CMSC B240 Computer Organization - Spring 2024
Lab Activity #2 Solutions

Question #1

Calculate the sums of the following unsigned binary integers:

1.1) 1101 + 1001

```
Carry:     1
         1 1 0 1
       + 1 0 0 1
       -------
       1 0 1 1 0
```

1.2) 10101 + 110

Think about how you can check that your answers are correct!

```
Carry:   1
         1 0 1 0 1
       +     1 1 0
       -----------
         1 1 0 1 1
```

In both cases, we can check that the solution is correct by converting to decimal:

- 1101 + 1001 → 13 + 9 = 22 → 10110
- 10101 + 110 → 21 + 6 = 27 → 11011

Question #2

Calculate the sums of the following 8-bit 2's-complement binary numbers; the results should also be expressed as 8-bit 2's-complement binary numbers:

2.1) 00010101 + 01001111

```
Carry:     1 1 1 1 1
         0 0 0 1 0 1 0 1
       + 0 1 0 0 1 1 1 1
       -----------------
         0 1 1 0 0 1 0 0
```

2.2) 00110000 + 01100001

```
Carry: 1 1
         0 0 1 1 0 0 0 0
       + 0 1 1 0 0 0 0 1
       -----------------
         1 0 0 1 0 0 0 1
```

2.3) 11110011 + 11111010

```
Carry: 1 1 1       1
         1 1 1 1 0 0 1 1
       + 1 1 1 1 1 0 1 0
       -----------------
       1 1 1 1 0 1 1 0 1
```

Here, we throw away the carry-out bit and get **1110 1101**.

2.4) In which cases did overflow occur? How could you tell?
Overflow occurred in the second one because we added two positive numbers (starting with 0) and the result was a negative number (starting with 1).

In this case we added 48 + 97. The sum would be 145, but the largest number we can represent is 01111111, which is 127, hence we have overflow.

It is tempting to say that we had overflow in the third one because of the carry-out bit. However, this is not overflow because we added two negative numbers and the sum was also negative: the first number is -13 (see below), the second is -6, and the sum is -19.

11110011 → subtract 1 → 11110010 → flip → 00001101 → -13
11111010 → subtract 1 → 11111001 → flip → 00000110 → -6
11101101 → subtract 1 → 11101100 → flip → 00010011 → -19

Question #3
In this question, you will see how you can use different techniques to perform subtraction using 8-bit 2's-complement binary numbers. In particular, we want to show how to calculate 27 - 15.

3.1) First, show how you can do this by performing binary subtraction on the two numbers, borrowing from columns to the left as needed:

```
    0 0 0 1 1 0 1 1
-   0 0 0 0 1 1 1 1
    ---------------
    0 0 0 0 1 1 0 0
```

Check that your result is correct! This is 8 + 4 = 12. Yeah!

3.2) Now calculate 27 - 15 as 27 + (-15) by first determining the binary representation of -15 and adding it to 27. You should get the same result as in Q3.1!

+15 is 00001111, so to get -15:
Flip → 11110000
Add 1 → 11110001

Now add:

```
Carry   1 1 1 1       1 1
          0 0 0 1 1 0 1 1
      +   1 1 1 1 0 0 0 1
        -----------------
        1 0 0 0 0 1 1 0 0
```

We get rid of the carry-out bit and have 0000 1100, which is 12. Yeah!

Question #4

Compute the following:

4.1) (0110 OR 0000) AND 1111

- 0110 OR 0000 → 0110
- 0110 AND 1111 → **0110**

4.2) NOT(1001 AND (1100 OR 0100))

We need to start with the inner parentheses:

- 1100 OR 0100 → 1100
- 1001 AND 1100 → 1000
- NOT 1000 → **0111**

Question #5

What would you expect to be printed by each of the following portions of Java code? Don't actually program it! Figure it out by hand!

5.1)

```
int x = 72;
int y = 40;
int z = 30;
System.out.println( (x & y) ^ z );
```

Let's first convert the three variables to binary:

- x = 72 → 1001000
- y = 40 → 101000
- z = 30 → 11110

Then x & y is (x AND y):

1 0 0 1 0 0 0
0 1 0 1 0 0 0
0 0 0 1 0 0 0

Then we do XOR with z:

0 0 0 1 0 0 0
0 0 1 1 1 1 0
0 0 1 0 1 1 0

This produces **22**.

5.2)

```
int a = 31;
int b = -31;
System.out.println( a & b );
```

- a = 31 = 11111
- b = -31 → start with 11111 → flip 00000 → add 1 → 00001
- 11111 & 00001 = **00001**
- A number "and" its negation will always be 1 in two's-complement

Question #6

Convert the following unsigned binary numbers to hexadecimal.

6.1) 10010011

10010011 → 1001 0011 → **x93**

6.2) 1101000110101111

1101000110101111 → 1101 0001 1010 1111 → **xD1AF**

6.3) 11110

11110 → 0001 1110 → **x1E**

Note that we need to put leading 0s in front of the number so that we have 8 digits.

