Chapter 5

Computer Architecture

These slides support chapter 5 of the book

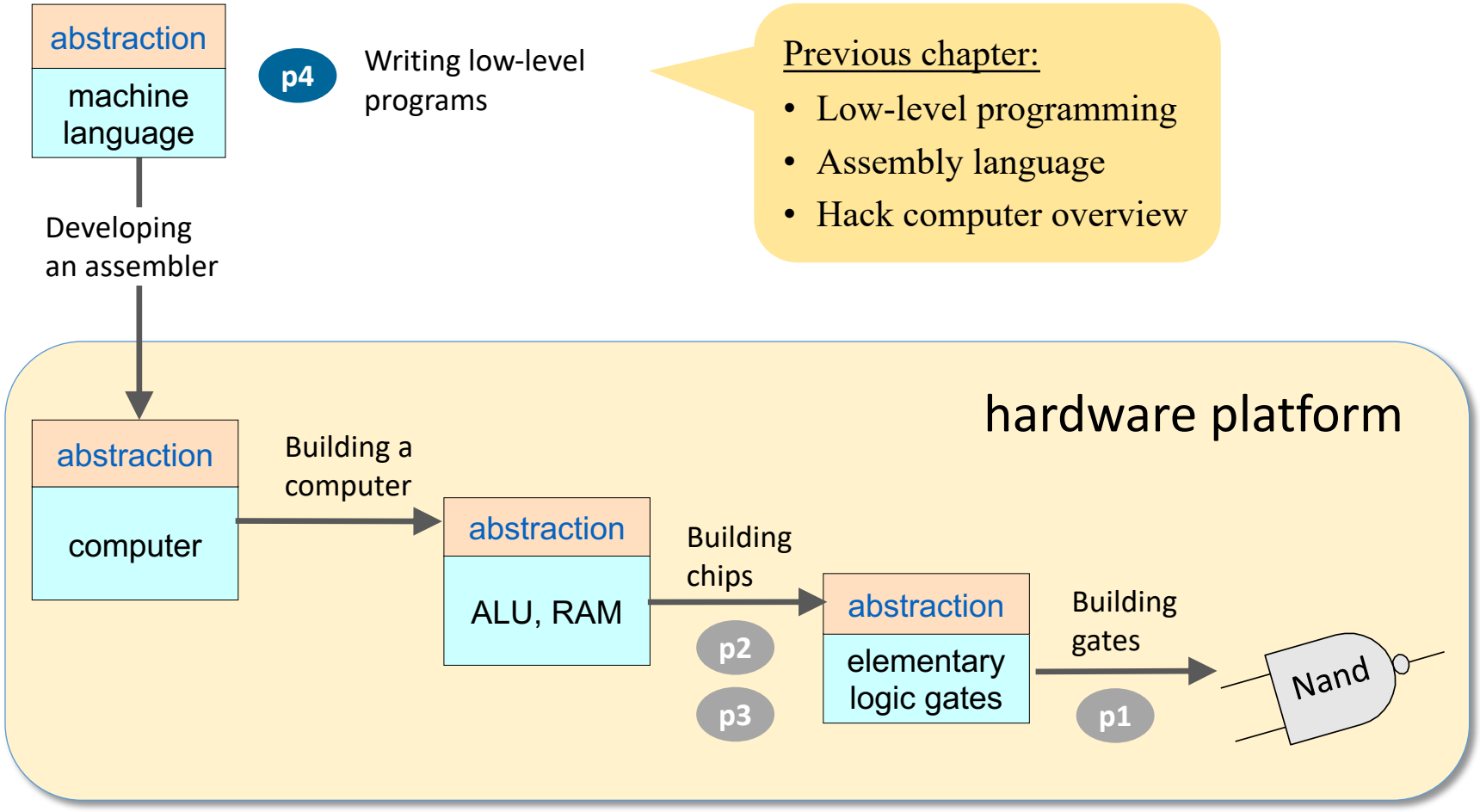
The Elements of Computing Systems

(1st and 2nd editions)

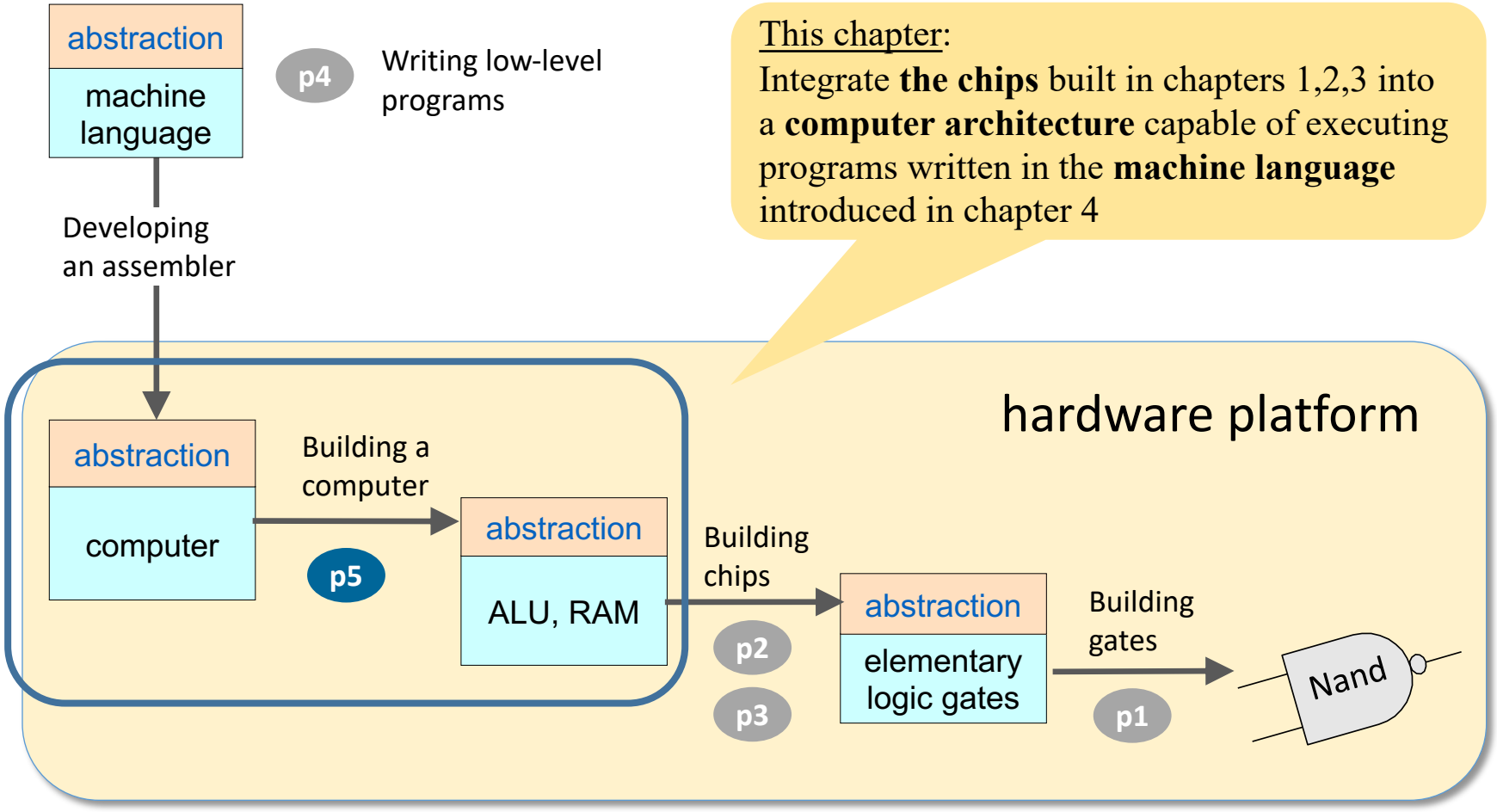
By Noam Nisan and Shimon Schocken

MIT Press

Nand to Tetris Roadmap (Part I: Hardware)

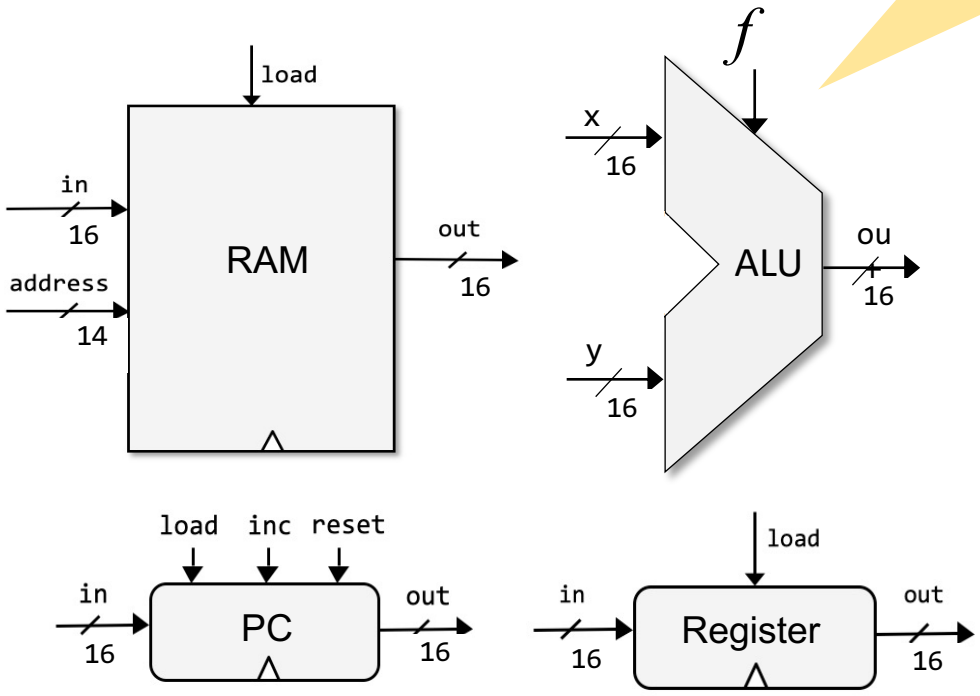


Nand to Tetris Roadmap (Part I: Hardware)



Nand to Tetris Roadmap (Part I: Hardware)

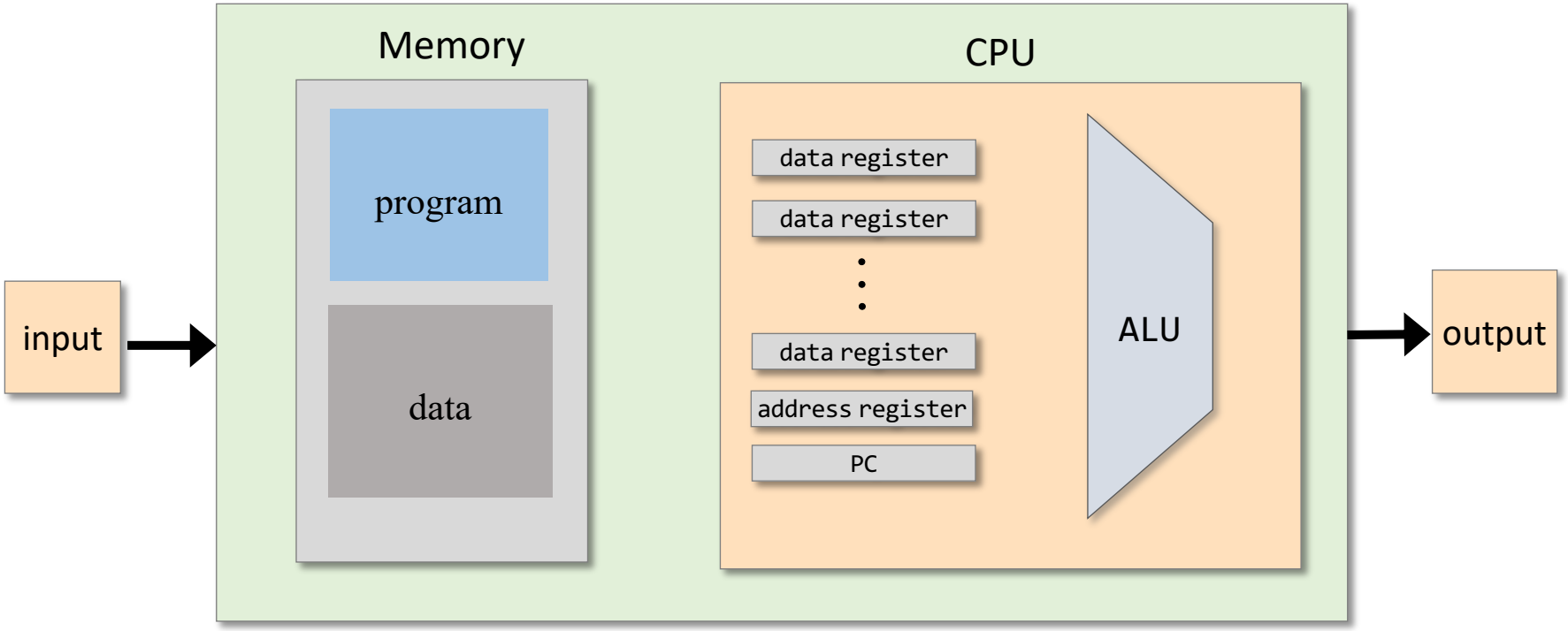
This chapter:
Integrate **the chips** built in chapters 1,2,3 into a **computer architecture** capable of executing programs written in the **machine language** introduced in chapter 4



