Solutions for Homework #8

Due 11:59pm Sunday, 20 November, 2016
Multiple submissions accepted.
Late homeworks are dinged 25% per full or partial day.

5.1 In this exercise we look at memory locality properties of matrix computation. The following code is written in C, where elements within the same row are stored contiguously. Assume each word is a 32-bit integer.

```
for (I=0; I<8; I++)
     for (J=0; J<8000; J++)
          A[I][J] = B[I][0] + A[J][I]
```

5.1.1 [5] <§5.1> How many 32-bit integers can be stored in a 16-byte cache block? Hint: don't worry about any of the extra stuff that has to accompany the integer like valid bits or tags, etc. Assume the memory is all for your data.
Answer**:** 4

5.1.2 [5] <§5.1> References to which variables exhibit temporal locality?
Answer**:** `I`, `J`, `A[I][J]`, `A[J][I]`and/or `B[I][0]`

Elaboration: We are asking which variables reference the same location repeatedly. In our inner loop, the variable I stays constant while J increments 8000 times, so that's one answer. `B[I][0]` also exhibits temporal locality as it references the same element in B 8000 times in a row. J is referenced repeatedly, so although its value changes, it's the same variable being referenced repeatedly. That qualifies. And by that argument, since A is referenced in every iteration, both `A[I][J]` and `A[J][I]`are accepted.

5.1.3 [5] <§5.1> References to which variables exhibit **spatial** locality?
Hint: choose one from among `I`, `J`, `A[I][J]` , `B[I][0]`, and `A[J][I]` .
Answer: `A[I][J]` and/or `J`.

Elaboration:
In C, as documented here, the first element of an array is a reference to the row. For example, with an array declaration of:

```
int a[3][4] = {
   {0, 1, 2, 3} ,   /*  initializers for row indexed by 0 */
   {4, 5, 6, 7} ,   /*  initializers for row indexed by 1 */
   {8, 9, 10, 11}   /*  initializers for row indexed by 2 */
};
```

references to `a[2][3]` will return value 6.

As we will see in a moment, references in MATLAB are identical – row first, then column. However, items in the same rows are located in adjacent memory locations in memory allocated by C. Therefore, to exhibit spatial locality, we want to increase the second parameter while holding the first constant. In our inner loop, that's `A[I][J]`.

==========

Locality is affected by both the reference order and data layout. The same computation can also be written below in **MATLAB**, which differs from the C code above by storing matrix elements within the same **column** contiguously in memory.

```
for I=1:8
     for J=1:8000
          A(I,J) = B(I,0) + A(J,I);
     end
end
```

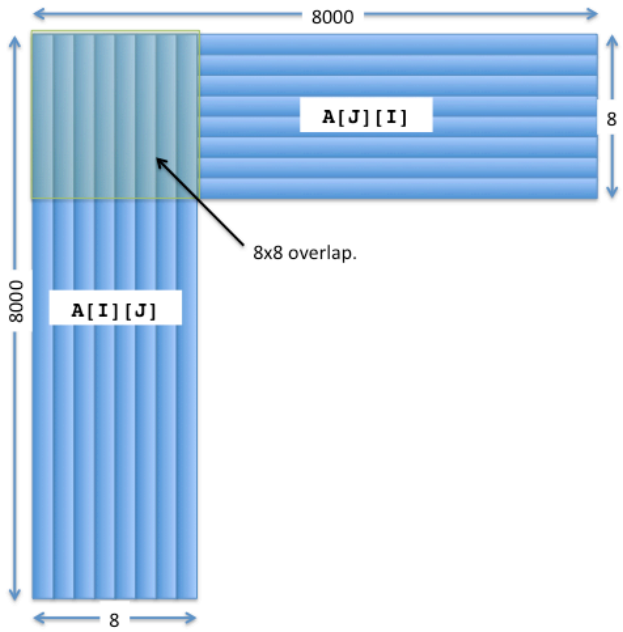
5.1.4 [10] <§5.1> How many 16-byte cache blocks are needed to store all 32-bit matrix elements being referenced?
Hint: I assumed the that the entries of A on the left and right of the equal sign could be shared, so I needed space for A and for the much smaller B. You can fit four 32-bit elements in a 16-byte block.
Answer**:** 32K (32768), or 31,986 16-byte cache blocks.

Elaboration: MATLAB and C need the same space.
B is an 8x1 matrix, it needs 2 blocks.
A ranges as follows:
For A(J,I): A(1:8000,1) → 2000 blocks …A(1:8000,8) → 2000 blocks = 16,000 blocks
For A(I,J): A(1,1:8000) → 2000 blocks …A(8,1:8000) → 2000 blocks = 16,000 blocks

They overlap when A(J,I)=A(1:8,1:8) and A(I,J)=A(1:8,1:8), so we need to remove 64 elements, or 16 blocks.

