Programming Assignment #2
Simulation of a Hardware Cache

CS 3220 / CS 5220 Spring 2025
40 points
Part I: due Saturday, Feb. 22nd (20 points)
Part II: due Saturday, Mar. 1st (20 points)

1 Simulating a Hardware Memory Cache

You'll write a program to simulate the behavior of a hardware memory cache. You can use the language of your choice (hopefully, Python). You may work either individually or with a partner.

If you work with a partner, tell me this in your code and make a note in Brightspace when you submit.

1.1 High-level summary

Here's the basic algorithm for a cache simulator:

```
while not done
  get a memory address A
  // this will be either of the form write(A, val) or val = read(A)
  // and the address could come from a file or from a test function
  extract the block offset, the index, and the tag from A
  calculate the target block in the cache
  check to see whether the block containing A is currently in the cache
  if the block containing A is in the cache
    // this is a cache hit
    if this is a read
      read the contents of the cache from the target block at the offset for A
    else
      write val to the target block at the offset for A
  else
    // this is a cache miss
    read the block containing A from memory
    write that block to the target block in the cache
    then either read or write, depending on whether this is a read or a write
```

You'll write two functions: the first is to read a word from address A; and the second is to write a word to address A. Each of these functions first looks in the cache. If the word at A is in the cache, then the read function will read the value (a single word); and the write function will write the value (again, a single word).

If the word at A is not in the cache, then you'll read a cache block from memory into the cache and update the cache data structures.

Remember: data is read from memory to the cache a whole block at a time.

A word is four bytes.

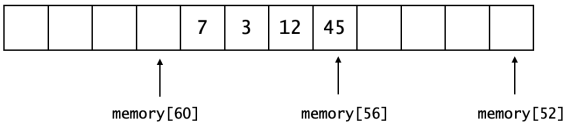
1.2 Memory hierarchy

Model the memory as an array of bytes. Each memory access will read or write one word (four bytes). Check that each address is aligned on a four-byte boundary, and assert (or exit) if it isn't. Also check that each memory access is in range, and assert (or exit) if it isn't.

So for example, if the size of the memory is `MEMSIZE` bytes, then every address must be in the range 0 to `MEMSIZE-1`, and every address must be an even multiple of four.

You'll model a little-endian system, with 32-bit words: so for example if `memory[56] = 45` and `memory[57] = 12` and `memory[58] = 3` and `memory[59] = 7`, then the word referenced by address 56 would be:

$$val = 45 + 12 \times 256 + 3 \times 256^2 + 7 \times 256^3.$$



When a range of values is loaded into the cache, the number of bytes loaded will thus be a multiple of four.

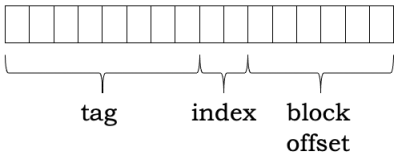
1.3 Cache structure

Recall how the index, tag, and block offset are computed from an address. For example, suppose we have the following:

- a 64K memory (hence 16-bit addresses)
- a 1K cache (1K is $2^{10} = 1024$)
- 64-byte cache blocks
- four-way associativity

Then:

- there are $2^{10}/2^6 = 16$ cache blocks (since $1024 = 2^{10}$ and $64 = 2^6$).
- and there are $16/4 = 4$ cache sets (because the cache is four-way set associative)
- so we need two bits for set selection (the index; two bits let us specify the value 0, 1, 2, or 3)
- and six bits for byte selection in a block (six bits lets us specify a value in the range 0 to 63)
- which means that the tag is $16 - 2 - 6 = 8$ bits in length



1.4 Parameters

Your simulator should let you specify the following parameters, as variables (in other words, do not hardcode these values in your program):

- the size of a memory address, in bits (and then set the size of the memory to 2 to the power #bits)
- the size of the cache, in bytes (must be a power of 2)
- this size of a cache block, in bytes (must be a power of 2)
- the associativity of the cache (must be a power of 2)
- whether the cache is a write-back or write-through cache (this will be for Part Two)

Again, do not hardcode any of these values. Use a variable or a `#define` to represent each of them.

