國立清華大學資訊工程學系

CS 410000 --- 計算機結構

111 學年度上學期

Final Project

(Due: 2023.01.17)

Topic

(Programming Project)

The cache behavior simulation

Goal

i. Study and implement cache policies (LRU replacement policy). [70%]

ii. Design a cache indexing scheme to minimize cache conflict miss. [30%]

Introduction

In hierarchy memory system, the fast but small SRAM (static random access memory), cache, is used to keep a subset data of main memory. It tries to "guess" what data will be used by the processor in the future. When the processor issues a memory instruction (lw or sw), the cache could save the long DRAM (dynamic random access memory) latency, if the data is found in the cache, so called cache hit. If the data does not in the cache, i.e. cache miss, the processor must wait for the additional cache miss penalty. Since the latency of DRAM is hundred times slower than that of SRAM, cache hit rate becomes a critical factor of overall system performance. Thus, for decades, computer architects develop several cache policies trying to identify those data that will be accessed in the near future.

There are several cache related performance policies, including cache allocation policies, cache replacement policies, cache write policies, cache indexing schemes, etc. All of them have critical impact on the cache performance. In this project, we are going to implement a cache behavior simulator targeting specific cache policies. Given a series of memory accesses, a cache configuration (including cache capacity, number of cache associative, etc.). Your program should trace the cache behavior and report the cache performance.

The cache allocation policy decides possible places where a data block could occupy. For example, direct-map policy decides a single location for each data block.

Figure 5.7 from the textbook [1] illustrates the direct mapping scheme with 1024 cache sets. In this case, there are 4 bytes for each cache block, and, thus, 2 bits for byte offset. The cache indexing needs another 10 bits of 32 address bits for indexing 1024 sets. The other 20 bits are cache tag. Under a cache access, cache hit or miss can be simply derived by comparing the cache tag on the corresponding allocation. If the cache tag matches the one of the address, then it's a cache hit. Otherwise, the cache entry should be replaced by the new data block and update the cache tag information correspondingly.

