Homework #5 – Caching and Virtual Memory

START EARLY

Due date: see course website

Directions:

- For short-answer questions, submit your answers in PDF format as a file called <NetID>-hw5.pdf. ***Word documents will not be accepted.***
- For programming questions, submit your source file using the filename specified in the question.
- **You must do all work individually, and you must submit your work electronically via Sakai.**
    - All submitted code will be tested for suspicious similarities to other code, and the test will uncover cheating, even if it is "hidden" (by reordering code, by renaming variables, etc.).

Q1. Cache policies

[5 points] Why are write-back caches usually also write-allocate? *Hint: see "Cache Interaction Policies with Main Memory" linked on the course site.*

Q2. Cache performance

[5] Your L1 data cache has an access latency of 1ns, and your L2 cache has an access latency of 10ns. Assume that 90% of your L1 accesses are hits, and assume that 100% of your L2 accesses are hits. What is the average memory latency as seen by the processor core?

Q3. Cache layout

[20] You have a 64-bit machine and you bought 4GB of physical memory. Pages are 64KB.

(a) [2] How many virtual pages do you have per process?
(b) [2] How many physical pages do you have?
(c) [2] In the translation from a virtual address to a physical address, how many bits of VPN are you mapping to how many bits of PPN (assuming you have just enough bits in the physical address for the amount of physical RAM present)?
(d) [2] How big does a page table entry (PTE) need to be to hold just a single PPN?
(e) [2] How many PTEs fit on a page, assuming PTEs are the size computed in part (d)?
(f) [2] How many pointers fit on a page?
(g) [2] How big would a flat page table be for a single process, assuming PTEs are the size computed in part (d)?

(h) [2] What are the virtual page offset bits for virtual address 35012? What are the physical page offset bits for virtual address 35012 after it has been translated?

(i) [4] Does a TLB miss always lead to a page fault? Why or why not?

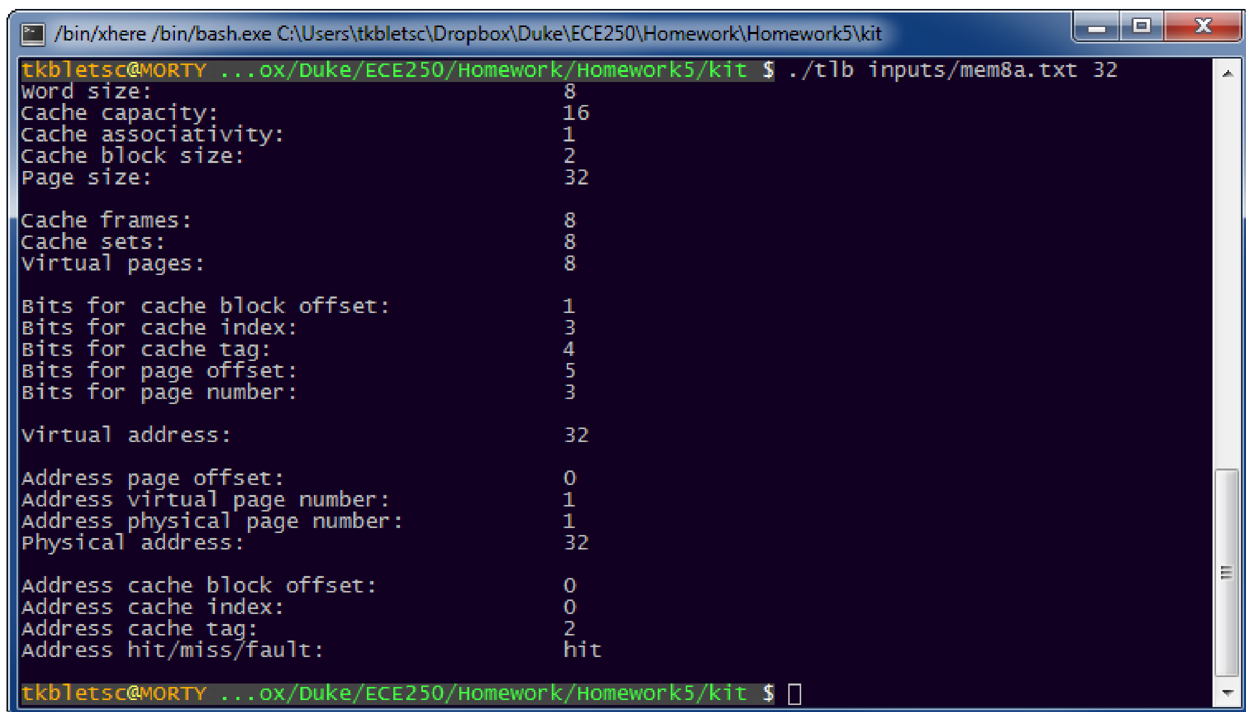
Q4. TLB Calculator Program

[100 points]

