--- Day 16: Chronal Classification ---

As you see the Elves defend their hot chocolate successfully, you go back to falling through time. This is going to become a problem.

If you're ever going to return to your own time, you need to understand how this device on your wrist works. You have a little while before you reach your next destination, and with a bit of trial and error, you manage to pull up a programming manual on the device's tiny screen.

According to the manual, the device has four registers (numbered 0 through 3) that can be manipulated by instructions containing one of 16 opcodes. The registers start with the value 0.

Every instruction consists of four values: an opcode, two inputs (named A and B), and an output (named C), in that order. The opcode specifies the behavior of the instruction and how the inputs are interpreted. The output, C, is always treated as a register.

In the opcode descriptions below, if something says "value A", it means to take the number given as A literally. (This is also called an "immediate" value.) If something says "register A", it means to use the number given as A to read from (or write to) the register with that number. So, if the opcode addi adds register A and value B, storing the result in register C, and the instruction addi 0 7 3 is encountered, it would add 7 to the value contained by register 0 and store the sum in register 3, never modifying registers 0, 1, or 2 in the process.

Many opcodes are similar except for how they interpret their arguments. The opcodes fall into seven general categories:

Addition:

- addr (add register) stores into register C the result of adding register A and register B.
- addi (add immediate) stores into register C the result of adding register A and value B.

Multiplication:

- mulr (multiply register) stores into register C the result of multiplying register A and register B.
- muli (multiply immediate) stores into register C the result of multiplying register A and value B.

Bitwise AND:

- banr (bitwise AND register) stores into register C the result of the bitwise AND of register A and register B.
- bani (bitwise AND immediate) stores into register C the result of the bitwise AND of register A and value B.

Bitwise OR:

- borr (bitwise OR register) stores into register C the result of the bitwise OR of register A and register B.
- bori (bitwise OR immediate) stores into register C the result of the bitwise OR of register A and value B.

Assignment:

- setr (set register) copies the contents of register A into register C. (Input B is ignored.)

- `seti` (set immediate) stores value `A` into register `C`. (Input `B` is ignored.)

Greater-than testing:

- `gtir` (greater-than immediate/register) sets register `C` to `1` if value `A` is greater than register `B`. Otherwise, register `C` is set to `0`.
- `gtri` (greater-than register/immediate) sets register `C` to `1` if register `A` is greater than value `B`. Otherwise, register `C` is set to `0`.
- `gtrr` (greater-than register/register) sets register `C` to `1` if register `A` is greater than register `B`. Otherwise, register `C` is set to `0`.

Equality testing:

- `eqir` (equal immediate/register) sets register `C` to `1` if value `A` is equal to register `B`. Otherwise, register `C` is set to `0`.
- `eqri` (equal register/immediate) sets register `C` to `1` if register `A` is equal to value `B`. Otherwise, register `C` is set to `0`.
- `eqrr` (equal register/register) sets register `C` to `1` if register `A` is equal to register `B`. Otherwise, register `C` is set to `0`.

Unfortunately, while the manual gives the `name` of each opcode, it doesn't seem to indicate the `number`. However, you can monitor the CPU to see the contents of the registers before and after instructions are executed to try to work them out. Each opcode has a number from `0` through `15`, but the manual doesn't say which is which. For example, suppose you capture the following sample:

```
Before: [3, 2, 1, 1]
9 2 1 2
After:  [3, 2, 2, 1]
```

This sample shows the effect of the instruction `9 2 1 2` on the registers. Before the instruction is executed, register `0` has value `3`, register `1` has value `2`, and registers `2` and `3` have value `1`. After the instruction is executed, register `2`'s value becomes `2`.

The instruction itself, `9 2 1 2`, means that opcode `9` was executed with `A=2`, `B=1`, and `C=2`. Opcode `9` could be any of the 16 opcodes listed above, but only three of them behave in a way that would cause the result shown in the sample:

- Opcode `9` could be `mulr`: register `2` (which has a value of `1`) times register `1` (which has a value of `2`) produces `2`, which matches the value stored in the output register, register `2`.
- Opcode `9` could be `addi`: register `2` (which has a value of `1`) plus value `1` produces `2`, which matches the value stored in the output register, register `2`.
- Opcode `9` could be `seti`: value `2` matches the value stored in the output register, register `2`; the number given for `B` is irrelevant.

None of the other opcodes produce the result captured in the sample. Because of this, the sample above `behaves like three opcodes`.

You collect many of these samples (the first section of your puzzle input). The manual also includes a small test program (the second section of your puzzle input) - you can `ignore it for now`.

Ignoring the opcode numbers, `how many samples in your puzzle input behave like three or more opcodes?`

Your puzzle answer was `618`.

--- Part Two ---

Using the samples you collected, work out the number of each opcode and execute the test program (the second section of your puzzle input).

What value is contained in register 0 after executing the test program?

Your puzzle answer was 514.

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.