# CS 2130 Final Exam

**Name** _____ **Computing ID** _____

You MUST clearly write your computing ID and name on the top of this page. Do this **BEFORE** you begin. Please write legibly.

Write your answers in the box labeled "Answer" when provided. In any multiple choice answer, **fill in the circle completely** for credit; **checkmarks, circles, lines, or other marks will be graded as an empty circle**.

If you are still writing when "pens down" is called, your exam will not be graded – even if you are still signing the honor pledge. So please do that first. Sorry to have to be strict on this!

There are 14 pages to this exam. Once the exam starts, please make sure you have all the pages. Questions are worth different amounts of points, so be sure to look over all the questions and plan your time accordingly.

This exam is CLOSED text book, closed notes, closed cell phone, closed smart watch, closed computer, closed neighbor, closed generative AI, closed smart glasses, etc. You may **not** discuss this exam with anyone until after the grades have been released. Please sign the honor pledge below.

On my honor as a student, I have neither given nor received aid on this exam. I will not discuss the content of this exam, even in vague terms, with *anyone* other than current course staff, until *after* grades have been released.

_____

*A crash reduces*
*Your expensive computer*
*To a simple stone.*

## Page 2: Toy ISA to C

Throughout the semester, we took a bottom-up approach, building from electricity on wires to C. The next few questions take a top down approach to Computer Systems and Organization.

Consider the following C function that calculates some "fun" exam math:

```
long multiply(long a, long b); // given in another file

long exammath(long x, long y) {
    long result = y + 16;
    if (x & 1)
       result += multiply(x, y);
    else
       result -= x;
    return result + 1;
}
```

1. [8 points] Rearrange the x86-64 assembly instructions to implement `exammath`. For each section, reorder the instructions by writing the number of the appropriate instruction on the lines provided to the right. Some order has been provided for you and the order in one section *may* influence the ordering in another section. *Each instruction may be used at most once and only within its group. Some instructions may not be used. All blanks will be filled.*

| Possible Instructions | Proper Ordering |
|---|---|
| 1. `addq 0x16, %rsi` | 3 `exammath:` |
| 2. `callq multiply` | ____ |
| 3. `exammath:` | ____ |
| 4. `je .label1` | ____ |
| 5. `leaq 16(%rsi), %rax` | ____ |
| 6. `pushq %rax` | ____ |
| 7. `pushq %rcx` | ____ |
| 8. `testq $1, %rdi` | 2 `callq multiply` |
| 9. `addq %rcx, %rax` | ____ |
| 10. `incq %rax` | ____ |
| 11. `jmp .label2` | ____ |
| 12. `label1:` | 12 `label1:` |
| 13. `label2:` | ____ |
| 14. `popq %rax` | ____ |
| 15. `popq %rcx` | ____ |
| 16. `popq %rcx` | 10 `incq %rax` |
| 17. `retq` | ____ |
| 18. `subq %rdi, %rax` | 17 `retq` |

**Page 3: Toy ISA to C, continued**

2. [8 points] Implement `exammath` using our Toy ISA and following our calling conventions. Part of the code has been provided for you; fill in the missing instructions. In this implementation, assume the `exammath` function is loaded at memory address `0x30` and the `multiply` function is loaded at memory address `0x99`. *Our Toy ISA is described on page 14.*

```
// hints given as {address}: {instruction byte(s)} : {scratch work}
// result = y + 16
    30: 03    : r0 = r3
    31: 62 __ : r0 += ??

// if (x & 1) ...



    36: 88    : push r2
    37: 68 44 : r2 = else (address 0x44)
    39: 76    : if r1 <= 0 jump to body of else (r2)


// body of if: result += multiply(x, y)









    41: 6C 47 : r3 = after_else (address 0x47)


// body of else: result -= x



// after else: return x + 1
```

Write your program in binary below. **Only the final program below will be graded for this question.** If you use fewer instructions, use no-ops to fill in the rest. We have filled in some of the instructions and immediate values already.

| 03 | 62 | | | | | 88 | 68 | 44 | 76 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 3a | 3b | 3c |

| | | | | 6C | 47 | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3d | 3e | 3f | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 |

## Page 4: Binary and Macros

3. [2 points] Convert the 8-bit two's complement binary number `11100100` into decimal.

| Answer |
| --- |
| |

4. [2 points] What is the value of `y` at the end of this code snippet? Answer in hexadecimal.

```
int x = 0x12345678;
char *p = (char *)&x;
int y = p[2];
```

| Answer |
| --- |
| |

5. [2 points] Which of the following C expressions are equivalent to `x & y`, where `x` and `y` are declared as `int`s? *Fill in the circle completely for all that apply.*

