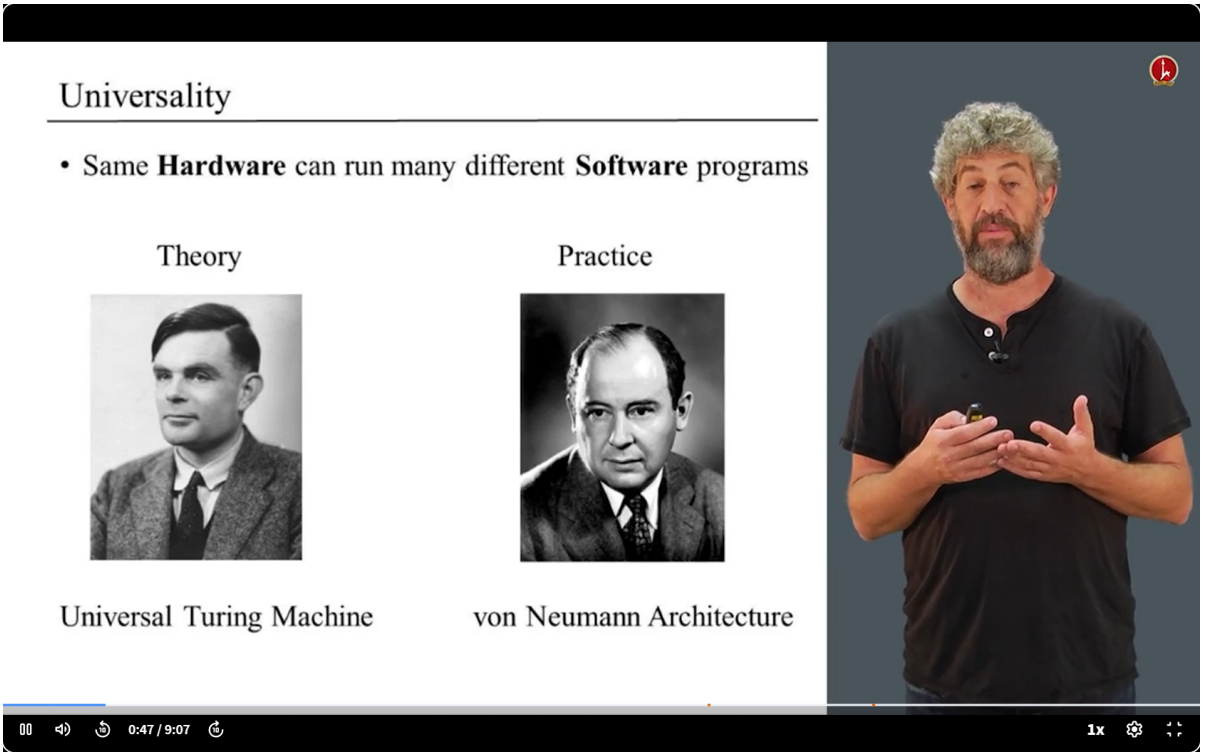
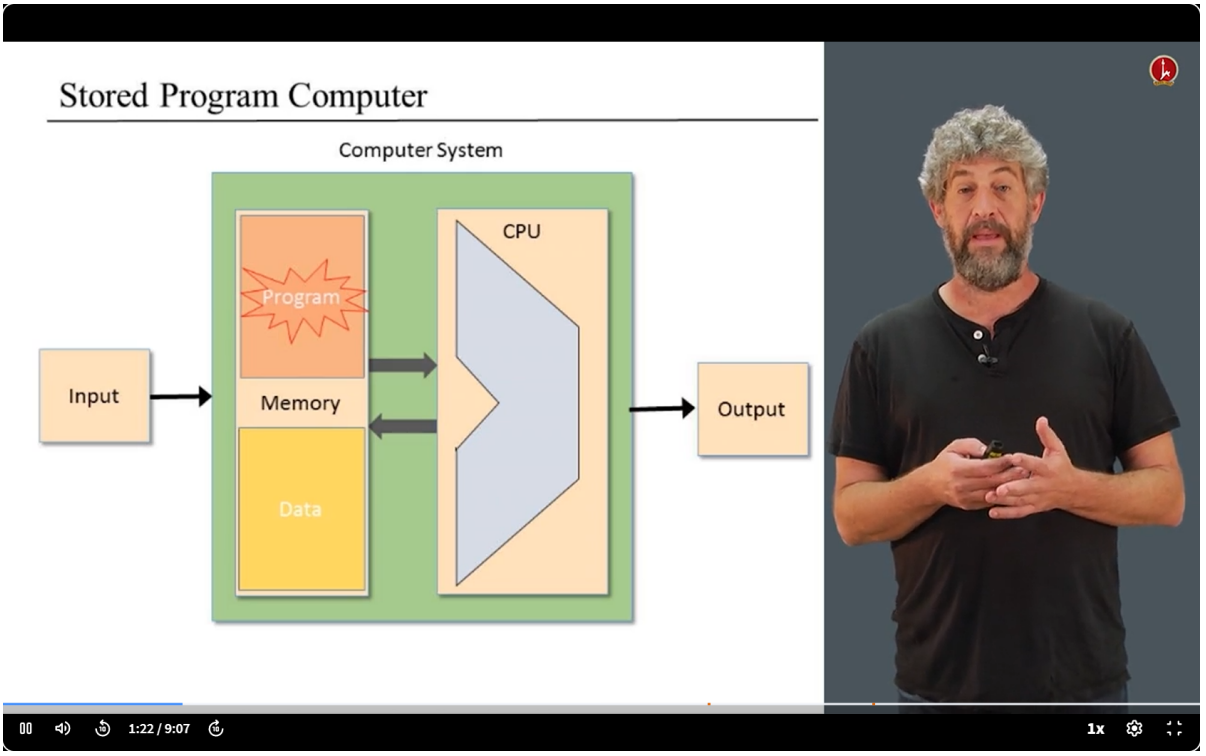
{% pdf /pdf/nand2tetris-proj5.pdf %}

5.1

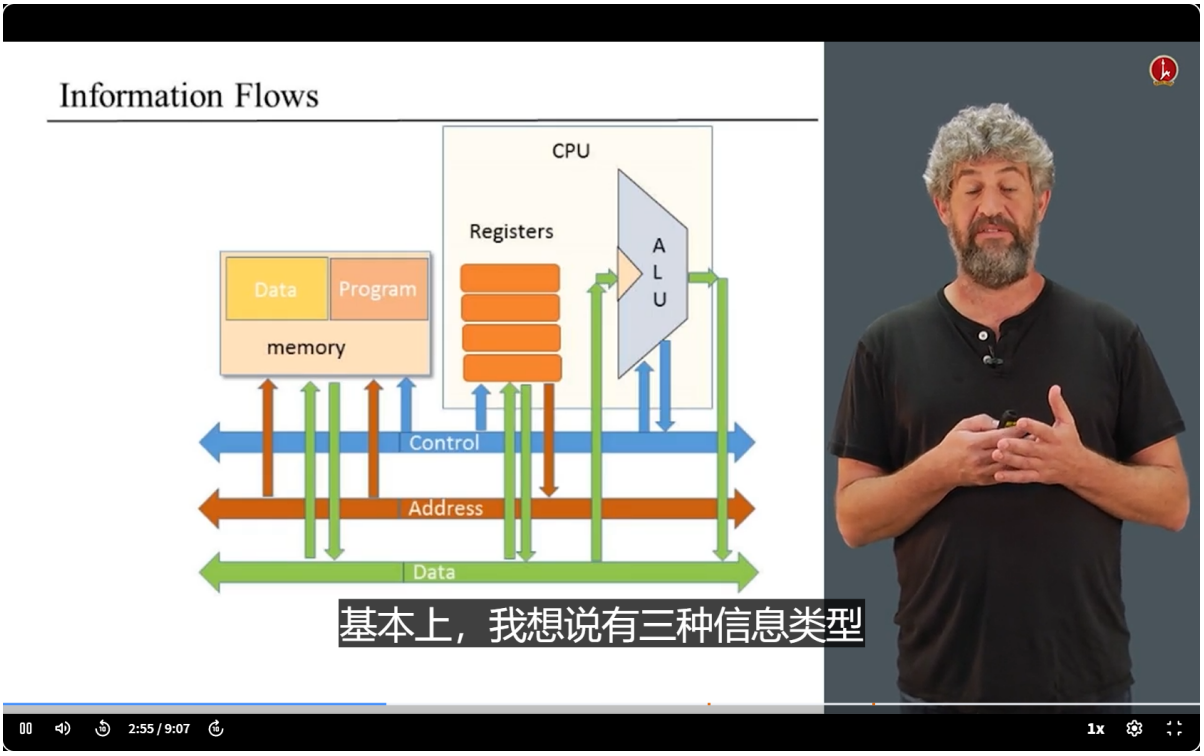




Elements

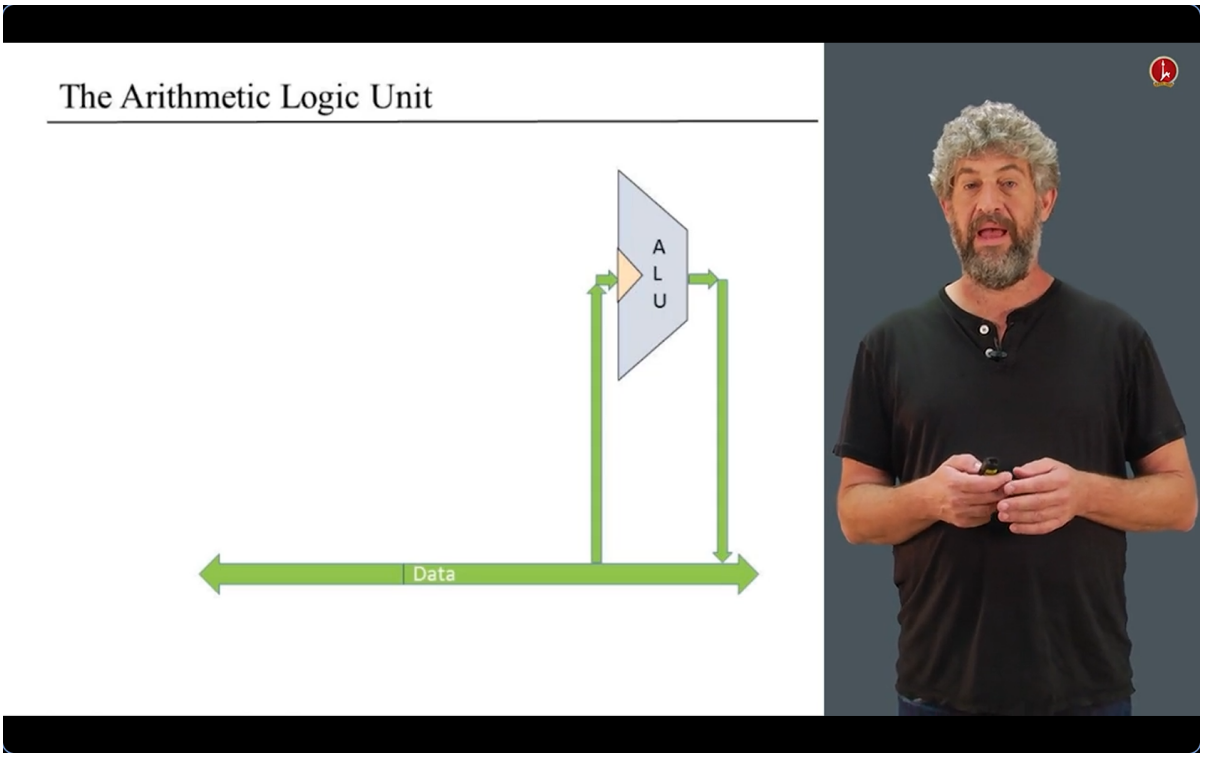
Infomation Flows

- **Element + 三条总线**
- （大概看看，具体的看后面）

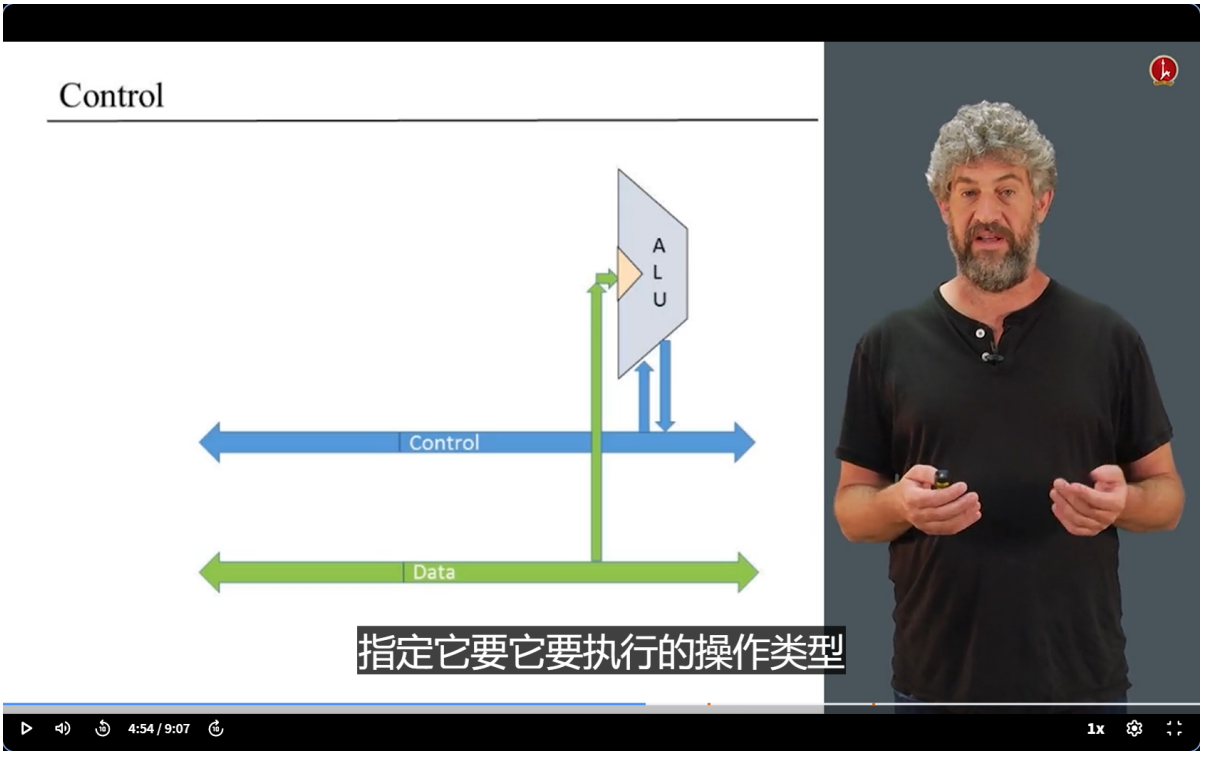


The Arithmetic Logic Unit

- data接入ALU，得到输出
- 输出值返回数据总线

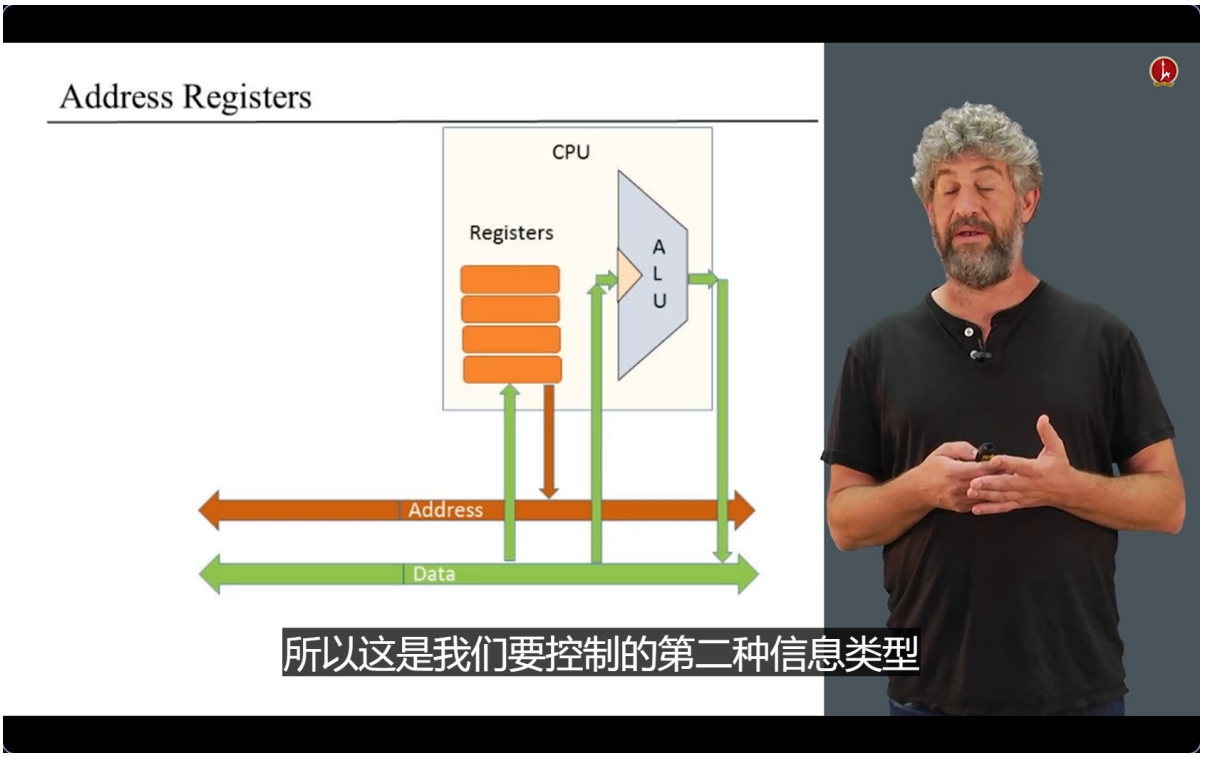


- control总线 告诉ALU进行何种操作
- ALU 返回告诉 Control 系统其他部分进行什么操作
- 当 CPU 从程序内存中获取到指令后，会对指令进行解析。对于指令中的控制信息，通过控制总线传出去（让算术逻辑单元、寄存器、内存等部件按要求操作，比如让算术逻辑单元进行加法运算等）
    - 例如如果ALU发现某个数字大于零，他将告诉Control下一个指令的跳转以及下一个指令是什么

