CS 3220 / CS 5220

Spring 2024

Cache Simulation Assignment

Additional Info

Memory

Memory consists of bytes

- and there are `memSize` bytes

Memory is grouped into regions called blocks; the block size is fixed

- for example, the block size might be 64; so each block is 64 bytes in size

Then in this case, each block will start on a 64-byte boundary

- i.e., the first address in each block will be an even multiple of 64
- and the last address will be `startAddress` + 63

`read_word()` will read four bytes at a time

Addresses

An address is just a number in the range of 0 to $memSize$ - 1

Assuming block size = 64, then to find the block that a particular address is in, do the following calculation:

- $blockNumber = \lfloor address/64 \rfloor$
- in other words, divide by 64 and round down (or do this as integer division)

The starting address of the block is then $64 \times blockNumber$

- and the last address in that block is $64 \times blockNumber + 63$

Example

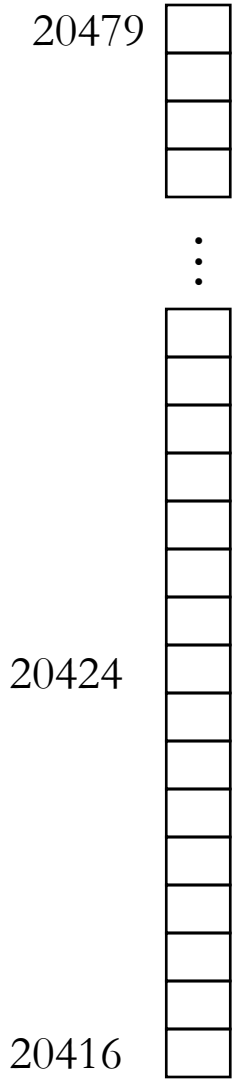
Suppose I do `read_word(20424)`

Then:

- $blockAddress = \lfloor {}^{20424}/_{64} \rfloor = 319$
- which means that this address is in block 319

And:

- the first address in this block is $64 \times 319 = 20416$
- the last address in this block is $20416 + 63 = 20479$
- the block offset for this address is $20424 - 24016 = 8$
- we can also see this since $20424 \; mod \; 64 = 8$



Example

So `read_word(20424)` will first look in the cache to see whether the block with the range 20416 to 20479 is in the cache

If it is, it will return the integer represented by the four bytes 20424 through 20427

- and it will treat this as a four-byte little-endian integer
- so the actual integer returned will be:

```
value = mem[20424] + 256*mem[20425] + 256*256*mem[20426] + 256*256*256*mem[20427]
```

Initializing Memory

"Initialize memory so that $memory[i] = i$ for each four-byte aligned value i with $0 \leq i \leq memSize$, where i is a four-byte integer."

What this means:

- the four bytes starting at memory[0] will be [0, 0, 0, 0]
- the four bytes starting at memory[4] will be [4, 0, 0, 0]
- the four bytes starting at memory[8] will be [8, 0, 0, 0]
- the four bytes starting at memory[256] will be [0, 1, 0, 0]
- the four bytes starting at memory[260] will be [4, 1, 0, 0]
- the four bytes starting at memory[39124] will be [212, 152, 0, 0]
- the four bytes starting at memory[39128] will be [216, 152, 0, 0]

because $39124 = 212 + 152 \times 256$

because $39128 = 216 + 152 \times 256$

Algorithm for Part One

Read only, with a direct-mapped cache

- read only: don't have to worry about dirty/clean
- read only: don't have to worry about write through vs. write back

Algorithm for Part One

The function `read_word(addr)` will do the following steps:

```
from addr, compute the tag t, index i and block offset b
look at the information in the cache for set i (there is only one block in the set)
if the block in set i is valid {
  if the tag for set i == t {
    // this is a hit
    return the word (the four bytes) at positions b, b+1, b+2, b+3 from the block in set i
  }
}
// this is a miss
compute the range of the desired block in memory: start to start+blocksize-1
read the blocksize bytes of memory from start to start+blocksize-1 into set i of the cache
set the valid bit for set i to true
set the tag set i to t
return the word at positions b, b+1, b+2, b+3 from the block in set i
```

start = blocksize * (addr // blocksize), using integer division

Example Data Structures

Example structures, in Python

```
# for a direct-mapped cache
MEMORY_SIZE      = 65536 # this is 2^16
CACHE_SIZE       = 1024  # this is 2^10
CACHE_BLOCK_SIZE = 64    # this is 2^6
ASSOCIATIVITY    = 1     # for Part One: direct-mapped cache


memory = bytearray(MEMORY_SIZE)
```

Example Data Structures

Helper function that I use, in Python

```
# helper function: compute the log base 2 of the input param

def logb2(val):
  i = 0
  assert val > 0
  while val > 0:
    i = i + 1
    val = val >> 1
  return i-1
```

Example Data Structures

Example structures, in Python

```
class CacheBlock:
  def __init__(self, cache_block_size):
    self.tag = -1
    self.dirty = False  # not needed for Part One
    self.valid = False
    self.data = bytearray(cache_block_size)


class CacheSet:
  def __init__(self, cache_block_size, associativity):
    self.blocks = [CacheBlock(cache_block_size) for i in range(associativity)]
    self.tag_queue = [-1 for i in range(associativity)] # not needed for Part One
```

Example Data Structures

Example structures, in Python

```
class Cache:
  def __init__(self, num_sets, associativity, cache_block_size):
    self.write_through = False  # not needed for Part One
    self.sets = [CacheSet(cache_block_size, associativity) for i in range(num_sets)]
    memory_size_bits = logb2(MEMORY_SIZE)
    self.cache_size_bits = logb2(CACHE_SIZE)
    self.cache_block_size_bits = logb2(CACHE_BLOCK_SIZE)
    self.index_length = logb2(NUM_SETS)
    self.block_offset_length = logb2(CACHE_BLOCK_SIZE)
```