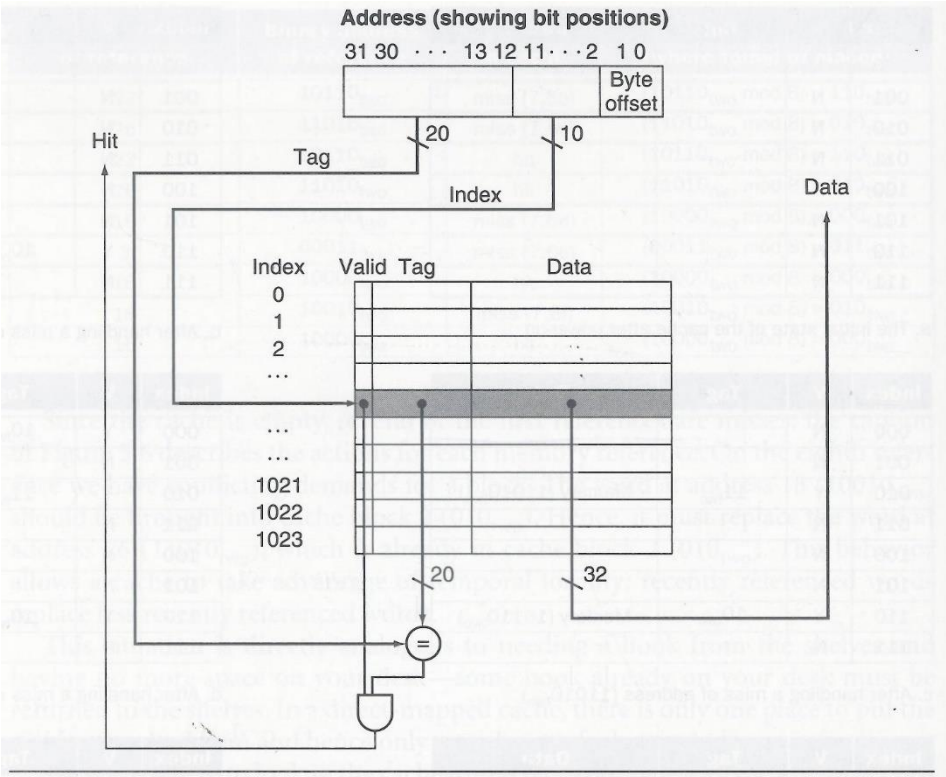


FIGURE 5.7 For this cache, the lower portion of the address is used to select a cache entry consisting of a data word and a tag. This cache holds 1024 words or 4 KB. We assume 32-bit addresses in this chapter. The tag from the cache is compared against the upper portion of the address to determine whether the entry in the cache corresponds to the requested address. Because the cache has 2^{10} (or 1024) words and a block size of one word, 10 bits are used to index the cache, leaving $32 - 10 - 2 = 20$ bits to be compared against the tag. If the tag and upper 20 bits of the address are equal and the valid bit is on, then the request hits in the cache, and the word is supplied to the processor. Otherwise, a miss occurs.

Another example of cache allocation policy could be a 4-way associative cache. Each data block has 4 possible allocations that can be occupied. Figure 5.17 from the textbook [1] shows a 4-way associative cache with only 256 cache sets. In this scenario, 8 bits of address are enough for indexing. The other 22 bits are cache tag. Apart from the direct map policy, there are four possible allocations for a data block. Each cache tag needs to be compared separately to decide whether it is a cache hit or miss under a cache access. If one of them is identical to the one of the address, then it is a cache hit. Otherwise, the cache replacement policy will decide a specific location among the possible locations for the new data block to occupy. **In this project, your program should take the number of associativity as an input variable and simulate the**

corresponding behavior. We may have hidden test cases with different associativity scenario for your program.

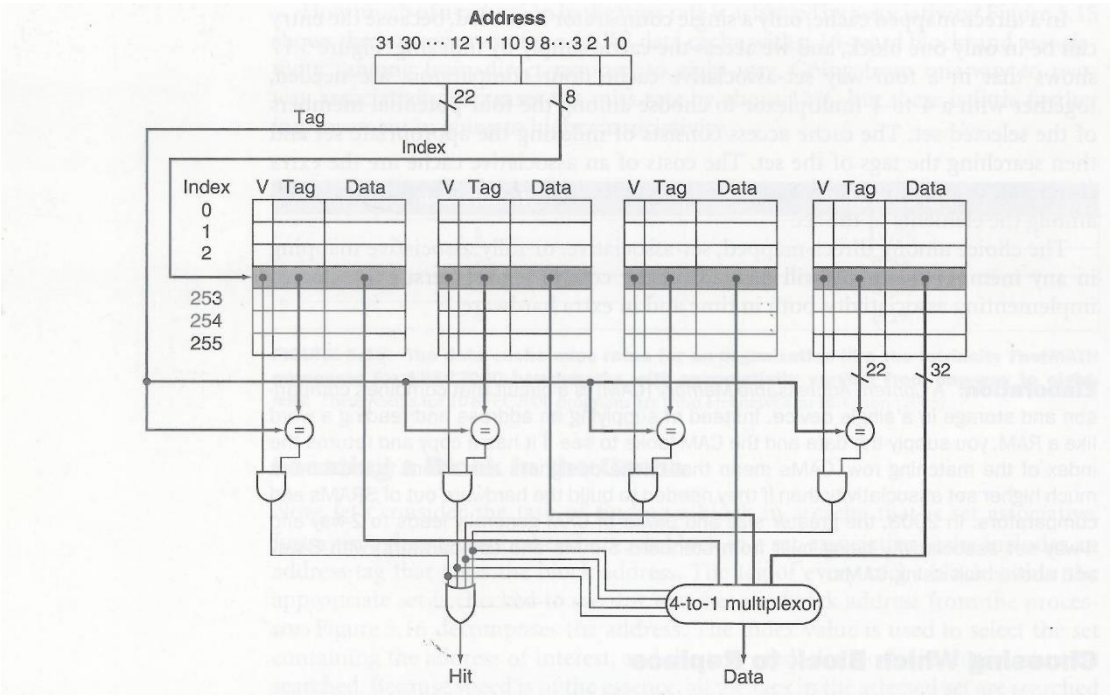
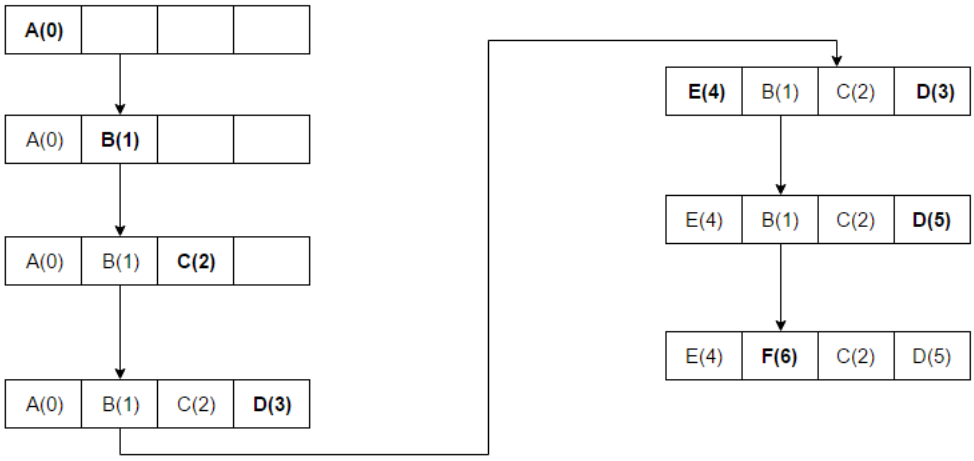