Total: 32,000-16+8 blocks

Graphically, here's how it looks:



That leaves us with:

$$16{,}000\ blks\ for\ A[J][I] + 16{,}000\ blks\ for\ A[I][J] - 16\ blks\ for\ overlap + 2\ blks\ for\ B$$
$$= 32{,}000 - 16 + 2$$
$$= 31986 \text{ blocks in the cache, roughly 32K 16-byte blocks, or 512KB.}$$

The formula will in fact require a 31986*16=511,776-byte cache, but you buy in powers of two. Too bad raises don't work that way ☺

If you want to read more detail about what technically we call "row-major" and "column-major" memory allocation, you might find this Wikipedia article interesting.

5.1.5 [5] <§5.1> References to which variables exhibit temporal locality?

Answer: Temporal means repeatedly accessing the same locations. "`I`" is repeatedly accessed, as is `B[I][0]`.

5.1.6 [5] <§5.1> References to which variables exhibit spatial locality?

Answer: `A[J][I]`

Elaboration: In MATLAB (as in C), the first variable of an array index is the row. So in the code from above:

```
for I=1:8
     for J=1:8000
          A(I,J) = B(I,0) + A(J,I);
     end
end
```

`A(J,I)` means the J^{th} row down, I^{th} element to the right, as shown in the example

here:

```
A = 16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1,
A(4,2)= 14.
```

In MATLAB, references to `A[J][I]` as `J` increases are adjacent. It's not `B[I][0]` because it's exactly the same element 8000 times, which is temporal locality. It's not `A[I][J]` because with `I` constant, `J` moves left to right, jumping through memory 8 locations at a time (there are 8 memory-adjacent elements per column in this array – the horizontal elements are not contiguous in memory).

5.2 Caches are important to providing a high-performance memory hierarchy to processors. Below is a list of 32-bit memory address references, given as word addresses.

```
3, 180, 43, 2, 191, 88, 190, 14, 181, 44, 186, 253
```

5.2.1 [10] <§5.3> For each of these references, identify the binary address, the tag, and the index given a direct-mapped cache with 16 one-word blocks. Also list if each reference is a hit or a miss, assuming the cache is initially empty.

Hint: use the table below for your answers. Remember the Tag refers to the block or chunk of data from which the data element in the cache came. The Index tells you which row of the cache it's in.

Word Address	Binary Address	Tag	Index	Hit/Miss
3				
180				
43				
2				
191				
88				
190				
14				
181				
44				
186				
253				

Answer:

Word Address	Binary Address	Tag	Index	Hit/Miss

3	0000 0011	0	3	M
180	1011 0100	11	4	M
43	0010 1011	2	11	M
2	0000 0010	0	2	M
191	1011 1111	11	15	M
88	0101 1000	5	8	M
190	1011 1110	11	14	M
14	0000 1110	0	14	M
181	1011 0101	11	5	M
44	0010 1100	2	12	M
186	1011 1010	11	10	M
253	1111 1101	15	13	M

Elaboration: Direct-mapped with 16 entries, each one word in size means we're breaking up our RAM into sections each of which has a word numbered 0 through 15 over and over and directing them towards one of 16 rows in our cache if called upon. In a 32-bit address system, that's a LOT of sections. We ignore 2 bits for the byte within a word, and 4 bits to cycle through the 16 variations in each segent, leaving 32-4-2=26 bits for the Tag. It must be 26 bits long to differentiate amongst all the possibe 16-word sections of RAM that have been loaded into a row of the cache. That's $2^{(32-4-2)}$ sections, about 67 million, to be exact. So for each of our 16 rows in our cache, there are 67M words that will compete for that index and thus that row of memory.

Mercifully, the authors have manufactured examples that all live within the first few 16-word blocks of RAM, so our tags can be written in decimal with a lot of leading 0's left off.

Index is the word address modulo 16, the remainder in other words. The tag is the leading 26 bits of the address which since our largest one is 253 is going to be mostly zeros (22 zeroes to be exact). So we can for now ignore the leading 22 bits (they're absolutely part of the address, and part of the tag, but we're dropping them for brevity).

Let's take an example of the address 190 (about half-way down). The key in the problem was when they told us these were *word* addresses in a system with 32-bit words. That means the 2 byte bits have already been dropped. 190 in binary is 1011 1110. That's what the table shows. The real binary address of this memory location would be 0000 0000 0000 0000 0000 0010 1111 1000. I underlined 190. Taking the lower 4 bits is needed to determine in which row of the cache we need to look for our particular word. Those lower 4 bits are 1110, or 14. Thus, an index of 14. The remaing 22 bits, ending in 1011 represent 11 in decimal. Thus our tag is 11. We've not used index 14 before, so it's a miss since we need to load the word from memory

and store "11" in the tag field.

5.2.2 [10] <§5.3> For each of these references, identify the binary address, the tag, and the index given a direct-mapped cache with two-word blocks and a total size of 8 blocks. Also list if each reference is a hit or a miss, assuming the cache is initially empty.

