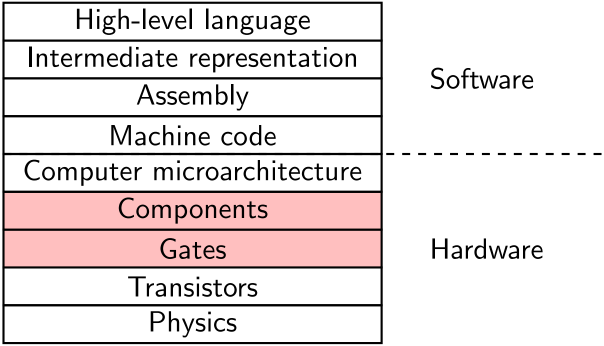
The fetch-execute cycle
COMSM1302 Overview of Computer Architecture

John Lapinskas, University of Bristol

Where the unit is going



High-level language

Intermediate representation

Assembly

Machine code

Computer microarchitecture

Components

Gates

Transistors

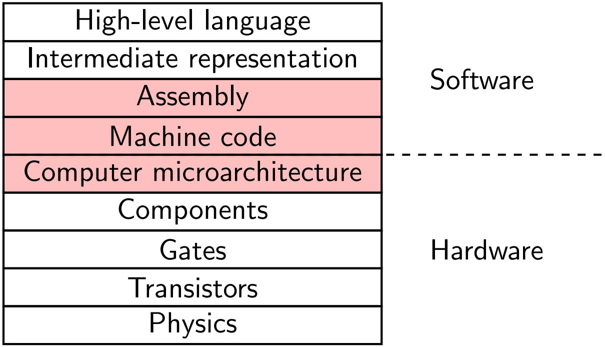
Physics

Software

Hardware

First part of unit: Focused on **hardware**.
Built components for a Hack CPU in labs (e.g. registers, the PC and ALU).

Where the unit is going

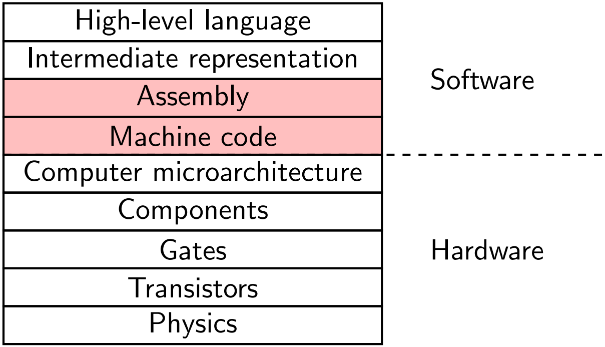


First part of unit: Focused on **hardware**.
Built components for a Hack CPU in labs (e.g. registers, the PC and ALU).

To understand the Hack architecture, we must think about **software**.

Assembly, machine code, and architecture are very tightly bound together, so we'll start by learning Hack assembly and drill down from there.

Where the unit is going



High-level language

Intermediate representation

Assembly

Machine code

Computer microarchitecture

Components

Gates

Transistors

Physics

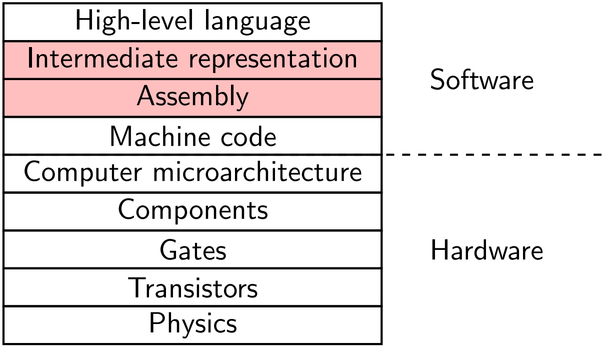
Software

Hardware

First part of unit: Focused on **hardware**.
Built components for a Hack CPU in labs (e.g. registers, the PC and ALU).

After building the Hack CPU, we will write a Hack **assembler** in C.

Where the unit is going


High-level language

Intermediate representation

Assembly

Machine code

Computer microarchitecture

Components

Gates

Transistors

Physics

Software

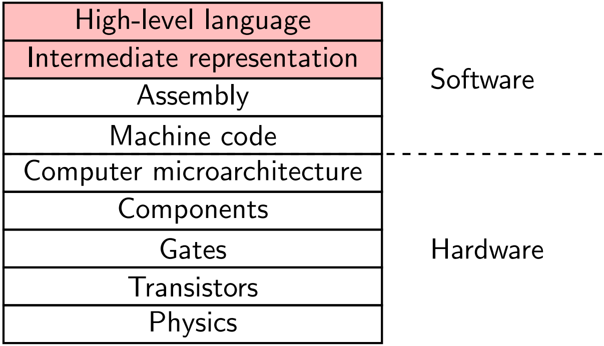
Hardware

First part of unit: Focused on **hardware**.
Built components for a Hack CPU in labs (e.g. registers, the PC and ALU).

After building the Hack CPU, we will write a Hack **assembler** in C.

We will then build towards a high-level language for Hack ("Minijack"), using C to write a **VM translator**

Where the unit is going



High-level language

Intermediate representation

Assembly

Machine code

Computer microarchitecture

Components

Gates

Transistors

Physics

Software

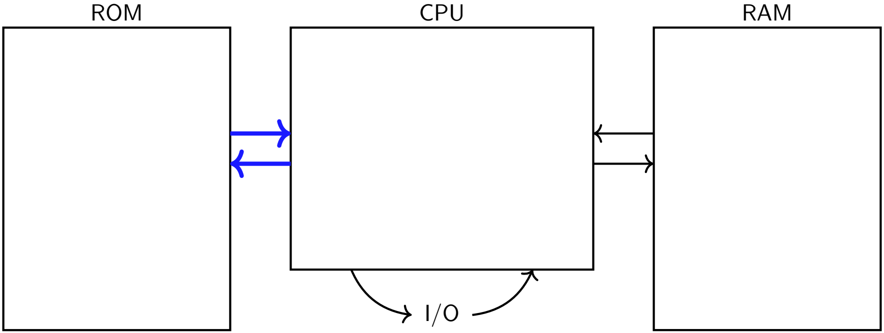
Hardware

First part of unit: Focused on **hardware**.
Built components for a Hack CPU in labs (e.g. registers, the PC and ALU).

After building the Hack CPU, we will write a Hack **assembler**.

