Your Name (first last)	# UC Berkeley Fall 2019 CS61C Quest	SID
← Name of person on left (or aisle)		Name of person on right (or aisle) →

Q1) [10 Points] **Negate** the following **nibble *binary/hex*** numbers, or write N/A if not possible. Remember to write your answer in the appropriate base. (A nibble is 4 bits)

(Unsigned) `0b0101`	*(Bias = -7)* `0b0100`	*(Bias = -7)* `0xF`	*(Two's Comp)* `0b1100`	*(Two's Comp)* `0xA`
`0b` **N/A**	`0b1010`	`0x` **N/A**	`0b0100`	`0x6`

...scratch space below...

(unsigned) 0b0101) We want to negate this value but we are using an unsigned representation so this is impossible, so the answer is N/A.

(Bias = -7) 0b0100) The value given's unsigned representation is 4. We we will plug it into the equation given in question 3: unsigned + bias value = final value, thus 4 + -7 = -3. We want to invert this so we want to represent the value 3. To find our unsigned representation, we can just plug the values back into the above equation: unsigned + -7 = 3 ⇒ unsigned = 10 thus 0b1010.

(Bias = -7) 0xF) 0xF == 0b1111. We will do the same as above: unsigned + bias value = final value ⇒ 15 + -7 => 8. (we also might have remembered that the standard biased notation gives the extra number to the positive side, so the range is [-7,8] and 0b1111 is the biggest bias number so it's 8) So the value we need to invert is 8, so we want to represent -8. The issue is since our bias is only -7, we cannot represent -8 even if we have our unsigned value as zero, so the answer is N/A.

(Two's Comp) 0b1100) For twos complement, we do not need to worry about converting this to decimal. We can use the trick of flipping the bits and adding one thus: ~0b1100 + 1 = 0b0011 + 1 = 0b0100

(Two's Comp) 0xA) First we have to convert this value into binary. We know that A = 10 in decimal thus the binary representation is 0b1010. Now we can apply the trick described for the last question: ~0b1010 + 1 = 0b0101 + 1 = 0b0110. We can now convert this back to hex, thus 0x6.

Q2) [6 Points] Which of the following sums will yield an **arithmetically incorrect result** when computed with **two's complement nibbles**?

[✓ Correct] Incorrect	[✓ Correct] Incorrect	Correct [✓ Incorrect]
`0xD + 0xE + 0xF`	`0x7 + 0x8`	`0x3 + 0x5`

...scratch space below...

For all of these problems, we are given nibbles which are just 4 bits. We are interpreting them as Two's Complement. This means we can only represent the values in the range [-8, 7]. Thus if we get a result which goes beyond this, we cannot represent this.

0xD + 0xE + 0xF) -3 + -2 + -1 = -6, which is in our range, thus the result is correctly representable.

0x7 + 0x8) 7 + -8 = -1 which is in our range, thus the result is correctly representable

0x3 + 0x5) 3 + 5 = 8 which is NOT in our range, thus we cannot represent it correctly, so it is incorrect.

Room

Q3) [12 Points] For each of the following representations, what is the *fewest number of bits* needed to cover the given range, which is inclusive of the endpoints (e.g., [1, 4] is the numbers 1, 2, 3 and 4). Write "N/A" if it is impossible. For the **Bias *Value*** (final value = unsigned + bias value), we'll let YOU specify whatever offset you wish to minimize the total number of bits needed for the Bias encoding.

Range	Unsigned	One's Comp	Two's Comp	Sign&Mag	Bias	Bias *Value*
`[  0, 10 ]`	4	5	5	5	4	0
`[ -4, -1 ]`	N/A	4	3	4	2	-4
`[  1,  4 ]`	3	4	4	4	2	1

...scratch space below...

Here are the ranges of each of the representations given n bits:
Unsigned: [0, $2^n–1$]. So [0,10] needs 4 bits [0,15] with 5 unused bit patterns 11-15, [-4,1] is impossible, and [1,4] needs 3 bits [0,7] with {0,5,6,7} bit patterns unused.
One's Complement Range: [-($2^{n–1}$–1), $2^{n–1}$–1]. [1,4] needs 4 bits [-7,7]
Two's Complement Range: [-$2^{n–1}$, $2^{n–1}$–1]. [-4,1] needs 3 bits [-4,3], but [1,4] needs 4 bits [-8,7].
Sign&Mag: [-($2^{n–1}$–1), $2^{n–1}$–1]. [0,10] needs 5 bits [-15,15], and [-4,1] needs 4 bits [-7,7]
Bias: [Bias Value, 2^n–1 + Bias Value]. A Bias Value of 0 is just unsigned, so that's 4 bits [0,15].To do -4,1, many answers work, but the simplest is Bias Value = -4, and only 2 bits are needed to represent [-4,-1] -- nice and efficient! This is the same idea with [1,4], if you only have 2 bits, you'd better bias to the smallest number to fit the representation perfectly, so Bias Value = 1.