Hint: use this table again. A two-word block loads two words at the same time. Here, though, the Tag points to the start of a set of two adjacent words in memory that are being loaded into the cache. So if you load word n, you'll be loading word n+1 or n-1 along with it, whichever you need to make sure that you start off that 2-word block with even one (word 0, 2, 4, and so on). So you'll have a hit if either your intended word or its mate are requested.

Word Address	Binary Address	Tag	Index	Hit/Miss
3				
180				
43				
2				
191				
88				
190				
14				
181				
44				
186				
253				

Answer:

Word Address	Binary Address	Tag	Index	Hit/Miss
3	0000 0011	0	1	M
180	1011 0100	11	2	M
43	0010 1011	2	5	M
2	0000 0010	0	1	H
191	1011 1111	11	7	M
88	0101 1000	5	4	M
190	1011 1110	11	7	H
14	0000 1110	0	7	M
181	1011 0101	11	2	H

44	0010 1100	2	6	M
186	1011 1010	11	5	M
253	1111 1101	15	6	M

Elaboration: we only need 3 bits to determine the index this time since we're only dealing with 8 cache entries (each has doubled in size to two words, so we'll always load or store a pair of words at a time from/to RAM). So if the bit left the index field, did it go to the tag field or the ignored (byte #) bits? What we've done here is decided to address every 8 bytes (2 words) in RAM with a single double-word #. So we don't care which of those 8 bytes someone was asking for, we'll pull an 8-byte block in its entirety every time. The robbed bit goes to the ignored bits for determining if it's a hit or not. We still have 26 bits for the tag, only 3 for the index, and now 3 ignored. Still 32 bits total. The only change to addressing is in the index field, which is going to range from 0 to 7 instead of 0 to 15.

Let's look again at 190. In binary it's 0000 0000 0000 0000 0000 0010 1111 1000. We've underlined the same 8 bits as before, still shown in the "binary" colunn. However, we're going to ignore the least-significant bit for the purpose of determining the index. 1011 will give us the tag still, really twenty-two zeroes followed by 1011. Decimal 11 as before. But the next quartet, 1110, is resolved to 7 because we're pushing the least significant bit into the byte cluster. 111 is 7, and 000 is the byte #. Because of this, both 190 and 191 have the same bits 5-3: 111, so both have index 7, along with the same tag # 11. A reference to word address 190 is not only a Hit as a result, but we're assured it was loaded into that 7th row of the cache alongside word 191 which was referenced first.

5.2.3 [20] <§§5.3, 5.4> You are asked to optimize a cache design for the given references. There are three direct-mapped cache designs possible, all with a total of 8 words of data: C1 has 1-word blocks, C2 has 2-word blocks, and C3 has 4-word blocks. In terms of miss rate, which cache design is the best? If the miss stall time is 25 cycles, and C1 has an access time of 2 cycles, C2 takes 3 cycles, and C3 takes 5 cycles, which is the best cache design?

Hint: Use the table below to map out your index numbers and hit/miss rate. This is just an extension of the prior exercise. When you have an n-word block, you always load all n words back to the nearest multiple of n. For example, if asked to find whether word 3 is present in a cache with 8-word blocks, you know that if it's a hit, you will find words 0 through 7 in that block, which of course includes 3.

Word Address	Binary Address	Tag	Cache 1		Cache 2		Cache 3	
			Index	Hit/Miss	Index	Hit/Miss	Index	Hit/Miss
3								

180								
43								
2								
191								
88								
190								
14								
181								
44								
186								
253								

Once you have the hits and misses, calculate the miss rate for each cache design as a % of all accesses. All misses is 100% miss rate.

Then calculate total cycles per cache design. Use # misses x 25 cycles per miss to capture the stall time, but don't forget to add in that cache's access cycle which is per access.

	Cache 1	Cache 2	Cache 3
Miss rate:			
Total Cycles:			

Circle the winner

Answer:

Word Address	Binary Address	Tag	Cache 1		Cache 2		Cache 3	
			Index	Hit/Miss	Index	Hit/Miss	Index	Hit/Miss
3	0000 0011	0	3	M	1	M	0	M
180	1011 0100	22	4	M	2	M	1	M
43	0010 1011	5	3	M	1	M	0	M
2	0000 0010	0	2	M	1	M	0	M
191	1011 1111	23	7	M	3	M	1	M
88	0101 1000	11	0	M	0	M	0	M
190	1011 1110	23	6	M	3	H	1	H
14	0000 1110	1	6	M	3	M	1	M
181	1011 0101	22	5	M	2	H	1	M
44	0010 1100	5	4	M	2	M	1	M
186	1011 1010	23	2	M	1	M	0	M
253	1111 1101	31	5	M	2	M	1	M

	Cache 1	Cache 2	Cache 3
Miss rate:	12/12=100%	10/12=83%	11/12=92%
Total Cycles:	12x25+12x2=324	10x25+12x3=286	11x25+12x5=335

Cache 2 is the winner

Performance:

C1:	12 accesses:	12 x 2	= 24 cycles
	12 misses:	12x25	= 300 cycles
	Total:	324 cycles	
C2:	12 accesses:	12 x 3	= 36 cycles
	10 misses:	10 x 25	= 250 cycles
	Total: 286 cycles		
C3:	12 accesses:	12 x 5	= 60cycles
	11 misses:	11 x 25	= 275 cycles
	Total: 335 cycles		

Cache 2 provides the best performance.

