

1 Introduction

In this activity you will learn about integer operations.

Before you begin, take a minute to assign roles to each member in your group. Try to switch up the roles as much as possible: please don't pick a role for yourself that you have already done more than once. Below is a summary of the four roles; write the name of the person taking that role next to the summary.

If your group only has three members, combine the roles of Facilitator and Process Analyst.

For this and all future activities, the Facilitator should also take the role of the reader and read the questions aloud to the group.

Facilitator Reads question aloud; keeps track of time and makes sure everyone contributes appropriately.

Quality Control Records all questions and answers, and provides team reflection to team and instructor.

Spokesperson Talks to the instructor and other teams. Compiles and runs programs when applicable.

Process Analyst Considers how the team could work and learn more effectively.

Fill in the following table showing which group member is performing each role:

Role	Person
Facilitator	
Quality Control	
Spokesperson	
Process Analyst	

Problem 6. Bit X_w of a bit representation is sometimes called the “sign bit.” What makes this bit different from all the other bits of a representation? Why do you think “sign bit” is an appropriate name for this bit?

It carries the sign information and the weight of the bit.

3 Model 1: Review of Limits/Negative Conversion

Questions: 7 / Allocated Time: 8 minutes

Problem 7. Complete the following table to indicate the most positive (i.e. largest) and most negative (i.e. smallest) number that can be represented with a given number of bits *when using two’s complement representation*.

Bits	Most Positive	Most Negative
1	0	-1
2	1	-2
3	3	-4
4	7	-8

Problem 8. Considering the pattern in the table above, give an expression for the *most positive* number that can be represented by a N -bit two’s complement number. (The textbook calls this number T_{\max} , T for two’s complement. Hint: T_{\max} will be related to a power of two in some way.)

$$T_{\max} = 2^{(n-1)} - 1$$

Problem 9. Considering the pattern in the table above, give an expression for the *most negative* number that can be represented by a N -bit two’s complement number. (The textbook calls this number T_{\min} . It will also be related to a power of two in some way.)

$$T_{\min} = -2^{(n-1)}$$

Problem 10. Add the following two binary numbers together. If there is a carry out from the leftmost bit, discard it (that is, give only the lowest eight bits of the sum).

$$\begin{array}{r}
 1111\ 1000 \\
 +\ 0010\ 0111 \\
 \hline
 1\ 0001\ 111\ 1 \\
 \hline
 \end{array}$$

The number is 31 if we keep only the 8 bits

$$-8 + 39 = 31$$

BBI 2

Problem 11. Convert all three numbers from the previous problem to decimal, treating the binary numbers as unsigned. Is the sum correct (modulo 256)?

248+39=287
287 % 256 = 31
Correct

Problem 12. Convert all three numbers from the previous problem to decimal, treating the binary numbers as *signed*. Is the sum still correct?

-8 + 39 = 31
Correct

Problem 13. Given what you observed in the previous question, does the architecture need multiple adders in hardware for signed and unsigned addition?

No need to have different hardware

4 Model 2: Bitwise Operations

Questions: 7 / Allocated Time: 8 minutes

In 1847, George Boole proposed a way to treat formal logic as a mathematical system, by treating “true” and “false” as the numbers 1 and 0, respectively, and defining mathematical operators based on the standard operations of formal logic—AND, OR, XOR, and NOT. The system he devised is now known as *Boolean algebra*.

Most programming languages allow you to work with “Boolean quantities” using the operators devised by Boole. In C, there are two variants of these operators, known as the *bitwise operators* and the *logical operators*. In this model we will study the bitwise operators, which treat integer values as vectors of bits.

Earlier this week, one of the steps to negating a signed integer was to invert all of the bits. This operation, complement (\sim in C), can be applied to any integer. For example, $\sim 0x0F0F$ is $0xF0F0$.

Problem 14. For each integer X below, compute its complement in hex and binary.