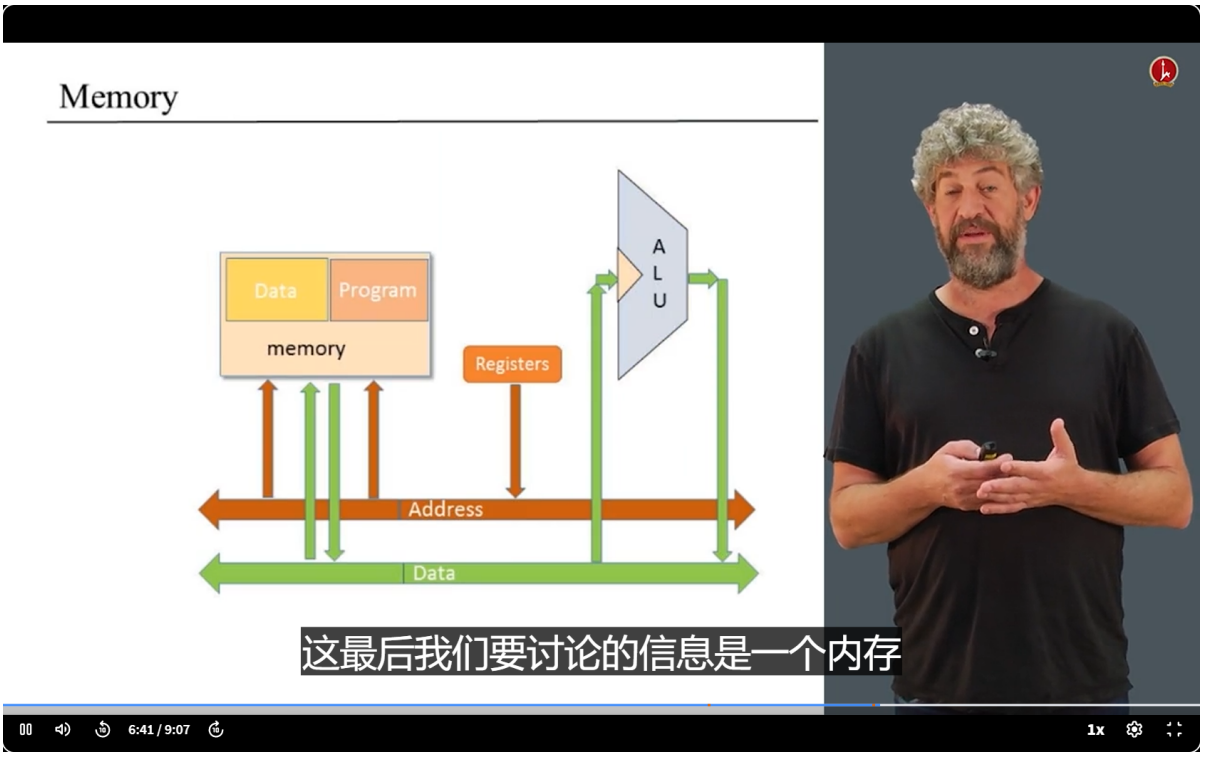


Address Register

- 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。）

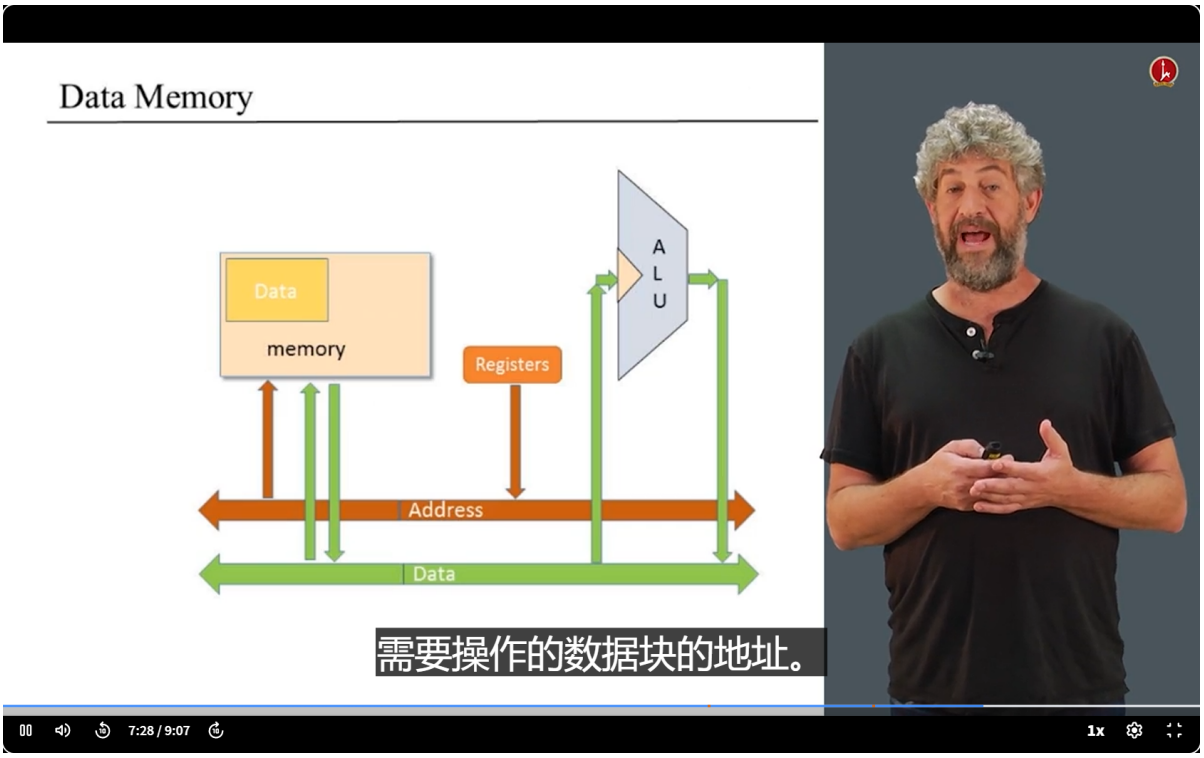


Memory



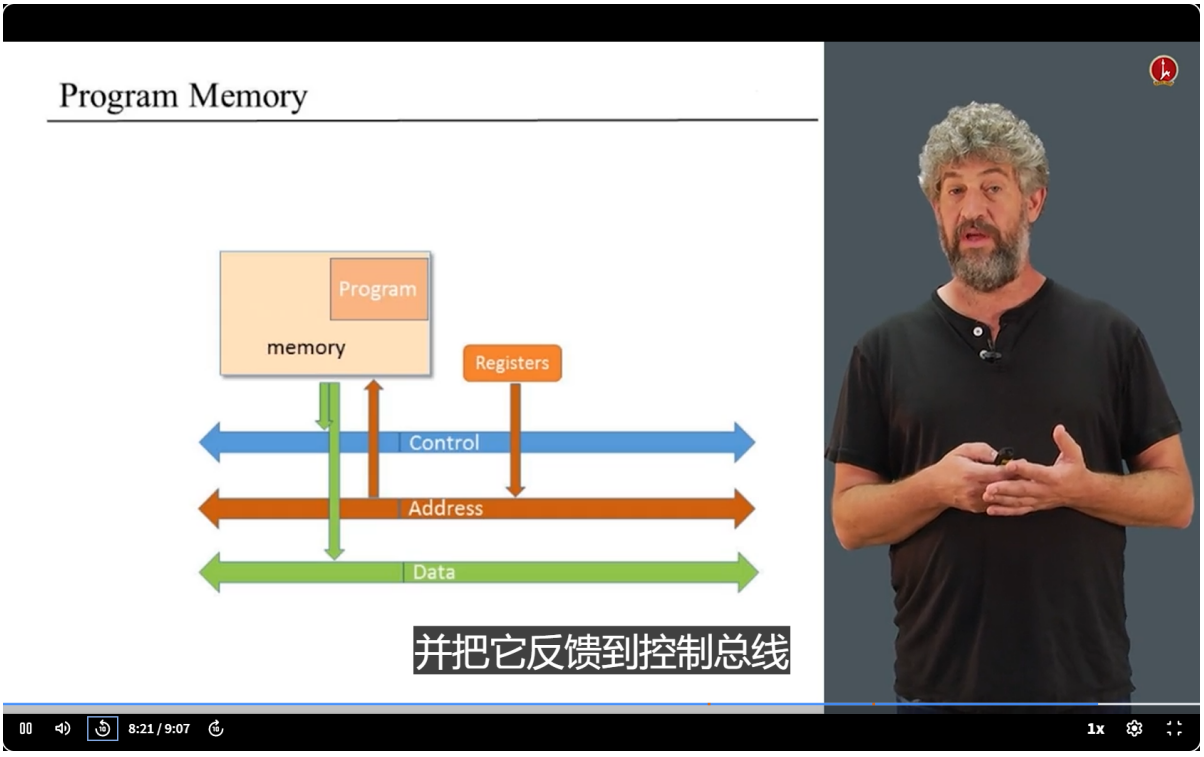
Data Memory

- Data Memory需要连接数据总线：因为输入和输出（这没问题）



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）。



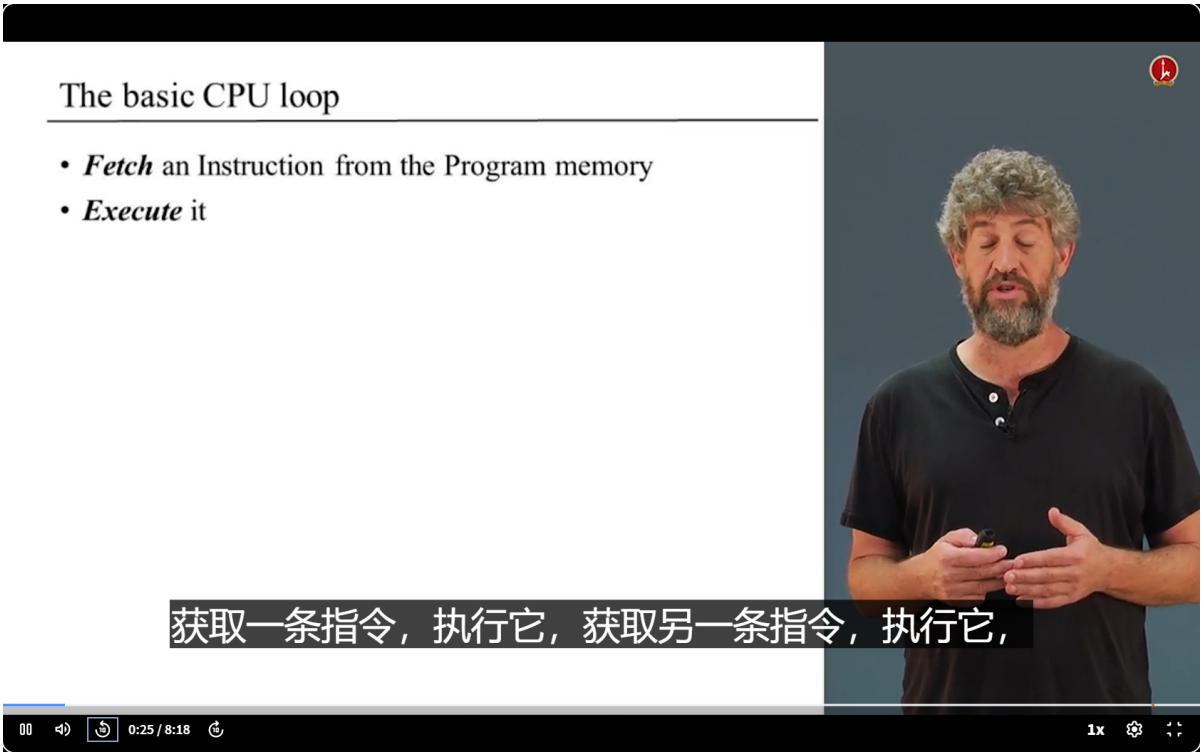
Next

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）**

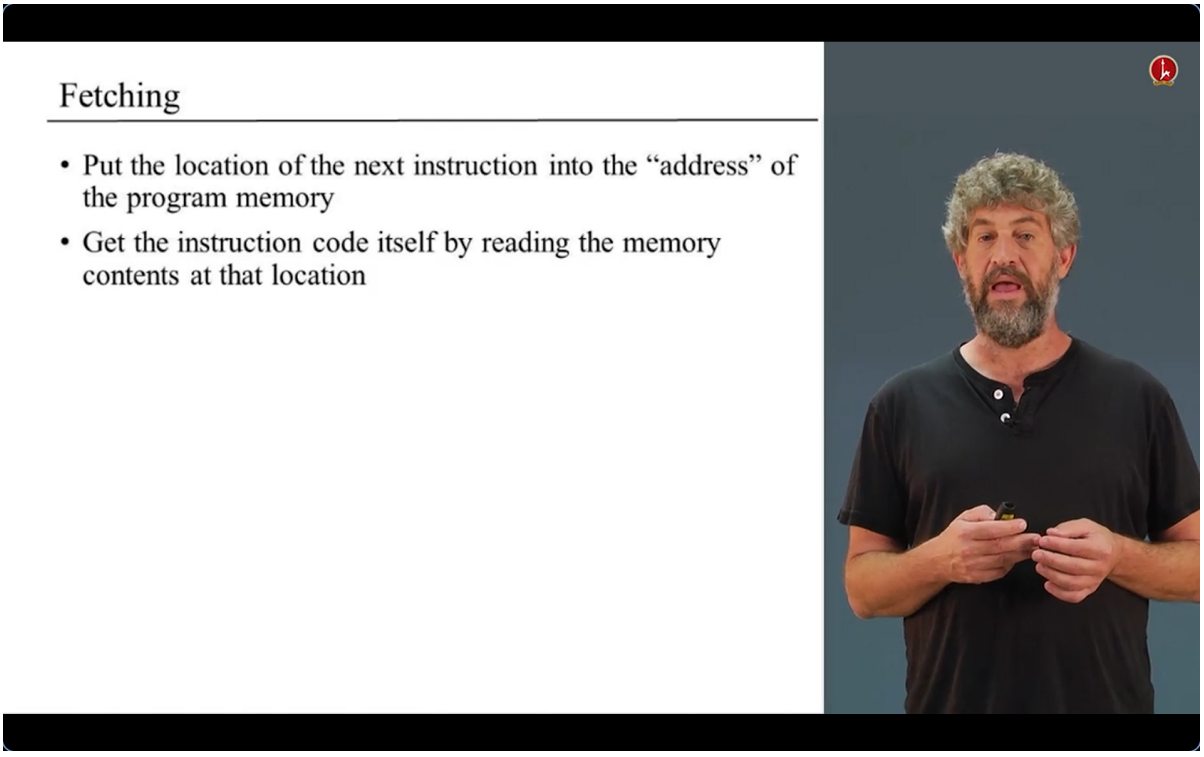
5.2 The Fetch-Execute Cycle

The basic CPU loop

一条条 获取-执行 程序指令

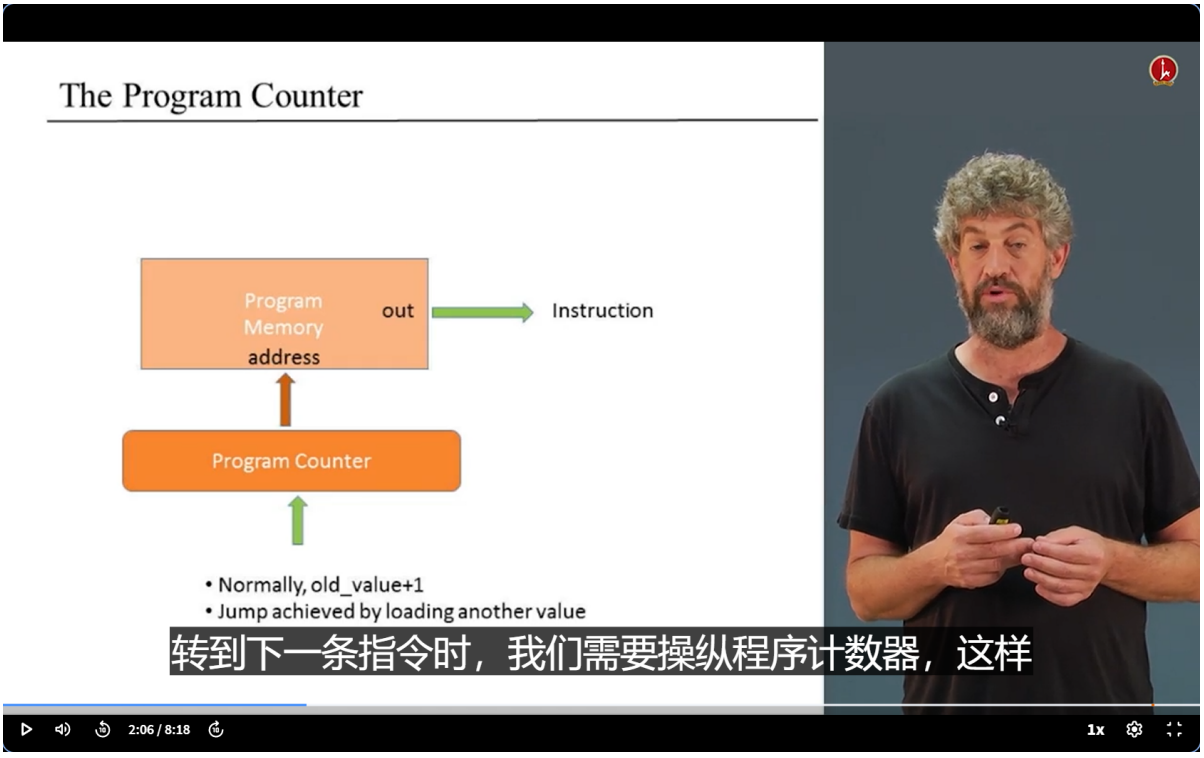


获取指令



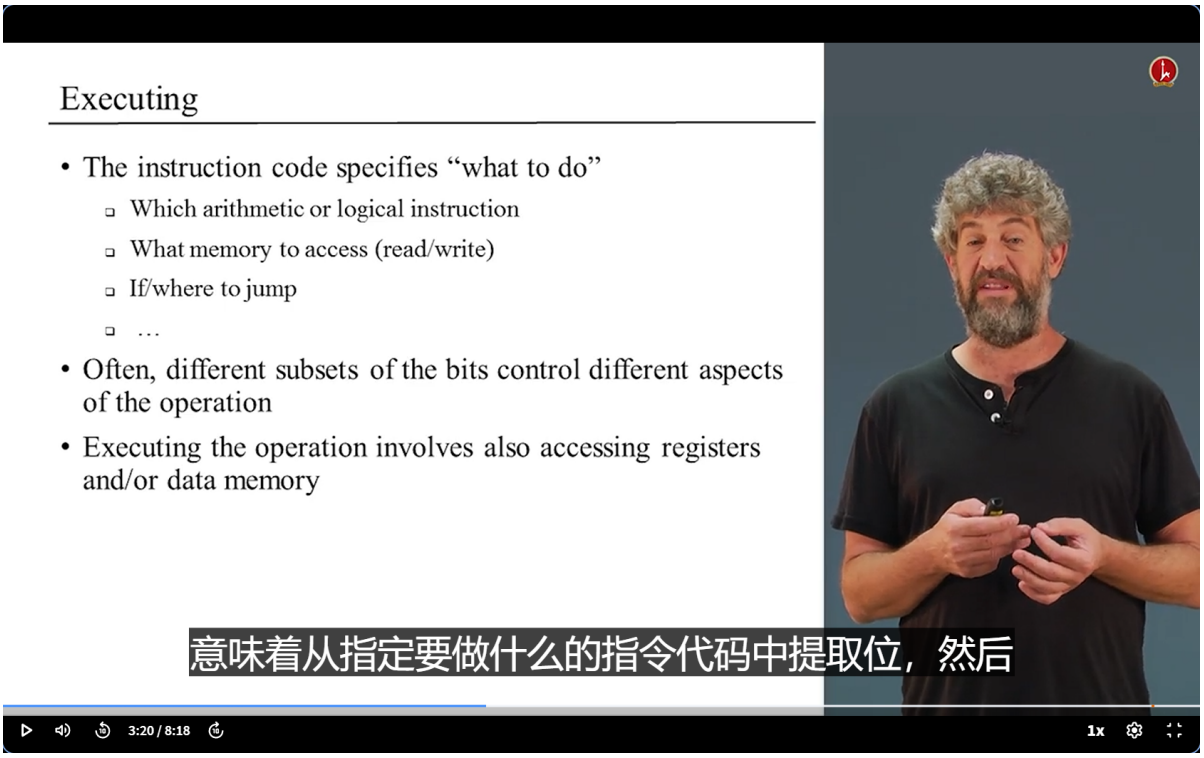
but下一条指令在哪里找呢?

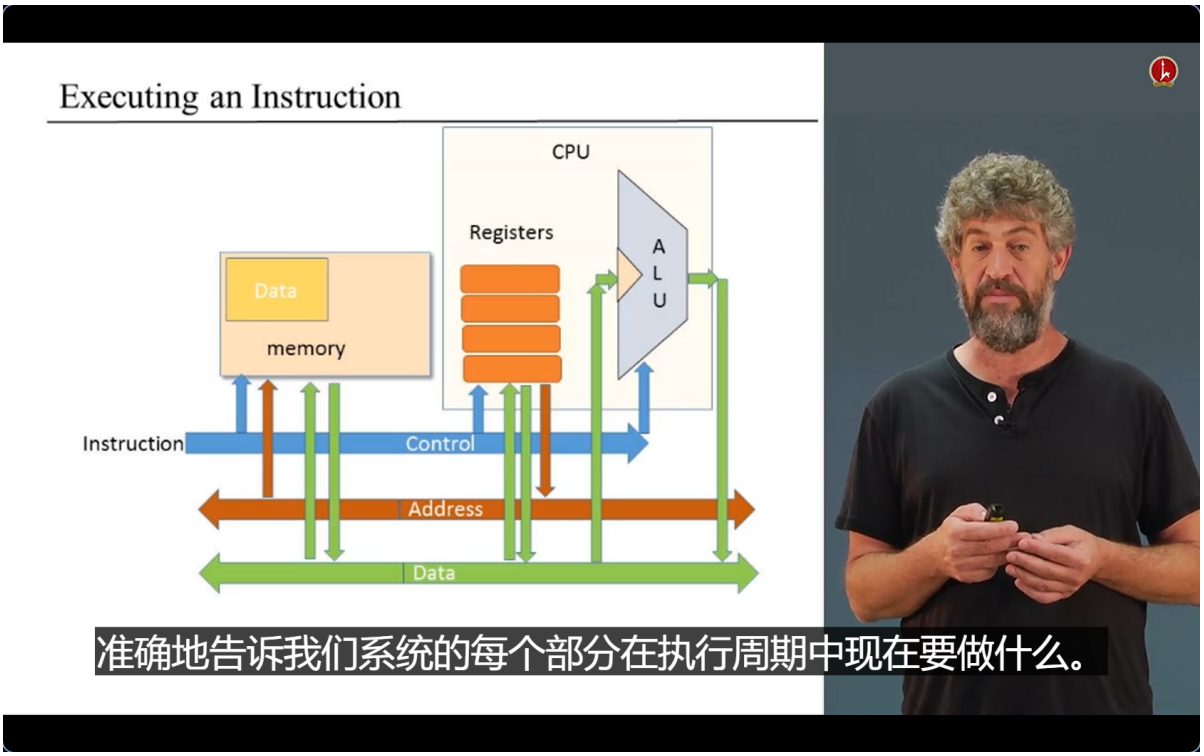
- 在内存里
- 在程序内存里
- 在程序计数器（program counter）里面（某个寄存器）
    - 它会根据指令的执行情况自动递增或根据跳转指令等进行修改，以指向下一条要取的指令**地址。**



执行指令

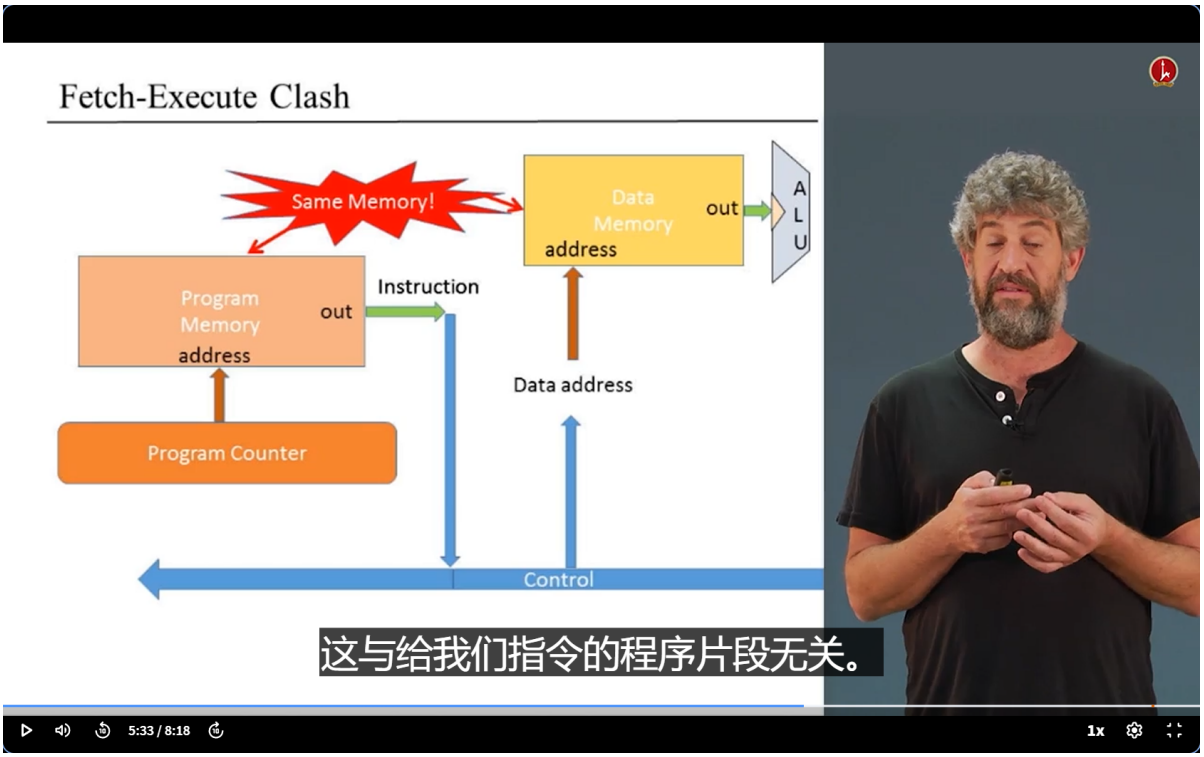
- 有了Instruction，我们需要执行它。
- 指令代码包含了很多信息（不同位有不同信息）
- 把取出的指令通过 控制总线 输出到CPU
    - 控制总线 告诉ALU计算什么指令，告诉数据来自哪个寄存器还是哪个数据存储器。



