
The simulation at-hand defines a 32KB, 8-way Set-Associative single-level cache implemented using SystemC to simulate memory access behaviors Cache Module transactions which returns the data to the CPU after accessing the specific Cache Line or Main Memory depending on address specifics and cache policy. The specific cache policy being used requires a *write-back* strategy where data is only written to the main memory when a cache line gets evicted, which requires that cache line to be "dirty" (thus not located in main memory when being evicted) and the "least-recently used" line in the 8-way set. Additionally, the cache policy defines a *write-allocate* strategy that requires the entire cache line to be loaded from Main Memory if the CPU sends an address that is not in the cache (Cache Miss) such that the combination of *write-back* and *write-allocate* resolves cache misses and maintains memory consistency.

II. Theory

From a basic broad perspective, the memory hierarchy contains three modules that each communicate with each other: the CPU, Cache, and Main Memory. The CPU executes instructions to read-from or write-to the Cache, which is a small and fast memory that has locality to the processor. These instructions are communicated using an address which contains a *Tag*, *Index*, and *Byte-in-line/Offset* where-in the *Index* searches for the set-in-cache containing the Cache Line with the desired data, while the *Tag* from the CPU address is compared to the *Tag* in the Cache line to determine if they match. If both *Tags* match, then it's a *Cache Hit*, which means the data being requested by the CPU from the Cache is the correct data, where-as the opposite *Cache Miss* means that the data is not there and the *Tags* do not match. The *Byte-in-line/Offset* is used to determine the Byte location in the Cache Line data, but in this simulation will be ignored due to the lack of real data being passed between modules.

During a *Cache Miss* or when the Cache Line is Invalid, the Cache must access the Main Memory to request the data directly in a much slower process. The case where a *Cache Miss* occurs means that the data being requested by the CPU is not in the Cache Line and needs to be pulled from memory; while the case where a Cache Line is Invalid means that the line has been invalidated for some reason, and the data in Cache cannot be sent back to the CPU, and thus must be read from Main Memory.

Writes that occur during *Cache Misses* use the *Write-back* policy to evict the least-recently-used line and write the new data from the CPU to the Cache Line. A special case occurs when a Cache Line is "Dirty" and a *Cache Miss* has occurred, meaning that the current Cache Line has newer data than what is currently stored in Main Memory, requiring that evicted data to be written to main memory before the new data from the CPU can be stored in the Cache Line.

III. Method

Looking at the implementation more specifically, communications between modules occur of "Ports" which use the SystemC library to take inputs through `sc_in`, send outputs through `sc_out`, and act as a shared data-transfer path. The CPU, Cache, and Main Memory Modules all use `sc_in<bool>` to receive a clock input signal to simulate system cycles, while the CPU will send an address to the Cache using `sc_out<uint64_t>` and the Cache will use the same to send an address to the Main Memory, both of which are coupled with a `sc_in<uint64_t>` in their partner modules to accept those addresses. Likewise, the CPU has a `sc_out<CacheMemory::Function>` matched by an `sc_in` in the Cache, and the Main Memory has a `sc_in<MainMemory::FunctionMem>` matched to a `sc_out` in the Cache, which are used specify whether a read or a write is being requested. These Ports also have matching opposite signals called `RetCode` which are used to communicate when those requested reads and writes have been completed, while the `sc_inout_rv<64>` is used to send data back and forth on a shared Port.

In the code, the CPU will send Read and Write requests through the ports and the cache will act on them depending on the Cache policies which may dictate a need to make Read or Write requests to the Main

Algorithm 1 Cache Module Initialization

```
1:  while true do
2:      Wait for CPU request event
3:      Read address, function (Read/Write) from CPU
4:      Extract tag, set index, byte offset from address
5:      Search cache set for matching tag
6:      if tag matches then
7:          cache_hit ← TRUE
8:      else
9:          cache_hit ← FALSE
10:     end if
11:     ...
12: end while
```

Algorithm 2 Cache Read Handling

```
    while true do
2:      ...
        if function = READ then
4:          if cache_hit then
                if Cache line is invalid then
6:                  Fetch data from main memory
                    Wait for single-cycle cache latency
8:                  Update existing cache line:
                      Set valid = TRUE
10:                   Set dirty = FALSE
                      Store fetched data
12:             end if
                Record read hit
14:         else
                Fetch data from main memory
16:             Wait for single-cycle cache latency
                Find least recently used (LRU) cache line
18:             if LRU line is dirty then
                    Write evicted data to main memory
20:             end if
                Update LRU cache line:
22:               Set tag = new tag
                  Set valid = TRUE
24:               Set dirty = FALSE
                  Store fetched data
26:             Record read miss
            end if
28:         Send data to CPU
            Signal read complete
30:         Wait for single-cycle cache latency
        end if
32:     ...
    end while
```

Memory. The CPU will send proper addresses with a *Tag*, *Index*, and *Byte-in-line/Offset*, but the data being sent and received to and from both the CPU and Main Memory will be placeholder data.

