Wawrzynek, Weaver
Fall 2021

CS 61C Midterm

Solutions last updated: Saturday, October 23, 2021

PRINT your name: ______________________ ______________________
(first) (last)

PRINT your student ID: ______________________

Read the following honor code and sign your name.

I understand that I may not collaborate with anyone else on this exam, or cheat in any way. I am aware of the Berkeley Campus Code of Student Conduct and acknowledge that academic misconduct will be reported to the Center for Student Conduct and may further result in, at minimum, negative points on the exam and a corresponding notch on Nick's Stanley Fubar demolition tool.

SIGN your name: ______________________

You have 110 minutes. There are 6 questions of varying credit (100 points total).

For questions with **circular bubbles**, you may select only one choice.

○ Unselected option (completely unfilled)

● Only one selected option (completely filled)

For questions with **square checkboxes**, you may select one or more choices.

■ You can select

■ multiple squares (completely filled).

Anything you write that you ~~cross out~~ will not be graded.

If an answer requires hex input, make sure you only use capitalized letters! For example, `0xDEADBEEF` instead of `0xdeadbeef`. Please include hex (`0x`) or binary (`0b`) prefixes in your answers. For all other bases, do not add the suffix or prefixes.

This page is intentionally left blank.

Q1 *Potpourri* (10 points)

Q1.1 (1.25 points) True or False: The compiler resolves define statements.

● True ○ False

Solution: True. In particular, the pre-processor, which is part of the compiler, examines the C code and replaces all instances of the defined variable with its value. This must be done on C code, so it happens in the compilation stage (later stages no longer use C code).

Grading: 1.25 points for True.

Q1.2 (1.25 points) True or False: The assembler is the step with the highest computational complexity among CALL.

○ True ● False

Solution: False. The compiler is more complex than the assembler.

Grading: 1.25 points for False.

Q1.3 (1.25 points) True or False: The assembler produces an executable.

○ True ● False

Solution: False. The assembler creates an object file. The linker creates an executable.

Grading: 1.25 points for False.

Q1.4 (1.25 points) True or False: In the loader, the program is placed in memory in preparation of running the code.

● True ○ False

Solution: True. Executable files (the program instructions after passing through the compiler, assembler, and linker) are stored on the disk, and the loader will place the executable in memory before running it.

Grading: 1.25 points for True.

Q1.5 (1.25 points) Convert `0xDA71` to a 16-bit binary value, including the prefix.

Solution: `0b1101 1010 0111 0001`

Remember that one hexadecimal digit (16 possible values) can be represented by four binary digits (4 digits, 2 possible values each, $2^4 = 16$). Converting each hexadecimal digit to its binary representation, we get `0xD = 0b1101`, `0xA = 0b1010`, `0xA = 0b0111`, and `0x1 = 0b0001`.

Note that the prefix is `0b` to denote binary values.

Other versions of this exam gave different hexadecimal values, but the conversion process is the same:

- `0x326A = 0b0011 0010 0110 1010`
- `0xB13C = 0b1011 0001 0011 1100`
- `0x126D = 0b0001 0010 0110 1101`

Grading: 1.25 points for the correct binary string. No partial credit, sorry.

Q1.6 (1.25 points) Convert `0x85` to decimal, assuming the data was stored as an unsigned one-byte integer.

Solution: 133

This question asks you to convert the provided one-byte (8 bits = 4 nibbles = 2 hexadecimal digits) hexadecimal number to a decimal number.

Hexadecimal is base-16, so this hexadecimal number has a 5 in the ones place and an 8 in the 16s place. In decimal, this is $(8 \times 16) + (1 \times 5) = 133$.

Other versions of this exam gave different hexadecimal values, but the conversion process is the same:

- `0x86 = 134`
- `0x87 = 135`
- `0x88 = 136`

Grading: 1.25 points for the correct decimal number. No partial credit, sorry.

Q1.7 (1.25 points) Convert `0x85` to decimal, assuming the data was stored as a 2's complement one-byte integer.

Solution: -123

This question asks you to convert a hexadecimal number to a two's complement integer. To do this, we start by writing out the hexadecimal number in binary (using the same process as Q1.5): `0x85 = 0b1000 0101`.

Recall that in 2's complement, the most-significant bit tells us whether the number is positive or negative. Here, the most-significant bit is 1, so the number must be negative.

If the number is negative, we need to invert all the bits and add 1 to determine the value of the number. Inverting all the bits gives us `0b0111 1010`, and adding 1 gives us `0b0111 1011`.

Now we can convert this binary number into a decimal number: $2^0+2^1+2^3+2^4+2^5+2^6 = 123$. Remember that we determined the number is negative, so our final answer is -123.

Other versions of this exam gave different hexadecimal values, but the conversion process is the same:

- `0x86 = -122`
- `0x87 = -121`
- `0x88 = -120`

Grading: 1.25 points for the correct decimal number. Half credit (0.675 points) only for the correct decimal number but the opposite sign (e.g. 123 instead of -123).

Q1.8 (1.25 points) Convert `0x85` to decimal, assuming the data was stored as a sign-magnitude one-byte integer.

Solution: -5

This question asks you to convert a hexadecimal number to a sign-magnitude integer. To do this, we start with the binary version of the number (which we wrote out in the previous part): `0x85 = 0b1000 0101`.

Recall that in sign-magnitude, the most-significant bit tells us whether the number is positive or negative. Here, the most-significant bit is 1, so the number must be negative.

In sign-magnitude, all the other bits of the number (except the most-significant bit) tell us the value of the number. Here, those bits are `0b000 0101`.

Now we can convert this binary number into a decimal number: $2^0 + 2^2 = 5$. Remember that we determined the number is negative, so our final answer is -5.

Other versions of this exam gave different hexadecimal values, but the conversion process is the same:

- `0x86 = -6`
- `0x87 = -7`
- `0x88 = -8`

Grading: 1.25 points for the correct decimal number. No partial credit, sorry.

