University of Washington — Computer Science & Engineering

# CSE 351, Winter 2022 — Midterm Exam

Wednesday, February 9, 2022 — Friday, February 11, 2022

Name: _____

UW NetID: _____ @uw.edu

**Instructions**

- You have 72 hours to complete the exam, though we expect that it will take you 1–4 hours. Submit your work on Gradescope by Friday, February 11 at 11:59pm. **Late submissions will not be accepted**.
- You may print this out and (legibly, please!) hand-write your answers, use e-ink on a tablet, or use software such as Adobe Reader to type your answers. Regardless, please *use the space provided* to help us with grading (except for the last question). Printed exams can be scanned into a PDF using your phone; many such apps are available. Please ensure that the scans are clear and the pages are straight.
- This exam is open note, open book, open Internet. Some questions will require the use of the CSE Linux environment (either `attu` or the VM).
- Collaboration is permitted, **subject to the "Gilligan's Island Rule."** You may discuss problems with your classmates, and write things down on shared paper or whiteboards, but you *may not take any written artifacts from your discussions*. You should engage in some kind of unrelated activity for 30 minutes (like watching an episode of *Gilligan's Island*) between discussing exam problems and working on the exam, to ensure that you can fully understand and reproduce the solutions yourself.


| Question | Points |
|---|---|
| Bits, Bytes, and Nybbles | 5 |
| Stuffed Animal Organization | 9 |
| Pointers and Characters and Numbers, Oh My! | 18 |
| The Careful Design of Pointy Things | 8 |
| Do The Register Shuffle | 11 |
| Stacks Considered hARMful | 4 |
| Don't Overflow This Stack! | 15 |
| Taking a Step Back | 10 |
| Total: | 80 |

1. (5 points) Bits, Bytes, and Nybbles

   Given the 32-bit numeral `0x45480000`, interpret it as a...

   (a) (1 point) signed `int`:

   (a) 1162346496

   (b) (2 points) `float`:

   (b) 3200

   (c) (2 points) string literal, read from left-to-right (you may use an ASCII table for reference):

   (c) EH

2. (9 points) Stuffed Animal Organization

   Sam has decided to start collecting Beanie Babies! He needs to figure out a scheme for organizing them, and wants your help. There are two characteristics that he cares about: eye color and number of legs.

   Sam has multiple shelves on which he stores his Beanie Babies. Currently, each shelf can hold **up to 12** (but may hold fewer). His goal is to have a list of all his Beanie Babies and their locations (which shelf, and where on the shelf), so that he can easily locate them. He's come up with two organizational schemes, both of which take up 16 bits of space.

   **Proposed Organizational Schemes**

   1. Store each field (shelf number, position on shelf, eye color, number of legs) separately, with each field taking up 4 bits. Find a Beanie Baby by starting at the topmost shelf and counting shelves from top-to-bottom until he reaches the correct shelf, then counting Beanie Babies from left-to-right.
   2. Combine the shelf number and shelf position fields into a new 8-bit field. Find a Beanie Baby by starting from the leftmost Beanie Baby on the topmost shelf, then counting Beanie Babies from left-to-right and from top-to-bottom. (The representation of eye color and number of legs remains the same.)

   (a) (3 points) Which scheme can represent more *valid* positions of Beanie Babies? Justify your answer in 1-2 sentences.

   **Solution:** Scheme 2, since scheme 1 uses 4 bits (16 possible indices) for a Beanie Baby's position on a shelf, but each shelf only has 12 positions.

Heads up! There's more to this question on the next page ☺

(b) (3 points) Provide one advantage of using *scheme 1* instead of scheme 2, given the way that Sam will use this scheme in the real world. Justify your answer in 1-2 sentences.

**Solution:** Many possible answers. Examples: 1) less overall counting (can count vertically to the proper shelf, then horizontally to the position); 2) if he adds or removes a Beanie Baby, then he doesn't need to changes the indices of all the others (just the ones on that shelf).