Elaboration: The same manufactured examples are fortunate for us in that they all have tag #s with twenty-two leading zeros followed by 4 bits of significance. Which are shown as the left 4 bits of the binary field and the decimal number in the Tag column. That's constant for all 3 designs. What changes is the dividing line between how many bits are needed for the index vs how many can be ignored as part of the byte #. We only have 8 words to deal with so we only need 3 bits in the first design to distinguish which of the 8 rows are being indexed by our address. Those are bits 5-3. Just to check, a double-word block has 8 bytes, needed bits 0-2 of the address. That uses the least 3, so we're good.

When we double the number of words per block for design #2, we cut in half the number of possibile block #s, and thus our index shrinks to 2 bits, bits 5-4. As a check, we double from 8 bytes to 16, so we now need 4 bits to track the byte number, bits 0-3, so the byte # and block # but up against each other, so we're good.

Lastly, design 3 has a just two blocks, needing a single bit to distinguish between them. The bit pushed out joins the least bits. The rest is just lookups and math.

The following parameters apply to problems 5.2.4, .5, and .6:
There are many different design parameters that are important to a cache's overall performance. Below are listed parameters for different direct-mapped cache designs.

Cache Data Size: 32 KiB

Cache Block Size: 2 words

Cache Access Time: 1 cycle

5.2.4 [15] <§5.3> Calculate the total number of bits required for the cache listed above, assuming a 32-bit address. Given that total size, find the total size of the

closest direct-mapped cache with 16-word blocks of equal size or greater. Explain why the second cache, despite its larger data size, might provide slower performance than the first cache.

Hint: Assume you have a 32KiB cache to start with. Figure out how many total blocks you have, and how many bits you are using for your index. Remember that the rest are all for your tag, less any you can ignore for word size or words per block. You must always add a valid bit to each block as well.

Answer**:** Changing from 2-word to 16-word blocks will change the index, but not the tag, if you restrict the cache to the same size, because it reduces the number of blocks by a factor of 8, forcing you to move bits from index to offset.

However, if you allow the cache to grow to keep the same number of blocks, then the extra bits you push into the offset can't come from the index, as you still need the same number of bits to differentiate among the same number of blocks. Instead, the additional offset bits will come from the tag field.

Because the hint was confusing, we'll accept either. But normally you'd want to specify whether you're expanding the cache or just reorganizing it. .

You need a valid bit always, but for this problem it's optional, because it's not mentioned.
You should not have an LRU bit because the cache is direct-mapped.
Including a dirty bit is optional.

32KiB = 32768 bytes
4 bytes per word
2 words per block
32768 ÷ 4 ÷ 2 = 4096 blocks
log2(4096)=12, so index is 12 bits
2 bits for 4 bytes/word + 1 bit for 2 words/block = 3 bits of offset per block
32 total address bits – 12 for index – 3 for offset = 17 bits remaining for tag

4096 blocks *
First we must compute the number of cache blocks in the initial cache configuration. For this, we divide 32 KiB by 4 (for the number of bytes per word) and again by 2 (for the number of words per block). This gives us 4096 blocks and a resulting index field width of 12 bits. We also have a word offset size of 1 bit and a byte offset size of 2 bits. This gives us a tag field size of 32 - 15 = 17 bits. These tag bits, along with one valid bit per block, will require 18 x 4096 = 73728 bits or 9216 bytes. The total cache size is thus 9216 + 32768 = 41984 bytes, or 355,872 bits.

The total cache size can be generalized to
datasize = blocks x blocksize x wordsize
wordsize = 4

tagsize = 32 - log2(blocks) - log2(blocksize) - log2(wordsize)

totalsize = datasize + ([1 for valid bit +][1 for dirty bit +] tagsize) x block count

With valid bit only, 32KiB Cache:
totalsize = 355,872 bits, 41984 bytes

With no valid bit, 32KiB Cache:
Totalsize =

Increasing from 2-word blocks to 16-word blocks will reduce the tag size from 17 bits to 14 bits.
In order to determine the number of blocks, we solve the inequality:
41984 <= 64 x blocks + 15 x blocks

Solving this inequality gives us 531 blocks, and rounding to the next power of two gives us a 1024-block cache.
The larger block size may require an increased hit time and an increased miss penalty than the original cache. The fewer number of blocks may cause a higher conflict miss rate than the original cache.