Q2 *Now, Where Did I Put Those Strings?* **(10 points)**

Consider the following code:

```
char *foo() {
    char *str1 = "Hello World";
    char str2[] = "Hello World";
    char *str3 = malloc(sizeof(char) * X);
    strcpy(str3, "Hello World");
    // INSERT CODE FROM PARTS 5-7
}
```

The `char *strcpy(char *dest, char *src)` copies the string pointed to by `src`, including the terminating null byte (`'\0'`), to the buffer pointed to by `dest`. The strings may not overlap, and the destination string `dest` must be large enough to receive the copy.

Q2.1 (1 point) Where is `*str1` located in memory?

○ code [✓ static] ○ heap ○ stack

Solution: Static

This question is asking about the location of `*str1`, the address stored in `str1`.

The code assigns the `str1` pointer to a hard-coded string `"Hello World"`. C will put this hard-coded string in static memory.

Grading: 1 point for selecting static.

Q2.2 (1 point) Where is `*str2` located in memory?

○ code ○ static ○ heap [✓ stack]

Solution: Stack

This question is asking about the location of `*str2`, the address stored in `str2`.

`str2` is a character array, and it is declared inside the `foo` function, so it is a local variable. Local variables are stored in stack memory.

Grading: 1 point for selecting stack.

Q2.3 (1 point) Where is `*str3` located in memory?

○ code ○ static [✓ heap] ○ stack

Solution: Heap

This question is asking about the location of `*str3`, the address stored in `str3`.

The code assigns the `str3` pointer to the result of calling `malloc`. Recall that `malloc` returns a pointer to memory on the heap, so `*str3` is on the heap.

Grading: 1 point for selecting heap.

Q2.4 (1 point) What is the minimum value of X needed for the code to have well-defined behavior?

Solution: 12

This question is asking how much space needs to be allocated on the heap in order to fit the string being copied into the heap memory.

`strcpy` will copy the string `"Hello World"`, including the null terminator byte, into the memory created by the `malloc` call. The string is 11 bytes (don't forget to include the space), and the null terminator is 1 byte, so in total, we need 12 bytes of memory to fully store this string.

Grading: 1 point for the correct answer (12). No partial credit, sorry.

Which of the following lines can be inserted into the function at the given line, with well-defined behavior? Select all that apply.

Q2.5 (1 point) Returning the string.

■ `return str1;` ■ `return str3;`

☐ `return str2;` ☐ None of the above

Solution: `return str3;`

`str1` is a pointer to static memory, which doesn't change throughout program execution, so `return str1;` is safe.

`str2` is a pointer to the stack. When the function returns, the string on the stack is erased, which causes `return str2;` to have undefined behavior.

`str3` is a pointer to the heap. Heap memory stays allocated until the programmer calls `free`. Since this function never calls `free`, the string on the heap will stay allocated, so `return str3;` is safe.

Grading: Each answer choice was graded independently. 1/3 of a point for correctly selecting `str1`, 1/3 of a point for correctly *not* selecting `str2`, and 1/3 of a point for correctly selecting `str3`. Selecting "None of the above" is worth 1/3 points (for correctly not selecting `str2`).

Q2.6 (1 point) Modifying the string.

☐ `str1[0] = 'J';` ■ `str3[0] = 'J';`

■ `str2[0] = 'J';` ☐ None of the above

Solution: `str2[0] = 'J';` and `str3[0] = 'J';`

`str1` is a pointer to static memory, which is read-only. Trying to modify static memory will result in an error.

`str2` is a pointer to the stack. Modifying stack memory is legal, so `str2[0] = 'J';` is safe.

`str3` is a pointer to the heap. Modifying heap memory is legal, so `str3[0] = 'J';` is safe.

Grading: Each answer choice was graded independently, just like in the previous subpart.

Q2.7 (1 point) Freeing the string.

☐ `free(str1);` ■ `free(str3);`

☐ `free(str2);` ☐ None of the above

Solution: `free(str3);`

Recall that `free` is used to free any allocated memory. This means that `free` can only be called on a pointer to the heap.

`str1` is a pointer to static memory. Calling `free` on a pointer to static memory will result in an error.

`str2` is a pointer to stack memory. Calling `free` on a pointer to stack memory will result in an error.

`str3` is a pointer to heap memory. Calling `free` on a pointer to heap memory is legal, so `free(str3);` is safe.

Grading: Each answer choice was graded independently, just like in the previous subparts.

Q2.8 (1 point) Printing the string.

■ `printf("%s\n", str1);` ■ `printf("%s\n", str3);`

■ `printf("%s\n", str2);` ☐ None of the above

Solution: `printf("%s\n", str1);` and `printf("%s\n", str2);` and `printf("%s\n", str3);`

Recall that `printf("%s\n", str1);` dereferences the `str1` pointer and prints out the string stored at that address (up until the null terminator).

Since `str1`, `str2`, and `str3` all point to a valid null-terminated string, it is safe to call `printf` on all three of them.

Grading: Each answer choice was graded independently, just like in the previous subparts.

Q2.9 (2 points) If this code was run on a little-endian system, what would `((uint32_t*) str1)[2]` evaluate to? Express your answer in hexadecimal, with the necessary prefix. Note that `uint32_t` refers to an unsigned 32-bit integer.

Solution: `0x00646C72`

This question requires you to think about endianness and how memory would be interpreted if casted to a different type.

First, note that `str1` contains the address of the bytes `Hello World`. These bytes are stored contiguously in memory; `H` is at the lowest address, and `d` is at the highest address. A null byte is stored immediately after `d` in memory.

Casting `str1` to `(uint32_t*)` doesn't change the fact that `str1` is still a pointer to the bytes `Hello World`, but those bytes are now being read as an array of `uint32_t` (32-bit = 4-byte unsigned integers), instead of an array of `char`s.

The `[2]` syntax says to dereference the pointer and look for the second element (zero-indexed) in the array. Each element in the array is 4 bytes long because of the cast. Thus the 0th element is bytes 0-4 (`Hell`), the 1st element is bytes 5-8 (`o Wo`), and the 2nd element is bytes 9-12 (`rld` and the null byte).

Finally, we need to interpret these four bytes (`rld` and the null byte) as a `uint32_t`. We can use the ASCII table to look up the bytes being stored in memory to represent these characters. From lowest to highest memory address, the bytes are `0x72` (`r`), `0x6C` (`l`), `0x64` (`d`), and `0x00` (null byte).

