16.482 / 16.561: Computer Architecture and Design

Spring 2015

Homework #8 Solution

1. *Virtual memory (30 points) Answer the following questions about a process that uses the page table below:*

Virtual page #	Valid bit	Reference bit	Dirty bit	Frame #
0	1	0	0	5
1	1	0	1	0
2	0	0	0	--
3	1	1	0	3
4	0	0	0	--
5	1	1	1	2
6	0	0	0	--
7	1	1	0	1

   a. *(10 points) Which pages above are candidates to be evicted on a page fault? Which, if any, are better candidates to evict?*

Solution: Any valid page in which the reference bit is set to 0 is a candidate to be evicted on a page fault—pages 0 and 1 in this case.

Pages in which you don't have to do any "work"—in this context, have any interaction with the disk—might be a better candidate. You could therefore argue that page 0 is a better candidate for eviction than page 1, because page 1 is dirty and will therefore require a write back to disk when it is evicted.

b. (20 points) Assuming 8 KB pages, what physical addresses would the virtual addresses below map to? Note that some virtual addresses may not have a valid translation, in which case you should note that address causes a page fault.

- *0xABCD*
- *0x1792*
- *0x4680*
- *0x5701*

Solution: The first step is to break down the addresses into page offsets and virtual page numbers. Since the page size is 8 KB = 2^{13} B, the offset field is 13 bits. The virtual page number is therefore the upper 3 bits of the address. We can now consider the translation of each address:

- 0xABCD = **101**0 1011 1100 1101_2
  - Page # = $101_2 = 5$ → Frame # = 2
  - Replace 5 with 2 → physical address = **010**0 1011 1100 1101_2 = **0x4BCD**
- 0x1792 = **000**1 0111 1001 0010_2
  - Page # = $000_2 = 0$ → Frame # = 5
  - Replace 0 with 5 → physical address = **101**1 0111 1001 0010_2 = **0xB792**
- 0x4680 = **010**0 0110 1000 0000_2
  - Page # = $010_2 = 2$
  - Page 2 is not valid → **no valid translation; page fault**
- 0x5701 = **010**1 0111 0000 0001_2
  - Page # = $010_2 = 2$
  - Page 2 is not valid → **no valid translation; page fault**

2. Cache optimizations: Multi-banked/non-blocking caches (40 points) Assume we have a system containing 64 blocks of memory, labeled B0-B63. (We do not need to know the block size for this problem.) This system has a 16-line, direct-mapped cache that is initially empty. Assume we have a program that accesses 20 of these blocks in the following order, with one access initiated per cycle unless a stall occurs:

63, 12, 28, 24, 42, 10, 14, 52, 51, 18, 0, 1, 3, 16, 4, 23, 8, 36, 27, 31

Note that, with an initially empty cache and no access to the same block twice, all of these accesses will miss. You should assume each miss takes 10 cycles to handle. This problem assesses the impact of several different cache organizations on those misses.

a. (5 points) Calculate the total time for these 20 accesses if the cache is not split into banks and is therefore a blocking cache (i.e., only one access can be handled at a time).

Solution: If the cache is not split into banks, then only one access can be handled at a time. Therefore, the total time is (20 accesses) * (10 cycles / access) = **200 cycles**

b. (15 points) Calculate the total time for these 20 accesses if the cache is divided evenly into four banks. Assume the blocks are not interleaved sequentially, so blocks are mapped to cache lines using normal direct mapping. In other words, B0 maps to cache line 0, B1 to cache line 1, and so on. Assume bank 0 holds cache lines 0-3, bank 1 holds cache lines 4-7, etc.

Solution: Remember, accesses to different banks can be overlapped and will start in consecutive cycles (for example, if an access to one bank starts in cycle i, the next access can start in cycle i+1). Since we are not using sequential interleaving, the blocks are mapped to cache lines—and therefore to banks—as shown in the table below:

Line #	Blocks mapped to this line	
0	0,16,32,48	Bank 0
1	1,17,33,49	
2	2,18,34,50	
3	3,19,35,51	
4	4,20,36,52	Bank 1
5	5,21,37,53	
6	6,22,38,54	
7	7,23,39,55	
8	8,24,40,56	Bank 2
9	9,25,41,57	
10	10,26,42,58	
11	11,27,43,59	
12	12,28,44,60	Bank 3
13	13,29,45,61	
14	14,30,46,62	
15	15,31,47,63	

b (cont.)
Now, we can determine the total access time by looking at each access and determining which ones can be overlapped. Note that up to 4 accesses can be overlapped—1 per bank. The table below shows each access, the bank it accesses, and the start and end time of those accesses. Another way to solve this problem is a "pipeline diagram" of sorts that might graphically show which accesses can overlap and therefore how long they take.

In total, the sequence takes **127 cycles** given this cache organization.

