Lecture 5

Computer Architecture

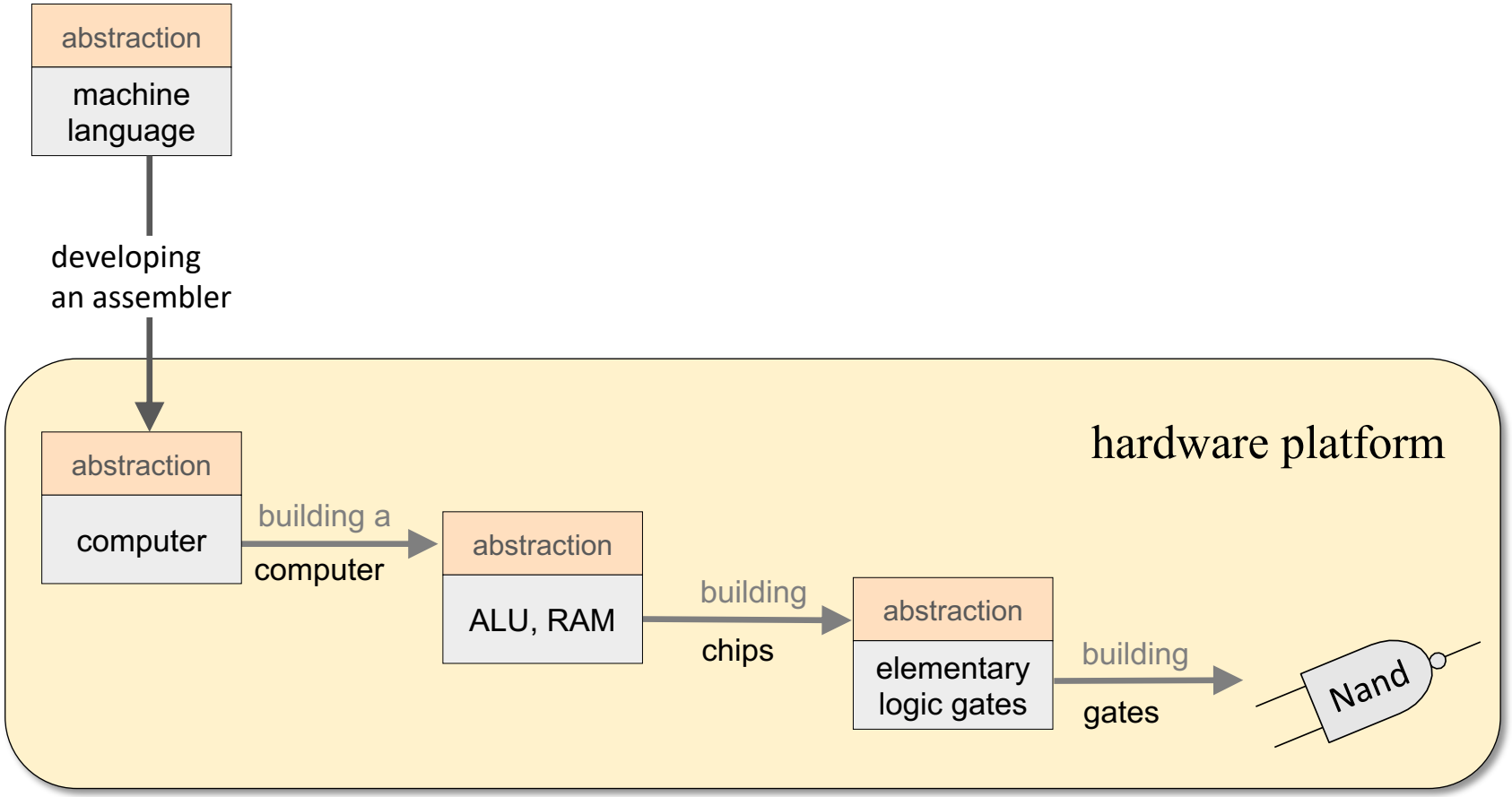
These slides support chapter 5 of the book