5.2.5 [20] <§§5.3, 5.4> Generate a series of read requests that have a lower miss rate on a 2 KiB 2-way set associative cache than the cache listed above. Identify one possible solution that would make the cache listed have an equal or lower miss rate than the 2 KiB cache. Discuss the advantages and disadvantages of such a solution.

Hint: It's all about entries with the same index field but with tag fields that change or not.
Answer**:** Associative caches are designed to reduce the rate of conflict misses. As such, a sequence of read requests with the same 12-bit index field but a different tag field will generate many misses. For the cache described above, the sequence 0, 32768, 0, 32768, 0, 32768, …, would miss on every access, while a 2-way set associate cache with LRU replacement, even one with a significantly smaller overall capacity, would hit on every access after the first two.

5.2.6 [15] <§5.3> The formula shown in Section 5.3 shows the typical method to index a direct-mapped cache, specifically (Block address) modulo (Number of blocks in the cache). Assuming a 32-bit address and 1024 blocks in the cache, consider a different indexing function, specifically (Block address [31:27] XOR Block address [26:22]). Is it possible to use this to index a direct-mapped cache? If so, explain why and discuss any changes that might need to be made to the cache. If it is not possible, explain why.
Answer**:** Yes, it is possible to use this function to index the cache. However, information about the five bits is lost because the bits are XOR'd, so you must include more tag bits to identify the address in the cache.

5.3 For a direct-mapped cache design with a 32-bit address, the following bits of the address are used to access the cache.

Tag	Index	Offset
31-10	9-5	4-0

5.3.1 [5] <§5.3> What is the cache block size (in words)?
Hint: The offset bits are those which are simply carried around the cache and added back to the final address if needed to handle a fault. The cache doesn't bother storing them. Since the 4 bytes within a word account for 2 of those bits, the rest are ignored because every combination of them is part of the cache's data entry.
Answer**:** 8

5.3.2 [5] <§5.3> How many entries does the cache have?
Hint: the tag bits are used to see if what is in the indexed row to which you've been pointed is the particular block you wanted, so it's not the tag bits.
Answer**:** 1 + (22/8/32) = 1.086

5.3.3 [5] <§5.3> What is the ratio between total bits required for such a cache implementation over the data storage bits?
Hint: Ignore the presence of a valid bit. They didn't specify that it had one, so we'll assume it does not. Page 390 walks through an example, "Bits in a Cache." The one confusing thing to me was the formula, which I'll retype but with annotations here for you: 32 bits in an address that could be used for a tag – 10 bits that must be used to index to narrow down to which block in the cache we're dealing with – 2 bits for there being 4 words in a block – 2 bits for there being 4 bytes in a word + 1 valid bit.

Starting from power on, the following byte-addressed cache references are recorded.

Address											
0	4	16	132	232	160	1024	30	140	3100	180	2180

Answer**:** 1 + (22/8/32) = 1.086

5.3.4 [10] <§5.3> How many blocks are replaced?
Answer**:** 3

5.3.5 [10] <§5.3> What is the hit ratio?
Answer**:** 0.25

5.3.6 [20] <§5.3> List the final state of the cache, with each valid entry represented as a record of <index, tag, data>.

Index	Tag	Data

Answer:

```
<Index, tag, data>
<0000012, 00012, mem[1024]>
<0000012, 00112, mem[16]>
<0010112, 00002, mem[176]>
<0010002, 00102, mem[2176]>
<0011102, 00002, mem[224]>
<0010102, 00002, mem[160]>
```

5.7 This exercise examines the impact of different cache designs, specifically comparing associative caches to the direct-mapped caches from Section 5.4. For these exercises, refer to the address stream shown in Exercise 5.2.

5.7.1 [10] <§5.4> Using the sequence of references from Exercise 5.2, show the final cache contents for a three-way set associative cache with two-word blocks and a total size of 24 words. Use LRU replacement. For each reference identify the index bits, the tag bits, the block offset bits, and if it is a hit or a miss.

Answer:

The cache would have 24 / 3 = 8 blocks per "way" and thus an index field of 3 bits.

