Homework Assignment #2
SAMPLE SOLUTION

Q1. (6 points) Please briefly explain: 1) what the "memory wall" challenge refers to in computer architecture; and 2) why do we need the memory hierarchy design.

ANSWER: The "memory wall" challenge refers to the growing performance gap (or speed gap, or bandwidth gap) between CPU and memory. The gap is further enlarging with the aggregate performance of multiple cores for multicore processors.

The memory hierarchy design is critical to bridge the performance gap between CPU and memory. It becomes even more important to bridge the gap given recent multicore processors.

Q2. (6 points) Please briefly explain what the temporal locality and spatial locality are. Please give an example for each of them, respectively.

ANSWER: The locality principle refers to programs tend to access a small proportion of instruction and data at any time. More specifically, the temporal locality refers to locality in time, i.e., instruction and data accessed recently are likely to be accessed again soon. The spatial locality refers to locality in space, i.e., the instruction and data near those accessed recently are likely to be accessed soon.

Temporal locality examples include instructions in a loop and the induction variables, i.e., those variables that get increased or decreased by a fixed amount on every iteration. (One example is enough)

Spatial locality examples include accessing instructions in a sequential/consecutive manner and accessing array data in a sequential/consecutive manner.

Q3. (10 points) Assume we have a direct-mapped cache. The cache size is 2^n blocks (thus *n* bits are used for the index), and the block size is 2^m words (2^{m+2} bytes). Assume we have 32-bit addresses.

a. What is the size (the number of bits) of each tag field?

ANSWER: given the assumption in this question, we have the address subdivision as shown below:

Tag	Index (i.e., block number): ***n* bits**	Byte offset within a block: ***m* + 2 bits**

Since we have 32-bit addresses, the tag field takes $32 - (n + m + 2)$ bits (or $30 - n - m$ bits).

b. What is the total number of bits needed for this direct-mapped cache, including valid field, tag field, and data field?

ANSWER: as shown below, this direct-mapped cache has 2^n cache blocks, with index from 0 to 2^n-1. Each cache block has valid field, tag field, and data field. The valid field takes 1 bit. The tag field takes 30-n-m bits. The data field (i.e., the block size) takes 2^m words = 2^{m+2} bytes = 2^{m+5} bits (or 32 * 2^m bits).

Index	Valid field	Tag field	Data field
0			
1			
2			
…			
…			
2^n-1			

Thus, the total number of bits needed for this direct-mapped cache is:

2^n * (valid field size + tag field size + data field size)

= 2^n * (1 + (30 - n – m) + 32 * 2^m)

= 2^n * (31 - n – m + 32 * 2^m)

Q4. (15 points) 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 (as a word is 4 bytes, the word addresses are byte addresses shifted right by 2 bits, i.e., byte addresses divided by 4)

```
0x03, 0xb4, 0x2b, 0x02, 0xbf, 0x58, 0xbe, 0x0e, 0xb5, 0x2c,
0xba, 0xfd
```

a. 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.

ANSWER: given the assumption in this question, and since each cache block has the size of one word, the word addresses given are also the block addresses. Additionally, since we have 16 (i.e., 2^4) cache blocks, the index field takes 4 bits. Thus, we have the memory byte address and block address as shown below:

Memory **byte address** subdivision:

Tag: **26 bits**	Index (i.e., block number): **4 bits**	Byte offset within a block: **2 bits**

Block address subdivision:

Tag: **26 bits**	Index (i.e., block number): **4 bits**

Then, we have the sequence of accesses as shown below. Note that, for each memory access, we need to derive the block address (same as the word address in this example), and the tag and index fields. The index field is used to select the cache block. After we select the cache block, and if the tag field in the cache matches with the tag field of the requesting block, we have a cache hit. Otherwise, we have a cache miss. In this example, each memory access results in a cache miss.

