ENCM 369 Winter 2018 Lab 10 for the Week of March 26

Steve Norman
Department of Electrical & Computer Engineering
University of Calgary

March 2018

Lab instructions and other documents for ENCM 369 can be found at
http://people.ucalgary.ca/~norman/encm369winter2018/

Administrative details

You may work in pairs on this assignment

You may complete this assignment individually or with *one* partner.

Students working in pairs must make sure both partners understand *all* of the exercises being handed in. The point is to help each other learn *all* of the lab material, not to allow each partner to learn only half of it! Please keep in mind that you will not be able to rely on a partner to do work for you on midterm #2 or the final exam.

Two students working together should hand in a *single assignment* with names and lab section numbers for both students on the cover page. Names should be complete and spelled correctly. If you as an individual are making the cover page, please get the information you need from your partner. For partners who are not both in the same lab section, please hand in the assignment to the collection box for the student whose last name comes first in alphabetical order.

Due Date

Note that the due date is affected by the Good Friday holiday March 30, 2018.

The Due Date for this assignment is 3:30pm Monday, April 2, 2018.
The Late Due Date is 3:30pm Tuesday, April 3, 2018.

The penalty for handing in an assignment after the Due Date but before the Late Due Date is 3 marks. In other words, X/Y becomes (X–3)/Y if the assignment is late. There will be no credit for assignments turned in after the Late Due Date; they will be returned unmarked.

How to package and hand in your assignments

Please see the Lab 1 instructions.

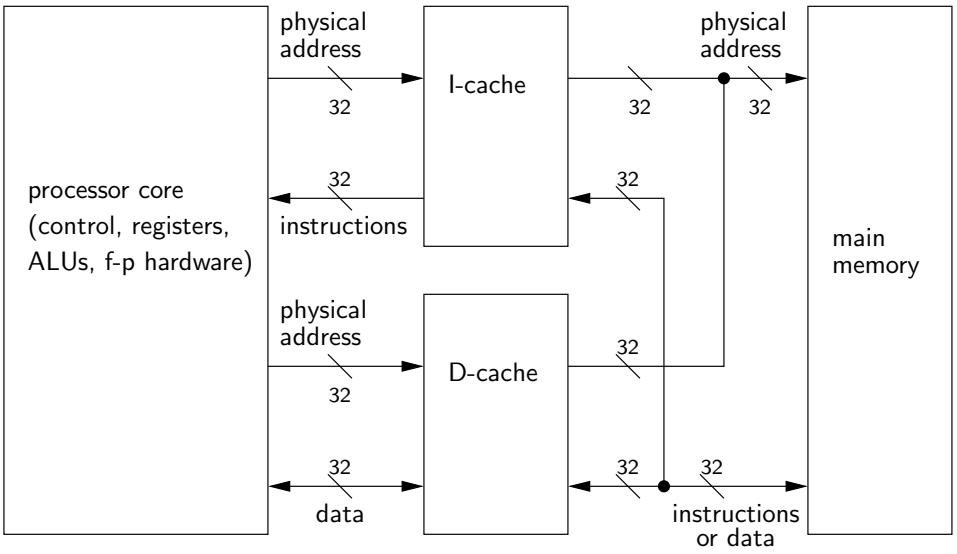
Exercise A: Tracing behaviour of an I-cache

Read This First

In learning about caches, it is useful to trace through all of the copying of bit patterns that occurs in a sequence of memory accesses.

What to Do

Figure 1: Computer with one level of cache, and no address translation.



The table in Figure 3 shows some of the main memory contents for a program running on a computer like the one shown in Figure 1, running the MIPS instruction set (except that this computer does *not* have delayed jumps and branches). Note that the first sequence of instructions is a complete procedure, but the second sequence is only part of a procedure. The second sequence includes a loop; within that loop there is a call to the procedure comprised of the first sequence of instructions. You will be asked to trace the interaction between the I-cache and the main memory starting with PC = `0x0040_27b8`, at the moment in time just before the `beq` instruction is fetched.

The I-cache for this computer is direct-mapped with 1024 sets, and the block in each set contains one instruction. This structure exactly matches an example that has been presented in a lecture. For this cache, main memory addresses are split as shown in Figure 2.

