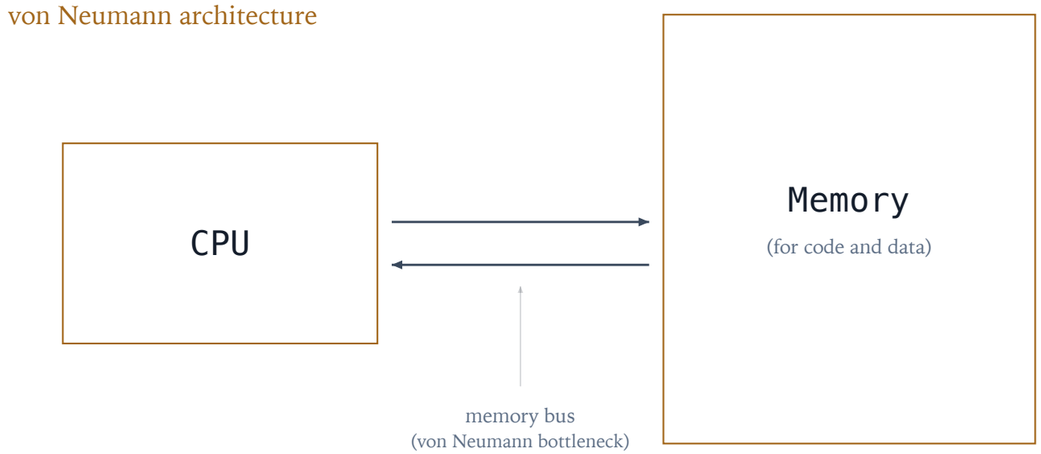
INTRODUCTION TO COMPUTER SCIENCE · WEEK 6 · STATION II

The Machine

Code and data in one memory, a processor that does one thing at a time — and an emulator that runs every program written for it, itself included.

VON NEUMANN · 1945

One memory for code and data. One processor. One bus between them.



the von Neumann architecture

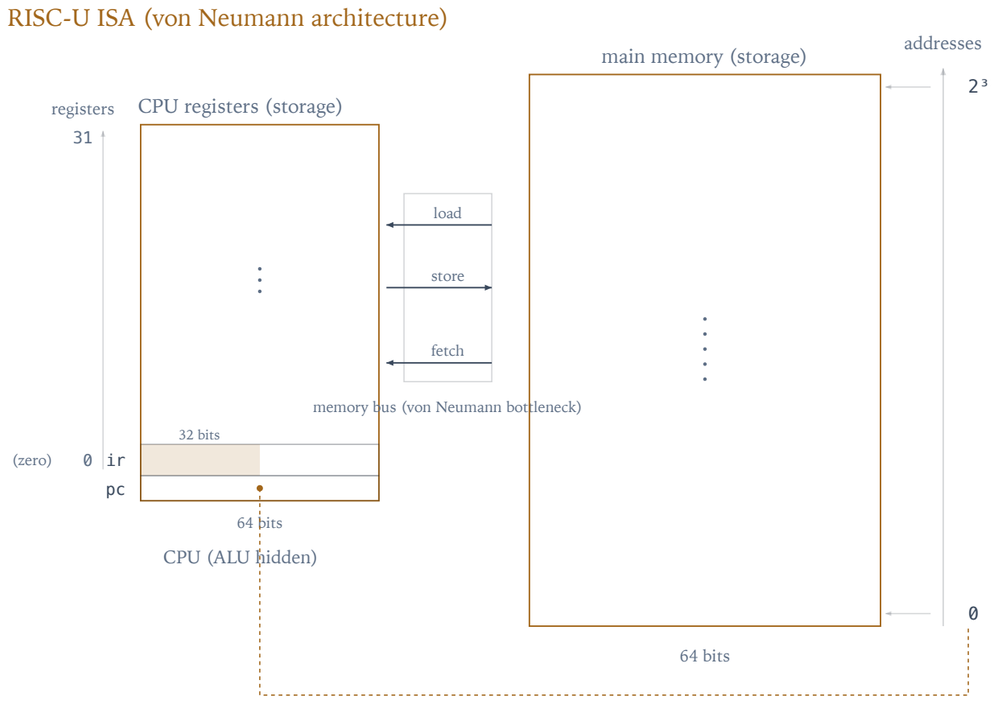
The processor fetches an instruction from memory, executes it, and fetches the next. Instructions and the data they work on live in the same memory, as bytes, which week 3 already told you: nothing in the bytes says which is which.

What says which is the program counter: a register holding the address of the next instruction. Whatever the pc points at is code, by definition, because it is about to be executed.

Every phone, laptop and server is this picture. Most of what is added is speed.