Overview

You will write a C program that takes two inputs on the command line: (1) a file describing a computer and its memory state (including word size, cache structure, page size, current cache content, and current page table), and (2) a single memory address. This program will do virtual to physical address translation as well as a cache lookup and determine, among other things, if the access is a page fault, cache hit, or cache miss.

See below for an example run:



Input file

The format of the input file is nothing but a whitespace-delimited sequence of numbers (hint: this is a indicator that you can get by with nothing but fscanf()). The format looks as follows:

```
<word-size> <cache-capacity> <cache-ways> <cache-block-size> <page-size>

 <tag> <tag> ... <tag>        (tags for set 0, as many as the associativity (ways))
 <tag> <tag> ... <tag>        (tags for set 1, as many as the associativity (ways))
   ...
 <tag> <tag> ... <tag>        (tags for set N-1, as many as the associativity (ways),
                               where N is the number of sets)

<ppn>                         (physical page number for virtual page 0; set to -1 if invalid)
<ppn>                         (physical page number for virtual page 1; set to -1 if invalid)
  ...
<ppn>                         (physical page number for virtual page N-1,
                               where N is the number of virtual pages; set to -1 if invalid)
```

All values above are decimal integer numbers. Specifically:

- **`<word-size>`**: The word size of the system, in bits.
- **`<cache-capacity>`**: Total size of the cache, in bytes.
- **`<cache-ways>`**: Associativity of the cache, i.e. the number of ways per set.
- **`<cache-block-size>`**: Block size of the cache, in bytes.
- **`<page-size>`**: Page size of the virtual memory system, in bytes.
- **`<tag>`**: An individual tag of something kept in cache. It's -1 if invalid.
    - Order doesn't matter. If a cache set has values {2,3,4,-1} it contains tags 2, 3, and 4; this is equivalent to {2,-1,3,4}, {-1,4,2,3}, etc.
- **`<ppn>`**: A physical page number in the page table. Set to -1 for invalid.

Note: while all provided input files will have this structure, your input can simply be a sequence of fscanf() calls to consume integers; there is no reason to care about line separation or to use fgets().

These files can get large, so to make your debugging easier, we're also provided a "verbose" variant of each input file where every value is specifically labeled. **This is for your human use only; there is no need for your program to ever look at, parse, or care about verbose-style input files.**

Below is an example input file as well as the equivalent verbose file.

Input file	Verbose file
`8 16 2 2 32`	`word_size=8 cache_capacity=16 cache_assoc=2 cache_blk_size=2 page_size=32`
`-1    24`	`[set    0]      -1      24`
`4    -1`	`[set    1]       4      -1`
`-1     5`	`[set    2]      -1       5`
`21    -1`	`[set    3]      21      -1`
`1`	`[vpage    0] 1`
`6`	`[vpage    1] 6`
`-1`	`[vpage    2] -1`
`4`	`[vpage    3] 4`
`2`	`[vpage    4] 2`
`0`	`[vpage    5] 0`
`-1`	`[vpage    6] -1`
`3`	`[vpage    7] 3`
	`** This is a verbose version of this test input to help with debugging; your program will never be fed this file. **`

This file indicates:

- The word size is 8 bits (1 byte)
- The cache size in total is 16 bytes
- The cache is 2-way (i.e. a set-associative cache, each set can hold two things)
- The cache block size is 2 bytes
- The page size is 32 bytes
- The cache has the following tags:
    - Set 0: tag 24 (plus an invalid entry)
    - Set 1: tag 4 (plus an invalid entry)
    - Set 2: tag 5 (plus an invalid entry)
    - Set 3: tag 21 (plus an invalid entry)
- The page table has the following entries:
    - Virtual page 0 maps to physical page 1
    - Virtual page 1 maps to physical page 6
    - Virtual page 2 is invalid
    - Virtual page 3 maps to physical page 4
    - Virtual page 4 maps to physical page 2
    - Virtual page 5 maps to physical page 0
    - Virtual page 6 is invalid
    - Virtual page 7 maps to physical page 1

