# Register Machine

Editorial

Task Author: Samuel Trajtenberg

## 1. Problem recap

We are given a *register machine* with 64 registers numbered 0 to 63, each holding a nonnegative integer and all initially 0. A program is a sequence of instructions, labelled $0, 1, 2, \ldots$ in input order, of three kinds:

- `INCREMENT x, i` — increase $\text{reg}_x$ by 1, then process instruction $i$.
- `DECREMENT x, i, j` — if $\text{reg}_x = 0$ process instruction $i$; otherwise decrease $\text{reg}_x$ by 1 and process instruction $j$.
- `HALT` — stop.

Execution starts at instruction 0. When the machine halts it opens the chest iff $\text{reg}_0 = N$. The machine breaks if it runs more than 50 000 000 instructions or jumps to a non-existent instruction.

We must output a program (by calling `add_increment`, `add_decrement`, `add_halt`) that leaves exactly $N$ in register 0, using *as few instructions as possible*, and at most 100. The interesting subtask is the last one: $1 \le N \le 1\,000\,000$ with the score increasing as the program gets shorter.

Throughout, we write $\text{reg}_k$ for the value stored at register $k$.

## 2. Warm-up subtasks

### Subtask 1 — $N \le 99$

Since $N$ is tiny, just emit $N$ increments on register 0 followed by a halt. This is one instruction per unit, easily within budget for $N \le 99$.

### Subtask 2 — $N$ a power of two

We need a way to multiply a register by 2. The following three-instruction loop does it:

```
0. DECREMENT 0, 3, 1
1. INCREMENT 1, 2
2. INCREMENT 1, 0
```

Each iteration removes 1 from register 0 and adds 2 to register 1, so when the loop exits register 1 holds twice the original register 0. Starting from a single increment and applying $\log_2 N$ doublings (alternating the source/target registers, or copying back) yields any power of two.

## 3. Subtask 3 — the full progression

This is where the problem becomes interesting. Below we walk through the sequence of ideas, each shaving instructions off the previous one, ending with the optimal construction. The instruction counts assume the worst case over $N \le 10^6$ (i.e. up to 20 bits).

### About 80 instructions (20 points)

Use the binary representation of $N$. Take $\log_2 N$ copies of the doubling loop, alternating between registers 1 and 2. Whenever the current power of two appears in the binary expansion of $N$, insert an extra

`INCREMENT 0` into that loop so the corresponding power of two is also added to register 0. This costs at most 4 instructions per bit and builds $N$ from lowest to highest bit.

Equivalently one can build from highest to lowest bit by placing the optional increments *between* the doubling loops rather than inside them:

```
INCREMENT Reg 0
Reg 1 <- 2 * Reg 0
INCREMENT Reg 1   (optional)
Reg 0 <- 2 * Reg 1
INCREMENT Reg 0   (optional)
Reg 1 <- 2 * Reg 0
...
```

Both variants use a similar number of instructions in the worst case.

## About 70 instructions (30 points)

To produce two consecutive 1-bits it is cheaper to multiply by 4 and then *decrement once*, rather than incrementing twice. This needs slightly more care — the value before the $\times 4$ step must be one higher so the subtraction does not disturb higher-order bits — but it brings the cost down to 7 instructions per 2 bits.

## About 60 instructions (40 points)

A different idea, resembling a binary counter:

```
0.  INCREMENT 0, 1
1.  DECREMENT 1, 2, 0
2.  INCREMENT 1, 3
3.  DECREMENT 2, 4, 0
4.  INCREMENT 2, 5
5.  DECREMENT 3, 6, 0
6.  INCREMENT 3, 7
...
39. DECREMENT 20, 40, 0
40. HALT
```

It scans registers $1, 2, 3, \ldots$: if a register holds 1 it resets it to 0 and jumps back to instruction 0; otherwise it sets it to 1 and moves on. If registers $1, \ldots, 20$ are initialised to the binary representation of $X$, the counter runs exactly $X + 1$ times before halting. So we initialise the registers to the binary representation of $N - 1$ (at most 19 extra instructions) and let the counter run. This uses at most 60 instructions.

## 50 instructions (50 points)

The bottleneck is now the counter, at 2 instructions per bit. We shrink it by using a smaller counter and *reusing* it. The difficulty is that the machine has no call stack, so we must engineer our own "return".

A first approach uses a register (say register 31) as a return address: the $i$-th "call" increments it $i$ times, and a chain of decrements at the end of the routine returns to the correct caller. Splitting the binary representation of $N$ into chunks of $X$ bits, using an $X$-bit counter and multiplying register 0 by $2^X$ between chunks, gives 56 instructions at the optimum ($X = 5$ or $X = 7$).

