202114146O159-邓钰川-作业2-2

3.20

0c000000=201326592.考虑无符号数和有符号数第一位是0没差异，所以两个情况都是201326592

3.21

0c000000=000011 00000000000000000000000000
查表可以知道应该是

jal	000011	addr

所应该是
jal 0×00000000

3.22

3.22 [10] <§3.5> What decimal number does the bit pattern 0×0C000000 represent if it is a floating point number? Use the IEEE 754 standard.

0×0c000000 = 0 0001 1000 0000 0000 0000 0000 0000 000
首位0表示正的
指数域是00011000=24，所以指数是24-128+1=-103
尾数域全是0不考虑
所以答案是2^{-103}

$$
\begin{aligned}
&0x0C000000 = 0000\ 1100\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000 = 0\ 00011000\ 00000000000000000000000 \\
&Sign:\ 0 \\
&Biased\ exponent:\ 00011000 = 24 \\
&24 - 127 = -103 \\
&Fraction:\ 00000000000000000000000 \\
&Mantissa:\ 1.0 \\
&Answer:\ \ +1.0 * 2^{-103}
\end{aligned}
$$

3.23

3.23 [10] <§3.5> Write down the binary representation of the decimal number 63.25 assuming the IEEE 754 single precision format.

$63.25 = 111111.01 = 1.1111101 * 2^5$

符号位0，指数域128+5-1=132=10000100

所以是0 1000 0100 1111 1010 0000 0000 0000 000（0x427D0000）

$$
\begin{aligned}
&63.25 = 111111.01(Binary) = 1.1111101 * 2^5 \\
&Fraction:\ 11111010000000000000000 \\
&Biased\ exponent:\ 5 + 127 = 132 = 10000100(Binary) \\
&Sign:\ 0 \\
&Answer:\ 0\ 10000100\ 11111010000000000000000
\end{aligned}
$$

3.27

3.27 [20] <§3.5> IEEE 754-2008 contains a half precision that is only 16 bits wide. The leftmost bit is still the sign bit, the exponent is 5 bits wide and has a bias of 15, and the mantissa is 10 bits long. A hidden 1 is assumed. Write down the bit pattern to represent -1.5625×10^{-1} assuming a version of this format, which uses an excess-16 format to store the exponent. Comment on how the range and accuracy of this 16-bit floating point format compares to the single precision IEEE 754 standard.

题目本身解答

首先是负数，符号位1

$-0.15625 = -1.01 * 2^{-3}$

指数域是16-1-3=12=1100

所以答案表示1011000100000000

3.27.1

$$
\begin{aligned}
&1.5625 * 10^{-1} = 0.15625 = 0.00101(Binary) = 1.01 * 2^{-3} \\
&Fraction:\ 0100000000 \\
&Biased\ exponent:\ -3 + 15 = 12 = 01100(Binary) \\
&Sign:\ 1 \\
&Answer:\ 1\ 01100\ 0100000000
\end{aligned}
$$

$$3.27.2$$

$$Range$$

$$Single Precision IEEE754$$

$$
\begin{aligned}
&Biased\ exponent:\\
&[00000001, 11111110] = [1, 254]\\
&[1-127, 254-127] = [-126, 127]\\
&Fraction: [1, 2)\\
&Range: \quad \pm[1, 2) * 2^{[-126,127]} = (-2^{128}, -2^{-126}] \cup [2^{-126}, 2^{128})
\end{aligned}
$$

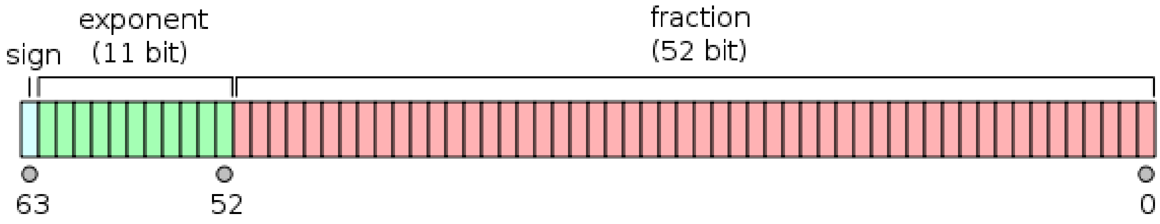
$$IEEE754-2008$$

$$
\begin{aligned}
&Biased\ exponent:\\
&[00001, 11110] = [1, 30]\\
&[1-15, 30-15] = [-14, 15]\\
&Fraction: [1, 2)\\
&Range: \quad \pm[1, 2) * 2^{[-14,15]} = (-2^{16}, -2^{-14}] \cup [2^{-14}, 2^{16})
\end{aligned}
$$