To facilitate testing, we have provided 8 test cases:

- **`mem8a.txt`** – A tiny 8-bit system with a direct-map cache
- **`mem8b.txt`** – A tiny 8-bit system with a 2-way set associative cache
- **`mem8c.txt`** – A tiny 8-bit system with a fully-associative cache
- **`mem16a.txt`** – A medium-sized 16-bit system with a 2-way cache
- **`mem16b.txt`** – A medium-sized 16-bit system with a 4-way cache
- **`mem16c.txt`** – A medium-sized 16-bit system with a smaller direct-map cache
- **`mem32a.txt`** – A big 32-bit system with a big cache with big pages
- **`mem32b.txt`** – A big 32-bit system with a reasonable cache and 4k pages

As mentioned previously, each file above has a corresponding "-verbose.txt" file with the same values but explicit labeling to ease debugging.

Files fed to your program will always meet the following rules:

- Files will always be of valid format and fully specify the cache and page table content
- Word size will always be 8, 16, or 32 bits
- Numbers will always be appropriate, e.g. no tag will never be larger than the maximum possible tag
- The system will always be possible and reasonable, e.g. we'll never have a cache bigger than memory.
- The cache and virtual memory parameters will always be such that the VPN field will never intersect with the cache index field (i.e. the system will always be compatible with the concept of a TLB).

Program output

For a memory access that's not a page fault, the output of your program should be as follows; grey comments have been added for clarity. All numbers printed should be decimal (base 10). This output should basically be computed in the order it's printed.

Output	Value	Comment	Group
Word size:	8	(First value of file)	System params
Cache capacity:	16	(Second value of file)	
Cache associativity:	1	(Third value of file)	
Cache block size:	2	(Fourth value of file	
Page size:	32	(Fifth value of file)	
Cache frames:	8	(Derived from params)	Cache/page stats
Cache sets:	8	(Derived from params)	
Virtual pages:	8	(Derived from params)	
Bits for cache block offset:	1	(Derived from above)	Address bit breakdowns
Bits for cache index:	3	(Derived from above)	
Bits for cache tag:	4	(Derived from above)	
Bits for page offset:	5	(Derived from above)	
Bits for page number:	3	(Derived from above)	
Virtual address:	0	(Given on command line)	From input
Address page offset:	0	(Computed from address)	Virtual to physical translation
Address virtual page number:	0	(Computed from address)	
Address physical page number:	6	(Looked up in page table)	
Physical address:	192	(Computed from above)	
Address cache block offset:	0	(Computed from phys addr)	Cache lookup
Address cache index:	0	(Computed from phys addr)	
Address cache tag:	12	(Computed from phys addr)	
Address hit/miss/fault:	miss	(Looked up in cache tags)	

In the event of a page fault, the lines shown in red above will not be printed (as there is no physical page to base them on). Instead, output will look like this:

```
Word size:                               8
Cache capacity:                          16
Cache associativity:                     1
Cache block size:                        2
Page size:                               32

Cache frames:                            8
Cache sets:                              8
Virtual pages:                           8

Bits for cache block offset:             1
Bits for cache index:                    3
Bits for cache tag:                      4
Bits for page offset:                    5
Bits for page number:                    3

Virtual address:                         64

Address page offset:                     0
Address virtual page number:             2
Address hit/miss/fault:                  fault
```

Spacing doesn't matter, but everything else does. To help, here's an example printf that produces a line of output above; note the use of the padded format specifier to provide nice spacing:

```
printf("%-40s %d\n","Word size:",word_size);
```

Restriction

In this assignment, **you may NOT use the modulus (%) operator**. You must use bitwise operations to determine the components of the address. Penalty: 50% off overall score.

Building and testing

You can simply build your program as per usual with g++:

```
g++ -g -o tlb tlb.c
```

It takes two arguments:

```
./tlb <memory-state-file.txt> <address>
```

The provided input files are in the **input** subdirectory, and you can specify any decimal address you wish ($\leq 2^{word_size}$), e.g.:

```
./tlb inputs/mem8a.txt 200
```

A suite of tests with expected results has been provided; these are in the **tests** subdirectory. For each test, we've provided the expected output. The tests themselves are listed in an appendix at the end of this document.