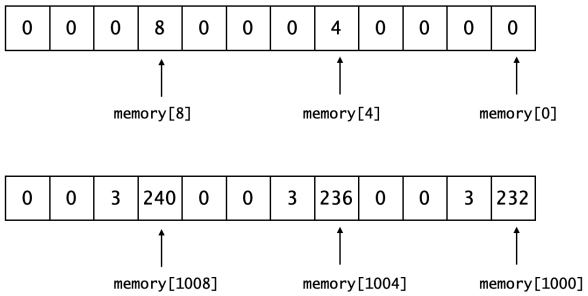
2 Part One

Implement a direct-mapped cache that supports only reads.

2.1 Initialization

Initialize the memory so that you'll know you're reading the correct values from memory (and from the cache). To do this, set four-byte words of the memory to the values $0, 1, 2, ..., m$, where $m = memSize/4 - 1$.

This diagram shows two regions of the memory and how they're initialized:



Note that $1000 = 3 \times 256 + 232$; $1004 = 3 \times 256 + 236$, etc.

By initializing the memory in this way, the value I obtain when I read from address A will be A.

2.2 Data structures

You're free to model the cache however you'd like, but I think it makes sense to consider the fundamental component of a cache to be a cache set, consisting of one or more cache blocks. Each set will then have k blocks, where k is the associativity. So, for example, a two-way associative cache of size 64K with 64-byte blocks will have 1024 blocks and 512 sets. A direct-mapped cache will have $k = 1$.

So, you can model your cache as a collection of cache sets (for example, as an array or list of sets). You might instead choose to model the cache as a collection of cache blocks, with some efficient way to mark set membership for each block.

Each cache block will need a tag (an integer) and the actual data of the block (an array/list of bytes). In Python, use a `bytearray`; in C use a `char[]`. Initialize the tag of each block to -1.

For Part One, each set will have a single block ($k = 1$).

The memory itself is just an array of bytes. It can be a global variable. In Python, use a `bytearray`; in C use a `char[]`.

2.3 Example cache

Suppose I want to simulate a direct-mapped cache in a system having these attributes:

- 64 K memory, with 16-bit addresses
- 1024-byte cache
- 64-byte cache blocks

This means the cache contains $1024/64 = 16$ cache blocks. Six bits are required for the block offset, and four bits for the index. This leaves $16 - 4 - 6 = 6$ bits for the tag.

Consider the address 46916 = 101101 1101 000100. The block offset is 4, the index is 13, and the tag is 45. A read from this memory location will correspond to the block at index = 13, starting at position 4 in the block. (This is a direct-mapped cache, so each set has only a single block.)

Another example: the address 13388 corresponds to 001101 0001 001100, so the block offset is 12, the index is 1, and the tag is 13.

2.4 Getting started

First, write a function that will take an address A and calculate the tag, the index, and the block offset values. In Python, the function would have this form:

```
def decode_address(A):
  # do the calculations
  return [tag, index, block_offset]
```

During decoding, print the block offset, index, and tag for each address, and make sure that you are calculating these values correctly.

Then, write a function to read from memory:

```
def read_word(A):
  # first look in the cache; if not found, then load the target block into the cache
  return word
```

See the description of the `read_word()` function in the accompanying slides.

2.5 Testing

Print information during each read: the address; the tag, index, and block offset; whether it's a read hit or read miss; the range of addresses in the target block in the cache; the range of memory addresses that will be read in during a read miss.

For the part-one code (direct-mapped cache, only reads), use this sequence of reads:

$0, 0, 60, 64, 1000, 1028, 12920, 12924, 12928$

You'll see my output from this sequence in `part-one-test.out`, in the `CacheSim` folder of the course repo.

3 Part Two

Handle read and write misses, associative caches, the tag queue, and write-through/write-back behavior.

3.1 Data structures

Now, each cache block will need two additional attributes: a dirty/clean flag and a valid/invalid flag. (OK to use a boolean variable for each.)

