CMPE 110 Computer Architecture
Fall 2015, Homework #3

Computer Engineering
UC Santa Cruz

November 19, 2015

Name: _____

Email: _____

Submission Guidelines:

- This homework is due on Friday 11/20/15.
- The homework must be submitted to ecommons by 7:59am.
  - Anything later is a late submission
- **Please write your name and your UCSC email address**
- **The homework should be "readable" without too much effort**
  - The homework must be typed and submitted as a single file in PDF format
  - Please name your homework file cmpe110-hw3-yourcruzid.pdf
  - Please keep your responses coherent and organized or you may lose points
- Provide details on how to reach a solution. **An answer without explanation gets no credit. Clearly state all assumptions.**
- Points: 64 = 16 + 16 + 12 + 20

Question	Part A	Part B	Part C	Part D	Part E	Part F	Total
1				-	-	-	
2							
3				-	-	-	
4				-	-	-	
Total							

Question 1. Instruction Cache (16 points)

In this problem, we will be exploring adding an instruction cache to a standard 5-stage pipelined processor. We will be using the MIPS assembly program shown below in Figure 1. The first column shows the instruction address for each instruction. Note that these addresses are byte addresses. The value of r1 is initially 64, meaning that there are 64 iterations in the loop. In this problem, we will be considering the execution of this loop with a direct-mapped instruction cache microarchitecture with eight 16-Byte cache lines. This means each cache line can hold four instructions and the bottom four bits of an instruction address are the block offset. Hint: The first instruction in Figure 1 (i.e., `addiu r1, r1, -1`), is in the middle of a cache line.

Address	Instruction	Iteration 1	Iteration 2
	`loop:`		
0x108	`addiu r1, r1, -1`		
0x10c	`addiu r2, r2, 1`		
0x110	`j foo`		
	`...`		
	`foo:`		
0x218	`addiu r6, r6, 1`		
0x21c	`bne r1, r0, loop`		

Figure 1: MIPS assembly loop

Question 1.A Catagorizing Cache Misses (4 points)

Fill out the table above. In the appropriate column, write *compulsory*, *conflict*, or *capacity* next to each instruction which misses in the instruction cache to indicate the type of instruction cache misses that occur in the first and second iteration of the loop. Assume that the instruction cache is initially completely empty.

- *Compulsory misses* occur when a memory access maps to a currently invalid (or empty) cache line or cache set.
- *Conflict misses* occur when multiple memory addresses map to the same cache line or cache set.
- *Capacity misses* occur when the cache is full and can no longer handle further memory access requests or contain the working set of data needed for program execution.

Question 1.B Average Memory Access Latency (6 points)

Calculate the instruction cache miss rate for 64 iterations of the loop. Calculate the average instruction cache memory access latency in cycles for 64 iterations of the loop. Assume the hit time is one cycle and that the miss penalty is 5 cycles. You must show your work, especially the various components of the average memory access latency. **Remark on which kind of miss is dominating the average memory access latency.**

Question 1.C Set-Associativity (6 points)

Qualitatively, predict how the cache performance would change if we replace the eight-entry, direct-mapped cache with an eight-entry, two-way, set-associative cache. Both caches have a one-cycle hit latency. Assume the set-associative cache address interleaves the sets across the ways using the least significant bits right after the block offset. **What kind of misses would be present with this kind of cache microarchitecture?**

Question 2. Cache Mapping and Access (16 points)

Consider a 512-KByte cache with 64-word cachelines (a cacheline is also known as a cache block, each word is 4-Bytes). This cache uses write-back scheme, and the address is 32 bits wide.

Question 2.A Direct-Mapped, Cache Fields (2 points)

Assume the cache is direct-mapped. Fill in the table below to specify the size of each address field.

Field	**Size (bits)**
Cacheline Offset	
Cacheline Index	
Tag	

Question 2.B Fully-Associative, Cache Fields (2 points)

Assume the cache is fully-associative. Fill in the table below to specify the size of each address field.

Field	**Size (bits)**
Cacheline Offset	
Cacheline Index	
Tag	

Question 2.C 16-Way Set-Associative, Cache Fields (2 point)

Assume the cache is 16-way set-associative. Fill in the table below to specify the size of each address field.

