CS 220 Spring 22 Exam 3 **Name:**____

For partial credit make sure to **show all work** where appropriate. Some answers are better than other answers. Full credit for the best answer.

1. [5] What floating-point number is represented by `0xC2FEA000`. You must show work to receive credit.

   Similar to practice problem 3.

   `1100 0010 1111 1110 1010 0000 0000 0000 0000`

   `exponent = 10000101 = 133 - 127 = 6`

   Including the implied leading 1 this is `-1.1111110101 X` 2^6

   Moving the decimal point to the right six places we have

   `-1111111.0101` which is `-127.3125` in decimal

2. [5] Assume we are multiplying the unsigned integers **1100 X 1011**. Trace the values of the multiplicand, multiplier, and result at every step.

   Similar to practice problem 4.

   This is 12 X 11 = 132 (so our final answer should be 132).

Multiplicand	Multiplier	Result
1100	1011	0 + 1100 = 1100
11000	101	1100 + 11000 = 100100
110000	10	100100 (because lsb of mplier is 0, no add)
1100000	1	100100 + 1100000 = 10000100 = 132

3. [2] When multiplying two **n**-bit numbers the result can have as many as **2n** bits.

4. [2] What is the purpose of the `-g` flag on gcc. **(practice problem 14)**

   Include debugging information in the object and/or machine code for use by gdb.

Solution

5. [2] What is the purpose of the **-c** flag on gcc. **(practice problem 18).**

   Used for separate compilation, only generate object code and not linked executable.

6. [2] In gdb, a location where code execution will be temporarily halted is called a …

   breakpoint

7. Consider a logic function with three inputs **A**, **B**, **C** and one output **Out**. The output should be a one when exactly one (and only one) of the inputs is a one.

   Similar practice problem #8.

   a. [5] Draw the truth table for the function.

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

   b. [5] Write out the sum-of-products logic equation for the function.

   See the highlighted rows above. This leads us to

$$Out = \overline{A}\,\overline{B}\,C + \overline{A}\,B\,\overline{C} + A\,\overline{B}\,\overline{C}$$

   c. [5] Minimize the equation as much as possible

   Lots of things you can do. I'll use the same trick I did on the study guide and duplicate the middle term and group them. Then factor and use an exclusive-or.

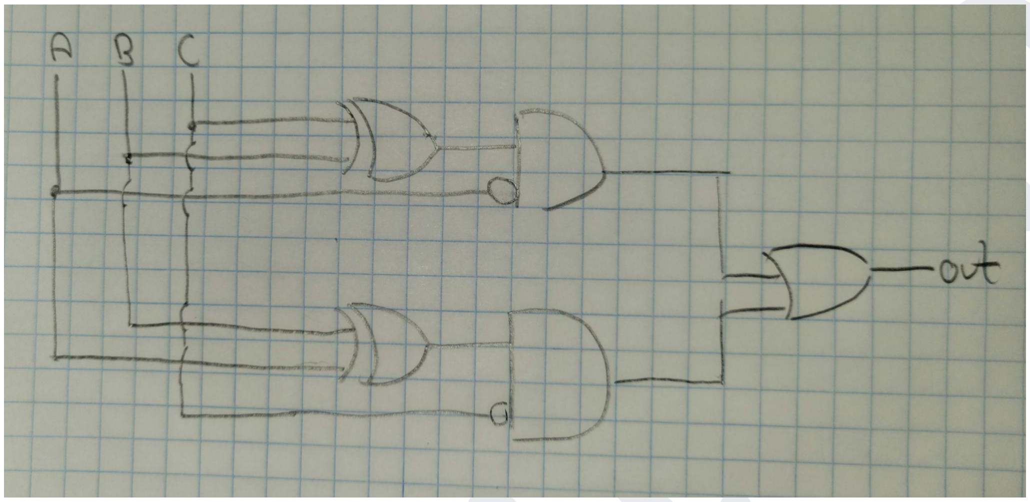
Solution

$$Out = (\overline{A}\,\overline{B}\,C + \overline{A}\,B\,\overline{C}) + (\overline{A}\,B\,\overline{C} + A\,\overline{B}\,\overline{C})$$

$$Out = \overline{A}(B \oplus C) + \overline{C}(A \oplus B)$$

d. [5] Draw the circuit diagram for the logic equation.

