**Overview of the single-cycle cache simulator**

Authors: Haoyi Shi & Haoyuan Du

**1. How does it work?**

We implement a cache simulator based on the previous single-cycle simulator. Cache structure is using for reducing the executing time (searching time) between processor and memory. Because when a processor wants a block of information from memory, it takes much time to look through the memory and find the target. But within the cache structure, it takes way less time to get the target from the cache. The cache structure has mainly three types: Direct mapping, Set Associate and Fully Associate.

The differences between these three types are that the cache has three dimentions which are the set\_size, the associativity\_number (n-way) and the block\_size. Thus, our simulator asks for three parameters and the machine code before simulating. In UST-3400 simulator, we use cache in three cases: loading the instructions from memory, loading an information when a lw instruction appears, storing an information when a sw instruction appears.

**Loading instructions from memory:** At the beginning, the simulator searches the target block in the cache, it will check all “Valid flag” in the appropriate set (set number is depended on PC in binary, and one set could have more than one entries depended on the associativity). If Valid flag is 1, then it will check the Tag. If the Tags are matched, then the processor successfully gets the instruction it needs. Otherwise, if no Tags are matched, then it has to ask cache to load new blocks from the memory.

Before actually loading a new block, the cache needs to evict the old blocks to memory (evict the least recently used blocks) if it’s dirty (have been updated by any reasons before) or evict them to nowhere if it’s not dirty. Finally, passing the new information to the processor.

**Loading an information when a lw instruction appears:** This case is very similar to “loading the instructions from memory”. However, instead of using PC as the address, we use the reg[b]+offset as the target address.

**Storing an information when a sw instruction appears:** Storing information is slightly different from loading information. For the last step, instead of loading the correct block to processor, the processor will send a block (word size – 32 bits) of information to the cache. Meanwhile, the “Dirty flag” of that entry will be set to 1.

**2. Some difficulties**

1. **Difficulties on tracking the lru (least recently used) entry:**

We used a little different way to track the lru because we start the project a little bit ealier. Basically, we also use the “adding and resetting” method to update each entry’s lru in a set. The initialize value for each entry is 0. When an entry is being used, we reset its lru to 0, and increment EVERY entries (include the one just being used) by 1. Therefore, each time when the cache needs to evict an entry of blocks, it will iterate through each lru value and find the largest one to evict. Then, reset the lru to 0, and increment every entries by 1 (as what I just mentioned). Our method keeps the differnces of lru among all entries so it's a valid method.

**2. Difficulties on allocating the 2D array:**

Since we rarely use malloc() function in c. To allocate a 2D array is a tough experience. Thanksfully, our best professor guide us by giving a nice and clear example which is way more make sense than the examples were searched from the GeeksforGeeks and StackOverflow.

As above, the cache is an array of pointers. Then, for each pointer in cache, we allocate the second dimension of memory space.

cachetype\*\* cache = (cachetype\*\*)malloc(sets\_number\*sizeof(cachetype\*));

for(int i = 0; i < sets\_number; ++i){

cache[i] = (cachetype\*)malloc(assoc\_number\*sizeof(cachetype));

}

Before Dr. Myre helps us, we use sizeof(int) instead of sizeof(cachetype). Also, we are not sure that either cache[][] or cache represents the whole cache structure or an entry in a cache structure. But now we understand, cache should be an entry of cache structure which has the same construction (in lecture ppt) such as: a valid bit, a dirty bit, a lru bit, a tag and n blocks.