Word address, i.e., block address	Block address in binary	Tag (high-order 26 bits of block address)	Index (low-order 4 bits of block address)	Hit/Miss
3	[0.....0] 0000 0011	[0.....0] 0000	0011	M
180	[0.....0] 1011 0100	[0.....0] 1011	0100	M
43	[0.....0] 0010 1011	[0.....0] 0010	1011	M
2	[0.....0] 0000 0010	[0.....0] 0000	0010	M
191	[0.....0] 1011 1111	[0.....0] 1011	1111	M
88	[0.....0] 0101 1000	[0.....0] 0101	1000	M
190	[0.....0] 1011 1110	[0.....0] 1011	1110	M
14	[0.....0] 0000 1110	[0.....0] 0000	1110	M
181	[0.....0] 1011 0101	[0.....0] 1011	0101	M
44	[0.....0] 0010 1100	[0.....0] 0010	1100	M
186	[0.....0] 1011 1010	[0.....0] 1011	1010	M
253	[0.....0] 1111 1101	[0.....0] 1111	1101	M

b. 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.

ANSWER: compared to part (a), the difference is that each cache block now has the size of two words (i.e., 8 bytes, or 2^3 bytes), thus the byte offset within a block takes the low-order 3 bits of the memory byte address. As such, the block address now has 29 bits (i.e., 32 – 3 = 29 bits). Additionally, since we have 8 (i.e., 2^3) cache blocks, the index field takes 3 bits. Thus, we have the memory byte address and block address as shown below:

Memory **byte address** subdivision:

Tag: **26 bits**	Index (i.e., block number): **3 bits**	Byte offset within a block: **3 bits**

Block address subdivision:

Tag: **26 bits**	Index (i.e., block number): **3 bits**

Then, we have the sequence of accesses as shown below. Similar to part (a), for each memory access, we need to derive the block address (now different from the word address in this example), and the tag and index fields. The index field is used to select the cache block. After we select the cache block, and if the tag field in the cache matches with the tag field of the requesting block, we have a cache hit. Otherwise, we have a cache miss. In this new example, we have 3 hits.

Word address	**Word address** in binary	**Block address** in binary	Tag (high-order 26 bits of block address)	Index (low-order 3 bits of block address)	Hit/Miss
3	[0.....0] 0000 0011	[0.....0] 0000 001	[0.....0] 0000	001	M
180	[0.....0] 1011 0100	[0.....0] 1011 010	[0.....0] 1011	010	M

43	[0.....0] 0010 1011	[0.....0] 0010 101	[0.....0] 0010	101	M
2	[0.....0] 0000 0010	[0.....0] 0000 001	[0.....0] 0000	001	**H**
191	[0.....0] 1011 1111	[0.....0] 1011 111	[0.....0] 1011	111	M
88	[0.....0] 0101 1000	[0.....0] 0101 100	[0.....0] 0101	100	M
190	[0.....0] 1011 1110	[0.....0] 1011 111	[0.....0] 1011	111	**H**
14	[0.....0] 0000 1110	[0.....0] 0000 111	[0.....0] 0000	111	M
181	[0.....0] 1011 0101	[0.....0] 1011 010	[0.....0] 1011	010	**H**
44	[0.....0] 0010 1100	[0.....0] 0010 110	[0.....0] 0010	110	M
186	[0.....0] 1011 1010	[0.....0] 1011 101	[0.....0] 1011	101	M
253	[0.....0] 1111 1101	[0.....0] 1111 110	[0.....0] 1111	110	M

This example shows larger cache block size can reduce misses in general because of spatial locality. On the other hand, as we discussed in class, if the cache capacity is fixed, larger block size results in a smaller number of cache blocks, which can increase conflict misses (or collision misses). The end result can be complicated and depend on applications too (i.e., depend on the sequence of memory accesses).

Q5. (12 points) Using the sequence of references from Q4, show the final cache contents for a three-way set associative cache with two-word blocks (i.e., the block size is 8 bytes) and a total size of 24 blocks. Use the LRU (least-recently used) replacement policy. For each reference, identify the index bits, the tag bits, and if it is a hit or a miss.

ANSWER: compared to Q3 (b), the difference is that, now we assume the cache is a three-way set associative cache, not direct-mapped cache anymore. Since each cache block has the size of two words (i.e., 8 bytes, or 2^3 bytes), thus the byte offset within a block also takes the low-order 3 bits of the memory byte address. The block address has 29 bits (i.e., 32 – 3 = 29 bits).

However, since now the cache is a three-way set associative cache, it means each set has three cache block slots. A cache block mapped to a specific set can go into any one of three slots in that set. If that set is full, we assume to use LRU replacement algorithm to kick out one block to make the room for the new block. Additionally, this cache has a total of 24 cache blocks. In other words, there're 24 / 3 = 8 sets. Since we have 8 (i.e., 2^3) sets, the index takes 3 bits. Note that, the index is now used to select a set, not a block number anymore. After a set is selected, a three-way comparator is used to compare three tags in a set against the requesting block simultaneously to determine whether it's a cache hit or a cache miss.

