e) I’m getting the message “cannot execute binary file”. In one or two sentences, explain what the problem is and how to fix it. (1pt w/explanation)

Assuming not a corrupted file, then the executable was compiled on another machine with a different architecture and can’t be read. Re-compile on the current machine.

f) In our 32-bit single-precision floating point representation, we decide to convert one significand bit to an exponent bit. How many denormalized numbers do we have relative to before? (Circle one)

More
Fewer
Half as many because lost a significand bit (1 pt)

Rounded to the nearest power of 2, how many denorm numbers are there in our new format? (Answer in IEC format) (1 pt)

22 significand bits + sign bit but not counting ±0, so exactly $2^{31}-2$ denoms

8 Mebi #s

Question 2: Flippin’ Fo’ Fun (10 points, 14 minutes)

Assume that the most significant bit (MSB) of $x$ is a 0. We store the result of flipping $x$’s bits into $y$. Interpreted in the following number representations, how large is the magnitude of $y$ relative to the magnitude of $x$? Circle ONE choice per row. (2 pts each)

|                  | $|y| < |x|$ | $|y| = |x|$ | $|y| > |x|$ | Can’t Tell |
|------------------|-----------|-----------|-----------|------------|
| Unsigned         |           |           |           |            |
| One’s Complement | $|y| < |x|$ |           |           |            |
| Two’s Complement | $|y| < |x|$ |           |           |            |
| Sign and Magnitude| $|y| < |x|$ | $|y| = |x|$ | $|y| > |x|$ |            |
| Biased Notation (e.g. FP exponent) | $|y| < |x|$ | $|y| = |x|$ | $|y| > |x|$ | Can’t Tell |

- In unsigned, a number with the MSB of 1 is always greater than one with a MSB of 0.
- In one’s complement, flipping all of the bits is the negation procedure, so the magnitude will be the same.
- In two’s complement, $y$ is a negative number. Its magnitude can be found by applying the negation procedure, which is flipping the bits and then adding 1, resulting in a larger magnitude than $x$.
- In sign and magnitude, the 2\textsuperscript{nd} MSB bit will determine the relative magnitudes of $x$ and $y$, so you can’t tell for certain.
- In biased notation, you read the number the same as unsigned but apply a constant bias to BOTH numbers, so the relation is the same as in unsigned numbers.
Question 1: Potpourri

a) We now need 6-bit register fields (rs, rt, rd) to specify 64 registers. Keeping opcode (6), that leaves 32-6-6-6=14 bits for the immediate field, and we recall the optimization that branches don’t count by bytes (because addresses are always a multiple of 4 byte) but by instructions, so a branch can reach $2^{14}$ instrs. (2 pts)

b) J-format instructions don’t have register fields, so nothing changes here. None. (1 pt)

c) We have 32-6-6-6-5=3 bits for funct. Since opcode=0, we have $2^3 = 8$ R-type instrs. (2 pts)

d) Recall that the “old-school” register file was 32-registers “high” x 32 bits/register “wide” = $2^5 x 2^5 = 2^{10}$ bits = 1 kibibits in total. If there are now a total of 64 registers, then the register file doubles in height, so it’d now have $2^{11}$ bits = 2 kibibits. If each bit cost $2^8 = 256$ cents each, that’s $2^{11}$ bits x $2^8$ cents/bit = $2^{19}$ cents = 512 kibicents. (2 pts)

e) 0xc14c0000 interpreted as a float is 0b1100 0001 0100 1100 0000 0000 0000 0000, or separated by the IEEE 754 fields: [1]100 0001 0100 1100 0000 0000 0000 0000. The first 1 tells us it’s negative. The second field is 130. 130 minus our bias of 127 is an exponent of 3. So now we can write this as we normally do: -1 x $1.10011 x 2^3$, (not forgetting the implicit leading 1) and the $2^3$ means we shift the binary point three spaces to the right, yielding the number -1100.111, which is -12.75. (3 pts)

f) The smallest positive normalized number has a sign bit of 0 and an exponent field of E=1 (remember that E=0 is reserved for denorms and ±0). The smallest number in magnitude will have a mantissa field of all zeros, yielding |0|0000 0001|0000 0000 0000 0000 0000 0000 0000 0000| = 0x00800000, which we interpret as $(-1)^0 x (1.0...0) x 2^{1-127} = 2^{126}$. (3 pts)