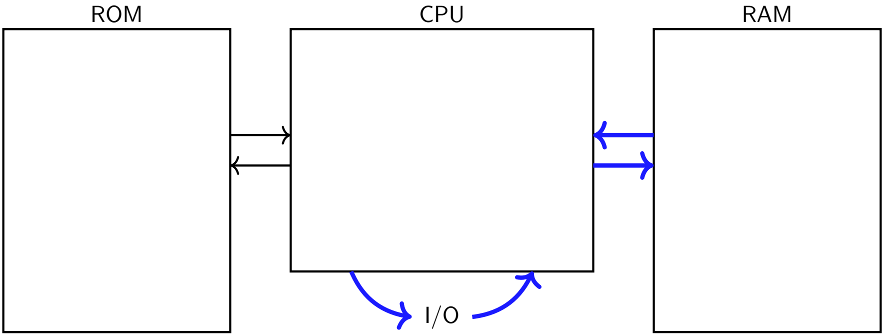
We will then build towards a high-level language for Hack ("Minijack"), using C to write a **VM translator**, then a (very simple!) **compiler**.

The Hack architecture



On each clock cycle, the Hack **central processing unit** (**CPU**) reads (**fetches**) one 16-bit binary **instruction** from **read-only memory** (**ROM**).

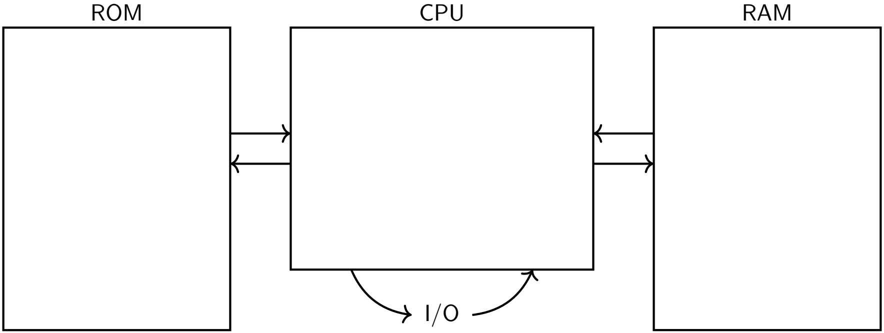
The Hack architecture



On each clock cycle, the Hack **central processing unit** (**CPU**) reads (**fetches**) one 16-bit binary **instruction** from **read-only memory** (**ROM**).

The CPU then **executes** this instruction, which may read from or write to the keyboard, screen, or 32KB of **random access memory (RAM)**.

The Hack architecture

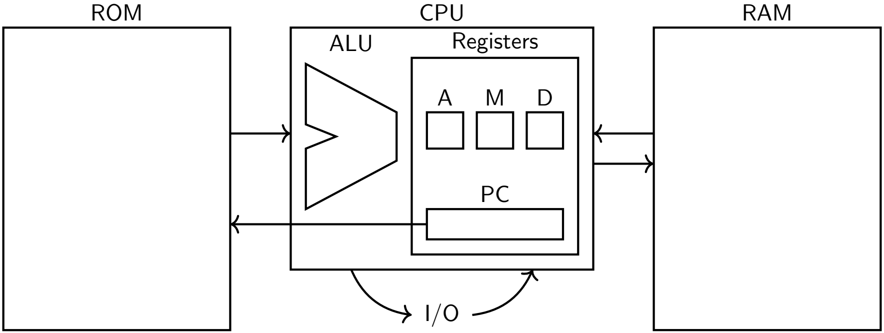


On each clock cycle, the Hack **central processing unit** (**CPU**) reads (**fetches**) one 16-bit binary **instruction** from **read-only memory** (**ROM**).

The CPU then **executes** this instruction, which may read from or write to the keyboard, screen, or 32KB of **random access memory (RAM)**.

This is called the **fetch-execute cycle**, and is common to all CPUs.
(Not all CPUs fetch from ROM, though — see later in unit.)

The Hack architecture

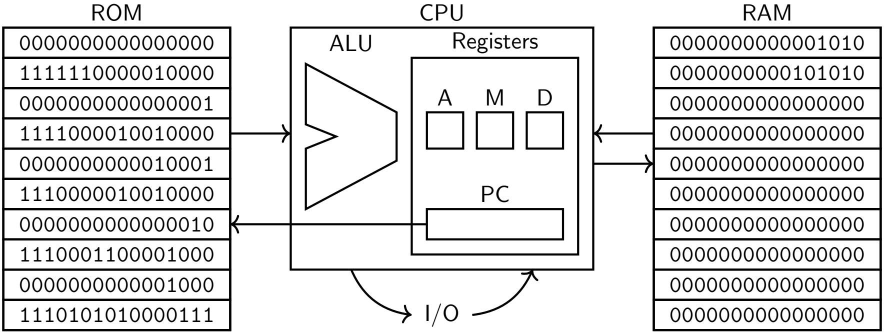


The most complex part of the CPU is an **arithmetic logic unit** (**ALU**), which handles arithmetic and boolean operations like +, -, and &.

The CPU also contains four registers including the **program counter** (**PC**), which holds the address of the next instruction for the CPU to fetch.

Why registers? Because we can only read one word from RAM per clock tick.

The Hack architecture



ROM
0000000000000000
1111110000010000
0000000000000001
1111000010010000
0000000000010001
1110000010010000
0000000000000010
1110001100001000
0000000000001000
1110101010000111

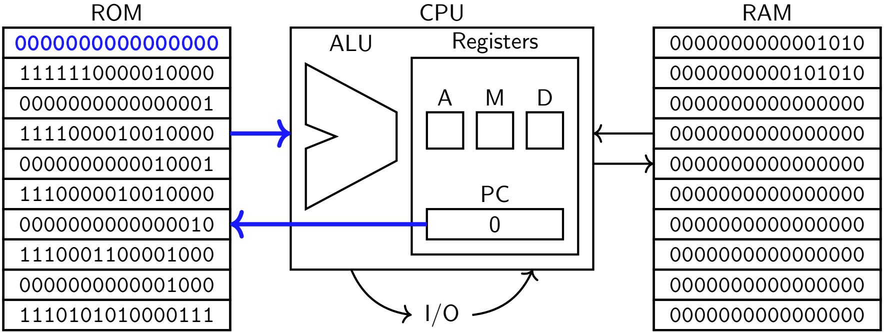
RAM
0000000000001010
0000000000101010
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000

You've made all these parts in labs — we'll put them together next week!

We'll talk about the *A*, *D* and *M* registers next video. First, here's a simple program to compute RAM[0] + RAM[1] + 17 and store the result in RAM[2]. (We write RAM[*i*] for the 16-bit word stored in RAM at address *i*.)

Don't worry about how it works yet — focus on the fetch-execute cycle.

