

2_homework

2.55

Compile and run the sample code that uses `show_bytes` (file `show-bytes.c`) on different machines to which you have access. Determine the byte orderings used by these machines.

Solution: do it by yourself!

2.56

Try running the code for `show_bytes` for different sample values.

Solution: do it by yourself!

2.57

Write procedures `show_short`, `show_long`, and `show_double` that print the byte representations of C objects of types `short`, `long`, and `double`, respectively. Try these out on several machines.

Solution:

```
1 // hw2_57.c
2 #include<stdio.h>
3
4 typedef unsigned char* byte_pointer;
5
6 void show_bytes(byte_pointer start, size_t len) {
7     int i;
8     for(i = 0; i < len; i++) printf(" %.2x", start[i]);
9     printf("\n");
10 }
11
12 void show_int(int x) {
13     show_bytes((byte_pointer)&x, sizeof(int));
14 }
15
16 void show_float(float x) {
17     show_bytes((byte_pointer)&x, sizeof(float));
18 }
19
20 void show_pointer(void *x) {
21     show_bytes((byte_pointer)&x, sizeof(void *));
22 }
23
24 void show_short(short x) {
25     show_bytes((byte_pointer)&x, sizeof(short));
26 }
27
28 void show_long(long x) {
29     show_bytes((byte_pointer)&x, sizeof(long));
30 }
31
32 void show_double(double x) {
33     show_bytes((byte_pointer)&x, sizeof(double));
34 }
35
36 int main()
37 {
38     short s = 100;
39     long l = 100;
40     double d = 100.0;
41
42     printf("Show short of %d:", s);
43     show_short(x);
44     printf("Show long of %ld:", l);
45     show_short(x);
```

```

46         printf("Show double of %lf:",d);
47         show_short(x);
48         return 0;
49     }

```

2.58

Write a procedure `is_little_endian` that will return 1 when compiled and run on a little-endian machine, and will return 0 when compiled and run on a big-endian machine. This program should run on any machine, regardless of its word size.

Solution:

```

1  // hw2_58.c
2  #include<stdio.h>
3      /* 指针法检测 */
4      int checkLitEndMachByPointer()
5      {
6          int a = 1;
7          char* p = (char*)&a;
8
9          if(1 == *p) return 1;
10         return 0;
11     }
12
13     /* 联合体法检测 */
14     int checkLitEndMachByUnion()
15     {
16         union{
17             char c;
18             int i;
19         }u;
20
21         u.i = 1;
22
23         if(1 == u.c) return 1;
24         return 0;
25     }
26
27     int main()
28     {
29         if(checkLitEndMachByPointer())
30         {
31             printf("Using pointer check little endian, this machine is little endian
32             type.\n");
33         }
34         else
35         {
36             printf("Using pointer check little endian, this machine is big endian
37             type.\n");
38         }
39
40         if(checkLitEndMachByUnion())
41         {
42             printf("Using union check little endian, this machine is little endian
43             type.\n");
44         }
45         else
46         {
47             printf("Using union check little endian, this machine is big endian
48             type.\n");
49         }
50         return 0;
51     }

```

```

parallels@ubuntu-linux-22-04-desktop:/media/psf/csapp/2_4.assets$ ./hw2_58
Using pointer check little endian, this machine is little endian type.
Using union check little endian, this machine is little endian type.

```

2.59

Write a C expression that will yield a word consisting of the least significant byte of x and the remaining bytes of y . For operands $x = 0x89ABCDEF$ and $y = 0x76543210$, this would give $0x765432EF$.

Solution:

```
(x&0x000000FF) | (y&0xFFFFF00)
0x000000EF | 0x76543200 = 0x765432EF
```

2.60

Suppose we number the bytes in a w -bit word from 0 (least significant) to $w/8 - 1$ (most significant). Write code for the following C function, which will return an unsigned value in which byte i of argument x has been replaced by byte b :

```
1 unsigned replace_byte (unsigned x, int i, unsigned char b);
```

Here are some examples showing how the function should work:

`replace_byte(0x12345678, 2, 0xAB) --> 0x12AB5678`

`replace_byte(0x12345678, 0, 0xAB) --> 0x123456AB`

Solution:

```
1  #include<stdio.h>
2
3  unsigned replace_byte (unsigned x, int n, unsigned char b)
4  {
5      char* p = &x; // *p - LSB, *(p+1), ... *(p+3) - MSB
6      *(p+n)=b;
7      return x;
8  }
9
10 int main()
11 {
12     // replace_byte(0x12345678, 2, 0xAB) --> 0x12AB5678
13     printf("Replace the 3rd byte of 0x12345678 with 0xAB is:
14 0x%X\n", replace_byte(0x12345678, 2, 0xAB));
15     // replace_byte(0x12345678, 0, 0xAB) --> 0x123456AB
16     printf("Replace the 1st byte of 0x12345678 with 0xAB is:
17 0x%X\n", replace_byte(0x12345678, 0, 0xAB));
18     return 0;
19 }
```

```
parallels@ubuntu-linux-22-04-desktop:/media/psf/csapp/2_homework.assets$ ./hw2_60
Replace the 3rd byte of 0x12345678 with 0xAB is: 0x12AB5678
Replace the 1st byte of 0x12345678 with 0xAB is: 0x123456AB
```

2.61

Write C expressions that evaluate to 1 when the following conditions are true and to 0 when they are false. Assume x is of type `int`.

- 1 A. Any bit of x equals 1.
- 2 B. Any bit of x equals 0.
- 3 C. Any bit in the least significant byte of x equals 1.
- 4 D. Any bit in the least significant byte of x equals 0.

Your code should follow the bit-level integer coding rules (page 164), with the additional restriction that you may not use equality (`==`) or inequality (`!=`) tests.

Solution:

A. `!(~x)`

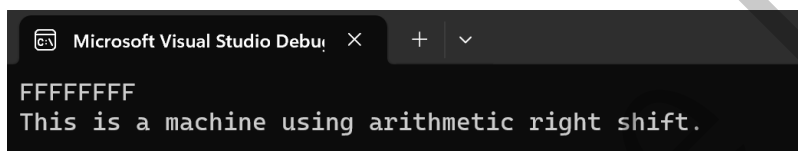
- B. !x
- C. !~(x&0xFF)
- D. !(x&0xFF)

2.62

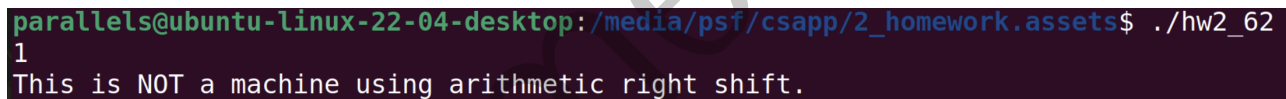
Write a function `int_shifts_are_arithmetic()` that yields 1 when run on a machine that uses arithmetic right shifts for data type `int` and yields 0 otherwise. Your code should work on a machine with any word size. Test your code on several machines.

Solution:

```
1  #include<stdio.h>
2
3  bool int_shifts_are_arithmetic()
4  {
5      return !~(0x80 >> 7); // if arithmetic, 0b 1000 000 >>7 will be 0b 1111 1111, otherwise,
                           // it will be 0b 0000 0001
6  }
7
8  int main()
9  {
10     if(int_shifts_are_arithmetic())
11         printf("This is a machine using arithmetic right shift.");
12     else
13         printf("This is NOT a machine using arithmetic right shift.");
14     return 0;
15 }
```



```
Microsoft Visual Studio Debug Console
FFFFFFFF
This is a machine using arithmetic right shift.
```



```
parallels@ubuntu-linux-22-04-desktop:/media/psf/csapp/2_homework.assets$ ./hw2_62
1
This is NOT a machine using arithmetic right shift.
```

2.63

Fill in code for the following C functions. Function `srl` performs a logical right shift using an arithmetic right shift (given by value `xsra`), followed by other operations not including right shifts or division. Function `sra` performs an arithmetic right shift using a logical right shift (given by value `xsrl`), followed by other operations not including right shifts or division. You may use the computation `8*sizeof(int)` to determine `w`, the number of bits in data type `int`. The shift amount `k` can range from 0 to `w - 1`.

```
1  unsigned srl(unsigned x, int k) {
2      /* Perform shift arithmetically */
3      unsigned xsra = (int) x >> k;
4      .....
5  }
6
7  int sra(int x, int k) {
8      /* Perform shift logically */
9      int xsrl = (unsigned) x >> k;
10     .....
11 }
```

Solution:

```
1  #include <stdio.h>
2
3  /* Perform right shift logically */
```

```

4  unsigned srl(unsigned x, int k) {
5      unsigned xsra = (int)x >> k; // right shift arithmetically
6      unsigned w = sizeof(int) * 8; // bit width of integer
7      unsigned mask = ~(0xFFFFFFFF << (w-k)); // 0b 00...0 11...1
8      return xsra & mask;
9  }
10
11 /* Perform right shift arithmetically */
12 int sra(int x, int k) {
13     int xsrl = (unsigned)x >> k; // right shift logically
14     unsigned w = sizeof(int) * 8; // bit width of integer
15     unsigned mask = 0xFFFFFFFF << (w - k); // 0b 11...1 00...0
16     unsigned m = 1 << (w-1); // get the sign bit
17     mask &= ! (x & m) - 1;
18     // if x > 0, x&m=0, !(x & m)-1=0, mask will be 0
19     // if x < 0, x&m will be non-zero, !(x&m) will be 0, !(x & m)-1 will be -1 which makes mask
keep same value.
20     return xsrl | mask;
21 }
22
23
24 int main()
25 {
26     unsigned test1 = 0x80000000;
27     printf("0x80000000 right shift logically for 3 bits is: 0x %X\n", srl(test1, 3));
28     printf("0x80000000 right shift logically for 8 bits is: 0x %X\n", srl(test1, 8));
29     printf("0x80000000 right shift logically for 16 bits is: 0x %X\n", srl(test1, 16));
30     printf("\n");
31     printf("0x80000000 right shift arithmetically for 3 bits is: 0x %X\n", sra(test1, 3));
32     printf("0x80000000 right shift arithmetically for 8 bits is: 0x %X\n", sra(test1, 8));
33     printf("0x80000000 right shift arithmetically for 16 bits is: 0x %X\n", sra(test1, 16));
34     printf("\n");
35     test1 = 0x70000000;
36     printf("0x70000000 right shift arithmetically for 3 bits is: 0x %08X\n", sra(test1, 3));
37     return 0;
38 }
39

```

2.64

Write code to implement the following function:

```

1  /* Return 1 when any odd bit of x equals 1; 0 otherwise.
2     Assume w=32 */
3
4  int any_odd_one(unsigned x);

```

Your function should follow the bit-level integer coding rules (page 164), except that you may assume that data type `int` has `w = 32` bits.

Solution:

```

1  #include <stdio.h>
2
3  /* Return 1 when any odd bit of x equals 1; 0 otherwise. Assume w=32 */
4  int any_odd_one(unsigned x)
5  {
6      // 0b 1010 1010 1010 1010 1010 1010 1010 1010
7      unsigned mask = 0xAAAAAAAA;
8      return (mask & x)&&1;
9  }
10
11 int main()
12 {
13     printf("Function any_odd_one returns %d with input of 0xA.\n", any_odd_one(0xA));
14     printf("Function any_odd_one returns %d with input of 0xB.\n", any_odd_one(0xB));
15     printf("Function any_odd_one returns %d with input of 0x4.\n", any_odd_one(0x4));
16     return 0;
17 }

```

2.65

Write code to implement the following function:

```
1  /* Return 1 when x contains an odd number of 1s; 0 otherwise.
2     Assume w=32 */
3
4  int odd_ones(unsigned x);
```

Your function should follow the bit-level integer coding rules (page 164), except that you may assume that data type `int` has `w = 32` bits.

Your code should contain a total of at most 12 arithmetic, bitwise, and logical operations.

