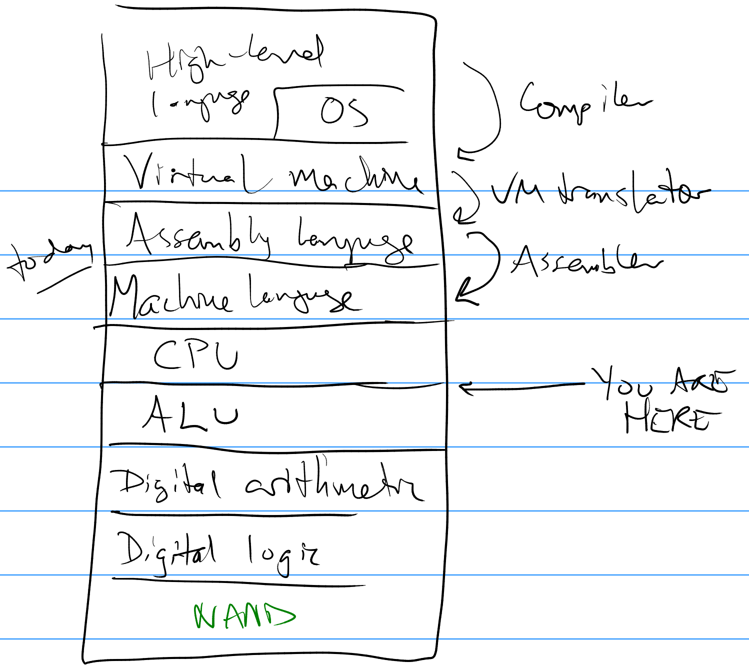


Machine language

Machine instructions expressed as integers / binary strings.
- pro: binary strings = circuit inputs!
- con: incomprehensible for humans?

Assembly language

Machine instructions expressed as symbols.
- pro: easier for humans to read
- Con: has to be translated → mach. lang.
- but 1-1 correspondence.

What abstractions / capabilities are available in assembly / machine language?

- Load values from RAM into CPU register.
- Store values from CPU register → RAM
- Add / subtract / AND / OR / etc — ie. ALU stuff
- Jumps (esp. conditional jumps)

↳ where do conditions come from?

zr, ng bits from ALU.

eg. to test $x > y$ — subtract $y - x$ and see whether result is negative.

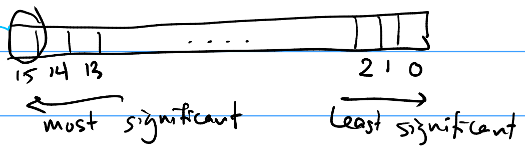
What about I/O? We can set things up so certain "memory addresses" really correspond to external devices.

Machine language for the Hack machine (ch. 4)

CPU has 2 16-bit registers (A, D). (+ PC)

- A stores memory addresses or arbitrary values.
- D just stores arbitrary values.

Instructions are 16 bits



2 types:

- MSB = 0 : A-instruction
  Just load the other 15 bits into the A register.
  Assembly : @ value
- MSB = 1 : C-instruction (C = "computation")
  Basic idea:
  - Value in D register + 1 more value → ALU
    either A-register OR from memory
  - ALU does an operation
  - ALU result → stored somewhere(s)
  - Possibly jump.

C-instruction:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	6
1	1	1													
always 1.			a	c_1	c_2	c_3	c_4	c_5	c_6	d_1	d_2	d_3	j_1	j_2	j_3

address bit: 0: use A register value; 1: look up value from RAM (using A register address)

c bits: Control the ALU.

d (destination bits): where should the ALU result be stored?
$d_1 = A$ $d_2 = M$ $d_3 = D$ (don't set d_1, d_2 @ the same time!)

j (jump) bits: when do we jump?

j_1 : neg
j_2 : zero
j_3 : pos.

jump to address in A register if conditions indicated by j bits are met.