Because the system is little-endian, the most-significant byte is stored at the highest memory address. In other words, we should read the integer starting from the highest address and ending at the lowest address. This gives us `0x00646C72`.

Grading: 2 points for the correct hexadecimal value. 1 point for starting from the second character, not the second word: `0x206F6C6C`. 1 point for wrong endianness: `0x0x726C6400` or `0x20576F72`. For any answer (fully correct or partial), 1 point was deducted if only one byte of data was given instead of the full 4 bytes.

Q3 *I C a Scheme* **(20 points)**

Consider the following C code:

```
union ExtraStuff {                  typedef struct ConsCell {
    char a[5];                          void *car;
    uint16_t b;                         void *cdr;
    int c;                              union ExtraStuff extra;
    double d;                       } cons;
};
```

Consider the following function: `cons *map(cons *c, (void *)(*f)(void *));`

`map` takes a pointer to a `cons` struct `c` and a function pointer `f`.

If the `cons` struct pointer is `NULL`, `map` returns `NULL`. Otherwise, it does the following:

1. Allocate a new `cons` struct. `ret` is a pointer to this new struct.
2. Set the contents of the `extra` union in `ret` to be all zeros.
3. Set the `car` field in `ret` to the result of calling `f` on the `car` pointer in `c`.
4. Set `cdr` field in `ret` to the result of calling `map` recursively on the `cdr` pointer in `c`.

Q3.1 (18 points) Complete the following code by filling in the blanks. This code should compile without errors or warnings. Each blank is worth 2 points.

```
cons *map(cons *c, (void *) (*f) (void *)) {
    cons *ret;


    if ( _____________________ ) return ______________________;


    ret = malloc(____________________________________________);


    ________________________extra_________________________ = 0;


    _____________________car = _______________________________;


    _____________________cdr = _______________________________;
    return ret;
}
```

Solution:

```
cons *map(cons *c, (void *) (*f) (void *)) {
    cons *ret;
    if ( c == NULL ) return NULL;
    ret = malloc(sizeof(cons));
    ret->extra.d = 0;
    car = f(c->car);
    ret->cdr = map(c->cdr, f);
    return ret;
}
```

Much of this question involves reading the provided description of the function and converting it to valid C code.

Blank 1 checks if the `cons` struct pointer is `NULL`.

Common mistakes:

- `*c == NULL` would be incorrect here, because we want to check if the pointer (i.e. the address stored in the variable) is NULL. We don't want to see if the value stored at the address in `cons` is NULL.
- `cons == NULL` would be incorrect here, because `cons` is the type of the variable, not the name of the variable itself. For the same reasons that you wouldn't write something like `int == NULL`, you can't write `cons == NULL` here.

Accepted solutions:

- `c == NULL`
- `!c`

Blank 2 returns `NULL` if `c` is NULL.

Accepted solutions:

- `NULL`
- `c` (if the first blank is correct, then `c` must be `NULL` if the if check passes)

Blank 3 asks you to input an argument to `malloc` that will allocate enough space for a new `cons` struct. The simplest and best way to do this is to use `sizeof(cons)`, which will return the number of bytes that a single `cons` struct takes up in memory.

Since we did not say anything about allocating the minimum space necessary, it was okay if you allocated more space than needed. For example, you could allocate space for two structs with `2 * sizeof(cons)`. You could also calculate the size of the struct by yourself (16 bytes) and input any number greater than or equal to 16.

Accepted solutions:

- `sizeof(cons)`
- `sizeof(struct ConsCell)` (equivalent because of the `typedef` statement)

- Any expression that is guaranteed to evaluate to at least 16, e.g. hard-coding the number `16`
- Partial credit: `sizeof(struct cons)` (invalid syntax)
- Partial credit: `sizeof(ConsCell)` (invalid syntax)

Blanks 4, 6, and 8 are immediately followed by the names of the struct fields: `car`, `cdr`, and `extra`. Also, the question says that the next steps are to set each of the fields in `ret`. We know that `ret` is a pointer to a struct from the provided line of code. To access a struct field from a pointer to the struct, we must first dereference the struct, and then access the correct field. The simplest and best way to do this is to use the C shorthand `ret->` notation. Recall that the `->` syntax will dereference a struct pointer and then access a struct field.

Common mistakes:

- `ret.` is incorrect because it fails to dereference the struct pointer. If `ret` was a struct (not a struct pointer), then you would be able to use this syntax.
- `*ret->` and variations are incorrect because they dereference the `ret` pointer twice. The `*` syntax dereferences the pointer once to access the struct, and then the `->` syntax dereferences the struct, which results in an error.

Accepted solutions:

- `ret->`
- `(*ret).` (note that the parentheses are required, because the dereference operator must happen before the struct access operator)
- Partial credit: `*ret.` (by C's order of operations, the struct access operator would take precedence over the dereference operator here, which is incorrect; you must dereference the pointer before accessing a struct field)

Blank 5 sets the contents of the `extra` union to be all zeros. This was probably the hardest blank in this question! The key observation is that the largest element in the union is `double d`, which is 8 bytes = 64 bits. (`char a[5]` is 5 bytes = 40 bits, `uint16_t b` is 2 bytes = 16 bits, and `int c` is 4 bytes = 32 bits.) Thus, if we set the largest element in the union to 0, all the other union elements using that same memory will also be set to 0.

Accepted solutions:

- `d.`
- Partial credit: `.double` (correctly identified the double, but used the type instead of the field name)
- Partial credit: `->d` (correctly identified the double, but tried to dereference the struct to reach the union field)

Note that the variables in the union were rearranged in some exam versions. For example, one version had `double a`, which would make the correct answer to this subpart `.a`, not `.d`.

Blank 7 sets the `car` field to the result of calling `f` on the `car` pointer in `c`. Remember that `c` is a struct pointer, so we have to use `->` syntax to dereference the struct and access the `car` pointer field inside the struct.