g) This was a hard question. We recall there were two infinities, $-\infty$ and $+\infty$ and that their formats were special; we’d reserved all ones in the exponent and zeros in the mantissa especially for it. So that means they look like 0bX111 1111 1000 0000 0000 0000 0000 0000 (= 0xF7F80000), where X is 0 for $+\infty$ and 1 for $-\infty$. Well the comment says to make them the same. What instruction (with an argument of simply “1”) can do that? Why shift left logical, which would push the leftmost bit off the edge yielding 0xFF000000. Now, the second blank needs to look at $a0$ and if it’s 0xFF000000 (either infinity) then $v0$ should be set to 0, otherwise set $v0$ to any non-zero value. We need something like “not-equal-to”, or (in C): $v0 = (a0 != 0xFF000000)$. The logical operation xor fits the bill, because xor is a “balancing” operation … when the arguments are perfectly “balanced” (i.e. equal), it is a zero. Otherwise it’s not. So xor is like “not equal to”, and xor (not xor) is “equal to”). Thus the answer is: (4 pts)

```
slh $a0 $a0 1
xor $v0 $a0 0xFF000000
jr $ra
```

IsNotInfinity: movl %edi, %eax
slh $1, %eax # make +/- Inf look the same
xorl $0xFF000000, %eax
ret
M3) What is that Funky Smell? Oh, it’s just Potpourri (10 pts)

a) This question asked for non-negative floating point numbers < 2. This did NOT include -0. Some important things to remember are that all positive denorm numbers count and the floating point representation of +2 is 0x40000000 (exponent of 0x80). So non-negative floating point numbers less than 2 are any combination where the 2 most significant bits are 0’s. This leaves any combination of the lower 30 bits, so there are $2^{30}$ such numbers. (1 pt)

+0.5 pt for value, +0.5 pt for work WITH correct value.

b) A jump instruction on a 32-bit MIPS system sets $PC=(PC+4[31:28], \text{target address, 0b00})$. Biggest jump would happen when you are right before a boundary (e.g. 0x0FFFFFFC). Then PC+4 is across the boundary at 0x10000000 and your farthest jump ends up at 0x1FFFFFFFC (assuming your target address field was all 1’s). This distance is $2^{28} = 256 \text{ MiB}$. (2 pts)

+1 pt for value, +1 pt for IEC format WITH correct value.

c) There were a number of necessary corrections, some of which could be combined with others to fit into the 5 given spaces. (7 pts)

+1 pt, Line 1: CHANGE return type of count_az() to int *.

+1 pt, Line 3: CHANGE count to a pointer to a malloc-ed array of 26 ints.

+1 pt, Line 4: ADD line to initialize array from Line 3 to zeros.

(the previous two could have been combined into a calloc call in Line 3)

+1 pt, Line 7: CHANGE &str to *str (need to dereference instead of getting address).

+1 pt, Line 7: CHANGE 0x97 constant to 97 (0x97 sits in extended ASCII codes).

(the previous two should have been combined into a single CHANGE)

+1 pt, Line 15: REMOVE free(str) because you need to return the array.

+1 pt, Line 15: ADD return count to return the array.

(the previous two should have been combined into a single CHANGE)
**Question 4: Let Me Float This Idea By You** (9 Points, 16 Minutes)

For a very simple household appliance like a thermostat, a more minimalistic microprocessor is desired to reduce power consumption and hardware costs. We have selected a **16-bit** microprocessor that does not have a floating-point unit, so there is no native support for floating point operations (no `float/double`). However, we’d still like to represent decimals for our temperature reading so we’re going to implement floating point operations in software (in C).

a) Define a new variable type called `fp`: *(1 pt)*

```c
typedef int fp; _______________________
```

Many people were not sure what to do here. 1 pt was given mainly to those who wrote a valid statement using `typedef` or the `#define` directive, or were close. Struct definitions were also accepted.

We have decided to use a representation with a **5-bit exponent field** while following all of the representation conventions from the MIPS 32-bit floating point numbers except **denorms**.

Fill in the following functions. Not all blanks need to be used. You can call these functions and assume proper behavior regardless of your implementation. Assume our hardware implements the C operator “>>” as *shift right arithmetic*.

b) *(1 pt)*

```c
/* returns -num */
fp negateFP(fp num) {
    return ____________________________;
}
```

If you assumed 32-bit type, then using 0x8000 was okay.

c) *(1 pt mask/shift, 1 pt bias)*

```c
/* returns the signed value of the exponent */
int getExp(fp num) {
    ________________________________
    return ____________________________;
}
```

0x7c00 to zero out everything but the exponent field, shift right by 10 to get the unsigned value, then subtract bias of $2^5 - 1 = 15$ to get the actual signed value.
d) (1 pt per line)

```c
/* multiplies floating point num by 2^n, while detecting over/underflow */
/* remember, there are no denoms */
fp multPow2(fp num,int n) {
    int exp = getExp(num) + n; /* get exponent or exponent + n */
    if(_exp > 15) exit(1); #overflow
    if(_exp < -15) exit(-1); #underflow
    _num &= 0x83FF; /* zero old exponent */
    return _num | ((exp + 15) << 10); /* set new exponent */
}
```

5 pts total:

First line: 1pt for trying to get the exponent by means of getExp(num) or manually retrieving it.
Second and third line: −0.5 pt each line if the numeric value on the right was close, but not correct.
Fourth and fifth: needed to correctly zero out the exponent field of num, and OR or add the modified exponent back into that field. 1pt for not forgetting to re-add the bias, and 1pt for getting the masking/shifting right.