At the moment in time mentioned above, the state of sets 494–499 of the cache is shown in Figure 4. Use `0x1000_1038` as the initial value for `$s2` and `0x1000_1040` as the initial value for `$s3`. Trace all the instruction fetches until after the instruction at address `0x0040_27cc` has been fetched. Record your answer in tabular form, using this trace of the first two instructions as a model:

address	tag	set	action
`0x0040_27b8`	`0x00402`	494	I-cache hit—no I-cache update
`0x0040_27bc`	`0x00402`	495	I-cache miss—instruction `0x0240_2021` is copied into instruction field in set 495, V-bit in that set is changed to 1, tag to `0x00402`

Hints: (1) Including the two instruction fetches given as examples, there will be a total of 18 instruction fetches. (2) There is a useful C program in `encm369w18lab10/exA`.

Figure 2: How instruction addresses are split for access to the cache of Exercise A.

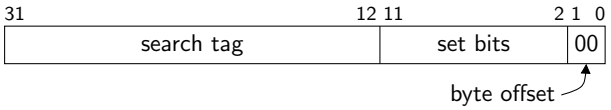


Figure 3: Small fragments of main memory contents for Exercise A.

address	instruction at address	disassembly of instruction
0x0040_17c0	0x8c99_0000	P1: lw $t9, ($a0)
0x0040_17c4	0x0019_c023	subu $t8, $zero, $t9
0x0040_17c8	0xac98_0000	sw $t8, ($a0)
0x0040_17cc	0x03e0_0008	jr $ra
⋮	⋮	⋮
0x0040_27b8	0x1253_0004	beq $s2, $s3, L2
0x0040_27bc	0x0240_2021	L1: addu $a0, $s2, $zero
0x0040_27c0	0x0c10_0f50	jal P1
0x0040_27c4	0x2652_0004	addiu $s2, $s2, 4
0x0040_27c8	0x1653_fffc	bne $s2, $s3, L1
0x0040_27cc	0x2414_0000	L2: addiu $s4, $zero, 0

Figure 4: Part of the initial state of the I-cache for Exercise A. (Only sets 494–499 in the cache are shown.)

set	valid	tag	instruction
494	1	0x00402	0x1253_0004
495	0	0x00000	0x0000_0000
496	1	0x00401	0x8c99_0000
497	1	0x00401	0x0019_c023
498	1	0x00401	0xac98_0000
499	1	0x00401	0x03e0_0008

What to Hand In

Hand in your completed table.

Exercise B: Analysis of direct-mapped caches

Read This First

The base-two logarithm

The *base-two logarithm* is a simple and useful concept that has many useful applications in computer systems, one of which is describing dimensions in caches. The $\log_2$ function is simply the inverse of the function $f(x) = 2^x$, in the same way that the ln function is the inverse of $f(x) = e^x$ and the $\log_{10}$ function (often written as simply log) is the inverse of $f(x) = 10^x$. For example,

$$\log_2 1 = 0,\ \log_2 4 = 2,\ \log_2 32 = 5,\ \text{and } \log_2 65536 = 16.$$

Dimensions within direct-mapped caches

Direct-mapped lookup is the simplest practical way to organize a data or instruction cache. It is illustrated in Figures 8.7 and 8.12 in the course textbook. The general structure of all direct-mapped caches is shown in Figure 5.

To search for an instruction word or data word in a direct-mapped cache, the main memory address of that word is broken into three or four parts:

- *tag:* used to distinguish a block address among a group of block addresses that all generate the same set bits;
- *set bits:* used to select one set among the S sets in the cache;
- *block offset:* used to select a single word within a multi-word block, so not necessary if the cache has one-word blocks;
- *byte offset:* can be assumed to be zero when the cache access is to an entire data or instruction word, but would be important for access to indivdual bytes in instructions such as MIPS `lb`, `lbu`, and `sb`.

How wide are each of these fields? Let's go right-to-left within an address. First,

$$\text{byte offset width} = \log_2(\text{number of bytes per word}).$$

So with 4-byte words, as in textbook and lecture examples, the byte offset width is $\log_2 4 = 2$, but with 8-byte words, the byte offset width would be $\log_2 8 = 3$.