Machine language program (example)

```
// 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



Typical computer architecture:

- Stored program concept
- General-purpose

In this chapter we build the Hack computer – a variant of this architecture.

Chapter 5: Computer Architecture

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

Chapter 5: Computer Architecture

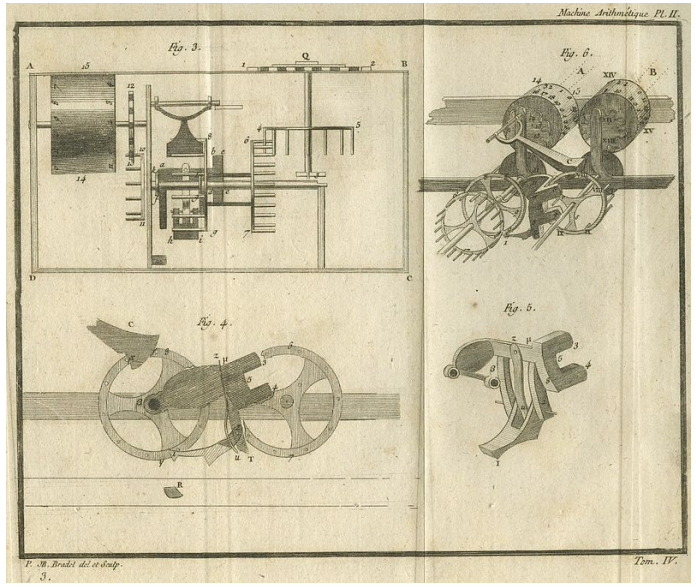
- Basic architecture
- Fetch-Execute cycle
- The Hack CPU
- Input / output
- Memory
- Computer
- Project 5: Chips
- Project 5: Guidelines

Early computers (17th century)

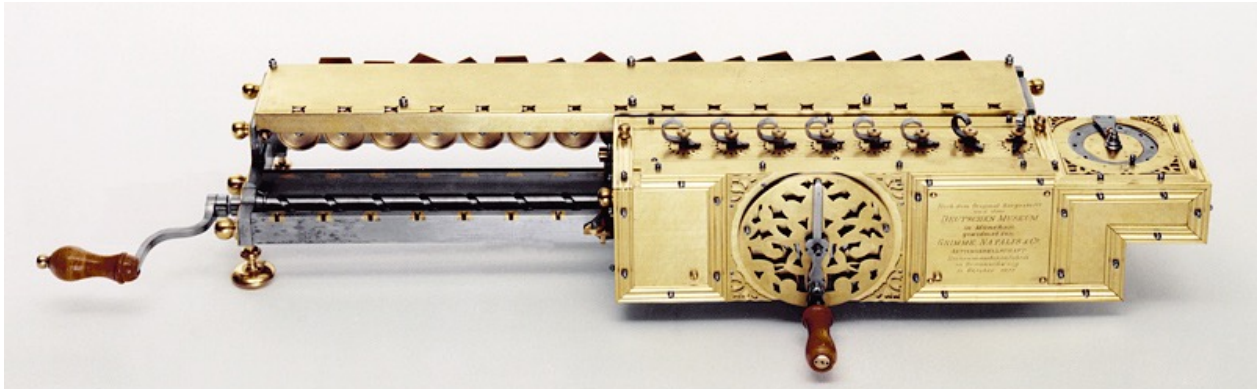
Blaise Pascal
1623-1662

Pascal's Calculator
(*Pascaline*, 1652)

- Add
- Subtract



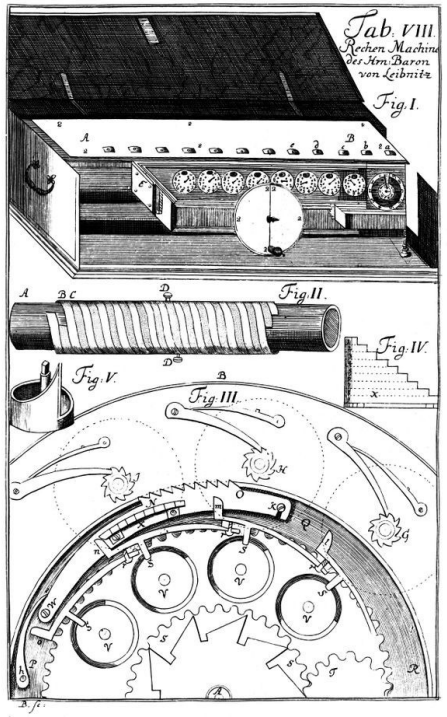
Early computers (17th century)



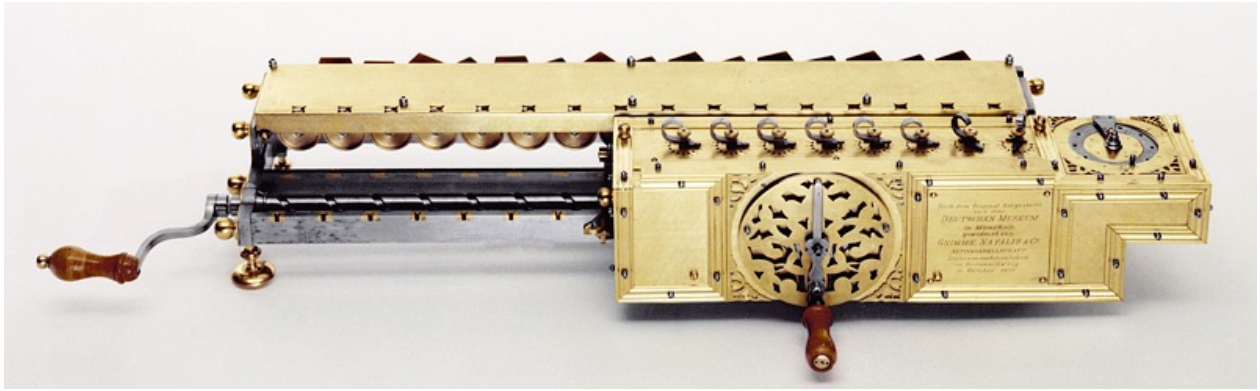
Gottfried Leibniz
1646-1716

Leibniz Calculator (1673)

- Add
- Subtract
- Multiply
- Divide.



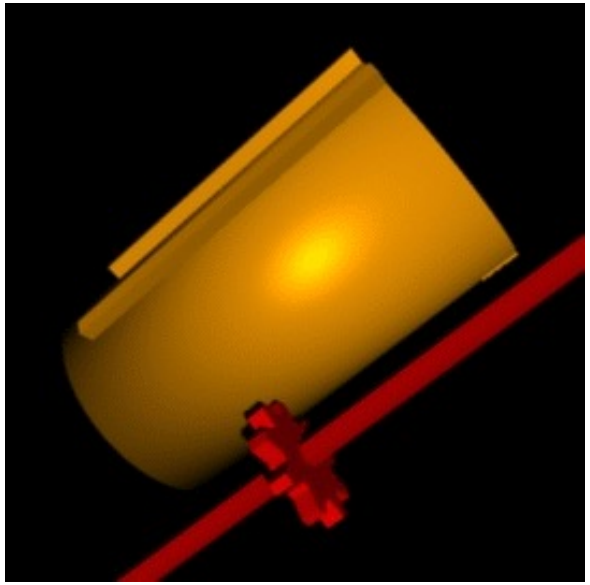
Early computers (17th century)



Gottfried Leibniz
1646-1716

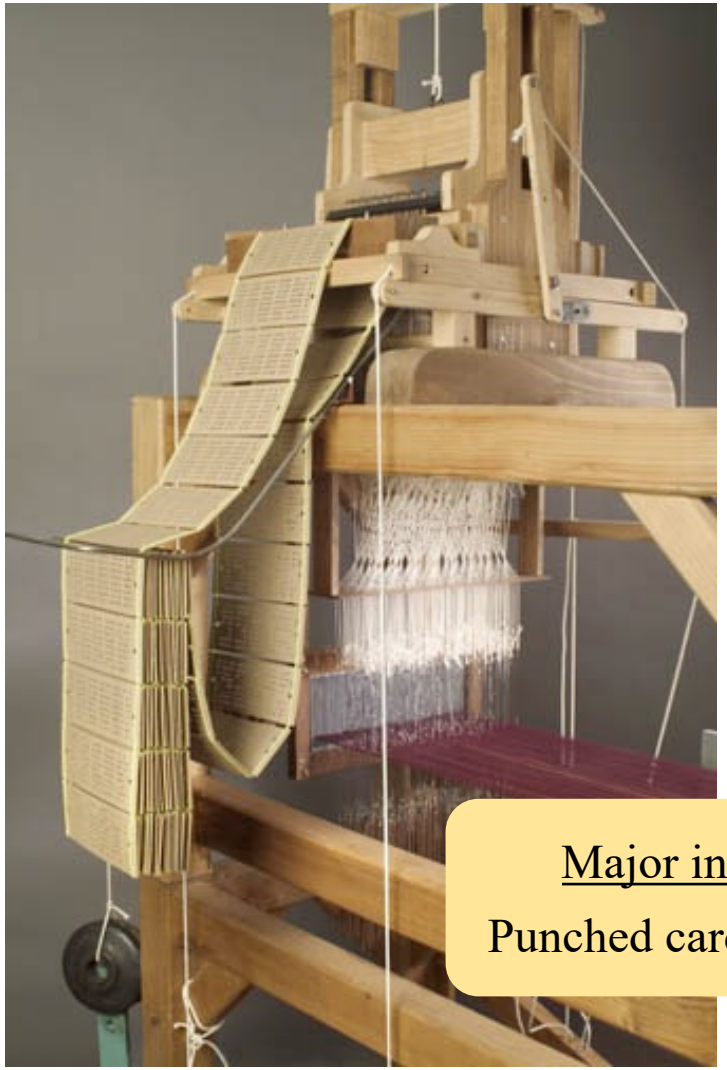
Leibniz Calculator (1673)

- Add
- Subtract
- Multiply
- Divide.



Side benefits:
Advances in gears / mechanical engineering

Early computers (19th century)



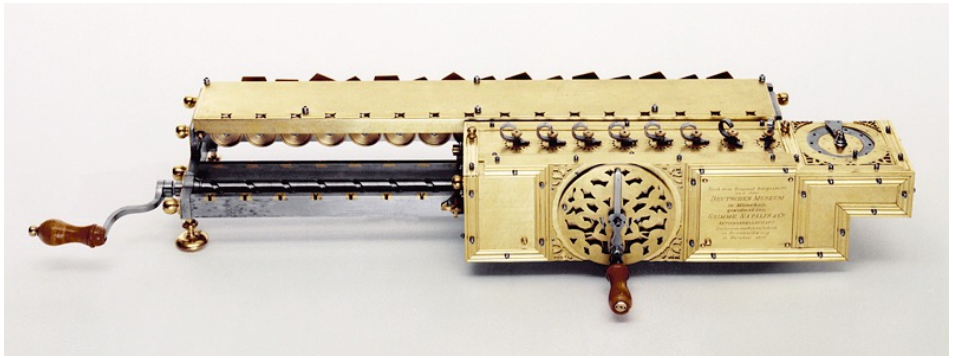
Jacquard Loom (1804)

Analytic Engine (1837)

Major innovation:

Punched cards = software

Early computers



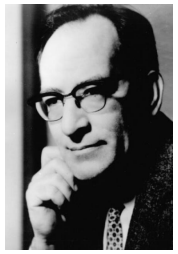
17th century: Hardware only / fixed / single purpose

Programmable!

19th century: Hardware / Software / General purpose

Modern computers (20th century)


John Von Neumann

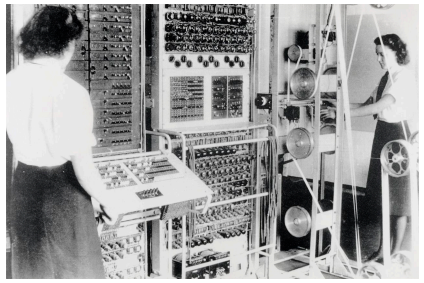

John Mauchly


Presper Eckert


John Atanassof

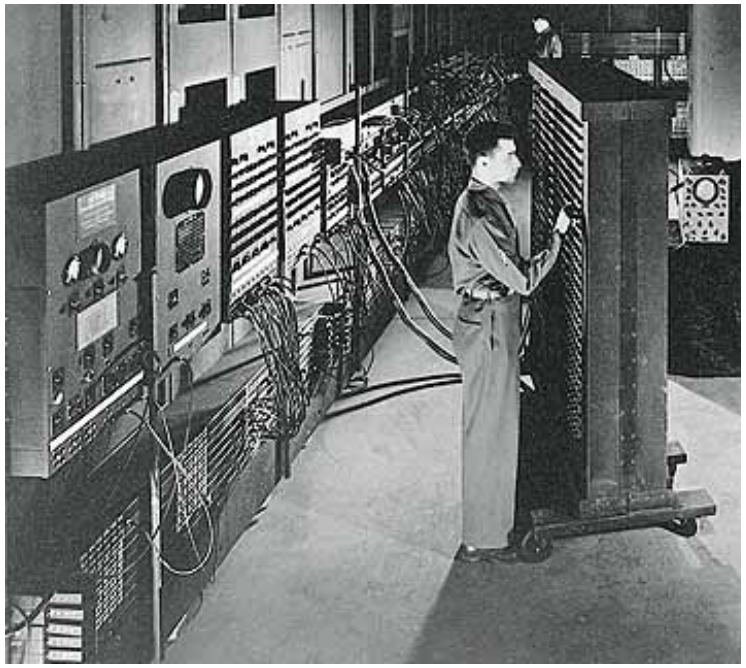

Howard Aiken


Konrad Zuse




Tommy Flowers

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



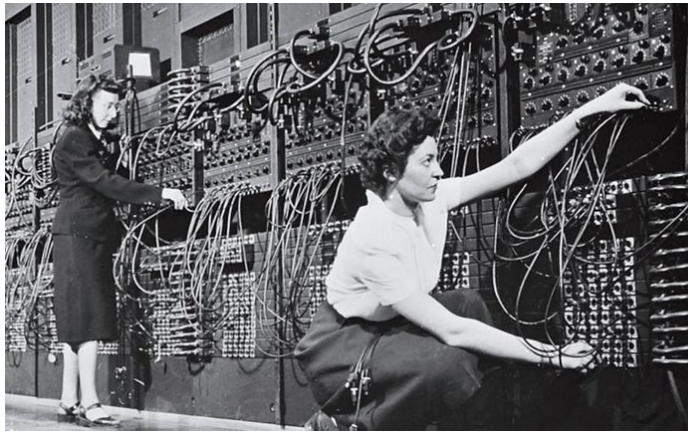
ENIAC: First digital, programmable, stored program computer

University of Pennsylvania, 1946,

(Borrowed key ideas from other early computers and innovators)

Modern computers (20th century)

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



ENIAC Women

Pioneered reusable code, subroutines, flowcharts, and many other programming innovations

Compilation pioneers

(Mark I)

Grace Hopper

Adele Koss

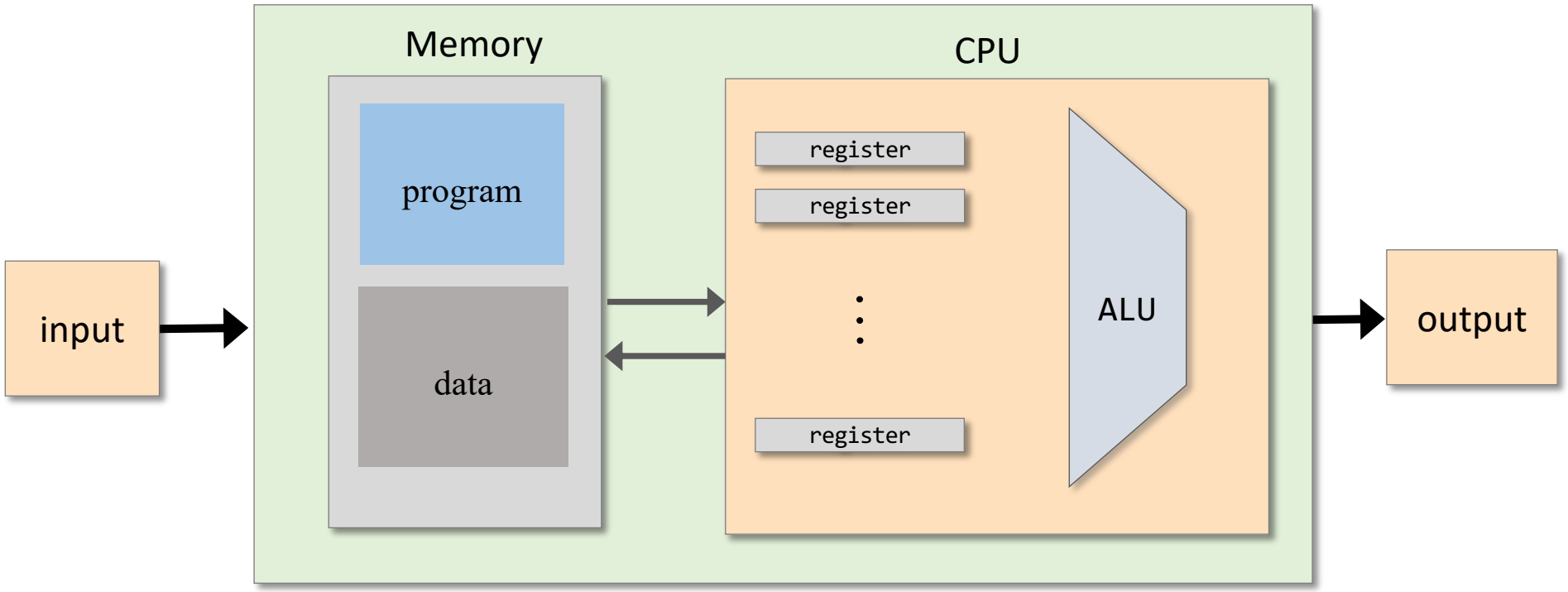
Modern computers (20th century)

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

Back in the 1950's, this was considered a radical idea:

"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, 1956 (Mark 1 computer architect)

Basic architecture

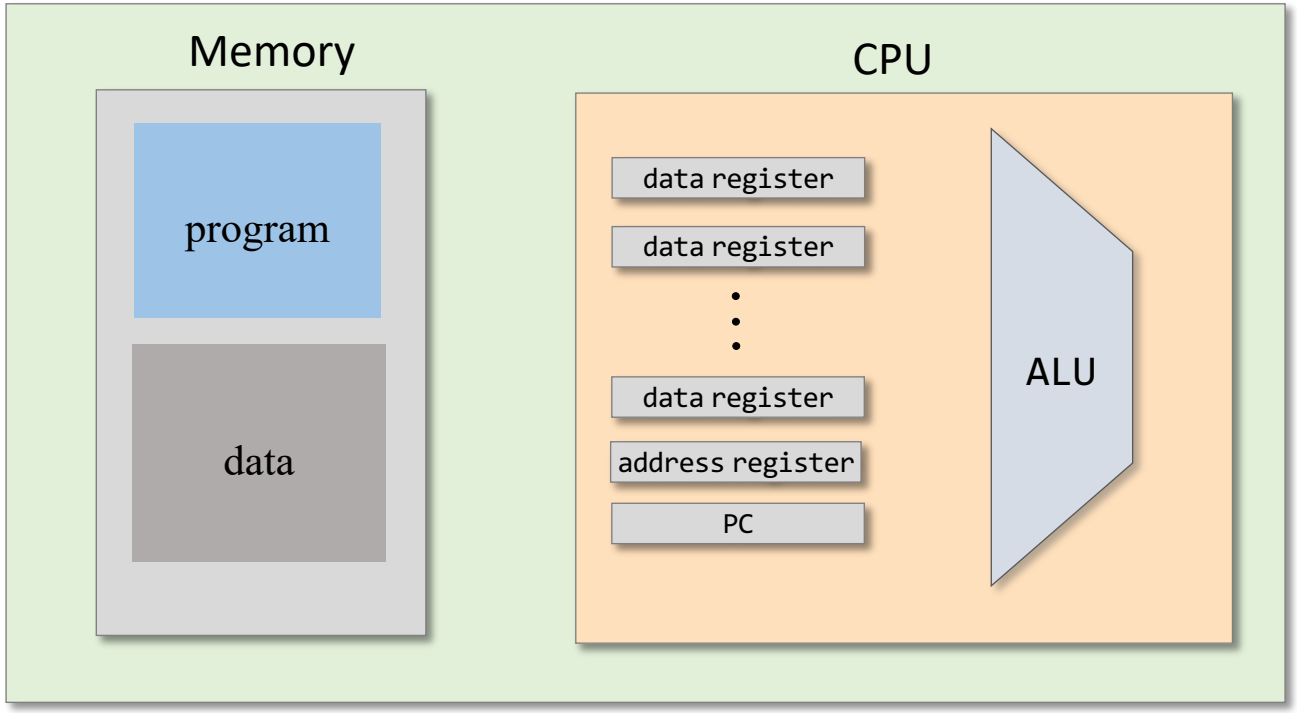


Stored program concept:

Same machine can run different programs

"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)

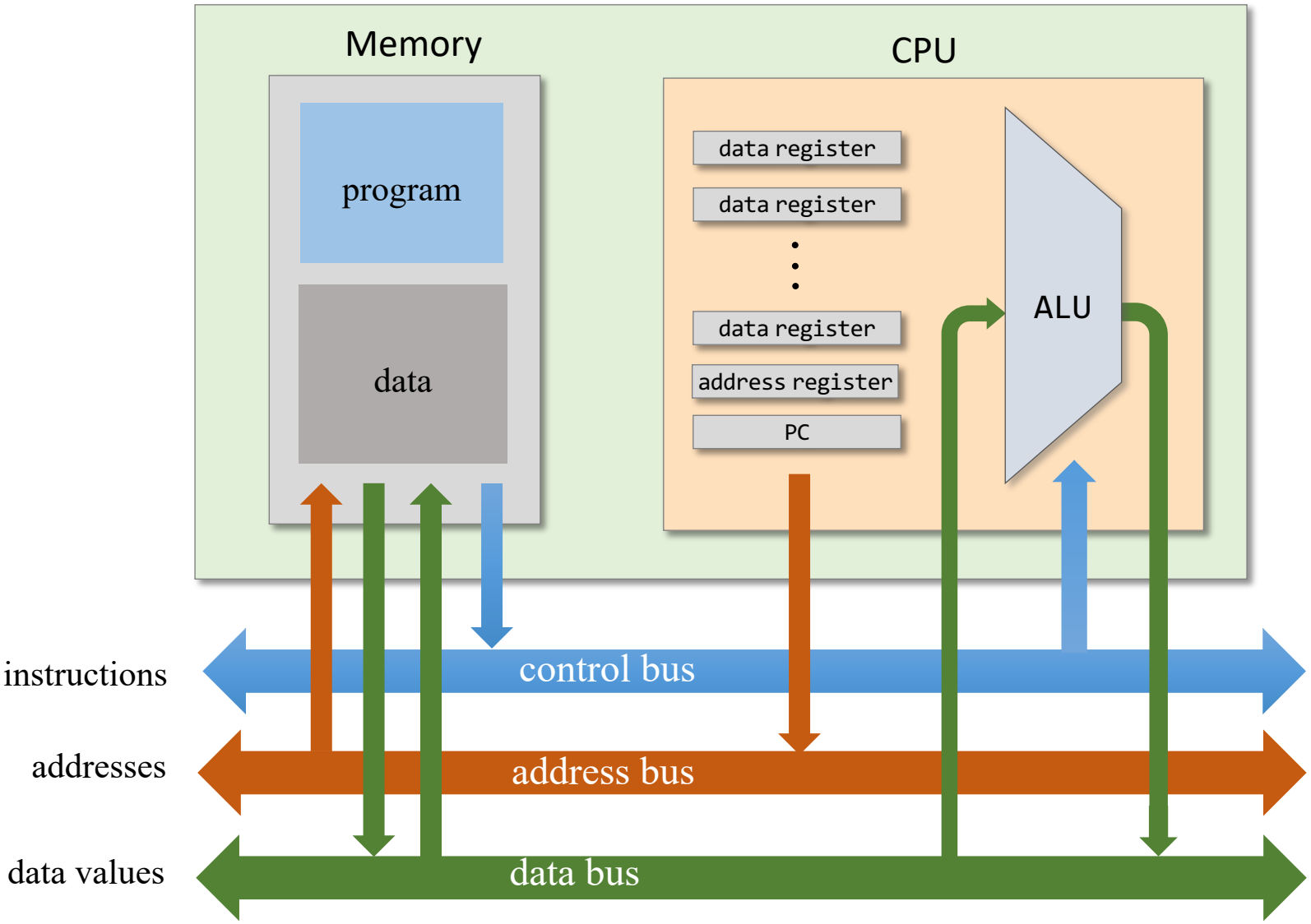
Basic architecture



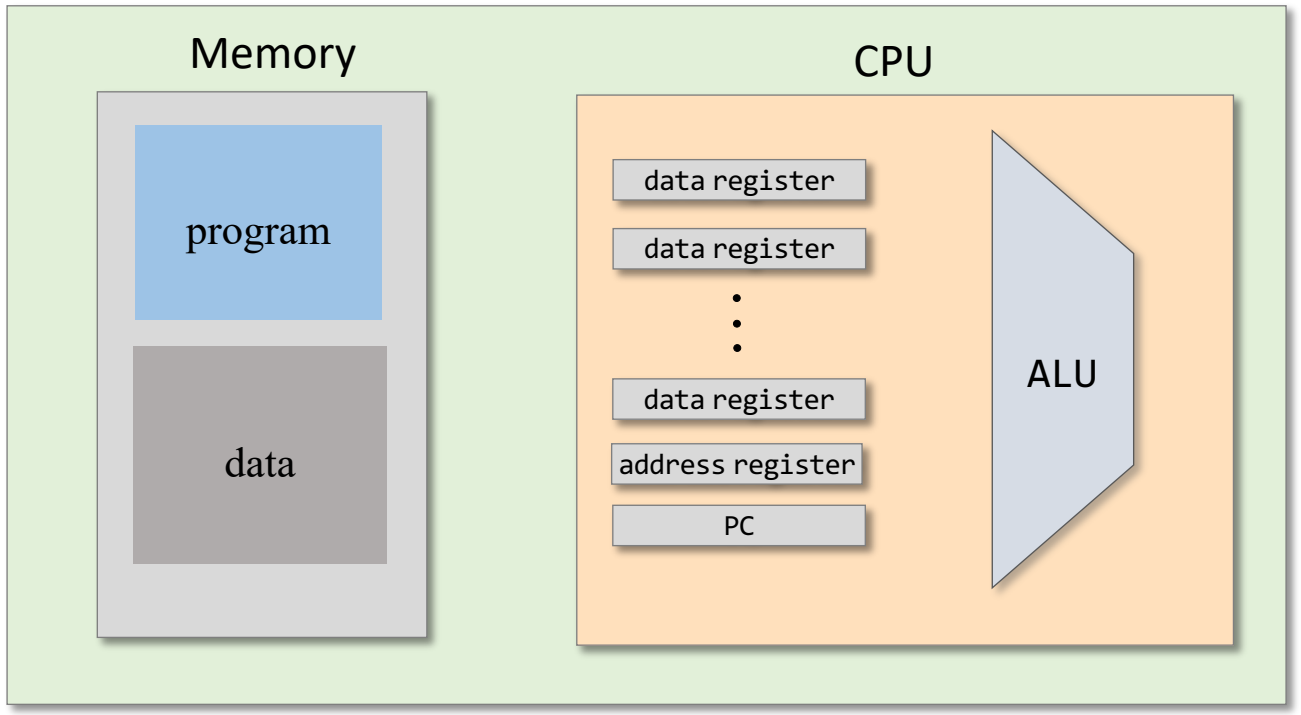
The computer is essentially a machine that manipulates *registers* under program control: *data* registers, *address* registers, a *program counter* (PC), *memory* registers (containing data and instructions).

How does information flow inside the computer?

Basic architecture



Basic architecture: Recap

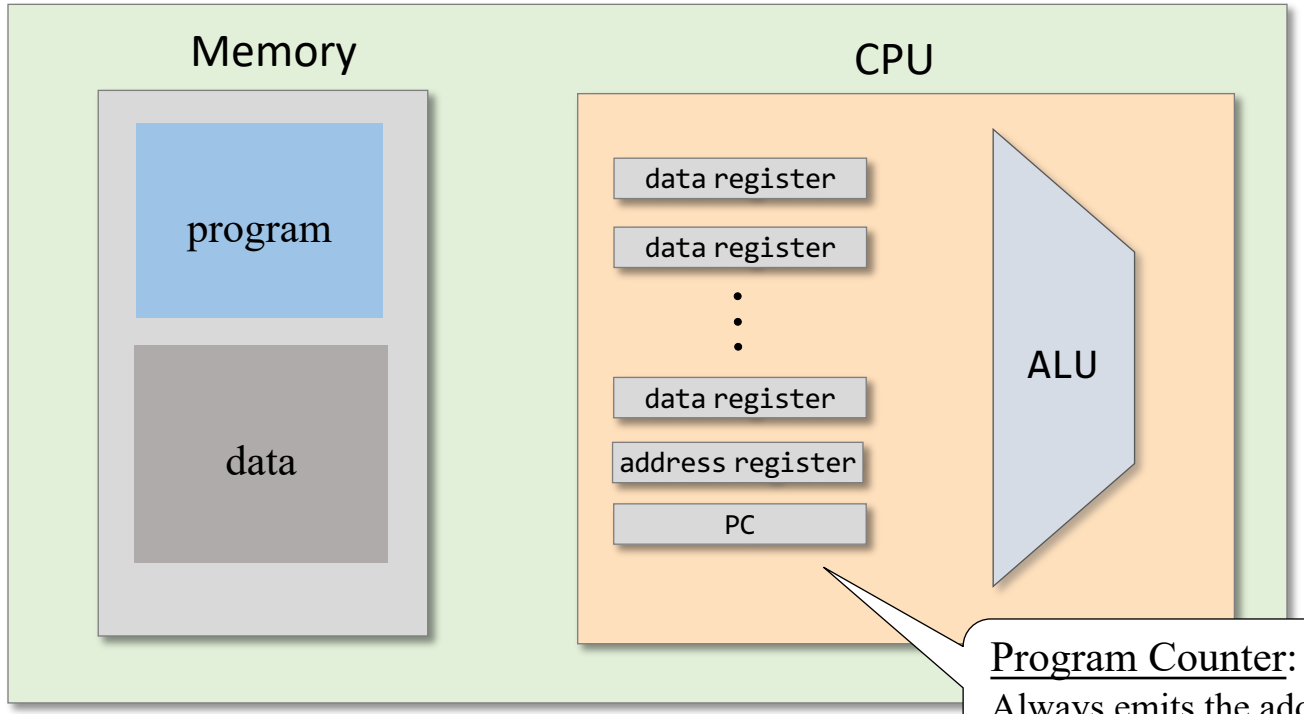


- General purpose computer
- A set of chips connected by buses
- Stored program concept
- Framework of most modern computers

Chapter 5: Computer Architecture

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

Computer architecture

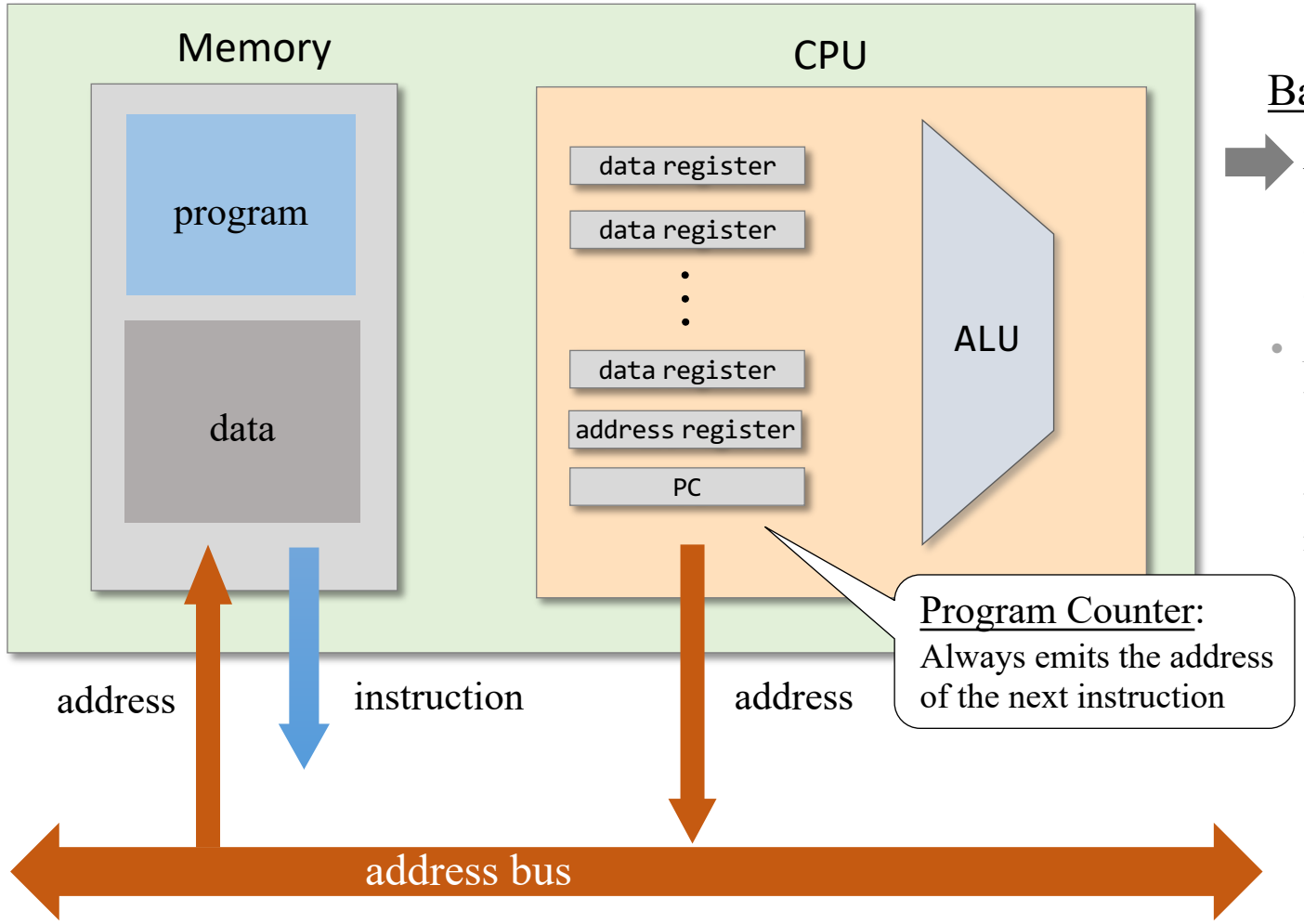


Basic loop:

- ***Fetch***
  an instruction
  (by supplying an address)
- ***Execute***
  the instruction

  (And… figure out the address of the next instruction)

Fetch an instruction



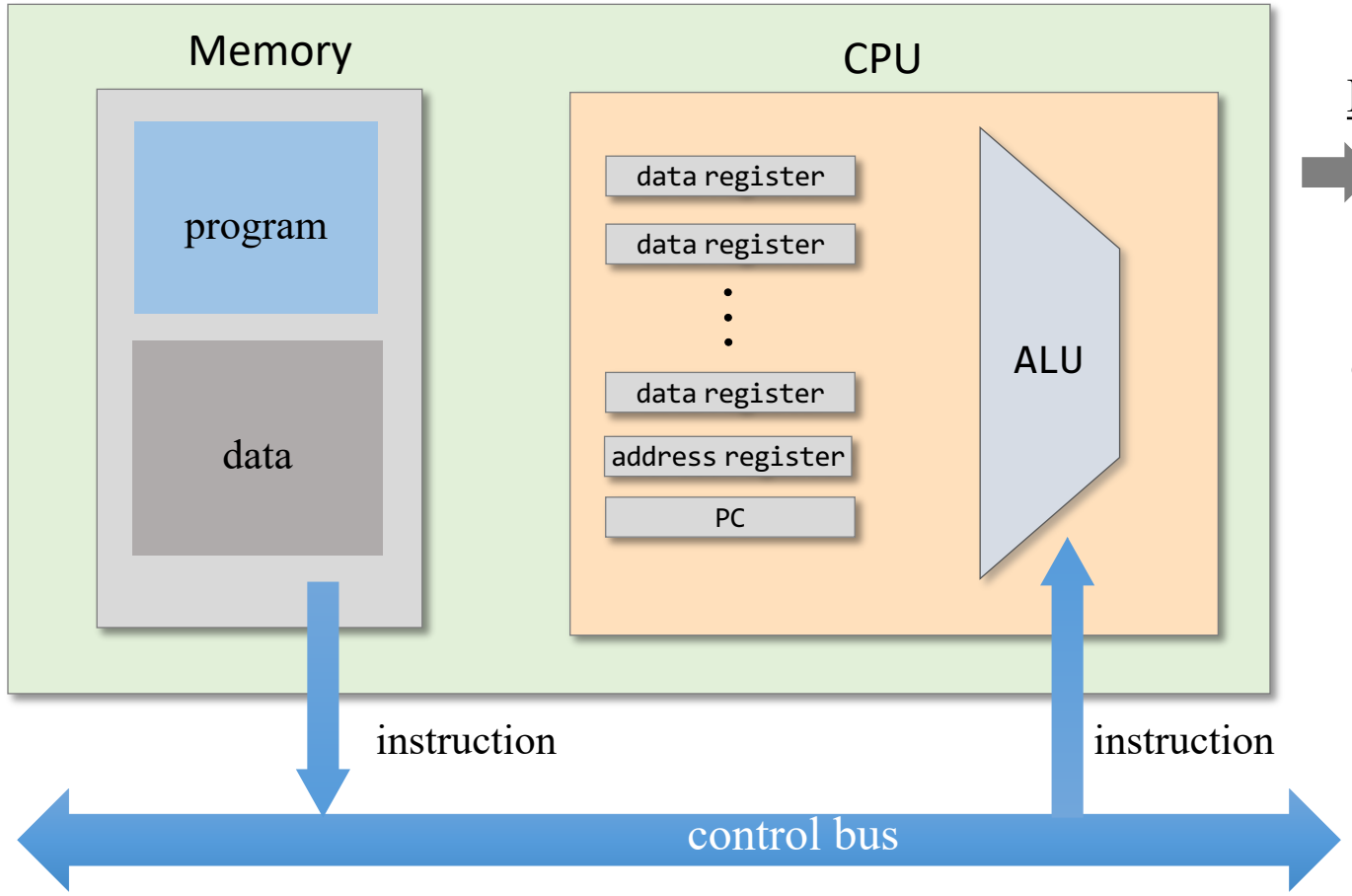
Program Counter:
Always emits the address of the next instruction

Basic loop:

- ***Fetch*** an instruction (by supplying an address)
- *Execute* the instruction

(And… figure out the address of the next instruction)

Fetch an instruction

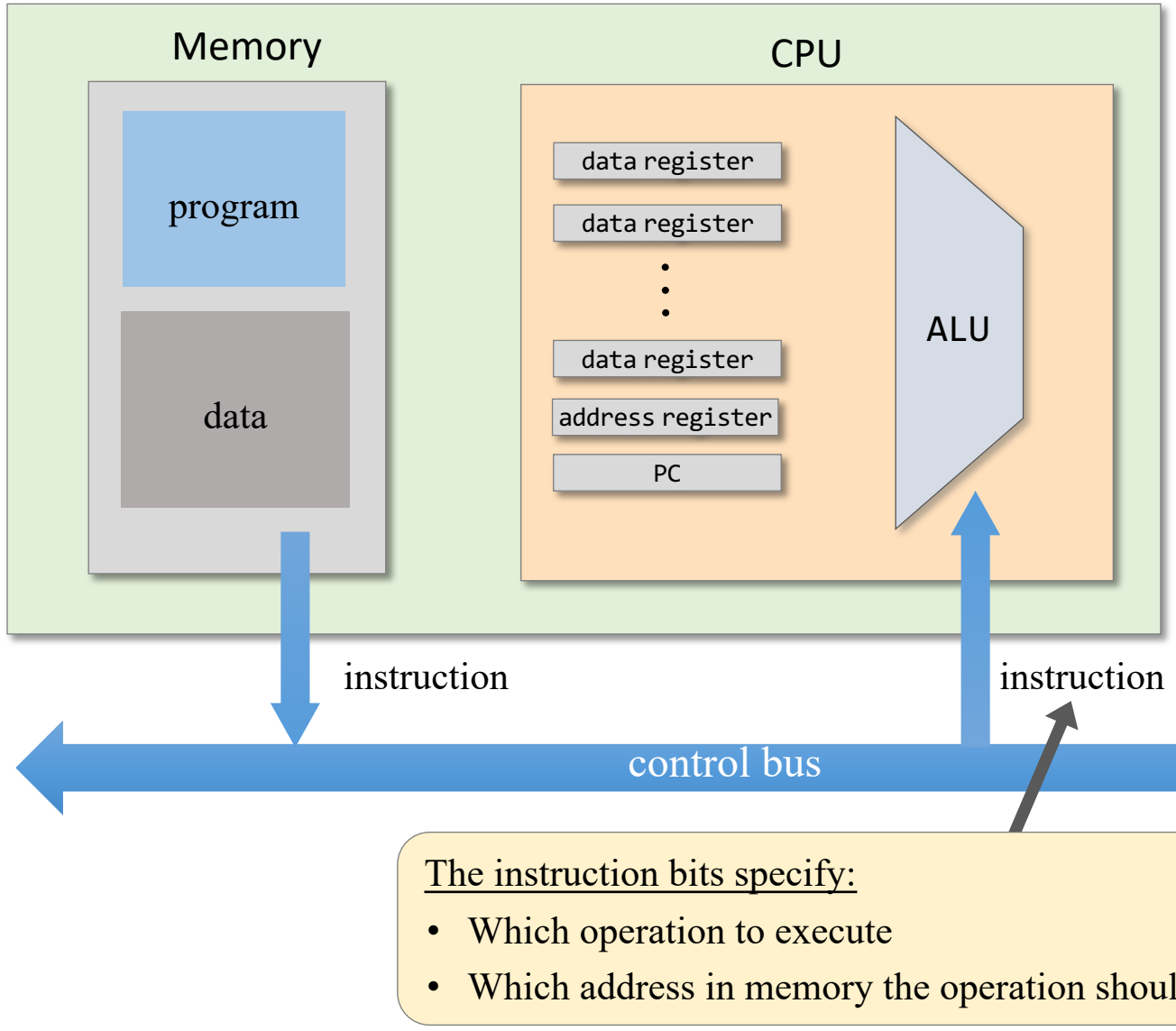


Basic loop:

- ***Fetch*** an instruction (by supplying an address)
- *Execute* the instruction

(And… figure out the address of the next instruction)

Execute the instruction



Basic loop:

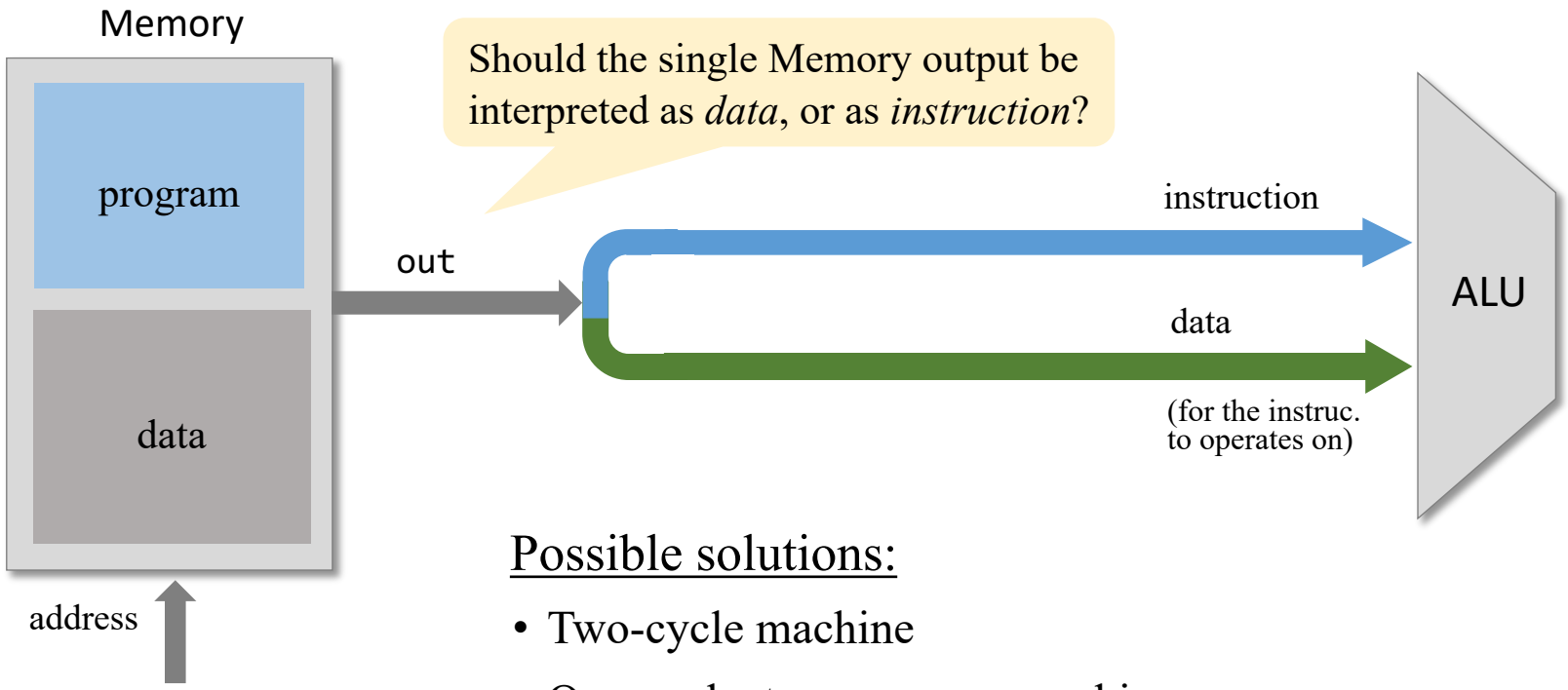
- *Fetch* an instruction (by supplying an address)
- ***Execute*** the instruction

(And… figure out the address of the next instruction)

The instruction bits specify:

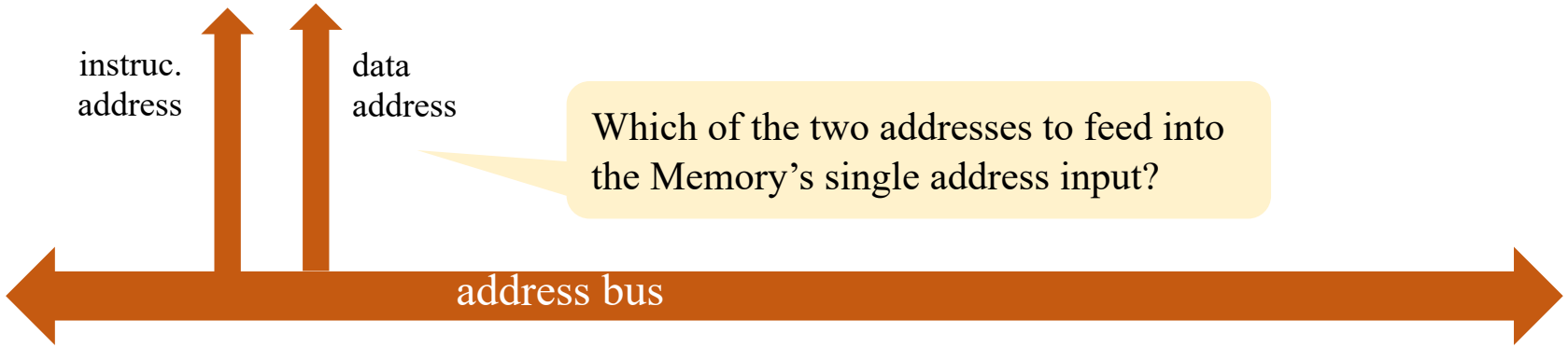
- Which operation to execute
- Which address in memory the operation should act on

Fetch – execute issues

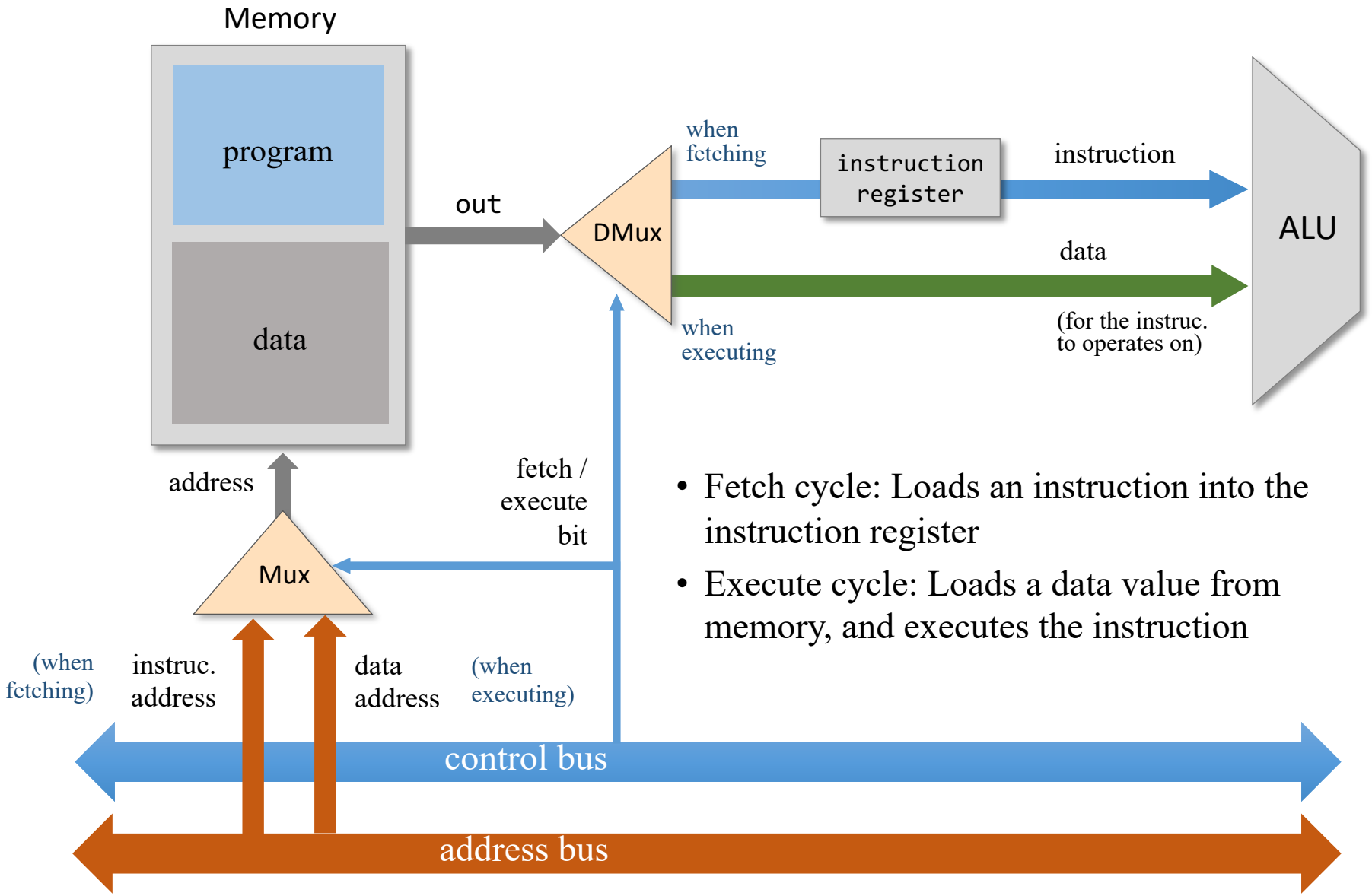


Possible solutions:

- Two-cycle machine
- One-cycle, two-memory machine

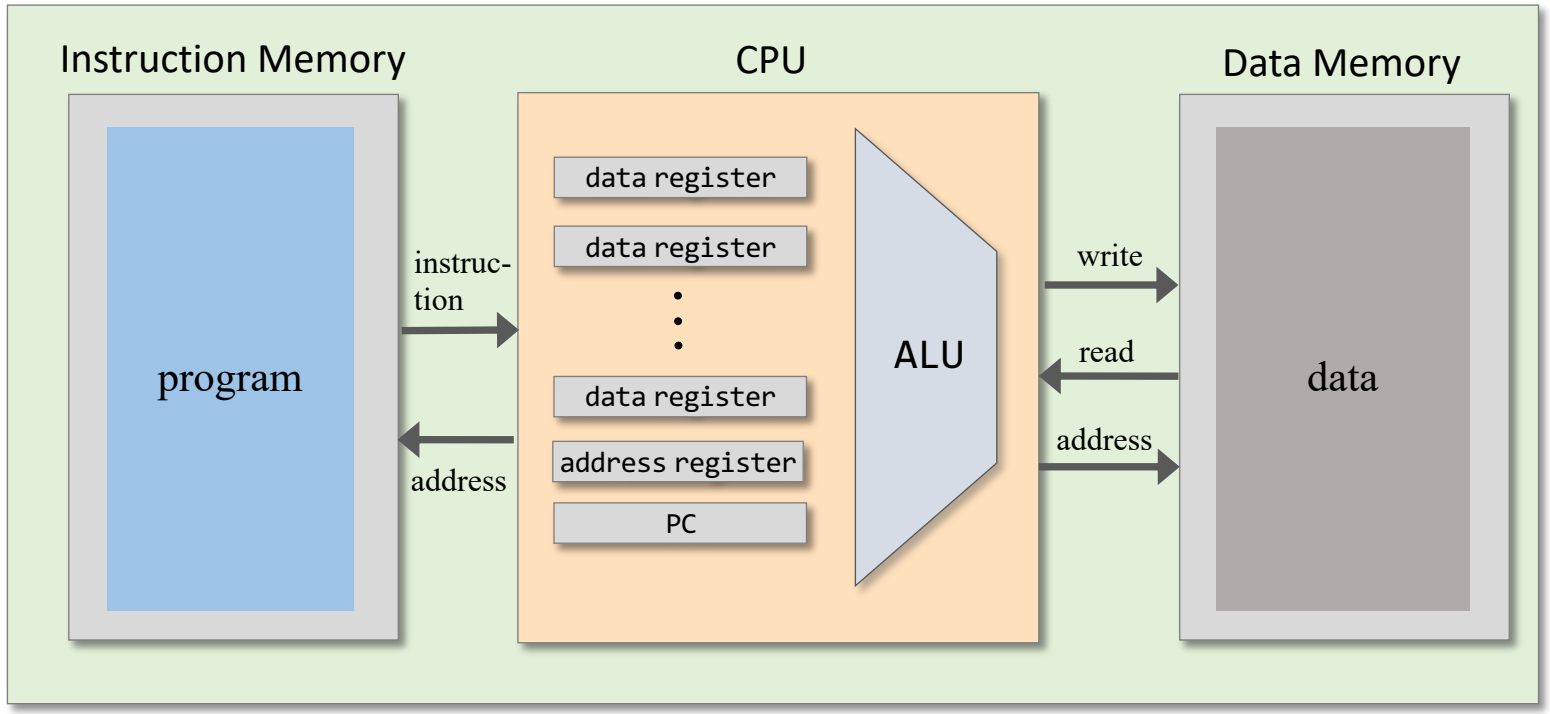


Two-cycle machine



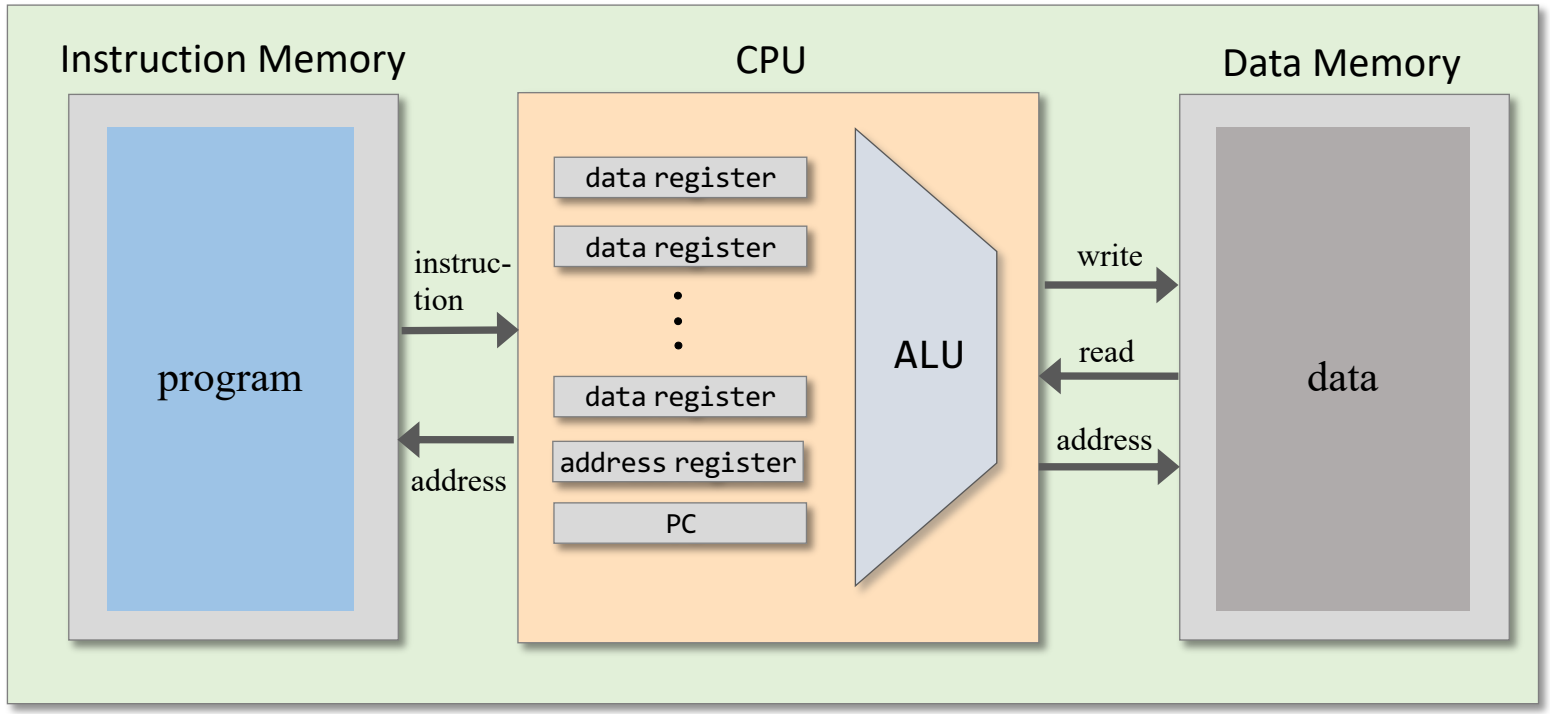
- Fetch cycle: Loads an instruction into the instruction register
- Execute cycle: Loads a data value from memory, and executes the instruction

Single cycle, two-memory machine



- Program and data are stored in two separate physical memories
- Both memories are accessed simultaneously, in the same cycle
  (Sometimes called ”Harvard architecture”)

Single cycle, two-memory machine



Advantages

- Simpler architecture
- Faster processing

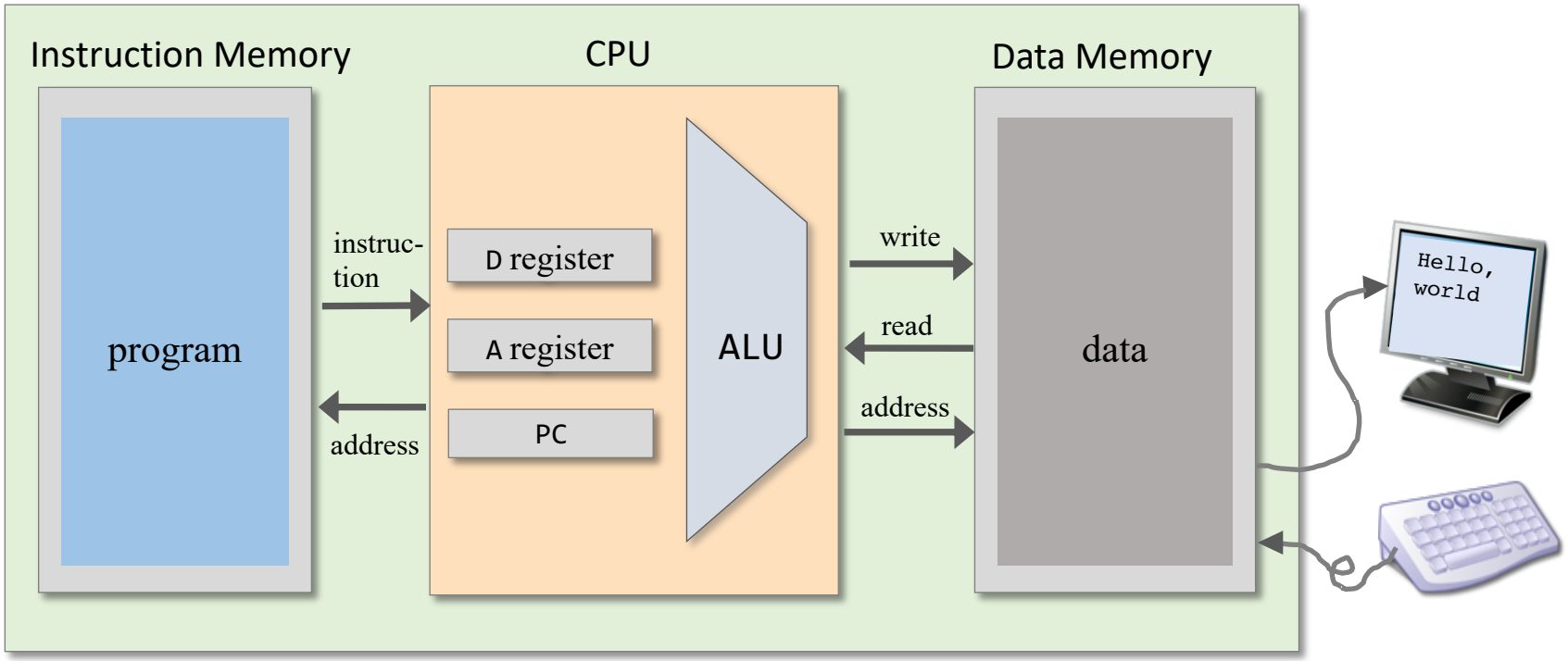
Disadvantages

- Two memory chips
- Separate address spaces

Chapter 5: Computer Architecture

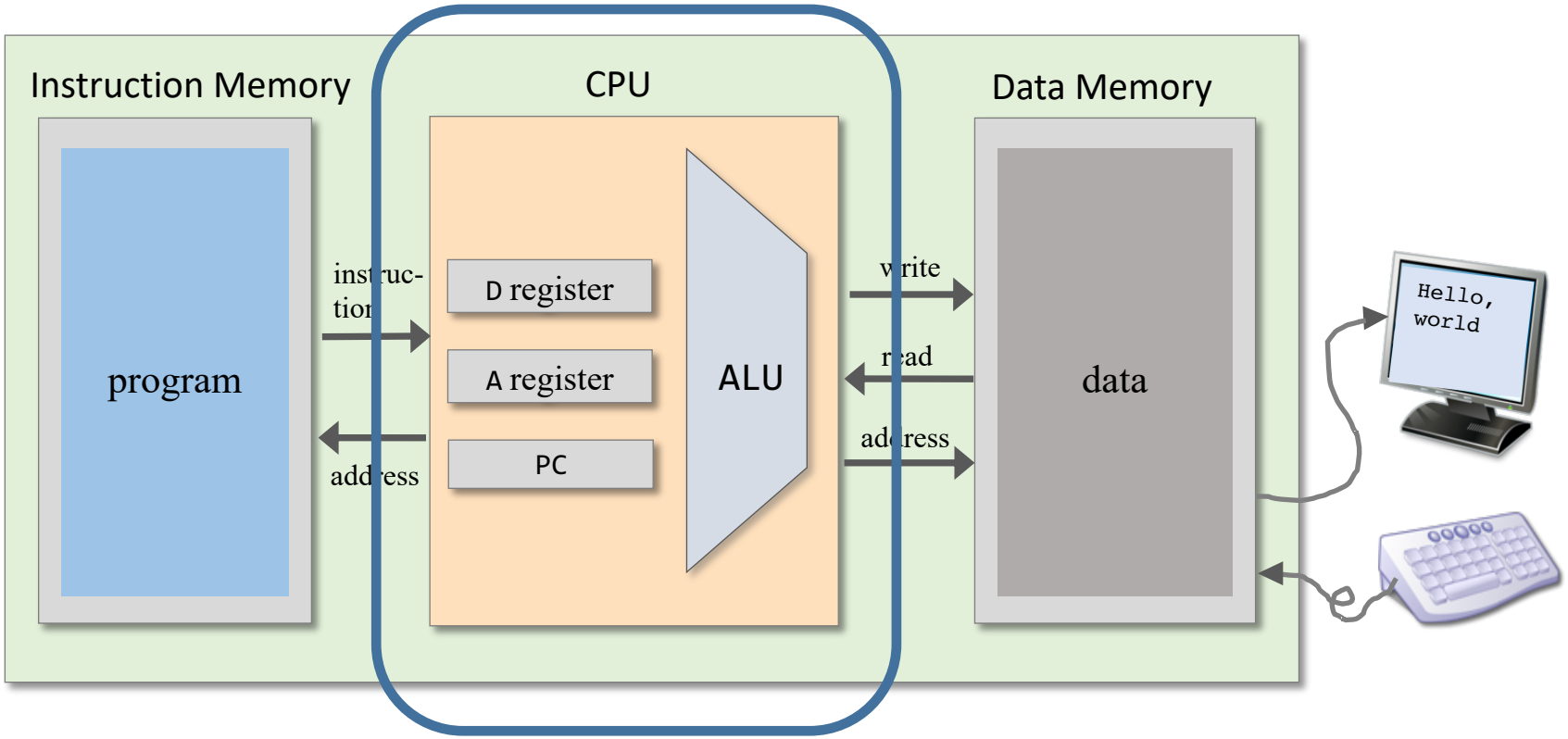
- ✓ Basic architecture
- ✓ Fetch-Execute cycle
- → The Hack CPU
  - Input / output
- Memory
- Computer
- Project 5: Chips
- Project 5: Guidelines

Hack computer

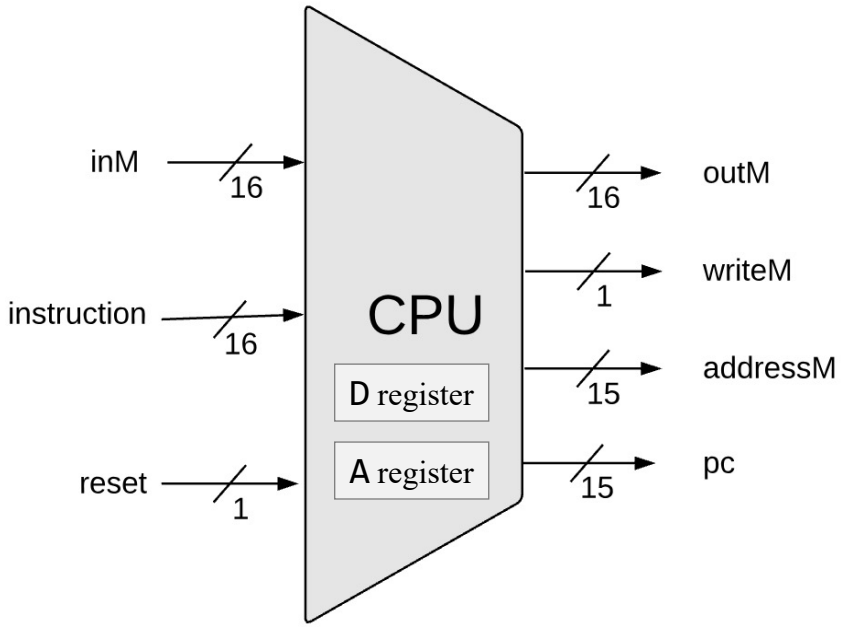


- Single cycle computer
- Two separate memory units

Hack computer



CPU abstraction



Instruction examples:

```
// D = RAM[5] + 1
@5
D=M+1
...
// RAM[3] = D
@3
M=D
...
```

CPU Abstraction:

Executes instructions written in the Hack machine language

CPU abstraction

Hack instructions (formal specification)

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` *vvvvvvvvvvvvvvv* (*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: `111`*accccccdddjjj*

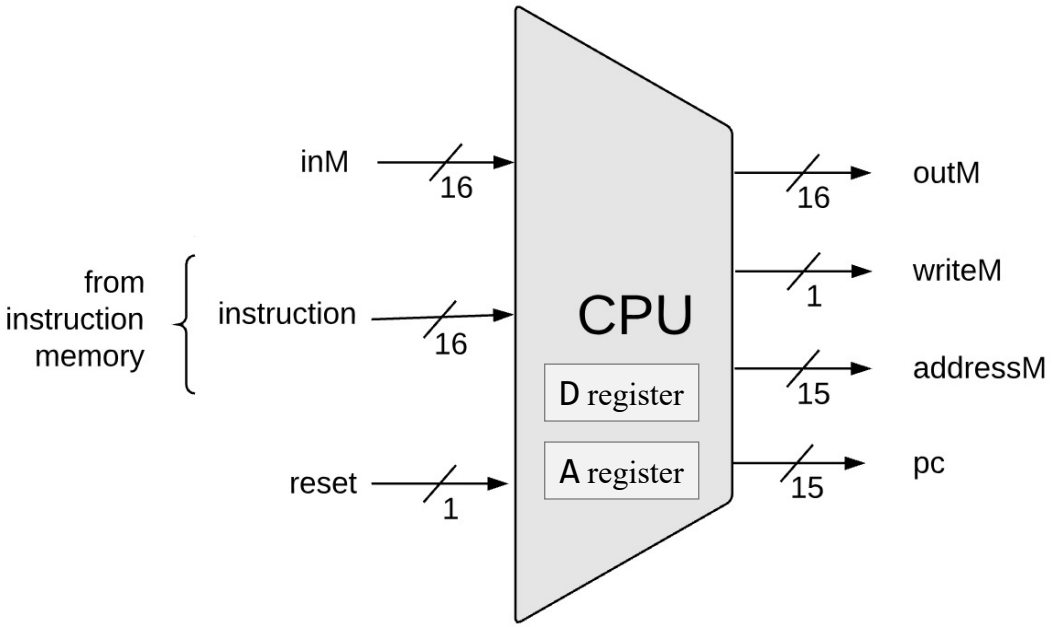
comp		*c*	*c*	*c*	*c*	*c*	*c*
0		1	0	1	0	1	0
1		1	1	1	1	1	1
-1		1	1	1	0	1	0
D		0	0	1	1	0	0
A	M	1	1	0	0	0	0
!D		0	0	1	1	0	1
!A	!M	1	1	0	0	0	1
-D		0	0	1	1	1	1
-A	-M	1	1	0	0	1	1
D+1		0	1	1	1	1	1
A+1	M+1	1	1	0	1	1	1
D-1		0	0	1	1	1	0
A-1	M-1	1	1	0	0	1	0
D+A	D+M	0	0	0	0	1	0
D-A	D-M	0	1	0	0	1	1
A-D	M-D	0	0	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

dest	*d*	*d*	*d*	Effect: store *comp* in:
null	0	0	0	the value is not stored
M	0	0	1	RAM[A]
D	0	1	0	D register (reg)
DM	0	1	1	RAM[A] and D reg
A	1	0	0	A reg
AM	1	0	1	A reg and RAM[A]
AD	1	1	0	A reg and D reg
ADM	1	1	1	A reg, D reg, and RAM[A]

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

CPU abstraction



Instruction examples:

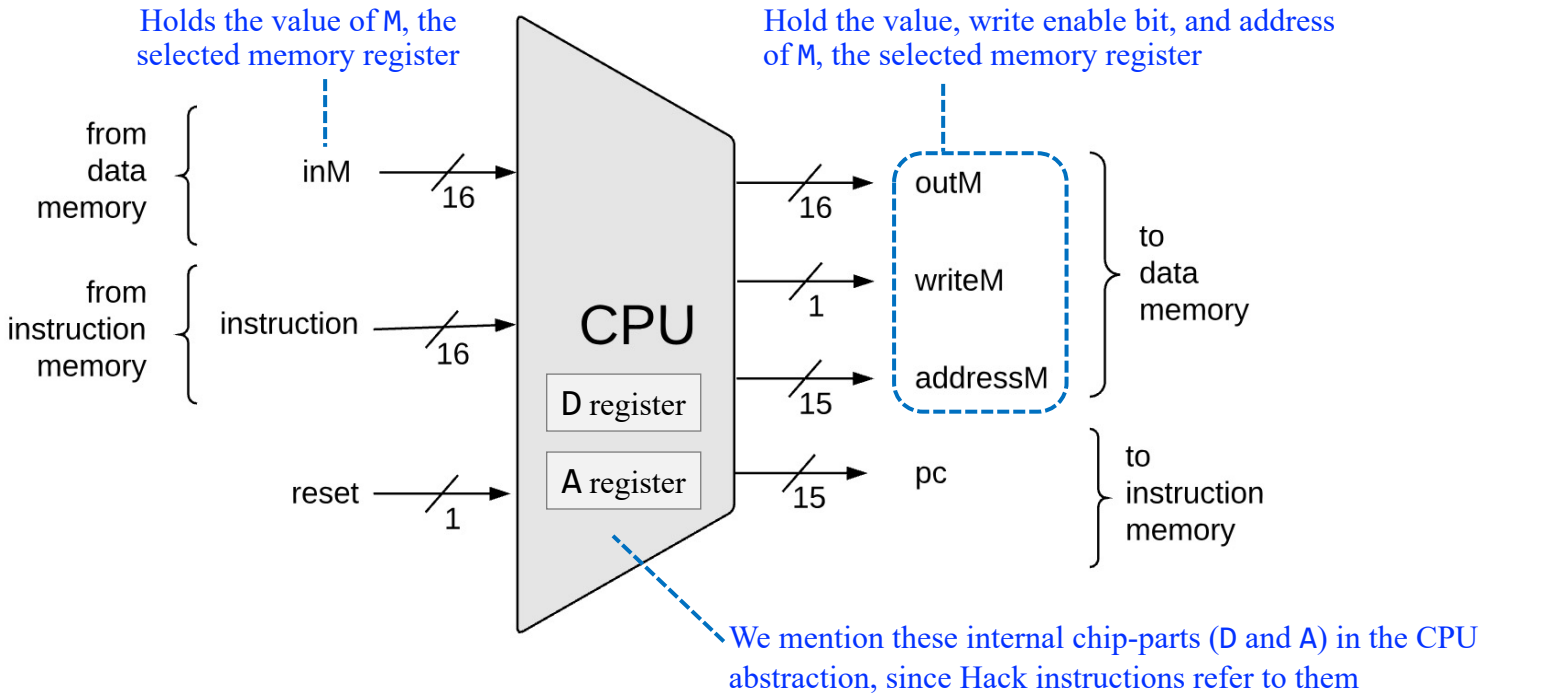
```
// D = RAM[5] + 1
@5
D=M+1
...
// RAM[3] = D
@3
M=D
...
```

CPU Abstraction:

Executes instructions written in the Hack machine language

Note that the selected memory register `M` can be input, or output, or both input and output (example: `M=M+1`)

CPU abstraction



Instruction examples:

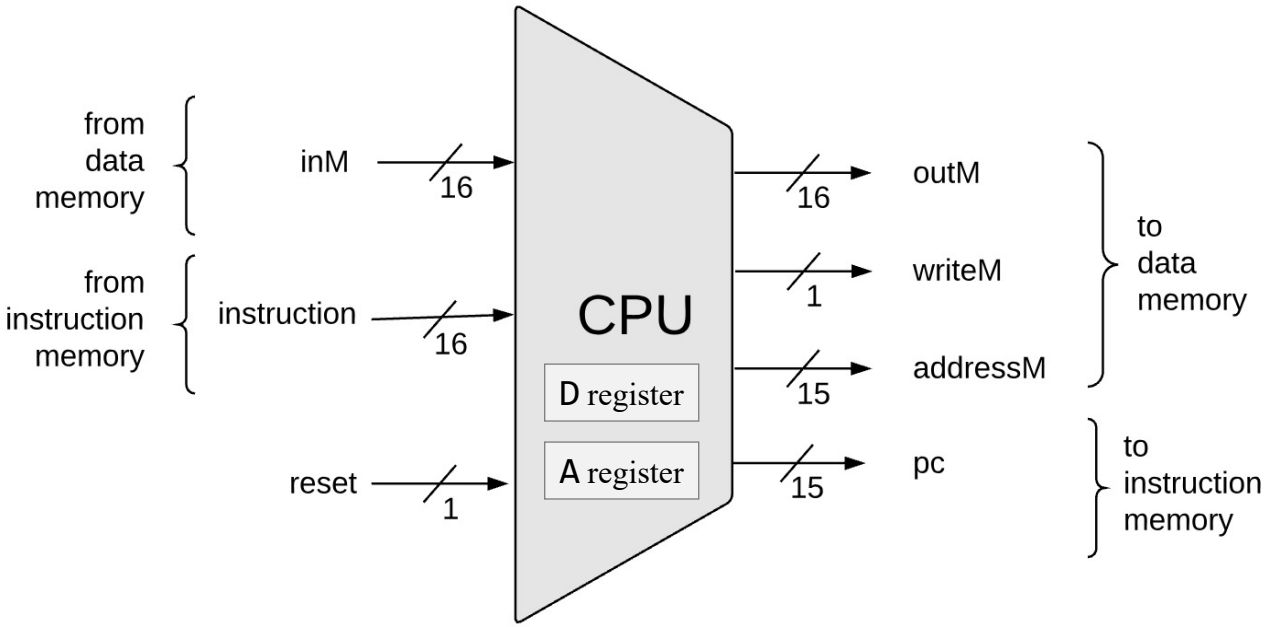
```
// D = RAM[5] + 1
@5
D=M+1
...
// RAM[3] = D
@3
M=D
...
```

CPU Abstraction:

Executes instructions written in the Hack machine language

Note that the selected memory register `M` can be input, or output, or both input and output (example: `M=M+1`)

CPU abstraction



Instruction examples:

```
// D = RAM[5] + 1
@5
D=M+1
…
// RAM[3] = D
@3
M=D
…
```

CPU operation:

1. Executes the instruction:

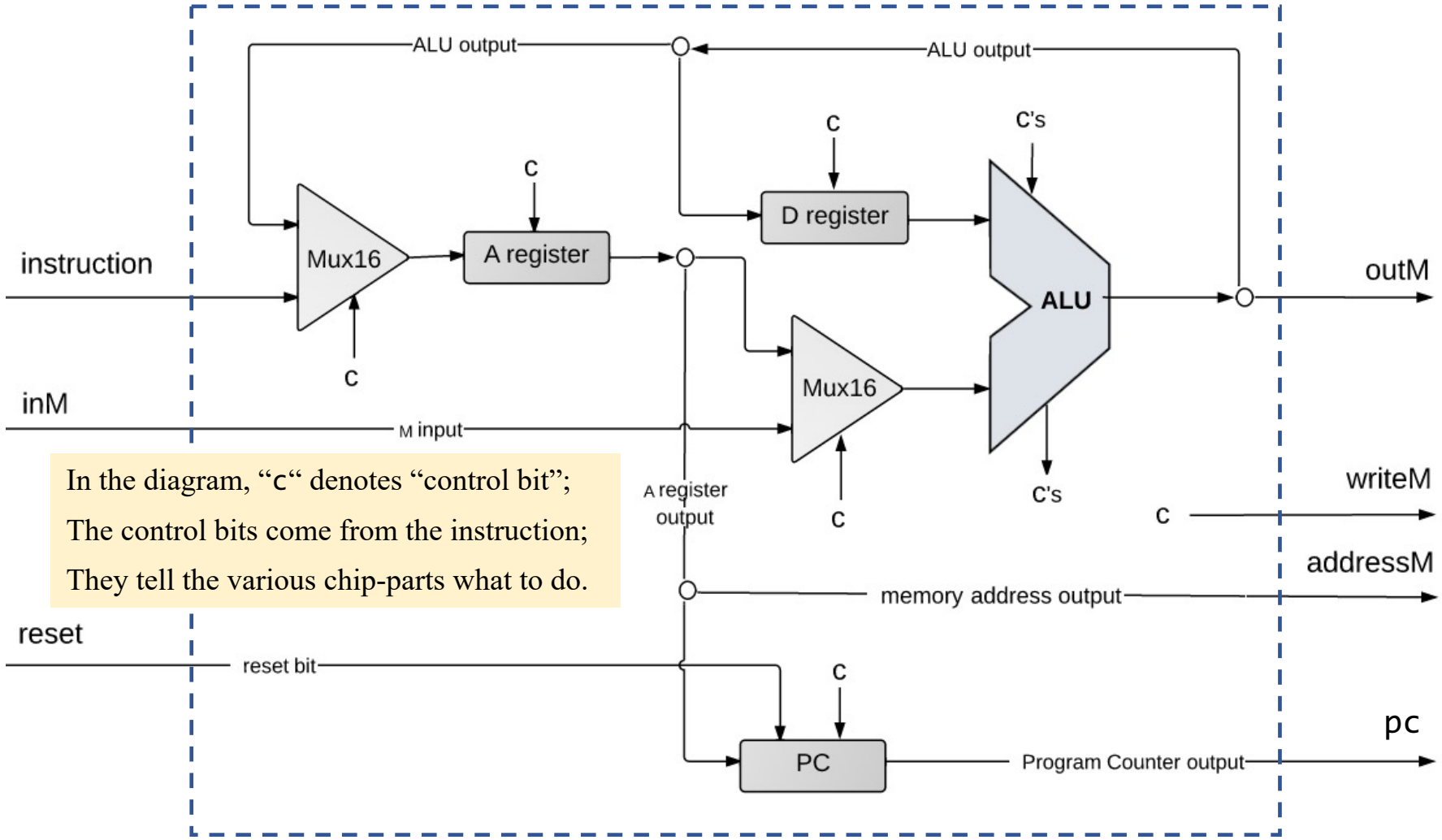
   If it's an A instruction (`@`*xxx*), sets the `A` register to *xxx*

   Else:

   - If the instruction uses `M` as input, gets this value from `inM`
   - Computes the ALU function specified by the instruction
   - If the instruction writes to `M`, puts the ALU output in `outM`, puts the register's address in `addressM`, and asserts the `writeM` bit

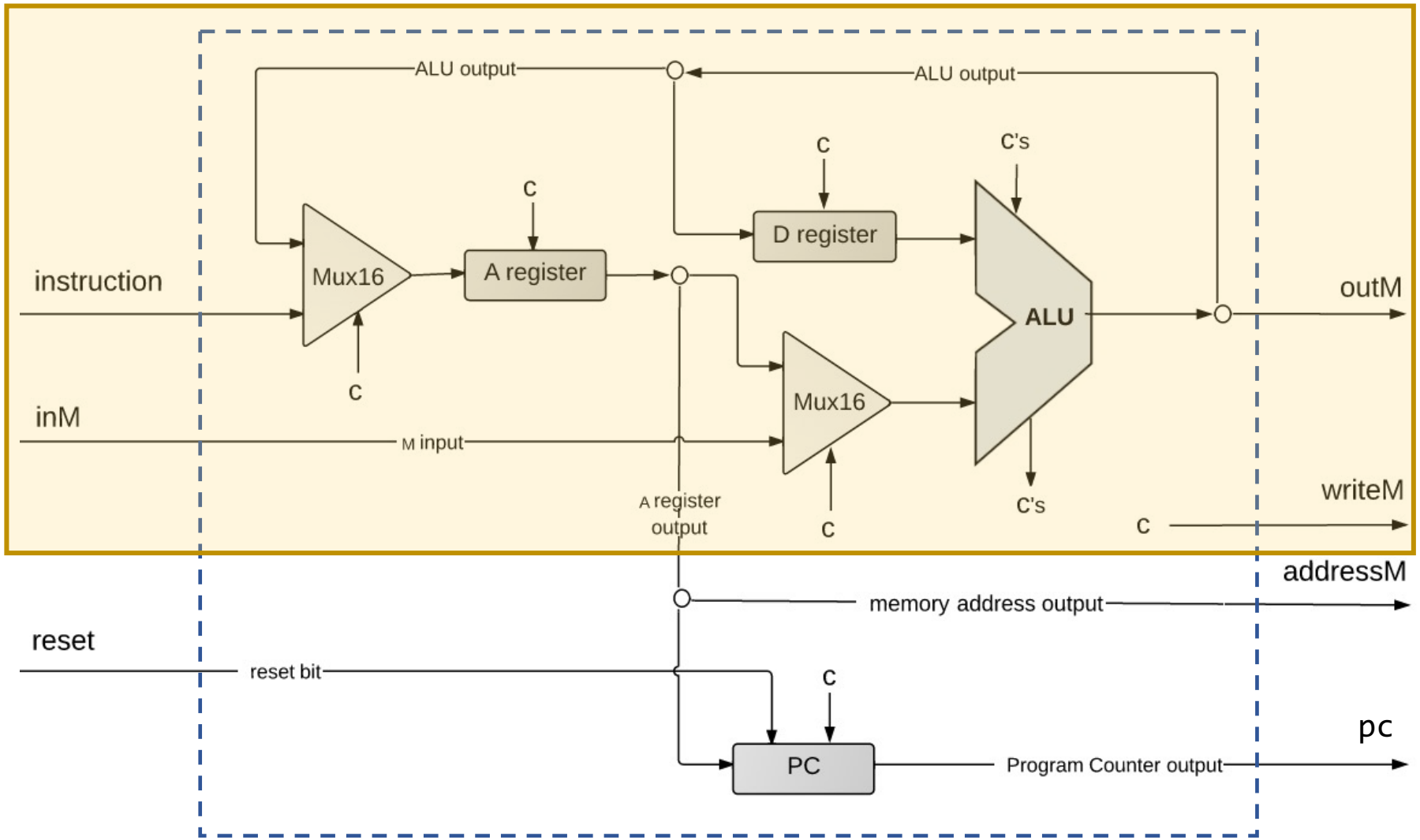
2. Figures out the address of the next instruction, and puts it in `pc`.

CPU implementation

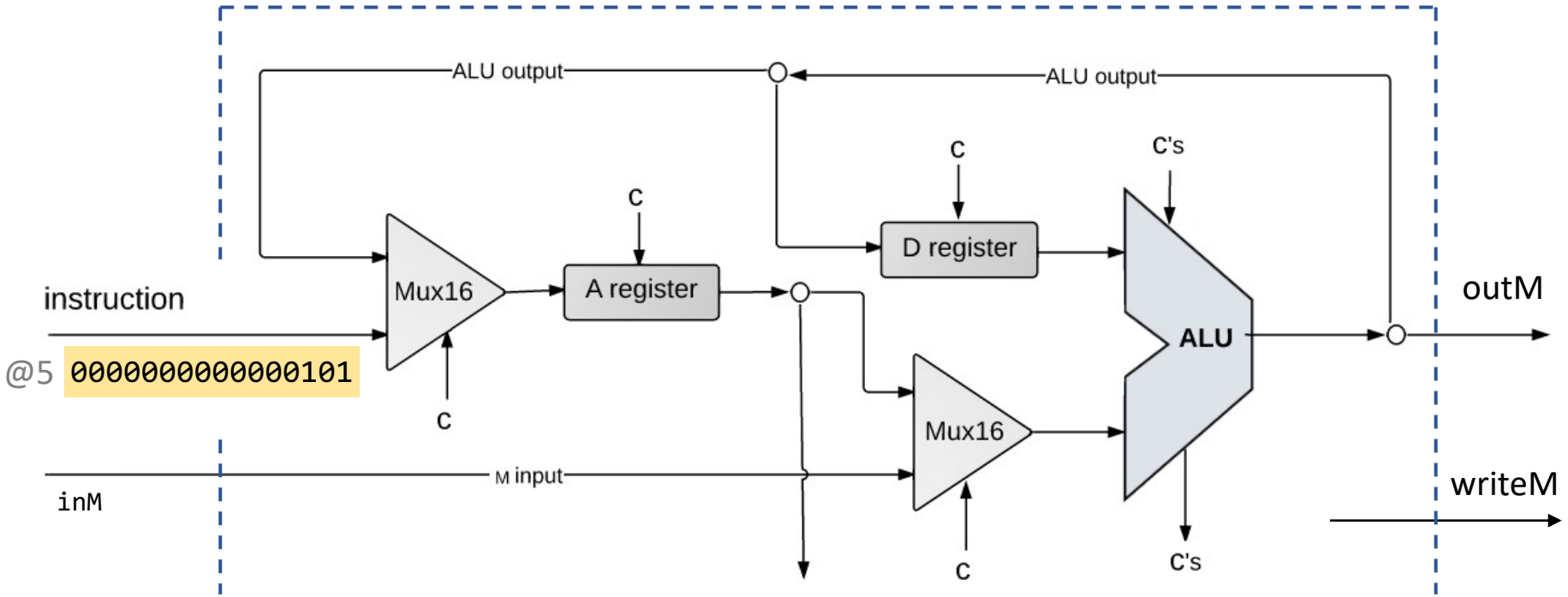


In the diagram, "c" denotes "control bit";
The control bits come from the instruction;
They tell the various chip-parts what to do.

CPU implementation



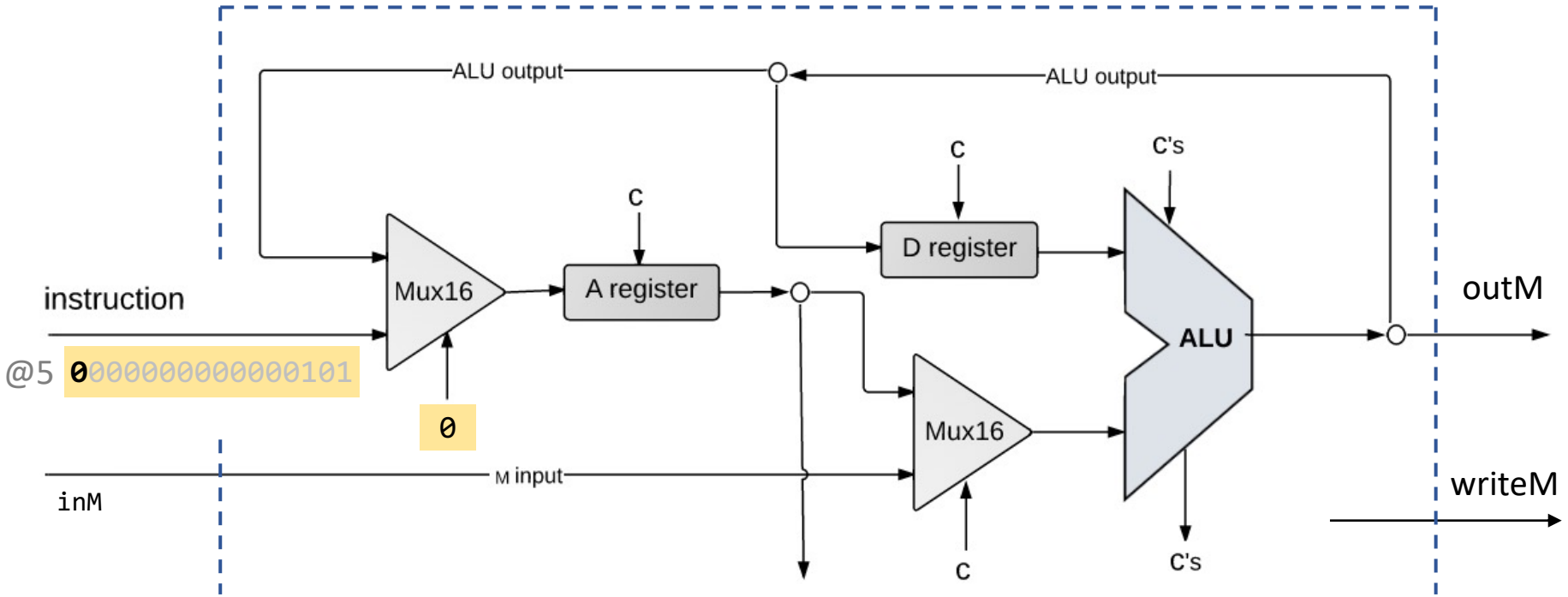
CPU implementation: Instruction handling



Handling A-instructions

Routes the instruction's MSB (op-code) to the Mux16 control bit

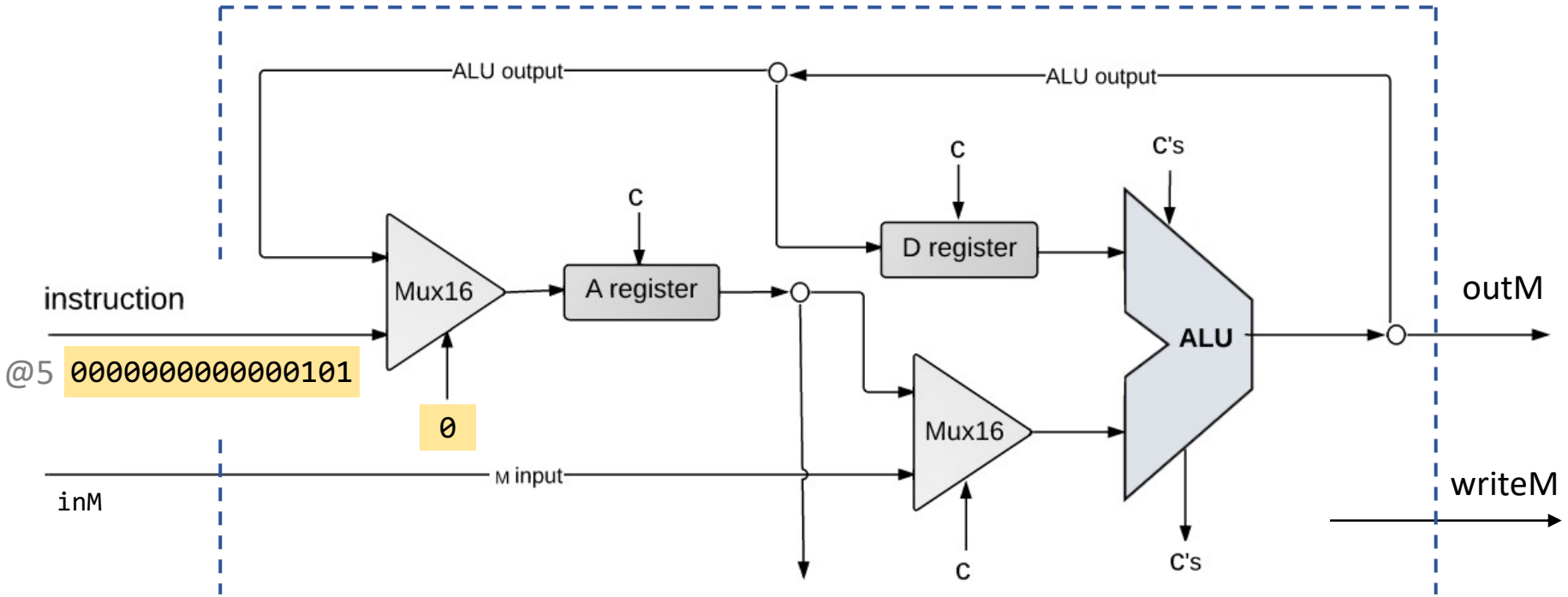
CPU implementation: Instruction handling



Handling A-instructions

Routes the instruction's MSB (op-code) to the Mux16 control bit

CPU implementation: Instruction handling



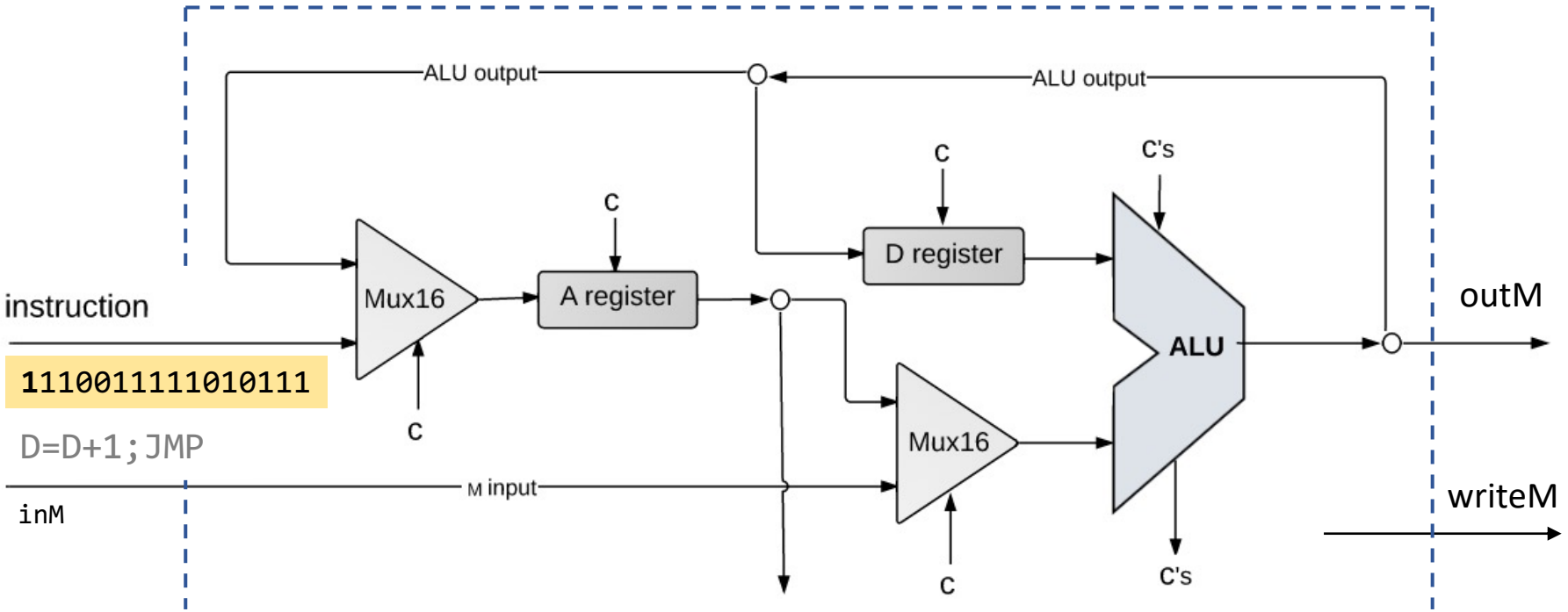
Handling A-instructions

Routes the instruction's MSB (op-code) to the Mux16 control bit

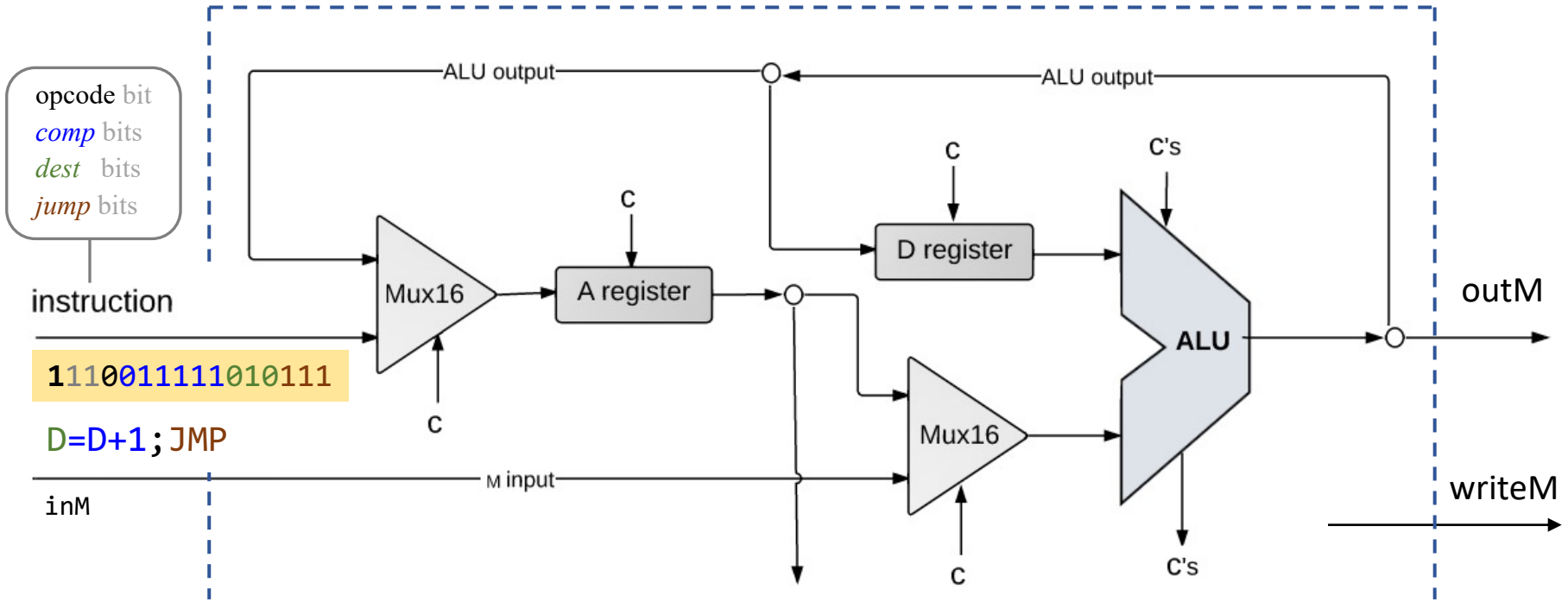
Result: A-register ← instruction (value)

(Exactly what the @*xxx* instruction specifies: "set A to *xxx*")

CPU implementation: Instruction handling



CPU implementation: Instruction handling



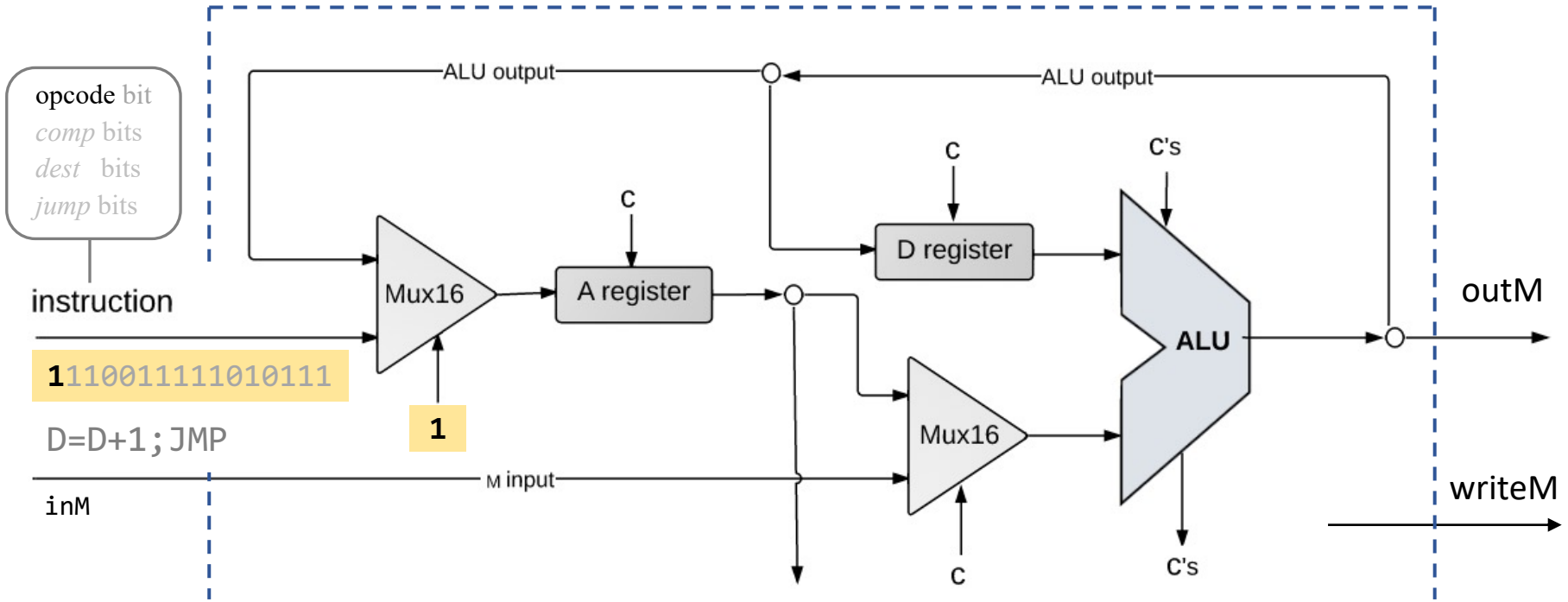
Handling C-instructions

Each instruction field (*opcode*, *comp* bits, *dest* bits, 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

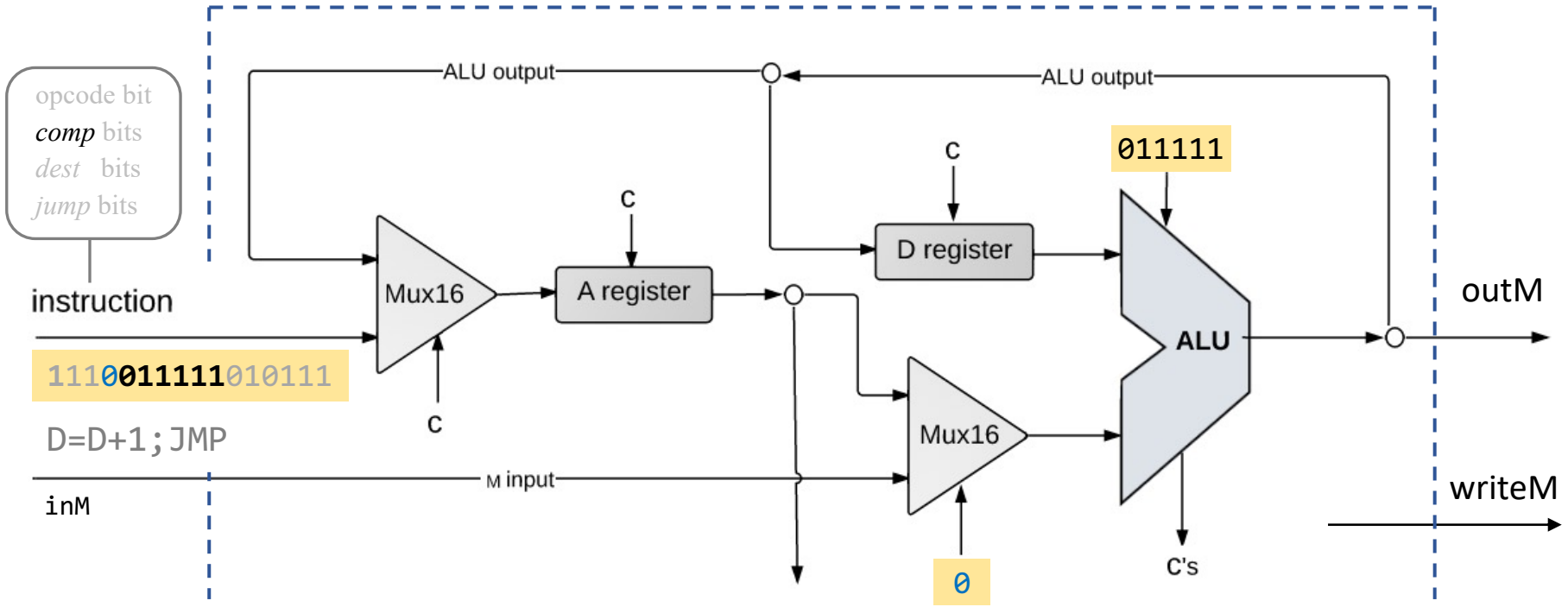


Handling C-instructions (the *opcode* bit):

Routes the instruction's MSB to the Mux16

Result: Prepares the A register to get the ALU output.

CPU implementation: Instruction handling



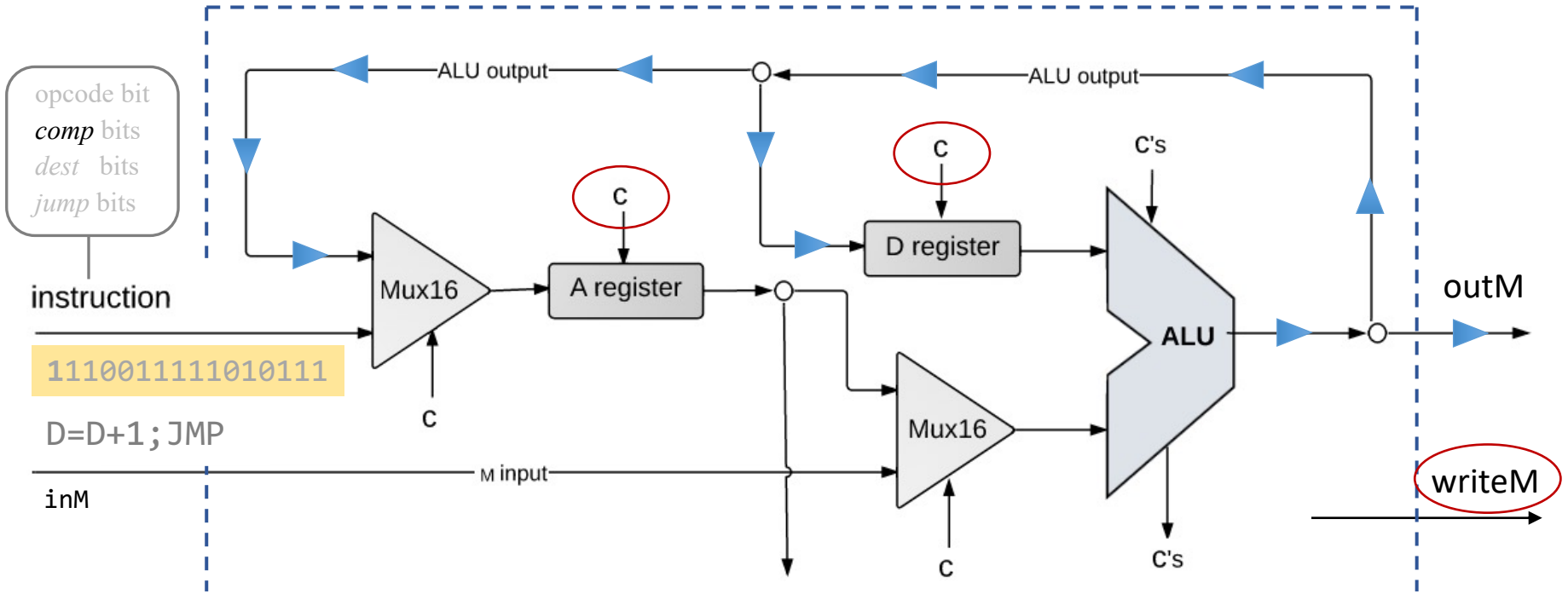
Handling C-instructions (the *comp*utation bits)

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

Result: the ALU computes the specified function

which becomes the ALU output

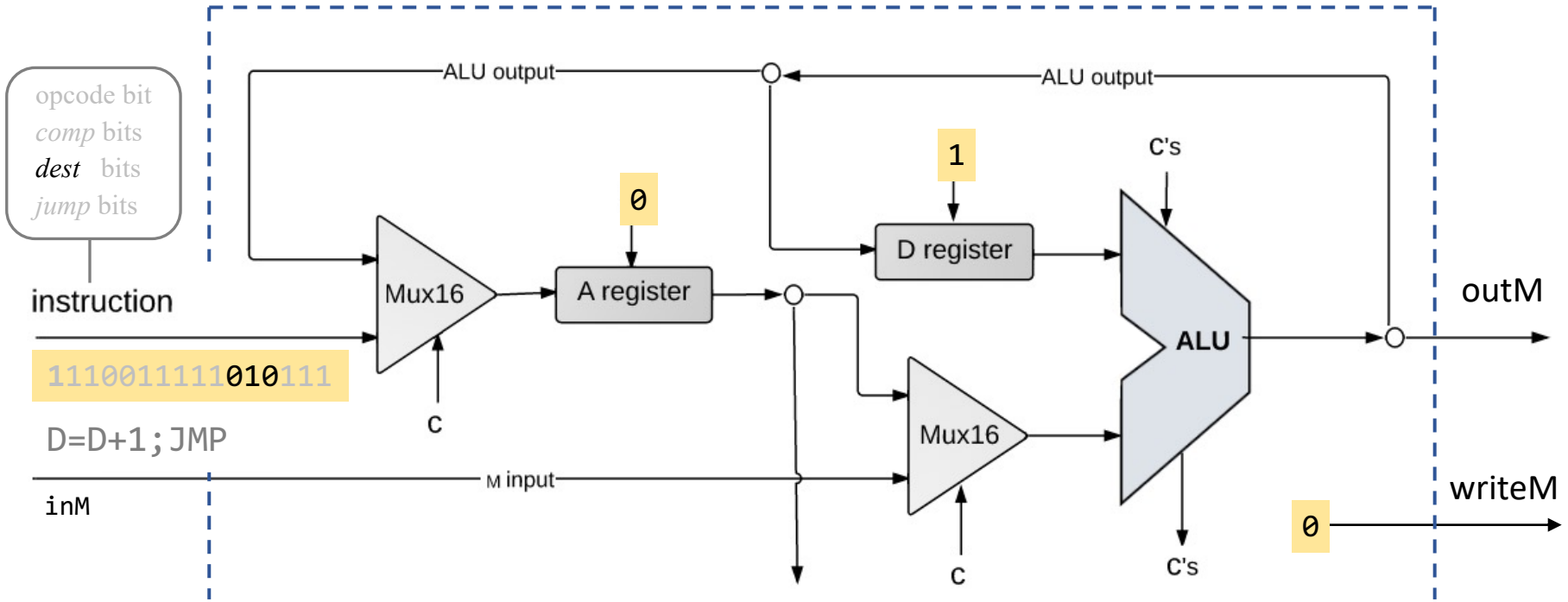
CPU implementation: Instruction handling



ALU output:

- Result of ALU calculation
- Fed simultaneously to D-register, A-register, data memory
- All enabled/disabled by control bits

CPU implementation: Instruction handling

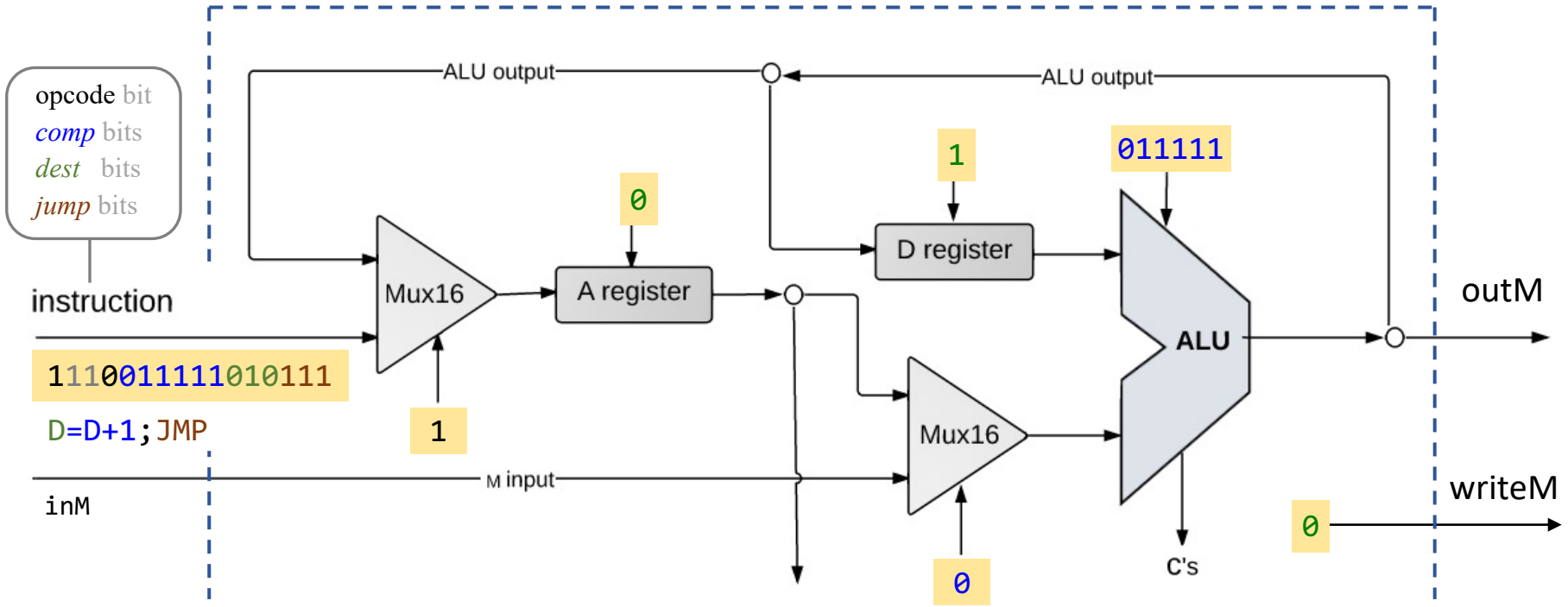


Handling C-instructions (the *dest*ination bits)

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

Result: Only the enabled destinations get the ALU output

CPU implementation: Instruction handling



Handling C-instructions (recap)

✓ Executes *dest* = *comp*

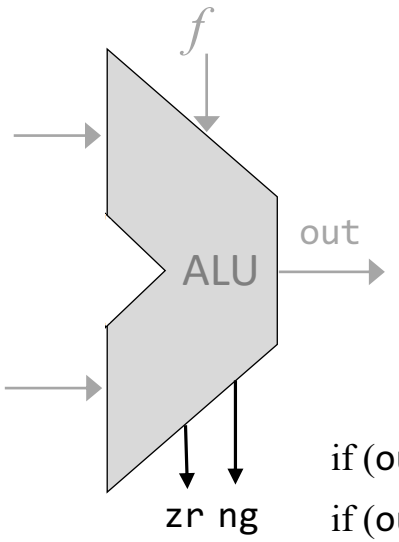
➡ Figures out which instruction to execute next

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`

jump	j1	j2	j3	*condition*
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



if (out == 0) zr = 1, else zr = 0

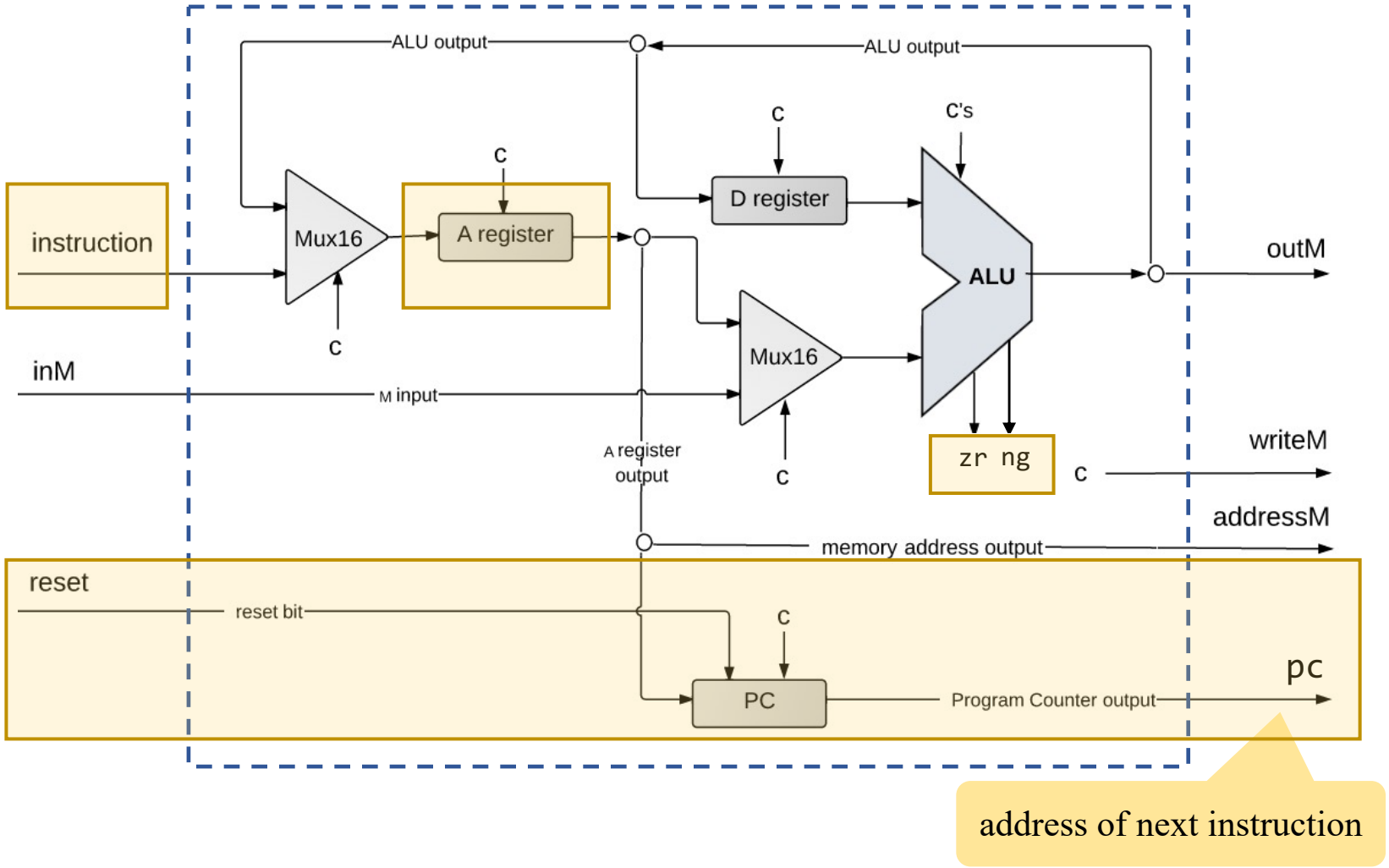
if (out < 0) ng = 1, else ng = 0

Jump decision:

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

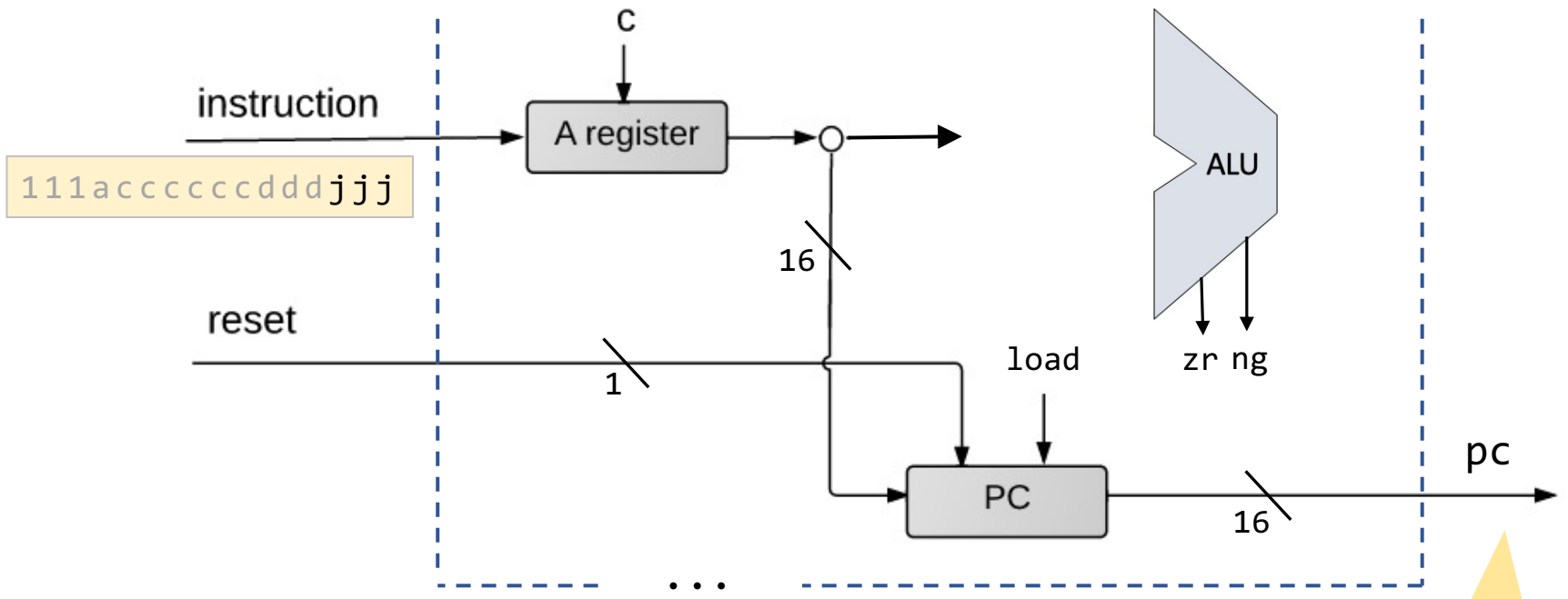
J can be computed using gate logic,
And then help compute the address of the next instruction

CPU implementation: Control



How to compute it?

CPU implementation: Control

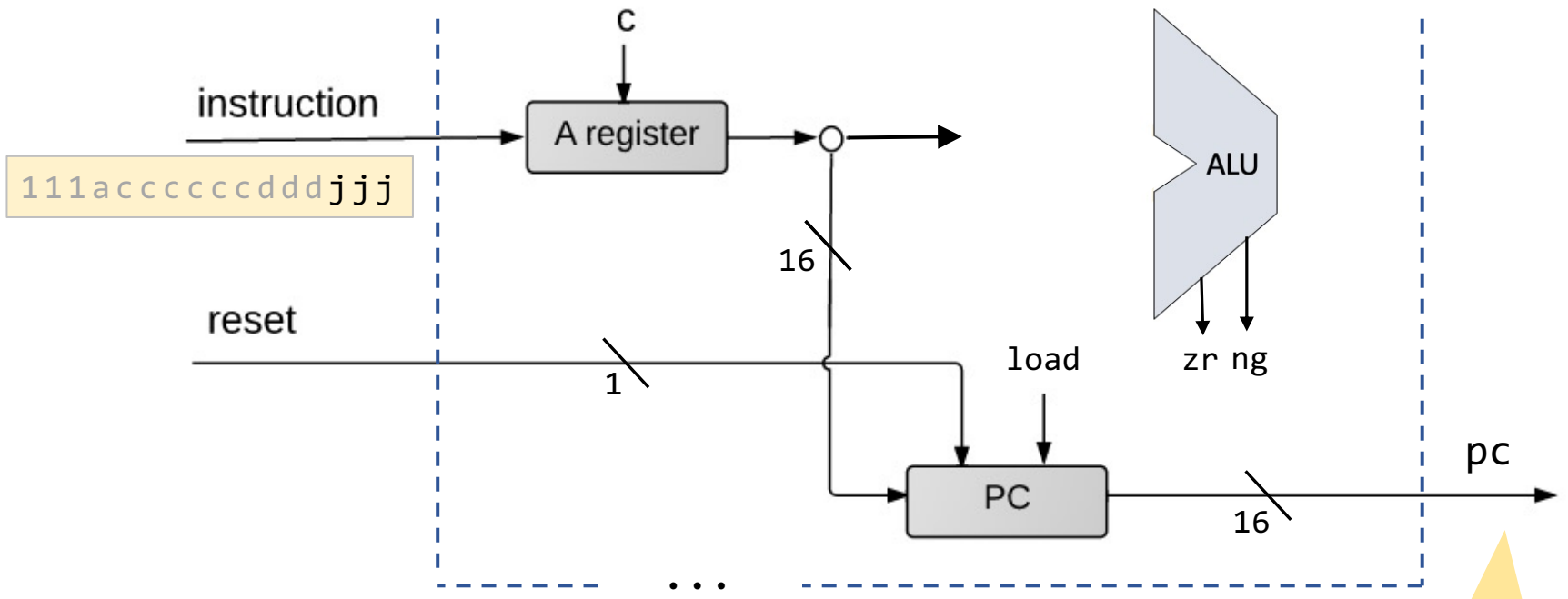


PC (program counter) abstraction:

Outputs the address of the next instruction:

- reset: `PC ← 0`
- no jump: `PC++`
- jump: if (*condition*) `PC ← A` `// A contains the address of the // jump destination`

CPU implementation: Control



PC (program counter) implementation:

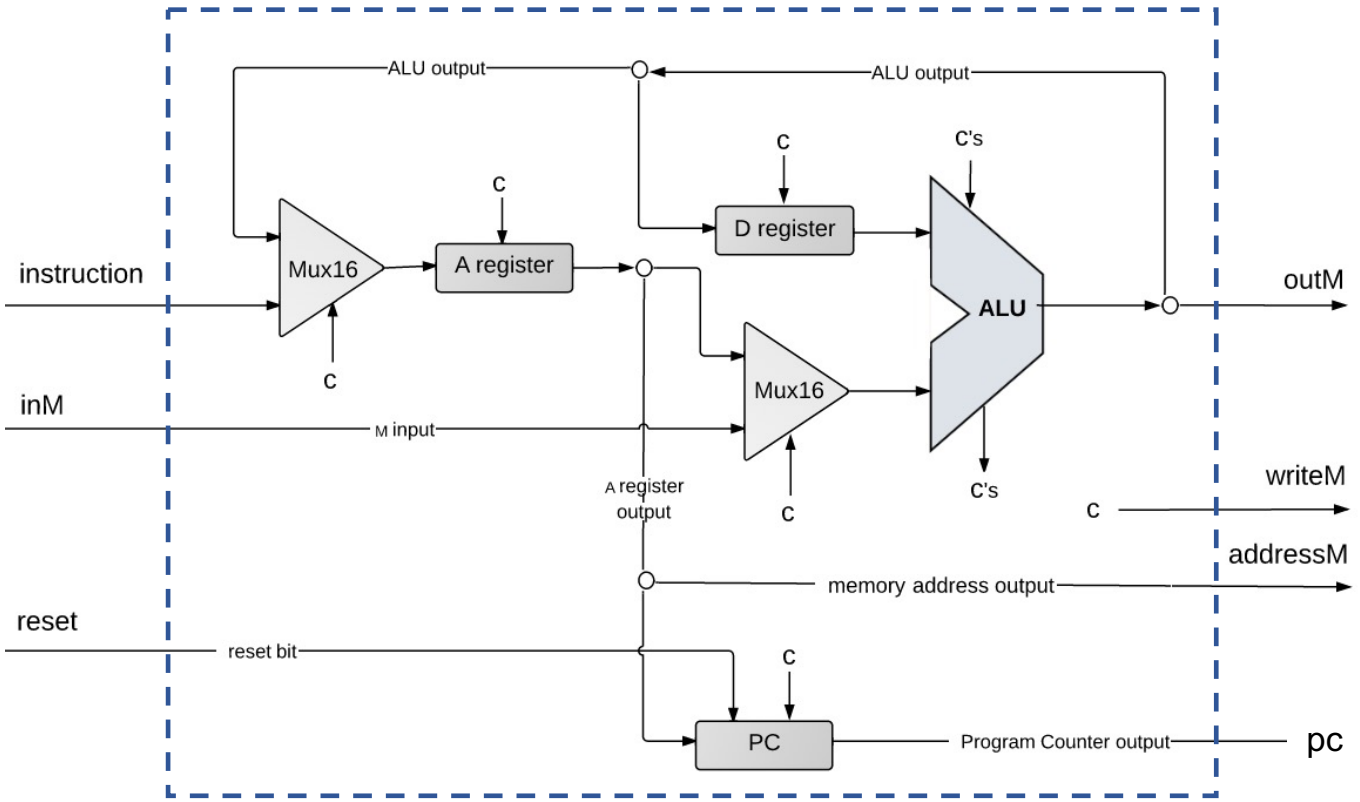
if (reset == 1) PC ← 0 // reset

else

if (*J*(jump bits, zr, ng) == 1) PC ← A // jump

else PC++ // next instruction

CPU implementation



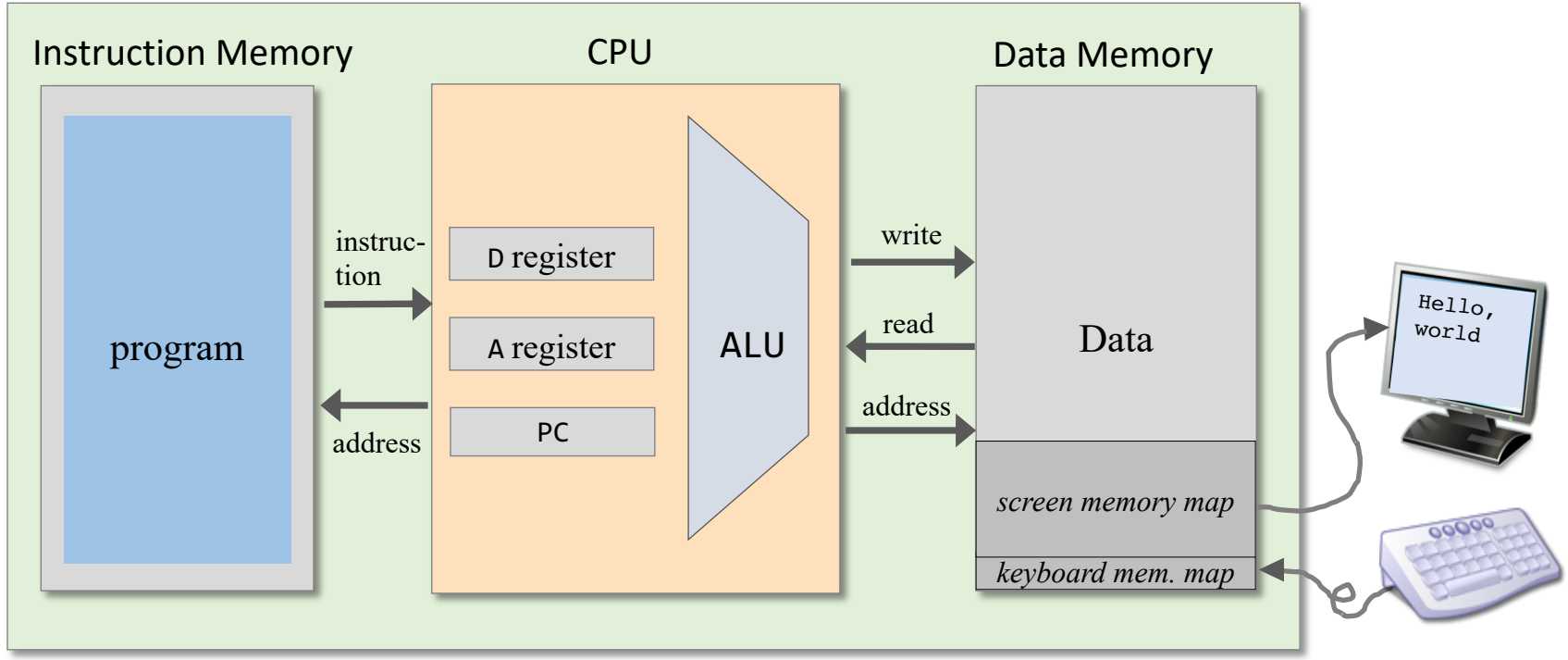
✓ Executes the current instruction

✓ Figures out which instruction to execute next.

Chapter 5: Computer Architecture

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

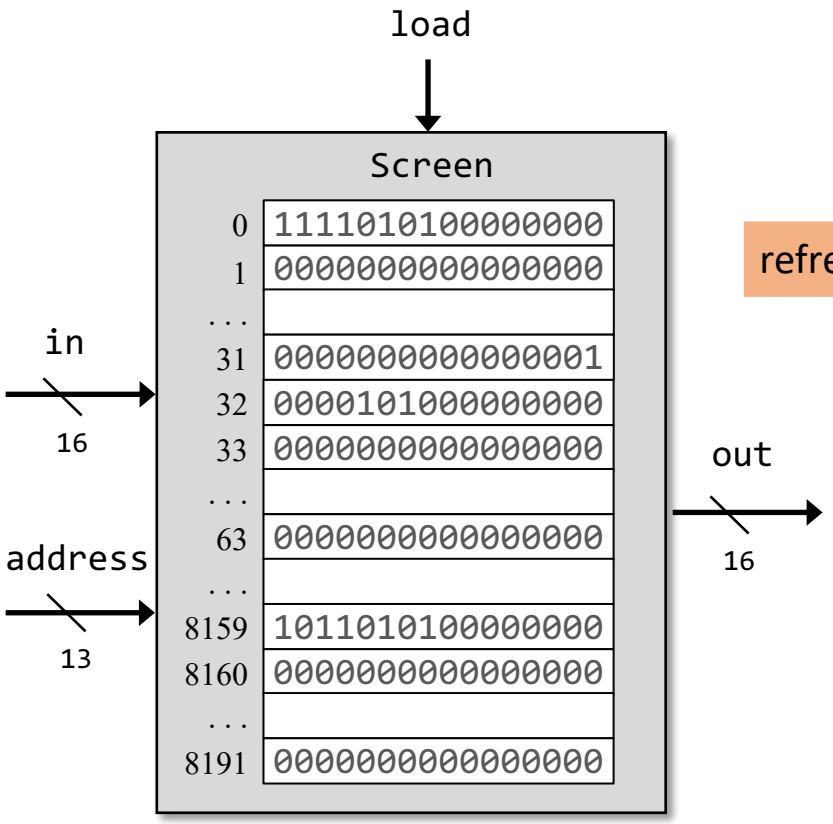
Hack computer



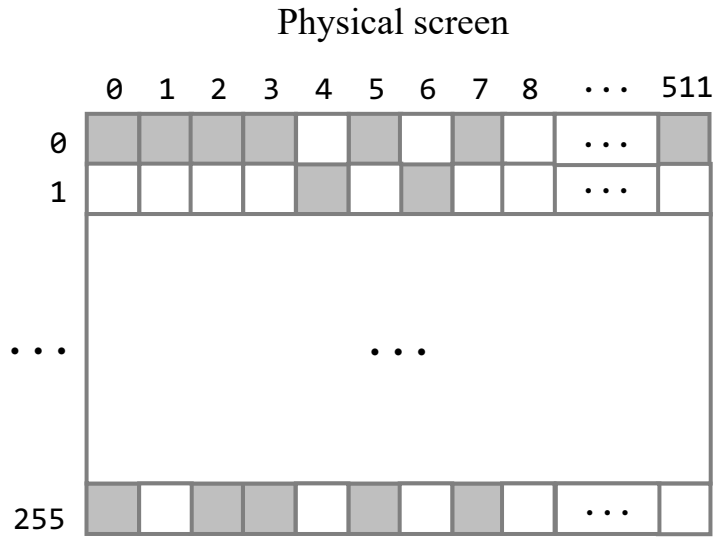
I/O devices

- Screen (black and white)
- Keyboard (regular)

Screen



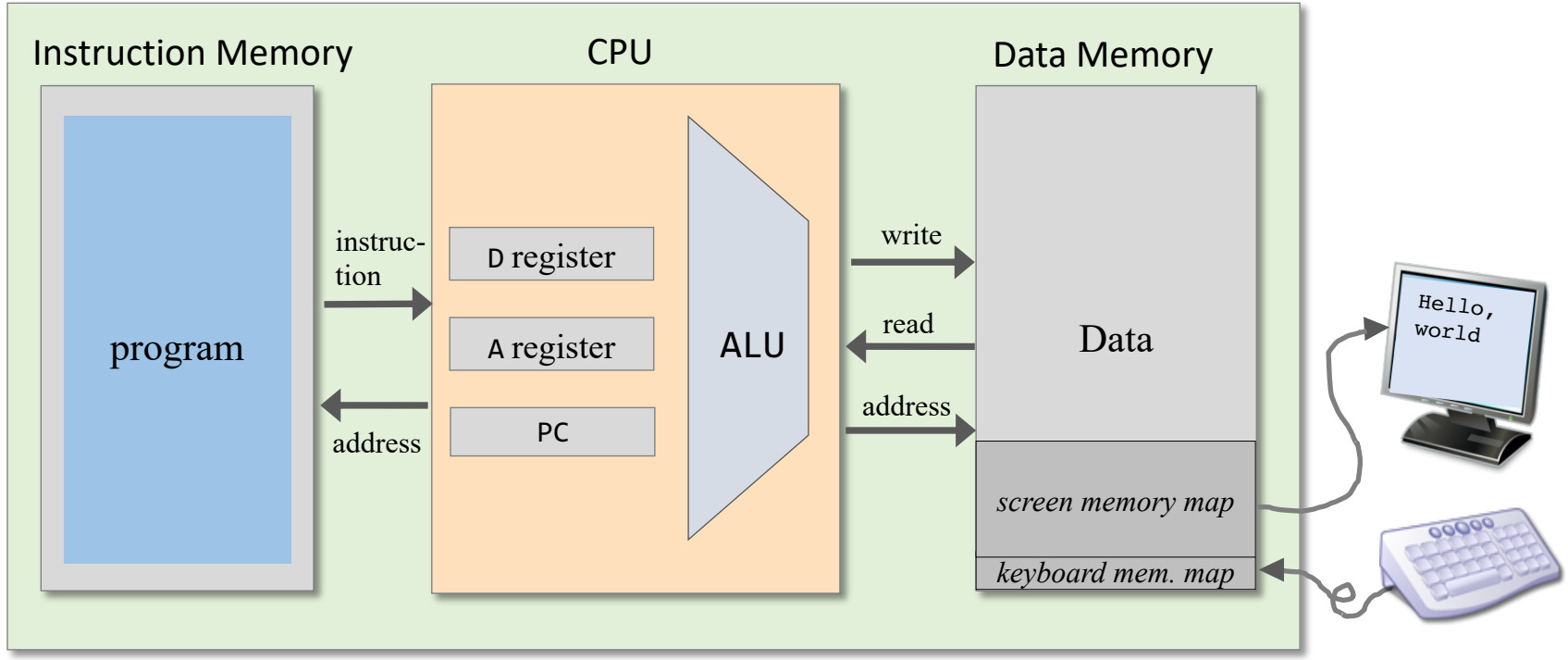
refresh



The screen memory map is implemented as an 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;
}
```