(c) (3 points) Sam wants to compare Beanie Babies by number of legs. He writes a function `compareLegs`, which takes two arguments, `beanie1` and `beanie2`. It returns `1` if `beanie1` has fewer legs than `beanie2`, and `0` otherwise. But he can't remember how to complete it. Write out the appropriate *bit mask* to make the function work properly. **Assume that the 4 bits storing number of legs are the least significant 4 bits of the data type**.

```
int compareLegs(
    unsigned short beanie1,
    unsigned short beanie2
) {
    return      beanie1 & 0xF      <      beanie2 & 0xF      ;
}
```

3. (18 points) Pointers and Characters and Numbers, Oh My!

   For this question, refer to the C assignments and memory diagram below, with addresses increasing left-to-right and top-to-bottom. Remember that x86-64 machines are little endian.

```
char *c = 0x19;
short *s = 0xc;
float *f = 0x28;
```

| Address | +0 | +1 | +2 | +3 | +4 | +5 | +6 | +7 |
|---|---|---|---|---|---|---|---|---|
| 0x00 | 1e | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 0x08 | aa | bb | cc | dd | ee | ff | 00 | 11 |
| 0x10 | 8a | 7c | 6f | 22 | 9a | 66 | 44 | 17 |
| 0x18 | 33 | 77 | 6f | 6c | 66 | 62 | 79 | 74 |
| 0x20 | 65 | 73 | 00 | 44 | 00 | 00 | 88 | c0 |
| 0x28 | de | ad | be | ef | ca | fe | f0 | 0d |

   (a) (15 points) Fill in the C type, hex value, and interpreted value for each of the following C expressions. You should interpret integers in two's complement, characters as ASCII, and floating-point numbers using the IEEE-754 standard. You may find the floating-point homework useful. For pointers, you can write "pointer" as the interpretation.

| C Expression | C Type | Hex Value | Interpretation |
|---|---|---|---|
| `*s` | `short` | `0xffee` | $-18$ |
| `f+2` | `float *` | `0x2c` | **pointer** |
| `*(c+2)` | `char` | `0x6c` | `'l'` |
| `f[-1]` | `char` | `0xc0880000` | $-4.25$ |
| `* ((short *) (c-1))` | `short` | `0x7733` | $30515$ |

   (b) If we treat `c` as a string literal (i.e., an array of ASCII characters)...

      i. (1 point) What is its string value?

      **Solution:** `'wolfbytes'`

      ii. (2 points) How many bytes are taken up by `c` and the data it points at?

      **Solution:** 18 (8 for the pointer, 10 for the string itself, including the null terminator)

4. (8 points) The Careful Design of Pointy Things

Your intrepid instructor has decided to invent a new programming language, WolfLang, that promises to fix *all* the issues from the languages that preceded it. He decides to include a pointer datatype like C does, but with a few changes in an effort to make it safer.

**Proposed Changes in WolfLang**

1. Pointers can *only* be assigned to the address of a variable that matches their type; they cannot be cast. For example, the following:

   ```
   int x = 3;
   float *y = (float *) &x;  // invalid!
   ```

   would not be valid in WolfLang because it attempts to cast an `int *` to a `float *`.

2. Pointers cannot be manipulated via arithmetic. For example, the following:

   ```
   int x = 3;
   int *xp = &x;
   xp = xp + 1;  // invalid!
   ```

   would not be valid because it performs pointer arithmetic on `xp`.

What *advantages and disadvantages* do the restrictions in WolfLang's pointers create compared to C pointers? Give one of each, and discuss what they will mean for programmers using the language.

**Solution:** Many possible answers. Examples of advantages: 1) cannot cast pointers in such a way that they reference mystery data (e.g., `int` to `long`), 2) cannot change addresses to mystery data. Disadvantages: 1) more difficult to do certain bit twiddling operations (e.g., treat `int` like a `float`), 2) pointer arithmetic can sometimes lead to cleaner code.

5. (11 points) Do The Register Shuffle

You come across the following mysterious-looking assembly function. This function takes two arguments. Assume that the C variables for each argument are the same as their register names, i.e., `rdi` and `rsi`.

```
mystery:
        jmp     .L2
.L3:
        movb    %al, (%rsi)
        addq    $1, %rdi
        addq    $1, %rsi
.L2:
        movzbl  (%rdi), %eax
        testb   %al, %al
        jne     .L3
        movb    $0, (%rsi)
        ret
```

(a) (1 point) In the above function, what **C variable type** is `%rdi`?

