

## 1 Pre-Check

This section is designed as a conceptual check for you to determine if you conceptually understand and have any misconceptions about this topic. Please answer true/false to the following questions, and include an explanation:

- T** 1.1 For the same cache size and block size, a 4-way set associative cache will have fewer index bits than a direct-mapped cache.
- F** 1.2 Any cache miss that occurs when the cache is full is a capacity miss.  
False. When the cache is full, you can still get compulsory misses (when a block of data is put in the cache for the first time) and conflict misses (if a fully associative cache with LRU replacement wouldn't have missed).
- F** 1.3 Increasing cache size by adding more blocks always improves (increases) hit rate.  
False. Whether this improves the hit rate for a given program depends on the characteristics of the program. As an example, it is possible for a program that only consists of a loop that runs through an array once to have each access be separated by more than one block (say, the block size is 8B, but we have an integer array and accessing every fourth element, so our access are separated by 16B). This makes every miss a compulsory miss, and there is no way for us to reduce the number of compulsory misses just by adding more blocks to our cache.

## 2 Understanding T/I/O

When working with caches, we have to be able to break down the memory addresses we work with to understand where they fit into our caches. There are three fields:

**Tag** - Used to distinguish different blocks that use the same index. Number of bits: ( $\#$  of bits in memory address) - Index Bits - Offset Bits

**Index** - The set that this piece of memory will be placed in. Number of bits:  $\log_2(\#$  of indices)

**Offset** - The location of the byte in the block. Number of bits:  $\log_2(\text{size of block})$

Given these definitions, the following is true:

$$\log_2(\text{memory size}) = \text{address bit-width} = \# \text{ tag bits} + \# \text{ index bits} + \# \text{ offset bits}$$

Another useful equality to remember is:

$$\text{cache size} = \text{block size} * \text{num blocks}$$

- 2.1 Assume we have a direct-mapped byte-addressed cache with capacity  $2^5$  32B and block size of 8B. Of the 32 bits in each address, which bits do we use to find the index of the cache to use?

T I O  
27 2 3 3~4 bits

- 2.2 Which bits are our tag bits? What about our offset?

5~31 bits 0~2 bits

- 2.3 Classify each of the following byte memory accesses as a cache hit (H), cache miss (M), or cache miss with replacement (R). Tip: Drawing out the cache can help you see the replacements more clearly.

Address	T/I/O	Hit, Miss, Replace
0x00000004	000/00/100	M compulsory
0x00000005	000/00/101	H
0x00000068	011/01/000	M compulsory
0x000000C8	110/01/000	R compulsory 第一次进入
0x00000068	011/01/000	R conflict 2个tag交替争抢
0x000000DD	110/11/101	M compulsory
0x00000045	010/00/101	R compulsory 第一次进入
0x00000004	000/00/100	R capacity 不同的tag进入同一位置
0x000000C8	110/01/000	R capacity

### 3 Cache Associativity

In the previous problem, we had a Direct-Mapped cache, in which blocks map to specifically one slot in our cache. This is good for quick replacement and finding out block, but not good for efficiency of space!

This is where we bring associativity into the matter. We define associativity as the number of slots a block can potentially map to in our cache. Thus, a Fully-Associative cache has the most associativity, meaning every block can go anywhere in the cache.

For an  $N$ -way associative cache, the following is true:

$$N * \# \text{ sets} = \# \text{ blocks}$$

$$2^{5/2^3/2}$$

T I O  
4 1 3

- 3.1 Here's some practice involving a 2-way set associative cache. This time we have an 8-bit address space, 8 B blocks, and a cache size of 32 B. Classify each of the following accesses as a cache hit (H), cache miss (M) or cache miss with replacement (R). For any misses, list out which type of miss it is. Assume that we have an LRU replacement policy (in general, this is not the case).