FIGURE 5.17 The implementation of a four-way set-associative cache requires four comparators and a 4-to-1 multiplexor. The comparators determine which element of the selected set (if any) matches the tag. The output of the comparators is used to select the data from one of the four blocks of the indexed set, using a multiplexor with a decoded select signal. In some implementations, the Output enable signals on the data portions of the cache RAMs can be used to select the entry in the set that drives the output. The Output enable signal comes from the comparators, causing the element that matches to drive the data outputs. This organization eliminates the need for the multiplexor.

When there is more than one possible allocation for data block to occupy, e.g. 4-way associative cache, the cache system has to follow a cache replacement policy to decide which one should be replaced. To achieve a higher hit rate, cache replacement policy will choose a data block that is least likely to be used in the near future. Many replacement policies have been proposed seeking to achieve a better hit rate considering hardware complexity and costs [4]. Among of them, LRU (least recently used) replacement policy is one of the most well-known cache replacement policy. In LRU replacement policy, the block replaced is the one that has been unused for the longest time. Consider the following figure as a 4-way associative, LRU example from Wikipedia [5]. The accessing sequence is: *A B C D E D F*. At the beginning, there is no valid data block in the cache. Thus, the first data blocks, *A*, *B*, *C*, and *D*, are cache misses. After that data block *E* will replace one of those data blocks and occupy the allocation. The LRU replacement policy will choose the one that is unused for the longest time, i.e. data block *A*. Then, the next access to data block *D* will have a cache hit. Finally, the data block *F* will replace the data block *B*, since it is unused for the longest time. **You are required to implement the LRU replacement policy in this project.**



As mentioned previously, cache holds a subset duplicated data from the main memory. After we update a data in the cache (sw instruction), the data will no longer be the same as the one in the main memory, i.e. data consistency issue. The cache write policies target this issue. It decides when to write the new data back to main memory. There are two basic cache write policies, namely write-through and write-back. Under the write-through policies, it always updates the new data all the way through cache and memory to guarantee that all data in cache and memory are consistent. For the write-back policy, on the other hand, it only updates the main memory when the data block is replaced (or evicted). To do so, it requires additional bit, dirty bit, to indicate whether the data block should be updated or not. **Since we target on a single level cache behavior in this project, your program does not need to consider the cache write policy.**

