Andrew ID: **SOLUTION**

Full Name:

Hint: This is an old school handwritten exam. There is no authenticated login. If we can't read your AndrewID, we won't easily know who should get credit for this exam. If we can't read either your AndrewID or Full Name, we're in real bind. Please write neatly :-)

18-213/18-613 Midterm Exam

Spring 2026

Instructions:

- Make sure that your exam is not missing any sheets (check page numbers at bottom)
- Write your Andrew ID and full name on this page (and we suggest on each and every page)
- This exam is closed book and closed notes.
- You may not use anything other than what we provide, except writing implements, such as pens and pencils, and a simple arithmetic calculator.
- Write your answers in the space provided for the problem.
- If you make a mess, clearly indicate your final answer.
- The exam has a maximum score of 100 points.
- The point value of each problem is indicated.
- **Good luck!**

Problem #	Scope	Max Points	Score
1	Data Representation: Ints	10	
2	Data Representation: Floats	10	
3	Data Representation: Structs, Unions, and Alignment	10	
4	Data Representation: Arrays	10	
5	Assembly: Simple	10	
6	Assembly: Loops, if-else, continue, break	10	
7	Assembly: Switch Statements	10	
8	Assembly: Calling Convention	10	
9	Caching	12	
10	Memory Hierarchy	8	
TOTAL	***Total points across all problems***	***100***	

1. Data Representation: Ints (10 points, 2 points per blank)

Fill in the five empty boxes in the table below when possible and indicate “UNABLE” when impossible.

	Machine 1: 6-bit w/2s complement signed	**Machine 2: 6-bit unsigned**
Binary representation of -10 decimal	**110110**	
Decimal value of 0x38	**-8**	**56**
Decimal value of (-24 - 10)	**30**	
Decimal representation of Tmin (Most negative)	**-32**	

Continued on next page.

2. Data Representation: Floats (10 points, 2 point per blank)

For this problem, please consider a floating point number representation based upon an IEEE-like floating point format as described below.

- Format:
  - The type is 7-bits wide
  - There is 1 sign bit.
  - There are k = 3 exponent bits.
  - You need to determine the number of fraction bits.

✓ Fill in the empty (and not grayed-out) boxes as instructed.
✓ When decimal values are requested, reduced fractions are okay.
✓ Should rounding be required, "round even"

	Format
Total Number of Bits (Decimal)	**7**
Number of Sign Bits (Decimal)	**1**
Number of Exponent Bits (Decimal)	**3**
Number of Fraction Bits (Decimal)	**3** (No points, just for you)
Bias (Decimal)	**3** (No points, just for you)
Largest number (Decimal value)	**15**
binary 1000101 (Decimal or Reduced Decimal Fraction)	**-5/32**
Smallest distance between any two adjacent points on the number line (Decimal)	**1/32**
Largest distance between any two adjacent points on the number line, excluding special values (Decimal)	**1**
There are multiple *positive* NaN bit patterns. For any given width, they all begin with the same prefix. **For this question's specific number format, list all bit patterns that can follow that prefix, but that do not represent a *positive* NaN.**	**000**

Continued on next page.

3. Structs, Unions, and Alignment (10 points)

For this question please assume "Natural alignment", in other words, please assume that each type must be aligned to a multiple of its data type size.

Please consider the following struct:

```
struct q3 {
  char c1;  // 1-byte type
  long l;   // 8-bytes type
  char c2;  // 1-byte type
  int i;
};
```

3(A)(4 point): What would you expect to be the value of the expression below?

```
sizeof(struct q3)
```

24 bytes

3(B)(3 points): Rewrite the struct above to minimize its size after alignment-mandated padding:

```
struct q3 {
  char c1;  // 1-byte type
  char c2;  // 1-byte type
  int i;    // 4-byte type
  long l;   // 8-bytes type
};
```

3(C)(3 points): How many bytes is your revised struct, also as reported by `sizeof()`?

16 bytes

Continued on next page.

4. Data Representation: Arrays (10 points)

Consider the following definition in an x86-64 system with 8-byte pointers and 4-byte `int`s:

Definition
`short numbers[3][2];`

4(A)(3 point): How many bytes are allocated to `numbers`? (Write "UNKNOWN" if not knowable). *Hint:* Think sizeof()

12 bytes

4(B) (4 point): If the address of `numbers[1][1]` is `0x20000`, what is the address of numbers[0][0]?