Solution:

To find a rule:

unsigned x (w=2)	x>>1	x^x>>1	odd number of 1s				
00	00	00	False				
01	00	01	True				
10	01	11	True				
11	01	10	False				
unsigned x (w=4)	x>>1	x=x^x>>1	x>>2	x=x^x>>2	odd number of 1s		
0000	0000	0000	0000	0000	False		
0001	0000	0001	0000	0001^0000=0001	True		
0010	0001	0011	0000	0011^0000=0011	True		
0011	0001	0010	0000	0010^0000=0010	False		
0100	0010	0110	0001	0110^0001=0111	True		
0101	0010	0111	0001	0110	False		
0110	0011	0101	0001	0100	False		
0111	0011	0100	0001	0101	True		
1000	0100	1100	0011	1111	True		
1001	0100	1101	0011	1110	False		
1010	0101	1111	0011	1100	False		
1011	0101	1110	0011	1101	True		
1100	0110	1010	0010	1000	False		
1101	0110	1011	0010	1001	True		
1110	0111	1001	0010	1011	True		
	0111	1000	0010	1010	False		
ux (w=8)	x>>1	x=x^x>>1	x>>2	x=x^x>>2	x>>4	x=x^x>>4	odd 1s?
0000 0000	0000 0000	0000 0000	0000 0000	0000 0000	0000 0000	0000 0000	False
0000 0001	0000 0000	0000 0001	0000 0000	0000 0001	0000 0000	0000 0001	True
...
1111 1110	0111 1111	1000 0001	0010 0000	1010 0001	0000 1010	1010 1011	True
1111 1111	0111 1111	1000 0000	0010 0000	1010 0000	0000 1010	1010 1010	False

w=16, >>1, >>2, >>4, >>8

w=32, >>1, >>2, >>4, >>8, >>16

To summarize and deduce logically:

1. Bit 0 after executing the code below shows if bit 0 and bit 1 contain odd number of 1s.

```
1 x ^= x >> 1;
```

2. Bit i after executing the code below shows if bit i and bit $i+1$ contain odd number of 1s.

```
1 x ^= x >> 1;
```

3. Bit 0 after executing the code below shows if x ($w=4$) contains odd number of 1s.

```
1 x ^= x >> 1;
2 x ^= x >> 2;
```

4. Bit i of x after executing the code below shows if bit i , bit $i+1$, bit $i+2$, bit $i+3$ contains odd number of 1s.

```
1 x ^= x >> 1;
2 x ^= x >> 2;
```

5. Bit i of x after executing the code below shows if bit i , bit $i+1$, ..., bit $i+7$ contains odd number of 1s.

```
1 x ^= x >> 1;
2 x ^= x >> 2;
3 x ^= x >> 4;
```

6. Bit i of x after executing the code below shows if bit i , bit $i+1$, ..., bit $i+31$ contains odd number of 1s.

```
1 x ^= x >> 1;
2 x ^= x >> 2;
3 x ^= x >> 4;
4 x ^= x >> 8;
5 x ^= x >> 16;
```

7. Bit 0 of x ($w=32$) after executing the code above shows if bit 0, bit 1, ..., bit 31 contains odd number of 1s, finally.

8. According to the result above, we write the code below:

```
1 // hw2_65.c
2 #include <stdio.h>
3
4 /* Return 1 when x contains an odd number of 1s; 0 otherwise. Assume w=32 */
5 int odd_ones(unsigned x)
6 {
7     x ^= x >> 1;
8     // odd number of 1s for i and i+1 bit of x
9
10    x ^= x >> 2;
11    // odd number of 1s for i and i+2 bit of x
12    // Combined above, odd number of 1s for i,i+1,i+2,i+3 bit of x
13
14    x ^= x >> 4;
15    // odd number of 1s for i and i+4 bit of x
16    // Combined above, odd number of 1s for i,i+1,i+2,i+3,i+4,i+5,i+6,i+7 bit of x
17
18    x ^= x >> 8;
19    // odd number of 1s for i and i+8 bit of x
```

```

20         // Combined above, odd number of 1s for i,i+1,...,i+15 bit of x
21
22         x ^= x >> 16;
23         // odd number of 1s for i and i+16 bit of x
24         // Combined above, odd number of 1s for i,i+1,...,i+31 bit of x
25
26         return x&0x1;
27     }
28
29     int main()
30     {
31         // 0x A = 0b 1010
32         printf("Function odd_ones returns %d with input of 0xA.\n", odd_ones(0xA));
33         // 0x B = 0b 1011
34         printf("Function odd_ones returns %d with input of 0xB.\n", odd_ones(0xB));
35         // 0x 4 = 0b 0100
36         printf("Function odd_ones returns %d with input of 0x4.\n", odd_ones(0x4));
37         return 0;
38     }

```

2.66

Write code to implement the following function:

```

1  /*
2
3  * Generate mask indicating leftmost 1 in x. Assume w=32.
4
5  * For example, 0xFF00 -> 0x8000, and 0x6600 --> 0x4000.
6
7  * If x = 0, then return 0.
8
9  */
10
11 int leftmost_one(unsigned x);

```

Your function should follow the bit-level integer coding rules (page 164), except that you may assume that data type `int` has `w = 32` bits.

Your code should contain a total of at most 15 arithmetic, bitwise, and logical operations.

Hint: First transform `x` into a bit vector of the form `[0 ... 011 ... 1]`

`[000111] = [000111]>>1 + 1 = [000100]`

`[01111111] = [01111111]>>1 + 1 = [01000000]`

Solution:

According to the hint, what we want is to transform `x` into a bit vector of the form `[0 ... 011 ... 1]`, which means all 0s on the left of leftmost 1 but all 1s on the right of leftmost 1.

We will see `'|'` operation to make sure replicate the leftmost 1 to all bits on the right of leftmost 1.

```

1  // hw2_66.c
2  #include <stdio.h>
3
4  /*
5  * Generate mask indicating leftmost 1 in x. Assume w=32.
6  * For example, 0xFF00 -> 0x8000, and 0x6600 --> 0x4000.
7  * If x = 0, then return 0.
8  */
9  int leftmost_one(unsigned x) {
10     x |= x >> 16; // To make sure all right bits after leftmost 1 are 1s
11     x |= x >> 8;
12     x |= x >> 4;
13     x |= x >> 2;
14     x |= x >> 1;
15     return (x >> 1) + 0x1;
16 }

```



```

17
18  int main()
19  {
20      // 0x A = 0b 1010
21      printf("Function leftmost_one returns 0x %x with input of 0xA.\n", leftmost_one(0xA));
22      // 0x B = 0b 1011
23      printf("Function leftmost_one returns 0x %x with input of 0xB.\n", leftmost_one(0xB));
24      // 0x 4 = 0b 0100
25      printf("Function leftmost_one returns 0x %x with input of 0x4.\n", leftmost_one(0x4));
26      return 0;
27  }

```

2.67

You are given the task of writing a procedure `int_size_is_32()` that yields 1 when run on a machine for which an `int` is 32 bits, and yields 0 otherwise. You are not allowed to use the `sizeof` operator. Here is a first attempt:

```

1  /* The following code does not run properly on some machines */
2
3  int bad_int_size_is_32() {
4      /* Set most significant bit (msb) of 32-bit machine */
5      int set_msb = 1 << 31;
6      /* Shift past msb of 32-bit word */
7      int beyond_msb = 1 << 32;
8      /* set_msb is nonzero when word size >= 32
9       * beyond_msb is zero when word size <= 32 */
10     return set_msb && !beyond_msb;
11 }

```

When compiled and run on a 32-bit SUN SPARC, however, this procedure returns 0. The following compiler message gives us an indication of the problem:

```

1      warning: left shift count >= width of type

```

1. In what way does our code fail to comply with the C standard?
2. Modify the code to run properly on any machine for which data type `int` is at least 32 bits.
3. Modify the code to run properly on any machine for which data type `int` is at least 16 bits.

Solution:

1. When the shift count is greater or equal to the width of type, the behavior is undefined.

```

1  #include<stdio.h>
2
3  int bad_int_size_is_32() {
4      /* Set most significant bit (msb) of 32-bit machine */
5      int set_msb = 1 << 31;
6      /* Shift past msb of 32-bit word */
7      int beyond_msb = 1 << 32;
8      /* set_msb is nonzero when word size >= 32
9       * beyond_msb is zero when word size <= 32 */
10     return set_msb && !beyond_msb;
11 }
12
13 int main()
14 {
15     if(bad_int_size_is_32())
16     {
17         printf("An int is 32 bits on this machine");
18     }
19     else
20     {

```

```

21         printf("An int is NOT 32 bits on this machine");
22     }
23     return 0;
24 }

```

2. code

```

1  #include<stdio.h>
2
3  int int_size_is_32()
4  {
5      int set_msb = 1 << 31;
6      int beyond_msb = set_msb << 1;
7      return set_msb && !beyond_msb;
8  }
9
10 int main()
11 {
12     if(int_size_is_32())
13     {
14         printf("An int is 32 bits on this machine\n");
15     }
16     else
17     {
18         printf("An int is NOT 32 bits on this machine\n");
19     }
20     return 0;
21 }

```

3. code

```

1  int int_size_is_32_for_16bit()
2  {
3      int set_msb = 1 << 15 << 15 << 1;
4      int beyond_msb = set_msb << 1;
5      return set_msb && !beyond_msb;
6  }

```

2.68

Write code for a function with the following prototype:

```

1  /*
2   * Mask with least significant n bits set to 1
3   * Examples: n = 6 --> 0x3F, n = 17 --> 0xFFFF
4   * Assume 1 <= n <= w
5   */
6
7  int lower_one_mask(int n);

```

Your function should follow the bit-level integer coding rules (page 164). Be careful of the case $n = w$.

Solution:

```

1  #include<stdio.h>
2
3  int lower_one_mask(int n) {
4      int w = sizeof(int) << 3;
5      int mask = (unsigned)-1 >> (w - n);
6      return mask;
7  }
8
9  int main()

```

```

10 {
11     int n = 10;
12     printf("Mask with least significant %d bits set to 1 is 0x%X\n", n, lower_one_mask(n));
13     return 0;
14 }

```

2.69

Write code for a function with the following prototype:

```

1  /*
2   * Do rotating left shift. Assume 0 <= n < w
3   * Examples when x = 0x12345678 and w = 32:
4   *   n=4 -> 0x23456781, n=20 -> 0x67812345
5   */
6
7  unsigned rotate_left(unsigned x, int n);

```

Your function should follow the bit-level integer coding rules (page 164). Be careful of the case $n = 0$.

Solution:

```

1  #include <stdio.h>
2
3  /*
4   * Do rotating left shift. Assume 0 <= n < w
5   * Examples when x = 0x12345678 and w = 32:
6   *   n=4 -> 0x23456781, n=20 -> 0x67812345
7   */
8  unsigned rotate_left(unsigned x, int n)
9  {
10     int w = sizeof(int) << 3;
11
12     int low_bits = (unsigned)x >> (w-n-1) >> 1;
13     int high_bits = x << n;
14     return high_bits | low_bits;
15 }
16
17 int main()
18 {
19     int x = 0x12345678;
20     int n = 4;
21     printf("When x = 0x%X, it will be 0x%X after rotating left shift by %d.\n", x,
22 rotate_left(x, n), n);
23
24     n = 0;
25     printf("When x = 0x%X, it will be 0x%X after rotating left shift by %d.\n", x,
26 rotate_left(x, n), n);
27
28     n = 20;
29     printf("When x = 0x%X, it will be 0x%X after rotating left shift by %d.\n", x,
30 rotate_left(x, n), n);
31     return 0;
32 }

```

2.70

Write code for the function with the following prototype:

```

1
2  /*
3   * Return 1 when x can be represented as an n-bit, 2's-complement
4   * number; 0 otherwise
5   * Assume 1 <= n <= w
6   */
7

```

```
8 int fits_bits(int x, int n);
```

Your function should follow the bit-level integer coding rules (page 164).

Solution:

```
1  #include<stdio.h>
2  /*
3   * Return 1 when x can be represented as an n-bit, 2's-complement
4   * number; 0 otherwise
5   * Assume 1 <= n <= w
6   */
7  int fits_bits(int x, int n) {
8      int test = x >> (n-1);
9      return (test == 0) || (test == -1); // 2 cases: x >= 0 or x < 0;
10 }
11
12 /*
13 int fits_bits(int x, int n){
14     int w = sizeof(int) << 3;
15     int shift_count = w - n;
16     return (x << shift_count >> shift_count) == x;
17 }*/
18
19 int main() {
20     int x = 0b0100;
21     int n = 3;
22     if(fits_bits(x,n)) printf("0x%X can be represented as an %d - bit, 2's - complement
23 number\n",x,n);
24     else printf("0x%X can NOT be represented as an %d - bit, 2's - complement number\n", x,
25 n);
26
27     n = 4;
28     if (fits_bits(x, n)) printf("0x%X can be represented as an %d - bit, 2's - complement
29 number\n", x, n);
30     else printf("0x%X can NOT be represented as an %d - bit, 2's - complement number\n", x,
31 n);
32
33     return 0;
34 }
```

2.71

You just started working for a company that is implementing a set of procedures to operate on a data structure where 4 signed bytes are packed into a 32-bit unsigned. Bytes within the word are numbered from 0 (least significant) to 3 (most significant). You have been assigned the task of implementing a function for a machine using two's-complement arithmetic and arithmetic right shifts with the following prototype:

```
1  /* Declaration of data type where 4 bytes are packed
2   into an unsigned */
3  typedef unsigned packed_t;
4
5  /* Extract byte from word. Return as signed integer */
6  int xbyte(packed_t word, int bytenum);
```

That is, the function will extract the designated byte and sign extend it to be a 32-bit int.