Field	**Size (bits)**
Cacheline Offset	
Cacheline Index	
Tag	

Question 2.D Direct-Mapped, Cache Transactions (4 points)

Assume the cache is direct-mapped. Fill in the table on the next page to identify the content of the cache after each of the following memory accesses. Assume the cache is initially empty (also called a "cold cache"). Specify if an entry causes another line to be replaced from the cache, and if an entry has to write its data back to memory. For the data column, specify the data in the block by referring to its address like M[address]. Write accesses will modify the data, so lets indicate the data after a write access with D[address].

Address	Request Type	Cachline Index	Hit or Miss?	Modified	Tag	Data	Caused Replace?	Write-back to Memory?
0x128	read					M[0x100]		
0xF40	write					D[0xF00]		
0xA00051	read							
0x093	write							
0x4000B44	read							

Question 2.E 16-Way Set-Associative, Cache Transactions (4 points)

Assume the cache is 16-way set-associative. Fill the table below to identify the content of the cache after each of the following memory accesses. Assume the cache is empty in the beginning (also known as cold cache). Specify if an entry causes another line to be replaced from the cache, and if an entry has to write its data back to memory. For the data column, specify the data in the block by referring to its address like M[address]. Write accesses will modify the data, so lets indicate the data after a write access with D[address].

Address	Request Type	Cachline Index	Hit or Miss?	Modified	Tag	Data	Caused Replace?	Write-back to Memory?
0x128	read					M[0x100]		
0xF40	write					D[0xF00]		
0xA00051	read							
0x093	write							
0x4000B44	read							

Question 2.F Overhead (2 points)

What is the overhead and actual size of the direct-mapped cache? What is the overhead and actual size of the 16-way set-associative cache? Does the structure change the overhead in terms of number of memory bits?

Question 3. Average Memory Access Time (12 points)

Considered two pipelined machines A and B, described in the table below.

Property	**Computer A**	**Computer B**
ISA	MIPS	x86
Clock Frequency	2 GHz	3 GHz
Base CPI	1 cycle/instruction	2 cycles/instruction
L1 Instruction Cache Hit Time	1 cycle	1 cycle
L1 Instruction Cache Miss Rate	2%	2%
L1 Data Cache Hit Time	1 cycle	2 cycles
L1 Data CAche Miss Rate	8%	5%
L2 Hit Time	15 cycles	12 cycles
L2 Global Miss Rate	3%	4%
Main Memory Access Time	250 cycles	300 cycles

Figure 2: Table containing properties of two different machines

Question 3.A Ideal System (4 points)

Assume a perfect memory system (100% of memory accesses hit in the L1 caches) and perfect branch prediction. Assume that MIPS programs execute 1.25x as many instructions as x86 programs. Which machine has the faster execution time and what is the speedup?

Question 3.B No L2 Cache (4 points)

Now assume each machine has only L1 caches (split iL1 cache and dL1 cache), no L2 cache, perfect branch prediction, and that 30% of the instructions are loads/stores. Which machine has the faster execution time and what is the speedup?

Question 3.C Full System (4 points)

Now assume each machine has both L1 caches and a unified L2 cache, and that 30% of the instructions are loads/stores. Which machine has the faster execution time and what is the speedup?

Question 4. Array vs. List Cache Behavior (20 points)

In this problem, you will explore the cache behavior for a basic operation on two common software data structures. Figure 3 illustrates a basic linear array and a doubly linked list. Each node in the doubly linked list has a 4-Byte value field, a 4-Byte pointer that points to the next node in the list, and a 4-Byte pointer that points to the previous node in the list. The previous pointer for the head node is defined to be zero, and the next pointer for the tail node is defined to be zero. For this problem, you should assume that both data structures contain 64 4-Byte values (i.e., the array is 64 elements long, and the linked list contains 64 nodes).

We wish to explore the performance of reversing the values in each data structure. Conventional wisdom in a basic computer science course on data structures might suggest that the time to complete this operation on both the array and linked list is $O(n)$ where n is the number of elements in the data structure. "Big-O" notation is useful when analyzing asymptotic behavior as n grows very large, but it abstracts many important "constant factors" that can dominate the performance for reasonable sized data structures on real architectures.