Lastly, the cache indexing scheme decides the cache set in which a data to be stored. If different data accessed frequently are mapped into a same location, those data blocks will keep kick others out from cache, and results in lots of cache misses, called the conflict misses. In this scenario, the system performance will be seriously degraded. For general cases (and for the above mentioned examples), designers usually choose the least significant bits (LSBs) of block address, i.e. without offset bits, to decide the cache set (the indexing bits in Figure 5.7 and Figure 5.17 from the textbook [1]). Nevertheless, some applications may have potential performance improvement under other indexing schemes [2]. **For this project, you are required to implement the LSB indexing scheme as baseline implementation. You will get the basic score for the baseline implementation (70%). In addition, the second topic of this project is to design a cache indexing scheme with the minimal cache miss count from the given addressing sequence (30%).**

Problem Definition

Give a cache with E sets, A-way set associativity, and O-bit offset. And, the addressing bus is M-bit. We need to select $K = \log_2 E$ bits among upper (M-O) address bits for indexing the cache.

There are totally $\binom{M-O}{K}$ possible valid indexing schemes. Your job is to (i) implement LSB scheme and (ii) find a valid indexing scheme with minimal cache misses under LRU replacement policy.

Example 1 – (direct-map or 1-way associative)

If the address is 8 bits ($a_7a_6a_5a_4a_3a_2a_1a_0$) and the cache has 8 sets, 4-byte block size, and uses direct map scheme, we need $\log_2 8 = 3$ bits for indexing 8 sets (#0~#7). Also, the lower $\log_2 4 = 2$ bits are offset bits that cannot be selected as indexing bits. Thus, there are $\binom{8-2}{3} = 20$ possible indexing schemes.

Access Example:

Reference Address	Indexing by ($a_4a_3a_2$) (LSB)	Allocation
00010000	00010000	#4
00101100	00101100	#3

The reference sequence is as follows:

Reference sequence ($a_7a_6a_5a_4a_3a_2a_1a_0$)	Indexing by ($a_4a_3a_2$) (LSB)		Indexing by ($a_7a_6a_5$)	
	Allocation	Cache hit/miss	Allocation	Cache hit/miss
00101100	#3	miss	#1	miss
00100000	#0	miss	#1	miss
00101100	#3	hit	#1	miss
00100000	#0	hit	#1	miss
Total cache miss count	2		4	

Example 2 – (2-way associative)

Assume the address is 8 bits ($a_7a_6a_5a_4a_3a_2a_1a_0$), the cache has 4 Sets with 4-byte block size and uses 2-way associative policy. We need $\log_2 4 = 2$ bits for indexing 4 sets (#0~#3) and $\log_2 4 = 2$ bits for offset. Hence, there are $\binom{8-2}{2} = 15$ possible indexing schemes.

The cache organization will be:

Set number	way0		way1	
#0	tag	data	tag	data
#1	tag	data	tag	data
#2	tag	data	tag	data
#3	tag	data	tag	data

Access Example:

Reference Address	Indexing by (a_3a_2) (LSB)	Allocation
`00010000`	`00010000`	#0
`00101100`	`00101100`	#3

The reference sequence is as follows:

Reference sequence ($a_7a_6a_5a_4a_3a_2a_1a_0$)	Indexing by (a_3a_2) (LSB)		Indexing by (a_5a_4)	
	Allocation	Cache hti/miss	Allocation	Cache hit/miss
`00000000`	#0 (way0)	miss	#0 (way0)	miss
`00010000`	#0 (way1)	miss	#1 (way0)	miss
`00100000`	#0 (way0)	miss	#2 (way0)	miss
`00000000`	#0 (way1)	miss	#0 (way0)	hit
`00101100`	#3 (way0)	miss	#2 (way1)	miss
`00000000`	#0 (way1)	hit	#0 (way0)	hit
`00101100`	#3 (way0)	hit	#2 (way1)	hit
Total cache miss count	5		4	

Input/output file example

Following the input/output file format is strictly necessary in this project. You will get no point without any excuse for any format violation that result in checking failure of the provided verifier.

