Solutions for Chapter 7 Exercises

7.1 There are several reasons why you may not want to build large memories out of SRAM. SRAMs require more transistors to build than DRAMs and subsequently use more area for comparably sized chips. As size increases, access times increase and power consumption goes up. On the other hand, not using the standard DRAM interfaces could result in lower latency for the memory, especially for systems that do not need more than a few dozen chips.

7.2–7.4 The key features of solutions to these problems:

- Low temporal locality for data means accessing variables only once.
- High temporal locality for data means accessing variables over and over again.
- Low spatial locality for data means no marching through arrays; data is scattered.
- High spatial locality for data implies marching through arrays.

7.5 For the data-intensive application, the cache should be write-back. A write buffer is unlikely to keep up with this many stores, so write-through would be too slow.

For the safety-critical system, the processor should use the write-through cache. This way the data will be written in parallel to the cache and main memory, and we could reload bad data in the cache from memory in the case of an error.

7.9 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–set.

Cache set	Address
0000	48
0001	
0010	2
0011	3
0100	4
0101	21
0110	6
0111	
1000	
1001	27
1010	
1011	11
1100	
1101	13
1110	
1111	

7.10 2–miss, 3–hit, 11–miss, 16–miss, 21–miss, 13–miss, 64–miss, 48–miss, 19–miss, 11–hit, 3–miss, 22–hit, 4–miss, 27–miss, 6–hit, 11–miss

Cache set	Address
00	[0, 1, 2, 3]
01	[4, 5, 6, 7]
10	[8, 9, 10, 11]
11	[12, 13, 14, 15]

7.11 C stores multidimensional arrays in row-major form. Therefore accessing the array in row-major form will be faster since there will be a greater degree of temporal and spatial locality. Column-major form, on the other hand, will result in capacity misses if the block size is more than one word.

7.12 The Intrinsity caches are 16 KB caches with 256 blocks and 16 words per block. Data is 64 bytes = 512 bytes. The tag field is 18 bits (32 – (8 + 6)).

Total bits = 256 × (Data + Tag + Valid)

= 256 × (512 bits + 18 bits + 1 bit)

= 135,936 bits

7.13 Simply extend the comparison to include the valid bit as the high-order bit of the cache tag and extend the address tag by adding a high-order "1" bit. Now the values are equal only if the tags match and the valid bit is a 1.

7.14 The miss penalty is the time to transfer one block from main memory to the cache. Assume that it takes 1 clock cycle to send the address to the main memory.

a. Configuration (a) requires 16 main memory accesses to retrieve a cache block, and words of the block are transferred 1 at a time.

   Miss penalty = $1 + 16 \times 10 + 16 \times 1 = 177$ clock cycles.

b. Configuration (b) requires 4 main memory accesses to retrieve a cache block and words of the block are transferred 4 at a time.

   Miss penalty = $1 + 4 \times 10 + 4 \times 1 = 45$ clock cycles.

c. Configuration (c) requires 4 main memory accesses to retrieve a cache block, and words of the block are transferred 1 at a time.

   Miss penalty = $1 + 4 \times 10 + 16 \times 1 = 57$ clock cycles

7.16 The shortest reference string will have 4 misses for C1 and 3 misses for C2. This leads to 32 miss cycles versus 33 miss cycles. The following reference string will do: 0x00000000, 0x00000020, 0x00000040, 0x00000041.

7.17 AMAT = Hit time + Miss rate × Miss penalty

AMAT = $2 \text{ ns} + 0.05 \times (20 \times 2\text{ns}) = 4 \text{ ns}$

7.18 AMAT = $(1.2 \times 2 \text{ ns}) + (20 \times 2 \text{ ns} \times 0.03) = 2.4 \text{ ns} + 1.2 \text{ ns} = 3.6 \text{ ns}$

Yes, this is a good choice.

7.19 Execution time = Clock cycle × IC × (CPI_{base} + Cache miss cycles per instruction)

Execution $\text{time}_{\text{original}} = 2 \times \text{IC} \times (2 + 1.5 \times 20 \times 0.05) = 7 \text{ IC}$

Execution $\text{time}_{\text{new}} = 2.4 \times \text{IC} \times (2 + 1.5 \times 20 \times 0.03) = 6.96 \text{ IC}$

Hence, doubling the cache size to improve miss rate at the expense of stretching the clock cycle results in essentially no net gain.

7.20

Reference	Band	Bank Conflict
3	3	No
9	1	No
17	1	Yes (with 9)
2	2	No
51	3	No
37	1	Yes (with 17)
13	1	Yes (with 37
4	0	No
8	0	Yes (with 4)
41	1	No
67	3	No
10	2	NO

A bank conflict causes the memory system to stall until the busy bank has completed the prior operation.

7.21 No solution provided.

7.22 No solution provided.

7.28 Two principles apply to this cache behavior problem. First, a two-way set-associative cache of the same size as a direct-mapped cache has half the number of sets. Second, LRU replacement can behave pessimally (as poorly as possible) for access patterns that cycle through a sequence of addresses that reference more blocks than will fit in a set managed by LRU replacement.

Consider three addresses—call them A, B, C—that all map to the same set in the two-way set-associative cache, but to two different sets in the direct-mapped cache. Without loss of generality, let A map to one set in the direct-mapped cache and B and C map to another set. Let the access pattern be A B C A B C A . . . and so on. The direct-mapped cache will then have miss, miss, miss, hit, miss, miss, hit, . . . , and so on. With LRU replacement, the block at address C will replace the block at the address A in the two-way set-associative cache just in time for A to be referenced again. Thus, the two-way set-associative cache will miss on every reference as this access pattern repeats.

