CMPU-224 Exam 1 Practice
Spring 2026

1. Fill in the blanks in the following table. Assume all numbers are **unsigned**. Do not put any leading zeros in your answers.

Decimal	**Hexadecimal**	**Binary**
235		
	0x11d	
		10011000
51		
	0xc4	
		1100011

2. You are given the following dump of memory below. The format of the memory dump is `<address>:<value>`. For example, the byte at memory address `0x425` is `0x94` and the byte at memory address `0x43a` is `0x04`.

```
420:47 421:2f 422:80 423:55 424:93 425:94 426:eb 427:7c 428:ad 429:fc 42a:1b 42b:af 42c:30 42d:ef 42e:56 42f:92
430:13 431:51 432:f2 433:26 434:c9 435:c2 436:57 437:68 438:4a 439:6d 43a:04 43b:ab 43c:31 43d:a3 43e:9a 43f:11
440:78 441:f0 442:fe 443:fd 444:76 445:4c 446:6e 447:43 448:6a 449:fa 44a:a1 44b:5e 44c:60 44d:b6 44e:61 44f:7d
450:0e 451:f9 452:52 453:0f 454:3d 455:3e 456:a5 457:19 458:bd 459:b4 45a:34 45b:ee 45c:a6 45d:3c 45e:f7 45f:e0
460:6b 461:67 462:d4 463:89 464:21 465:63 466:83 467:f3 468:65 469:87 46a:e1 46b:86 46c:50 46d:8d 46e:e3 46f:d2
470:46 471:e6 472:24 473:bf 474:2d 475:f5 476:53 477:6c 478:f1 479:e2 47a:b8 47b:a4 47c:fb 47d:cb 47e:0a 47f:de
480:4e 481:9c 482:07 483:01 484:72 485:d1 486:a2 487:c6 488:02 489:12 48a:b1 48b:69 48c:d3 48d:5b 48e:cf 48f:10
490:bb 491:22 492:98 493:f6 494:4d 495:d0 496:a0 497:5c 498:c1 499:36 49a:cc 49b:1c 49c:25 49d:5d 49e:5a 49f:8e
4a0:44 4a1:d6 4a2:79 4a3:90 4a4:70 4a5:ca 4a6:03 4a7:7b 4a8:85 4a9:dc 4aa:c0 4ab:dd 4ac:29 4ad:58 4ae:74 4af:1d
4b0:b3 4b1:84 4b2:d5 4b3:ae 4b4:33 4b5:aa 4b6:09 4b7:ea 4b8:e7 4b9:b5 4ba:8b 4bb:62 4bc:9e 4bd:20 4be:75 4bf:7a
```

What are the values of the following expressions below? **Give your answers in hexadecimal**.

Expression	**Value**
Little-endian **`char`** at address **`0x447`**	0x
Big-endian **`char`** at address **`0x438`**	0x
Little-endian **`short`** at address **`0x46c`**	0x
Big-endian **`short`** at address **`0x468`**	0x
Little-endian **`int`** at address **`0x480`**	0x
Big-endian **`int`** at address **`0x47c`**	0x

3. Short answer and multiple choice

(a) Given the **base-5** number 124_5, what is value of that number in a decimal (**base-10**) representation?

_____

(b) Given the decimal (base-10) number 47_{10}, what is value of that number in a (**base-3**) representation?

_____

(c) You have a the following **12-bit** two's complement number: `0xA42`

What is the value of the above number converted to a **16-bit** two's complement number. Give your answer in **hexadecimal**.

0x_____

(d) You have the following C code:

```
signed short a = 0xC287;
signed char b;
b = a;
```

What is the value of `b`? Give your answer in **decimal**. _____

(e) You have the following C code:

```
signed char a = 0xF8;
signed char b = 0x21;
signed char c = a + b;
```

What is the value of `c`? Give your answer in **decimal**. _____

(f) You have the following C code:

```
y = x * 24;
```

Fill in the blanks below with the **decimal** numbers that give an equivalent statement to the one above.

y = (x << _____) + (x << _____);

(g) Which of the following decimal numbers are **not** able to be represented as a **8-bit** two's complement number? **Select all that apply**.

- ◯ -0
- ◯ 0
- ◯ -128
- ◯ 128

(h) When sign-extending a two's complement number, what is the correct procedure? **Select one**.

- ◯ Flip the bits and add 1.
- ◯ Fill the new bits on the left with copies of the most significant bit (MSB)
- ◯ Fill the new bits on the left with zeros.
- ◯ Fill the new bits on the left with ones.

(i) Under which condition is overflow **not** possible when adding two n-bit two's complement numbers? **Select one**.

