Q1) Float, float on... (6 points)
Consider an 8-bit “minifloat” SEEEMEMM (1 sign bit, 4 exponent bits, 3 mantissa bits). All other properties of IEEE754 apply (bias, denormalized numbers, ∞, NaNs, etc). The bias is -7.

a) How many NaNs do we have? ____ 6 ____

b) What is the bit representation (in hex) of the next minifloat bigger than the minifloat represented by the hexadecimal value is 0x3F? ____ 0x40 ____

c) What is the bit representation (in hex) of the encoding of -2.5? ____ C1 ____

Standard float calculation: -2.5 = -10.1 = -1.01 x 10^1 ⇒ 1 EEEEE 01, where EEEEE + -15 = 1 so EEEEE = 16, so it's 1 10000 01 = C1

This is basically the value when you start counting by 2s, since once you start counting by 2s and always round down, your sum doesn't increase. There are 2 mantissa bits, so there are always 4 numbers in the range \([2^i, 2^{i+1})\). So you ask yourself, what gap has exactly 4 numbers between consecutive values of \([2^i, 2^{i+1})\), meaning when are you counting by 1? Easy, \([4-8) \Rightarrow \{4, 5, 6, 7\}. When you get to 8 you’re counting by 2s since you have to cover 8 to 16 with only 4 numbers: \{8, 10, 12, 14\}. So it’s 8.0 and you didn’t have to do any work trying to encode numbers in and out of minifloats, since that was what question c was supposed to be about.

```c
minifloat should_be_a_billion() {
    minifloat sum = 0.0;
    for (unsigned int i = 0; i < 1000000000; i++) { sum = sum + 1.0; }
    return(sum);
}
```

Q2) How can I bring you to the C of madness... (4 points)
On the quest, you saw mystery, which should really have been called is_power_of_2, since it took in an unsigned integer and returned 1 when the input was a power of 2 and 0 when it was not. Well, it turns out we can write that in one line! What should the blanks be so that it works correctly?

```c
int is_power_of_2(unsigned int N) { return (N != 0) __ 1 __ (N __ 2)); }
/* ^ is bitwise xor, ~ is bitwise not */
```

<table>
<thead>
<tr>
<th>i</th>
<th>ii</th>
<th>iii</th>
<th>iv</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0 &amp; &amp;</td>
<td>0 &amp; &amp; &amp;</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0 &amp; &amp; &amp;</td>
<td>0 &amp; &amp; &amp; &amp;</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0 ^ &amp;</td>
<td>0 ^ &amp; &amp;</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0 ^ &amp; &amp;</td>
<td>0 ^ &amp; &amp; &amp;</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0 &amp;</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0 &amp; &amp;</td>
<td>0 &amp;</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0 &amp; &amp; &amp;</td>
<td>0 &amp; &amp; &amp;</td>
</tr>
</tbody>
</table>
Q3) Cache money, dollar bills, y’all. (18 points)

We have a 32-bit machine, and a 4 KiB direct mapped cache with 256B blocks. We run the following code from scratch, with the cache initially cold.

```c
uint8_t addup() {
    uint8_t A[1024], sum = 0; // 8-bit unsigned
    touch(A);
    for (int i = 0; i < 1024; i++) { sum += A[i]; }
    return(sum-1);
}

void touch(uint8_t *A) {
    // Touch random location
    // in A between
    // A[0] to A[1023], inclusive
    A[random(0,1024)] = 0;
}
```

a) Assume sum has the smallest possible value after the loop. What would addup return? ____ 255

The smallest value is 0, when you subtract 1 you have “negative overflow” back to the biggest representable number of an 8-bit unsigned value, which is 255.

b) Let A = 0x100061C0. What cache index is A[0]? ____ 1

We tried to design this cache so the numbers are easy. 256B blocks is 8 bits for the offset, or the last two nibbles. 4KiB bytes in cache / 256B bytes/block = 16 blocks or 4 bits there, which is the 3rd nibble from the right. So the index is 1.

c) Let A = 0x100061C0. If the cache has a hit at i = 0 in the loop, what is the maximum value returned by random? ____ 63

If we look at the 8 bits of offset, the value if C0, which is 0b1100 0000, (¼ of the way from the left since the top two bits are 11) so any value of random between 0 and 0b11 1111 would leave us in that block (before the 8 bit offset bubbles over, and another index is randomly touched), and 0b11 1111 is 63.

For d and e, assume we don’t know where A is.

d) What’s the fewest misses caused by the loop? ____ 3

For the fewest misses, let’s assume A is block aligned. There are only 4 blocks ever used by the cache in this case, and touch gives us one free hit, so that’s 3 compulsory misses.

e) What’s the most misses caused by the loop? ____ 4

For the most misses, let’s assume A is not block aligned. There are only 5 blocks ever used by the cache in this case, and touch gives us one free hit, so that’s 4 compulsory misses.

f) If we change to a fully associative LRU cache, how would c, d, e’s values change? (select 1 per box)
The array is well smaller than the array, nothing gets kicked out, so nothing would change

c: ○ up  ○ down  ● same
d: ○ up  ○ down  ● same
e: ○ up  ○ down  ● same

Q4) RISC-V business: I’m in a CS61C midterm & I’m being chased by Guido the killer pimp... (14 points)

a) Write a function in RISC-V code to return 0 if the input 32-bit float = ∞, else a non-zero value. The input and output will be stored in a0, as usual.