Address	T/I/O	Hit, Miss, Replace
0b0000/0/100 ①		M compulsory
0b0000/0/101 ②	<div style="display: inline-block; border: 1px solid black; padding: 2px;"> <div style="border: 1px solid black; padding: 2px;">0</div> <div style="border: 1px solid black; padding: 2px;">0</div> </div> set 0	H
0b0110/1/000 ③	<div style="display: inline-block; border: 1px solid black; padding: 2px;"> <div style="border: 1px solid black; padding: 2px;">0 0 0</div> <div style="border: 1px solid black; padding: 2px;">0 0</div> </div> set 1	M compulsory
0b1100/1/000 ④		M compulsory
0b0110/1/000 ⑤		H
0b1101/1/101 ⑥		R compulsory
0b0100/0/101 ⑦		M compulsory
0b0000/0/100 ⑧		H
0b1100/1/000 ⑨		R capacity

- 3.2 What is the hit rate of our above accesses?

$$\frac{1}{3}$$

## 4 The 3 C's of Cache Misses

4.1 Go back to questions 2 and 3 and classify each M and R as one of the 3 types of misses described below:

1. **Compulsory:** First time you ask the cache for a certain block. A miss that must occur when you first bring in a block. Reduce compulsory misses by having longer cache lines (bigger blocks), which bring in the surrounding addresses along with our requested data. Can also pre-fetch blocks beforehand using a hardware prefetcher (a special circuit that tries to guess the next few blocks that you will want).
2. **Conflict:** Occurs if, hypothetically, you went through the ENTIRE string of accesses with a fully associative cache (with an LRU replacement policy) and wouldn't have missed for that specific access. Increasing the associativity or improving the replacement policy would remove the miss.
3. **Capacity:** Capacity misses are independent of the associativity of your cache. If you hypothetically ran the ENTIRE string of memory accesses with a fully associative cache (with an LRU replacement policy) of the same size as your cache, and it was a miss for that specific access, then this miss is a capacity miss. The only way to remove the miss is to increase the cache capacity.

Note: The test you can use to see if a miss is a conflict miss is the same as the test you can use to see if a miss is a capacity miss.

Note: There are many different ways of fixing misses. The name of the miss doesn't necessarily tell us the best way to reduce the number of misses.


## 5 Code Analysis

Given the follow chunk of code, analyze the hit rate given that we have a byte-addressed computer with a total memory of **1 MiB**. It also features a **16 KiB** Direct-Mapped cache with **1 KiB** blocks. Assume that your cache begins cold.

```
#define NUM_INTS 8192    // 213
int A[NUM_INTS];        // A lives at 0x10000
int i, total = 0;
for (i = 0; i < NUM_INTS; i += 128) {
    A[i] = i;            // Line 1
}
for (i = 0; i < NUM_INTS; i += 128) {
    total += A[i];       // Line 2
}
```

$2^{20}$   
 $2^{14}$      $2^{10}$   
 T I 0  
 6 4 10

$2^8 = 256$  个 int  
 1 行  $2^{10}$  B  
 int  
 4B

cache: 16 lines    Memory: 64 caches  


5.1 How many bits make up a memory address on this computer?

20

5.2 What is the T:I:O breakdown?

6 4 10

5.3 Calculate the cache hit rate for the line marked Line 1:

50%

5.4 Calculate the cache hit rate for the line marked Line 2:

50%

## 6 AMAT

Recall that AMAT stands for Average Memory Access Time. The main formula for it is:

$$\text{AMAT} = \text{Hit Time} + \text{Miss Rate} * \text{Miss Penalty}$$

In a multi-level cache, there are two types of miss rates that we consider for each level.

*access / 到达L1级别的access次数*

**Global:** Calculated as the number of accesses that missed at that level divided by the total number of accesses to the cache system.