For this problem, you should **assume we have a very large, fully-associative data cache such that there are no capacity nor conflict misses. Assume a write-back, no write-allocate cache.** The data cache uses 16-Byte cache lines and is initially empty. All data cache accesses will result in either a hit or a compulsory miss. You should also assume that the array data structure is cache-line aligned, each node in the linked list is also cache-line aligned, and there is only one linked list node per cache line. Cache-line aligned means that the first element of the array is at the very beginning of a cache line, and that value field for each linked list node is also at the very beginning of a cache line.

The data cache has a hit latency of one cycle and a miss penalty of four cycles. This means if an instruction stalls in M waiting for a cache miss, it will remain in the M stage for a total of five cycles (one cycle for the hit latency and four cycles for the miss penalty). The processor should stall for both read and write misses.

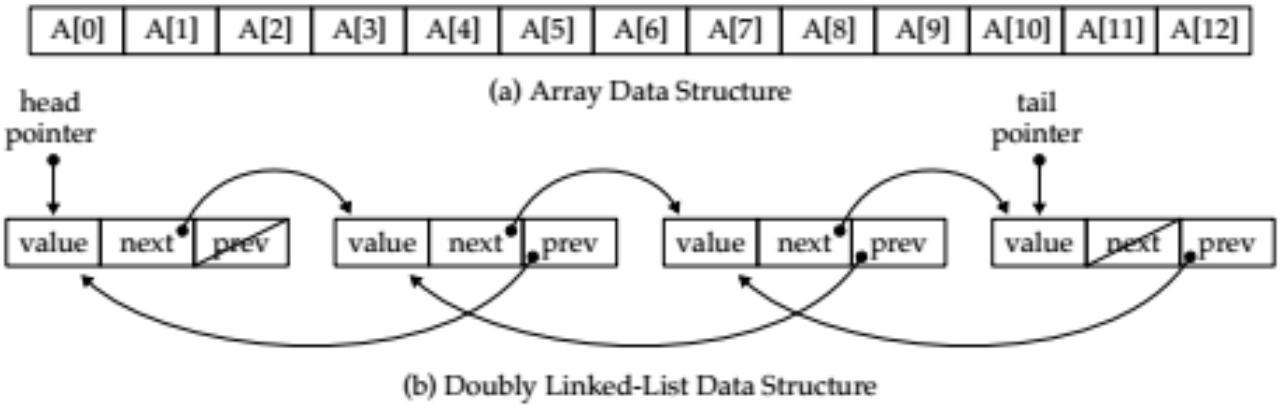

Figure 3: Linear Array and Doubly Linked-List Data Structures

Question	Number of Instructions	CPI	Execution Time (cyc)	CPI Breakdown			
				Useful Work	RAW Stalls	Control Squashes	Memory Stalls
4.A							
4.B							

Figure 4: Execution Time for Reverse Operation on Array and Linked List Data Structures

Question 4.A Analyzing Performance of an Array Data Structure (7 points)

Figure 5 on the next page shows C-code which implements the reverse operation for an array data structure with an even number of elements. Figure 6 shows corresponding MIPS assembly code for the function. Recall that arguments are stored in r4 and r5. Note that we are organizing the loop a bit differently than some of the other loops we have studied in this course to better match the assembly code and also to better match the code used for the linked list. We use a few setup instructions to create a pointer to the last element in the array. We then iteratively move the forward and reverse pointers, swapping the data as we go along.

Draw a pipeline diagram illustrating at least one iteration of the loop shown in Figure 6. You do not need to show the four setup instructions used to calculate the initial reverse pointer. You should draw as many iterations as you need in order to determine the steady-state behavior of the loop. Carefully consider which memory accesses hit or miss in the data cache. **Use your pipeline diagram to estimate the overall execution time in cycles for this operation. Fill in the corresponding row of Figure 4.** You will need to decompose the CPI into its various components based on what stalls and/or squashes occur in the execution. **You must show your work.**

```
1  void array_reverse( int* A, int n ) {
2
3    # Code only works for even n
4    assert( n % 2 == 0 )
5
6    int* fwd_ptr = &A[0];
7    int* rev_ptr = &A[n-1];
8    int* rev_ptr_last = 0;
9
10   do {
11     # Swap values at fwd and rev pointers
12     int temp = *fwd_ptr;
13     *fwd_ptr = *rev_ptr;
14     *rev_ptr = temp;
15
16     # Save rev pointer for exit condition check
17     rev_ptr_last = rev_ptr;
18
19     # Update fwd and rev pointers
20     fwd_ptr++;
21     rev_ptr--;
22   }
23   while ( fwd_ptr != rev_ptr_last );
24
25 }
```

