PES, Section 2.6
Shift ops

1. What is the value (in hex) of 0xAAAA << 3?

   1010 1010 1010 1010 << 3
   ~~101~~0 1010 1010 1010 000
   0101 0101 0101 0000
   0x5550

2. What is the value of 21 >> 2?

   The key is to remember that this is the decimal value 21, not the hex value 0x21!

   One approach is to interpret >> 2 as integer division (quotient) by 4.
   21 / 4 = 5

   Alternately, we can convert 21 to binary and perform the shift. Using 8 bits (i.e., a char), we can represent 21 as 0001 0101 (16 + 4 + 1).

   0001 0101 >> 2
   00 0001 01~~01~~
   0000 0101
   5

3. Write a single RIMS-compatible C-language statement that copies the values of A7…A5 to B2…B0, inverts the values of A4…A2 and copies them to B7…B5, and copies the values of A1…A0 to B4…B3.

```
B =  ( (A & 0xE0) >> 5) |
     ( ((~A) & 0x1C) << 3) |
     ( (A & 0x03) << 3);
```

4. A parking lot has eight spaces, each with a sensor connected to RIM input A7, A6, ..., or A0. A RIM input being 1 means a car is detected in the corresponding space. Spaces A7 and A6 are reserved handicapped parking. Write a RIM C program that:

   (1) Sets B0 to 1 if both handicapped spaces are full, and
   (2) Sets B7..B5 equal to the number of available non-handicapped spaces.

   The first part can be accomplished by masking and shifting

```
unsigned char full = ((A & 0x80) >> 7) & ((A & 0x40) >> 6);
```

   The second part is best handled in several steps:

```
unsigned char cnt = ((A & 0x20) >> 5) +
                    ((A & 0x10) >> 4) +
                    ((A & 0x08) >> 3) +
                    ((A & 0x04) >> 2) +
                    ((A & 0x02) >> 1) +
                    (A & 0x01);
```

   Now, we need to shift cnt left to put the 3 bits in positions B7..B5, while writing to B0 at the same time:

```
B = (cnt << 5) | full;
```

   Note that it is not possible to write to B0 in one statement, followed by B7…B5 in the next statement, e.g.:

```
B0 = full;
B = (cnt << 5);
```

   The second statement would overwrite the value of B0.

5. Binary coded decimal (BCD) is encodes decimal (base-10) numbers in which the value of each digit (0-9) is represented by a 4-bits (values in the range 10-15 are not allowed). BCD takes advantage of the fact that any one decimal numeral can be represented by a four bit pattern. The most obvious way of encoding digits is "natural BCD" (NBCD), where each decimal digit is represented by its corresponding four-bit binary value, as shown in the following table. This is also called "8421" encoding.

Decimal	BCD 8421
0	0 0 0 0
1	0 0 0 1
2	0 0 1 0
3	0 0 1 1
4	0 1 0 0
5	0 1 0 1
6	0 1 1 0
7	0 1 1 1
8	1 0 0 0
9	1 0 0 1

With 8 bits of I/O, RIMS can express a 2-digit BCD number in the range 0-99. Assume that A3-A0 and B3-B0 represent the lower digits, and A7-A4 and B7-B4 represent the upper digits.

Write a short sequence of RIMS-compatible C language statements that interprets the value of A as a 2-digit BCD number adds 1 to it, and outputs the value to B. (assume that 99 + 1 = 00). Assume that illegal BCD inputs (i.e., hex values in the range A-F) cannot occur as inputs.

```
unsigned char lower = A & 0x0F;
unsigned char upper = A & 0xF0 >> 4;
unsigned char carry = 0;

if( lower < 9 )
    lower++;
else {
    lower = 0;

    // carry propagates from lower-order BCD digit to upper-order BCD digit
    if( upper < 9 ) upper++;
    else upper = 0;
}

B = (upper << 4) | lower;
```

