Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Fall Term 2018

SYSTEMS PROGRAMMING AND COMPUTER ARCHITECTURE
Assignment 11: Caches & Virtual Memory

Assigned on: **5th Dec 2018**
Due by: **11th 23:59 Dec 2018**

Part I: Pen & Paper Exercises, Cache

Question 1

The following problem concerns basic cache lookups

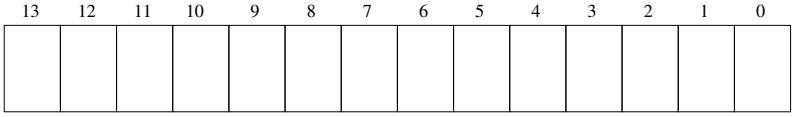
- The memory is byte adressable
- Memory accesses are to **1-byte words** (not 4-byte words)
- Physical addresses are 14 bits wide.
- The cache is 4-way set associative, with a 4-byte block and 64 total lines

In the following table, **all numbers are given in hexadecimal**. The *Index* column contains the set index for each set of 4 lines. The *Tag* columns contain the tag value for each line. The *V* column contains the valid bit for each line. The *Bytes 03* columns contain the data for each line, numbered left-to-right starting with byte 0 on the left.

The contents of the cache are as follows:

The box below shows the format of a physical address. Indicate (by labeling the diagram) the fields that would be used to determine the following:

CO The block offset within the cache line
CI The cache index
CT The cache tag



For the given physical address, indicate the cache entry accessed and the cache byte value returned **in hex**.

If there is a cache miss, enter "-" for "Cache byte returned".

4-way Set Associative Cache																
Index	Tag	V	Bytes 0–3	Tag	V	Bytes 0–3	Tag	V	Bytes 0–3	Tag	V	Bytes 0–3				
0	0C	0	03 3E CD 38	A0	0	16 7B ED 5A	40	0	8E 4C DF 18	58	0	FB B7 12 02				
1	3A	1	A9 76 2B EE	54	0	BC 91 D5 92	98	1	80 BA 9B F6	84	1	48 16 81 0A				
2	26	0	75 F7 3F C6	78	1	9E 3A 0F DA	26	1	00 4C B6 A8	5E	1	92 04 E5 2E				
3	B8	1	E0 22 19 3A	D2	0	02 B3 8F B6	D4	1	25 31 E1 02	C2	0	18 09 73 02				
4	54	1	86 B8 F0 C6	4C	1	AA 29 AE 16	56	1	76 46 80 6E	1C	1	13 EA A8 66				
5	F6	0	04 2A 32 6A	9E	0	B1 86 56 0E	CC	0	96 30 47 F2	06	1	F8 1D 42 30				
6	BE	0	2F 7E 3D A8	C0	0	27 95 A4 74	C4	1	07 11 6B D8	8A	1	C7 B7 AF C2				
7	A0	0	D6 A4 89 92	10	0	FD FE D6 DA	76	0	DE D5 CD 4A	E2	0	7C 68 3A 1A				
8	F0	1	ED 32 0A A2	E4	1	BF 80 1D FC	14	1	EF 09 86 2A	BC	1	25 44 6F 1A				
9	30	1	1E C2 AE 60	08	0	5C 3E DF F2	CA	0	25 CF 84 DA	5C	1	F1 6B DC DE				
A	38	1	5D 4D F7 DA	82	1	69 C2 8C 74	9C	1	A8 CE 7F DA	3E	1	FA 93 EB 48				
B	3A	1	61 C6 5E 74	64	0	03 97 BA 62	80	1	F8 11 72 12	E0	1	C5 EC 76 4E				
C	D4	0	17 52 75 2C	AE	0	62 89 EF 18	8E	0	BB 7D 8C 7C	68	0	26 57 7F C2				
D	DC	1	54 9E 1E FA	B6	1	DC 81 B2 14	00	0	B6 1F 7B 44	74	0	10 F5 B8 2E				
E	D6	0	14 9A 0D 4A	EA	1	C8 1D E6 6E	38	1	F3 38 F3 5C	64	0	6C 8F BD A8				
F	7E	1	32 21 1C 2C	FA	1	22 C2 DC 34	BE	1	BA DD 37 D8	B8	0	E7 A2 39 BA				

Figure 1: Cache content[1]

Physical address: `2BB2`

a) Physical address format (one bit per box)

13	12	11	10	9	8	7	6	5	4	3	2	1	0

b) Physical memory reference

Parameter	Value
Cache Offset (CO)	0x
Cache Index (CI)	0x
Cache Tag (CT)	0x
Cache Hit? (Y/N)	
Cache Byte returned	0x

