Assignment 1

Lokesh Mohanty (SR no: 21014)

August 2022

1. Number of iterations = 150, in

```
float f = 1.0;
while (f != 0.0) f = f/2.0;
```

Proof: We know that as per IEEE 754 standard a real number(float) is stored in C as | s | e | f | where

$$\text{Normalized form} \rightarrow (-1)^s \times 1.f \times 2^{(e-127)}$$
$$\text{Denormalized form} \rightarrow (-1)^s \times 0.f \times 2^{-126}$$

1.0 is expressed as $(-1)^0 \times 1.0 \times 2^{127-127}$ i.e., `0x30000000`. Every time 1.0 is divided by 2, the value of e decreases by 1. Hence in normalized form, e takes values from 127 to 1 (i.e., the loop runs **126** times).

The next division by 2 takes it to denormalized form (i.e., $(-1)^0 \times 1.0 \times 2^{1-127}$ to $(-1)^0 \times 0.1 \times 2^{-126}$). After this for every division, the 1 in f shifts right by 1. Since f has a size of 23 bits, the lowest value will be twenty two 0's followed by a single 1 (i.e., the loop runs $\mathbf{1+22}$). After this the loop runs another time to finally make $f = 0.0$.

Hence, total number of the times the loop will iterate is $\boxed{126 + 1 + 22 + 1 = \mathbf{150}}$

2.

(a) The C shift operator << can be used to multiply a 32 bit unsigned int by powers of 2

Proof: Lets assume that out of the 32 bits, n bits are 1s and thier positions are $a_0, a_1 ... a_{n-1}$ i.e., the decimal form is $\sum_{i=0}^{n-1} 2^{a_i}$. When we left shift (<<) it, all the positions of the 1 bits moves up by 1 i.e., $\forall\, i$, $a'_i = a_i + 1$ where a'_i are the new positions and 0 is inserted at the LSB. Hence the decimal form becomes

$$\sum_{i=0}^{n-1} 2^{a'_i} = \sum_{i=0}^{n-1} 2^{a_i+1} = \sum_{i=0}^{n-1} (2^{a_i} \times 2)$$
$$= \boxed{2 \sum_{i=0}^{n-1} 2^{a_i}}$$

i.e., the new number is 2 times the previous number. And similarly repeating this k times multiplies the number by 2^k.

Hence the shift operator (<<) can be used to multiply a 32 bit unsigned int by powers of 2.

But when a_{n-1} is 31, shifting it to left removes it instead of increasing it as it goes out of storage. Hence this is valid only when the resultant number is within 0 and $2^{32} - 1$ (i.e., the range of an unsigned int).

(b) The C shift operator >> can be used to divide a 32 bit unsigned int by powers of 2

Proof: Lets assume that out of the 32 bits, n bits are 1s and thier positions are $a_0, a_1 ... a_{n-1}$ i.e., the decimal form is $\sum_{i=0}^{n-1} 2^{a_i}$. When we right shift (>>) it, all the positions of the 1 bits moves down by 1 i.e., $\forall\, i$, $a'_i = a_i - 1$ where a'_i are the new positions and 0 is inserted at the MSB. Hence

the decimal form becomes

$$\sum_{i=0}^{n-1} 2^{a'_i} = \sum_{i=0}^{n-1} 2^{a_i - 1} = \sum_{i=0}^{n-1} (2^{a_i}/2)$$
$$= \boxed{(\frac{1}{2}) \sum_{i=0}^{n-1} 2^{a_i}}$$

i.e., the new number is (1/2) times the previous number. And similarly repeating this k times divides the number by 2^k.

Hence the shift operator (>>) can be used to divide a 32 bit unsigned int by powers of 2.

When a_0 is 0, shifting it to right removes it instead of decreasing it as it goes out of storage. And this is also the desired action as the fraction part is removed in the int type in C.