The Elements of Computing Systems

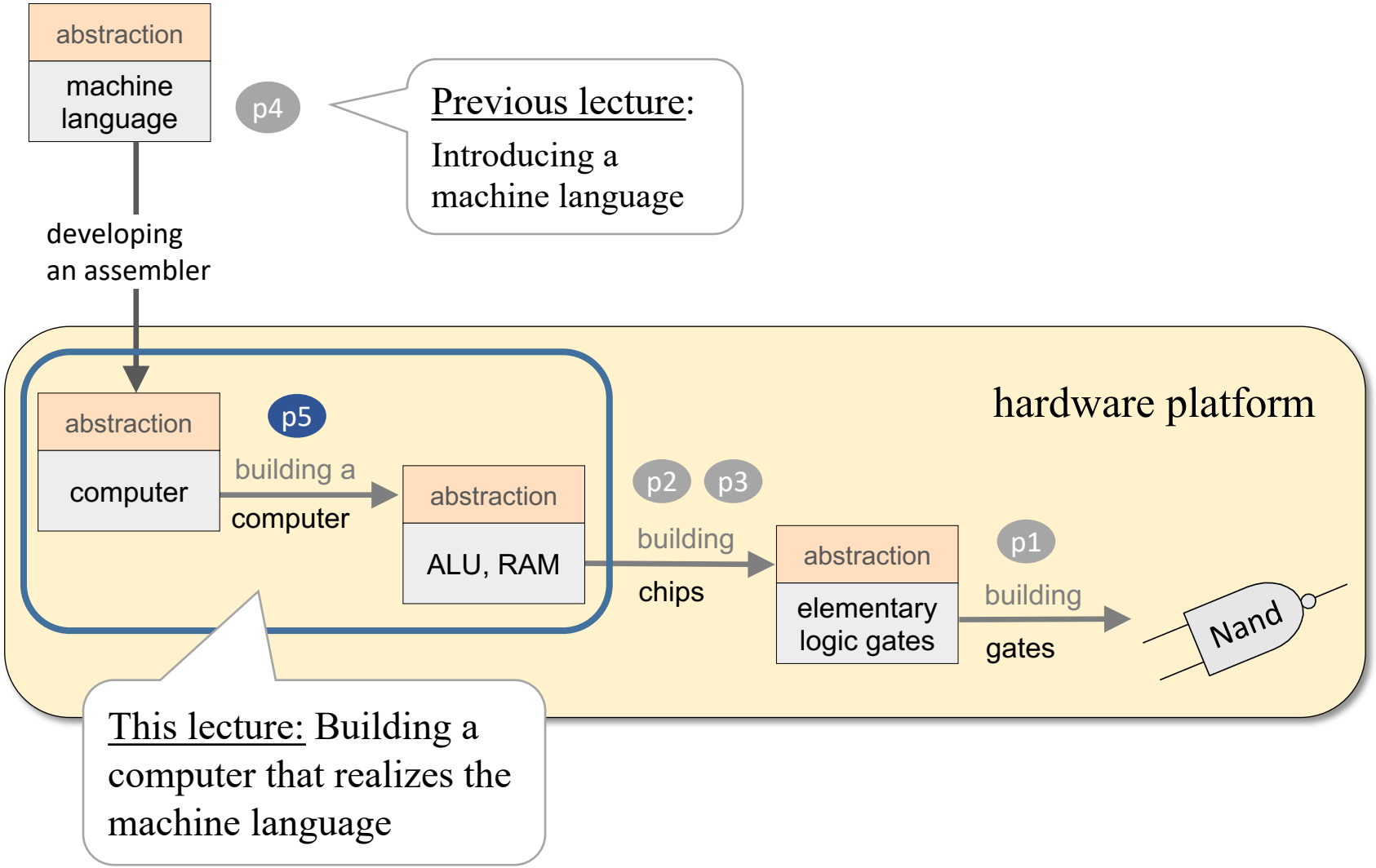
By Noam Nisan and Shimon Schocken

MIT Press, 2021

Nand to Tetris Roadmap: Hardware



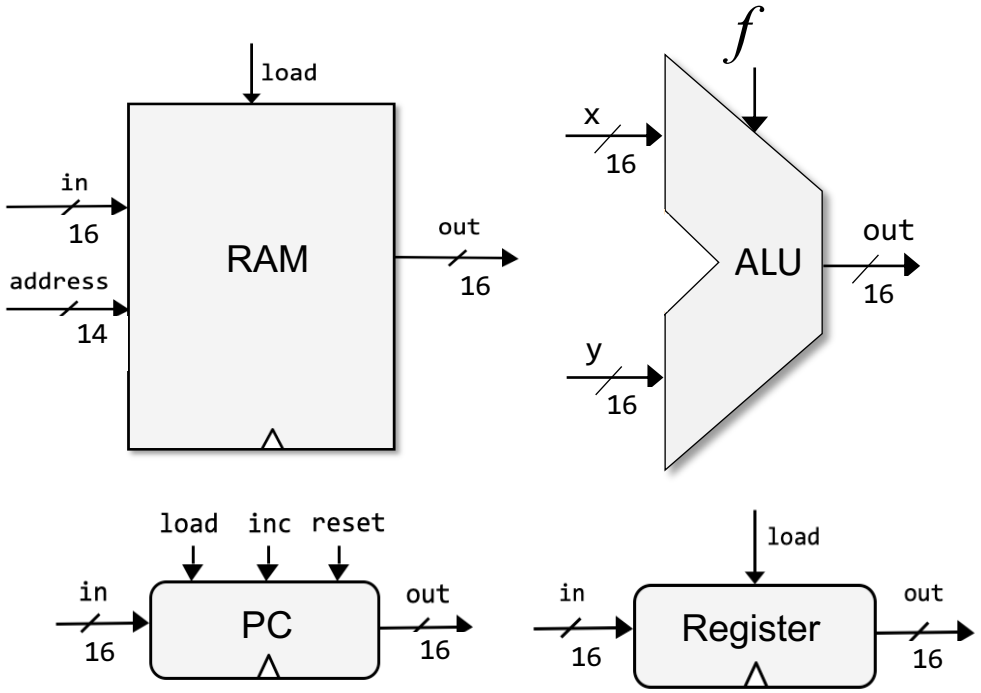
Nand to Tetris Roadmap: Hardware



Nand to Tetris Roadmap: Hardware

The challenge

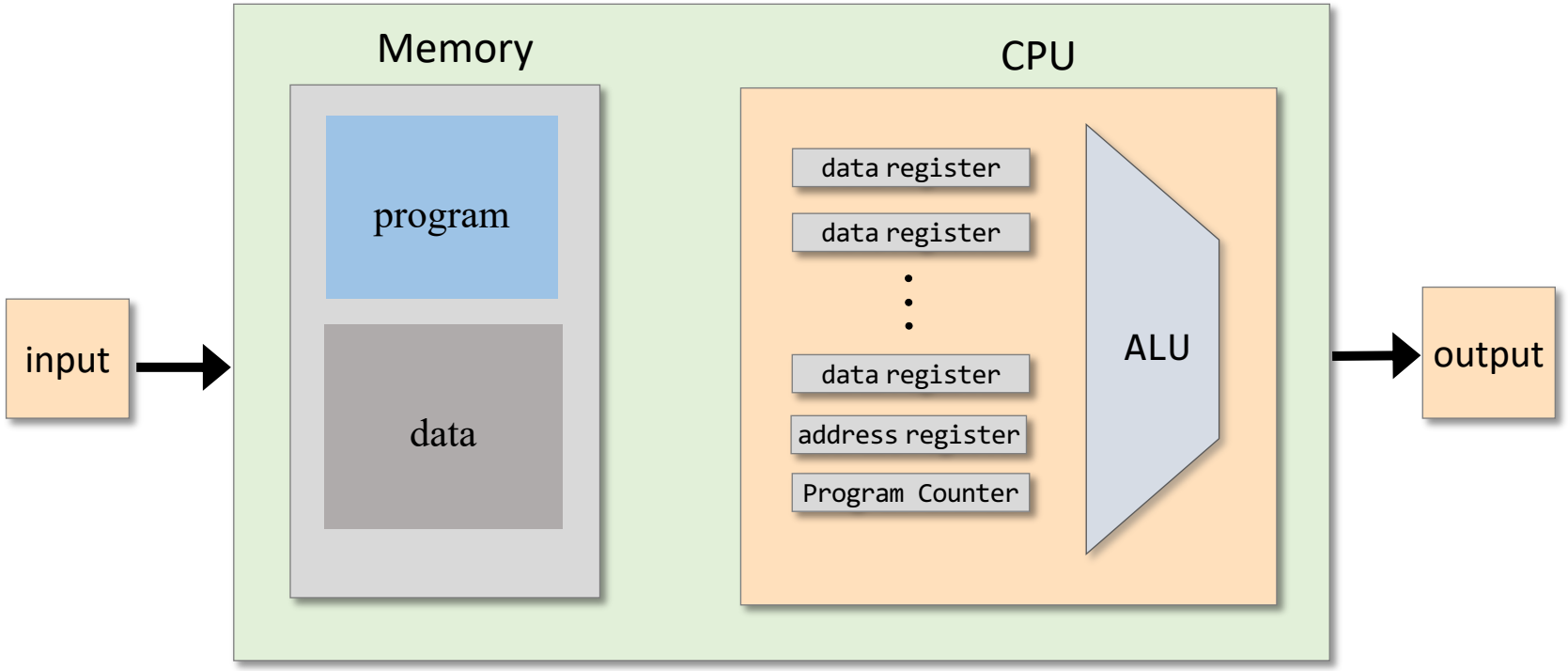
Integrate the chips built in chapters 1, 2, 3 into an architecture that…