The Hack architecture



ROM

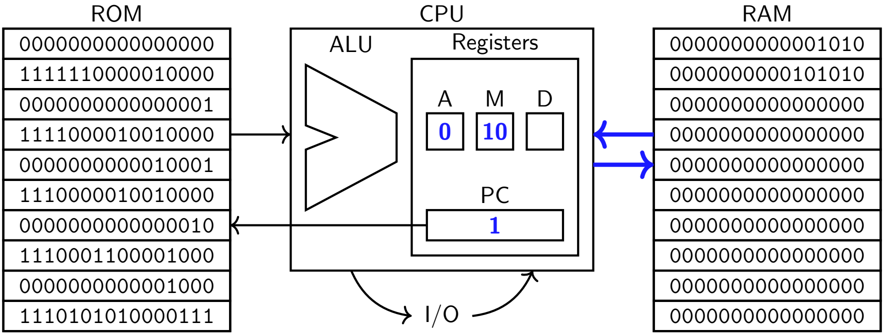
0000000000000000
1111110000010000
0000000000000001
1111000010010000
0000000000010001
1110000010010000
0000000000000010
1110001100001000
0000000000001000
1110101010000111

RAM

0000000000001010
0000000000101010
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000

The first instruction stores 0 in A. The PC auto-increments to 1.
The M register always holds RAM[A]. Here, that's RAM[0] $=$ 10.

The Hack architecture

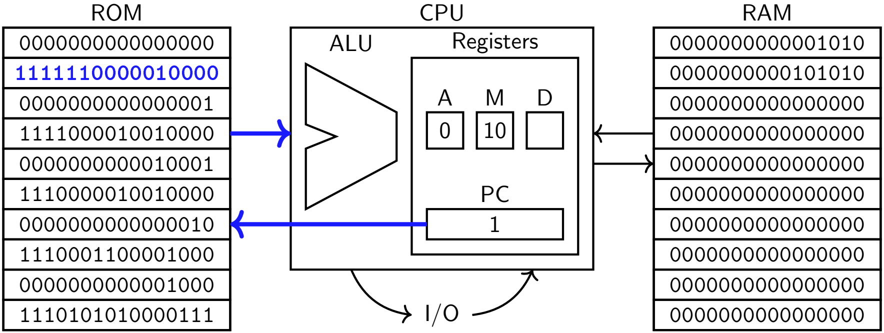


ROM
0000000000000000
1111110000010000
0000000000000001
1111000010010000
0000000000010001
1110000010010000
0000000000000010
1110001100001000
0000000000001000
1110101010000111

RAM
0000000000001010
0000000000101010
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000

The first instruction stores 0 in A. The PC auto-increments to 1.
The M register always holds RAM[A]. Here, that's RAM[0] $= 10$.

The Hack architecture



ROM

0000000000000000
1111110000010000
0000000000000001
1111000010010000
0000000000010001
1110000010010000
0000000000000010
1110001100001000
0000000000001000
1110101010000111

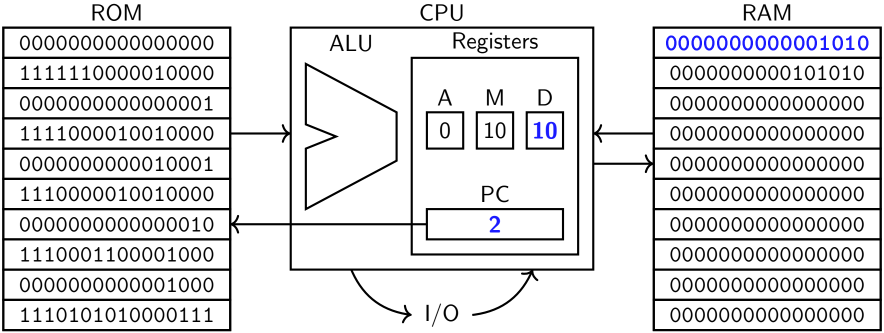
RAM

0000000000001010
0000000000101010
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000

The first instruction stores 0 in A. The PC auto-increments to 1.
The M register always holds RAM[A]. Here, that's RAM[0] $= 10$.

The second instruction reads RAM[A] from M and stores it in D.
The PC auto-increments to 2.

The Hack architecture



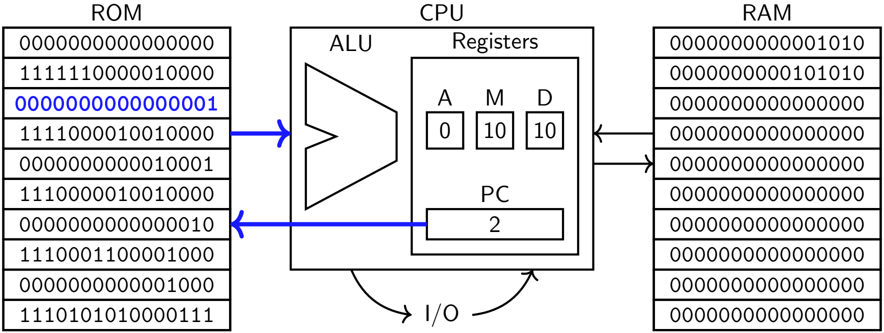
ROM
0000000000000000
1111110000010000
0000000000000001
1111000010010000
0000000000010001
1110000010010000
0000000000000010
1110001100001000
0000000000001000
1110101010000111

RAM
0000000000001010
0000000000101010
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000

The first instruction stores 0 in A. The PC auto-increments to 1. The M register always holds RAM[A]. Here, that's RAM[0] $= 10$.

The second instruction reads RAM[A] from M and stores it in D. The PC auto-increments to 2.

The Hack architecture

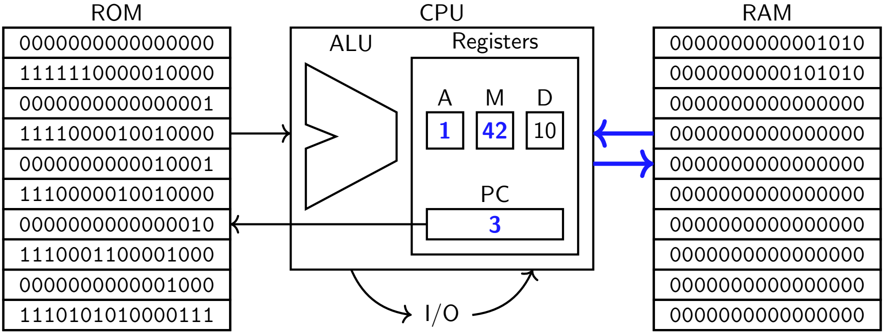


ROM
0000000000000000
1111110000010000
0000000000000001
1111000010010000
0000000000010001
1110000010010000
0000000000000010
1110001100001000
0000000000001000
1110101010000111

RAM
0000000000001010
0000000000101010
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000

The next two instructions read RAM[1] the same way, but this time add it to D rather than storing it in D. Again, the PC auto-increments.