In this project, we assume data blocks are byte addressable, and the byte address is used. For the input/output files, the index of the MSB (the first bit) always has the biggest indexing number. Meanwhile, the index of the LSB (the last bit) is always 0. For example, the last address in the following file, reference.lst, is 001011, which means bit indexes 5, 4, 2 (a_5, a_4, a_2) are 0, and bit indexes 3, 1, 0 (a_3, a_1, a_0) are 1.

Input file example: cache.org

```
Address_bits: 6
Number_of_sets: 4
Associativity: 2
Block_size: 1
```

Input file example: reference.lst

```
.benchmark testcase1
000000
000100
001000
000000
001011
000000
001011
.end
```

For the output file, indexing bits should be printed from MSB to LSB (from small to big).

Output file example: index.rpt

```
Address bits: 6
Number of sets: 4
Associativity: 2
Block size: 1

Indexing bit count: 2
Indexing bits: 3 2
Offset bit count: 0

.benchmark testcase1
000000 miss
000100 miss
001000 miss
000000 hit
001011 miss
000000 hit
001011 hit
.end

Total cache miss count: 4
```

Requirements

- Your program should follow the requirements.
  - Implemented in C/C++ programming language and compliable to g++ compiler. Two binary code should be generated, namly `arch_final_lsb` and `arch_final_opt`, representing using LSB indexing scheme and your developed indexing scheme accordingly.
  - **Pass the I/O filenames from command**, as following :

    ```
    $ ./arch_final_lsb cache.org reference.lst index.rpt
    $ ./arch_final_opt cache.org reference.lst index.rpt
    ```

    where arch_final is your binary code, cache.org, reference.lst are input files, index.rpt is the output file name. (The input files are ASCII format.)
    Notice that you will get no point if not follow the IO format requirement!!
  - Output the total cache miss count into the output file, i.e. index.rpt.
  - Terminate within a minute.
- Report; including flow-chart and description of algorithm and discussion.
- Demonstration. The time will be shown on class website before the final exam.

Grading

- Output file format – The output file should be generated in output file format.
- Correctness – The number of indexing bits and the count of cache miss should be correct.
- Cache miss count – The count of cache miss is the main performance criterion of the cache system.
- Run Time – Your program should not run over a minute.
- Demonstration – You have to daemon your code and the flow of your algorithm to TAs clearly.

Notice that this project plays an important role on the final score.
You are not going to pass the course if not doing this project!

Related Work

For the second topic, a previous work, “Zero Cost Indexing for Improved Processor Cache Performance “, is proposed by Tony Givargis [2]. To refer to this algorithm in your project is welcome, or your new idea is encouraged and will be given additional bonus.

References

[1] David A. Patterson and John L. Hennessy. 2008. Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design) (4th ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[2] Tony Givargis. 2006. Zero cost indexing for improved processor cache performance. ACM Trans. Des. Autom. Electron. Syst. 11, 1 (January 2006), 3-25. DOI=http://dx.doi.org/10.1145/1124713.1124715

[3] Code::Blocks, http://www.codeblocks.org/

[4] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. 2010. High performance cache replacement using re-reference interval prediction (RRIP). In Proceedings of the 37th annual international symposium on Computer architecture (ISCA '10). ACM, New York, NY, USA, 60-71. DOI: https://doi.org/10.1145/1815961.1815971

[5] Wikipedia, cache replacement policies – least recently used (LRU) https://en.wikipedia.org/wiki/Cache_replacement_policies#Least_recently_used_(LRU)