



Lecture 5

Computer Architecture

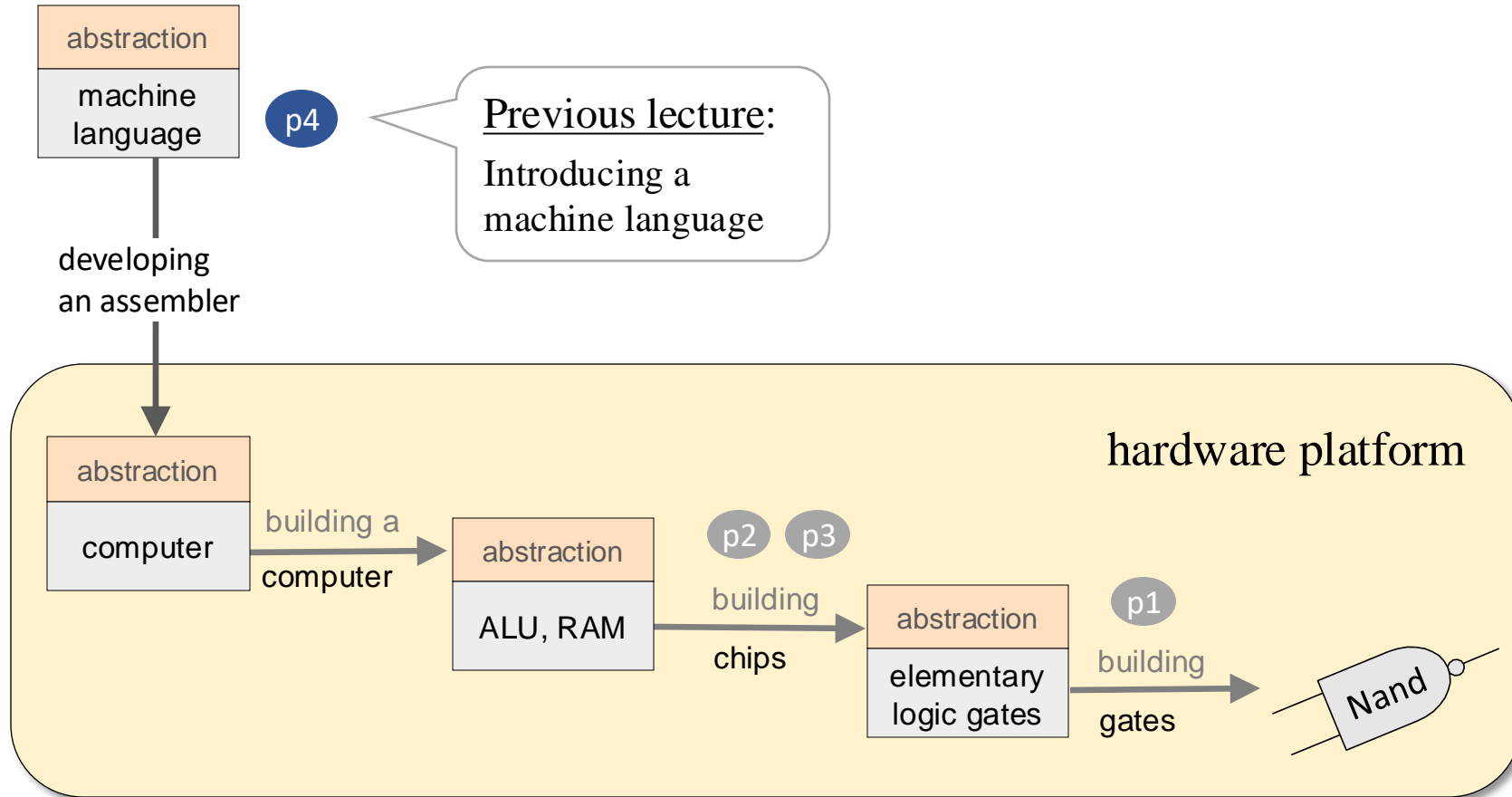
These slides support chapter 5 of the book

The Elements of Computing Systems

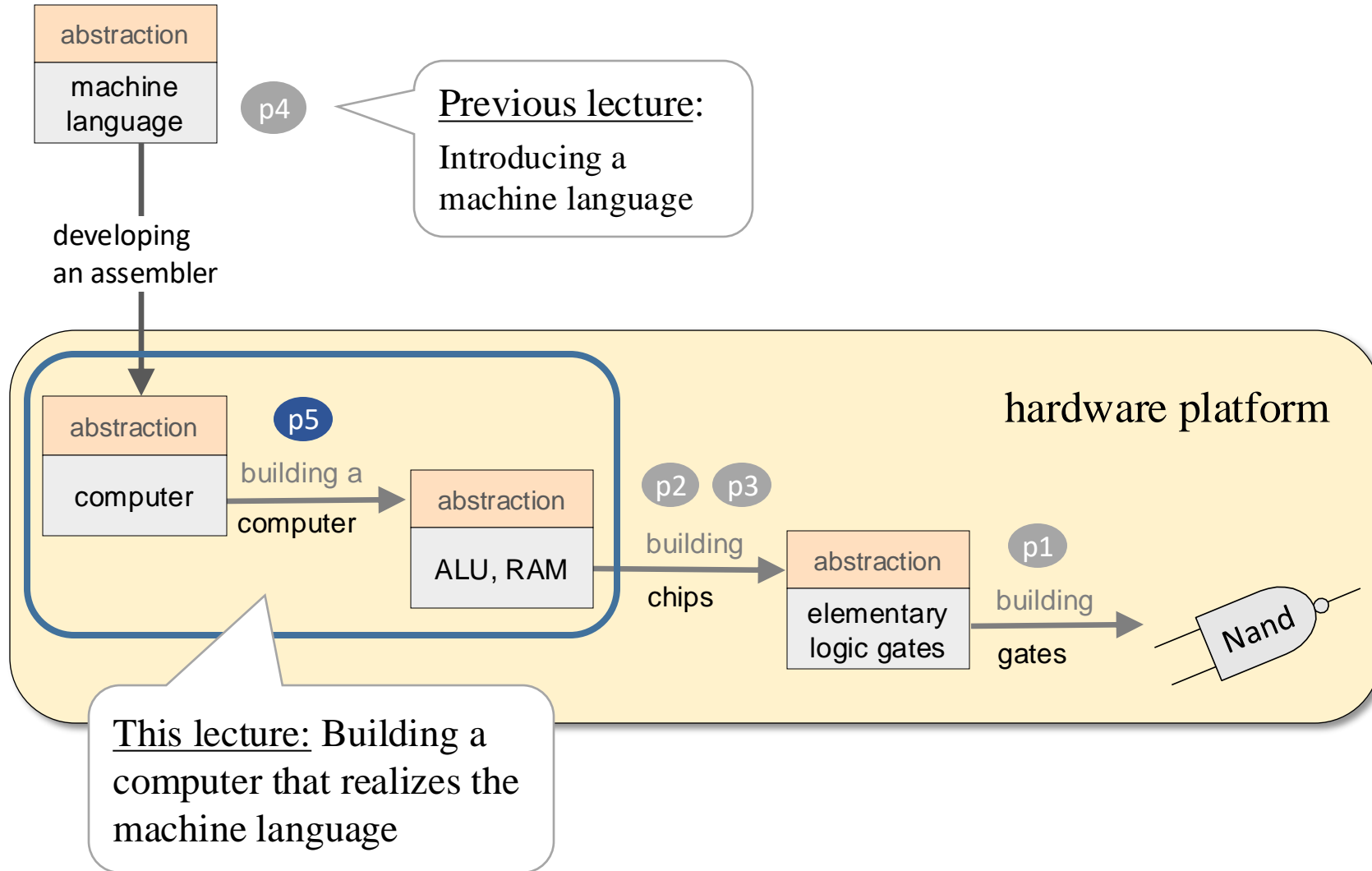
By Noam Nisan and Shimon Schocken

MIT Press, 2021

Nand to Tetris Roadmap: Hardware



Nand to Tetris Roadmap: Hardware

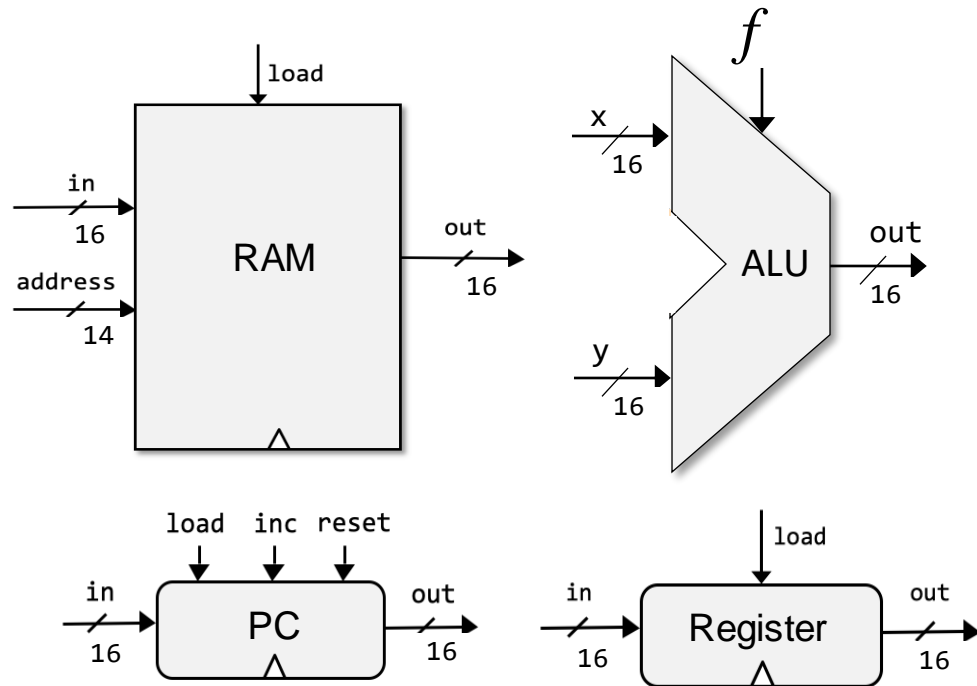


Nand to Tetris Roadmap: Hardware

The challenge

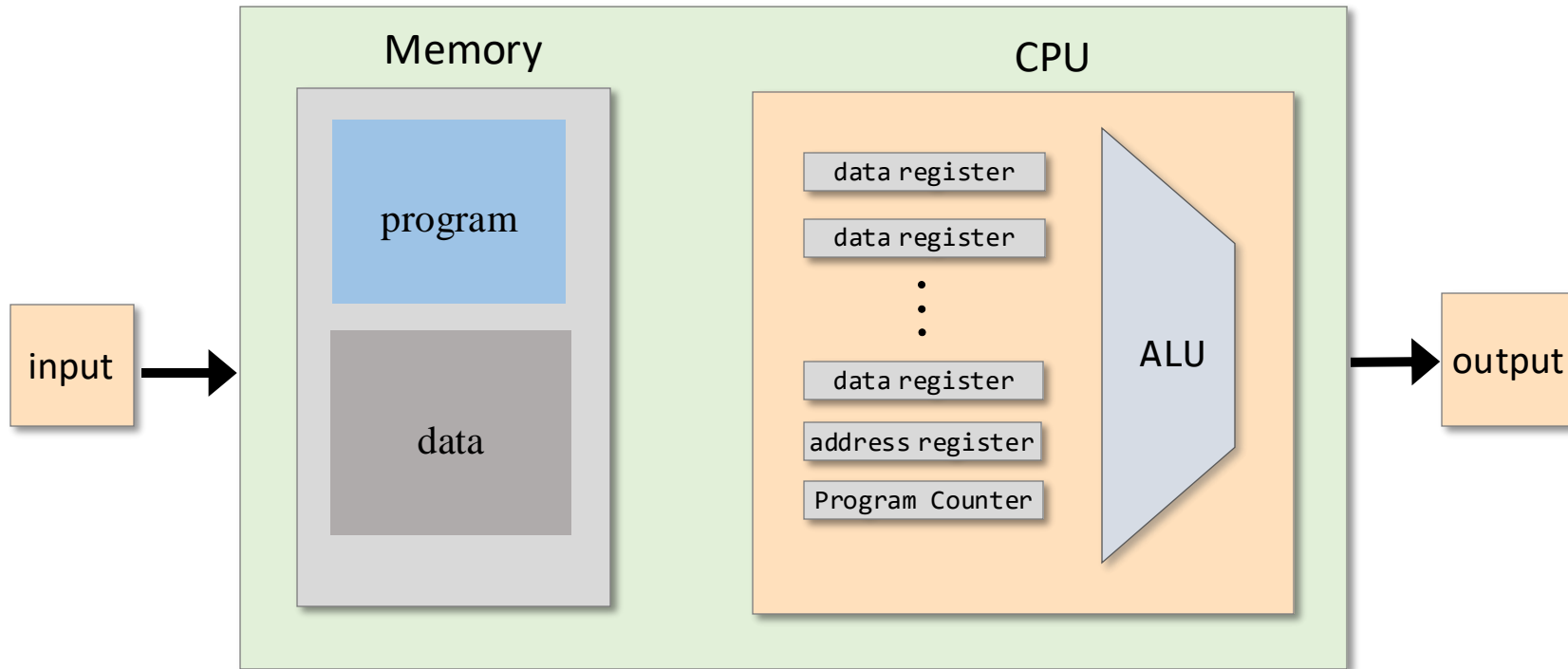
Integrate the chips built
in chapters 1, 2, 3...

... into an architecture that
executes *any program* written in
the machine language introduced
in chapter 4



```
// Computes R1 = 1 + 2 + 3 + ... + R0
// i = 1
@i
M=1
// sum = 0
@sum
M=0
(LOOP)
// if(i > R0) goto STOP
@i
D=M
@R0
D=D-M
@STOP
D;JGT
...
```

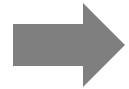
Computer Architecture



- Processor, registers, memory
- Stored program concept
- *General-purpose*

We'll build the Hack computer – a variant of this architecture.

Computer Architecture



Basic architecture

- Fetch-Execute cycle
- The Hack CPU
- Input / output
- Memory
- Computer
- Project 5: Chips
- Project 5: Guidelines

Early computers: 17th century

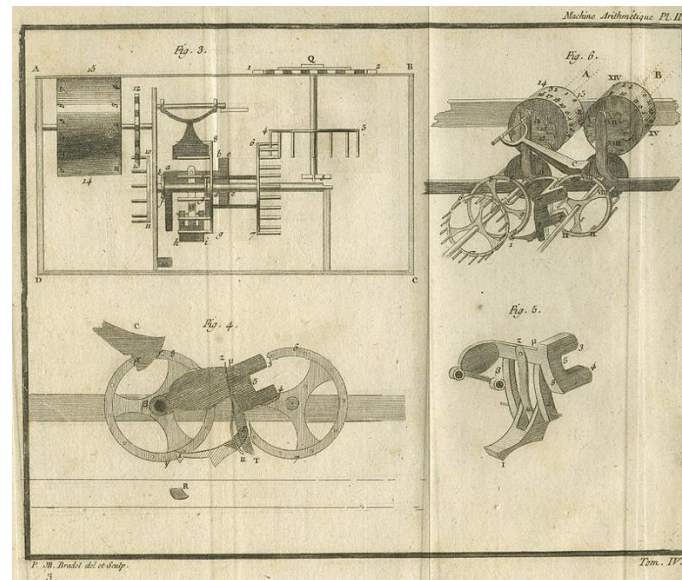


Blaise Pascal
1623 – 1662

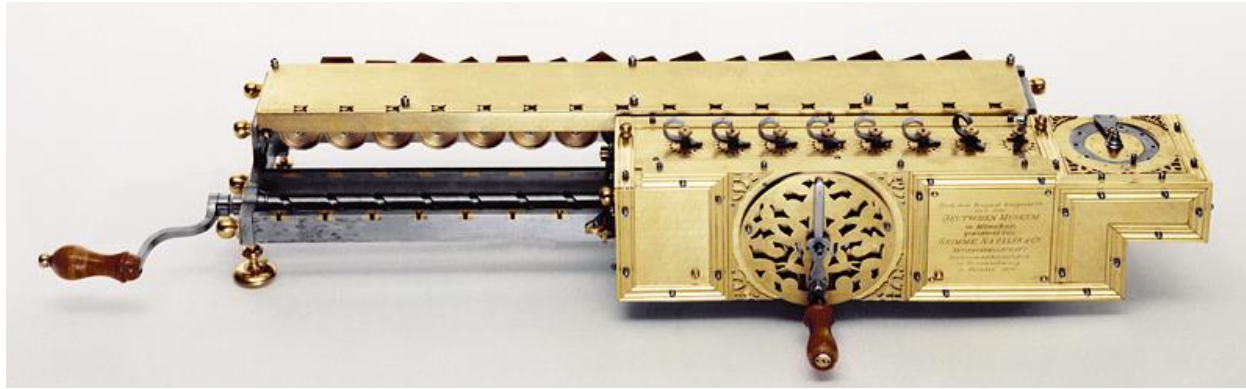


Pascal's Calculator (*Pascaline*, 1652)

- Add
- Subtract



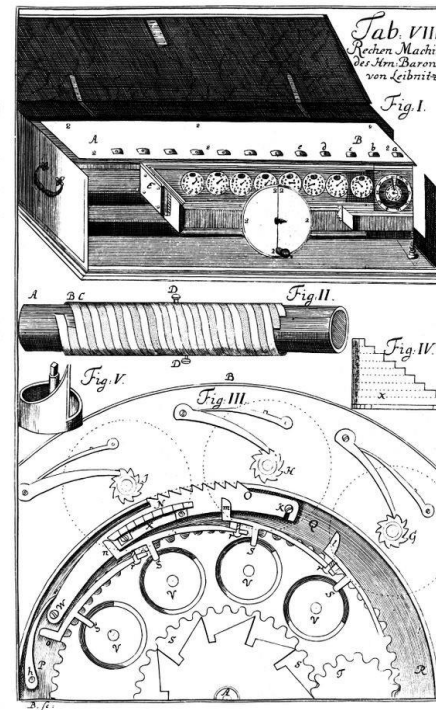
Early computers: 17th century



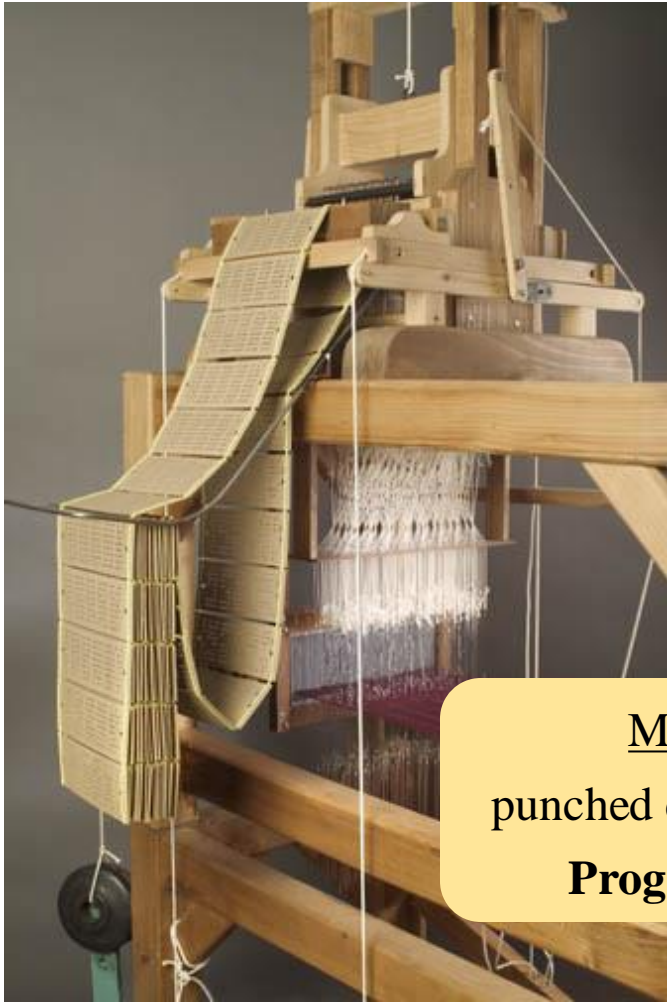
Gottfried Leibniz
1646 – 1716

Leibniz Calculator (1673)

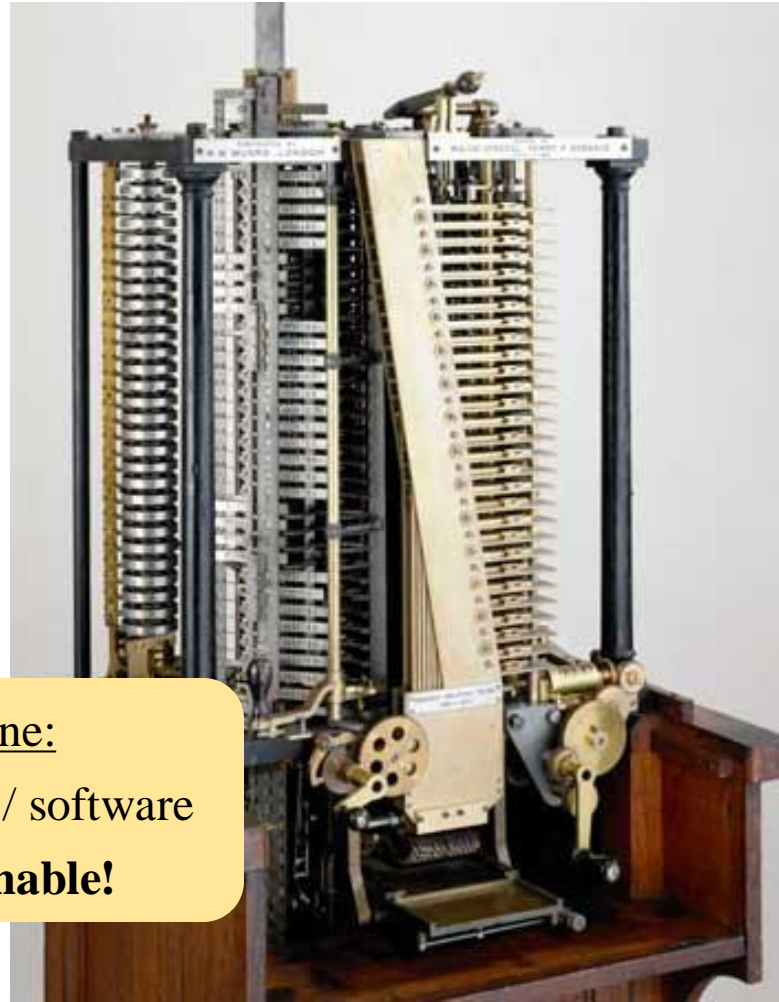
- Add
- Subtract
- Multiply
- Divide.



Early computers: 19th century



mechanical loom
(Jacquard, 1804)



mechanical calculator
(Babbage, 1837)

Milestone:
punched cards / software
Programmable!

Modern computers: 20th century



John Von Neumann



John Mauchly



Presper Eckert



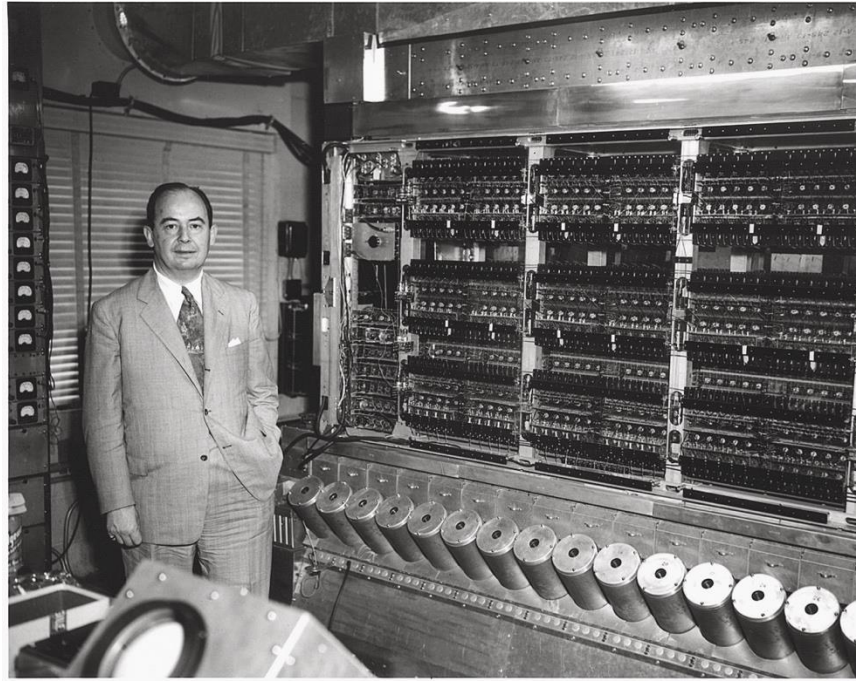
John Atanasoff



Howard Aiken



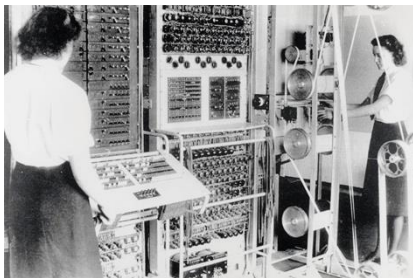
Konrad Zuse



ENIAC: First digital, programmable,
stored program computer

University of Pennsylvania, 1946,

(Inspired by many other
early computers and innovators)



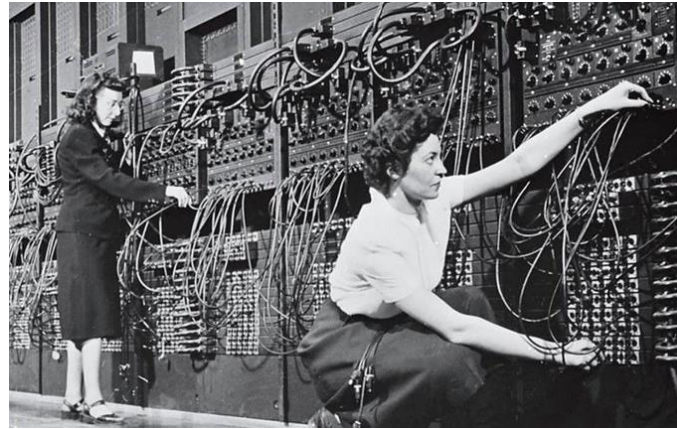
Tommy Flowers

Colossus: First digital, programmable,
computer, UK, 1945

Modern computers: 20th century



Kathleen McNulty, Jean Jennings, Frances Snyder,
Marlyn Wescoff, Frances Bilas, Ruth Lichterman



Eniac women

Pioneered reusable code, subroutines,
flowcharts, compilation, ...
many other software innovations