0x1FFFA

4(C)(3 points): Consider the original definition of the struct we provided in Question 3 (Structs) above, as well as the definition of `array4C` given below, what is the distance, measured in bytes, between the address of `array4C[2].c1` and the address of `array4C[4].i`?

```
struct q3 array4C[6];
```

68 Bytes

Continued on next page.

5. Assembly: Simple (10 points)

How many times is puts() called in the assembly below?

```
(gdb)
Dump of assembler code for function main:
   0x0000000000001149 <+0>:     endbr64
   0x000000000000114d <+4>:     push   %rbp
   0x000000000000114e <+5>:     mov    %rsp,%rbp
   0x0000000000001151 <+8>:     sub    $0x10,%rsp
   0x0000000000001155 <+12>:    movl   $0x80,-0x4(%rbp)
   0x000000000000115c <+19>:    jmp    0x1171 <main+40>
   0x000000000000115e <+21>:    sarl   $0x2,-0x4(%rbp) # Shift Arithmetic right
   0x0000000000001162 <+25>:    lea    0xe9b(%rip),%rax
   0x0000000000001169 <+32>:    mov    %rax,%rdi
   0x000000000000116c <+35>:    call   0x1050 <puts@plt>
   0x0000000000001171 <+40>:    cmpl   $0x0,-0x4(%rbp)
   0x0000000000001175 <+44>:    jg     0x115e <main+21>
   0x0000000000001177 <+46>:    nop
   0x0000000000001178 <+47>:    nop
   0x0000000000001179 <+48>:    leave
   0x000000000000117a <+49>:    ret
End of assembler dump.
```

4 times.

Continued on next page.

6. Assembly: Assembly: Loops, if-else, continue, break (10 points)

Consider the following code:

```
(gdb) disassemble loop
Dump of assembler code for function loop:
   0x0000000000001149 <+0>:     endbr64
   0x000000000000114d <+4>:     push   %rbp
   0x000000000000114e <+5>:     mov    %rsp,%rbp
   0x0000000000001151 <+8>:     mov    %edi,-0x14(%rbp)
   0x0000000000001154 <+11>:    mov    %esi,-0x18(%rbp)
   0x0000000000001157 <+14>:    movl   $0x0,-0x8(%rbp)
   0x000000000000115e <+21>:    mov    -0x18(%rbp),%eax
   0x0000000000001161 <+24>:    mov    %eax,-0x4(%rbp)
   0x0000000000001164 <+27>:    cmpl   $0x0,-0x14(%rbp)
   0x0000000000001168 <+31>:    jle    0x11c2 <loop+121>
   0x000000000000116a <+33>:    jmp    0x1193 <loop+74>
   0x000000000000116c <+35>:    mov    -0x18(%rbp),%eax
   0x000000000000116f <+38>:    mov    %eax,%edx
   0x0000000000001171 <+40>:    neg    %edx
   0x0000000000001173 <+42>:    cmovns %edx,%eax
   0x0000000000001176 <+45>:    mov    %eax,-0x18(%rbp)
   0x0000000000001179 <+48>:    jmp    0x1183 <loop+58>
   0x000000000000117b <+50>:    addl   $0x1,-0x8(%rbp)
   0x000000000000117f <+54>:    subl   $0x1,-0x18(%rbp)
   0x0000000000001183 <+58>:    cmpl   $0x0,-0x18(%rbp)
   0x0000000000001187 <+62>:    jg     0x117b <loop+50>
   0x0000000000001189 <+64>:    mov    -0x4(%rbp),%eax
   0x000000000000118c <+67>:    mov    %eax,-0x18(%rbp)
   0x000000000000118f <+70>:    subl   $0x1,-0x14(%rbp)
   0x0000000000001193 <+74>:    cmpl   $0x0,-0x14(%rbp)
   0x0000000000001197 <+78>:    jg     0x116c <loop+35>
   0x0000000000001199 <+80>:    jmp    0x11c8 <loop+127>
   0x000000000000119b <+82>:    mov    -0x18(%rbp),%eax
   0x000000000000119e <+85>:    mov    %eax,%edx
   0x00000000000011a0 <+87>:    neg    %edx
   0x00000000000011a2 <+89>:    cmovns %edx,%eax
   0x00000000000011a5 <+92>:    mov    %eax,-0x18(%rbp)
   0x00000000000011a8 <+95>:    jmp    0x11b2 <loop+105>
   0x00000000000011aa <+97>:    addl   $0x1,-0x8(%rbp)
   0x00000000000011ae <+101>:   subl   $0x1,-0x18(%rbp)
   0x00000000000011b2 <+105>:   cmpl   $0x0,-0x18(%rbp)
   0x00000000000011b6 <+109>:   jg     0x11aa <loop+97>
   0x00000000000011b8 <+111>:   mov    -0x4(%rbp),%eax
   0x00000000000011bb <+114>:   mov    %eax,-0x18(%rbp)
   0x00000000000011be <+117>:   addl   $0x1,-0x14(%rbp)
   0x00000000000011c2 <+121>:   cmpl   $0x0,-0x14(%rbp)
   0x00000000000011c6 <+125>:   js     0x119b <loop+82>
   0x00000000000011c8 <+127>:   mov    -0x8(%rbp),%eax
   0x00000000000011cb <+130>:   pop    %rbp
   0x00000000000011cc <+131>:   ret
```

