

5.1

Universality

- Same **Hardware** can run many different **Software** programs

Theory



Universal Turing Machine

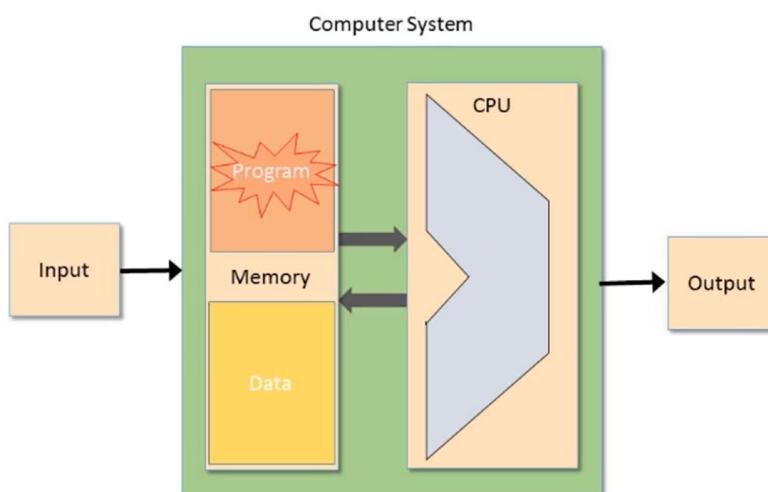
Practice



von Neumann Architecture

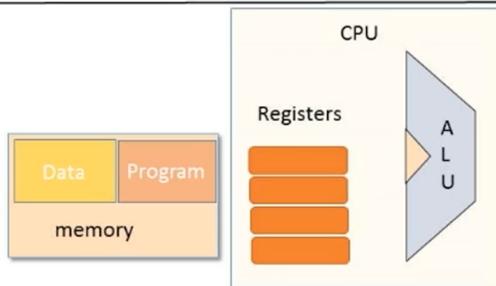


Stored Program Computer



Elements

Elements



我们的CPU将由两个主要成分组成，两个主要组件

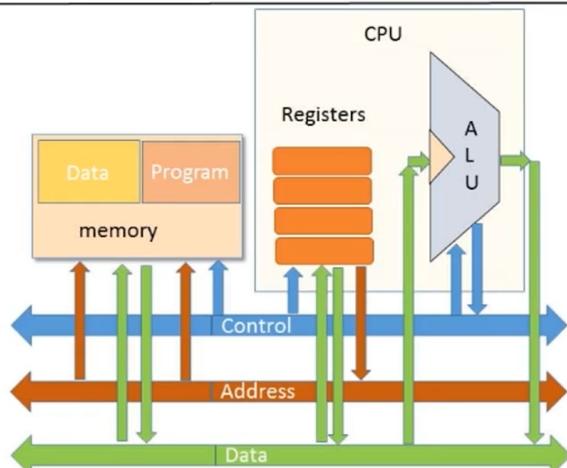
00 40 5 1:53 / 9:07 ⏪

1x ⌂ ⌂

Information Flows

- Element + 三条总线
- (大概看看，具体的看后面)

Information Flows



基本上，我想说有三种信息类型

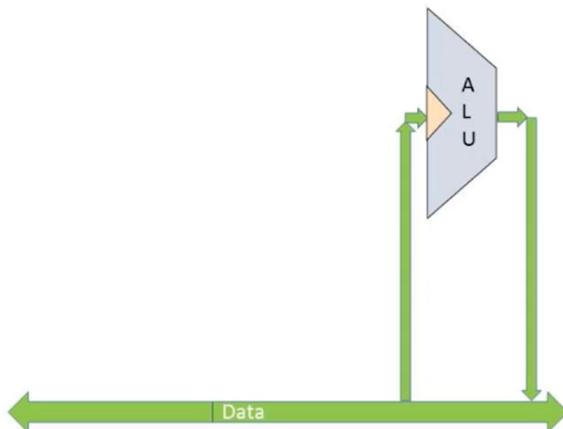
00 40 5 2:55 / 9:07 ⏪

1x ⌂ ⌂

The Arithmetic Logic Unit

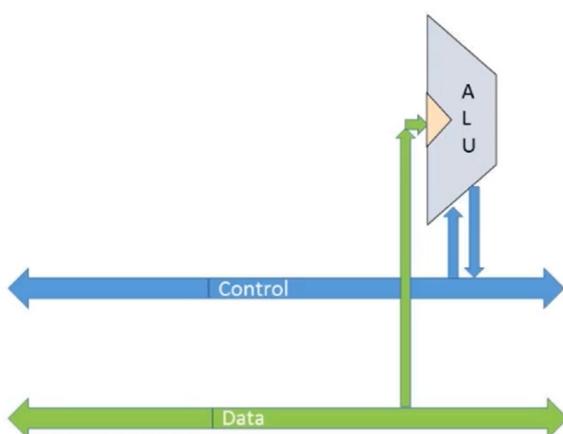
- data接入ALU，得到输出
- 输出值返回数据总线

The Arithmetic Logic Unit



- control总线 告诉ALU进行何种操作
- ALU 返回告诉 Control 系统其他部分进行什么操作
- 当 CPU 从程序内存中获取到指令后，会对指令进行解析。对于指令中的控制信息，通过控制总线传出去（让算术逻辑单元、寄存器、内存等部件按要求操作，比如让算术逻辑单元进行加法运算等）
 - 例如如果ALU发现某个数字大于零，他将告诉Control下一个指令的跳转以及下一个指令是什么

Control



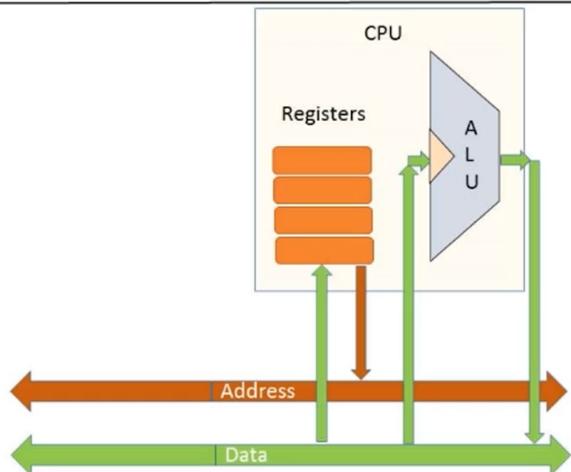
指定它要执行的操作类型



Address Register

- 寄存器储存中间结果。所以数据总线接到Register里
- 有的寄存器还用作地址储存器，所以register要接入地址总线
 - (将数字，也就是地址放入寄存器中，然后它就指定了我们想要访问的位置。)

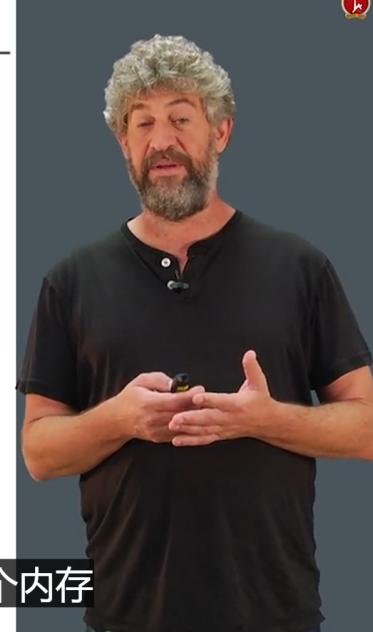
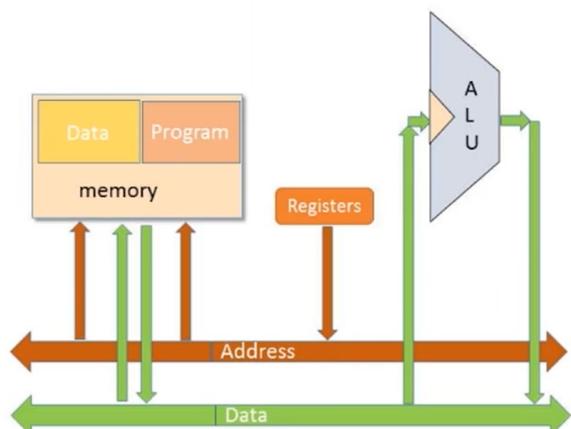
Address Registers



所以这是我们要控制的第二种信息类型

Memory

Memory

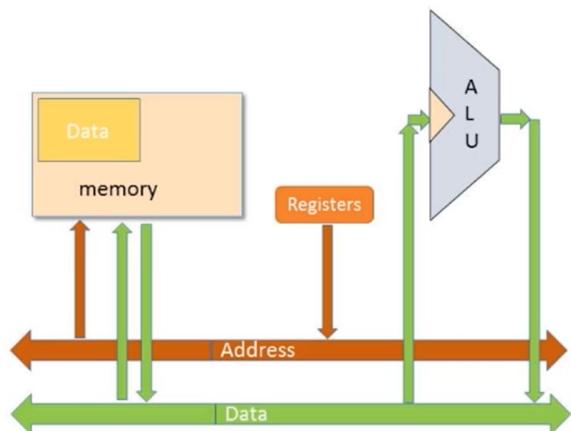


这最后我们要讨论的信息是一个内存

Data Memory

- Data Memory需要连接数据总线：因为输入和输出（这没问题）

Data Memory



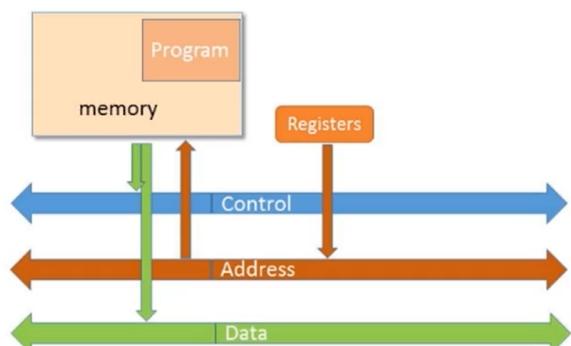
需要操作的数据块的地址。



Program Memory

- 程序内存
- 需要从PM取出CPU所需要的程序指令，所以连接地址总线
 - 例如，（**程序指令是 CPU 找内存要的**），CPU 要从程序内存里拿指令时，得先把**下一条指令**的地址发出去（通过地址总线），程序内存收到地址后，按地址找到指令再传给 CPU（通过数据总线）。

Program Memory



并把它反馈到控制总线



Next

我们在下一单元要做的，是更仔细地研究最内层的循环，也就是我们的硬件应该执行的基本操作：即从内存，程序内存中取出一条指令，并恰当地使用系统的其他部分来执行它。这被称为取指 - 执行周期，这将是下一单元的内容。 **(一条一条执行程序指令)**

5.2 The Fetch-Execute Cycle

The basic CPU loop

一条条 获取-执行 程序指令

The basic CPU loop

- **Fetch** an Instruction from the Program memory
- **Execute** it

获取一条指令，执行它，获取另一条指令，执行它，

00 0:25 / 8:18 1x

A screenshot of a video player interface. The main content area shows a slide with the title "The basic CPU loop" and a bulleted list: "• Fetch an Instruction from the Program memory" and "• Execute it". Below the slide, a subtitle in Chinese reads "获取一条指令，执行它，获取另一条指令，执行它，". At the bottom of the screen, there is a navigation bar with icons for volume, brightness, and search, and a progress bar indicating the video is at 0:25 of 8:18.