An automated testing tool, **hw5test.py**, has also been provided. It works much like the tool from homework 1.

```
Auto-tester for Duke CS/ECE 250, Homework 5, Summer 2017

Usage: hw5test.py [options] <suite>

Options:
  -h, --help     show this help message and exit
  -v, --verbose  Print extra info.

Where <suite> is one of:
  ALL             : Run all program tests
  CLEAN           : Remove all the test output produced by this tool
  tlb             : Run tests for tlb
```

You can run the tests as follows:

```
./hw5test.py tlb
```

These tests are not guaranteed to be exhaustive – you may wish to test additional addresses or even develop new machine files to find any corner-case bugs in your tool.

If you fail a test case, you can see your actual output as **tests/tlb_actual_##.txt**, and you can see the diff (a comparison between the actual vs. expected output) in **tests/tlb_diff_##.txt**.

Tips for success

- The input format is simple; don't overthink it. It's nothing but fscanf(): read the system parameters, use them to figure out the number of cache and page table entries, then read those.
- You should compute the outputs in the order they're printed. As a side effect, you can develop your program iteratively, adding additional output as you go until you have the full program.
- Copy/paste the field labels into the printf() code shown earlier to ensure your labels match perfectly so you pass the automated tests. Retyping them is needless, error-prone work.
- The cache can simply be a 2D array; the page table simply a 1D array.
- You do NOT have to worry about memory leaks on this assignment.
- Bit manipulation tips:
    - You can get a bit string of N ones with the expression: **((1<<N)-1)**

- Here's a simple implementation of base-2 log using only integer math, that way you don't have to mess with the math library:

```
int log2(int n) {
      int r=0;
      while (n>>=1) r++;
      return r;
}
```

- With regard to the "verbose" files: the lines numbers match between the two, so you can scan through the verbose file to find something, then see the corresponding value on the same line of the actual input file. This is their primary purpose.
- You can represent every number in this assignment as a 32-bit integer, but potentially large values (such as addresses) must be unsigned; you can printf unsigned decimal values using the "`%u`" specifier.
- All integer inputs and outputs are expressed as decimal numbers.
- Have fun *!!*

Appendix A: Provided test cases

#	input file	hit/miss/fault?	expected output
00	mem8a.txt	miss	tests/tlb_expected_00.txt
01	mem8a.txt	miss	tests/tlb_expected_01.txt
02	mem8a.txt	hit	tests/tlb_expected_02.txt
03	mem8a.txt	hit	tests/tlb_expected_03.txt
04	mem8a.txt	hit	tests/tlb_expected_04.txt
05	mem8a.txt	hit	tests/tlb_expected_05.txt
06	mem8a.txt	miss	tests/tlb_expected_06.txt
07	mem8a.txt	miss	tests/tlb_expected_07.txt
08	mem8a.txt	fault	tests/tlb_expected_08.txt
09	mem8a.txt	fault	tests/tlb_expected_09.txt
10	mem8a.txt	miss	tests/tlb_expected_10.txt
11	mem8a.txt	fault	tests/tlb_expected_11.txt
12	mem8a.txt	miss	tests/tlb_expected_12.txt
13	mem8b.txt	miss	tests/tlb_expected_13.txt
14	mem8b.txt	miss	tests/tlb_expected_14.txt
15	mem8b.txt	hit	tests/tlb_expected_15.txt
16	mem8b.txt	hit	tests/tlb_expected_16.txt
17	mem8b.txt	hit	tests/tlb_expected_17.txt
18	mem8b.txt	hit	tests/tlb_expected_18.txt
19	mem8b.txt	miss	tests/tlb_expected_19.txt
20	mem8b.txt	miss	tests/tlb_expected_20.txt
21	mem8b.txt	fault	tests/tlb_expected_21.txt

