Project 4 - UST-3400 Cache Simulator
Ben Frey and Joe Lambrecht
Professor Myre

UST-3400 Cache Simulator Documentation

UST-3400 Specifications:
32-Bit Architecture
Memory: 65536 words
Registers: 8

Cache Design

The design of the implemented cache simulator is dynamic based upon the user's intended cache configuration. The block size of each cache entry, the number of sets, and the set associativity of the cache are specified in the program input by the user with the flags -b -s, and – a, respectively. This allows for flexibility in program simulation by modifying certain cache specifications. Each cache entry consists of a valid bit to declare whether the entry is a legitimate entry. There is also a dirty bit to denote whether the cache data differs from memory. Each cache entry has a tag which is a unique identifier for each entry derived from the memory address. The block stores the data for the cache entry.

In a UST-3400 based cache simulator implementation, three operations of are special concern: an instruction fetch, a load word, and a store word. These three operation types facilitate all necessary communication between the processor and memory. By focusing on the implementation of these three operations, the cache will ideally become transparent to the ISA – meaning that any arbitrary UST-3400 program will run natively on the cache simulator implementation.

For any of these three operation types, when a dirty entry is encountered in cache, the entry is written back to memory (write back policy) before the existing cache entry is modified.

Simulator Implementation

The simulator begins by checking the cache for an entry with a tag matching the program counter. If the entry exists, the data for that entry is accessed and used as the instruction code. The LRU is updated. If the entry does not exist, memory is accessed at that address. A cache entry is created with the tag and block offset from the address, the data from memory, and a dirty bit is initialized as 0 (not dirty). The instruction is then sources from this cache entry and the entry's LRU is updated. This is known as an instruction fetch, one of the three important operations outlined in *Cache Design*, and occurs at the start of every UST-3400 processor cycle.

After finding the instruction code associated with each line, the simulator calls to the specific conditional for that instruction. Each instruction is handled differently, although some similarities exist within each format type.

Type R

For the R-type instructions, there are four segments: Register A, Register B, and Destination Register. After this is completed and error checking is performed, the instructions can be performed.

ADD

The ADD instruction reads the value from the register specified in Register A and the value from the register specified in Register B. It then adds these two values together and writes the summed value to the register specified in Destination Register.

NAND

The NAND instruction reads the value from the register specified in Register A and the value from the register specified in Register B. It then performs a bitwise AND on these two values, then a bitwise NOT. The resulting integer is written to the register specified in Destination Register.

Type I

For the I-type instructions, three segments are accessed: Register A, Register B, and the immediate field. After this is completed, the instructions can be performed.

LW

The LW instruction reads the value in Reg. B and sums that with the offset in the immediate field. The cache is checked for the address corresponding to the ALU result. If the associated tag is in the cache, the LRU for the entry is updated and the data for that address is written to the register specified by Reg. A. If the address corresponding to the ALU result is not in the cache, the address is found in memory and stored in the cache (if possible, a new entry is allocated, otherwise an existing entry is evicted and written back to memory if necessary), then written to the register specified by Reg. A.

SW

The SW instruction reads the value in Reg. B and sums that with the offset in the immediate field. The cache is checked for the address corresponding to the ALU result. If the result is in the cache, the LRU is updated, the data is stored in the cache entry, and the entry is marked as dirty. If the address corresponding to the ALU result is not in the cache, a new cache entry is created (if possible, a new entry is allocated, otherwise an existing entry is evicted and written back to memory if necessary) with the address from the ALU result, the data from Reg. A, and a dirty bit marked as true.

BEQ

The BEQ instruction reads the value stored in Register A and the value stored in Register B and compares them in the ALU. If the two are identical, a branch is performed. The PC is increased by 1, then increased by an additional amount equal to the value stored in the immediate field.

Type J
For the J-type instruction, there are two segments: Register A and Register B. The sole J-type instruction, JALR, then sets the current PC equal to the value read in from the register specified by Register B.

Type O
For the O-type instructions, the NOOP instruction simply increments the PC by 1, while the HALT instruction checks the cache for dirty entries and writes their data back to memory, then terminates the while loop in main, ending the program.

Design Considerations
During the design process of the cache simulator, few changes were made from the simulation program's conception to its final implementation. Initially, tags were given an invalid value of -1 such that the program would be able to find unallocated cache entries. Of course, through proper implementation of the valid bit, this design choice was revoked to take full advantage of the valid bit.

Another wrinkle during the design process inspired the inception of the test case "lwTest.2.1.8". Through the way an LRU value is updated in the cache simulator, it possible for a cache entry to be recently used and then remain "cold" for long enough such that its LRU value is incremented until it maxes out (at seven in this case). Upon a miss in the cache, the program previously would look for the first maximum LRU of seven within the set, evicting the entry if necessary. At the same time, it was possible for unallocated entries to be present within the set, meaning that eviction was unnecessary. Once this issue was resolved the cache simulator was restored to ideal working order.

Implementations Issues
No current known errors exist in the implementation of the UST-3400 cache simulator.