获取指令

Fetching

- Put the location of the next instruction into the “address” of the program memory
- Get the instruction code itself by reading the memory contents at that location

A screenshot of a video player interface. The main content area shows a slide with the title "Fetching" and a bulleted list: "• Put the location of the next instruction into the ‘address’ of the program memory" and "• Get the instruction code itself by reading the memory contents at that location". To the right of the slide, a video frame shows a man with a beard and curly hair, wearing a black t-shirt, speaking and gesturing with his hands. The video player has a progress bar at the bottom.

but下一条指令在哪里找呢?

- 在内存里
- 在程序内存里
- 在程序计数器 (program counter) 里面 (某个寄存器)
 - 它会根据指令的执行情况自动递增或根据跳转指令等进行修改, 以指向下一条要取的指令地址。

The Program Counter

Program Memory address

out

Instruction

Program Counter

• Normally, old_value+1
• Jump achieved by loading another value

转到下一条指令时, 我们需要操纵程序计数器, 这样

▶ ⏪ ⏴ 2:06 / 8:18 ⏵ 1x ⏹ ⏺

执行指令

- 有了Instruction, 我们需要执行它。
- 指令代码包含了很多信息 (不同位有不同信息)
- 把取出的指令通过 控制总线 输出到CPU
 - 控制总线 告诉ALU计算什么指令, 告诉数据来自哪个寄存器还是哪个数据存储器。

Executing

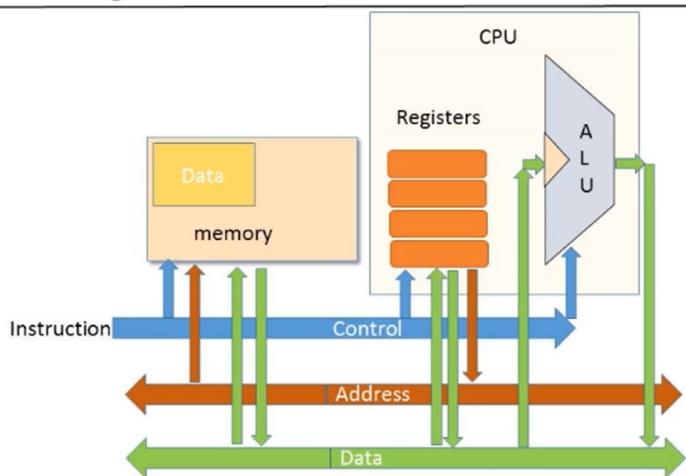
- The instruction code specifies “what to do”
 - Which arithmetic or logical instruction
 - What memory to access (read/write)
 - If/where to jump
 - ...
- Often, different subsets of the bits control different aspects of the operation
- Executing the operation involves also accessing registers and/or data memory



意味着从指定要做什么的指令代码中提取位，然后



Executing an Instruction

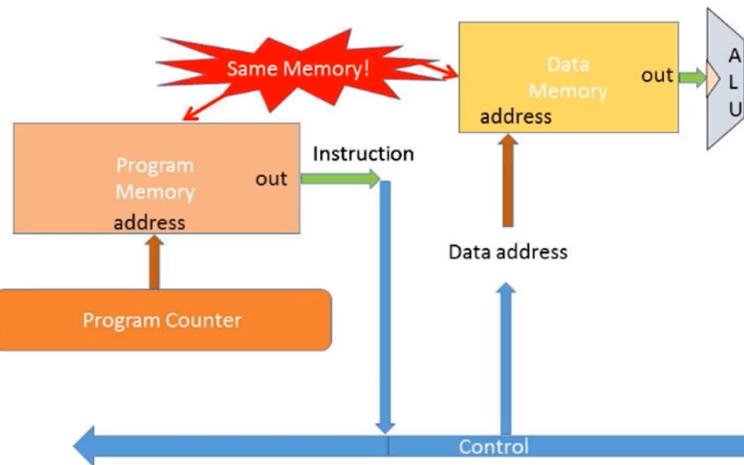


准确地告诉我们系统的每个部分在执行周期中现在要做什么。

获取-执行 冲突 Clash

- 简单来说，就是因为内存地址输入在同一时刻的唯一性，而取指令和执行指令时都需要使用内存地址来获取不同的内容（指令和数据），从而导致了冲突。

Fetch-Execute Clash



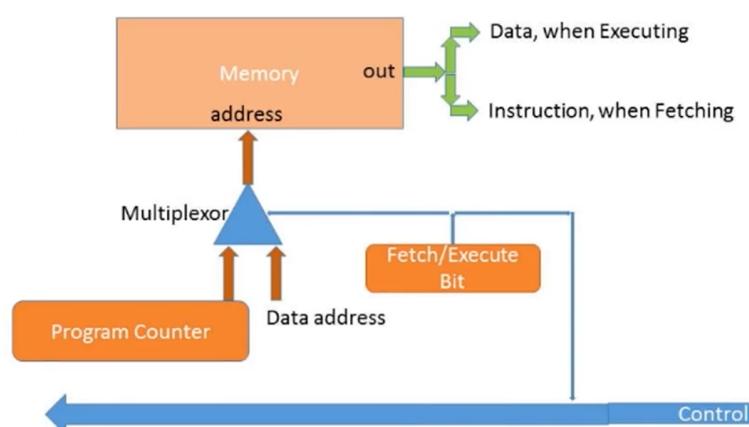
这与给我们指令的程序片段无关。

▶ ⏪ ⏴ 5:33 / 8:18 ⏵

1x ⌂ ⌂

- 解决方法：一个一个做。通过一个Mux解决 (+a Fetch/Execute Bit)

Solution: Multiplex

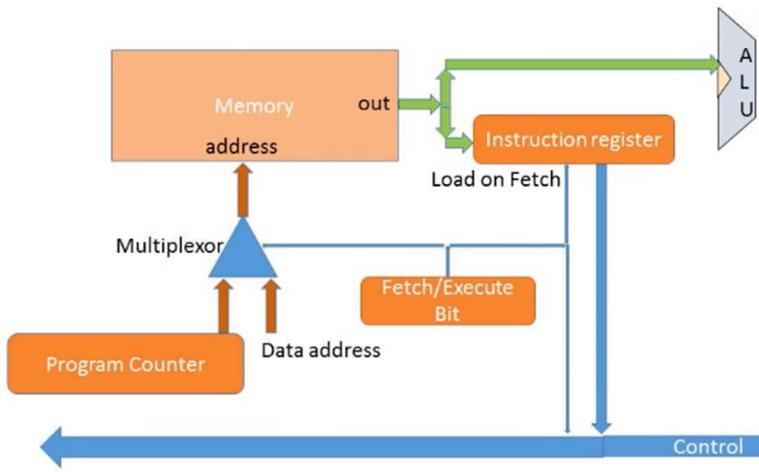


部分，又要进入系统内部执行时用来实际放置

▶ ⏪ ⏴ 6:30 / 8:18 ⏵

1x ⌂ ⌂

Instruction register



而且，我们可以



- 更简单的解决方法：把内存分为两个部分，一个部分作为数据存储器，另一个办法作为程序存储器。

Simpler Solution: Harvard Architecture

- Variant of von Neumann Architecture
- Keep Program and Data in two separate memory modules
- Complication avoided

另一个单元存放程序存储器。



Next

- 这一节，我们讨论了计算机的一般架构



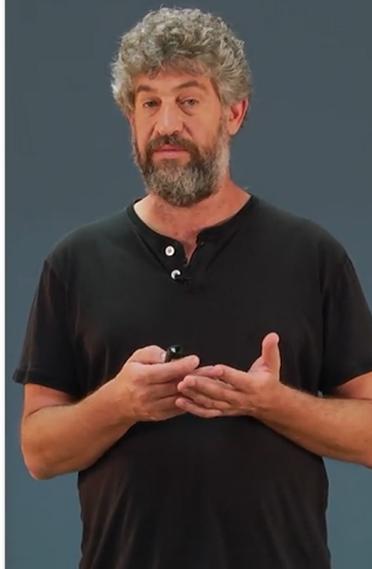
Week 5 / Unit 5.2

The Fetch-Execute Cycle

Coming Up:

The HACK Central processing Unit

计算机的一般架构。



- 下一节，我们讨论我们的HACK计算机



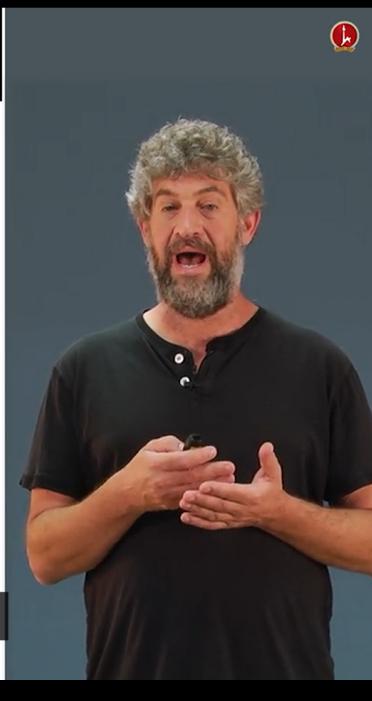
Week 5 / Unit 5.2

The Fetch-Execute Cycle

Coming Up:

The HACK Central processing Unit

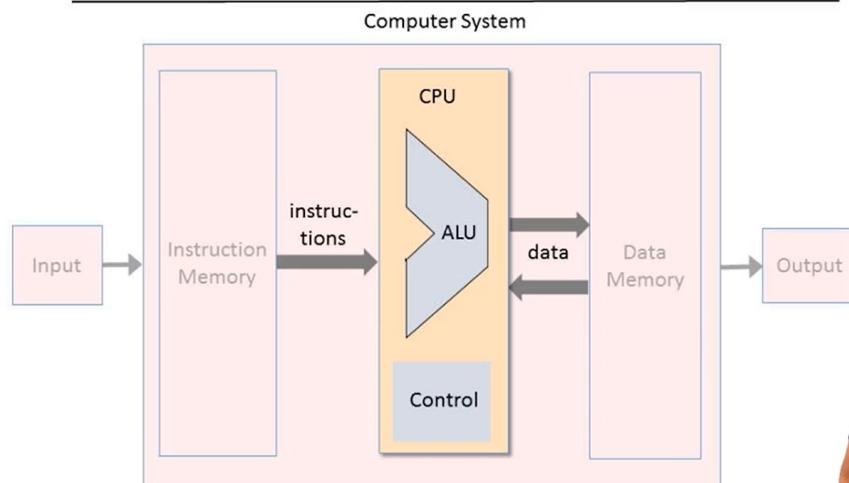
以及它究竟是如何建造的。



5.3 Central Processing Unit

CPU: interface and implementation

The Central Processing Unit



接下来应该获取和执行哪条指令的地方。

The Hack CPU: Abstraction

A 16-bit processor, designed to:

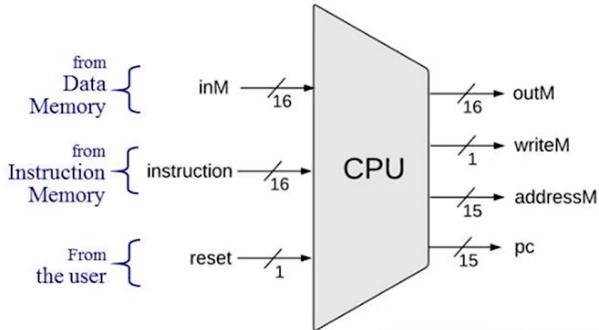


- Execute the current instruction
- Figure out which instruction to execute next
(instructions written in the Hack language)