… executes any program written in the Hack machine language introduced in lecture 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



Typical characteristics

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

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

Computer Architecture

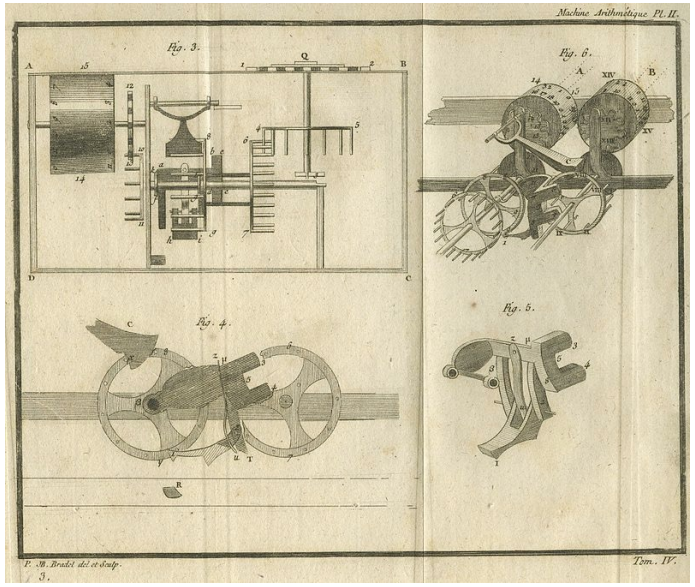
- Basic architecture
- Fetch-Execute cycle
- The Hack CPU
- Input / output
- Memory
- Computer
- Project 5: Chips
- Project 5: Guidelines

Early computers: 17th century

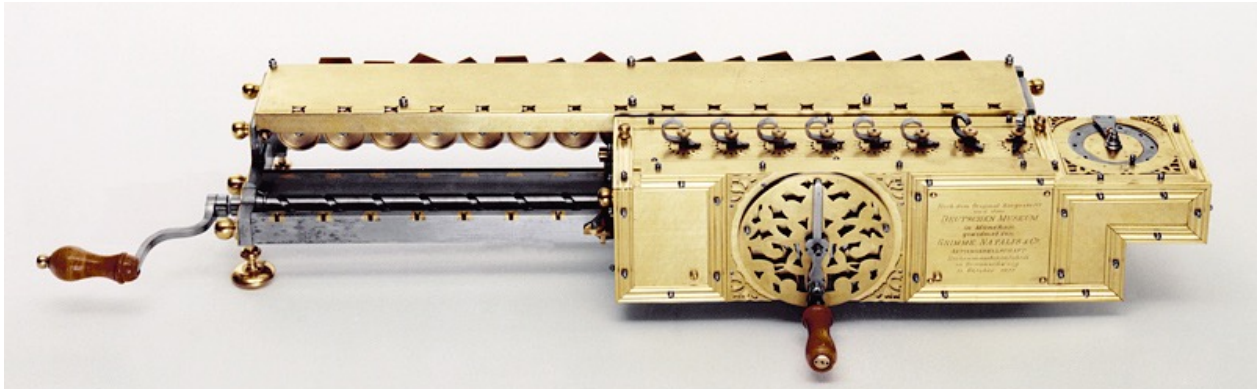
Blaise Pascal
1623 – 1662

Pascal's Calculator
(*Pascaline*, 1652)

- Add
- Subtract



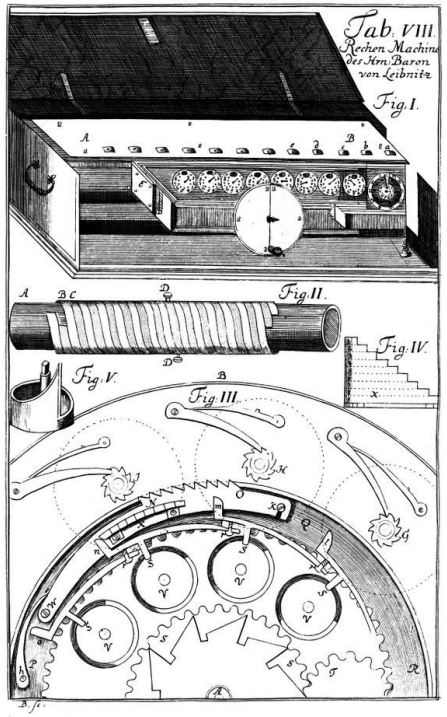
Early computers: 17th century



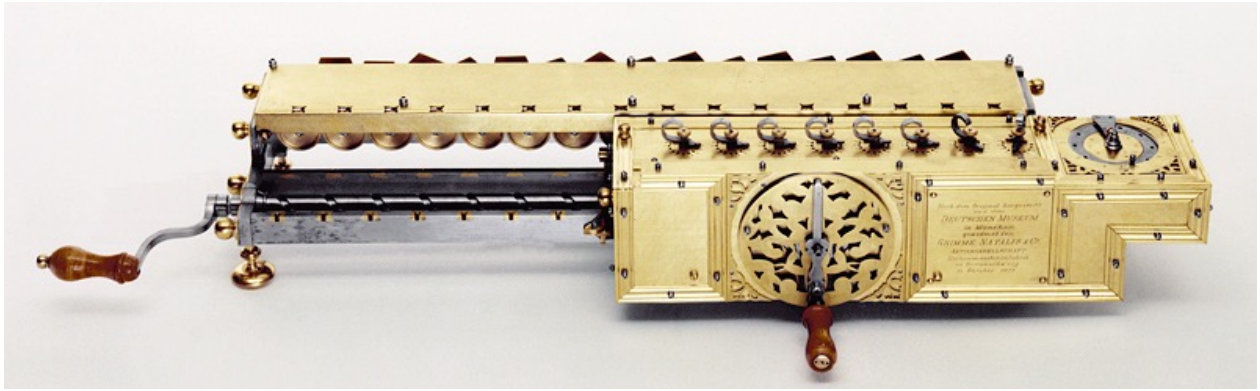
Gottfried Leibniz
1646 – 1716

Leibniz Calculator (1673)

- Add
- Subtract
- Multiply
- Divide.



