Cache Simulator Algorithm

Cache line structure members:

Valid bit (initialize to 0)
Dirty bit (initlaize to 0)
Tag (initlaize to 0)
Index
Cache_block_index // "way" number
LRU "timer" count (initialize to 0)
MRU (initialize to 0)

1. Ask user (scanf) for cache's num_sets, num_ways (max 8-way, so 2,4,8), line_size (32-128B, so 32,64,128), and replacement_policy (True LRU = 0, 1-bit LRU = 1). User provides exponent for num_sets, num_ways, and line_size (no error checking)

2. Using scanf options, create cache architecture. This is done by creating an array of structs, where length=num_sets*num_ways. Populating index and cache block members based on num_sets and num_ways. Use a nested loop to store each cache block per index.

   Pseudocode:

```
index_value = 0
for(int = 0; i < num_sets*num_ways; int++)
        current_block = (i % num_ways) + 1      // Set cache block #s 1-user input
        cache_array[i].cache_block_index = current_block
        cache_array[i].index = set_index
        IF(current_block == num_ways)
                set_index++

Declare work flag named request_serviced      // whenever flag is set, go to next request
```

In a while(fgets != EOF)…

3. Parse trace and grab first request of list, and each consecutive memory request per loop iteration. Split access type and hex address from file string and save into variables *access_type* and *hex_address* respectively. Increment #cache_accesses.

4. Based on line_size, determine index bits start, and range determined by num_sets, and save in "temp" index variable. Determine byte select bits for index field start and index bit length from number of ways to ultimately determine and store index field in "temp" index variable to service request. Store remaining MSBs of address after the index field in "temp" tag variable. Ignore byte bits (subtract cache line size).

   Pseudocode on following page.

Pseudocode:

```
trace_parse(trace_str):
store trace_str[0] as access_type as trace_str[2:9] as hex_addr_str
        trace_str[0] = access_type
        strcpy(hex_addr_str, trace_str+2, 8)
num_index_bits = log2(num_sets)
num_bytesel_bits = log2(line_size)
tag_length = 32 – num_index_bits – num_bytesel_bits
call strtol(hex_addr_str, NULL, 16)       // pass string to convert, NULL, and base 16
request_index = extract_bits(hex_addr_str, num_index_bits, num_bytesel_bits)
request_tag = extract_bits(hex_addr_str, tag_length, num_index_bits + num_bytesel_bits)
```

5. IF (replacement policy = 0 (LRU)), increment all timer variables of caches lines with valid=1. Anytime a cache line is accessed or replaced, the count is reset to 0. Cache block with largest count in accessed will be evicted if cache set is full. If two or more counts are equal, first block with that count is replaced.

   For 1-bit LRU policy (replacement policy variable = 1), cache line's MRU is set anytime it is accessed. First cache line with MRU = 0 in requested cache set will be evicted, if no open blocks. If all cache blocks have MRU set, clear all MRUs in cache set and evict first block.

6. Determine read, write, or invalidate requests request by checking access type (read=0, write=1, invalidate=2). For each access type, determine hit or miss. If there is a miss, determine if there is room in the cache (cache set = request index, first invalid cache line) or cache line to evict to add the requested data to cache. Use a flag to know when request has been serviced by cache and to avoid unnecessary checking (for example, no point checking if it was a miss if it was a cache hit).

   Increment the following cache statistics:
   Accesses each time a new request is grabbed from .txt file.
   Reads each time the access type = 0.
   Writes when access type = 1.
   Invalidates when access type = 2.
   Hits if cache set = request's index field, cache tag = request's tag field, cache line is valid.
   Misses if tag is invalid or not found in cache set.
   Evictions when miss and no blocks are invalid (evict a block based on replacement policy).
   Writebacks when a dirty cache line is evicted.

   Pseudocode on following page.