因此，鉴于这是CPU的抽象概念，接下

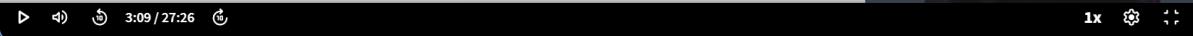
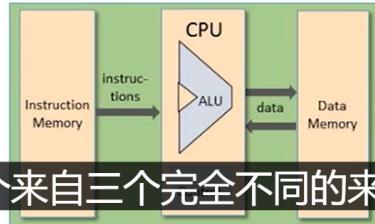
Hack CPU Interface



Inputs:

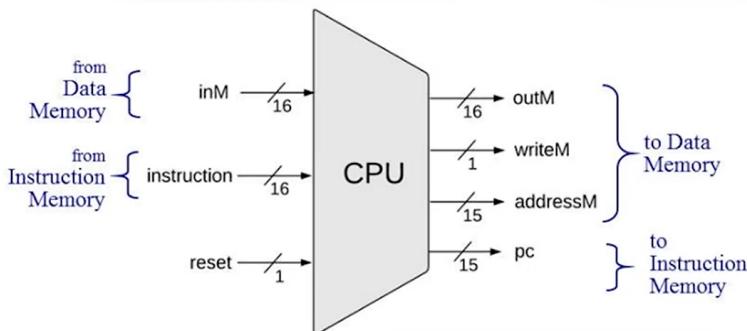
- Data value
- Instruction
- Reset bit

我们有三个来自三个完全不同的来源。



1x

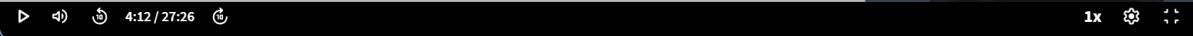
Hack CPU Interface



Outputs:

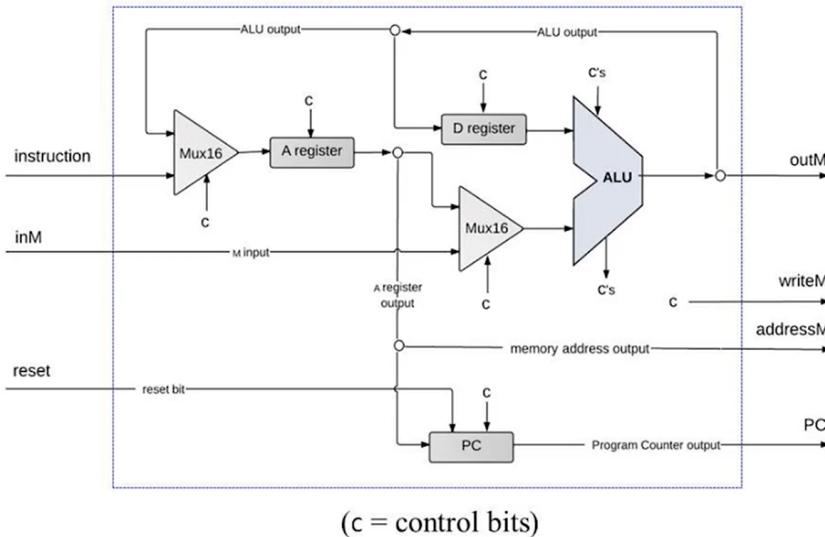
- Data value
- Write to memory? (yes / no)
- Memory Address
- Address of next ins

在右侧，我们可以看到 ALU 的输出。



1x

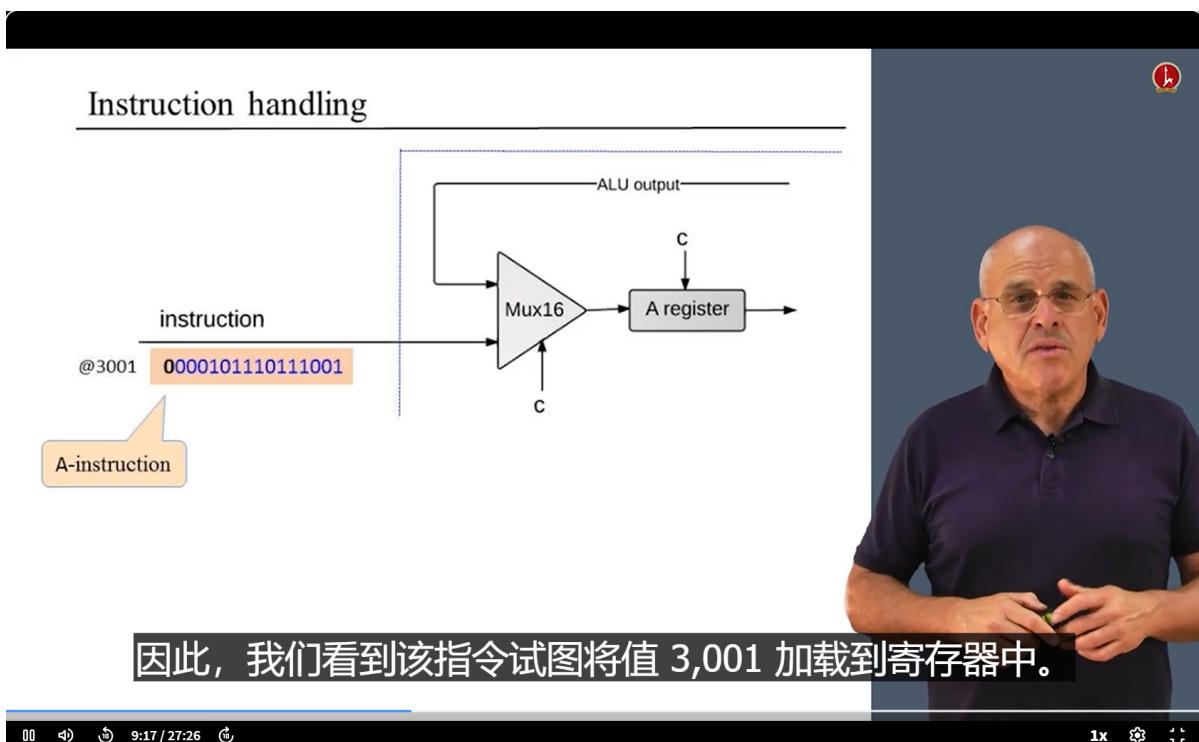
Hack CPU Implementation



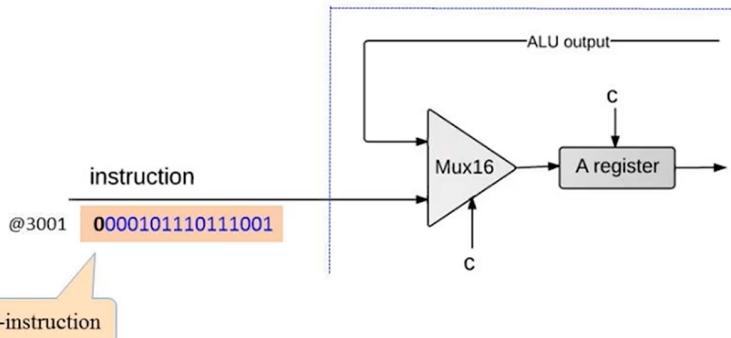
CPU: Instruction handling

handle A-instruction

- 解码：分离操作码和其他字段（15位的地址或是值）
- 确定是A指令后，取出后15位值，把它放入A寄存器中。
- CPU 同时做的另一件事：取出A寄存器的输出，并通过我们称之为“M地址”的输出将其输出到CPU外部。



Instruction handling

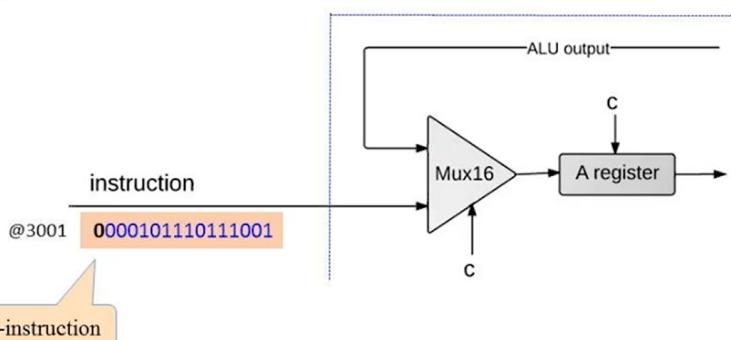


CPU handling of an A-instruction:

- Decodes the instruction into op-code + 15-bit value

首先，它必须对指令进行解码。

Instruction handling



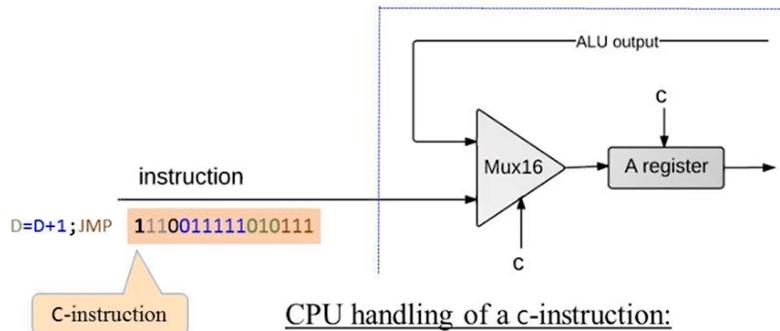
CPU handling of an A-instruction:

- Decodes the instruction into op-code + 15-bit value
- Stores the value in the A-register
- Outputs the value (not shown in this diagram)

handle C-instruction

- 解码：分离操作码和其他三个字段

Instruction handling



CPU handling of a c-instruction:

The instruction bits are decoded into:

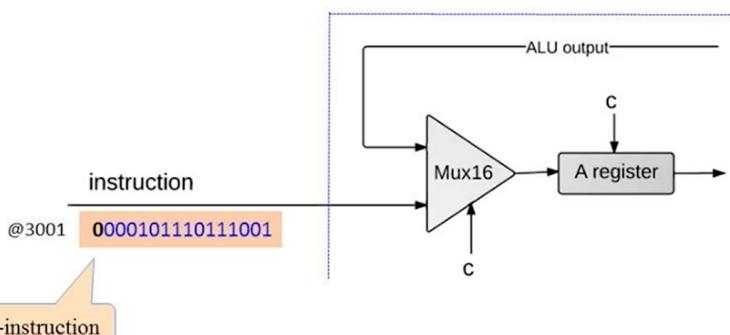
- Op-code
- ALU control bits
- Destination load bits
- Jump bits



A register -- again

- 我们发现：A寄存器也可以由 ALU的输出 提供数据，而不一定是来自指令输入。 (是不是有时候 ALU发现该跳转了，所以输出指令给A register?)
- 在某些情况下是操作码为0的A指令，在这种情况下我们希望输入来自指令。在其他情况下是操作码为1的C指令，在这种情况下，我们希望以某种方式路由A寄存器的输入，使得输入来自ALU。
 - 例如，如果 C 指令是要计算某个数值与 A 寄存器当前值的和，那么 ALU 会从 D 寄存器（或其他数据源）和 A 寄存器获取操作数进行加法运算，**运算结果可能需要再存回 A 寄存器**，所以就需要将 ALU 的输出路由到 A 寄存器作为其输入。