**Local:** Calculated as the number of accesses that missed at that level divided by the total number of accesses to that cache level. *access / 到达当前级别的access次数*

- 6.1 • An L2\$, out of 100 total accesses to the cache system, missed 20 times. What is the global miss rate of L2\$?

$$20 / 100$$

- 6.2 If L1\$ had a miss rate of 50%, what is the local miss rate of L2\$?

$$20 / (100 \cdot 50\%)$$

Suppose your system consists of:

1. An L1\$ that has a hit time of 2 cycles and has a local miss rate of 20% *20/100*
2. An L2\$ that has a hit time of 15 cycles and has a global miss rate of 5% *5/100*
3. Main memory where accesses take 100 cycles

- 6.3 What is the local miss rate of L2\$?

$$5 / (100 \cdot 20\%) = 25\%$$

- 6.4 What is the AMAT of the system?

$$L_2: 15 + \frac{0.25}{0.25} \cdot 100 = 20 \quad 40$$

$$L_1: 2 + 0.2 \cdot 20 = 6 \quad 10$$

- 6.5 Suppose we want to reduce the AMAT of the system to 8 cycles or lower by adding in a L3\$. If the L3\$ has a local miss rate of 30%, what is the largest hit time that the L3\$ can have?

$$2 + 0.2 (15 + 0.25 (x + 0.3 \cdot 100)) = 8$$

$$x = 30$$

## 1 Pre-Check

This section is designed as a conceptual check for you to determine if you conceptually understand and have any misconceptions about this topic. Please answer true/false to the following questions, and include an explanation:



1.1

Depending on the context, the same sets of bits may represent different things.

1.2

It is possible to get an overflow error in Two's Complement when adding numbers of opposite signs. 相反数相加. 正+负

1.3

If you interpret a N bit Two's complement number as an unsigned number, negative numbers would be smaller than positive numbers.

1.4

If you interpret an N bit Bias notation number as an unsigned number (assume there are negative numbers for the given bias), negative numbers would be smaller than positive numbers.

## 2 Unsigned Integers

~进制

2.1

If we have an  $n$ -digit unsigned numeral  $d_{n-1}d_{n-2}\dots d_0$  in *radix* (or *base*)  $r$ , then the value of that numeral is  $\sum_{i=0}^{n-1} r^i d_i$ , which is just fancy notation to say that instead of a 10's or 100's place we have an  $r$ 's or  $r^2$ 's place. For the three radices binary, decimal, and hex, we just let  $r$  be 2, 10, and 16, respectively.

Let's try this by hand. Recall that our preferred tool for writing large numbers is the IEC prefixing system:

$$\begin{array}{llll} \text{Ki (Kibi)} = 2^{10} & \text{Gi (Gibi)} = 2^{30} & \text{Pi (Pebi)} = 2^{50} & \text{Zi (Zebi)} = 2^{70} \\ \text{Mi (Mebi)} = 2^{20} & \text{Ti (Tebi)} = 2^{40} & \text{Ei (Exbi)} = 2^{60} & \text{Yi (Yobi)} = 2^{80} \end{array}$$

- (a) Convert the following numbers from their initial radix into the other two common radices:

$$\begin{array}{lll} 1. \text{0b10010011} & 3+32+256=147 & \text{0x93} \\ 2. 63 & \text{0b 0011 111} & \text{0x 3F} \\ 3. \text{0b00100100} & 26 & \text{0x 2A} \\ 4. 0 & \text{0b0} & \text{0x0} \\ 5. 39 & \text{0b 0010 011} & \text{0x 27} \\ 6. 437 & \text{0b 001 011 0101} & \text{0x 1B5} \\ 7. \text{0x0123} & \text{0b 0000 0001 0010 0011} & 291 \end{array}$$

- (b) Convert the following numbers from hex to binary:

1. 0xD3AD
2. 0xB33F
3. 0x7EC4

- (c) Write the following numbers using IEC prefixes:  $\text{KMGTPEZY}$

$$\begin{array}{lllll} \bullet 2^{16} & 64 \text{ KB} & \bullet 2^{27} & 128 \text{ MB} & \bullet 2^{43} & 8 \text{ TB} \\ \bullet 2^{34} & 16 \text{ GB} & \bullet 2^{61} & 2 \text{ EB} & \bullet 2^{47} & 128 \text{ TB} \\ & & & & \bullet 2^{59} & 512 \text{ PB} \end{array}$$

- (d) Write the following numbers as powers of 2:

$$\begin{array}{lll} \bullet 2 \text{ Ki} & 2^1 & \bullet 512 \text{ Ki} & 2^{19} \\ \bullet 256 \text{ Pi} & 2^{18} & \bullet 64 \text{ Gi} & 2^{36} \\ & & \bullet 16 \text{ Mi} & 2^{24} \\ & & \bullet 128 \text{ Ei} & 2^{47} \end{array}$$

### 3 Signed Integers

**3.1** Unsigned binary numbers work for natural numbers, but many calculations use negative numbers as well. To deal with this, a number of different schemes have been used to represent signed numbers, but we will focus on two's complement, as it is the standard solution for representing signed integers.