Cache Simulator Algorithm

Pseudocode:

```
Clear flag (request_resolved = 0)
IF(replacement_policy == 0)      // LRU policy
        IF(strcmp(access_type,"0")==0)
                Num_reads++
                FOR(all cache slots)      // looking for hit
                        IF(request_resolve == 1)
                                Break;
                        IF(request_index = struct index & request_tag = struct tag & valid =1)
                                num_hits++
                                request_resolve = 1
                                cache_array[i].LRU_Count = 0
                declare replacement_index
                declare max_lru_count
                FOR(all cache slots)      // miss, looking for invalid (open slot)
                        IF(request_resolve == 1)
                                Break;
                        IF(request_index == struct index && valid==0)
                                num_misses++
                                Valid = 1
                                Dirty=0
                                Cache_array[replacement_index].tag = request_tag
                                Cache_array[replacement_index].lru_count = 0
                                Request_resolve = 1
                FOR(all cache slots)      // miss and find lru block (satisfy LRU)
                        IF(request_resolve == 1)
                                Break;
                        IF(index match && cache_block member =1) // finding max lru_count
                                // finding starting point
                                Replacement_index = loop index "I"
                                Max_lru_count = cache array[i].lru_count
                        ELSE IF(index match)     // comparing start point to find max lru_count
                                IF(cache array[i].lru_count > max_lru_count)
                                        Max_lru_count = cache array[i].lru_count
                                        Replacement_index = loop index "I"
                // Do replacement
                IF(request_resolve == 1)
                        Break;
                num_misses++
                num_evictions++
                Cache_array[replacement_index].tag = request_tag
                Cache_array[replacement_index].lru_count = 0
                Request_resolve = 1
                IF(cache_array[replacement_index].dirty == 1)  // if modified cache block
                        Num_writebacks++
```

```
ELSE IF(strcmp(access_type,"1")==0)
        Num_write++
        FOR(all cache slots)      // looking for hit
                IF(request_resolve == 1)
                        Break;
                IF(request_index = struct index & request_tag = struct tag & valid =1)
                        Num_hits++
                        Dirty=1
                        Request_resolve = 1
        declare replacement_index
        declare max_lru_count
        FOR(all cache slots)      // miss, looking for invalid (open slot)
                IF(request_resolve == 1)
                        Break;
                IF(request_index == struct index && valid==0)
                        num_misses++
                        Valid = 1
                        Dirty = 1
                        Cache_array[replacement_index].tag = request_tag
                        Cache_array[replacement_index].lru_count = 0
                        Request_resolve = 1
        FOR(all cache slots)      // miss and find lru block (satisfy LRU)
                IF(request_resolve == 1)
                        Break;
                IF(index match && cache_block member =1)// finding max lru_count
                        // finding starting point
                        Replacement_index = loop index "I"
                        Max_lru_count = cache array[i].lru_count
                ELSE IF(index match)    // comparing start point to find max lru_count
                        IF(cache array[i].lru_count > max_lru_count)
                                Max_lru_count = cache array[i].lru_count
                                Replacement_index = loop index "I"
        // Do replacement
        IF(request_resolve == 1)
                Break;
        num_misses++
        num_evictions++
        Cache_array[replacement_index].tag = request_tag
        Cache_array[replacement_index].lru_count = 0
        Request_resolve = 1
        IF(cache_array[replacement_index].dirty == 1)  // if modified cache block
                Num_writebacks++
        ELSE(dirty = 0)
                cache_array[replacement_index].dirty = 1

ELSE(strcmp(access_type,"2")==0)
        Num_invalidates++
        FOR(all cache slots)      // looking for hit
```

```
                    IF(request_index = struct index & request_tag = struct tag & valid =1)
                        num_hits++
                        num_evictions++
                        valid = 0
                        if(dirty == 1)
                            dirty = 0
                            num_writebacks++
                    ELSE(miss)
                        Num_misses++