Accepted solutions:

- `f(c->car)` or `f((*c).car)` (note that `(*c).car` is equivalent to `c->car`)
- `(*f)(c->car)` or `(*f)((*c).car)` (C lets you call a function pointer with or without the dereference symbol, as long as you add parentheses to dereference the function pointer first)
- Partial credit: `*f(c->car)` or `*f((*c).car)` (by C's order of operations, the function will be called, and the pointer returned by the function will be dereferenced, which is not what we want; we want to assign the pointer to the `ret->car` pointer)
- Partial credit: `f(*c.car)` (same reasoning as why `*ret.` fails in blanks 4, 6, and 8)

Blank 9 calls `map` recursively on the `cdr` pointer in `c`. Note that `map` takes two arguments, so the function pointer `f` must be provided as the second argument. Again, remember that `c` is a struct pointer, so we have to use `->` syntax to dereference the struct and access the `cdr` pointer field inside the struct.

Accepted solutions:

- `map(c->cdr, f)`
- `(*map)(c->cdr, f)` (same reasoning as why dereferencing the function pointer is okay in blank 8)
- Partial credit: `*map(c->cdr, f)` (same reasoning as blank 8)
- Partial credit: `map(c->cdr)` (forgot to supply `f` argument)
- Partial credit: `map(c->cdr, (void*) (*f) (void*))` (supplied the `f` argument with the type in addition to the variable)

Grading: Each blank was graded independently. 2 points for any accepted solution. 1 point for any partial credit solution listed. No credit for any of the common mistakes listed.

Q3.3 (2 points) On a 32-bit architecture, what is `sizeof(cons)`?

Solution: 16

This question asks you to calculate the memory that a `cons` struct takes up in memory.

First, remember that C calculates sizes in terms of bytes, so `sizeof` will return the size of the struct in bytes (not bits).

On a 32-bit architecture, a pointer is 32 bits = 4 bytes long. This means that the pointers `void *car` and `void *car` each take up 4 bytes in memory, for a total of 8 bytes.

The size of a union is the size of the largest element in the union. The largest element in the union is the `double` field, which is 8 bytes = 64 bits. (`char a[5]` is 5 bytes = 40 bits, `uint16_t b` is 2 bytes = 16 bits, and `int c` is 4 bytes = 32 bits.)

Note that the actual union (not a pointer to the union) is stored in the struct. If the struct had instead contained `union ExtraStuff *extra`, then the pointer would take up 4 bytes in memory.

In total, the struct takes up 8 bytes for the void pointers and 8 bytes for the union, for a total of 16 bytes.

Note that no struct or union padding is necessary here, because every field size is a multiple of 4 bytes and thus is already word-aligned.

`Grading`: 2 points for the correct size (16). The answer "16 bytes" was accepted for full credit, even though the `sizeof(cons)` expression in C would only return the number "16", not the text "16 bytes".

Q4 *To Float or Not to Float* (20 points)

Consider a floating point system that has 16 bits with 7 bits of exponent and an exponent bias of -63, which otherwise follows all conventions of IEEE-754 floating point numbers (including denorms, NaNs, etc.). In this question, we will compare this system to an unsigned 16-bit integer system.

Q4.1 (4 points) What is the value of floating point number `0xC2C0` in decimal?

Solution: -14

First, we convert the hexadecimal number to binary: `0xC20 = 0b1 1000010 11000000`. We've added spaces to distinguish the sign bit, the 7 exponent bits, and the remaining 8 significand bits.

The sign bit is 1, so the number is negative.

The unsigned exponent bits `1000010` are equal to 66 in decimal. Adding the exponent bias gives us an exponent of 66-63 = 3.

The significand bits are `11000000`. Adding the implicit 1 gives us `1.11000000`. Since there is a 1 in the ones digit, the 1/2s digit, and the 1/4s digit, this number is $1 + 1/2 + 1/4 = 1.75$.

Putting these together, the final number is -$2^3 * 1.75 = -14$.

Other versions of this exam gave different hexadecimal values, but the conversion process is the same:

- `0xC220` $= -9$
- `0xC280` $= -12$
- `0xC260` $= -11$

Grading: 1 point for sign, 1.5 points each for correct exponent and significand

Q4.2 (1 point) Which representation has more representable numbers? Count +0, -0, $+\infty$, and $-\infty$ as 4 different representable numbers.

○ The floating point number

[✓ The unsigned 16-bit integer]

○ Both systems can represent the same number of values

Solution: There are a total of 2^{16} bit patterns in either system, since 16 bits store 2^{16} possible values. This means we only need to think about which bit patterns as numbers and which bit patterns are not numbers.

In the integer system, every bit pattern represents a different number.

In the floating point system, some bit patterns represent NaNs, which are not numbers ("NaN" stands for "Not a Number").

Since the floating point system has some bit patterns that aren't numbers, and the integer system has no bit patterns that aren't numbers, the integer system can represent more numbers.

Q4.3 (3 points) How many more numbers can be represented? Write 0 if both systems can represent the same number of values.

Solution: 510

From the previous part, we know that the only difference in representable numbers comes from NaNs. Thus the question is asking how many NaNs exist in the floating point system.

Recall that NaNs are represented by all ones in the exponent, any value in the sign bit, and any non-zero value in the significand (a zero in the significand would represent infinity).

The significand is 8 bits, so there are $2^8 - 1$ possible non-zero values in the significand. There are 2 possible bits in the sign bit. In total, there are $2 \times (2^8 - 1) = 510$ NaNs.

Thus, the unsigned integer can store 510 more unique numbers.

Grading: Half credit was awarded for forgetting the infinity (512), and forgetting negative NaNs (255), but not both.

Q4.4 (4 points) Out of all numbers representable by this floating point system, what is the largest number that can also be represented as an unsigned 16-bit integer?

Solution: $2^{16} - 2^{7} = 65408$

The unsigned number can represent any nonnegative integer less than 2^{16}, so we're looking for the largest integer less than 2^{16} that can be represented by the floating point number. To do this, we can try to create a 16-bit integer with the floating point number, and how we can maximize the number created through this process.

The significand has 8 bits plus the implicit 1 (e.g. `1.1111 1111`), so to represent a 16-bit integer, we would need an exponent of 15 to create `1 1111 1111 0000 000`.

Note that the lower 7 bits of any number created in this process will always be 0, because they are not part of the significand. Thus all we can do to maximize this number is adjust the significand to be as large as possible. The largest significand would be all 1s, as shown above.