X (hex)	X (bin)	$\sim X$ (bin)	$\sim X$ (hex)
0xCAFE	1100 1010 1111 1110	0011 0101 0000 0001	0x3501
0x3C3C	0011 1100 0011 1100	1100 0011 1100 0011	0xC3C3
0x0000	0000 0000 0000 0000	1111 1111 1111 1111	0xFFFF

Problem 15. There are three other bitwise operators: AND (& in C), OR (|), and XOR (^). Unlike ~, these are *binary* operators. When applied to two bits a and b : $a \& b$ is 1 when (and only when) both operands are 1. $a | b$ is 1 when at least one operand is 1. And $a \wedge b$ is 1 when only 1 operand is 1. (The X is for eXclusive, i.e. one but not both.) Complete the table below.

a	b	$a \& b$	$a b$	$a \wedge b$
0	0	0	0	0
1	0	0	1	1
0	1	0	1	1
1	1	1	1	0

Problem 16. When applied to integers, &, |, and ^ work on each pair of bits independently. Fill in the following table:

0x1 = 0b0001		
X (Dec)	X (Bin)	X & 0x1
-2	1110	0000
-1	1111	0001
0	0000	0000
1	0001	0001
2	0010	0000

Problem 17. For which numbers was $X \& 0x1$ not 0000? What is a common property of these integers?

-1 and 1
LSB is 1

Problem 18. Many times in systems programming, we use bits as *flags*. There will be a set of constants FLAG_X, FLAG_Y, etc. each of which has a numeric value with only one bit in its unsigned representation equal to 1. Then an unsigned int variable can hold any combination of these flags.

If F is a variable holding flags, we can test whether a particular one of those flags, say FLAG_Y, is true with the expression $(F \& \text{FLAG_Y}) == \text{FLAG_Y}$. Explain what this expression calculates.

It isolates the bit at FLAG_Y

Problem 19. The second argument to the C library function open (not to be confused with fopen) provides a concrete example of how flags are typically used in systems programming. The header fileunistd.h declares both open and a set of flag constants

whose names all begin with `O_` (O is for open). Here is one common way to call `open`, supplying a *combination* of `O_` flags: open a file for writing (`O_WRONLY`), create it if it doesn't exist (`O_CREAT`), and erase the previous contents if it does exist (`O_TRUNC`).

```
int fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC,
               DEFFILEMODE);
```

What is the effect of the `|` operator in this example?

Take a union of all the flags. Combine the effect of all flags. Condition is satisfied if one of the flags is true

Problem 20. De Morgan's laws of logical duality say that, for all x and y , we have

$$\begin{aligned}\sim(x \ \& \ y) &== \sim x \ | \ \sim y \\ \sim(x \ | \ y) &== \sim x \ \& \ \sim y\end{aligned}$$

Fill in the table below to verify the first of these laws for a selection of inputs.

x	y	$x \ \& \ y$	$\sim x \ \ \sim y$	Equal?
0xF	0x1			
0x5	0x7			
0x3	0xC			

5 Model 3: Logical Operations

Questions: 4 / Allocated Time: 5 minutes

The other variant of Boolean operators in C is the *logical* operations. These are closer to Boole's original idea; there are only two possibilities for their inputs and their outputs. Because C did not originally have a genuine Boolean type,¹ the logical operators work on integer values, just like the bitwise operators do. The difference is that the logical operators treat *any nonzero value* as "true". Only zero is treated as "false". Furthermore, no matter what the inputs to the logical operators are, their output will be either 0 (false) or 1 (true).

The three logical operators are NOT (!),² AND (&&), and inclusive OR (||). There is no logical XOR operator.

¹A genuine Boolean type was added in the 1999 revision of the C standard, but we are old-fashioned in this course and we mostly won't use it.

²Trivia: when reading code out loud, `!` is often pronounced "bang."

Problem 21. Of the 16 possible 4-bit values, how many are considered as false by the logical operators, and how many are considered as true?