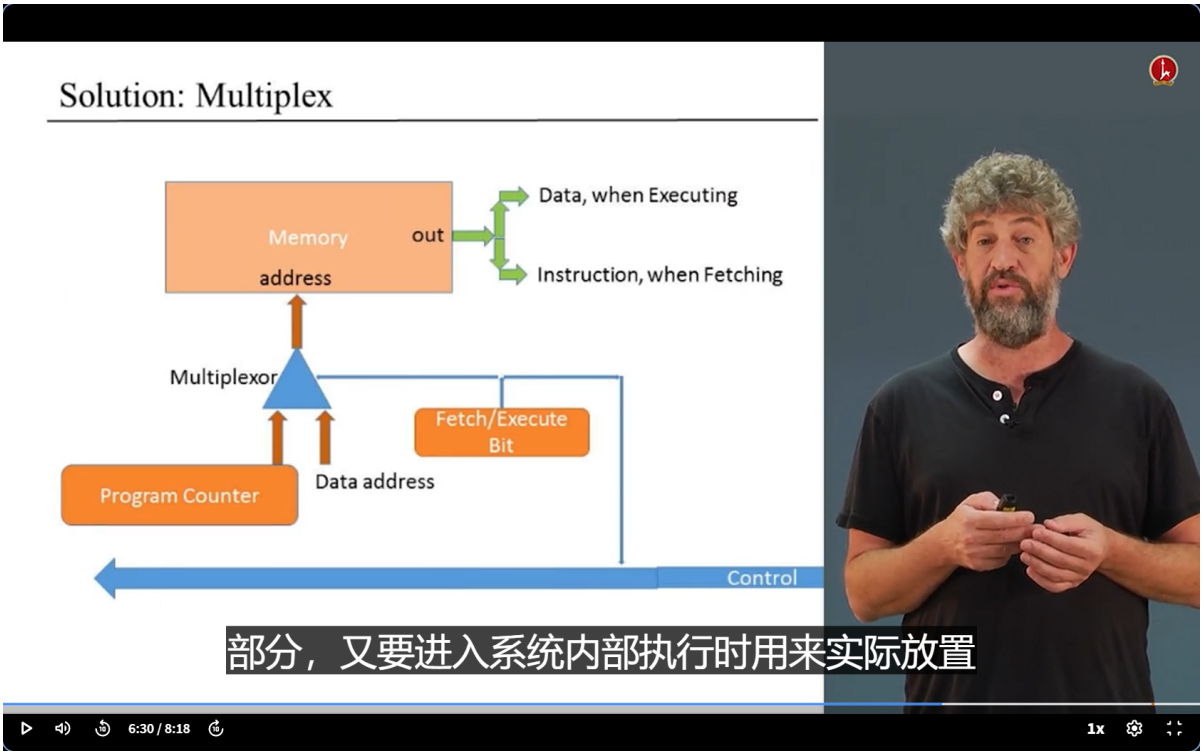


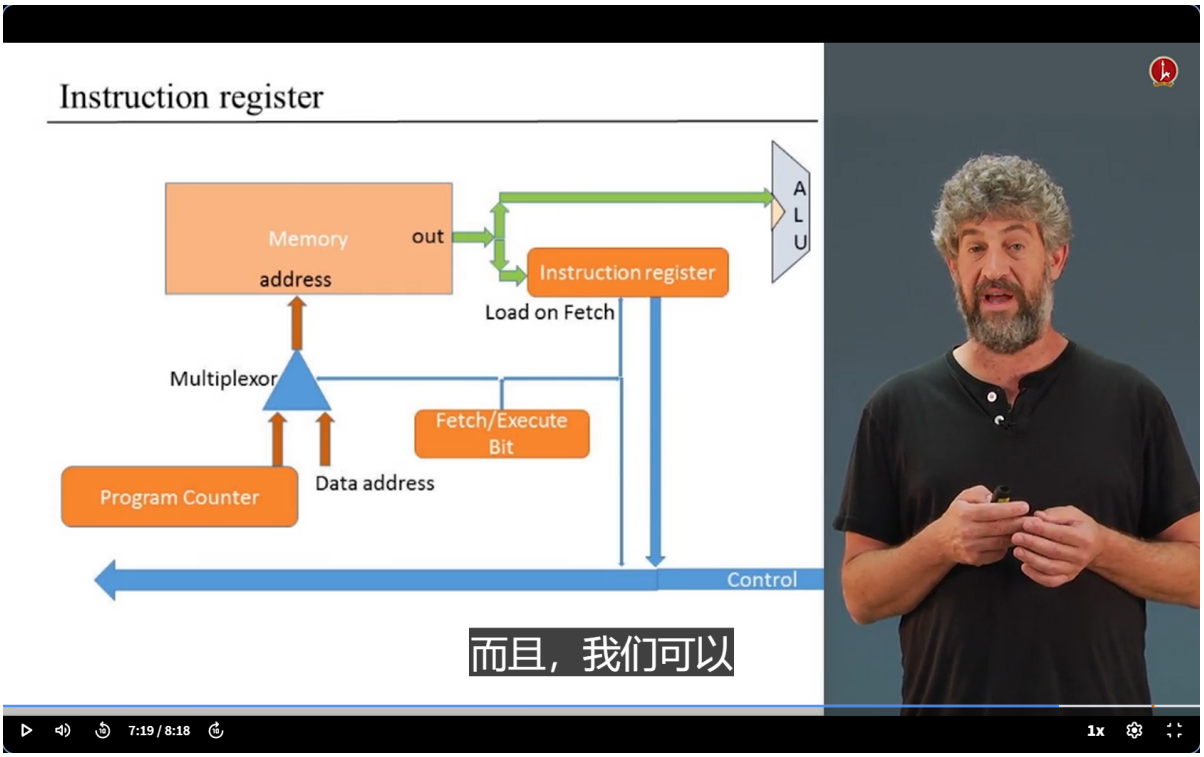
获取-执行 冲突Clash

- 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。

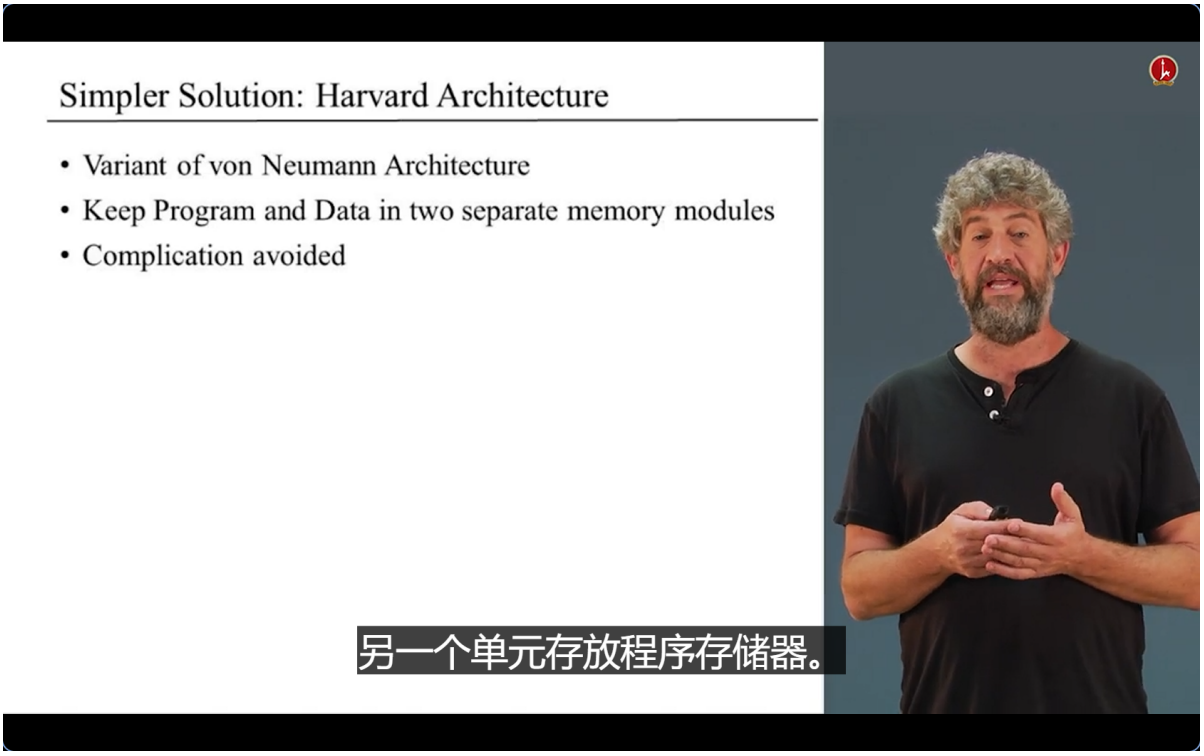


- 解决方法：一个一个做。通过一个Mux解决（+a Fetch/Execute Bit)





- 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。



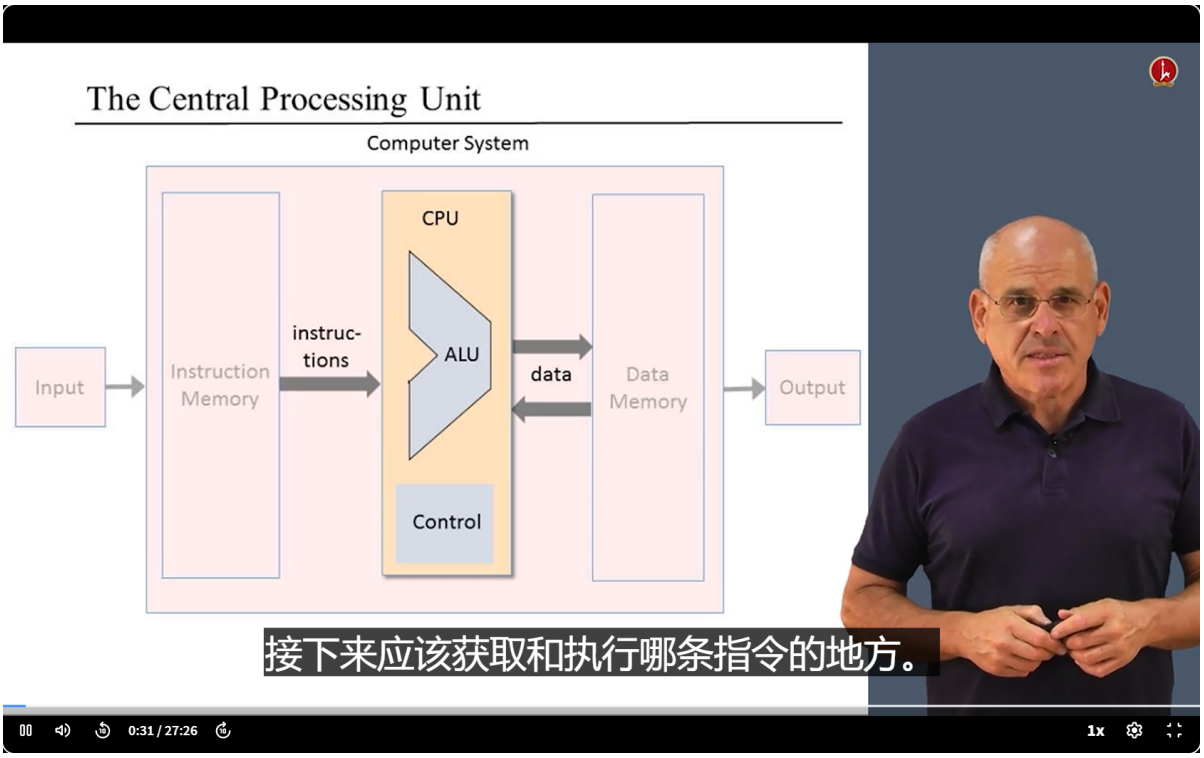
Next

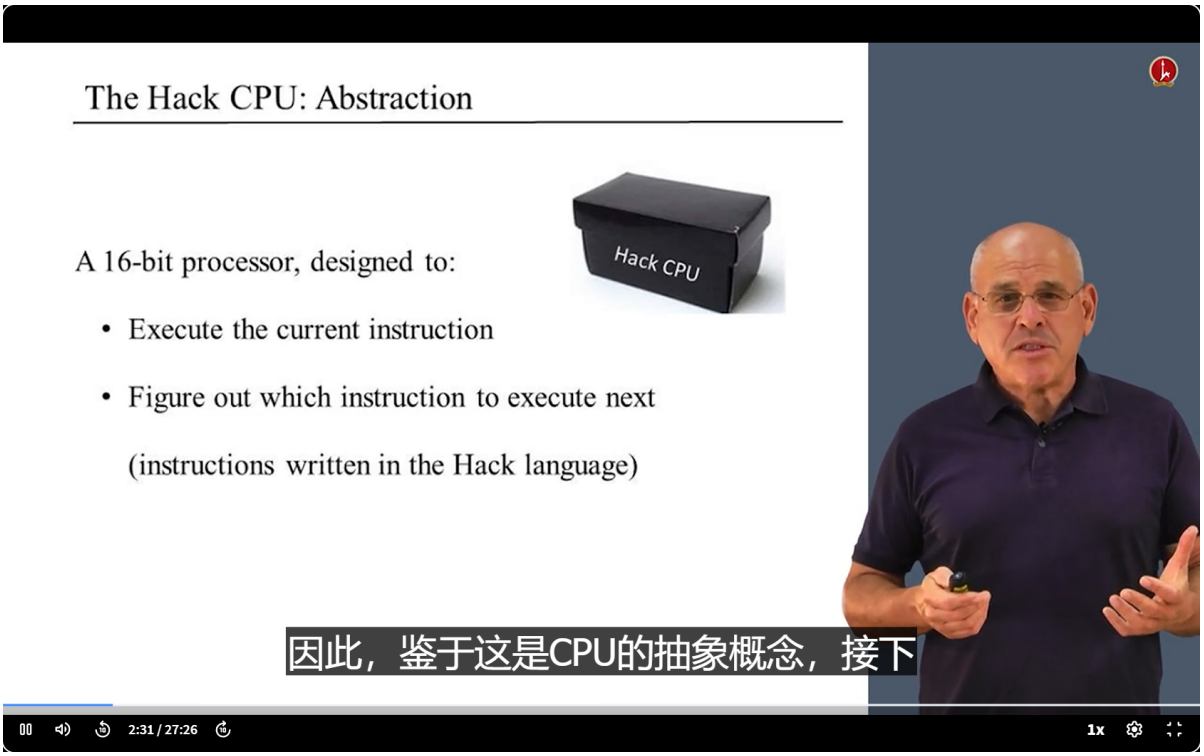
- 这一节，我们讨论了计算机的一般架构

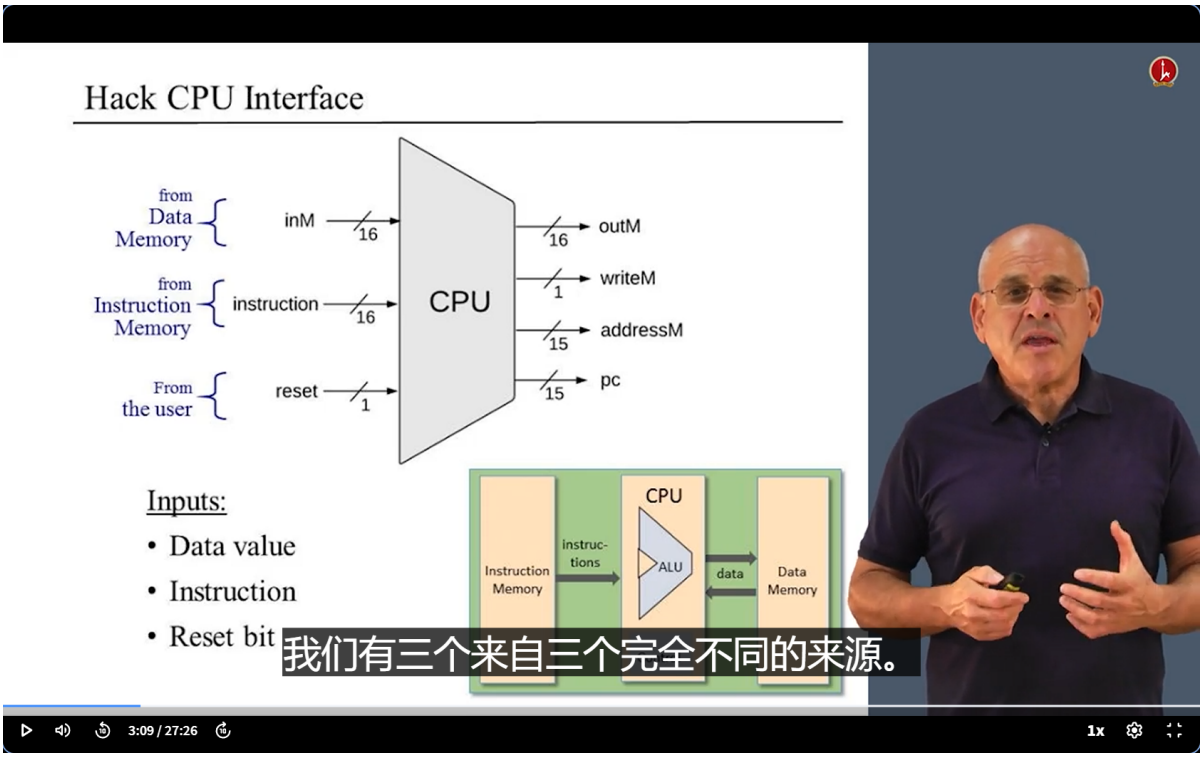
- 下一节，我们讨论我们的HACK计算机

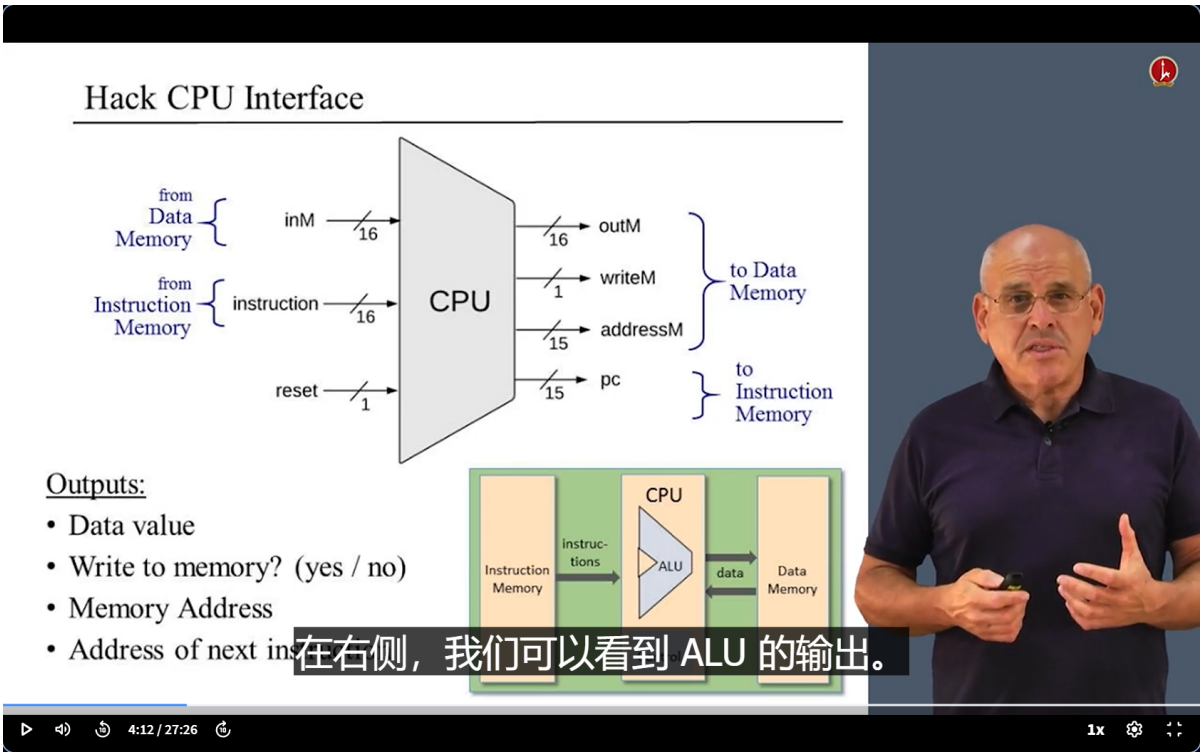
5.3 Central Processing Unit

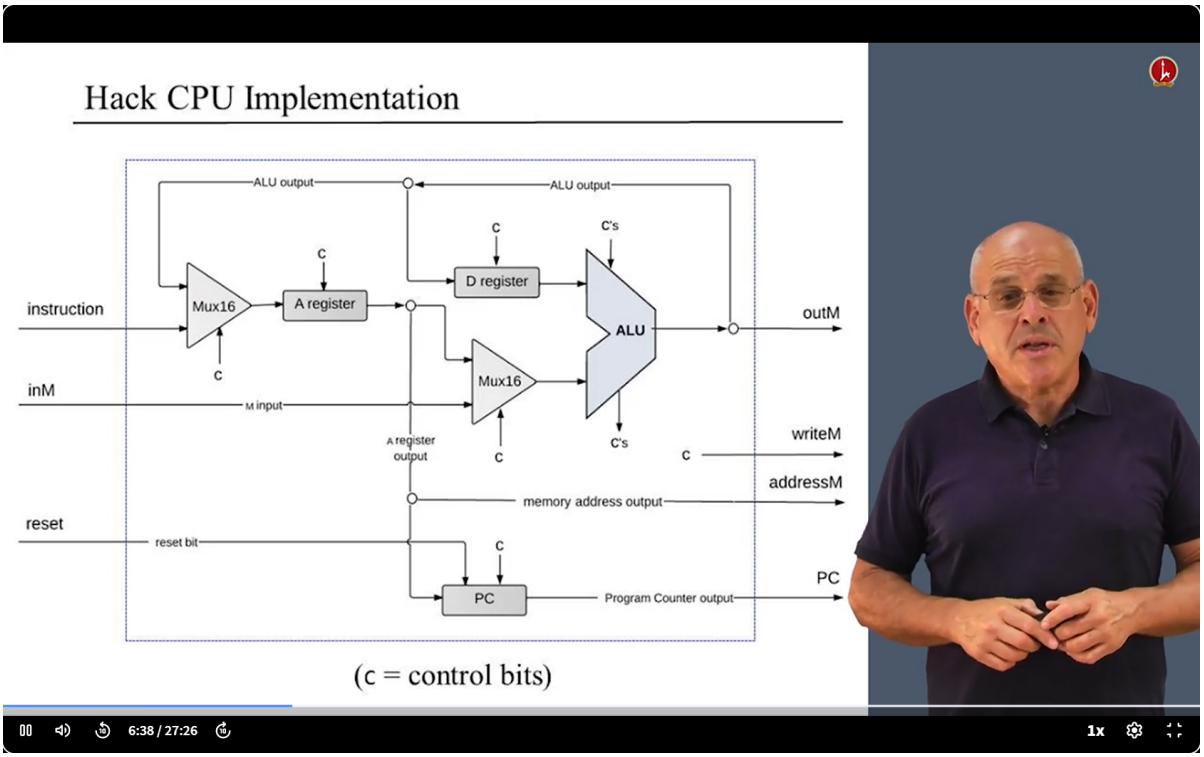
CPU: interface and implementation


The Central Processing Unit
Computer System
Input
Instruction Memory
instruc-tions
CPU
ALU
Control
data
Data Memory
Output
接下来应该获取和执行哪条指令的地方。
0:31 / 27:26
1x