In other words, the value we want is `0b1.11111111` $\times 2^{15}$, which is equal to $2^{16} - 2^{7} = 65408$.

Grading: Half credit was awarded for $2^{16} - 1$ and $2^{16} - 2^{8}$.

Q4.5 (4 points) What is the smallest positive number representable by this floating point system that isn't representable by the unsigned 16-bit integer?

Solution: 2^{-70}

Floating point numbers can represent fractional numbers between 0 and 1, but integers cannot represent fractional numbers between 0 and 1. Thus we are looking for the smallest positive number representable by the floating point number.

The smallest positive numbers representable in floating point are the denorms. The smallest denorm can be obtained by using a denorm exponent of 0 and the smallest possible mantissa. This gives us 2^{-70}.

Grading: Half credit was awarded for 2^{-71}.

Q4.6 (4 points) What is the smallest positive number representable by the unsigned 16-bit integer that isn't representable by this floating point system?

Solution: $2^9 + 1$

Intuitively, floating point numbers can represent all smaller integers 1, 2, 3, etc. but eventually, there will be an integer that the floating point number skips over (the gaps between numbers get wider as the number gets larger). Thus we are looking for the smallest positive integer that is not representable by the floating point number.

If we make the exponent exactly equal to the number of bits in the significand, then we can use the entire significand to represent a positive integer. The significand has 8 bits, so we can set the exponent to 71-63=8 and use the 8 bits of the significand and the implicit 1 to represent all integers up to 2^9.

After 2^9, the exponent must be increased to 72-63=9. This will add a 0 to the end of the bits of the significand, which means that odd numbers are no longer representable after 2^9. Thus the smallest positive integer that cannot be represented by the floating point number is $2^9 + 1$.

Grading: Half credit was awarded for $2^8 + 1$.

Q5 *A RISC-y Program* (20 points)

In addition to storing the `ra` register and the `s` registers, the stack can also store local variables. You have access to the following labels defined externally:

- `Password`: a pointer to a statically-stored string `"secretpass"`
- `Get20chars`: A function defined as follows:
  - Input: `a0` is a pointer to a buffer
  - Effect: Reads characters from stdin, and fills the buffer pointed to by `a0` with the read data, null-terminating the string. Your code may assume that the input is at most 19 characters, not including the null-terminator.
  - Output: None

The function `verifypassword` is defined as follows:

- Input: No register input; however, the function receives a string input from stdin.
- Output: a0 returns 1 if the input from stdin is exactly `"secretpass"`, and 0 otherwise.

Q5.1 (12 points) Complete the function `verifypassword`. Each line contains exactly one instruction or pseudoinstruction.

```
 1: verifypassword:
 2:     addi sp, sp, -24 # Make space for a 20-byte buffer

 3:     sw ______________ 20(sp)

 4:     ________________________________________

 5:     jal ra Get20chars

 6:     ______________ t0 Password

 7:     ______________ t1 sp

 8: Loop:

 9:     ______________ t2 0(t0)

10:     ______________ t3 0(t1)

11:     ________________________________________

12:     ________________________________________

13:     addi t0 t0 ______________

14:     addi t1 t1 ______________

15:     ________________________________________

16: Pass:

17:     ________________________________________

18:     ________________________________________

19: Fail:

20:     ________________________________________

21: End:

22:     ________________________________________

23:     ________________________________________

24:     ________________________________________
```

Solution:

```
 1: verifypassword:
 2:     addi sp, sp, -24 # Make space for a 20-byte buffer
 3:     sw ra 20(sp)
 4:     mv a0, sp
 5:     jal ra Get20chars
 6:     la t0 Password
 7:     mv t1 sp
 8: Loop:
 9:     lb t2 0(t0)
10:     lb t3 0(t1)
11:     bne t2, t3, Fail
12:     beq t2, x0, Pass
13:     addi t0 t0 1
14:     addi t1 t1 1
15:     j Loop
16: Pass:
17:     li a0, 1
18:     j End
19: Fail:
20:     li a0, 0
21: End:
22:     lw ra, 20(sp)
23:     addi sp, sp, 24
24:     jr ra
```

Line 3: First, we note that Line 2 allocated 24 bytes of space on the stack. Line 3 is storing a 4-byte register value to the stack space that was just allocated. This looks a lot like the function prologues that we've seen before in projects and labs!

A quick refresher on function prologues and epilogues: according to calling convention, there are some callee-saved registers whose original values must be preserved after the function returns. To achieve this, we usually store the original register values on the stack at the beginning of the function (the prologue), and we restore the original register values at the end of the function (the epilogue).

In the provided code, there is only one callee-saved register already being used: `ra` at line 5. All the other registers being used are `t` registers which do not need to be saved.

Thus we should save the original value of `ra` on the stack so that we can restore it later.

Accepted solutions:

- `ra`
- `x1`

Line 4: First, we note that Line 5 is calling the `Get20chars` function. Before we call a function, we need to make sure that its arguments are properly loaded into the argument (`a`) registers.

According to the question, the `Get20chars` function takes in one argument, in the `a0` register. Thus, this line is probably going to involve putting something into the `a0` register.

According to the question, the argument in `a0` is a pointer to a buffer in memory, which the function will fill up with (at most 19) input characters. In other words, the `Get20chars` function requires 20 bytes of space in memory to store the input characters, so `a0` needs to store the address of some 20-byte space in memory where we can store a string. Where do we have 20 bytes of space in memory right now? We allocated 24 bytes on the stack in line 2, and we used 4 of those bytes in line 3, so we have 20 bytes on the stack that we can use! (The first sentence in this question mentions storing local variables on the stack, and the comment on Line 2 reminds you that there is a 20-byte buffer on the stack, to try and direct you toward this solution.)

What is the address of this space on the stack? Remember that the stack pointer `sp` always contains the address of the bottom of the stack. Since the stack grows down to make space (see line 2), the 20 bytes of space is located at the bottom of the stack, and the address of this space is the address stored in `sp`. Note that in Line 3, we stored the saved `ra` register value at the top of the 24-byte space on the stack, so the 20-byte buffer starts at the very bottom of the stack, where `sp` is pointing.