Room

For this page, assume all `mallocs` are successful, all necessary libraries are `#included`, and any heap accesses outside what the program allocates is a segmentation fault.

Q4) [12 Points] Which of the following are possible, if perhaps unlikely, results of attempting to compile and run this code? (select ALL that apply)

```
int main() {
    int32_t *str = (int32_t *) malloc(sizeof(int32_t) * 3);
    printf("%s", (char *) str); // A char is 8 bits.
    return 0;
}
```

- ▢ Compilation error due to invalid typecast
- ▢ Runtime typecasting error
- ● A segmentation fault
- ● The program prints the empty string
- ● The program prints **`CS61C`**
- ▢ The program prints **`CS61C rocks!`**

- Compilation or runtime typecast error: Won't happen, the cast of the bits on the right matches the value expected on the left: `(int32_t *)` matches `str`'s type, and `(char *)` matches `%s`.
- A segmentation fault: Possible; this can occur if no byte in val is a null terminator (`'\0'`), since inside `printf` it will keep reading the string `str` until it finds the null terminator. Note that because `malloc` doesn't clear out the data already in that space, it is indeterminate whether this will occur, or if any valid string is outputted.
- The program prints the empty string "": Possible; this can occur if the 0th byte in `str` is a null terminator
- The program prints the string "`CS61C`": Possible; this can occur on the off chance that the first six bytes of `str` match the sequence `"C", "S", "6", "1", "C", "\0"`
- The program prints the string "`CS61C rocks!`": Impossible; "`CS61C rocks!`" is 12 characters, and since all strings must contain a null terminator, that would require 13 bytes to store. Since `str` is only 12 bytes long, "`CS61C rocks!`" cannot be outputted. Remember that we stated that any memory accesses the program did not allocate will cause a segfault. This means that the allocation of `str` is on the heap, which means that we only have 12 bytes to work with – accessing that 13th byte to get the `"\0"` would error.

Q5) [10 Points] Each of the following evaluate to an address in memory. In other words, they "point" somewhere. Where in memory do they **point**?

	Code	*Static*	*Stack*	*Heap*
`arr`	○	●	○	○
`arr[0]`	○	●	○	○
`dest`	○	○	●	○
`dest[0]`	○	○	○	●
`&arrPtr`	○	○	●	○

- `arr` is an array of character pointers, so it itself is a pointer to the first element in the array. Since the array is declared globally, its contents are placed in the static portion of memory, so `arr` points to **static**.
- `arr[0]` is a string literal (therefore, a pointer to

Q6) [10 Points] The program below runs through the array of strings, doing something to each of the characters and putting the results in the `dest` array.

What are the first 8 characters the program prints? (Note: The program DOES compile and run without error.)

G O _ B E A R S

This function capitalizes all letters in `arr`, the letter's ASCII encoding has its 5th bit turned off by the bitwise and. Let's go through that slowly.
A's binary representation is `0b0100 0001`
a's binary representation is `0b0110 0001`
(hmm, you might say, it only differs in the 32's place, and that remains true for all the letters since there are only 26 of them and 32 possible bit patterns for the lowest 5 bits)

Room

the first character in the string) so it points to the **static** portion of memory.

- `dest` is an array declared in the `main` function, so all of the elements in `dest` are in the stack. Since `dest` is a pointer to the first element, it points to the **stack**.
- Here, we notice the line where we set `dest[i]` to be the result of a `malloc`. Since `malloc` allocates space in the heap, `dest[0]` is a pointer into the **heap**.
- `arrPtr` is declared at the top of `main`, so it is in the stack. The address of `arrPtr` is therefore a pointer into the **stack**.

(1 << 5) left shifts the 1 in the binary representation 5 spots to the left to make `0b0010 0000` (`0x20`), and when it is negated it becomes `0b1101 1111`. Whenever you bitwise and (&) a value with a mask (here `0b1101 1111` is the mask), all the values that match up with the mask's 1 stay, all others become zero. It's a way to set to 0 whatever bits you want. So we are turning OFF the 32's bit, which take any letter ([a-z] with the 32's bit on as well as [A-Z] with the 32's bit already off) and maps it to [A-Z]. So, in effect, it capitalizes all the letters of the two strings.

```c
// The ASCII values for 'A', 'B', etc. are 65, 66, ... ⇐⇐⇐⇐⇐⇐ Important
// The ASCII values for 'a', 'b', etc. are 97, 98, ... ⇐⇐⇐⇐⇐⇐ Important
char *arr[] = {"Go", "Bears"};

int main() {

   char **arrPtr = arr;
   char *dest[2];
   int j;
   for (int i = 0; i < 2; i++) {
       char *currString = *arrPtr;
       dest[i] = (char *) malloc(strlen(currString) + 1);
       for (j = 0; j < strlen(currString); j++) {
           dest[i][j] = currString[j] & ~(1 << 5); // ⇐ Hint: Focus on this line!⇐
       }
       dest[i][j] = '\0';
       arrPtr++;
   }
   printf("%s %s", dest[0], dest[1]);
}
```

Room