CS220 Spring 22 Exam 3 Study Questions

1. Convert -6.625 to 32-bit IEEE single precision format

   0xC0D40000   You will need to show work for partial credit.

2. Convert 123.457 to 32 32-bit IEEE single precision format.

   This one is trickier because the binary expansion goes on for a full 23 bit length mantissa and also rounding bits. You would have to use the program we wrote to generate the bits (otherwise you are doing a ton of calculations by hand).

   123 is **01111011**

   The binary expansion of .457 is …

```
ehar@raspberrypi:~/CS220Spring22/float $ python3 float2bin.py 0.457
011101001111110111110011101101100100010110100001110011
```

   This generates way more bits than we need. But that's OK. Combine the 123 and the .457 we get …

   1111011.011101001111110111110011101101100100010110100001110011

   Writing in scientific notation we get

   1.111011011101001111110111110011101101100100010110100001110011 x 2⁶

   So the exponent is 6 + 127 = 133 in binary is **10000101**.

   Laying it out we get

```
0     10000101     11101101110100111111011     11
sign  exponent     mantissa                    rounding bits
```

   Writing in groups of four

```
0100 0010 1111 0110 1110 1001 1111 1011     11
4    2    F    6    E    9    F    B
```

   We need to round the last hex digit to a C because of the rounding bits. So the final answer is 0x42F6E9FC.

CS220 Spring 22 Exam 3 Study Questions

3. What floating-point number is represented by **0x41BA0000**.

23.25

a. **Write out the bits 01000001101110100000000000000000**
b. **The exponent is 10000011 = 131. Subtract 127, so exponent is 4.**
c. **Binary scientific notation is 1.0111010 X 2^4 (don't forget implied leading one bit)**
d. **Rewriting 10111.01 which is 23.25.**

4. Assume we are multiplying the unsigned integers **1011 X 1011**. Trace the values of the multiplicand, multiplier, and result at every step. (We are not covering this algorithm until Monday April 11).

multiplicand	**multiplier**	**result**
1011	1011	0 + 1011
10110	101	1011 + 10110 = 100001
101100	10	100001 + 1011000 = 1111001
1011000	1	final answer is 1111001 = 121_{ten}

5. The swap function below exchanges the two double values pointed to by **x** and **y**. Write **swap** as an ARM assembly language function. Full credit for the most concise version.

```
void swap(double *x, double *y) {
     double tmp = *x;
     *x = *y;
     *y = tmp;
}
```

```
swap:
    vldr.f64 d0, [r0]  // d0 = *x
    vldr.f64 d1, [r1]  // d1 = *y
    vstr.f64 d1, [r0]  // *x = d1
    vstr.f64 d0, [r1]  // *y = d0
    bx lr
```

6. Write a recursive C function that implements the declaration below. **popcount** counts the number one bits in the binary representation of its argument. For example, **popcount(30)** is 4 because 30 in binary is 11110, which has four one bits.

```
extern int popcount(unsigned int n);
```

```
int popcount(unsigned int n) {
    if (n == 0)
        return 0;
    else
        return popcount(n >> 1) + (n & 1);
}
```

7. Write **popcount** as an ARM assembly language function.

```
popcount:
    push { r4, lr }
    mov r4, r0          // save n in r4

    cmp r4, #0          // base case
    bne else
    mov r0, #0
    pop { r4, pc }
else:                   // recursive case
    lsr r0, r4, #1
    bl popcount
    and r1, r4, #1
    add r0, r0, r1
    pop { r4, pc }
```

8. Consider the logic function with three inputs **A**, **B**, **C** and one output **Out**. **Out** should be 1 when exactly two inputs are 1.

   a. Draw the truth table for this function.

A	B	C	Out
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

b. Write the sum-of-products logic equation for this function.

$$Out = \overline{A}BC + A\overline{B}C + AB\overline{C}$$

c. Minimize the logic equations

There are several solutions, but it isn't clear if they are really simpler (or smaller). You could either group the first two terms and factor out a C or the last two terms and factor out an A. Or you could do both by temporarily replicating the middle term (because $X = X + X$)