RISC-U

32 registers, an instruction register, a program counter, and 4 GB of bytes.



the whole machine state

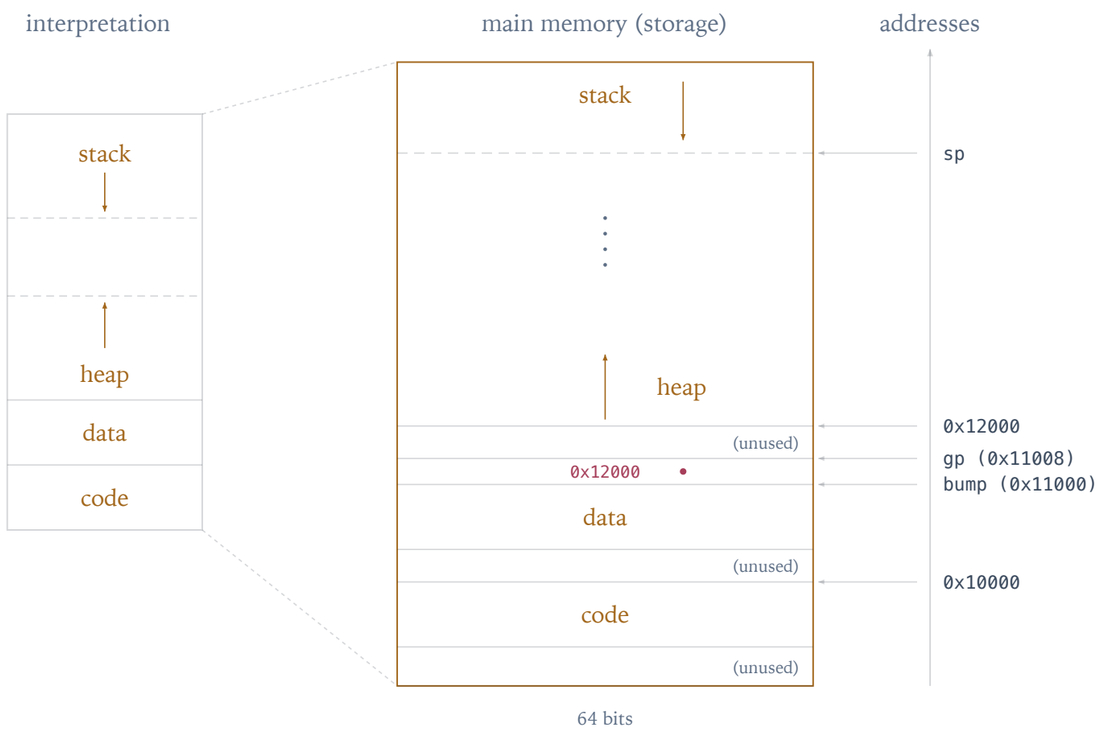
Registers: 32 boxes of 64 bits, the processor's own scratch space. Register `zero` is always 0. Selfie uses eighteen: `sp` for the stack, `gp` for globals, `t0`–`t6` as temporaries, `a0`–`a7` for arguments and results, `ra` for return addresses.

Memory: 2^{32} bytes, addressed 0 to $2^{32}-1$, read and written eight bytes at a time.

The cycle: fetch the 32 bits at `pc` into `ir`, decode them, execute them, and set `pc` to the next instruction — usually `pc + 4`.

LAYOUT

Where a program's bytes go: code, data, heap, stack.



the memory layout of a RISC-U program

Code at the bottom, from address 0x10000: the instructions. Then the data: the global variables, such as week 4's `c`, addressed relative to `gp`.

Then the heap, growing upwards, for memory a program asks for while it runs. And the stack at the top, growing downwards, for procedure calls and their local variables, addressed relative to `sp`.

The `-16(gp)` in week 4's loop is now readable: the global variable `c` lives sixteen bytes below the global pointer.

IN THE TERMINAL

One instruction at a time.

```
$ ./selfie -c tiny.c -d 1
…
tiny.c: pc==0x10158(~6): ld t0,-16(gp): gp==0x11010,mem[0x11000]==0 |- t0==0(0x0) -> t0==0(0x0)
tiny.c: pc==0x1015C(~6): addi t1,zero,7: zero==0(0x0) |- t1==0(0x0) -> t1==7(0x7)
tiny.c: pc==0x10160(~6): sltu t0,t0,t1: t0==0(0x0),t1==7(0x7) |- t0==0(0x0) -> t0==1(0x1)
tiny.c: pc==0x10164(~6): beq t0,zero,6: t0==1(0x1),zero==0(0x0) |- pc==0x10164 -> pc==0x10168
…
tiny.c: 64-bit mipster terminating 64-bit RISC-U binary tiny.c with exit code 7
tiny.c: summary: 107 executed instructions in total
```

Each line: where the pc is, which source line that came from, the instruction, what the registers held before the bar, and what changed after the arrow. The machine's whole life is lines like this. There are 107 of them for the tiny program, and 85,754 for selfie printing its synopsis.

