

# CMPU-224 Exam 1 Solutions

## Spring 2026

Name: \_\_\_\_\_

This is a closed book, closed notes quiz. No electronic devices are allowed. You have 75 minutes to complete the quiz. There are a total of 13 questions and 100 points. This quiz has 14 pages. Make sure you have all 14 pages before starting the quiz.

There should be enough space on the quiz for your answers. If you need more space to work out a problem, blank paper will be available, just ask.

**Good Luck!**

1. (6 points) 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 0x6f5 is 0x61 and the byte at memory address 0x70a is 0xc9.

```
6f0:81 6f1:d5 6f2:14 6f3:2f 6f4:e5 6f5:61 6f6:0c 6f7:86 6f8:18 6f9:74 6fa:49 6fb:08 6fc:57 6fd:f0 6fe:c3 6ff:91
700:e3 701:55 702:32 703:bb 704:ae 705:62 706:3a 707:69 708:f9 709:71 70a:c9 70b:9f 70c:ab 70d:01 70e:97 70f:cb
710:a9 711:8d 712:9b 713:3e 714:fb 715:29 716:87 717:21 718:7b 719:35 71a:43 71b:2e 71c:42 71d:f3 71e:17 71f:39
720:84 721:9a 722:ce 723:9c 724:67 725:3f 726:16 727:4f 728:56 729:44 72a:ec 72b:d9 72c:e6 72d:92 72e:eb 72f:8c
730:a4 731:6c 732:23 733:d6 734:65 735:26 736:94 737:b0 738:ca 739:68 73a:a6 73b:1a 73c:ea 73d:53 73e:25 73f:f6
740:59 741:f4 742:96 743:47 744:6f 745:1f 746:a0 747:b5 748:be 749:64 74a:bf 74b:c6 74c:40 74d:a3 74e:63 74f:77
750:b2 751:dc 752:8b 753:0b 754:2a 755:78 756:0a 757:dd 758:5d 759:46 75a:50 75b:fd 75c:a8 75d:11 75e:d7 75f:7a
760:7e 761:ad 762:cd 763:02 764:58 765:2b 766:2d 767:4d 768:b6 769:73 76a:98 76b:de 76c:b3 76d:88 76e:1e 76f:09
770:3b 771:d0 772:85 773:4b 774:31 775:24 776:a1 777:f5 778:f2 779:d3 77a:a7 77b:fc 77c:28 77d:db 77e:04 77f:cf
780:2c 781:76 782:c5 783:05 784:c7 785:6a 786:7f 787:af 788:3d 789:5f 78a:a5 78b:60 78c:0e 78d:e4 78e:8a 78f:75
```

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

Expression	Value
Little-endian <b>char</b> at address <b>0x723</b>	0x9c
Big-endian <b>char</b> at address <b>0x715</b>	0x29
Little-endian <b>short</b> at address <b>0x738</b>	0x68ca
Big-endian <b>short</b> at address <b>0x72e</b>	0xeb8c
Little-endian <b>int</b> at address <b>0x740</b>	0x4796f459
Big-endian <b>int</b> at address <b>0x73c</b>	0xea5325f6

2. (3 points) Convert the base-4 number  $213_4$  to base-10: 39<sub>10</sub>

**Solution:** To convert  $213_4$  to base-10, we expand the number using powers of 4:

$$213_4 = (2 \times 4^2) + (1 \times 4^1) + (3 \times 4^0)$$

$$213_4 = (2 \times 16) + (1 \times 4) + (3 \times 1)$$

$$213_4 = 32 + 4 + 3$$

$$213_4 = 39_{10}$$

3. Convert the following 8-bit two's complement numbers to 16-bit two's complement numbers. Give your answers in hexadecimal.

- (a) (3 points) 0xB4: 0xFFB4

**Solution:** The most significant bit of 0xB4 ( $10110100_2$ ) is 1 (negative). To extend it to 16 bits, we sign-extend by padding the upper 8 bits with 1s.

$$111111110110100_2 = 0xFFB4$$

(b) (3 points) 0x3A: 0x003A

**Solution:** The most significant bit of 0x3A ( $00111010_2$ ) is 0 (positive). To extend it to 16 bits, we pad the upper 8 bits with 0s.

$$000000000111010_2 = 0x003A$$

4. What are the smallest (most negative) and largest (most positive) 15-bit two's complement numbers? Give your answers in **hexadecimal**.

(a) (3 points) Smallest 15-bit number: 0x4000

**Solution:** The smallest (most negative) two's complement number has the sign bit set to 1 and all other bits set to 0. For a 15-bit number, this is:

$$100\ 0000\ 0000\ 0000_2 = 0x4000$$

(This corresponds to the decimal value  $-16384$ ).