We can return more cheaply. Consider:

```
0. INCREMENT 1, 1
1. DECREMENT 1, 3, 2
2. INCREMENT 2, i1
```

```
3. DECREMENT 2, 5, 4
4. INCREMENT 3, i2
5. DECREMENT 3, 7, 6
6. INCREMENT 4, i3
...
```

The first time the section starting at instruction 1 is entered it jumps to $i_1$, the second time to $i_2$, then $i_3$, and so on. This returns in $2n - 1$ instructions instead of $O(n^2)$, and with $X = 2$ or $X = 3$ the whole solution fits in 50 instructions.

## 43 instructions (57 points)

With $X$ so small, the counter is barely worth it: instead of laying out bits and summing them, we can just use `INCREMENT` instructions directly. Combining the return trick with "construct from highest to lowest bit" gives:

```
0.  INCREMENT 2, N19
1.  INCREMENT 0, 2

// Multiply by 2
2.  DECREMENT 0, 5, 3
3.  INCREMENT 1, 4
4.  INCREMENT 1, 2
5.  DECREMENT 1, 7, 6
6.  INCREMENT 0, 5

// Jump back to either 1 or 2, depending on the bit of N
7.  DECREMENT 2, 9, 8
8.  INCREMENT 3, N18
9.  DECREMENT 3, 11, 10
10. INCREMENT 4, N17
...
37. DECREMENT 17, 39, 38
38. INCREMENT 18, N3
39. DECREMENT 18, 41, 40
40. INCREMENT 19, N2
41. DECREMENT 19, 42, N1
42. INCREMENT 0, 43   (skip if N0 = 0)
43. HALT
```

Here each $N_i$ is 1 if bit $i$ of $N$ is 1, and 2 otherwise. This uses only 43 instructions.

## 32 instructions (68 points)

We can perform the return even more efficiently. Use register 2 as a counter for the number of iterations so far; at the end of each loop increment it, then copy its value to register 3 with:

```
0. DECREMENT 2, 5, 1
1. INCREMENT 3, 2
2. INCREMENT 4, 0
3. DECREMENT 4, 5, 4
4. INCREMENT 2, 3
```

Now one decrement per branch suffices, reducing the cost from 2 to 1 instruction per loop (at a higher fixed cost). Combined with the previous idea this gives a 32-instruction solution.

### 25 instructions (75 points)

The final lever is base choice: we need not work in binary. With a larger base $k$ we either pay extra increments for the last digit, or (if we reuse the increments at the start of the program) add a check so we do not shift on the last iteration. Weighing the number of iterations against the cost of multiplying register 0 by $k$:

| Base $k$ | No. iterations | Multiply by $k$ | Total instructions |
|---|---|---|---|
| 2 | 20 | 5 | 32 |
| 3 | 13 | 6 | 28 |
| **4** | **10** | **6** | **26** |
| 5 | 9 | 8 | 28 |
| 6 | 8 | 7 | 27 |
| 7 | 8 | 10 | 31 |
| 8 | 7 | 8 | 29 |
| 9 | 7 | 8 | 30 |
| 10 | 6 | 9 | 31 |

The best base is 4, at 26 instructions. With base 4 we can save one final instruction by handling the last digit specially:

- If it is 0, halt immediately.
- If it is 1, add one extra increment.
- If it is 2, increment the counter for the second doubling loop before the last iteration — this adds 2 to the final answer.
- If it is 3, add 1 to $N$ initially (reducing to the 0 case) and append a single decrement at the end.

In every case the last digit costs just one instruction, bringing the total to 25.

## 4. The optimal solution — 23 instructions

Let us try to write the number digit by digit in an unspecified base $b$.

To do so, we will need a sequence of instruction that is able to multiply by $b$ the number written in 0. The easiest way to do this is to first copy $8 * \text{reg}_0$ to $\text{reg}_1$ and then copy $\text{reg}_1$ to $\text{reg}_0$, as follows:

```
x:       increment(1, x + 1)
x + 1:   increment(1, x + 2)
x + 2:   increment(1, x + 3)
...
x + b-1: increment(1, x + b)
x + b:   decrement(0, x + b + 1, x)
x + b+1: decrement(1, Q, x + b + 2)
x + b+2: increment(0, x + b + 1)
```

We notice that if we start execution from instruction $x + b$ with $\text{reg}_1 = 0$ we get exactly the behaviour described before: while possible we decrement from register 0 and perform $b$ additions of register 1, then while possible we decrement from register 1 and perform a single addition on register 0. At the end the snippet jumps to instruction $Q$.

What's more, this snippet can actually already perform operations of type $\text{reg}_0 \leftarrow b \cdot \text{reg}_0 + d$ for all $0 \leq d < b$, if instead of starting execution from instruction $x + b$ we start from instruction $x + b - d$.