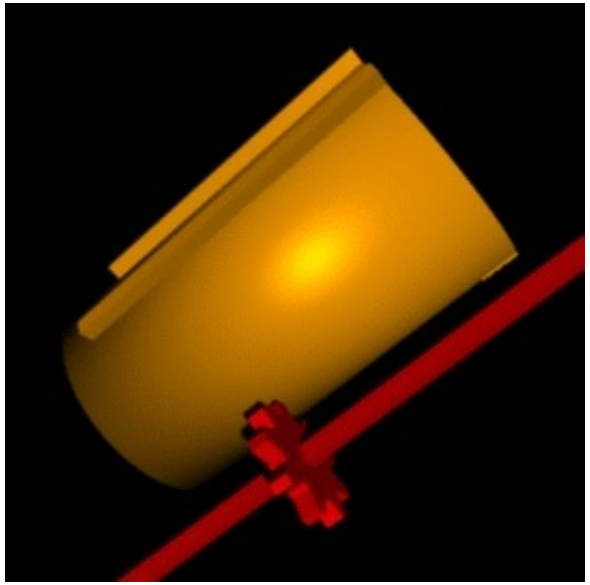
Early computers: 17th century



Gottfried Leibniz
1646 – 1716

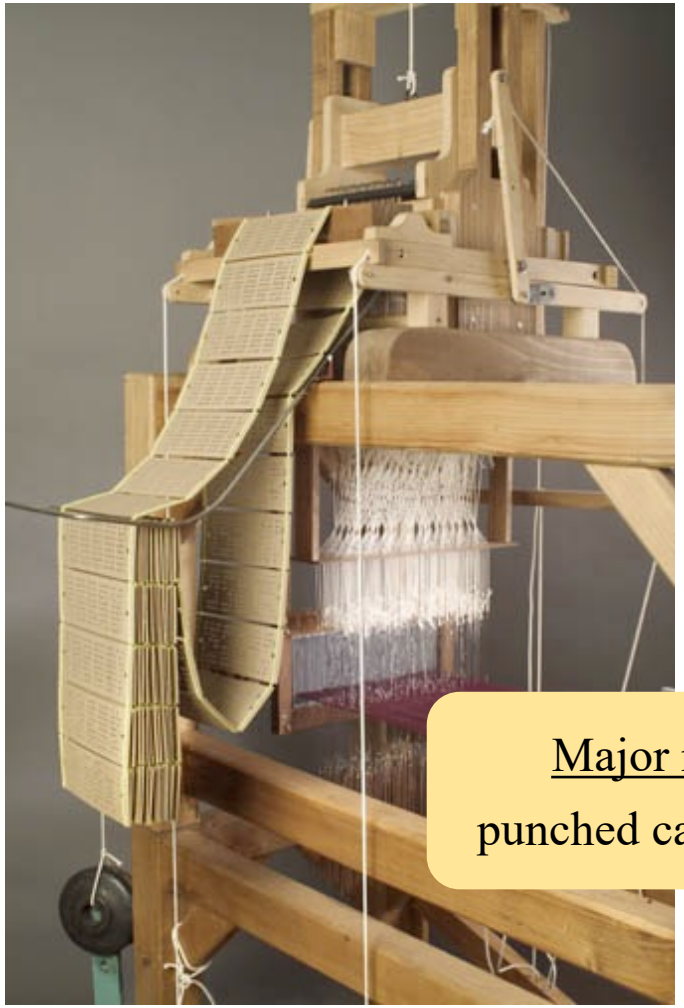
Leibniz Calculator (1673)

- Add
- Subtract
- Multiply
- Divide.



Side benefits:
Advances in gears / mechanical engineering

Early computers: 19th century

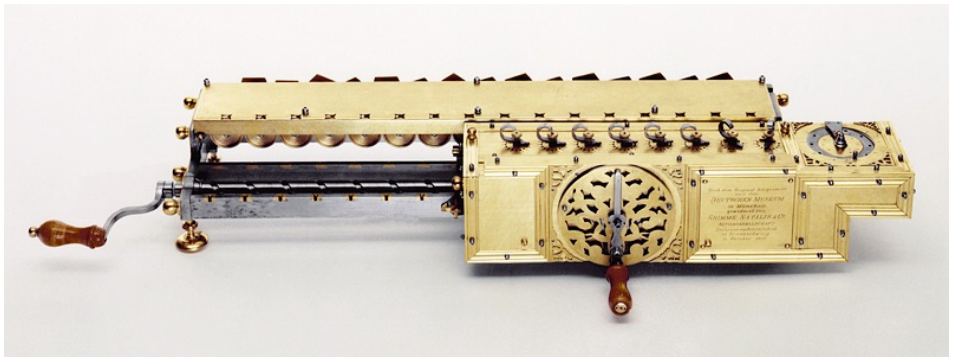


mechanical loom
(Jacquard, 1804)

mechanical calculator
(Babbage, 1837)

Major milestone:
punched cards / software

Early computers (recap)



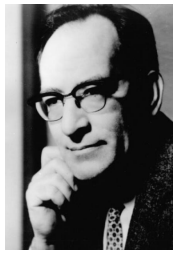
17th century: Hardware only / fixed / single purpose

Programmable!