ELSE(replacement_policy == 1)  // 1-bit LRU
    IF(strcmp(access_type,"0")==0), THEN call cache_read()
        Num_reads++
        FOR(all cache slots)      // looking for hit
            IF(request_resolve == 1)
                Break;
            IF(request_index = struct index & request_tag = struct tag & valid =1)
                num_hits++
                Cache_array[i].mru = 1
                Request_resolve = 1
        FOR(all cache slots)      // miss, looking for invalid (open slot)
            IF(request_resolve == 1)
                Break;
            IF(request_index == struct index && valid==0)
                num_misses++
                Valid = 1
                Dirty=0
                Cache_array[replacement_index].tag = request_tag
                Cache_array[replacement_index].mru = 1
                Request_resolve = 1
        FOR(all cache slots)      // miss and find mru=0
            IF(request_resolve == 1)
                Break;
            IF(index match && valid=1 && mru=0)
                Cache_array[i].tag = request_tag
                Cache_array[i].mru = 1
                Num_evictions++
                Num_misses++
                Request_resolve = 1
                IF(dirty=1)
                    Num_writebacks++
                    Cache_array[i].Dirty=0
        Declare replacement_index
        FOR(all cache slots)      // miss and find all mru=1
            IF(request_resolve == 1)
                Break;
```

```
                IF(index match && Cache_array[i].mru = 1)
                        Cache_array[i].mru = 0
                IF(index match && cache_block == 1)
                        Replacement_index = i
        // Do replacement
        IF(request_resolve == 1)
                        Break;
        Cache_array[replacement_index].tag = request_tag
        Cache_array[replacement_index].mru = 1
        Num_evictions++
        Num_misses++
        Request_resolve = 1
        IF(cache_array[replacement_index].dirty == 1)
                Num_writebacks++
                Cache_array[i].Dirty=0

ELSE IF(strcmp(access_type,"1")==0)
        Num_write++
        FOR(all cache slots)    // looking for hit
                IF(request_resolve == 1)
                        Break;
                IF(request_index = struct index & request_tag = struct tag & valid =1)
                        num_hits++
                        Cache_array[i].mru = 1
                        Request_resolve = 1
        FOR(all cache slots)    // miss, looking for invalid (open slot)
                IF(request_resolve == 1)
                        Break;
                IF(request_index == struct index && valid==0)
                        num_misses++
                        Valid = 1
                        Dirty= 1
                        Cache_array[replacement_index].tag = request_tag
                        Cache_array[replacement_index].mru = 1
                        Request_resolve = 1
        FOR(all cache slots)    // miss and find mru=0
                IF(request_resolve == 1)
                        Break;
                IF(index match && valid=1 && mru=0)
                        Cache_array[i].tag = request_tag
                        Cache_array[i].mru = 1
                        Num_evictions++
                        Num_misses++
                        Request_resolve = 1
                        IF(dirty=1)
                                Num_writebacks++
                        ELSE(dirty = 0)
                                Cache_array[i].dirty = 1
```

```
Declare replacement_index
FOR(all cache slots)      // miss and find all mru=1
        IF(request_resolve == 1)
                Break;
        IF(index match && Cache_array[i].mru = 1)
                Cache_array[i].mru = 0
        IF(index match && cache_block == 1)
                Replacement_index = i
// Do replacement
IF(request_resolve == 1)
        Break;
Cache_array[replacement_index].tag = request_tag
Cache_array[replacement_index].mru = 1
Num_evictions++
Num_misses++
Request_resolve = 1
IF(cache_array[replacement_index].dirty == 1)
        Num_writebacks++
ELSE(Cache_array[replacement_index].Dirty=0)
        Cache_array[replacement_index].dirty = 1


ELSE(strcmp(access_type,"2")==0)
        Num_invalidates++
        FOR(all cache slots)      // looking for hit
                IF(request_index = struct index & request_tag = struct tag & valid =1)
                        num_hits++
                        num_evictions++
                        valid = 0
                        if(dirty == 1)
                                dirty = 0
                                num_writebacks++
                ELSE(miss)
                        Num_misses++
```

7. Display the following cache simulation results.
    a. Number of cache accesses
    b. Number of cache reads
    c. Number of cache writes
    d. Number of invalidates
    e. Number of cache hits
    f. Number of cache misses
    g. Cache hit ratio = #reads / #accesses
    h. Cache miss ratio = #misses / #accesses
    i. Number of cache evictions
    j. Number of cache writebacks