- Most significant bit has a negative value, all others are positive. So the value of an  $n$ -digit two's complement number can be written as  $\sum_{i=0}^{n-2} 2^i d_i - 2^{n-1} d_{n-1}$ .
- Otherwise exactly the same as unsigned integers.
- A neat trick for flipping the sign of a two's complement number: flip all the bits and add 1.



- Addition is exactly the same as with an unsigned number.
- Only one 0, and it's located at 0b0.

For questions (a) through (c), assume an 8-bit integer and answer each one for the case of an unsigned number, biased number with a bias of -127, and two's complement number. Indicate if it cannot be answered with a specific representation.

(a) What is the largest integer? What is the result of adding one to that number?

1. Unsigned? 255 0
2. Biased? 128 -127
3. Two's Complement? 127 -128

(b) How would you represent the numbers 0, 1, and -1?

1. Unsigned? 0x00 0x01 N/A
2. Biased? 0b 0111 1111 0b 1000 0000 0b 0111 1110
3. Two's Complement? 0b 0000 1000 0b 0000 1001 0b 1111 1111

(c) How would you represent 17 and -17?

1. Unsigned? 0b 0001 0001 N/A
2. Biased? 0b 1001 0000 0b 0110 1110
3. Two's Complement? 0b 0001 0001 0b 1110 1111

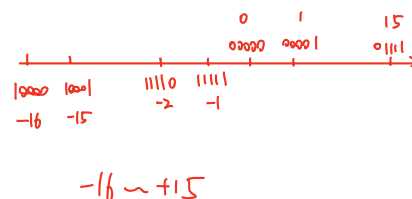
(d) What is the largest integer that can be represented by *any* encoding scheme that only uses 8 bits?

There is no such integer. For example, an arbitrary 8-bit mapping could choose to represent the numbers from 1 to 256 instead of 0 to 255.

(e) Prove that the two's complement inversion trick is valid (i.e. that  $x$  and  $\bar{x} + 1$  sum to 0).

$$\begin{array}{r} 3 \quad -3 \\ 0001 + 1101 = 10000 = 0 \end{array}$$

Note that for any  $x$  we have  $x + \bar{x} = 0b1 \dots 1$ . Adding 0b1 to 0b1...1 will cause the value to overflow, meaning that  $0b1 \dots 1 + 0b1 = 0b0 = 0$ . Therefore,  $x + \bar{x} + 1 = 0$



(f) Explain where each of the three radices shines and why it is preferred over other bases in a given context.

Decimal is the preferred radix for human hand calculations, likely related to the fact that humans have 10 fingers.

Binary numerals are particularly useful for computers. Binary signals are less likely to be garbled than higher radix signals, as there is more "distance" (voltage or current) between valid signals. Additionally, binary signals are quite convenient to design circuits, as we'll see later in the course.

Hexadecimal numbers are a convenient shorthand for displaying binary numbers, owing to the fact that one hex digit corresponds exactly to four binary digits.

## 4 Arithmetic and Counting

4.1

Addition and subtraction of binary/hex numbers can be done in a similar fashion as with decimal digits by working right to left and carrying over extra digits to the next place. However, sometimes this may result in an overflow if the number of bits can no longer represent the true sum. Overflow occurs if and only if two numbers with the same sign are added and the result has the opposite sign.

- (a) Compute the decimal result of the following arithmetic expressions involving 6-bit Two's Complement numbers as they would be calculated on a computer.

Do any of these result in an overflow? Are all these operations possible?