Instruction handling



CPU handling of an A-instruction:

- Decodes the instruction into op-code + 15-bit value
- Stores the value in the A-register
- Outputs the value from the A-register



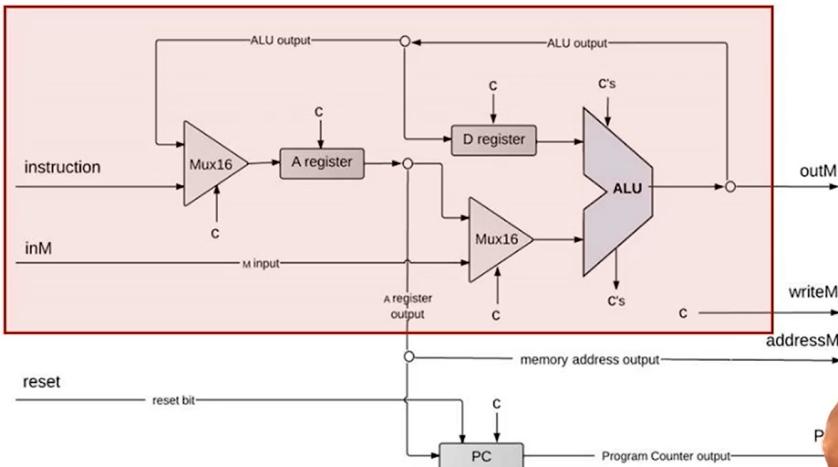
Summary

• We have seen how the ALU can be used to calculate the address of the next instruction (branching) or the sum of two memory locations (addition).

• In both cases, the ALU output was stored back into the A register.

CPU: ALU operation

ALU operation

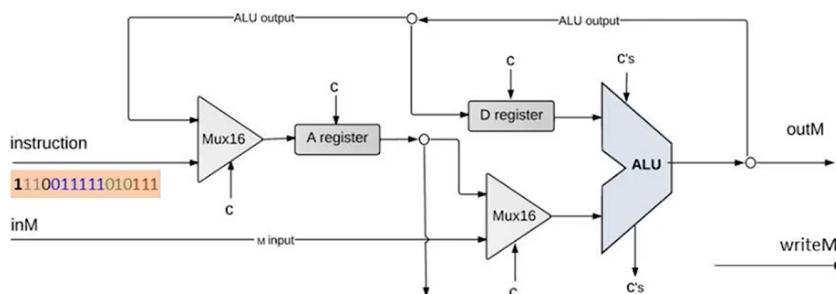


介绍下一节，我想讨论的是哪个是 ALU 的操作，

ALU operation: inputs

- remember: C指令由不同的位字段组成，每个字段都有不同的含义。

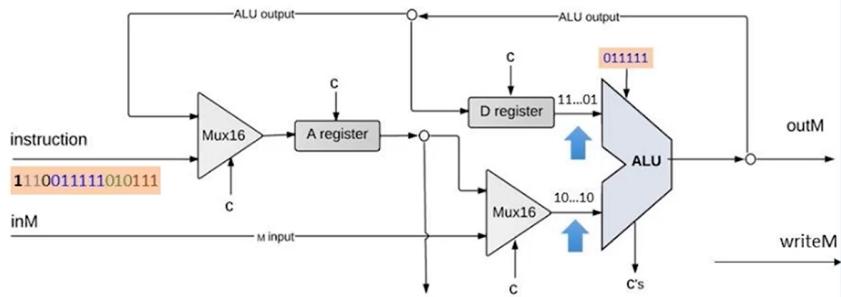
ALU operation: inputs



我们看到一个 C 指令进来，而不是我们之前看到的指令。

- ALU 有两个输入源，最终输入 depend on 控制位c和011111(now)
- Control bit **from the instruction**

ALU operation: inputs



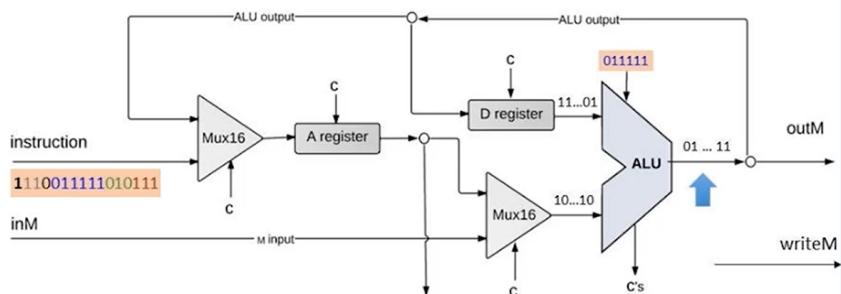
00 40 5 14:56 / 27:26 6

1x ⚡

ALU operation: outputs

- ALU 的输出被送到三个地方。

ALU operation: outputs

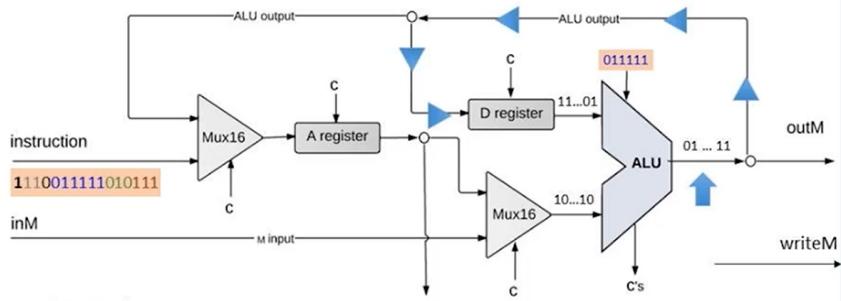


输出同时被馈送到三个不同的目的地。

00 40 5 14:56 / 27:26 6

1x ⚡

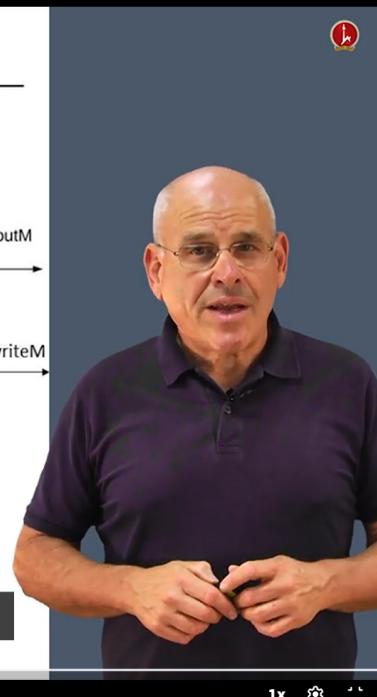
ALU operation: outputs



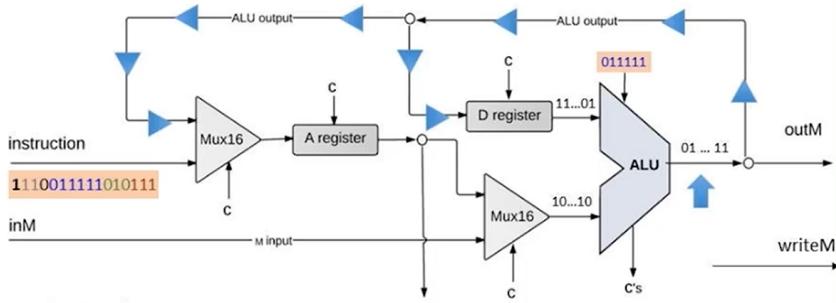
ALU data output:

- Result of ALU calculation, fed simultaneously to:
D-register, A-register, M-register

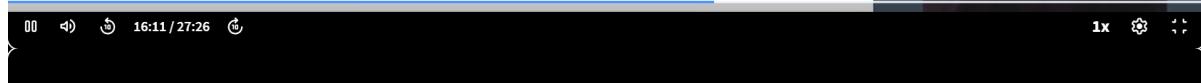
第一个目的地是 D 寄存器。



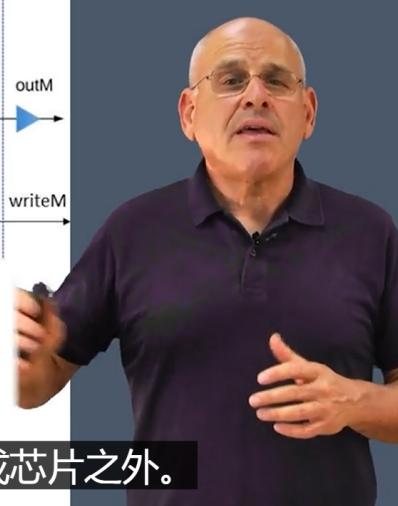
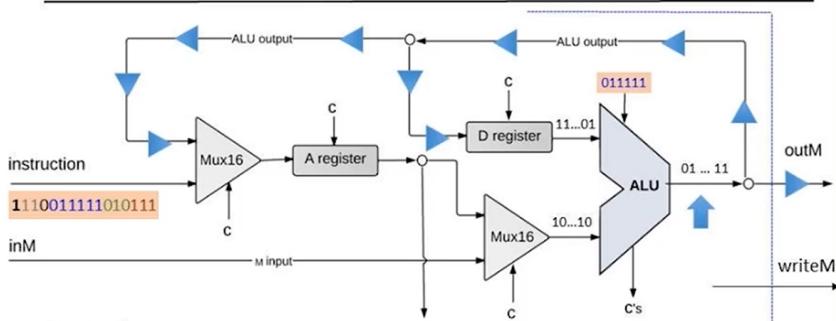
ALU operation: outputs



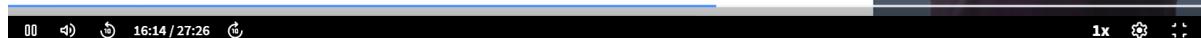
第二个目的地是 A 寄存器，它通过多路复用器。



ALU operation: outputs

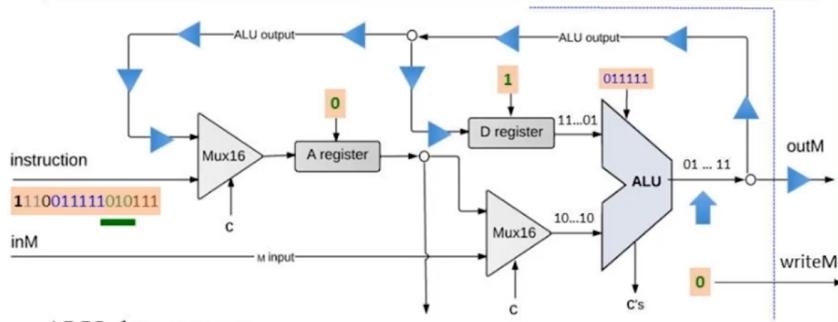


可以这么说，最终的目的地在接口或芯片之外。



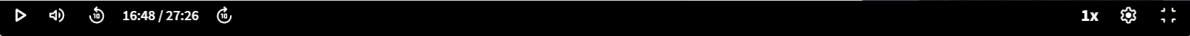
- 我们有三个目的地，这些位决定是否打开D寄存器、A寄存器和数据存储器来接受ALU的输出。
(即有选择性地接收)

ALU operation: outputs

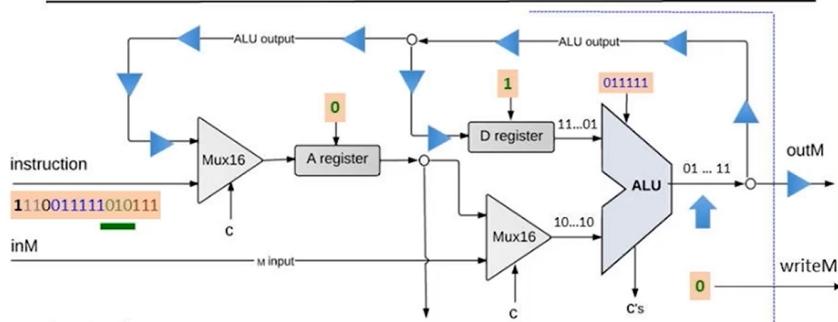


ALU data output:

- Result of ALU calculation, fed simultaneously to:
D-register, A-register, M-register
- Which register *actually* received the incoming value is determined by the instruction's **destination bits**.