Physical address: `098B`

a) Physical address format (one bit per box)

13	12	11	10	9	8	7	6	5	4	3	2	1	0

b) Physical memory reference

Parameter	Value
Cache Offset (CO)	0x
Cache Index (CI)	0x
Cache Tag (CT)	0x
Cache Hit? (Y/N)	
Cache Byte returned	0x

Question 2

Consider a direct mapped cache of size 1K bytes. Assume that the cache starts empty and that local variables and computations take place completely within the registers (only the matrix is in memory). Remember that `sizeof(int) == 4`.

```
1  int sum_nth_array_elements_twice(int m[128], int n) {
2
3      int i;
4      int tot = 0;
5
6      for (i = 0; i < 128; i += n) {
7              tot += m[i];
8      }
9      for (i = 0; i < 128; i += n) {
10             tot += m[i];
11     }
12     return tot;
13 }
```

a) What is the cache miss rate if the block size is 4 bytes and n is 1?

b) What is the cache miss rate if the block size is 16 bytes and n is 2?

Part II: Pen & Paper Exercises: Virtual Memory

Question 3

A virtual memory system is described by a number of parameters:

	Description
N	Number of addresses in virtual address space ($N = 2^n$)
M	Number of addresses in physical address space ($M = 2^m$)
P	Page size ($P = 2^p$) (in bytes)

A virtual address consists of:	
VPO	Virtual page offset (bytes)
VPN	Virtual page number

A physical address consists of:	
PPO	Physical page offset (bytes)
PPN	Physical page number

For a memory system with $n = 30$ and $m = 22$, determine the number of bits in the VPN, VPO, PPN, and PPO for the following page sizes:

P	# VPN bits	# VPO bits	# PPN bits	# PPO bits
512 Bytes				
1 KB				
2 KB				

Question 4

Consider a small memory system with a TLB and a L1 data cache. We make the following assumptions to simplify the question:

- The memory is byte addressable, each access is always to **a single byte**.
- Virtual addresses are 14 bits wide.
- Physical addresses are 12 bits wide.
- The page size is 64 bytes.
- The TLB is two-way set associative with 8 total entries.
- The L1 (data) cache is physically addressed, direct mapped, and has a 4-byte block size; there are 16 sets.

At some point in the execution of a program, the page table (only the top 16 entries are shown), the TLB, and the cache have these contents:

TLB: (all values are hexadecimal)

Set	Tag	PPN	Valid	Tag	PPN	Valid
0	05		0	12	42	1
1	02	20	1	04	32	1
2	01	22	1	07		0
3	01	22	1	02		0

Page table: (all values are hexadecimal)

VPN	PPN	Valid
00		
01	02	1
02	03	1
03		
04		
05		
06	22	1
07	22	1

VPN	PPN	Valid
08		
09	20	1
0a		
0b		
0c		
0d	04	1
0e		
0f		

Cache: (all values are hexadecimal)

Index	Tag	Valid	Block[0]	Block[1]	Block[2]	Block[3]
00	00	1	de	ad	fa	ce
01	31	0				
02	24	1	02	13	e1	de
03	22	1	22	23	e2	2e
04	21	0				
05	22	0				
06	18	0				
07	22	1	9a	01	00	de
08	20	0				
09	22	1	83	1a	09	ce
0a	20	1	0d	1f	f1	d0
0b	3a	0				
0c	3f	0				
0d	24	1	be	fb	57	02
0e	23	0				
0f	22	1	cf	7a	9b	a0

Blank entries indicate that contents of this block are not relevant.

For a given virtual address, indicate the VPN, the TLB index, the TLB tag, if there is a page fault and (if appropriate) the PPN.

To help you answer these questions, start with the virtual address in binary. Then show the physical address (in binary) and indicate the byte offset, the cache index and tag, if there is a cache hit, and the byte value retrieved from the cache (if appropriate). If there is a cache miss, enter "—" for "Cache byte returned". If there is a page fault, enter "—" for "PPN" and leave parts (c) and (d) blank.

a) Virtual address is 0x0268

(a) Virtual address (in binary)

13	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*

(b) Address translation

Parameter	Value
VPN	
TLB index	
TLB tag	
TLB hit (y/n)	
Page fault (y/n)	
PPN	

(c) Physical address (in binary)

