HW01: Number Representation and C

CSE 20221 Logic Design

Names: Aaron Wang and Ethan Little

Preamble:

1. Make sure you start from the Google Docs copy of this assignment, found in Google Drive.

2. Copy this assignment to your Google Drive folder and enter your answers in the boxes provided; **each empty box should be filled in with an answer**. You can either type your answers directly or scan handwritten work as long as it is legible. For solutions that require code, use a fixed-width font (Consolas preferred) no smaller than 10 points with proper indentation. Save your solutions as a single PDF file and upload them to Gradescope.

3. See the checklist at the end of the document for a list of what to turn in, and point totals.

4. Type your names at the top of this document - you are encouraged to work in groups of 2.

5. Always double-check your final PDF prior to uploading to Gradescope. In past semesters, students occasionally encountered problems copying output from a terminal to a Google Doc.

6. When copying text or taking a screenshot, please use dark text on a light background, or light text on a dark background, otherwise it is very difficult to read.

Introduction

The C programming language was originally developed in 1972 by Dennis Ritchie at AT&T Bell Laboratories for the purpose of writing the Unix operating system. Because of this, it has a lot of features that let you get close to the hardware and basic operations of the processor. The purpose of this exercise is to explore some of these features and relate them to concepts that you will see in Logic Design, including the representation of numbers and letters and Boolean and bit-level operations.

For this activity, we assume that you have some familiarity with a C-like language, and also know how to log into a Linux box and edit files. If you don't, please visit the instructor or a TA in office hours for a review, and partner with someone who does.

Part 1: Three Different Interpretations of the Same 32 Bits

In a programming language, a data type is a way of describing the representation and properties of a variable, such as how many bits, how to interpret them, and what operations can be performed on them. C has a variety of built-in numerical data types, we will start by focusing on one of them: `int`. A C `int` is an ordered set of 32 bits that can represent both positive and negative integers, as well as characters in various formats. For example, binary `1100001` (decimal 97) is the \`a\` character in ASCII; the remaining leftmost bits are all 0s when using a 32-bit `int`. Extended character sets, like Unicode and its variants (UTF-8, UTF-16, UTF-32, etc.) can be used to encode a wide range of characters from different languages, as well as emojis.

Key point: Deep inside the computer hardware, an `int` variable is just a set of 32 1s and 0s without any particular meaning. C (and other programming languages) lets us interpret these bits in a variety of different ways.

Given the following ordered set of 32 bits: `00000000000000011111011000000000`
answer the questions below.

What number do these bits represent as a decimal integer?	**`128,512`**
What number do these bits represent as a hexadecimal integer?	**`0x1F600`**
What letter do these bits represent as a UTF-32 character? (You should look it up, or write a C program to print it. Hint: it's an emoji)	😀

Part 2: Displaying an `int` in Decimal, Hex, and Character Forms in C

C uses the `printf` statement to display messages to the terminal. Here's "Hello, world!" in C:

```
#include <stdio.h>
int main()
{
    printf(“Hello, world!\n”);
}
```

Program notes:

- The `printf` function is declared in `stdio.h` so you need to include it.
- This version of `printf` takes a single argument called the *format string*, which specifies the message to be displayed.
- `‘\n’` is the new line character

Using your favorite text editor (e.g., `vim`), write and save the program as `hello.c`
It is best **not** to copy/paste any of this code, things like "smart" quotation marks will cause syntax errors.
Compile the program with the following command: `gcc hello.c`
Run the program by typing: `a.out` (or `./a.out` depending on how you've set your path)

To display the results of expressions as part of a message, `printf` uses placeholders that specify the format of the value. In the following program fragment, `%d` is the placeholder for displaying the result of an expression in decimal format. The format string has 3 placeholders corresponding to the 3 additional arguments `x, y, x + y`. More formally, these placeholders are called *conversion specifiers*.

```
    int x = 2;
    int y = 3;

    printf("x = %d  y = %d  sum = %d\n", x, y, x + y);
```

Add the additional lines to your "Hello, world!" program, run it, and observe the result

`printf` uses different conversion specifiers to display the values in different formats. Initially, we'll consider three of these:

- `%d`: display as signed decimal
- `%x`: display as hex
- `%c`: display as ASCII character