Word Address	Binary Address	Tag	Index	Hit/ Miss	Way 0	Way 1	Way 2
3	0000 0011	0	1	M	Tag(Index 1)=0, T(I2)=11		
180	1011 0100	11	2	M	T(I1)=0, T(I2)=11, T(I5)=2		
43	0010 1011	2	5	M	T(I1)=0, T(I2)=11, T(I5)=2		
2	0000 0010	0	2	C	T(I1)=0, T(I2)=11, T(I5)=2, T(I2)=0		
191	1011 1111	11	7	M	T(I1)=0, T(I2)=11, T(I5)=2, T(I2)=0, T(I7)=11		
88	0101 1000	5	4	M	T(I1)=0, T(I2)=11, T(I5)=2, T(I2)=0, T(I7)=11, T(I4)=5		
190	1011 1110	11	7	H	T(I1)=0, T(I2)=11, T(I5)=2, T(I2)=0, T(I4)=5, T(I7)=11		
14	0000 1110	0	7	M	T(I1)=0, T(I2)=11, T(I5)=2, T(I2)=0, T(I4)=5, T(I7)=11	T(I7)=0	
181	1011 0101	11	2	H	T(I1)=0, T(I2)=11, T(I5)=2, T(I2)=0, T(I4)=5, T(I7)=11	T(I7)=0	

44	0010 1100	2	6	M	T(I1)=0, T(I2)=11, T(I5)=2, T(I2)=0, T(I4)=5, T(I7)=11, T(I6)=2	T(I7)=0	
186	1011 1010	11	5	M	T(I1)=0, T(I2)=11, T(I5)=2, T(I2)=0, T(I4)=5, T(I7)=11, T(I6)=2	T(I7)=0, T(I5)=11	
253	1111 1101	15	6	M	T(I1)=0, T(I2)=11, T(I5)=2, T(I2)=0, T(I4)=5, T(I7)=11, T(I6)=2	T(I7)=0, T(I5)=11, T(I6)=15	

5.7.2 [10] <§5.4> Using the references from Exercise 5.2, show the final cache contents for a fully-associative cache with one-word blocks and a total size of 8 words. Use LRU replacement. For each reference identify the index bits, the tag bits, and if it is a hit or a miss.

Answer:

Tag	Hit/Miss	Contents
3	M	3
180	M	3, 180
43	M	3, 180, 43
2	M	3, 180, 43, 2
191	M	3, 180, 43, 2, 191
88	M	3, 180, 43, 2, 191, 88
190	M	3, 180, 43, 2, 191, 88, 190
14	M	3, 180, 43, 2, 191, 88, 190, 14
181	M	181, 180, 43, 2, 191, 88, 190, 14
44	M	181, 44, 43, 2, 191, 88, 190, 14
186	M	181, 44, 186, 2, 191, 88, 190, 14
253	M	181, 44, 186, 253, 191, 88, 190, 14

5.7.3 [15] <§5.4> Using the references from Exercise 5.2, what is the miss rate for a fully-associative cache with two-word blocks and a total size of 8 words, using LRU replacement? What is the miss rate using MRU (most recently used) replacement? Finally what is the best possible miss rate for this cache, given any replacement policy?

Answer:

Address	Tag	Hit/Miss	Contents
3	1	M	1
180	90	M	1, 90
43	21	M	1, 90, 21
2	1	H	1, 90, 21
191	95	M	1, 90, 21, 95
88	44	M	1, 90, 21, 95, 44
190	95	H	1, 90, 21, 95, 44
14	7	M	1, 90, 21, 95, 44, 7
181	90	H	1, 90, 21, 95, 44, 7
44	22	M	1, 90, 21, 95, 44, 7, 22
186	143	M	1, 90, 21, 95, 44, 7, 22, 143
253	126	M	1, 90, 126, 95, 44, 7, 22, 143

The final reference replaces tag 21 in the cache, since tags 1 and 90 had been reused at time=3 and time=8 while 21 hadn't been used since time=2.
Miss rate = 9/12 = 75%
This is the best possible miss rate, since there were no misses on any block that had been previously evicted from the cache. In fact, the only eviction was for tag 21, which is only referenced once.

Multilevel caching is an important technique to overcome the limited amount of space that a first-level cache can provide while still maintaining its speed. Consider a processor with the following parameters:

Base CPI, No Memory Stalls	Processor Speed	Main Memory Access Time	First-Level Cache Miss Rate per Instruction	Second-Level Cache, Direct-Mapped Speed	Global Miss Rate with Second-Level Cache, Direct-Mapped	Second-Level Cache, Eight-Way Set Associative Speed	Global Miss Rate with Second Level Cache, Eight-Way Set Associative
1.5	2 GHz	100 ns	7%	12 cycles	3.50%	28 cycles	1.50%

5.7.4 [10] <§5.4> Calculate the CPI for the processor in the table using: 1) only a first-level cache, 2) a second-level direct-mapped cache, and 3) a second-level eight-way set-associative cache. How do these numbers change if main memory access time is doubled? If it is cut in half?
Answer:
L1 only:
.07 x 100 = 7 ns
CPI = 7 ns / .5 ns = 14

Direct mapped L2:
.07 x (12 + 0.035 x 100) = 1.1 ns
CPI = ceiling(1.1 ns/.5 ns) = 3
8-way set associated L2:
.07 x (28 + 0.015 x 100) = 2.1 ns
CPI = ceiling(2.1 ns / .5 ns) = 5