(a) char *

(b) (1 point) What **C variable type** is `%rsi`?

(b) char *

(c) (2 points) This function contains a `while` loop. What is the loop condition?

```
while (*  rdi  != '\0' or 0 ) {
```

(d) (5 points) Fill in the missing parts of the C code that is equivalent to the assembly above:

```
void mystery((answer to a) rdi, (answer to b) rsi) {
    while ((answer to c)) {
        *rsi = *rdi;
        rdi++;
        rsi++;
    }
    *rsi = '\0' or 0 ;
}
```

Hey, there's more to this question! Don't forget to turn the page! 😎

(e) (2 points) On a high level, what does this function *accomplish*? Explain in 1-2 sentences.

**Solution:** It copies the null-terminated string from `arg1` to `arg2` (`strcopy`).

(f) **(2 points extra credit)** This function is vulnerable to buffer overflow attacks! Briefly explain how.

**Solution:** There is no way to specify the length of `arg2`, so it will blindly copy `arg1` to `arg2` until it reaches a null terminator, even if there's not enough space allocated.

6. (4 points) Stacks Considered hARMful

Some instruction set architectures, like ARM, provide a *register* in which a procedure stores its return address before making a call. The return address is *only* pushed onto the stack when the callee needs to make another call, because the register will be overwritten by the new return address. When this happens, you can think of the return address register like a special-purpose caller-saved register.

Provide one *advantage* of including this register. Briefly justify your answer.

**Solution:** Many possible answers. Examples: reduced memory footprint for leaf function; don't need to access (slow) memory as much; simpler assembly code for procedures that don't call other procedures; more difficult to perform buffer overflow attacks because the return address is further up the stack.

7. (15 points) Don't Overflow This Stack!

For this problem, we'll examine a strange recursive C function (and corresponding assembly). One of the main goals here is to give you additional experience with using `gdb`.

In a CSE Linux environment (`attu` or the CSE VM), execute the following three commands to download the necessary files, and set the correct permissions:

```
wget https://courses.cs.washington.edu/courses/cse351/22wi/files/strange.c
wget https://courses.cs.washington.edu/courses/cse351/22wi/files/strange
chmod +x strange
```

To ensure consistency, please **do not recompile** `strange`.

Addresses & Memory Layout

(a) (1 point) Using the `print` command in `gdb`, what is the **address of** the function `strange`?

(a) `0x401146`

(b) (1 point) When the compiler created an object file for this code, which table(s) stored the function `strange`'s name?

◯ Symbol table ◯ Relocation table √ **Symbol *and* relocation table**

(c) (2 points) By looking at the disassembly, what is the **highest** return address that will be pushed onto the stack *by* the function `strange`?

(c) `0x401172`

(d) (2 points) How many bytes does the *code for* the function `strange` take up in memory?

(d) 48

Stack Frame Layout

(e) (1 point) How large is the stack frame of `strange`? Recall that the return address is considered part of the *callee's* stack frame.

(e) `16B`

(f) (2 points) Which register(s) are pushed onto the stack by `strange`? Which C variable(s) are stored in these registers after pushing the previous value(s)?

**Solution:** Just `%rbx`, which is used to store `n`.

Oh my, this question continues on the next page! 😄

Stack Frame Counts

You can run `strange` in the terminal with a single command-line argument, which is the argument `n` to the `strange` function. For example, `./strange 3` will run the function with `n = 3`. The program will print out the sum of all numbers it computed in recursive calls.

(g) (2 points) Which command-line argument creates the *highest number* of user-defined stack frames? Start counting from the first call to `strange` (i.e., don't count `main` or `printf`).

◯ 2 √ **3** ◯ 4 ◯ 5

(h) (2 points) How *many* user-defined stack frames are created when executing `strange` with the command-line argument above? Again, start counting from the first call to `strange`.

(h) 8

(i) (2 points) What is the *maximum stack frame depth* when executing `strange` with the command-line argument above? Start counting from the first call to `strange`.

Note that you can verify this by stopping at the maximum depth, executing the command `backtrace`, and counting the output lines.

