CS61C
Spring 2026

C, Floating Point, IEC
Discussion 3

1 Floating Point

1.1 Convert the following single-precision floating point numbers from hexadecimal to decimal or from decimal to hexadecimal using the IEEE 754 Floating Point Standard. You may leave your answer as an expression.

a) `8.25`

Answer: `0x41040000`

First, we write `8.25` into binary. Splitting `8.25` into its integer and decimal portions, we can determine that `8` will be encoded in binary as `1000` and `0.25` will be `.01` (the 1 corresponds to the 2^{-2} place). So, `8.25 = 1000.01`. In normalized form, we get $1.00001_2 \times 2^3$.

Our floating point representation has three parts: the sign, exponent and significand. Our number is positive, so our sign bit -1^S will be 0. Solving for the significand, we know that our number will have a non-zero exponent. This is a normalized number, so we will have a leading 1 for our mantissa. Thus, our significand will be the digits after the decimal point, `00001000`. Finally, we can solve for the exponent. The power in the normalized form is 3, and we must use the bias in reverse to find what exponent we encode in binary. 3 subtracted by a bias of `-127` results in `130`, so our exponent is `0b10000010`. Our final binary number concatenated is `0 100 0001 0 000 0100 0000 0000 0000 0000`, or `0x41040000`.

b) `39.5625`

Answer: `0x421E4000`

Writing `39.5625` in binary results in the bits `100111.1001`. In normalized form, we get 1.001111001×2^5. We can find our floating point exponent with $2^5 = 2^{\text{exp} + \text{bias}} = 2^{\text{exp} - 127} \Rightarrow \text{exp} = 132 =$ `0b10000100`. From the normalized form, our significand is `0b00111100100` and our sign bit is zero. Our final binary number is `0b0 100 0010 0001 1110 0100 0000 0000 0000` which is `0x421E4000`.

c) `0x00000F00`

Answer: $(2^{-12} + 2^{-13} + 2^{-14} + 2^{-15}) * 2^{-126}$

For `0x00000F00`, splitting up the hexadecimal gives us a sign bit and exponent bit of 0, and a significand of `0b 000 0000 0000 1111 0000 0000`. Since the exponents bits are 0, we know that this is a denormalized positive number. We can find out the true power by adding the bias + 1 to get a power of `-126`.

Then, we can evaluate the mantissa by inspecting the bits that are 1 to the right of the decimal point, and finding the corresponding negative power of two. This results in the mantissa evaluated as $2^{-12} + 2^{-13} + 2^{-14} + 2^{-15}$. Combining these get the extremely small number $(-1)^0 * 2^{-126} * (2^{-12} + 2^{-13} + 2^{-14} + 2^{-15})$

d) `0`

Answer: `0x00000000` or `0x80000000`

To represent zero in floating point, the significand will be zero but we need to set the correct exponent. Recall that any non-zero exponent will indicate a *normalized* float which means the significand has a leading 1. Zero's binary representation does not have a leading 1, so we need an exponent of zero to indicate a *denormalized* float which implies a leading zero in front of the significand. The remaining sign bit determines the sign of zero, allowing positive zero to be represented as `0x00000000` and negative zero as `0x80000000`.

e) `0xFF94BEEF`

Answer: NaN

Certain exponent fields are reserved for representing special values. Floating point representations with exponents of 255 and a zero significand are reserved for ± ∞, and exponents of 255 with a nonzero significand are reserved for representations of NaN. Deconstructing the fields of this number gives an exponent of `0b11111111 = 255` and a nonzero significand which indicates that this represents NaN. Note that there are many possible ways to represent NaN.

f) ∞

Answer: `0x7F800000`

Certain exponent fields are reserved for representing special values. Floating point representations with exponents of 255 and a zero significand are reserved for $\pm\infty$, and exponents of 255 with a nonzero significand are reserved for represntations of NaN. Because we need to represent positive infinity, we use a 0 for the sign bit, 255 for the exponent field, and a zero significand giving `0b01111111100000000000000000000000` or `0x7F800000` for positive ∞. Note that $-\infty$ would be `0xFF800000`.

g) `1/3`

Answer: N/A - impossible to represent in single-precision floating point, we can only approximate it

2 More Floating Point

As we saw above, not every number can be represented perfectly using floating point. For this question, we will only look at positive numbers.

2.1 What is the next smallest number larger than 2 that can be represented completely?

Answer: $2 + 2^{-22}$

First,we start at the number 2 and write it in the normalized form. Then we increment the number by the smallest amount possible, which is the same as incrementing the significand by 1 at the rightmost location, which adds 2^{-22}.

Normalized: $2 = 10.000... = 1.000...00 \times 2^1$

Increment: $2 = (1.000...00 + 2^{-23}) \times 2^1 = (1 + 2^{-23}) \times 2 = 2 + 2^{-22}$

2.2 What is the next smallest number larger than 4 that can be represented completely?

Answer: $4 + 2^{-21}$

Similarly, we write 4 in its normalized form, and increment it by the smallest amount possible. This is the same as incrementing the significand by 1 at the rightmost location.