Doubled memory access time, L1 only:
.07 x 200 = 14 ns
CPI = 14 ns / .5 ns = 28

Doubled memory access time, direct mapped L2:
.07 x (12 + 0.035 x 200) = 1.3 ns
CPI = ceiling(1.3 ns/.5 ns) = 3

Doubled memory access time, 8-way set associated L2:
.07 x (28 + 0.015 x 200) = 2.2 ns
CPI = ceiling(2.2 ns / .5 ns) = 5

Halved memory access time, L1 only:
.07 x 50 = 3.5 ns
CPI = 3.5 ns / .5 ns = 7

Halved memory access time, direct mapped L2:
.07 x (12 + 0.035 x 50) = 1.0 ns
CPI = ceiling(1.1 ns/.5 ns) = 2

Halved memory access time, 8-way set associated L2:
Chapter 5 Solutions S-11
.07 x (28 + 0.015 x 50) = 2.1 ns
CPI = ceiling(2.1 ns / .5 ns) = 5

5.7.5 [10] <§5.4> It is possible to have an even greater cache hierarchy than two levels. Given the processor above with a second-level, direct-mapped cache, a designer wants to add a third-level cache that takes 50 cycles to access and will reduce the global miss rate to 1.3%. Would this provide better performance? In general, what are the advantages and disadvantages of adding a third-level cache?

5.7.6 [20] <§5.4> In older processors such as the Intel Pentium or Alpha 21264, the second level of cache was external (located on a different chip) from the main processor and the first-level cache. While this allowed for large second-level caches, the latency to access the cache was much higher, and the bandwidth was typically lower because the second-level cache ran at a lower frequency. Assume a 512 KiB off-chip second-level cache has a global miss rate of 4%. If each additional 512 KiB of cache lowered global miss rates by 0.7%, and the cache had a total access time a 50 cycles, how big would the cache have to be to match the performance of the second-level direct-mapped cache listed above? Of the eight-way set-associative cache?

5.9 This Exercise examines the single-error-correcting, double-error-detecting (SEC/DED) Hamming code.

5.9.1 [5] <§5.5> What is the minimum number of parity bits required to protect a 128-bit word using the SEC/DED code?

5.9.2 [5] <§5.5> Section 5.5 states that modern server memory modules (DIMMs) employ SEC/DED ECC to protect each **64** bits with 8 parity bits. Compute the

cost/performance ratio of this code to the code from 5.9.1. In this case, cost is the relative number of parity bits needed while performance is the relative number of errors that can be corrected. Which is better?
Hint: Chapter 5.5 has 64 bits protected by 8 error bits, so a total of 72 bits to protect the 64 data bits. That's known as a (72,64) code. The ratio of total bits to data bits gives you the cost, and the ratio of errors detected to total bits gives you the performance. Both are percentages. Calculate that for both schemes, and then divide the cost by the performance and get a factor for each. This is the cost/performance ratio, and the lower one wins.

Note to grader: the original problem statement was to protect 4 bits. 8 parity bits on 4 data bits will correct 3+ errors, I believe. But that is not the desired solution. If someone worked the problem with 4 data bits, count it correct.

Answer: The (72,64) code described in the chapter requires an overhead of 8/64 = 12.5% additional bits to tolerate the loss of any single bit within 72 bits, providing a protection rate of 1.4%. The (137,128) code from an overhead of 9/128 = 7.0% additional bits to tolerate the loss of any single bit within 137 bits, providing a protection rate of 0.73%. The cost/performance of both codes is as follows:

(72,64) code → 12.5/1.4 = 8.9
(136,128) code →_ 7.0/0.73 = 9.6
The (72,64) code has a better cost/performance ratio.

5.9.3 Consider an SEC code that protects 8-bit words with 4 parity bits. If we read the value 0x375 is there an error? If so, correct it.

Hint: use the bit numbering from section 5.5, where you number bit from the left starting at 1. Use this table to fill out the bits so you can tell the parity bits apart from the data bits. You'll need to follow the steps in the example problem on page 420-21 of the book.

Answer: 0x375 is 0011 0111 0101
We use this mapping to show which bits are covered by which parity bit.

	bit position	1	2	3	4	5	6	7	8	9	10	11	12
		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8
Coverage Matrix	p1	x		x		x		x		x		x	
	p2		x	x			x	x			x	x	
	p4				x	x	x	x					x
	p8								x	x	x	x	x
	p16												
	p32												
	p64												

If we lay out our 12 bits in this pattern, we can XOR each row. Seeing position p8

XOR to 1 tells us that there is an error in that bit.

	data string	001101110101											
	Bitwise split	0	0	1	1	0	1	1	1	0	1	0	1
	Bit Position	1	2	3	4	5	6	7	8	9	10	11	12
	data bit position			1		2	3	4		5	6	7	8
	data bit value	0	0	1	1	0	1	1	1	0	1	0	1
	bit position	1	2	3	4	5	6	7	8	9	10	11	12
	XOR	p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8
Re-calculated Bits	p1	0		1		0		1		0		0	
	p2	0		0	1			1	1			1	0	
	p4	0				1	0	1	1					1
	p8	1								1	0	1	0	1
	p16	0												
	p32	0												
	p64	0												
Bit in error		8												

Flipping bit 8 gives us the corrected result:

Corrected String	0	0	1	1	0	1	1	0	0	1	0	1					
	001101100101																
	Error-corrected string shown																

Extra problem:
Read the extra reading for HW #7 that is alongside the homework assignment in Blackboard, entitled "ErrorCorrectionAndDetectionSupplement.pdf".

