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

```
/*
 * negate - return -x
 *   Example: negate(1) = -1.
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 5
 *   Rating: 2
 */
int negate(int x) {

  /* Write solution below. */

          return  ~x + 1;
```

2: 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;
```

3: 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.

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

.nt *abc[3] = { a, c, b };
```

Expression	Type	Value	Guaranteed
abc[2][1]	int	6	yes
abc[3][-1]	int	?	no
&abc[0][10]	int *	90	no
abc[0][12]	int	5	no
abc[2][-2]	int	?	no
abc[1]	int *	70	no

: 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) {
        x--;
        y--;
   }
   return( y );
 }
```

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

```
hat_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
```

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

```
return y-x;
```

We know $y \le 0 \le x$, so the if statement is unnecessary. The loop just implements subtraction.

5: The following C-code fragment (including some deleted material)

```
int foo( int x, int y, int n )

     do {
          x += n;
          y *= n;
          n--;



        } while ( n > 0 && y < n          );


    return {             x;               }
```

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 oto form.

```
 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;
 }
```

hat does the function above compute? I'll just write the for loop

WHILE form

```
i = 1;
while (i != 0) {
   ans = ans | (x&i)^(y&i);
   i = i << 1;
}
return ans;
```

DO WHILE form

```
i = 1;
if (i != 0) {
   do {
      ans = ans | (x&i)^(y&i);
      i = i << 1;
   } while (i != 0);
}
return ans;
```

GOTO form

```
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
```

What does the function compute?

X ^ y