- ◯ When adding two negative numbers.
- ◯ When adding two positive numbers.
- ◯ When there is a carry-out bit from the MSB.
- ◯ When adding a positive number and a negative number.

4. Assume we are running code on a **12-bit** machine. All values in this system are represented as 12-bit two's complement numbers. `TMax` is the largest (most positive) valued integer represented in this system and `TMin` represents the smallest (most negative) valued integer in the system. **For each of the expressions given below, write out the corresponding bit representation as a three digit hexadecimal number.** For example, if the binary representation for a given expression is `000000000001`, your answer would be `0x001`.

Expression	**Hexadecimal Representation**
`TMin`	
`TMax`	
`-0xA53`	
`~0xA53`	
`!0xA53`	
`0xA53 << 4`	
`0xA53 >> 4`	
`0xA53 ^ 0x0F0`	
`0xA53 & 0x0F0`	
`0xA53 | 0x0F0`	

5. Recall that floating point numbers are represented in the form $V = (-1)^s * M * 2^E$ where M is encoded in `frac`, and E is encoded in `exp`. Assume we have the following tiny **12-bit** floating point representation where bits 0-6 represent the `frac` field (7 bits), bits 7-10 represent the `exp` field (4 bits), and bit 11 represents the sign bit (1 bit).

   (a) What is the value of negative infinity ($-\infty$) in the floating point system described above?

   Give your answer in hexadecimal. 0x_____

   (b) In this system, which of the following is a representation of NaN? (select one).

   - ◯ `0x70F`
   - ◯ `0x7F0`
   - ◯ `0x780`
   - ◯ `0x078`

   (c) In the above system, which of the following is a representation of a positive denormalized number that is greater than zero? (select one)

   - ◯ `0x000`
   - ◯ `0x804`
   - ◯ `0x080`
   - ◯ `0x040`

(d) What is the value of decimal fraction `36.2` in the floating point system described above?

Give your answer in hexadecimal. 0x_____

(e) You are given the following hexadecimal number `0xF4E` in the floating point system described above. What is the decimal representation of that number? Give your answer as a decimal number, representing the fractional part of the number (if it has one) as a fraction (e.g., 3/4).

Give your answer as a **decimal fraction**. _____

6. For this problem, you are given an unsigned binary fractional number. Round each fraction to binary integer (whole number) using the "round to even" rounding mode. **Give your answer in binary**.

Binary Fraction	**Rounded Binary**
`101.100`	
`110.100`	
`111.010`	
`110.101`	
`100.100`	
`101.100`	

7. Fill in the following table showing the effects of the following instructions in terms of the value in the destination register. The answer in the "`Value in a0`" column will be the hexadecimal value of the `a0` register. Assume each instruction is independent of the others (i.e., all the registers have the values in the table below at the start of every question). Give your answer as **eight hexadecimal digits, two digits per box**. For example, if the value in the `a0` register is the decimal integer `11`, your answer would be 00 00 00 0B, with the least significant byte being the rightmost one, and the most significant byte being the leftmost one.

Register	Value
a1	0xFB
a2	0xDF
a3	0x59E6
a4	0x3489
a5	0x39E516FC
a6	0xACB1E780

Instruction	Value in a0
`addi a0, a1, 5`	
`and a0, a2, a6`	
`or a0, a1, a5`	
`xor a0, a2, a1`	
`srli a0, a5, 8`	
`srai a0, a6, 4`	
`lui a0, 0x4582`	
`li a0, 0x39D5`	
`mv a0, a4`	
`not a0, a2`	
`neg a0, a2`	
`slli a0, a3, 0xc`	

8. Consider the following C program and its output:

```
#include <stdio.h>

struct question {
  int   a;
  char  b;
  short c;
  char *d;
};

int main(void) {
  struct question x;
  char *y;

  // Fill in our struct
  x.a = 0x498D94EF;
  x.b = 'V';
  x.c = 0xAC54;
  x.d = &(x.b);

  printf("Struct x is at memory address %p.\n", &x);
  return 0;
}
Struct x is at memory address 0x2b2aaf00.
```

(a) What is the total size of the structure in bytes? _____

(b) Assume you have the following C code utilizing the above structure:

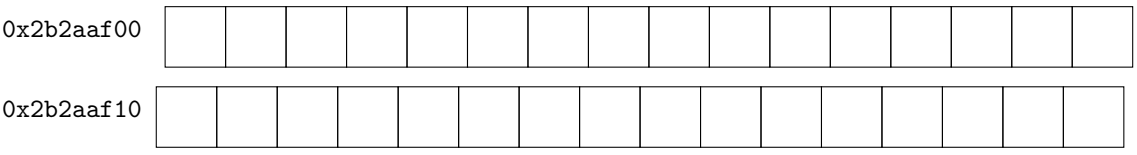
```
char* getPtr(struct question* q) {
  return q->d;
}
```