Add new lines to your program that assign the decimal value `42` to an `int` variable (you can reuse `x`) and then use a single `printf` statement to display that variable in signed decimal, hex, and ASCII character formats. Compile and run it. Enter the representation of decimal `42` in hex and ASCII character formats in the boxes below.

hex	**2a**	ASCII	*****

Add new lines to your program to assign the hex value `0x3E` to the variable and recompile and rerun it. Enter your results in the other two formats below.

decimal	**62**	ASCII	**>**

Add new lines to your program to assign the ASCII character `'q'` (in single quotes) to the variable and recompile and rerun it. Enter your results in the other two formats below.

decimal	**113**	hex	**71**

Part 3: Negative Integers

Add new lines to your program to display the decimal numbers -1, -2, and -3 in hex format; do not use any variables for this part. Compile and run it and answer the questions below.

What were your results for -1, -2, -3?	`-1: 0xffffffff`
	`-2: 0xfffffffe`
	`-3: 0xfffffffd`

Do you see a pattern? How does C represent negative integers in 32 bits?	Twos complement notation - it flips every bit to its complement and then adds 1 to the negative magnitude.
Add new lines to your program to take the hex values `0x7ffffffe, 0x7fffffff`, `0x80000000, 0x80000001` and print them in decimal format. What output do you get? Use your intuition to explain the result.	`0x7ffffffe:` 2147483646 `0x7fffffff:` 2147483647 `0x80000000:` -2147483648 `0x80000001:` -2147483647 If the most significant bit is 1 the number is negative
What does the most significant bit (furthest	If the bit is 1 the `int` is negative.

to the left) tell us about the sign of an `int`?	If the bit is 0 the `int` is positive.

Part 4: Unsigned Integers

A C `int` can represent 2^{32} different values. If we interpret these as signed integers, half of the values are negative and half are positive (including zero). For some applications we don't need negative numbers and would prefer to consider all 2^{32} combinations as positive integers. C provides the `unsigned int` data type for this purpose. To display an `int` as an unsigned value, use the `%u` conversion specifier with `printf`. Answer the following:

What should be the hex representation for the largest unsigned integer in C?	**`0xffffffff`**
Add lines to your program to assign this value to a new **`int`** variable, and then display this hex value as an *unsigned* integer using `%u`. What output do you get? **Note:** if you weren't quite right, revise your guess based on the results and try again. You can just report your final answers in these boxes.	**`4294967295`**

Part 5: Floating Point Numbers

The C data types `float` and `double` allow for storage and manipulation of single- and double-precision floating-point data, respectively. These types follow the IEEE-754 floating-point standard. With floating-point numbers, we may want to specify the number of digits after the decimal point. We can use the %f conversion specifier along with the precision in the form "`%.Nf`" where N is the number of digits after the decimal point. Example: `printf("%.5f", z);` will print z with N = 5 digits after the decimal point. (Sidenotes: technically, %f converts to *double-precision* for printing; the behavior of "`%.N`" differs depending on the conversion modifier used.)

Add lines to your program to create a **single**-precision floating point variable (i.e., use **`float`**) and assign it the value 48.15162342. Then, print this value using the %f conversion specifier with 20 digits after the decimal point. What output do you get?	**48.15162277221679687500**
Add lines to your program to create a double-precision floating point variable (i.e., use **`double`**) and assign it the value 48.15162342. Then, print this value using the %f conversion specifier with 20 digits after the decimal point. What output do you get?	**48.15162341999999995323**
What is one example of a number with at least one significant digit after the decimal point that can be represented exactly? (i.e., 1.0, 2.0, etc. are not good answers) Tip: you can use this tool to visualize and see what is requested vs. what is actually stored.	**1.0625**

Part 6: Bitwise Boolean Operations

The three Boolean operations, AND, OR, and NOT are the mathematical foundation upon which all of digital logic is built. While you may not frequently use them in your normal day-to-day program, they are behind the scenes in every computation you perform and we will use them extensively in designing hardware. They also appear frequently in system software when it is necessary to manipulate data at the bit level. In Boolean logic, we usually think of a 1 as representing "true" and 0 as representing "false". Given two Boolean (true/false) variables `a` and `b`, the three fundamental Boolean operations on them are defined as follows