19th century: Hardware / software / general purpose

Modern computers: 20th century


John Von Neumann

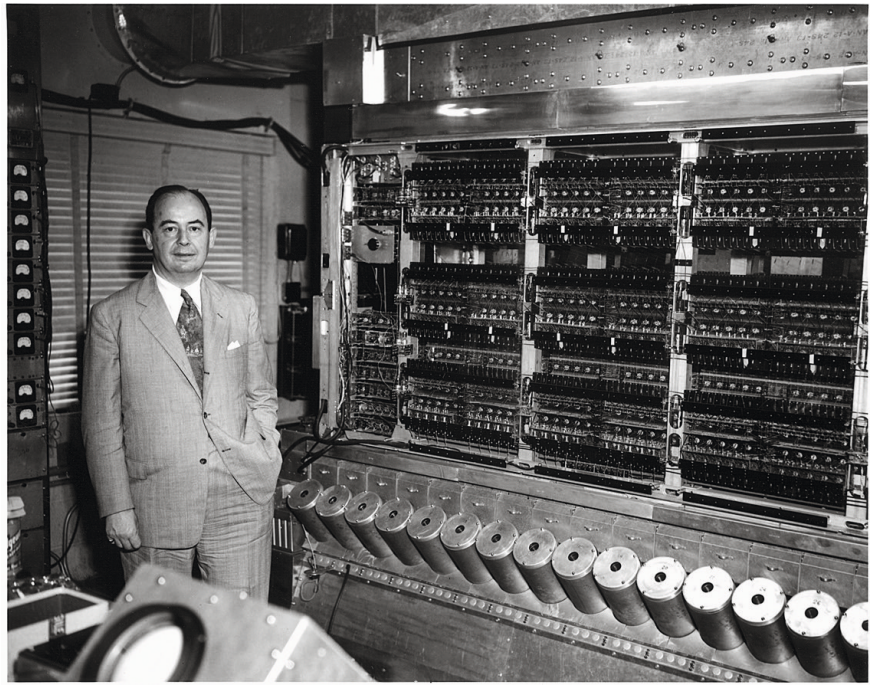

John Mauchly

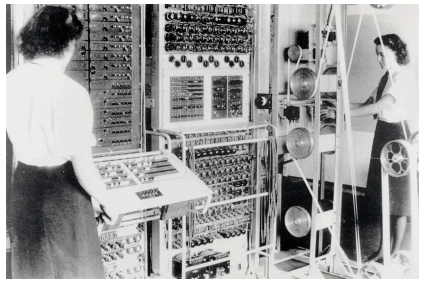

Presper Eckert


John Atanasoff


Howard Aiken


Konrad Zuse






Tommy Flowers

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

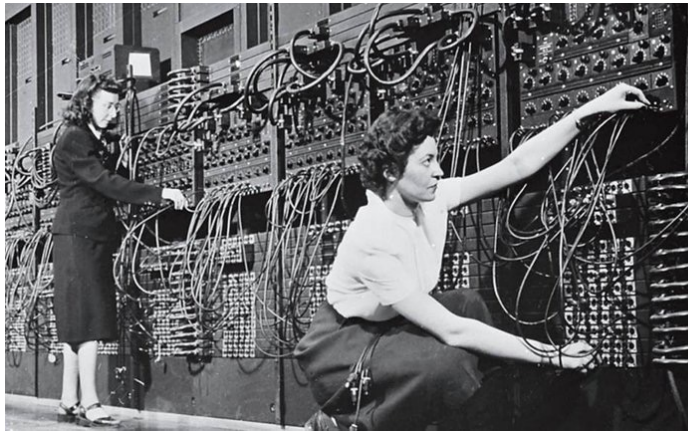
ENIAC: First digital, programmable, stored program computer

University of Pennsylvania, 1946,

(Inspired by many 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

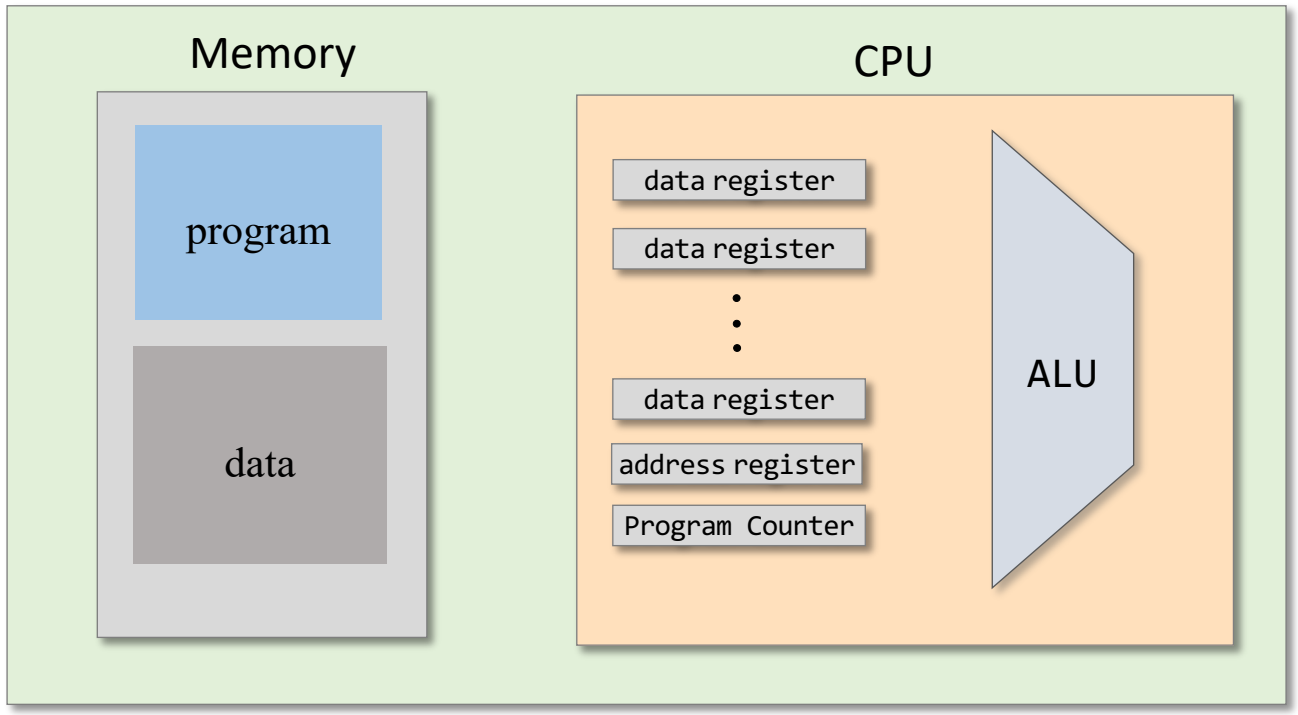
Modern computers: 20th century

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

"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)