Next,

$$\text{block offset width} = \log_2(\text{number of words per block}).$$

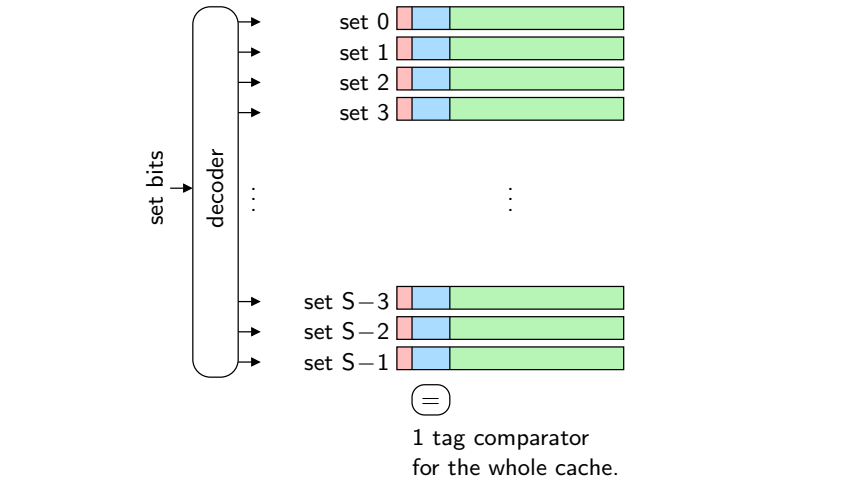
For example, in textbook Figure 8.12, there are 4 words per block, so the block offset is $\log_2 4 = 4$ bits wide. Note that that gives you four different bit patterns to choose one of the four words within a block: 00, 01, 10, 11. Note also that $\log_2 1 = 0$, consistent with the idea that if the block size is one word, no bits from the address should be used as a block offset—that is what you see in textbook Figure 8.7.

Moving on, let's let S stand for the number of sets within the cache. Then

$$\text{number of set bits} = \log_2 S.$$

I hope you can see a general pattern here: If X is a power of two, then $\log_2 X$ bits are needed to select one of X things.

Figure 5: General organization of a direct-mapped cache. Wiring and some logic components have been left out to reduce clutter.



Finally, the tag is everything in an address that hasn't already been used:

tag width =
address width − number of set bits − block offset width − byte offset width

The *capacity* of a cache is usually defined as the maximum number of bytes of data or instructions that a cache can hold. Note that that definition *excludes* storage of status bits and tag bits. Let's let Bpl ("bytes per line") stand for the size of a block. (*Cache line* is a synonym for *cache block*, and unlike "block", "line" does not confusingly start with the letter b.) From Figure 5 it should be clear that for a direct-mapped cache

$$C = S \times \text{Bpl},$$

and if we want to think of block size measured in words instead of bytes,

$$C = S \times \text{words per block} \times \text{bytes per word}.$$

Example calculations

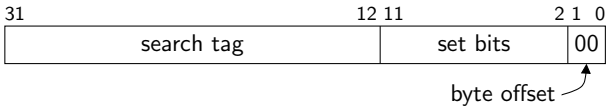
1. *Suppose it has been decided that a direct-mapped cache should have 1024 entries, each with one 32-bit data word. How should addresses be split into parts? And what will the capacity of this cache be?*

   Byte offset: A 32-bit word is 4 bytes, so the width is $\log_2 4 = 2$.

   Block offset: There is no block offset, because there is only one word per block.

   Set bits: $S = 1024$, so we need $\log_2 1024 = 10$ set bits.

Tag: The tag is all the bits to the left of the set bits. So addresses should be split this way:



Capacity is

$$\begin{aligned} C &= 1024\,\text{blocks} \times 1\,\frac{\text{word}}{\text{block}} \times 4\,\frac{\text{bytes}}{\text{word}} \\ &= 4096\,\text{bytes} \\ &= 4\,\text{KB}. \end{aligned}$$

Remark: This has been a review of the organization of the cache used in Exercise A of this lab.

2. *Suppose it has been decided to build a direct-mapped cache with a (very tiny) capacity of 32 bytes, in which each block holds 4 32-bit words. What is S, the number of sets? And how should addresses be split into parts?*

   To find S, solve for it in this equation:

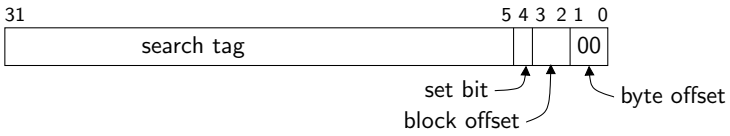
$$32\,\text{bytes} = 2^5\,\text{bytes} = S \times 2^2\,\frac{\text{words}}{\text{block}} \times 2^2\,\frac{\text{bytes}}{\text{word}}$$

   That gives

$$S = \frac{2^5\,\text{bytes}}{2^2\,\frac{\text{words}}{\text{block}} \times 2^2\,\frac{\text{bytes}}{\text{word}}} = 2^{5-2-2}\,\text{blocks} = 2^1\,\text{blocks} = 2\,\text{blocks}.$$

   (The equation gives S as a number of blocks rather than a number of sets, but that's fine, because in a direct-mapped cache there is one block per set.)