- `NOT i`: true `(1)` if `i` is false `(0)`; false `(0)` if i is true `(1)`
- `i AND j`: true `(1)` if both `i` and `j` are true `(1)`; false `(0)` otherwise
- `i OR j`: true `(1)` if `i` or `j` or both are true `(1)`; false `(0)` otherwise

The table below lists all 4 possible combinations of values for i and j. Complete the table by first filling in the results for the "combinations" columns for the AND and OR operations, following the example for the NOT operation. Then, the 4 bits across each row of the combinations can be represented by a single hex digit. Fill in the hex column with the digits for each row, following the example that `i:1010 = 0xA` and `j:1101 = 0xD`.

	combinations				hex
i	1	0	1	0	0xA
j	1	1	0	1	0xD
~i	0	1	0	1	0x5
i & j	1	0	0	0	0x8
i | j	1	1	1	0	0xE

The bitwise Boolean operators in C for AND `(&)`, OR `(|)`, and NOT `(~)` work just like this table, performing operations between column-aligned bits of an `int`. For example, the following statements evaluate the AND operation and print the result in hex.

```
int i = 0xA;
int j = 0xD;
printf("i & j = %x\n", i & j);
```

Add lines to your program that evaluate and print the results for each of the 3 fundamental Boolean operations and answer the question below:

What results did you get for NOT i? If it is different from the table above, explain why.	`0xfffffff5`
	`This is because it converts all of the bits to their complements, not just the ones that we see because i is an integer.`

A common bit-level operation is "zeroing out" certain bits in a variable, leaving others at their original values. We do this by AND-ing the variable with another integer that serves as a *mask*. For example, when executing "`i & j`" we can think of `j` as a mask - any bit set to 0 in `j` will cause the result to be 0 in that bit position, regardless of what `i` stores in that position; any 1 in `j` will result in the original value in that bit position in `i` to be retained.

Similarly, we can set bits in a variable to `1` by OR-ing that variable with a mask. For example, when executing "`i | j`" we can again think of `j` as a mask - any bit set to 0 in `j` will result in the original value in that bit position in `i` to be retained; any bit set to 1 in `j` will cause the result to be 1 in that bit position, regardless of what `i` stores in that position.

What 32-bit value in hex could you use as a mask to zero out all but the lowest 8 bits of an `int`? Write the value in the box and <u>add lines to your program</u> to test this and display the result. Pick a *meaningful* test case to demonstrate the mask is working.	**`0xff`**
	`0xaaaaaaaa & 0xff = 0xaa`

What 32-bit value in hex could you use as a mask to set to `1` all but the lowest 8 bits of an `int`? Write the value in the box and <u>add lines to your program</u> to test this and display the result. Pick a *meaningful* test case to demonstrate the mask is working.	**`0xffffff00`**
	`0xaaaaaaaa & 0xffffff00 = 0xffffffaa`

Part 7: Shifting

Another bit-level operation is shifting the bits in a variable to the left or right. The C shift operators are as follows:

- `a << b` shifts the bits of `a` to the left by `b` bits, shifting in zeros from the right.
- `a >> b` shifts the bits of `a` to the right by `b` bits, shifting in copies of the most significant[1] bit from the left (which preserves the sign of an `int`).

Answer the following:

Write the decimal number 64 in binary. Shift the bits by 1 bit to the left, shifting in a 0 on the right side. What is the decimal value of the result? Shift the bits to the left by one more bit. Now what do you get? In general, what does shifting a number to the left by one bit do to its value? <u>Add lines to your program</u> that demonstrate a shift to the left by 1 bit on an `int` and display the result in decimal and hex. The `int` should start at 64.	**64_{10}=1000000_2** **128; 256** In general, shifting the number to the left by one bit doubles the value (multiplies by the base). **`128, 0x80`** **`256, 0x100`**

What would shifting to the right do? Explain, and then <u>add lines to your program</u> to demonstrate a shift to the right by 1 on an `int` and display the result in decimal and hex. Try this with two ints, one with a starting value of 64 and one with a starting value of -64.	In general, shifting the number to the right by one should do the opposite so it should half the value (divide by the base).
	`32;  0x20`
	`-32; 0xffffffe0`

[1] The "most significant" bit is the leftmost bit and the "least significant" bit is the rightmost bit.