Therefore, $\lceil \log_b N \rceil$ accurate calls to this snippet will be enough to obtain our desired number.
We now just need a way to keep track of how many digits we have written.

Consider the following snippet:

```
0:      decrement(2, 3, 1)
1:      increment(3, 2)
2:      increment(4, 0)
3:      decrement(4, 5, 4)
4:      increment(2, 3)
5:      increment(2, 6)
6:      decrement(3, K0, 7)
7:      decrement(3, K1, 8)
...
6 + h:  decrement(3, Kh, 6 + h + 1)
7 + h:  halt()
```

When executed from instruction 0, with $\text{reg}_3 = \text{reg}_4 = 0$ the lines from 0 to 5 perform the assignment $(\text{reg}_2, \text{reg}_3) \leftarrow (\text{reg}_2 + 1, \text{reg}_2)$ (thus counting how many times we have executed the snippet) and then pass the execution to line 6. From line 6 onward we decrement register 3 as long as possible, and when it reaches 0 we can choose which instruction to jump to.
If we jump to appropriate positions of the first snippet, and then jump back to the second snippet by setting $Q = 0$, we can effectively write the digits of our number in register 0.
Once we have no more digits to write, we jump to a halt instruction.

For a naive implementation we would set $h = \lceil \log_b N \rceil - 1$, but we can actually save one last instruction by starting directly from the appropriate instruction inside the first snippet.
In this way we can have $h = \lceil \log_b N \rceil - 2$ and therefore use in total

$$6 + \lceil \log_b N \rceil + 3 + b = 9 + b + \lceil \log_b N \rceil$$

instructions.

A quick search reveals that for the optimal bases $b \in \{4, 5, 6\}$ this uses 23 instructions.

Out of these, base 4 has the easiest implementation. Below is a model solution in C++.

## Model solution (base 4, 23 instructions) (Credit: Francesco Vercellesi)

```cpp
#include <vector>
#include <tuple>
using namespace std;

void add_increment(int x, int i);
void add_decrement(int x, int i, int j);
void add_halt();

void program_machine(int N) {
  auto perm = [&](int i) {
    int start = 20 - ((N >> 18) & 3);
    if (i == 0) return start;
    if (i == start) return 0;
    return i;
  };
```

```
  vector<tuple<int, int, int>> instr;
  auto increment = [&](int x, int i) { instr.emplace_back(x, perm(i), -1); };
  auto decrement = [&](int x, int i, int j) { instr.emplace_back(x, perm(i),
perm(j)); };
  auto halt = [&] { instr.emplace_back(-1, -1, -1); };

  decrement(2, 3, 1);
  increment(3, 2);
  increment(4, 0);
  decrement(4, 5, 4);
  increment(2, 3);
  increment(2, 6);
  decrement(3, 20 - ((N >> 16) & 3), 7);
  decrement(3, 20 - ((N >> 14) & 3), 8);
  decrement(3, 20 - ((N >> 12) & 3), 9);
  decrement(3, 20 - ((N >> 10) & 3), 10);
  decrement(3, 20 - ((N >>  8) & 3), 11);
  decrement(3, 20 - ((N >>  6) & 3), 12);
  decrement(3, 20 - ((N >>  4) & 3), 13);
  decrement(3, 20 - ((N >>  2) & 3), 14);
  decrement(3, 20 - ((N >>  0) & 3), 15);
  halt();
  increment(1, 17);
  increment(1, 18);
  increment(1, 19);
  increment(1, 20);
  decrement(0, 21, 16);
  decrement(1, 0, 22);
  increment(0, 21);

  auto insert = [&](int idx) {
    auto [x, i, j] = instr[perm(idx)];
    if (x == -1) return add_halt();
    if (j == -1) return add_increment(x, i);
    add_decrement(x, i, j);
  };

  for (int i = 0; i < 23; i++)
    insert(i);
}
```

The `perm` helper swaps instruction 0 with the chosen entry point `start` of the multiply block, so that execution begins at the right digit position — this is the trick that lets us reuse the leading increments and reach $h = \lceil \log_b N \rceil - 2$.

## 5. Summary of the progression

| Idea | Instructions | Key insight |
|---|---|---|
| Binary, increments in loops | 80 | 4 instr / bit |
| Decrement after $\times$ 4 | 70 | 7 instr / 2 bits |
| Binary counter | 60 | counter runs $X\{+\}1$ times |
| Reused counter + cheap return | 50 | return in $2n\{-\}1$ |

| Idea | Instructions | Key insight |
|---|---|---|
| Direct increments, hi-to-lo | 43 | skip the counter |
| Copy-counter return | 32 | 1 instr / loop |
| General base (base 4) | 25 | special last digit |
| Shared multiply + counter | **23** | reuse entry points |