**CSE 351 Section 2 – Pointers, Bit Operators, Integers**

**Pointers**

A pointer is a variable that holds an address. C uses pointers explicitly. If we have a variable `x`, then `&x` gives the address of `x` rather than the value of `x`. If we have a pointer `p`, then `*p` gives us the value that `p` points to, rather than the value of `p`.

Consider the following declarations and assignments:

```c
int x;
int *ptr;
ptr = &x;
```

1) We can represent the result of these three lines of code visually as shown.
   The variable `ptr` stores the address of `x`, and we say “`ptr` points to `x`.”
   `x` currently doesn’t contain a value since we did not assign `x` a value!

2) After executing `x = 5;`, the memory diagram changes as shown.
3) After executing `*ptr = 200;`, the memory diagram changes as shown.
   We modified the value of `x` by dereferencing `ptr`.

**Pointer Arithmetic**

In C, arithmetic on pointers (`++`, `+`, `--`, `-`) is scaled by the size of the data type the pointer points to. That is, if `p` is declared with pointer `type* p`, then `p + i` will change the value of `p` (an address) by `i*sizeof(type)` (in bytes). If there is a line `*p = *p + 1`, regular arithmetic will apply unless `*p` is also a pointer datatype.

**Exercise:**

Draw out the memory diagram after sequential execution of each of the lines below:

```c
int main(int argc, char **argv) {
    int x = 410, y = 350;   // assume &x = 0x10, &y = 0x14
    int *p = &x;            // p is a pointer to an integer
    *p = y;
    p = p + 4;
    p = &y;
    x = *p + 1;
}
```

<table>
<thead>
<tr>
<th>Line 1:</th>
<th>Line 2:</th>
<th>Line 3:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line 4:</td>
<td>Line 5:</td>
<td>Line 6:</td>
</tr>
</tbody>
</table>
C Bitwise Operators

| & | 0 1 | ← AND (\&) outputs a 1 only when both input bits are 1. | 1 | 0 1 |
|---|---|---|---|
| 0 | 0 0 | 0 | 0 1 |
| 1 | 0 1 | OR (|) outputs a 1 when either input bit is 1. | → | 1 1 1 |

<table>
<thead>
<tr>
<th>^</th>
<th>0 1</th>
<th>← XOR (^) outputs a 1 when either input is exclusively 1.</th>
<th>~</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1 0</td>
<td>NOT (~) outputs the opposite of its input.</td>
<td>→</td>
</tr>
</tbody>
</table>

Masking is very commonly used with bitwise operations. A mask is a binary constant used to manipulate another bit string in a specific manner, such as setting specific bits to 1 or 0.

**Exercises:**

1) What happens when we fix/set one of the inputs to the 2-input gates? Let \( x \) be the other input.

Fill in the following blanks with either 0, 1, \( x \), or \( \overline{x} \) (NOT \( x \)):

\[
\begin{align*}
\text{x } \& \text{ 0} &= \underline{\phantom{1}} \\
\text{x } \mid \text{ 0} &= \underline{\phantom{1}} \\
\text{x } ^ \text{ 0} &= \underline{\phantom{1}} \\
\text{x } \& \text{ 1} &= \underline{\phantom{1}} \\
\text{x } \mid \text{ 1} &= \underline{\phantom{1}} \\
\text{x } ^ \text{ 1} &= \underline{\phantom{1}} \\
\end{align*}
\]

2) Bit Manipulation/Number Representation exercises:

**Bit Extraction:** Returns the value (0 or 1) of the 19th bit (counting from LSB). Allowed operators: >>, &, |, ~.

```c
int extract19(int x) {
    return ____________________________;
}
```

**Subtraction:** Returns the value of \( x-y \). Allowed operators: >>, &, |, ~, +.

```c
int subtract(int x, int y) {
    return ____________________________;
}
```

**Equality:** Returns the value of \( x==y \). Allowed operators: >>, &, |, ~, +, ^, !.

```c
int equals(int x, int y) {
    return ____________________________;
}
```

**Divisible by Eight?** Returns the value of \( (x\%8)==0 \). Allowed operators: >>, <<, &, |, ~, +, ^, !.

```c
int divisible_by_8(int x) {
    return ____________________________;
}
```