Repeat the last two questions but replace both `int` variables with `unsigned int` variables; keep the initial values the same (i.e., 64 and -64). Add a line to print the value of the `unsigned int` assigned the value -64 (spoiler: it will become a large positive number, check your work if it does not!). Do the behaviors of the left and right shift operators change, and if so, how?	**`128, 0x80`**
	`256, 0x100`
	
	`32, 0x20`
	`2147483616, 0x7fffffe0`
	
	The left shift works the same.
	The right shift is the same for a positive number but for a negative number it displays the complement of what would be half of the negative number because it can not display negatives.

What to Turn In

Fill in the boxes on the previous pages with answers to all the questions and then copy your completed C program and output in the boxes below. Save your Google Doc as a PDF and upload it to **Gradescope** for grading. Note Gradescope can be accessed via our Canvas site, or by visiting gradescope.com. Make sure your program has sufficient comments so that the graders can clearly identify the parts of the program. Use a fixed-width font (Consolas is preferred) for your code with proper indentation. Below is a checklist of everything your program should do:

Checklist for Question Boxes (25 points)

This is a summary of the question boxes above.

	Deliverable	**Points**
1.	Part 1: binary number in 3 formats (3 boxes to fill)	3
2.	Part 2: tables listing conversion of 42, 3E, ‘q’ (6 boxes to fill)	3
3.	Part 3: negative numbers, patterns, most significant bit (4 boxes to fill)	4
4.	Part 4: unsigned int hypothesis and result (2 boxes to fill)	2
5.	Part 5: floating-point single- , double-precision, and 1 exact number (3 boxes to fill)	3
6.	Part 6: table for i & j, i | j; NOT i result; two masks (1 table and 3 boxes to fill)	6
7.	Part 7: shifting results and explanation (3 boxes to fill)	4

Checklist for Program (60 points)

This is a summary of everything your program should do. Make sure to include your program and output in the boxes below.

	Deliverable	**Points**
1.	display “Hello, world!”	1
2.	printf displaying x, y and x+y	2
3.	assign 42 to variable; display in hex, decimal, and as a character	2
4.	assign 0x3E to variable; display in hex, decimal, and as a character	2

5.	assign ‘q’ to variable; display in hex, decimal, and as a character	2
6.	display signed decimals -1, -2, and -3 as hex values (no variables should be used)	6
7.	display the hex values 0x7ffffffe, 0x7fffffff, 0x80000000, 0x80000001 as signed decimals (no variables should be used)	4
8.	assign the largest unsigned integer to an int in hex, and display it as an unsigned decimal using %u	5
9.	48.15162342 as a single-precision and a double-precision floating-point number with 20 digits after the decimal point	5
10.	display results from using the Boolean AND, OR, and NOT operators (&, |, ~)	5
11.	use a mask and the AND operation to zero out all but the 8 least-significant bits of an int, and print the result	5
12.	use a mask and the OR operation to set to ones all but the 8 least-significant bits of an int, and print the result	6
13.	demonstrate shift left on 64	4
14.	demonstrate shift right on positive and negative integers (64, -64)	6
15.	demonstrate shift left and right on unsigned integers (repeat 13-14 with unsigned int)	5