(b) (3 points) Largest 15-bit number: 0x3FFF

**Solution:** The largest (most positive) two's complement number has the sign bit set to 0 and all other bits set to 1. For a 15-bit number, this is:

$$011\ 1111\ 1111\ 1111_2 = 0x3FFF$$

(This corresponds to the decimal value 16383).

5. (8 points) Write the following function, `bool_bits()`.

```
/*
 * bool_bits - return 1 if any of the bits of byte 2 (bits 16 to 23)
 * of the input are one. Bits are numbered from 0 (least significant) to 31
 * (most significant). The value of the other bits do not matter for this function.
 *
 * Examples: bool_bits(0x00ff0000) = 1
 *           bool_bits(0xff00ffff) = 0
 *           bool_bits(0x00100000) = 1
 */
```

Legal operations:

! ~ & ^ | + << >>

and any 1-byte (0 to 255) numeric constants.

**Solution:**

```
int bool_bits(int x) {
    int result = x >> 16;    // shift byte3 to the LSB
    result = result & 0xff;  // mask off upper byte
    return !!result;        // check for any ones
}
```

6. 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) (3 points) What is the value of positive infinity ( $+\infty$ ) in the floating point system described above?

Give your answer in hexadecimal. 0x\_\_\_\_\_ **780** \_\_\_\_\_

**Solution:** The correct answer has a zero in the msb (sign bit), all ones in the exp field, and all zeros in the frac field.

(b) (3 points) In this system, which of the following is a representation of a negative NaN? (select one).

- 0x780
- 0xF80
- 0xF81
- 0x081

**Solution:** In this system, a NaN has an exponent field with all ones, and a fractional field that is non zero. Since we want a negative NaN, the sign bit must be 1.

(c) (3 points) In the above system, which of the following is a representation of a negative denormalized number? (select one)

- 0x900
- 0x002
- 0x882
- 0x802

**Solution:** The correct answer has a 1 in the sign bit, an exponent field of all zeros.

(d) (3 points) What is the value of decimal fraction 13.4 in the floating point system described above?

Give your answer in hexadecimal. 0x\_\_\_\_\_ **556** \_\_\_\_\_

**Solution:**

$$s = 0$$

Next, write out 13.4 as a binary fraction:

Integer part:

$$13 = 8 + 4 + 1 = 1101$$

Fractional part:

$$0.4 * 2 = 0.8 \quad (\text{Take the } 0)$$

$$0.8 * 2 = 1.6 \quad (\text{Take the } 1)$$

$$0.6 * 2 = 1.2 \quad (\text{Take the } 1)$$

$$0.2 * 2 = 0.4 \quad (\text{Take the } 0)$$

$$0.4 * 2 = 0.8 \quad (\text{Take the } 0 - \text{ the pattern } 0110 \text{ now repeats})$$

Putting it together:

$$1101.0110[0110]$$

Now normalize:

$$1.1010110[0110] * 2^3$$

$$E = 3$$

and the bias is 7 so

$$exp = 3 + 7 = 10 \text{ which is } 1010 \text{ in binary.}$$

The frac field needs to be truncated and rounded (using round to even) to 7 bits.

$$frac = 1010110 \text{ } 01 \text{ so round down (truncate) to } 1010110.$$

Putting it all together:

$$0 \text{ } 1010 \text{ } 1010110$$

$$\text{In hex: } 0101 \text{ } 0101 \text{ } 0110 = 0x556.$$

- (e) (3 points) You are given the following hexadecimal number 0x4A4 in the floating point system described above. What is the decimal representation of that number?

Give your answer as a **decimal fraction**, representing the fractional part of the number (if it has one) as a fraction (e.g., 3/4).

5 1/8

**Solution:** First, convert to binary to recover  $s$ ,  $exp$ , and  $M$ .

0100 1010 0100

0 1001 0100100

$s = \text{positive (0)}$

$exp = 9$

$e = 2 (9 - 7)$

$frac = .0100100$

$M = 1.0100100$

Value =  $1.0100100 * 2^2 = 101.00100 = 4 + 1 + 1/8 = 5 \frac{1}{8}$

7. Round the following binary fractions to exactly 3 fractional bits using the “round to even” rounding rules.

(a) (2 points)  $1.01101_2 \rightarrow \underline{\quad 1.011_2 \quad}$

**Solution:** We want to keep 3 fractional bits: 1.011. The remaining bits to discard are  $01_2$ . Since  $01_2$  is less than exactly half ( $10_2$ ), we round down (truncate). The result is  $1.011_2$ .