Continued on next page.

6(A)(2 points): How many loops does this function have? How do you know?

4. There are 4 backward jumps: +62, +78, +109, and +125

6(B)(2 points): How many nested loops are there? How do you know?
Hint: When counting the number of nested loops note that outermost loops do not count as a nested loops.

2. There are two loop where the beginning and end of the loop are entirely within other loops, note +62 and +109.

6(C)(2 points): Are there any uses of the ?-operator? If so, how many and how do you know?

Yes. There are two (2) cmovs: +42 and +89

Continued on next page.

7. Assembly: Switch Statements (10 points)

Consider the following compiled from C Language code containing a switch statement and no if statements. Remember that the **jump table keeps offsets from its own start address. The address of each code block is the address of the beginning of the jump table plus the value of the code block's jump table entry.** You'll see this address before the relevant jump in the assembly. It might make things easier for you to note the address indicated by the lowest jump table entry and think of the other entries relative to that one.

```
(gdb) disassemble foo
Dump of assembler code for function foo:
   0x0000555555555169 <+0>:     endbr64
   0x000055555555516d <+4>:     push   %rbp
   0x000055555555516e <+5>:     mov    %rsp,%rbp
   0x0000555555555171 <+8>:     mov    %edi,-0x4(%rbp)
=> 0x0000555555555174 <+11>:    mov    %esi,-0x8(%rbp)
   0x0000555555555177 <+14>:    mov    -0x8(%rbp),%eax
   0x000055555555517a <+17>:    sub    $0x4,%eax
   0x000055555555517d <+20>:    cmp    $0x9,%eax
   0x0000555555555180 <+23>:    ja     0x5555555551e2 <foo+121>
   0x0000555555555182 <+25>:    mov    %eax,%eax
   0x0000555555555184 <+27>:    lea    0x0(,%rax,4),%rdx
   0x000055555555518c <+35>:    lea    0xe71(%rip),%rax        # 0x555555556004
   0x0000555555555193 <+42>:    mov    (%rdx,%rax,1),%eax
   0x0000555555555196 <+45>:    cltq
   0x0000555555555198 <+47>:    lea    0xe65(%rip),%rdx        # 0x555555556004
   0x000055555555519f <+54>:    add    %rdx,%rax
   0x00005555555551a2 <+57>:    notrack jmp *%rax
   0x00005555555551a5 <+60>:    addl   $0x2,-0x4(%rbp)
   0x00005555555551a9 <+64>:    jmp    0x5555555551e8 <foo+127>
   0x00005555555551ab <+66>:    mov    -0x4(%rbp),%eax
   0x00005555555551ae <+69>:    mov    %eax,%edx
   0x00005555555551b0 <+71>:    shr    $0x1f,%edx
   0x00005555555551b3 <+74>:    add    %edx,%eax
   0x00005555555551b5 <+76>:    sar    %eax
   0x00005555555551b7 <+78>:    mov    %eax,-0x4(%rbp)
   0x00005555555551ba <+81>:    jmp    0x5555555551e8 <foo+127>
   0x00005555555551bc <+83>:    addl   $0x1,-0x4(%rbp)
   0x00005555555551c0 <+87>:    jmp    0x5555555551e8 <foo+127>
   0x00005555555551c2 <+89>:    mov    -0x4(%rbp),%eax
   0x00005555555551c5 <+92>:    imul   -0x8(%rbp),%eax
   0x00005555555551c9 <+96>:    mov    %eax,-0x4(%rbp)
   0x00005555555551cc <+99>:    jmp    0x5555555551e8 <foo+127>
   0x00005555555551ce <+101>:   subl   $0x1,-0x4(%rbp)
   0x00005555555551d2 <+105>:   addl   $0x2,-0x4(%rbp)
   0x00005555555551d6 <+109>:   mov    -0x4(%rbp),%eax
   0x00005555555551d9 <+112>:   imul   -0x8(%rbp),%eax
   0x00005555555551dd <+116>:   mov    %eax,-0x4(%rbp)
   0x00005555555551e0 <+119>:   jmp    0x5555555551e8 <foo+127>
   0x00005555555551e2 <+121>:   mov    -0x8(%rbp),%eax          # Note that this is -3618
   0x00005555555551e5 <+124>:   sub    %eax,-0x4(%rbp)
   0x00005555555551e8 <+127>:   mov    -0x4(%rbp),%eax
   0x00005555555551eb <+130>:   pop    %rbp
   0x00005555555551ec <+131>:   ret
End of assembler dump.

(gdb) x/16d  0x555555556000
0x555555556000: 131073  -3679   -3679   -3673
0x555555556010: -3673   -3618   -3656   -3650
0x555555556020: -3638   -3634   -3630   680997
0x555555556030: 990059265       60      6       -4112
```

