An Unexpected Computer

Universal computation with echo, ed, test, and exec.

Programming typically relies on variables, loops, and if-statements. A lone utility can sometimes be universal, such as `find`[1] with its ability to create loops.

What if we instead assemble universality from several mundane OS tools, none of them Turing-complete alone?

1 Minsky Machines

We need a model simple enough to implement. A Minsky register machine[2] is one of the simplest known Turing-complete models[3]. It has unsigned integer registers and just two computational instructions:

- `INC(R)` increments register R.
- `JZDEC(R, S)` tests R; if zero, branch to state S; if nonzero, decrements and proceeds to the next instruction.

For unconditional jump, we add `JMP(S)`.

Table 1 shows a program that computes 2×3 by repeated addition.

State	Instructions
0	`JZDEC(A, 3); JMP(1)`
1	`JZDEC(B, 2); INC(C); INC(D); JMP(1)`
2	`JZDEC(D, 0); INC(B); JMP(2)`
3	`PRINT(C); HALT`

Table 1: Multiplier: $A = 2, B = 3 \rightarrow C = 6$

2 POSIX Mapping

Let's transpose this model using only basic primitives. A register is a file, where N is encoded as N newline characters. This encoding allows blind operations with standard tools: `echo` appends without knowing the value, `ed 1d` deletes one line, `test -s` checks nonzero without reading. Table 2 summarizes the mapping.

Minsky	POSIX
Register	File (N = number of newlines)
`INC(R)`	`echo >> R`
`JZDEC(R, S)`	`test -s R || exec ./S` `(echo 1d;echo w) | ed -s R`
`JMP(S)`	`exec ./S`
Halt	`exit`

Table 2: Minsky-to-POSIX mapping

3 Shell Implementation

Listing 1 implements the multiplier[4] described in table 1 as a shell script[5]. Run it and see: `Result: 6`.

The script creates four registers and four state scripts in `/tmp/CPU`. You can then boot the machine with `exec ./0`. From that point on, no shell variables, loops, functions or regex are used. The OS keeps replacing the running process image until state `3` calls `exit`. Your `/tmp/CPU` directory literally *becomes* the computer.

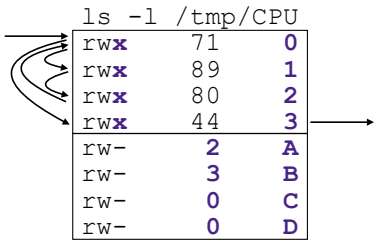

Figure 1: The whole machine is made of state files and register files. The size of the registers is their value.

4 Conclusion

We didn't build a Minsky machine from scratch. We wired one from mundane tools already in the system, illustrating how universal computation can emerge when trivial tools are composed.

Next question: can you build a programming language and a compiler on top of it?

Listing 1: `mul.sh`

```
#!/bin/sh
mkdir -p /tmp/CPU; cd /tmp/CPU
printf '\n\n' >A; printf '\n\n\n' >B; :>C; :>D
printf '#!/bin/sh\n test -s A || exec ./3; (echo 1d;echo w) | ed -s A; exec ./1'                  > 0
printf '#!/bin/sh\n test -s B || exec ./2; (echo 1d;echo w) | ed -s B; echo>>C; echo>>D; exec ./1' > 1
printf '#!/bin/sh\n test -s D || exec ./0; (echo 1d;echo w) | ed -s D; echo>>B; exec ./2'          > 2
printf '#!/bin/sh\n printf "Result: ";wc -l < C;exit'                                              > 3
chmod +x 0 1 2 3; exec ./0
```

[1] Keigo Oka, *Turing Completeness of GNU find: From mkdir-assisted Loops to Standalone Computation*, arXiv:2602.20762, 2026.
[2] `https://esolangs.org/wiki/Minsky_machine`
[3] M. Minsky, *Computation: Finite and Infinite Machines*, Prentice-Hall, 1967, Ch. 14
[4] A 6-state Fibonacci program is also available at `https://seriot.ch/uc/fib.sh`.
[5] Instant run: `curl -O https://seriot.ch/uc/mul.sh && chmod +x mul.sh && ./mul.sh`