Assignment 0 Solutions

1. Base systems (10 pts)

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>10</td>
<td>0x2</td>
</tr>
<tr>
<td>10</td>
<td>1010</td>
<td>0xA</td>
</tr>
<tr>
<td>16</td>
<td>1 0000</td>
<td>0x10</td>
</tr>
<tr>
<td>42</td>
<td>10 1010</td>
<td>0x2A</td>
</tr>
<tr>
<td>4096</td>
<td>1 0000 0000 0000</td>
<td>0x1000</td>
</tr>
<tr>
<td>3</td>
<td>11₂</td>
<td>0x3</td>
</tr>
<tr>
<td>5</td>
<td>101₂</td>
<td>0x5</td>
</tr>
<tr>
<td>21</td>
<td>10101₂</td>
<td>0x15</td>
</tr>
<tr>
<td>19</td>
<td>10011</td>
<td>0x13</td>
</tr>
<tr>
<td>63</td>
<td>111111</td>
<td>0x3F</td>
</tr>
<tr>
<td>4294967295</td>
<td>1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111</td>
<td>0xFFFFFFFF</td>
</tr>
<tr>
<td>3735928559</td>
<td>1101 1110 1010 1101 1011 1110 1110 1110 1111</td>
<td>0xDEADBEEF</td>
</tr>
</tbody>
</table>

2. Computer arithmetic (20 pts)

Part A (0 pts)
Example Solutions:
1. 33
2. 35
3. 46
4. 1 (any number is correct)
5. 3014
6. 3
7. 2, 9
8. 54, 4
9. 55 (any nonzero number is correct)
10. 42139
11. 0, 0, 0

Explanations by number:

1. ________ & 57 == 33
2. ________ & 57 == 33

First, let's figure out the binary representation of each 33 and 57:
\[
57 = 2^5 + 2^4 + 2^3 + 2^0 = 111001 \\
33 = 2^5 + 2^0 = 100001
\]

We are using \&, which is bitwise AND, and we want to solve for the blank, represented by xxxxxx here:

\[
\begin{array}{c}
\text{xxxxxx} \\
\text{& 111001} \\
\text{-------} \\
\text{100001}
\end{array}
\]

The solution can be anything that has the 1st (least significant) and 6th bit set to 1, and the 4th and 5th bit set to 0. Here are a few examples of possible solutions:
\[
100001 = 33 \\
100111 = 39
\]

Answer: ...xxxxxxxxxxx100xx1 where x can be 1 or 0!

3. ________ ^ 46 == 0

Bitwise XOR flips bits in its input. So, if you bitwise XOR some number with its own inverse, you get 0.

Answer: 46
4. \[ \quad | \quad \neg 0 \quad = \quad 4294967295 \]

\(\neg 0\) (not 0) is really \((\neg 0 \cdot 0000 0000 \rightarrow 1\ldots 1111 1111) = (2^{32} - 1). 4294967295\) is equal to \((2^{32} - 1)\) as well. Since we’re working with bitwise OR, and every bit in the second operand \((\neg 0)\) is set to 1, the right hand side is always \((2^{32} - 1)\) regardless of what the first operand (the blank) is.

**Answer:** any number works, since applying OR to anything and 1 is 1.

5. \[ \quad - \quad 100000 \quad = \quad 429470310 \]

If we use regular old math to figure out what goes in the blank, we would find that the blank would be filled with the number 429470310. However, 429470310 does not fit into a 32 bit integer. The most significant bit (the leftmost bit) gets truncated, and the leftover bits equal 3014 in decimal.

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>4294970310</td>
<td>0000 0000 0000 0000 0000 1011 1100 0110</td>
<td>33</td>
</tr>
<tr>
<td>3014</td>
<td>0000 0000 0000 0000 0000 1011 1100 0110</td>
<td>32</td>
</tr>
</tbody>
</table>

This is equivalent to subtracting \(2^{32}\) from 429470310, which also gives us 3014.

**Answer:** 3014

6. \[ 15 \quad >> \quad \quad = \quad 1 \]

We are using bitwise left shift here. Since our goal is to only have the least significant (leftmost) bit set to 1, we will need to shift 15 to the left by 3.

\[
15 = 0000 \ldots 0000 1111
\]

\[
1 = 0000 \ldots 0000 0001
\]

**Answer:** 3

7. \[ \quad << \quad \quad = \quad 1024 \]

1024 = 100 0000 0000 in binary

We are using bitwise right shift here. Our goal is to have the 11th bit set to 1 using right shift. There are a few combinations you could put in the blanks to fulfill this equation:

\[
1 << 10 \quad \text{(Shift 1 to the left 10 spots)}
\]

\[
2 << 9 \quad \text{(Shift 10 in binary to the left 9 spots)}
\]

\[
4 << 8 \quad \text{(Shift 100 in binary to the right 8 spots)}
\]

and so on.
Answer: 1 << 10, or 2 << 9, or 4 << 8, etc. You could also pick numbers that shifted some bits off the top, such as 2147483649 << 10.

8. \[ \text{________} + \sim \text{________} = 49 \]

Answer: One easy answer is \[49 + \sim 4294967295 = 49, \text{since} \sim 4294967295 = 0.\]

Many other answers are possible too. For example:
\[
\begin{align*}
50 + \sim 0 & = 49 \\
51 + \sim 1 & = 49 \\
54 + \sim 4 & = 49 \\
\end{align*}
\]

and so on.

9. \[(\text{________} \text{&& 42139}) = 1\]

We are using the logical AND operator, meaning we want to treat the operands as booleans. In C, any nonzero number is considered TRUE, and zero is considered FALSE. Thus, any number other than 0 should satisfy the equation.