The Hack architecture



ROM

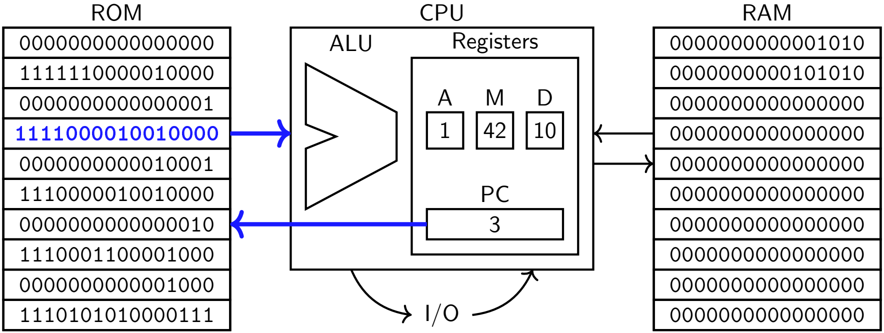
0000000000000000
1111110000010000
0000000000000001
1111000010010000
0000000000010001
1110000010010000
0000000000000010
1110001100001000
0000000000001000
1110101010000111

RAM

0000000000001010
0000000000101010
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000

The next two instructions read RAM[1] the same way, but this time add it to D rather than storing it in D. Again, the PC auto-increments.

The Hack architecture



ROM

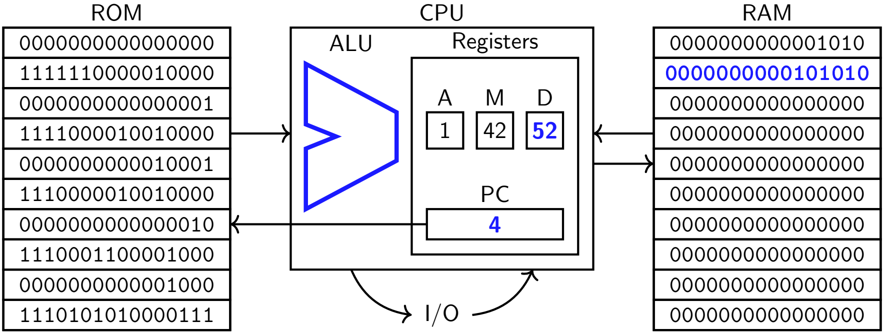
0000000000000000
1111110000010000
0000000000000001
1111000010010000
0000000000010001
1110000010010000
0000000000000010
1110001100001000
0000000000001000
1110101010000111

RAM

0000000000001010
0000000000101010
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000

The next two instructions read RAM[1] the same way, but this time add it to *D* rather than storing it in *D*. Again, the PC auto-increments.

The Hack architecture



ROM

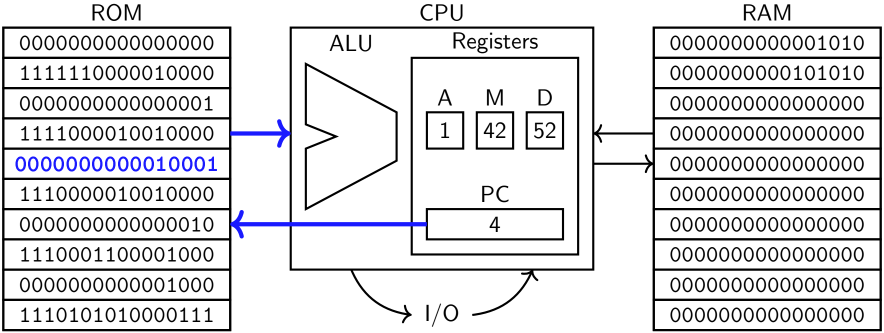
0000000000000000
1111110000010000
0000000000000001
1111000010010000
0000000000010001
1110000010010000
0000000000000010
1110001100001000
0000000000001000
1110101010000111

RAM

0000000000001010
0000000000101010
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000

The next two instructions read RAM[1] the same way, but this time add it to D rather than storing it in D. Again, the PC auto-increments.

The Hack architecture



ROM

0000000000000000
1111110000010000
0000000000000001
1111000010010000
0000000000010001
1110000010010000
0000000000000010
1110001100001000
0000000000001000
1110101010000111

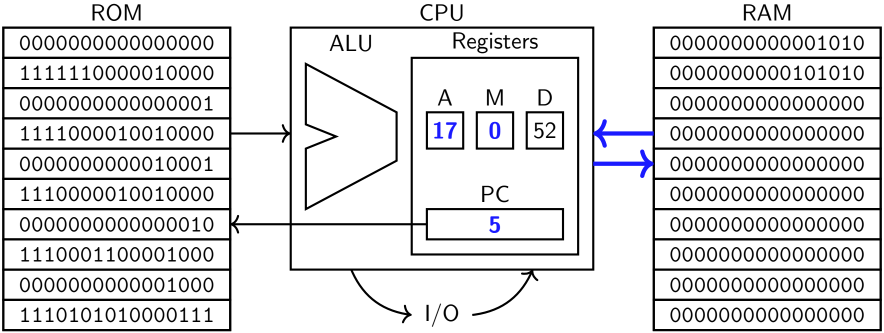
RAM

0000000000001010
0000000000101010
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000

The next two instructions read RAM[1] the same way, but this time add it to D rather than storing it in D. Again, the PC auto-increments.

The next two instructions load 17 into A directly, and add it to D.
So now D contains RAM[0] + RAM[1] + 17.

The Hack architecture



ROM

0000000000000000
1111110000010000
0000000000000001
1111000010010000
0000000000010001
1110000010010000
0000000000000010
1110001100001000
0000000000001000
1110101010000111

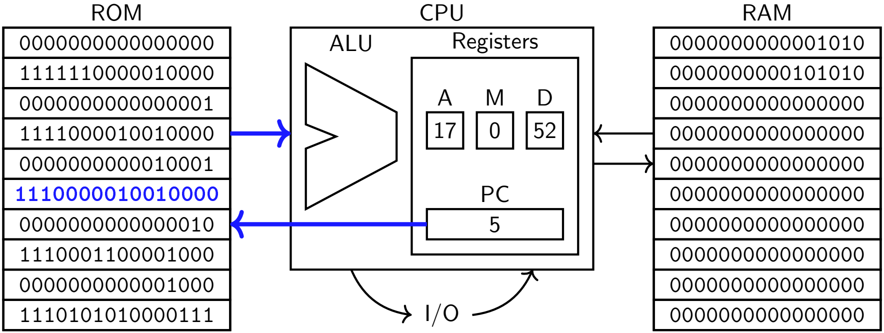
RAM

0000000000001010
0000000000101010
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000

The next two instructions read RAM[1] the same way, but this time add it to *D* rather than storing it in *D*. Again, the PC auto-increments.