7.29

Address size:	k bits
Cache size:	S bytes/cache
Block size:	$B = 2^b$ bytes/block
Associativity:	A blocks/set

Number of sets in the cache:

$$\text{Sets/cache} = \frac{\text{(Bytes/cache)}}{\text{(Blocks/set)} \times \text{(Bytes/block)}}$$

$$= \frac{S}{AB}$$

Number of address bits needed to index a particular set of the cache:

$$\text{Cache set index bits} = \log_2 \text{(Sets/cache)}$$

$$= \log_2 \left(\frac{S}{AB}\right)$$

$$= \log_2 \left(\frac{S}{A}\right) - b$$

Number of bits needed to implement the cache:

Tag address bits/block = (Total address bits) – (Cache set index bits) – (Block offset bits)

$$= k - \left(\log_2\left(\frac{S}{A}\right) - b\right) - b$$

$$= k - \log_2\left(\frac{S}{A}\right)$$

Number of bits needed to implement the cache = sets/cache $\times$ associativity $\times$ (data + tag + valid):

$$= \frac{S}{AB} \times A \times \left(8 \times B + k - \log_2\left(\frac{S}{A}\right) + 1\right)$$

$$= \frac{S}{B} \times \left(8B + k - \log_2\left(\frac{S}{A}\right) + 1\right)$$

7.32 Here are the cycles spent for each cache:

Cache	Miss penalty	I cache miss	D cache miss	Total miss
C1	6 + 1 = 7	4% x 7 = 0.28	6% x 7 = 0.42	$0.28 \times \frac{0.42}{2} = 0.49$
C2	6 + 4 = 10	2% x 10 = 0.20	4% x 10 = 0.4	$0.20 \times \frac{0.4}{2} = 0.4$
C3	6 + 4 = 10	2% x 10 = 0.20	3% x 10 = 0.3	$0.20 \times \frac{0.3}{2} = 0.35$

Therefore C1 spends the most cycles on cache misses.

7.33 Execution time = CPI × Clock cycle × IC

We need to calculate the base CPI that applies to all three processors. Since we are given CPI = 2 for C1,

$$\text{CPI\_base} = \text{CPI} - \text{CPI}_{\text{misses}} = 2 - 0.49 = 1.51$$

$$\text{ExC1} = 2 \times 420 \text{ ps} \times \text{IC} = 8.4 \times 10^{-10} \times \text{IC}$$

$$\text{ExC2} = (1.51 + 0.4) \times 420 \text{ ps} \times \text{IC} = 8.02 \times 10^{-10} \times \text{IC}$$

$$\text{ExC3} = (1.51 + 0.35) \times 310 \text{ ps} \times \text{IC} = 5.77 \times 10^{-10} \times \text{IC}$$

Therefore C3 is fastest and C1 is slowest.

7.34 No solution provided.

7.35 If direct mapped and stride = 256, then we can assume without loss of generality that array[0] . . . array[31] is in block 0. Then, block 1 has array[32] . . . [63] and block 2 has array[64] . . . [127] and so on until block 7 has [224] . . . [255]. (There are 8 blocks × 32 bytes = 256 bytes in the cache.) Then wrapping around, we find also that block 0 has array[256] . . . [287], and so on.

Thus if we look at array[0] and array[256], we are looking at the same cache block. One access will cause the other to miss, so there will be a 100% miss rate. If the stride were 255, then array [0] would map to block 0 while array [255] and array [510] would both map to block 7. Accesses to array [0] would be hots, and accesses to array [255] and array [510] would conflict. Thus the miss rate would be 67%.

If the cache is two-way set associative, even if two accesses are in the same cache set, they can coexist, so the miss rate will be 0.

7.38 No solution provided.

7.39 The total size is equal to the number of entries times the size of each entry. Each page is 16 KB, and thus, 14 bits of the virtual and physical address will be used as a page offset. The remaining 40 – 14 = 26 bits of the virtual address constitute the virtual page number, and there are thus 2^{26} entries in the page table, one for each virtual page number. Each entry requires 36 – 14 = 22 bits to store the physical page number and an additional 4 bits for the valid, protection, dirty, and use bits. We round the 26 bits up to a full word per entry, so this gives us a total size of $2^{26} \times 32$ bits or 256 MB.

7.40 No solution provided.

7.41 The TLB will have a high miss rate because it can only access 64 KB (16×4 KB) directly. Performance could be improved by increasing the page size if the architecture allows it.

7.42 Virtual memory can be used to mark the heap and stack as read and write only. In the above example, the hardware will not execute the malicious instructions because the stack memory locations are marked as read and write only and not execute.

7.45 Less memory—fewer compulsory misses. (Note that while you might assume that capacity misses would also decrease, both capacity and conflict misses could increase or decrease based on the locality changes in the rewritten program. There is not enough information to definitively state the effect on capacity and conflict misses.)

Increased clock rate—in general there is no effect on the miss numbers; instead, the miss penalty increases. (Note for advanced students: since memory access timing would be accelerated, there could be secondary effects on the miss numbers if the hardware implements prefetch or early restart.)

Increased associativity—fewer conflict misses.

7.46 No solution provided.

7.47 No solution provided.

7.48 No solution provided.

7.49 No solution provided.

7.50 No solution provided.

7.51 No solution provided.

7.52 This optimization takes advantage of spatial locality. This way we update all of the entries in a block before moving to another block.