Your predecessor (who was fired for incompetence) wrote the following code:

```
1  /* Failed attempt at xbyte */
2  int xbyte(packed_t word, int bytenum)
3  {
4      return (word >> (bytenum << 3)) & 0xFF;
5  }
```

A. What is wrong with this code?

B. Give a correct implementation of the function that uses only left and right shifts, along with one subtraction.

Solution:

A. Failed at negative number, because packed_it is unsigned int which means only logical right shifts happen. Thus the function cannot return as signed integer.

B.

```
1  #include<stdio.h>
2
3  /* Declaration of data type where 4 bytes are packed
4     into an unsigned */
5  typedef unsigned packed_t;
6
7  /* Failed attempt at xbyte */
8  int xbyte(packed_t word, int bytenum)
9  {
10     return (word >> (bytenum << 3)) & 0xFF;
11 }
12
13 /* Correct xbyte*/
14 int correct_xbyte(packed_t word, int bytenum) // suppose word = 0xFFF80000 bytenum = 2
15 {
16     int w = sizeof(int) << 3; // w = 32
17     int temp_shift = word << (w - ((bytenum + 1) << 3)); // temp_shift = 0xF8000000
18     int result = temp_shift >> (w - 1 << 3); // result = 0xFFFFFFFF8
19     return result;
20 }
21
22 int main()
23 {
24     packed_t packedword = -10;
25     int bytenum = 0;
26     printf("xbytes: extract bytenumber%d from word 0x%08X. Return 0x%08X as signed
27 integer.\n",bytenum,(int)packedword, xbyte(packedword,bytenum));
28     printf("correct_xbyte: extract bytenumber%d from word 0x%08X. Return 0x%08X as signed
29 integer.\n", bytenum, (int)packedword, correct_xbyte(packedword, bytenum));
30     packedword = 0xFFF80000;
31     bytenum = 2;
32     printf("correct_xbyte: extract bytenumber%d from word 0x%08X. Return 0x%08X as signed
33 integer.\n", bytenum, (int)packedword, correct_xbyte(packedword, bytenum));
34     return 0;
35 }
```

2.72

You are given the task of writing a function that will copy an integer val into a buffer buf, but it should do so only if enough space is available in the buffer.

Here is the code you write:

```
1  /* Copy integer into buffer if space is available */
2  /* WARNING: The following code is buggy */
3  void copy_int(int val, void *buf, int maxbytes) {
4
5     if (maxbytes-sizeof(val) >= 0)
6         memcpy(buf, (void *) &val, sizeof(val));
7
8 }
```

This code makes use of the library function memcpy. Although its use is a bit artificial here, where we simply want to copy an int, it illustrates an approach commonly used to copy larger data structures.

You carefully test the code and discover that it always copies the value to the buffer, even when maxbytes is too small.

A. Explain why the conditional test in the code always succeeds. Hint: The sizeof operator returns a value of type size_t.

B. Show how you can rewrite the conditional test to make it work properly.

Solution:

A. sizeof() always return unsigned, which make maxbytes-sizeof(val) is always greater or equal to 0.

B.

```
1         if (maxbytes >= (int)sizeof(val))
```

2.73

Write code for a function with the following prototype:

```
1  /* Addition that saturates to TMin or TMax */
2  int saturating_add(int x, int y);
```

Instead of overflowing the way normal two's-complement addition does, saturating addition returns TMax when there would be positive overflow, and TMin when there would be negative overflow. Saturating arithmetic is commonly used in programs that perform digital signal processing.

Your function should follow the bit-level integer coding rules (page 164).

Solution:

```
1  #include<stdio.h>
2  #include<limits.h>
3
4  /* Addition that saturates to TMin or TMax */
5  int saturating_add(int x, int y)
6  {
7      /*
8       * if x > 0, y > 0 but sum < 0, it's a positive overflow
9       * if x < 0, y < 0 but sum >= 0, it's a negative overflow
10      */
11      int sum = x + y;
12      int pos_flag = !(INT_MIN & x) && !(INT_MIN & y) && (INT_MIN & sum); // INT_MIN = 0x 8000
13      0000 int neg_flag = (INT_MIN & x) && (INT_MIN & y) && !(INT_MIN & sum);
14
15      pos_flag && (sum = INT_MAX);
16      neg_flag && (sum = INT_MIN);
17
18      return sum;
19  }
20
21  int main()
22  {
23      int x = 5;
24      int y = 5;
25      printf("0x%x saturating_add 0x%x equals 0x%x.\n", x, y, saturating_add(x, y));
26      x = INT_MAX;
27      printf("0x%x saturating_add 0x%x equals 0x%x.\n", x, y, saturating_add(x, y));
28      x = -5;
29      y = INT_MIN;
30      printf("0x%x saturating_add 0x%x equals 0x%x.\n", x, y, saturating_add(x, y));
31      return 0;
32  }
```

2.74

Write a function with the following prototype:

```

1  /* Determine whether arguments can be subtracted without overflow */
2  int tsub_ok(int x, int y);

```

This function should return 1 if the computation $x - y$ does not overflow.

Solution:

```

1  #include <stdio.h>
2  #include <limits.h>
3
4  /* Determine whether arguments can be subtracted without overflow */
5  int tsub_ok(int x, int y)
6  {
7      /*
8       * if x > 0, y < 0 but res <= 0, it's a positive overflow
9       * if x < 0, y > 0 but res >= 0, it's a negative overflow
10      */
11      int res = x - y;
12      int pos_flag = !(INT_MIN & x) && (INT_MIN & y) && ((INT_MIN & res) || (res == 0)); // INT_MIN =
0x 8000 0000
13      int neg_flag = (INT_MIN & x) && !(INT_MIN & y) && !(INT_MIN & res) || (res == 0);
14      return !(pos_flag || neg_flag);
15  }
16
17  int main()
18  {
19      int x = 5;
20      int y = 6;
21      printf("tsub_ok() return %d when check %d - %d\n", tsub_ok(x, y), x, y);
22
23      x = INT_MIN;
24      printf("tsub_ok() return %d when check %X - %X\n", tsub_ok(x, y), x, y);
25
26      x = INT_MAX;
27      y = -5;
28      printf("tsub_ok() return %d when check %X - %X\n", tsub_ok(x, y), x, y);
29      return 0;
30  }

```

2.75

Suppose we want to compute the complete $2w$ -bit representation of $x \cdot y$, where both x and y are unsigned, on a machine for which data type unsigned is w bits. The low-order w bits of the product can be computed with the expression $x * y$, so we only require a procedure with prototype

```

1  unsigned unsigned_high_prod(unsigned x, unsigned y);

```

that computes the high-order w bits of $x \cdot y$ for unsigned variables. We have access to a library function with prototype

```

1  int signed_high_prod(int x, int y);

```

that computes the high-order w bits of $x \cdot y$ for the case where x and y are in two's-complement form. Write code calling this procedure to implement the function for unsigned arguments. Justify the correctness of your solution.

Hint: Look at the relationship between the signed product $x \cdot y$ and the unsigned product $x' \cdot y'$ in the derivation of Equation 2.18.

Solution:

We know the original equation 2.18:

$$x' *^u_w y' = (x' \cdot y') \bmod 2^w$$

```
1  $=((x+x_{w-1}2^w) \cdot (y+y_{w-1}2^w))\bmod 2^w$
```

Because $w = 32$, and we can have 64 bit width, we don't need $\bmod 2^w$

$$x' \cdot y' = (x + x_{w-1}2^w) \cdot (y + y_{w-1}2^w) = x \cdot y + x \cdot y_{w-1}2^w + y \cdot x_{w-1}2^w + y_{w-1}2^w \cdot x_{w-1}2^w$$

Our goal is to get the high 32 bits of unsigned multiplication, so we right shift the equation above by 32 bits .

$$(x' \cdot y') \gg 32 = (x \cdot y) \gg 32 + x \cdot y_{w-1} + y \cdot x_{w-1} + y_{w-1} \cdot x_{w-1}2^w$$

$$= \text{signed_high_prod}(x, y) + x \cdot y_{w-1} + y \cdot x_{w-1} + y_{w-1} \cdot x_{w-1} \cdot 2^w$$

The last $y_{w-1} \cdot x_{w-1} \cdot 2^{32}$ can be discarded because the bit width is 32 bits.

Therefore, we get the code below:

```
1
2  #include <stdio.h>
3  #include <inttypes.h>
4
5  int signed_high_prod(int x, int y)
6  {
7      int64_t res = (int64_t) x * y;
8      res = res >> 32;
9      return res;
10 }
11
12 unsigned unsigned_high_prod(unsigned x, unsigned y)
13 {
14     // $=signed\_high\_prod(x,y) + x\cdot y_{w-1}+y\cdot x_{w-1}+y_{w-1}\cdot x_{w-1}2^w$
15     unsigned y31 = y >> 31;
16     unsigned x31 = x >> 31;
17     return signed_high_prod(x, y) + x * y31 + y * x31;
18 }
19
20 int main()
21 {
22     unsigned x = 0xFFFFFFFF;
23     unsigned y = 0xA;
24     printf("unsigned_high_prod returns 0x%X with 0x%X + 0x%X\n", unsigned_high_prod(x,y),x,y);
25
26     return 0;
27 }
```

2.76

The library function `calloc` has the following declaration:

```
1  void *calloc(size_t nmemb, size_t size);
```

According to the library documentation, "The `calloc` function allocates memory for an array of `nmemb` elements of size bytes each. The memory is set to zero. If `nmemb` or `size` is zero, then `calloc` returns `NULL`."

Write an implementation of `calloc` that performs the allocation by a call to `malloc` and sets the memory to zero via `memset`. Your code should not have any vulnerabilities due to arithmetic overflow, and it should work correctly regardless of the number of bits used to represent data of type `size_t`.

As a reference, functions `malloc` and `memset` have the following declarations:

```
1  void *malloc(size_t size);
2  void *memset(void *s, int c, size_t n);
```

Solution:

Overflow check, we may refer to Practice Problem 2.35:


```

1  /* Determine whether arguments can be multiplied without overflow */
2  int tmult_ok(int x, int y) {
3
4      int p = x*y;
5      /* Either x is zero, or dividing p by x gives y */
6      return !x || p/x == y;
7
8  }

```

So we write the code below:

```

1  #include <stdio.h>
2  #include <limits.h>
3
4  void* calloc(size_t ele_count, size_t ele_size)
5  {
6      if (ele_count == 0 || ele_size == 0) return NULL;
7
8      size_t size = ele_count * ele_size;
9      if (!ele_count || size / ele_count != ele_size) return NULL; // overflow check
10
11     void* p = malloc(size);
12     if (p == NULL) return NULL;
13     memset(p, 0, size);
14     return p;
15 }
16
17 int main()
18 {
19     size_t ele_count = UINT_MAX;
20     size_t ele_size = 2;
21     printf("calloc(0x%X, 0x%X) returns 0x%p.\n", ele_count, ele_size, calloc(ele_count,
22     ele_size));
23
24     ele_count = 0;
25     ele_size = 2;
26     printf("calloc(0x%X, 0x%X) returns 0x%p.\n", ele_count, ele_size, calloc(ele_count,
27     ele_size));
28
29     ele_count = 5;
30     ele_size = 2;
31     printf("calloc(0x%X, 0x%X) returns 0x%p.\n", ele_count, ele_size, calloc(ele_count,
32     ele_size));
33     return 0;
34 }

```

2.77

Suppose we are given the task of generating code to multiply integer variable x by various different constant factors K . To be efficient, we want to use only the operations $+$, $-$, and \ll . For the following values of K , write C expressions to perform the multiplication using at most three operations per expression.