The Hack CPU: Abstraction
A 16-bit processor, designed to:
Hack CPU
• 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
2:31 / 27:26
1x


Hack CPU Interface
from Data Memory
inM
16
from Instruction Memory
instruction
16
From the user
reset
1
CPU
16
outM
1
writeM
15
addressM
15
pc
Inputs:
• Data value
• Instruction
• Reset bit
CPU
Instruction Memory
instruc-tions
ALU
data
Data Memory
我们有三个来自三个完全不同的来源。
3:09 / 27:26
1x

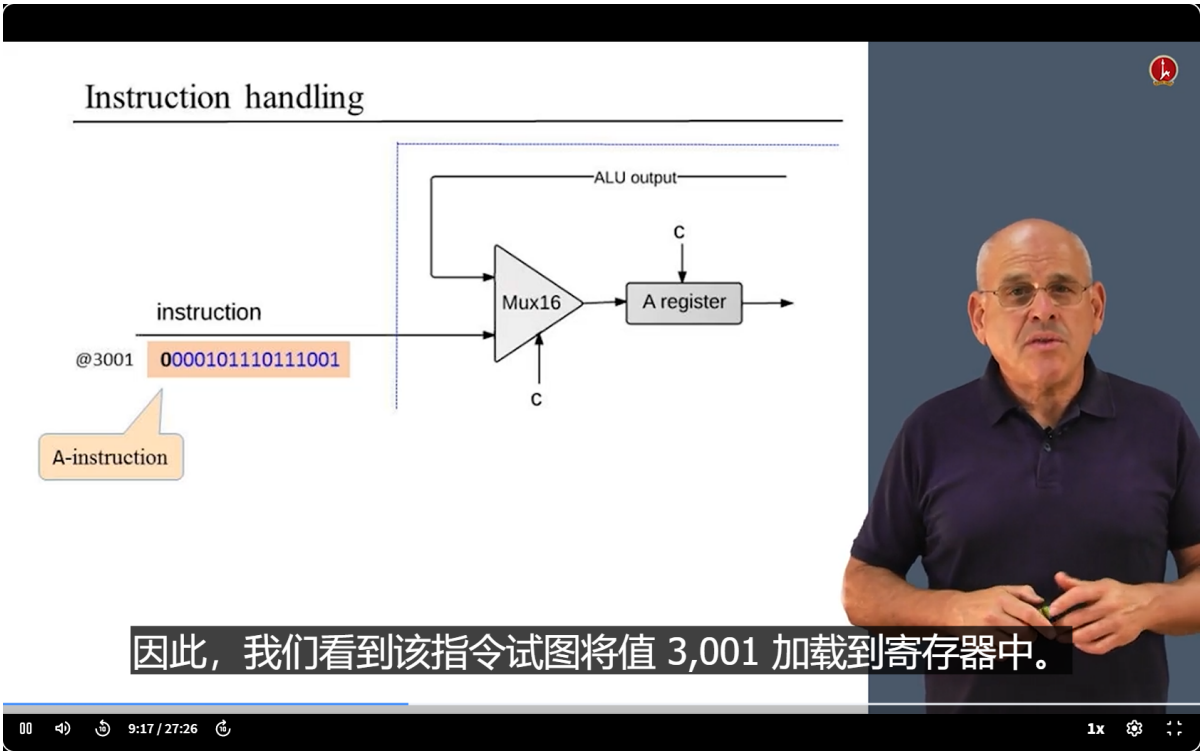

Hack CPU Interface
from Data Memory
inM
16
from Instruction Memory
instruction
16
reset
1
CPU
16
outM
1
writeM
15
addressM
15
pc
to Data Memory
to Instruction Memory
Outputs:
• Data value
• Write to memory? (yes / no)
• Memory Address
• Address of next ins
CPU
Instruction Memory
instruc-tions
ALU
data
Data Memory
在右侧，我们可以看到 ALU 的输出。
4:12 / 27:26
1x

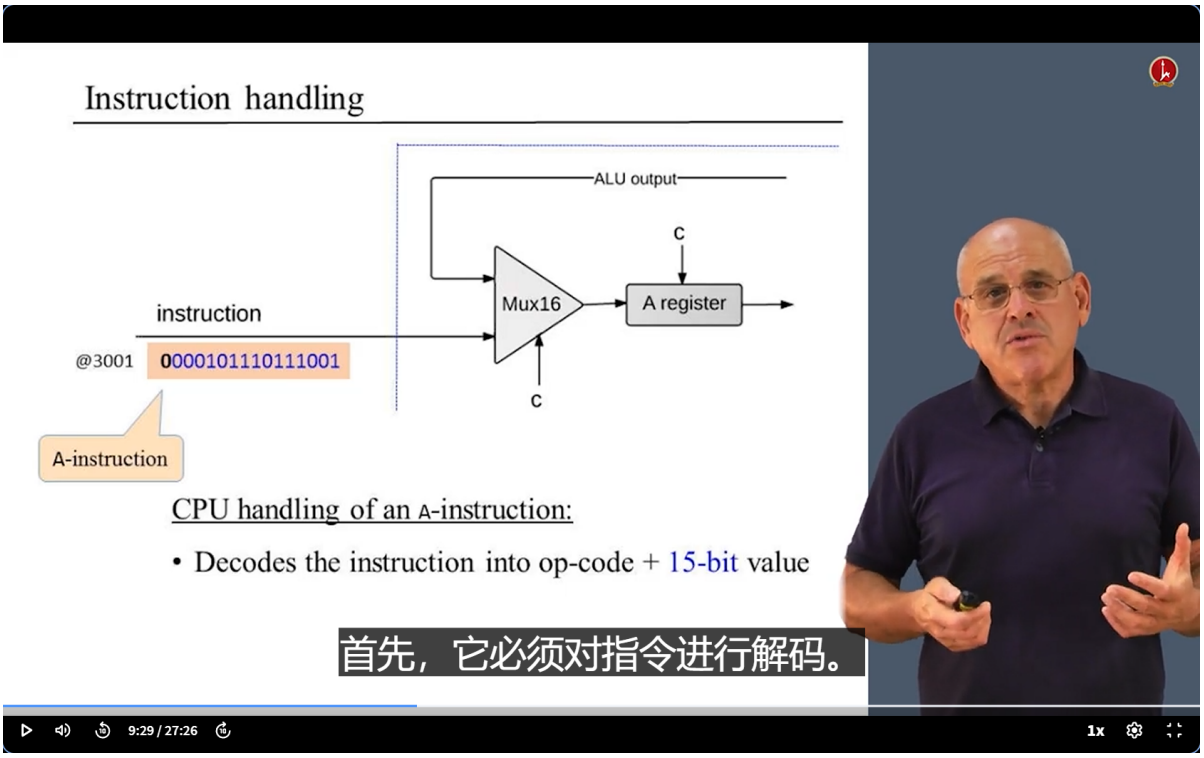


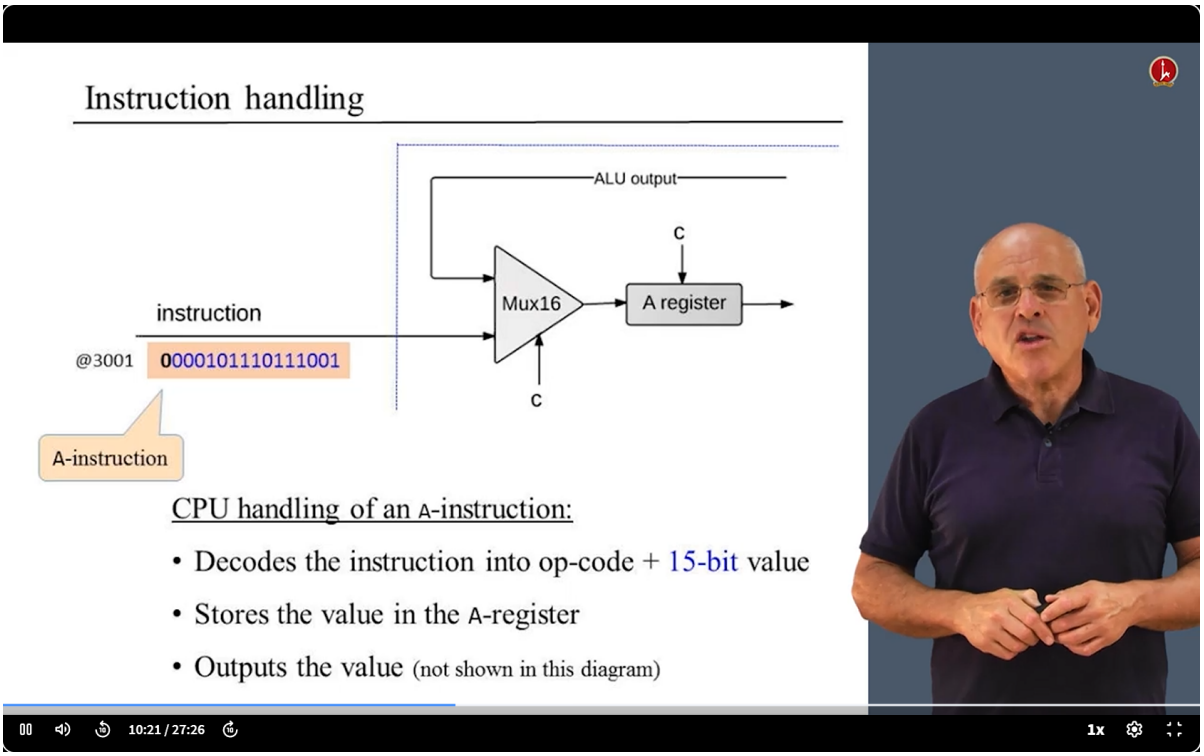
CPU: Instruction handling

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。

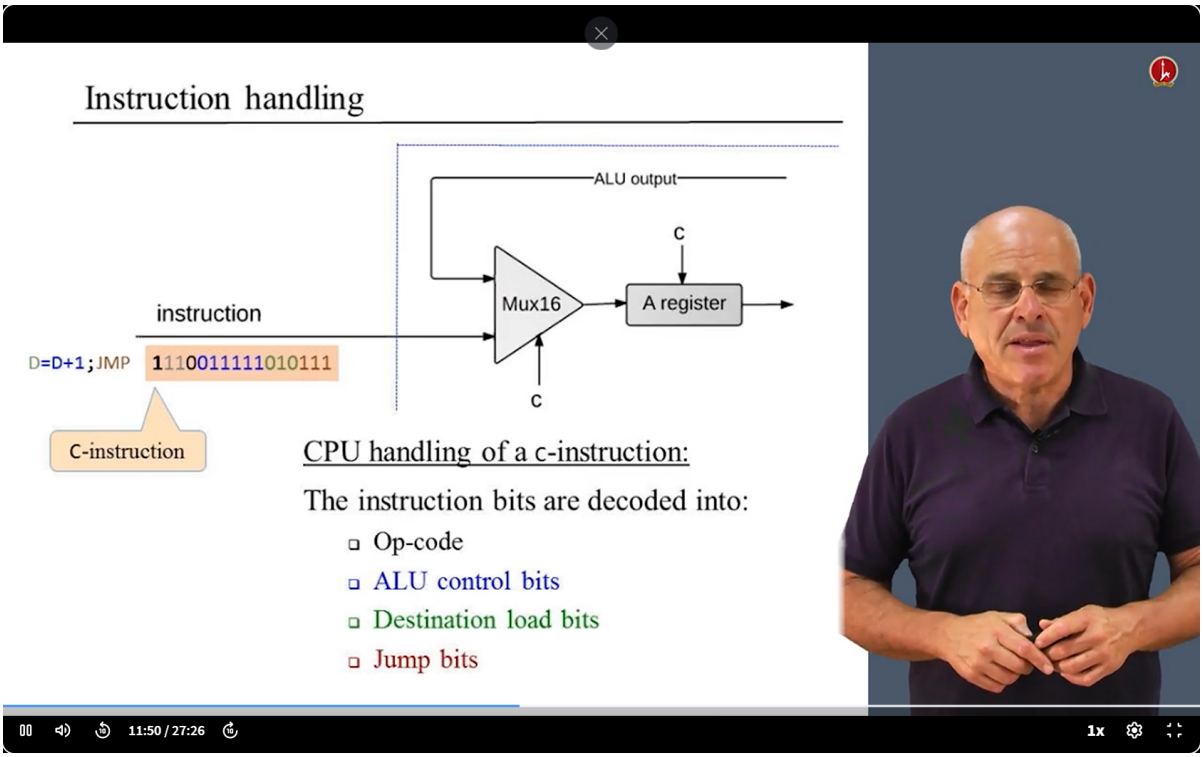






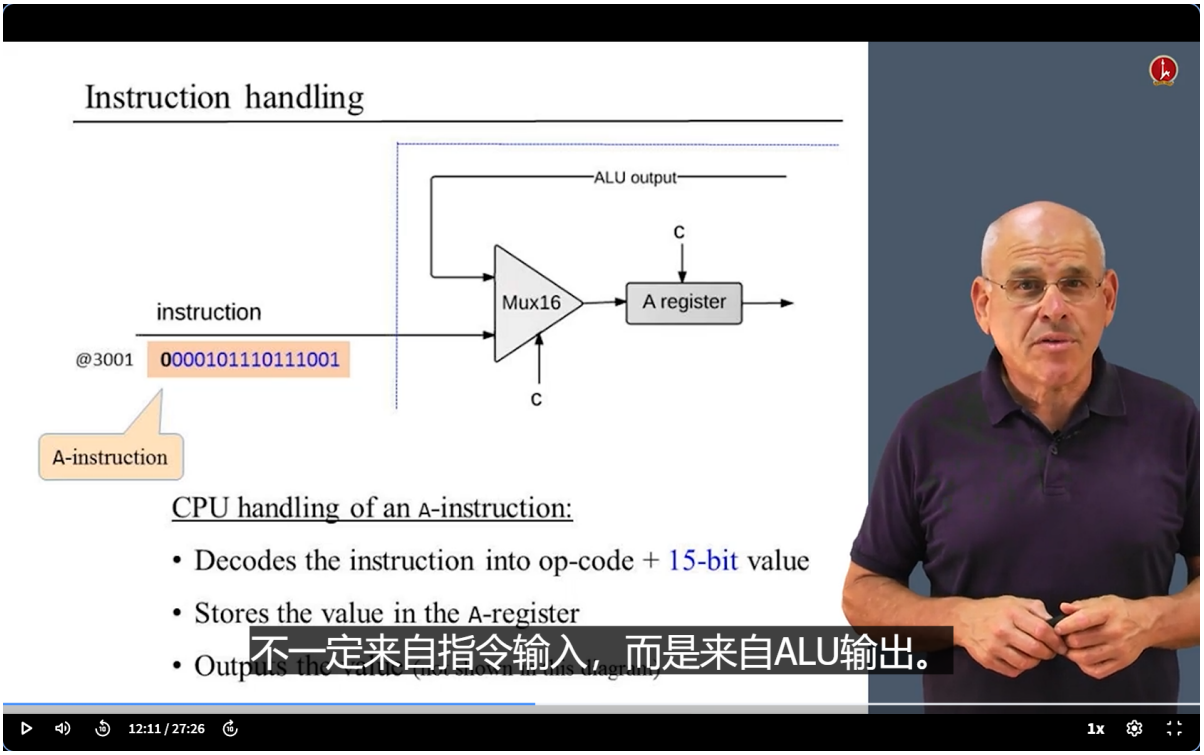
handle C-instruction

- 解码：分离操作码和其他三个字段

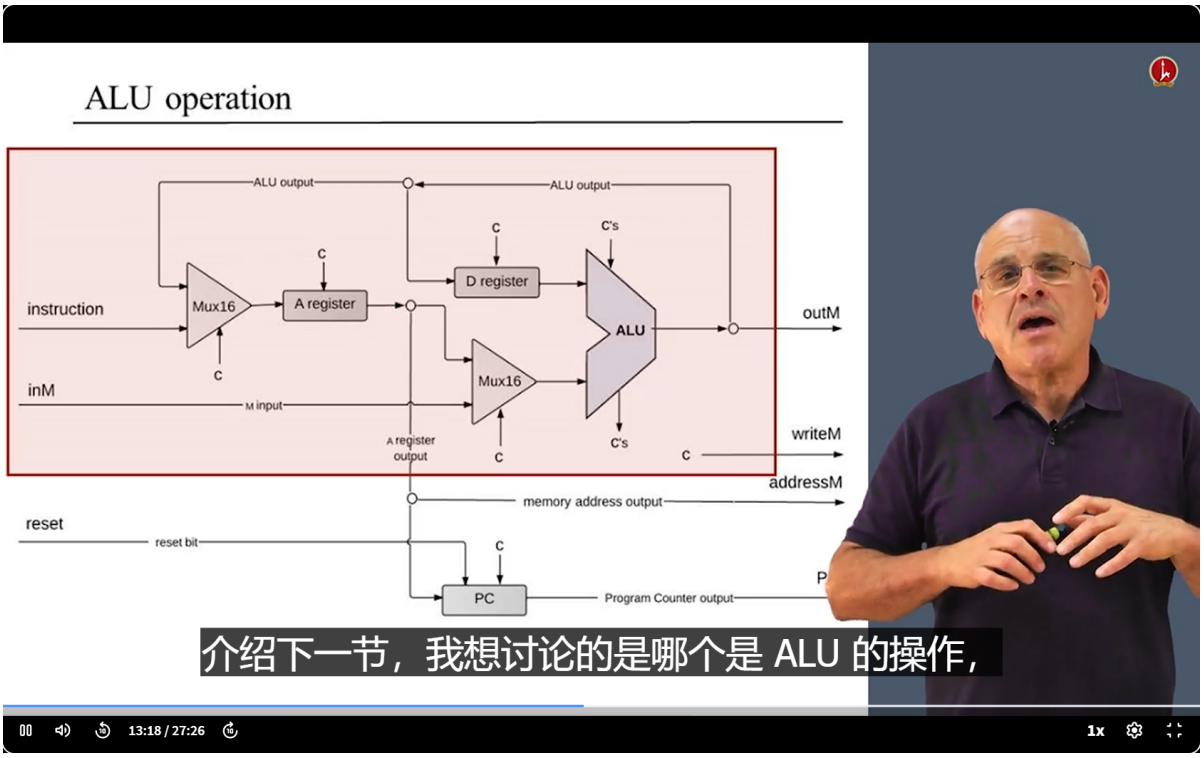


A regester -- again

- 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 regester？）
- 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。