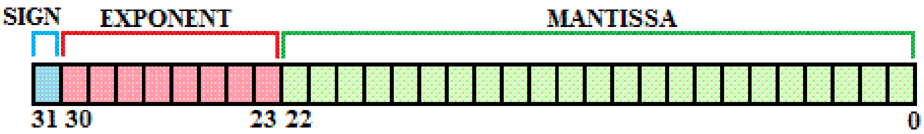
补充

半精度是英伟达在2002年搞出来的，双精度和单精度是为了计算，而半精度更多是为了降低数据传输和存储成本。很多场景对于精度要求也没那么高，例如分布式深度学习里面，如果用半精度的话，比起单精度来可以节省一半传输成本。考虑到深度学习的模型可能会有几亿个参数，使用半精度传输还是非常有价值的。

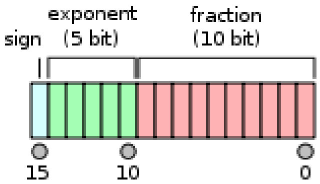
双精度浮点数



单精度浮点数



半精度浮点数



```C++
numpy里面的转换代码
npy_uint32 npy_halfbits_to_floatbits(npy_uint16 h)

{

npy_uint16 h_exp, h_sig;

npy_uint32 f_sgn, f_exp, f_sig;

h_exp = (h&0x7c00u);

f_sgn = ((npy_uint32)h&0x8000u) << 16;

switch (h_exp) {

case 0x0000u: /* 0 or subnormal */

h_sig = (h&0x03ffu);

/* Signed zero */

if (h_sig == 0) {

return f_sgn;

}

/* Subnormal */

h_sig <<= 1;

while ((h_sig&0x0400u) == 0) {

h_sig <<= 1;

h_exp++;

}
```

```
f_exp = ((npy_uint32)(127 - 15 - h_exp)) << 23;

f_sig = ((npy_uint32)(h_sig&0x03ffu)) << 13;

return f_sgn + f_exp + f_sig;

case 0x7c00u: /* inf or NaN */

/* All-ones exponent and a copy of the significand */

return f_sgn + 0x7f800000u + (((npy_uint32)(h&0x03ffu)) <<
13);

default: /* normalized */

/* Just need to adjust the exponent and shift */

return f_sgn + (((npy_uint32)(h&0x7fffu) + 0x1c000u) << 13);

}

}

npy_uint64 npy_halfbits_to_doublebits(npy_uint16 h)

{

npy_uint16 h_exp, h_sig;

npy_uint64 d_sgn, d_exp, d_sig;

h_exp = (h&0x7c00u);

d_sgn = ((npy_uint64)h&0x8000u) << 48;

switch (h_exp) {

case 0x0000u: /* 0 or subnormal */

h_sig = (h&0x03ffu);
```

```
/* Signed zero */

if (h_sig == 0) {

return d_sgn;

}

/* Subnormal */

h_sig <<= 1;

while ((h_sig&0x0400u) == 0) {

h_sig <<= 1;

h_exp++;

}

d_exp = ((npy_uint64)(1023 - 15 - h_exp)) << 52;

d_sig = ((npy_uint64)(h_sig&0x03ffu)) << 42;

return d_sgn + d_exp + d_sig;

case 0x7c00u: /* inf or NaN */

/* All-ones exponent and a copy of the significand */

return d_sgn + 0x7ff0000000000000ULL +

(((npy_uint64)(h&0x03ffu)) << 42);

default: /* normalized */

/* Just need to adjust the exponent and shift */

return d_sgn + (((npy_uint64)(h&0x7fffu) + 0xfc000u) << 42);
```

```
    }
  }
```

3.29

3.29 [20] <§3.5> Calculate the sum of 2.6125×10^1 and $4.150390625 \times 10^{-1}$ by hand, assuming A and B are stored in the 16-bit half precision described in Exercise 3.27. Assume 1 guard, 1 round bit, and 1 sticky bit, and round to the nearest even. Show all the steps.

$26.125 = 1.1010001000 * 2^4$

$0.4150390625 = 1.1010100111 * 2^{-2}$

分别可以表示成为

1.101000100000+1.0000011010100111=1.101010001010

1.1010100011 * 2^4 =11010.100011=26.546875

$$2.6125 * 10^1 = 26.125 = 11010.001(Binary) = 1.1010001000 * 2^4$$
$$4.150390625 * 10^{-1} = 0.4150390625 = 0.0110101001(Binary) = 1.1010100100 * 2^{-2}$$

$$1.1010100011 \times 2^4$$