- A. $K = 17$
- B. $K = -7$
- C. $K = 60$
- D. $K = -112$

Solution:

$$x \times K = x \times 17 = x \times (2^4 + 2^0) = x \ll 4 + x$$

```

1  int K17(int x)
2  {
3      return x<<4+x;
4  }

```

$$x \times K = x \times -7 = x \times (2^0 - 2^3) = x - x \ll 3$$

Copyright 2024 Maxime Lionel. All rights reserved.
For any question, please contact ydzhang89@163.com

```

1  int Kneg7(int x)
2  {
3      return x-x<<3;
4  }

```

$x \times K = x \times 60 = x \times (2^5 + 2^4 + 2^3 + 2^2) = x \ll 5 + x \ll 4 + x \ll 3 + x \ll 2$ - discarded because of more than 3 operations.

$x \times K = x \times 60 = x \times (2^6 - 2^2) = x \ll 6 - x \ll 2$

```

1  int K60(int x)
2  {
3      return x<<6-x<<2;
4  }

```

$x \times K = x \times -112 = x \times (2^4 - 2^7) = x \ll 4 - x \ll 7$

```

1  int Kneg112(int x)
2  {
3      return x<<4-x<<7;
4  }

```

2.78

Write code for a function with the following prototype:

```

1  /* Divide by power of 2. Assume 0 <= k < w-1 */
2  int divide_power2(int x, int k);

```

The function should compute $x/2^k$ with correct rounding, and it should follow the bit-level integer coding rules (page 164).

Solution:

We may refer to the chapter of **Two's-complement division by a power of 2**, thus get the formula below:

$x/2^k = (x < 0 ? x + (1 \ll k) - 1 : x) \gg k$

Then we get the code below:

```

1  #include <limits.h>
2  #include <stdio.h>
3
4
5  /* Divide by power of 2. Assume 0 <= k < w-1 */
6  int divide_power2(int x, int k)
7  {
8      // $x/2^k=(x<0? x+(1<<k)-1 : x)>>k$
9      int sign_bit = ((x & INT_MIN) == INT_MIN);
10     sign_bit && (x = x + (1 << k) - 1);
11     return x >> k;
12 }
13
14 int main()
15 {
16     int x = 8;
17     int k = 3;
18     printf("divide_power2(0x%X, 0x%X) returns 0x%X.\n", x, k, divide_power2(x, k));
19     x = -17;
20     printf("divide_power2(0x%X, 0x%X) returns 0x%X.\n", x, k, divide_power2(x, k));
21     return 0;
22 }

```

2.79

Write code for a function `mul3div4` that, for integer argument `x`, computes $3 * x / 4$ but follows the bit-level integer coding rules (page 164). Your code should replicate the fact that the computation $3*x$ can cause overflow.

Solution:

$$x \times 3/4 = x \times (2^1 + 2^0)/2^2 = (x \ll 1 + x) \gg 2$$

We can use the `divide_power2` function from 2.78.

We get the code below:

```
1  #include <limits.h>
2  #include <stdio.h>
3  #include <assert.h>
4
5  /* Divide by power of 2. Assume 0 <= k < w-1 */
6  int divide_power2(int x, int k)
7  {
8      // $x/2^k=(x<0? x+(1<<k)-1 : x)>>k$
9      int sign_bit = ((x & INT_MIN) == INT_MIN);
10     sign_bit && (x = x + (1 << k) - 1);
11     return x >> k;
12 }
13
14 int mul3div4(int x)
15 {
16     // x*3/4 = x * (2^1 + 2^0) / 2^2
17     int res = (x << 1) + x;
18     return divide_power2(res, 2);
19 }
20
21 int main()
22 {
23     int x = 10;
24     assert(mul3div4(x)==3*x/4);
25     x = INT_MAX;
26     assert(mul3div4(x)==3*x/4);
27     x = INT_MIN;
28     assert(mul3div4(x)==3*x/4);
29     return 0;
30 }
```

2.80

Write code for a function `threefourths` that, for integer argument `x`, computes the value of $\frac{3}{4}x$, rounded toward zero. It should not overflow. Your function should follow the bit-level integer coding rules (page 164).

Solution:

$$x/2^k = (x < 0 ? x + (1 \ll k) - 1 : x) \gg k$$

To calculate $\frac{3}{4}x$ and avoid overflow, we may choose to divide by 4 first then multiply by 3. But there's one issue to solve that if we divide by 4 first, means we will discard the last 2 bits directly, we may lose the precision, because there's a multiplication of 3 after. Therefore, we write the code below:

```
1  #include <stdio.h>
2  #include <limits.h>
3  #include <assert.h>
4
5  int threefourths(int x)
6  {
7      // First, we divide x into high 30 bits and low 2 bits.
8      int mask2bits = 0x3;
9      int high30 = ~mask2bits & x;
10     int low2 = mask2bits & x;
11
12     // For high 30 bits, we may divide by 4 first
13     high30 = high30 >> 2;
14     int high30_res = (high30 << 1) + high30;
```

```

15
16
17     // For low 2 bits, we may multiply 3 first then divide.
18     low2 = (low2 << 1) + low2;
19     int sign_bit = (x & INT_MIN) == INT_MIN;
20     int bias = (1 << 2) - 1;
21     (sign_bit == 1) && (low2 += bias);
22     int low2_res = low2 >> 2;
23
24     return high30_res + low2_res;
25 }
26
27 int main()
28 {
29     int x = 8;
30     printf("threefourths(0x%X) returns 0x%X.\n", x, threefourths(x));
31     x = 9;
32     printf("threefourths(0x%X) returns 0x%X.\n", x, threefourths(x));
33     x = 10;
34     printf("threefourths(0x%X) returns 0x%X.\n", x, threefourths(x));
35     x = -8;
36     printf("threefourths(0x%X) returns 0x%X.\n", x, threefourths(x));
37     x = -9;
38     printf("threefourths(0x%X) returns 0x%X.\n", x, threefourths(x));
39     x = -10;
40     printf("threefourths(0x%X) returns 0x%X.\n", x, threefourths(x));
41     return 0;
42 }

```

2.81

Write C expressions to generate the bit patterns that follow, where a^k represents k repetitions of symbol a . Assume a w -bit data type. Your code may contain references to parameters j and k , representing the values of j and k , but not a parameter representing w .

A. $1^{w-k}0^k$

B. $0^{w-k-j}1^k0^j$

- $A(k) = 1^{w-k}0^k$
- $\sim A(k) = 0^{w-k}1^k$
- $(\sim A(k)) << j = 0^{w-k-j}1^k0^j$

Solution:

```

1  #include <stdio.h>
2
3  // $1^{w-k}0^k$
4  int A(int k)
5  {
6      int sample = -1;
7      return sample << k;
8  }
9
10 // $0^{w-k-j}1^k0^j$
11 int B(int k, int j)
12 {
13     return (~A(k)) << j;
14 }
15
16 int main()
17 {
18     int k = 16;
19     printf("A(%d) returns 0x%08X.\n", k, A(16));
20     int j = 8;
21     printf("B(%d,%d) returns 0x%08X.\n", k,j, B(16,8));
22     return 0;
23 }

```

2.82

We are running programs where values of type `int` are 32 bits. They are represented in two's complement, and they are right shifted arithmetically. Values of type `unsigned` are also 32 bits.

We generate arbitrary values `x` and `y`, and convert them to unsigned values as follows:

```
1  /* Create some arbitrary values */
2  int x = random();
3  int y = random();
4  /* Convert to unsigned */
5  unsigned ux = (unsigned) x;
6  unsigned uy = (unsigned) y;
```

For each of the following C expressions, you are to indicate whether or not the expression always yields 1. If it always yields 1, describe the underlying mathematical principles. Otherwise, give an example of arguments that make it yield 0.

- A. $(x < y) == (-x > -y)$
- B. $((x + y) << 4) + y - x == 17 * y + 15 * x$
- C. $\sim x + \sim y + 1 == \sim (x + y)$
- D. $(ux - uy) == -(\text{unsigned})(y - x)$
- E. $((x >> 2) << 2) <= x$

Solution:

A. $(x < y) == (-x > -y)$

False.

When `x = INT_MIN`, `y = 0`,

`-x` will still be `INT_MIN`.

B. $((x + y) << 4) + y - x == 17 * y + 15 * x$

True.

C. $\sim x + \sim y + 1 == \sim (x + y)$

$\sim x = \sim x + 1$, $\sim y = \sim y + 1$

$\sim x + \sim y + 1 = -x - 1 + -y - 1 + 1 = -(x + y) - 1 = \sim (x + y) + 1 - 1$

True.

D. $(ux - uy) == -(\text{unsigned})(y - x)$

True.

E. $((x >> 2) << 2) <= x$

True.

Code:

```
1  #include <limits.h>
2  #include <stdio.h>
3  #include <assert.h>
4  #include <stdlib.h>
5  #include <time.h>
6
7  // A. (x < y) == (-x > -y)
8  int A(int x, int y)
9  {
10     return (x < y) == (-x > -y);
11 }
12
13 // B. ((x+y)<<4) + y-x == 17*y+15*x
14 int B(int x, int y)
15 {
16     return ((x + y) << 4) + y - x == 17 * y + 15 * x;
17 }
18
19 // C. ~x + ~y + 1 == ~(x + y)
20 int C(int x, int y)
21 {
22     return ~x + ~y + 1 == ~(x + y);
23 }
24
```

```

25 // D. $(ux-uy) == -(unsigned)(y-x)$
26 int D(unsigned ux, unsigned uy, int x, int y)
27 {
28     return (ux - uy) == -(unsigned)(y - x);
29 }
30
31 // E. $((x>>2)<<2)<=x$
32 int E(int x, int y)
33 {
34     return ((x >> 2) << 2) <= x;
35 }
36
37 int main()
38 {
39     /* Create some arbitrary values */
40     srand((unsigned)time(0)); // set seed for rand(). rand() will use the seed to generate
    random number.
41                                     // time(0) - seconds since January 1, 1970
42     int x = rand();
43     int y = rand();
44     /* Convert to unsigned */
45     unsigned ux = (unsigned)x;
46     unsigned uy = (unsigned)y;
47
48     /*
49     A. $(x<y) == (-x>-y)$
50     B. $((x+y)<<4) + y-x == 17*y+15*x$
51     C. $~x+~y+1 == ~(x+y)$
52     D. $(ux-uy) == -(unsigned)(y-x)$
53     E. $((x>>2)<<2)<=x$
54     */
55     assert(!A(INT_MIN, 0));
56     assert(B(x, y));
57     assert(C(x, y));
58     assert(D(ux, uy, x, y));
59     assert(E(x, y));
60
61     return 0;
62 }

```

2.83

Consider numbers having a binary representation consisting of an infinite string of the form $0.yyyyyy \dots$, where y is a k -bit sequence. For example, the binary representation of $\frac{1}{3}$ is $0.01010101 \dots$ ($y = 01$), while the representation of $\frac{1}{5}$ is $0.001100110011 \dots$ ($y = 0011$).