Grace Hopper



Adele Koss

Compilation pioneers

Modern computers: 20th century

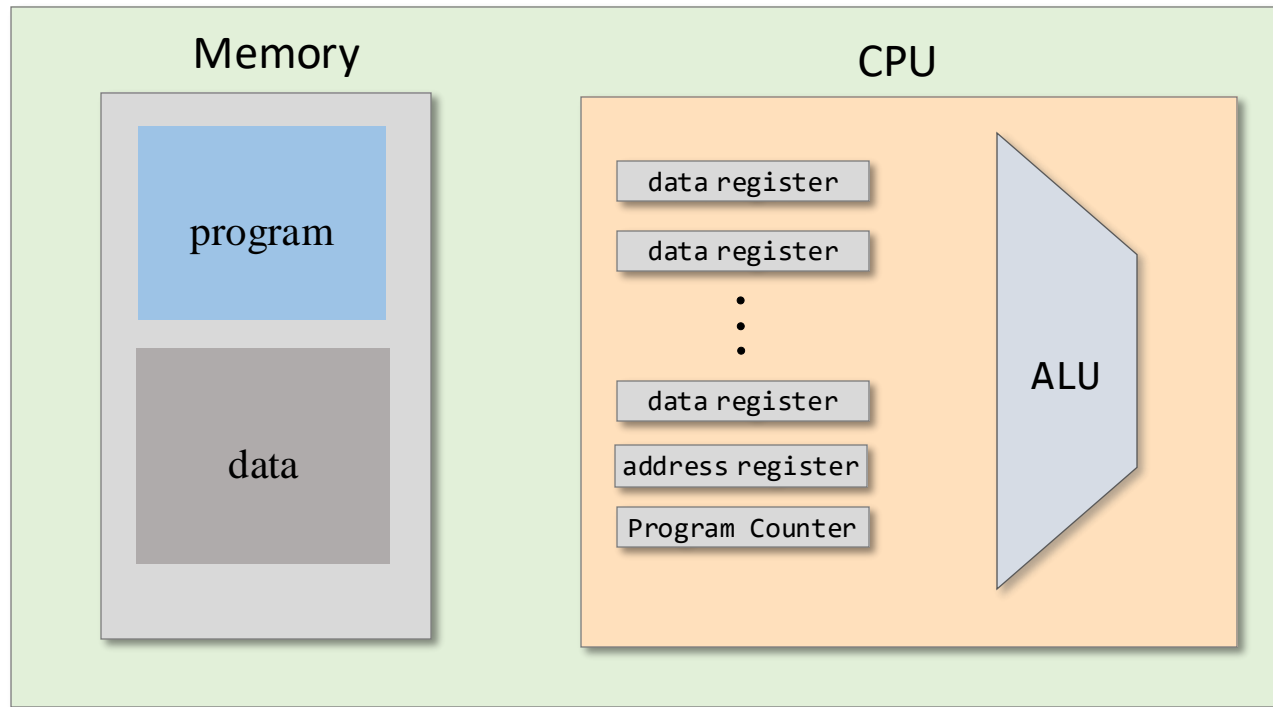
Same **hardware** can run many different programs (**software**)



“If it should turn out that the basic logic of a machine designed for the numerical solution of differential equations coincides with the logic of a machine intended to make bills for department stores, I would regard this as the most amazing coincidence I have ever encountered” — Howard Aiken (Mark 1 computer architect, 1956)

“The *stored program computer*, as conceived by Alan Turing and delivered by John von Neumann, broke the distinction between numbers that *mean things* and numbers that *do things*. Our universe would never be the same” ([George Dyson](#))

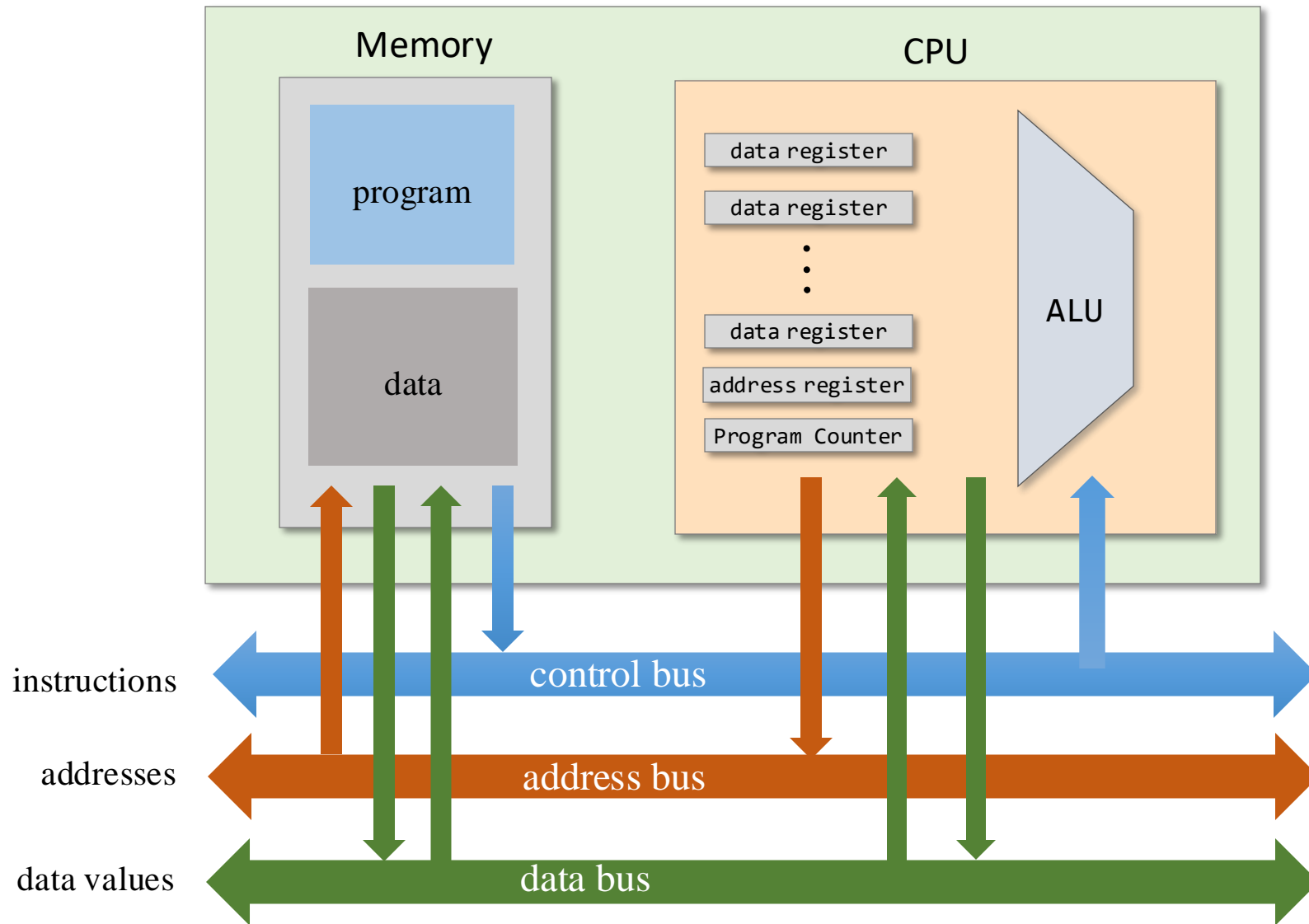
Basic architecture



Computer:

A machine that uses instructions to manipulates data

Basic architecture



The computer can be viewed as a set of chips, connected by pathways (buses).

