**EE480 Assignment 2: Multi Cycle AXA**

Implementor’s Notes

Evan Helton, Michael Probst, Sairam Sri Vatsavai

University of Kentucky, Lexington, KY USA

evan.helton@uky.edu, mppr222@uky.edu, ssr226@uky.edu

**Abstract**

The objective of this project was to design, implement, and test a multi-core processor where each core has a fully-associative cache with 8, 16-bit long cache lines. The cache uses a least-recently-used (LRU) replacement policy and can handle transactional memory violations.

1. **General Approach**
   1. **Design**

We have designed this processor using the pipelined implementation from the previous project. We divided the implementation of this processor into 4 phases. First, we developed a cache for a single-core processor. Second, to interface the processor with the slow memory through the cache by using an arbiter. Third, to incorporate a second core into the processor, and develop cache coherence. Lastly, to handle transaction memory violations.

* 1. **Phase 1 – Cache**

The cache line is in total 36 bits long. The lines are designed as follows:

|  |  |  |
| --- | --- | --- |
| Bits | Name | Purpose |
| 35 | LINE INIT | Has this line been initialized? |
| 34 | TRAN | Has this line been used in a transaction |
| 33 | USED | 1-bit Timestamp for LRU policy |
| 32 | DIRTY | Has the value stored in this line been modified? |
| 31:16 | LINE MEMORY | Memory location associated with a line |
| 15:0 | LINE VALUE | Value associated with a line |

The cache processes requests using a simple state machine with the following states:

Standby - Default state. Cache waits until the core requires a value from a memory address. Then, it searches each line until it finds the requested memory address. After searching, the cache enters the Check Hit state.

Check Hit - If there is a hit, then the USED bit of the hit cache line is set to 1 and the value in the hit line is returned to the core. Also, the cache checks if the USED bits of all lines are 1. If so, the USED bits are all set to 0. If there is a miss, the cache checks if there is an uninitialized line. If so, the cache will read its new values into that line. If not, the LRU policy will take into effect and if the least-recently-used line is dirty, the value of that line will be sent to the arbiter to be written to memory and the line will be flushed. Finally, the cache sends the requested memory address to the arbiter and enters the Read state.

Read - The cache will remain in the Read state, stalling the processor, until the arbiter signals that slow memory has retrieved the value at the requested memory address. Once the value is received, it is written to the line determined in the Check Hit state along with the memory at which it was found. The processor is then resumed and the cache returns to the Standby state.

* 1. **Phase 2 – Slow memory Interface**

When the cache has miss, the arbiter is signaled to call the slow memory when given a valid address. The arbiter then passes this address to the slow memory and waits for the request to be completed. Once completed, the arbiter sends the address and the value at the indicated address back to the cache.

* 1. **Phase 3 – Multi Core Processor**

To incorporate a second core into the processor we needed to add more signals such that each core is distinguishable and can interact with other parts of the processor independently. We added a feature so that the core could start reading from instruction memory at a given offset, ensuring that each core can perform different tasks.

In order to handle concurrent requests to the slow memory from the cores, we needed to integrate logic within the arbiter to determine which core was given access to the slow memory.

* 1. **Phase 4 – Transaction Handling**

When the core identifies a SIGTMV error condition, it notifies the cache that the core is now running in transaction mode. If a core is in transaction and the other core attempts to read or write any address that is in this core's cache. If such transaction memory violation occurs, the processor will halt both cores.

1. **Testing**

A test program was developed for both cores throughout the implementation of this processor, testing that each phase was working correctly before moving on to the next. We identified AXA instructions that tested filling the cache lines and the LRU replacement policy. This also allowed us to test the functionality of receiving a value from slow memory. We ensured that only one core gets access to the slow memory by making both the cores access slow memory concurrently. We have tested the functionality of transaction memory, but only at an abstract level.

1. **Known Limitations**

We were not able to thoroughly test the functionality of transactional memory violation handling.

We did not implement the functionality of writing all lines in the cache to memory when a transaction occurs.

We did not implement the logic of the cache checking the other core’s cache when it has a miss.