- ◯ `~(~x | ~y)`
- ◯ `~x ^ ~y`
- ◯ `!(!x | !y)`
- ◯ `~x & ~y`
- ◯ `(x ^ y) ^ (x | y)`

6. [2 points] Given the C code below that defines macro `THING`, what value is placed in variable `x`?

```
#define THING(k) 7 - k
int x = 10 * THING(3)
```

| Answer |
| --- |
| |

## Page 5: Circuits

7. [7 points] In the space below, draw a circuit with three 1-bit inputs **x**, **y**, and **z** that outputs the result **(x + y) - z**. Label your outputs as **a** and **b**, with **a** as the low-order bit, **b** as the high-order bit. Additionally, if **x** and **y** are 0 and **z** is 1, the circuit will output 00. Construct the circuit using only wires and gates from the set {and, or, not, xor}. Clearly label your input and output wires. *Hint: writing out a truth table can help design and/or check your circuit.*

## Page 6: Writing C

8. [11 points] Implement the `csostrsubseq` function, included in the manual page on this exam. You may **not** use any library functions.

```
const char *csostrsubseq(const char *str, const char *sub) {
```

```
}
```

## Page 7: Memory

Consider the following C code, shown with line numbers which are not part of the code itself. The code contains one or more memory errors. Use this code for the next 2 questions.

```
1  typedef struct {int *vals; int size;} arrayList;
2
3  arrayList *insert(int value, arrayList *a) {
4      int *new_vals = (int *) malloc(a->size + 1);
5      for (int i = 0; i <= a->size; i++) {
6          new_vals[i] = a->vals[i];
7      }
8      a->vals = new_vals;
9      a->vals[a->size] = value;
10     a->size++;
11     return a;
12 }
```

9. [4 points] The code above has one memory leak.

   A. After which line should we add a `free`? For example, if we should free before the `return a;`, write **10**.

   Answer

   B. What should be freed? That is, what should we pass to the call `free();`?

   Answer

10. [4 points] For each of the following memory errors, give the line where the error occurs in the code on the previous page. If the error does not occur, put an X in on the line.

    ____ Escaping pointer

    ____ Fails to use `sizeof` or uses `sizeof` incorrectly

    ____ Possible heap buffer overflow

    ____ Possible stack buffer overflow

11. [8 points] True/False questions on memory. For each of the statements below, fill in the **T** circle *completely* if you think the statement is True. If you think it's False, fill in the **F** circle *completely*.

    (T) (F) Each process is only provided access to a small segment of user space memory; it is not provided a view of all of memory.

    (T) (F) The kernel reserves lower addresses (close to 0) to catch errors such as dereferencing a `NULL` pointer.

    (T) (F) Garbage is defined as all memory on the heap that your program has `malloc`ed where there is no longer a pointer for that memory on the stack.

    (T) (F) Accessing one element beyond the end of an array on the stack will always result in a segmentation fault.

**Page 8: C**

12. [2 points] Given the following C code snippet, which of the expressions below access the value in the array `a` at index 3? *Fill in the circle completely for all that apply.*

```
int a[10]; int *p = a + 3;
```

- ○ `a[3]`
- ○ `*(a + 3)`
- ○ `*(a + 12)`
- ○ `p[0]`
- ○ `*(p)`
- ○ `a + 3`

Consider the following C code, shown with line numbers which are not part of the code itself. Use this code for the next 2 questions.

```
1 int inc(int x) { return x + 1; }
2 void print(_____________________, int x) {
3     printf("%d\n", f(x));
4 }
5
6 int main(int argc, char *argv[]) {
7     if (argc > 1) { ______________________________ }
8     return 0;
9 }
```

13. [2 points] Which of the following should be used in the blank on line 2 to allow `print` to take a function pointer `f` as its first parameter?

- ○ `int *f(int)`
- ○ `(int *)f(int)`
- ○ `int (*f)(int)`
- ○ `function *f`

14. [2 points] Which of the following should be used in the blank on line 7 so that the program prints `26` when run as `./a.out 25` ?

- ○ `print(inc, atoi(argv[0]));`
- ○ `print(inc, atoi(argv[1]));`
- ○ `print(*inc, atoi(argv[1]));`
- ○ `print(&inc, atoi(argv[0]));`

## Page 9: More C

15. [10 points] True/False questions. For each of the statements below, fill in the **T** circle *completely* if you think the statement is True. If you think it's False, fill in the **F** circle *completely*.