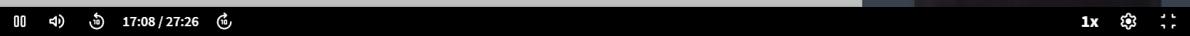
ALU operation: outputs



ALU data output:

- Result of ALU calculation, fed simultaneously to:
D-register, A-register, M-register
- Which register *actually* received the incoming value is determined by the instruction's **destination bits**.

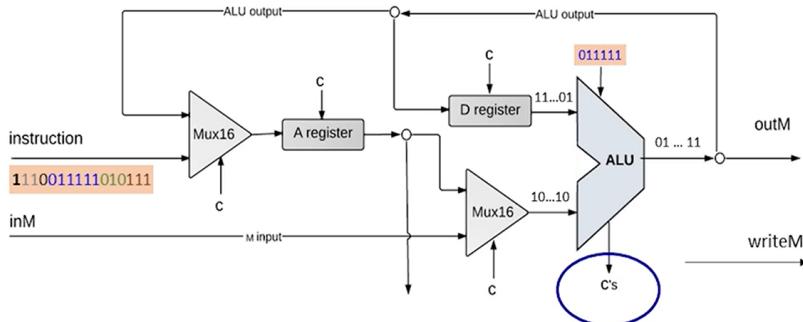
那么 ALU 计算一些东西真是太好了，但是这个值会丢失。可以@@



Control bit c's?

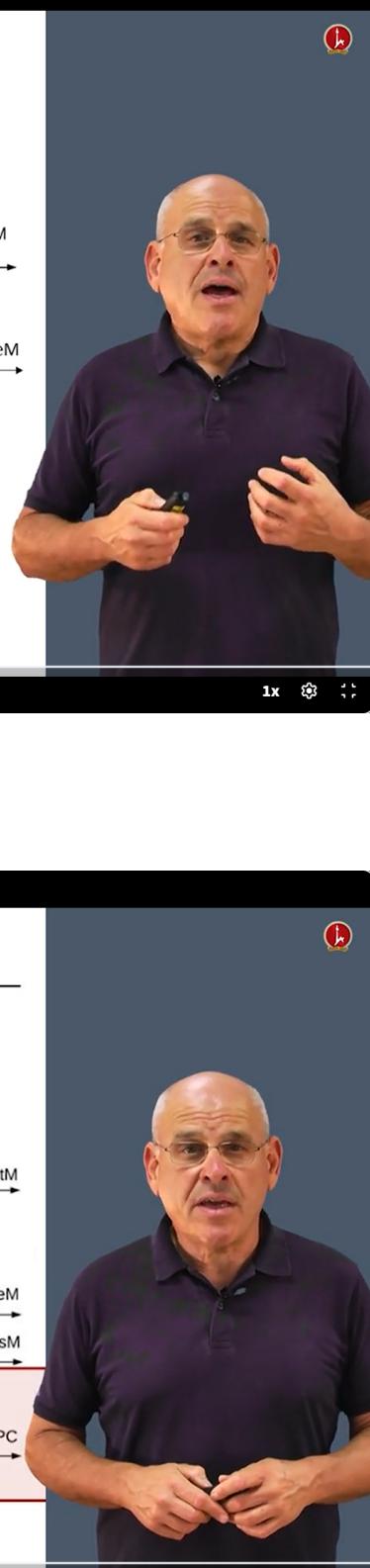
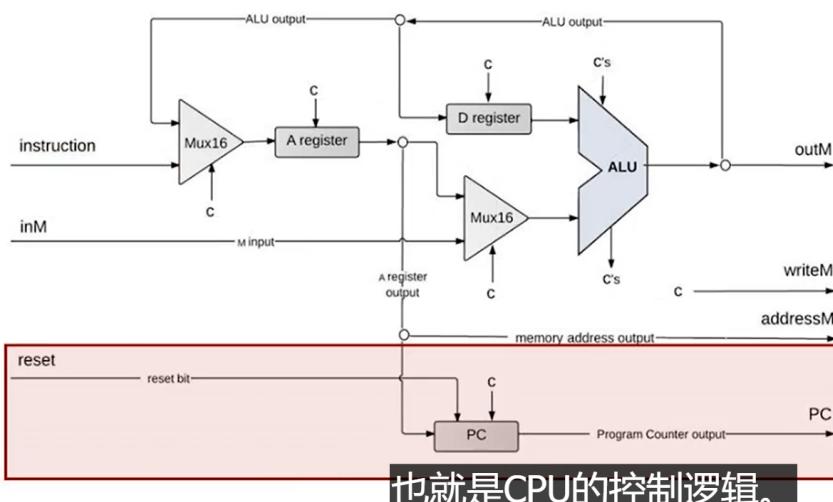
- 这两个控制位用于记录ALU的输出是负还是零。
- important for what follows.

ALU operation: outputs



CPU: Control (logic)

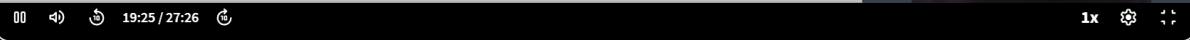
Control



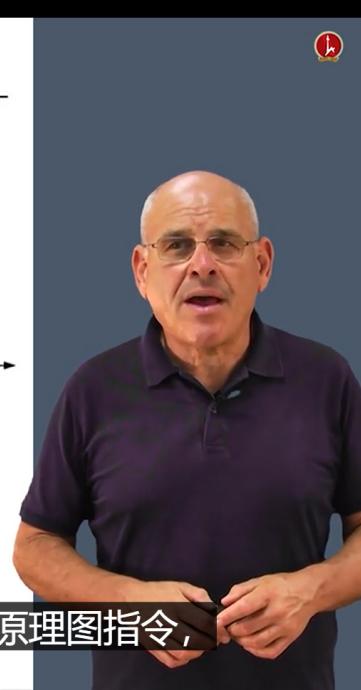
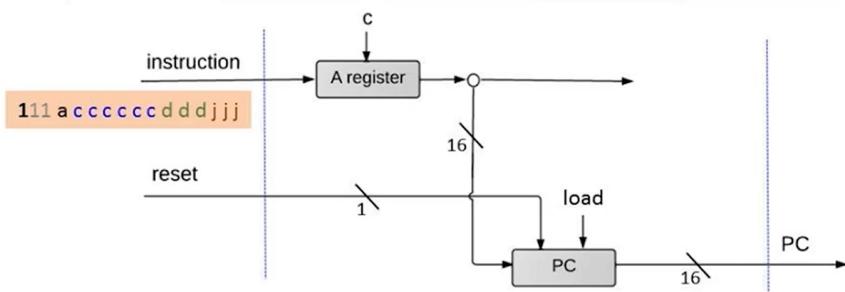
Possible outside view of the Hack computer



- The computer is loaded with some program
- When you push **reset**, the program starts running.



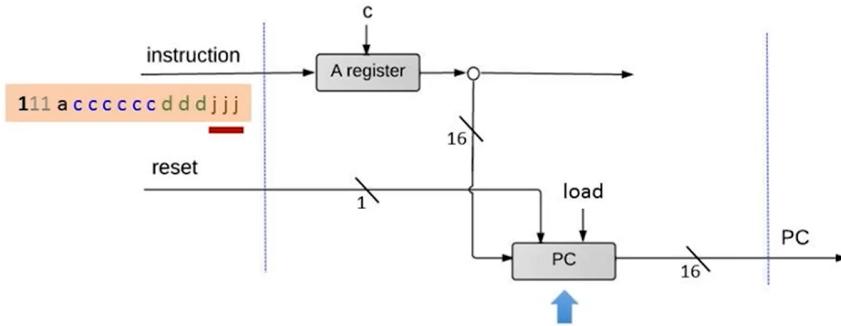
Control (abstraction)



因此，这里有一个指令的示例，它是一条原理图指令，

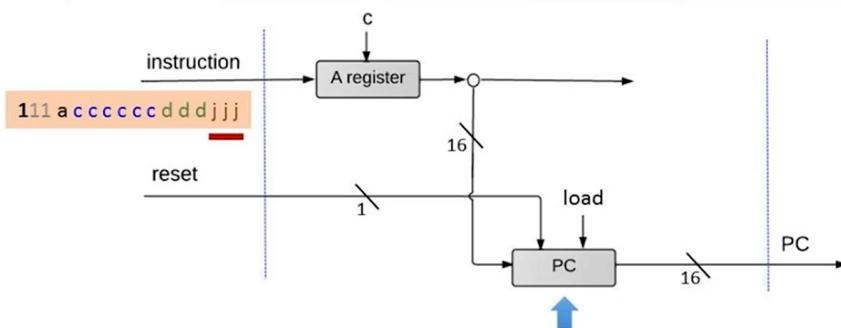


Control (abstraction)



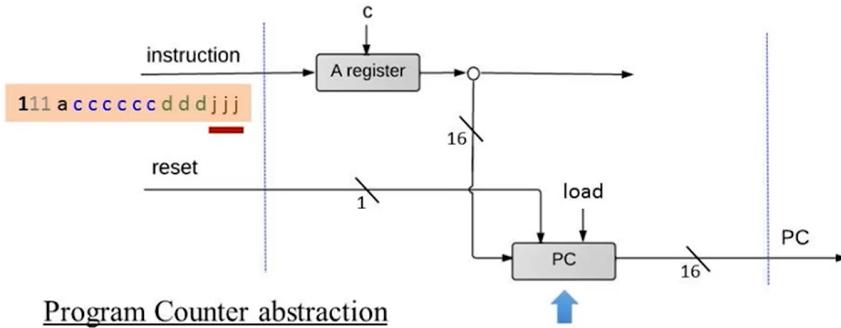
好吧，我们故事的主角是一个不起眼的寄存器，叫做程序计数器，

Control (abstraction)



它实际上是一个计数器，用我们的术语来说，程序计数器也被称为PC。

Control (abstraction)



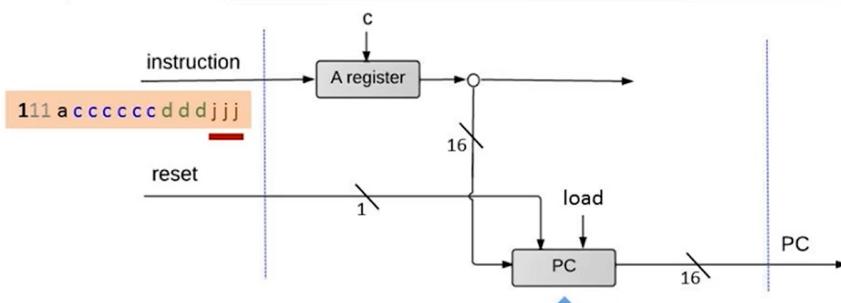
Program Counter abstraction

Emits the address of the next instruction:



做的一件事就是始终发出必须执行的下一条指令的地址。

Control (abstraction)



Program Counter abstraction

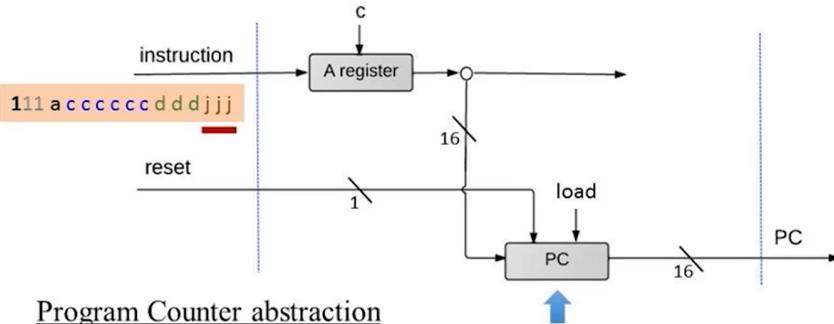
Emits the address of the next instruction:

To start / restart the program's execution: $PC = 0$



好吧，在这种情况下，我们必须将 PC 设置为零，

Control (abstraction)



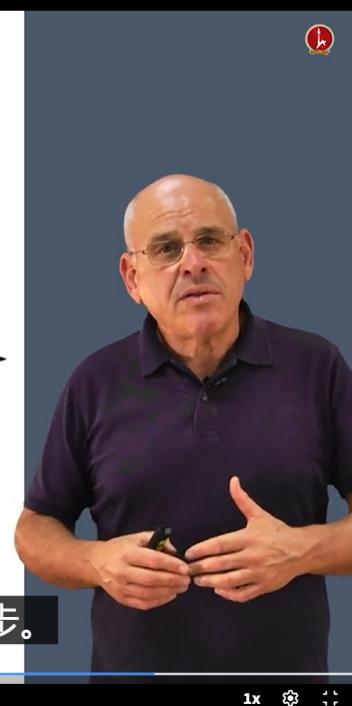
Program Counter abstraction

Emits the address of the next instruction:

To start / restart the program's execution: $PC = 0$

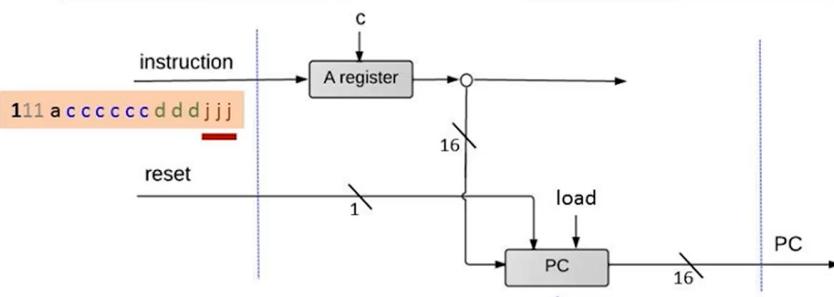
- no jump: $PC++$

然后我们希望程序计数器增加一步。



- if 程序员已经事先将想要跳转到的地址存入了A寄存器。如果我们执行PC等于A，程序计数器将输出接下来必须执行的下一条指令的地址。

Control (abstraction)



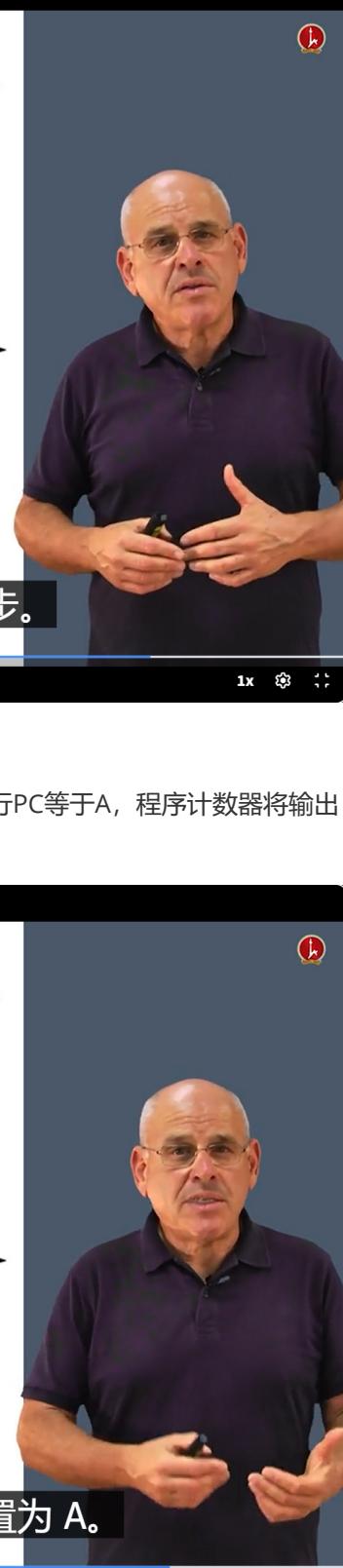
Program Counter abstraction

Emits the address of the next instruction:

To start / restart the program's execution: $PC = 0$

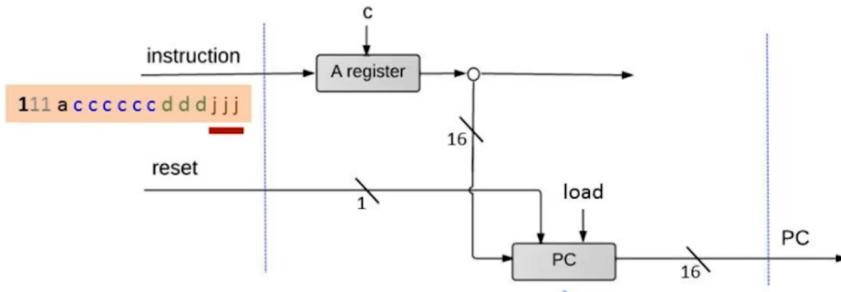
- no jump: $PC++$
- goto: $PC = A$

如果是无条件转到，我们想将 PC 设置为 A。



- 如果我们有一个条件跳转，我们必须查看ALU的输出，并决定这个跳转是否应该实际发生。
 - 例如，假设我们有一个条件跳转指令是“如果寄存器 A 中的值大于寄存器 B 中的值，则跳转到地址 X”。在执行这条指令之前，会先让 ALU 进行 $A - B$ 的运算，然后根据 ALU 的输出结果（如果结果大于 0，表示 A 大于 B）来决定是否跳转到地址 X。

Control (abstraction)



Program Counter abstraction

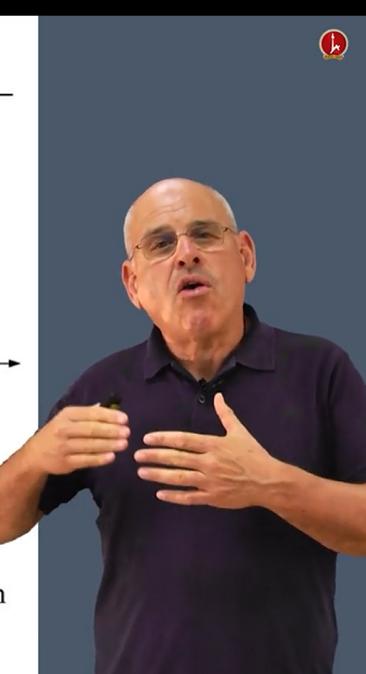
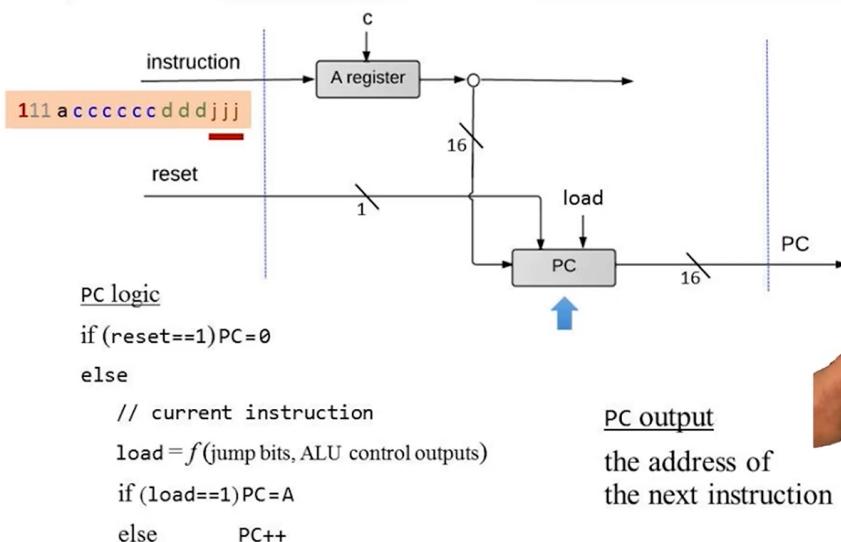
Emits the address of the next instruction:

To start / restart the program's execution: $PC = 0$

- no jump: $PC++$
- goto: $PC = A$
- conditional goto: if the condition is true $PC = A$ else $PC++$

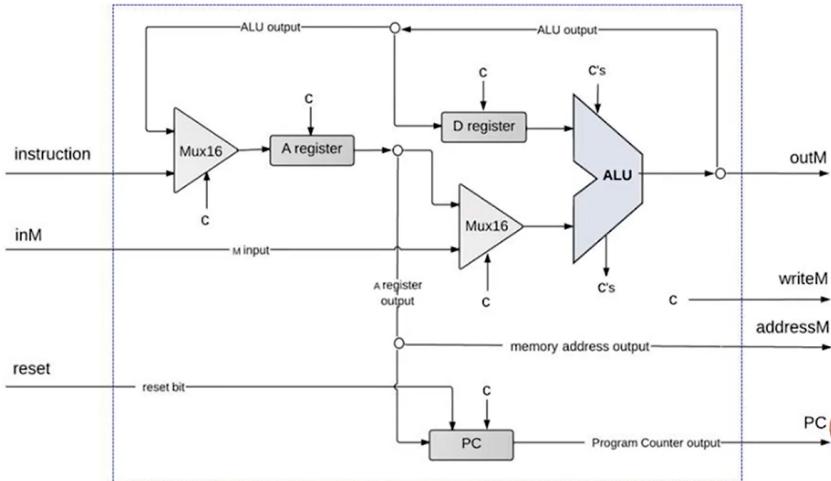


Control (implementation)



Last : we finish CPU

Hack CPU Implementation

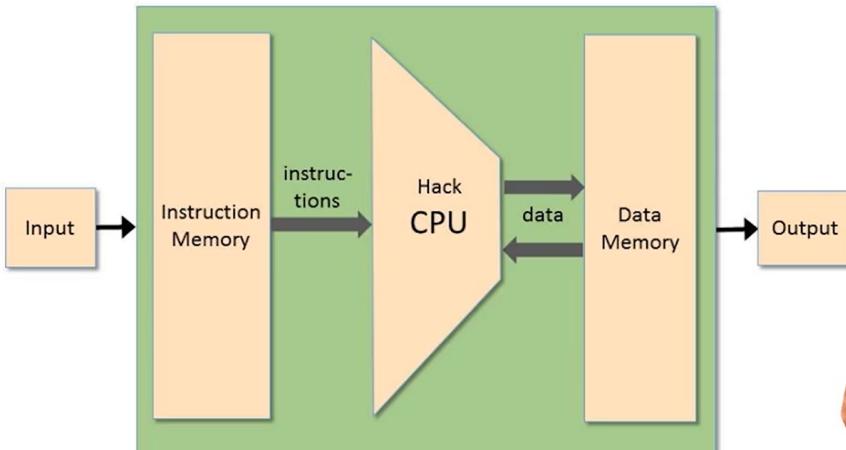


All that's left to do is actually build it.

