CS2323 Computer Architecture

RISC-V CACHE SIMULATOR

Kaushal Morankar CS23BTECH11037

Introduction

In this assignment,which is a continuation of the previous assignment, we had to build a cache simulator, which simulated various tasks like printing the configuration status, calculating and displaying no of hits,misses, accessing, writing all the reads and writes along with their set and tag values in a separate file(dump command), etc. We also had to integrate various replacement policies like LRU,FIFO,Random, along with the 2 different writing approaches, write-back and write-through.

Main Implementation Approach

- Initialize a few important global variables: Hits,misses for no of hits and misses,lines,block_sizes for no_of_lines(no of sets) and the block size.
- To integrate reading from cache inside our previous load/store functions of the simulator,we define a boolean value cache_on to know if we are part of the cache_simulation and have to read from cache or not.
- Define a struct CacheEntry to denote an entry in the cache for a particular set, containing various fields like valid bit,dirty bit,tags, and also the data. Here, we store all the entries for a particular set(if associativity is 4, then 4 entries) together for a set, so a vector is used(size of vector will be the associativity number)

- The cache is in the form of a map, which maps a string(set) to its cache entries.

Reading from cache

- In the load instructions, first we check whether to read from cache or not using the cache_on boolean value.
- Then, we first dissect the input address into its set and tag value,and then search in the map for that index, then search all the entries in the set and match the tag. If hit, then we directly read from this map.
- If it’s a miss and we need to replace, then select which cache entry in the set to evict according to the replacement policy, and replace it.
- If valid bit 0, then no worry about replacing, directly put from memory to cache.
- Update the variable misses or hits accordingly.
- Similar set of code for all the load functions ld,lb,lh,lw,lbu,lhu,lwu.

Writing from cache

- Similar to the load part, we first dissect the address and search for the index in the cache map first, then replace according to the write policy.
- Writeback policy: If hit, then we check if the dirty bit is 1 then we store this old value to memory and put the new data value into the cache. If miss, then according to the replacement policy we find which block to evict(again if the dirty bit is 1 then store into memory before evicting), then store new data after evicting. If miss is due to valid bit being 0, then simply put the data in this cache entry, and put the tag values and set valid bit 1 and dirty bit 1.
- Writethrough policy: Here, the code for writing to cache will be similar to writeback, but with each write to cache, we simultaneously write to memory also.
- Allocation: Since we do allocate for write back, for write back when we have a miss, we add the new data to the cache. For write through, since it is without

allocate, in case of miss, we don’t write to the cache map ,we directly only write to memory.

Replacement policies

- FIFO: We maintain a vector pair, which stores the index and a queue of the cache entries in that index. The queue stores the cache entries in the order in which they are accessed, so that the first entry in the queue will be the one which was “first in”, so that entry will be replaced. If that entry is dirty, store it in the memory before evicting it. Pop this from queue, and replace with the new tag, and push this index back to the queue.
- LRU: We maintain a list for each index,containing the index and the associated tags with it.
- LFU: We have also included the LFU replacement policy which checks how often the tag occurs and replaces based on the count.
- Random: Using a random generator, generate a random number from 0 to the no of ways, and then evict that block(if dirty then put in to the memory of evicting)

Implementation of commands

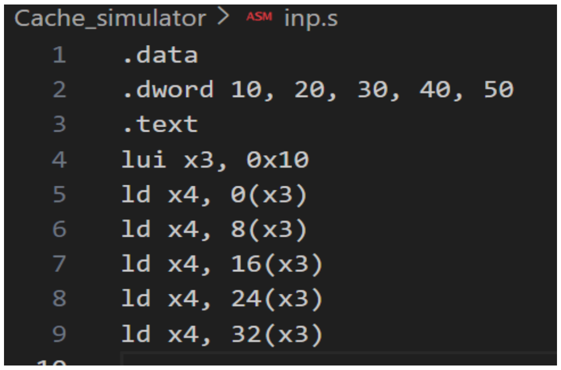
- *cache_sim enable config_file*: Take input from the config_file and set values for the different parameters(lines,block sizes,associativity,write and replacement policy). Find no of bits required to represent the set, to represent the set in binary format. Set boolean value to true. Clear the contents of the previous map for new simulation.
- *cache_sim disable*: Set boolean value cache_on to false.
- *cache_sim status*: Print if cache enabled or not, along with the configuration settings.