   The address split is: byte offset, $\log_2 4 = 2$ bits; block offset, $\log_2 4 = 2$ bits; set bits, $\log_2 2 = 1$ bit; tag, $32 - 1 - 2 - 2 = 27$ bits. Graphically, that's:



   Remark: This example has been a review of the organization of the cache in textbook Figure 8.12.

What to Do

Write well-organized solutions to the following problems.

3. Suppose the specification for the cache of textbook Figure 8.12 is changed. The block size is now supposed to be 8 words, and the capacity has been increased to a much more practical size of 32 KB.

   (a) What is S, the number of sets?

   (b) How should addresses be split into parts? Show this with a diagram like the previously given examples—include bit numbers marking the boundaries between parts.

(c) The cache will be built using SRAM cells, one SRAM cell for every V-bit, every bit within a tag, and every bit within a block of data or instruction words. How many SRAM cells are needed for the whole cache? Show your work carefully.

4. The term "64-bit processsor" usually describes a processor in which general-purpose registers are 64 bits wide, and memory addresses *within the processor core and in pointer variables* are managed as 64-bit patterns. However, 2^{64} bytes of DRAM is an enormously larger quantity of storage than can practically be connected to a single processor chip, so a typical 64-bit design might use only the least significant 44 bits of an address to access caches and main memory. This is illustrated in Figure 6.

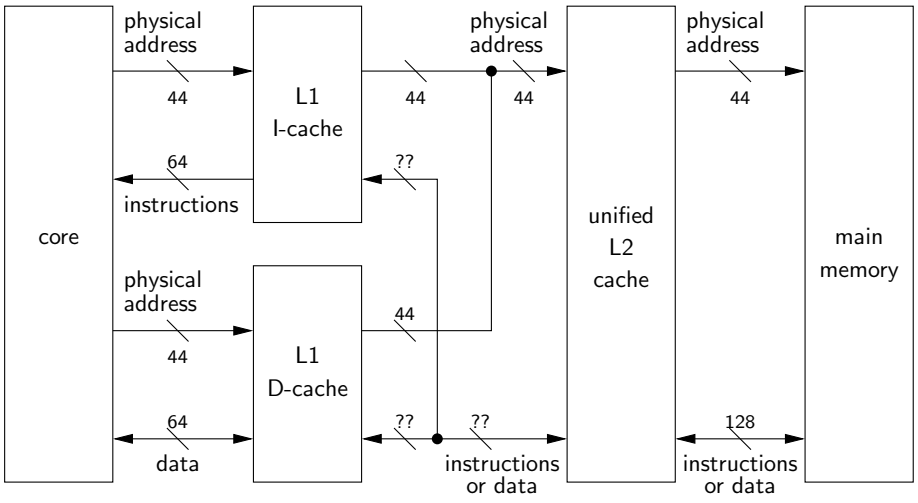
   Do the following calculations for the computer of Figure 6. We'll assume direct-mapped design for all three caches, and we'll consider the word size to be *64 bits*.

   (a) The capacity of the L1 D-cache is 64 KB. The block size is 64 *bytes*. Draw a diagram to show how a 44-bit address input to the cache would be split into these fields: tag, set bits, block offset, and byte offset. Indicate exactly how wide each field is.

   (b) Repeat part (a) for the L2 cache, which has a capacity of 2 MB. The block size is again 64 *bytes*.

   (c) Use your results from (b) to determine how many one-bit SRAM cells are needed for all the V-bits, tags, and data/instruction blocks in the L2 cache.

Figure 6: Simplifed view of memory organization of a recent single-core 64-bit system. (The major simplification here is the omission of address translation.) The core can fetch two 32-bit instructions at once from from the Level 1 I-cache. The width of the data/instruction bus between the Level 1 and Level 2 caches is unspecified but should be much wider than 64 bits to support speedy transfers of large blocks.



Exercise C: Simulating a Cache

Read This First

In this exercise you will work with a C program to simulate some aspects of the behaviour of data caches. For two different sequences of memory accesses you will determine which accesses are cache hits and which accesses are cache misses. (This is relatively easy to do; a complete simulation that modeled write-through or write-back to main memory would be a much more complex problem.)