Continued on next page.

```
(gdb) x/16x 0x555555556000
0x555555556000 <_IO_stdin_used>:        0x00020001      0xffffff1a1      0xffffff1a1
0xffffff1a7
0x555555556010: 0xffffff1a7      0xffffff1de      0xffffff1b8      0xffffff1be
0x555555556020: 0xffffff1ca      0xffffff1ce      0xffffff1d2      0x000a6425
0x555555556030: 0x3b031b01       0x0000003c       0x00000006       0xffffeff0
```

7(A)(2 points): At what address does the jump table shown above begin? How do you know?

0x555555556004
See +35 and +47. That's the base of where it can jump

7(B)(3 points): What range of values are handled by the jump table? How do you know?

4-13. See +17 and +20. It adds so 4 is the 0th entry and the max entry, above which it jumps past the table to the default case, is, therefore, 9+4=13.

7(C)(2 points): Is there a default case? If so, at what address does it begin? How do you know?

Yes. See +23: ja 0x5555555551e2 <foo+121>

7(D)(3 points) Which case(s), if any, fall through to the next case <u>after executing some of their own code</u>? How do you know? *Hint*: Give the case number not the address.

Cases 11 falls to case 12, which in turn falls to case 13.
Case 11 starts 30 bytes before the default case, <+101>: subl $0x1,-0x4(%rbp)
Case 12 starts 4 bytes after that – and there is no jump out before then.
Case 13 starts 4 bytes after that – and there is no jump out before then.
The jump out is after case 13: <+119>: jmp 0x5555555551e8 <foo+127>

Continued on next page.

8. Assembly: Calling Convention (10 points)

8(A)(5 points): Consider the following code, taking note of the comment.

```
void fun(int a1, int b2, int c3, int d4, int e5, int f6, int g7) {
  // Recall: Only the 1st 6 arguments are passed in registers
  // Recall: Pointers/Addresses are 8-bytes
  // Note: &g7 = 0x7fffffffe770
  unsigned char buffer[2048];

  printf ("%d %d %d %d %d %d %d\n", a1, b2, c3, d4, e5, f6, g7);

}
```

At what address will this functions return address be found? How do you know?

0x7fffffffe768. It comes just above g7. But, since the stack needs to be aligned, it is 8 bytes before.

8(B)(5 points): Why does the argument build area come above (higher address) than the return address on the stack? Please don't explain that the stack grows downward. Instead, please focus on the order in which these are pushed onto the stack.