Fill in the missing blank below for the assembly language implementation of the above function.

```
getPtr:

    _____
    ret
```

(c) We see from running the program that `struct x` resides at memory address `0x2b2aaf00`. Show the memory representation of the struct below in hexadecimal. Put one pair hexadecimal digits (representing one byte) in each box. If a byte of memory is used as padding for the struct, put a single `X` for that box. If a byte memory is not used for the struct, leave that box blank (you may have blank boxes).



9. You are given the following RISC-V assembly and C code. Fill in the missing bits of the C program. **Note**: each blank should be either a single number, variable, or operation.

RISC-V assembly	**C code**

question:
101a8: mv a5,a0
101ac: li a0,0
101b0: j 101bc <question+0x14>
101b4: addi a5,a5,4
101b8: addi a1,a1,-1
101bc: blez a1,101d4 <question+0x2c>
101c0: lw a4,0(a5)
101c4: li a3,23
101c8: blt a3,a4,101b4 <question+0xc>
101cc: add a0,a0,a4
101d0: j 101b4 <question+0xc>
101d4: ret
	int question(int a[], int len) {
	
	int sum = 0;
	
	int val;
	
	while (_____ > _____) {
	
	_____ = *a;
	
	if (_____ < _____) {
	
	sum = _____ _____ _____;
	
	}
	
	a = a _____ _____ ;
	
	len = _____ - _____;
	
	}
	
	return sum;
	
	}

10. The following C program and its assembly language version are shown below. Fill in the missing blanks in the assembly language output.

C code

```
// node structure for a singly linked list
struct Node {
  int data;
  struct Node* next;
};

// Searches for a target value in a linked list
// head - pointer to the first node of the list
// target - integer value to search for
// return 1 if the target is found, 0 otherwise

int search_list(struct Node* head, int target) {
  // Start at the beginning of the list
  struct Node* current = head;

  // Traverse the list until we reach the end
  while (current != NULL) {
    if (current->data == target) {
      return 1; // Match found!
    }
    // Move to the next node
    current = current->next;
  }

  return 0; // Match not found.
}
```

RISC-V assembly

```
search_list:

  101a8:  j    101b0

  101ac:  lw   a0, ________(a0)

  101b0:  ________ a0, 101c0

  101b4:  lw   a5, 0(a0)

  101b8:  ________ a5, a1, 101ac

  101bc:  li   a0, ________

  101c0:  ret
```

11. You have the following C and assembly code.

C code	RISC-V assembly

`void getstr(int len, char *buf){`
`  int c = getchar();`
`  len--; // account for '\0'`
`  while (c != EOF && c != '\n' && len > 0) {`
`  *buf = c;`
`  buf++;`
`  c = getchar();`
`  len--;`
`  }`
`  *buf = '\0';`
`}`
`void target() {`
`  printf("You called the target!\n");`
`  exit(1);`
`}`
`void get_input(void) {`
`  char buf[20];`
`  getstr(200, buf); // should have been 20`
`}`
	`target:`
	`  10208:  addi sp,sp,-16`
	`  1020c:  sw  ra,12(sp)`
	`  10210:  lui  a0,0x14`
	`  10214:  addi a0,a0,-1532 # 13a04`
	`  10218:  jal  10774 <puts>`
	`  1021c:  li  a0,1`
	`  10220:  jal  100b4 <exit>`
	
	`get_input:`
	`  10224:  addi sp,sp,-48`
	`  10228:  sw  ra,44(sp)`
	`  1022c:  addi a1,sp,12`
	`  10230:  li  a0,200`
	`  10234:  jal  101a8 <getstr>`
	`  10238:  lw  ra,44(sp)`
	`  1023c:  addi sp,sp,48`
	`  10240:  ret`

If you look at the above code, you see that `getstr` now takes a length argument as it's first parameter, which it checks before writing the buffer. Unfortunately, in this example, the call to `get_input` has a typo, and the length was set to 200 instead of 20. This means that this code is vulnerable to buffer overflow attacks. Create an exploit string that causes the call to `get_input()` to return to `target()` instead of main. Give your answer as pairs of hexadecimal characters ready to be read by the `hex2raw` program (two hexadecimal characters per box and 4 bytes per line). If a byte is used for padding and its value is not important to the exploit, put an `X` for that box instead of a pair of hexadecimal characters. Leave a box blank if it is not needed your exploit.