22	mem8b.txt	fault	tests/tlb_expected_22.txt
23	mem8b.txt	miss	tests/tlb_expected_23.txt
24	mem8b.txt	fault	tests/tlb_expected_24.txt
25	mem8b.txt	miss	tests/tlb_expected_25.txt
26	mem8c.txt	miss	tests/tlb_expected_26.txt
27	mem8c.txt	miss	tests/tlb_expected_27.txt
28	mem8c.txt	hit	tests/tlb_expected_28.txt
29	mem8c.txt	hit	tests/tlb_expected_29.txt
30	mem8c.txt	hit	tests/tlb_expected_30.txt
31	mem8c.txt	hit	tests/tlb_expected_31.txt
32	mem8c.txt	miss	tests/tlb_expected_32.txt
33	mem8c.txt	miss	tests/tlb_expected_33.txt
34	mem8c.txt	fault	tests/tlb_expected_34.txt
35	mem8c.txt	fault	tests/tlb_expected_35.txt
36	mem8c.txt	miss	tests/tlb_expected_36.txt
37	mem8c.txt	fault	tests/tlb_expected_37.txt
38	mem8c.txt	miss	tests/tlb_expected_38.txt
39	mem16a.txt	miss	tests/tlb_expected_39.txt
40	mem16a.txt	miss	tests/tlb_expected_40.txt
41	mem16a.txt	fault	tests/tlb_expected_41.txt
42	mem16a.txt	hit	tests/tlb_expected_42.txt
43	mem16a.txt	hit	tests/tlb_expected_43.txt
44	mem16a.txt	fault	tests/tlb_expected_44.txt
45	mem16a.txt	miss	tests/tlb_expected_45.txt
46	mem16a.txt	hit	tests/tlb_expected_46.txt
47	mem16a.txt	miss	tests/tlb_expected_47.txt
48	mem16a.txt	hit	tests/tlb_expected_48.txt
49	mem16b.txt	fault	tests/tlb_expected_49.txt
50	mem16b.txt	miss	tests/tlb_expected_50.txt
51	mem16b.txt	hit	tests/tlb_expected_51.txt
52	mem16b.txt	fault	tests/tlb_expected_52.txt
53	mem16b.txt	hit	tests/tlb_expected_53.txt
54	mem16b.txt	miss	tests/tlb_expected_54.txt
55	mem16b.txt	hit	tests/tlb_expected_55.txt
56	mem16b.txt	fault	tests/tlb_expected_56.txt
57	mem16c.txt	fault	tests/tlb_expected_57.txt
58	mem16c.txt	miss	tests/tlb_expected_58.txt
59	mem16c.txt	miss	tests/tlb_expected_59.txt
60	mem16c.txt	hit	tests/tlb_expected_60.txt
61	mem16c.txt	miss	tests/tlb_expected_61.txt

62	mem16c.txt	hit	tests/tlb_expected_62.txt
63	mem16c.txt	miss	tests/tlb_expected_63.txt
64	mem16c.txt	miss	tests/tlb_expected_64.txt
65	mem16c.txt	fault	tests/tlb_expected_65.txt
66	mem32a.txt	miss	tests/tlb_expected_66.txt
67	mem32a.txt	miss	tests/tlb_expected_67.txt
68	mem32a.txt	miss	tests/tlb_expected_68.txt
69	mem32a.txt	miss	tests/tlb_expected_69.txt
70	mem32a.txt	miss	tests/tlb_expected_70.txt
71	mem32a.txt	miss	tests/tlb_expected_71.txt
72	mem32a.txt	miss	tests/tlb_expected_72.txt
73	mem32a.txt	fault	tests/tlb_expected_73.txt
74	mem32a.txt	miss	tests/tlb_expected_74.txt
75	mem32a.txt	fault	tests/tlb_expected_75.txt
76	mem32a.txt	hit	tests/tlb_expected_76.txt
77	mem32a.txt	hit	tests/tlb_expected_77.txt
78	mem32b.txt	fault	tests/tlb_expected_78.txt
79	mem32b.txt	fault	tests/tlb_expected_79.txt
80	mem32b.txt	fault	tests/tlb_expected_80.txt
81	mem32b.txt	fault	tests/tlb_expected_81.txt
82	mem32b.txt	miss	tests/tlb_expected_82.txt
83	mem32b.txt	fault	tests/tlb_expected_83.txt
84	mem32b.txt	miss	tests/tlb_expected_84.txt
85	mem32b.txt	miss	tests/tlb_expected_85.txt
86	mem32b.txt	miss	tests/tlb_expected_86.txt
87	mem32b.txt	miss	tests/tlb_expected_87.txt
88	mem32b.txt	hit	tests/tlb_expected_88.txt
89	mem32b.txt	hit	tests/tlb_expected_89.txt