Read the four lines as a sentence: load c into t0, put 7 into t1, is t0 less than t1, and if so do not branch. The loop condition, decided by the machine in four steps, with no idea what a loop is.

SYSTEM CALLS

The one instruction that asks for help: `ecall`.

Thirteen instructions move numbers around. The fourteenth stops the machine and hands control to whoever is running it, with a request number in `a7`: exit, read, write, open, more memory please.

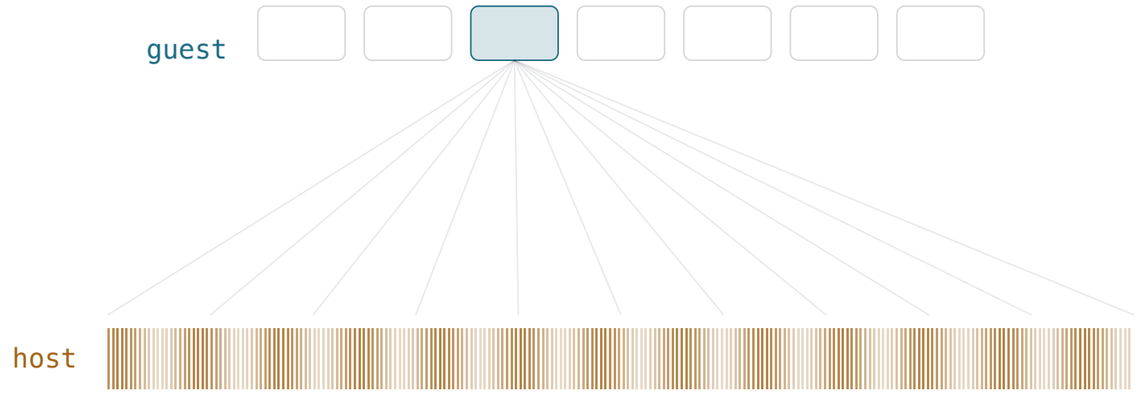
That is how a program prints, reads a file, or ends. It cannot do any of it itself; it asks. Whoever answers is the operating system — and in selfie, that is the emulator, or, in week 10, a hypervisor.

EXCEPTIONS

An `ecall` is a request. A division by zero, or an access to memory the program does not own, is the same mechanism uninvited: the machine stops and hands control up. Week 8's third run ended exactly this way, and week 10 builds an operating system out of nothing but handling these.

EMULATION

Self-execution: an emulator that runs its own machine code.



one guest instruction, about 2,600 on the host

```
$ ./selfie -c selfie.c -o selfie.m -m 2 -l selfie.m -r
…
> selfie.m { -c { source } | -o binary | … }
```

`mipster` is a page of C* per group of instructions: fetch, decode, do what the table of week 4 says, repeat. It is the machine, written down.

So it can run any RISC-U program — including selfie, which contains mipster, which then runs selfie. The second command of week 1. Slower by a factor of about 2,600 per level, and semantically identical.

STATION II

Deterministic, finite, and universal.

DETERMINISTIC

Same state, same input, same next state, always. The 107 lines of the trace are the same every time you run them. A computer never does anything you did not, in the end, tell it to.

FINITE

2^{35} bits of memory, so $2^{34{,}359{,}738{,}368}$ states. Big enough that we pretend it is unbounded — and week 9 says exactly what that pretence buys and costs.

UNIVERSAL

Fourteen instructions are enough to run any program that any computer can run, given enough time and memory. Everything you have ever seen a computer do can be done by this machine, more slowly.

You have now seen the whole of it: bits, a table of fourteen lines, and a loop that reads the table. Everything else in computing is software.

BEFORE NEXT WEEK

Recommended exercises.

1. Read the Machine chapter: Model, Processor, Memory, Instructions, Emulation.
2. Trace `tiny.c` with `-d 1` and find the nine loop instructions executing seven times. How many lines does the loop take in total?
3. For each of the fourteen instructions, write one sentence saying what it does, without looking.
4. Where in memory does `c` live? Find its address in the trace, and find the store that writes 7 into it.
5. Run self-execution and time it against running selfie directly. Estimate the factor.

NEXT WEEK

Uncountability: the diagonal, twice. Programs are countable; what programs do is not. The result everything else hangs on.