Do the following problems from that reading:
1. A 12-bit Hamming code word containing 8 bits of data and 4 parity bits is read from memory. What was the original 8-bit data word that was written into memory if the 12-bit word read out is:

(a) 010011111000 (b) 011101010010 (c) 010000000101

Answer:

(a) bit 7 is in error. Original data: 0110 1000
(b) no error. Original data: 1010 0010
(c) bit 4 is in error. Original data: 0000 0101

4. A modified single-error-correcting, double-error-detecting Hamming code for

four bits of data D3 , D5 , D6 , and D7 has the following parity bit equations:

P1 = D3⊕ D5⊕ D6
P2 = D3⊕ D5⊕ D7
P4 = D3⊕ D6⊕ D7
P8 = D5⊕ D6⊕ D7

(a) Find the binary values of the four check bits for a single error in each of the eight bit positions of the code.

Answer:

Bit	C1	C2	C3	C4
1	1	0	0	0
2	0	1	0	0
3	1	1	1	0
4	0	0	1	0
5	1	1	0	1
6	1	0	1	1
7	0	1	1	1
8	0	0	0	1

Elaboration: XOR is a mathematical function that results in a "1" if the number of "1" is odd, "0" if it's even. Very helpful for these calculations.

Further, if you XOR the parity bit along with the data bits it covers, you should always get a "0" since the parity bit is supposed to make the "1" an even number. So it's fast to determine if a parity bit and its corresponding data is in error. Doing this final check is usually written as an equation with C on one side and XORs of the parity and data bits on the other. For our problem, the check equations are:

C1 = P1⊕D3⊕D5⊕D6
C2 = P2⊕D3⊕D5⊕D7
C4 = P4⊕D3⊕D6⊕D7
C8 = P8⊕D5⊕D6⊕D7

From this we can start setting each bit into error and see which C values change. Let's take the first bit, #1, represented by P1. If P1 is in error, then C1 will be 1. No other C values will change, because none include P1.
For bit #2 to be in error, P2, only C2 is a "1".
For bit #3 to be in error, C1, C2, and C3 change, because all reference D3. And so on.

(b) Assuming that either a single or double error has occurred, indicate the type of error for each of the following words read from memory:
(1) 00110011

(2) 01100100
(3) 01000101

Answer:
(1) impossible result – error in bit 13.
(2) single error in bit 8/p8.
(3) single error in bit 7/d7.

Elaboration:
In (1), 00110011 is P1 P2 D3 P4 D5 D6 D7 P8
Applying our equations from above,
C1 = P1⊕D3⊕D5⊕D6 = 0⊕1⊕0⊕0 = 1. Error.
C2 = P2⊕D3⊕D5⊕D7 = 0⊕1⊕0⊕1 = 0. No error.
C4 = P4⊕D3⊕D6⊕D7 = 1⊕1⊕0⊕1 = 1. Error.
C8 = P8⊕D5⊕D6⊕D7 = 1⊕0⊕0⊕1 = 1. No error.

C8C4C2C1 = 1101 = 13_{10}. Bit 13 is in error, except we have no bit 13.

In (2), 01100100 maps as follows:
C1 = P1⊕D3⊕D5⊕D6 = 0⊕1⊕0⊕1 = 0. No error.
C2 = P2⊕D3⊕D5⊕D7 = 1⊕1⊕0⊕0 = 0. No error.
C4 = P4⊕D3⊕D6⊕D7 = 0⊕1⊕1⊕0 = 0. No error.
C8 = P8⊕D5⊕D6⊕D7 = 0⊕0⊕1⊕0 = 1. Error.

C8C4C2C1 = 1000 = 8_{10}. Bit 8 is in error.

In (3), 01000101 maps as follows:
C1 = P1⊕D3⊕D5⊕D6 = 0⊕0⊕0⊕1 = 1. Error.
C2 = P2⊕D3⊕D5⊕D7 = 1⊕0⊕0⊕0 = 1. Error.
C4 = P4⊕D3⊕D6⊕D7 = 0⊕0⊕1⊕0 = 1. Error.
C8 = P8⊕D5⊕D6⊕D7 = 1⊕0⊕1⊕0 = 0. Error.

C8C4C2C1 = 0111 = 7_{10}. Bit 7 is in error.