- Let $Y = B2U_k(y)$, that is, the number having binary representation y . Give a formula in terms of Y and k for the value represented by the infinite string. Hint: Consider the effect of shifting the binary point k positions to the right.
- What is the numeric value of the string for the following values of y ?
 - 101
 - 0110
 - 010011

Solution:

- what is y ? - y is a k -bit sequence.
what is k ? - k is the width of y .
what is Y ? - Y is the number having binary representation y .
So:
 $0.yyy \dots yyy \ll k = y.yyy \dots yyy = Y + 0.yyy \dots yyy$
 $Y = 0.yyy \dots yyy \ll k - 0.yyy \dots yyy = 0.yyy \dots yyy(2^k - 1)$
 $0.yyy \dots yyy = Y / (2^k - 1)$
- We may use the equation above, so only need to find values of Y and k .
 - $Y = 101_2 = 5_{10}, k = 3, result = \frac{5}{7}$

- (b) $Y = 0110_2 = 6_{10}, k = 4, result = \frac{2}{5}$
 (c) $Y = 010011_2 = 18_{10}, k = 6, result = \frac{19}{31}$

2.84

Fill in the return value for the following procedure, which tests whether its first argument is less than or equal to its second. Assume the function `f2u` returns an unsigned 32-bit number having the same bit representation as its floating-point argument. You can assume that neither argument is NaN. The two flavors of zero, `+0` and `-0`, are considered equal.

```
1 // Test if x <= y
2 int float_le(float x, float y) {
3     unsigned ux = f2u(x);
4     unsigned uy = f2u(y);
5
6     /* Get the sign bits */
7     unsigned sx = ux >> 31;
8     unsigned sy = uy >> 31;
9
10    /* Give an expression using only ux, uy, sx, and sy */
11    return ;
12 }
```

Solution:

Condition 1: $x = +0.0$ or -0.0 , $y = +0.0$ or -0.0 , then $x == y$

We do not need sign bit, so just `<< 1`.

```
C x << 1 == 0 && y << 1 == 0
```

Condition 2: $x < 0$, $y > 0$

We need to get the sign bit.

```
C unsigned x_signbit = ux >> 31; unsigned y_signbit = uy >> 31; x_signbit && !y_signbit
```

Condition 3: $x > 0$, $y > 0$

```
C !x_signbit && !y_signbit && ux <= uy f = s | E | M
```

Condition 4: $x < 0$, $y < 0$

```
C x_signbit && y_signbit && ux >= uy
```

Code:

```
1 #include <stdio.h>
2 #include <assert.h>
3
4 unsigned f2u(float x)
5 {
6     return *(unsigned*)&x;
7 }
8
9 // Test if x <= y
10 int float_le(float x, float y) {
11     unsigned ux = f2u(x);
12     unsigned uy = f2u(y);
13
14     /* Get the sign bits */
15     unsigned sx = ux >> 31;
16     unsigned sy = uy >> 31;
17
18     /* Give an expression using only ux, uy, sx, and sy */
19     return (ux << 1 == 0 && uy << 1 == 0) // Condition 1
20         || (sx && !sy) // Condition 2
21         || (!sx && !sy && ux <= uy) // Condition 3
22         || (sx && sy && ux >= uy); // Condition 4
23 }
24
25 int main()
26 {
27     assert(float_le(+0.0, -0.0));
28     assert(float_le(-0.8, 0.9));
29     assert(float_le(5.0, 9.0));
```

```

30     assert(float_le(-9.0, -5.0));
31     return 0;
32 }

```

2.85

Given a floating-point format with a k -bit exponent and an n -bit fraction, write formulas for the exponent E , the significand M , the fraction f , and the value V for the quantities that follow. In addition, describe the bit representation.

- The number 7.0
- The largest odd integer that can be represented exactly
- The reciprocal of the smallest positive normalized value

Solution:

$$V = (-1)^s \times M \times 2^E$$

$$bias = 2^{k-1} - 1$$

A.

$$7.0 = 111.000_2 = 1.11_2 \times 2^2$$

so:

$$f = 0.110..0$$

$$E = e - bias \text{ then } e = E + bias = 10_2 + bias = 2^{k-1} + 1 = 100..01_2$$

$$\text{Therefore, } 7.0_{10} = 0\ 10..01\ 110..0$$

B.

$$\text{First } s = 0$$

Next we know if odd, the last bit of f must be 1, if biggest, $f = 0.1...1$ with n 1s,

$$M = 1 + f = 1.1...1 \text{ with } n \text{ 1s.}$$

Then E must be equal to n :

$$e - bias = n$$

$$e = bias + n = 2^{k-1} + n - 1$$

$$\text{Therefore, the result is: } 0\ 1...(n-1)_{10}\ 11...11 = 2^{n+1} - 1$$

C.

$$M = 1.0...0$$

because it's normalized, the smallest e will be $0...01$.

$$\text{Then } E = e - bias = 1 - bias$$

$$\text{Therefore, } V = 2^{1-bias}$$

$$\text{For reciprocal, } V_r = 2^{bias-1}$$

$$\text{Then, } s = 0, f = 0.0...0, E = bias - 1, e = 2 \times bias - 1 = 2 \times (2^{k-1} - 1) - 1 = 2^k - 3 = [111...101]_2$$

$$\text{the final } V = 0\ 111...101\ 000...000$$

2.86

Intel-compatible processors also support an "extended-precision" floating-point format with an 80-bit word divided into a sign bit, $k = 15$ exponent bits, a single integer bit, and $n = 63$ fraction bits. The integer bit is an explicit copy of the implied bit in the IEEE floating-point representation. That is, it equals 1 for normalized values and 0 for denormalized values. Fill in the following table giving the approximate values of some "interesting" numbers in this format:

Description	Value	Decimal
Smallest positive denormalized	00..00..01	$2^{-61-2^{14}}$
Smallest positive normalized	00..010..0	$2^{2-2^{14}}$
Largest normalized	01..101..1	$(2 - 2^{-63}) \times 2^{bias}$

This format can be used in C programs compiled for Intel-compatible machines by declaring the data to be of type `long double`. However, it forces the compiler to generate code based

on the legacy 8087 floating-point instructions. The resulting program will most likely run much slower than would be the case for data type float or double.

Solution:

$$bias = 2^{k-1} - 1 = 2^{14} - 1$$

Smallest positive denormalized:

$$e = [0000..000]_2 = 0$$

$$M = f = 0.00...001$$

$$E = 1 - bias = 2 - 2^{14}$$

$$V = (-1)^s \times M \times 2^E = 2^{-63} \times 2^{(2-2^{14})} = 2^{-2^{14}-61}$$

Smallest positive normalized:

$$e = [000..001] = 1$$

$$E = e - bias = 2 - 2^{14}$$

$$M = 1 + f = 1$$

$$V = (-1)^s \times M \times 2^E = 2^{2-2^{14}}$$

Largest normalized:

$$e = [111..110] = 2^{15} - 2$$

$$E = e - bias = 2^{15} - 2 - 2^{14} + 1 = 2^{14} - 1$$

$$M = 1 + f = 1.11...11_2$$

$$V = (-1)^s \times M \times 2^E = 1.11...11_2 \times 2^{2^{14}-1} = (2 - 2^{-63}) \times 2^{2^{14}-1}$$

2.87

The 2008 version of the IEEE floating-point standard, named IEEE 754-2008, includes a 16-bit "half-precision" floating-point format. It was originally devised by computer graphics companies for storing data in which a higher dynamic range is required than can be achieved with 16-bit integers. This format has 1 sign bit, 5 exponent bits ($k = 5$), and 10 fraction bits ($n = 10$). The exponent bias is $2^{5-1} - 1 = 15$.

Fill in the table that follows for each of the numbers given, with the following instructions for each column:

Hex: The four hexadecimal digits describing the encoded form.

M: The value of the significand. This should be a number of the form x or $\frac{x}{y}$, where x is an integer and y is an integral power of 2. Examples include 0, $\frac{67}{64}$ and $\frac{1}{256}$

E: The integer value of the exponent.

V: The numeric value represented. Use the notation x or $x \times 2^z$, where x and z are integers.

D: The (possibly approximate) numerical value, as is printed using the %f formatting specification of printf.

As an example, to represent the number $\frac{7}{8}$, we would have $s = 0$, $M = \frac{7}{4}$ and $E = -1$. Our number would therefore have an exponent field of 01110_2 (decimal value $15 - 1 = 14$) and a significand field of 1100000000_2 , giving a hex representation 3B00. The numerical value is 0.875.

You need not fill in entries marked —.

Description	Hex	M	E	V	D
-0				-0	-0.0
Smallest value > 2					
512				512	512.0
Largest denormalized					
$-\infty$		—	—	$-\infty$	$-\infty$
Number with hex	3BB0				

Solution:

Description	Hex	M	E	V	D
-0	0x8000	0	-14	-0	-0.0
Smallest value > 2	0x4001	$1 + 2^{-10}$	1	$\frac{1025}{512}$	2.00195312
512	0x 6000	1	9	512	512.0
Largest denormalized	0x 03FF	1	-14	$\frac{1023}{2^{24}}$	6.09755516e-5
$-\infty$	0x FC00	—	—	$-\infty$	$-\infty$
Number with hex	3BB0	$\frac{123}{64}$	-1	$\frac{123}{128}$	0.9609375

1 sign bit, $k = 5$, $n = 10$

$\text{bias} = 2^{k-1} - 1 = 15$

-0:

$s = 1$

$e = 0, f = 0$

Hex = 8000

$M = f = 0$

$E = 1 - \text{bias} = -14$

Smallest value > 2:

if $V == 2$, $s = 0$, $M = 1$, $E = 1$, then $M \times 2^E == 2$

if $M == 1$, $f = 0$

if $E == 1$, $e - \text{bias} = 1$, then $e = 16 = [10000]_2$

if we want a value just above 2, we can set $f = 0000000001_2$ and $M = 1 + f = 1 + 2^{-10}$

Therefore, Hex = 0b 0 10000 0000000001 = 0x 4001

$V = (1 + 2^{-10}) \times 2 = \frac{1025}{512}$

512:

$512 = 2^9$

so $s = 0$, $M = 1$, $E = 9$

if $M == 1$, $f = 0$

if $E == 9$, $e = \text{bias} + 9 = 24 = [11000]$

Hex = 0b 0 11000 0000000000 = 0x 6000

$V = 512$

Largest denormalized:

$e = 0$, $s = 0$, $E = -14$

we only need settle M

$M = f = 0.111..11_2$

Hex = 0b 0 00000 1111111111 = 0x 03FF

$V = (1 - 2^{-10}) \times 2^{-14} = \frac{1023}{2^{24}}$

$-\infty$:

$e = [11111]_2$

$s = 1$

$f = 0000000000$

Hex = 0b 1 11111 0000000000 = 0x FC00

Number with hex 3BB0:

hex = 0x 3BB0 = 0b 0 01110 1110110000

$s = 0$,

$e = 0b 01110 = 14$, $E = e - \text{bias} = -1$

$f = 0.1110110000_2$, $M = 1 + f = 1.1110110000_2 = \frac{123}{64}$

$V = 1.1110110000_2 \times 2^{-1} = \frac{123}{128}$

2.88

Consider the following two 9-bit floating-point representations based on the IEEE floating-point format.

1. Format A

There is 1 sign bit.

There are $k = 5$ exponent bits. The exponent bias is 15. There are $n = 3$ fraction bits.

2. Format B

There is 1 sign bit.

There are $k = 4$ exponent bits. The exponent bias is 7. There are $n = 4$ fraction bits.

In the following table, you are given some bit patterns in format A, and your task is to convert them to the closest value in format B. If rounding is necessary you should round toward $+\infty$. In addition, give the values of numbers given by the format A and format B bit patterns. Give these as whole numbers (e.g., 17) or as fractions (e.g., $17/64$ or $17/26$).

FormatA - Bits	FormatA - Value	FormatB - Bits	FormatB - Value
1 01111 001			
0 10110 011			
1 00111 010			
0 00000 111			
1 11100 000			
0 10111 100			

Solution:

FormatA - Bits	FormatA - Value	FormatB - Bits	FormatB - Value
1 01111 001	$-\frac{9}{8}$	1 0111 0010	$-\frac{9}{8}$
0 10110 011	176	0 1110 0110	176
1 00111 010	$-\frac{5}{1024}$	1 0000 0101	$-\frac{5}{1024}$
0 00000 111	$\frac{7}{2^{17}}$	0 0000 0001	$\frac{1}{2^{10}}$
1 11100 000	-2^{13}	1 1110 1111	-248
0 10111 100	3×2^7	0 1111 0000	$+\infty$

A: $s = 1$, $k = 5$, $n = 3$, bias = 15

B: $s = 1$, $k = 4$, $n = 4$, bias = 7

1 01111 001:

Format A:

$$s_A = 1$$

$$e_A = [01111]_2 = 15 - \text{normalized value}$$

$$E_A = e_A - \text{bias}_A = 0$$

$$f_A = 0.001_2$$

$$M_A = 1 + f_A = 1.001_2 = \frac{9}{8}$$

$$V_A = (-1) \times \frac{9}{8} \times 2^0 = -\frac{9}{8}$$

Format B:

$$s_B = 1$$

$$E_B = E_A \rightarrow e_B = 7 = [0111]_2$$

$$f_B = f_A = 0.0010_2$$

$$V_B = -\frac{9}{8} = [1 \ 0111 \ 0010]$$

0 10110 011:

Format A:

$$s = 0$$

$$e = [10110]_2 = 22 - \text{normalized value}$$

$$E = e - \text{bias} = 7$$

$$f = 0.011_2$$

$$M = 1 + f = 1.011_2 = \frac{11}{8}$$

$$V = \frac{11}{8} \times 2^7 = 176$$

Format B:

$$E_B = 7 \rightarrow e_B = 7 + 7 = 14 = [1110]_2$$

$$f_B = 0.0110 = f_A$$

$$V_B = 176 = [0 \ 1110 \ 0110]$$

1 00111 010:

Format A:

$$s_A = 1$$

$$e_A = [00111]_2 = 7 - \text{normalized value}$$

$$E_A = e_A - \text{bias}_A = -8$$

$$f_A = 0.010_2$$

$$M_A = 1 + f_A = 1.010_2 = \frac{5}{4}$$

$$V_A = (-1) \times \frac{5}{4} \times 2^{-8} = -\frac{5}{1024}$$

Format B:

$$s_B = 1$$

$$E_B = E_A \rightarrow e_B = E_A + \text{bias}_B$$

Normalized: $[0001] \leq e_B \leq [1110] \rightarrow -6 \leq E_B \leq 7$ - to discard!