00 26:47 / 27:26

1x

The overall computer architecture



它是迄今为止架构中最复杂、最有趣的元素。



Week 5 / Unit 5.3

Central Processing Unit

Coming Up:

The Hack Computer



不是现在，而是在下一个单位之后。



Week 5 / Unit 5.3

Central Processing Unit

Coming Up:

The Hack Computer



Hack 计算机的整体架构，然后我们将动手并

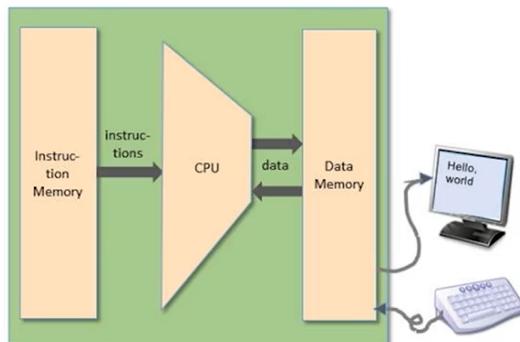


5.4 The Hack Computer

The Hack Computer

Abstraction:

A computer capable of running programs written in the Hack machine language



Implementation:

Built from the Hack chip-set.

我们也可以自下而上地描述这台计算机。

00 1:48 / 27:59

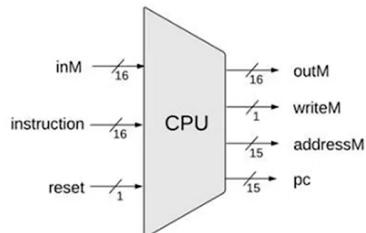
1x

Hack CPU Operation

Hack CPU Operation

Sample Hack instructions:

D = D-A
@17
M = M+1



CPU 应该如何处理这些指令？

00 2:57 / 27:59

1x

- 如果指令中提到了助记符“D”和“A”，CPU就会操作位于CPU内部的相应的**D寄存器**和**A寄存器**。
- 如果指令是一条A指令，在这种情况下，CPU会取出数据值，即这条指令中所谓的15位的“x”，然后将其放入A寄存器中。
- 如果指令的右侧(xxx=M)包含“M”，这个值将从“inM”读取。
- 如果指令的左侧(M=xxx)包含“M”，那么ALU的输出将通过“outM”输出，并且“writeM”位会 $\rightarrow 1$ 。

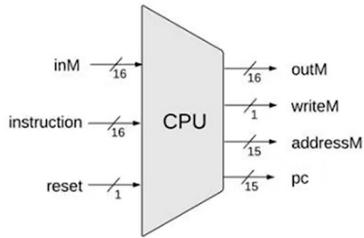
Hack CPU Operation

Sample Hack instructions:

D = D-A

@17

M = M+1



The CPU executes the instruction according to the Hack language specification:

- If the instruction includes D and A, the respective values are read from, and/or written to, the CPU-resident D-register and A-register
- If the instruction is @x, then x is stored in the A-register; this value is emitted by addressM
- If the instruction's RHS includes M, this value is read from inM
- If the instruction's LHS includes M, then the ALU output is emitted by outM, and the writeM bit is asserted.



00 4:17 / 27:59 1x

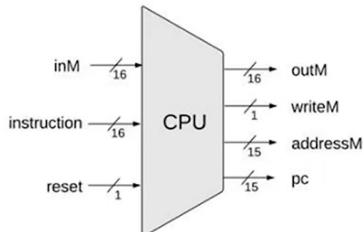
- 看看CPU如何处理跳转指令

Hack CPU Operation (continued)

Sample Hack instructions:

@100

D = D-1; JEQ



If (reset==0)

The CPU logic uses the instruction's jump bits and the ALU's output to decide if there should be a jump

If there is a jump: PC is set to the value of the A-register

Else (no jump): PC++

The updated PC value is emitted by pc

If (reset==1)

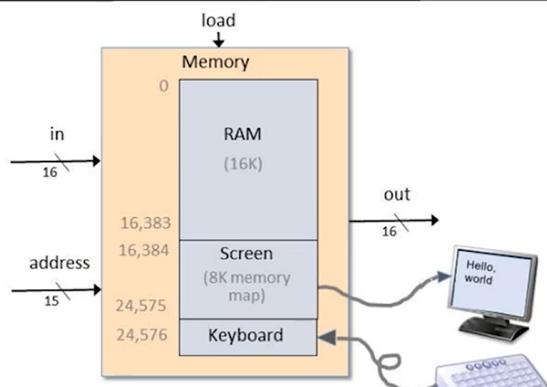
PC is set to 0, pc emits 0 (causing a program restart)



00 5:55 / 27:59 1x

Memory

Memory

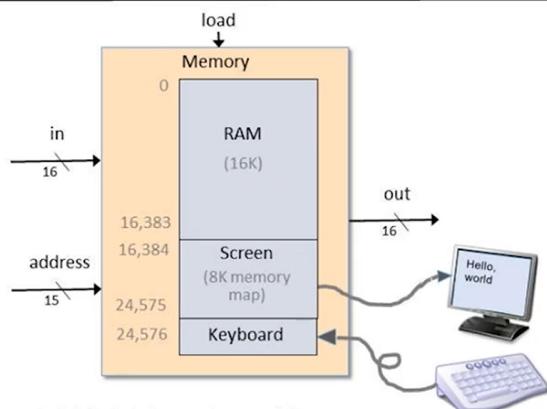


Abstraction:

- Address 0 to 16383: data memory
- Address 16384 to 24575: screen memory map
- Address 24576: keyboard memory map



Memory Implementation



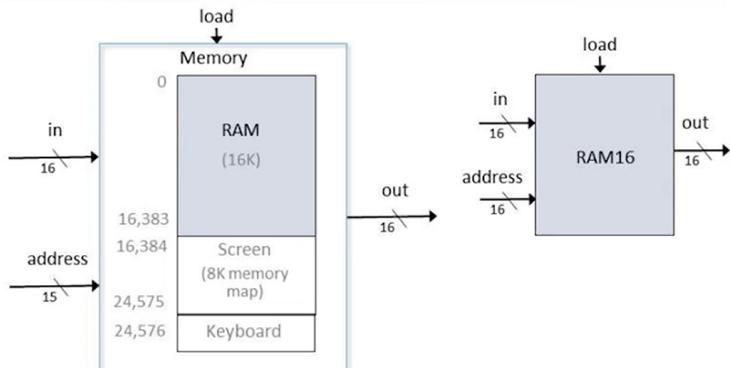
RAM: 16-bit / 16K RAM chip

Screen: 16-bit / 8K memory chip with a raster display side-effect

Keyboard: 16-bit register with a keyboard site-effect

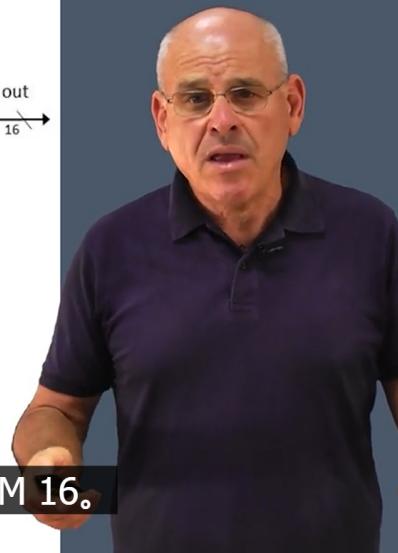


RAM

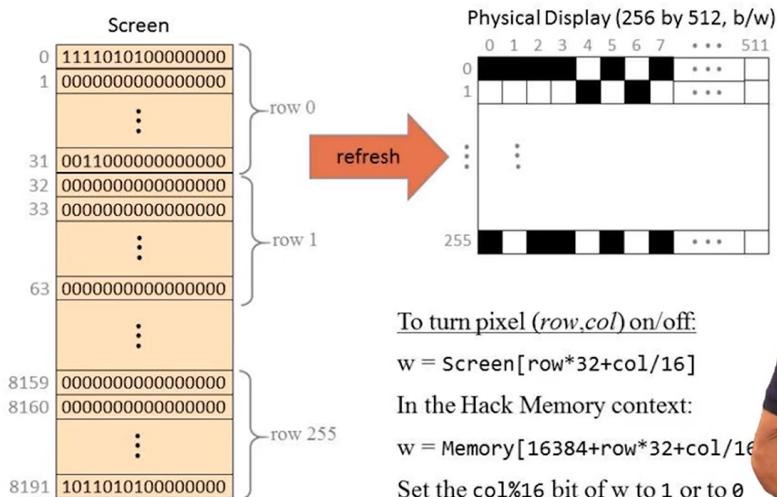


Implemented by a 16-bit, 16K RAM chip

你制造的其中一个芯片叫做 RAM 16。



Screen memory map



To turn pixel (row, col) on/off:

$w = \text{Screen}[row * 32 + col / 16]$

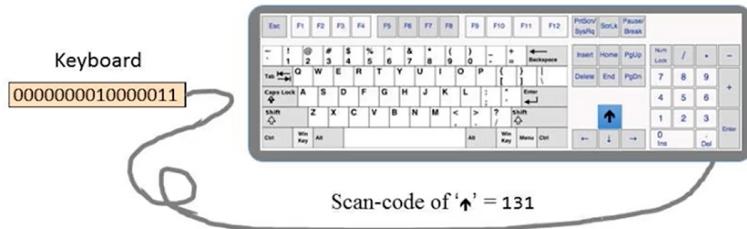
In the Hack Memory context:

$w = \text{Memory}[16384 + row * 32 + col / 16]$

Set the $\text{col} \% 16$ bit of w to 1 or to 0



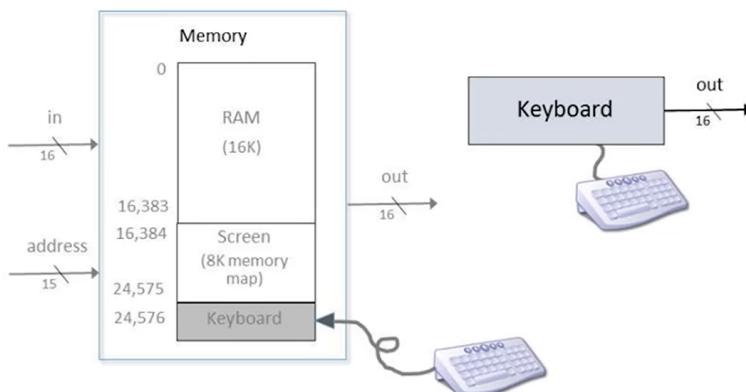
Keyboard memory map



When a key is pressed on the keyboard, the respective scan code appears in the Keyboard chip