Common mistakes:

- Load instructions like `lw a0 0(sp)` would not work here because they would dereference the address in `sp` and load a value from stack memory into `a0`. In this question, we want `a0` to contain the address of the buffer on the stack, not a value loaded from the stack.
- Store instructions like `sw a0 0(sp)` would not work here because they store the value in `a0` onto the stack. In this question, we want to put the argument into `a0`, and store instructions would not modify the value in the register.
- `la a0, sp` would not work because the register `sp` is not in memory and does not have an address.

Accepted solutions:

- Any instruction that puts the value of `sp` in `a0`, including: `mv a0, sp` or `addi a0, sp, 0` or `add a0, x0, sp`
- Partial credit: `mv a0 (0)sp` (incorrect syntax for `mv`)
- Partial credit: `addi a0, sp, 4` (incorrectly assumed that the saved value of `ra` was below the 20-byte buffer on the stack and used the address 4 bytes above the bottom of the stack instead)

Line 6: Lines 6 and 7 initialize `t0` and `t1`, which we can see will be used repeatedly in the loop. We initialize `t0` to be the address of the start of the `Password` string, and we initialize `t1` to be the address of the start of the user's input string. This will let us compare each character of the strings in a loop later.

One way to reason this out: this line is putting a value into `t0`. We see at line 9 that `t0` will be used as an address (because it's in parentheses). `la` is an instruction that loads addresses into a register.

Another way to reason this out is to note that the `Password` label is involved in this instruction, so we must use an instruction that uses a label. Branches and jumps don't make sense here because we aren't implementing any loops or if/else conditions yet, so by process of elimination, we should use `la` here.

Common mistakes:

- Load instructions like `lw` would not work here because they do not use labels. This would lead to a syntax error.

Accepted solutions: `la`

Line 7: One way to reason this out is to note that only two registers are provided, so we probably want a pseudoinstruction that uses only two registers. `mv` is the only logical pseudoinstruction to use here.

Accepted solutions: `mv`

Lines 9-10: We want to use the loop to compare each character of the input string with each character of the provided password string. In lines 6-7, we initialized `t0` and `t1` with the addresses of the strings. Now we need to dereference those addresses to load each character of the string from memory. Since each character is one byte long, `lb` is the appropriate load instruction to use here.

One way to reason this out is to note that the `0(t0)` syntax is used only in load and store instructions. The addresses in `t0` and `t1` already contain data (the hardcoded password string and the input string, respectively), so we don't want to store to those addresses. This means we must be loading from those addresses.

Accepted solutions:

- `lb`
- `lbu`

Line 11: Now that we've loaded a character from each string from memory, we need to check if those characters match. If they don't match, we can stop checking characters in a loop and jump to the fail case.

Accepted solutions:

- `bne t2, t3, Fail` or `bne t3, t2, Fail`
- Partial credit: Using the address `t0` instead of the value `t2`, and/or using the address `t1` instead of the value `t3` (e.g. `bne t0, t1, Fail`)

Line 12: If we've reached the end of the string without branching to the Fail case, then we've checked that every character is equal, and we should enter the Pass case.

Remember that strings end with a null terminator, so if either loaded character is 0 (null byte), then we know that we reached the end of both strings. Note that Line 11 will check for us that both strings have reached the null terminator, since it checks that every character is equal. This is why the Pass branch must come after the Fail branch–if they were swapped, then this function would jump to Pass whenever the shorter string terminates, even if the longer string still has extra characters left.

Accepted solutions:

- `beq t2, x0, Pass` or `beq t2, zero, Pass`
- `beq t3, x0, Pass` or `beq t3, zero, Pass`
- Partial credit: Using the address `t0` instead of the value `t2`, and/or using the address `t1` instead of the value `t3` (e.g. `beq t0, x0, Pass`)
- Partial credit: Swapping lines 11 and 12 (only applies if both lines 11 and 12 are fully correct)
- Partial credit: `NULL` or `\0` or `0x00` instead of `x0` or `zero` (incorrect syntax for a null byte in RISC-V)

Lines 13-14: Once we finish checking a pair of characters, we need to increment the addresses to the next character. The next time the loop runs, the load instructions will now load the next character in the string.

Recall that characters are 1 byte, and RISC-V indexes memory by bytes, so we should increment the addresses in `t0` and `t1` by 1.

Accepted solutions: `1`

Line 15: If we didn't jump to the Fail or Pass case, after we increment the addresses, we need to keep executing the loop and check the next character. To do this, we should jump back to the start of the loop. This is not a function call, and we don't care about saving our return value, so a simple jump instruction is sufficient.

There is an alternate solution here that combines Line 12 (pass check) and Line 15 (jumping to loop). As in the standard Line 12, we check if `t2` (or `t3`, which must be equal at this point) is equal to 0 (the null byte). If so, we've reached the end of the string, so we should continue execution normally to the Pass case immediately following the loop. If not, then we need to loop again, so we jump back to the start of the loop. This logic is captured in `bne t2, x0, Loop`. With this alternate solution, Line 12 can be blank (or any innocuous instruction that doesn't affect program execution, e.g. `add t6, x0, x0`).

Accepted solutions:

- `j Loop`
- `jal x0 Loop`
- `jal Loop` (this is not the best answer because it unnecessarily saves a useless return address into `ra`, but it won't break the program because the saved value of `ra` is restored in the function epilogue)
- Partial credit: `jr Loop` or `jalr Loop` (incorrect jump instructions, because these jump to addresses in registers, not a label)
- `bne t2, x0, Loop` or `bne t3, x0, Loop` (see alternate solution above)

Line 17: If the check passes, we should load 1 into the return value register `a0`.

One way to reason this out: This line of code only executes if we jump to the Pass label. The only line of code that is unique to the case where the check passes is putting 1 in the return value register, indicating success. Other cleanup lines of code (e.g. function epilogue, return instruction) are common to both the Pass and Fail cases, so they are more suited to be in the End label.

Accepted solutions:

- Any instruction that puts the value 1 in a0, including: `li a0, 1` or `addi a0, x0, 1`

Line 18: If the check passes, we don't want to execute the code under the Fail case, so we should jump to the End label. Like in Line 15, this is not a function call, and we don't care about saving our return value, so a simple jump instruction is sufficient.

