CMPU 224 Quiz 1 Practice Problems

1. Perform the following number conversions:

   A. `0x48EA034D` to binary: **`100 1000 1110 1010 0000 0011 0100 1101`**

   B. binary `1 0001 0111 0010 1011 0101 0101` to hexadecimal: 0x**`1172B55`**

   C. `0xD4317` to binary: **`11010100001100010111`**

   D. binary `0010110110001110` to hexadecimal: **`2D8E`**

   E. decimal `94` to binary: **`1011110`**

   F. binary `01001101` to decimal: **`77`**

2. Fill in the blanks in the following table. Assume all numbers are **unsigned**.

Decimal	**Hexadecimal**	**Binary**
`455`	**`0x1c7`**	**`111000111`**
`213`	`0xd5`	**`11010101`**
`335`	**`0x14f`**	`101001111`
`159`	**`0x9f`**	**`10011111`**
`282`	`0x11a`	**`100011010`**
`153`	**`0x99`**	`10011001`

3. A single byte can be represented by 2 hexadecimal digits. Fill in the missing entries in the following table, giving the decimal, binary, and hexadecimal values of different byte patterns.

Decimal	Binary	Hexadecimal
0	0000 0000	0x00
167	**1010 0111**	**0xA7**
62	**0011 1110**	**0x3E**
188	**1011 1100**	**0xBC**
55	0011 0111	**0x37**
136	1000 1000	**0x88**
243	**1111 0011**	0xF3

4. Assume we are running code on a 6-bit machine using two's complement arithmetic for signed integers. Values of type `int` and `unsigned int` are represented using 6 bits. `TMax` is the largest integer represented in this system and `TMin` represents the smallest valued integer in the system. Fill in the empty boxes in the table below. The following definitions are used in the table below. Note: You do not need to fill in entries marked with "-".

```
int x = -15;
unsigned ux = x;
```

Expression	Decimal Representation	Binary Representation
Zero	0	**00 0000**
-	-6	**11 1010**
-	-10	**11 0110**
-	**20**	01 0100
-	**-26**	10 0110
ux	**49**	**11 0001**
x << 4	**16**	**01 0000**
x >> 1	**-8**	**11 1000**
TMax	**31**	**01 1111**
TMin	**-32**	**10 0000**
!x	**0**	**00 0000**
~x	**14**	**00 1110**

5. Assume we are running code on a **8-bit** machine using two's complement arithmetic for signed integers. Values of type `int` and `unsigned int` are represented using **8 bits**. `TMax` is the largest integer represented in this system and `TMin` represents the smallest valued integer in the system. Fill in the empty boxes in the table below. The following definitions are used in the table below.

```
int x = -42;
unsigned int ux = x;
```

Expression	Decimal Representation	Binary Representation
Zero	**0**	**0000 0000**
TMax	**127**	**0111 1111**
TMin	**-128**	**1000 0000**
x	**-42**	**1101 0110**
ux	**214**	**1101 0110**
x << 3	**-80**	**1011 0000**
x >> 3	**-6**	**1111 1010**
ux >> 2	**53**	**0011 0101**
!x	**0**	**0000 0000**
!!x	**1**	**0000 0001**
-x	**42**	**0010 1010**
~x	**41**	**0010 1001**

6. Suppose we have the following bytes in memory:

Mem addr:	0x200	0x201	0x202	0x203	0x204
values:	0x03	0xF1	0x0A	0x55	0xDA

And the following C code:

```
char *cp = 0x200;
short *sp = 0x200;
int *ip = 0x200;
```

What are the values of `*cp`, `*sp`, and `*ip` if the system is a little-endian Linux system?

`*cp` = **0x3**
`*sp` = **0xF103**
`*ip` = **0x550AF103**

What would the values be on a big-endian system?

`*cp` = **0x3**
`*sp` = **0x03F1**
`*ip` = **0x03F10A55**

7. Suppose that `x` and `y` have byte values of `0x66` and `0x39`, respectively. Fill in the following table indicating the byte values of the different C expressions:

Expression	**Value**
`x & y`	**0x20**
`x | y`	**0x7F**
`~x | ~y`	**0xDF**
`x & !y`	**0x00**
`x && y`	**0x01**
`x || y`	**0x01**
`!x || !y`	**0x00**
`x && ~y`	**0x01**

8. Fill in the table below showing the effects of the different shift operations on single-byte quantities. The best way to think about shift operations is to work with binary representations. Convert the initial values to binary, perform the shifts, and then convert back to hexadecimal. Each of the answers should be 8 binary digits or two hexadecimal digits.