- *cache_sim invalidate*: Put all the dirty blocks currently in the cache into the memory, and then clear all the data stored in the cache, and also the data stored in the fifo,lru vectors
- *cache_sim dump myFile.out:* Iterate through the map, and accordingly print the set,tag values and if clean or dirty.
- *cache_sim stats:* Print the accesses,hits and misses and the hit rate
- After each load store instruction, in the load and store function, we write to the file filename.output using an ofstream object the address, the set and tag values, whether it was a miss or hit and if the bit is clean or dirty

Checking of testcases

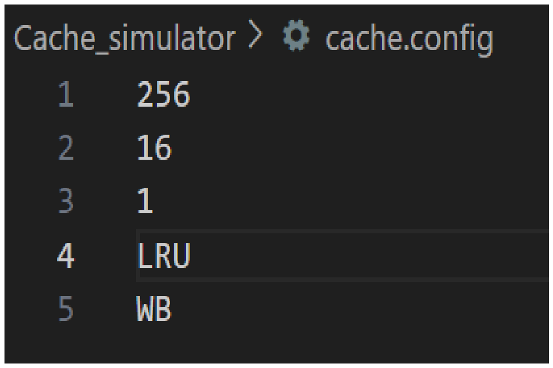
Test cases include the intensive testing of the homework 4 codes with all the variations possible and given in the assignment to code.

- **Input 1**



Cache_simulator > inp.s

```
.data
.dword 10, 20, 30, 40, 50
.text
lui x3, 0x10
ld x4, 0(x3)
ld x4, 8(x3)
ld x4, 16(x3)
ld x4, 24(x3)
ld x4, 32(x3)
```



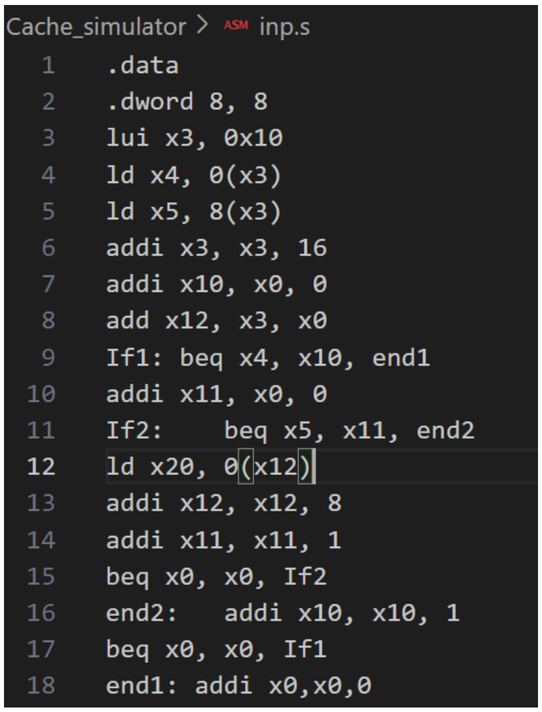
Cache_simulator > cache.config

```
256
16
1
LRU
WB
```

Simulation

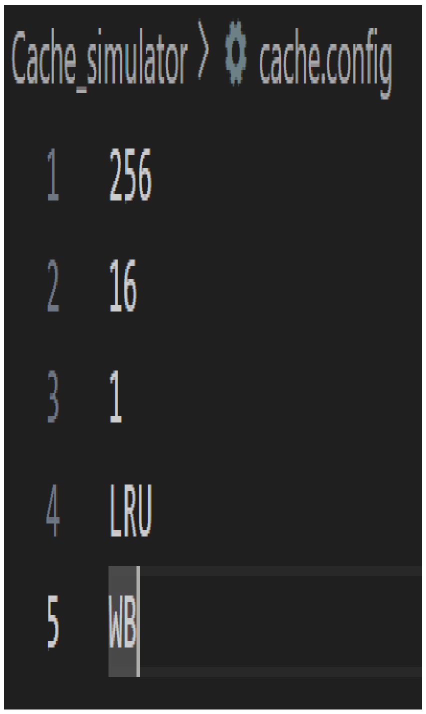
```
load inp.s

cache_sim enable cache.config

run
Executed lui x3, 0x10; pc=0x00000000
Executed ld x4, 0(x3); pc=0x00000004
Executed ld x4, 8(x3); pc=0x00000008
Executed ld x4, 16(x3); pc=0x0000000c
Executed ld x4, 24(x3); pc=0x00000010
Executed ld x4, 32(x3); pc=0x00000014
D-Cache statistics: Accesses=5, Hit=2, Miss=3, Hit Rate=0.4

cache_sim dump filename.ext
```