Denormalized: $e_B = 0$

$$E_B = -6$$

$$\text{To make sure, } V_B = V_A = (-1) \times M_B \times 2^{-6} = -\frac{5}{1024} = -\frac{5}{2^{10}}$$

$$\text{so } M_B = \frac{5}{16} = 0.0101_2 = f$$

$$V_B = 1 \ 0000 \ 0101$$

0 00000 111:

Format A:

$$s_A = 0$$

$$e_A = [00000]_2 - \text{denormalized value}$$

$$E_A = 1 - \text{bias}_A = -14$$

$$f_A = 0.111_2$$

$$M_A = f_A = \frac{7}{8}$$

$$V_A = \times \frac{7}{8} \times 2^{-14} = \frac{7}{2^{17}}$$

Format B:

In denormalized form:

$$s_B = 0$$

$$e_B = 0000$$

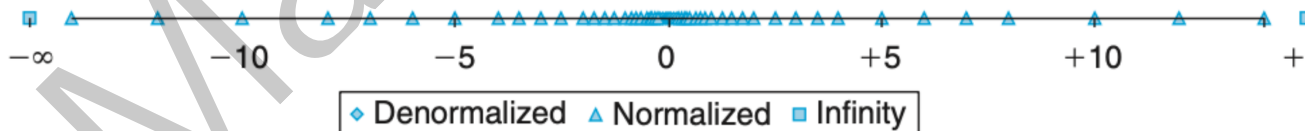
$$E_B = -6$$

$$f_B = 0.0001_2 = M_B$$

$$V_B = \frac{1}{2^{10}} = 0 \ 0000 \ 0001$$

because it's rounding toward $+\infty$ and $\frac{1}{2^{10}}$ is already the smallest value in denormalized form, it's the correct result.

Why no need to consider the normalized form? because the smallest value of normalized form must be larger than that of denormalized form.



1 11100 000:

Format A:

$$s_A = 1$$

$$e_A = [11100]_2 = 28 - \text{normalized value}$$

$$E_A = e_A - \text{bias}_A = 13$$

$$f_A = 0.000_2$$

$$M_A = 1$$

$$V_A = (-1) \times 1 \times 2^{13} = -2^{13}$$

Format B:

$$s_B = 1$$

$$E_B = E_A \rightarrow e_B = 28 + 7 = 35 \text{ we cannot fulfill}$$

in normalized form: $-6 \leq E_B \leq 7$, we find that cannot meet the exponent.

$$\text{suppose } E_B = 7 \rightarrow e_B = 1110_2$$

$f_B = 1111_2$
 $V_B = -1 \times 1.1111_2 \times 2^7 = -248 = 1\ 1110\ 1111$
 0 10111 100:
Format A:
 $s_A = 0$
 $e_A = [10111]_2 = 23$ - normalized form
 $E_A = e_A - bias_A = 8$
 $f_A = 0.100_2$
 $M_A = 1 + f_A = 1.100_2 = \frac{3}{2}$
 $V_A = \frac{3}{2} \times 2^8 = 3 \times 2^7$
Format B:
 $s_B = 0$
 if $E_B = E_A \rightarrow e_B = 15$ which is special value
 $V_B = +\infty = 0\ 1111\ 0000$

2.89

We are running programs on a machine where values of type `int` have a 32-bit two's-complement representation. Values of type `float` use the 32-bit IEEE format, and values of type `double` use the 64-bit IEEE format.

We generate arbitrary integer values `x`, `y`, and `z`, and convert them to values of type `double` as follows:

```

1  /* Create some arbitrary values */
2
3  int x = random();
4  int y = random();
5  int z = random();
6
7
8  /* Convert to double */
9
10 double dx = (double) x;
11 double dy = (double) y;
12 double dz = (double) z;
```

For each of the following C expressions, you are to indicate whether or not the expression always yields 1. If it always yields 1, describe the underlying mathematical principles. Otherwise, give an example of arguments that make it yield 0. Note that you cannot use an IA32 machine running gcc to test your answers, since it would use the 80-bit extended-precision representation for both `float` and `double`.

1. `(float) x == (float) dx`
2. `dx - dy == (double) (x-y)`
3. `(dx+dy)+dz==dx+(dy+dz)`
4. `(dx*dy)*dz==dx*(dy*dz)`
5. `dx/dx==dz/dz`

Solution:

1. `(float) x == (float) dx`: right
2. `dx - dy == (double) (x-y)`: wrong when `x = 0`, `y = INT_MIN`
3. `(dx+dy)+dz==dx+(dy+dz)`: right
4. `(dx*dy)*dz==dx*(dy*dz)`: wrong
 $(1e20\ 1e20)\ 1e(-20) = +\infty$
 $1e20\ (1e20\ 1e(-20)) = 1e20$
5. `dx/dx==dz/dz`: wrong when `dx != 0` and `dz == 0`

2.90

You have been assigned the task of writing a C function to compute a floating-point representation of 2^x . You decide that the best way to do this is to directly construct the IEEE single-precision representation of the result. When x is too small, your routine will return 0.0. When x is too large, it will return $+\infty$. Fill in the blank portions of the code that follows to compute the correct result. Assume the function `u2f` returns a floating-point value having an identical bit representation as its unsigned argument.

```
1  float fpwr2(int x)
2  {
3
4      /* Result exponent and fraction */
5      unsigned exp, frac;
6      unsigned u;
7
8      if (x < 0) {
9          /* Too small. Return 0.0 */ exp = ;
10         frac = ;
11
12     } elseif(x<0){ /* Denormalized result */ exp = ;
13     frac = ;
14
15     } else if (x < 0) { /* Normalized result. */ exp = ;
16     frac = ;
17
18     } else {
19         /* Too big. Return +oo */ exp = ;
20         frac = ;
21
22     }
23
24     /* Pack exp and frac into 32 bits */
25     u = exp << 23 | frac;
26     /* Return as float */
27     return u2f(u);
28 }
29 }
```

Solution:

floating-point is single precision:

$s = 1, k = 8, n = 23$

$bias = 2^{k-1} - 1 = 127$

For positive normalized,

For max, $e = [11111110]$, so $E = e - bias = 127$

$V_{max} = (-1)^0 \times 1.11...11_2 \times 2^{127}$ is in range $(2^{127}, 2^{128})$

For min, $e = [00000001]$, so $E = -126$

$V_{min} = (-1)^0 \times 1.00...00 \times 2^{-126} = 2^{-126}$

For positive denormalized,

$E = 1 - bias = -126$

$M = 0.f = 0.111...11_2$

$V_{max} \approx 2^{-126}$

$M_{min} = f_{min} = 0.000...001_2$

$V_{min} = (-1)^0 \times 0.00...001 \times 2^{-127} = 2^{-149}$

$0 \rightarrow 2^{-149} \rightarrow 2^{-126} \rightarrow 2^{127} \rightarrow +\infty$

1. $0 \rightarrow 2^{-149} \approx 0.0$ then $e = 0, f = 0$

2. $2^{-149} \rightarrow 2^{-126}$: $e = 0, f$ is in range $[00...00]$ and $[11...10]$

we know $E = 2^{-126}$

in denormalized form, $M = 0.f$

$2^x = 2^{-126} \times 2^{x+126}$, then $f = 2^{x+126} \times 2^{23} = 2^{x+149} = 1 \ll (x + 149)$

3. $2^{-126} \rightarrow 2^{127}$:

we wanna do 2^x , in normalized case, $M = 1 + 0.f$ cannot be represented into 2^x , so f have to be 0.

$f = 0$, e is in range $[0000\ 0001]$ to $[1111\ 1110]$
 for $2^x = 2^{x+bias-bias}$, then $e = x + bias = x + 127$
 4. $2^{127} \rightarrow +\infty$: $e = [1111\ 1111]$, $f = 0$

Therefore, the code is like:

```

1  #include <limits.h>
2  #include <stdio.h>
3  #include <math.h>
4  #include <assert.h>
5
6
7  float u2f(unsigned x)
8  {
9      unsigned* p_x = &x;
10     return *(float*)p_x;
11 }
12
13 float fpwr2(int x)
14 {
15     /* Result exponent and fraction */
16     unsigned exp, frac;
17     unsigned u;
18
19     /*if x < smallest positive denormalized, return 0.0 (denormalized) */
20     if (x < -149) {
21         /* Too small. Return 0.0 */
22         exp = 0;
23         frac = 0;
24     }
25     /*if x < smallest positive normalized / largest positive denormalized, return
    (denormalized)*/
26     else if (x < -126)
27     { /* Denormalized result */
28         exp = 0;
29         frac = 1 << (x + 149);
30     }
31     else if (x < 128)
32     { /* Normalized result. */
33         exp = x + 127;
34         frac = 0;
35     }
36     else {
37         /* Too big. Return +oo */
38         exp = 0xFF;
39         frac = 0;
40     }
41
42     /* Pack exp and frac into 32 bits */
43     u = (exp << 23 | frac) & INT_MAX;
44     /* Return as float */
45     return u2f(u);
46 }
47
48 int main()
49 {
50     assert(fpwr2(0) == powf(2, 0));
51     assert(fpwr2(100) == powf(2, 100));
52     assert(fpwr2(-100) == powf(2, -100));
53     assert(fpwr2(10000) == powf(2, 10000));
54     assert(fpwr2(-10000) == powf(2, -10000));
55     return 0;
56 }

```

2.91

Around 250 B.C., the Greek mathematician Archimedes proved that $\frac{223}{71} < \pi < \frac{22}{7}$. Had he had access to a computer and the standard library `<math.h>`, he would have been able to determine that the single-precision floating-point approximation of π has the hexadecimal

representation 0x40490FDB. Of course, all of these are just approximations, since π is not rational.

A. What is the fractional binary number denoted by this floating-point value?

B. What is the fractional binary representation of $\frac{22}{7}$? Hint: See Problem 2.83.

C. At what bit position (relative to the binary point) do these two approximations to π diverge?

Solution:

A.

0x40490FDB = 0100 0000 0100 1001 0000 1111 1101 1011 = 0 10000000 10010010000111111011011

s = 0, e = 1000 0000, f = 10010010000111111011011

bias = 127

$V = (-1)^0 \times (1 + 0.f) \times 2^{e-127} = 11.0010010000111111011011_2$

B.

From 2.83, we know that:

$0.yyy\dots yyy = Y/(2^k - 1)$

y is a k-bit sequence

k is the width of y;

Y is the number having binary representation y;

So $\frac{22}{7} = 3\frac{1}{7}$,

suppose $Y = 1$, then $2^k - 1 = 7$, so we know the width of $y = k = 3$, the value of y is $Y = 1$, therefore, $y = 001$, Finally, $\frac{1}{7} = 0.001001001\dots$

$3\frac{1}{7} = 3 + \frac{1}{7} = 0b11.001001(001)\dots$

C.

$\frac{223}{71} = 3\frac{10}{71}$

$\frac{10}{71} \times 2 = \frac{20}{71}$, it's less than 1, so the 1st bit after fraction point is 0. $\rightarrow 0.0$

$\frac{20}{71} \times 2 = \frac{40}{71} \rightarrow 0.00$

$\frac{40}{71} \times 2 = \frac{80}{71} \rightarrow 0.001$

$\frac{9}{71} \times 2 = \frac{18}{71} \rightarrow 0.0010$

$\frac{18}{71} \times 2 = \frac{36}{71} \rightarrow 0.00100$

$\frac{36}{71} \times 2 = \frac{72}{71} \rightarrow 0.001001$

$\frac{1}{71} \times 2 = \frac{2}{71} \rightarrow 0.0010010$

$\frac{2}{71} \times 2 = \frac{4}{71} \rightarrow 0.00100100$

$\frac{4}{71} \times 2 = \frac{8}{71} \rightarrow 0.001001000$

so the 9th bit is different.