We have the memory byte address and block address as shown below:

Memory **byte address** subdivision:

Tag: **26 bits**	Index (i.e., **set number**): **3 bits**	Byte offset within a block: **3 bits**

Block address subdivision:

Tag: **26 bits**	Index (i.e., **set number**): **3 bits**

Then, we have the sequence of accesses as shown below. Similar to Q3, for each memory access, we need to derive the block address, and the tag and index fields. The index field is used to select the set, not a direct-mapped cache block anymore. After a set is selected, a three-way comparator

is used to compare three tags in a set against the requesting block simultaneously to determine whether it's a cache hit (if a match is found) or a cache miss (no match is found). In this new example, we still have 3 hits. As none of 8 sets are full yet, the LRU replacement policy is not being used. The attached Excel form shows a more detailed view with 8 sets shown same time.

Word Address	**Block address** in binary	Tag	Index (**set number**)	Hit / Miss	Way 0 (tag)	Way 1 (tag)	Way 2 (tag)
3	[0.....0] 0000 001	[0.....0] 0000	001	M	[0.....0] 0000		
180	[0.....0] 1011 010	[0.....0] 1011	010	M	[0.....0] 1011		
43	[0.....0] 0010 101	[0.....0] 0010	101	M	[0.....0] 0010		
2	[0.....0] 0000 001	[0.....0] 0000	001	**H**	[0.....0] 0000		
191	[0.....0] 1011 111	[0.....0] 1011	111	M	[0.....0] 1011		
88	[0.....0] 0101 100	[0.....0] 0101	100	M	[0.....0] 0101		
190	[0.....0] 1011 111	[0.....0] 1011	111	**H**	[0.....0] 1011		
14	[0.....0] 0000 111	[0.....0] 0000	111	M	[0.....0] 1011	[0.....0] 0000	
181	[0.....0] 1011 010	[0.....0] 1011	010	**H**	[0.....0] 1011		
44	[0.....0] 0010 110	[0.....0] 0010	110	M	[0.....0] 0010		
186	[0.....0] 1011 101	[0.....0] 1011	101	M	[0.....0] 0010	[0.....0] 1011	
253	[0.....0] 1111 110	[0.....0] 1111	110	M	[0.....0] 0010	[0.....0] 1111	

Q6. (12 points) Assume that main memory accesses take 70 ns and that memory accesses are 36% of all instructions. The following table shows data for L1 caches attached to each of two processors, P1 and P2.

	L1 size	L1 miss rate	L1 hit time
P1	2 KiB	8.0%	0.66ns
P2	4 KiB	6.0%	0.90ns

a. What is the Average Memory Access Time for P1 and P2?

ANSWER: given that AMAT = Hit Time + Miss Rate * Miss Penalty, we have:

AMAT for P1: 0.66ns + 8% * 70ns = 6.26ns

AMAT for P2: 0.90ns + 6% * 70ns = 5.1ns

b. Assuming that the L1 hit time determines the cycle times for P1 and P2, i.e., the clock rates are 1.52GHz and 1.11 GHz for P1 and P2, respectively. Assuming a base CPI of 1.0 without any memory stalls, what is the total CPI for P1 and P2, respectively?

ANSWER: given that CPI = CPI + (Miss Rate) * (Memory Access Instruction) * (Miss Penalty in Cycles), we have:

Total CPI for P1: 1 + 8% * 36% * (70/0.66) ≈ 4.05 CPI

Total CPI for P2: 1 + 6% * 36% * (70/0.90) ≈ 2.68 CPI

c. Assuming P1 now is added with an additional L2 cache, with the following table shows data for this L2 cache, what is the AMAT for P1 with the addition of an L2 cache?

L2 size	L2 miss rate	L2 hit time
1 MiB	20%	5.62ns

ANSWER: given that: AMAT = Hit Time (L1) + Miss Rate (L1) * Miss Penalty (L1), and Miss Penalty (L1) = Hit Time (L2) + Miss Rate (L2) * Miss Penalty (L2)

We have: Miss Penalty (L1) = 5.62ns + 20% * 70ns = 19.62ns

Therefore, the AMAT for PI with the addition of an L2 cache is: 0.66ns + 8% * 19.62ns ≈ 2.23ns.