The next two instructions load 17 into *A* directly, and add it to *D*.
So now *D* contains $\text{RAM}[0] + \text{RAM}[1] + 17$.

The Hack architecture



ROM

```
0000000000000000
1111110000010000
0000000000000001
1111000010010000
0000000000010001
1110000010010000
0000000000000010
1110001100001000
0000000000001000
1110101010000111
```

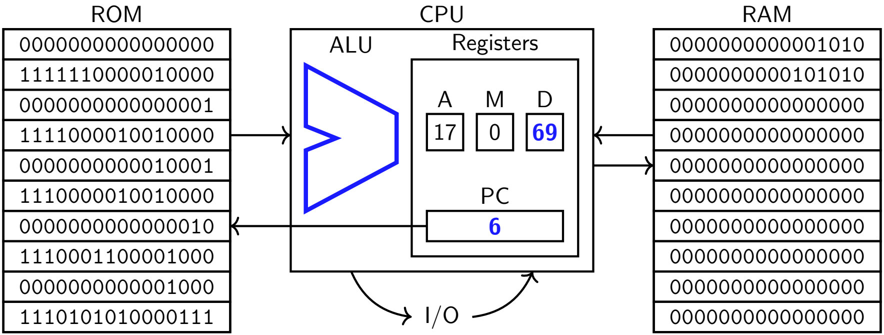
RAM

```
0000000000001010
0000000000101010
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
```

The next two instructions read RAM[1] the same way, but this time add it to *D* rather than storing it in *D*. Again, the PC auto-increments.

The next two instructions load 17 into *A* directly, and add it to *D*.
So now *D* contains RAM[0] + RAM[1] + 17.

The Hack architecture

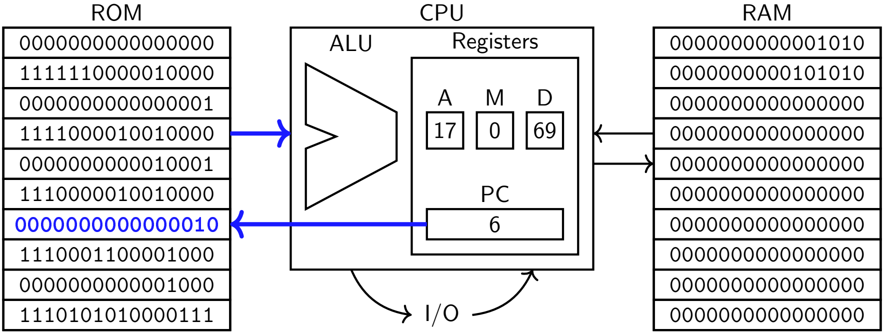


ROM	RAM
0000000000000000	0000000000001010
1111110000010000	0000000000101010
0000000000000001	0000000000000000
1111000010010000	0000000000000000
0000000000010001	0000000000000000
1110000010010000	0000000000000000
0000000000000010	0000000000000000
1110001100001000	0000000000000000
0000000000001000	0000000000000000
1110101010000111	0000000000000000

The next two instructions read RAM[1] the same way, but this time add it to D rather than storing it in D. Again, the PC auto-increments.

The next two instructions load 17 into A directly, and add it to D.
So now D contains $\mathrm{RAM}[0] + \mathrm{RAM}[1] + 17$.

The Hack architecture



ROM
0000000000000000
1111110000010000
0000000000000001
1111000010010000
0000000000010001
1110000010010000
0000000000000010
1110001100001000
0000000000001000
1110101010000111

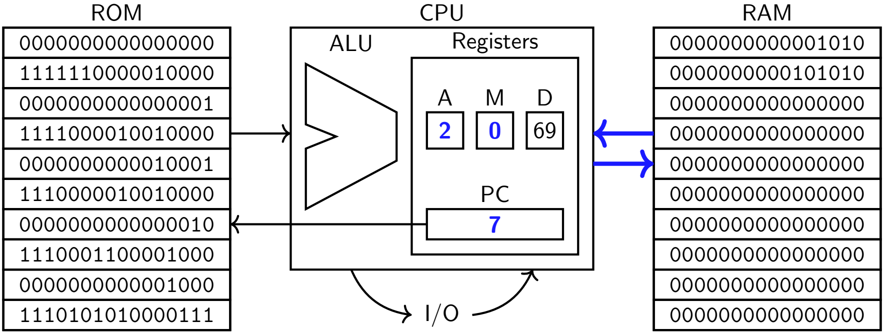
RAM
0000000000001010
0000000000101010
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000

The next two instructions read RAM[1] the same way, but this time add it to *D* rather than storing it in *D*. Again, the PC auto-increments.

The next two instructions load 17 into *A* directly, and add it to *D*.
So now *D* contains RAM[0] + RAM[1] + 17.

The next two instructions copy *D* into RAM[2]. Nice! Just like reading from RAM, we always write to RAM[*A*] — so we had to load 2 into *A* first.

The Hack architecture



ROM

0000000000000000
1111110000010000
0000000000000001
1111000010010000
0000000000010001
1110000010010000
0000000000000010
1110001100001000
0000000000001000
1110101010000111

RAM

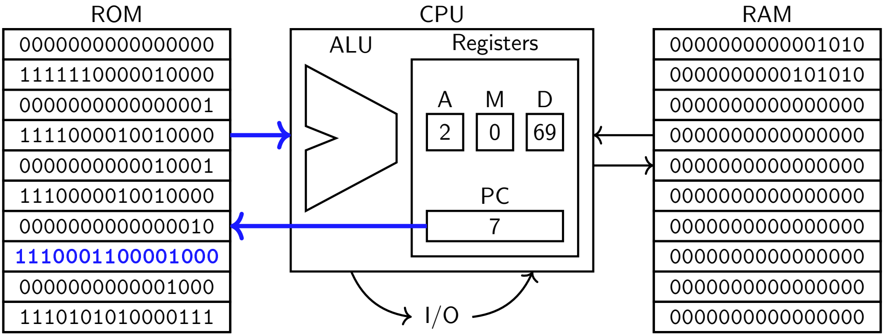
0000000000001010
0000000000101010
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000

The next two instructions read RAM[1] the same way, but this time add it to D rather than storing it in D. Again, the PC auto-increments.

The next two instructions load 17 into A directly, and add it to D.
So now D contains $\text{RAM}[0] + \text{RAM}[1] + 17$.

The next two instructions copy D into RAM[2]. Nice! Just like reading from RAM, we always write to RAM[A] — so we had to load 2 into A first.