Keyboard interface and usage



To read the keyboard:

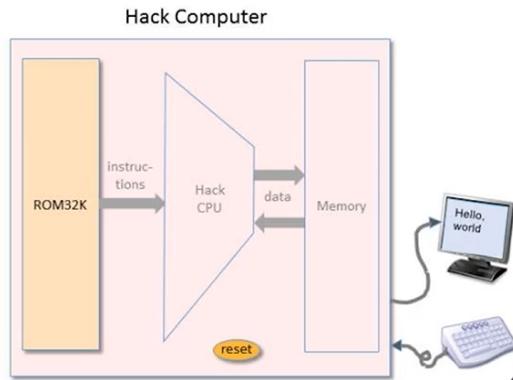
- Probe the output of the Keyboard register
- In the Hack Memory context: probe memory register 24576



- 指令存储器 Instruction Memory (ROM: Read - Only Memory)

- 在计算机运行之前，需要将编写好的 Hack 程序数据固化到 ROM32K 芯片中。
- 程序中的每一条指令都按照一定的顺序和格式存储在 ROM 的不同存储单元中，每个存储单元都有唯一对应的地址。
- 在 Hack 计算机运行时，程序计数器 (PC) 会不断地向 ROM 的地址端口提供地址，ROM 根据这些地址输出相应的指令，然后 CPU 从 ROM 的输出中获取指令并执行，从而实现整个程序的运行。
 - 例如，当 PC 输出地址为 0 时，ROM 会从地址为 0 的存储单元中取出对应的 Hack 指令，传递给 CPU 去执行，然后 PC 再递增，指向下一个地址，ROM 又输出下一条指令，如此循环，使得程序能够按顺序执行。

Instruction Memory (ROM)

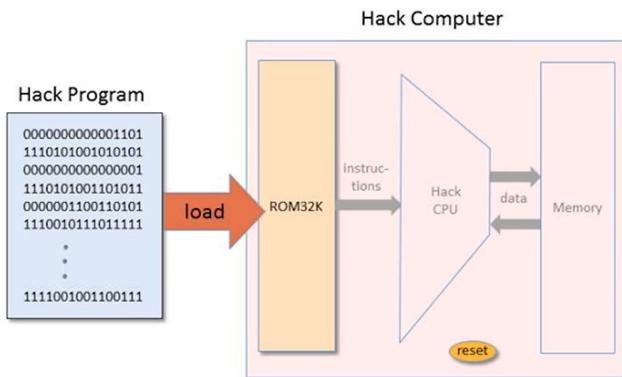


使用一种我们称之为ROM 32K的芯片来实现这个指令存储器。

00 40 5 14:27 / 27:59

1x ⚙

Instruction Memory (ROM)

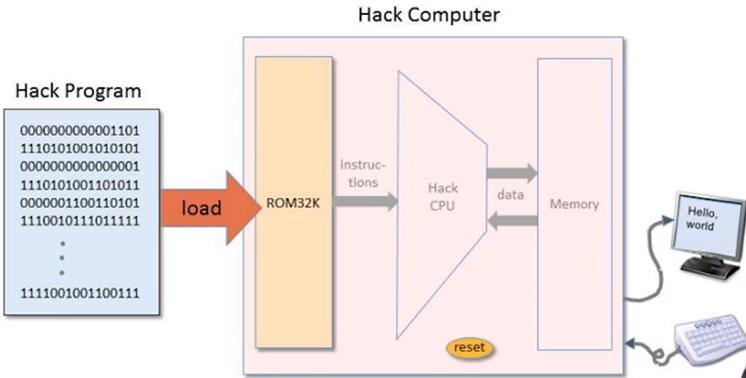


To run a program on the Hack computer:

- Load the program into the ROM

特别是，我们将其加载到ROM 32K芯片中。

Instruction Memory (ROM)



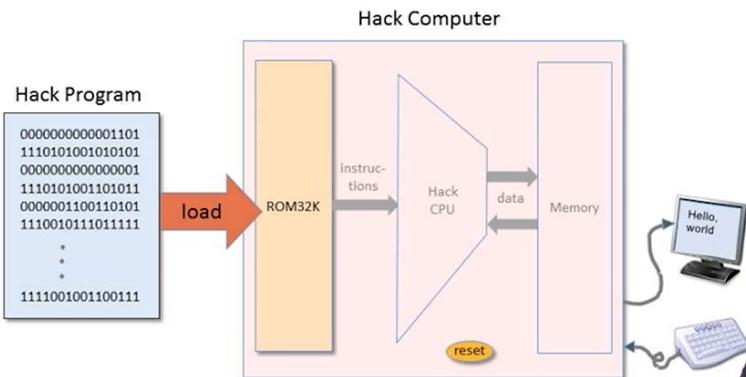
To run a program on the Hack computer:

- ❑ Load the program into the ROM
- ❑ Press "reset" **这就是我们想要实施的障碍。**
- ❑ The program starts running.



00 40 5 15:04 / 27:59 1x ⚙️ ⏹

Loading a program

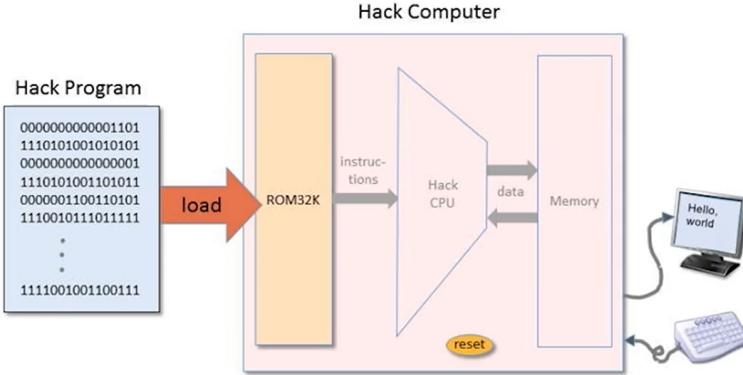


- Hardware implementation: plug-and-play ROM chips



00 40 5 16:03 / 27:59 1x ⚙️ ⏹

Loading a program

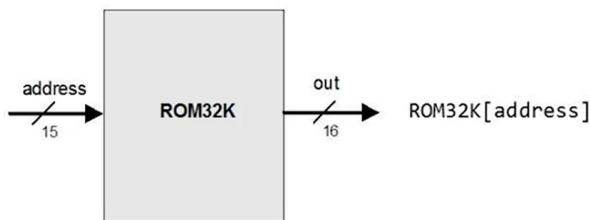


- Hardware implementation: plug-and-play ROM chips
 - Hardware simulation: programs are stored in text files;
- 实现它的另一种方法是使用硬件模拟



00 40 5 16:10 / 27:59 1x ⚙️ ⌂

ROM interface



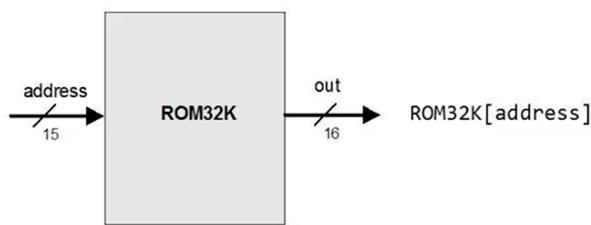
- Implemented as a built-in ROM32K chip
- Contains a sequence of Hack instructions (program)



是通过地址输入选择的寄存器的内容。

00 40 5 23:19 / 27:59 1x ⚙️ ⌂

ROM interface

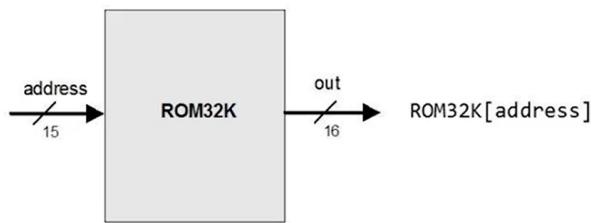


- Implemented as a built-in ROM32K chip
- Contains a sequence of Hack instructions (program)



因此，如果我在地址输入中输入 17，

ROM interface

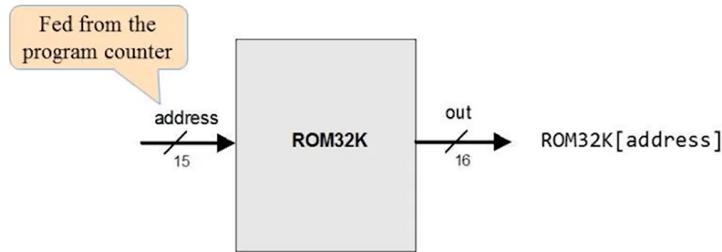


- Implemented as a built-in ROM32K chip
- Contains a sequence of Hack instructions (program)



就会出现寄存器编号为 17 的内容。

ROM interface



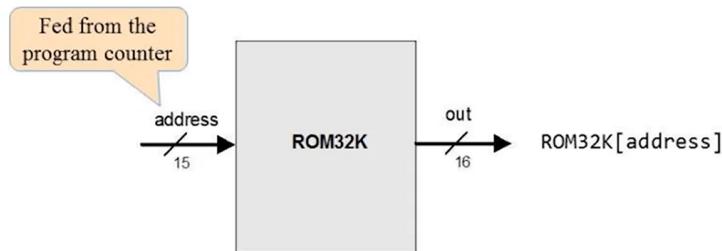
- Implemented as a built-in ROM32K chip
- Contains a sequence of Hack instructions (program)



因此，拿起这个 ROM 并将其连接到程序计数器很有意义，

00 ⏪ ⏴ 23:33 / 27:59 ⏵ 1x ⏹ ⏷

ROM interface



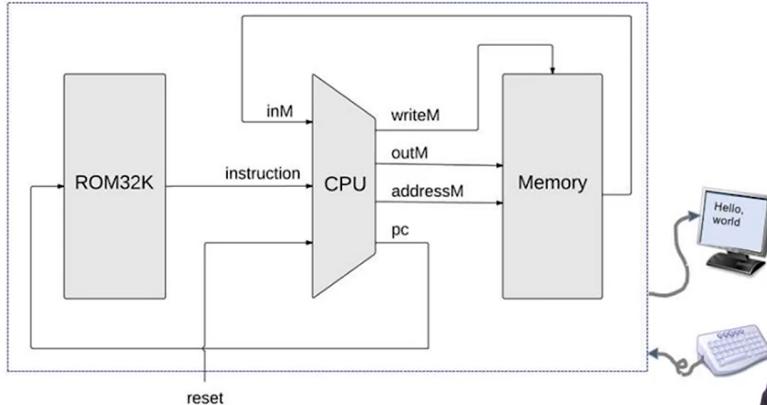
- Implemented as a built-in ROM32K chip
- Contains a sequence of Hack instructions (program)



因为程序计数器总是允许下一条指令的地址。

00 ⏪ ⏴ 23:39 / 27:59 ⏵ 1x ⏹ ⏷

Hack Computer implementation



"We ascribe beauty to that which is simple; which has no superfluous parts; which exactly answers its end; which stands related to all things; which is the mean of many extremes."

-- Ralph Waldo Emerson