- **Input 2:**



Cache_simulator > inp.s

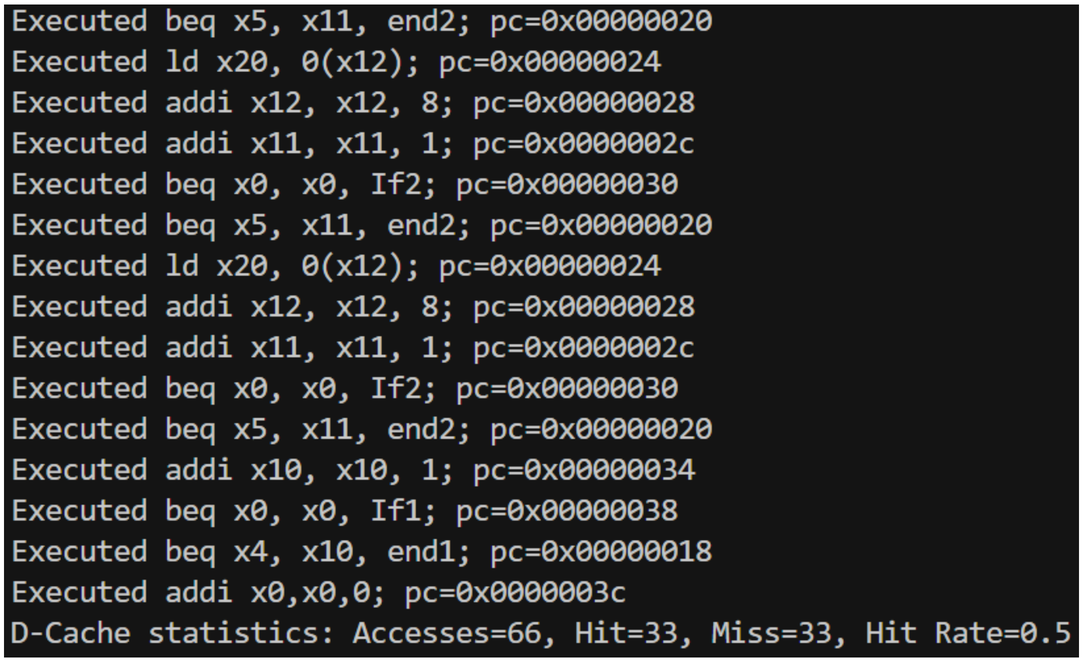
```
.data
.dword 8, 8
lui x3, 0x10
ld x4, 0(x3)
ld x5, 8(x3)
addi x3, x3, 16
addi x10, x0, 0
add x12, x3, x0
If1: beq x4, x10, end1
addi x11, x0, 0
If2:    beq x5, x11, end2
ld x20, 0(x12)
addi x12, x12, 8
addi x11, x11, 1
beq x0, x0, If2
end2:   addi x10, x10, 1
beq x0, x0, If1
end1: addi x0,x0,0
```