Q7. (10 points) Given three variables: the cache capacity, the block size, and the cache associativity, and assume one variable changes and the other two are fixed, please fill in the below table with "increase", "decrease", or "same".

ANSWER:

	Compulsory misses	Capacity Misses	Conflict misses	Hit time	Miss penalty
Larger block size, same cache capacity, same associativity	Decrease (block size increased, the misses caused by the first word access reduced)	Increase (# of blocks reduced)	Increase (# of blocks reduced)	Decrease (# of blocks reduced, i.e., search time reduced)	Increase (block size increased, i.e., the latency fetching a missing block increased)
Larger cache capacity, same block size, same associativity	Same (same block size)	Decrease (capacity increased)	Decrease (# of blocks increased)	Increase (# of blocks increased, i.e., search time increased)	Same (same block size, i.e., the latency fetching a missing block is the same)
Higher associativity, same cache capacity, same block size	Same (same block size)	Same (same cache capacity)	Decrease (higher associativity reduces conflict misses)	Increase (higher associativity requires more tag comparisons, increasing the search time)	Same (same block size, i.e., the latency fetching a missing block is the same)

Q8. (6 points) Please list two sample compiler optimization techniques that can reduce the cache miss rate and also briefly explain why.

ANSWER: Two sample compiler optimization techniques discussed in the class include loop interchange technique and blocking technique.

The loop interchange technique can swap nested loops so that programs can access memory in a sequential/consecutive manner, instead of a strided manner, which can result in better spatial locality and thus reduce the cache miss rate.

The blocking technique partitions matrices into smaller, sub-matrices (or "(matrix) blocks", but not to confuse with cache blocks), so that they fit into the cache, instead of accessing entire rows or columns, which improves both temporal and spatial locality and thus reduce the cache miss rate.

Q9. (8 points) Please explain what virtual memory is. Please briefly explain these terminologies: virtual address, physical address, page, page fault, page table, and Translation Look-aside Buffer (TLB).

ANSWER: Virtual memory is a concept that uses main memory as a "cache" for secondary (disk) storage. With the virtual memory concept, the addresses generated by CPU are now virtual addresses, forming a virtual address space, and need to be translated into physical addresses (physical memory addresses) to access the memory.

Virtual address: an address generated from CPU, corresponding to a location in virtual address space.

Physical address: an address in the physical main memory.

Page: since virtual memory is like a "cache" for secondary storage, the "cache block" of virtual memory is called a page.

Page fault: if a page is not in the main memory, like a "cache miss", it's called page fault.

Page table is a structure that performs virtual-to-physical address translation. It's indexed by the virtual page number, and the page table entry (PTE) stores the translated physical page number.

Translation Look-aside Buffer (TLB): TLB is a cache that keeps track recently used address translations to avoid an access to the page table.

Q10. (15 points) Please pick one of the papers listed below (the paper PDF file can be downloaded from Blackboard), read the paper in detail, and write a short summary of the paper you studied. Please limit to maximum 400 words, and please focus on what problem is studied in the paper and what are the key conclusions.

[1] J. Leidel and Y. Chen. HMC-Sim-2.0: A Co-Design Infrastructure for Exploring Custom Memory Cube Operations. The International Journal of Parallel Computing (ParCo), Volume:

68, Pages: 77 - 88, 2017.

[2] J. Leidel and Y. Chen. HMC-SIM: A Simulation Framework for Hybrid Memory Cube Devices. Journal of Parallel Processing Letters, Volume: 24, Issue: 04, Pages: 1465 - 1474, December 2014.

[3] W. Xie, Y. Chen and P. Roth. Parallel-DFTL: A Flash Translation Layer that Exploits Internal Parallelism in Solid State Drives. In Proceedings of the 11th IEEE International Conference on Networking, Architecture, and Storage (NAS'16), Pages: 1 - 10, 2016. DOI Best Paper Nominee.

[4] K. Zhang, Z. Wang, Y. Chen, H. Zhu and X.-H. Sun. PAC-PLRU: A Cache Replacement Policy to Salvage Discarded Predictions from Hardware Prefetchers. In the Proc. of the 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'11), 2011.

N/A

THE END.