(b) (2 points)  $1.01011_2 \rightarrow \underline{\quad 1.011_2 \quad}$

**Solution:** We want to keep 3 fractional bits: 1.010. The remaining bits to discard are  $11_2$ . Since  $11_2$  is greater than exactly half ( $10_2$ ), we round up by adding  $0.001_2$ .  $1.010_2 + 0.001_2 = 1.011_2$ .

(c) (2 points)  $1.11010_2 \rightarrow \underline{\quad 1.110_2 \quad}$

**Solution:** We want to keep 3 fractional bits: 1.110. The remaining bits to discard are exactly  $10_2$ , which is exactly halfway (a tie). We look at the least significant bit we are keeping, which is a 0 (even). Therefore, we round to even by leaving it as is (rounding down). The result is  $1.110_2$ .

(d) (2 points)  $1.10110_2 \rightarrow \underline{\quad 1.110_2 \quad}$

**Solution:** We want to keep 3 fractional bits: 1.101. The remaining bits to discard are exactly  $10_2$ , which is exactly halfway (a tie). We look at the least significant bit we are keeping, which is a 1 (odd). Therefore, we round to even by rounding up.  $1.101_2 + 0.001_2 = 1.110_2$ .

8. (16 points) For each of the following assembly code snippets, determine the final decimal value stored in register a0.

Prior to the execution of *each* snippet, assume the following initial machine state:

**Registers:**

- t0 = 16
- t1 = 32
- t2 = -4
- t3 = 0x100
- x0 = 0 (always)

**Memory (Little-Endian):**

Address	Value (32-bit Word)
0x100	0x00000005 (5)
0x104	0x0000000A (10)
0x108	0xFFFFFFFF (-1)
0x10C	0x00000000 (0)

Give your answers in **decimal**.

(a) `addi a0, t3, 4`  
`lw a0, 4(a0)`

a0 is \_\_\_\_\_ -1 \_\_\_\_\_ (in decimal)

**Solution:** a0 gets 0x104. lw loads word at 0x108.

(b) `srai a0, t2, 2`  
`slli a0, a0, 1`

a0 is \_\_\_\_\_ -2 \_\_\_\_\_ (in decimal)

**Solution:**  $-4 \gg 2 = -1$ .  $-1 \ll 1 = -2$ .

(c) `lh a0, 8(t3)`  
`addi a0, a0, 5`

a0 is \_\_\_\_\_ 4 \_\_\_\_\_ (in decimal)

**Solution:** lh loads 0xFFFF, sign-extends to -1.  $-1 + 5 = 4$ .

(d) `lui a0, 2`  
`srlr a0, a0, 11`

a0 is \_\_\_\_\_ 4 \_\_\_\_\_ (in decimal)

**Solution:**  $2 \ll 12 = 8192$ .  $8192 \gg 11 = 4$ .

(e) `lbu a0, 8(t3)`  
`addi a0, a0, 1`

a0 is 256 (in decimal)

**Solution:** `lbu` loads `0xFF`, zero-extends to 255.  $255 + 1 = 256$ .

(f) `sltu a0, t2, t1`  
`xori a0, a0, 1`

a0 is 1 (in decimal)

**Solution:** `unsigned(-4) < 32` is False (0).  $0 \text{ XOR } 1 = 1$ .

(g) `lw a0, 4(t3)`  
`ori a0, a0, 5`

a0 is 15 (in decimal)

**Solution:**  $10 \text{ OR } 5$  ( $1010 \text{ OR } 0101$ ) = 15 (1111).

(h) `mul a0, t0, t2`  
`sub a0, a0, t1`

a0 is -96 (in decimal)

**Solution:**  $16 * -4 = -64$ .  $-64 - 32 = -96$ .

9. Consider the following structure declaration, assuming a 32-bit architecture (RV32) where pointers are 4 bytes and doubles are 8 bytes:

```
struct record {
    short    id;
    double   score;
    char     *name;
    char     grade;
    int      rank;
};
```

- (a) (2 points) Give the total size of the structure in bytes (i.e., the value of the `sizeof(struct record)` expression). Include any required padding.

`sizeof(struct record)` returns     **32**    

**Solution:** Each field is placed at the nearest offset that is a multiple of its size. Starting from offset 0: `id` (short, size 2) sits at offset 0–1, then 6 bytes of padding bring us to offset 8 so that `score` (double, size 8) is 8-byte aligned. `name` (pointer, size 4) follows at offset 16, `grade` (char, size 1) at offset 20, then 3 bytes of padding place `rank` (int, size 4) at offset 24. The struct ends at byte 28, but must be padded to a multiple of 8 (the largest alignment), giving a final size of **32 bytes**.