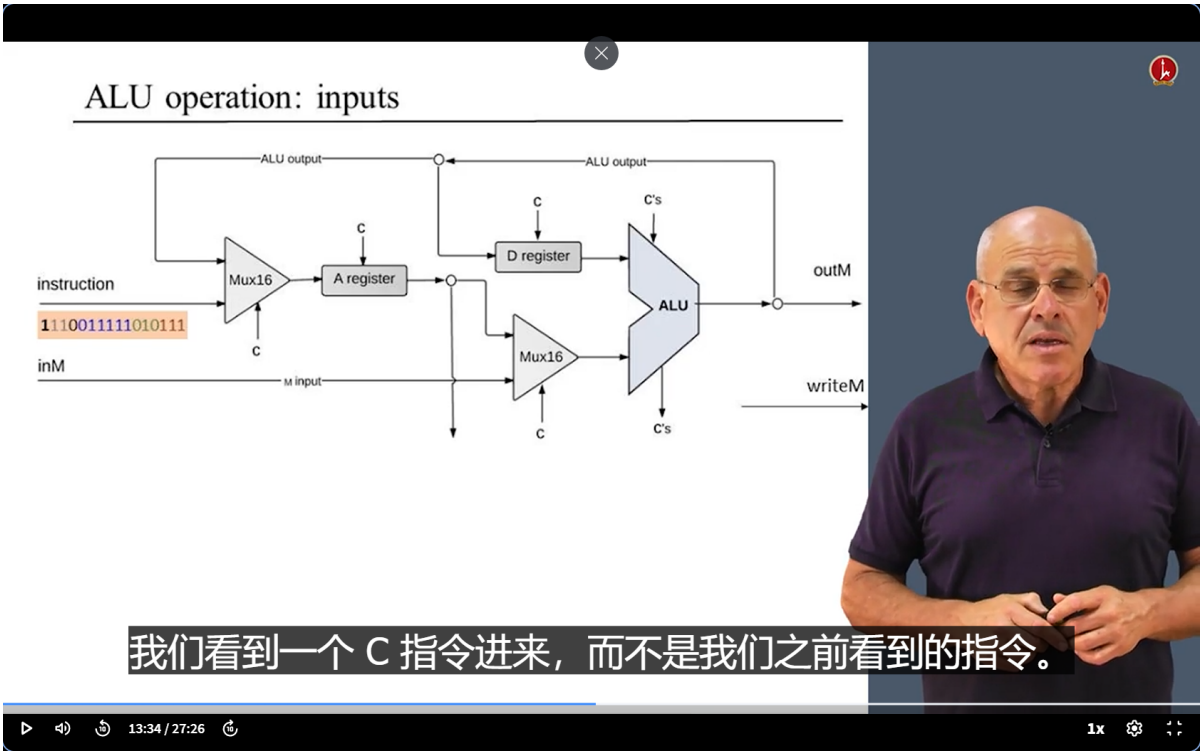


CPU: ALU operation

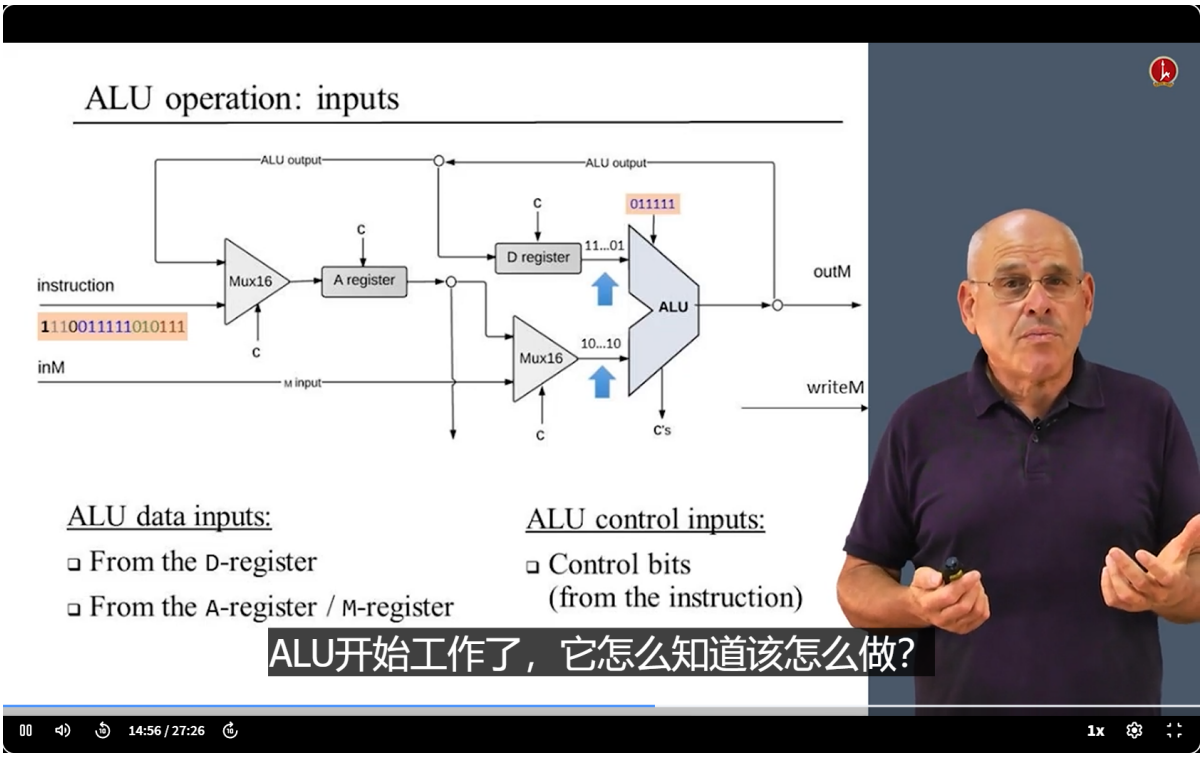


ALU operation: inputs

- remember：C指令由不同的位字段组成，每个字段都有不同的含义。

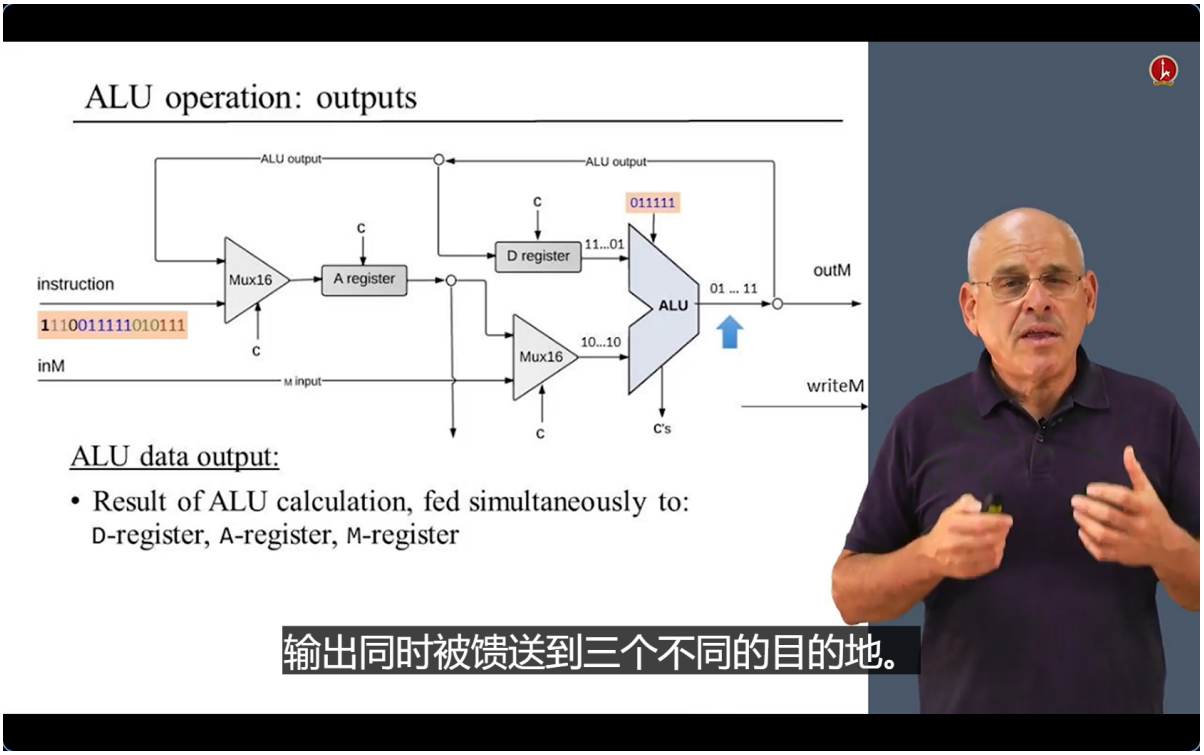


- ALU 有两个输入源 , 最终输入depend on 控制位c和011111(now)
- Control bit **from the instruction**

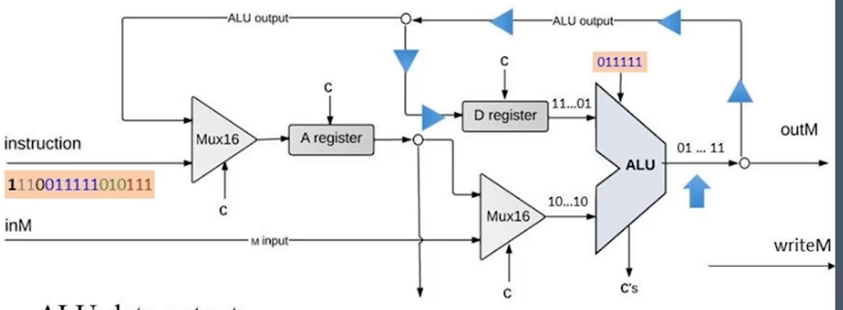


ALU operation: outputs

- ALU 的 输出被送到三个地方。



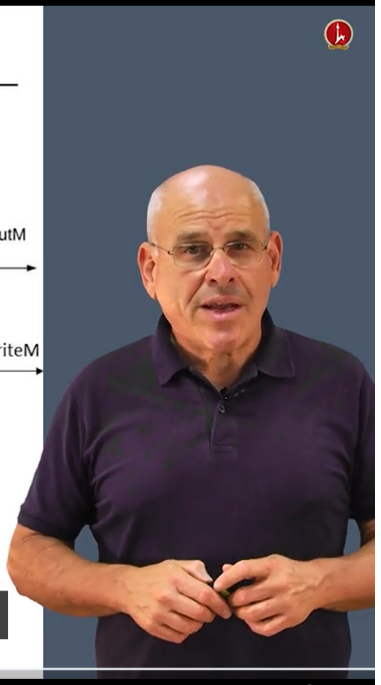
ALU operation: outputs



ALU data output:

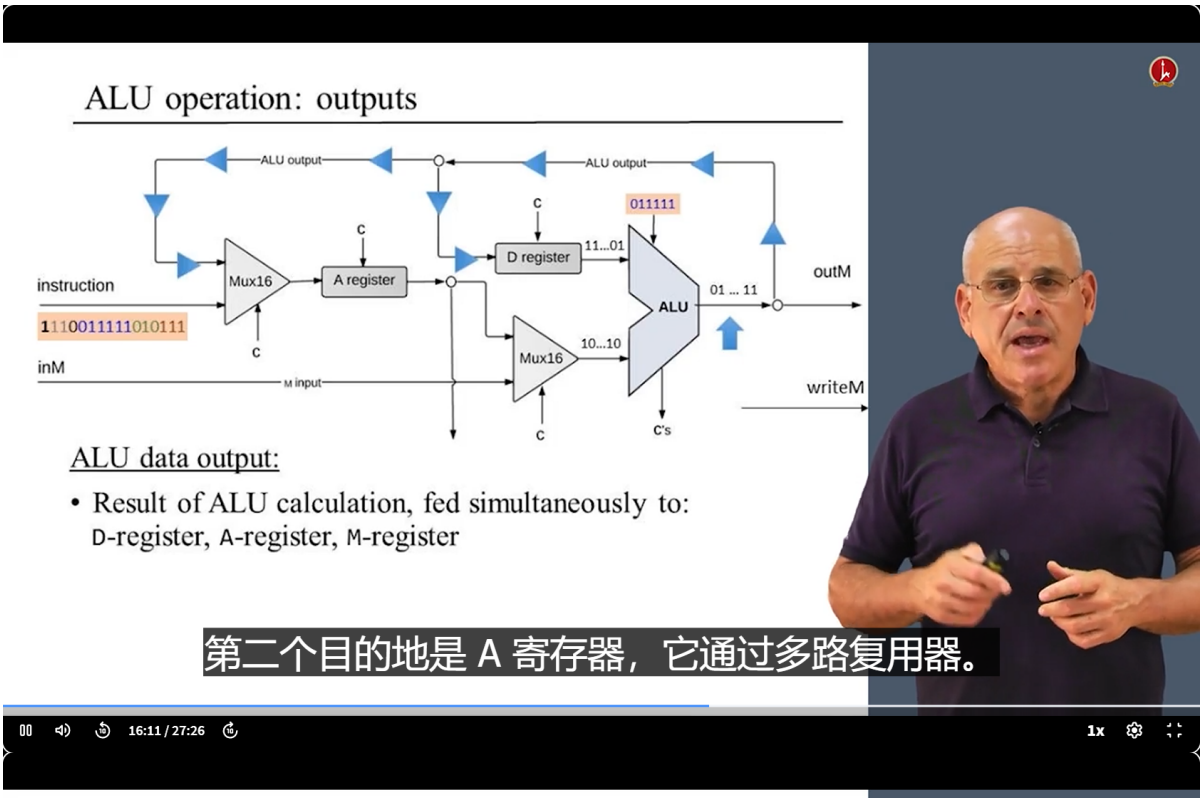
- Result of ALU calculation, fed simultaneously to: D-register, A-register, M-register

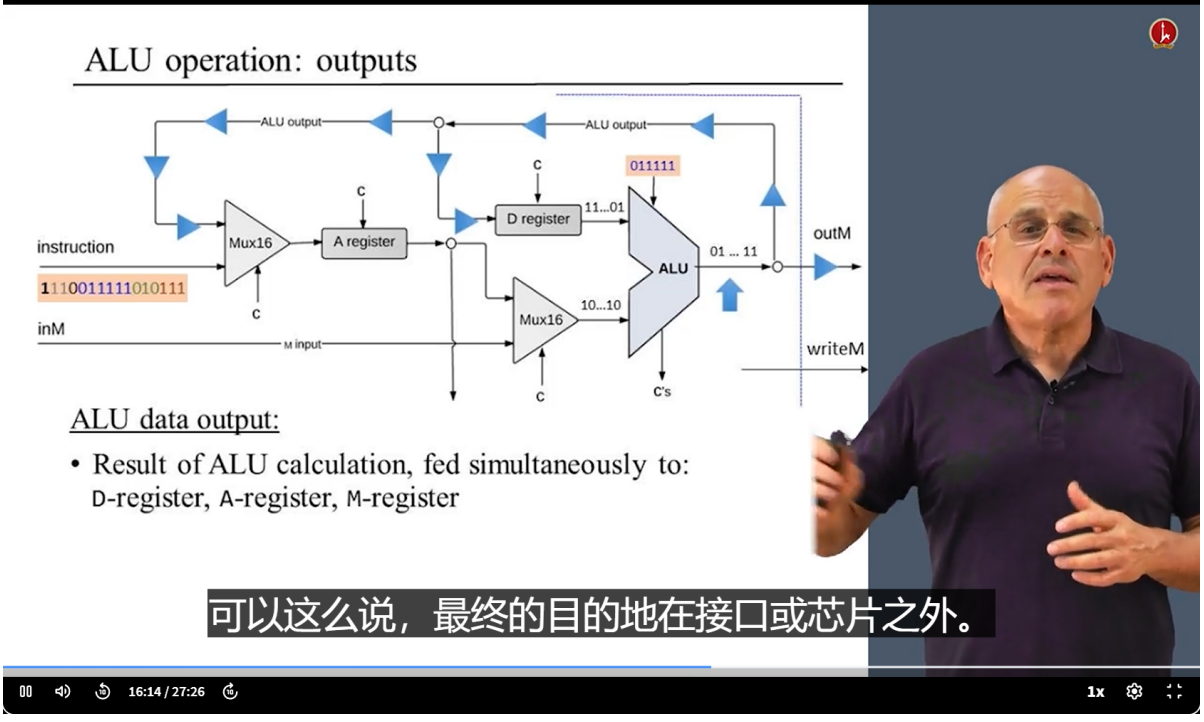
第一个目的地是 D 寄存器。



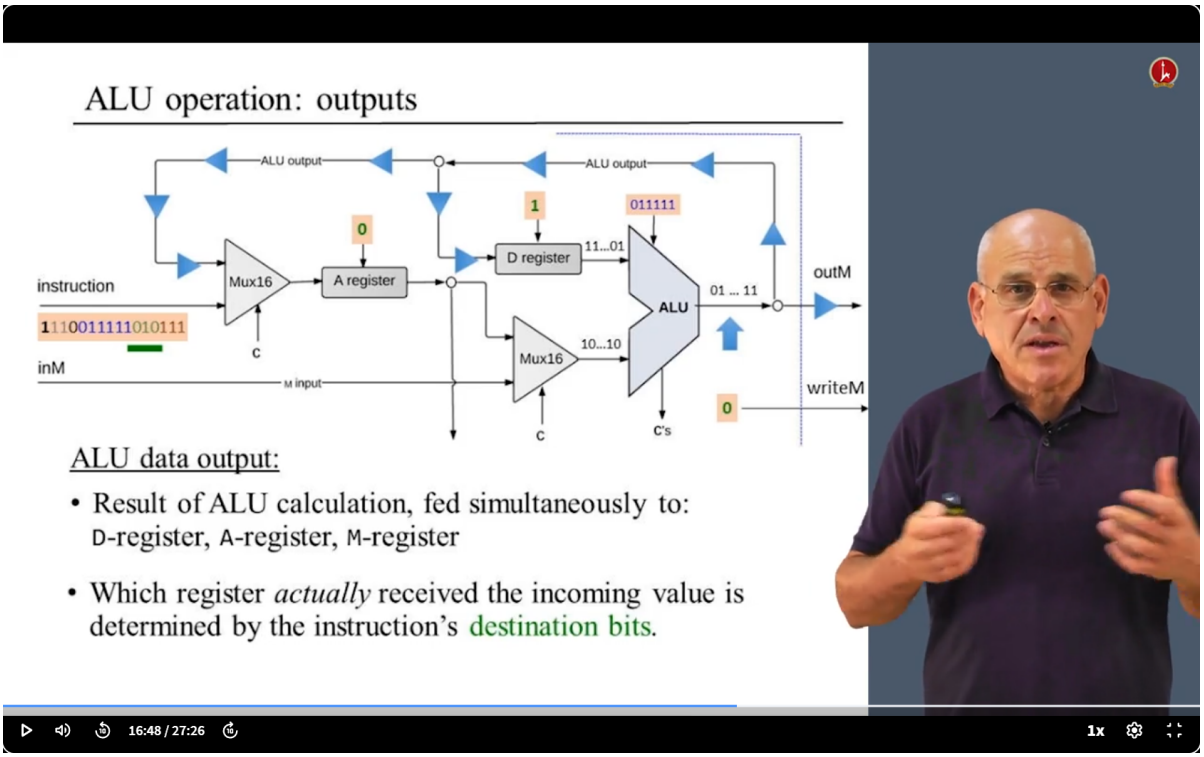
16:05 / 27:26

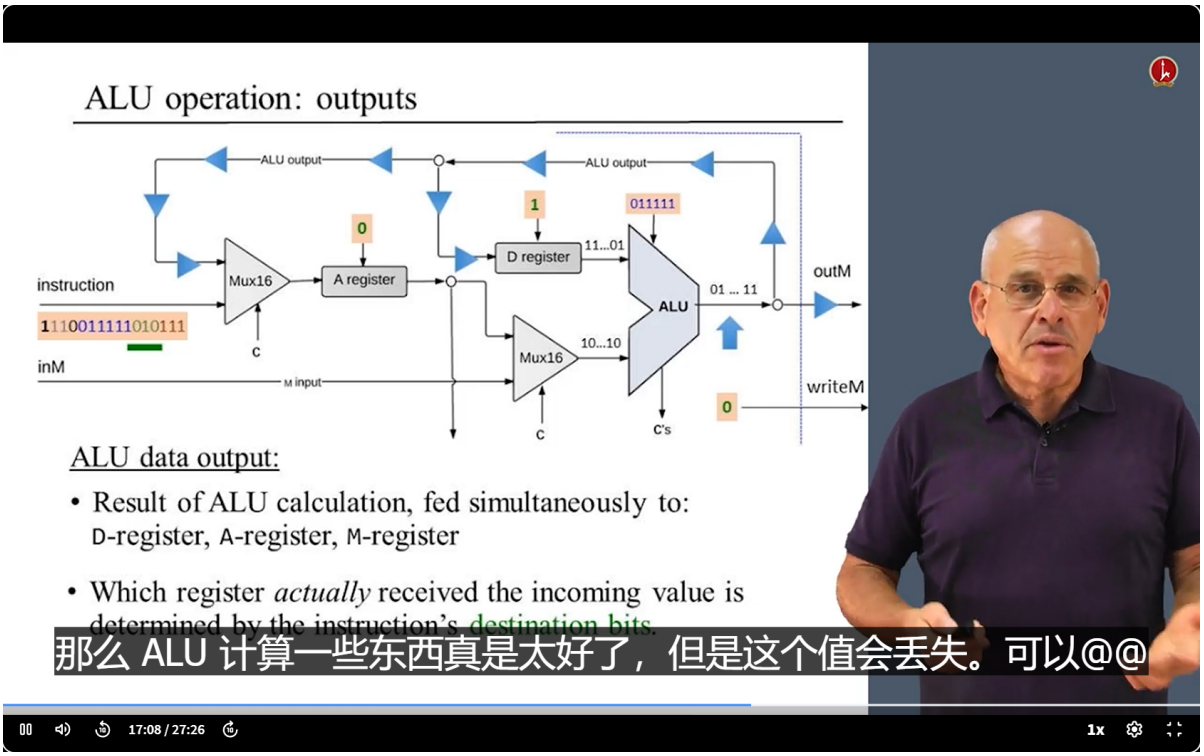
1x





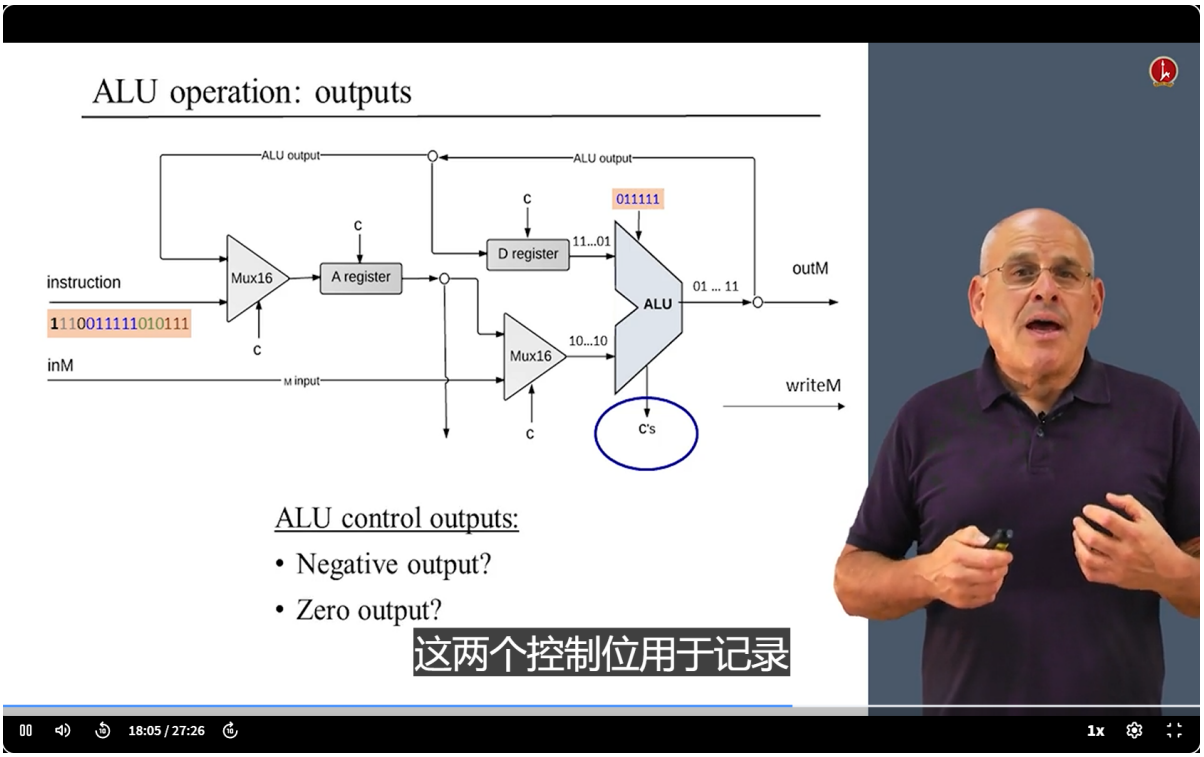
- 我们有三个目的地位，这些位决定是否打开D寄存器、A寄存器和数据存储器来接受ALU的输出。（即有选择性地接收）