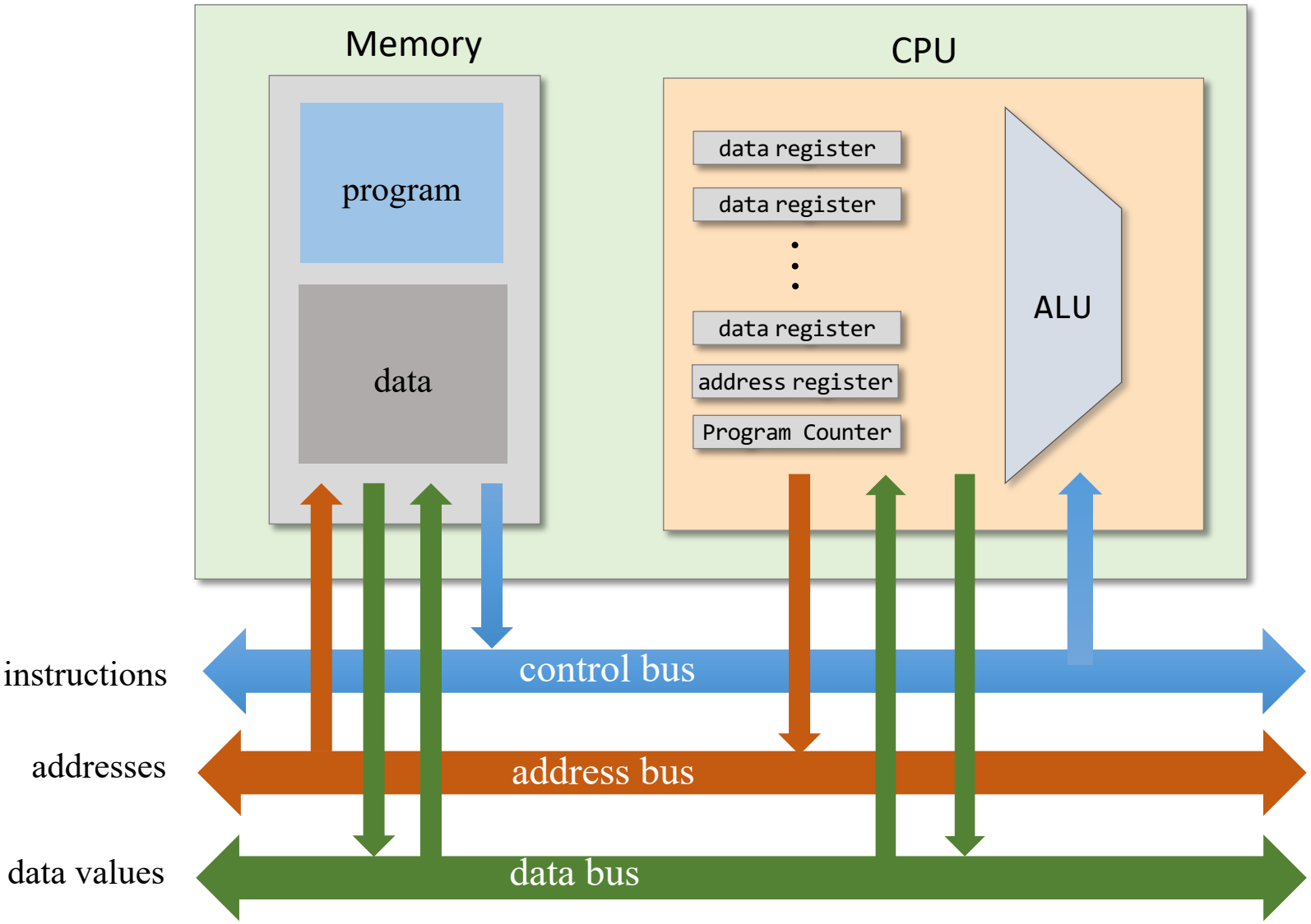
Basic architecture



Computer:

A machine that uses instructions to manipulates data

Basic architecture

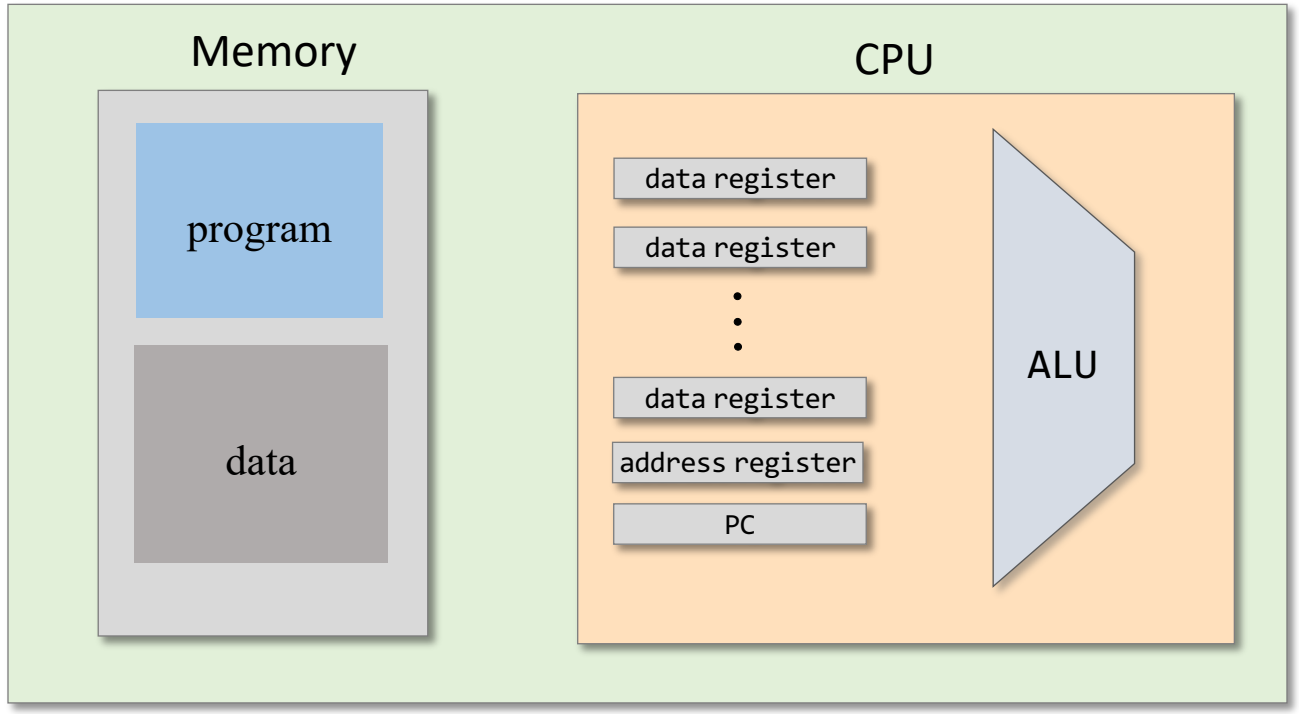


The computer can also be viewed (technically) as a set of chips, connected by buses.

