Aaron Gartner
Duran Carlson
CISC 340
Simulator with Cache Overview

Simulator Overview:

This program is a modification of an already existing single-cycle simulator, adjusted to replicate the behavior of a cache, customizable by block size, number of sets, and set associativity.

The program starts by using getopt to detect a file to be read in through the “-f” option argument. The block size in words, number of sets, and set associativity are also read in through the “-b”, “-s”, and “-a” flags, respectively. The program automatically ends if there is no input to be found. After a file is found, we read the lines to determine how many there are, then rewind the pointer.

Before building our state struct, we first initialize our cache as a 2D array of cache entry structs, each containing the entry’s valid bit, dirty bit, LRU value, tag, and block contents. The number of rows in this array equals the number of sets, and the number of columns equals the set associativity. We build a state struct containing the PC, memory contents, register contents, and cache hits/misses, all of which are initialized to 0. The lines of the file are read again, converted to integers (as the input file is expected to consist of decimal machine code instructions), then placed in the corresponding memory addresses. Helper functions are used to select bits from the integer instruction through a combination of bit shifting and AND operations, then assigning these values to register indices, the offset field, or whatever purpose they serve. The opcode is selected first, determining which instruction has been reached. Until a halt instruction is reached, the pc will be incremented, and whatever instruction is detected is executed accordingly. However, the only thing actually being printed out is the actions of our cache.

There are three situations where the cache needs to be interacted with: fetching an instruction, a LW instruction, and a SW instruction. The appropriate memory address is grabbed (the instruction address for a fetch, the address in the offset field for LW and SW), the set index is determined, and we check to see if the appropriate block is already inside the set by checking to see if there is a tag match and the matching entry is valid. If there is a match, the hits counter is incremented, the way index is set to this entry’s column, and the LRU value of the matching entry is set to 0, along with the LRU values of the other valid entries being updated appropriately. If no match is found, the miss counter is incremented, and the appropriate block must be inserted into the set. If any of the ways in the set are empty (i.e. invalid), the block is simply placed in the first invalid entry, marked valid, and the LRU values of the other valid entries are updated appropriately. If the set is full, however, the entry with the highest LRU value is evicted. If the entry’s dirty bit is 1, the block is written back to memory. The new entry is then inserted in the evicted entry’s place, its LRU value is set to 0, the LRU values of the other

entries are updated appropriately, and the block is updated. The completion of this process means the block was successfully loaded into our cache from memory. When this whole process is completed, if it was following a LW instruction, the contents associated with the memory address is transferred to the specified register. If it was following a SW instruction, the contents of the specified register are assigned to that memory address inside the cache entry, and the entry's dirty bit is set to 1.

When a halt instruction is reached, all remaining entries with a dirty bit of 1 get written back to memory, all entries get invalidated, the final number of hits and misses gets printed, and the program ends.

Simulator Shortcomings/Difficulties:

One of the major shortcomings in our opinion was not using functions for cache activity. The result of this was when a change needed to be made we had to change two other places in our code rather than just changing it in one. Another difficulty we had was getting our LRU to work properly. We had to literally print out our cache as a 2D matrix to figure out how our sets were being updated, at which point we reached the conclusion our LRU was not working as intended. We also spent an unfortunate amount of time trying to figure out how to determine block size/number of sets/associativity inside of the run function using only the cache as a reference, when we should have just sent those values in as additional arguments to the run function, which we eventually did.