- (b) (2 points) Assume you have the following C code utilizing the above structure:

```
long get_rank(struct record *x) {
    return x->rank;
}
```

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

```
get_rank:
    lw a0, 24(a0)
    ret
```

- (c) (2 points) If the structure declaration above was rearranged to minimize the space of the structure, what would be the new size of the structure?

Total size of rearranged and minimized `struct` in bytes:     **24**    

**Solution:** For part (a), the offsets are: `id` at 0, 6 bytes of padding, `score` at 8, `name` at 16, `grade` at 20, 3 bytes of padding, `rank` at 24. Total size = 28 bytes. The max alignment requirement is 8 (for the double), so it must be padded to a multiple of 8, making it 32.

For part (b), `rank` is a 32-bit integer located at offset 24, requiring an `lw` instruction.

For part (c), arranging largest to smallest (`score`, `name`, `rank`, `id`, `grade`) gives sizes  $8+4+4+2+1 = 19$  bytes. Pad to the nearest multiple of 8 = 24 bytes.

10. (4 points) You have the following two-dimensional array of 16-bit integers (shorts) declared as follows:

```
short grid[8][6];
```

Assume the start of the array is at address `0x4000`. You have an array access `grid[x][y]`. What are the indices (`x` and `y` below) of the array that refers to the array element at address `0x403A`?

`x` is     4    

`y` is     5    

**Solution:** Address `0x403A` - `0x4000` = `0x3A` (58 in decimal). Since each element is a `short` (2 bytes), the element index is  $58/2 = 29$ . The row length is 6, so `x` (row) =  $29/6 = 4$ , and `y` (column) =  $29\%6 = 5$ .

11. (16 points) 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
<pre> question: 101d0: mv   a5,a0 101d4: li   a0,0 101d8: j    101e4 &lt;question+0x14&gt; 101dc: addi a5,a5,4 101e0: addi a1,a1,-1 101e4: blez a1,10200 &lt;question+0x30&gt; 101e8: lw   a4,0(a5) 101ec: andi a3,a4,1 101f0: bnez a3,101dc &lt;question+0xc&gt; 101f4: bge  a2,a4,101dc &lt;question+0xc&gt; 101f8: addi a0,a0,1 101fc: j    101dc &lt;question+0xc&gt; 10200: ret </pre>	<pre> int question(int a[], int len, int threshold) {     int count = 0;     int val;     while (__len__ &gt; __0__) {         __val__ = *a;         if ((val __&amp;__ 1) == 0) {             if (val __&gt;__ threshold) {                 __count__ = count + 1;             }         }         a = a + __1__;         len = len - 1;     }     return __count__; } </pre>

**Solution:** The RISC-V calling convention places the three arguments in `a0` (`a[]`), `a1` (`len`), and `a2` (`threshold`). The first two instructions (`mv a5,a0` then `li a0,0`) save the array pointer into `a5` and repurpose `a0` as `count`, initialised to 0. The unconditional jump to `101e4` implements a **while** loop by checking the condition before the first iteration. The condition `blez a1` branches when `a1 ≤ 0`—the negation of `len > 0`—giving the first two blanks: **len** and **0**. Inside the loop, `lw a4,0(a5)` dereferences the current pointer, filling `val = *a`. `andi a3,a4,1` computes `val & 1`, so the operator is `&`. The branch `bge a2,a4` skips the increment when `threshold ≥ val`, so the inner condition passes only when `val > threshold`. When both conditions hold, `addi a0,a0,1` increments `count`. After the body, `addi a5,a5,4` advances the pointer by 4 bytes (one `int`), corresponding to `a = a + 1` in pointer arithmetic. Finally, `ret` returns `a0`, i.e. `count`.

12. (3 points) How does a stack canary mitigate stack smashing attacks?

- By placing a randomized, secret value between local variables and the return address, and verifying it hasn't changed before returning**
- By marking the stack memory region as non-executable, preventing any injected shellcode from running
- By randomizing the memory locations of the stack, heap, and libraries every time the program runs
- By statically analyzing the code at compile time to ensure no array bounds are violated

**Solution:** If a contiguous buffer overflow occurs, it will overwrite this secret value before reaching the return address, causing the integrity check to fail and the program to safely abort.

13. (3 points) Return-Oriented Programming (ROP) is an exploit technique primarily designed to bypass which of the following security defenses?

- Address Space Layout Randomization (ASLR)
- Stack canaries
- Non-executable memory protections**
- Control Flow Integrity (CFI)

**Solution:** Because the NX (non-executable) bit prevents attackers from executing their own injected shellcode on the stack, ROP strings together pieces of existing, executable code (gadgets) that are already present in memory.