The sequences of memory accesses were generated by determining the data memory accesses that would be made in sorting an array of 3000 integers using two different sort algorithms on a machine similar to a 32-bit MIPS computer. Memory is stored in 32-bit words, and byte addresses are 32-bits in length. All accesses to data memory by the sort procedures are loads and stores of *words*, at addresses that are multiples of four.

These sequences are available in text files. Here are the first ten lines of one of the files:

```
r 804e6ac
r 804fe1c
w 804e6ac
r 804e6a8
r 804fe14
r 804fe18
w 804e6a8
w 804fe18
r 804e6a4
r 804fe0c
```

`r` means read (load) and `w` means write (store). The addresses are in hexadecimal notation, even though there is no leading `0x`.

Note that the actual data values are not included in the file, just the addresses used to access data. It turns out that to count cache hits and misses, the sequence of addresses used is all that really matters.

What to Do, Part I

Download the files from `encm369w18lab10/exC`. Note that two of the files in the directory are large—if you are close to using up your disk quota (or if the file server is having a bad day) your copy command might fail. To check that the copy was successful, use the Cygwin `ls -l` command (lower-case L, not number 1), to check the sizes of the large files—they should be

`heapsort_trace.txt`	1251240 bytes
`mergesort_trace.txt`	1824492 bytes

(The files might get a little bigger if the Web browser downloading them converts them to the Microsoft Windows text file format.)

Read the C source file `sim1.c` carefully. The program does a simulation of a data cache with 1024 words in 1-word blocks. Build an executable and run it with both data files, following the instructions in a comment near the top of the source file.

Here is some information about two memory access traces:

- In `heapsort_trace.txt` all data memory accesses are to the array elements. Each of the 3000 elements is read at least once, and is likely to be read and written many more times.

- In `mergesort_trace.txt` the data memory accesses are to all of the array elements, to a 1500-word array of temporary storage needed by the mergesort algorithm, and to 72 words of stack used to manage a sequence of recursive function calls. So the total number of different words accessed is 4572.

In both cases the 1024-word cache is much too small to hold all of the different data memory words being accessed. If you naïvely guess that access to memory words is truly random, you would expect very high miss rates, such as $(3000 - 1024)/3000 = 65.9\%$ or worse for the heapsort run, and $(4572 - 1024)/4572 = 77.6\%$ or worse for the mergesort run. However, you should see much lower miss rates due to *locality of reference* in the memory accesses.

What to Do, Part II

Let's examine the effect of changing the block size of the cache of Part I, while maintaining capacity.

Make a copy of `sim1.c` called `sim2.c`, and edit it to simulate a direct-mapped cache with 256 four-word blocks. You will not have to edit many lines of the C code, but to do it correctly you will have to do some calculations like the ones in Exercise B, then think carefully about how to isolate the correct set and tag bits.

Run the program using the two given input files. Copy and paste records of your programs runs into a file and print that file along with your source file `sim2.c`.

Also, answer this question:

> Compare results obtained in this part with results from Part I. Do they suggest that there is significant *spatial* locality of reference in the memory accesses done by the heapsort and mergesort algorithms? Give a brief explanation.

What to Do, Part III

Now let's consider a direct-mapped cache with the same 4 KB capacity as in Part II, but with the block size quadrupled to sixteen words.

Make another copy of `sim1.c`; call this one `sim3.c`. Edit it to simulate a direct-mapped cache with the appropriate number of sixteen-word blocks.

Run the program using the two given input files. Copy and paste records of your programs runs into a file and print that file along with your source file `sim3.c`.

Final Note

Cache-friendliness (a tendency towards low miss rates) is an important factor in performance of algorithms on modern computer hardware.

But don't use this exercise to draw any firm conclusions about which of the two sort algorithms is more cache-friendly. (My own suspicion is that mergesort may often be significantly more cache-friendly than heapsort.) The caches being simulated are unrealistically small, and using one run of each algorithm for a single array hardly constitutes enough input data for a good experiment.

Researchers doing simulation experiments to compare cache designs will use traces of trillions of memory accesses from many different programs in order to ensure that performance is being measured for many different patterns of memory access.

What to Hand In

Hand in the printouts from Parts II and III, and your answer to the question in Part II. Please label the printouts clearly so your TA's don't have to guess which printout is from which part.