Bit-Level Floating-Point Coding Rules

In the following problems, you will write code to implement floating-point functions, operating directly on bit-level representations of floating-point numbers. Your code should exactly replicate the conventions for IEEE floating-point operations, including using round-to-even mode when rounding is required.

To this end, we define data type `float_bits` to be equivalent to unsigned:

```
1 /* Access bit-level representation floating-point number */
2
3 typedef unsigned float_bits;
```

Rather than using data type `float` in your code, you will use `float_bits`. You may use both `int` and unsigned data types, including unsigned and integer constants and operations. You may not use any unions, structs, or arrays. Most significantly, you may not use any floating-point data types, operations, or constants. Instead, your code should perform the bit manipulations that implement the specified floating-point operations.

The following function illustrates the use of these coding rules. For argument `f`, it returns ± 0 if `f` is denormalized (preserving the sign of `f`), and returns `f` otherwise.


```

1  /* If f is denorm, return 0. Otherwise, return f */
2  float_bits float_denorm_zero(float_bits f) {
3
4      /* Decompose bit representation into parts */
5      unsigned sign = f >> 31;
6      unsigned exp  = f >> 23 & 0xFF;
7      unsigned frac = f      & 0x7FFFFF;
8
9      if (exp == 0) {
10         /* Denormalized. Set fraction to 0 */
11         frac = 0;
12     }
13
14     /* Reassemble bits */
15     return (sign << 31) | (exp << 23) | frac;
16 }

```

2.92

Following the bit-level floating-point coding rules, implement the function with the following prototype:

```

1  /* Compute -f. If f is NaN, then return f. */
2  float_bits float_negate(float_bits f);

```

For floating-point number f , this function computes $-f$. If f is NaN, your function should simply return f .

Test your function by evaluating it for all 2^{32} values of argument f and comparing the result to what would be obtained using your machine's floating-point operations.

Solution:

```

1  #include <stdio.h>
2  #include <limits.h>
3  #include <assert.h>
4
5  /* Access bit-level representation floating-point number */
6  typedef unsigned float_bits;
7
8  float u2f(unsigned x)
9  {
10     unsigned* p_x = &x;
11     return *(float*)p_x;
12 }
13
14 unsigned f2u(float f)
15 {
16     return *(unsigned*)&f;
17 }
18
19 /* Compute -f. If f is NaN, then return f. */
20 float_bits float_negate(float_bits f)
21 {
22     /* Decompose bit representation into parts */
23     unsigned sign = f >> 31;
24     unsigned exp  = f >> 23 & 0xFF;
25     unsigned frac = f & 0x7FFFFF;
26
27     /* NaN */
28     if (exp == 0xFF && frac != 0) return f;
29
30     /* Reassemble bits */
31     return (~sign << 31) | (exp << 23) | frac;
32 }
33
34 int main()
35 {
36     unsigned x = 0;

```

```

37     while (x <= UINT_MAX)
38     {
39         assert(-u2f(x) == u2f(float_negate(x)));
40         x++;
41         printf("0x%X ", x);
42     }
43     return 0;
44 }

```

2.93

Following the bit-level floating-point coding rules, implement the function with the following prototype:

```

1  /* Compute |f|. If f is NaN, then return f. */
2  float_bits float_absval(float_bits f);

```

For floating-point number f , this function computes $|f|$. If f is NaN, your function should simply return f .

Test your function by evaluating it for all 2^{32} values of argument f and comparing the result to what would be obtained using your machine's floating-point operations.

Solution:

```

1  #include <stdio.h>
2  #include <limits.h>
3  #include <assert.h>
4  #include <math.h>
5
6  /* Access bit-level representation floating-point number */
7  typedef unsigned float_bits;
8
9  float u2f(unsigned x)
10 {
11     unsigned* p_x = &x;
12     return *(float*)p_x;
13 }
14
15 unsigned f2u(float f)
16 {
17     return *(unsigned*)&f;
18 }
19
20 /* Compute -f. If f is NaN, then return f. */
21 float_bits float_absval(float_bits f)
22 {
23     /* Decompose bit representation into parts */
24     unsigned sign = 0; // to align absolute value
25     unsigned exp = f >> 23 & 0xFF;
26     unsigned frac = f & 0x7FFFFF;
27
28     /* NaN */
29     if (exp == 0xFF && frac != 0) return f;
30
31     /* Reassemble bits */
32     return (sign << 31) | (exp << 23) | frac;
33 }
34
35 int main()
36 {
37     unsigned x = 0;
38     while (x <= UINT_MAX)
39     {
40         assert(fabs(u2f(x)) == u2f(float_absval(x))); // fabs() - calculate the absolute value
41         // of a floating-point value.
42         x++;
43         printf("0x%X \n", x);
44     }
45     return 0;

```

2.94

Following the bit-level floating-point coding rules, implement the function with the following prototype:

```
1  /* Compute 2*f. If f is NaN, then return f. */
2  float_bits float_twice(float_bits f);
```

For floating-point number f , this function computes $2.0 \times f$. If f is NaN, your function should simply return f .

Test your function by evaluating it for all 2^{32} values of argument f and comparing the result to what would be obtained using your machine's floating-point operations.

Solution:

```
1  #include <stdio.h>
2  #include <limits.h>
3  #include <assert.h>
4  #include <math.h>
5
6  /* Access bit-level representation floating-point number */
7  typedef unsigned float_bits;
8
9  float u2f(unsigned x)
10 {
11     unsigned* p_x = &x;
12     return *(float*)p_x;
13 }
14
15 unsigned f2u(float f)
16 {
17     return *(unsigned*)&f;
18 }
19
20 /* Compute -f. If f is NaN, then return f. */
21 float_bits float_twice(float_bits f)
22 {
23     /* Decompose bit representation into parts */
24     unsigned sign = f >> 31; // to align absolute value
25     unsigned exp = f >> 23 & 0xFF;
26     unsigned frac = f & 0x7FFFFFFF;
27
28     /* NaN */
29     if (exp == 0xFF && frac != 0) return f;
30
31     /* Denormalized value */
32     if (exp == 0)
33     {
34         frac = frac << 1;
35     }
36     /* infinity */
37     else if (exp == 0xFE)
38     {
39         return f;
40     }
41     else exp += 1;
42
43     /* Reassemble bits */
44     return (sign << 31) | (exp << 23) | frac;
45 }
46
47 int main()
48 {
49     unsigned x = 0;
50     while (x < UINT_MAX)
51     {
52         assert((u2f(x))*2 == u2f(float_twice(x)));
```

```

53         x++;
54         printf("0x%X ", x);
55     }
56     return 0;
57 }
58

```

2.95

Following the bit-level floating-point coding rules, implement the function with the following prototype:

```

1  /* Compute 0.5*f. If f is NaN, then return f. */
2  float_bits float_half(float_bits f);

```

For floating-point number f , this function computes $0.5 \times f$. If f is NaN, your function should simply return f .

Test your function by evaluating it for all 2^{32} values of argument f and comparing the result to what would be obtained using your machine's floating-point operations.

Solution:

We will use rounding to even rule when we need to do rounding.

Here's some points we need to talk about:

In denormalized mode:

- Denormalized: $e=0$, $M=f$.
- when do halving, means $f \gg 1$.
- because of rounding-to-even rule, we need to consider last 2 bits of f
 - if $f = \dots 11$, after $f \gg 1$, we need to add 1 to make it be even.
 - else, we just $f \gg 1$, no need to add 1.
 - if $f = \dots 10$, after $f \gg 1$, $f = \dots 1$
 - if $f = \dots 00$, after $f \gg 1$, $f = \dots 0$
 - if $f = \dots 01$, after $f \gg 1$, $f = \dots 0$

```

1  /* Denormalized value */
2  if (exp == 0)
3  {
4      frac = frac >> 1;
5      frac += carrybit;
6  }

```

In normalized mode when $e = 1$:

- there's one exception that is $e=0..01$, when do the halving, it will transit to the denormalized mode.
- in normalized mode: $V_n = (1 + 0.f_n) \times 2^{1-bias}$ when $e = 1$
- in denormalized mode: $V_d = 0.f_d \times 2^{1-bias}$
- so we get that, if we wanna do the halving, we only need to do
 - firstly $e \gg 1$ which makes $e = 0$.
 - secondly $1.f_n \gg 1 = 0.f_d$ to make $M_n \times 0.5 = M_d$
- Also we need to consider the round-to-even rule on $1.f_n \gg 1 = f_d$

```

1  /* Normalized transit to denormalized */
2  else if (exp == 0x1)
3  {
4      exp = 0;
5      frac = ((frac >> 1) | 0x400000) + carrybit;
6  }

```

In normalized mode when $e \neq 1$:

```
1 else exp -= 1;
```

Fullcode:

```
1  #include <stdio.h>
2  #include <limits.h>
3  #include <assert.h>
4  #include <math.h>
5
6  /* Access bit-level representation floating-point number */
7  typedef unsigned float_bits;
8
9  float u2f(unsigned x)
10 {
11     unsigned* p_x = &x;
12     return *(float*)p_x;
13 }
14
15 unsigned f2u(float f)
16 {
17     return *(unsigned*)&f;
18 }
19
20 /* Compute 0.5*f. If f is NaN, then return f. */
21 float_bits float_half(float_bits f)
22 {
23     /* Decompose bit representation into parts */
24     unsigned sign = f >> 31; // to align absolute value
25     unsigned exp = f >> 23 & 0xFF;
26     unsigned frac = f & 0x7FFFFFFF;
27
28     /* NaN */
29     if (exp == 0xFF && frac != 0) return f;
30
31     int carrybit = (frac & 0x3) == 0x3; // When do round-to-even, if or not need to carry bit
32
33     /* Denormalized value */
34     if (exp == 0)
35     {
36         frac = frac >> 1;
37         frac += carrybit;
38     }
39     /* Normalized transit to denormalized */
40     else if (exp == 0x1)
41     {
42         exp = 0;
43         frac = ((frac >> 1) | 0x400000) + carrybit;
44     }
45     else exp -= 1;
46
47     /* Reassemble bits */
48     return (sign << 31) | (exp << 23) | frac;
49 }
50
51 int main()
52 {
53     unsigned x = 0;
54     while (x < UINT_MAX)
55     {
56         assert((u2f(x)) / 2 == u2f(float_half(x)));
57         x++;
58         printf("0x%X ", x);
59     }
60     return 0;
61 }
```

2.96 ◆◆◆◆

Following the bit-level floating-point coding rules, implement the function with the following prototype:

```

1  /*
2   * Compute (int) f.
3   * If conversion causes overflow or f is NaN, return 0x80000000
4   */
5  int float_f2i(float_bits f);

```

For floating-point number f , this function computes $(\text{int}) f$. Your function should round toward zero. If f cannot be represented as an integer (e.g., it is out of range, or it is NaN), then the function should return 0x80000000.

Test your function by evaluating it for all 2^{32} values of argument f and comparing the result to what would be obtained using your machine's floating-point operations.