Normalized: $4 = 100.000... = 1.000...00 \times 2^2$

Increment: $(1.000...00 + 2^{-23}) \times 2^2 = (1 + 2^{-23}) \times 4 = 4 + 2^{-21}$

2.3 What is the largest odd number that we can represent? Hint: at what power can we only represent even numbers?

Answer: $2^{24} - 1$

To find the largest odd number we can represent, we want to find when odd numbers will stop appearing. Because we are always multiplying the significand by a power of 2^x, we will only be able to represent even numbers when the exponent grows large enough. In particular, odd numbers will stop appearing when the significand's LSB has a step size (distance between each successive number) of 2, so the largest odd number will be the first number with a step size of 2, subtracted by 1. After this number, only even numbers can be represented in floating point.

We can think of each binary digit in the significant as corresponding to a different power of 2 to get to a final sum. For example, 0b1011 can be evaluated as $2^3 + 2^1 + 2^0$, where the MSB is the 3rd bit and corresponds to 2^3 and the LSB is the 0th bit at 2^0.

We want our LSB to correspond to 2^1, so by counting the number of mantissa bits (23) and including the implicit 1, we get a total exponent of 24. The smallest number with this power would have a mantissa of 00..00, so after taking account of the implicit 1 and subtracting, this gives $2^{24} - 1$

3 C Generics

3.1 True or False: In C, it is possible to directly dereference a `void *` pointer, e.g.

```
... = *ptr;
```

False. To dereference a pointer, we must know the number of bytes to access from memory at compile time. A `void *` pointer contains the address for an arbitrary region of memory without a known size, so they cannot be dereferenced – they must be typecast beforehand (e.g. `... = *((int *) ptr)`)

3.2 Generic functions (i.e., generics) in C use `void *` pointers to operate on memory without the restriction of types. Generic pointers do not support dereferencing, as the number of bytes to access from memory is not known at compile-time. They instead use byte handling functions such as `memcpy` and `memmove`.

Implement `rotate`, which will prompt the following program to generate the provided output.

```
int main(int argc, char *argv[]) {
  int array[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  print_int_array(array, 10);
  rotate(array, array + 5, array + 10);
  print_int_array(array, 10);
  rotate(array, array + 1, array + 10);
  print_int_array(array, 10);
  rotate(array + 4, array + 5, array + 6);
  print_int_array(array, 10);
  return 0;
}
```

Output:

```
$ ./rotate
Array: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Array: [6, 7, 8, 9, 10, 1, 2, 3, 4, 5]
Array: [7, 8, 9, 10, 1, 2, 3, 4, 5, 6]
Array: [7, 8, 9, 10, 2, 1, 3, 4, 5, 6]
```

```
void rotate(void *front, void *separator, void *end) {
  size_t width = (char *) end - (char *) front;
  size_t prefix_width = (char *) separator - (char *) front;
  size_t suffix_width = width - prefix_width;
  char temp[prefix_width];
  memcpy(temp, front, prefix_width);
  memmove(front, separator, suffix_width);
  memcpy((char *) end - prefix_width, temp, prefix_width);
}
```

See slides provided under “Discussion Resources” for a visual walkthrough of the solution.

4 IEC Prefixes and Symbols

$\boxed{4.1}$ Convert the following quantities into the number of bytes each term represents (you may leave your answers in terms of powers of 2).

a) 1 KiB

Answer: 2^{10} Bytes

We can use the 61C reference card (or the precheck worksheet) for the values of the SI prefixes. A Kibi- is 2^{10}, so 1 KiB = 2^{10} bytes.

b) 32 MiB

Answer: 2^{25} Bytes

We know that one MiB = 2^{20} Bytes, so we have $32 \times 2^{20} = 2^5 \times 2^{20} = 2^{25}$ bytes.

c) 16 Gib

Answer: 2^{31} Bytes

16 Gib = $2^4 \times 2^{30} = 2^{34}$. Notice that we have 16 Gib which is 16 Gibi*bits* – one byte is $8 = 2^3$ bits, so 2^{34} / $2^3 = 2^{31}$ Bytes

d) 20 KiB

Answer: 5×2^{12} Bytes

We can factor 20 into 4×5 for a solution in terms of powers of 2. 20 KiB = $(4 \times 5) \times 2^{10} = (2^2 \times 5) \times 2^{10} = 5 \times 2^{12}$ Bytes.

$\boxed{4.2}$ Rewrite the following quantities using IEC Prefixes.

a) 2048 B

Answer: 2 KiB

$2048 = 2^{11} = 2^{10+1}$

$= 2^1 \times 2^{10} = 2$ KiB.

b) 2^{38} B

Answer: 256 GiB

2^{38} can be rewritten as $2^{30+8} = 2^8 \times 2^{30} = 256 \times 2^{30} = 256$ GiB.