1.  $0b011001 - 0b000111$       ① 
$$\begin{array}{r} 011001 \\ 000111 \\ \hline 010010 = 18 \end{array}$$
2.  $0b100011 + 0b111010$       ② 
$$\begin{array}{r} 100011 \\ 111010 \\ \hline 101101 = 01101 = 9 \end{array}$$
      overflow 截断且溢出
3.  $0x3B + 0x06$       ③ 
$$\begin{array}{r} 0011011 \\ 0000110 \\ \hline 0100001 = 000001 \end{array}$$
      截断, 未溢出,  $-5+6=1$
4.  $0xFF - 0xAA$       ④ 
$$\begin{array}{r} 1111111 \\ 10101010 \\ \hline 01010101 \end{array}$$
      不可能, 因为6bit 无法表示

- (b) What is the least number of bits needed to represent the following ranges using any number representation scheme.

1. 0 to 256      257      9
2. -7 to 56       $56+1+7 = 64$       6
3. 64 to 127 and -64 to -127      128      7
4. Address every byte of a 12 TiB chunk of memory

$$12 \text{ TiB} = 12 \times 2^{40} \text{ bytes} = 2^{44} \text{ bytes}$$

44

## 1 Pre-Check

This section is designed as a conceptual check for you to determine if you conceptually understand and have any misconceptions about this topic. Please answer true/false to the following questions, and include an explanation:

- 1.1 True or False: C is a pass-by-value language.
- 1.2 What is a pointer? What does it have in common to an array variable?
- 1.3 If you try to dereference a variable that is not a pointer, what will happen? What about when you free one?
- 1.4 When should you use the heap over the stack? Do they grow?

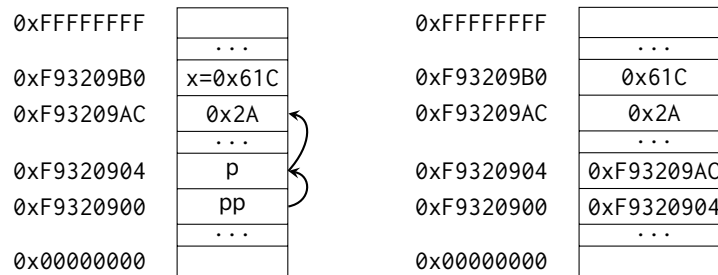
## 2 C

C is syntactically similar to Java, but there are a few key differences:

1. C is function-oriented, not object-oriented; there are no objects.
2. C does not automatically handle memory for you.
  - Stack memory, or *things that are not manually allocated*: data is garbage immediately after the *function in which it was defined* returns.
  - Heap memory, or *things allocated with malloc, calloc, or realloc*: data is freed only when the programmer explicitly frees it!
  - There are two other sections of memory that we learn about in this course, *static* and *code*, but we'll get to those later.
  - In any case, allocated memory always holds garbage until it is initialized!
3. C uses pointers explicitly. If `p` is a pointer, then `*p` tells us to use the value that `p` points to, rather than the value of `p`, and `&x` gives the address of `x` rather than the value of `x`.

On the left is the memory represented as a box-and-pointer diagram.

On the right, we see how the memory is really represented in the computer.



Let's assume that **int\*** **p** is located at **0xF9320904** and **int** **x** is located at **0xF93209B0**. As we can observe:

- **\*p** evaluates to **0x2A** ( $42_{10}$ ).
- **p** evaluates to **0xF93209AC**.
- **x** evaluates to **0x61C**.
- **&x** evaluates to **0xF93209B0**.

Let's say we have an **int \*\*pp** that is located at **0xF9320900**.

2.1 What does **pp** evaluate to? How about **\*pp**? What about **\*\*pp**?

2.2 The following functions are syntactically-correct C, but written in an incomprehensible style. Describe the behavior of each function in plain English.