Answer: Any nonzero number.

10. \[(\text{________} = 42139) = 1\]

In C, the == operator will evaluate to 1 (equivalent to TRUE) only if the both values are equal; otherwise the expression will evaluate to 0, or FALSE. Thus, \[42139 = 42139\] evaluates to 1, which is equal to the right hand side.

Answer: 42139

11. Use the same number for all three blanks: (It should differ from all numbers above)
\[\text{________} + \sim \text{________} + 1 = \text{________}\]

Answer: 0 + \sim 0 + 1 = 0

This is the only possible answer. \sim x\ has 1 bit exactly where \(x\) has 0 bits, and vice versa. So we can see that adding \(x + \sim x\) cannot cause a carry at any bit position. This means that \(x + \sim x\) equals \(\sim 0 = 4294967295\), no matter what \(x\) is. Because \(4294967295 + 1 = 0\) in computer arithmetic, all three blanks must be filled in with 0.

Part B (5 pts)
What are the unique solutions?

Answer: 3, 5, 6, 10, 11
Part C (5 pts)
Which operators are not associative?

-, /, %, ==, !=, <, >, <=, >=, <<, >>

3. Base -2 encoding (20 pts)

Part A (12 pts)

-8: 1000     +8: 11000
-7: 1001     +7: 11011
-6: 1110     +6: 11010
-5: 1111     +5: 101
-4: 1100     +4: 100
-3: 1101     +3: 111
-2: 10       +2: 110
-1: 11       +1: 1
0: 0

Part B (4 pts per sum)

max
The max is the sum of all the even bits:
\[ \text{max} = (-2)^0 + (-2)^2 + ... + (-2)^{2n - 2} \]

This series can be reduced to
\[ \text{max} = \sum_{i=0}^{n-1} 4^i \]
\[ = \frac{4^n - 1}{3} \]

min
The min is the sum of all the odd bits:
\[ \text{min} = (-2)^1 + (-2)^3 + ... (-2)^{2n - 1} \]

This is equivalent to just multiplying the max by (-2):
\[ \text{min} = -2 * \sum_{i=0}^{n-1} 4^i \]
\[ = -1 * \sum_{i=0}^{n-1} 2^{2i + 1} \]
\[ = -2 * \frac{4^n - 1}{3} \]
4. C programming (20 pts)

Part A (4 pts per blank)
Let the binary representation of $x$ be represented by “abcdefgh”, where each character (eg, a and b) represents a bit which is either 1 or 0.

The idea then is to solve for these characters in the following equation:

$$128a + 64b + 32c + 16d + 8e + 4f + 2g + h = x$$

To test whether a character, or bit, is set to 1 or 0, you can check whether or not $x$ is greater than or equal to the coefficient of that character. If it is, the bit is set to 1, and we should subtract out the coefficient from $x$. To move to the next coefficient, we divide by 2. This continues until we run out of coefficients.

In the program, you can imagine that we are testing one character at a time from left to right. We store the current coefficient in the variable $b$, so $b$ initially equals 128, the first coefficient. Once we have determined whether the a bit is set, we move on to the next character and also to the next coefficient by dividing by 2 to get 64.

First blank: $b > 0$
Second blank: $b = b/2$;

Part B (12 pts)
Basically anything can go here, as long as some part of the logic changed. Here are a few examples:

```c
void print_binary2(unsigned char x) {
    int b = 128;
    while (b > 0) {
        if (x & b)
            printf("1");
        else
            printf("0");
        b = b >> 1;
    }
}

void print_binary2(unsigned char x) {
    int bits[8];
    int b = 1;
    for (int i = 7; i >= 0; i--) {
```
bits[i] = (x & b) ? 1 : 0;
b <<= 1;
}

for (int i = 0; i < 8; i++)
    printf("%d", bits[i]);
}

5. Additional C self-assessment (not graded)

Part A
Answer: 10

Part B
Answer: 1 + (2 * 4) + (8 / 4) = 11

Part C
Answer:
- i should start at 0, not 1.
- arr is dereferenced twice, since arr[i] is dereferenced. It should just be arr[i].
- There are should be a break statement in each case, including the default. Because there are no break statements, more things are printed than desired. For example, if arr[i] is 0, then "ZERO\nONE\nNONE\n" is printed.

Part D
#include <string.h>
void underscoresize(char *dst, const char *src) {
    int len = strlen(src);
    for (int i = 0; i <= len; i++) {
        if (src[i] == ' ')
            dst[i] = '_';
        else
            dst[i] = src[i];
    }
}
Part E

```c
struct node *remove_nth(struct node *head, int n) {
    // check for invalid lists or n
    if (head == NULL || n < 0)
        return head;

    // go to the node to delete
    struct node *currNode = head;
    int i = 0;
    while (i < n) {
        if (currNode == NULL)
            return head;
        currNode = head->next;
        i++;
    }

    // adjust the links
    if (currNode->next != NULL)
        (currNode->next)->prev = (currNode->prev);
    if (currNode->prev != NULL)
        (currNode->prev)->next = (currNode->next);

    // we’re deleting the head, mark the next node as the head
    if (i == 0)
        head = nextNode;
    return head;
}
```

Part F

**Answer:** An input of 1 causes an overflow in the variable a. For example, when a equals 1, then the program as is would return `fib(0) + fib(-1)`, which is not correct. Even worse, a is an unsigned char, so the -1 would just wrap around to the largest number. Here’s a possible fix:

```c
unsigned fib(unsigned a) {
    if (a == 0) // a is unsigned, so it is never less than 0
        return 1;
    else if (a == 1)
        return 1;
    else
        return fib(a - 1) + fib(a - 2);
}
```