Advent of Code [About] [Events] [Shop] [Settings] [Log Out] hyattj-osu 28* {:year 2020} [Calendar] [AoC++] [Sponsors] [Leaderboard] [Stats]

--- Day 14: Docking Data ---

As your ferry approaches the sea port, the captain asks for your help again. The computer system that runs this port isn't compatible with the docking program on the ferry, so the docking parameters aren't being correctly initialized in the docking program's memory.

After a brief inspection, you discover that the sea port's computer system uses a strange bitmask system in its initialization program. Although you don't have the correct decoder chip handy, you can emulate it in software!

The initialization program (your puzzle input) can either update the bitmask or write a value to memory. Values and memory addresses are both 36-bit unsigned integers. For example, ignoring bitmasks for a moment, a line like $\boxed{\text{mem}[8]} = 11$ would write the value $\boxed{11}$ to memory address $\boxed{8}$.

The bitmask is always given as a string of 36 bits, written with the most significant bit (representing 2^35) on the left and the least significant bit (2^0 , that is, the 1s bit) on the right. The current bitmask is applied to values immediately before they are written to memory: a 0 or 1 overwrites the corresponding bit in the value, while an 1 leaves the bit in the value unchanged.

For example, consider the following program:

This program starts by specifying a bitmask (mask =). The mask it specifies will overwrite two bits in every written value: the 2s bit is overwritten with n, and the 64s bit is overwritten with n.

The program then attempts to write the value [1] to memory address [8]. By expanding everything out to individual bits, the mask is applied as follows:

So, because of the mask, the value [73] is written to memory address [8] instead. Then, the program tries to write [101] to address [7]:

This time, the mask has no effect, as the bits it overwrote were already the values the mask tried to set. Finally, the program tries to write $\boxed{0}$ to address $\boxed{8}$:

64 is written to address 8 instead, overwriting the value that was there previously.

To initialize your ferry's docking program, you need the sum of all values left in memory after the initialization program completes. (The entire 36-

Our sponsors help make Advent of Code possible:

GitHub - We're hiring engineers to make GitHub fast. Interested? Email fast@github.com with details of exceptional performance work you've done in the past.

bit address space begins initialized to the value ① at every address.) In the above example, only two values in memory are not zero - 101 (at address 7) and 64 (at address 8) - producing a sum of 165.

Execute the initialization program. What is the sum of all values left in memory after it completes? (Do not truncate the sum to 36 bits.)

Your puzzle answer was 8471403462063.

```
--- Part Two ---
```

For some reason, the sea port's computer system still can't communicate with your ferry's docking program. It must be using version 2 of the decoder chip!

A version 2 decoder chip doesn't modify the values being written at all. Instead, it acts as a memory address decoder. Immediately before a value is written to memory, each bit in the bitmask modifies the corresponding bit of the destination memory address in the following way:

- If the bitmask bit is 0, the corresponding memory address bit is unchanged.
- If the bitmask bit is \square , the corresponding memory address bit is overwritten with \square .

A floating bit is not connected to anything and instead fluctuates unpredictably. In practice, this means the floating bits will take on all possible values, potentially causing many memory addresses to be written all at once!

For example, consider the following program:

When this program goes to write to memory address 42, it first applies the bitmask:

After applying the mask, four bits are overwritten, three of which are different, and two of which are floating. Floating bits take on every possible combination of values; with two floating bits, four actual memory addresses are written:

```
00000000000000000000000000000000011010
(decimal 26)

0000000000000000000000000000011011
(decimal 27)

000000000000000000000000000111010
(decimal 58)

00000000000000000000000000111011
(decimal 59)
```

Next, the program is about to write to memory address 26 with a different bitmask:

This results in an address with three floating bits, causing writes to eight memory addresses:

000000000000000000000000000000000000	(decimal 16)
000000000000000000000000000000000000000	(decimal 17)
	(decimal 18)
000000000000000000000000000000000000000	(decimal 19)
000000000000000000000000000000000000000	(decimal 24)
000000000000000000000000000000000000000	(decimal 25)
000000000000000000000000000000000000000	(decimal 26)
000000000000000000000000000000000011011	(decimal 27)

The entire 36-bit address space still begins initialized to the value 0 at every address, and you still need the sum of all values left in memory at the end of the program. In this example, the sum is 208.

Execute the initialization program using an emulator for a version 2 decoder chip. What is the sum of all values left in memory after it completes?

Your puzzle answer was 2667858637669.

Both parts of this puzzle are complete! They provide two gold stars: **

At this point, you should return to your Advent calendar and try another puzzle.

If you still want to see it, you can get your puzzle input.

You can also [Share] this puzzle.