Accepted solutions:

- `j End`
- `jal x0 End`
- `jal End` (this is not the best answer because it unnecessarily saves a useless return address into ra, but it won't break the program because the saved value of ra is restored in the function epilogue)
- Partial credit: `jr End` or `jalr End` (incorrect jump instructions, because these jump to addresses in registers, not a label)

Line 20: If the check fails, we should load 0 into the return value register a0. The reasoning from Line 17 also applies to solving this blank.

Accepted solutions:

- Any instruction that puts the value 0 in a0, including: `li a0, 0` or `addi a0, x0, 0` or `mv a0, x0` or `add a0, x0, x0`

Line 22: At the end of the function, we need to undo any setup we did at the beginning of the function. In particular, we should write a function epilogue to restore all the saved register values on the stack.

Since we stored the original value of ra 20 bytes above the stack pointer in the prologue, we should load from that same location in memory to restore ra.

Accepted solutions:

- `lw ra, 20(sp)`
- Partial credit: `lw ra, 24(sp)` (off-by-one)

Line 23: As part of the function epilogue, we should also increment the stack pointer back up to delete the space we allocated earlier in the function.

In Line 2, we decremented the stack pointer by 24, so we should now increment the stack pointer by 24.

Accepted solutions:

- `addi sp, sp, 24`
- Partial credit: Swapping lines 22 and 23 (only applies if both lines 22 and 23 are fully correct)

Line 24: Finally, we need to return from the function.

Accepted solutions:

- `ret`
- `jr ra`
- `jalr x0, ra, x0`

Grading: Each blank was graded independently, except for cases where a correct answer was placed one line before or after its intended location. Full points for any accepted solution. Half points for any partial credit solution listed. No credit for any of the common mistakes listed.

These lines were worth 0.5 points: 3, 5, 6, 9, 10, 13, 14, 17, 20, 24

These lines were worth 1 point: 4, 11, 12, 15, 18, 22, 23

Q5.2 (4 points) Translate `addi sp, sp, -24` to its machine-language hexadecimal representation, with the appropriate prefix.

Solution: `0xFE810113`

Start with the opcode: `addi` has opcode `0010011`, so we now have:

`-------------------------0010011`

We know that this is an I-type instruction now, so we can follow that format on the green card. We can fill in the funct3 for `addi`, which is `000`:

`-----------------000-----0010011`

Next, we can fill in both registers `rd` and `rs1`, since they are both `sp = x2 = 00010`:

`------------00010000000100010011`

Finally, we can fill in the immediate, which is `-24`. To write this in two's complement binary, start by writing 24 in unsigned binary: `0b00011000`. Then flip the bits: `0b11100111`, and add one: `0b11101000`. Finally, sign-extend this value to 12 bits: `0b111111101000`.

`11111110100000010000000100010011`

Converting this to hexadecimal gives us `0xFE810113`.

Grading: 4 points for a completely correct answer. If the answer was not fully correct, we gave partial credit as follows:

- 1 point for the correct opcode (hex instruction looks like `XXXXXX13` or `XXXXXX93`) and func3 (hex instruction looks like `XXXX0XXX` or `XXXX8XXX`)
- 1 point for the correct registers (hex instruction looks like `XXX1K1KX`, where `K < 8`)
- 2 points for the correct immediate (hex instruction starts with `0xFE8`)
- 1 point for an immediate of 24 (hex instruction starts with `0x018`)
- 2 points for the correct answer in binary

Q5.3 (4 points) Assume that `verifypassword` is located at `0x00001000`, and `Get20chars` is located at `0x00000f00`, and that line 4 is exactly one instruction (not a pseudoinstruction). Translate the line `jal ra Get20chars` to its machine-language hexadecimal representation, with the appropriate prefix.

Solution: `0xEF5FF0EF`

Start with the opcode: `jal` has opcode `1101111`, so we now have:

`-------------------------1101111`

We know that this is an I-type instruction now, so we can follow that format on the green card. We can fill in the register `rd`, which is `ra = x1 = 00001`:

`--------------------000011101111`

Next, we need to calculate the immediate. Remember that addresses are PC-relative, so we need to calculate the distance in memory between the current instruction and the `Get20chars` function.

According to the assumption, `verifypassword` starts at `0x00001000`, and `Get20chars` starts at `0x00000f00`. The distance between these two functions is `0x00001000 - 0x00000f00 = 0x100 = 256` bytes. In other words, from the beginning of `verifypassword`, we have to jump 256 bytes backwards to reach the `Get20chars` function.

However, we aren't starting exactly at the beginning of the `verifypassword` function. Labels aren't stored in memory, so Line 2 is the first instruction in the function. The `jal` instruction at Line 5 is 3 instructions = 12 bytes after Line 2 (using the assumption that Line 4 is not a pseudoinstruction). Thus we actually have to jump another 12 bytes backwards to reach the start of the function, then another 256 bytes backwards to reach `Get20chars`, for a total of 256+12=268 bytes backwards.

Now that we have our immediate of -268 (negative because we are jumping to a lower address in memory), we can convert it to two's complement binary. Start by writing 268 in unsigned binary: `0b000100001100`. Then flip the bits: `0b111011110011`, and add one: `0b111011110100`. Finally, sign-extend this value to 21 bits: `0b111111111111011110100`.

Now we have to rearrange this number to fit in the instruction format. Start with the 20th bit (zero-indexed) `0b111111111111011110100`:

`1-------------------000011101111`

Then encode bits 10 through 1 `0b111111111111011110100`:

`11101111010---------000011101111`

Then encode bit 11 `0b111111111111011110100`:

`111011110101--------000011101111`

Then encode bits 19 through 12 `0b111111111111011110100`:

`11101111010111111111000011101111`

Note that the 0th bit was never encoded in the instruction, because it is guaranteed to always be 0.

Converting this to hexadecimal gives us `0xEF5FF0EF`.

