CSE 351 Section 2 – Pointers and Bit Operators

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 \( \text{type}* p \), then \( p + i \) will change the value of \( p \) (an address) by \( i\text{sizeof(type)} \) (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;
}
```
C Bitwise Operators

<table>
<thead>
<tr>
<th>&amp;</th>
<th>0</th>
<th>1</th>
<th>← AND (&amp;) outputs a 1 only when both input bits are 1.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>OR (|) outputs a 1 when either input bit is 1.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>^</th>
<th>0</th>
<th>1</th>
<th>← XOR (^) outputs a 1 when either input is exclusively 1.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>NOT (~) outputs the opposite of its input.</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 \( x \)):
   \[
   x \& 0 = \_0\_0\_ \\
   x | 0 = \_x\_x\_ \\
   x \& 1 = \_x\_x\_ \\
   x | 1 = \_1\_1\_ \\
   x ^ 0 = \_x\_x\_ \\
   x ^ 1 = \_\overline{x}\_\overline{x}\_ \\
   \]

2) Bit Manipulation/Number Representation exercises:

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

   ```
   int extract19(int x) {
       return (x >> 18) & 0x1;
   }
   ```

   Subtraction: Returns the value of \( x - y \). Allowed operators: \( \gg, \&, |, \sim, + \).

   ```
   int subtract(int x, int y) {
       return x + ((~y) + 1);
   }
   ```

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

   ```
   int equals(int x, int y) {
       return !(x ^ y);
   }
   ```

   Divisible by Eight? Returns the value of \( (x \% 8) == 0 \). Allowed operators: \( \gg, \ll, \&, |, \sim, +, ^, ! \).

   ```
   int divisible_by_8(int x) {
       return !(x << 29);
   }
   ```

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

   ```
   int greater_than_0(int x) {
       /* invert and check sign; we need the third operand for the T_min case */
       return ((~x + 1) >> 31) & 0x1 & ~(x >> 31) \_OR\_ !x & ~(x >> 31);
   }
   ```
3) Implement the following C function using control structures and bitwise operators.

```c
int num_pairs_opposite(int x) {
    int count = 0;
    for (int i = 0; i < 16; i++) {  // 32 bits in an integer
        int bit0 = x & 1;
        int bit1 = (x >> 1) & 1;
        count += bit0 ^ bit1;
        x >>= 2;
    }
    return count;
}
```

**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 (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>0111 1111</td>
<td>0111 1111</td>
</tr>
<tr>
<td>1111 1111</td>
<td>0000 0000</td>
</tr>
</tbody>
</table>

2) How do you represent (if possible) the following numbers: 39, -39, 127?
<table>
<thead>
<tr>
<th>Unsigned:</th>
<th>Two's Complement:</th>
</tr>
</thead>
<tbody>
<tr>
<td>39: 0010 0111</td>
<td>39: 0010 0111</td>
</tr>
<tr>
<td>127: 0111 1111</td>
<td>127: 0111 1111</td>
</tr>
</tbody>
</table>

3) Compute the following sums in binary using your Two’s Complement answers from above. *Answer in hex.*

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a.</td>
<td>39 -&gt; 0b 0 0 1 0 0 1 1 1</td>
<td>b.</td>
</tr>
<tr>
<td></td>
<td>+ (-39) -&gt; 0b 1 1 0 1 1 0 0 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x 0 0 &lt;- 0b 0 0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>c.</td>
<td>39 -&gt; 0b 0 0 1 0 0 1 1 1</td>
<td>d.</td>
</tr>
<tr>
<td></td>
<td>+ (-127) -&gt; 0b 1 0 0 0 0 0 0 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x A 8 &lt;- 0b 1 0 1 0 1 0 0 0</td>
<td></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.)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a.</td>
<td>39 + (-39)</td>
<td>b.</td>
</tr>
<tr>
<td></td>
<td>Unsigned: 0 overflow</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Two's Complement: 0 no overflow</td>
<td></td>
</tr>
<tr>
<td>c.</td>
<td>39 + (-127)</td>
<td>d.</td>
</tr>
<tr>
<td></td>
<td>Unsigned: 168 no overflow</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Two's Complement: -88 no overflow</td>
<td></td>
</tr>
</tbody>
</table>