Computer architecture

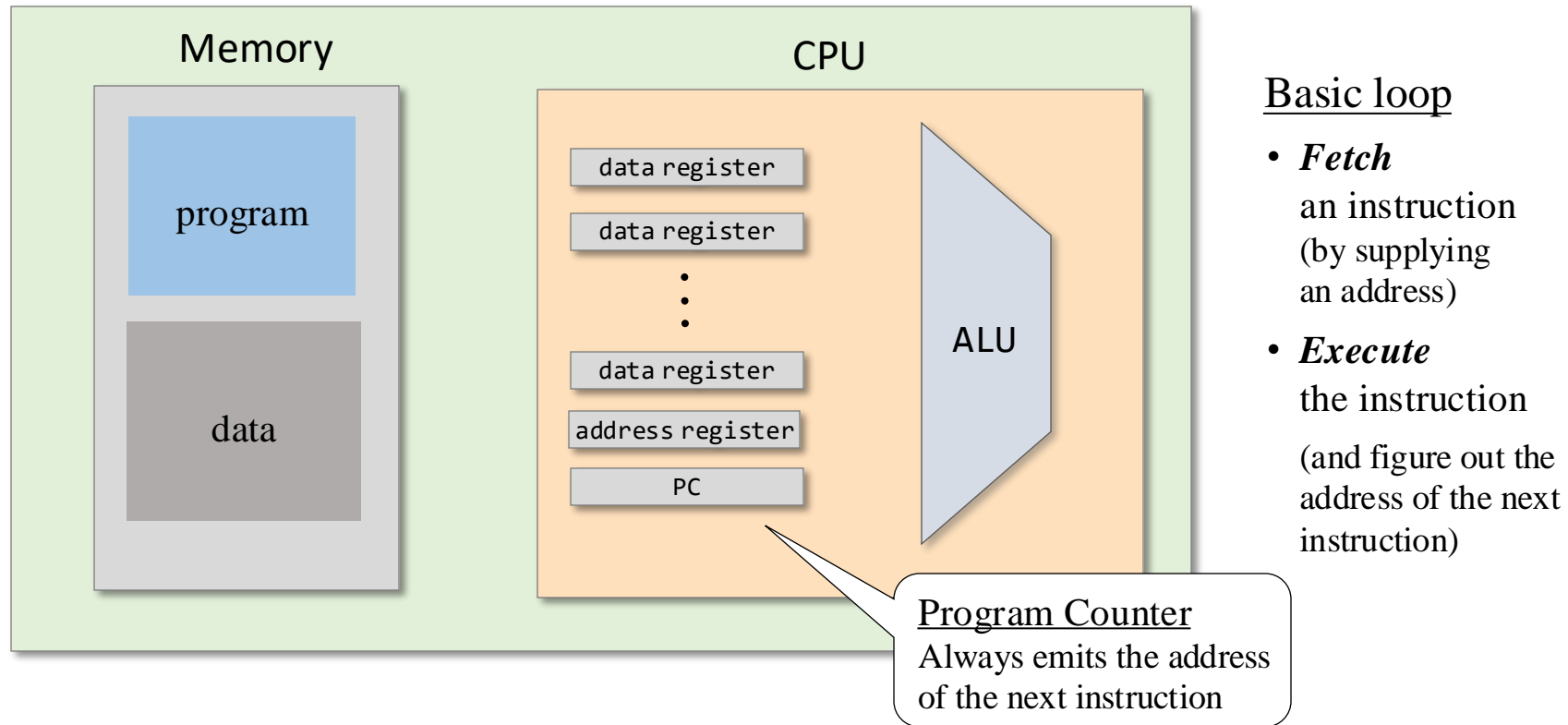
✓ Basic architecture

➡ Fetch-Execute cycle

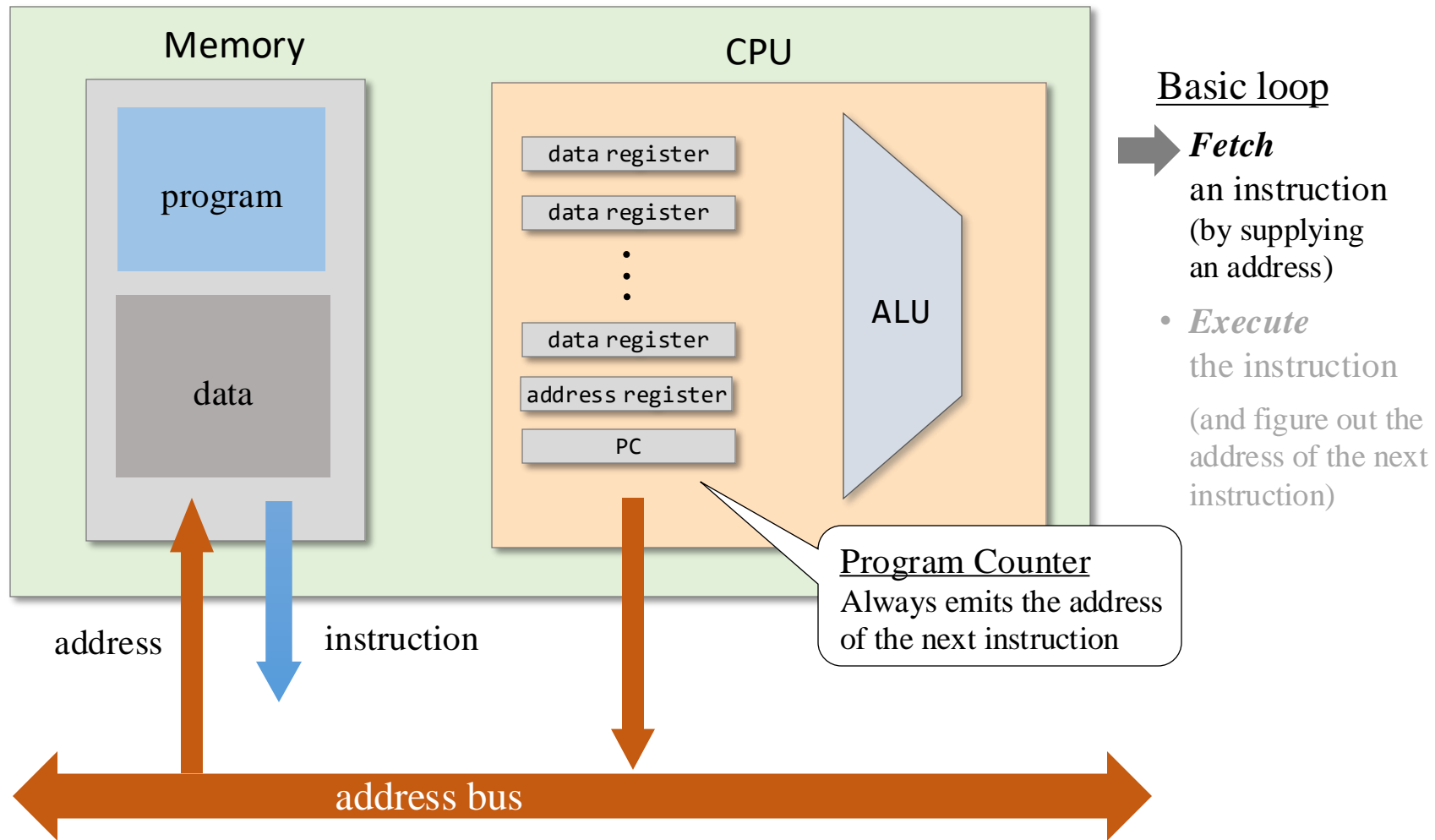
- The Hack CPU
- Input / output

- Memory
- Computer
- Project 5: Chips
- Project 5: Guidelines

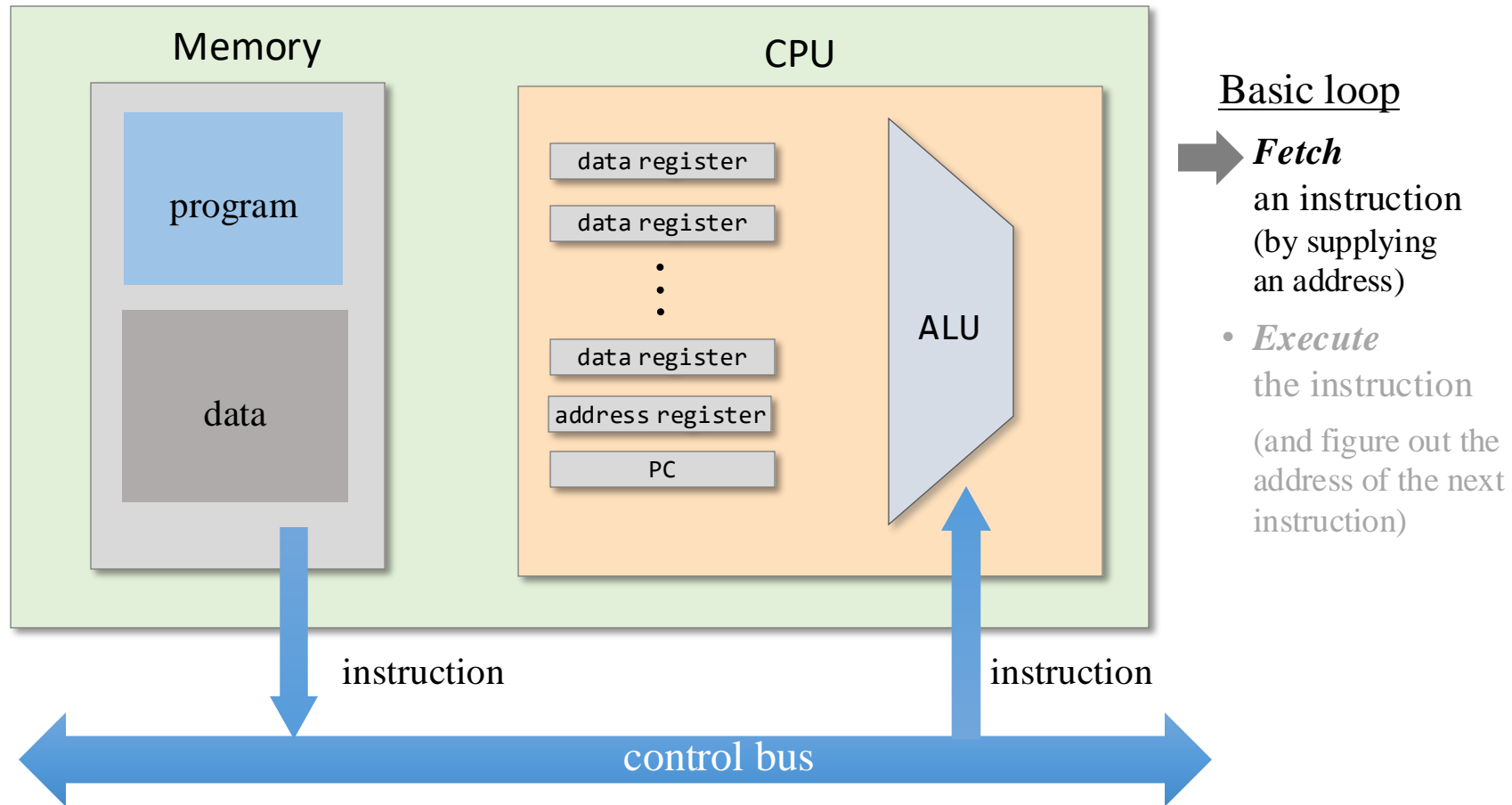
Computer architecture



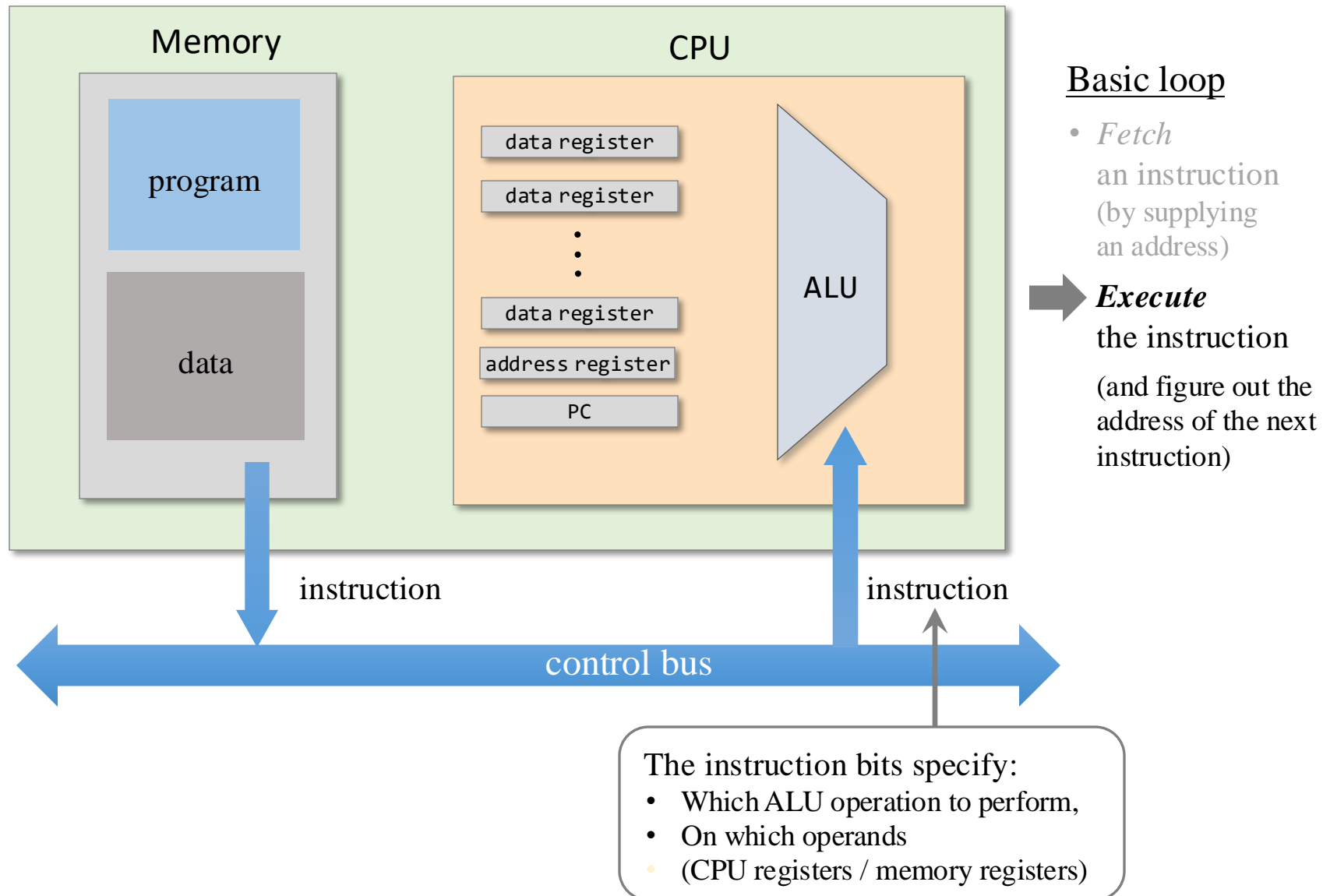
Fetch an instruction



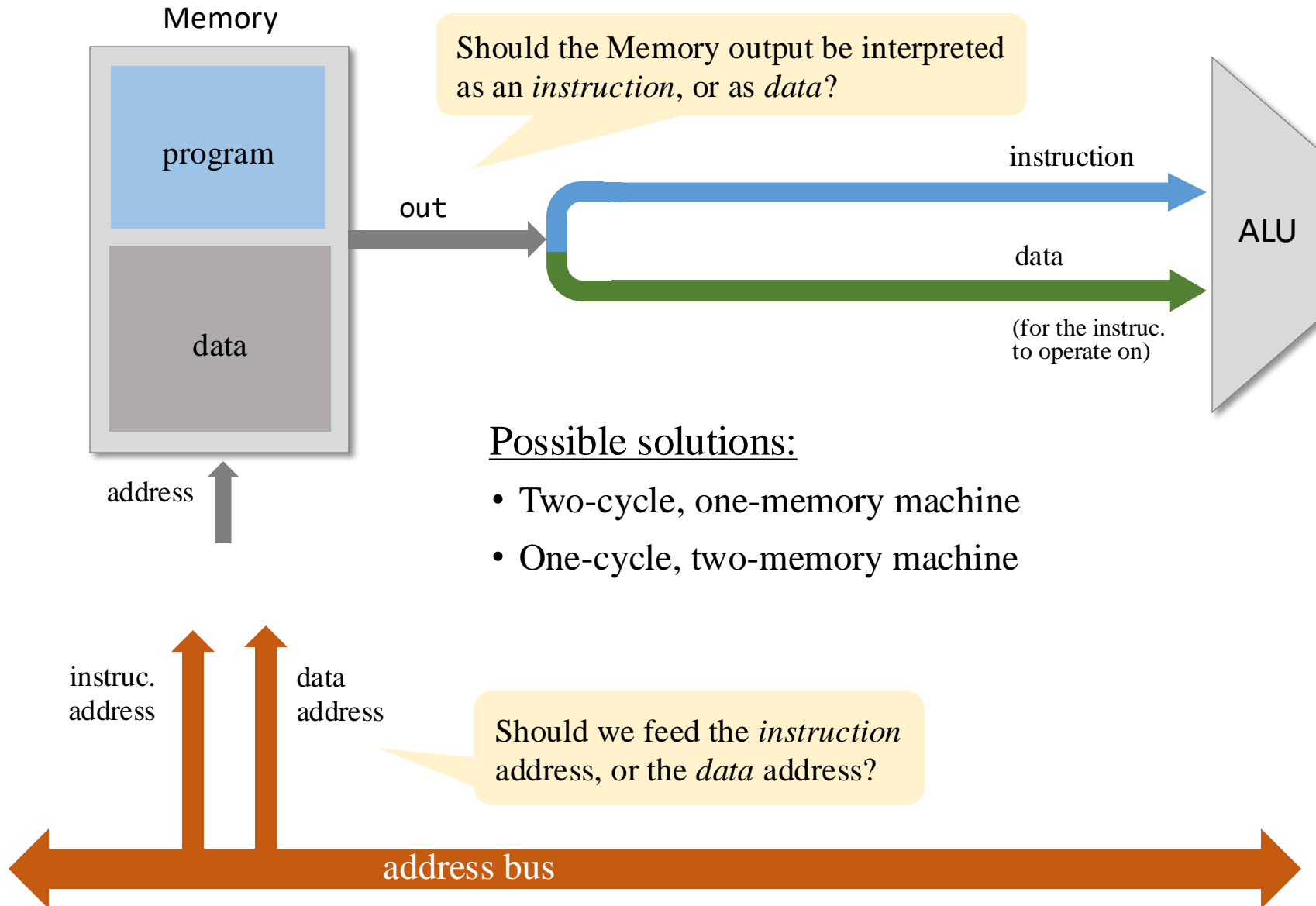
Fetch an instruction



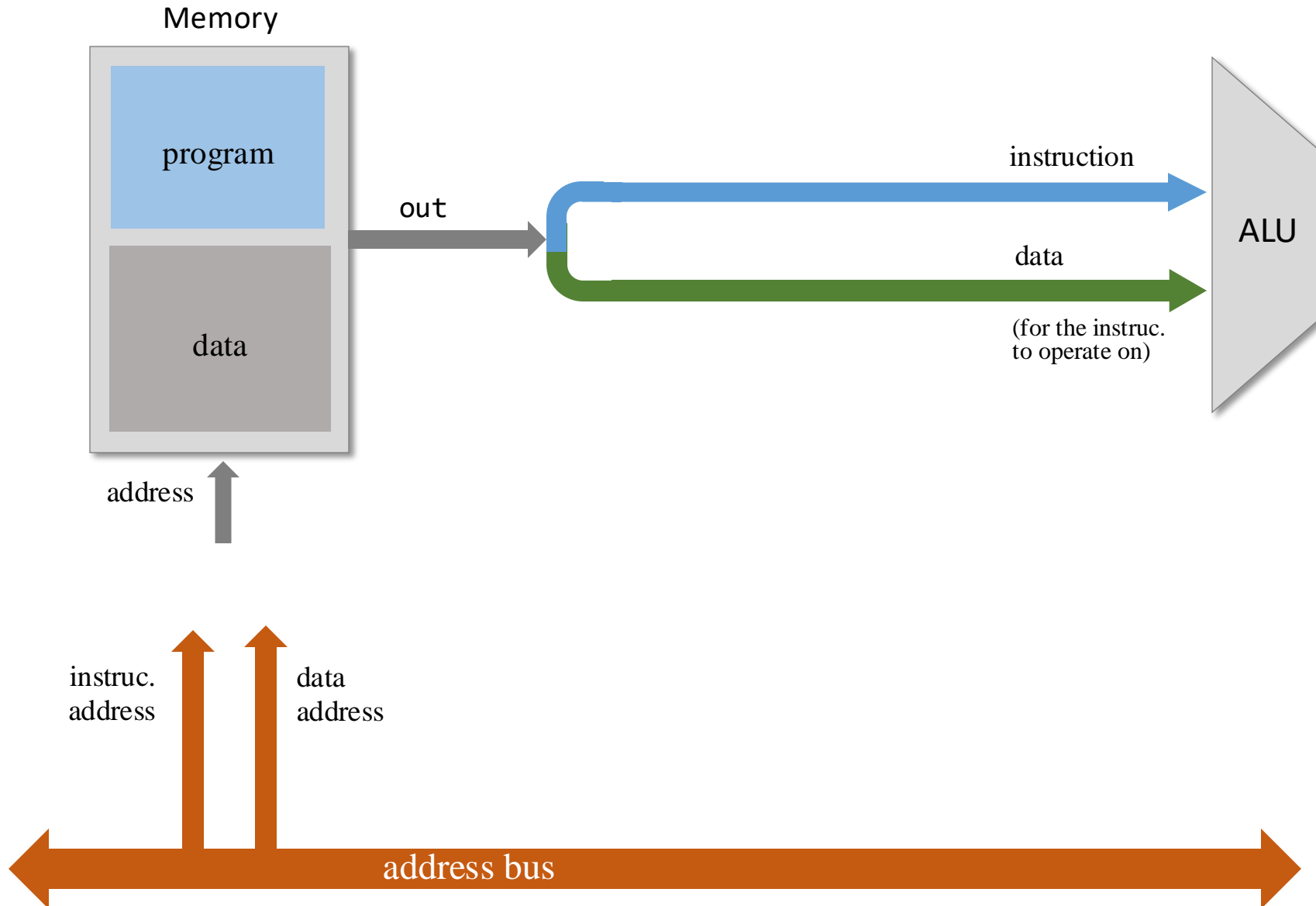
Execute the instruction



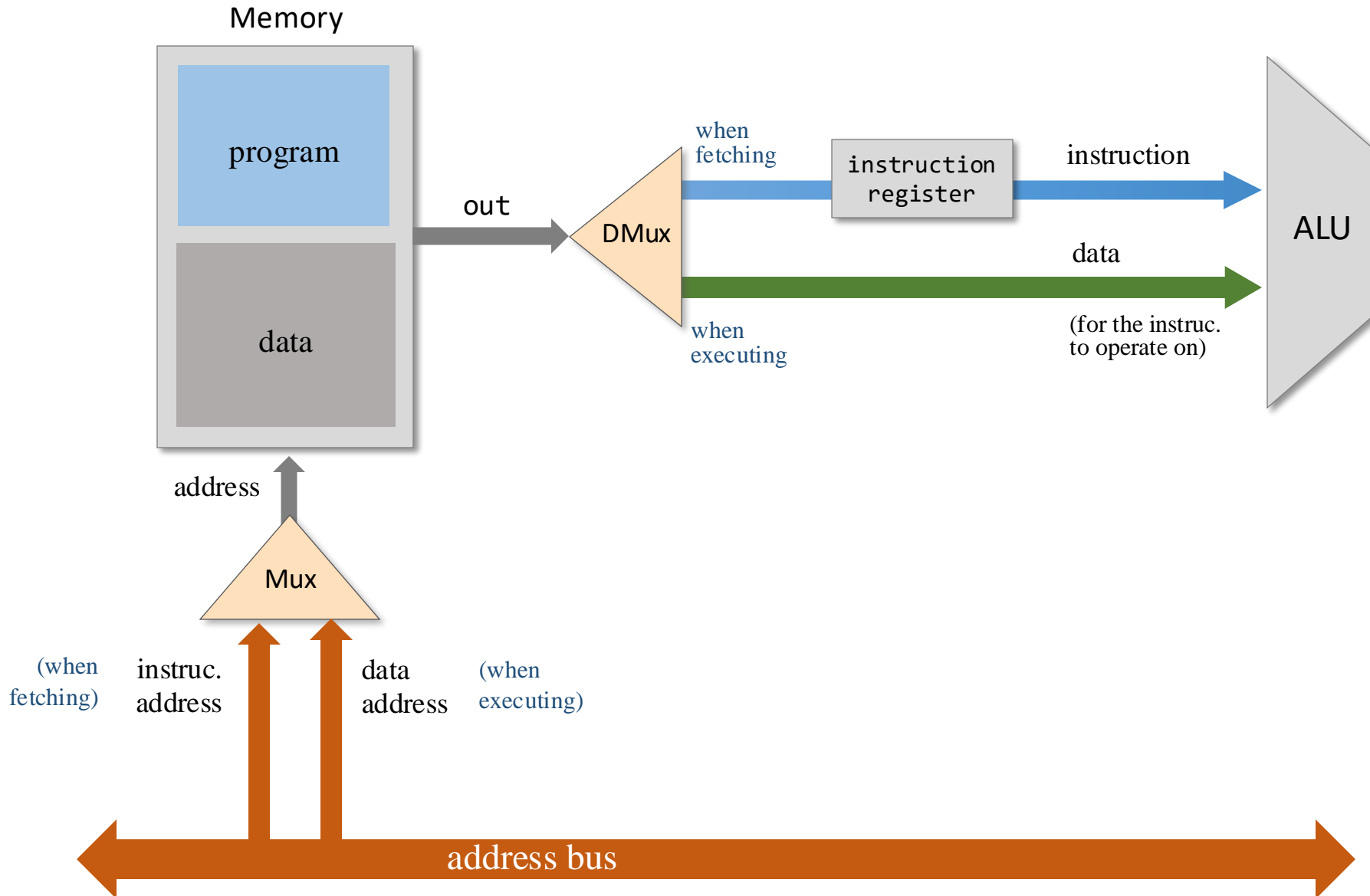
Fetch – Execute issues



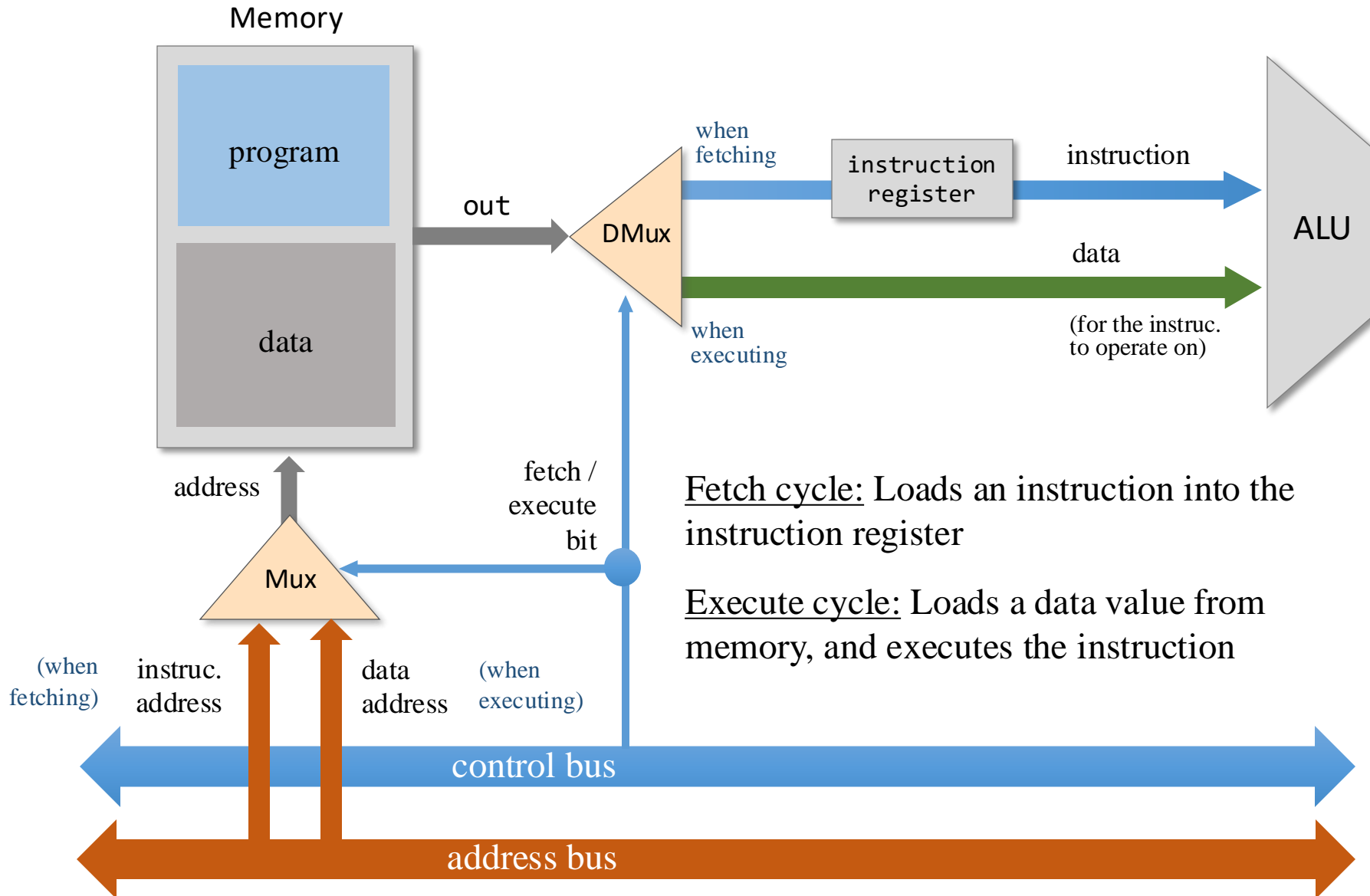
Two-cycle, one-memory machine



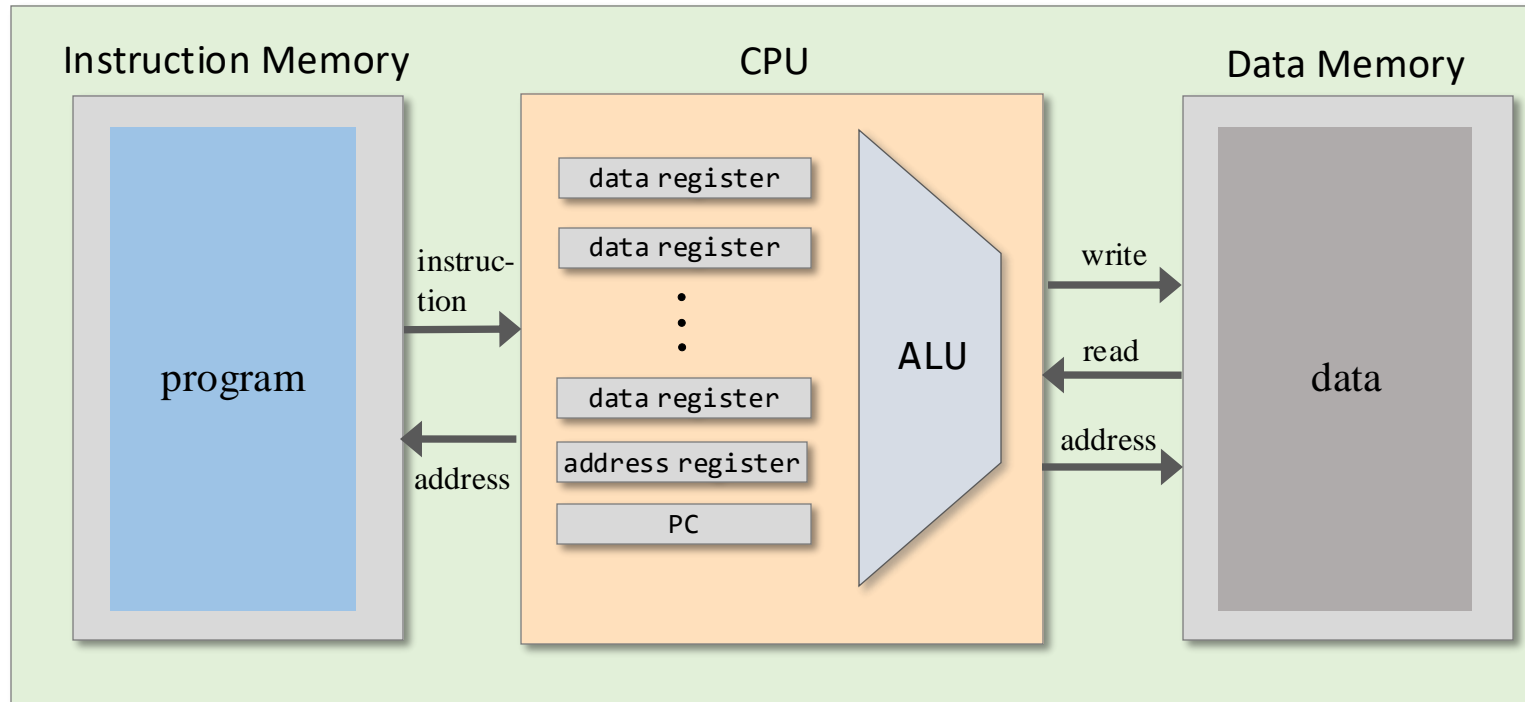
Two-cycle, one-memory machine



Two-cycle, one-memory machine

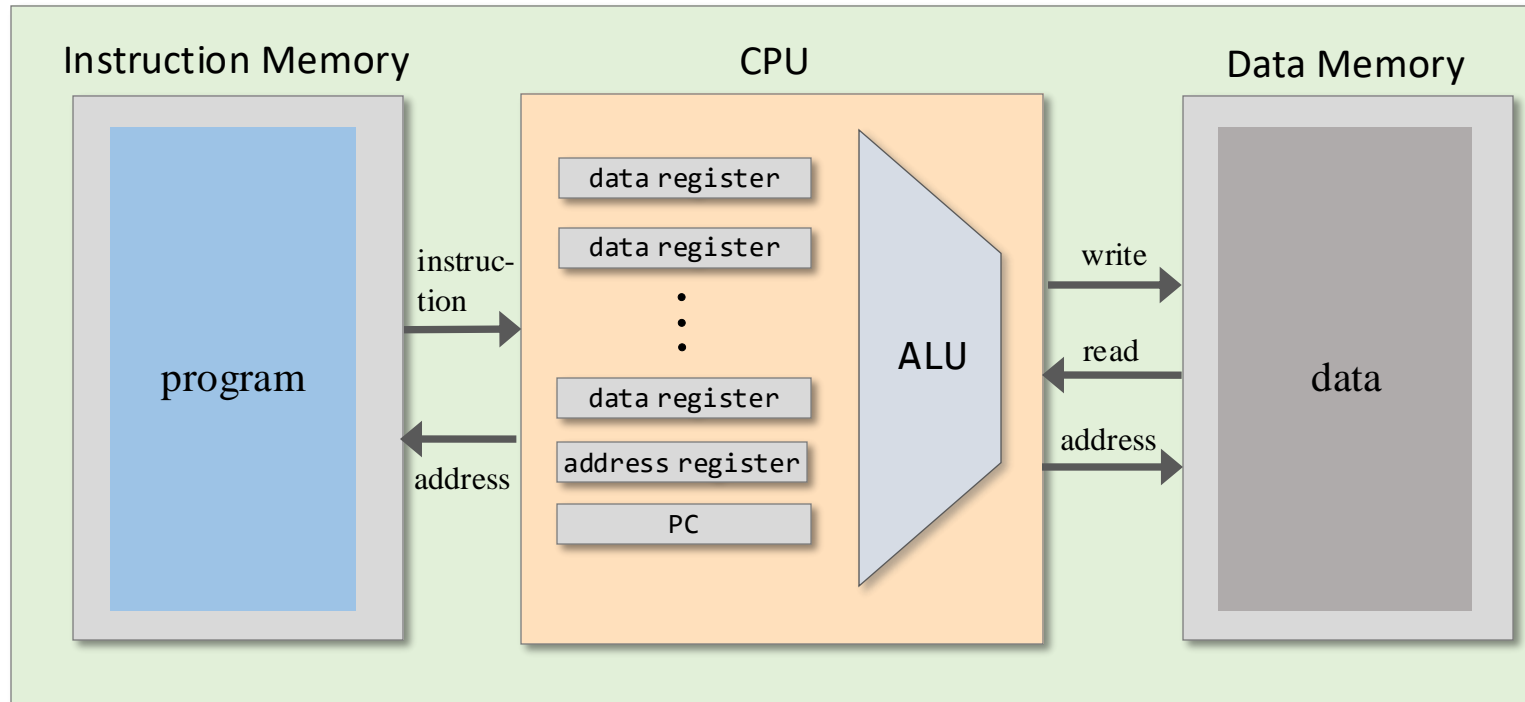


Single cycle, two-memory machine



- Instructions and data are stored in two separate physical memories
 - Both memories are accessed simultaneously, in the same cycle
- (for historical reasons, referred to as "Harvard architecture")

Single cycle, two-memory machine



Advantages

- Simple architecture
- Fast processing

Disadvantages

- Two memory chips
- Separate address spaces

Chapter 5: Computer Architecture

- ✓ Basic architecture
- ✓ Fetch-Execute cycle

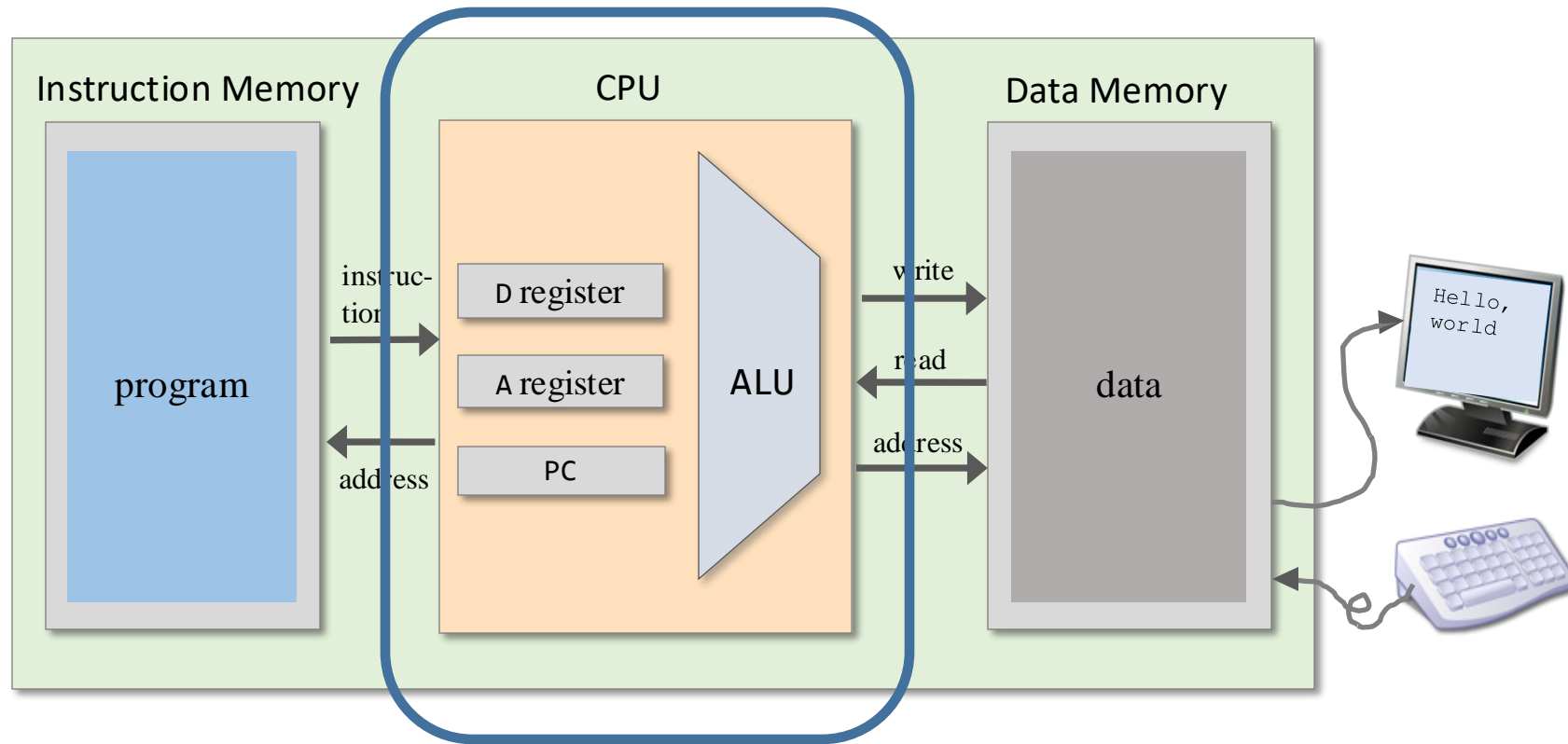


The Hack CPU

- Input / output

- Memory
- Computer
- Project 5: Chips
- Project 5: Guidelines

The Hack computer



CPU abstraction

A chip that implements the Hack instruction set:

A instruction Symbolic: *@xxx* (*xxx* is a decimal value ranging from 0 to 32767, or a symbol bound to such a decimal value)

Binary: *0 vvvvvvvvvvvvvvvv* (*vv ... v* = 15-bit value of *xxx*)

C instruction Symbolic: *dest = comp; jump* (*comp* is mandatory.
If *dest* is empty, the = is omitted;
If *jump* is empty, the ; is omitted)

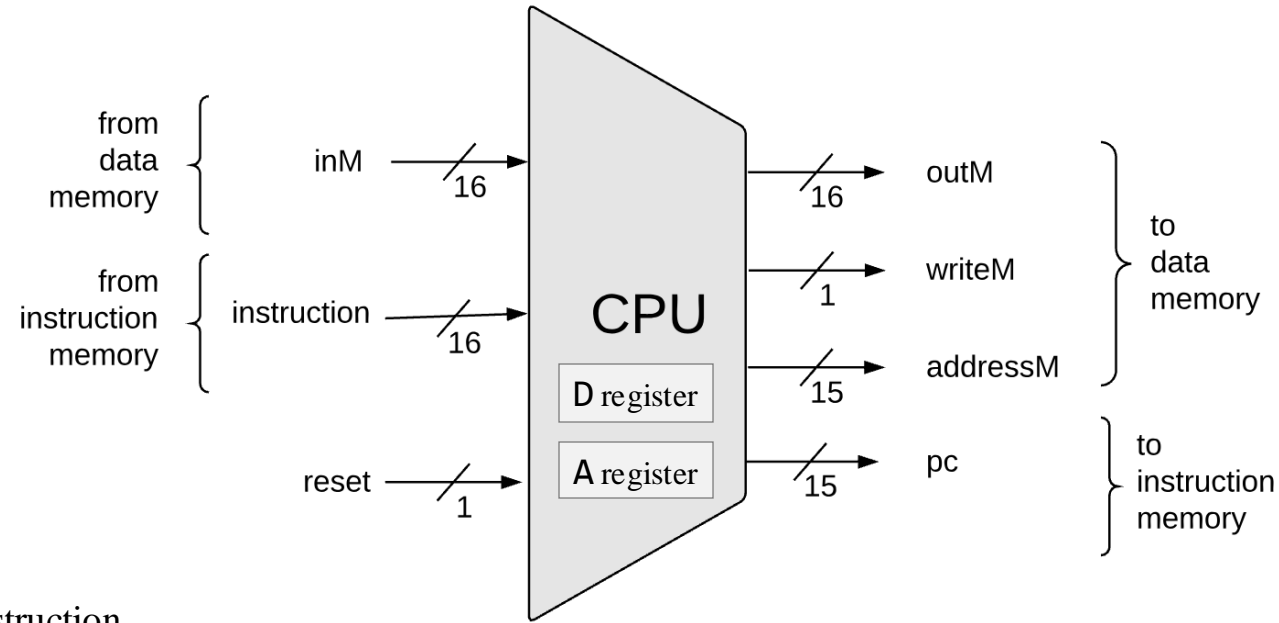
Binary: *111a ccccccc d d j j j*

<i>comp</i>		<i>c</i>	<i>c</i>	<i>c</i>	<i>c</i>	<i>c</i>	<i>c</i>	<i>dest</i>	<i>d</i>	<i>d</i>	<i>d</i>	Effect: store <i>comp</i> in:
0		1	0	1	0	1	0	null	0	0	0	the value is not stored
1		1	1	1	1	1	1	M	0	0	1	RAM[A]
-1		1	1	1	0	1	0	D	0	1	0	D register (reg)
D		0	0	1	1	0	0	DM	0	1	1	RAM[A] and D reg
A	M	1	1	0	0	0	0	A	1	0	0	A reg
!D		0	0	1	1	0	1	AM	1	0	1	A reg and RAM[A]
!A	!M	1	1	0	0	0	1	AD	1	1	0	A reg and D reg
-D		0	0	1	1	1	1	ADM	1	1	1	A reg, D reg, and RAM[A]
-A	-M	1	1	0	0	1	1					
D+1		0	1	1	1	1	1					
A+1	M+1	1	1	0	1	1	1					
D-1		0	0	1	1	1	0					
A-1	M-1	1	1	0	0	1	0					
D+A	D+M	0	0	0	0	1	0					
D-A	D-M	0	1	0	0	1	1					
A-D	M-D	0	0	0	1	1	1					
D&A	D&M	0	0	0	0	0	0					
D A	D M	0	1	0	1	0	1					

a == 0 *a* == 1

<i>jump</i>	<i>j</i>	<i>j</i>	<i>j</i>	Effect:
null	0	0	0	no jump
JGT	0	0	1	if <i>comp</i> > 0 jump
JEQ	0	1	0	if <i>comp</i> = 0 jump
JGE	0	1	1	if <i>comp</i> ≥ 0 jump
JLT	1	0	0	if <i>comp</i> < 0 jump
JNE	1	0	1	if <i>comp</i> ≠ 0 jump
JLE	1	1	0	if <i>comp</i> ≤ 0 jump
JMP	1	1	1	unconditional jump

CPU abstraction

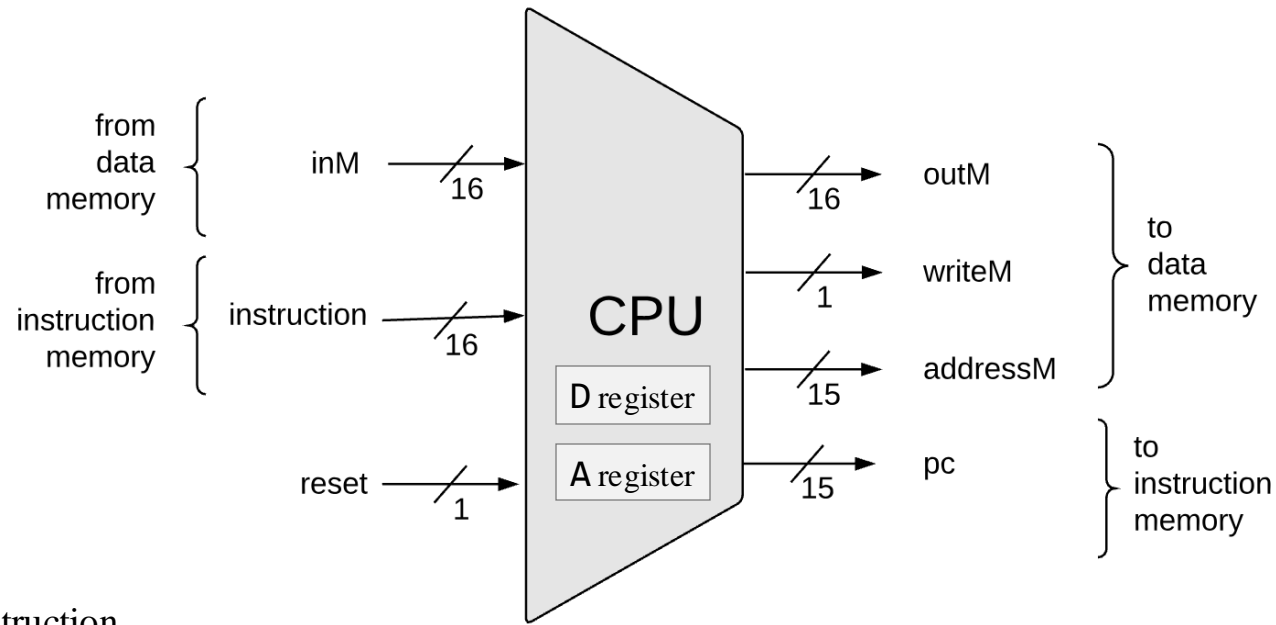


Instruction examples:

```
// D = RAM[5] + 1
@5
DM=M+1
...
// goto 200
@200
0;JMP
...
```

