Processor — CPSPC 2004 Praha, Day 4

Submit:	`processor*.out`	Time limit:	4 hours
Input:	`processor*.in`	Memory limit:	of your own
Output:	`processor*.out`	Points:	100

One of the leaders of the today's technological innovation [...][1], the famous inventor prof. I. N. Tell[2] has designed a prototype of a new processor, called Hexadecadium. However, the instruction encoding used by this processor is somewhat wasteful[16] and as everybody knows, 640 KB ought to be enough for anybody[640], so he also has to invent a method for optimizing programs to decrease their length.

Prof. Tell wants to create an ultimate optimizer — that is a program which, given a source code of another program P, outputs the shortest (that is with the minimum number of instructions in the source file) possible program Q which still computes the same as P (i.e., for each input value, the output of Q is identical to the output of P). The only exceptions can be inputs for which P doesn't halt — in such cases, the output of Q can be arbitrary and Q needn't halt as well.

But this task turned out to be extremely difficult[†] and although prof. Tell firmly believes that he will finally accomplish that, he also needs to sell his processors as soon as possible. So he has decided to simplify the problem by considering only programs whose input and output are a single 32-bit number and also temporarily replace the optimizing program by cheap labor. He distributes the programs to his employees and they produce the shortest equivalent program they are able to. Of course, their wages depend on the length of the produced programs. You have correctly spotted the chance to gain a small fortune during your stay in Prague.

Description of the processor

Hexadecadium has 26 registers called `a` to `z`. Each of these registers contains a single 32-bit unsigned integer.

It is controlled by a program, which is a sequence of at most 1 000 instructions of the following form:

[*label*:] [`if` *value relation value*] *command*

[the parts in square brackets can be omitted]. The *label* has no effect on the meaning of the instruction, but it allows to refer to the instruction in jumps and indexed values. The `if` part allows to make the instruction executed conditionally (see below). There are the following *commands*:

- *register* = *value* — write the *value* to the given *register*, where *value* is either:
  - *number* — a decimal number (0...4294967295), or
  - `#`*number* — a hexadecimal number (0...`ffffffff`), or
  - *register* — the value of the given register, or
  - *label*[*value*] — contents of the `data` instruction *value* instructions after *label* (*value* zero uses the instruction at *label*, 1 the following one etc.). If no such instruction exists (either the instruction is not `data` or it would be after the end of the program), zero is returned instead. The *value* inside the square brackets must contain no further square brackets.
- *register* = *value operator value* — perform a single operation on two values and put the result to *register* (if a single register occurs at both sides of the =, the new value is written to the register after both *values* are read). The following *operators* are available:
  - `+`, `-`, `*` — standard arithmetic operations; if they overflow, the result is taken modulo 2^{32},
  - `/`, `%` — standard integer division and remainder operations; division of x by 0 gives result 0 and remainder x,
  - `&`, `|`, `^` — bitwise *and*, *or* and *xor*,
  - `<<`, `>>` — bitwise shifts: x `<<` y gives x shifted by y bits to the left (if $y \geq 32$, the result is always zero), x `>>` y is the same to the right [$1 << 5 = 32$, $9 >> 2 = 2$],
  - `<<<`, `>>>` — bitwise rotations: rotation does the same as a shift, only the bits shifted out from the 32-bit value appear at the opposite side [$9 >>> 2 =$ `#40000002`].
- `jump` *label* — jump to the instruction marked by *label* and continue execution there.
- `halt` — stop program execution.
- `data` [`#`]*number* — this command doesn't do anything except for acting as a place-holder for the given number, which then can be accessed by *label*[*value*].

Each instruction can also be preceded by an optional condition of type `if` *value relation value*, where *relation* is `=`, `<>`, `<`, `>`, `<=` or `>=`. Then the instruction works as described above only if the condition is satisfied, otherwise it does nothing.

In addition to instructions, the program can also contain comments: whenever the `'` character occurs, everything to the end of the line is ignored. Blank lines are ignored as well. The whole language is case-insensitive.

---

[1] 1463 more marketing buzzwords have been omitted to preserve the reader's sanity

[2] a grand-grand-...-son of the legendary Swiss hero William Tell

[16] 16 kilobytes per instruction

[640] Bill Gates, 1981

[†] The mathematicians have of course long ago proven that this is in general impossible, but orthodox businessmen of course don't believe in such a nonsense...

Execution of programs

When a program is started, register `a` contains the input (a single 32-bit number), registers `b` to `z` contain zeroes and the execution starts with the first instruction of the program. The program halts when it executes the `halt` instruction or "falls over the end of the program" (i.e., it tries to execute an instruction after the end of the program). Then the value in register `z` will be interpreted as the output of the program and the contents of all remaining registers will be ignored.

After executing 1 000 000 instructions, the program is considered to be looping forever. (This means that if the original program doesn't halt within this time for a particular input x, your program can produce an arbitrary input or loop forever.)

Input and output

You will find the programs you should optimize in input files `processor`n`.in` ($1 \leq n \leq 5$). These can be found in the directory `/aux/mo/public/problems/processor/`. You should submit the optimized versions of these programs (i.e., the shortest equivalent programs you are able to produce). This is accomplished by writing them to files `processor`n`.out` and calling `submit processor` n. Remember that `submit` has to be called for each output separately. If you want to be sure you've submitted everything, the `status` command will be your friend.

The simulator

A simulator of the Hexadecadium processor is available to you. You can run it by

`hdsim` [`-g`] *program* [*input* …]

where *program* is the name of the file with your program, *input* is the a number used as the input for the program (beware, `#` is used as a comment marker by the Unix shell, so you need to write it as `\#`). If you give no *input*, the program will be only checked for syntax errors. If you give multiple *input*s, the program will be run separately for each of them. Finally, the simulator prints the output of the program.

If you use the `-g` switch, the simulator will print a detailed trace of the program's execution.

Scoring

Each program you submit will be evaluated separately. It will be compared with the original program on a set of inputs. If it turns out to give a different answer on any input, you will score 0 points. If your program will be longer or of the same length as the original program, you will also score 0 points. In all other cases, you will get points according to the following formula:

$$\max(20 - 5 \cdot (y - b),\ 1),$$

where y is the length of your solution (in instructions) and b is the length of the best solution known (including ours).

Example 1

This program counts the number of divisors of the given number:

```
        b=1                     ' 1 is the first divisor candidate
pool:   c=a%b                   ' does it divide a?
        if c=0 z=z+1            ' if so, increase z
        b=b+1                   ' next divisor candidate
        if b<=a jump pool       ' try again
```

Example 2

The following program returns the binary mirror image of the given number (i.e., a number which is written in binary as the same as the given number, but backwards):

```
        b=32                    ' bits in an integer
loop:   c=a&1                   ' extract the lowest bit from a
        a=a>>1
        z=z<<1                  ' and put it to the bottom of z
        z=z|c
        b=b-1                   ' iterate
        if b>0 jump loop
```