ECE 586

Computer Architecture

Summer 2016

Final project report

‘Cache Simulator’

Venkata Hemanth Tulimilli

tulimil2@pdx.edu

Table of contents

1. Introduction:

Cache; one of the main component of a microprocessor is what we have to design. But how to design a totally important component? What is the size of data memory we have to consider for designing a cache simulator? What kind of memory are we designing; Data cache or Instruction cache or a Shared cache? What are the number of lines per set? What is the size of each line/ segment/ paragraph? What kind of algorithm are we using for handling cache misses? These are few questions we get in mind when we decide to design a cache.

Answers for each and every question differ from processor to processor and are totally based on trade-offs. But for the current implementation, we have the values fixed and are given in the below table.

Parameter	Description/ Choice
Line Size	32 Bytes
Associativity	4
Sets	1024
Data Size	128 kBytes
Replacement Policy	LRU
Miss Penalty	50 clock cycles
Interaction Policy with Main Memory	Write Allocate or Write Back

Table 1: Functional Specifications of L1 Cache for this project

Now we have required specifications. But we have to know few more concepts before proceeding in implementing the simulator.

1. We have line-size, number of lines/ associativity, and number of sets. But we do not know about how to implement replacement policy and interaction policy with main memory.
2. To implement each of these policies, we have to have extra bits for each cache line to know which line to be replaced when there is a cache miss or anything like that.
3. Therefore for replacement policy - LRU, calculations goes this way. We have associativity as 4. Therefore the number of bits for each line to implement replacement policy is Log_2 (4) which is 2. So for each line, we need 2 extra bits to implement replacement policy.
4. For implementing Interaction Policy with main memory, we use a single bit named “dirty bit” to keep tab on whether the data in specified line is modified without changing the same in main memory or not.

5. Also while initializing cache, we have a "valid bit" which states whether a cache line is being used or is it empty. By the help of this bit, we can increase the reachability of data in caches without any ambiguity of whether or not to delete a cache line.
6. Last but not the least, we have to have a particular label which specifies which 32 Bytes of main memory are present in a cache line. We name that field/label as "tag". For a given 32 bit byte addressable cache, we can find out the number of bits we need to save "tag" field as follows. There are 1024 sets and 32 bytes in each line. So the number of bits we need to represent 1024 sets is Log_2 (1024) which is 10 bits. Similarly we need Log_2 (32) bits which is 5 bits to represent number of bytes in a line. Therefore number of tag bits = 32 bits - 10 bits - 5 bits = 17 bits.

So now we are about to design a cache simulator for a data memory of 128 kBytes but we end up having a larger total cache memory including control bits. Which is 128 kB + 1024*4*(17 + 1 + 1 + 2)/8 B =138.5 kBytes.

Now that we know what are the control bits and how it changes the total cache memory, lets get started with how to approach implementation.

2. Project Tasks:

This project of developing a level 1 cache simulator can be simply classified into 6 stages.

1. Calculating required cache control fields; shown in Introduction part above.
2. Developing code for reading the input trace file and accessing each and every memory access from main memory instead of cache.
3. Implementing direct mapped cache which basically mean we have only a single line for every set.
4. Extending direct mapped cache implementation to 4 way set associative cache with LRU.
5. Implementing interaction policy(Write-back) for the L1 cache.
6. Implementing Debug commands and testing.

3. Implementation:

By following Project Tasks Implementation has been brought down to three main functions.

1. main()
2. check_cache()
3. set_lru()

3.1 main():

Main logic(heart) of the project lies in here beginning from reading input trace file to writing back modified cache line when there is a cache miss.

Local Control Flags:

hit_flag: This flag is set only if there is a cache hit for each and every memory reference instruction.

empty_flag: This flag is set if there are any empty cache lines available in a given set number(extracted from memory address).

version_flag: Used when encountered "-v" in an input trace file.

debug_flag: Used when encountered "-d" in an input trace file.

trace_flag: Used when encountered "-t" in an input trace file.

Program flow:

→ initialize cache

→ get memory access instruction from input trace file

→ check cache if it is a cache hit or a cache miss and it returns cache line number where the address is found(hit) or where data from memory should be placed(miss)

→ if required, write evicted cache line back to memory when dirty bit is set

→ if cache miss, replace cache line data with new data from memory and write tag field

→ getting memory access instructions from input trace file continues to repeat until End Of File is hit or "-v" debug command is encountered

3.2 check_cache():

It takes memory access instruction from input trace file as an input argument and search cache lines in the given set for a cache hit or cache miss. Then updating hit_flag is really important as the next code flow is totally dependent on hit_flag. This function also updates empty_flag which is used in the next function.

If there is a cache hit, it returns the hit line number to specify which line number is to be seen to find required data. If it is a cache miss, it returns the least recently used line number to specify which line number should be evicted.

3.3 set_lru():

This function takes line number as input and set the LRU bit field depending on hit_flag as well as empty_flag.