1. Executes the current instruction
2. Figures out which instruction to execute next

CPU abstraction



Instruction examples:

```
// D = RAM[5] + 1
@5
DM=M+1
...
// goto 200
@200
0; JMP
...
```

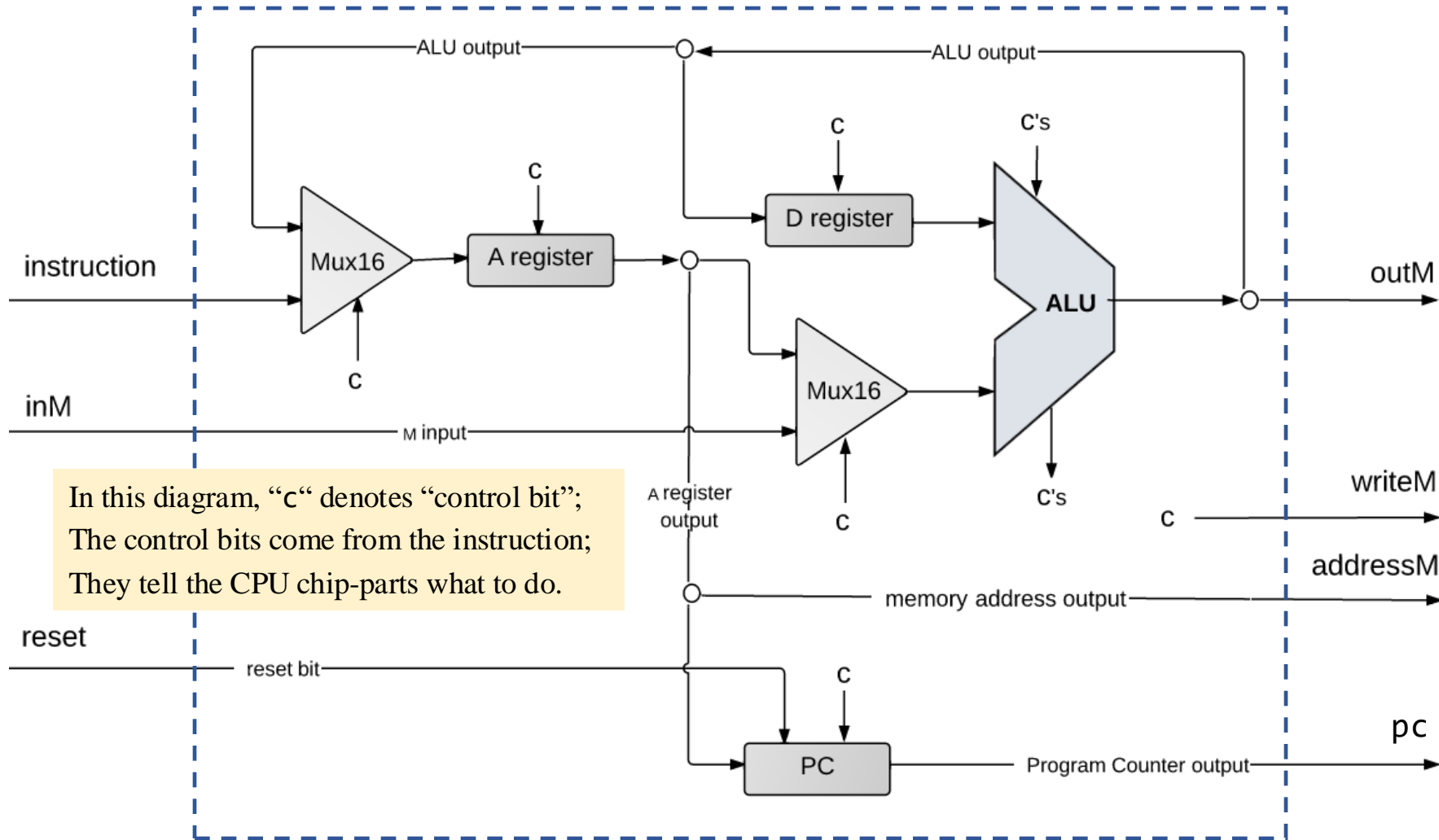
1. Executes the current instruction:

If it's an A-instruction (@*xxx*), sets the A register to *xxx*

Else (C-instruction):

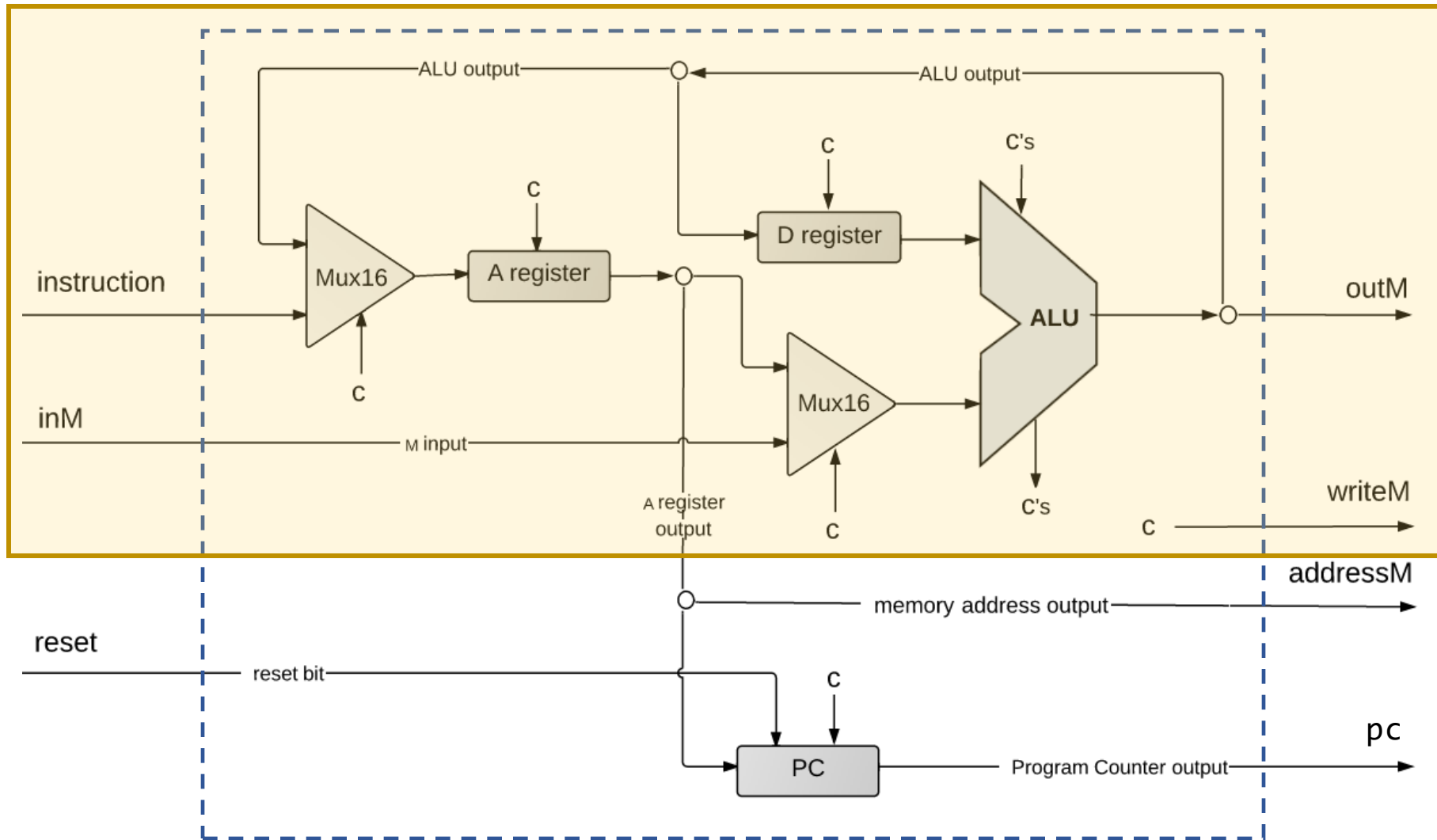
- Computes the ALU function specified by the instruction (on the values of A, D, inM)
- Puts the ALU output in A, D, outM, as specified by the instruction
- If the instruction writes to M, sets addressM to A and asserts writeM

CPU implementation



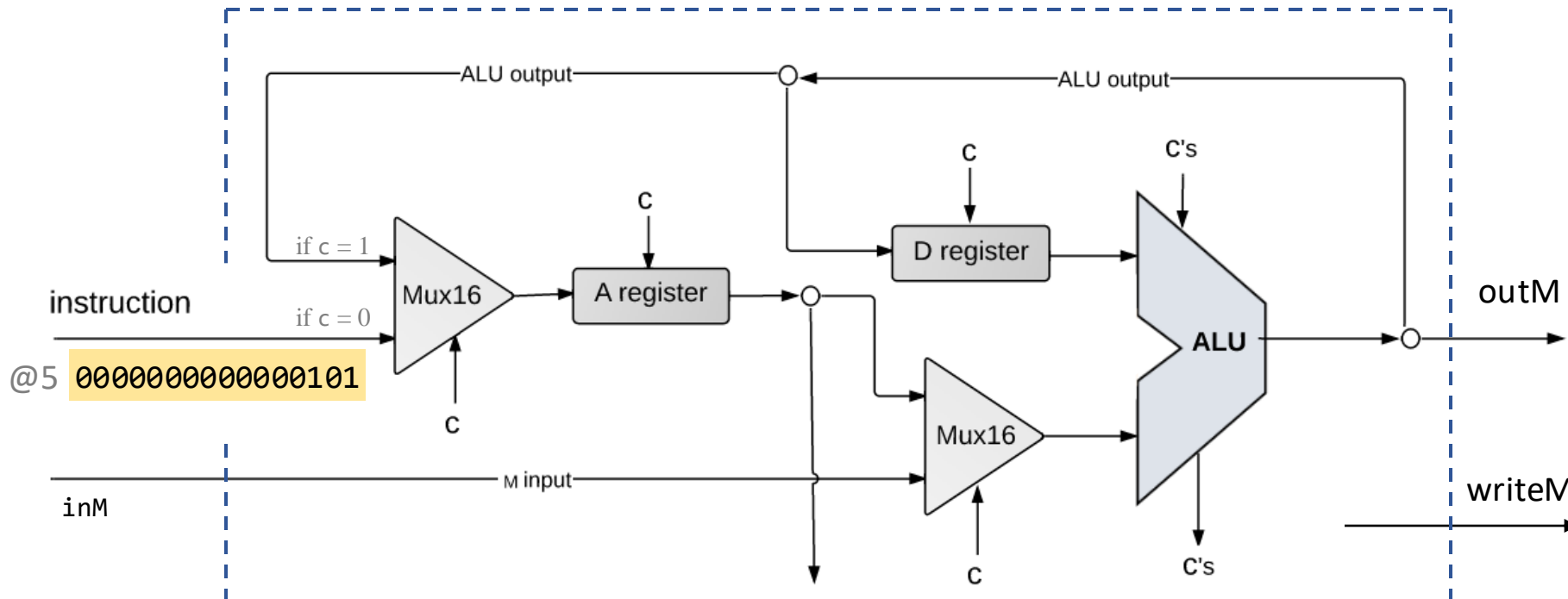
Hack CPU architecture

CPU implementation: Instruction handling



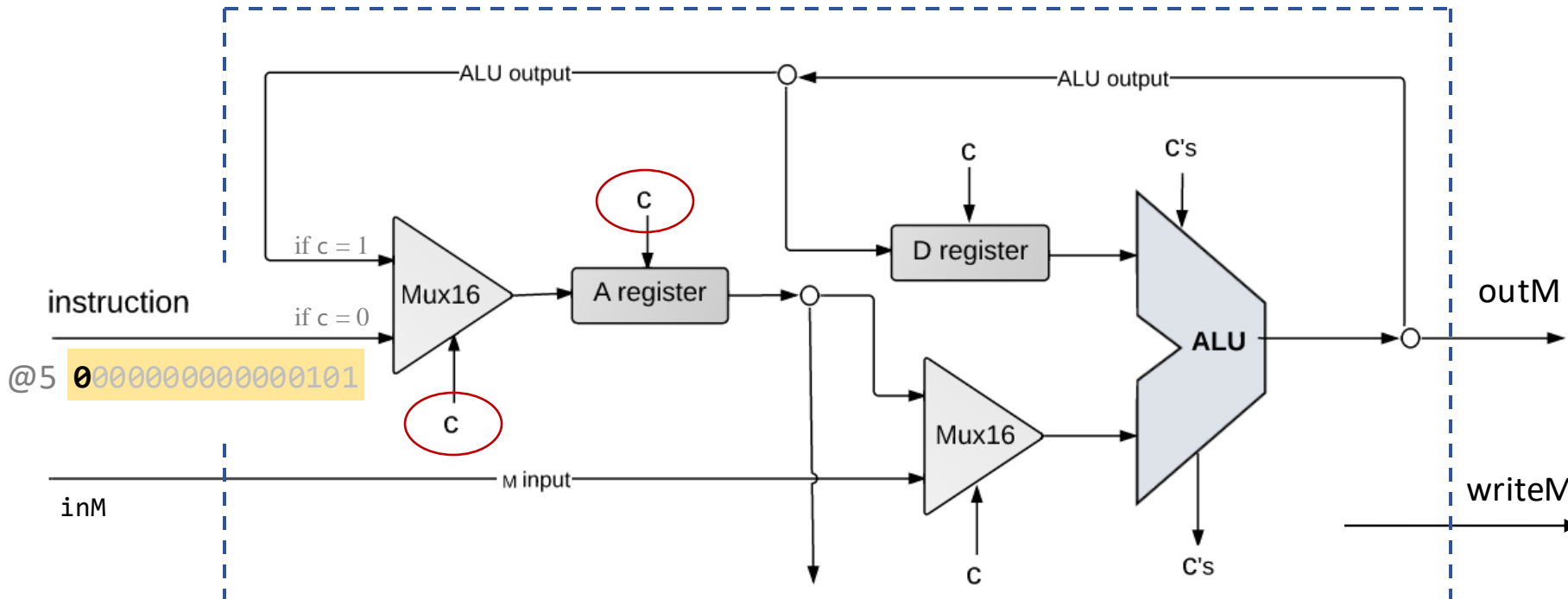
Hack CPU architecture

CPU implementation: Instruction handling



Handling A-instructions

CPU implementation: Instruction handling



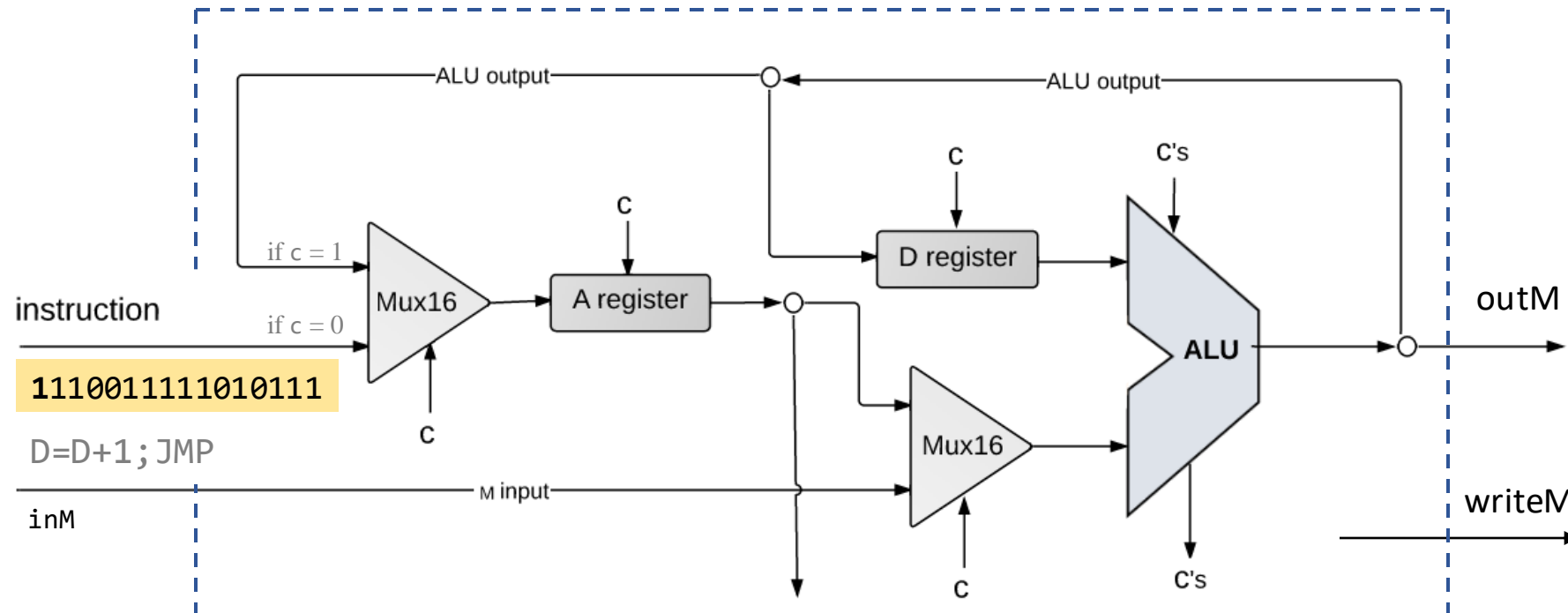
Handling A-instructions

Use the instruction's MSB (op-code) to manipulate the control bits of the A register and the Mux16 before it.

Effect: A-register \leftarrow instruction (treated as a value)

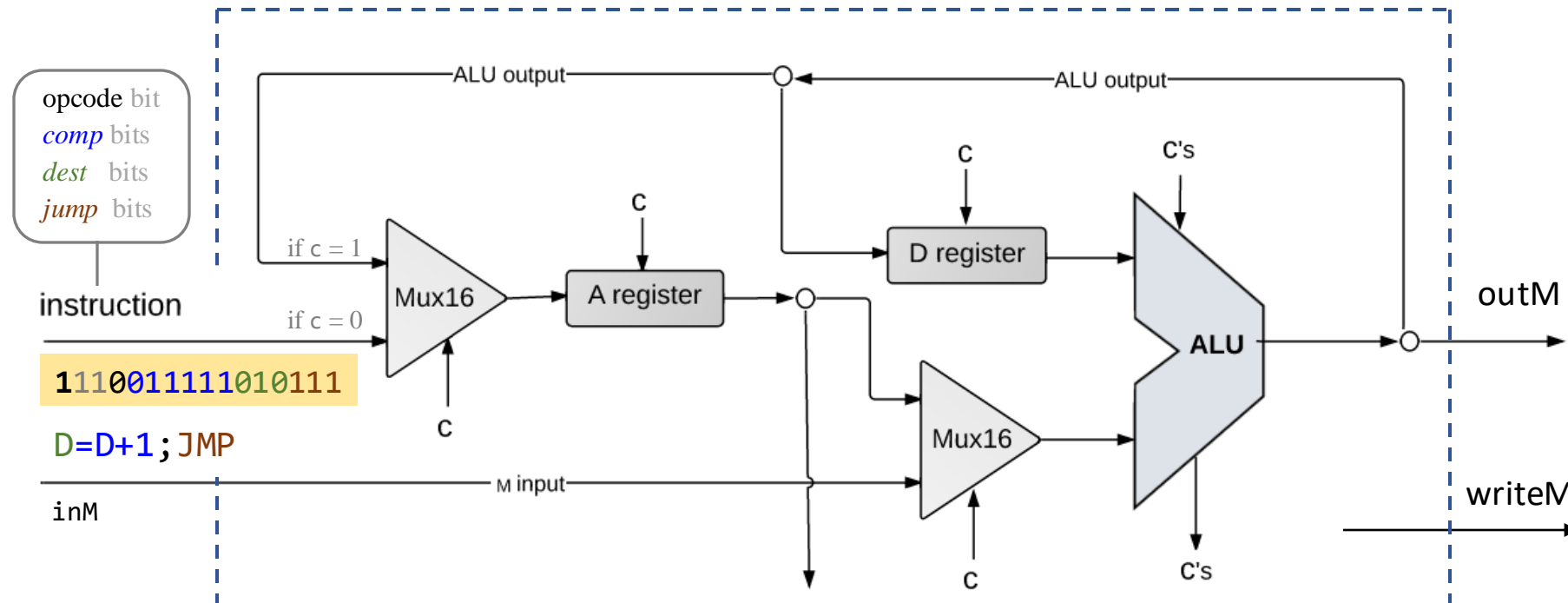
(Exactly what the @xxx instruction specifies: “set A to xxx”)

CPU implementation: Instruction handling



Handling C-instructions

CPU implementation: Instruction handling



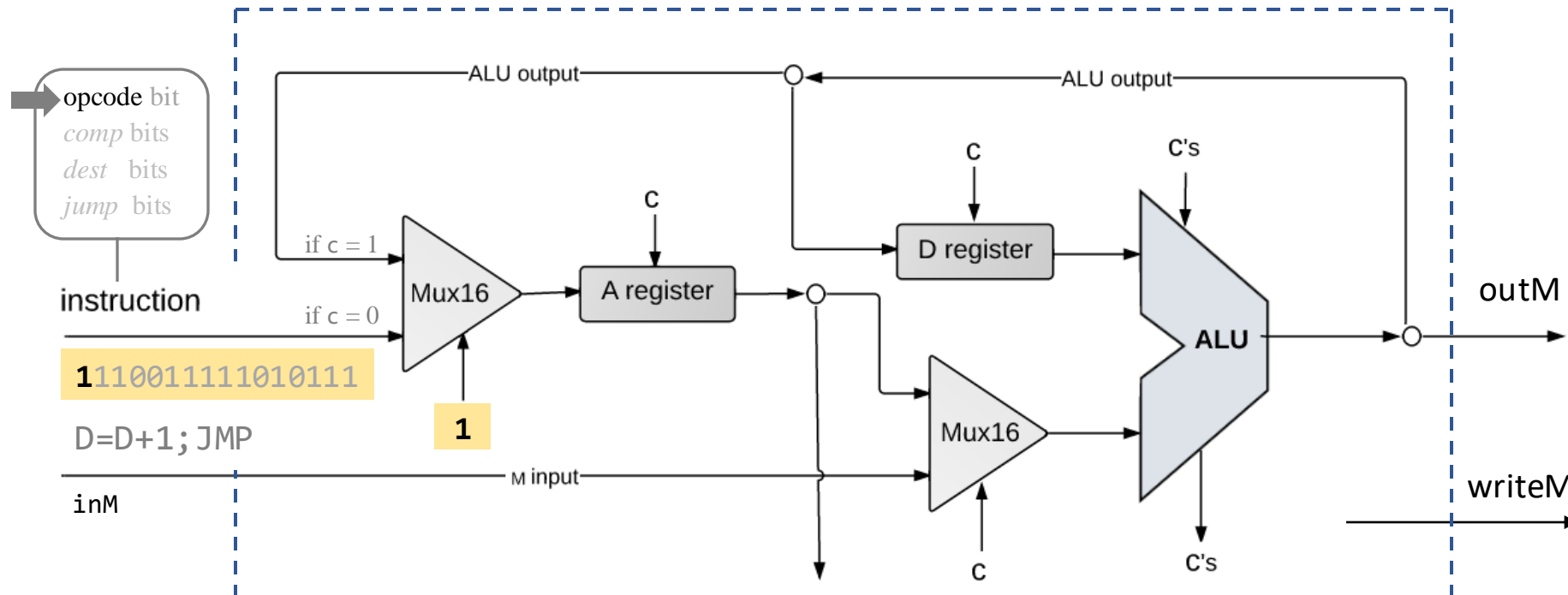
Handling C-instructions

Each instruction field (*opcode*, *comp*, *dest*, and *jump* bits) is handled separately

Each group of bits is used to "tell" a CPU chip-part what to do

Taken together, the chip-parts end up executing the instruction.

CPU implementation: Instruction handling

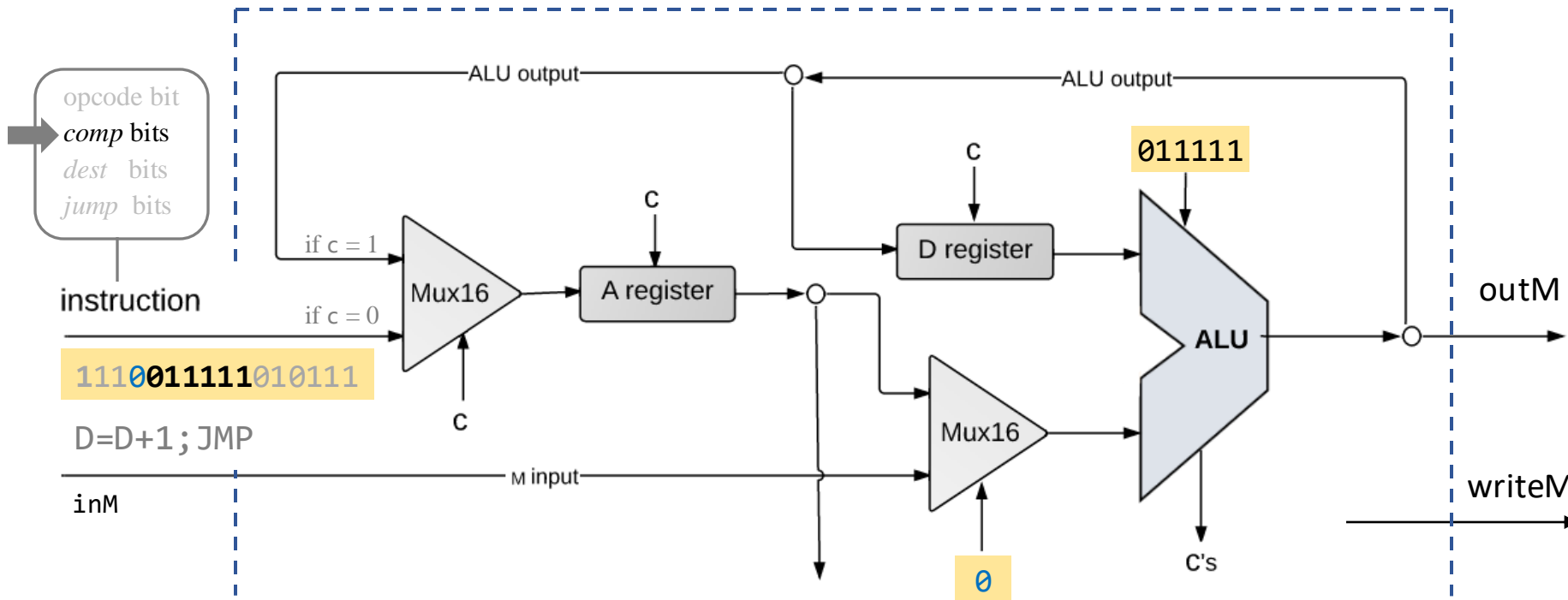


Handling C-instructions: The *opcode* bit

Routes the instruction's MSB to the Mux16

Effect: Primes the A register to get the ALU output.