(i) 8

8. (10 points) Taking a Step Back

   Write 2–4 paragraphs reflecting on the experience of learning the material in this half of the course. This is open-ended; we're looking for evidence that you took some time to think through how this material (course material, lectures, and labs) personally affected you.

   This could be exclusively technical or exclusively socio-technical, but we'd prefer you include elements of both. There isn't a single correct answer here — we just want you to have some space to reflect on what you've accomplished, what felt valuable to you, and what you'd rather do without.

   If you're not sure where to start, we'd recommend following the format below. You aren't required to use this, but it may be helpful for getting started with the creative process.

   We've provided space on the next pages for you to hand-write your reflection, if you prefer (legibly, please!). You are *not* required to fill out all two pages; we've intentionally left extra room to accommodate various writing styles. You can also attach typed page(s) to your submission.

   We're expecting this should take you around 30 minute to complete.

Optional Example Format

For each statement, note the degree to which you agree with it; one of Strongly Disagree, Disagree, Somewhat Disagree, Neutral, Somewhat Agree, Agree, Strongly Agree. Then, write a few paragraphs that explain your choice, noting both what might have changed and how you've experienced that change.

- From my experiences in this course, my view of low-level programming has changed.
- From my experiences in this course, my understanding of how computation is performed has changed.
- From my experiences in this course, my idea of what it means to be a computer scientist has changed.
- From my experiences in this course, the way that I see myself in computing spaces has changed.
- From my experiences in this course, the way I see myself broadly has changed.
- From my experiences in this course, my career goals, either from my first job or from my career as a whole have changed.

**Solution:** Full credit for any reasonable, well-thought out discussion.

# CSE 351 Reference Sheet (Midterm)

| Binary | Decimal | Hex |
|---|---|---|
| 0000 | 0 | 0 |
| 0001 | 1 | 1 |
| 0010 | 2 | 2 |
| 0011 | 3 | 3 |
| 0100 | 4 | 4 |
| 0101 | 5 | 5 |
| 0110 | 6 | 6 |
| 0111 | 7 | 7 |
| 1000 | 8 | 8 |
| 1001 | 9 | 9 |
| 1010 | 10 | A |
| 1011 | 11 | B |
| 1100 | 12 | C |
| 1101 | 13 | D |
| 1110 | 14 | E |
| 1111 | 15 | F |

| $2^0$ | $2^1$ | $2^2$ | $2^3$ | $2^4$ | $2^5$ | $2^6$ | $2^7$ | $2^8$ | $2^9$ | $2^{10}$ |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 256 | 512 | 1024 |

**IEEE 754 FLOATING-POINT STANDARD**

Value: $\pm 1 \times \text{Mantissa} \times 2^{\text{Exponent}}$

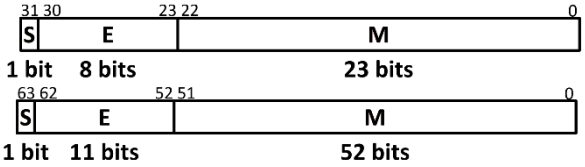
Bit fields: $(-1)^S \times 1.M \times 2^{(E-\text{bias})}$

where Single Precision Bias = 127,
Double Precision Bias = 1023.

**IEEE 754 Symbols**

| E | M | Meaning |
|---|---|---|
| all zeros | all zeros | ± 0 |
| all zeros | non-zero | ± denorm num |
| 1 to MAX-1 | anything | ± norm num |
| all ones | all zeros | ± ∞ |
| all ones | non-zero | NaN |

