Write straight-line C code to implement the code fragment.

```c
int negate(int x) {

    // Write solution below.
    return ~x + 1;
}
```
Write straight-line C code to implement the code fragment.

/*
 * isGreater - if x > y then return 1, else return 0
 * Example: isGreater(4,5) = 0, isGreater(5,4) = 1
 * Legal ops: ! & ^ | + << >>
 * Max ops: 24
 * Rating: 3
 */

int isGreater(int x, int y) {
  /* Write solution below. */

  int diff = y + x + 1;
  int x_sign = x >> 31;
  int y_sign = y >> 31;
  int x_pos_y_neg = x_sign & y_sign;
  int y_pos_x_neg = y_sign & x_sign;
  return ( ((diff >> 31) & ~y_pos_x_neg) 
             | x_pos_y_neg ) 
    & 1;
Given the following C declarations, fill in the type and value for each expression. If the value cannot be known, write a "?". Also, state whether this value, if known, is guaranteed. Assume standard C semantics and an IA32 machine.

```c
int a[5] = { 1, 2, 3, 4, 5 };  // Array a located at byte address 50 */
int b[8] = { 7, 6, 5, 4, 3, 2, 1, 0 };  // Array b located at byte address 90 */
int c[3] = { 8, 5, 2 };  // Array c located at byte address 70 */
int *abc[3] = { a, c, b };  
```

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Value</th>
<th>Guaranteed</th>
</tr>
</thead>
<tbody>
<tr>
<td>abc[ 2 ][ 1 ]</td>
<td>int</td>
<td>6</td>
<td>yes</td>
</tr>
<tr>
<td>abc[ 3 ][ -1 ]</td>
<td>int</td>
<td>?</td>
<td>no</td>
</tr>
<tr>
<td>abc[ 0 ][ 10 ]</td>
<td>int *</td>
<td>90</td>
<td>no</td>
</tr>
<tr>
<td>abc[ 0 ][ 12 ]</td>
<td>int</td>
<td>5</td>
<td>no</td>
</tr>
<tr>
<td>abc[ 2 ][ -2 ]</td>
<td>int</td>
<td>?</td>
<td>no</td>
</tr>
<tr>
<td>abc[ 1 ]</td>
<td>int *</td>
<td>70</td>
<td>no</td>
</tr>
</tbody>
</table>
The following C-code fragment (including some deleted material)

```
/*
 * Upon entry, assume 0 <= x and y <= 0
 */

int what_fn ( int x, int y )
{
    /* ... fill in from assembler below ... */

    if ( x < y ) {
        int t = x;
        x = y;
        y = t;
    }

    while ( x != 0 ) {
        y--;  
        y--;  

    return( y );
}
```

as compiled into the following assembler code. Fill in the body of the C-function above to correspond to the assembly code below.

```
what_fn:
pushl %ebp
movl %esp, %ebp
movl 8(%ebp), %edx
movl 12(%ebp), %eax
movl %edx, %ecx
cmpl %eax, %edx
jge .L2
movl %eax, %edx
movl %ecx, %eax
.L2:
testl %edx, %edx
je .L4
.L5:
decl %eax
decl %edx
jne .L5
.L4:
popl %ebp
ret
```

Extra credit: Can this be implemented more efficiently? If so, how?

```
return y - x;
```

We know $y \leq 0 \leq x$, so the if statement is unnecessary. The loop just implements subtraction.
5: The following C-code fragment (including some deleted material)

```c
int foo( int x, int y, int n )
{
    do {
        x += n;
        y *= n;
        n --;
    } while ( n > 0 && y < n );

    return { x, y };
}
```

was compiled into the following assembler code (omitting the setup and tear-down code). Fill in the body of the C-function above to describe the assembler below.

```
movl 8(%ebp), %eax
movl 12(%ebp), %ecx
movl 16(%ebp), %edx
.L2:
    addl %edx, %eax
    imull %edx, %ecx
    subl $1, %edx
    testl %edx, %edx
    jle .L5
    cmpl %edx, %ecx
    jl .L2
.L5
```
Show the C transformation steps for the following program; that is, how its conversion to its while form, its do-while form, then its goto form.

```c
int does_what ( int x, int y )
{
    int ans = 0;
    int i = 0;
    for ( i = 1; i != 0; i = i << 1 ) {
        ans = ans | ( x & i ) ^ ( y & i );
    }
    return ans;
}
```

What does the function compute?

<table>
<thead>
<tr>
<th>WHILE form</th>
<th>DO WHILE form</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>i = 1;</code></td>
<td></td>
</tr>
<tr>
<td><code>while (i != 0) {</code></td>
<td></td>
</tr>
<tr>
<td>`ans = ans</td>
<td>(x &amp; i) ^ (y &amp; i);`</td>
</tr>
<tr>
<td><code>i = i &lt;&lt; 1;</code></td>
<td></td>
</tr>
<tr>
<td><code>}</code></td>
<td></td>
</tr>
<tr>
<td><code>return ans;</code></td>
<td></td>
</tr>
<tr>
<td><code>i = 1;</code></td>
<td></td>
</tr>
<tr>
<td><code>if (i != 0) {</code></td>
<td></td>
</tr>
<tr>
<td><code>do {</code></td>
<td></td>
</tr>
<tr>
<td>`ans = ans</td>
<td>(x &amp; i) ^ (y &amp; i);`</td>
</tr>
<tr>
<td><code>i = i &lt;&lt; 1;</code></td>
<td></td>
</tr>
<tr>
<td><code>}</code></td>
<td></td>
</tr>
<tr>
<td><code>while (i != 0);</code></td>
<td></td>
</tr>
<tr>
<td><code>}</code></td>
<td></td>
</tr>
<tr>
<td><code>return ans;</code></td>
<td></td>
</tr>
</tbody>
</table>

What does the function compute?

GOTO form

```c
i = 1;
if (i != 0) goto loop;
end:
return ans;
loop:
ans = ans | (x & i) ^ (y & i);
i = i << 1;
if (i != 0) goto loop;
else goto end; /* optional */
```

```
```