Cache_simulator > cache.config

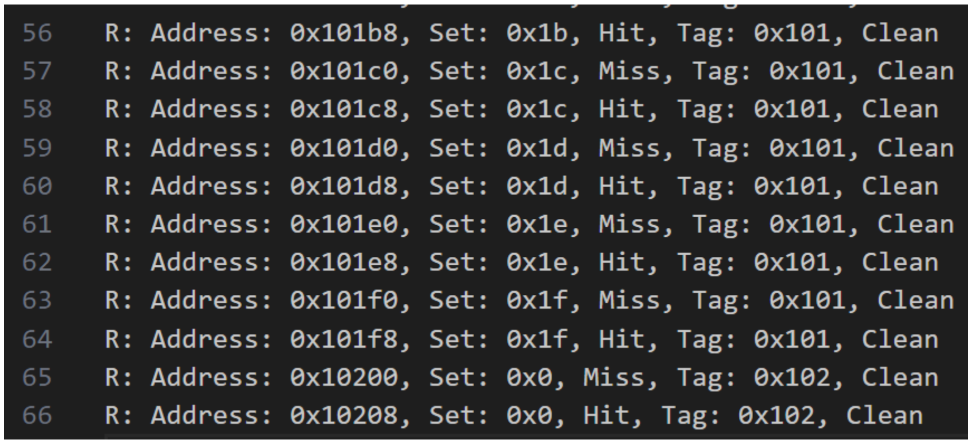
```
256
16
1
LRU
WB
```

Simulation



```
Executed beq x5, x11, end2; pc=0x00000020
Executed ld x20, 0(x12); pc=0x00000024
Executed addi x12, x12, 8; pc=0x00000028
Executed addi x11, x11, 1; pc=0x0000002c
Executed beq x0, x0, If2; pc=0x00000030
Executed beq x5, x11, end2; pc=0x00000020
Executed ld x20, 0(x12); pc=0x00000024
Executed addi x12, x12, 8; pc=0x00000028
Executed addi x11, x11, 1; pc=0x0000002c
Executed beq x0, x0, If2; pc=0x00000030
Executed beq x5, x11, end2; pc=0x00000020
Executed addi x10, x10, 1; pc=0x00000034
Executed beq x0, x0, If1; pc=0x00000038
Executed beq x4, x10, end1; pc=0x00000018
Executed addi x0,x0,0; pc=0x0000003c
D-Cache statistics: Accesses=66, Hit=33, Miss=33, Hit Rate=0.5
```

Inp.output



```
R: Address: 0x101b8, Set: 0x1b, Hit, Tag: 0x101, Clean
R: Address: 0x101c0, Set: 0x1c, Miss, Tag: 0x101, Clean
R: Address: 0x101c8, Set: 0x1c, Hit, Tag: 0x101, Clean
R: Address: 0x101d0, Set: 0x1d, Miss, Tag: 0x101, Clean
R: Address: 0x101d8, Set: 0x1d, Hit, Tag: 0x101, Clean
R: Address: 0x101e0, Set: 0x1e, Miss, Tag: 0x101, Clean
R: Address: 0x101e8, Set: 0x1e, Hit, Tag: 0x101, Clean
R: Address: 0x101f0, Set: 0x1f, Miss, Tag: 0x101, Clean
R: Address: 0x101f8, Set: 0x1f, Hit, Tag: 0x101, Clean
R: Address: 0x10200, Set: 0x0, Miss, Tag: 0x102, Clean
R: Address: 0x10208, Set: 0x0, Hit, Tag: 0x102, Clean
```

Drawbacks and Challenges faced

Challenges Encountered

1. **Implementation of Write-Back Policy**:
    - One of the most significant challenges was implementing the **write-back policy** in the cache simulation. Unlike a write-through policy where data is written directly to both the cache and main memory, a write-back policy requires updates to be made only to the cache initially. This necessitated careful handling of the dirty bit to ensure data was correctly written back to memory only when a block is evicted. Managing these scenarios and ensuring consistency between cache and main memory added complexity to the design.
2. **Replacement Policies**:
    - Another major hurdle was implementing and testing different **replacement policies** (such as LRU, FIFO, and Random). Designing efficient algorithms for determining which cache block to evict while maintaining performance was difficult, especially ensuring that policies worked seamlessly with varying cache sizes and associativity levels.

Drawbacks of the Simulator

1. **Lack of Pipelining Support**:
    - The current simulator is designed for basic instruction execution and cache simulation but does not support **pipelined execution**.
2. **No Multilevel Cache Simulation**:
    - The simulator is restricted to single-level cache modeling and does not support **multilevel cache hierarchies** (e.g., L1, L2, L3).

3. **Simplified Memory Operations**:
    - The simulator currently abstracts and simplifies memory operations. Real-world cache simulators often handle various memory operations such as block prefetching, cache coherence in multi-core processors, and more detailed memory latency modeling, which are not covered here.

Additional Challenges

- **Debugging and Testing**: Verifying the correctness of the cache behavior across different scenarios, especially when simulating edge cases with varying access patterns, required extensive debugging and testing.
- **Performance Optimization**: Ensuring that the simulator ran efficiently with larger input sizes was a challenge. Optimizing the data structures and algorithms to minimize runtime and memory usage required additional effort.
- **Handling Corner Cases**: Managing scenarios like cache thrashing, handling full cache conditions, and ensuring correct operation when repeatedly accessing the same data block were challenging.