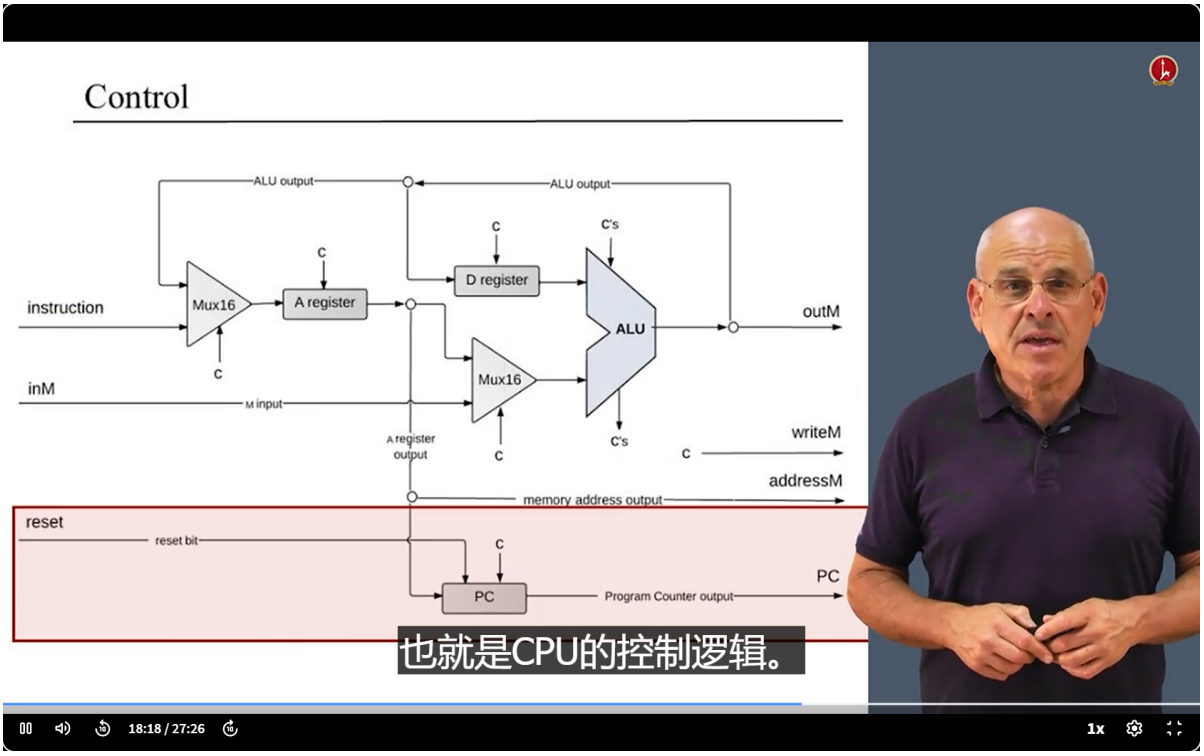


Control bit c's?

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



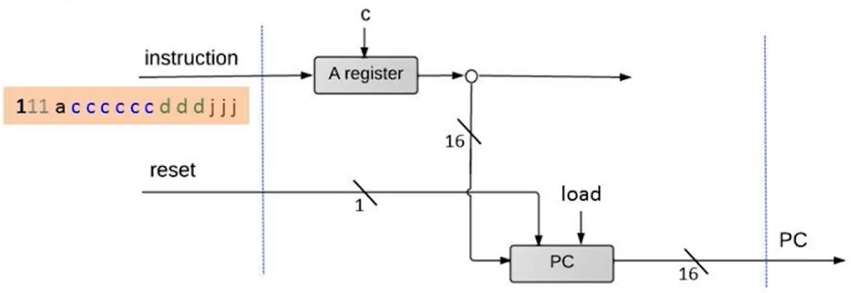
CPU：Control （logic



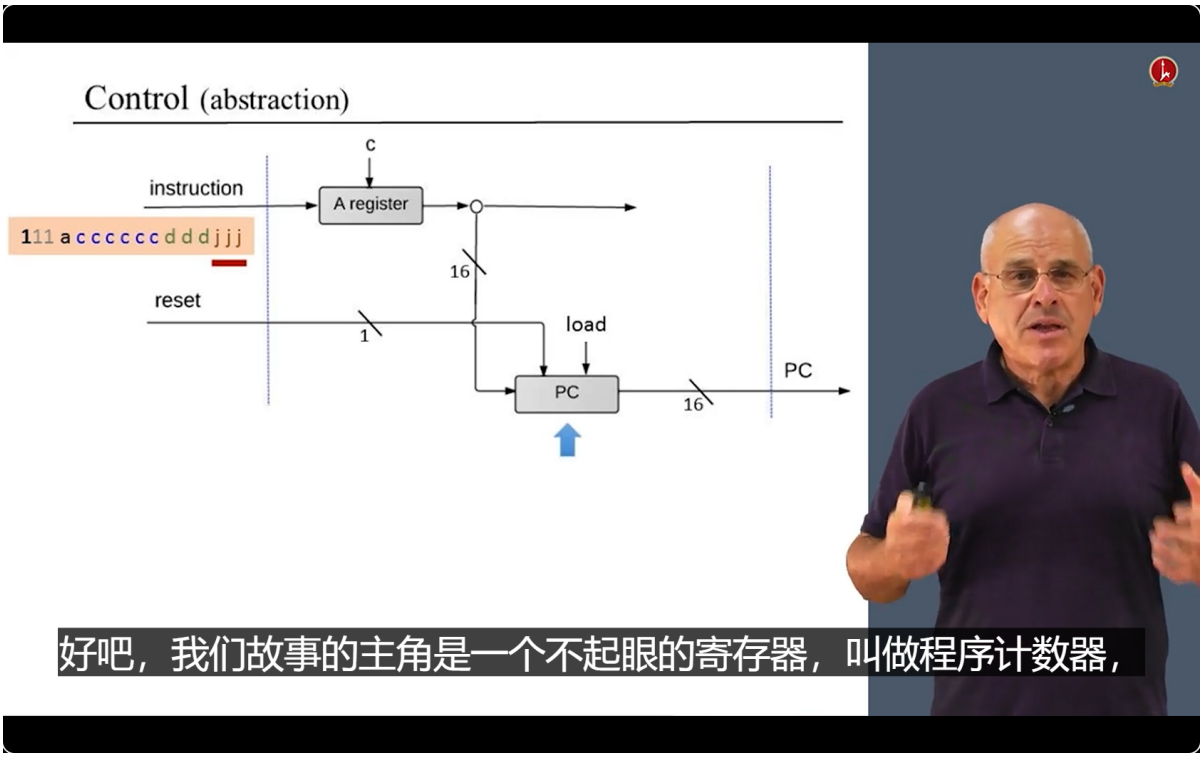
Possible outside view of the Hack computer

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

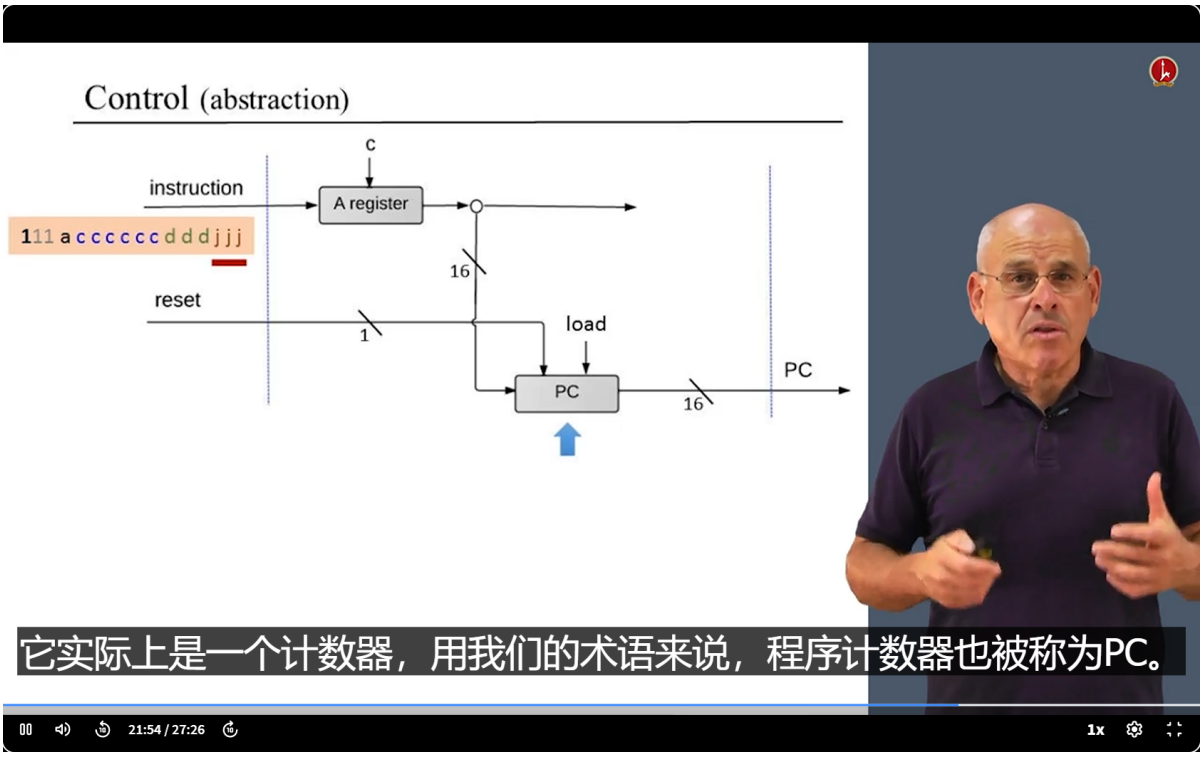
Control (abstraction)



因此，这里有一个指令的示例，它是一条原理图指令，

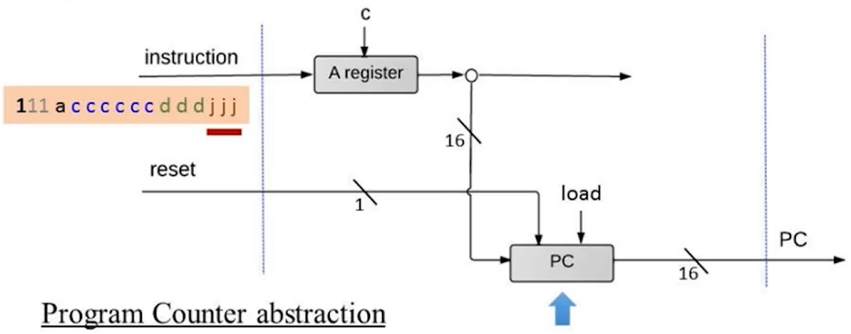


好吧，我们故事的主角是一个不起眼的寄存器，叫做程序计数器，



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)

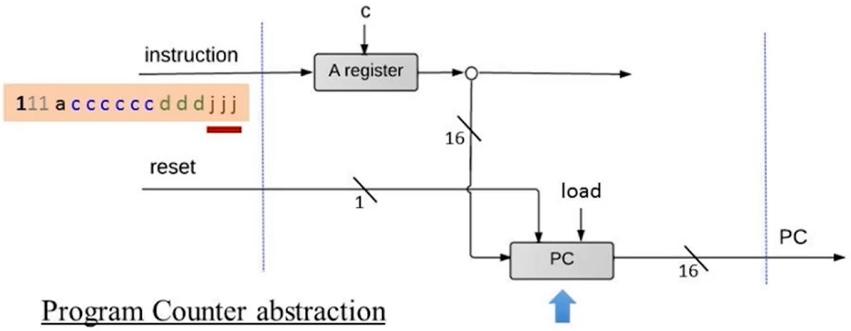


Program Counter abstraction

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)

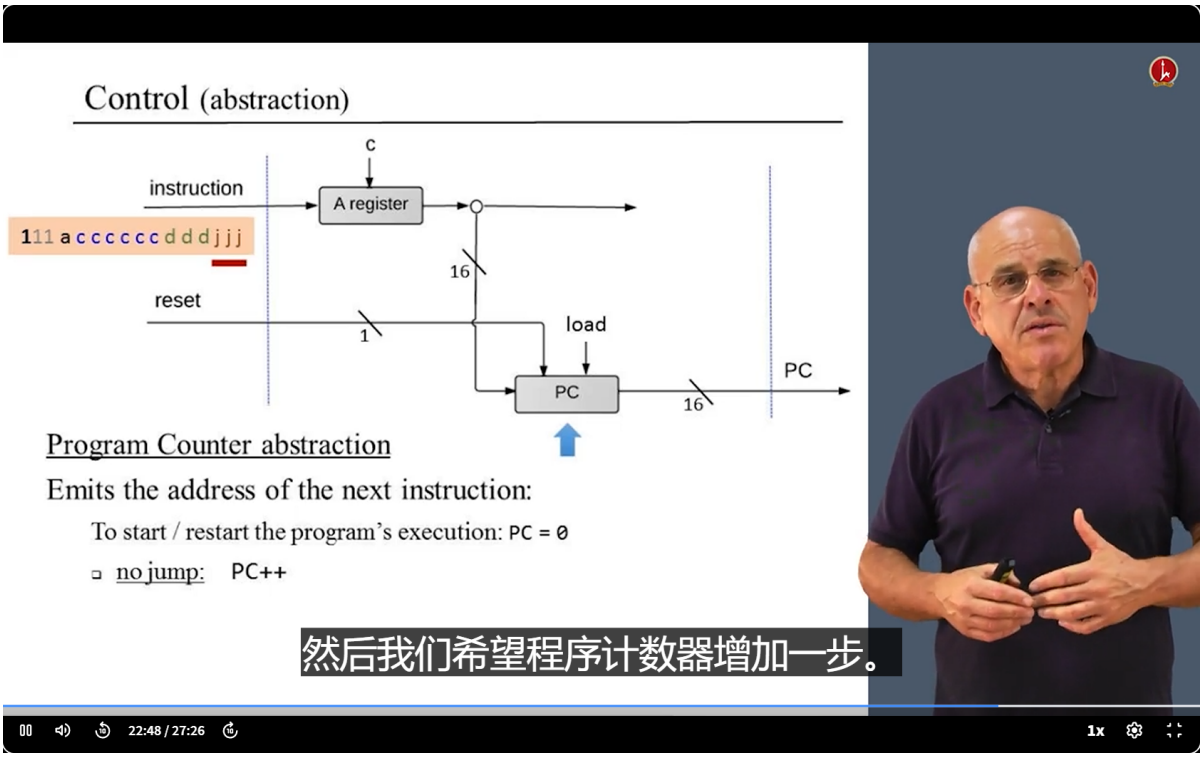


Program Counter abstraction

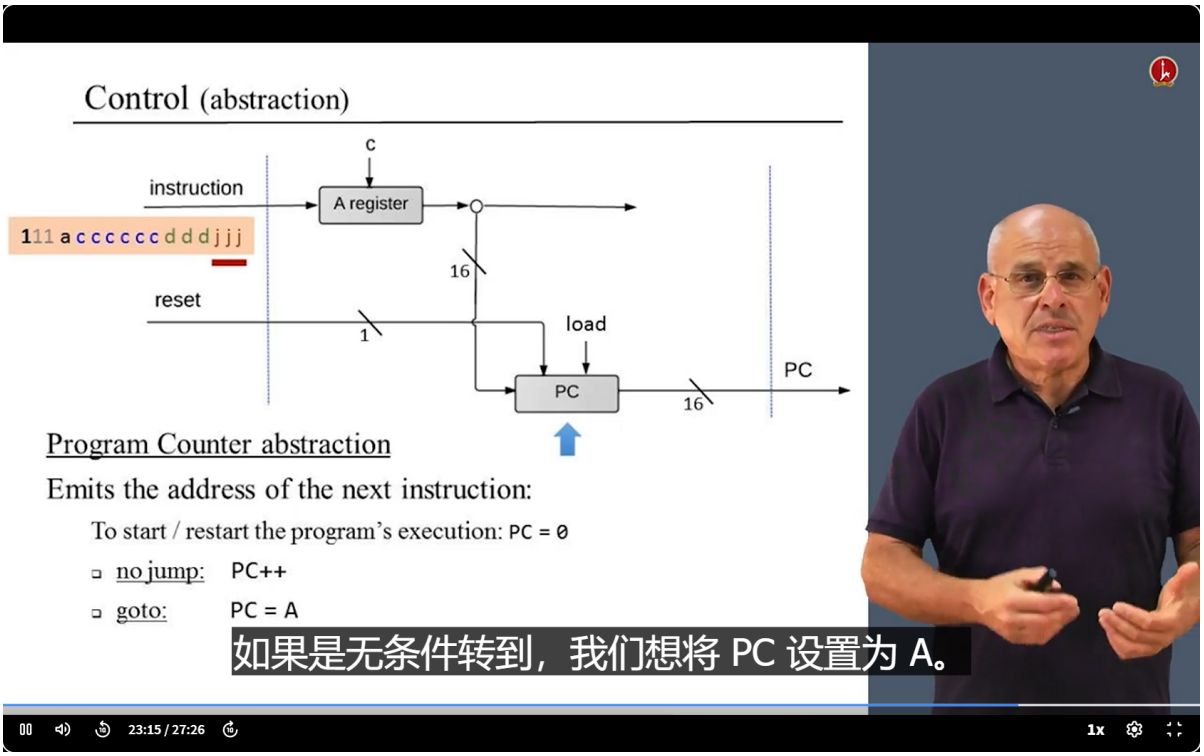
Emits the address of the next instruction:

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，

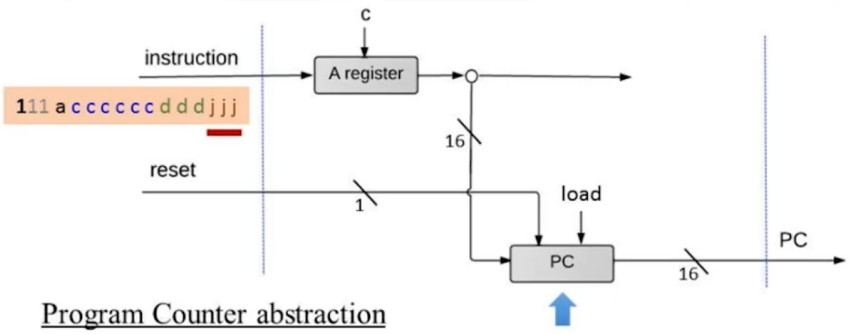


- if 程序员已经事先将想要跳转到的地址存入了A寄存器。如果我们执行PC等于A，程序计数器将输出接下来必须执行的下一条指令的地址。



- 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)

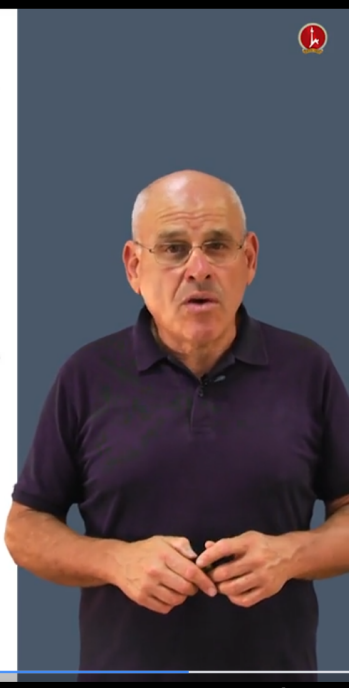


Program Counter abstraction

Emits the address of the next instruction:

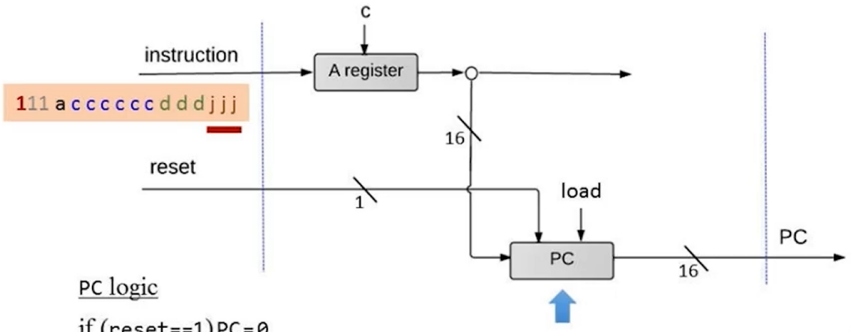
To start / restart the program's execution: `PC = 0`

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



23:46 / 27:26 1x

Control (implementation)