CPU implementation: Instruction handling

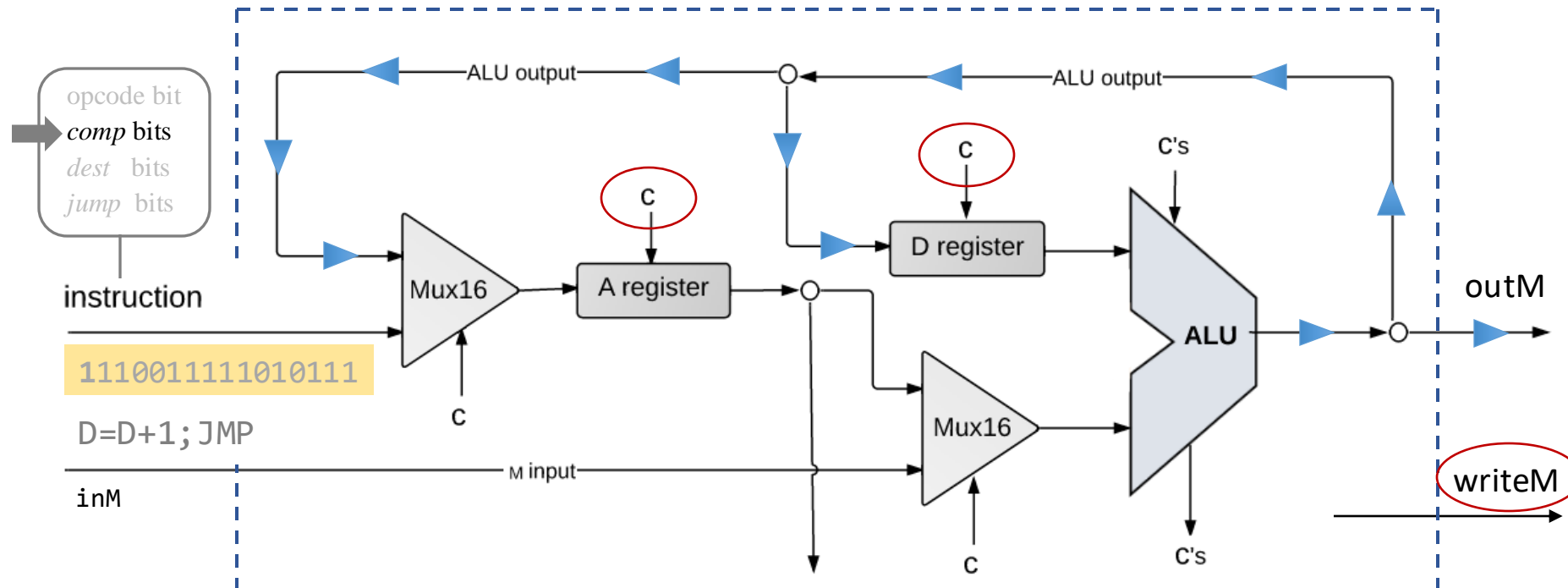


Handling C-instructions: The *computation* bits

- Routes the instruction's *c*-bits to the ALU control bits
- Routes the instruction's *a*-bit to the Mux16

Effect: the ALU computes the specified function
and emits the resulting value to the ALU output

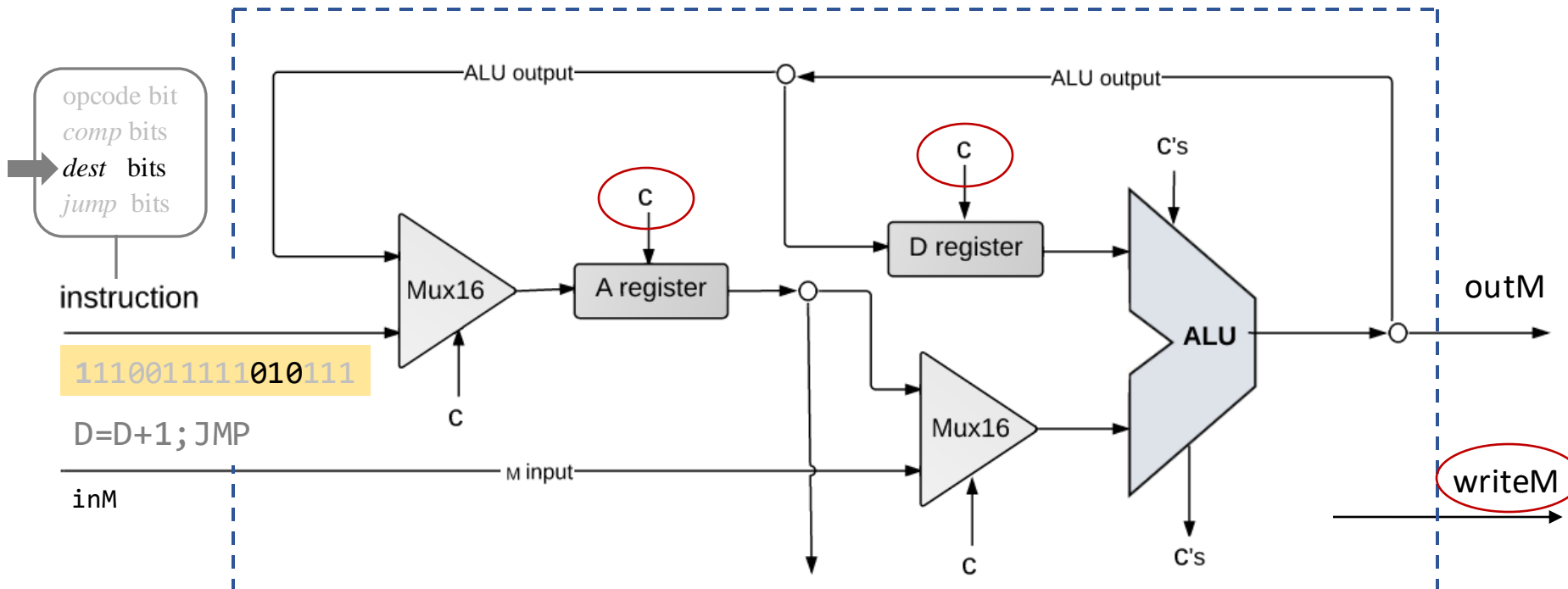
CPU implementation: Instruction handling



ALU output:

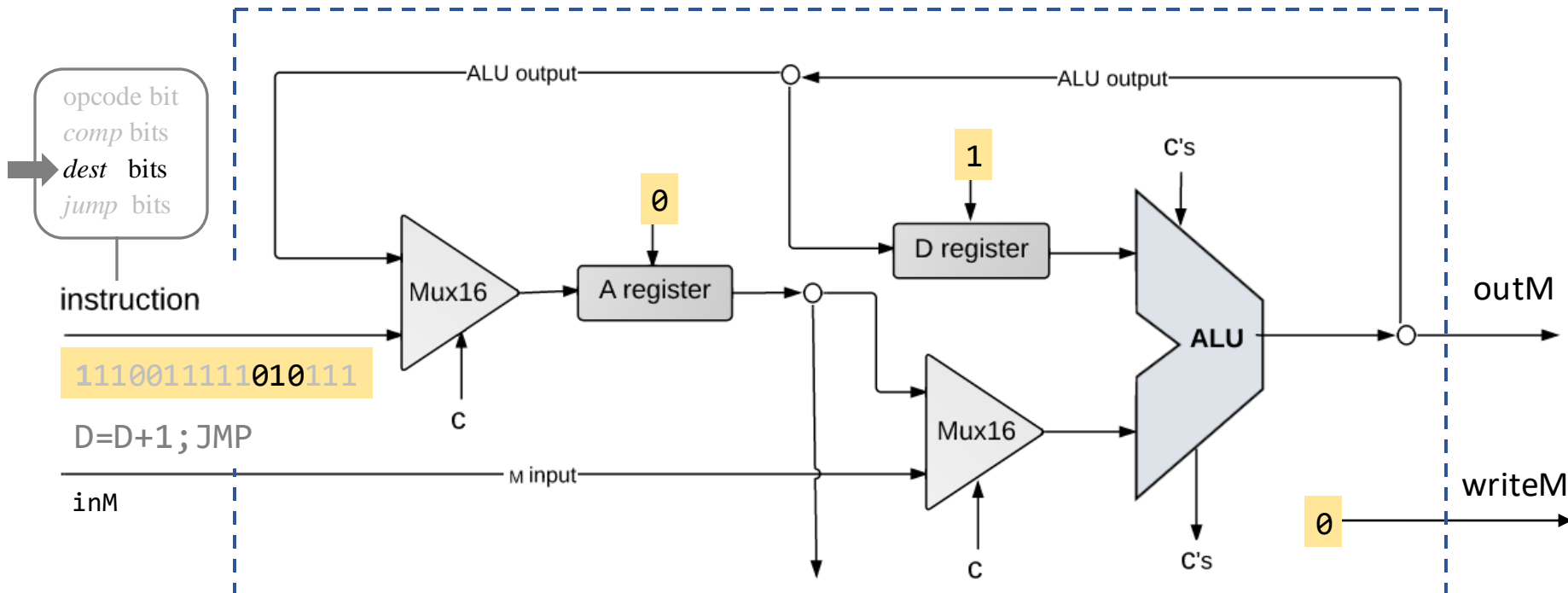
- Result of ALU calculation
- Fed simultaneously to D-register, A-register, data memory
- Each enabled/disabled by its control bit

CPU implementation: Instruction handling



Handling C-instructions: The *destination* bits

CPU implementation: Instruction handling

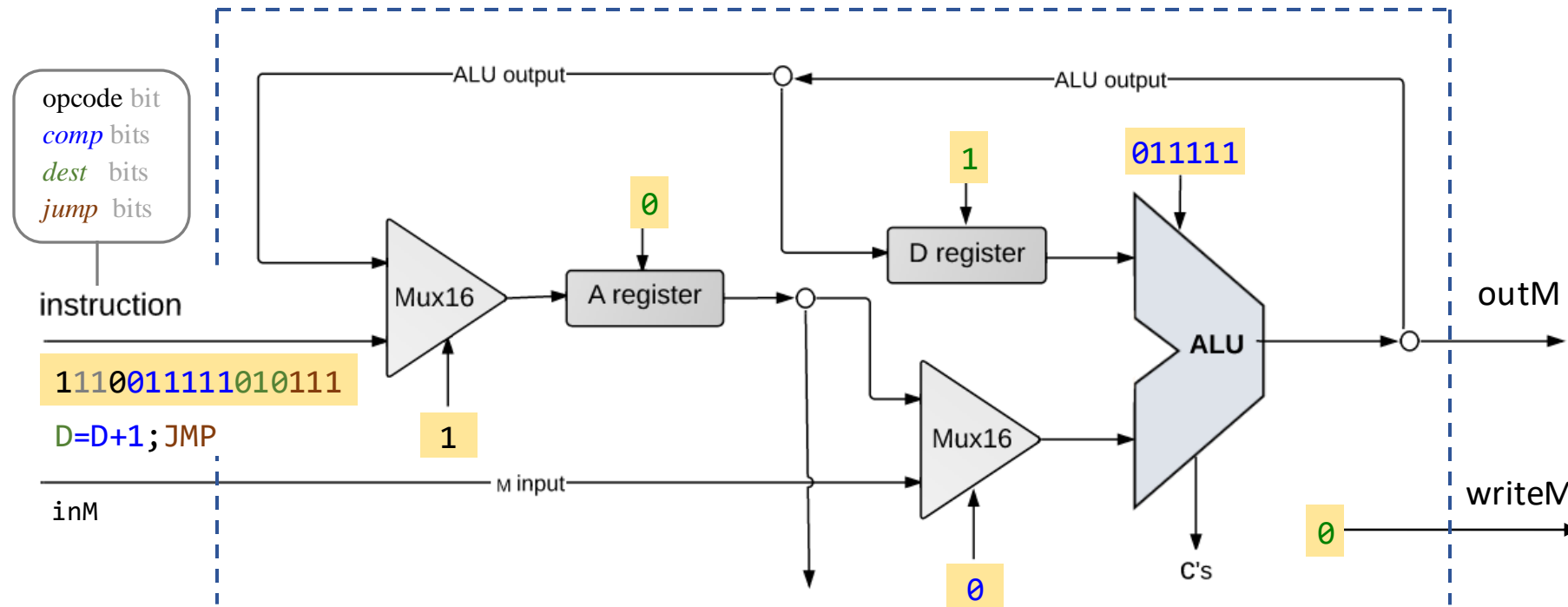


Handling C-instructions: The *destination* bits

Routes the instruction's d-bits to the control (load) bits of the A-register, D-register, and to the `writeM` bit

Effect: Only the enabled destinations commit to the ALU output

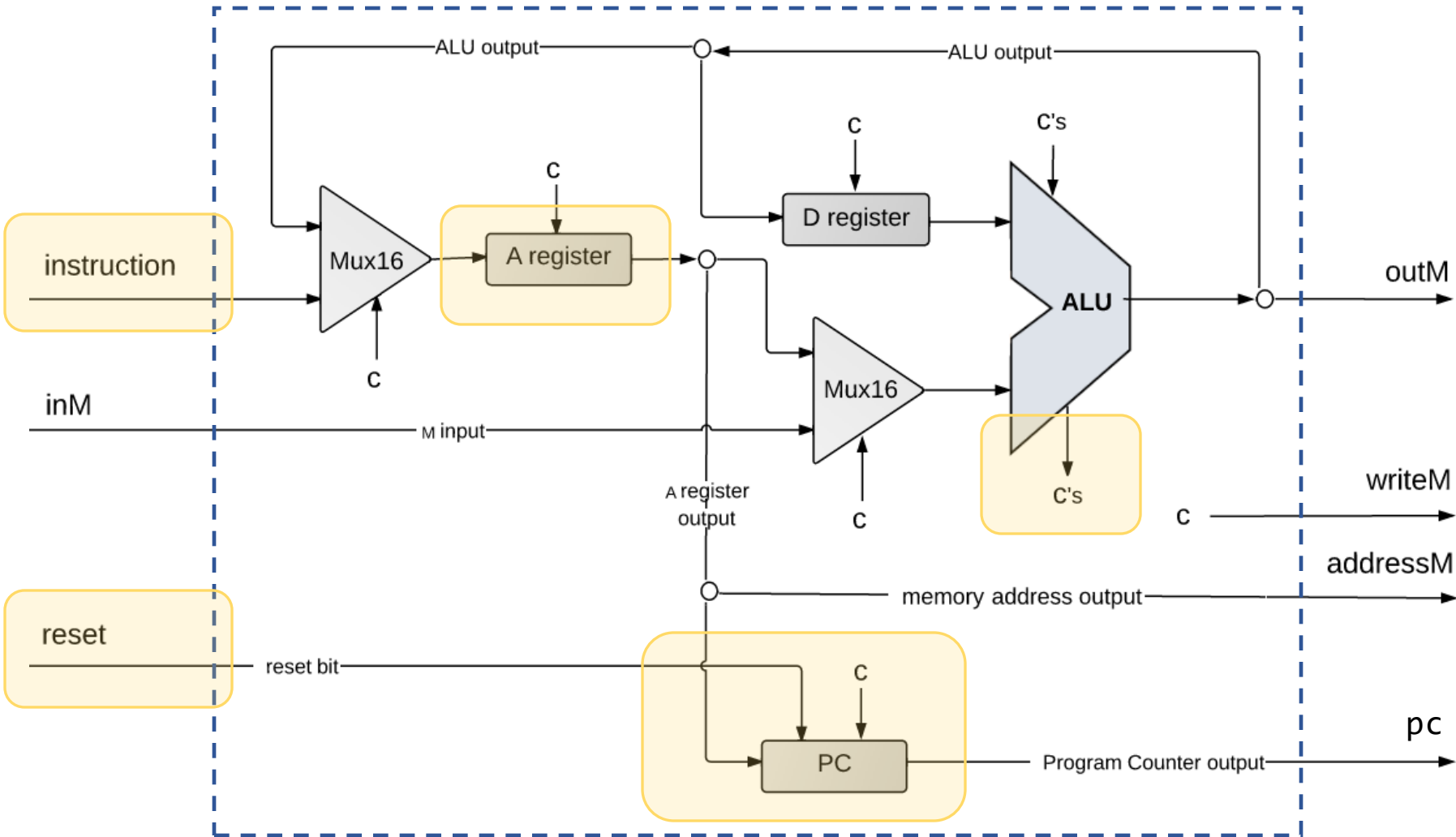
CPU implementation: Instruction handling



Handling C-instructions: Recap

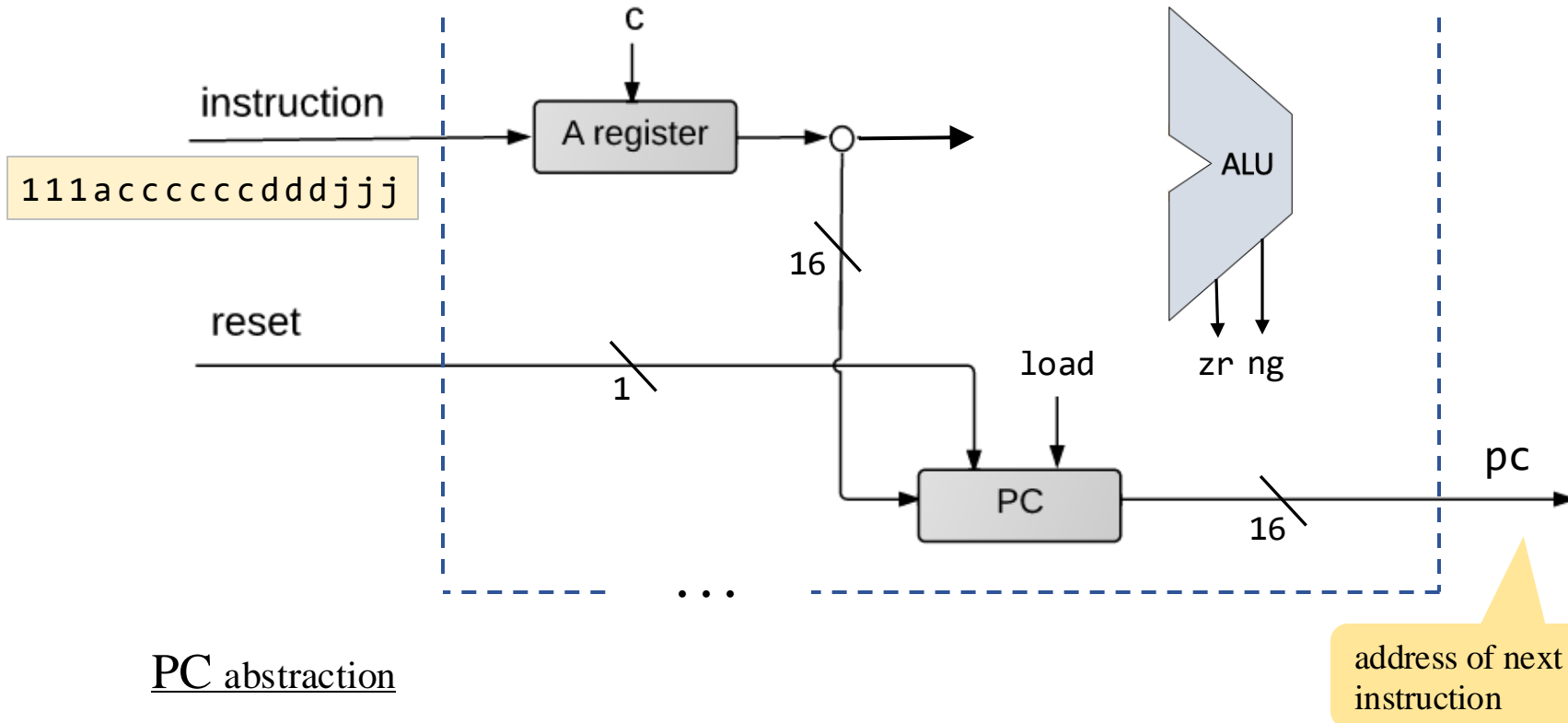
- ✓ Executes $dest = comp$
- ➔ Figures out which instruction to execute next

CPU implementation: Control



Hack CPU architecture

CPU implementation: Control



PC abstraction

Outputs the address of the next instruction, has three states:

reset: $PC \leftarrow 0$

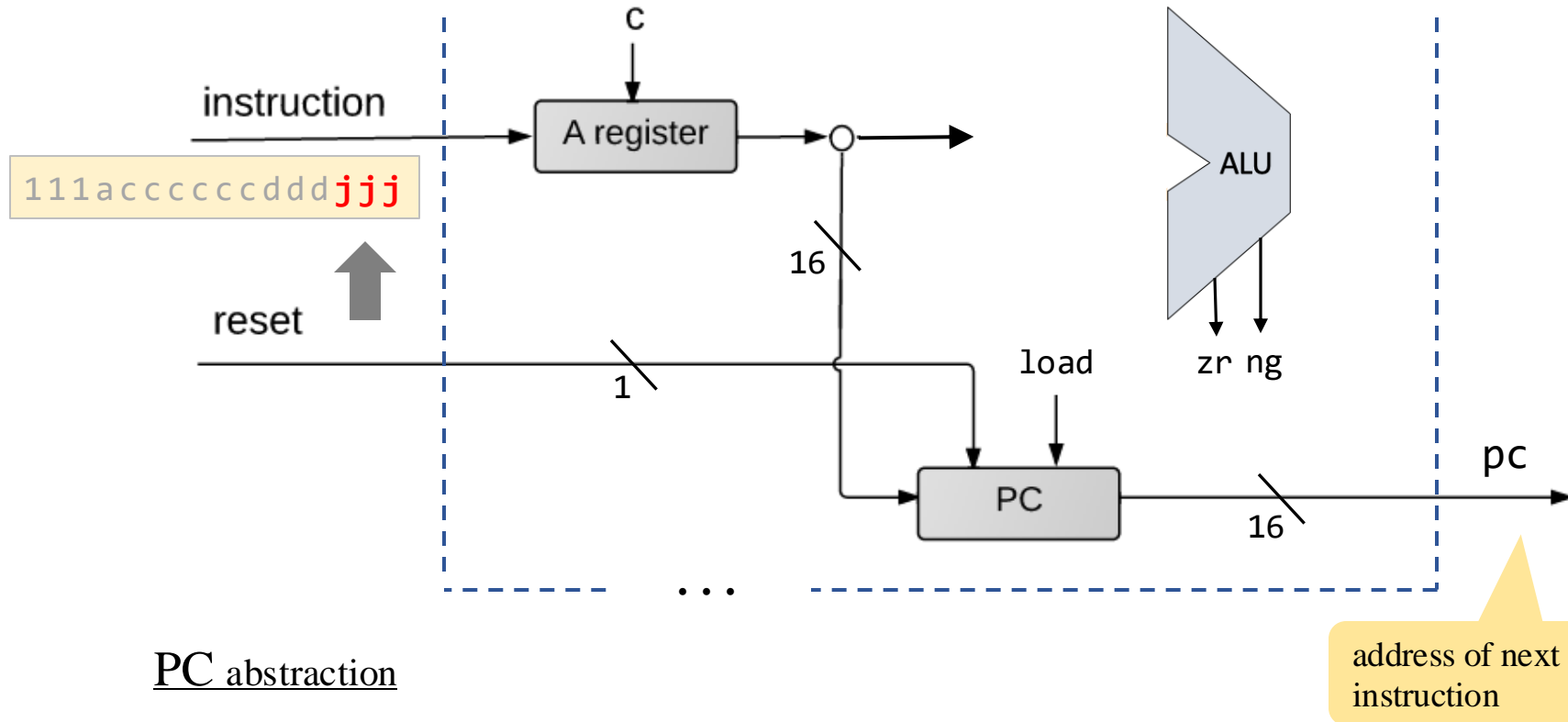
no jump: PC++

```

jump:    if (condition) PC ← A    // Note: A was already set to the jump address

```

CPU implementation: Control



PC abstraction

Outputs the address of the next instruction, has three states:

reset: $PC \leftarrow 0$

no jump: $PC++$

jump: if (*condition*) $PC \leftarrow A$ // Note: A was already set to the jump address

CPU implementation: Control

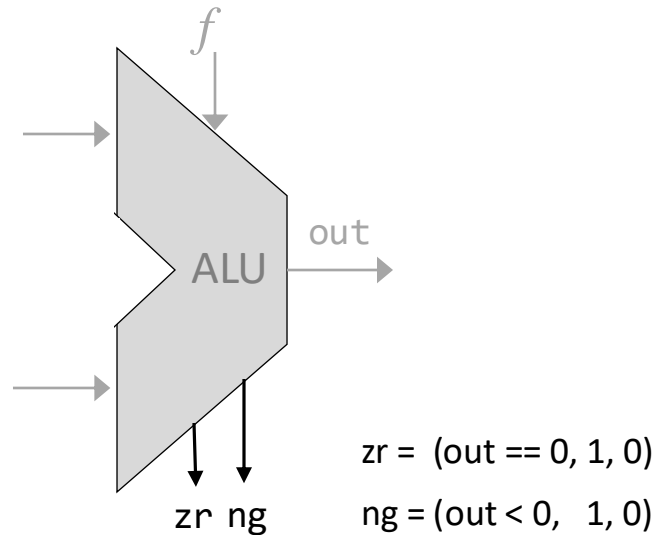
Symbolic
syntax:

dest = comp ; jump

Binary
syntax:

1 1 1 a c c c c c c d d d j1 j2 j3

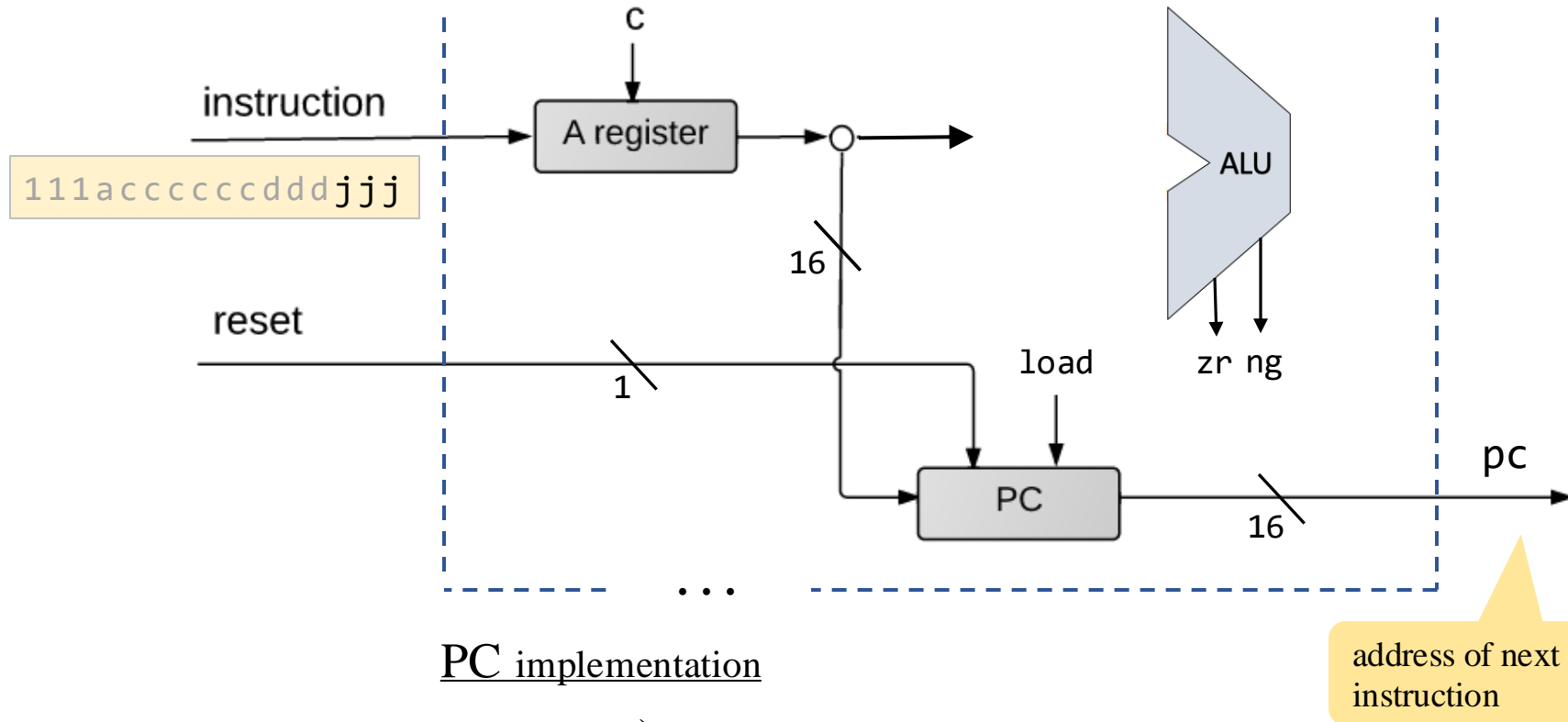
<i>jump</i>	j1	j2	j3	<i>condition</i>
null	0	0	0	no jump
JGT	0	0	1	if (ALU out > 0) jump
JEQ	0	1	0	if (ALU out = 0) jump
JGE	0	1	1	if (ALU out ≥ 0) jump
JLT	1	0	0	if (ALU out < 0) jump
JNE	1	0	1	if (ALU out ≠ 0) jump
JLE	1	1	0	if (ALU out ≤ 0) jump
JMP	1	1	1	Unconditional jump



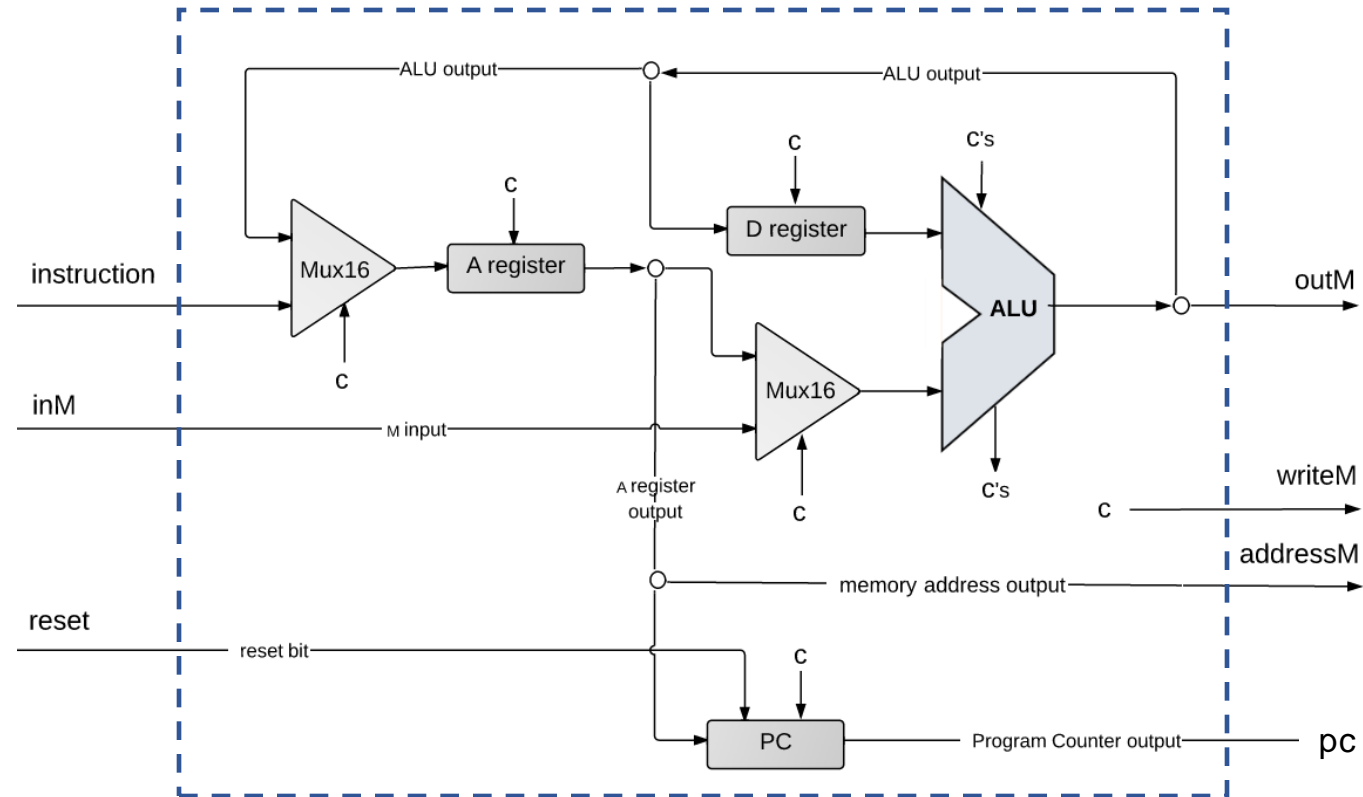
We can use gate logic to compute:

$J(j1, j2, j3, zr, ng) = 1$ if *condition* is true
0 otherwise

CPU implementation: Control



CPU implementation



Executes the current instruction



Figures out which instruction to execute next.

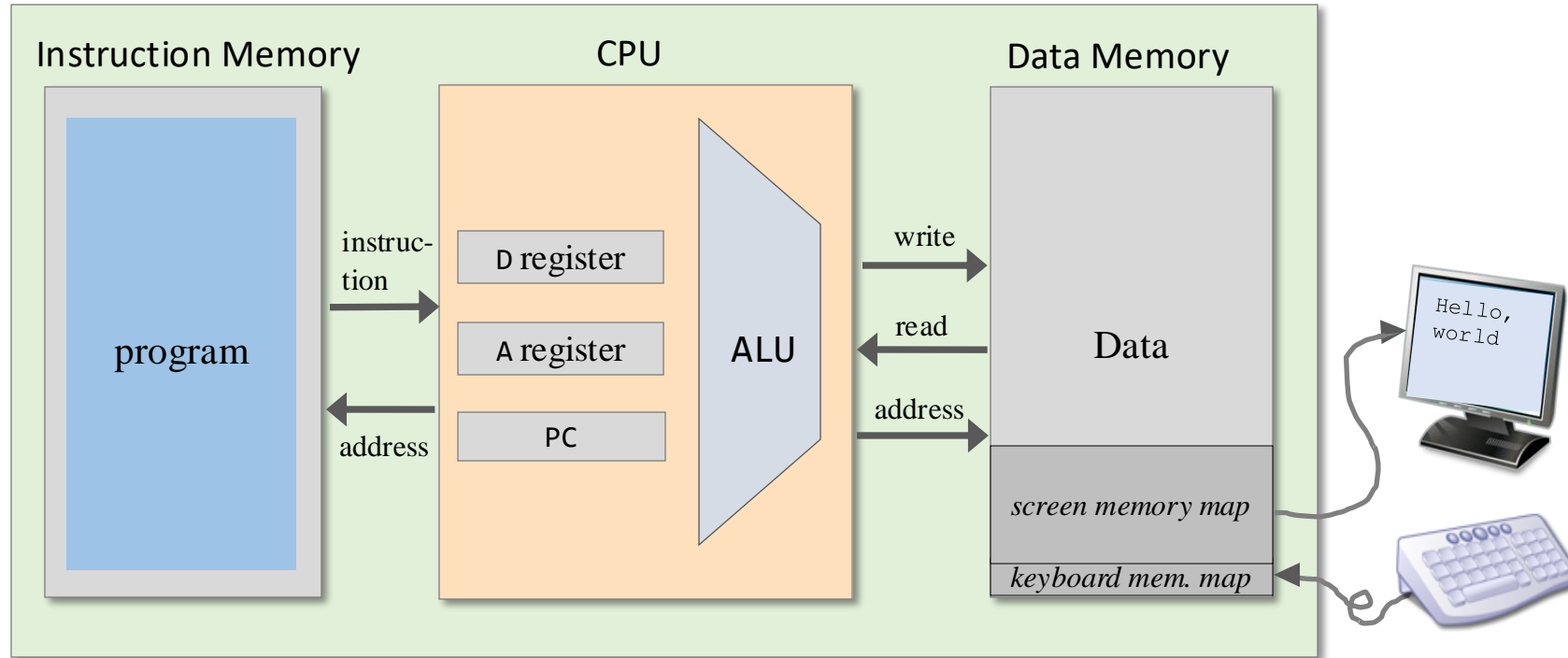
Computer Architecture

- ✓ Basic architecture
- ✓ Fetch-Execute cycle
- ✓ The Hack CPU

➡ Input / output

- Memory
- Computer
- Project 5: Chips
- Project 5: Guidelines

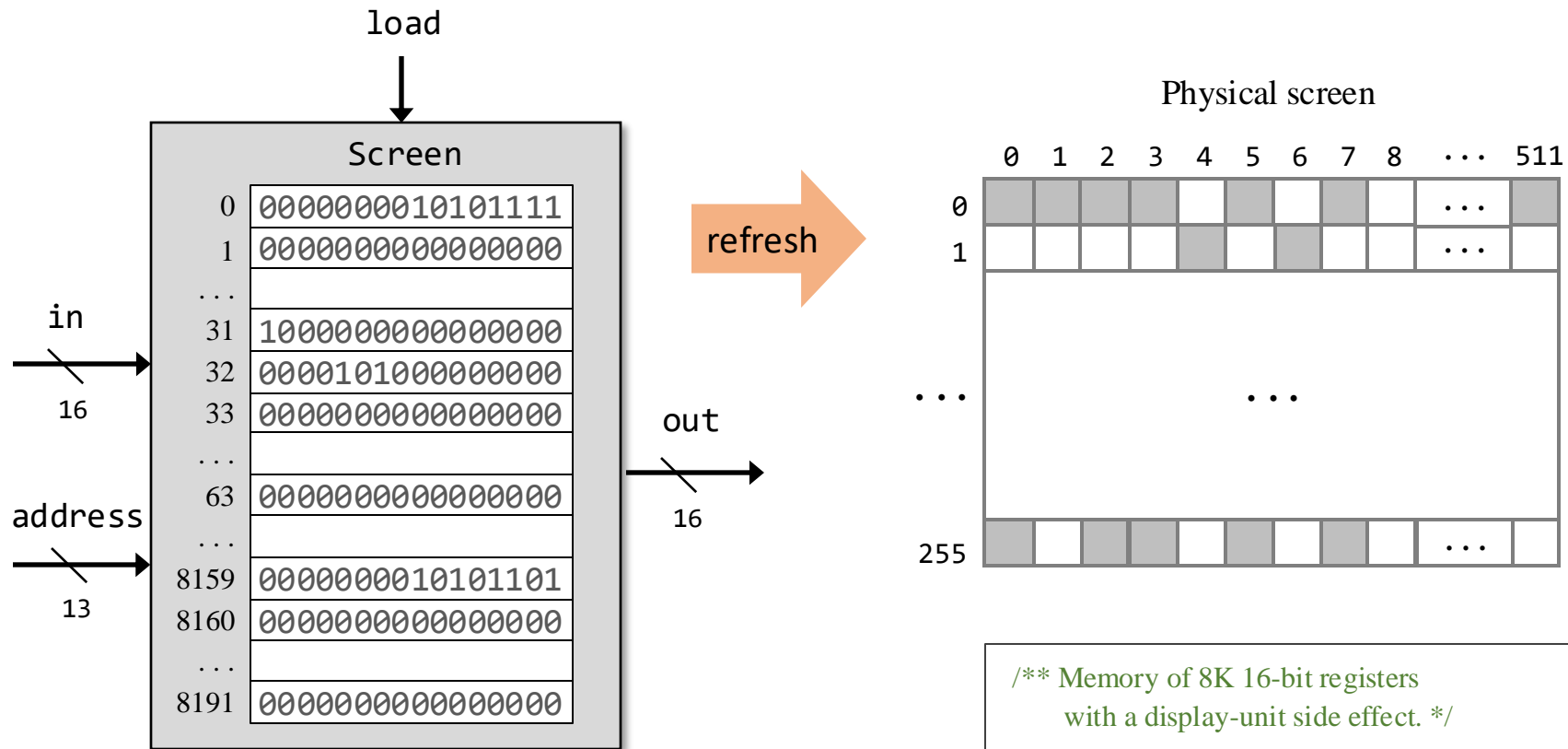
Hack computer



I/O devices

- Screen
- Keyboard

Screen

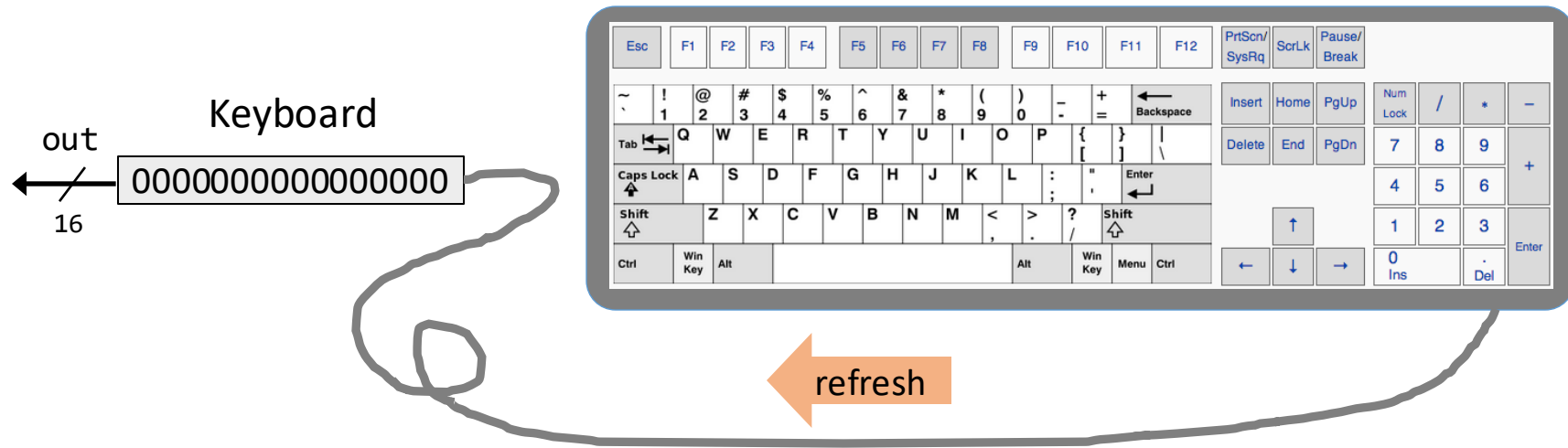


Implemented as a built-in 8K
memory chip named Screen

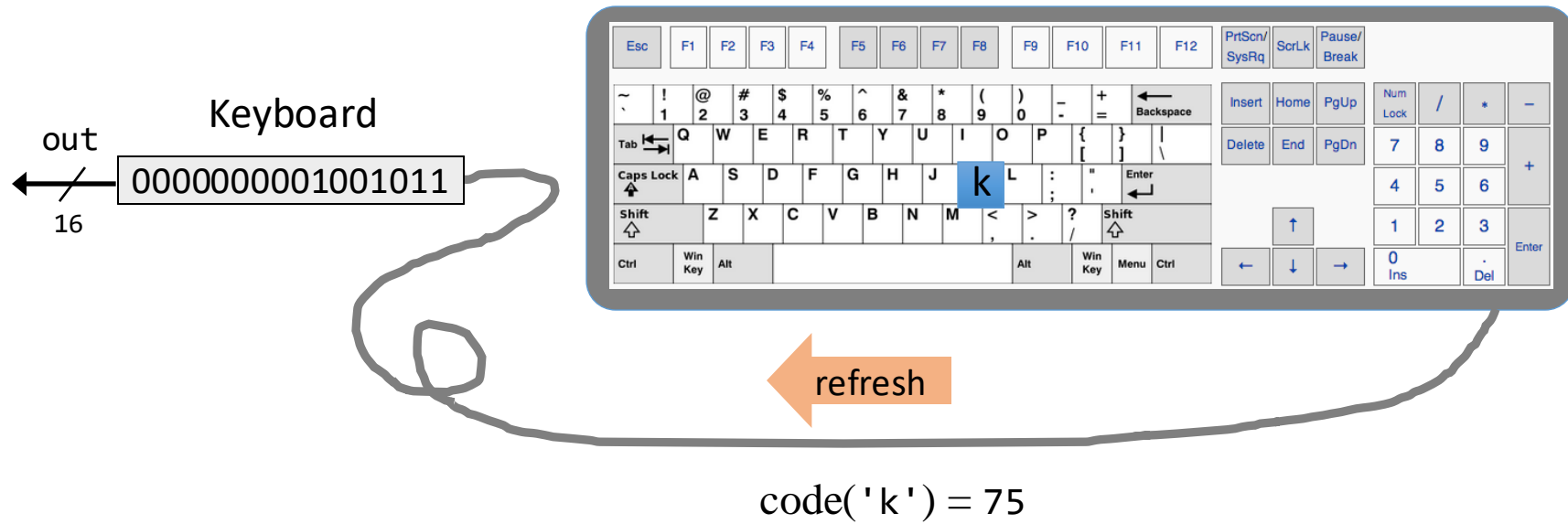
```
/** Memory of 8K 16-bit registers  
    with a display-unit side effect. */
```

```
CHIP Screen {  
    IN  address[13], in[16], load;  
    OUT out[16];  
    BUILTIN Screen;  
    CLOCKED in, load;  
}
```

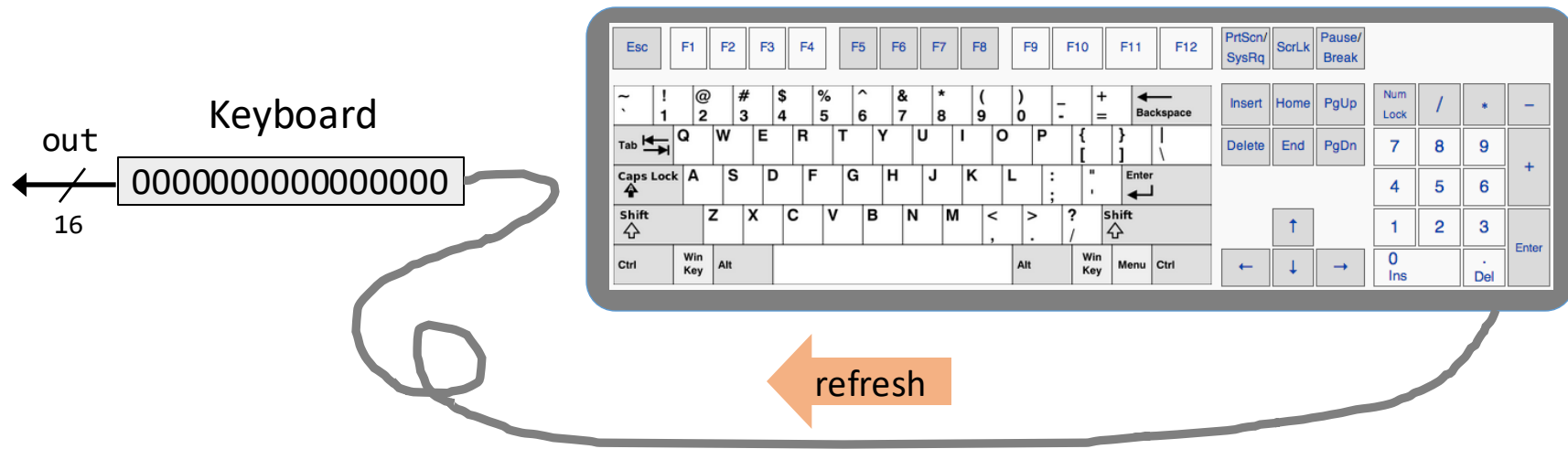
Keyboard



Keyboard



Keyboard



Implemented as a built-in
16-bit memory register
named Keyboard

```
/* 16-bit register, outputs the character code of the currently  
   pressed keyboard key, or 0 if no key is pressed */  
CHIP Keyboard {  
    OUT  
    out[16];  
    BUILTIN Keyboard;  
}
```

Chapter 5: Computer Architecture

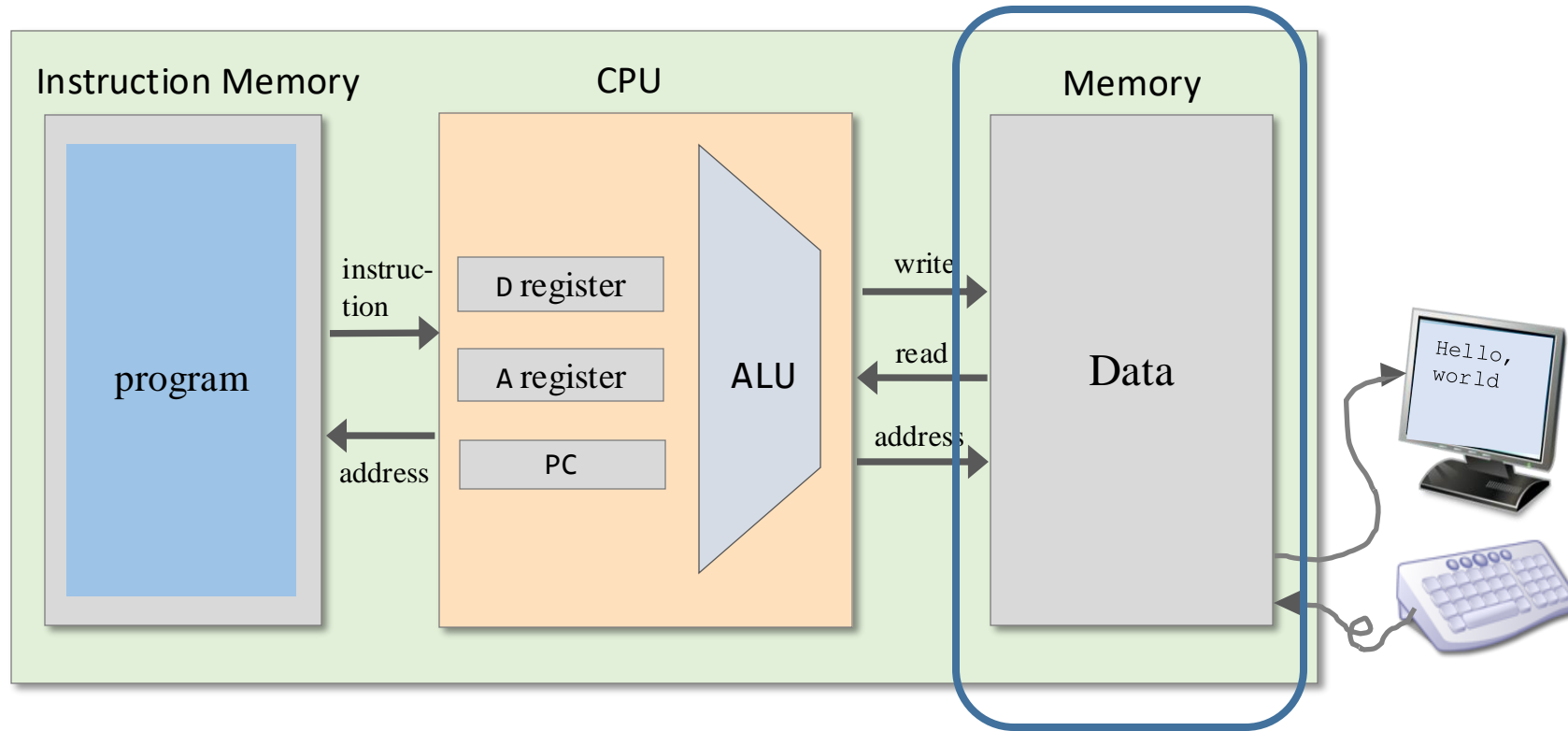
- ✓ Basic architecture
- ✓ Fetch-Execute cycle
- ✓ The Hack CPU
- ✓ Input / output