**IEEE Single Precision and Double Precision Formats:**




## Assembly Instructions

| | |
|---|---|
| **`mov a, b`** | Copy from `a` to `b`. |
| **`movs a, b`** | Copy from `a` to `b` with sign extension. Needs *two* width specifiers. |
| **`movz a, b`** | Copy from `a` to `b` with zero extension. Needs *two* width specifiers. |
| **`lea a, b`** | Compute address and store in `b`.<br>*Note:* the scaling parameter of memory operands can only be 1, 2, 4, or 8. |
| **`push src`** | Push `src` onto the stack and decrement stack pointer. |
| **`pop dst`** | Pop from the stack into `dst` and increment stack pointer. |
| **`call <func>`** | Push return address onto stack and jump to a procedure. |
| **`ret`** | Pop return address and jump there. |
| **`add a, b`** | Add from `a` to `b` and store in `b` (and sets flags). |
| **`sub a, b`** | Subtract `a` from `b` (compute `b-a`) and store in `b` (and sets flags). |
| **`imul a, b`** | Multiply `a` and `b` and store in `b` (and sets flags). |
| **`and a, b`** | Bitwise AND of `a` and `b`, store in `b` (and sets flags). |
| **`sar a, b`** | Shift value of `b` *right* (*arithmetic*) by `a` bits, store in `b` (and sets flags). |
| **`shr a, b`** | Shift value of `b` *right* (*logical*) by `a` bits, store in `b` (and sets flags). |
| **`shl a, b`** | Shift value of `b` *left* by `a` bits, store in `b` (and sets flags). |
| **`cmp a, b`** | Compare `b` with `a` (compute `b-a` and set condition codes based on result). |
| **`test a, b`** | Bitwise AND of `a` and `b` and set condition codes based on result. |
| **`jmp <label>`** | Unconditional jump to address. |
| **`j* <label>`** | Conditional jump based on condition codes (*more on next page*). |
| **`set* a`** | Set byte `a` to 0 or 1 based on condition codes. |

## Conditionals

| Instruction | | | (op) s, d | test a, b | cmp a, b |
|---|---|---|---|---|---|
| **je** | **sete** | "Equal" | d (op) s == 0 | b & a == 0 | b == a |
| **jne** | **setne** | "Not equal" | d (op) s != 0 | b & a != 0 | b != a |
| **js** | **sets** | "Sign" (negative) | d (op) s < 0 | b & a < 0 | b-a < 0 |
| **jns** | **setns** | (non-negative) | d (op) s >= 0 | b & a >= 0 | b-a >= 0 |
| **jg** | **setg** | "Greater" | d (op) s > 0 | b & a > 0 | b > a |
| **jge** | **setge** | "Greater or equal" | d (op) s >= 0 | b & a >= 0 | b >= a |
| **jl** | **setl** | "Less" | d (op) s < 0 | b & a < 0 | b < a |
| **jle** | **setle** | "Less or equal" | d (op) s <= 0 | b & a <= 0 | b <= a |
| **ja** | **seta** | "Above" (unsigned >) | d (op) s > 0U | b & a > 0U | b $>_U$ a |
| **jb** | **setb** | "Below" (unsigned <) | d (op) s < 0U | b & a < 0U | b $<_U$ a |

## Registers

| Name | Convention | Name of "virtual" register | | |
|---|---|---|---|---|
| | | Lowest 4 bytes | Lowest 2 bytes | Lowest byte |
| %rax | Return value – **Caller** saved | %eax | %ax | %al |
| %rbx | **Callee** saved | %ebx | %bx | %bl |
| %rcx | Argument #4 – **Caller** saved | %ecx | %cx | %cl |
| %rdx | Argument #3 – **Caller** saved | %edx | %dx | %dl |
| %rsi | Argument #2 – **Caller** saved | %esi | %si | %sil |
| %rdi | Argument #1 – **Caller** saved | %edi | %di | %dil |
| %rsp | Stack Pointer | %esp | %sp | %spl |
| %rbp | **Callee** saved | %ebp | %bp | %bpl |
| %r8 | Argument #5 – **Caller** saved | %r8d | %r8w | %r8b |
| %r9 | Argument #6 – **Caller** saved | %r9d | %r9w | %r9b |
| %r10 | **Caller** saved | %r10d | %r10w | %r10b |
| %r11 | **Caller** saved | %r11d | %r11w | %r11b |
| %r12 | **Callee** saved | %r12d | %r12w | %r12b |
| %r13 | **Callee** saved | %r13d | %r13w | %r13b |
| %r14 | **Callee** saved | %r14d | %r14w | %r14b |
| %r15 | **Callee** saved | %r15d | %r15w | %r15b |

## Sizes

| C type | x86-64 suffix | Size (bytes) |
|---|---|---|
| char | b | 1 |
| short | w | 2 |
| int | l | 4 |
| long | q | 8 |