**CS220 Spring 22 Exam 3 Study Questions**

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

   **C0D40000**

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

   **42F6E9FC**

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

   **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 | |
| 10110 | 101 | 1011 |
| 101100 | 10 | 100001 |
| 1011000 | 1 | 100001 |
| 10110000 | | 11111001 |

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;}
```

.global swap

.fpu vfp //tell processor what floating point units are used (vector floating point unit)

```
.cpu cortex-a53

.text

        vldr.f64 d0, [r0] //put data at location r0 into d0

        vldr.f64 d1, [r1] //put data at location r1 into d1

        vstor.f64 d1, [r0] //put data d1 into location at r0

        vstor.f64 d0, [r1] //put data d0 into location at r1

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

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

```
.cpu cortex-a53

.global popcount

.text

popcount:
```

```
        push { r4, r5, lr }
        mov r4, r0 // n is in r0
        cmp r4, #0
        beq end
        bl else

end:
        mov r0, #0
        pop { r4, r5, lr}
        bx lr

else:
        and r5, r4, #1
        lsr r4, r4, #1
        mov r0, r4
        bl popcount
        add r5, r5, r0
        mov r0, r5
        pop {r4, r5, lr }
        bx lr
```

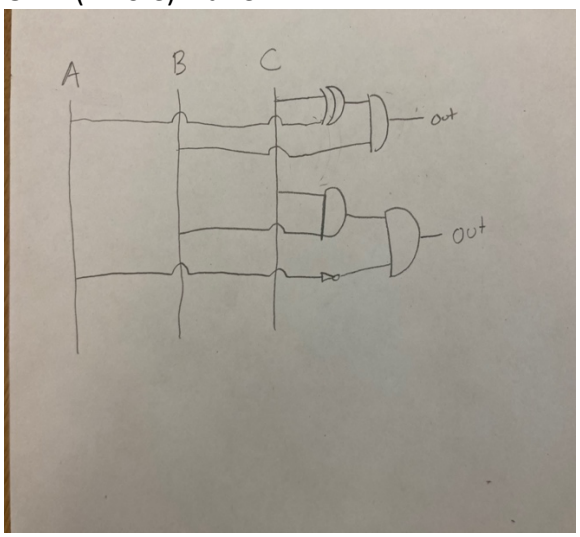
8. Consider the logic function with three inputs **A**, **B**, **C** and one output **Out**. **Out** should be 1 when exactly two inputs are 1.
    0. Draw the truth table for this function.
    1. Write the sum-of-products logic equation for this function.
    2. Minimize the logic equations

3. Draw the circuit diagram for the logic equation.

| A | B | C | Out |
|---|---|---|---|
| 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 |

Lowercase letter = knotted

1.) O = ABc + AbC + aBC
2.) O = A(B +o C) + aBC
3.)




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){
	for(int x = 0; x < n; x++){
		double *address = &vec[x];
		*address = factor * vec[x];
	}
}

// could also be done like this I believe
	double [] scaled;
	for(int x = 0; x < n; x++){
```

```
        double temp = vec[x];
        temp = temp * factor;
        scaled[x] = temp;
    }
    vec = scaled;
```

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 Data (data segment?)** 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?

Compiles so that it can be run in the debugger.

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

Compiler produces assembly code
16. What does the **`-o`** flag on the gcc compiler do?

Creates an executable with desired compiled files into named executable
17. What does the **`-O3`** flag on the gcc compiler do?

Optimizes code to the third level
18. What does the **`-c`** flag on the gcc compiler do?

Compiles code individually without need of a main file or corresponding files making an object file.
19. What program do we use to reverse engineer machine code files?

Object dump

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

8 bytes (32 bits)

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

Program uses more RAM memory over time until memory is full and “spills over” with no available space left. When memory is allocated but not deallocated.

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

X is not a global variable so it cannot be accessed and thus not able to be pointed to within the main as it is only accessible within seven.

```
#include <stdio.h>
int *seven() {

     int x = 7;

     return &x;

}

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

}
```

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

The variable x is now accessible globally which allows it to be pointed to  within the main.

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

}

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

}
```

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.

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

```
u_int32_t rev(u_int32_t x){

        u_int32_t rev = 0;
```

```
        while(x>0){
                rev <<= 1;
                rev = rev|(x&1);
                x >>=1;
        }
        return rev;
}
```

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?

```
u_int32_t add(u_int32_t x, u_int32_t y){
// ^ bitwise exclusive or
        u_int32_t s = 0; // total sum
        bit_t C = 0; // carry: initial carry is always 0
        for (int i = 0; i < 32; i++){
                bit_t X = x & 1; //extract last x bit
                bit_t Y = y & 1;
                bit_t S = X ^ Y ^ C;
                C = (X & Y) | (X & C) | (Y & C);

                x = x >> 1;
                y = y >> 1;

                // get S into s
                //s = (s << 1) | S;
```

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