Your circuit drawing should at least be consistent with your equation above. The little circle in front of an input is shorthand notation for a not-gate.



8. Consider the following (rather ridiculous) C function.

```
void f(int vec[], int n) {
    int x = 99;
    int *y = malloc(sizeof(int));
    static double pi = 3.14159;
    printf("%d %X %d %f\n", x, y, vec[3], pi + n);
}
```

Practice problems 11, 12, 13, 22, 23.

a. [2] Memory for **x** is allocated on/in the **stack.**

b. [2] Memory for **y** is allocated on/in the **stack.**

c. [2] Memory for what **y** points to is allocated on/in the **heap.**

d. [2] How many bytes did the call to **malloc** allocate? **4**

e. [2] Memory for **pi** is allocated on/in **global memory.**

Solution

f. [5] To avoid creating a memory leak, can the function that called **f** free the memory that was allocated by **malloc**? Briefly explain why or why not.

No. This function will cause a memory leak and it cannot be fixed without modifying it. A calling function would need access to the pointer that was returned by malloc. But since the function does not return anything and the calling function does not have access to y, then that pointer cannot be freed.

9. [5] Assume **x** is an integer variable, write a C statement that will set the 8th bit in **x** to a 1 (where the 0th bit is the least significant bit).

This is related to practice problem 25.

```
x = (1 << 8) | x;
```

10. Assume a direct mapped cache with four rows and two instructions per row. The function below computes x^y by multiplying **x** by itself **y** times.

```
104ac:      e3a02001      mov     r2, #1
104b0:      e3510000      cmp     r1, #0
104b4:      da000002      ble     104c4
104b8:      e0020092      mul     r2, r2, r0
104bc:      e2411001      sub     r1, r1, #1
104c0:      eaffffa       b       104b0
104c4:      e1a00002      mov     r0, r2
104c8:      e12fff1e      bx      lr
```

a. [5] What address ranges constitute the loop body?

```
104b0 to 104c0
```

b. [5] Which row and column in the cache does the **bx** instruction map to? Express answer in decimal.

bx is at address 104c8. Writing out the bits we have …

```
0001 0000 0100 1100 1000

000100000100110     01     0       00
tag                 row    column  always 0
```

So 104c8 mapps to row 1 and column 0.

Solution

c. [5] When computing 2^{10} what would the cache hit ratio be for the code?

Since we fetch two instructions at a time we have the address sequence

```
104ac - miss
104b0 - hit   first loop iteration starts here
104b4 - miss
104b8 - hit
104bc - miss
104c0 - hit   first loop iteration ends here
```

9 more loop iterations are all hits, so 45 hits

The last two instructions are miss and hit. So we have

```
mhmhmh + (hhhhh)9 + mh = 49hits/(4 misses + 49 hits)

49/53 = 92.4% hit ratio.
```

d. [2] Does the code contain any ***compulsory*** misses? Briefly explain why or why not.

Yes, all of the misses are compulsory because it is the first time they were accessed.

e. [2] Does the code contain any ***conflict*** misses? Briefly Explain why or why not.

No, there are 8 sequential addresses and 8 locations in the cache. Each address maps to a different cache location.

f. [2] Does the code contain any ***capacity*** misses? Explain why or why not.

No, a direct mapped cache cannot have a capacity miss.

Solution

11. Answer questions about the recursive function below. `u_int32_t` is just an `unsigned int`.

```
u_int32_t what(u_int32_t n, u_int32_t r) {
    if (n == 0)
        return r;
    else
        return what(n >> 1, (r << 1) | (n & 1));
}
```

This was practice problem 24, but written recursively. The bit operations are similar to popcount, practice problem 6.

a. [10] Neatly draw the runtime stack of a call to `what(13, 0)` up to the point where we hit the base case. Show the values of **n** and **r** on each call.

Since these are bit operations, easier to write values in binary.

```
what(1101, 0)
      ↓
what(110, 1)
      ↓
what(11, 10)
      ↓
what(1, 101)
      ↓
what(0, 1011)
      ↓
    1011
```

b. [2] Briefly describe what the function computes. That is, what is the net effect of the function.

Reverses the bits in n.

Solution