Computer Architecture

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

Computer architecture

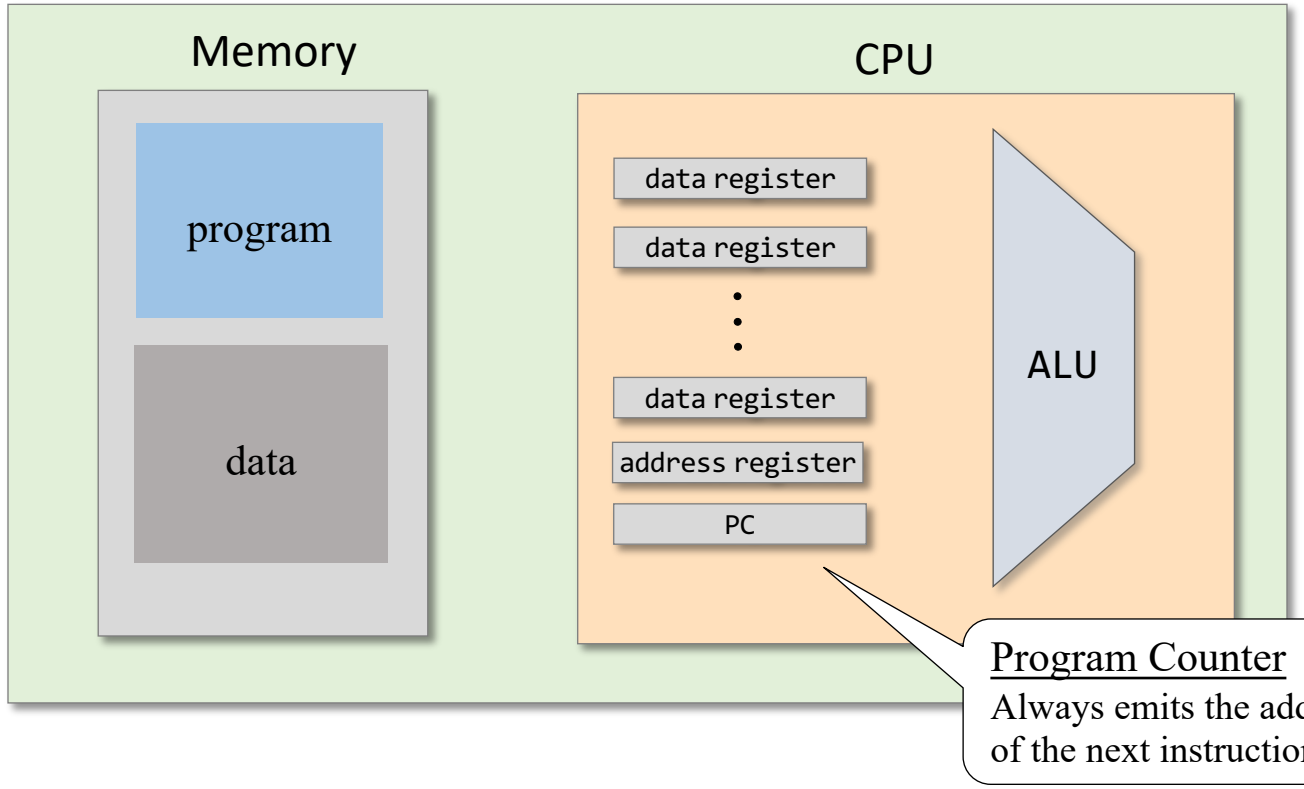


Basic loop

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

  (and figure out the
  address of the next
  instruction)

Computer architecture

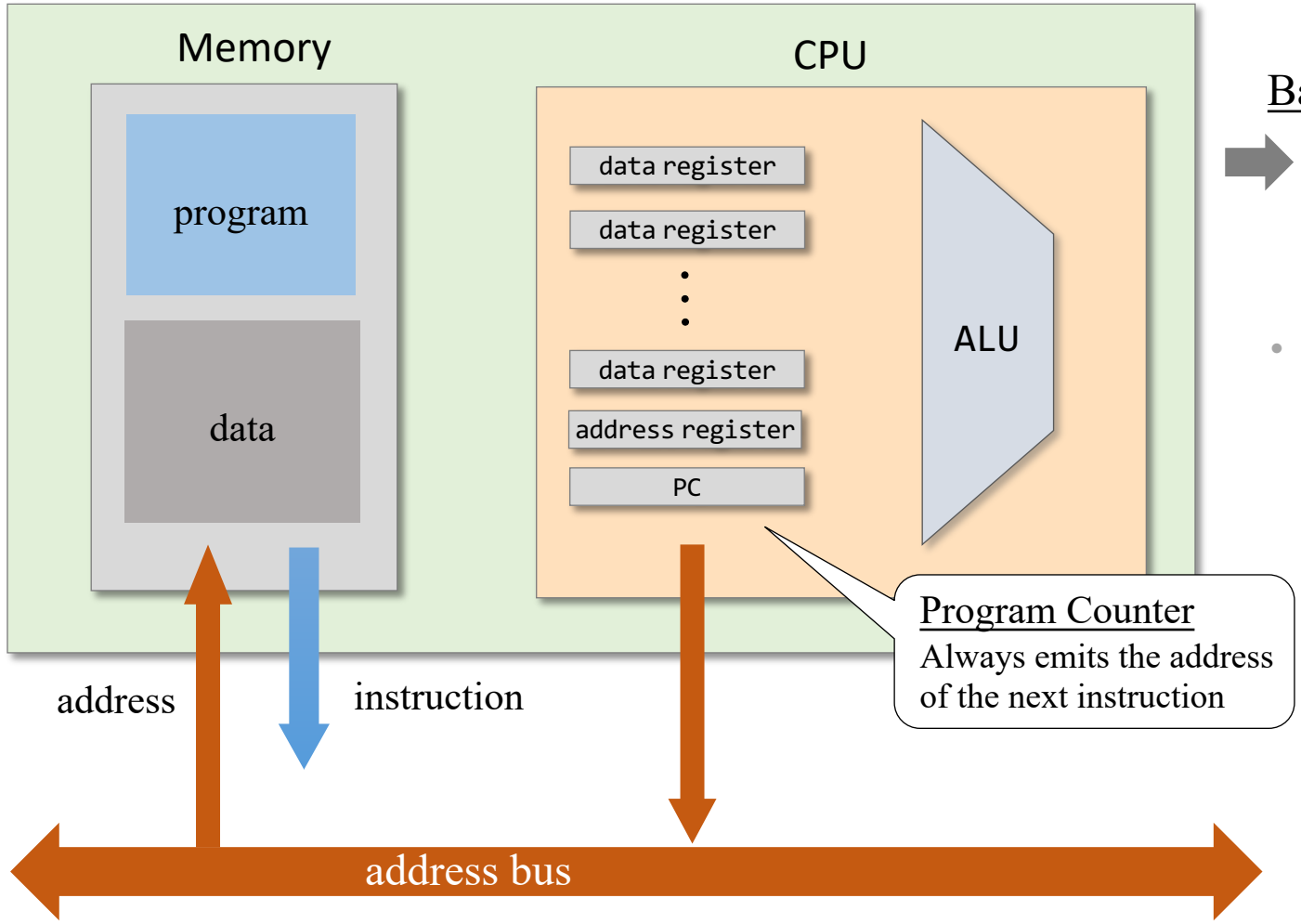


Basic loop

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

  (and figure out the address of the next instruction)