Solution:

Case 1: Normalized Values

- Pre-condition: $\text{exp} \neq [00\dots 0] \text{ or } [11\dots 1]$
- $V = (-1)^s \times (1 + 0.f_{n-1}\dots f_1 f_0) \times 2^{[e_{k-1}\dots e_1 e_0] - (2^{k-1} - 1)}$

Case 2: Denormalized Values (非规格化)

- Pre-condition: $\text{exp} = [00\dots 0]$
- $V = (-1)^s \times 0.f_{n-1}\dots f_1 f_0 \times 2^{1 - (2^{k-1} - 1)}$

Case 3: Special Values

- Pre-condition: $\text{exp} = [11\dots 1]$
- If $f = [00\dots 0]$ and $s = 0$, then $V = +\infty$
- If $f = [00\dots 0]$ and $s = 1$, then $V = -\infty$
- If $f = \text{nonzero}$, then $V = \text{NaN}$

Then we need to think about set checkpoint of 1 and $+\infty$

- $[-1, 1]$: return 0 based on rounding to zero rule.
 - $[1, \text{INT_MAX}]$ or $[\text{INT_MIN}, -1]$: return based on rounding to zero rule.
 - $[\text{INT_MAX}, +\infty]$ or $[-\infty, \text{INT_MIN}]$: return 0x80000000
- Checkpoint values on floating representations:
- Value 1 or -1: $f = 0$ and $e = \text{bias} = 0x7F$
 - $M = 1 + 0.f = 1$
 - $E = e - \text{bias} = 0$
 - thus $V = 1$ or -1 based on your sign bit.
 - INT_MAX or INT_MIN
 - INT_MAX : $0x7FFFFFFF \approx 1.1\dots 1_2 \times 2^{30} \approx 2^{31}$
 - INT_MIN : $0x80000000 = 2^{31}$
 - So when $E \geq 31$, it's overflow and we should return 0x80000000, then in this case that $e \geq \text{bias} + 31$.
 - Infinity
 - * when $v == +\infty$ or $-\infty$, $f=0$ and $e = 0xFF$
- Some ideas we get:
- value from -1 to +1 (all denormalized values and part of normalized values):

```

1  if(e <= bias && e >= 0)
2  {
3      result = 0;
4  }

```

- value from $\text{INT_MIN}/\text{INT_MAX}$ to $-\infty$ or $+\infty$:

```

1  if(e >= bias + 31)
2  {
3      result = 0x80000000;
4  }

```

- value from INT_MIN to -1 and from +1 to INT_MAX (Normalized values):

```

1  if(e > bias && e < bias + 31)
2  {
3      E = e - bias;
4      M = 0x800000 | f; // this M already shift n=23 bits
5      //result = M >> 23 << E = M << E-23;
6      if(E > 23) result = M << (E - 23);
7      else result = M >> (23 - E);
8  }

```

- Note: $M \times 2^{-23} \times 2^E = M \times 2^{E-23}$

Integrate all code:

```

1  #include <stdio.h>
2  #include <limits.h>
3  #include <assert.h>
4  #include <math.h>
5
6  /* Access bit-level representation floating-point number */
7  typedef unsigned float_bits;
8
9  float u2f(unsigned x)
10 {
11     unsigned* p_x = &x;
12     return *(float*)p_x;
13 }
14
15 unsigned f2u(float f)
16 {
17     return *(unsigned*)&f;
18 }
19
20 /*
21  * Compute (int) f.
22  * If conversion causes overflow or f is NaN, return 0x80000000
23  */
24 int float_f2i(float_bits f)
25 {
26     /* Decompose bit representation into parts */
27     unsigned sign = f >> 31; // to align absolute value
28     unsigned exp = f >> 23 & 0xFF;
29     unsigned frac = f & 0x7FFFFF;
30     unsigned result = 0;
31     int E;
32     int bias = 0x7F;
33     unsigned M;
34
35     if (exp == 0xFF && frac != 0) return 0x80000000;
36
37     if (exp < bias && exp >= 0) // value from -1 to +1 (all denormalized values and part of
38         normalized values)
39     {
40         result = 0;
41         return result;
42     }
43     else if (exp >= bias + 31) // value from INT_MIN/INT_MAX to negative infinity or
44         positive infinity:
45     {
46         result = 0x80000000;
47         return result;
48     }
49     else // (e > bias && e < bias + 31): value from INT_MIN to -1 and from +1 to INT_MAX
50         (Normalized values)
51     {
52         E = exp - bias;
53         M = 0x800000 | frac; // this M already shift n=23 bits
54         //result = M >> 23 << E = M << E-23;
55         if (E > 23) result = M << (E - 23);
56         else result = M >> (23 - E);
57     }
58 }

```

```

58     /* Reassemble bits */
59     return (sign << 31) | result;
60 }
61
62 int main()
63 {
64     unsigned x = 0x3F800000; // 0 01111111 000000000000000000000000
65     while (x < UINT_MAX)
66     {
67         float fx = *(float*) & x;
68         int ix = float_f2i(x);
69         assert((int)fx == ix);
70         x++;
71         printf("Original unsigned: 0x%08X, Integer: %08d, Floating: %08f\n", x, ix, fx);
72     }
73     return 0;
74 }

```

2.97 ◆◆◆◆

Following the bit-level floating-point coding rules, implement the function with the following prototype:

```

1  /* Compute (float) i */
2  float_bits float_i2f(int i);

```

For argument *i*, this function computes the bit-level representation of (float) *i*.

Test your function by evaluating it for all 2^{32} values of argument *f* and comparing the result to what would be obtained using your machine's floating-point operations.

Solution:

Our goal is to extract sign bit, exponent bits and fraction bits from the integer. For sign bit, we need:

- extract the MSB
- one point to notice: for negative integer, it's two's complement encoding. We need to change it back to unsigned encoding and store the sign bit.

```

1  if ((INT_MIN & i) == 0)
2  {
3      sign = 0;
4      u = (unsigned)i;
5  }
6  else{
7      sign = 1;
8      u = (unsigned)(~i + 1); // to transite from 2's complement to unsigned encoding.
9  }

```

For exponent bits, we need:

- calculate the bit width of the integer (width).
 - For example: 1111's width 4, 1.111×2^3
- then *f* width (frac_width) should be width - 1.
- $E = \text{exp} - \text{bias} = \text{width} - 1$, then $\text{exp} = \text{width} - 1 + \text{bias}$.
- because the range of floating representation is far beyond the the range of integer, we do not need to talk about overflow.

```

1  exp = frac_width + bias;

```

For fraction bits (*f*), we need:

- *f* should be the *i* removing the MSB.
- some special conditions:

- if `frac_width > n` which is 23 in single-floating representation, we need to take rounding (round-to-even) into consideration.
- when doing round-to-even, in some cases, `exp` also need to add 1.

```

1  if (frac_width < 24) // n = 23 is the precision of single-floating representation. No rounding
    required.
2      {
3          frac = raw_frac << (23 - frac_width);
4      }
5  else
6      {
7          frac = raw_frac >> (frac_width - 23);
8      }

```

Rounding (3 cases - rounding to even):

- `rounding_part > rounding_halfflag`
- `rounding_part < rounding_halfflag`
- `rounding_part == rounding_halfflag`: rounding to even

```

1          if (rounding_part > rounding_halfflag) // round to closest
2      {
3
4          // frac = (frac + rounding_halfflag) & (~rounding_mask);
5          raw_frac = raw_frac + rounding_halfflag;
6          if (bit_width(raw_frac) > frac_width) // if(there's carry bit) exp need to add
7      {
8          exp++;
9          //rounding_mask = (rounding_mask << 1) + 1;
10         raw_frac = raw_frac & (~rounding_mask);
11         //frac_width++;
12     }
13
14 }
15 else if (rounding_part < rounding_halfflag) // round to closest
16 {
17     raw_frac = raw_frac & (~rounding_mask);
18 }
19 else // round to even
20 {
21     unsigned odd_bit = raw_frac & (~rounding_mask) & ((rounding_mask << 1) + 1);
22     if (odd_bit)
23     {
24         raw_frac = raw_frac + rounding_part;
25         if (bit_width(raw_frac) > frac_width) // if(there's carry bit) exp need to
26     {
27         exp++;
28         //rounding_mask = (rounding_mask << 1) + 1;
29         raw_frac = raw_frac & (~rounding_mask);
30         //frac_width++;
31     }
32     }
33     else
34     {
35         raw_frac = raw_frac & (~rounding_mask);
36     }
37 }

```

Full code:

```

1  #include <stdio.h>
2  #include <limits.h>
3  #include <assert.h>
4  #include <math.h>
5
6  /* Access bit-level representation floating-point number */
7  typedef unsigned float_bits;
8

```

```

9 // Caculate the binary width of ux
10 unsigned bit_width(unsigned ux)
11 {
12     if ((ux & INT_MIN) != 0)
13     {
14         return 0x20;
15     }
16
17     unsigned width = 0;
18     while (ux != 0)
19     {
20         width++;
21         ux = ux >> 1;
22     }
23     return width;
24 }
25
26 // Generate the bit mast of x bits.
27 unsigned bit_mask(unsigned x)
28 {
29     unsigned w = sizeof(unsigned) << 3;
30     if (x == 0) return 0;
31     return (unsigned)(-1) >> (w - x);
32 }
33
34 float u2f(unsigned x)
35 {
36     unsigned* p_x = &x;
37     return *(float*)p_x;
38 }
39
40 unsigned f2u(float f)
41 {
42     return *(unsigned*)&f;
43 }
44
45 /* Generate fraction bits based on the u.*/
46 /* For example, if u = 0b 11, then the function will return 0b 1100 0000 ... 0000*/
47 unsigned frac_gen(unsigned u)
48 {
49     unsigned ret = u;
50     while ((0x80000000 & ret) == 0)
51     {
52         ret = ret << 1;
53     }
54     return ret;
55 }
56
57 /* Compute (float) i */
58 float_bits float_i2f(int i)
59 {
60     /* Decompose bit representation into parts */
61     unsigned sign;
62     unsigned frac;
63     unsigned exp;
64
65     /* Other useful local variables */
66     unsigned width; // bit width of integer i
67     unsigned u; // unsigned representation of integer i
68     int bias = 0x7F; // k = 8
69     unsigned frac_mask; // frac = frac_mask & u
70     unsigned frac_width; // the width of frac
71     unsigned raw_frac; // fraction on the rightmost.
72     unsigned rounding_part, rounding_mask, rounding_halfflag;
73     float_bits result;
74
75     // get the sign bit and transite 2's complement encoding to unsigned encoding.
76     if ((INT_MIN & i) == 0)
77     {
78         sign = 0;
79         u = (unsigned)i;
80     }
81     else {
82         sign = 1;
83         u = (unsigned)(~i + 1);

```

```

84     }
85
86     // exception: i == 0
87     if (i == 0) {
88         frac = 0;
89         exp = 0;
90         return sign << 31 | exp << 23 | frac;
91     }
92     // exception; i == INT_MIN
93     else if (i == INT_MIN) {
94         sign = 1;
95         exp = bias + 31;
96         frac = 0;
97         return sign << 31 | exp << 23 | frac;
98     }
99
100    width = bit_width(u);
101    frac_width = width - 1;
102    frac_mask = bit_mask(frac_width);
103    raw_frac = u & frac_mask;
104
105    if (frac_width < 24) // n = 23 is the precision of single-floating representation. No
rounding required.
106    {
107        exp = frac_width + bias;
108        frac = raw_frac << (23 - frac_width);
109    }
110    else // consider rounding-to-even rule
111    {
112        exp = frac_width + bias;
113        rounding_mask = bit_mask(frac_width - 23);
114        rounding_part = rounding_mask & raw_frac;
115        rounding_halfflag = (rounding_mask + 1) >> 1;
116        if (rounding_part > rounding_halfflag) // round to closest
117        {
118
119            // frac = (frac + rounding_halfflag) & (~rounding_mask);
120            raw_frac = raw_frac + rounding_halfflag;
121            if (bit_width(raw_frac) > frac_width) // if(there's carry bit) exp need to add
122            {
123                exp++;
124                //rounding_mask = (rounding_mask << 1) + 1;
125                raw_frac = raw_frac & (~rounding_mask);
126                //frac_width++;
127            }
128
129        }
130        else if (rounding_part < rounding_halfflag) // round to closest
131        {
132            raw_frac = raw_frac & (~rounding_mask);
133        }
134        else // round to even
135        {
136            unsigned odd_bit = raw_frac & (~rounding_mask) & ((rounding_mask << 1) + 1);
137            if (odd_bit)
138            {
139                raw_frac = raw_frac + rounding_part;
140                if (bit_width(raw_frac) > frac_width) // if(there's carry bit) exp need to
add
141                {
142                    exp++;
143                    //rounding_mask = (rounding_mask << 1) + 1;
144                    raw_frac = raw_frac & (~rounding_mask);
145                    //frac_width++;
146                }
147            }
148            else
149            {
150                raw_frac = raw_frac & (~rounding_mask);
151            }
152        }
153        frac = raw_frac >> (frac_width - 23);
154        frac = frac & 0x7FFFFF;
155    }
156

```

```

157     return sign << 31 | exp << 23 | frac;
158 }
159
160 int main()
161 {
162     //int ix = INT_MIN + 1; // 0 01111111 000000000000000000000000
163     //int ix = INT_MIN+1;
164     //int ix = 0;
165     int ix = INT_MAX - 0x100;
166     while (ix < INT_MAX)
167     {
168         float_bits fbx = float_i2f(ix);
169         float fx = *(float*)&fbx;
170         float fix = (float)ix;
171         assert(fix == fx);
172         printf("ix value: %08d, Floating from float_i2f: %08f, Floating from casting:
173         %08f\n", ix, fx, fix);
174         ix++;
175     }
176     return 0;

```