***ret* needs to be a simple, consistent operation implemented in hardware. By ensuring that the return address is the last thing pushed onto the stack, *ret* can do a simple pop and jump. If the order were reversed, how would *ret* know how much to pop off of the stack to get to the return address? Maybe the caller could clean it up before the return, but, technically speaking, it doesn't even know what the callee put there, just what it used.**

Continued on next page.

9. Caching (12 points)
Part A: Cache Configuration (4 points)

Given a model described as follows:

- Associativity: 2-way set associative
- Sets: 4
- Total size: 128 bytes (not counting meta data)
- Replacement policy: Set-wise LRU
- 8-bit addresses

9(A)(1) (1 point) How many bits for the set index?

2
4 sets: 00, 01, 10, 11

9(A)(2) (1 point) How many bytes are in a block?

16 bytes = (128 bytes total) / (4 sets) / (2 blocks/set)

9(A)(3) (1 point) How many bits for the block offset?

4 bits
Log_2(16 bits) = 4 bits

9(A)(4) (1 point) How many bits for the tag?

(8 bits total) – (2 bits set index) – (4 bits offset) = 2 bit tags

Continued on next page.

9(B) Cache Trace (8 points, 1 point each line, ½ point each entry):

Below is an address trace giving a sequence of memory accesses. The trace begins at the beginning of time, e.g. the cache is empty.

For each of the following addresses, please indicate if it hits, or misses.

If it misses, further categorize it a capacity miss, a conflict miss, or a compulsory (cold) miss, or note whether the miss results in allocating an unused entry or replacing/evicting a valid entry, as indicated within the table.

If the address replaces another **valid** entry in the cache, please indicated which address it replaced in the designated column.

Mark your answers in the table. Addresses should be in hex, just as we presented them to you.

Address	**Circle one (per row):**		**Circle one (per row):**				**Replaced address, if applicable**
0x00							
0x10							
0x00	**Hit**	Miss	Capacity	Compulsory	Conflict	**N/A**	
0x40	Hit	**Miss**	**Allocate**	Replace		N/A	
0x10	**Hit**	Miss	Capacity	Compulsory	Conflict	**N/A**	
0x80	Hit	**Miss**	Allocate	**Replace**		N/A	**0x00**
0x00	Hit	**Miss**	Capacity	Compulsory	**Conflict**	N/A	**0x40**
0x40	Hit	**Miss**	Allocate	**Replace**		N/A	**0x80**
0x80	Hit	**Miss**	Capacity	Compulsory	**Conflict**	N/A	**0x00**
0x10	**Hit**	Miss	Capacity	Compulsory	Conflict	**N/A**	

Continued on next page.

10. Memory Hierarchy (8 points)

10(A) (4 points) The Shark machines, like many servers and workstations, incorporate a 3-level caching scheme. For each cache, please explain if it is segregated or unified, why, and its unique role in the hierarchy (4 points).

The level 1 and level 2 caches are per-core caches. The level 1 cache is usually segregated to ensure that there is at least a certain amount of both likely useful instructions and likely useful data ready to flow into the processor. The level 2 cache is generally unified to ensure flexibility, since it isn't knowable at the time the hardware is made what the instruction density vs data density will be, e.g. the size of the working set of each, so the partitioning needs to be flexible. The unified cache provides this.

The level three cache is usually per processor, across cores. It provides a way for data and instructions to be shared across cores, e.g. by threads or as scheduling migrates work.

10(B) (4 points) In multicore systems, it is common for per-core caches, e.g. Level-1 and Level-2, to be write-through and the system-wide caches, e.g. Level-3 cache, to be write-back. Why?

Write-back caches do a better job of shielding the memory beneath them from repeated writes, e.g. a loop control variable. But, in doing so, they also don't push the updated value down to a shared level of memory. In an environment with multiple cores, this means that another core can load a stale value from a lower level of memory, or fail to have its now-stale value updated.

Write-through caches both updated the lower-level memory and also generate bus traffic that can be used for invalidating stale entries in peer caches (and communicating present residency).

For correctness, we often use write-through caches at layers where peer caches exist to support snoopy protocols, etc, and write-back caches for lower-level shared caches, where that isn't needed, so we can shield the lower level storage layers from repeated writes.

All done! Great work! See you next week!