x		x << 3		Logical x >> 2		Arithmetic x >> 2	
Hex	**Binary**	**Binary**	**Hex**	**Binary**	**Hex**	**Binary**	**Hex**
0xC3	**1100 0011**	**0001 1000**	**0x18**	**0011 0000**	**0x30**	**1111 0000**	**0xF0**
0x75	**0111 0101**	**1010 1000**	**0xA8**	**0001 1101**	**0x1D**	**0001 1101**	**0x1D**
0x87	**1000 0111**	**0011 1000**	**0x38**	**0010 0001**	**0x21**	**1110 0001**	**0xE1**
0x66	**0110 0110**	**0011 0000**	**0x30**	**0001 1001**	**0x19**	**0001 1001**	**0x19**

9. Assume we are representing numbers as 4-bit 2's complement numbers. Fill in the table below to determine the inverse of the given number.

x		-x	
Hex	**Decimal**	**Decimal**	**Hex**
0	**0**	**0**	**0**
5	**5**	**-5**	**B**
8	**-8**	**-8**	**8**
D	**-3**	**3**	**3**
F	**-1**	**1**	**1**

10. For this problem you'll solve a puzzle by writing a small C function. You will implement `getHex()` which extracts hex digit `n` from word `x`. Hex digits are numbered from 0 (Least Significant Hex Digit) to 7 (Most Significant Hex Digit). You can assume `n` will only have the values from 0 to 7.

Examples:

`getHex(0x12345678,1)` returns `7`
`getHex(0xabcdef01,7)` returns `10 (0xa)`

Legal operations: `! ~ & ^ | + << >>` and any numeric constants from 0 to 255. (0x00 to 0xFF).

```
int getHex(int x, int n) {

    int mask = 0xf; mask for the hex digit
    int shift = n << 2; // how much to shift to get our hex digit to
                        // the low-order 4 bits (lowest hex digit)
    return (x >> shift) & mask;
}
```

11. Write the following function, `swapNybble()`, which takes an `unsigned char` as input returns an `unsigned char` that has the lower 4-bits swapped with the upper 4 bits of the input. For example, `swapNybble(0xAB)` would return `0xBA`.

```
unsigned char swapNybble(unsigned char c){
   // Returns an unsigned char with the upper 4 bits swapped with
   // the lower 4 bits.

   unsigned char lower = c >> 4 & 0xf
   unsigned char upper = c << 4
   return upper | lower
}
```

12. Write the following function, `clearByte()`, which clears (sets all the bits to zero) all the bits of a byte given by `n` from word `x`. Bytes are numbered from 0 (Least Significant Byte) to 3 (Most Significant Byte). You can assume `n` will only have the values from 0 to 3.

Examples:

`clearByte(0x12345678,1)` returns `0x12340078`
`clearByte(0xabcdef01,3)` returns `0x00cdef01`

Legal operations: `! ~ & ^ | + << >>` and any numeric constants from 0 to 255. (0x00 to 0xFF).

```
int clearByte(int x, int n) {

// To clear the byte we need a mask of all zeros in the byte we want to
    // clear, with all ones for the other bytes.

    // To get the mask, shift in eight ones to the correct bit position and
    // then flip all the bits.  To get the correct shift amount we need to
    // multiply the input n (the byte number) by 8.  We can do this by
    // shifting n by 3, which is the same as n*8.

    int shift = n << 3;

    // Now create the mask by shifting all ones (0xFF) into the correct
    // position and flipping all the bits.

    int mask = ~(0xFF << shift);

    // Now all we have to do is AND the mask with x to get our answer.

    return x & mask;

}
```

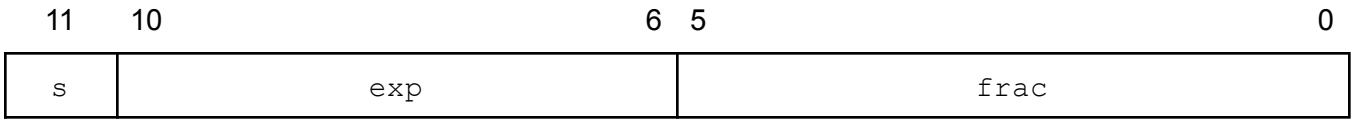
13. Fill in the missing information in the following table:

Fractional value	**Binary representation**	**Decimal representation**
3/4	**0.11**	**0.75**
5/16	**0.0101**	**0.3125**
2 11/16	10.1011	**2.6875**
1 1/8	1.001	**1.125**
5 7/8	**101.111**	5.875
3 3/16	**11.0011**	3.1875

14. Show how the following binary fractional values would be rounded to the nearest half (1 bit to the right of the binary point), according to the round-to-even rule. In each case, show the numeric values both before and after rounding.