Main function repeats check_cache() and set_lru() repeatedly for each input in the input trace file.

4. Version Revisions:

Each and every version revision is based on project tasks.

Version 0.1: Implement reading from the input trace and directly accessing memory from main memory.

Version 1.0: Implement Direct mapped cache for all 1024 sets.

Version 1.1: Extend the implementation from Direct Mapped cache to 4-way set associative cache with LRU.

Version 1.2: Implement write-back policy for the 4-way set associative cache using dirty bit.

Version 1.3: Implement Debug commands and verifying whether it works or not.

5. Output Specs:

The below table 2 represents output specifications for 4 way set associative cache with 1024 sets.

Operations	Output by Simulator
Clock cycles with cache	133499700
Clock cycles without cache	16000000000
Accesses	32000000
Reads	24000000
Writes	8000000
Hits	29980000
Misses	2020000
Read hits	21980000
Read misses	2020000
Write hits	8000000
Write misses	0
Stream-ins	2020000
Stream-outs	9994
Miss ratio(%)	6.3125
Average Memory access time	4.1718 clock cycles per instruction

Table 2: Output Specifications obtained after executing Ultimate_Trace.txt which is the trace file obtained from HW1

If we implement the same cache but with 2 way set associative cache, we get miss ratio and average memory access time as 6.77% and 4.4361 clock cycles per instruction respectively whereas if we implement direct mapped cache with no change in any other input specifications, we get miss ratio and average memory access time as 9.47% and 6.0635 clock cycles per instruction.

6. Testing and Debugging:

S. No	Memory Access String, hex address
1	-d r 0x00000000 r 0x00050000 r 0x000A0000 w 0x000A0000 r 0x00000008 r 0x00050640 r 0x000A0000 w 0x000A0000
2	-d w 0x00000000 w 0x10000000 w 0x20000000 w 0x30000000 -t w 0x40000000 w 0x10000000 w 0x40000000 w 0x20000000 w 0x00000000 w 0x10000000 w 0x20000000 w 0x30000000
3	-t -d w 0x00000000 w 0x10000000 w 0x20000000 w 0x30000000 w 0x40000000 r 0x10000000 r 0x40000000 r 0x20000000 r 0x00000000 w 0x10000000 w 0x20000000 r 0x30000000 r 0x00000000 w 0x00000000 w 0x00000000 -d -t -d -t -d -d -d w 0x00000000 w 0x00000000 w 0x20000000 w 0x20000000 w 0x10000000 w 0x30000000 r 0x40000000 r 0x50000000 -v

Table 3: Sample Memory access Trace files for testing

Sample trace 1: While executing this trace bug I found is regarding Valid bit setting and I overcome it by setting valid bit.

Sample trace 2: While executing this trace bug I found is regarding LRU setting where I incremented LRU bits for every cache line and I overcome it by checking valid bit and introduced an if statement to make sure only LRU bits which are more recently used than the present line are incremented.

Sample trace 3: While executing this trace bug I found is regarding repeated debug commands and I overcome it by introducing another while loop which finds out if there are repeating debug commands as shown in Fig. 1 below.

```
// Handling "-v", "-t" and "-d" arguments and making sure arg contains either 'r' or 'w' only.
while (arg[0] == '-')
{
    switch (arg[1])
    {
        case 'v':
        case 'V':   version_flag = TRUE;
                    break;
        case 't':
        case 'T':   trace_flag = TRUE;
                    break;
        case 'd':
        case 'D':   debug_flag = TRUE;
                    break;
    }

    // If input trace file ends with "-v", "-t" and "-d", it will end the execution
    if (fscanf(fp, "%s", arg) == EOF)
    {
        break;
    }
}
```

Fig. 1: Logic implementation to neglect repeating debug commands

7. Conclusion:

Implementation of a cache in a uniprocessor system reduces average memory access time from 50 clock cycles to 4.1718 clock cycles per instruction which is a great achievement for a given cache specifications. As we see from the above Output Specs, change in associativity from 4 to 2 and 2 to 1 increases miss ratio as well as average memory access time per instruction. To find optimized cache specifications, we run through each and every possibility of varying associativity, cache size as well as many other factors. But the more the associativity is, the less miss ratio and average access time is.

8. Acknowledgements:

For any project to be successful the efforts have to be carried out in the right direction. But the advices, help and support and few little pushes in the right direction are equally necessary. First and foremost I would like to thank Professor Herbert Mayer for giving me this opportunity, sharing his ideas and pushing in the right direction of learning through the span of the course. The course structure that he has planned has made me good kind of workaholic which gave me the satisfaction to complete this interesting project successfully, in time. I would like to thank Sai Kiran Cherupally for his help and advices throughout the semester.

9. References:

1. https://www.d.umn.edu/~gshute/arch/cache-addressing.xhtml
2. http://web.cecs.pdx.edu/~herb/ece586s16/
3. Portland State University, ECE-586, Lecture Slides