Initially, mark each cache block as invalid. When you load a block from memory into the cache, mark that block as valid and clean. When you write to particular block in the cache, mark that block as dirty.

Each cache set will have a tag queue. Initialize the tag queue for each set to invalid values, such as -1. (Zero is a valid tag, so you must not initialize the tag-queue entries to zeroes.) The size of the tag queue for each set is the associativity. So for example, each tag queue for a four-way set-associative cache will be an array (or list) of length four.

Initialize the tag for each block to -1, for the same reason.

3.2 Block replacement

Use the least-recently-used (LRU) algorithm to control block replacement in a set: if you need to replace one of the blocks in a set, then pick the block that was least recently used. The tag queue will show which block in a set should be replaced, because you'll keep the tags in LRU order in the tag queue.

Here's an example of how the queue will work. Suppose we have a four-way associative cache: then the tag queue for each set will have four positions. Now suppose the queue for a particular set is currently [4, 8, 12, 16], and that the most recently accessed tag is always kept in the last position.

- then, after an access having tag=4, the queue will be [8, 12, 16, 4]
- and after an access having tag=12, the queue will be [8, 16, 4, 12]
- and after an access having tag=4, the queue will be [8, 16, 12, 4]
- and after an access having tag=6, the queue will be [16, 12, 4, 6]
- and after another access having tag=4, the queue will be [16, 12, 6, 4]

In this way, the least-recently accessed tag is always in the first position. When you need to replace one of the blocks in this set, you'll replace the block having the tag that is in the first position.

(Another way of thinking about this: the tag queue is a priority queue of fixed size in which the priority is the access time; and later access time means higher priority.)

At a high level, your cache consists of a group of arrays/lists representing the cache blocks. Each of these arrays will have auxiliary information with it (a tag, the dirty/clean bit and the valid/invalid bit, and some way to show which set this array belongs to). A read from memory thus consists of copying a range of values from your memory array to one of the arrays representing a cache block. A write to memory consists of copying a value to one of these arrays. The setting for write-through vs. write-back will determine when and if you also copy data from one of the cache-block arrays back to your memory array if the target block in the cache is dirty.

All reads or writes will be for a single word (four bytes). A write-through cache will write a single word to the memory (in addition to writing the word to the cache).

3.3 Functions

Create two functions: one to read a word from memory, and one to write a word to memory.

Most of the actions of reading and writing are the same, so it makes sense to consolidate the actions into a single function `access_memory()`. This is essentially your `read_word()` from Part One, with additional logic.

For example: in Python, my functions would look like this:

```
read_word(address)
write_word(address, word)
```

where `read_word()` returns the value read from the specified address and `write_word()` writes the provided word to the specified address.

And each of these will then call `access_memory()`:

```
def read_word(address):
  return access_memory(address, None, AccessType.READ)

def write_word(address, word):
  access_memory(address, word, AccessType.WRITE)
```

Again, check each address for four-bit alignment and for range $0 \leq address < memSize$, where $memSize$ is the number of bytes in the memory.

In C, I might do this:

```
word readWord(unsigned int address);
void writeWord(unsigned int address, word w);
```

`word` is a typedef to an `int`.

The `access_memory()` function first looks in the cache. If the desired word is found in the cache, then it is returned (for a read) or modified (for a write). Otherwise, the cache block containing the desired word is brought into memory, and then the read or write occurs.

3.4 Output

In order to observe the behavior of the simulated cache, print output describing what happens in response to a read or write. For example, with a 64 K memory (16-bit addresses), a 1K cache, 64-byte blocks, and associativity = 1 (a direct-mapped cache), then in response to a read from address 56132, your functions should print out a string in this form (details will depend on whether this is a hit or miss, and whether an eviction is necessary):

```
read miss + replace [addr=56132 index=13 tag=54: word=56132 (56128 - 56191)]
```

If there is a read or write miss with a replacement necessary, then print out which tag, in which block index was evicted (the block index is the index of a block in its set):

```
[evict tag 4, in blockIndex 0]
```

And after each read or write, print the tag queue for the set that was accessed, in this format:

```
[ 12 20 32 54 ]
```

Check: 56132 = 110110 1101 000100, so the block offset is 000100 = 4; four bits are needed for the index (1024 / 64 = 16), giving the index 1101; and the tag is 110110 = 54.