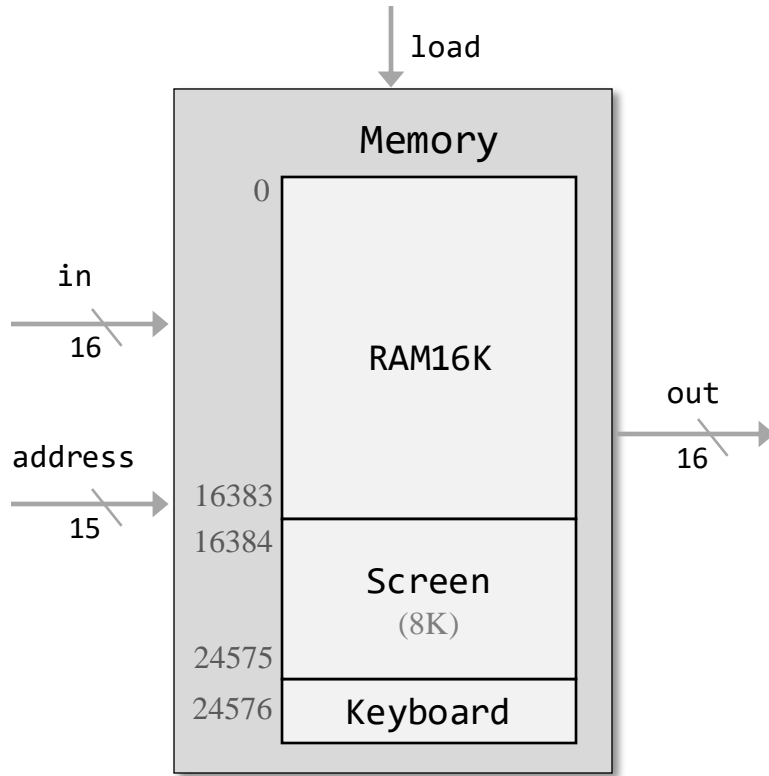
Memory

- Computer
- Project 5: Chips
- Project 5: Guidelines

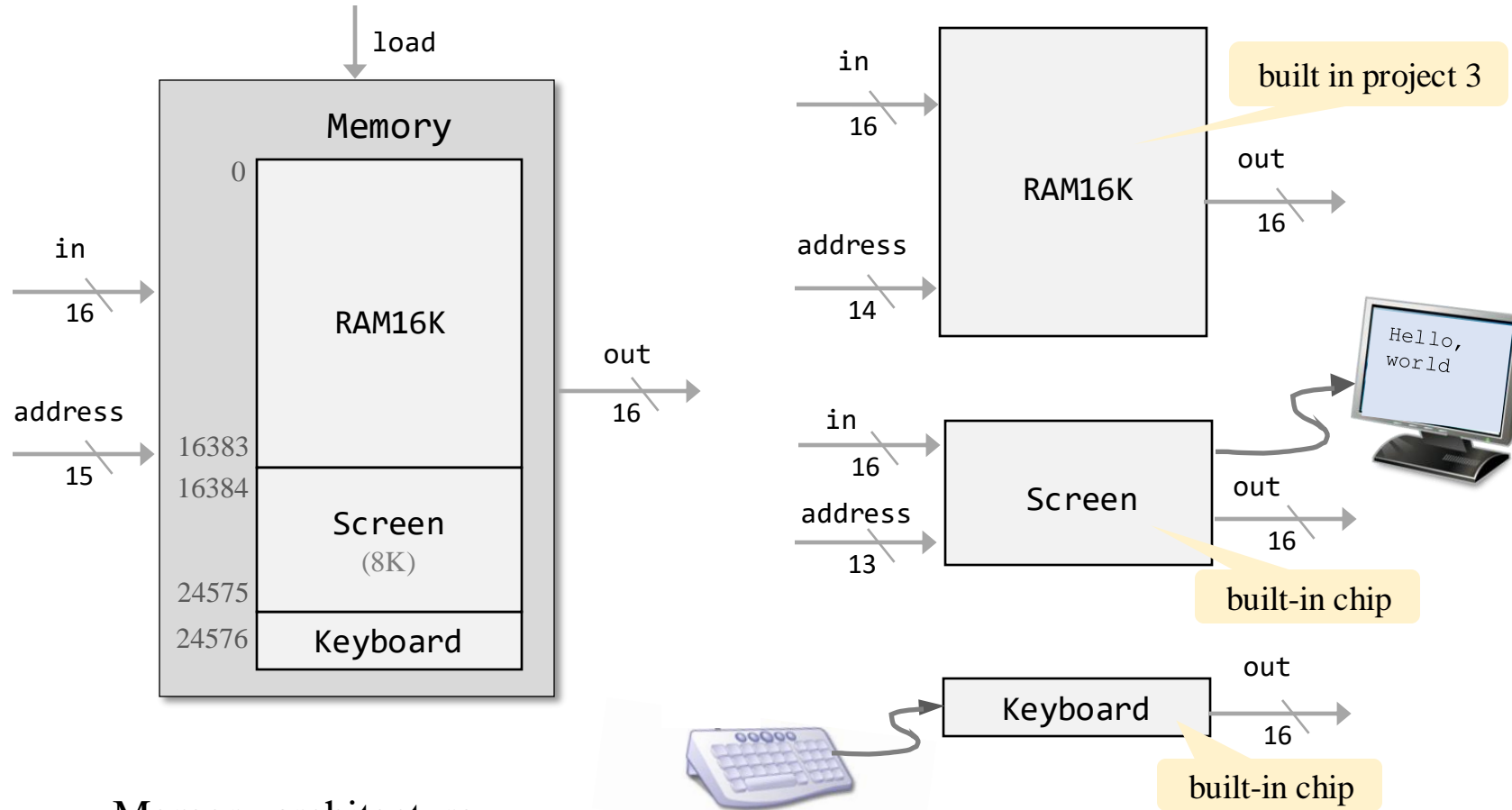
Memory



Memory: Implementation



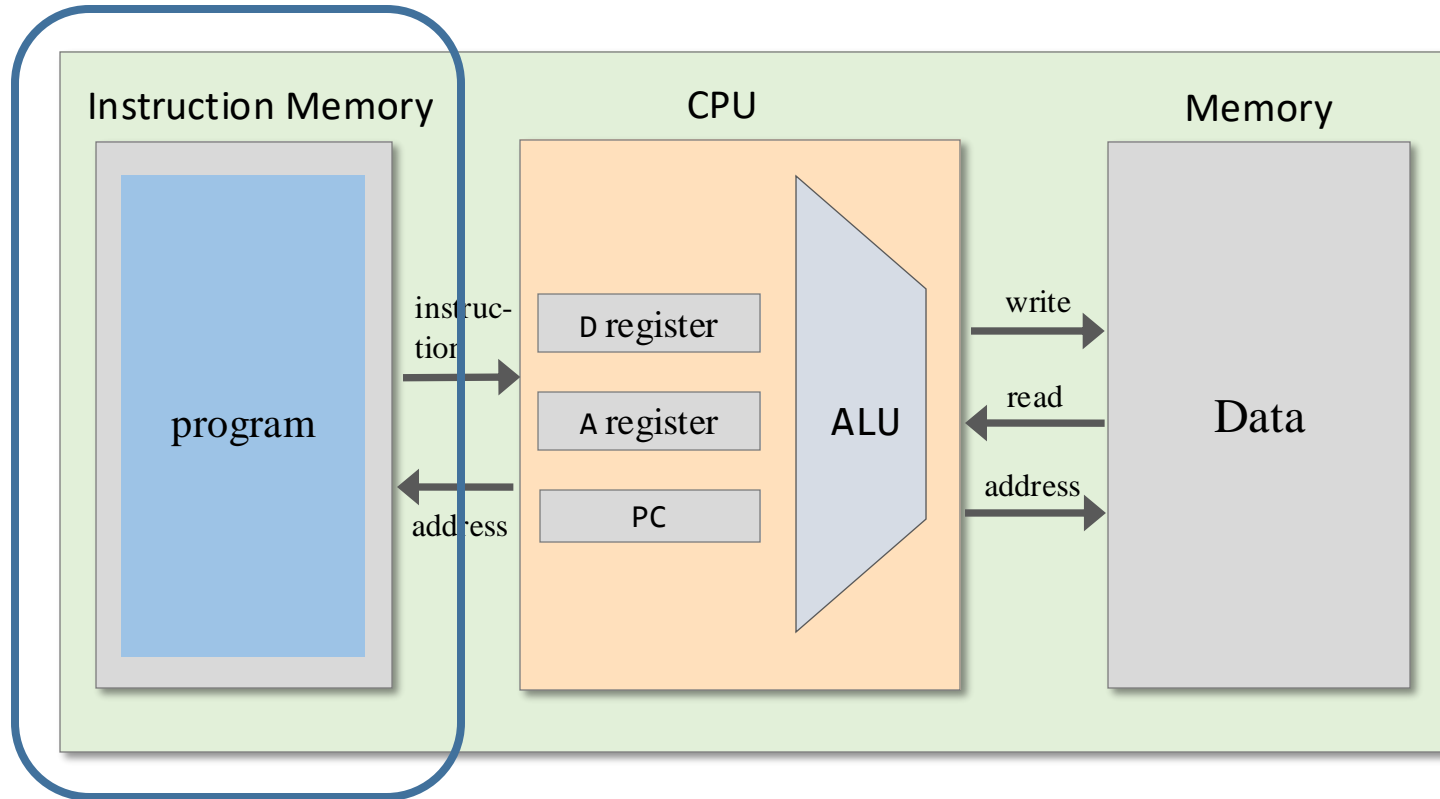
Memory: Implementation



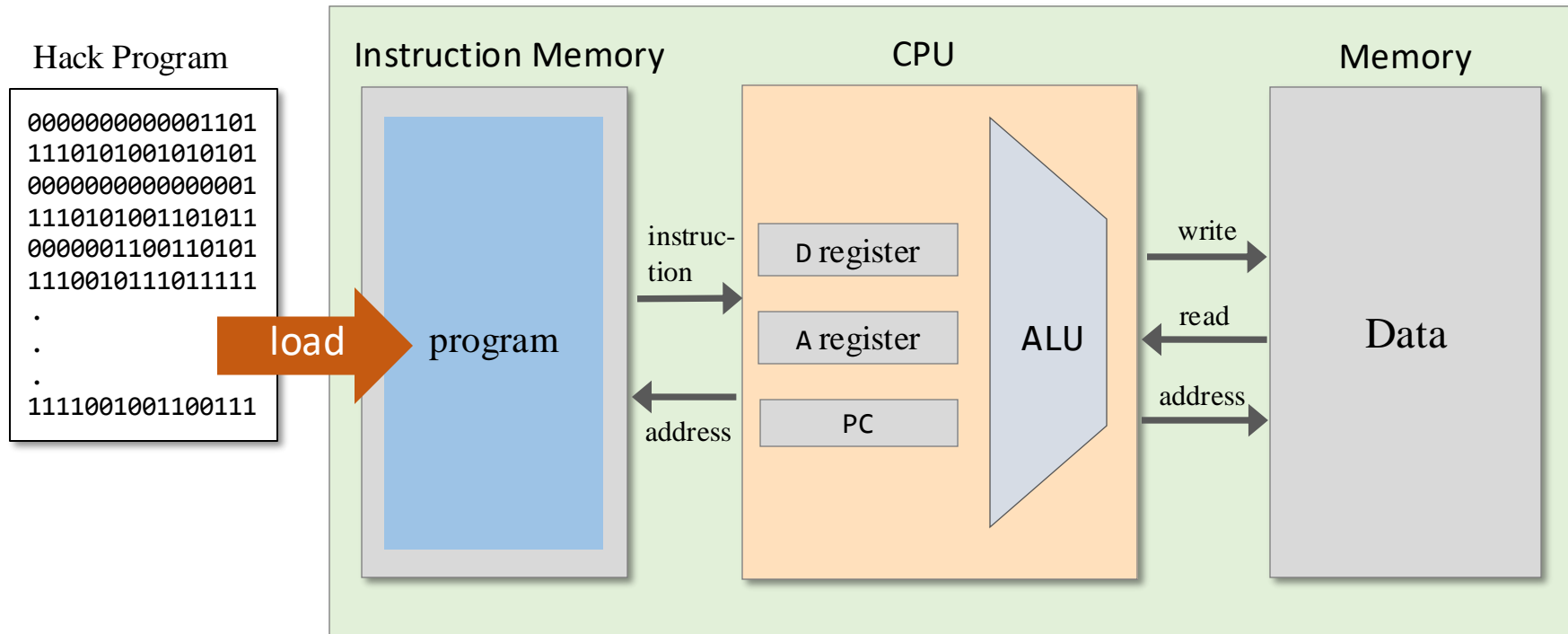
Memory architecture

- An aggregate of three chip-parts: RAM16K, Screen, Keyboard
- Single address space, 0 to 24576 (0x6000)
- Maps the address input onto the address input of the relevant chip-part.

Instruction memory



Instruction memory



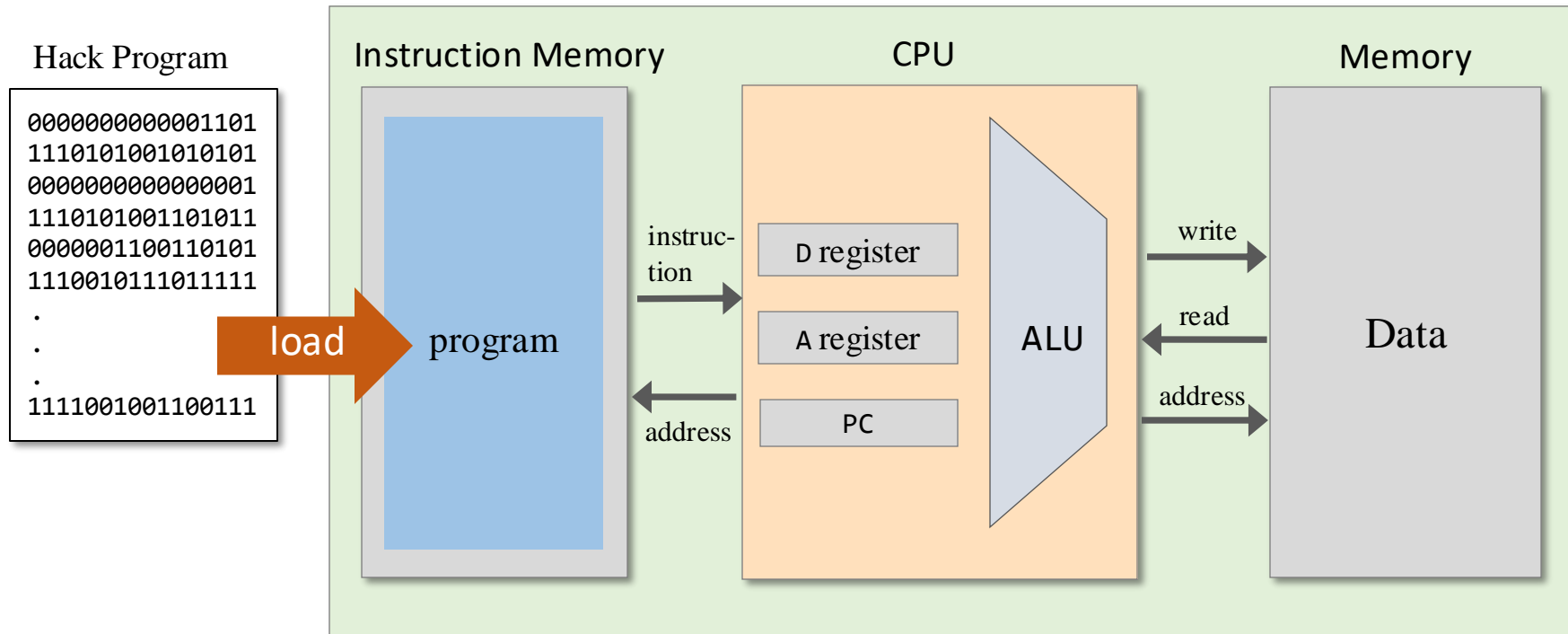
Instruction memory

Implemented as a built-in plug-and-play chip named ROM32K (pre-loaded with a program)

Loading a program

- Physical: Replace the ROM chip
- Simulator: Load a file containing instructions

Instruction memory



Instruction memory

Implemented as a built-in plug-and-play chip named ROM32K (pre-loaded with a program)

```
/** Read-Only memory (ROM),  
    acts as the Hack computer instruction memory. */  
CHIP ROM32K {  
    IN  address[15];  
    OUT out[16];  
    BUILTIN ROM32K;  
}
```

Chapter 5: Computer Architecture

- Basic architecture

✓ Memory

- Fetch-Execute cycle

➡ Computer

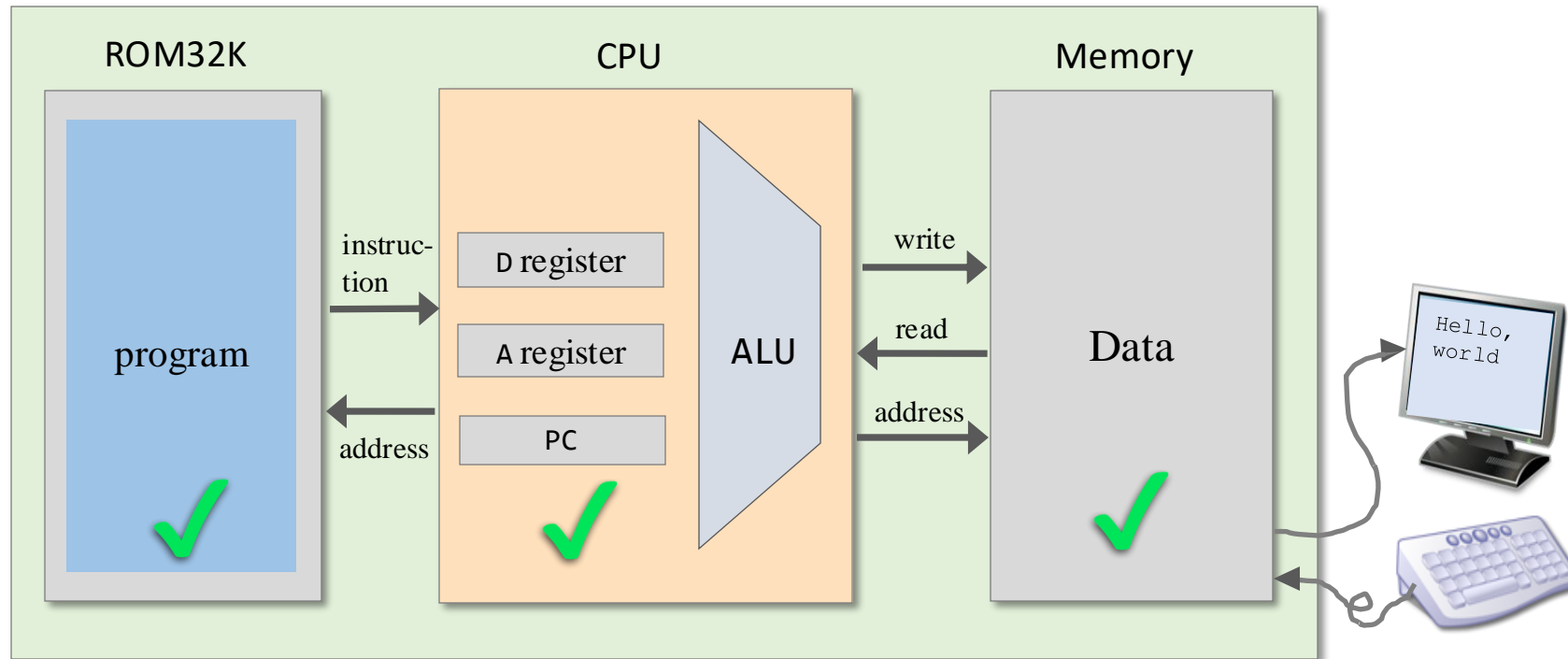
- The Hack CPU

- Project 5: Chips

- Input / output

- Project 5: Guidelines

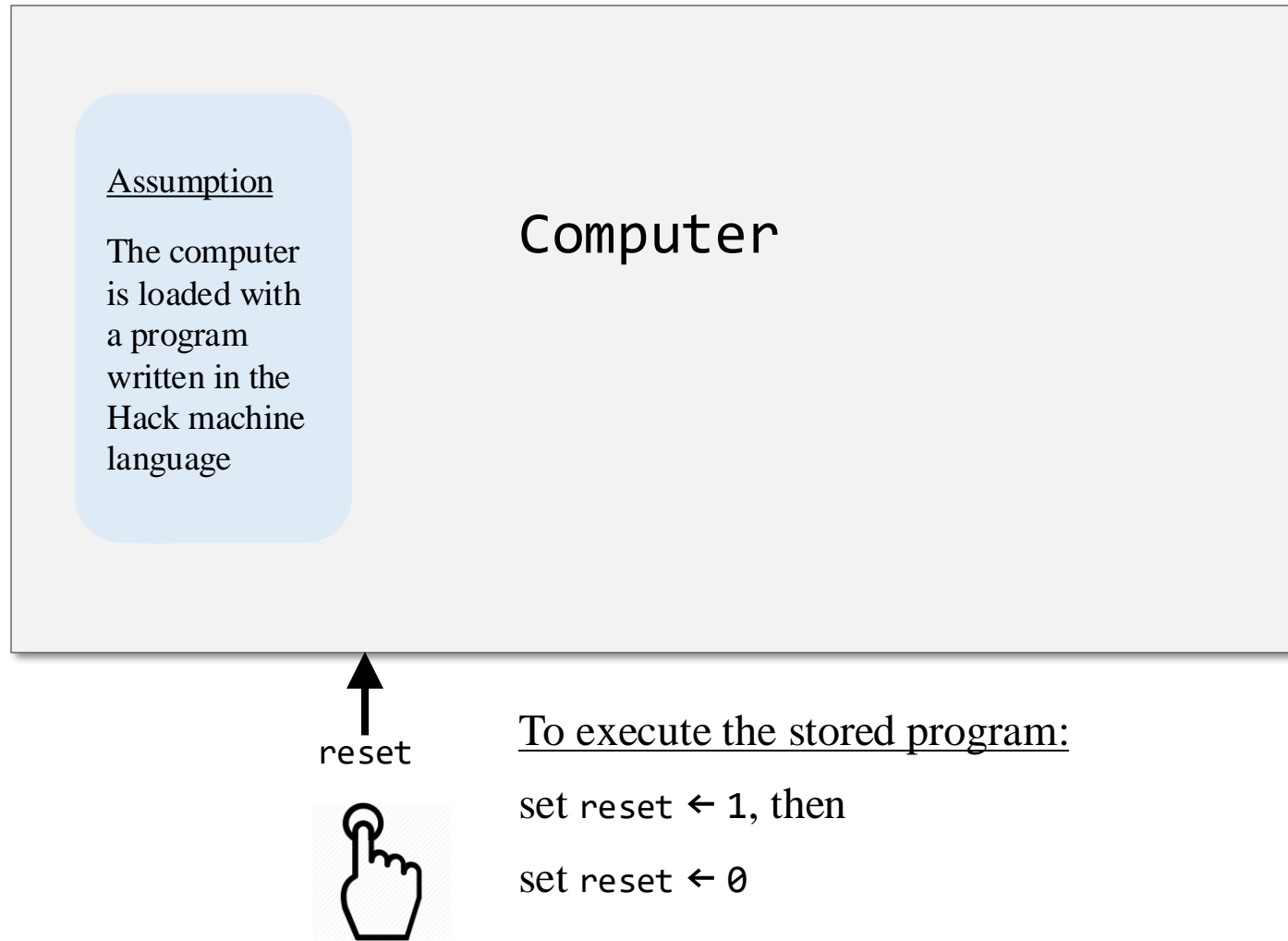
Hack computer architecture



Remaining challenge

Integrate into a single chip, named Computer

Computer abstraction



Computer abstraction

Assumption

The computer is loaded with a program written in the Hack machine language

Computer

if ($\text{reset} == 1$), executes the *first* instruction in the stored program

if ($\text{reset} == 0$), executes the *next* instruction in the stored program

↑
reset

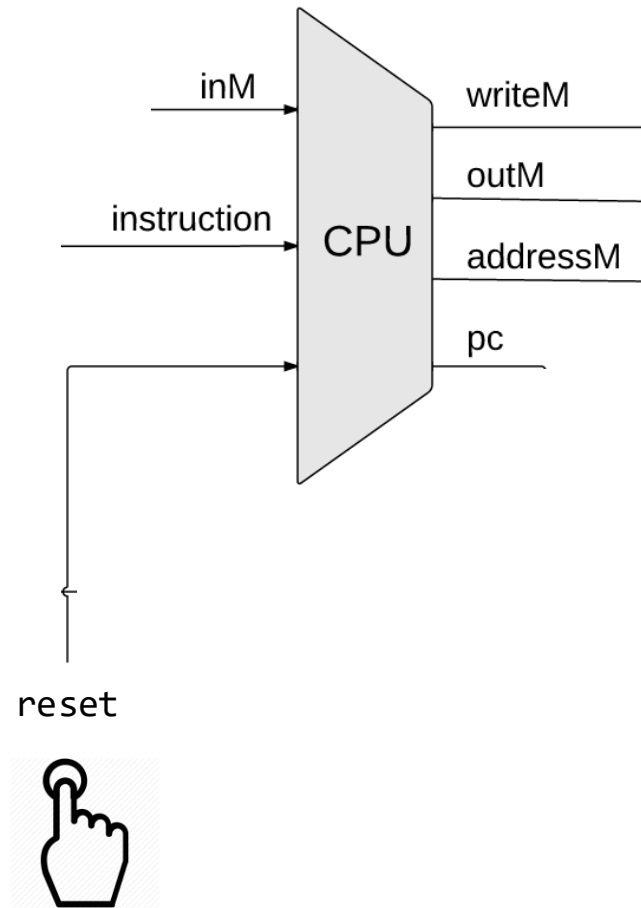


To execute the stored program:

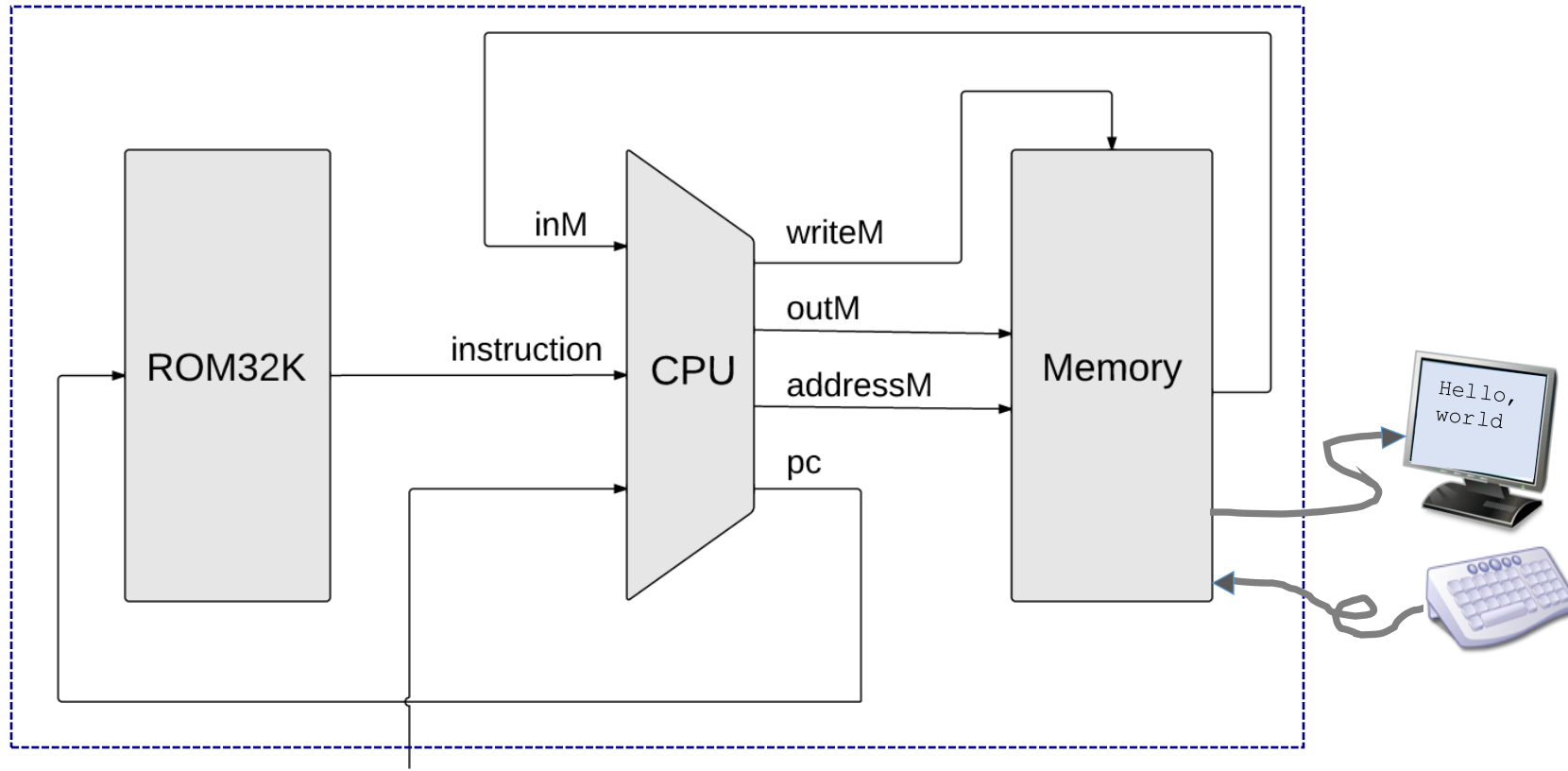
set $\text{reset} \leftarrow 1$, then

set $\text{reset} \leftarrow 0$

Computer implementation



Computer implementation



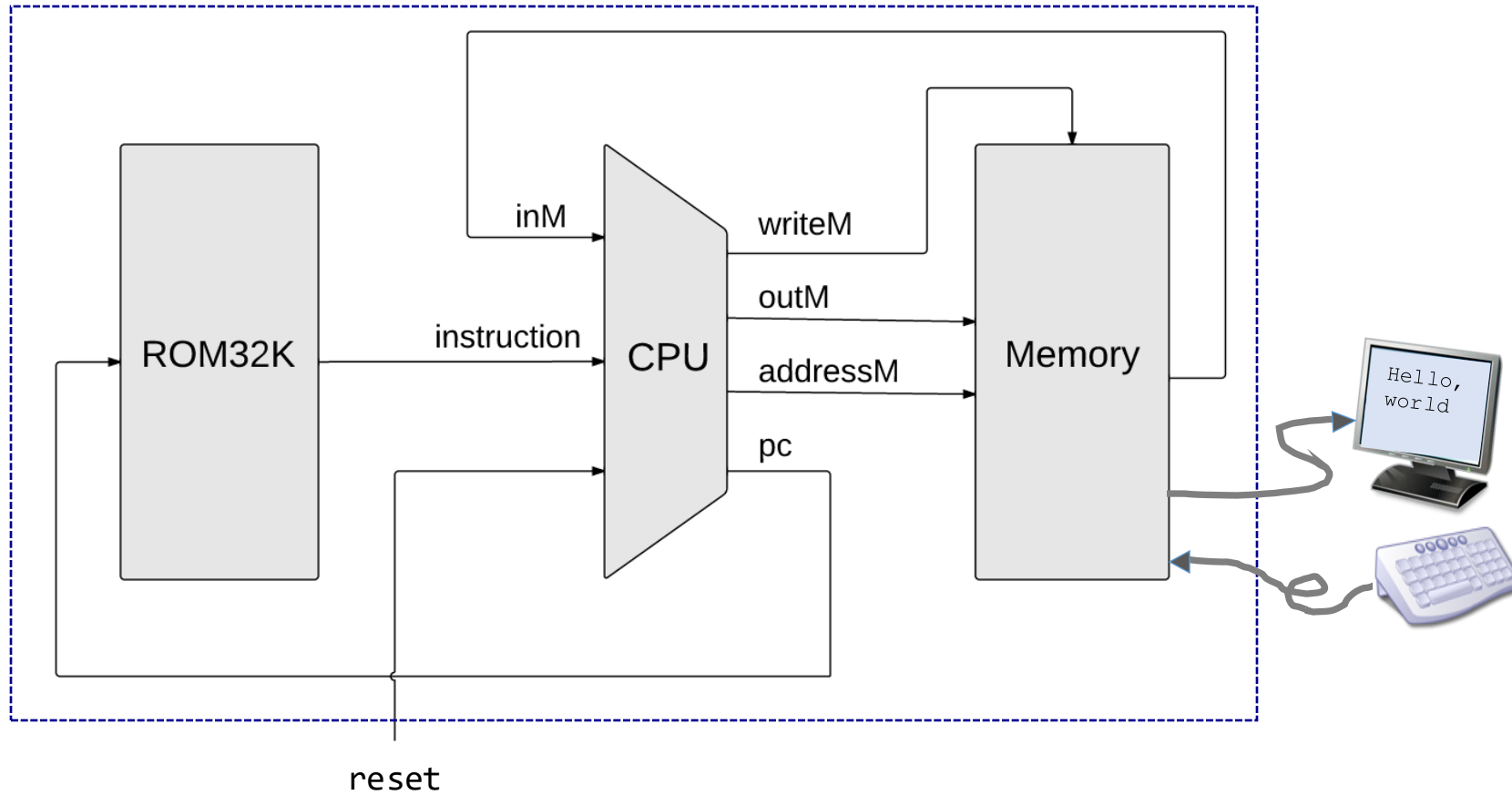
reset



Implementation

Connect the three chip-parts
along these lines.