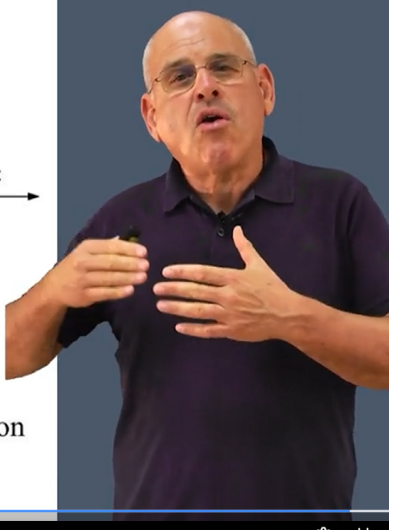


PC logic

```
if (reset==1) PC=0
else
    // current instruction
    load = f(jump bits, ALU control outputs)
    if (load==1) PC=A
    else         PC++
```

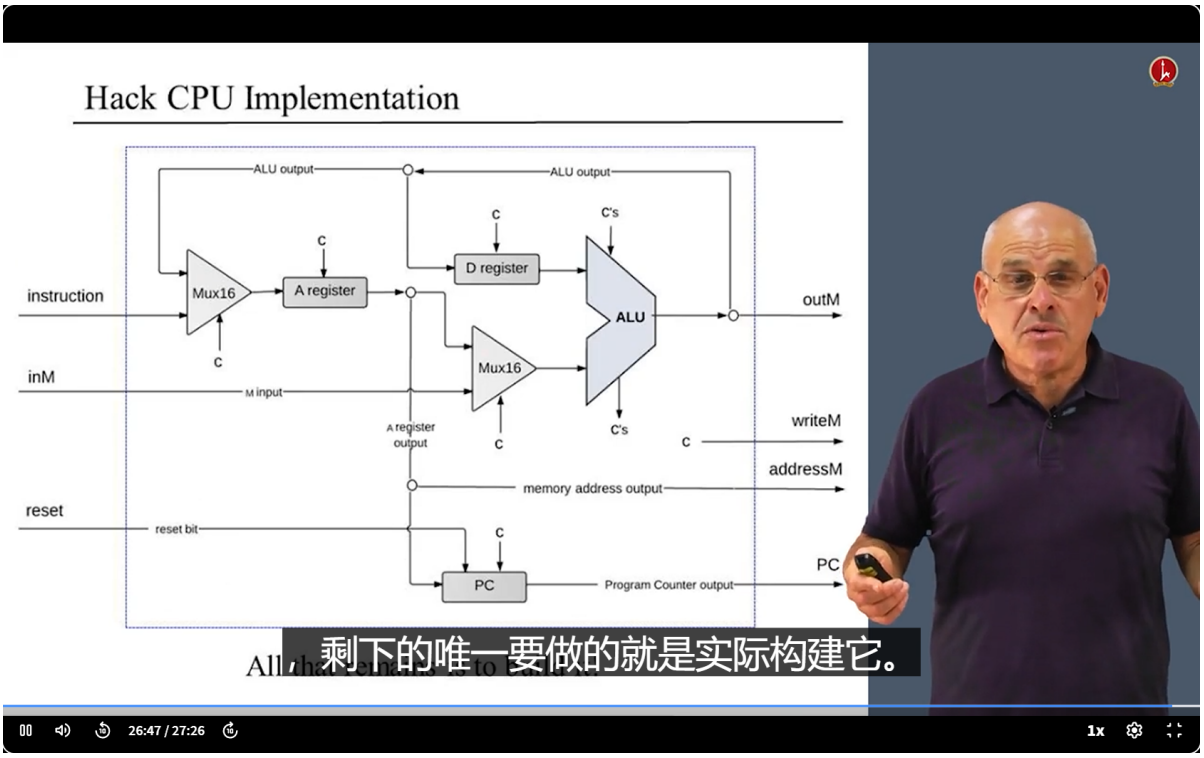
PC output

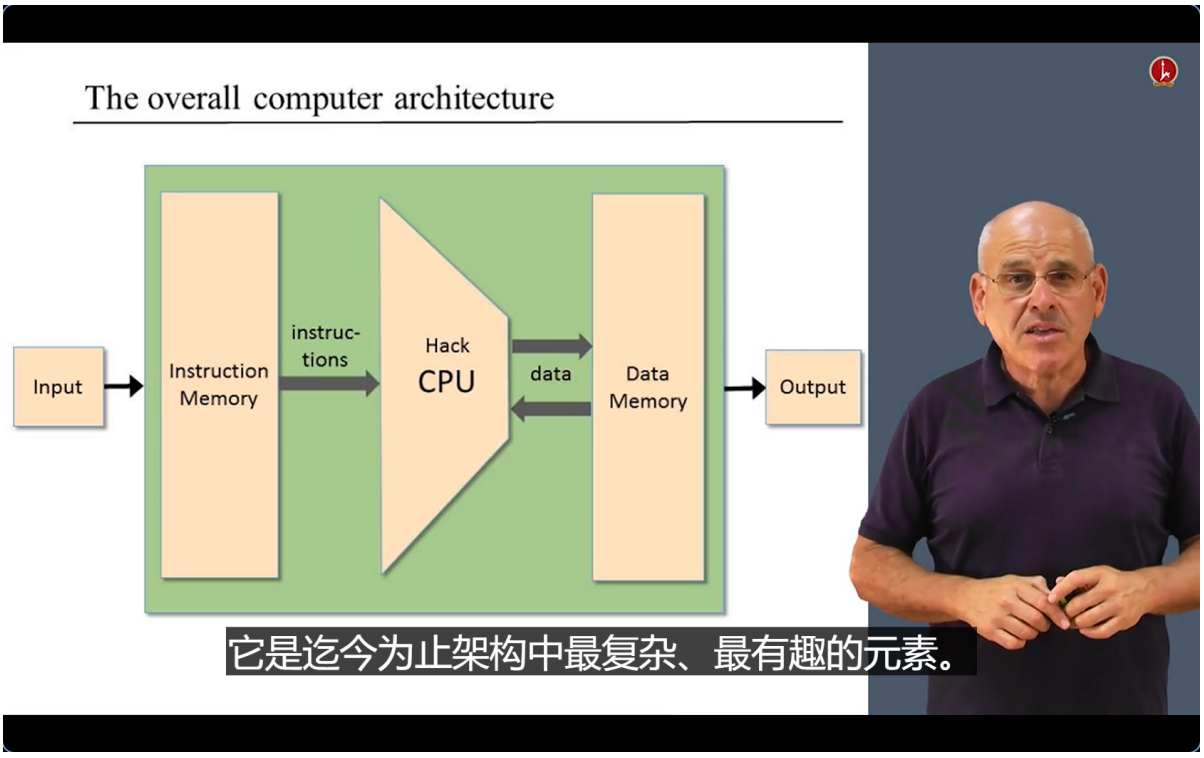
the address of
the next instruction



26:32 / 27:26 1x

Last： we finish CPU

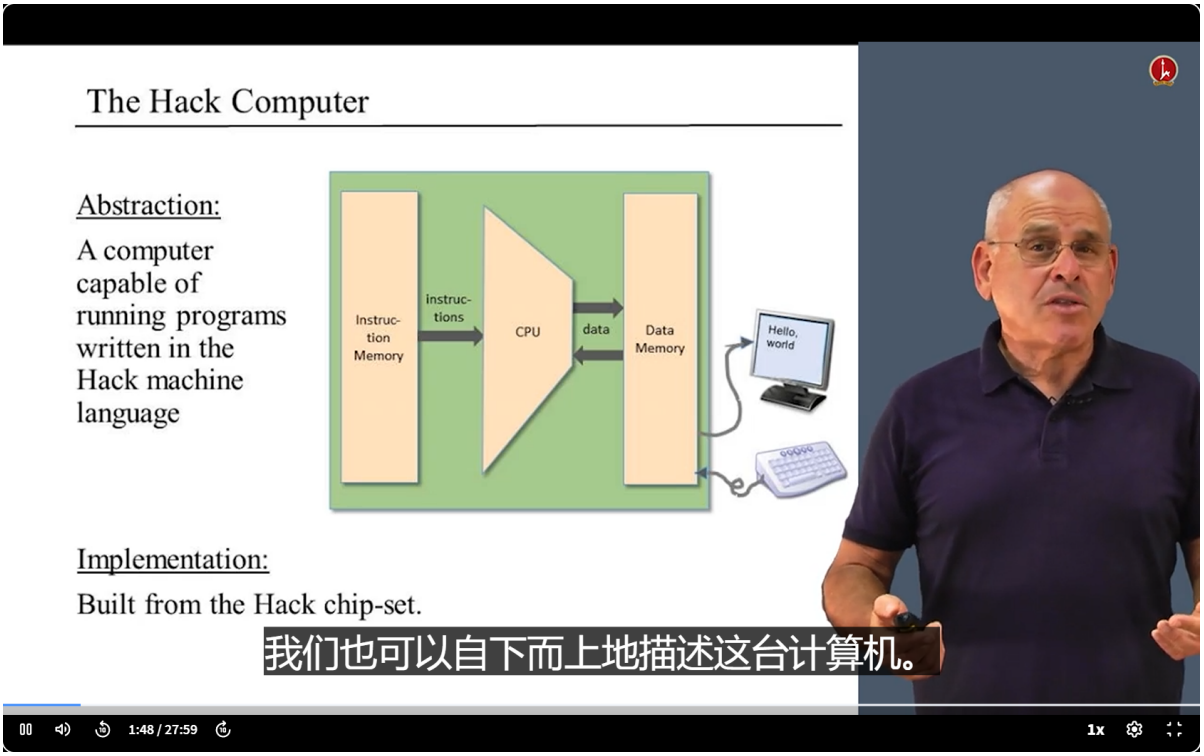




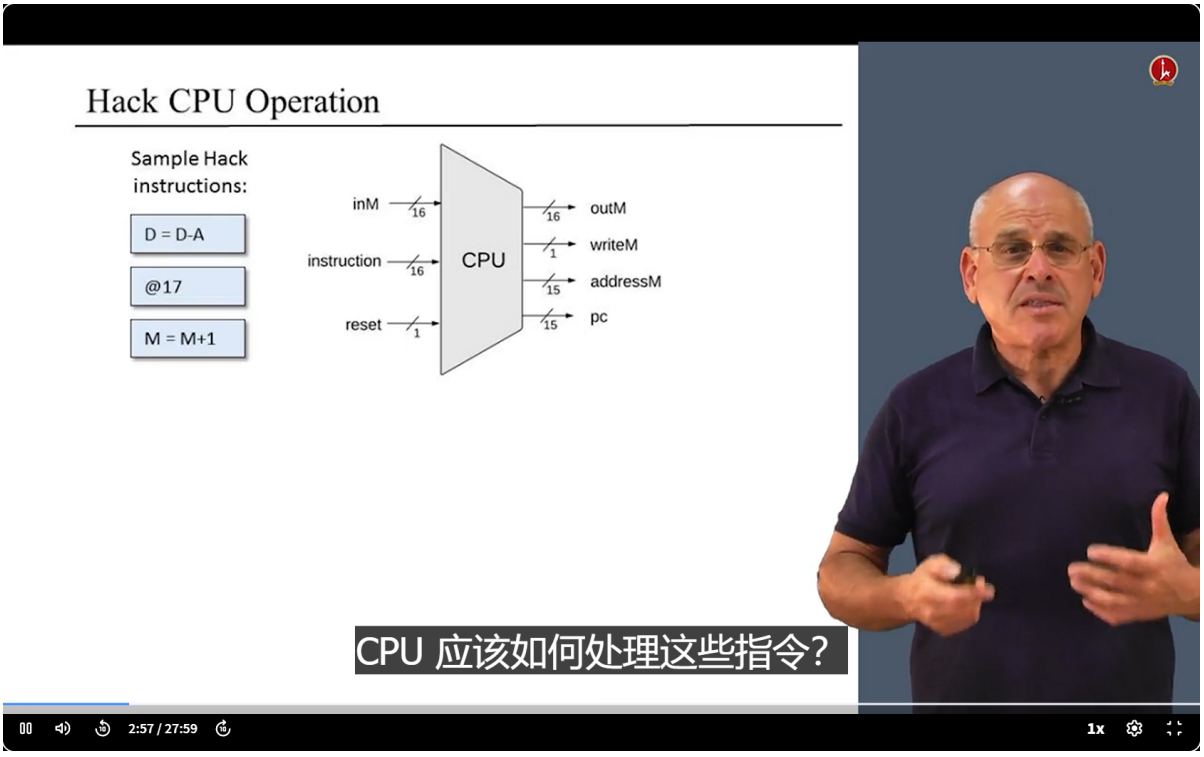

From Nand to Tetris
Building a Modern Computer from First Principles
www.nand2tetris.org
Week 5 / Unit 5.3
Central Processing Unit
Coming Up:
The Hack Computer
不是现在，而是在下一个单位之后。
27:09 / 27:26
1x


From Nand to Tetris
Building a Modern Computer from First Principles
www.nand2tetris.org
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

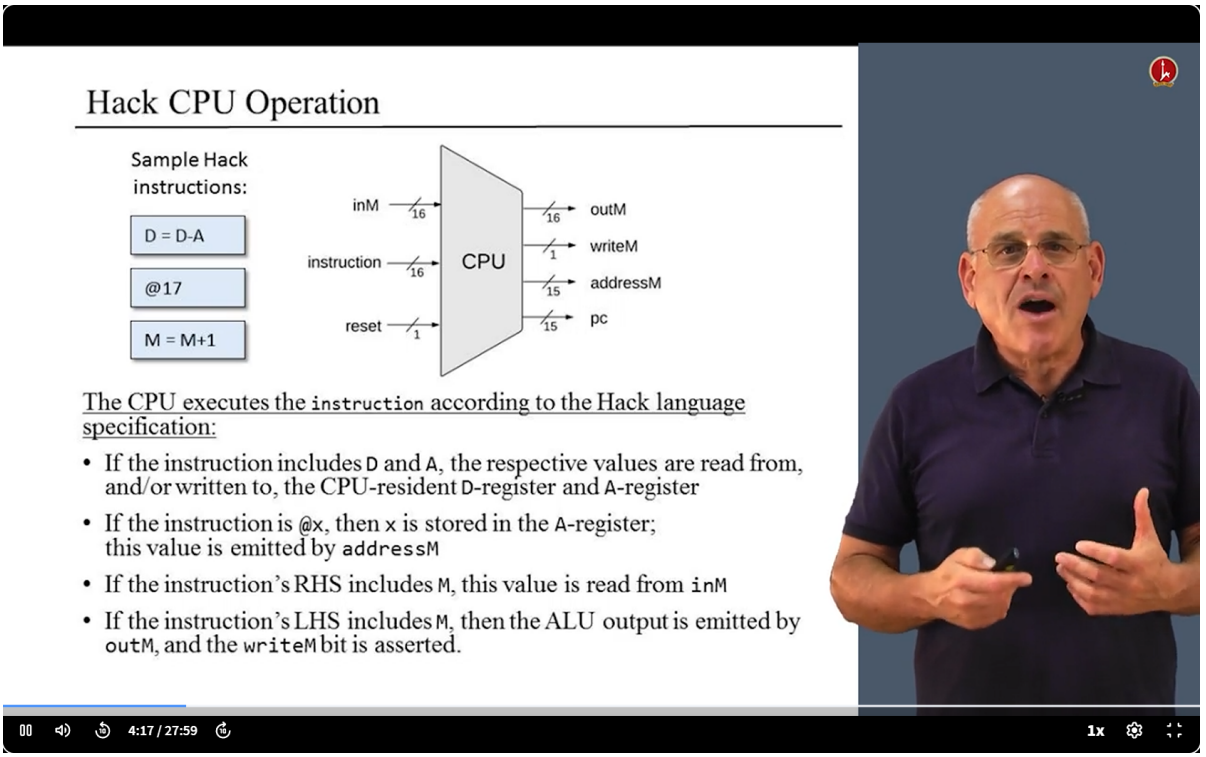
5.4 The Hack Computer



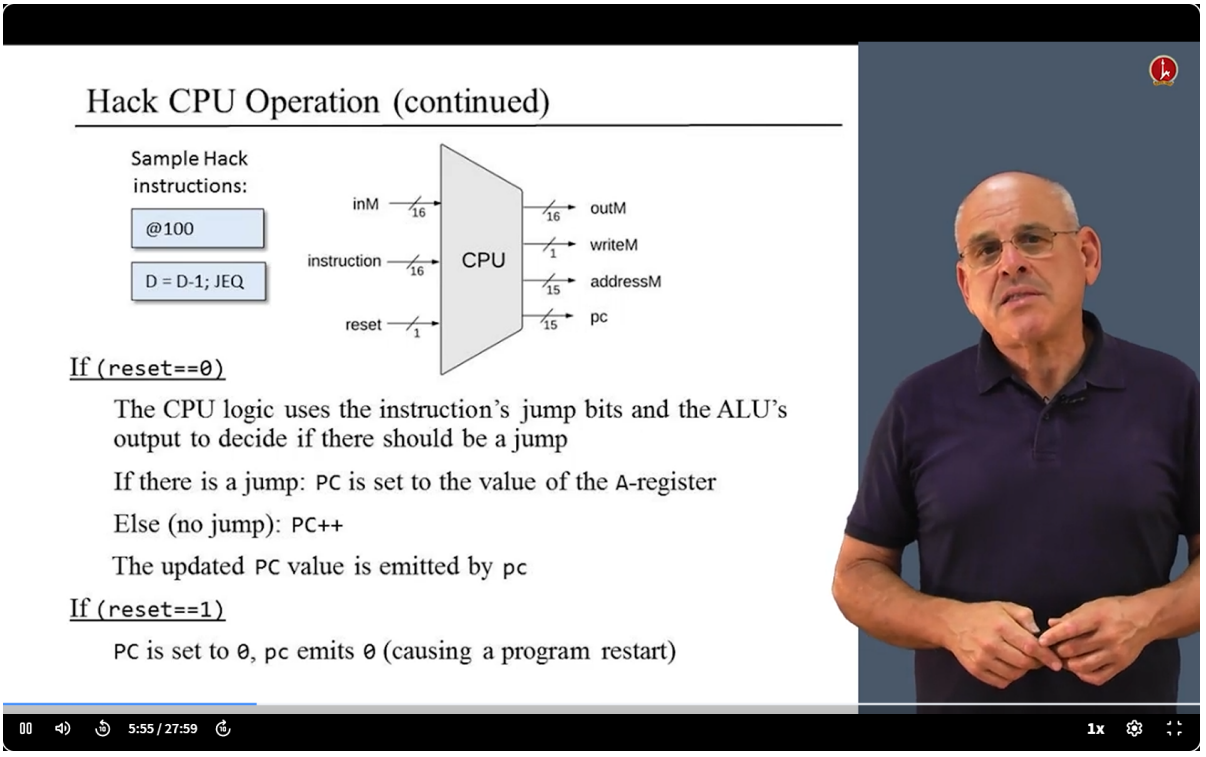
Hack CPU Operation



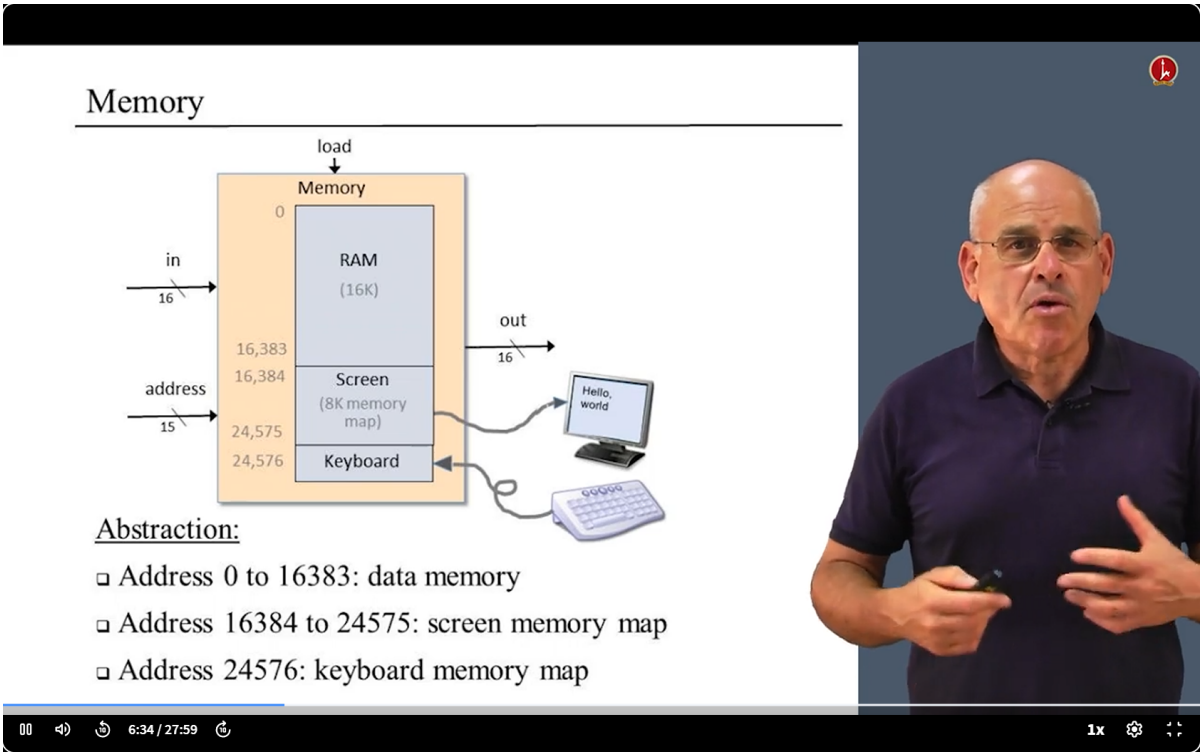
- 如果指令中提到了助记符“D”和“A”，CPU就会操作位于CPU内部的相应的**D寄存器**和**A寄存器**。
- 如果指令是一条A指令，在这种情况下，CPU会取出数据值，即这条指令中所谓的15位的“x”，然后将其放入A寄存器中。
- 如果指令的右侧(xxx=M)包含 “M”，这个值将从 “inM” 读取。
- 如果指令的左侧(M=xxx)包含 “M”，那么 ALU 的输出将通过 “outM” 输出，并且 “writeM” 位会→1。