Figure 5: C-Code for Reverse on Array Data Structure

```
1    # r4: A
2    # r5: n
3
4    addiu r5,  r5, -1
5    addiu r12, r0, 4
6    mul   r5,  r5, r12
7    addu  r5,  r5, r4
8
9    # r4: fwd_ptr
10   # r5: rev_ptr
11
12 loop:
13   lw    r12, 0(r4)
14   lw    r13, 0(r5)
15   sw    r12, 0(r5)
16   sw    r13, 0(r4)
17   addiu r14, r5, 0
18   addiu r4,  r4, 4
19   addiu r5,  r5, -4
20   bne   r4,  r14, loop
21
22   jr    r31
```

Figure 6: Assembly for Reverse on Array Data Structure

Question 4.B Analyzing Performance of a Linked-List Data Structure (7 points)

Figure 7 on the next page shows C-code which implements the reverse operations for a linked-list data structure with an even number of elements. Figure 8 shows corresponding MIPS assembly code for the function. Recall that arguments are stored in r4 and r5. We need additional load instructions to retrieve the next and previous pointers for iterating through the list. Note that we use a non-zero offset when accessing these next and previous pointers since we know ahead of time where these fields are located relative to the value field.

Draw a pipeline diagram illustrating at least one iteration of the loop shown in Figure 8. You should draw as many iterations as you need in order to determine the steady-state behavior of the loop. Carefully consider which memory accesses hit or miss in the data cache. **Use your pipeline diagram to estimate the overall execution time in cycles for this operation. Fill in the corresponding row of Figure 4.** You will need to decompose the CPI into its various components based on what stalls and/or squashes occur in the execution. **You must show your work.**

```
void list_reverse( node* head, node* tail ) {

  # Code only works for even n
  assert( n % 2 == 0 )

  int* fwd_ptr = head;
  int* rev_ptr = tail;
  int* rev_ptr_last = 0;

  do {

    # Swap values at fwd and rev pointers
    int temp = fwd_ptr->value;
    fwd_ptr->value = rev_ptr->value;
    rev_ptr->value = temp;

    # Save rev pointer for exit condition check
    rev_ptr_last = rev_ptr;

    # Update fwd and rev pointers
    fwd_ptr = fwd_ptr->next;
    rev_ptr = rev_ptr->prev;

  }
  while ( fwd_ptr != rev_ptr_last );

}
```

Figure 7: C-Code for Reverse on Linked-List Data Structure

```
  # r4: fwd_ptr
  # r5: rev_ptr

loop:
  lw    r12, 0(r4)
  lw    r13, 0(r5)
  sw    r12, 0(r5)
  sw    r13, 0(r4)
  addiu r14, r5, 0
  lw    r4,  4(r4)
  lw    r5,  8(r5)
  bne   r4,  r14, loop

  jr    r31
```

Figure 8: Assembly for Reverse on Linked-List Data Structure

Question 4.C Comparison of Data Structures (6 points)

Compare the performance of the two data structures and generalize your results by answering the following questions.

- Which data structure performs better in this specific example and why?
- How would the execution time change as a function of cache line size assuming we always only allocate one linked-list node per cache line?
- How would the execution time change if we assumed larger cache lines and that the memory allocator organizes multiple linked-list nodes on the same cache line?
- How does the execution time change as the number of elements in the data structure grows asymptotically large? How does this relate to the theoretical asymptotic behavior of $O(n)$?
- How would the execution time change if both data structures were already present in the cache such that there were no cache misses?
- Can we draw any broad conclusions about the cache behavior of more regular array- or matrix-based data structures vs. more irregular list-, tree-, or graph-based data-structures that make extensive use of dynamic memory allocation and pointers?