(c) The C shift operator << can be used to multiply a 32 bit signed int by powers of 2

Proof: The storage method for positive signed int is same as that of unsigned int. Hence the left shift operator (<<) can be used to multiply 32 bit positive signed int by powers of 2. And just like the case of unsigned int, the resultant int should not exceed the range of a signed it (i.e., $2^{31} - 1$)

In case of negative signed int, lets assume that out of the 32 bits, n bits are 1s and thier positions are $a_0, a_1 ... a_{n-1}$ ($a_{n-1} = 31$ for negative signed int and $a_{n-2} = 30$ as multiplying by 2 shouldn't overflow it) i.e., the decimal form is (2's complement)

$$-2^{a_{n-1}} \sum_{i=0}^{n-2} 2^{a_i} = -2^{a_{n-1}} + 2^{a_{n-2}} + \sum_{i=0}^{n-3} 2^{a_i} = \boxed{-2^{a_{n-2}} + \sum_{i=0}^{n-3} 2^{a_i}}$$

When we left shift (<<) it, all the positions of the 1 bits move up by 1 i.e., $\forall\, i,\ a'_i = a_i + 1$ where a'_i are the new positions and 0 is inserted at the

LSB (i.e., a 1 is removed and the 1 bits are from $a'_0 \rightarrow a'_{n-2}$). Hence the decimal form becomes

$$\begin{aligned} -2^{a'_{n-2}} + \sum_{i=0}^{n-3} 2^{a'_i} &= -2^{a_{n-2}+1} + \sum_{i=0}^{n-3} 2^{a_i+1} \\ &= -2^{a_{n-2}}(2) + (2)\sum_{i=0}^{n-3} 2^{a_i} \\ &= \boxed{(2)(-2^{a_{n-2}} + \sum_{i=0}^{n-3} 2^{a_i})} \end{aligned}$$

i.e., the new number is (1/2) times the previous number. And similarly repeating this k times divides the number by 2^k.

Hence the shift operator (>>) can be used to divide a 32 bit negative signed int by powers of 2.

When a_{n-2} is not 30 (i.e., 31st bit is 0), shifting it to left makes the number positive instead of multiplying it as it goes out of storage. Hence this is also valid only when the resultant number is within the range of signed int.

(d) The C shift operator >> can be used to divide a 32 bit signed int by powers of 2

Proof: The storage method for positive signed int is same as that of unsigned int. Hence the right shift operator (>>) can be used to divide 32 bit positive signed int by powers of 2.

In case of negative signed int, lets assume that out of the 32 bits, n bits are 1s and thier positions are $a_0, a_1 ... a_{n-1}$ ($a_{n-1} = 31$ for negative signed int) i.e., the decimal form is $-2^{a_{n-1}} + \sum_{i=0}^{n-2} 2^{a_i}$ (2's complement).

When we right shift (>>) it, all the positions of the 1 bits move down by 1 i.e., $\forall\, i$, $a'_i = a_i - 1$ where a'_i are the new positions and 1 is inserted at the MSB (i.e., an extra 1 is added and the 1 bits are from $a'_0 \rightarrow a'_n$ where

$a'_n = a_{n-1}$) (arithmetic shift). Hence the decimal form becomes

$$\begin{aligned}
-2^{a'_n} + \sum_{i=0}^{n-1} 2^{a'_i} &= -2^{a'_n} + 2^{a'_{n-1}} + \sum_{i=0}^{n-2} 2^{a_i - 1} \\
&= -2^{a_{n-1}} + 2^{a_{n-1}-1} + \sum_{i=0}^{n-2} 2^{a_i - 1} \\
&= -2^{a_{n-1}-1} + \sum_{i=0}^{n-2} 2^{a_i - 1} \\
&= \boxed{\left(\frac{1}{2}\right)\left(-2^{a_{n-1}} + \sum_{i=0}^{n-2} 2^{a_i}\right)}
\end{aligned}$$

i.e., the new number is (1/2) times the previous number. And similarly repeating this k times divides the number by 2^k.

Hence the shift operator (>>) can be used to divide a 32 bit negative signed int by powers of 2.

When a_0 is 0, shifting it to right removes it instead of decreasing it as it goes out of storage. And this is also the desired action as the fraction part is removed in the int type in C. And since 1 is added at MSB, the number always remains negative as it should be.