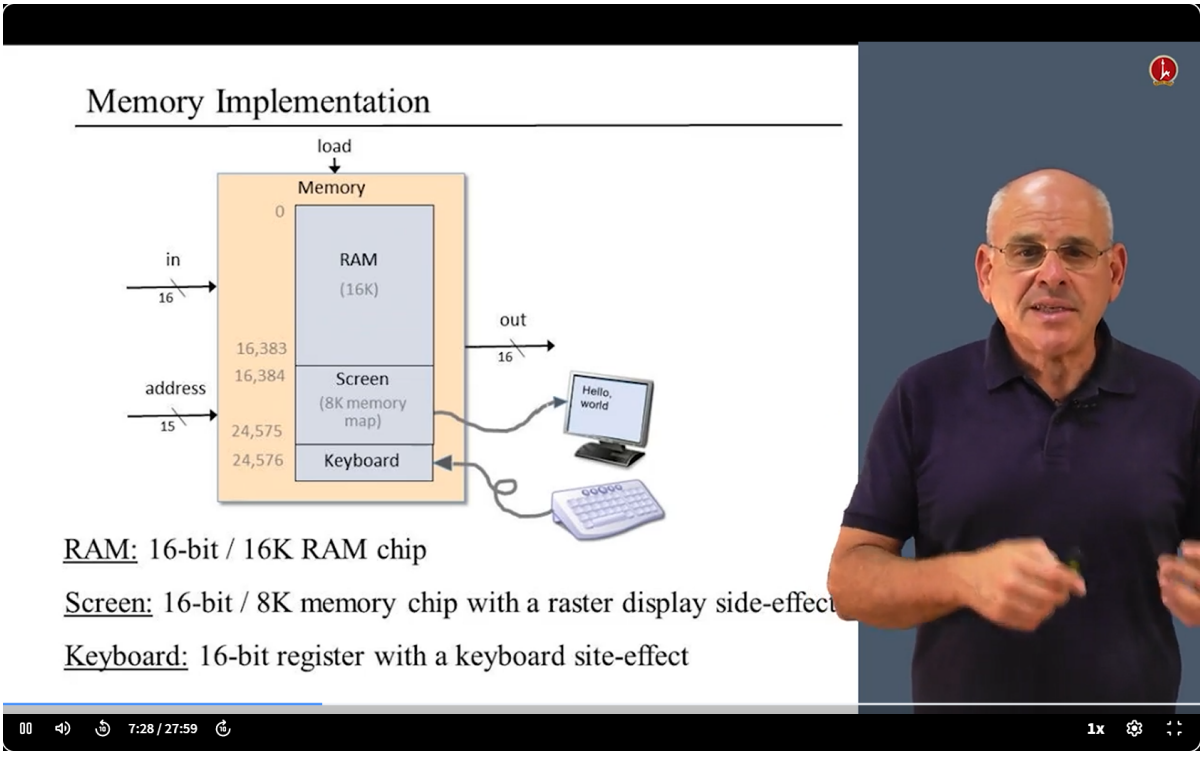


- 看看CPU如何处理跳转指令

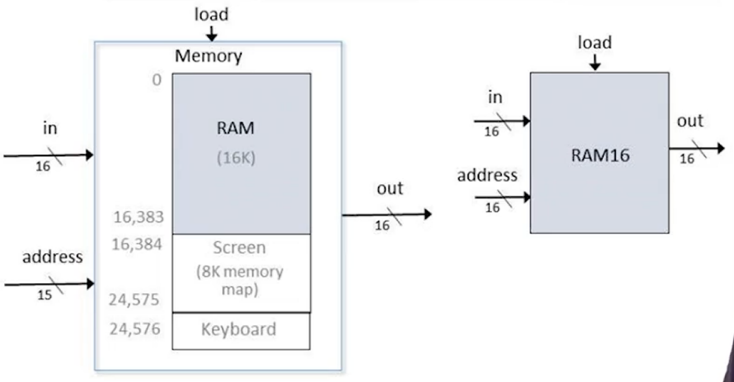


Memory


Memory
load
Memory
0
RAM
(16K)
16,383
16,384
Screen
(8K memory map)
24,575
24,576
Keyboard
in
16
address
15
out
16
Hello, world
Abstraction:
Address 0 to 16383: data memory
Address 16384 to 24575: screen memory map
Address 24576: keyboard memory map
6:34 / 27:59
1x


Memory Implementation
load
Memory
0
RAM
(16K)
16,383
16,384
Screen
(8K memory map)
24,575
24,576
Keyboard
in
16
address
15
out
16
Hello, world
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
7:28 / 27:59
1x

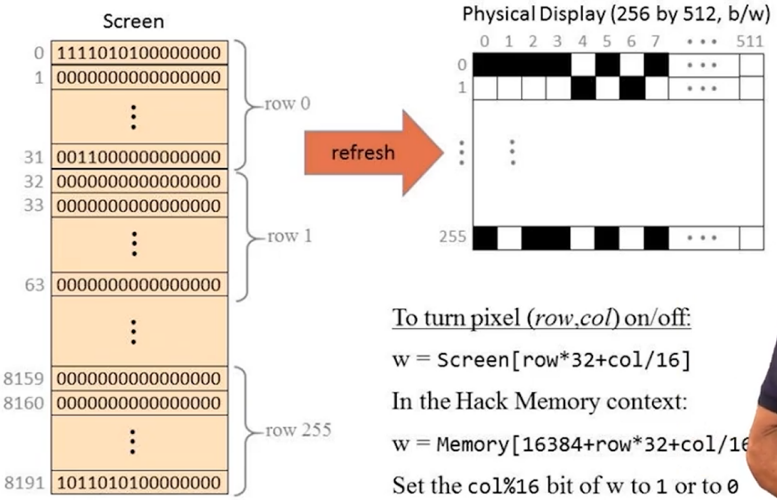
RAM



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



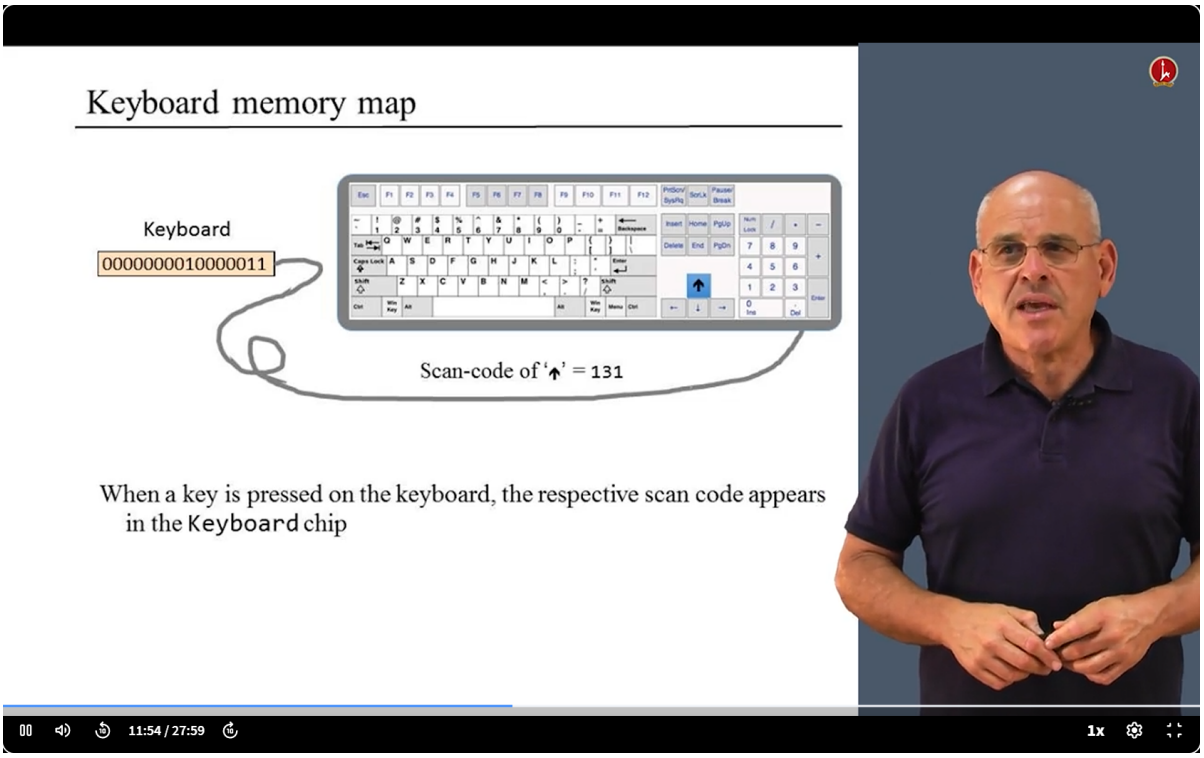
To turn pixel (*row*,*col*) on/off:

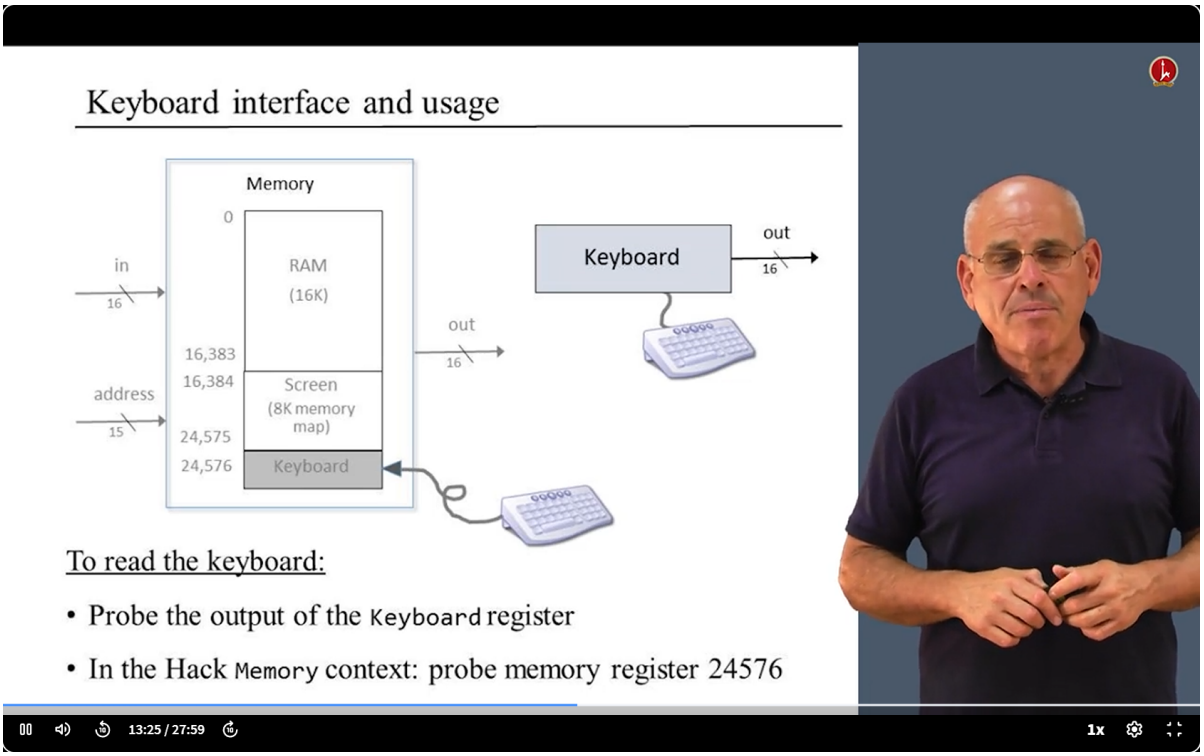
w = `Screen[row*32+col/16]`

In the Hack Memory context:

w = `Memory[16384+row*32+col/16`

Set the `col%16` bit of w to `1` or to `0`



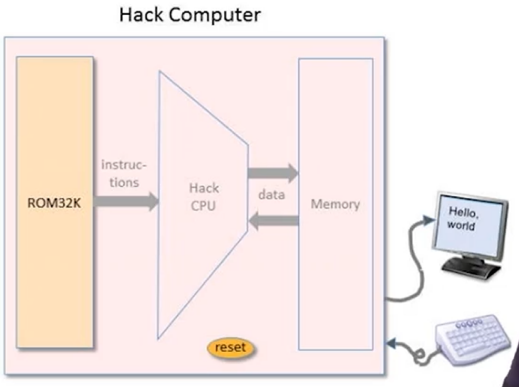


- 指令存储器 Instruction Memory（ROM：Read - Only Memory）

- 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。

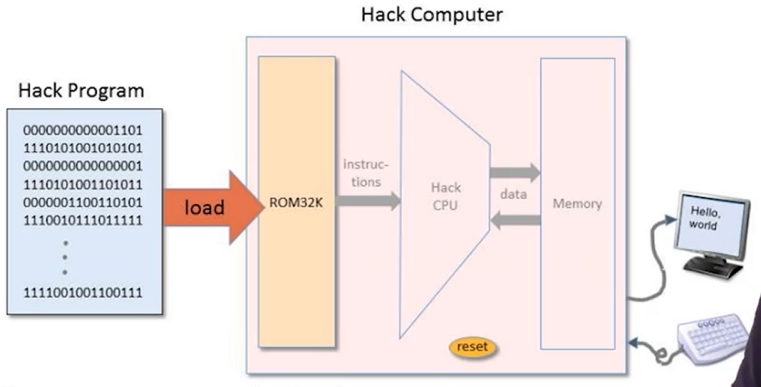
Instruction Memory (ROM)



使用一种我们称之为ROM 32K的芯片来实现这个指令存储器。

14:27 / 27:59 1x

Instruction Memory (ROM)

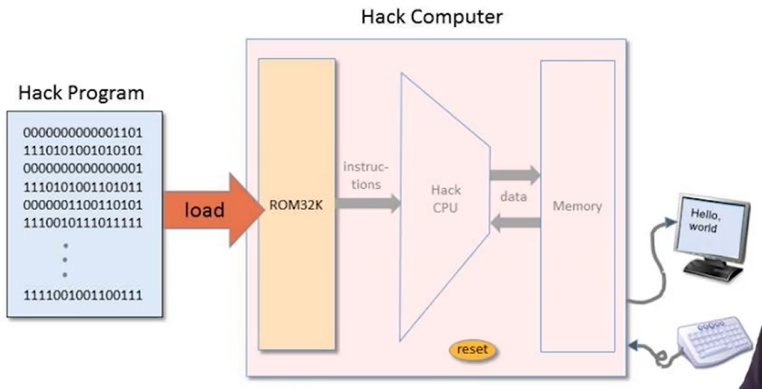


To run a program on the Hack computer:

- Load the program into the ROM

特别是，我们将其加载到ROM 32K芯片中。

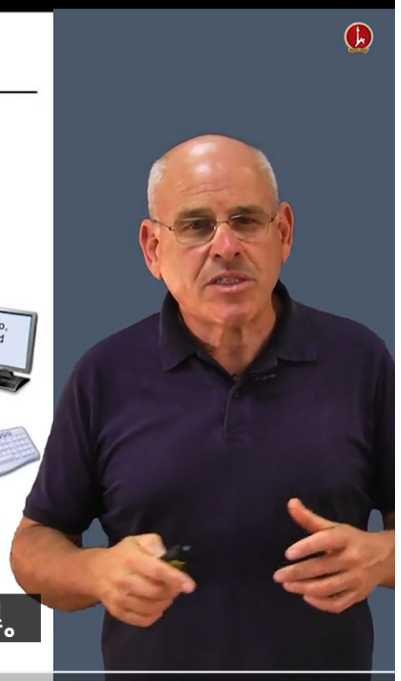
Instruction Memory (ROM)



To run a program on the Hack computer:

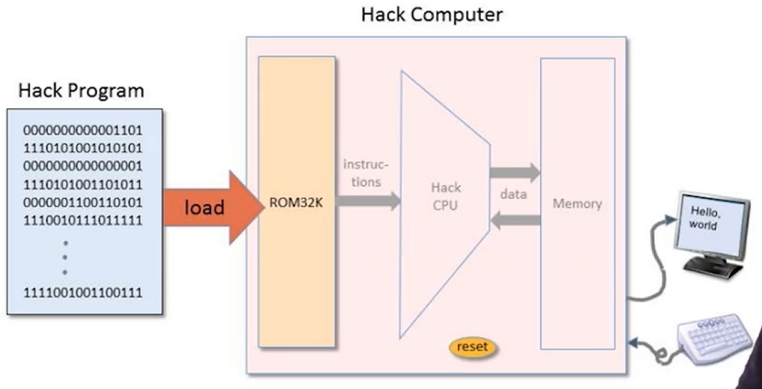
- Load the program into the ROM
- Press "reset"
- The program starts running.

这就是我们想要实施的障碍。



15:04 / 27:59 1x

Loading a program

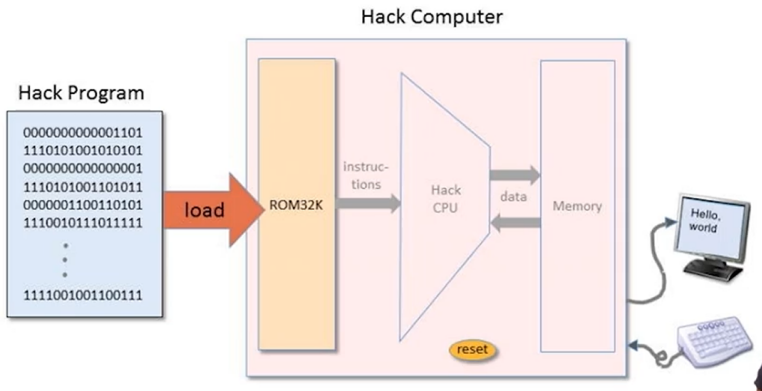


- Hardware implementation: plug-and-play ROM chips

你想玩另一款游戏，你，你插入另一个卡带等等。

16:03 / 27:59 1x

Loading a program



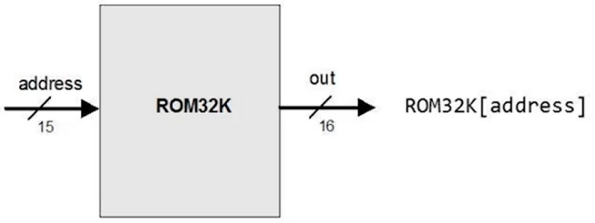
- Hardware implementation: plug-and-play ROM chips
- Hardware simulation: programs are stored in text files;

built-in ROM chip

实现它的另一种方法是使用硬件模拟

16:10 / 27:59 1x

ROM interface

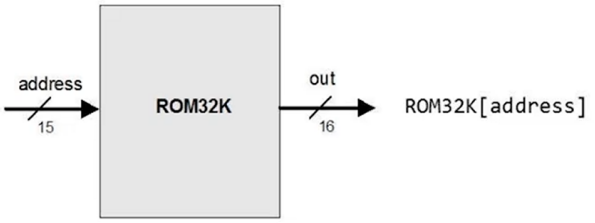


- Implemented as a built-in ROM32K chip
- Contains a sequence of Hack instructions (program)

是通过地址输入选择的寄存器的内容。

23:19 / 27:59 1x

ROM interface

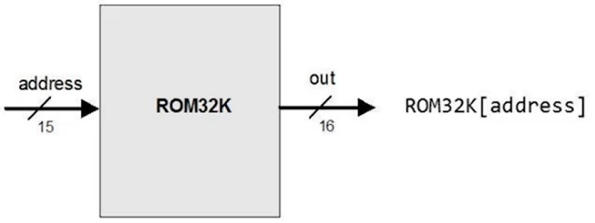


- Implemented as a built-in ROM32K chip
- Contains a sequence of Hack instructions (program)

因此，如果我在地址输入中输入 17，

23:24 / 27:59 1x

ROM interface

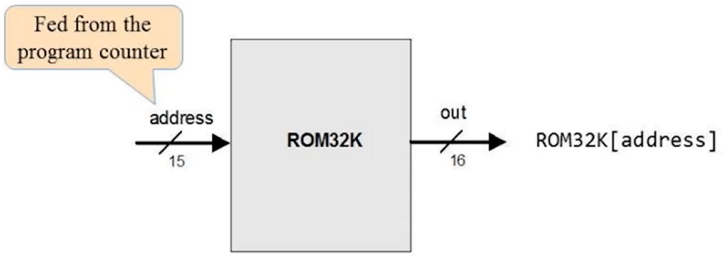


- Implemented as a built-in ROM32K chip
- Contains a sequence of Hack instructions (program)

就会出现寄存器编号为 17 的内容。

23:28 / 27:59 1x

ROM interface

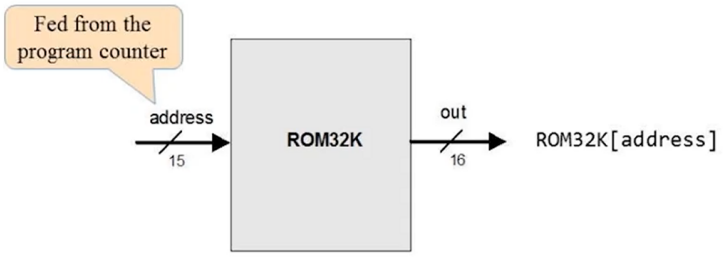


- Implemented as a built-in ROM32K chip
- Contains a sequence of Hack instructions (program)

因此，拿起这个 ROM 并将其连接到程序计数器很有意义，

23:33 / 27:59 1x

ROM interface

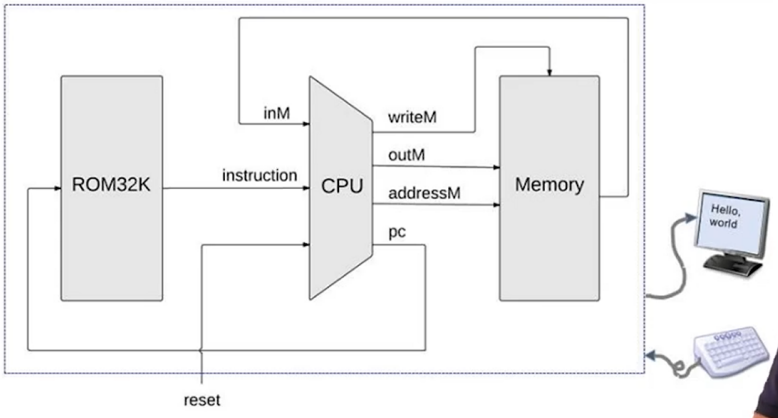


- Implemented as a built-in ROM32K chip
- Contains a sequence of Hack instructions (program)

因为程序计数器总是允许下一条指令的地址。

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