Binary	Value	Rounded Binary	Rounded Value
110.010 → **Even round down**	**6.25**	**110.0**	**6.0**
110.011 **Round up**	**6.375**	**110.1**	**6.5**
110.110 → **Even round up**	**6.75**	**111.0**	**7.0**
111.001 **Round down**	**7.125**	**111.0**	**7.0**

15. Recall that floating point numbers are represented in the form $V = (-1)^s \times M \times 2^E$ where M is encoded in `frac`, and E is encoded in `exp`. Assume we have the following tiny **12-bit** floating point representation where bits 0-5 represent the fraction field, bits 6-10 represent the exponent field, and bit 11 represents the sign bit, as shown below.



a. What is the floating point representation of the number 1.0? Write out the binary bits in the boxes below.

11	10	9	8	7	6	5	4	3	2	1	0
0	**0**	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**0**	**0**	**0**

b. What is the Bias (used to calculate E) for this floating point number system? **$2^{(5-1)}-1 = 15$**

c. An `exp` field of all ones and a `frac` field of not equal to zero represents what type of number? **NaN (Not a Number)**

d. An `exp` field of all zeros represents what? **A denormalized number**

e. If `frac` has the value of `101110` for a normalized value, what is the value of `M` (in binary)?

Since the number is normalized, M is the frac with a leading 1.

1.101110

f. Write down the floating point representation of the following binary number with the representation system described above.

Take the binary representation and seperate into the three parts: s, exp, and frac. s = 0, so it is a positive number. Exp has the value of 20. With the bias of 15, E is 20-15 = 5. M is frac with a leading 1, or 1.001001. E is 5, so we move the binary point 5 places to the right: 100100.1 giving us 36.5 as the answer.

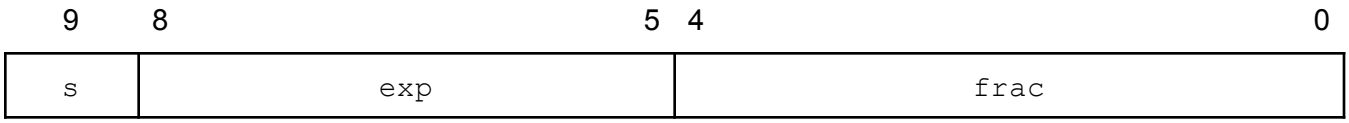
Binary	**Floating Point**
0 10100 001001	**36.5**

g. Write down the binary representation of the following floating point number with the representation described above.

We need to determine the values for s, exp, and frac. The floating point number is negative, so s is 1. To determine exp, and frac, first write out the fraction in binary: 1010.11 which we will then encode. Take that fraction and shift the binary point till we have a leading one, 1.01011 (done by shifting the binary point 3 places). Frac is the fractional part of that number, 010110. E is 3 as we need to shift M 3 places to the right to give us our number (1010.11), therefore exp is 15 + 3 = 18 as we have to add the bias to E. 18 is 10010, which is the value of frac. Putting it all together gives us the binary representation shown below.

Binary	Floating Point
1 10010 010110	-10.75

16. Recall that floating point numbers are represented in the form $V = (-1)^s \times M \times 2^E$ where M is encoded in `frac`, and E is encoded in `exp`. Assume we have the following tiny **10-bit** floating point representation where bits 0-4 represent the fraction field, bits 5-8 represent the exponent field, and bit 9 represents the sign bit, as shown below.



Write down the floating point representation of the following binary number with the representation system described above.

Binary	**Floating Point**
1101101110	**-23.0**

MSB is 1, so the number is negative.

exp = 1011 ⇒ 11
bias = 7
E = exp - bias = 4

frac = 01110 ⇒ M = 1.01110

V = 10111.0 ⇒ -23.0

Write down the binary representation of the following floating point number with the representation described above.

Binary	**Floating Point**
0 1001 11001	7.125

MSB is 0 (positive number)

7.125 = 111.001
M = 1.11001
frac = 11001
E = 2 ⇒ exp = 9 ⇒ 1001

17. Assume the following values are stored at the indicated memory addresses and registers:

Address	Value
0x100	0xFF
0x108	0xAB
0x110	0x13
0x118	0x11

Register	Value
%rax	0x100
%rcx	0x1
%rdx	0x3

Fill in the following table showing the values for the indicated operands:

Operand	Value
%rax	**0x100**
0x108	**0xAB**
$0x108	**0x108**
(%rax)	**0xFF**
8(%rax)	**0xAB**
0xD(%rax, %rdx)	**0x13**
260(%rcx, %rdx)	**0xAB**
0xFC(,%rcx, 4)	**0xFF**
(%rax, %rdx, 8)	**0x11**

18. Suppose `%rax` has the value of 42, `%rcx` has the value 5, indicate the value that will be stored in register `%rdx` for each of the instructions:

```
leaq -5(%rax), %rdx              37
leaq 0x45, %rdx                  69
leaq (%rax, %rcx), %rdx          47
leaq (%rax, %rcx, 2), %rdx       52
leaq 0x21(%rcx, %rcx, 1), %rdx   43
```