Question #7

Convert the following hexadecimal numbers to binary.

7.1) xABC

xABC → 1010 1011 1100 → **101010111100**

7.2) x10

x10 → 0001 0000 → 00010000 or just **10000**

Question #8

Convert the following decimal numbers to hexadecimal. See if you can figure out how to do this without first converting to binary!

It's possible to first convert to binary, of course, but we can convert to hexadecimal using the same approach as we did for converting to binary, but by using "% 16" and "/ 16" instead of "% 2" and "/ 2".

8.1) 82

82 % 16 = 2 → _ 2
82 / 16 = 5
5 % 16 = 5 → 5 2
5 / 16 = 0 → done
So the solution is **x52**.

8.2) 171

171 % 16 = 11 → _ B
171 / 16 = 10
10 % 16 = 10 → A B
10 / 16 = 0 → done
So the solution is **xAB**.

Question #9

Express the decimal value 78.5 using the IEEE 32-bit floating point representation, i.e. the representation used by the Java "float" datatype. Write your answer in hexadecimal.

The first thing we need to do is convert 78.5 to binary using scientific notation. We can do the 78 part using the approach we've seen before:

78 % 2 = 0 → _ _ _ _ _ _ 0
78 / 2 = 39
39 % 2 = 1 → _ _ _ _ _ 1 0
39 / 2 = 19
19 % 2 = 1 → _ _ _ _ 1 1 0
19 / 2 = 9
9 % 2 = 1 → _ _ _ 1 1 1 0
9 / 2 = 4
4 % 2 = 0 → _ _ 0 1 1 1 0
4 / 2 = 2
2 % 2 = 0 → _ 0 0 1 1 1 0
2 / 2 = 1
1 % 2 = 1 → 1 0 0 1 1 1 0

We never saw how to convert the fractional part of a decimal number to binary, but 0.5 is of course ½ = 2^{-1} so we know that's 0.1.

Thus, 78.5 can be represented as 1001110.1.

But we're not done yet because we need to put this in the IEEE floating-point format.

First, we rewrite 1001110.1 as $1.0011101 * 2^6$.

Then we can convert this to the IEEE format:

- The first bit is 0 since the number is positive
- The next eight bits are for the exponent part, which is 127 + 6 = 133, which is represented as 10000101 (left as an exercise for the reader!)
- The remaining 23 bits are for the fraction part, which is 0011101 followed by 16 zeroes

Thus, the binary representation is **0 10000101 00111010000000000000000**.

But we're *still* not done because the question asks us to write this in hexadecimal, which we can do by converting each group of four binary digits.

0 10000101 00111010000000000000000 → 0100 0010 1001 1101 0000 0000 0000 0000
4 2 9 D 0 0 0 0

So the answer is **x429D0000**

Question #10

In addition to representing integers and floating point numbers in binary, we can also represent single characters such as ‘a’, ‘b’, ‘A’, ‘B’, ‘?’, etc.

A common representation is **ASCII**, in which each character has a unique 8-bit value.

A nice aspect of ASCII is that the letters of the alphabet are grouped together:

- Uppercase ‘A’ is represented using 01000001 = x41 = 65
- Uppercase ‘B’ is one greater than ‘A’, i.e. its representation is 01000010 = x42 = 66
- Uppercase ‘C’ is one greater than ‘B’, i.e. its representation is 01000011 = x43 = 67

and so on. The same applies to lowercase letters as well. You can find ASCII tables at the back of your textbook or online.

The Java programming language uses **Unicode** for representing characters, which is a 16-bit representation. ASCII representations are kept the same in Unicode, e.g. ‘A’ is x0041 in Unicode, and the letters of the alphabet are still grouped together, e.g. ‘B’ is x0042 and so on.

Based on this, what do you think is printed out by each of the following lines of Java code? Don’t actually program it! Figure it out by hand!

10.1)

```
System.out.println((char)('A' + 1));
```

Java will treat ‘a’ as the number x41 = 65, so when we add 1, we get x42 or 66. We can then cast it to a char, and it will use the value x42 or 66 to get the corresponding Unicode character, which is **‘B’**.

10.2)

```
System.out.println((int)('A' + 1));
```

Again Java will treat ‘A’ as the number 65, so when we add 1, we get 66. But now we’re going to print it as an int, so it prints **66**. In fact, we don’t need to cast it, but it’s better to make it clear.

10.3)

```
System.out.println((int)('A' + 'B'));
```

Now Java will treat ‘A’ as the number 65, and we’ve seen that ‘b’ is 66, so the sum is **131**, which is printed as an int.

10.4)

```
System.out.println((char)('M' - 5));
```

We can do subtraction with chars as well. You could solve this by figuring out that ‘M’ is 77, and that 77 - 5 = 72, which is the encoding of ‘H’. Or you can do some alphabet magic and realize that ‘H’ is five letters before ‘M’!