Andrew ID:

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

Fall 2025

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 and Effective Memory Access Time	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: 4-bit w/2s complement signed	**Machine 2: 4-bit unsigned**
Binary representation of 10 decimal	UNABLE	1010
Decimal value of 1010	-6	
Decimal value of (-3 - 8)	5	
Decimal representation of Umax		15

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 8-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)	**8**
Number of Sign Bits (Decimal)	**1**
Number of Exponent Bits (Decimal)	**3**
Number of Fraction Bits (Decimal)	**4** (No points, just for you)
Bias (Decimal)	**3** (No points, just for you)
Largest number (Decimal value)	**15 ½**
binary 10000110 (Decimal)	**-3/32**
Smallest distance between any two adjacent points on the number line (Decimal)	**1/64**
Largest distance between any two adjacent points on the number line, excluding special values (Decimal)	**1/2**
Infinity (Binary)	**01110000**

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 {
  char c;        // 1-byte type
  double d;      // 8-byte type
  short s[3];    // 2-byte type
  int i[3];      // ints are a 4-byte type
} partB;
```

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

```
sizeof(struct partB)
```

40 Bytes

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

```
struct {
  char c;        // 1-byte type
  short s[3];    // 2-byte type
  int i[3];      // ints are a 4-byte type
  double d;      // 8-byte type
} partB;
```

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

32 Bytes

Continued on next page.

4. Data Representation: Arrays (10 points)
Part A: Array Size and Layout (5 points)

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

Definition
`int numbers[2][5];`

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

40 bytes

4(B) (4 point): If the address of `numbers[0][1]` is `0x10000`, what is the address of numbers[1][4]?

0x10020

4(C)(3 points): Consider the original definition of the struct we provided in Question 3 (Structs) above, as well as the definition of `array3D` given below, what is the distance, measured in bytes, between the address of `array3D[3].c` and the address of `array3D[4].s[1]`?

```
struct partB array3D[10];
```

58 Bytes

Continued on next page.

5. Assembly: Simple (10 points)

Write out the simplest possible mathematical expression in terms of *n* that captures the arithmetic performed by the code sample below:

```
    movl    %edi, %eax          # eax = n
    shl     $3, %eax
    leal    (%rax,%rdi), %eax
```

9n

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

Consider the following code:

```
loop:
        pushq   %r12
        leaq    .LC0(%rip), %r12
        pushq   %rbp
        movl    %esi, %ebp
        pushq   %rbx
        movl    %edi, %ebx
        cmpl    %esi, %edi
.L0:
        jg      .L2
        jge     .L1
.L3:
        movl    %ebx, %edx
        movq    %r12, %rsi
        movl    $1, %edi
        xorl    %eax, %eax
        call    __printf_chk@PLT
        addl    $1, %ebx
        cmpl    %ebx, %ebp
        jne     .L3
.L1:
        popq    %rbx
        popq    %rbp
        popq    %r12
        ret
.L2:
        movl    %ebp, %edx
        movq    %r12, %rsi
        movl    $1, %edi
        xorl    %eax, %eax
        call    __printf_chk@PLT
        addl    $1, %ebp
        cmpl    %ebp, %ebx
        jne     .L2
        jmp     .L1
```

Continued on next page.

6(A)(2 points): How many arguments does this function receive (and use)? How do you know?
Hint: The first 6 arguments are passed via registers %rdi, %rsi, %rdx, %rcx, %r8, and %r9, in that order, and %rax is used for return values.

%rdi and %rsi are the 1st and 2nd arguments, respectively, and are used without otherwise being given values.

6(B)(2 points): Does this function return a value? How do you know?
Hint: The first 6 arguments are passed via registers %rdi, %rsi, %rdx, %rcx, %r8, and %r9, in that order, and %rax is used for return values.

The function does not return a value, since %rax is not used to hold a return result before ret.

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

Two. There are two backward jumps.

6(D)(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.

None. No loop is contained within the body of any other loop.

6(E)(2 points): Are there any conditionals that do not function to determine whether or not a loop should continue or stop? If so, what does it, or they, do? If not, how do you know?

One. The two jumps at .L0 determine exactly which one of two loops should be run, or if neither loop should run at all.

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:
   0x0000000000001210 <+0>:     endbr64                          # 0x1210 is 4624 decimal
   0x0000000000001214 <+4>:     lea    -0x4(%rsi),%eax
   0x0000000000001217 <+7>:     cmp    $0x6,%eax
   0x000000000000121a <+10>:    ja     0x1260 <foo+80>
   0x000000000000121c <+12>:    lea    0xde1(%rip),%rdx    # %rdx = 0x2004 (8196 decimal)
   0x0000000000001223 <+19>:    movslq (%rdx,%rax,4),%rax
   0x0000000000001227 <+23>:    add    %rdx,%rax
   0x000000000000122a <+26>:    notrack jmp *%rax
   0x000000000000122d <+29>:    nopl   (%rax)
   0x0000000000001230 <+32>:    sub    $0x4,%edi
   0x0000000000001233 <+35>:    lea    (%rdi,%rdi,2),%eax
   0x0000000000001236 <+38>:    ret
   0x0000000000001237 <+39>:    nopw   0x0(%rax,%rax,1)
   0x0000000000001240 <+48>:    mov    %edi,%eax
   0x0000000000001242 <+50>:    shr    $0x1f,%eax
   0x0000000000001245 <+53>:    add    %eax,%edi
   0x0000000000001247 <+55>:    mov    %edi,%eax
   0x0000000000001249 <+57>:    sar    %eax
   0x000000000000124b <+59>:    ret
   0x000000000000124c <+60>:    nopl   0x0(%rax)
   0x0000000000001250 <+64>:    lea    0x1(%rdi),%eax
   0x0000000000001253 <+67>:    ret
   0x0000000000001254 <+68>:    nopl   0x0(%rax)
   0x0000000000001258 <+72>:    lea    (%rdi,%rdi,4),%edi
   0x000000000000125b <+75>:    lea    (%rdi,%rdi,1),%eax
   0x000000000000125e <+78>:    ret
   0x000000000000125f <+79>:    nop
   0x0000000000001260 <+80>:    mov    %edi,%eax
   0x0000000000001262 <+82>:    sub    %esi,%eax
   0x0000000000001264 <+84>:    ret                              # 0x1264 is 4708 decimal
End of assembler dump.


(gdb) x/20xgw 0x2000
0x2000: 0x00020001      0xfffff22c      0xfffff22f      0xfffff23c
0x2010: 0xfffff23c      0xfffff25c      0xfffff24c      0xfffff254
0x2020: 0xfffff0af      0xfffff0b3      0xfffff0d3      0xfffff0d3
0x2030: 0xfffff0ec      0xfffff0e0      0xfffff0e6      0x000a6425
0x2040: 0x3b031b01      0x0000003c      0x00000006      0xfffefe0

(gdb) x/20dgw 0x2000
0x2000: 131073  -3540   -3537   -3524
0x2010: -3524   -3492   -3508   -3500
0x2020: -3921   -3917   -3885   -3885
0x2030: -3860   -3872   -3866   680997
0x2040: 990059265       60      6       -4128
```