Computer implementation



Computer Architecture

- Basic architecture
- Fetch-Execute cycle
- The Hack CPU
- Input / output

✓ Memory

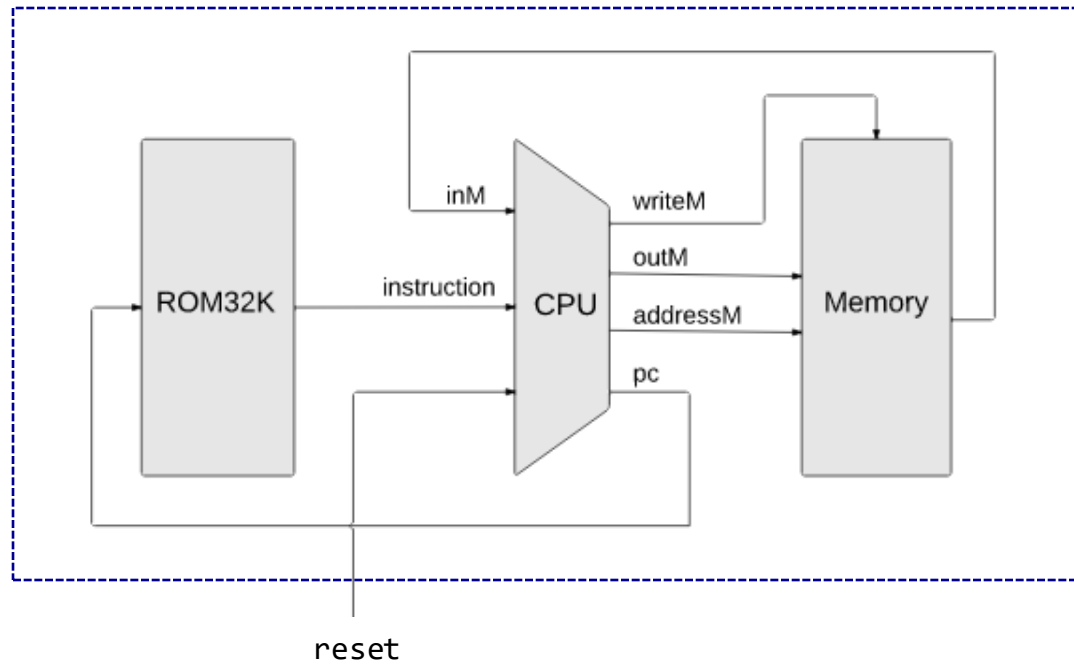
✓ Computer



Project 5: Chips

- Project 5: Guidelines

Computer Architecture



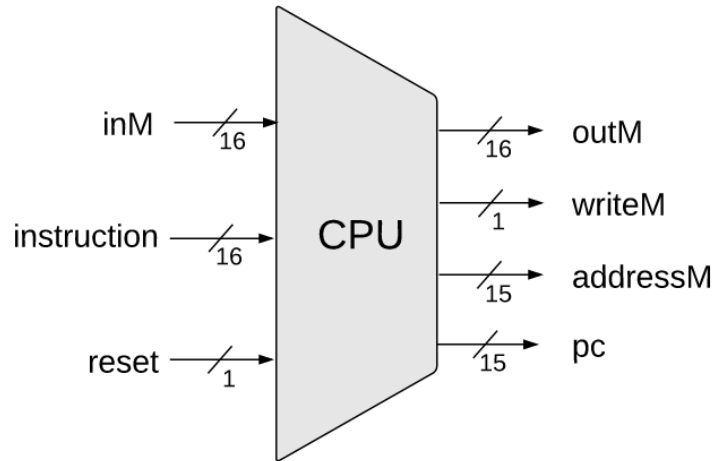
Project 5



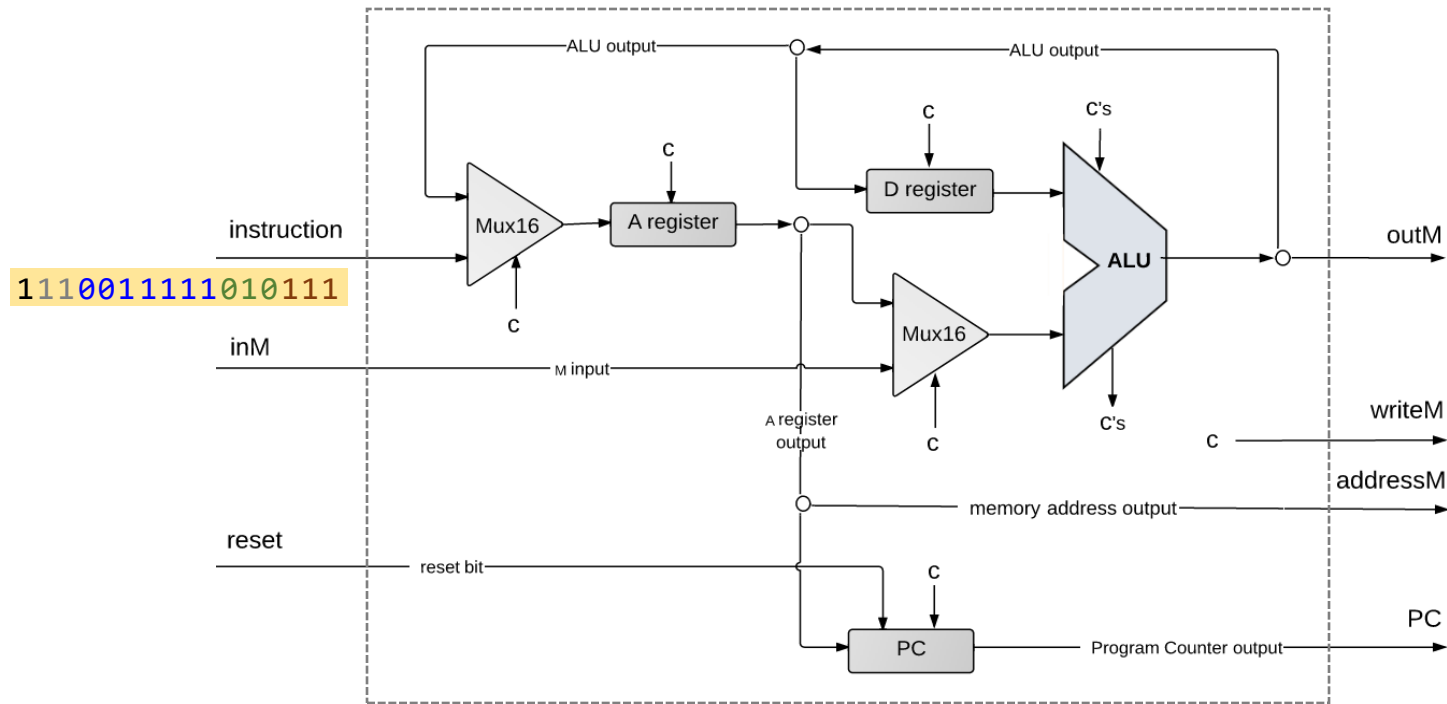
- CPU
- Memory
- Computer

CPU

```
/** Central Processing unit.  
    Executes instructions written in Hack machine language.  
CHIP CPU {  
    IN  
        inM[16],           // Value of M (RAM[A])  
        instruction[16],   // Instruction to execute  
        reset;             // Signals whether to execute the first instruction  
                           // (reset==1) or next instruction (reset == 0)  
  
    OUT  
        outM[16]           // Value to write to the selected RAM register  
        writeM,            // Write to the RAM?  
        addressM[15],      // Address of the selected RAM register  
        pc[15];            // Address of the next instruction  
  
    PARTS:  
        //// Put your code here  
}
```



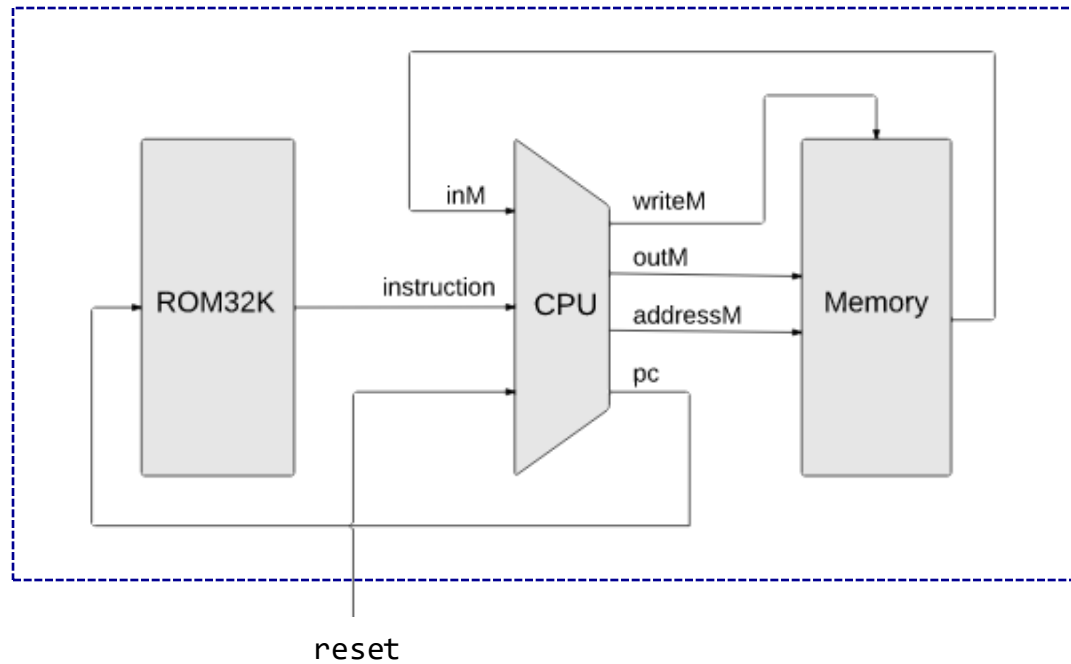
CPU



Implementation tips

- All the chip-parts seen here were built in projects 1, 2, 3
- But: we'll use their built-in versions
- Use HDL to unpack the instruction bits and connect them to the control bits of chip-parts
- Use gate logic to compute the address of the next instruction.

Computer Architecture



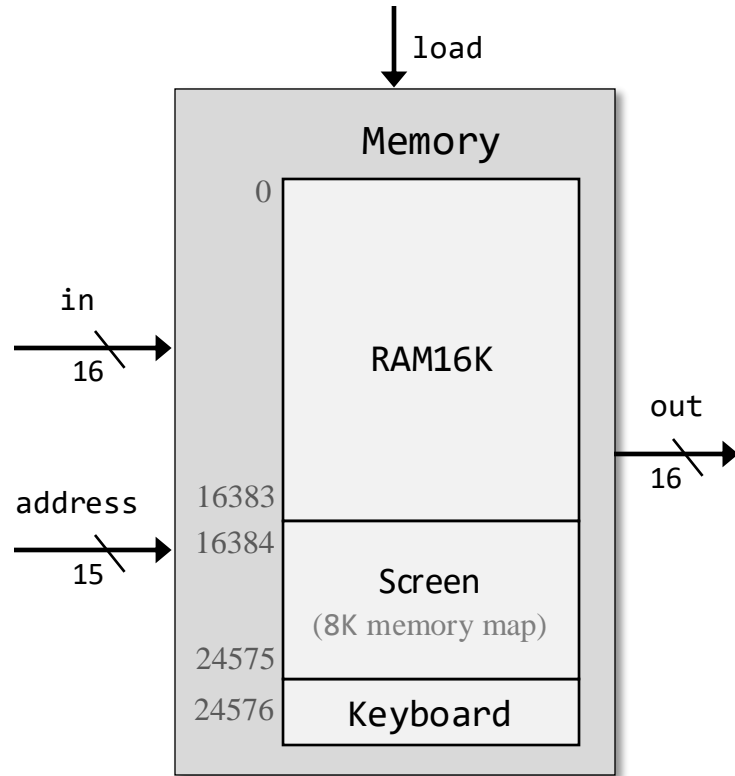
Project 5:

✓ CPU

➡ Memory

- Computer

Memory



Memory.hdl

```
/** Complete address space of the computer's data memory,
    including RAM, screen memory map, and keyboard memory map.
    Outputs the value of the memory location specified by address.
    If (load==1), the in value is loaded into the memory location specified by address.

    Addressing space rules:
    Only the upper 16K+8K+1 words of the memory are used.

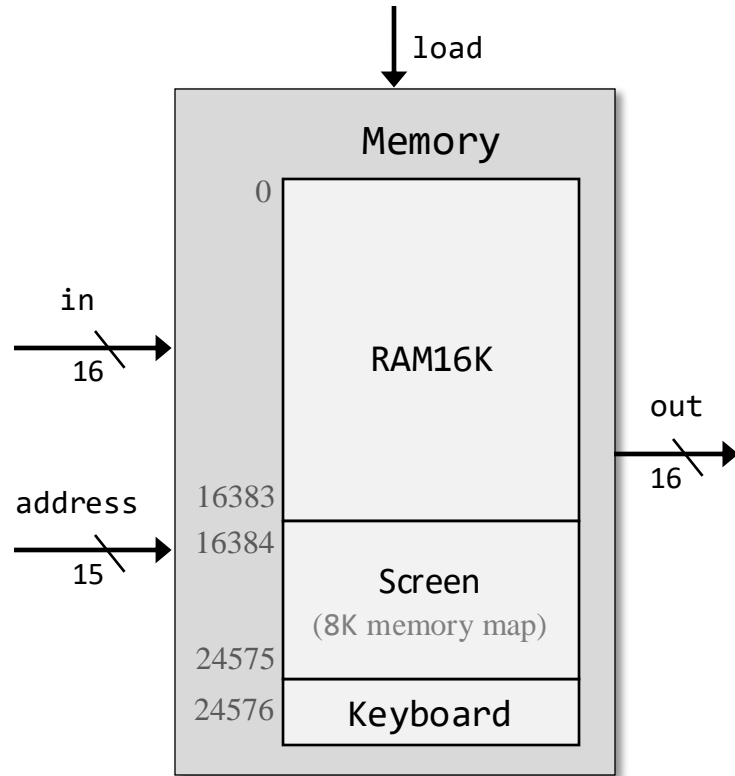
    Access to address 0 to 16383 (0x0000 to 0x3FFF) results in accessing the RAM;
    Access to address 16384 to 24575 (0x4000 to 0x5FFF) results in accessing the Screen memory map;
    Access to address 24576 (0x6000) results in accessing the Keyboard memory map.
*/

CHIP Memory {
    IN    address[15], in[16], load;
    OUT   out[16];
    PARTS:
        //// Put your code here.
}
```

Implementation tips

- Two bits in the address input can be used to determine the target memory-part (RAM16K, Screen, Keyboard)
- The remaining bits of the address input are the target address within the target memory-part
- Do the read/write specified by the Memory's inputs, and output the value of the selected memory-part[address] to the Memory's out output.

Memory



Memory.hdl

```
/** Complete address space of the computer's data memory,
    including RAM, screen memory map, and keyboard memory map.
    Outputs the value of the memory location specified by address.
    If (load==1), the in value is loaded into the memory location specified by address.

    Addressing space rules:
    Only the upper 16K+8K+1 words of the memory are used.

    Access to address 0 to 16383 (0x0000 to 0x3FFF) results in accessing the RAM;
    Access to address 16384 to 24575 (0x4000 to 0x5FFF) results in accessing the Screen memory map;
    Access to address 24576 (0x6000) results in accessing the Keyboard memory map.
*/

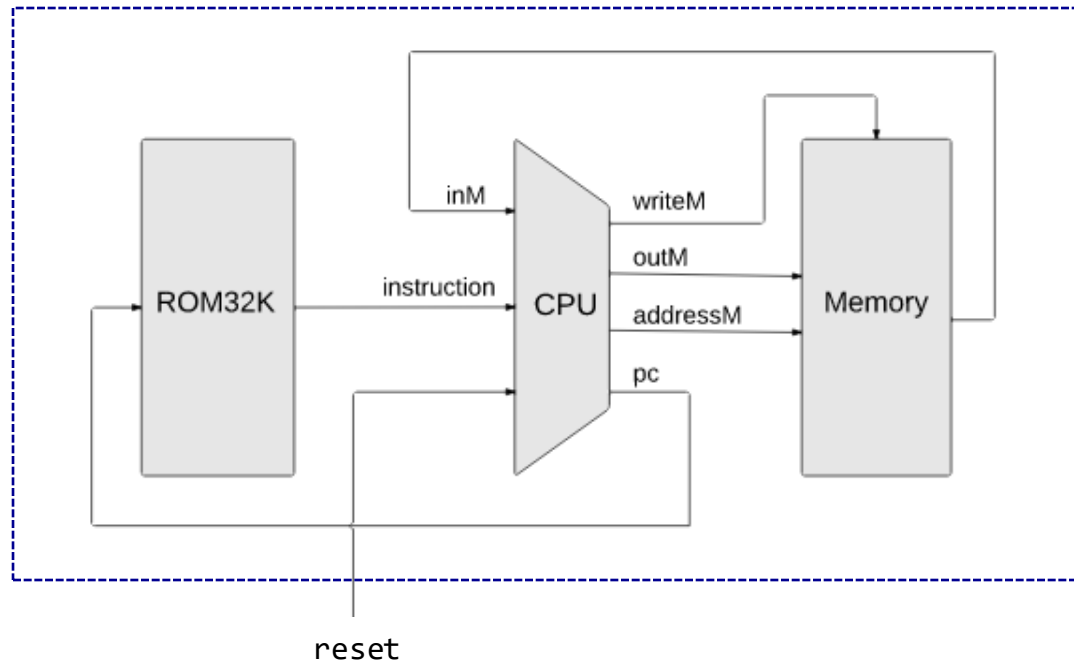
CHIP Memory {
    IN    address[15], in[16], load;
    OUT   out[16];
    PARTS:
        //// Put your code here.
}
```

Implementation tips (continued)

- Use builtin memory-parts (RAM16K, Screen, Keyboard), and builtin logic gates, as needed.
- Optional: Start by building a basic Memory chip that outputs two bits, say, loadRAM and loadScreen, indicating if the addressed memory-part is the RAM or the Screen

Then complete the final Memory chip, in which these two bits can become internal pins.

Computer



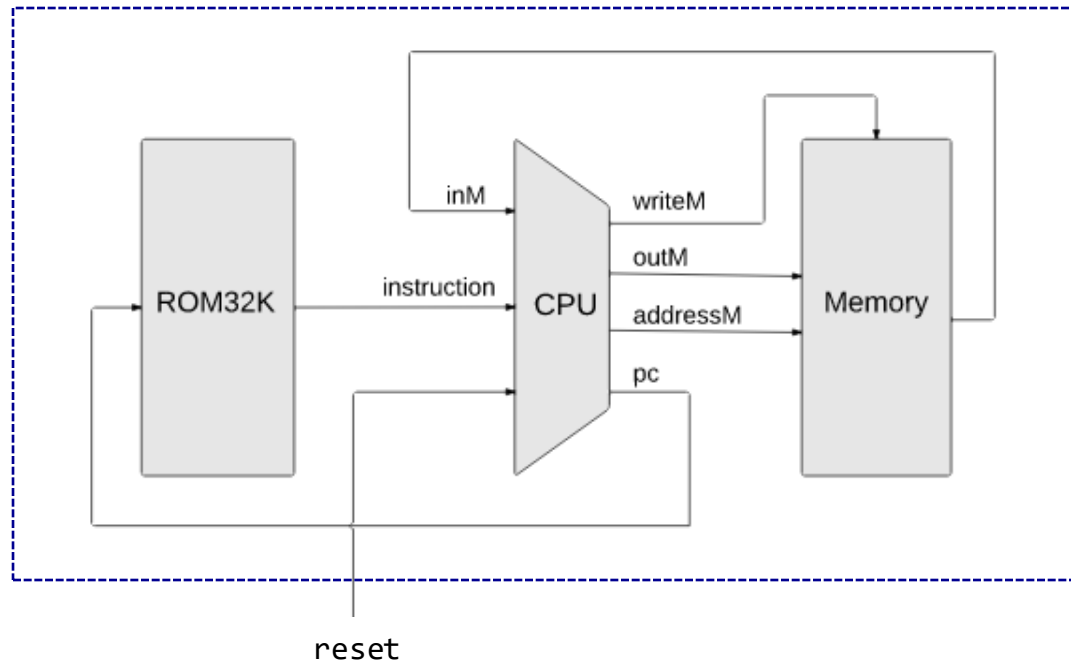
Project 5:

✓ CPU

✓ Memory

➡ Computer

Computer



```
/** The Hack computer, including CPU, RAM and ROM, loaded with a program.  
    When (reset==1), the computer executes the first instruction in the program;  
    When (reset==0), the computer executes the next instruction in the program. */
```

```
CHIP Computer {  
    IN reset;  
    PARTS:  
        // Put your code here.  
}
```

Implementation tips

Use the built-in ROM32K

Follow the diagram, and use HDL to connect the three chip-parts

Computer Architecture

- Basic architecture
 - Fetch-Execute cycle
 - The Hack CPU
 - Input / output
- ✓ Memory
 - ✓ Computer
 - ✓ Project 5: Chips
 - ➔ Project 5: Testing

Testing the Computer chip

Testing logic

- Load `Computer.hdl` into the hardware simulator
- Load a Hack program into the ROM32K chip-part
- Run the clock enough cycles to execute the program

`Computer.hdl`

```
/** The Hack computer, including CPU, RAM and ROM,
    loaded with a program. */
CHIP Computer {
    IN reset;
    PARTS:
        //// Completed HDL code
}
```

Test programs

- `Add.hack`:
 $\text{RAM}[0] \leftarrow 2 + 3$
- `Max.hack`:
 $\text{RAM}[2] \leftarrow \max(\text{RAM}[0], \text{RAM}[1])$
- `Rect.hack`:
Draws a rectangle of $\text{RAM}[0]$ rows of 16 pixels each.

Testing the Computer chip

Testing logic

- Load Computer.hdl into the hardware simulator
- Load a Hack program into the ROM32K chip-part
- Run the clock enough cycles to execute the program

Test programs

- Add.hack:
RAM[0] $\leftarrow 2 + 3$
- ➔ Max.hack:
RAM[2] $\leftarrow \max(\text{RAM}[0], \text{RAM}[1])$
- Rect.hack:
Draws a rectangle of RAM[0] rows of 16 pixels each.

ComputerMax.tst

```
load Computer.hdl,
output-file ComputerMax.out,
compare-to ComputerMax.cmp,
output-list time reset ARegister[] DRegister[] PC[]
          RAM16K[0] RAM16K[1] RAM16K[2];

// Loads a Hack program (that computes R2 = max(R0,R1))
ROM32K load Max.hack,

// Test 1: computes max(3,5)
set RAM16K[0] 3,
set RAM16K[1] 5,
output;
repeat 14 {
    tick, tock, output;
}

// Resets the PC
set reset 1,
tick, tock, output;

// Test 2: computes max(23456,12345)
set reset 0,
set RAM16K[0] 23456,
set RAM16K[1] 12345,
output;
repeat 14 {
    tick, tock, output;
}
```


Testing the Computer chip

Testing logic

- Load `Computer.hdl` into the hardware simulator
- Load a Hack program into the ROM32K chip-part
- Run the clock enough cycles to execute the program

Test programs

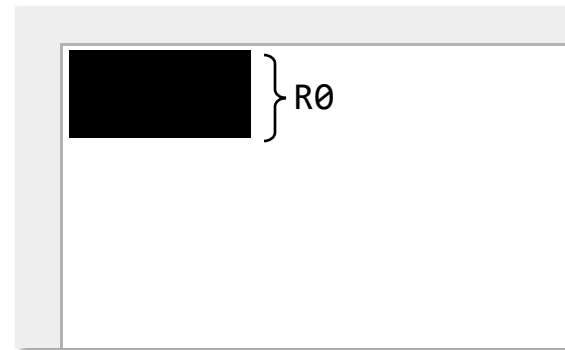
- `Add.hack`:
 $\text{RAM}[0] \leftarrow 2 + 3$
- `Max.hack`:
 $\text{RAM}[2] \leftarrow \max(\text{RAM}[0], \text{RAM}[1])$



`Rect.hack`:

Draws a rectangle of $\text{RAM}[0]$ rows of 16 pixels each.

`Rect.hack` output:



Test script

- `ComputerRect.tst`
- Inspect it, understand the testing logic.

What's next?