    (T) (F) `read(0, buf, 400)` may return fewer than 400 bytes.

    (T) (F) A system call transitions execution from user mode to kernel mode through a hardware-supported mechanism such as the `syscall` instruction.

    (T) (F) The function `write` may defer invoking a system call depending on the kernel's buffering policy.

    (T) (F) By using `#include <myfile.h>`, the C preprocessor will look for and copy in `myfile.h` from the current directory.

    (T) (F) Header files, like the ones we viewed in class, use the `#ifndef` and `#define` directives to avoid circular includes.

16. [2 points] Which of the following will **NOT** evaluate to **false** when used in a conditional, such as an `if` statement?

    ◯ `'\0'`

    ◯ `NULL`

    ◯ `'0'`

    ◯ `0` — the number zero

17. [2 points] Consider the following C code. What does the program print to standard output?

```
char msg[] = {'C', 'S', 'O', '1', '!', '\0'};
char *p = strpbrk(msg, "!S1");
*(p + 1) = 'X';

printf(" %ld %s ", strlen(msg), msg);
write(1, p, 1);
puts("");
```

| Answer |
| --- |
| |

## Page 10: C and Memory

Consider the following code, shown with line numbers which are not part of the code itself. Use this code to answer questions 18-20. Assume that the code was compiled with no optimizations.

```
1  typedef struct {
2     size_t id; char *name; int balance; int type;
3  } account;
4
5  int main() {
6     account first[10];
7     account *second[20];
8
9     for (int i = 0; i < 20; i++)
10        second[i] = (account *) malloc(sizeof(account));
11    for (int j = 0; j < 10; j++)
12        _________________________  = 2130;
13    return 0;
14 }
```

18. [4 points] Which of the following could be used in the blank on line 12 to set `balance` in the first 10 accounts of each array to `2130`? *Fill in the circle completely for all that apply.*

A. In `first`:

- ◯ `(first[j]).balance`
- ◯ `(*(first + j)).balance`
- ◯ `(first[j])->balance`
- ◯ `(first + j)->balance`

B. In `second`:

- ◯ `(second[j]).balance`
- ◯ `(**(second + j)).balance`
- ◯ `(second[j])->balance`
- ◯ `(*(second + j))->balance`

19. [4 points] How many bytes are allocated for `first` (and any of its structs):

A. On the **stack**?

| Answer |
| --- |
| |

B. On the **heap**?

| Answer |
| --- |
| |

20. [4 points] How many bytes are allocated for `second` (and any of its structs):

A. On the **stack**?

| Answer |
| --- |
| |

B. On the **heap**?

| Answer |
| --- |
| |

## Page 11: More Questions

21. [2 points] In this course, we primarily used the CS department portal to compile and run our code. Assuming that you have written code in a file named `final.c` that uses functions from `list.c` as defined in `list.h`, write the command to compile your code with 1 level of optimization.

```
mst3k@portal04:~$ _____
```

22. [6 points] True/False questions. Suppose you found a security vulnerability in code that you did not write. For each of the statements below, fill in the **T** circle *completely* if you think the statement is True. If you think it's False, fill in the **F** circle *completely*.

(T) (F) You should immediately report it to the author of the software.

(T) (F) If the author of the software fixes the vulnerability, you should not report it on a public repository, such as CVE.

(T) (F) It is customary to give the authors a week before publishing the vulnerability in the media.

23. [2 points] Consider the following code that connects a socket, as in homework 10. What should be included in the blanks on lines 10 and 14 for the client to read input sent by the server into `buffer`?

```
1  struct sockaddr_in ipOfServer;
2  memset(&ipOfServer, 0, sizeof(struct sockaddr_in));
3  ipOfServer.sin_family = AF_INET;
4  ipOfServer.sin_addr.s_addr = inet_addr("127.0.0.1");
5  ipOfServer.sin_port = htons(49999);
6  int s = socket(AF_INET, SOCK_STREAM, 0);
7  int connection = connect(s, (struct sockaddr*)&ipOfServer,
       sizeof(ipOfServer));
8  struct pollfd fds[2];
9  fds[0].fd = 1;      fds[0].events = POLLIN;
10 fds[1].fd = _____; fds[1].events = POLLIN;
11 char buffer[4000];
12 for(;;) {
13     if (poll(fds, 2, 60000) > 0) {
14         if (fds[1].revents & POLLIN) {read(____, buffer, 4000);}
15     }
16 }
```

| Answer |
| --- |
| |

## Page 12: Assumptions and Man Pages

**Assumptions**. Assume unless otherwise specified:

- All necessary `#include`s have been used
- `char`, `short`, `int`, and `long` are 8-, 16-, 32-, and 64-bits long, respectively
- The compiler pads pointers where it is allowed to do so such that `sizeof(struct X)` is:
  - an even multiple of the size of its largest field
  - the smallest such multiple big enough to store all its fields