11	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*

(d) Physical memory reference

Parameter	Value
Byte offset	
Cache index	
Cache tag	
Cache hit (y/n)	
Cache byte returned	

b) Virtual address is 0x0197

(a) Virtual address (in binary)

13	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*

(b) Address translation

Parameter	Value
VPN	
TLB index	
TLB tag	
TLB hit (y/n)	
Page fault (y/n)	
PPN	

(c) Physical address (in binary)

11	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*

(d) Physical memory reference

Parameter	Value
Byte offset	
Cache index	
Cache tag	
Cache hit (y/n)	
Cache byte returned	

c) Virtual address is 0x035e

(a) Virtual address (in binary)

13	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*

(b) Address translation

Parameter	Value
VPN	
TLB index	
TLB tag	
TLB hit (y/n)	
Page fault (y/n)	
PPN	

(c) Physical address (in binary)

11	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*

(d) Physical memory reference

Parameter	Value
Byte offset	
Cache index	
Cache tag	
Cache hit (y/n)	
Cache byte returned	

d) Virtual address is 0x021a

(a) Virtual address (in binary)

13	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*

(b) Address translation

Parameter	Value
VPN	
TLB index	
TLB tag	
TLB hit (y/n)	
Page fault (y/n)	
PPN	

(c) Physical address (in binary)

11	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*

(d) Physical memory reference

Parameter	Value
Byte offset	
Cache index	
Cache tag	
Cache hit (y/n)	
Cache byte returned	

Hand In Instructions for Part I & II

This part is a paper exercise. If you want your solution to be revised please hand it in during your exercise class on the due date.

Part III: Cachelab

Overview

This exercise will help you understand the impact that cache memories can have on the performance of your C programs.

You will write a small C program (about 200-300 lines) that simulates the behavior of a cache memory.

Downloading the assignment

Download the cachelab-handout.tar from the course website.

You must run this code on a 64-bit x86-64 machine. Start by copying `cachelab-handout.tar` to your working directory in which you plan to do your work. Then give the command

```
linux> tar xvf cachelab-handout.tar
```

This will create a directory called `cachelab-handout` that contains a number of files. You will be modifying one file: `csim.c`. To compile the program, type:

```
linux> make clean
linux> make
```

Reference Trace Files

The `traces` subdirectory of the handout directory contains a collection of *reference trace files* that you will use to evaluate the correctness of the cache simulator you write. The trace files are generated by a Linux program called `valgrind`. For example, typing

```
linux> valgrind --log-fd=1 --tool=lackey -v --trace-mem=yes ls -l
```

on the command line runs the executable program "`ls -l`", captures a trace of each of its memory accesses in the order they occur, and prints them on `stdout`.

`Valgrind` memory traces have the following form:

```
I 0400d7d4,8
 M 0421c7f0,4
 L 04f6b868,8
 S 7ff0005c8,8
```

Each line denotes one or two memory accesses. The format of each line is

```
[space]operation address,size
```

The *operation* field denotes the type of memory access: "I" denotes an instruction load, "L" a data load, "S" a data store, and "M" a data modify (i.e., a data load followed by a data store). There is never a space before each "I". There is always a space before each "M", "L", and "S". The *address* field specifies a 64-bit hexadecimal memory address. The *size* field specifies the number of bytes accessed by the operation.

Writing a Cache Simulator

You will implement the simulator in the `csim.c` file. The simulator takes a `valgrind` memory trace as input, simulates the hit/miss behavior of a cache memory on this trace, and outputs the total number of hits, misses, and evictions.

We have provided you with the binary executable of a *reference cache simulator*, called `csim-ref`, that simulates the behavior of a cache with arbitrary size and associativity on a `valgrind` trace file. It uses the LRU (least-recently used) replacement policy when choosing which cache line to evict.

The reference simulator takes the following command-line arguments:

```
Usage: ./csim-ref [-hv] -s <s> -E <E> -b <b> -t <tracefile>
```

- `-h`: Optional help flag that prints usage info
- `-v`: Optional verbose flag that displays trace info
- `-s <s>`: Number of set index bits ($S = 2^s$ is the number of sets)
- `-E <E>`: Associativity (number of lines per set)
- `-b <b>`: Number of block bits ($B = 2^b$ is the block size)
- `-t <tracefile>`: Name of the `valgrind` trace to replay

The command-line arguments are based on the notation (s, E, and b) from page 597 of the CS:APP2e textbook. For example:

```
linux> ./csim-ref -s 4 -E 1 -b 4 -t traces/yi.trace
hits:4 misses:5 evictions:3
```

The same example in verbose mode:

```
linux> ./csim-ref -v -s 4 -E 1 -b 4 -t traces/yi.trace
L 10,1 miss
M 20,1 miss hit
L 22,1 hit
S 18,1 hit
L 110,1 miss eviction
L 210,1 miss eviction
M 12,1 miss eviction hit
hits:4 misses:5 evictions:3
```

Your job is to fill in the `csim.c` file so that it takes the same command line arguments and produces the identical output as the reference simulator. Notice that this file is almost completely empty. You'll need to write it from scratch.

