Chapter 7 Solutions: For More Practice

7.6–7.8 The key features of solutions to these problems:

- Low temporal locality for code means no loops and no reuse of instructions.
- High temporal locality for code means tight loops with lots of reuse.
- Low spatial locality for code means lots of jumps to far away places.
- High spatial locality for code means no branches/jumps at all.

7.15 Effective CPI = Base CPI + Miss rate per instruction × Miss penalty

For memory configurations (a), (b), and (c):

a. Effective CPI = 1.2 + 0.005 × 177 = 2.085 clocks/instruction
b. Effective CPI = 1.2 + 0.005 × 45 = 1.425 clocks/instruction
c. Effective CPI = 1.2 + 0.005 × 57 = 1.485 clocks/instruction

Now

Speedup of A over B = Execution time B/Execution time A

and

Execution time = Number of instructions × CPI × Clock cycle time

We have the same CPU running the same software on the various memory configurations, so the number of instructions and the clock cycle time are fixed. Thus, speed can be compared by comparing CPI.

Speedup of configuration (b) over configuration (a) = 2.085/1.425 = 1.46

Speedup of configuration (b) over configuration (c) = 1.485/1.425 = 1.04

7.23 The largest direct-mapped cache with two-word blocks would be 512 KB in size. The address breakdown is 13 bits for tag, 16 bits for index, 1 bit for block offset and 2 bits for byte offset. The SRAMs will be used such that 8 of them form the data array (2 × 4) and 2 of them form the tag array (2 × 1).

7.24 The largest direct-mapped cache with four-word blocks would be 1024 KB in size. The address breakdown is 12 bits for tag, 16 bits for index, 2 bits for block offset, and 2 bits for byte offset. The SRAMs will be used such that 16 of them form the data array (2×8) and 2 of them form the tag array (2×1).

7.25

Reference	Hit or miss
2	Miss
3	Miss
11	Miss
16	Miss
21	Miss
13	Miss
64	Miss
48	Miss
19	Miss
11	Hit
3	Miss
22	Miss
4	Miss
27	Miss
6	Miss
11	Miss

Here is the final state of the cache with blocks given in LRU order:

Set #	Block 0 address (most recent)	Block 1 address (most recent)
0	48	64
1		
2	2	
3	11	27
4	4	
5	13	21
6	6	22
7		

7.26

Reference	Hit or miss
2	Miss
3	Miss
11	Miss
16	Miss
21	Miss
13	Miss
64	Miss
48	Miss
19	Miss
11	Hit
3	Hit
22	Miss
4	Miss
27	Miss
6	Miss
11	Hit

Here is the final state of the cache:

Block #	Address	LRU order
0	11	Most recently used
1	6	
2	27	
3	4	
4	22	
5	3	
6	19	
7	48	
8	64	
9	13	
10	21	
11	16	
12	2	
13		
14		
15		Least recently used

7.27

Reference	Hit or miss
2	Miss
3	Hit
11	Miss
16	Miss
21	Miss
13	Miss
64	Miss
48	Miss
19	Miss
11	Miss
3	Miss
22	Miss
4	Miss
27	Miss
6	Hit
11	Miss

Here is the final state of the cache with blocks given in LRU order:

Block	Addresses	LRU order
0	[8, 9, 10, 11]	(most recent)
1	[4, 5, 6, 7]	
2	[24, 25, 26, 27]	
3	[20, 21, 22, 23]	(least recent

7.30 Fully associative cache of 3K words? Yes. A fully associative cache has only one set, so no index is used to access the cache. Instead, we need to compare the tag with every cache location (3K of them).

Set-associative cache of 3K words? Yes. Implement a three-way set-associative cache. This means that there will be 1024 sets, which require a 10-bit index field to access.

Direct-mapped cache of 3K words? No. (Cannot be done efficiently.) To access a 3K-word direct-mapped cache would require between 11 and 12 index bits. Using 11 bits will allow us to only access 2K of the cache, while using 12 bits will cause us to map some addresses to nonexistent cache locations.

7.31 Fully associative cache of 300 words? Yes. A fully associative cache has only 1 set, so no index is used to access the cache. Instead, we need to compare the tag with every cache location (300 of them).

Set-associative cache of 300 words? Yes. Implement a 75-way set-associative cache. This means that there will be 4 sets, which require a 2-bit index field to access.

Direct-mapped cache of 300 words? No. (Cannot be done efficiently.) To access a 300-word direct-mapped cache would require between 8 and 9 index bits. Using 8 bits will allow us to only access 256 words of the cache, while using 9 bits will cause us to map some addresses to nonexistent cache locations.

7.36 We can build a 384 KB cache. The address breakdown is 14 bits for tag, 16 bits for set index, 0 bits for block offset, and 2 bits for byte offset. A total of 18 chips will be required; 6 for overhead.

7.37 No solution provided.

7.43 Using a 32-bit virtual address and 4 KB page size, the virtual address is partitioned into a 20-bit virtual page number and a 12-bit page offset. We divide the virtual page number into two 10-bit fields. The first field is the page table number and is used as an index into the first-level page table. The size of the first-level page table in 2^{10} entries $\times$ 4 bytes/entry = 2^{12} bytes = one page.

7.44 Pages are 4-KB in size and each entry uses 32 bits, so we get 1K worth of page table entries in a page. Each of these entries points to a physical 4-KB page, making it possible to address $2^{10} \times 2^{12} = 2^{22}$ bytes = 4 MB of memory. But only half of these are valid, so 2 MB of our virtual address space would be in physical memory. If there are 1K worth of entries per page table, the page table offset will occupy 10 bits and the page table number also 10 bits. Thus, we only need 4 KB to store the first-level page table as well.