The Hack architecture



ROM

0000000000000000
1111110000010000
0000000000000001
1111000010010000
0000000000010001
1110000010010000
0000000000000010
1110001100001000
0000000000001000
1110101010000111

RAM

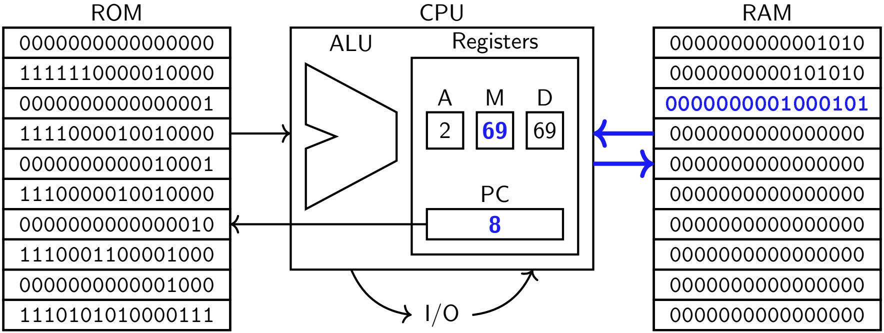
0000000000001010
0000000000101010
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000

The next two instructions read RAM[1] the same way, but this time add it to *D* rather than storing it in *D*. Again, the PC auto-increments.

The next two instructions load 17 into *A* directly, and add it to *D*.
So now *D* contains $\text{RAM}[0] + \text{RAM}[1] + 17$.

The next two instructions copy *D* into RAM[2]. Nice! Just like reading from RAM, we always write to RAM[*A*] — so we had to load 2 into *A* first.

The Hack architecture



ROM
0000000000000000
1111110000010000
0000000000000001
1111000010010000
0000000000010001
1110000010010000
0000000000000010
1110001100001000
0000000000001000
1110101010000111

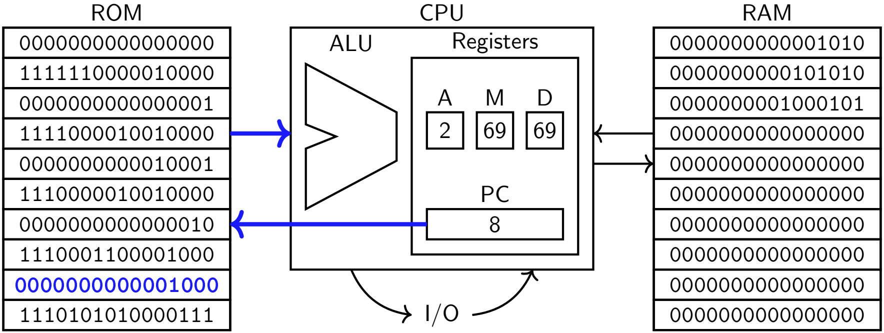
RAM
0000000000001010
0000000000101010
0000000001000101
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000

The next two instructions read RAM[1] the same way, but this time add it to *D* rather than storing it in *D*. Again, the PC auto-increments.

The next two instructions load 17 into *A* directly, and add it to *D*.
So now *D* contains RAM[0] + RAM[1] + 17.

The next two instructions copy *D* into RAM[2]. Nice! Just like reading from RAM, we always write to RAM[*A*] — so we had to load 2 into *A* first.

The Hack architecture



ROM
0000000000000000
1111110000010000
0000000000000001
1111000010010000
0000000000010001
1110000010010000
0000000000000010
1110001100001000
0000000000001000
1110101010000111

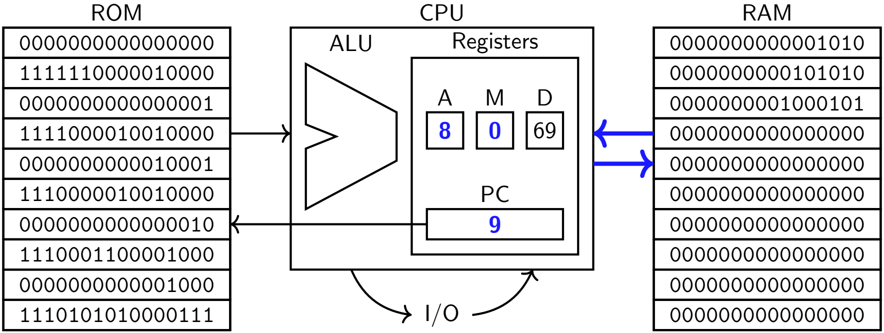
A	M	D
2	69	69

RAM
0000000000001010
0000000000101010
0000000001000101
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000

We then enter an infinite loop. All Hack programs end this way (there's no "halt" instruction).

Notice how the CPU always fetches the instruction whose address is given by the PC — manipulating the PC is how we implement loops and conditionals.

The Hack architecture



ROM
0000000000000000
1111110000010000
0000000000000001
1111000010010000
0000000000010001
1110000010010000
0000000000000010
1110001100001000
0000000000001000
1110101010000111

CPU

ALU

Registers

A	M	D
8	0	69

PC

9

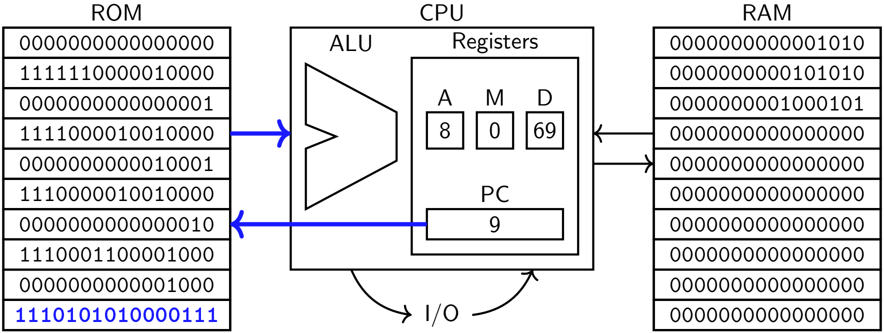
I/O

RAM
0000000000001010
0000000000101010
0000000001000101
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000

We then enter an infinite loop. All Hack programs end this way (there's no "halt" instruction).

Notice how the CPU always fetches the instruction whose address is given by the PC — manipulating the PC is how we implement loops and conditionals.