PES, Section 2.7
Bit access

1. What is the output value (in hexadecimal) of the following sequence of operations?

```
unsigned char x = 0x33;        // 0011 0011
x = SetBit(x, 1, 0);           // 0011 0001
x = x << 2;                    // 1100 0100
x = SetBit(x, 4, 1);           // 1101 0100
x = SetBit(x, 3, 0);           // 1101 0100
x = x >> 1;                    // 0110 1010
x = SetBit(x, 2, 1);           // 0110 1110
```

2. What is the value of B after executing each of the following two statements? Assume that the input provided to A is the constant value 10, and that the value does not change.

```
B = (a == 1) ? 20 : 30;                 B = 30

B = (a == 10) ? 20 : 30;                B = 20
```

3. Write a RIMS-compatible C-language for-loop that sets B to the reverse of A, in other words B7 = A0, B6 = A1, etc.

```
unsigned char i;
unsigned char tmp = 0;
for(i = 0; i < 8; i++)
    tmp = SetBit(tmp, i, GetBit(A, 7-i));
B = tmp;
```

4. Write a RIMS-compatible C-language for-loop that counts the number of times a bit of A is **followed** by a bit of the opposite parity (01 or 10) and writes the value to B. For example 00100110 has 4 cases: 00100110, 00100110, 00100110, 00100110.

```
unsigned char i;
unsigned char cnt = 0;
for(i = 0; i < 7; i++)
    cnt = (GetBit(A, i) ^ GetBit(A, i+1)) ? cnt+1 : cnt;
B = cnt;
```

Note: Be careful with the loop bounds for counter i. It would be an error to call GetBit(A, -1) or GetBit(A, 8).

PES, Section 2.8
Rounding and Overflow

1. The following program was used in Section 2.8 of PES to illustrate the problem of rounding, which occurs in Fahrenheit-to-Celsius conversion:

```
#include "RIMS.h"
unsigned char C2F_uc(unsigned char C) {
   unsigned char F;
   F = (9/5)*C + 32;
   return F;
}
void main() {
   while (1) {
      B = C2F_uc(A);
   }
}
```

The following Table shows the actual Fahrenheit values (real numbers), the closest integer approximation (i.e., round-up, round-down), the results computed by the above program, and a variation that performs division later: F = (9*C)/5 + 32.

Celsius	Fahrenheit (actual)	Fahrenheit (integer)	Fahrenheit (from above program)	Fahrenheit (late division)
0	32	32	32	32
1	33.8	34	33	33
2	35.6	36	34	35
3	37.4	37	35	37
4	39.2	39	36	39
5	41	41	37	41
6	42.8	43	38	42
7	44.8	45	39	44
8	46.4	46	40	46
9	48.2	48	41	48

Rewrite the function C2F_uc to compute the values in the table listed as Fahrenheit (integer).

```
unsigned char C2F_uc(unsigned char C) {
   unsigned char F;
   F = (9*C)/5 + 32;
   if( (9*C)%5 >= 3)
      F++;
   return F;
}
```

2. In the preceding example, we replaced F = (9/5)*C + 32 with F= (9*C)/5 + 32. Making this change could introduce a new type of error that would not have occurred originally. What is the error?

   9*C could overflow, depending on the value of C.

   (9/5)*C could not overflow, because (9/5) simplifies to 1, and 1*C does not overflow (presuming that C is an unsigned char to begin with).

3. When computing (a + b + c)/3, the possibility of integer overflow occurs. One proposed remedy was to compute the division earlier, i.e., (a/3) + (b/3) + (c/3). Give an example, assuming that overflow does not occur, where (a + b + c)/3 computes a more accurate result than (a/3) + (b/3) + (c/3).

   a = b = c = 1

   (a + b + c)/3 = (1 + 1 + 1)/3 = 3/3 = 1

   (a/3) + (b/3) + (c/3) = (1/3) + (1/3) + (1/3) = 0 + 0 + 0 = 0