One of those. 0000. Only zero is false.
The rest of 15 are considered true.

Problem 22. Evaluate the following expression: $(0x3 \ \&\& \ 0xC) == (0x3 \ \& \ 0xC)$. Show your work.

$0x1 == 0x0 = 0x0$

Problem 23. Fill in the following table to determine whether $!!X == X$ is true for all values of X.

X	!X	!!X	!!X == X
-1	0	1	false
0	1	0	true
1	0	1	true
2	0	1	false

Problem 24. Now let's do the same calculation for \sim instead of $!$. Will we find the same thing?

X	$\sim X$	$\sim\sim X$	$\sim\sim X == X$
-1	0	-1	true
0	-1	0	true
1	-2	1	true
2	-3	2	true

6 Model 4: Multiplication, Division, and Bit Shifts

Questions: 11 / Allocated Time: 30 minutes

We observed yesterday that when you append a 0 bit to the right of a binary number, its value is doubled. This operation is called *left shift* because all of the bits in the original number move left one place. C provides an operator for shifting numbers left, possibly by *several* places: $x \ll n$ produces the number x shifted left by n places. (n must be nonnegative and strictly less than the number of bits in the value x .)

Problem 25. Assuming that each of the following x -values is a 32-bit number, and bits shifted out of the lowest 32 bits are discarded, fill in the table with the result of each left-shift operation.

x	n	$x \ll n$
0x30	1	0x60
0x5A	4	0x5A0
0b0000 0000 0000 0000 0000 0001 0001 1101	31	0x8000 0000

Problem 26. Given the expression $X = (0x1 \ll 2) \mid (0x1 \ll 1)$, what is the value of X in decimal and binary?

$$0100 \mid 0010 = 0110 = 6$$

Problem 27. The compiler can often detect simple multiplication and replace it with shifts and addition. What is an equivalent expression to $x * 6$?

$$x \ll 2 + x \ll 1$$

Problem 28. Given the largest 3-bit unsigned integer, what is its value squared? How many bits does this value require?

$$\begin{aligned} 111 &= 7 \\ 7^2 &= 49 = 110001 \\ &6 \text{ bits} \end{aligned}$$

Problem 29. What is the result from the previous question if it must be stored in 3 bits?

$$\text{Discard excess bits: } 001 = 3$$

The inverse operation of left shift is called *logical right shift*. To logical right shift a number by n places, you delete the n least significant bits, move all of the surviving bits to the right until the lowest surviving bit is in the ones place, and fill in on the left with zeroes. C provides this operation as the \gg operator. (Note: \gg is only *guaranteed* to perform this operation when its left operand is unsigned. See below.)

Problem 30. Compute the following right shifts.

		x	n	x >> n
0011 0000	0001 1000	0x30	1	0x18
0101 1010	0000 0101	0x5A	4	0x05
0001 0001	0000 0010	0x11	3	0x02

Problem 31. Convert the values of x and $x \gg n$ in the previous question to decimal. To what common operation is right shift equivalent?

48 24
90 5
17 2

Divide by 2 (integer division)

Problem 32. Suppose we right shift the negative number -2 by one place. What value *should* this produce to preserve the equivalence discussed above?

-1

Problem 33. With 4-bit integers, what is the binary for -2 ? After logical right shifting by 1, what (decimal) value do you get?

1110, 0111 = 7

Problem 34. (*Advanced*) How might you change the right shift operation to make it correctly handle signed integers?

Fill the copies of the sign bit. AKA arithmetic shift

Problem 35. (*Advanced*) One way to visualize a decimal number in binary is to repeatedly divide by 2 and compute the remainder. Fill in the blanks in this loop to make it compute this visualization. Assume `saveNextBit(b)` is a helper function that takes in a bit (either 0 or 1), then records the given bit to the right of all currently recorded bits.

```
while (x > 0) {  
    saveNextBit( x & 0x1 );  
    x = x >> 1 ;  
}
```