Grading: 4 points for a completely correct answer. If the answer was not fully correct, we gave partial credit as follows:

- 0.5 points for the correct opcode (hex instruction ends in `EF` or `6F`)
- 0.5 points for the correct register (hex instruction looks like `XXXXX0KX`, where $\texttt{K} \geq 8$)
- 3 points for an immediate of -268 (hex instruction starts with `EF5FF`)

If the immediate was not correct, we gave partial credit for that rubric item as follows:

- 1.5 points for an immediate of 268 (hex instruction starts with `10C00`)
- 1.5 points for not ignoring the 0th bit (hex instruction starts with `DE9FF`)
- 1.5 points for an immediate of -272 (hex instruction starts with `EF1FF`)
- 1.5 points for an immediate of -244 (hex instruction starts with `F0DFF`)
- 1.5 points for an immediate of -12 (hex instruction starts with `FF5FF`)

Q6 *Testception* **(20 points)**

Recall the following information from the previous question:

You have access to the following labels defined externally:

- `Password`: a pointer to a statically-stored string "secretpass"
- `Get20chars`: A function defined as follows:
  - Input: a0 is a pointer to a buffer
  - Effect: Reads characters from stdin, and fills the buffer pointed to by a0 with the read data, null-terminating the string. Your code may assume that the input is at most 19 characters, not including the null-terminator.
  - Output: None

The function `verifypassword` is defined as follows:

- Input: No register input; however, the function receives a string input from stdin.
- Output: a0 returns 1 if the input from stdin is exactly "secretpass", and 0 otherwise.

Propose a suite of 4-6 tests you would use to verify that an implementation of this function works properly. Your test suite does not need to be comprehensive, but each test should test something different. We will only count your best 4 tests when grading.

Each test should consist of a list of inputs, expected outputs, and a one-sentence justification for why this test is useful. A useful test without proper justification will not receive credit.

Some cases you can test include:

- A generic case that returns true
- A generic case that returns false
- Other calling convention checks
- Edge cases

`rand0` through `rand11` is a constant set of 12 randomly generated numbers, which you can use in your tests.

A valid example is provided below; you may not reuse the example.

	Test 0

Input(s)	`a0-a7 = rand0-rand7`
	`stdin = "secretpass"`

Output(s)	`a0 = 1`
Justification	Check for using unset `a` registers.

	Test 1
Input(s)	
Output(s)	
Justification	

	Test 2
Input(s)	
Output(s)	
Justification	

	Test 3
Input(s)	
Output(s)	
Justification	

	Test 4
Input(s)	
Output(s)	
Justification	

	Test 5
Input(s)	
Output(s)	
Justification	

	Test 6
Input(s)	
Output(s)	
Justification	

Solution:

Grading: Each test was graded independently. In order to qualify for points, a test needed to have a valid purpose, a reasonable justification, no errors, and not be redundant with another test. Each correct test was awarded 5 points. No partial credit was given on individual tests, in part because only 4 out of 6 tests needed to be correct.

A list of common valid tests were as follows.

Generic tests:

- One generic true test (ex. Input: `stdin = "secretpass"`, Output: `a0 = 1`)
- One generic false test (ex. Input: `stdin = "youshallnotpass"`, Output: Output: `a0 = 1`)

Calling convention tests:

- A test on t register calling conventions (ex. Input: `t0-t6 = rand0-rand6`,

  `stdin = "secretpass"`, Output: `a0 = 1`)
- A test on s register calling conventions (ex. Input: `s0-s11 = rand0-rand11`,

  Output: `s0-s11=rand0-rand11`)
- A test to confirm that the sp was restored (ex. Input: `s0 = sp`, Output: `sp=s0`)
- A test to confirm that data on the stack is unchanged (ex. Input: `sp -=4, sw rand0 0(sp)`, Output: `0(sp) = rand0`)

Note that any calling convention test could be listed twice; once for a true case, and once for a false case (since a solution could feasibly forget calling convention on only one branch).

Edge case tests:

- Empty string (ex. Input: `stdin=""`)
- Password stays constant after the function call (ex. Output: `Password="secretpass"`)
- A string of length 19 or more doesn't overwrite needed stack data

  (ex. Input: `stdin="nineteen characters"`)
- "secretpass" is a strict prefix of the input (ex. Input: `stdin="secretpassword"`)
- The input is a strict prefix of "secretpass" (ex. Input: `stdin="secret"`)
- Edge Case: The input differs only on the first character (ex. Input: `stdin="mecretpass"`)
- The input is the same length as "secretpass" (ex. Input: `stdin="passsecret"`)
- The input has punctuation or numbers (ex. Input: `stdin="12!$"`)
- The input is capitalized (ex. Input: `stdin="SeCreTPass"`)

A list of common tests which **weren't** considered valid were as follows:

- S-register or stack tests without explicitly checking values in the output

- Setting the sp to a random location. A correct solution may use the stack, and setting the sp to a random location is a good way of overwriting somewhere randomly (or segfaulting).
- Setting the ra to a random location. jal sets the ra to the line after the function call anyway.
- Any test involving checking for restoring t or a registers, or the ra. This is not mandated by calling convention.
- A test on a register calling conventions. This was already provided as an example, and thus could not be used unless justification explicitly put it under a test of a registers in the false case.
- Tests which attempted to modify Password or send input via a0. The specification of this question was that verifypassword return true when the input was "secretpass", not when it matched the string stored in Password or in a0.
- Tests which attempted to access a register that did not exist. In particular, any test which attempted to set the nonexistent register t7 was automatically incorrect.
- Tests which relied on the specific implementation of verifypassword used in question 5 (such as checking t3 or t4 for specific values).
- Tests which attempted to put string data after a null terminator, or which tried to send a string without a null terminator. C strings are by definition ended at the null terminator, and stdin generally doesn't automatically add a terminator to the end of its data anyway, so these tests are redundant with any other tests written.
- Tests which tried to check that input differentiated the null terminator from the ASCII character "0"; the null terminator is a character in its own right with ASCII value 0, and is written as `\x2F\x30` only for reading convenience. The two are entirely different, so such a justification doesn't really work.
- Any test where the expected output was Error. This question didn't specify any case in which a correct solution would error, so no input should expect to cause an error.
- Any other test which didn't adhere to spec, could return a failure on a correct solution, or had no justification.