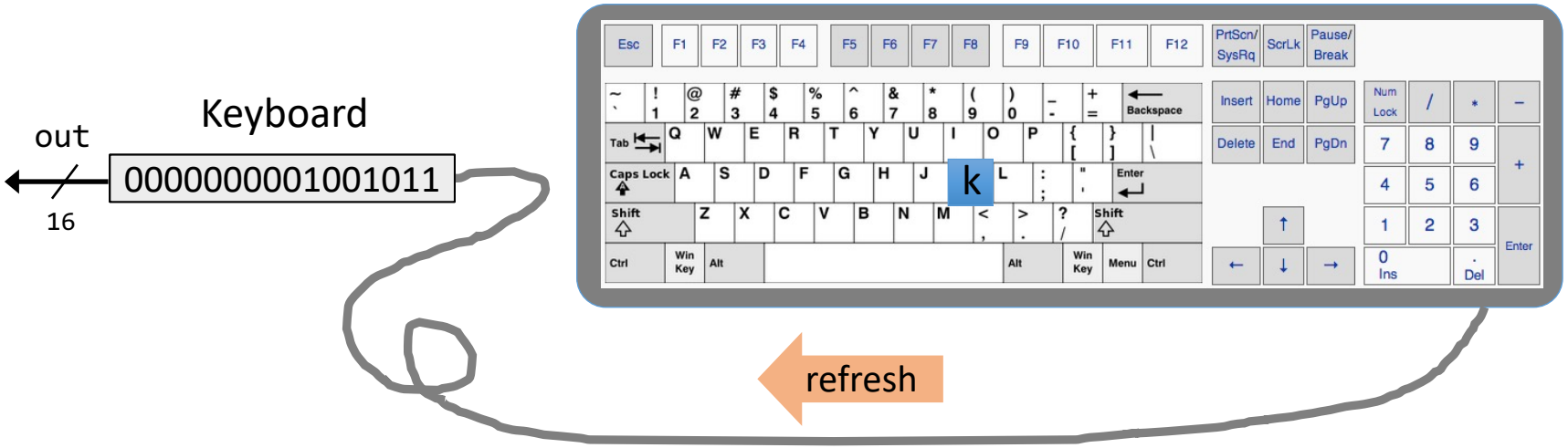
Hack computer



I/O devices

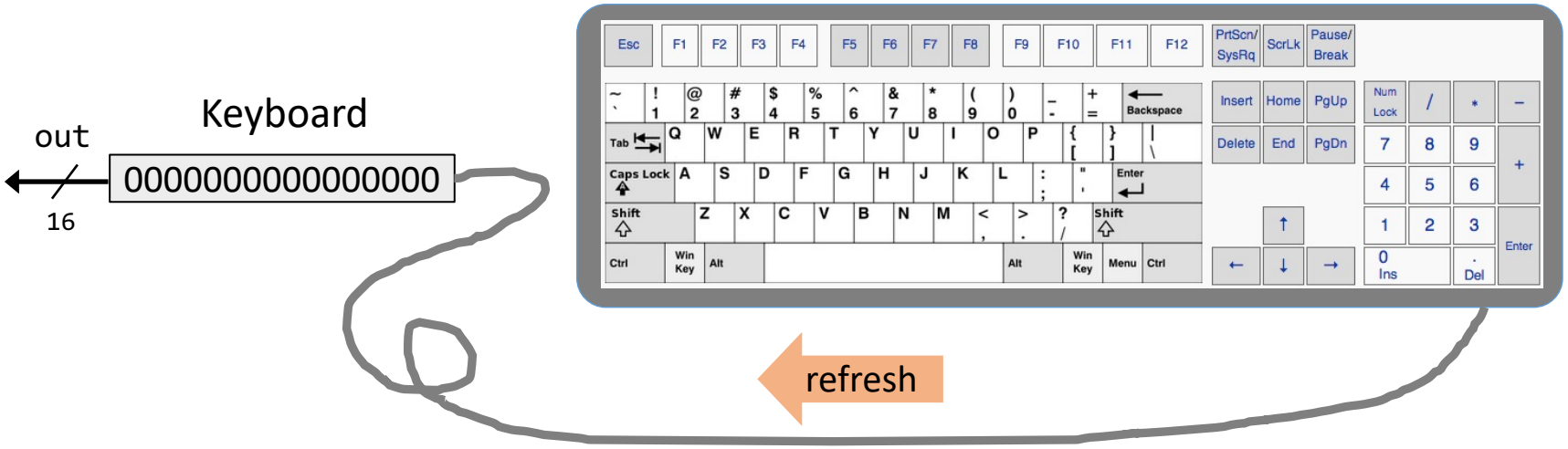
- Screen (black and white)
- Keyboard (regular)

Keyboard



code('k') = 75

Keyboard



The *keyboard memory map* is implemented as a 16-bit memory register named `Keyboard`