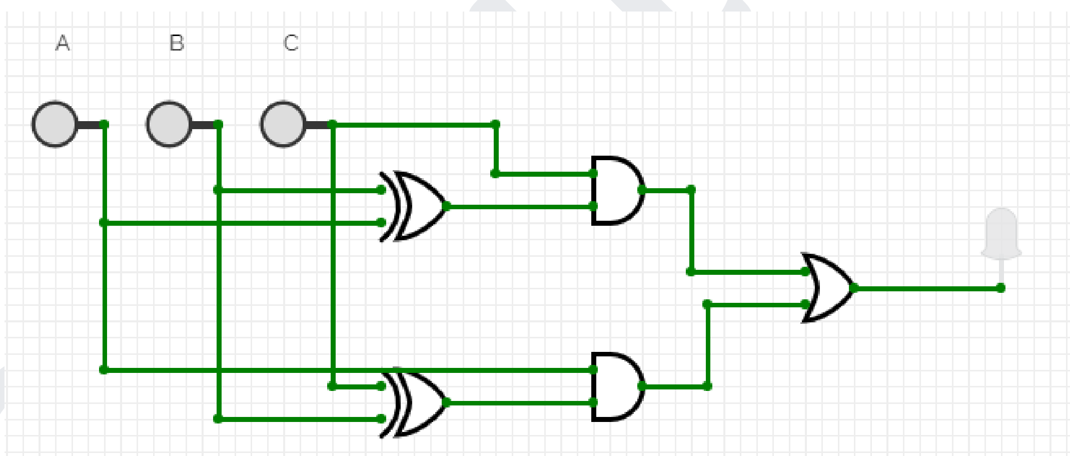
$$Out = \overline{A}BC + A\overline{B}C + A\overline{B}C + AB\overline{C}$$

$$Out = C(A \oplus B) + A(B \oplus C)$$

I guess this final equation uses five gates where the SOP form used eleven.

d. Draw the circuit diagram for the logic equation.

I was messing around with this circuit editor at https://circuitverse.org/simulator



9. Write a C function **`scale`** that takes a factor and multiplies each item in the array by the factor.

```
extern void scale(double factor, double [] vec, int n);

void scale(double factor, double *vec, int n) {
    int i = 0;
    while (i < n) {
        vec[i] = vec[i] * factor;
        i++;
    }
}
```

So this C function is boring and easy. What I meant to ask was to write the assembly language version.

```
scale:
    mov r2, #0           // i - loop counter

while:
    cmp r2, r1           // loop condition, r1 has size of array
    bge endwhile
    lsl r3, r2, #3       // r3 is the offset into the array, why #3?
    add r3, r0, r3       // address of element i
    vldr.f64 d1, [r3]    // vec[i]
    vmul.f64 d1, d1, d0
    vstr.f64 d1, [r3]
    add r2, r2, #1
    b while

endwhile:
    bx lr
```

10. Make sure you understand the four areas of program memory; code, global data, stack, and heap and how memory is allocated for each.

11. Static function local variables in C are allocated in/on **global** memory.

12. Local variables in C are allocated in/on **stack** memory.

13. Memory allocated using `malloc` is **heap** memory.

14. What does the `-g` flag on the gcc compiler do?

    Includes debugging information in the object file so that it can be used by GDB.

15. What does the `-S` flag on the gcc compiler do?

    Generate a .s assembly file.

16. What does the `-o` flag on the gcc compiler do?

    Specify an output file name.

17. What does the `-O3` flag on the gcc compiler do?

    Turns on the highest level of code optimization.

18. What does the `-c` flag on the gcc compiler do?

    Don't link into an executable, only produce .o object files.

19. What program do we use to reverse engineer machine code files?

    objdump with the -d flag.

20. How many bytes is a C **double**? **8**

21. Briefly describe what a *memory leak* is?

    Memory that has been allocated (usually by malloc) and was never deallocated, or freed (usually by free).

22. Consider the following C program. Why might it have a segmentation fault?

```
#include <stdio.h>
int *seven() {
    int x = 7;
    return &x;
}

int main() {
    int *y = seven();
    printf("%d\n", *y);
}
```

Because function `seven` returns an address in stack memory that no longer exists when the function returns.

23. The following variation of the program seems to work OK. Why?

```
#include <stdio.h>

int *seven() {
    static int x = 7;
    return &x;
}

int main() {
    int *y = seven();
    printf("%d\n", *y);
}
```

Because static variables are not allocated on the stack like local variables are, the address of `x` in function `seven` persists beyond function invocation.

24. Write a function **`rev`** that takes an unsigned integer **x** and reverses the bits in **x**. Use bit operations only, don't use strings or arrays.

    Keep extracting the least significant bit and and push int on to a new result integer.

    ```
    u_int32_t rev(u_int32_t n) {
        u_int32_t m = 0;  // result

        // build up the int in reverse
        while (n > 0) {
            m = (m << 1) | (n & 1);
            n = n >> 1;
        }
        return m;
    }
    ```

    Here's a one-liner Python version. But it uses strings. Can you explain how it works?

    ```
    def rev(n):
        return int((bin(n)[2:])[::-1],2)
    ```

    a. Modify the **add** function in **adder.c** we wrote to call **rev**.

    ```
        …
        return rev(s);
    }
    ```

25. There is a simple fix to the **add** function in **adder.c** file that does not need to reverse the bits. What is it?

    The function has the line `s = (s << 1) | S;` which shifts the result left and then bitwise-or in the sum bit as the least significant bit.

    Instead, the sum bit S all the way to the left i positions and then or it in to the result like so …

    ```
    s = (S << i)  | s;
    ```