The Hack architecture



ROM

0000000000000000
1111110000010000
0000000000000001
1111000010010000
0000000000010001
1110000010010000
0000000000000010
1110001100001000
0000000000001000
1110101010000111

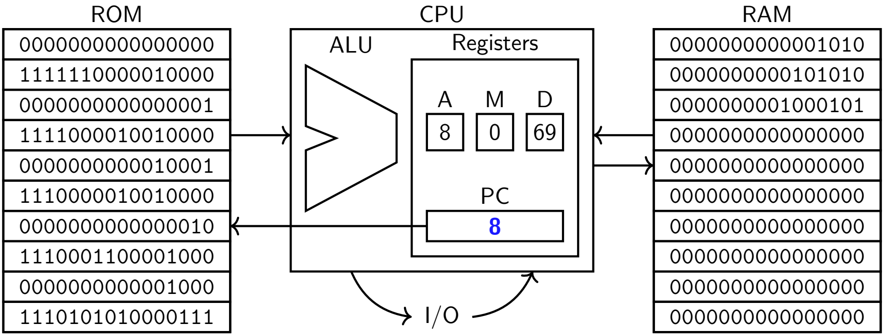
RAM

0000000000001010
0000000000101010
0000000001000101
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000

We then enter an infinite loop. All Hack programs end this way (there's no "halt" instruction).

Notice how the CPU always fetches the instruction whose address is given by the PC — manipulating the PC is how we implement loops and conditionals.

The Hack architecture



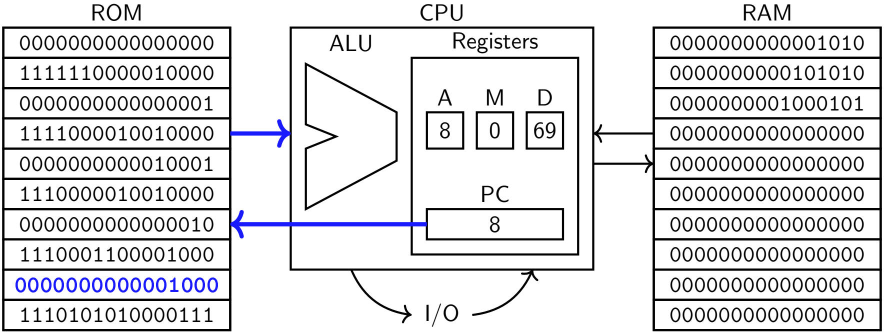
ROM
0000000000000000
1111110000010000
0000000000000001
1111000010010000
0000000000010001
1110000010010000
0000000000000010
1110001100001000
0000000000001000
1110101010000111

RAM
0000000000001010
0000000000101010
0000000001000101
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000

We then enter an infinite loop. All Hack programs end this way (there's no "halt" instruction).

Notice how the CPU always fetches the instruction whose address is given by the PC — manipulating the PC is how we implement loops and conditionals.

The Hack architecture



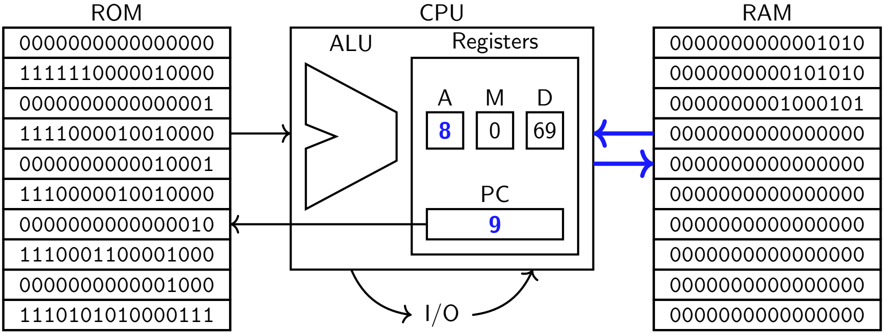
ROM
0000000000000000
1111110000010000
0000000000000001
1111000010010000
0000000000010001
1110000010010000
0000000000000010
1110001100001000
0000000000001000
1110101010000111

RAM
0000000000001010
0000000000101010
0000000001000101
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000

We then enter an infinite loop. All Hack programs end this way (there's no "halt" instruction).

Notice how the CPU always fetches the instruction whose address is given by the PC — manipulating the PC is how we implement loops and conditionals.

The Hack architecture

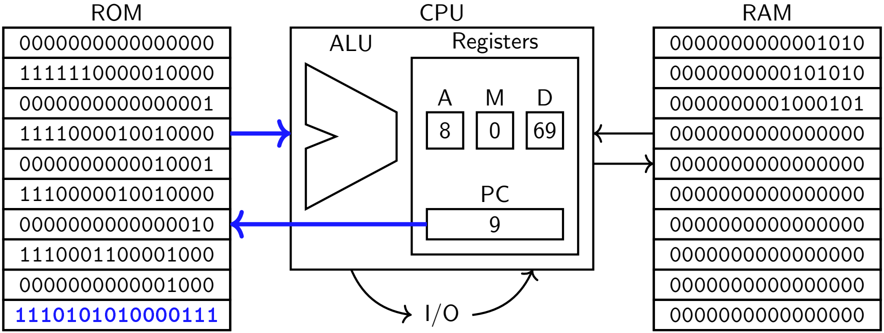


ROM	RAM
0000000000000000	0000000000001010
1111110000010000	0000000000101010
0000000000000001	0000000001000101
1111000010010000	0000000000000000
0000000000010001	0000000000000000
1110000010010000	0000000000000000
0000000000000010	0000000000000000
1110001100001000	0000000000000000
0000000000001000	0000000000000000
1110101010000111	0000000000000000

We then enter an infinite loop. All Hack programs end this way (there's no “halt” instruction).

Notice how the CPU always fetches the instruction whose address is given by the PC — manipulating the PC is how we implement loops and conditionals.

The Hack architecture



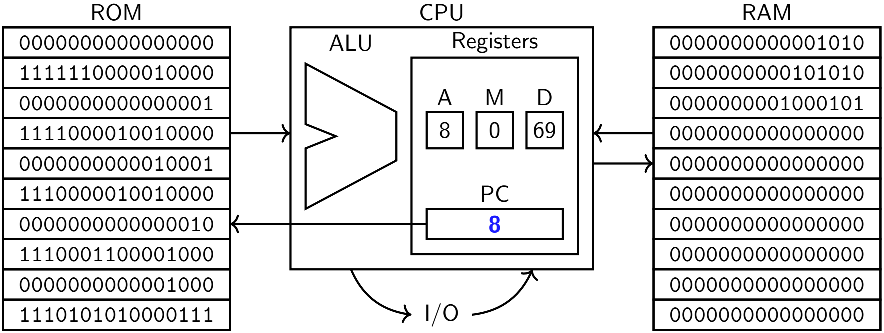
ROM
0000000000000000
1111110000010000
0000000000000001
1111000010010000
0000000000010001
1110000010010000
0000000000000010
1110001100001000
0000000000001000
1110101010000111

RAM
0000000000001010
0000000000101010
0000000001000101
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000

We then enter an infinite loop. All Hack programs end this way (there's no "halt" instruction).

