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	

2 Model 0: Review of Representation

Questions: 6 / Allocated Time: 7 minutes

Problem 1. Add the following two unsigned binary numbers together. Give the result in both binary and decimal.

1					
	1	0	1	0	
+	1	0	0	0	
1	0	0	1	0	= 18

Problem 2. How many bits were required to represent the *sum* in the previous question?

Five—one more than either of the addends.

Problem 3. If you only have 4 bits to represent this sum, what number(s) might you provide? Explain.

The usual approach is to "truncate" the answer, throwing away the fifth (leftmost) bit. In this case the result would be 2.

The *bit representation* of a number X is a string of bits

$$\langle X \rangle = \langle X_w, X_{w-1}, \ldots, X_2, X_1, X_0 \rangle$$

where each X_i is either 1 or 0. w is called the *bit width* of X.

Problem 4. Supposing $\langle X \rangle$ is an *unsigned* bit representation (that is, it can only represent numbers greater than or equal to 0), give a formula for the value of X in terms of the X_i. (Hint: a summation will be required.)

$$X = \sum_{i=0}^{w} 2^i \cdot X_i$$

Problem 5. Give a similar formula for when $\langle X \rangle$ is a *two's complement* bit representation (can represent any integer, as long as w is big enough).

$$X = -2^w \cdot X_w + \sum_{i=0}^{w-1} 2^i \cdot X_i$$

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?

Bit X_w is the only bit whose place-value is different in the signed (two's complement) interpretation than in the unsigned interpretation. It's called the "sign bit" because, in the two's complement interpretation, the number X is negative if and only if X_w is 1.

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.)

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

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.)

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

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).

	1100 0000
	1111 1000
+	0010 0111
	0001 1111

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

$1111\,1000_2 = 248_{10}$. $0010\,0111_2 = 39_{10}$. $0001\,1111_2 = 31_{10}$.
$248 + 39 = 287 \equiv 31 \pmod{256}$.
The sum is correct.

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

$1111\,1000_2 = -8_{10}$. $0010\,0111_2 = 39_{10}$. $0001\,1111_2 = 31_{10}$.
$-8 + 39 = 31$.
The sum is still 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, signed and unsigned addition are the same operation (as long as two's complement representation is used for negative numbers).

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 (~ in C), can be applied to any integer. For example, `~0x0F0F` is `0xF0F0`.

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

`X` (hex)	`X` (bin)	`~X` (bin)	`~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 `^` 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 `^` 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:

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. Generalizing, X `&` `0x1` is nonzero for all odd integers.

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 & FLAG_Y) == FLAG_Y`. Explain what this expression calculates.

The subexpression `F & FLAG_Y` will evaluate as zero if flag `FLAG_Y` is not set, or as the value of `FLAG_Y` if it is set. Comparing the result to the value of `FLAG_Y` *canonicalizes* the value produced by the overall expression, making it be 0 if the flag is not set, or 1 if it is.

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 file `unistd.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?

Each bit in `A|B` will be set if the corresponding bit in `A` *or* the corresponding bit in `B` is set, so the effect of the `|` operator is to combine the flags. The number passed as the second argument to `open` will have the bits for all three of `O_WRONLY`, `O_CREAT`, and `O_TRUNC` set.

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

```
~(x & y) == ~x | ~y
~(x | y) == ~x & ~y
```

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

x	y	`x & y`	`~x | ~y`	Equal?
0xF	0x1	0xE	0xE	Yes
0x5	0x7	0xA	0xA	Yes
0x3	0xC	0xF	0xF	Yes

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.

[1] A 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.

[2] Trivia: 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 value is false: `0000`. All of the other 15 values are nonzero and therefore considered true.

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

`(0x3 && 0xC) == 1` because both 0x3 and 0xC are nonzero.
`(0x3 & 0xC) == 0x0` because each individual bit is only set in one of the two input values.
So the two sides of the original expression are not equal.
Note that in most cases `(a && b) != (a & b)` *even if* both sides evaluate to nonzero values.

Side note: the parentheses in all these expressions are necessary, because the *precedence* of both `&` and `&&` is *lower* than that of `==`. This is because people use the logical operators to combine the results of comparisons (e.g. `x >= 2 && x < 10`) much more often than they compare the results of logical operations to anything.

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	No
0	1	0	Yes
1	0	1	Yes
2	0	1	No

No, it isn't.

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

`X`	`~X`	`~~X`	`~~X == X`
−1	0	−1	Yes
0	−1	0	Yes
1	−2	1	Yes
2	−3	2	Yes

Unlike logical negation, bitwise complement is its own inverse.

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, is 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 << 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 << n`
0x30	1	0x060
0x5A	4	0x5A0
0x11D	31	0x8000 0000

Problem 26. Given the expression `X = (0x1 << 2) | (0x1 << 1)`, what is the value of X in decimal and binary?

$$X = 6_{10} = 0110_2$$

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

Two possible answers are `(x << 2) + (x << 1)` and `(x + x + x) << 1`.

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

The largest 3-bit unsigned integer is $111_2 = 7_{10}$. Its value squared is $49_{10} = 110001_2$, which requires 6 bits.

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

Assuming excess bits are discarded, $001_2 = 1_{10}$.

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 >> operator. (Note: >> 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`
0x30	1	0x18
0x5A	4	0x05
0x11	3	0x02

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

`x`	`n`	`x >> n`
48	1	24
90	4	5
17	3	2

A single right shift is equivalent to dividing by 2, so right shifting by N is equivalent to dividing by 2^N with *truncation* (not rounding) to the nearest integer.

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

To preserve equivalence with division by 2^N, we should have $-2 >> 1 = -1$.

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

$-2_{10} = 1110_2$ in two's complement. After right shifting by 1, we get $0111_2 = 7_{10}$.

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

We need to fill in the vacated places on the left with copies of the sign bit, instead of always filling in with zeroes. Thus, for instance, $1110_2 >> 1 = 1111_2 = -1_{10}$. This adjusted operation is called *arithmetic* right shift. Notice that when shifting unsigned numbers, you want the original version: $(1110_2 = 14_{10}) >> 1$ should produce $0111_2 = 7_{10}$, not $1111_2 = 15_{10}$. Inconveniently, C does *not* guarantee that >> with a signed left operand will perform arithmetic right 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;
}
```