Additionally, the Cache itself is specified as 32KB with 8 sets per Cache and 32 byte line sizes, making each set 256 bytes (Number of Sets $\times$ Line Size) with 128 sets per Cache (Cache Size / Set Size).

A. Cache Module

Algorithm 1 initializes the Cache Module by starting a continuous while loop and waiting for CPU requests to occur, at which point the Cache will receive the Read/Write request and address containing the *Tag*, *Index*, and *Byte-in-line/Offset*. The Cache Set is identified using the *Index*, and is then searched for a matching *Tag* such that if a matching *Tag* is found, there is a Cache Hit, but if not, there is a Cache Miss.

When receiving a Read request from the CPU in Algorithm 2, the Cache determines whether a Cache Hit or Cache Miss occurs to decide the next action: If a Cache Hit occurs, the Cache line is checked for validity, and if it is valid, the data gets sent directly to the CPU, while and invalid line requires data to be fetched from Main Memory and stored in the line where a Cache Hit occurred. If a Cache Miss occurs, the data must be fetched from Main Memory while the least-recently-used cache line is evicted to Main Memory before storing the new data in the Cache Line and sending this data to the CPU.

On receipt of a Write request from the CPU in Algorithm 3, the Cache will read the data from the shared I/O data port and if a Cache Hit has occurred, the data will be written immediately to the Cache Line and the CPU will be informed of a completed task. If a Cache Miss occurs, the least-recently-used Cache line is found and evicted to the Main Memory if the line is ”Dirty”, and in either case where the line is Dirty or not, the new data gets stored in the least-recently-used cache line.

Once the Read or Write has been completed according the CPU request, the Least-Recently-Used queue is updated so that accessed cache line is moved to the most-recently-used position and all other cache lines in the Cache set iterate down one position. This then means the current request is completed and the CPU can begin waiting for another request from the CPU.

Algorithm 3 Cache Read Handling

```
while true do
    ...
    if function = WRITE then
        Read data from CPU
        if cache_hit then
            Record write hit
        else
            Fetch data from main memory
            Wait for single-cycle cache latency
            Find least recently used (LRU) cache line
            if LRU line is dirty then
                Write evicted data to main memory
            end if
            Record write miss
        end if
        Update cache line (hit index OR LRU line):
          Set tag = new tag
          Set valid = TRUE
          Set dirty = TRUE
          Store new data
        Signal write complete
        Wait for single-cycle cache latency
    end if
    Update LRU queue for the accessed cache line
end while
```

B. Memory Module

Algorithm 4 Memory Module Execution

```
while true do
    Wait for memory function request from cache
    Read address, function (Read/Write) from Cache
    Initialize placeholder data
    if Function is READ then
        Send placeholder data to Cache
        Signal read completion to Cache
        Wait for zero-time synchronization
    else if Function is WRITE then
        Read data from Cache
        Signal write completion to Cache
        Wait for zero-time synchronization
    end if
end while
```

The Memory Module code acts as an abstract system module that reads the communications from the Cache when a Read or Write request is made to the Main Memory. Much like the Cache module, this module reads the address and function (READ or WRITE) from the Cache communication, but doesn't use the same *Tag*, *Index*, *Byte-in-line/Offset* structure since the Main Memory uses Direct Memory Locations that are addressed differently from the CPU-to-Cache requests. The specifics of the Main Memory structure and communication is not in-scope of this project, but the template exists to better simulate and understand the process by which the Cache reads and writes to Main Memory.

IV. Results

The resulting Cache Misses and Cache Hits for Read and Write Requests from the CPU seem to be within specifications, with larger simulations using traces for matrix multiplication and fast Fourier transforms reaching Hit rates of around 96%. Additionally, using the "dbg_p1" trace to simulate a debugging set of Read and Write instructions, the resulting Hits and Misses for both Read and Write in tables I, II, III are identical.

The only difference is in the resulting Simulation Time, which is a result of the assignment specifications not specifying the cache and memory access times exactly, although the difference is within 1%, which is within a reasonable error range. Additionally, with 55 Cache Misses in total and a Memory Latency of 100 cycles, the estimated number of cycles for this number of Misses would be around 5500 assuming one Main Memory access occurs per Miss and some number of single-cycle cache latencies occur during operations.

A. Simulation Results - dbg_p1

TABLE I: Read Operations

.	CPU	Reads	RHit	RMiss
SIM	0	40	19	21
Expected	0	40	19	21

TABLE II: Write Operations

.	CPU	Writes	WHit	WMiss
SIM	0	60	26	34
Expected	0	60	26	34

TABLE III: Hit Rates (%)

.	CPU	RHitrate	WHitrate	Hitrate	Sim Time
SIM	0	47.5	43.33	45	5727 ns
Expected	0	47.5	43.33	45	5661 ns