Notice how the CPU always fetches the instruction whose address is given by the PC — manipulating the PC is how we implement loops and conditionals.

The Hack architecture



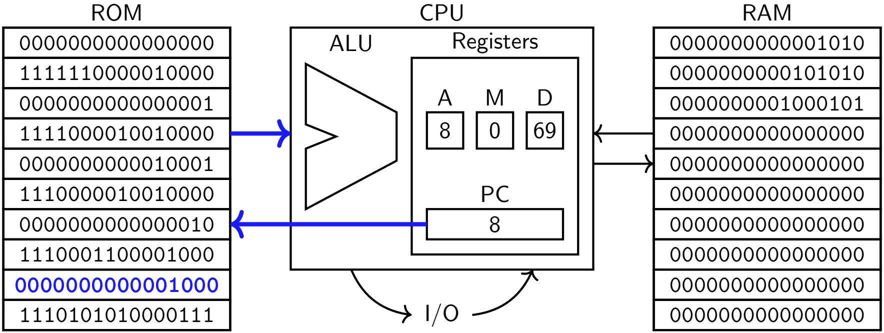
ROM
0000000000000000
1111110000010000
0000000000000001
1111000010010000
0000000000010001
1110000010010000
0000000000000010
1110001100001000
0000000000001000
1110101010000111

RAM
0000000000001010
0000000000101010
0000000001000101
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000

We then enter an infinite loop. All Hack programs end this way (there's no "halt" instruction).

Notice how the CPU always fetches the instruction whose address is given by the PC — manipulating the PC is how we implement loops and conditionals.

The Hack architecture

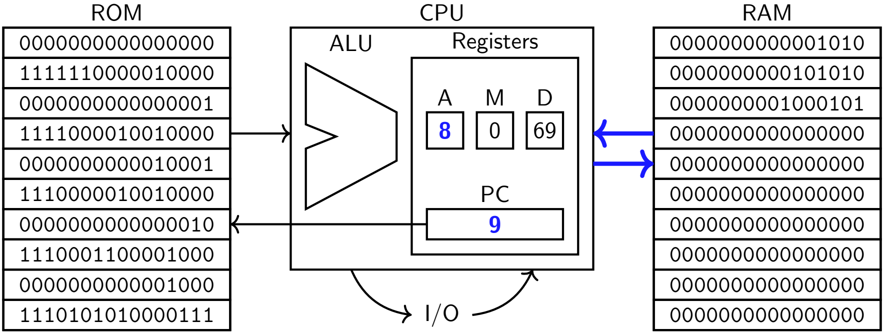


ROM	RAM
0000000000000000	0000000000001010
1111110000010000	0000000000101010
0000000000000001	0000000001000101
1111000010010000	0000000000000000
0000000000010001	0000000000000000
1110000010010000	0000000000000000
0000000000000010	0000000000000000
1110001100001000	0000000000000000
0000000000001000	0000000000000000
1110101010000111	0000000000000000

We then enter an infinite loop. All Hack programs end this way (there's no "halt" instruction).

Notice how the CPU always fetches the instruction whose address is given by the PC — manipulating the PC is how we implement loops and conditionals.

The Hack architecture



ROM

```
0000000000000000
1111110000010000
0000000000000001
1111000010010000
0000000000010001
1110000010010000
0000000000000010
1110001100001000
0000000000001000
1110101010000111
```

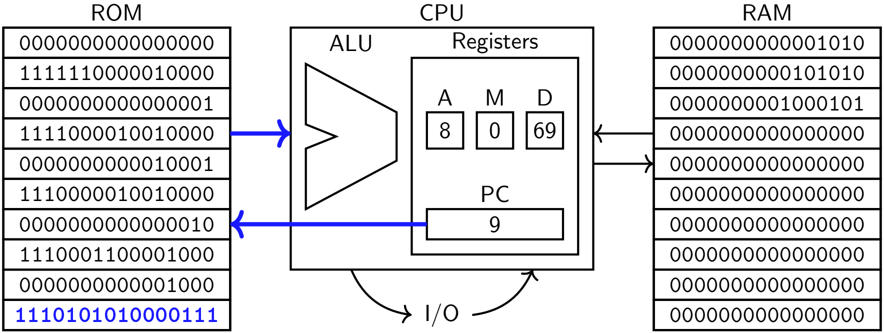
RAM

```
0000000000001010
0000000000101010
0000000001000101
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
```

We then enter an infinite loop. All Hack programs end this way (there's no “halt” instruction).

Notice how the CPU always fetches the instruction whose address is given by the PC — manipulating the PC is how we implement loops and conditionals.

The Hack architecture



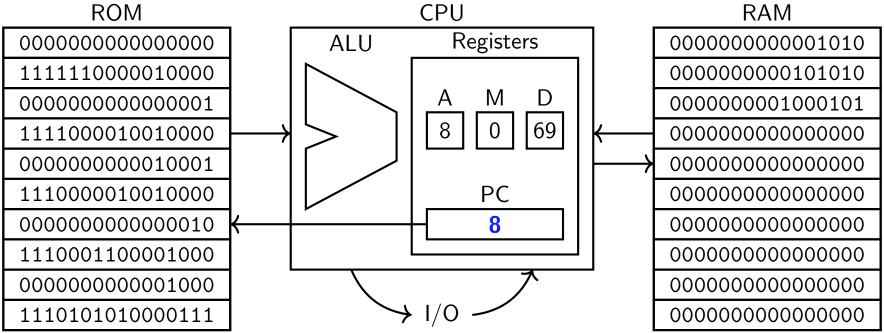
ROM
0000000000000000
1111110000010000
0000000000000001
1111000010010000
0000000000010001
1110000010010000
0000000000000010
1110001100001000
0000000000001000
1110101010000111

RAM
0000000000001010
0000000000101010
0000000001000101
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000

We then enter an infinite loop. All Hack programs end this way (there's no "halt" instruction).

Notice how the CPU always fetches the instruction whose address is given by the PC — manipulating the PC is how we implement loops and conditionals.

The Hack architecture



ROM

0000000000000000
1111110000010000
0000000000000001
1111000010010000
0000000000010001
1110000010010000
0000000000000010
1110001100001000
0000000000001000
1110101010000111

RAM

0000000000001010
0000000000101010
0000000001000101
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000

We then enter an infinite loop. All Hack programs end this way (there's no "halt" instruction).

Notice how the CPU always fetches the instruction whose address is given by the PC — manipulating the PC is how we implement loops and conditionals.

Demonstration of CPU simulator

[See video.]