Other:

−1 pt for left shifting the exponent by n instead of adding.
If you didn’t add the 15 bias because in getExp() you didn’t subtract the 15 bias, then I didn’t mark you off for that.
**Question 5: Floating Point (10 pts)**

Assume integers and IEEE 754 single precision floating point are **32 bits wide**.

(a) Convert from IEEE 754 to decimal: **0xC0900000** [3 pts]

S = 1, E = 0b1000 0001, M = 0010…0; 

\[-1.001_2 \times 2^2 = -100.1_2\]

\[-4.5\]

(b) What is the smallest positive integer that is a power of 2 that can be represented in IEEE 754 but not as a signed int? You may leave your answer as a power of 2. [2 pts]

Largest 32-bit signed int is \[2^{31} - 1\].

\[2^{31}\]

(c) What is the **smallest positive** integer \(x\) such that \(x + 0.25\) can't be represented? You may leave your answer as a power of 2. [3 pts]

Need \(2^{-2}\) digit to run off end of mantissa, so

\[10000000000000000000000.012 = 1.000000000000000000000001 \times 2^{22}\]

\[2^{22}\]

(d) We have the following word of data: **0xFFC00000**. Circle the number representation below that results in the most negative number. [1 pt]

<table>
<thead>
<tr>
<th>Unsigned Integer (positive number)</th>
<th>Two’s Complement (negative number)</th>
<th>Floating Point (NaN)</th>
</tr>
</thead>
</table>

(e) If we decide to stray away from IEEE 754 format by making our Exponent field 10 bits wide and our Mantissa field 21 bits wide. This gives us (circle one): [1 pt]

MORE PRECISION // LESS PRECISION

**Question 6: Performance (4 pts)**

We are using a processor with clock period of 1 ns.

(a) Program A contains 1000 instructions with a CPI of 1.2. What is the CPU time spent executing program A? [2 pts]

\[\text{CPU Time} = 1000 \times 1.2 \times 1 \text{ ns} = 1200 \text{ ns} = 1.2 \mu\text{s}\]

(b) Program B contains 500 instructions but accesses memory more frequently, what is the maximum CPI that program B can have without executing slower than program A? [2 pts]

Half as many instructions, so can have twice as big CPI.

2.4
**Question 1: Number Representation (8 pts)**

a) Convert \(0x1A\) into base 6. Don’t forget to indicate what base your answer is in! [1 pt]

\[
0x1A = 0b1\,1010 = 16 + 8 + 2 = 26 = 4 \times 6^1 + 2 \times 6^0 \]

\[42_6\]

b) In IEEE 754 floating point, how many numbers can we represent in the interval \([10,16)\)? You may leave your answer in powers of 2. [3 pts]

\[10 = 0b1010 = 1.01 \times 2^3 \text{ and } 16 = 0b10000 = 1.0 \times 2^4\]

Count all numbers with Exponent of \(2^3\) and Mantissa bits of the form \{1b’0, 1b’1, 21{1b’X}\} and \{1b’1, 22{1b’X}\}, for a total of \(2^{21} + 2^{22}\) numbers.

\[2^{22} + 2^{21} = 3 \times 2^{21}\]

c) If we use 7 Exponent bits, a denorm exponent of -62, and 24 Mantissa bits in floating point, what is the largest positive power of 2 that we can multiply with 1 to get underflow? [2 pts]

Smallest denorm is \(2^{-62} \times 0.0000\,0000\,0000\,0000\,0000\,0000\,0001 = 2^{-86}\), which is representable. So the next smaller power of 2 is unrepresentable and causes underflow.

\[2^{-87}\]

Local phone numbers in the USA typically have 7 decimal digits, which use the symbols 0 to 9. For example, Jenny Tutone’s phone number is:

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Line Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>867</td>
<td>5309</td>
</tr>
</tbody>
</table>

d) How many unique phone numbers can be encoded by this scheme? [1 pt]

\[10^7\]

e) How many bits would we need to represent a phone number if we treated it as a single 7-digit decimal? You may use \(\log()\) and \(\text{ceil()\ in your answer and the variable } E\) to represent the correct answer to part (d). [1 pt]

\[\text{ceil}(\log_2(E))\]