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

The Hack character set

key	code
(space)	32
!	33
“	34
#	35
$	36
%	37
&	38
‘	39
(	40
)	41
*	42
+	43
,	44
-	45
.	46
/	47

key	code
0	48
1	49
…	…
9	57

key	code
:	58
;	59
<	60
=	61
>	62
?	63
@	64

key	code
A	65
B	66
C	…
…	…
Z	90

key	code
[	91
/	92
]	93
^	94
_	95
`	96

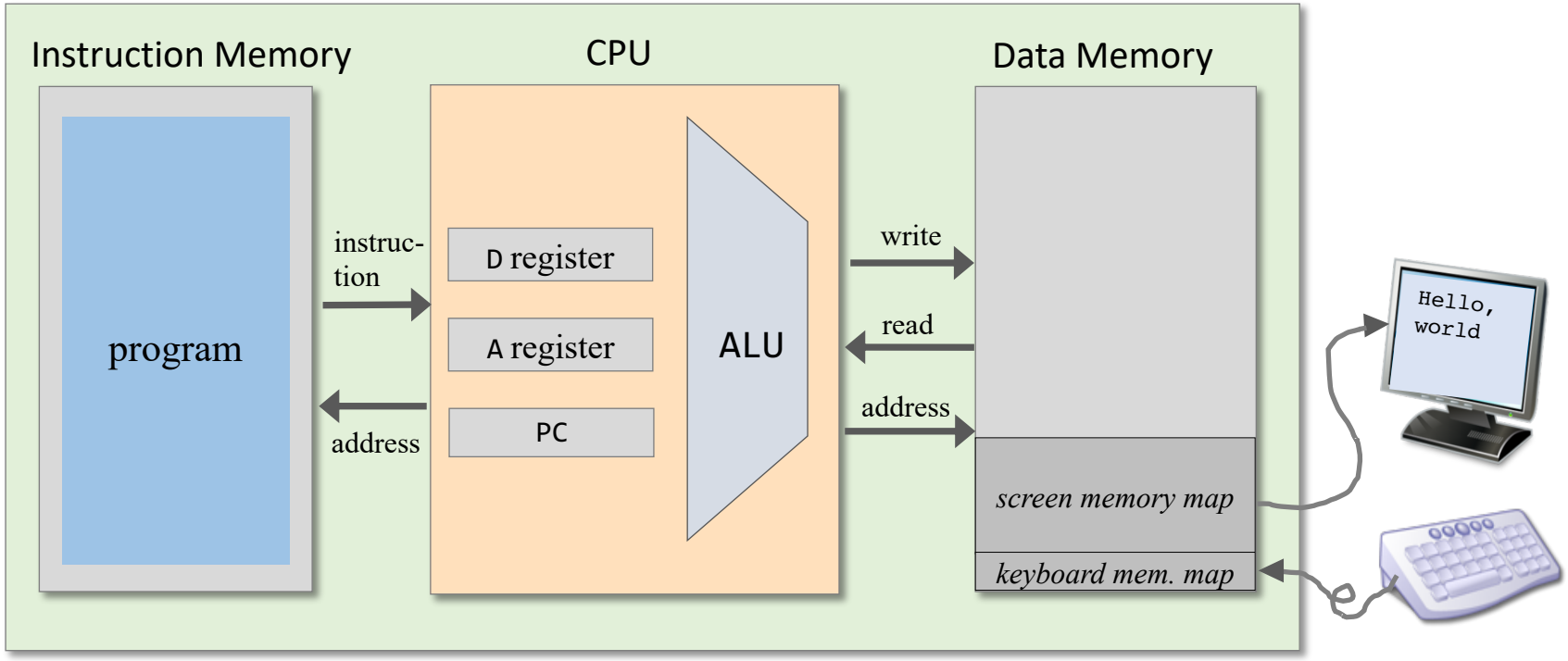
key	code
a	97
b	98
c	99
…	…
z	122

key	code
{	123
|	124
}	125
~	126

key	code
newline	128
backspace	129
left arrow	130
up arrow	131
right arrow	132
down arrow	133
home	134
end	135
Page up	136
Page down	137
insert	138
delete	139
esc	140
f1	141
. . .	. . .
f12	152

(Subset of Unicode)

Hack computer



I/O devices

- ✓ Screen (black and white)
- ✓ Keyboard (regular)

More I/O devices can be added, as needed

Each requiring a memory map, and an interaction contract

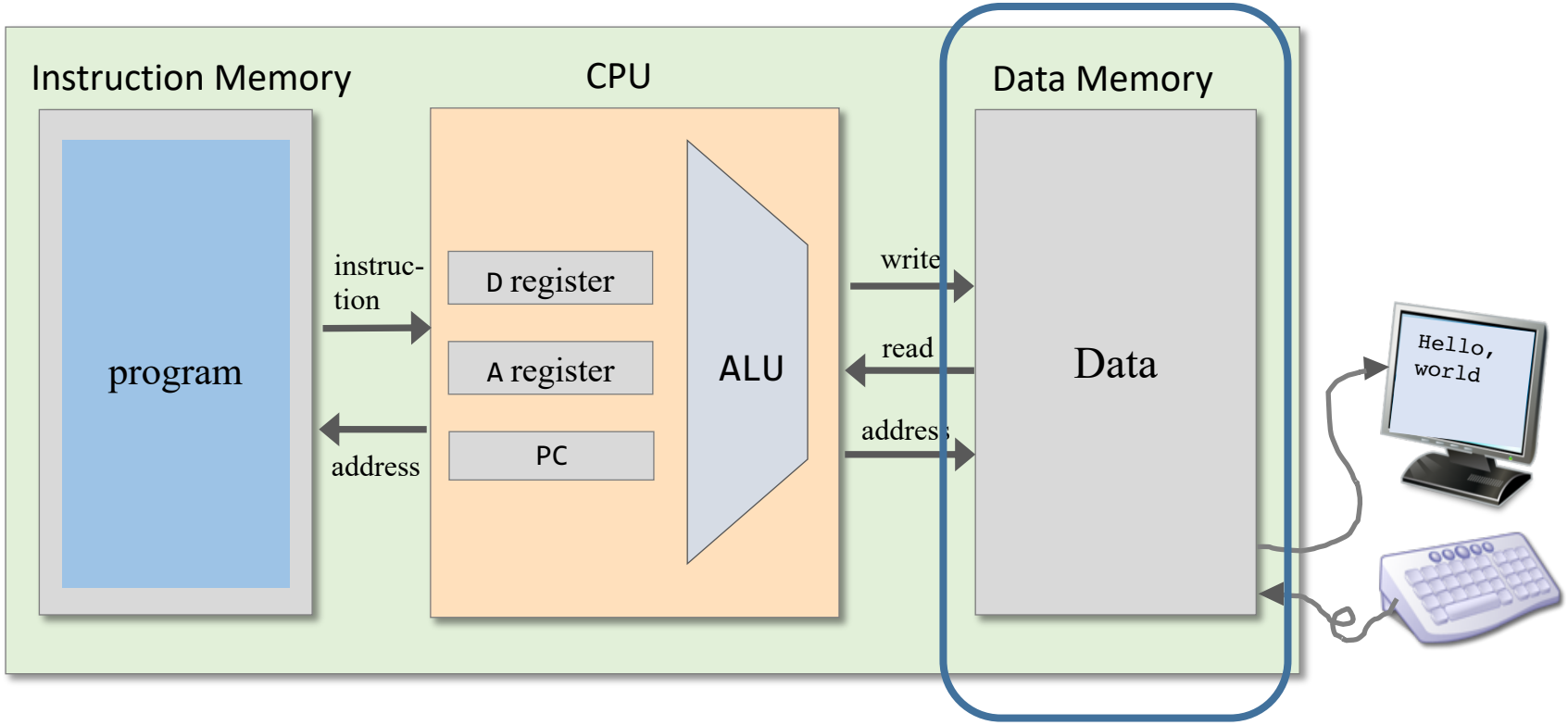
Managed jointly by the hardware and the OS.

Chapter 5: Computer Architecture

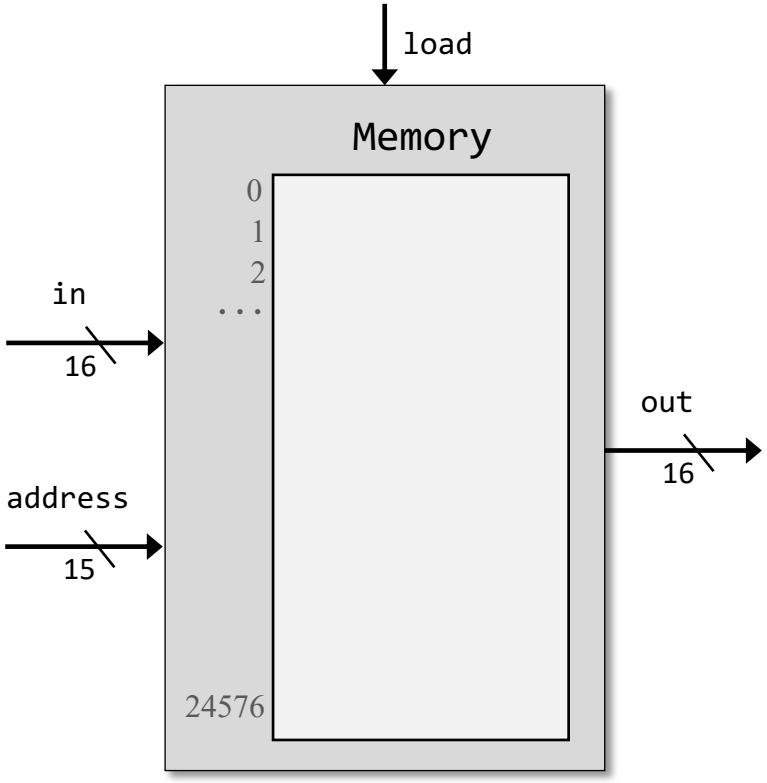
- Basic architecture
- Fetch-Execute cycle
- The Hack CPU
- Input / output

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

Memory


Instruction Memory
CPU
Data Memory
program
instruc-
tion
address
D register
A register
PC
ALU
write
read
address
Data
Hello,
world

Memory: Abstraction



Reading register *i*:

`address` ← *i*

Probe `out`

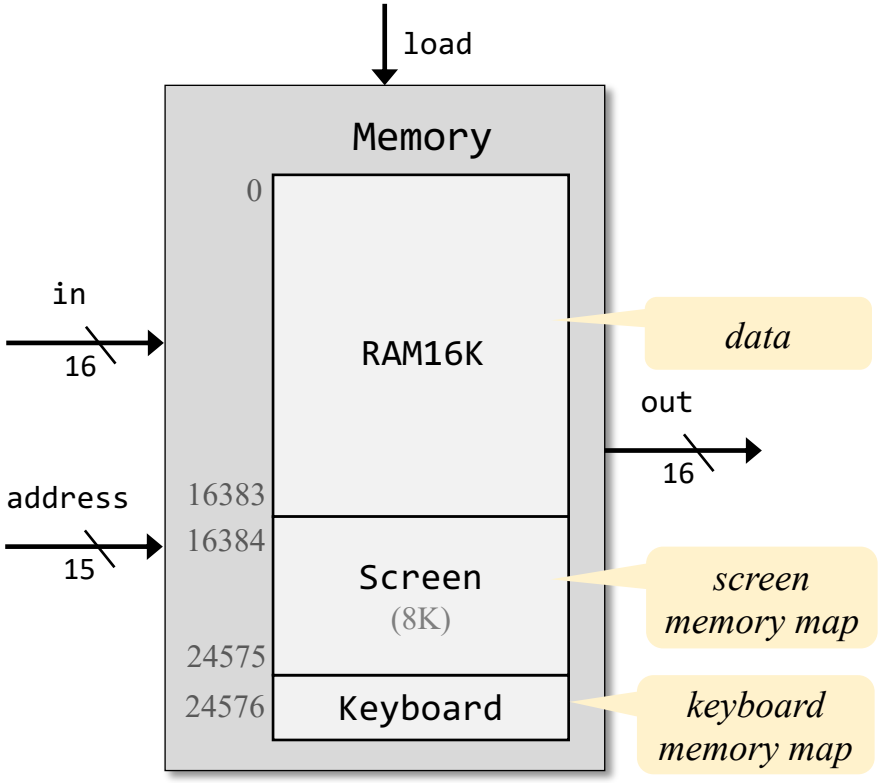
Setting register *i* to *v*:

`in` ← *v*

`address` ← *i*

`load` ← `1`

Memory: Implementation



Reading register *i*:

`address` ← *i*

Probe `out`

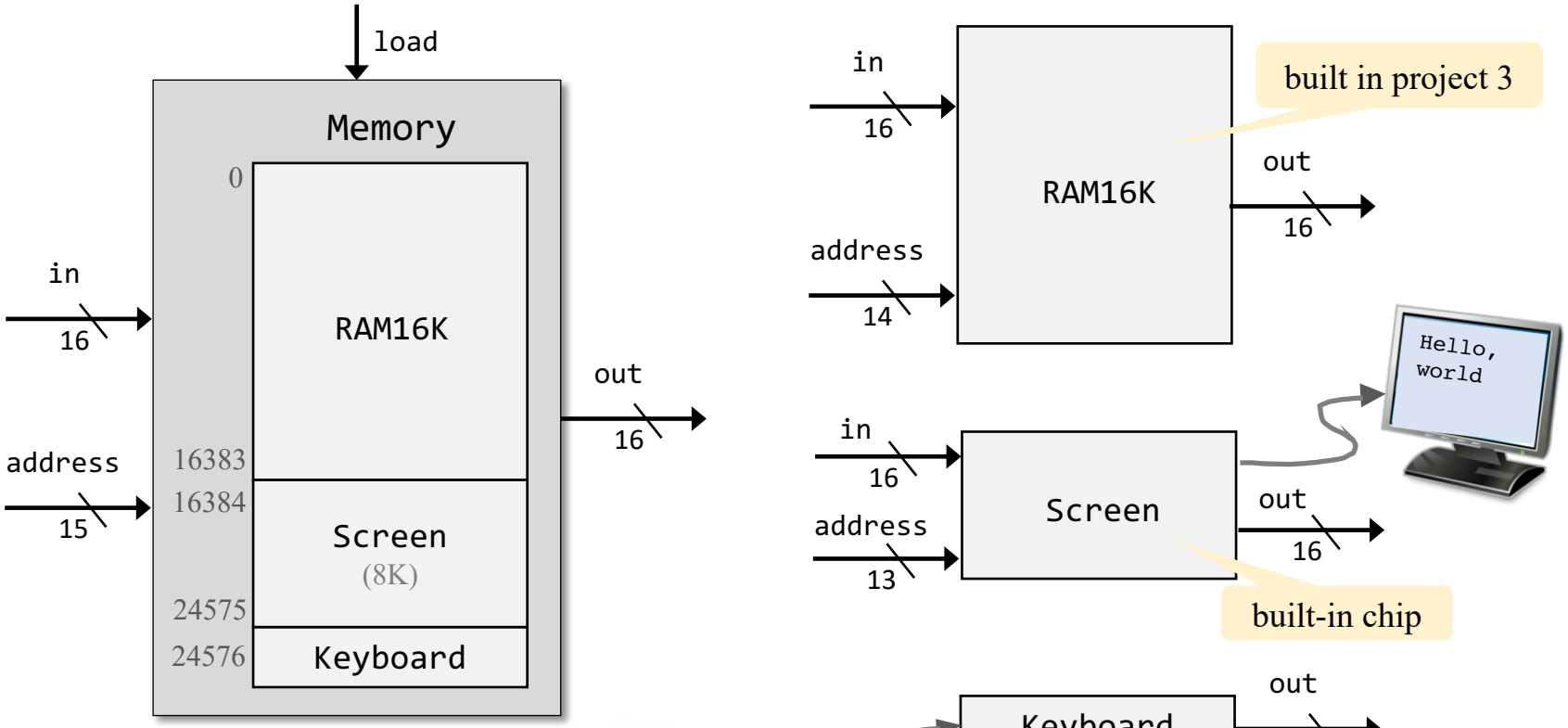
Setting register *i* to *v*:

`in` ← *v*

`address` ← *i*

`load` ← `1`

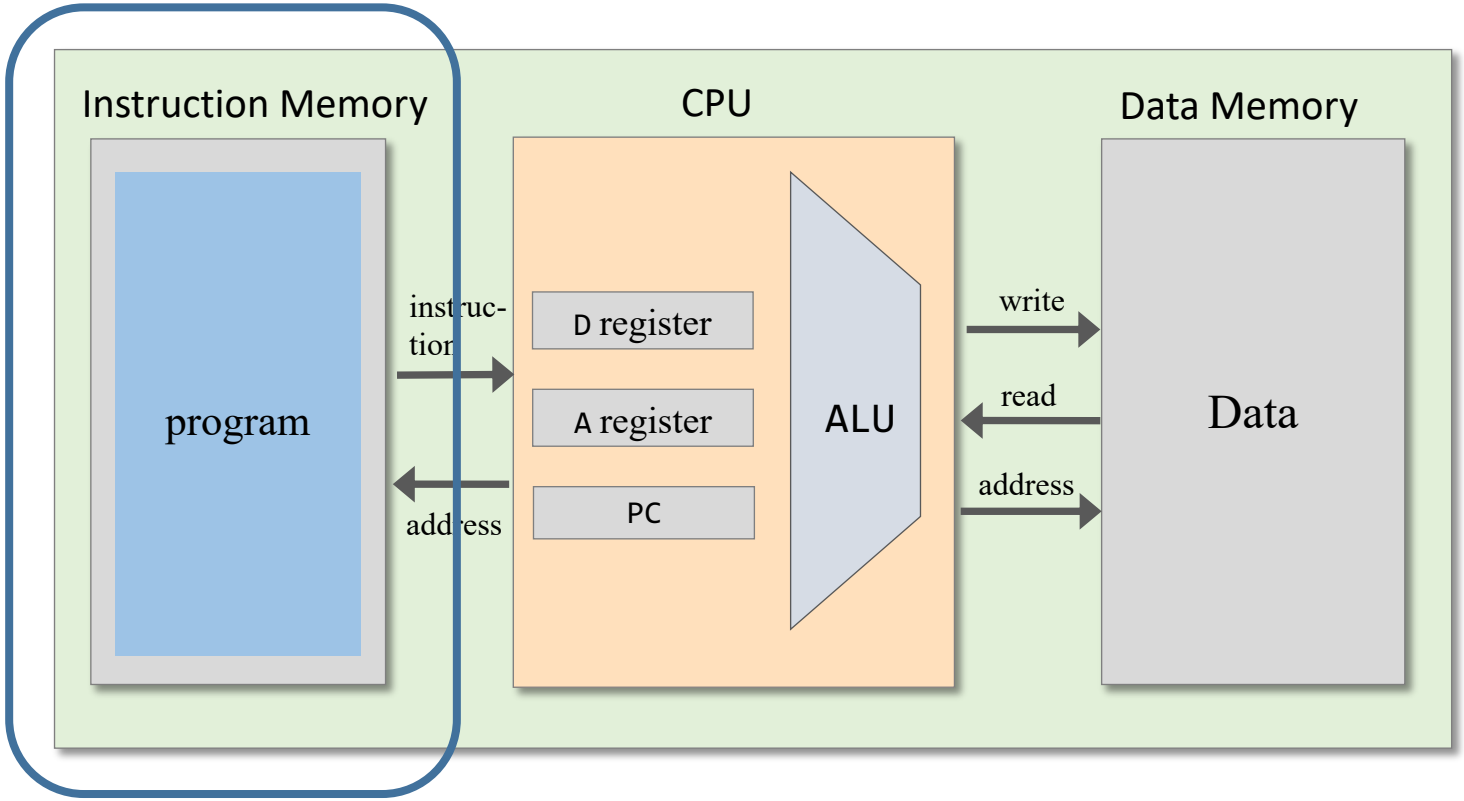
Memory: Implementation



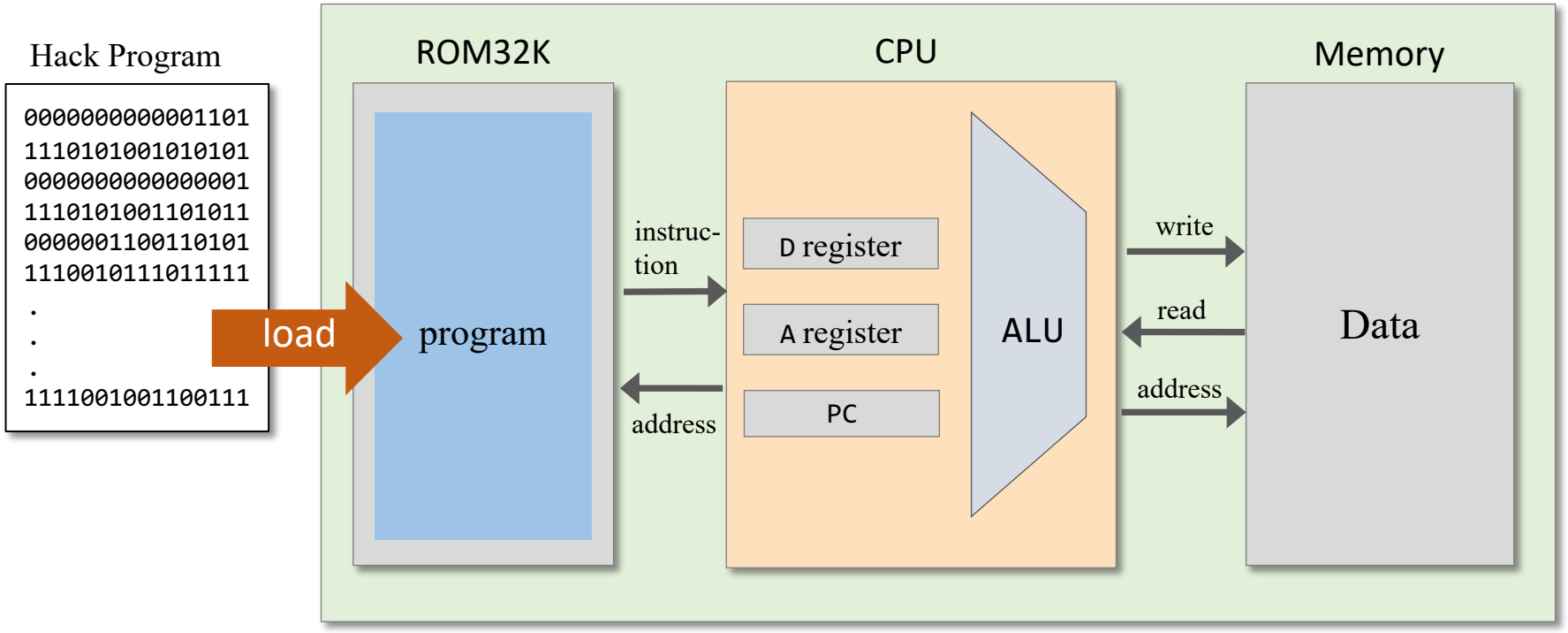
Memory architecture

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

Instruction memory



Instruction memory



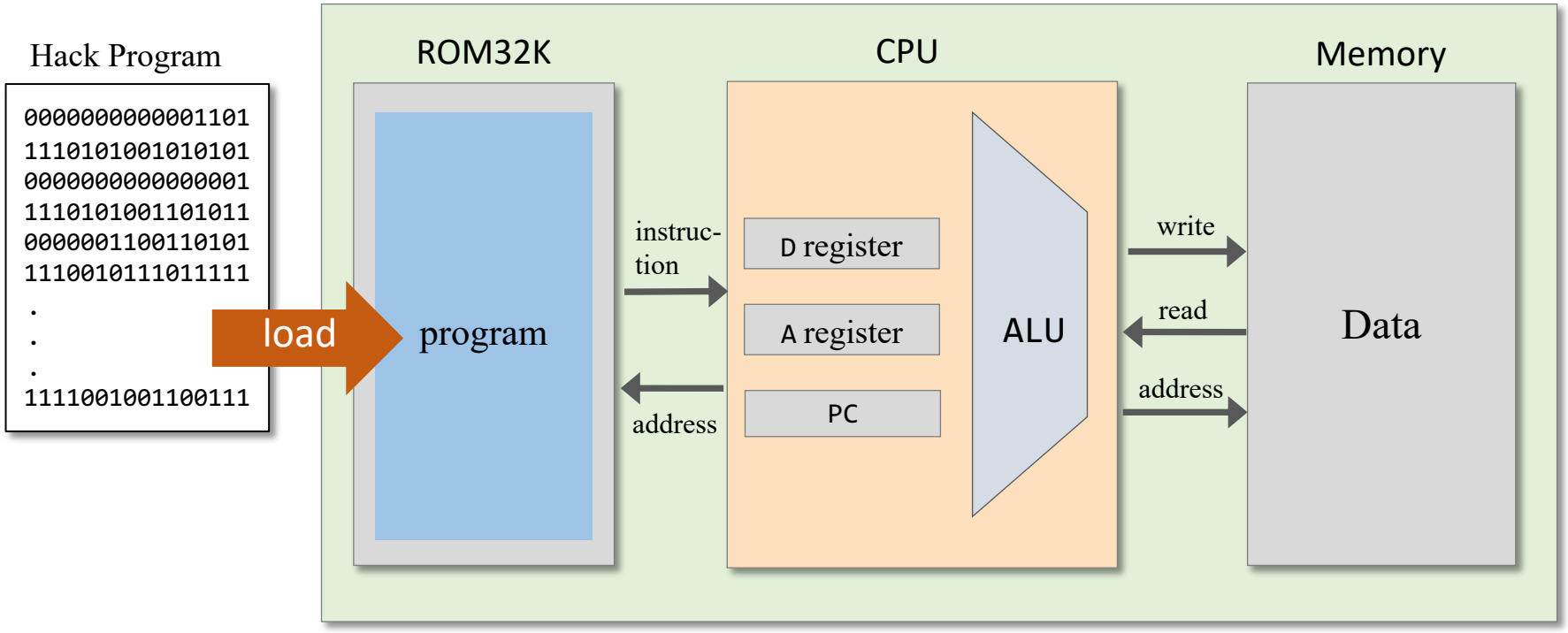
Hardware implementation

Plug-and-play ROM 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;
}
```

Instruction memory



Hardware implementation

Plug-and-play ROM chip, named `ROM32K`

(pre-loaded with a program)

Hardware simulation

- Programs are stored in text files;
- The simulator software features a *load-program* service.

Chapter 5: Computer Architecture

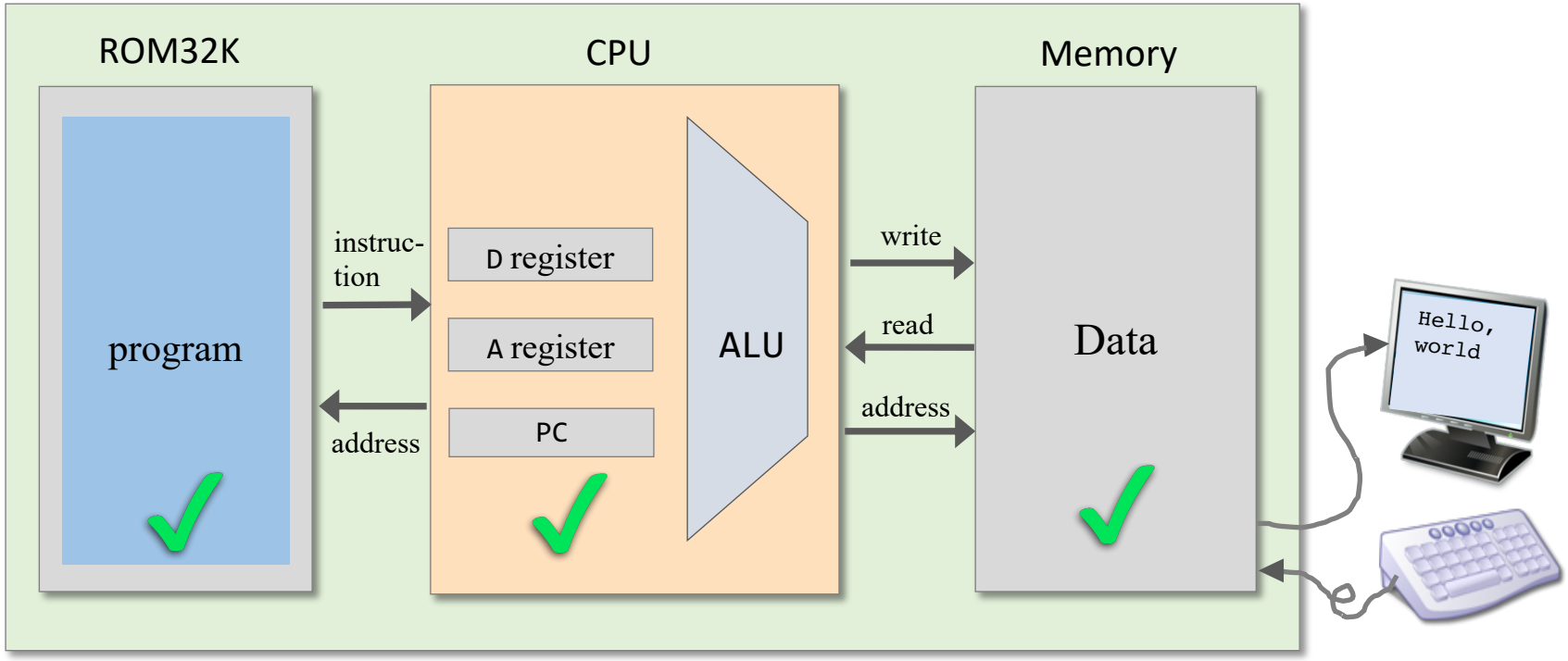
- Basic architecture
- Fetch-Execute cycle
- The Hack CPU
- Input / output

✓ Memory

Computer

- Project 5: Chips
- Project 5: Guidelines

Hack computer architecture



Remaining challenge

Integrate into a single `Computer` chip

Computer abstraction

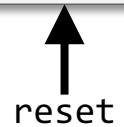
Assumption:

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

`Computer` abstraction:

if (`reset == 1`), executes the *first* instruction in the stored program

if (`reset == 0`), executes the *next* instruction in the stored program



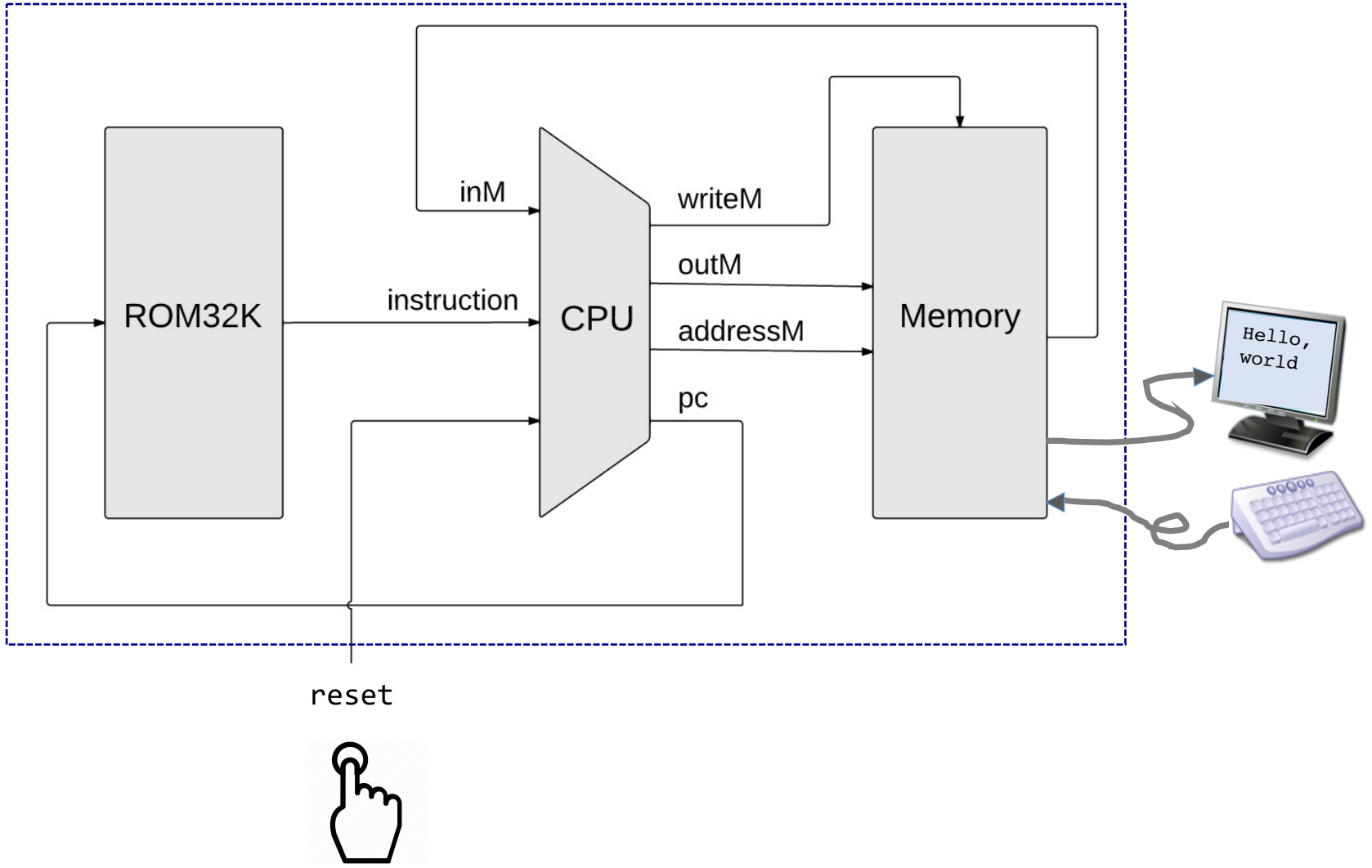
like pushing and releasing a button

To execute the stored program:

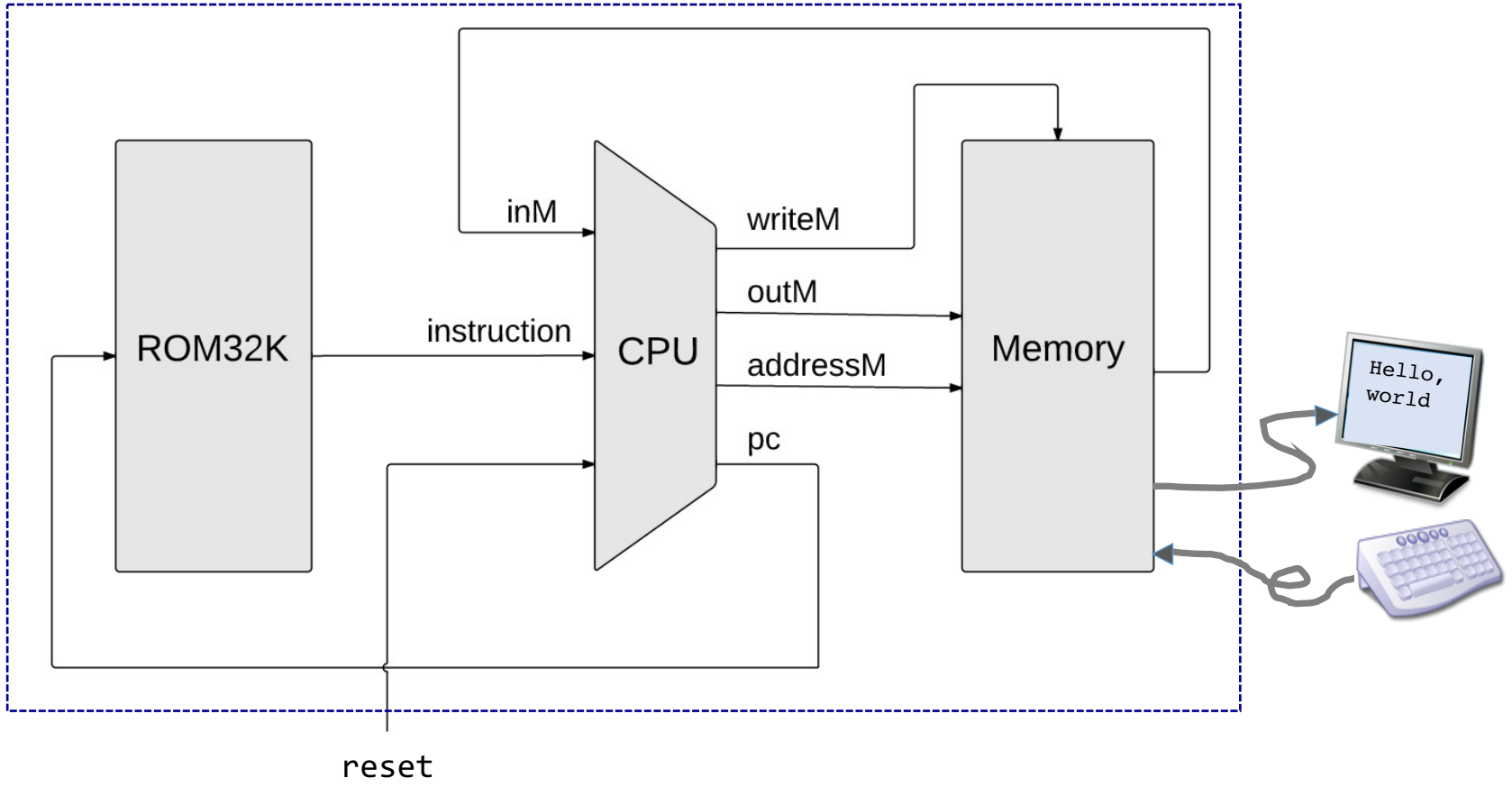
set `reset ← 1`, then

set `reset ← 0`

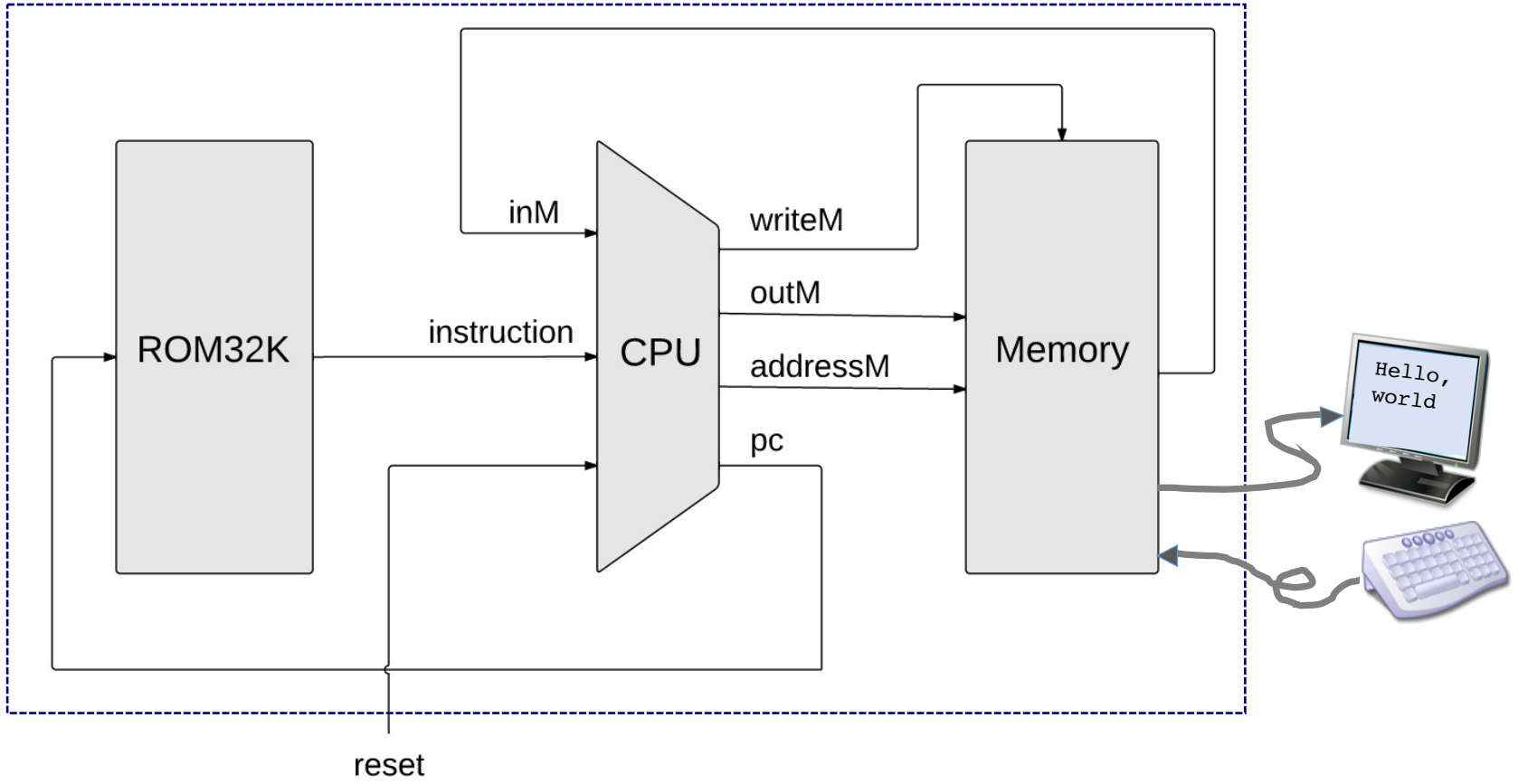
Computer implementation



Computer implementation



Computer implementation



"*Make everything as simple as possible, but no simpler.*"

– Albert Einstein

Chapter 5: Computer Architecture

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

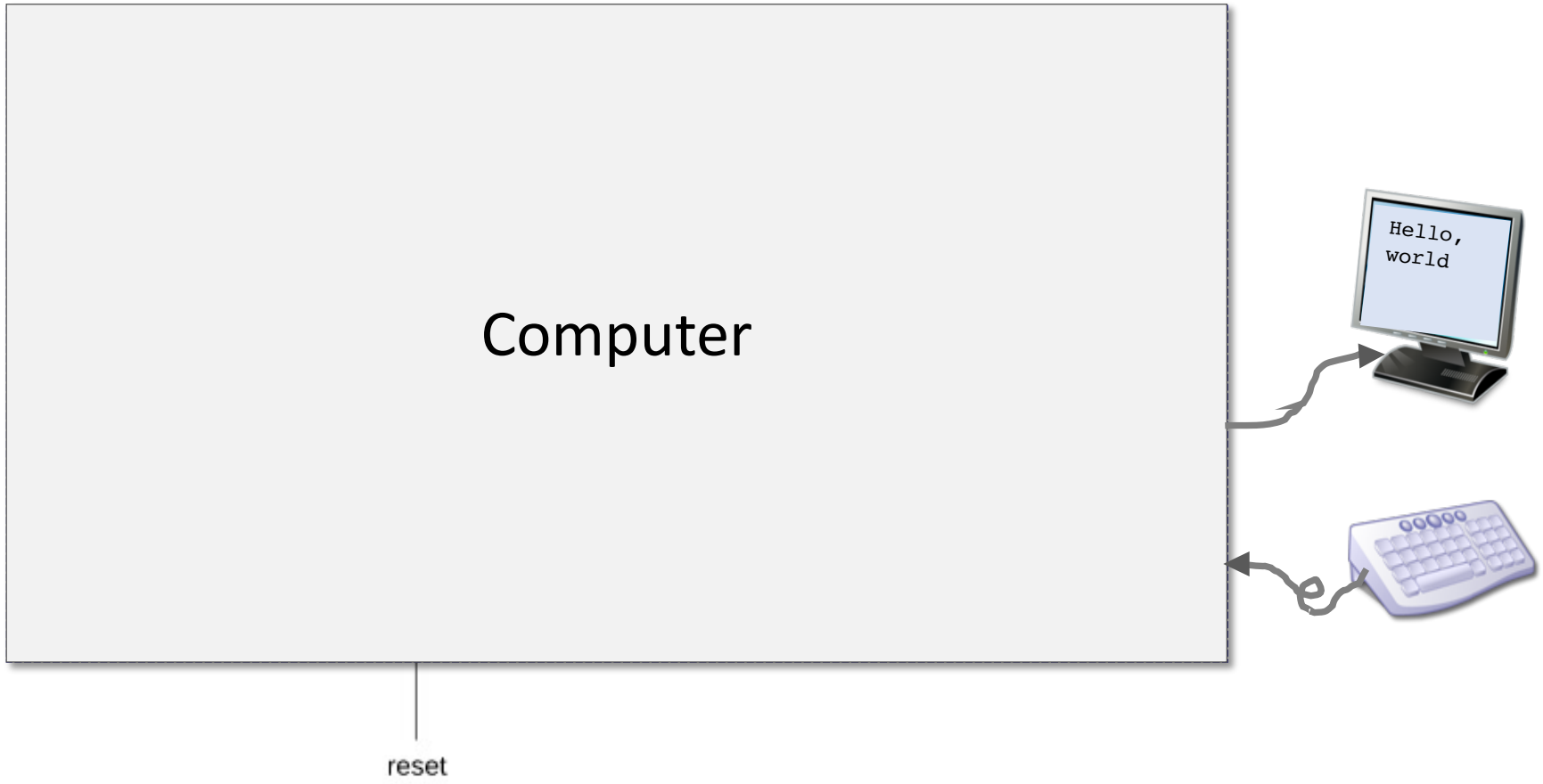
✓ Memory

✓ Computer

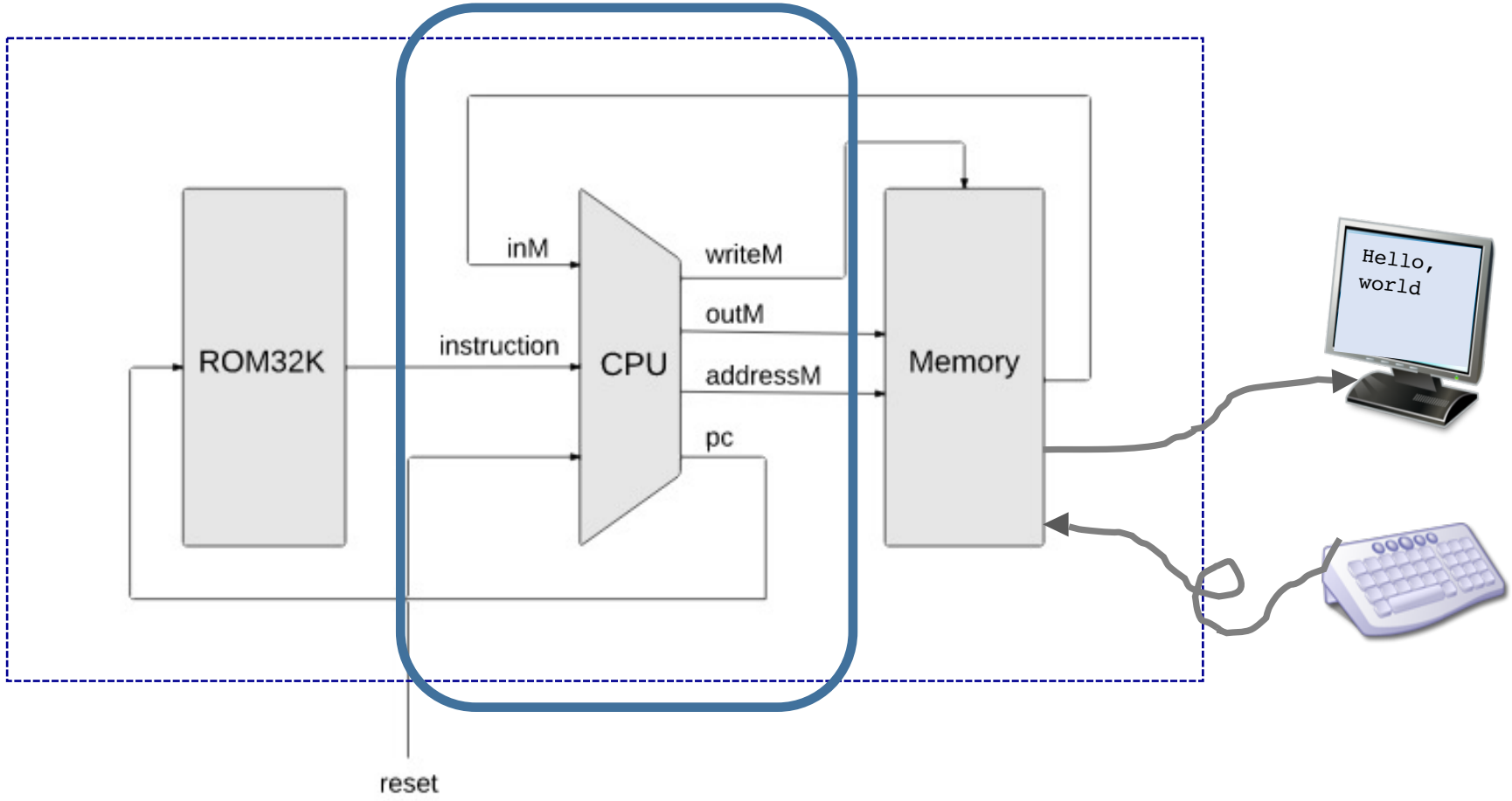
Project 5: Chips

- Project 5: Guidelines

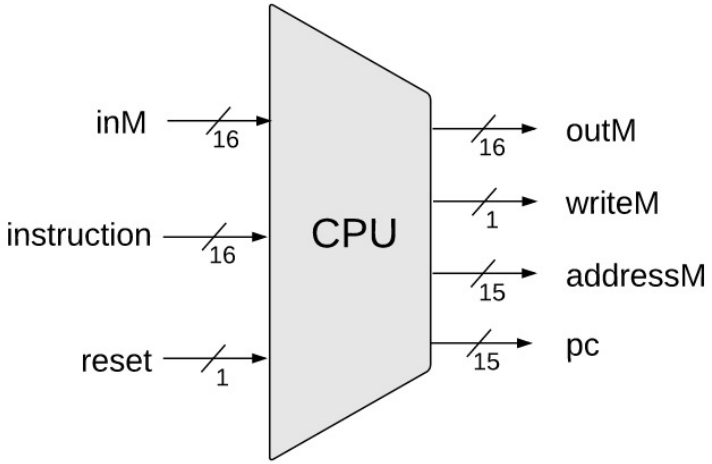
Hack computer



Hack computer

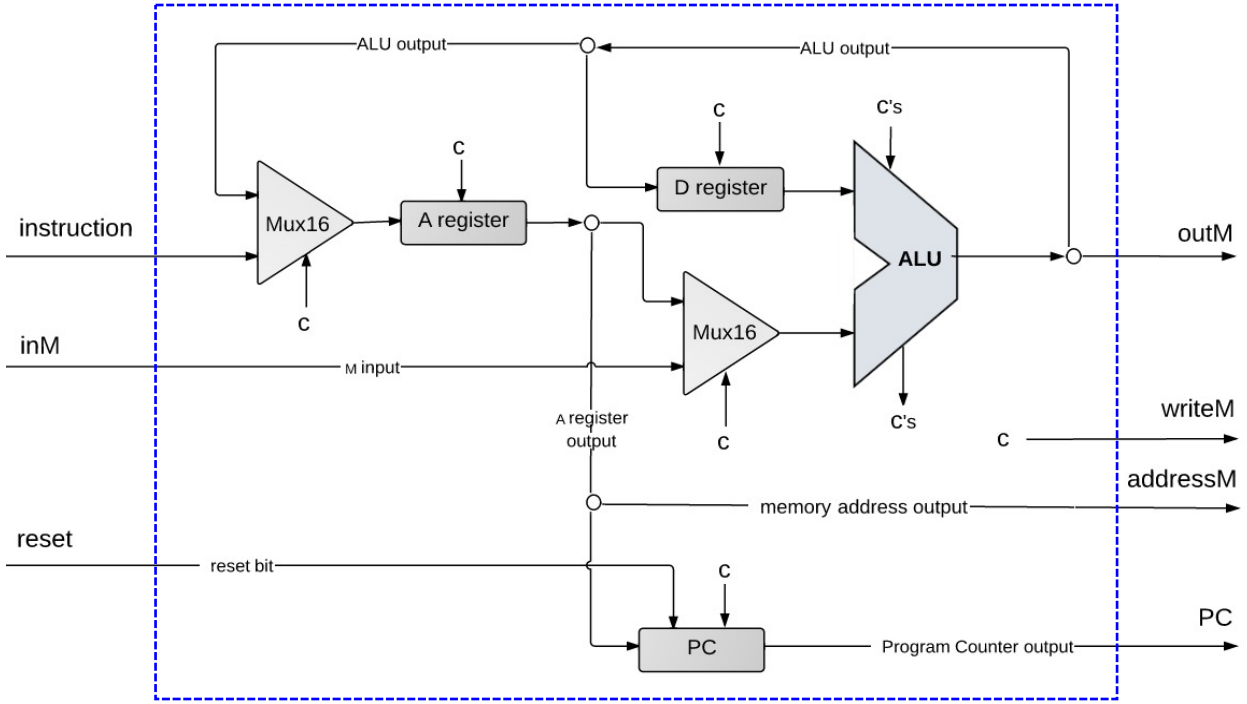


CPU




```
/** 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 you code here:
}
```

CPU implementation



Chip parts:

Built in project 1 and project 2

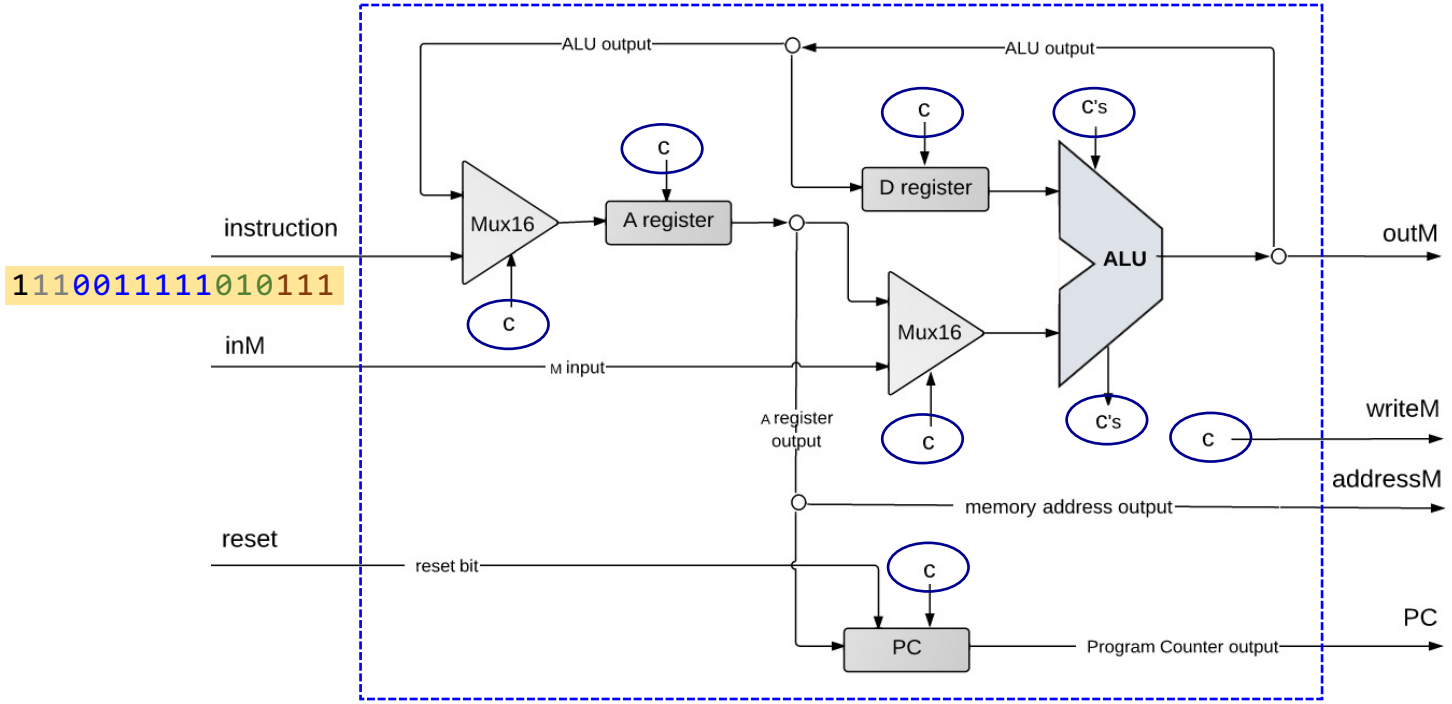
- `Mux16`
- `ALU`
- `And`, `Or`, `Not` ...

Built in project 3

- `ARegister`
- `DRegister`
- `PC`

`ARegister` and `Dregister` are built-in `Register` chips

CPU implementation



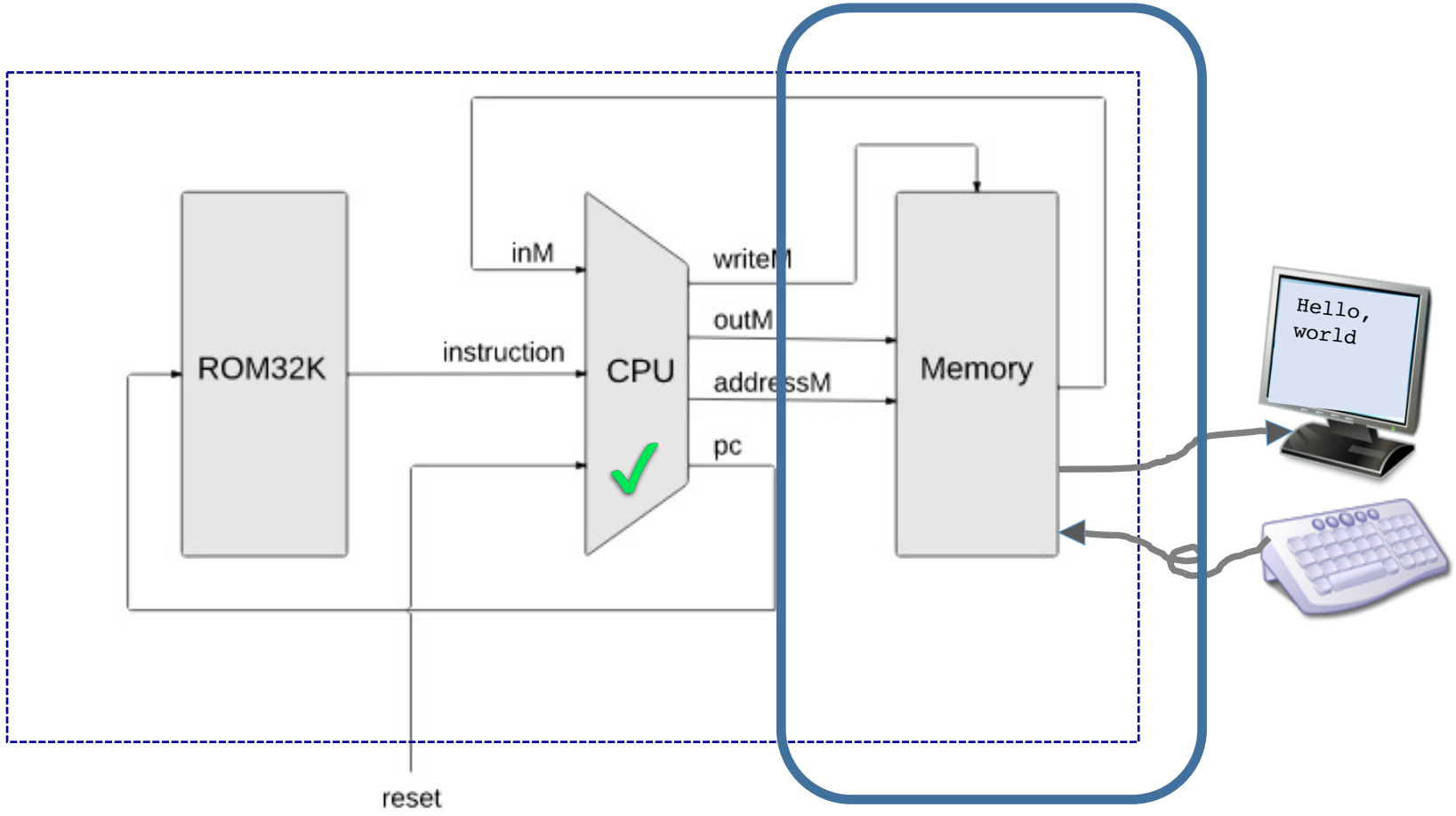
Implementation

- Use HDL to route instruction bits to chip-parts
- Use gate logic to compute the address of the next instruction

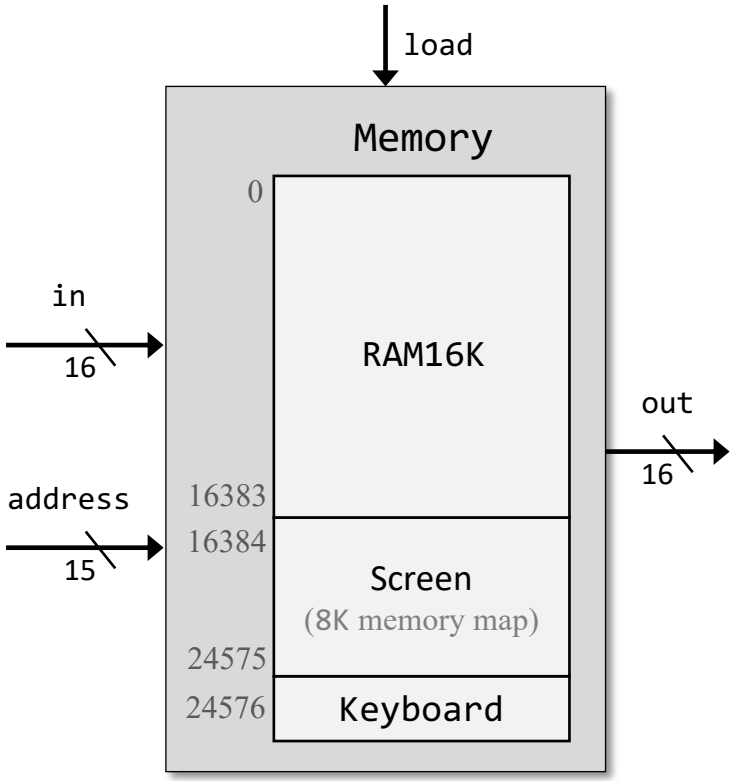
Tips

- No need for “helper chips”
- Use logic gates and HDL for implementing everything.

Computer


ROM32K
instruction
CPU
inM
writeM
outM
addressM
pc
reset
Memory
Hello,
world

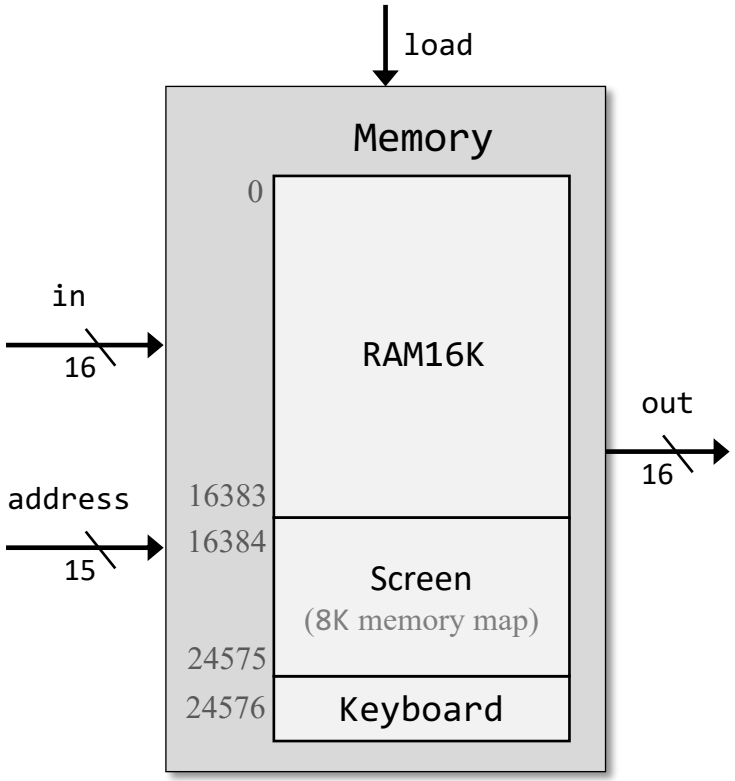
Memory



Memory.hdl

```
/** Complete address space of the computer's data memory,
    including RAM and memory mapped I/O.
    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.
    Address space rules:
    Only the upper 16K+8K+1 words of the memory are used.
    Access to address 0 to 16383 results in accessing the RAM;
    Access to address 16384 to 24575 results in accessing
                  the Screen memory map;
    Access to address 24576 results in accessing the Keyboard
                  memory map.
 */
CHIP Memory {
  IN   address[15], in[16], load;
  OUT  out[16];
  PARTS:
     // Put your code here.
}
```

Memory implementation




```
/** Memory of 16K 16-bit registers */
CHIP RAM16K {
  IN
    address[14], in[16], load;
  OUT
    out[16];

}
```

built in project 3

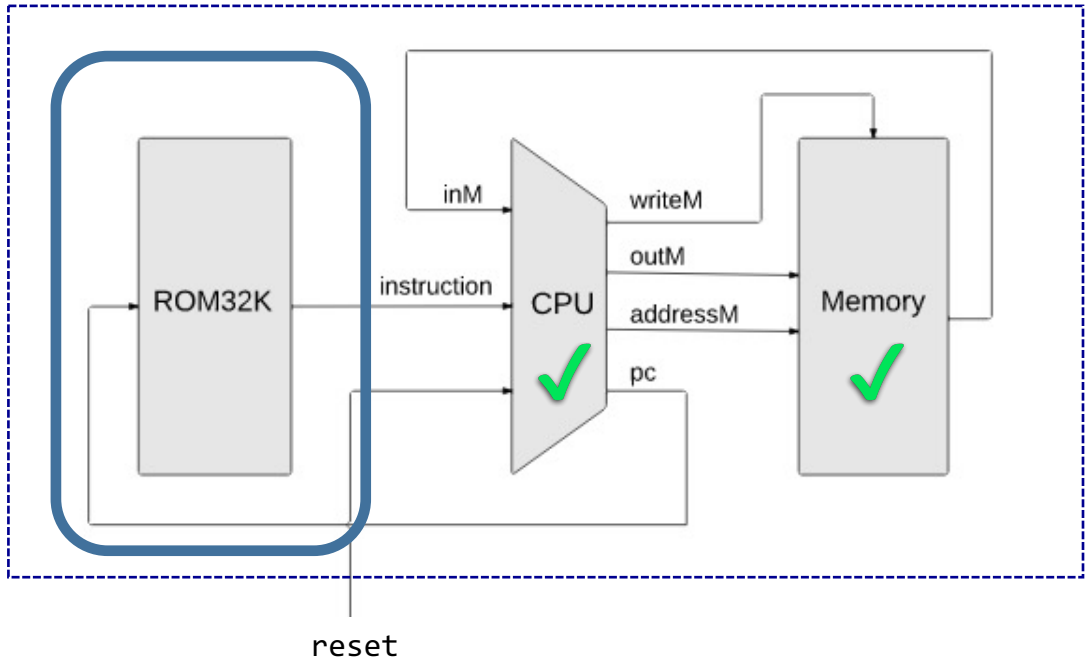
```
/** Memory of 8K 16-bit registers
    with a display unit side effect. */
CHIP Screen {
  IN
    address[13], in[16], load;
  OUT
    out[16];