**Greater than Zero?** Returns the value of \( x>0 \). Allowed operators: >>, &, |, ~, +, ^, !.

```c
int greater_than_0(int x) {
    return ____________________________;
}
```
3) Implement the following C function using control structures and bitwise operators.

```c
int num_pairs_opposite(int x) {
    // returns the number of pairs of bits that are the
    // opposite of each other (i.e. 0 and 1 or 1 and 0)
    // bits are "paired" by taking adjacent bits
    // starting at the lsb (0) and pairs do not overlap.
    // For example, there are 16 distinct pairs in a 32-bit integer
    return 0; // Placeholder for the implementation
}
```

Signed Integers with Two's Complement
Two's complement is the standard for representing signed integers:
- The most significant bit (MSB) has a negative value; all others have positive values (same as unsigned)
- Binary addition is performed the same way for signed and unsigned
- The bit representation for the negative value (additive inverse) of a Two's Complement number can be found by:
  flipping all the bits and adding 1 (i.e. $-x = \overline{x} + 1$).

The “number wheel” showing the relationship between 4-bit numerals and their Two's Complement interpretations is shown on the right:
- The largest number is 7 whereas the smallest number is -8
- There is a nice symmetry between numbers and their negative counterparts except for -8

**Exercises**: (assume 8-bit integers)
1) What is the largest integer? The largest integer + 1?

<table>
<thead>
<tr>
<th>Unsigned</th>
<th>Two’s Complement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2) How do you represent (if possible) the following numbers: 39, -39, 127?
3) Compute the following sums in binary using your Two's Complement answers from above. Answer in hex.

<table>
<thead>
<tr>
<th></th>
<th>Unsigned:</th>
<th>Two's Complement:</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>39</td>
<td>0b_ _ _ _ _ _ _ _</td>
</tr>
<tr>
<td></td>
<td>+(-39)</td>
<td>0b_ _ _ _ _ _ _ _</td>
</tr>
<tr>
<td></td>
<td>0x_ _ &lt;- 0b_ _ _ _ _ _ _</td>
<td>0x_ _ &lt;- 0b_ _ _ _ _ _ _</td>
</tr>
<tr>
<td>b</td>
<td>127</td>
<td>0b_ _ _ _ _ _ _ _</td>
</tr>
<tr>
<td></td>
<td>+(-39)</td>
<td>0b_ _ _ _ _ _ _ _</td>
</tr>
<tr>
<td></td>
<td>0x_ _ &lt;- 0b_ _ _ _ _ _ _</td>
<td>0x_ _ &lt;- 0b_ _ _ _ _ _ _ _</td>
</tr>
<tr>
<td>c</td>
<td>39</td>
<td>0b_ _ _ _ _ _ _ _</td>
</tr>
<tr>
<td></td>
<td>+(-127)</td>
<td>0b_ _ _ _ _ _ _ _</td>
</tr>
<tr>
<td></td>
<td>0x_ _ &lt;- 0b_ _ _ _ _ _ _</td>
<td>0x_ _ &lt;- 0b_ _ _ _ _ _ _ _</td>
</tr>
<tr>
<td>d</td>
<td>127</td>
<td>0b_ _ _ _ _ _ _ _</td>
</tr>
<tr>
<td></td>
<td>+39</td>
<td>0b_ _ _ _ _ _ _ _</td>
</tr>
<tr>
<td></td>
<td>0x_ _ &lt;- 0b_ _ _ _ _ _ _</td>
<td>0x_ _ &lt;- 0b_ _ _ _ _ _ _ _</td>
</tr>
</tbody>
</table>

4) Interpret your answers from 2 & 3 and indicate if overflow has occurred for each of the representations. (For values that cannot be represented, interpret as Two’s Complement, then convert to unsigned.)

<table>
<thead>
<tr>
<th></th>
<th>Unsigned:</th>
<th>Two's Complement:</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>39+(-39)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Two's Complement:</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>39-127</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Two's Complement:</td>
<td></td>
</tr>
</tbody>
</table>