**x86-64 Callee-saved Registers:** `%rbx`, `%rsp`, `%rbp`, `%r12`, `%r13`, `%r14`, `%r15`

---

STRPBRK(3) Linux Programmer's Manual STRPBRK(3)

NAME

`strpbrk` - search a string for any of a set of bytes

SYNOPSIS

```
#include <string.h>

char *strpbrk(const char *s, const char *accept);
```

DESCRIPTION

The `strpbrk()` function locates the first occurrence in the string `s` of any of the bytes in the string `accept`.

RETURN VALUE

The `strpbrk()` function returns a pointer to the byte in `s` that matches one of the bytes in `accept`, or `NULL` if no such byte is found.

**Page 13: Man Page**

CSOSTRSUBSEQ (3) Linux Programmer's Manual CSOSTRSUBSEQ(3)

NAME

`csostrsubseq, csostrnsubseq` - locate a subsequence

SYNOPSIS

```
#include <cso1final.h>

const char *csostrsubseq(const char *str, const char *sub);

const char *csostrnsubseq(const char *str, const char *sub, size_t n);
```

DESCRIPTION

The `csostrsubseq()` function determines whether `sub` is a subsequence of `str`. A string `sub` is a subsequence of `str` if all characters in `sub` appear in `str` in the same order, but not necessarily contiguously.

The `csostrnsubseq()` function is similar, except it considers only the first (at most) `n` bytes of `str` and `sub`.

RETURN VALUE

The `csostrsubseq()` and `csostrnsubseq` functions return a pointer to the beginning of the subsequence located in `str`, or `NULL` if the substring is not found.

EXAMPLES

Finding a subsequence that is not in the string returns `NULL`:

```
#include <cso1final.h>
char *ret = csosubstrseq("abcdef", "axd");
```

Finding a subsequence that is contiguous in the string returns a pointer to `cdef`:

```
#include <cso1final.h>
char *ret = csosubstrseq("abcdef", "cd");
```

Finding a subsequence that is not contiguous in the string returns a pointer to `bcdef`:

```
#include <cso1final.h>
char *ret = csosubstrseq("abcdef", "bdf");
```

Finding an empty subsequence in the string returns the `str` pointer (i.e., `abcdef`):

```
#include <cso1final.h>
char *ret = csosubstrseq("abcdef", "");
```

## Page 14: Our Toy ISA

**Our Example Toy ISA**. This is the ISA described in class and used in Lab 4-5 and Homework 3-4. Each instruction is one (or two) bytes, defined as:


If the reserved bit (bit 7) in our instruction is 0, the following table defines our instruction encoding.

| icode | b | behavior |
|---|---|---|
| 0 | | `rA = rB` |
| 1 | | `rA &= rB` |
| 2 | | `rA += rB` |
| 3 | 0 | `rA = ~rA` |
| | 1 | `rA = !rA` |
| | 2 | `rA = -rA` |
| | 3 | `rA = pc` |
| 4 | | `rA` = read from memory at address `rB` |
| 5 | | write `rA` to memory at address `rB` |
| 6 | 0 | `rA` = read from memory at `pc + 1` |
| | 1 | `rA` &= read from memory at `pc + 1` |
| | 2 | `rA` += read from memory at `pc + 1` |
| | 3 | `rA` = read from memory at the address stored at `pc + 1` |
| | | For icode 6, increase `pc` by 2 at end of instruction |
| 7 | | Compare `rA` as 8-bit two's-complement to `0` |
| | | if `rA <= 0` set `pc = rB` |
| | | else increment `pc` as normal |

If the reserved bit (bit 7) in our instruction is 1, the following table defines our extended instruction encoding to provide function calls and stack operations.

| icode | b | behavior |
|---|---|---|
| 0 | 0 | Decrement `rsp` and push contents of `rA` to the stack |
| | 1 | Pop the top value from the stack into `rA` and increment `rsp` |
| | 2 | Push `pc + 2` onto the stack, `pc` = read from memory at `pc + 1` |
| | 3 | `pc` = pop the top value from the stack |
| | | For icode 0, if `b` is not 2, update the `pc` as normal |

ToyISA calling conventions: Register 1 is callee-save. Function arguments will be placed in registers 2 and 3; the return value is stored in register 0.