Block #	Bank	Start cycle	End cycle
63	3	1	10
12	3	11	20
28	3	21	30
24	2	22	31
42	2	32	41
10	2	42	51
14	3	43	52
52	1	44	53
51	0	45	54
18	0	55	64
0	0	65	74
1	0	75	84
3	0	85	94
16	0	95	104
4	1	96	105
23	1	106	115
8	2	107	116
36	1	116	125
27	2	117	126
31	3	118	127

c. (15 points) Calculate the total time for these 20 accesses if the cache is divided evenly into four banks and the blocks are interleaved sequentially across those four banks.

Solution: Sequential interleaving yields a different cache layout than in part (b); the idea behind sequential interleaving is to place blocks with spatial locality in different banks. The relevant mapping for this problem is shown in the first table on the next page

Given this mapping, we can overlap a few more accesses and complete this access sequence in **114 cycles**, as shown in the second table on the next page.

Mapping of blocks to cache lines for part (c):

Line #	Blocks mapped to this line	
0	0,16,32,48	Bank 0
1	4,20,36,52	
2	8,24,40,56	
3	12,28,44,60	
4	1,17,33,49	Bank 1
5	5,21,37,53	
6	9,25,41,57	
7	13,29,45,61	
8	2,18,34,50	Bank 2
9	6,22,38,54	
10	10,26,42,58	
11	14,30,46,62	
12	3,19,35,51	Bank 3
13	7,23,39,55	
14	11,27,43,59	
15	15,31,47,63	

Bank #, start, and end times for each access in sequence:

Block #	Bank	Start cycle	End cycle
63	3	1	10
12	0	2	11
28	0	12	21
24	0	22	31
42	2	23	32
10	2	33	42
14	2	43	52
52	0	44	53
51	3	45	54
18	2	53	62
0	0	54	63
1	1	55	64
3	3	56	65
16	0	64	73
4	0	74	83
23	3	75	84
8	0	84	93
36	0	94	103
27	3	95	104
31	3	105	114

d. (5 points) Determine the ideal mapping of memory blocks to cache banks for this program (i.e., the mapping that gives you the minimum total time for all 20 accesses).

Solution: The ideal mapping for this particular program would be one that would allow you to always have four simultaneous cache accesses going at once. Therefore, every four consecutive blocks in the sequence should be in different banks. It also makes sense to map blocks to different lines within the bank whenever possible to potentially avoid later cache misses.

The simplest scheme is to place the first block in line 0 of bank 0, the next block in line 0 of bank 1, and so on. We show the results of that scheme below. Note that only the 20 blocks that are accessed during the program are mapped for this problem. The remaining 44 blocks in the system don't matter for this particular example.

Line #	Blocks mapped to this line	
0	63, 8	Bank 0
1	42	
2	51	
3	3	
4	12, 36	Bank 1
5	10	
6	18	
7	16	
8	28, 27	Bank 2
9	14	
10	0	
11	4	
12	24, 31	Bank 3
13	52	
14	1	
15	23	

3. *Cache optimizations: Early restart/critical word first (30 points) You are studying a system with the following memory characteristics. The 32-bit processor contains a single 64 KB direct-mapped cache with 256-byte blocks. A 32-bit bus connects the cache to 2 GB of off-chip memory. The bus is capable of supplying a word to the cache every 250 µs.*

Determine the average latency for a cache miss under three conditions:
- *Sequential cache block fill (i.e., start with word 0 and fill all words in block)*
- *Early restart*
- *Critical word first with early restart*

In addition, show the speedup over sequential block fill for both early restart and critical word first with early restart. (In other words, how much faster is each of the enhanced techniques than sequential block fill?) You should assume that the requested word is, on average, in the middle of the cache block (since cache blocks hold an even number of words, choose the first "middle" word—word 1 in a 4-word block, word 3 in an 8-word block, etc.).

Solution: First, note that the problem contains far more information than you really need. The important points are:
- Each word is 32 bits, or 4 bytes, since we have a 32-bit processor.
- Each cache block therefore holds 256 / 4 = 64 words.
- The memory is capable of supplying a single word at a time, and it takes 250 µs to send each word.
- On average, the desired word is halfway through the block—in this case, it's the 32nd word to enter the cache during a sequential fill.

Let's take each of these 1 by 1:
- In sequential cache block fill, the cache returns the desired data to the processor only when the entire block is full. Therefore, the time required to fill the block is:

  (64 words) x (250 µs/word) = 16000 µs = **16 ms**

- If we use early restart alone, the cache will supply the data as soon as the desired word reaches the cache, effectively ending the miss at that point. In this case, we only need to fill half the block to get that word, so the time required is:

  (32 words) x (250 µs/word) = 8000 µs = **8 ms**

  The speedup of this technique over sequential fill is 16 ms / 8 ms = **2**.

- If we use critical word first with early restart, memory will supply the desired word first, rather than always fetching from word 0. The cache can then supply that word to the processor, effectively ending the word after 1 memory bus cycle, which takes **250 µs**.

  The speedup of this technique over sequential fill is 16 ms / 250 µs = **64**.