- (a) Recall that the ternary operator evaluates the condition before the **?** and returns the value before the colon (**:**) if true, or the value after it if false.

```
1 int foo(int *arr, size_t n) {
2     return n ? arr[0] + foo(arr + 1, n - 1) : 0;
3 }
```

- (b) Recall that the negation operator, **!**, returns 0 if the value is non-zero, and 1 if the value is 0. The **~** operator performs a *bitwise not* (NOT) operation.

```
1 int bar(int *arr, size_t n) {
2     int sum = 0, i;
3     for (i = n; i > 0; i--)
4         sum += !arr[i - 1];
5     return ~sum + 1;
6 }
```

- (c) Recall that **^** is the *bitwise exclusive-or* (XOR) operator.

```
1 void baz(int x, int y) {
2     x = x ^ y;
```

```

3     y = x ^ y;
4     x = x ^ y;
5 }
```

(d) (Bonus: How do you write the *bitwise exclusive-nor* (XNOR) operator in C?)

### 3 Programming with Pointers

3.1 Implement the following functions so that they work as described.

(a) Swap the value of two **ints**. *Remain swapped after returning from this function.*

```
void swap(
```

(b) Return the number of bytes in a string. *Do not use strlen.*

```
int mystrlen(
```

3.2 The following functions may contain logic or syntax errors. Find and correct them.

(a) Returns the sum of all the elements in **summands**.

```

1  int sum(int* summands) {
2      int sum = 0;
3      for (int i = 0; i < sizeof(summands); i++)
4          sum += *(summands + i);
5      return sum;
6  }
```

(b) Increments all of the letters in the **string** which is stored at the front of an array of arbitrary length, **n**  $\geq$  **strlen(string)**. Does not modify any other parts of the array's memory.

```

1  void increment(char* string, int n) {
2      for (int i = 0; i < n; i++)
```

```

3         *(string + i)++;
4     }

```

(c) Copies the string `src` to `dst`.

```

1 void copy(char* src, char* dst) {
2     while (*dst++ = *src++);
3 }

```

(d) Overwrites an input string `src` with “61C is awesome!” if there’s room. Does nothing if there is not. Assume that `length` correctly represents the length of `src`.

```

1 void cs61c(char* src, size_t length) {
2     char *srcptr, replaceptr;
3     char replacement[16] = "61C is awesome!";
4     srcptr = src;
5     replaceptr = replacement;
6     if (length >= 16) {
7         for (int i = 0; i < 16; i++)
8             *srcptr++ = *replaceptr++;
9     }
10 }

```

## 4 Memory Management

4.1 For each part, choose one or more of the following memory segments where the data could be located: **code**, **static**, **heap**, **stack**.

- (a) Static variables
- (b) Local variables
- (c) Global variables
- (d) Constants
- (e) Machine Instructions
- (f) Result of `malloc`
- (g) String Literals

- 4.2 Write the code necessary to allocate memory on the heap in the following scenarios
- (a) An array `arr` of  $k$  integers
  - (b) A string `str` containing  $p$  characters
  - (c) An  $n \times m$  matrix `mat` of integers initialized to zero.
- 4.3 What's the main issue with the code snippet seen here? (Hint: `gets()` is a function that reads in user input and stores it in the array given in the argument.)

```

1  char* foo() {
2      char buffer[64];
3      gets(buffer);
4
5      char* important_stuff = (char*) malloc(11 * sizeof(char));
6
7      int i;
8      for (i = 0; i < 10; i++) important_stuff[i] = buffer[i];
9      important_stuff[i] = '\0';
10     return important_stuff;
11 }
```

Suppose we've defined a linked list `struct` as follows. Assume `*lst` points to the first element of the list, or is `NULL` if the list is empty.

```

struct ll_node {
    int first;
    struct ll_node* rest;
}
```

- 4.4 Implement `prepend`, which adds one new value to the front of the linked list. Hint: why use `ll_node **lst` instead of `ll_node*lst`?

```
void prepend(struct ll_node** lst, int value)
```

- 4.5 Implement `free_ll`, which frees all the memory consumed by the linked list.

```
void free_ll(struct ll_node** lst)
```

## 1 Pre-Check

This section is designed as a conceptual check for you to determine if you conceptually understand and have any misconceptions about this topic. Please answer true/false to the following questions, and include an explanation:

F 1.1 True or False. The goals of floating point are to have a large range of values, a low amount of precision, and real arithmetic results

T 1.2 True or False. The distance between floating point numbers increase as the absolute value of the numbers increase.