(If you use 2 lines=3pts. 3 lines=2 pts)

```riscv
isNotInfinity: lui a1, 0x7F800
             xor a0, a0, a1
             ret
```
Consider the following RISC-V code run on a 32-bit machine:

```assembly
beq a0, x0, done
rs1 rs2 label

done: li a0, 1
ret

fun:  beq a0, x0, done
addi sp, sp, -12
addi a0, a0, -1
sw ra, 8(sp)
sw a0, 4(sp)
sw s0, 0(sp)
jal fun
mv s0, a0
lw a0, 4(sp)
jal fun
add a0, a0, s0
lw s0, 0(sp)
lw ra, 8(sp)
ret
```

beq a0, x0, done
rs1 rs2 label
addi sp, sp, 12
ret

---

**c)** What is the hex value of the machine code for the underlined line? (choose ONE)

- [ ] 0xFE050CE3
- [ ] 0xFE050FE3
- [ ] 0xFE050EFA
- [ ] 0xFE050EE3
- [ ] 0xFE050EEA

---

**d)** What is the one-line C disassembly of `fun` with recursion, and generates the same # of function calls:

```c
uint32_t fun(uint32_t a0) { return !a0 ? 1 : fun(a0-1) + fun(a0-1) }
```

**e)** What is the one-line C disassembly of `fun` that has no recursion (i.e., see if you can optimize it):

```c
uint32_t fun(uint32_t a0) { return 1 << a0 }
```

**f)** Show the call and the return value for the largest possible value returned by (e) above:

```c
fun( 31 ) => 2^31
```

---

**Q5** What's that smell? Oh, it's potpourri! (18 points)

a) What's the ideal speedup of a program that's 90% parallel run on an ∞-core machine? __10__

\[
\frac{1}{s} = \frac{1}{1/(1/10)} = 10x
\]

b) How many times faster is the machine in (a) than a 9-core machine in the ideal case? __2__

\[
\frac{1}{s + (1-s)/9} = \frac{1}{1/10 + (9/10)/9} = \frac{1}{1/10 + 1/10} = \frac{1}{2/10} = 10/2 = 5 \text{ (so } 2x \text{ faster)}
\]

c) What was NOT something companies were doing (yet) to reap PUE benefits? (select ALL that apply)

- [ ] Do away with air conditioners
- [x] Turn the servers completely off (not in an idle state) when not in use.
- Elevate cold aisle temperatures
- Have a UPS (Uninterruptible Power Supply) for the building in case of a power outage
- Pack the servers in freight containers to control air flow

**d)** What was NOT something Dave Patterson talked about in his Turing talk? (select ALL that apply)
- Dennard scaling is going strong
- Machine learning researchers are pushing the boundaries of architecture
- Some researchers have found that floating point has too much range, so they made their own floats
- VLIW (Very Long Instruction Word) architectures are an exciting new area of research
- Quantum computers are at least a decade off from solving the global thirst for computation

**e)** The value of memory pointed to by $x_1$ is 10. Two cores run the following code concurrently:

```
| lw  x2,0(x1)  | lw  x3,0(x1) |
| addi x2,x2,1 | add x3,x3,x3 |
| sw  x2,0(x1)  | sw  x3,0(x1) |
```

...what are possible values of the memory afterward? (select ALL that apply)
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22

**f)** The final machine code bits for `beq` are known after: compiler • assembler • linker • loader
This stage handles forward references: compiler • assembler • linker • loader
This stage reads a DLL: compiler • assembler • linker • loader

**g)** All 61C students were asked to record the time they spent learning C, and we wrote some Spark code to calculate the AVERAGE time every student spent learning C. (select ONE per column)

```
CRDD = sc.parallelize([('Ana', 10), ('Sue', 50), ('Ana', 20)])

def C_init(L):    return (L[0], _____ii_____)  


def C_avg(L):     return [(L[0], L[1][0]/L[1][1])]  

CRDD._____i_____(C_init).reduceByKey(C_sum)._____iii_____(C_avg).collect() 
[('Ana', 15), ('Sue', 50)]
```

- **i)** `map` • `flatMap` • `reduce` • `reduceByKey`
  - ii) `__L[1]` • `__L[0],__L[1]`
  - iii) `map` • `flatMap` • `reduce` • `reduceByKey`

**h)** Mark all **necessary** conditions to convert this code to SIMD for a ≈4x boost. (select ALL that apply)

```
void shift_vector( int *X, int n, int s ) {  for(int i=0; i < n; i++) 
   X[i] += s;   }
```

- Loop needs to be unrolled
- The CPU must have multiple cores
- It needs to use 128-bit registers
- There needs to exist a tail case, for when $n$ is not divisible by 4
- $x[i] += s$ needs to change to $x[i] = x[i] + s$
- $i$ needs to be declared as an **unsigned** int
- SIMD instructions need to be added

Remember, if you didn't do as well as you'd hoped, remember that there's always the final exam which will allow you to clobber this midterm. (also note that if this exam was too hard, the eventual overall average of the class will be lower, and we'll ooch you all up, so you have that going for you, which is good)