Program

```c
#include <stdio.h>
int main()
{
    // part 2
    printf("Hello World!\n");
    int x = 2;
    int y = 3;
    printf("x = %d  y = %d  sum = %d\n", x, y, x + y);

    x = 42;
    printf("signed decimal: %d  hexadecimal: 0x%x  ASCII: %c\n", x, x, x);
    x = 0x3E;
    printf("signed decimal: %d  hexadecimal: 0x%x  ASCII: %c\n", x, x, x);
    x = 'q';
    printf("signed decimal: %d  hexadecimal: 0x%x  ASCII: %c\n", x, x, x);

    // part 3
    printf("hexadecimal: 0x%x\n", -1);
    printf("hexadecimal: 0x%x\n", -2);
    printf("hexadecimal: 0x%x\n", -3);

    printf("decimal: %d\n", 0x7ffffffe);
    printf("decimal: %d\n", 0x7fffffff);
    printf("decimal: %d\n", 0x80000000);
    printf("decimal: %d\n", 0x80000001);

    // part 4
    printf("decimal: %u\n", 0xffffffff);

    // part 5
    float z = 48.15162342;
    printf("float: %.20f\n", z);
    double w = 48.15162342;
    printf("float: %.20f\n", w);

    // part 6
    int i = 0xA;
    int j = 0xD;
    printf("i & j = 0x%x\n", i & j);
    printf("i | j = 0x%x\n", i | j);
    printf("~i = 0x%x\n", ~i);

    int k = 0xaaaaaaaa;
    printf("k & 0x000000ff = 0x%x\n", k & 0x000000ff);
    printf("k | 0xffffff00 = 0x%x\n", k | 0xffffff00);

    // part 7
    int a = 64;
    printf("64 << 1 = %d, 0x%x\n", a << 1, a << 1);
    printf("64 << 1 = %d, 0x%x\n", a << 2, a << 2);

    printf("64 >> 1 = %d, 0x%x\n", a >> 1, a >> 1);
    a = -64;
    printf("-64 >> 1 = %d, 0x%x\n", a >> 1, a >> 1);

    unsigned int b = 64;
    printf("64 << 1 = %d, 0x%x\n", b << 1, b << 1);
    printf("64 << 1 = %d, 0x%x\n", b << 2, b << 2);

    printf("64 >> 1 = %d, 0x%x\n", b >> 1, b >> 1);
    b = -64;
    printf("-64 >> 1 = %d, 0x%x\n", b >> 1, b >> 1);
}
```

Output

```
Hello World!
x = 2  y = 3  sum = 5
signed decimal: 42  hexadecimal: 0x2a  ASCII: *
signed decimal: 62  hexadecimal: 0x3e  ASCII: >
signed decimal: 113  hexadecimal: 0x71  ASCII: q
hexadecimal: 0xffffffff
hexadecimal: 0xfffffffe
hexadecimal: 0xfffffffd
decimal: 2147483646
decimal: 2147483647
decimal: -2147483648
decimal: -2147483647
decimal: 4294967295
float: 48.15162277221679687500
float: 48.15162341999999995323
i & j = 0x8
i | j = 0xf
~i = 0xfffffff5
k & 0x000000ff = 0xaa
k | 0xffffff00 = 0xffffffaa
64 << 1 = 128, 0x80
64 << 1 = 256, 0x100
64 >> 1 = 32, 0x20
-64 >> 1 = -32, 0xffffffe0
64 << 1 = 128, 0x80
64 << 1 = 256, 0x100
64 >> 1 = 32, 0x20
-64 >> 1 = 2147483616, 0x7fffffe0
```

Extra Practice Problems (Ungraded)

These will not be graded, so no need to turn them in, but they make for good practice problems for future assignments and exams. Solutions will be provided.

Enter your solutions in the boxes provided. You can either type them directly or insert images of neatly written results. Show all work.

Problem 2: Convert the following 8-bit unsigned binary numbers to hexadecimal and decimal.

A. 00101001
B. 1001110
C. 11010111
D. 10011010

Problem 3: Convert the following unsigned decimal integers to 8-bit unsigned binary numbers and hexadecimal numbers. If it is not possible to do so, state why not.

A. 17
B. 163
C. 271
D. 93

Problem 4: Evaluate each of the following arithmetic expressions. Assume that all given values are 8-bit unsigned integers, represented in hex. Show your solutions as 8-bit unsigned integers in both binary and hex, and state whether or not the operation causes an overflow.

A. AC + 46
B. DA + 61

Problem 5: Convert the following 8-bit signed 2's complement binary numbers to hexadecimal and decimal.

A. 00101001
B. 01001110
C. 11010111
D. 10011010

Problem 6: Convert the following signed decimal integers to 8-bit 2's complement binary numbers and hexadecimal numbers. If it is not possible to do so, state why not.

A. 171
B. -142
C. 65

D. -65

Problem 7: Evaluate each of the following arithmetic expressions. Assume that all given values are 8-bit signed 2's complement integers, represented in hex. Show your solutions as 8-bit 2's complement integers in both binary and hex, and state whether or not the operation causes an overflow.

A. 0C + 3B
B. 9B + 5C
C. 57 + 32
D. 6E – 23
E. 9B – 23