Continued on next page.

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

0x2004. It is the address used as the base in <+12> - <+26>. It is given in the comment.

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

4 – 10. Note the offset of -4 in the leal @ <+4>, which shifts 4 to the 0th entry of the table. Notice the compare and ja at <+7> and <+10> that go directly to the default case for values greater than 10. The comparison is against 6 – but that is after the offset by -4 is already made.

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

Yes. There is a guard jump at <+7> and <+10> to 0x1260 <foo+80>, which is also where case 8 in the jump table gets us.

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

Case 4 starts at 0x1230. Case 5 starts 3 bytes from that at 0x1233. The 1st return case 4 encounters is at 0x1236, which is after case 5 starts at 0x1233.

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]; // 0x7fffffffdf50

  gets(buffer);

}
```

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

0x7fffffffe768. Since pointers are 8 bytes, the return address will be 8-bytes above the 7th argument in the stack.

8(B)(5 points): Consider (a) the choice of general purpose registers for arguments and the return value, and (b) the direction of the stack movement, e.g. growing up vs growing down. Which of these is configured by the x86-64 instruction set architecture (ISA), and which of these is purely convention? How can you tell by looking at the ISA?

The choice among the general-purpose registers is purely convention. No instruction within the ISA assumes anything about this.

The direction of the stack growth is set by the ISA because call pushes arguments and stack pops them. One could certainly work around this, but that wouldn't be the design intent and would require more instructions for the same effect.

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: 2
- Total size: 64 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?

1 bits are required to index into 2 sets: 0 and 1.

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

64B / 2 Sets / 2 Blocks/Set = 16B/Block

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

4 bits.
16B/Block (from above)
4 bits are required to index 16B

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

3 Bits
8 bits (address) – 1 bit (set index) – 4 bits (block offset) = 3 bits (tag)

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. Mark your answers in the table.

Address	**Circle one (per row):**	**Circle one (per row):**
0x30		
0X5A		
0X3F	**Hit** Miss	Capacity Compulsory Conflict **N/A**
0X65	Hit **Miss**	**Allocate** Replace N/A
0X8F	Hit **Miss**	Capacity **Compulsory** Conflict N/A
0XA0	Hit **Miss**	Allocate **Replace** N/A
0X65	Hit **Miss**	Capacity Compulsory **Conflict** N/A
0XBF	Hit **Miss**	Allocate **Replace** N/A
0x31	**Hit** Miss	Capacity Compulsory Conflict **N/A**
0x5A	Hit **Miss**	**Capacity** Compulsory Conflict N/A

10. Memory Hierarchy and Effective Access time (8 points)

Imagine an embedded system with a FLASH-based main memory layered beneath anSRAM cache.

- The SRAM cache has a 1nS access time.
- The FLASH main memory has a 51ns access time.

Your goal is to design a system with a system with a memory access time of 11nS, or better. What is the maximum SRAM cache miss rate that can be tolerated? Round up to a whole number.

FOR SIMPLICITY, AVOID COMPLEX CALCULATION AND LEAVE YOUR ANSWER AS A SIMPLE FRACTION

What is the maximum acceptable miss rate to achieve a system performance of 25nS?

MISS_RATE = 20%

11ns = 1 ns + MISS_RATE*(51ns – 1ns)
11ns = 1n + MISS_RATE*50ns
10ns = MISS_RATE*50ns
10ns/50ns = MISS_RATE
0.2 = MISS_RATE
MISS_RATE = 0.2

All done! Great work! See you next week!