Programming Rules

- Your simulator must work correctly for arbitrary s, E, and b. This means that you will need to allocate storage for your simulator's data structures using the `malloc` function. Type "man malloc" for information about this function.

- For this exercise, we are interested only in data cache performance, so your simulator should ignore all instruction cache accesses (lines starting with "I"). Recall that `valgrind` always puts "I" in the first column (with no preceding space), and "M", "L", and "S" in the second column (with a preceding space). This may help you parse the trace.
- For the grader to evaluate your implementation, you must call the function `printSummary`, with the total number of hits, misses, and evictions, at the end of your `main` function:

```
printSummary(hit_count, miss_count, eviction_count);
```

- For this this exercise, you should assume that memory accesses are aligned properly, such that a single memory access never crosses block boundaries. By making this assumption, you can ignore the request sizes in the `valgrind` traces.

Evaluation

You can run your cache simulator using different cache parameters and traces. There are eight test cases:

```
linux> ./csim -s 1 -E 1 -b 1 -t traces/yi2.trace
linux> ./csim -s 4 -E 2 -b 4 -t traces/yi.trace
linux> ./csim -s 2 -E 1 -b 4 -t traces/dave.trace
linux> ./csim -s 2 -E 1 -b 3 -t traces/trans.trace
linux> ./csim -s 2 -E 2 -b 3 -t traces/trans.trace
linux> ./csim -s 2 -E 4 -b 3 -t traces/trans.trace
linux> ./csim -s 5 -E 1 -b 5 -t traces/trans.trace
linux> ./csim -s 5 -E 1 -b 5 -t traces/long.trace
```

You can use the reference simulator `csim-ref` to obtain the correct answer for each of these test cases. During debugging, use the `-v` option for a detailed record of each hit and miss.

For each test case, your implementation should output the correct number of cache hits, misses and evictions for that test case.

Working on the exercise

We have provided you with an autograding program, called `test-csim`, that tests the correctness of your cache simulator on the reference traces. Be sure to compile your simulator before running the test:

```
linux> make
linux> ./test-csim
                        Your simulator     Reference simulator
Points (s,E,b)    Hits  Misses  Evicts    Hits  Misses  Evicts
     3 (1,1,1)       9       8       6       9       8       6  traces/yi2.trace
     3 (4,2,4)       4       5       2       4       5       2  traces/yi.trace
     3 (2,1,4)       2       3       1       2       3       1  traces/dave.trace
     3 (2,1,3)     167      71      67     167      71      67  traces/trans.trace
     3 (2,2,3)     201      37      29     201      37      29  traces/trans.trace
     3 (2,4,3)     212      26      10     212      26      10  traces/trans.trace
     3 (5,1,5)     231       7       0     231       7       0  traces/trans.trace
     6 (5,1,5)  265189   21775   21743  265189   21775   21743  traces/long.trace
    27
```

For each test, it shows the number of points you earned, the cache parameters, the input trace file, and a comparison of the results from your simulator and the reference simulator.

Here are some hints and suggestions for working on your simulator:

- Do your initial debugging on the small traces, such as `traces/dave.trace`.
- The reference simulator takes an optional `-v` argument that enables verbose output, displaying the hits, misses, and evictions that occur as a result of each memory access. You are not required to implement this feature in your `csim.c` code, but we strongly recommend that you do so. It will help you debug by allowing you to directly compare the behavior of your simulator with the reference simulator on the reference trace files.
- You can use the `getopt` function to parse your command line arguments. You'll need the following header files:

  ```
  #include <getopt.h>
  #include <stdlib.h>
  #include <unistd.h>
  ```

  See "`man 3 getopt`" for details.
- Each data load (L) or store (S) operation can cause at most one cache miss. The data modify operation (M) is treated as a load followed by a store to the same address. Thus, an M operation can result in two cache hits, or a miss and a hit plus a possible eviction.

Handing in Your Work

If you want to hand-in your solution you can upload your code in a folder called assignment12 to SVN.