F 1.3 True or False. Floating Point addition is associative.

## 2 Floating Point

The IEEE 754 standard defines a binary representation for floating point values using three fields.

- The *sign* determines the sign of the number (0 for positive, 1 for negative).
- The *exponent* is in **biased notation**. For instance, the bias is -127 which comes from  $-(2^{8-1} - 1)$  for single-precision floating point numbers.
- The *significand* or *mantissa* is akin to unsigned integers, but used to store a fraction instead of an integer.

The below table shows the bit breakdown for the single precision (32-bit) representation. The leftmost bit is the MSB and the rightmost bit is the LSB.

1	8	23
Sign	Exponent	Mantissa/Significand/Fraction

For normalized floats:

$$\text{Value} = (-1)^{\text{Sign}} * 2^{\text{Exp} + \text{Bias}} * 1.\text{significand}_2$$

For denormalized floats:

$$\text{Value} = (-1)^{\text{Sign}} * 2^{\text{Exp} + \text{Bias} + 1} * 0.\text{significand}_2$$

Exponent	Significand	Meaning
0	Anything	Denorm
1-254	Anything	Normal
255	0	Infinity
255	Nonzero	NaN



Note that in the above table, our exponent has values from 0 to 255. When translating between binary and decimal floating point values, we must remember that there is a bias for the exponent.

- 2.1 How many zeroes can be represented using a float?

2

- 2.2 What is the largest finite positive value that can be stored using a single precision float?

$$\begin{array}{l} \text{exp} = 254 \\ \text{sig} = 23 \uparrow 1 \end{array} \quad 2^{254-127} \cdot 1.111 \dots 11 = (1 + (1-2^{-23})) \times 2^{127}$$

- 2.3 What is the smallest positive value that can be stored using a single precision float?

$$2^{-149}$$

- 2.4 What is the smallest positive normalized value that can be stored using a single precision float?

$$2^{1-127} \cdot 1.000 \dots 0 = 2^{-126}$$

- 2.5 Cover the following single-precision floating point numbers from binary to decimal or from decimal to binary. You may leave your answer as an expression.

- 0x00000000  $0$
- 8.25  $1000.01 = 1.00001 \times 2^3$   $\text{exp} = 3 + 127 = 130$   
*在 sig 部分左移*
- 0x00000F00  $\text{exp} = 0$  *denormalized*  
 $(2^{-12} + 2^{-13} + 2^{-14} + 2^{-15}) \times 2^{-126}$
- 39.5625  $1001111001 \times 2^5$   $127 + 5 = 132 = 10000100$   
 $\text{sig} = 00111001$
- 0xFF94BEEF  $\text{NaN}$
- $-\infty$   $0x \text{FF}800000$

### 3 More Floating Point Representation

Not every number can be represented perfectly using floating point. For example,  $\frac{1}{3}$  can only be approximated and thus must be rounded in any attempt to represent it. For this question, we will only look at positive numbers.

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

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

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

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

- 3.3 Define stepsize to be the distance between some value  $x$  and the smallest value larger than  $x$  that can be completely represented. What is the step size for 2? 4?

$$\begin{array}{l} 2^{-22} \\ 2^{-21} \end{array}$$

- 3.4 Now let's see if we can generalize the stepsize for normalized numbers (we can do so for denorms as well, but we won't in this question). If we are given a normalized number that is not the largest representable normalized number with exponent value  $x$  and with significand value  $y$ , what is the stepsize at that value? Hint: There are 23 significand bits.

$$2^{x - 127 - 23}$$

- 3.5 Now let's apply this technique. What is the largest odd number that we can represent? Part 4 should be very useful in finding this answer.

To find the largest odd number we can represent, we want to find when odd numbers will stop appearing. This will be with step size of 2.

As a result, plugging into Part 4:  $2 = 2^{x-150} \rightarrow x = 151$

This means the number before  $2^{151-127}$  was a distance of 1 (it is the first value whose stepsize is 2) and no number after will be odd. Thus, the odd number is simply subtracting the previous step size of 1. This gives,  $2^{24} - 1$