Here's another example of the information you should print:

```
read miss + replace [addr=17536 index=2 tag=68: word=17536 (17536 - 17599)]
[evict tag 32, in blockIndex 1]
[write back (8320 - 8383)]
[ 36 44 64 68 ]
```

Print the eviction information only for a read or write miss + replace, and print the write-back info only for a write-back cache (and only if the evicted cache block is dirty).

Here's an example of the information to print for a write:

```
write miss + replace [addr=8320 index=2 tag=32: word=7 (8320 - 8383)]
[evict tag 28, in blockIndex 1]
[write back (7296 - 7359)]
[ 16 12 20 32 ]
```

And again, print the eviction information only for a miss + replace, and print the write-back info only for a write-back cache (and only if the evicted cache block is dirty).

3.5 Grad students

Graduate students —as you design and implement your cache, keep the following in mind: I will be asking you to run your cache simulator using real x86 trace files, in which the addresses are in the range zero to 2^{48}. You won't actually be reading from or writing to simulated memory, so you should have a flag such as `use_memory`, which you can set to true (for the main part of the assignment) or false (for the grad portion). I'll also ask you to report stats: the total number of reads and writes; the number of read hits/misses and the number of write hits/misses. And you should have a flag to suppress debug output.

4 Testing

In the class gitlab site, you'll find these files:

- `part-two-test-one-wb.out`
- `part-two-test-one-wt.out`
- `part-two-test-two-wb.out`
- `part-two-test-two-wb.out`
- `part-two-test-three-wb.out`
- `part-two-test-three-wt.out`

They show my output for a sequence of reads and writes with various cache configurations and various read and write sequences. The specific settings appear at the beginning of each file. For each cache configuration, there are two versions: wb is write-back; wt is write-through. Verify that you are getting the same results. All of these correspond to the sequences of reads/writes that appear in `part-two-addresses.txt`.

5 What to Submit

Submit your source code.

6 Graduate Students

Students taking the course for graduate credit, and undergraduates who want a bit of extra credit: add a command-line option that will represent the name of an ASCII file. This file will have lines this form:

- a line having a single hexademical value, representing an instruction address
- or a line having a hexadecimal value, followed by a space and either `R` or `W`, and a second hexadecimal value, representing a data address. "R" represents a read from the data address, and "W" represents a write to that data address

Here are a few example lines:

```
0x7f0dff69386a
0x7f0dff69386c
0x7f0dff69386f R 0x7f0dff8b3e00
0x7f0dff693876 W 0x7f0dff8b3bd8
0x7f0dff69387d
0x7f0dff693884
0x7f0dff693887 R 0x7f0dff8b3f90
```

If a filename is provided, then process each line of the file using your cache simulator. Keep track of reads, read misses, read hits, writes, write misses, and write hits. At the end, print info and statistics, like this:

```
----------------------------------------
cache size = 2048
cache block size = 64
cache #blocks = 32
cache #sets = 8
cache associativity = 4
cache tag length = 27
write back
----------------------------------------
# reads = 17600
# read misses = 2662 (15.12%)
```

```
# read hits = 14938 (84.88%)
# writes = 8800
# write misses = 142 (1.61%)
# write hits = 8658 (98.39%)
```

I've created trace files from different programs and put them in gitlab:

- `curl-portion.atrace.out`
- `cholesky-full.atrace.out`
- `rand-accesses.atrace.out`

Try your simulator on each. Vary the characteristics of the cache.

Important note: you don't actually have to read/write data from/to your simulated cache or from/to your simulated memory! Just analyze the behavior of the cache by collecting statistics.

Then, examine the behavior of instruction accesses compared to data accesses. You can do this one of two ways: either create two different caches in your program; or, much simpler, modify your program to process only the instructions or only the data accesses and do two runs. (Remember: all instruction accesses are reads.)

Study the effect of split caches, using the same total cache size as a single unified cache.