    BUILTIN Screen;

}
```

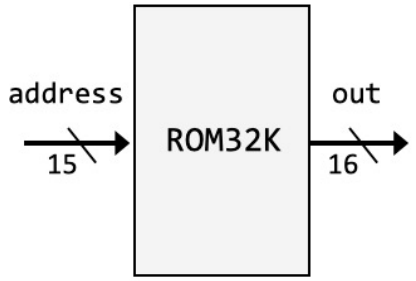
```
/** 16-bit register with a
    keyboard input side effect */
CHIP Keyboard {
  OUT
    out[16];
    BUILTIN Keyboard;
}
```

Implementation tip:

Use logic gates for mapping the `address` input on the corresponding address input of the relevant chip-part.

Instruction memory

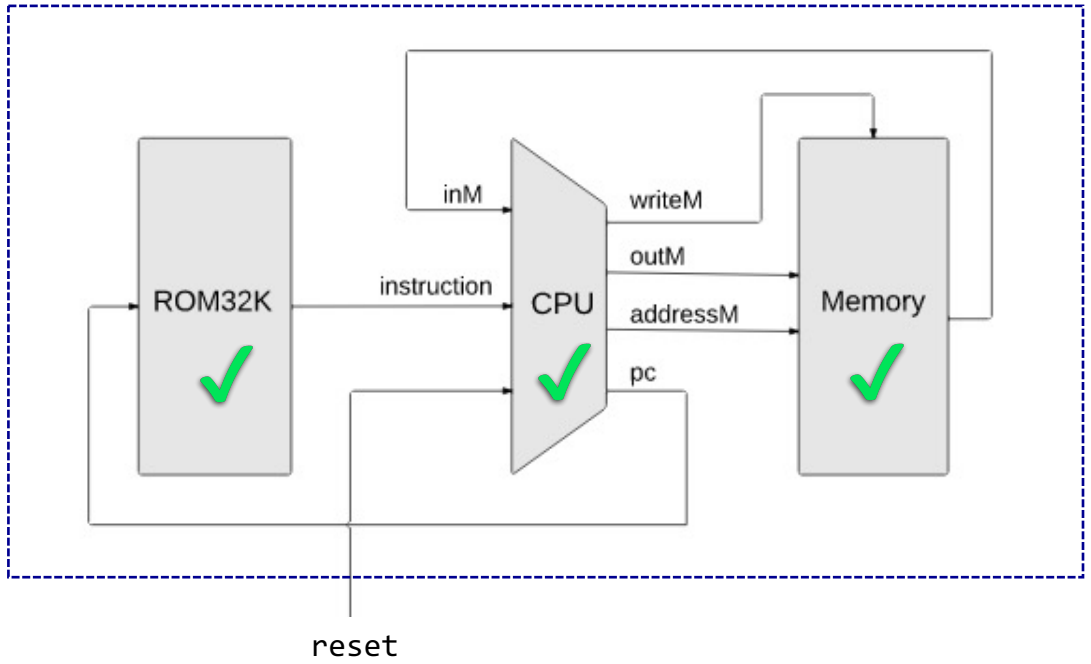




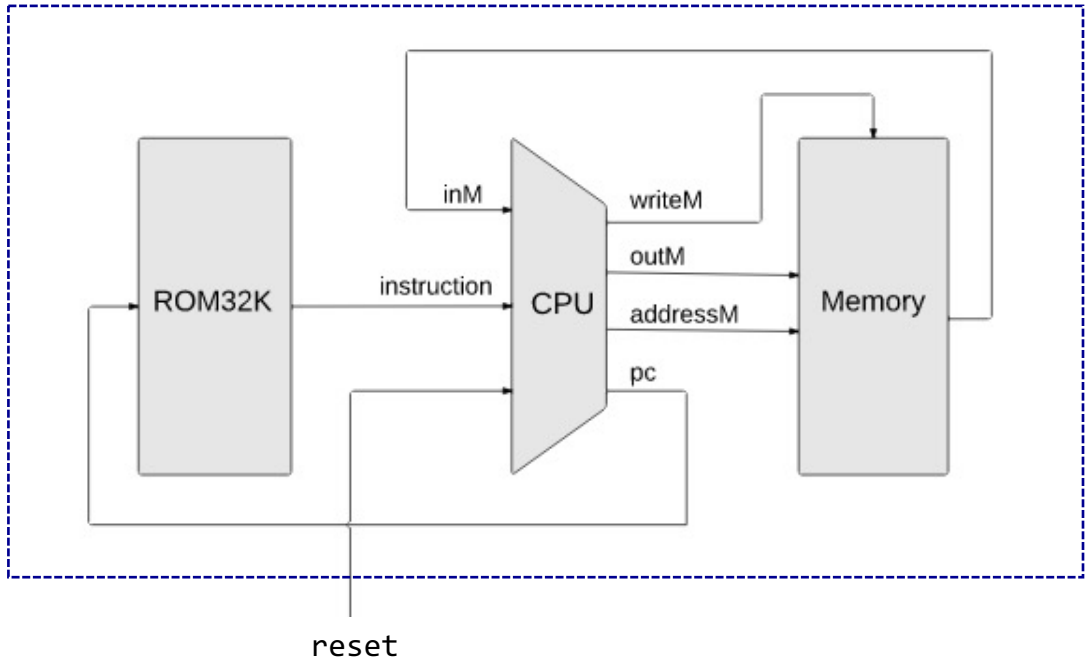
ROM32K.hdl

```
/** Read-Only memory (ROM),
    acting as the Hack computer instruction memory. */
CHIP ROM32K {

    IN  address[15];

    OUT out[16];

    BUILTIN ROM32K;

}
```

Computer


ROM32K
instruction
CPU
inM
writeM
outM
addressM
pc
Memory
reset

Computer implementation



Computer.hdl

```
/** 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.
}
```

Chapter 5: Computer Architecture

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

✓ Memory

✓ Computer

✓ Project 5: Chips

➡ Project 5: Guidelines

Project 5

Build three chips:

- `Memory.hdl` chip-parts: `RAM16K`, `Screen`, `Keyboard`
- `CPU.hdl` chip-parts: `ARgister`, `DRegister`, `PC`, `ALU`, ...
- `Computer.hdl` chip-parts: `CPU`, `Memory`, `ROM32K`

(All the chip-parts should be built-in chips, except for `Memory` and `CPU`)

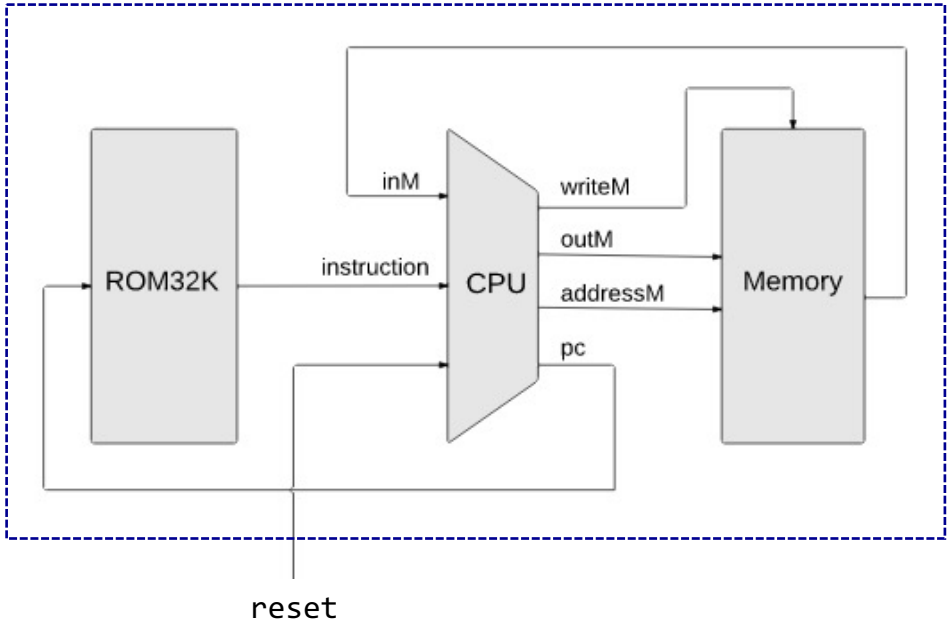
Tools

- Text editor
- Hardware simulator

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.
 * 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.
}
```

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]` ← `2 + 3`
- `Max.hack`:
  `RAM[2]` ← *max*(`RAM[0]`, `RAM[1]`)
- `Rect.hack`:
  Draws a rectangle of `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]` ← `2 + 3`
- `Max.hack`:
  `RAM[2]` ← *max*(`RAM[0]`, `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 executes 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

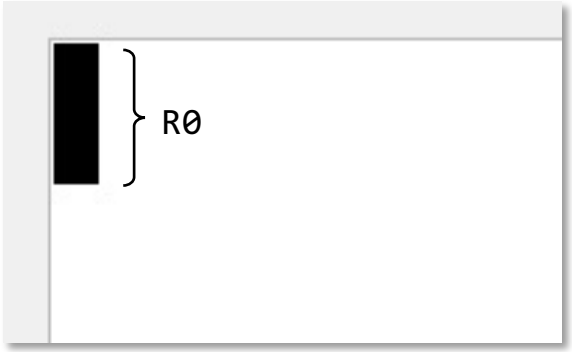
Testing logic:

- 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`:
  `RAM[0]` ← `2 + 3`
- `Max.hack`:
  `RAM[2]` ← *max*(`RAM[0]`, `RAM[1]`)
- `Rect.hack`:
  Draws a rectangle of `RAM[0]` rows of 16 pixels each.

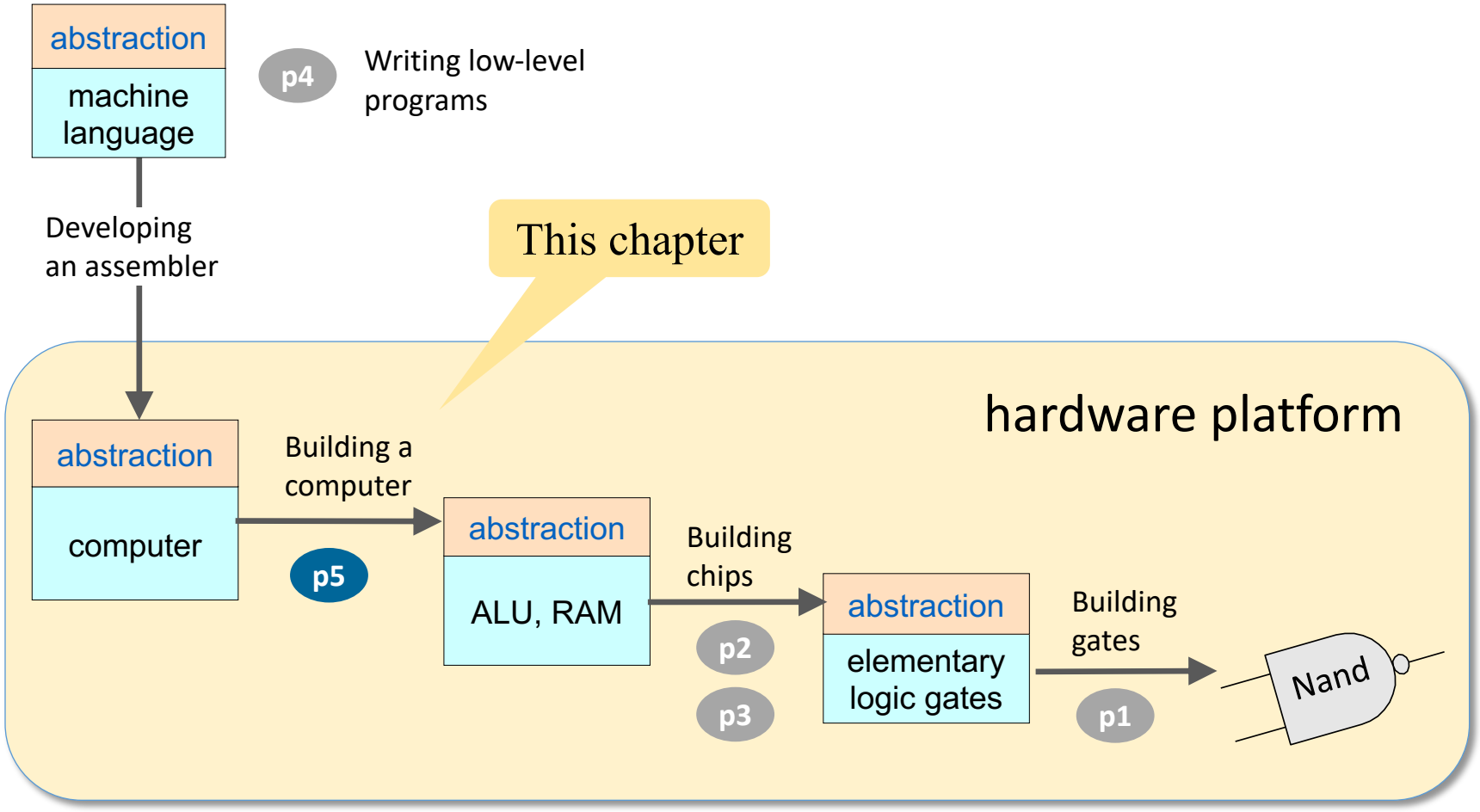
`Rect.hack` output:



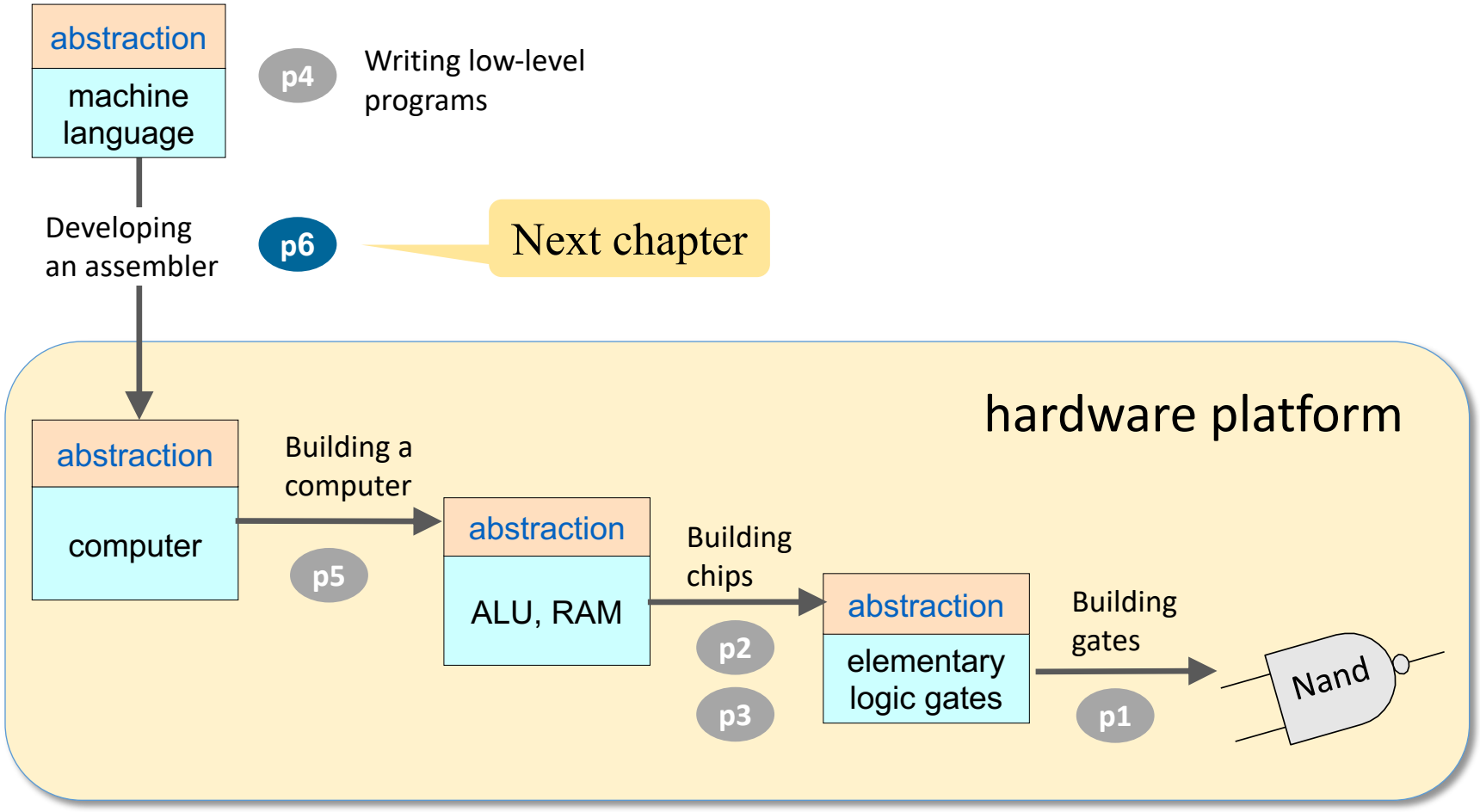
Test script

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

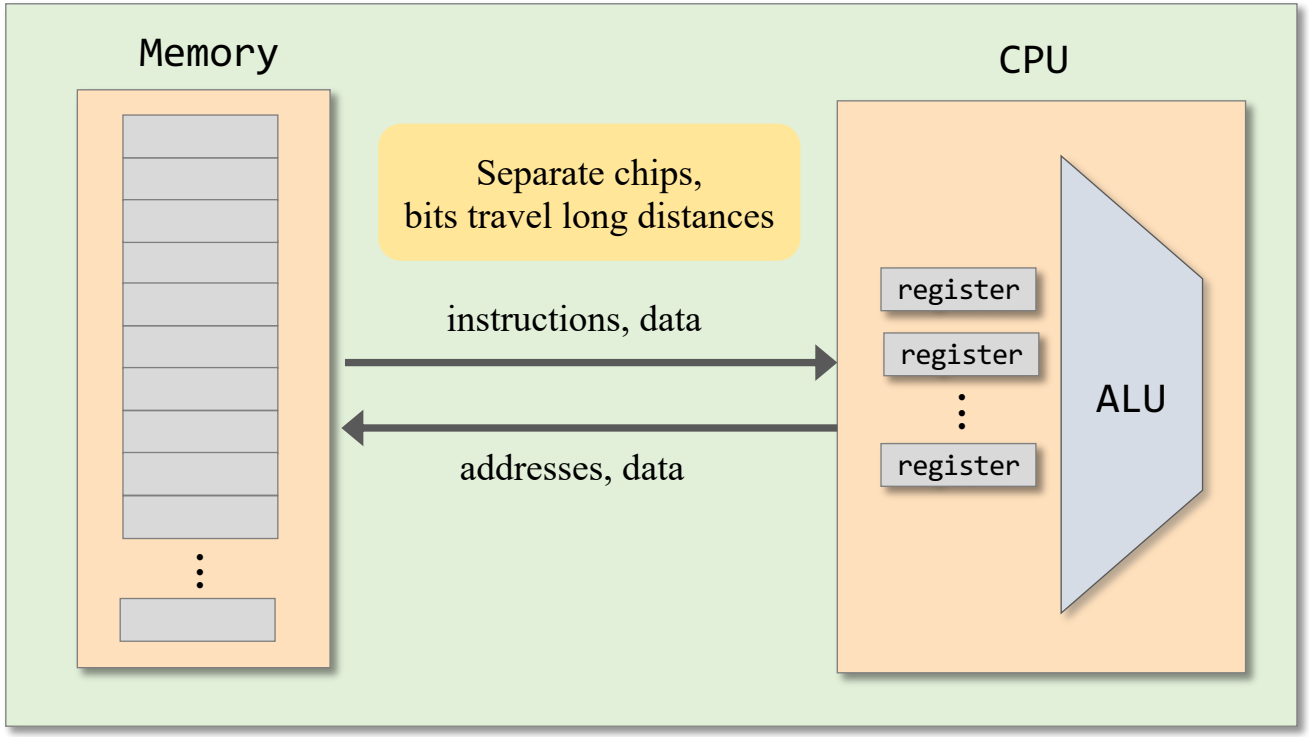
What's next?


abstraction
machine language
p4
Writing low-level programs
Developing an assembler
This chapter
hardware platform
abstraction
computer
Building a computer
p5
abstraction
ALU, RAM
Building chips
p2
p3
abstraction
elementary logic gates
Building gates
p1
Nand

What's next?


abstraction
machine language
p4
Writing low-level programs
Developing an assembler
p6
Next chapter
hardware platform
abstraction
computer
Building a computer
p5
abstraction
ALU, RAM
Building chips
p2
p3
abstraction
elementary logic gates
Building gates
p1
Nand

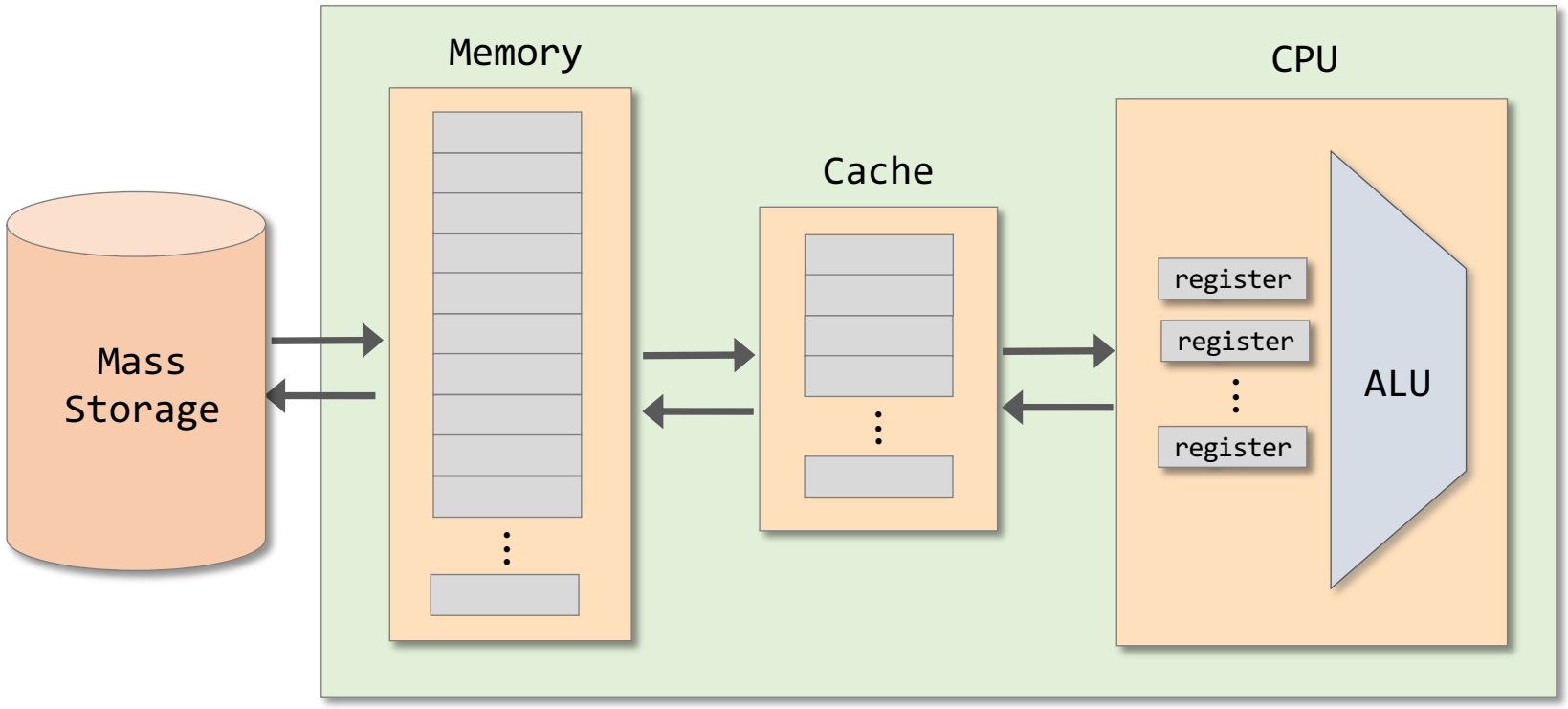
End note: Memory hierarchy



Challenges

- Slow access time
- Limited memory space

End note: Memory hierarchy



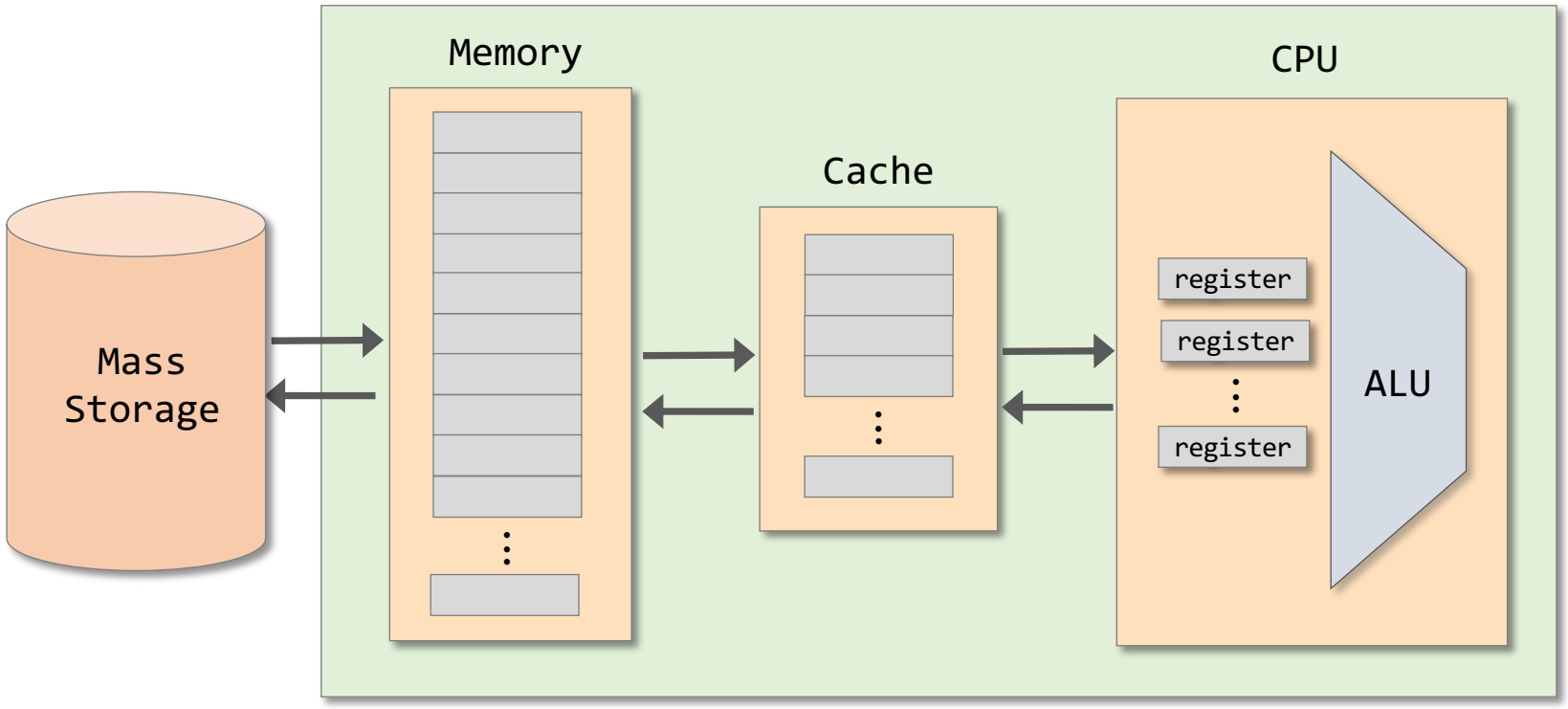
Challenges

- Slow access time
- Limited memory space

Typical solutions

- Cache memory
- External storage

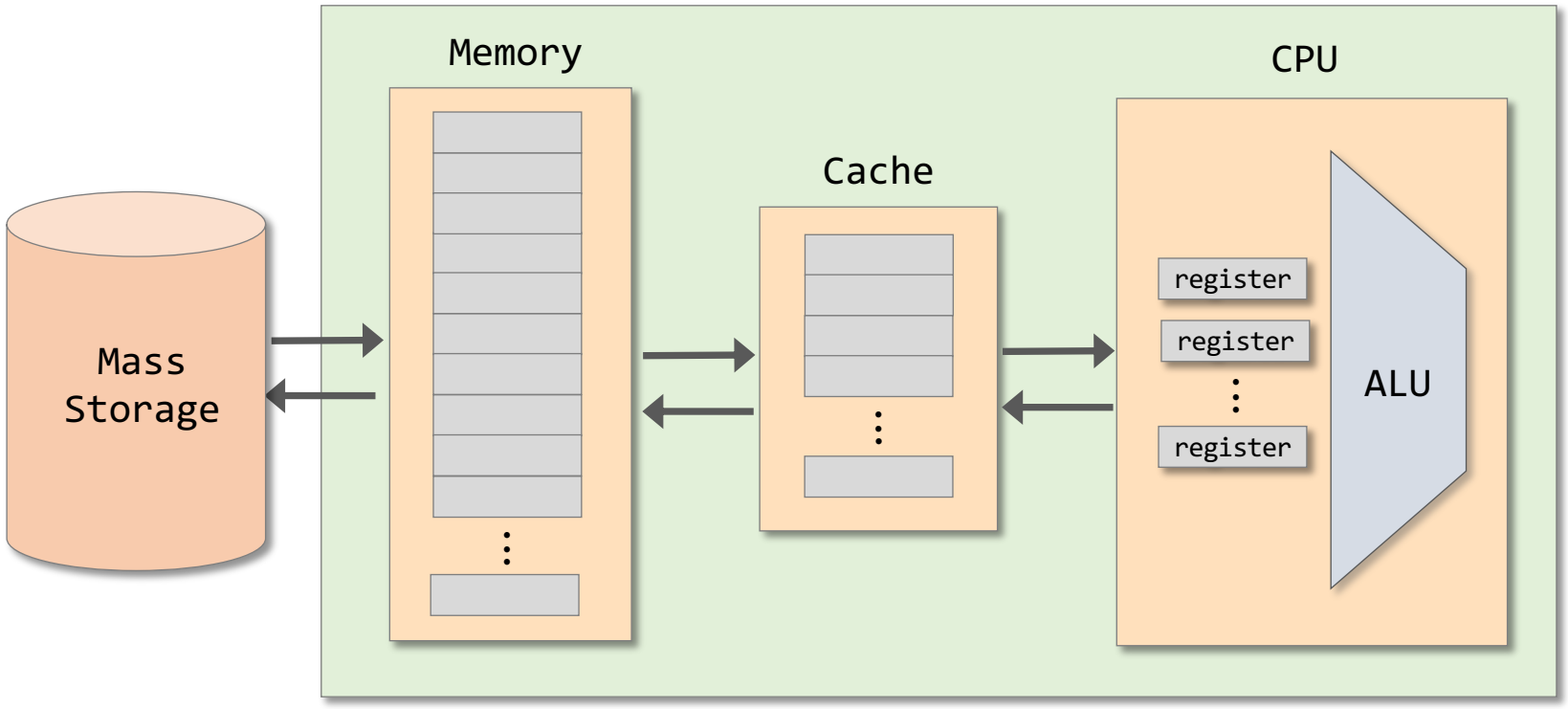
End note: Memory hierarchy



Memory hierarchy:

more storage space, slower access time

End note: Memory hierarchy



- Mass storage and cache memory are possible *extensions*
- The basic Hack computer is built without them.