Fetch an instruction

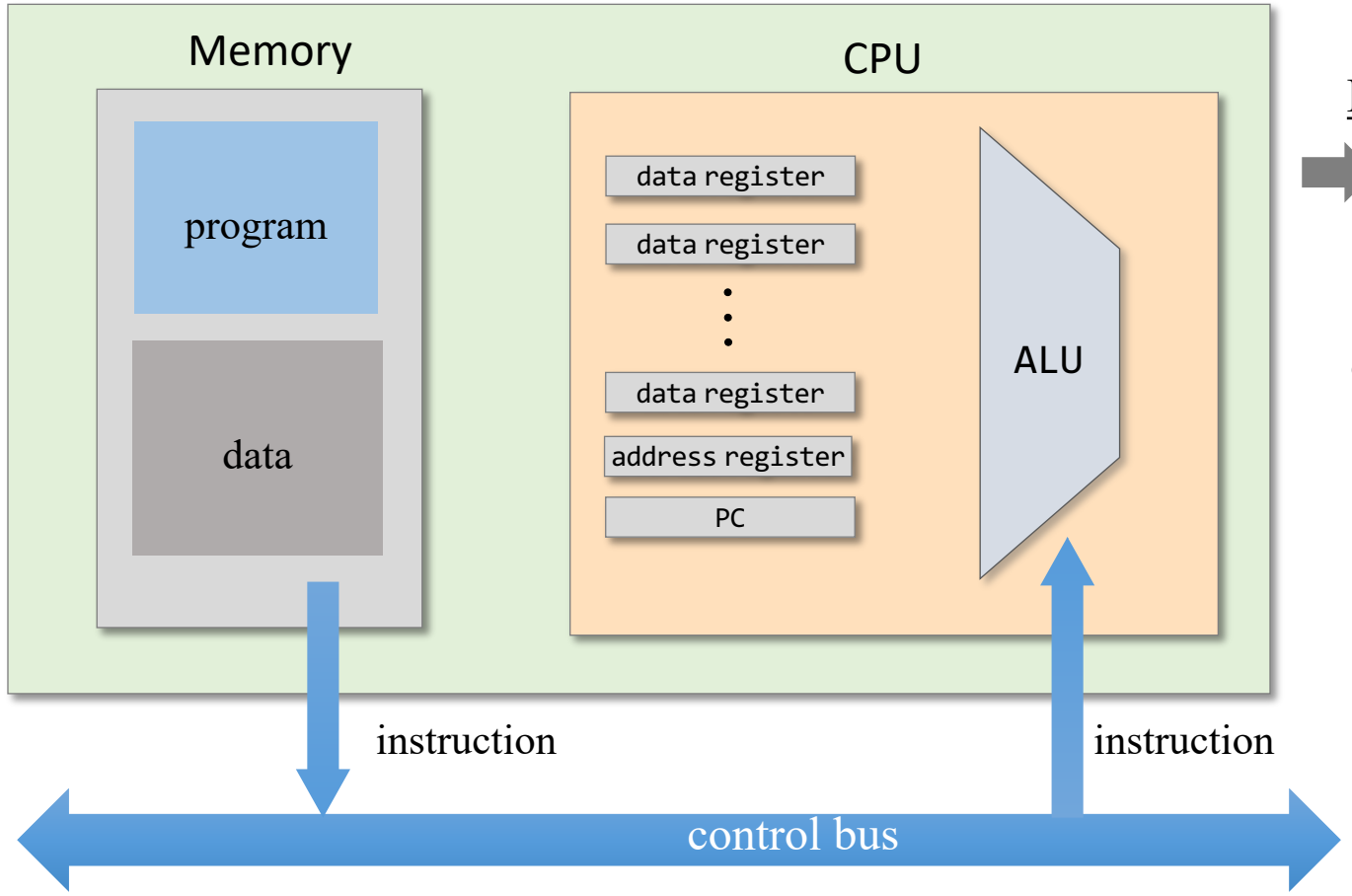


Basic loop

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

  (and figure out the address of the next instruction)

Fetch an instruction

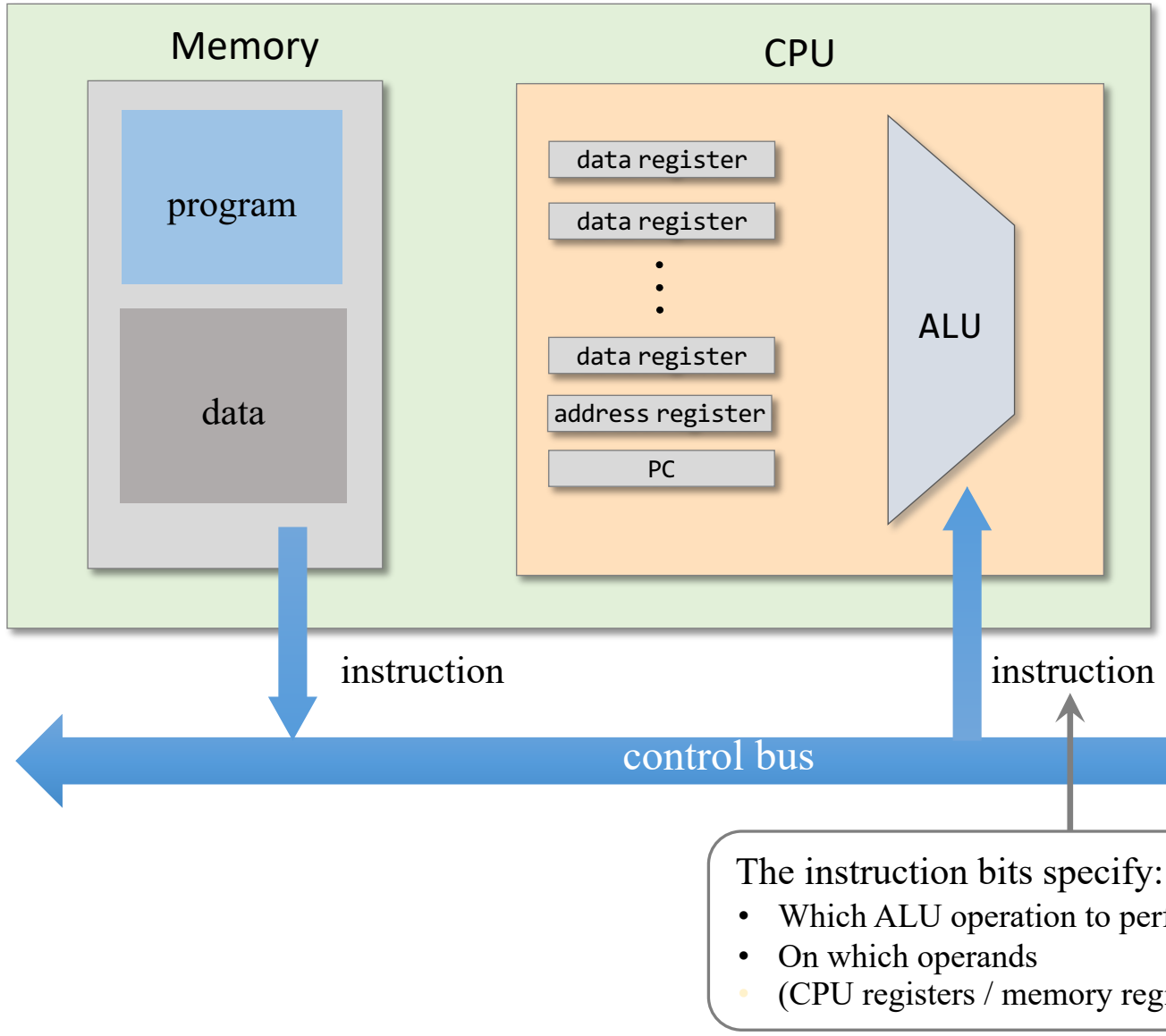


Basic loop

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

  (and figure out the address of the next instruction)

Execute the instruction

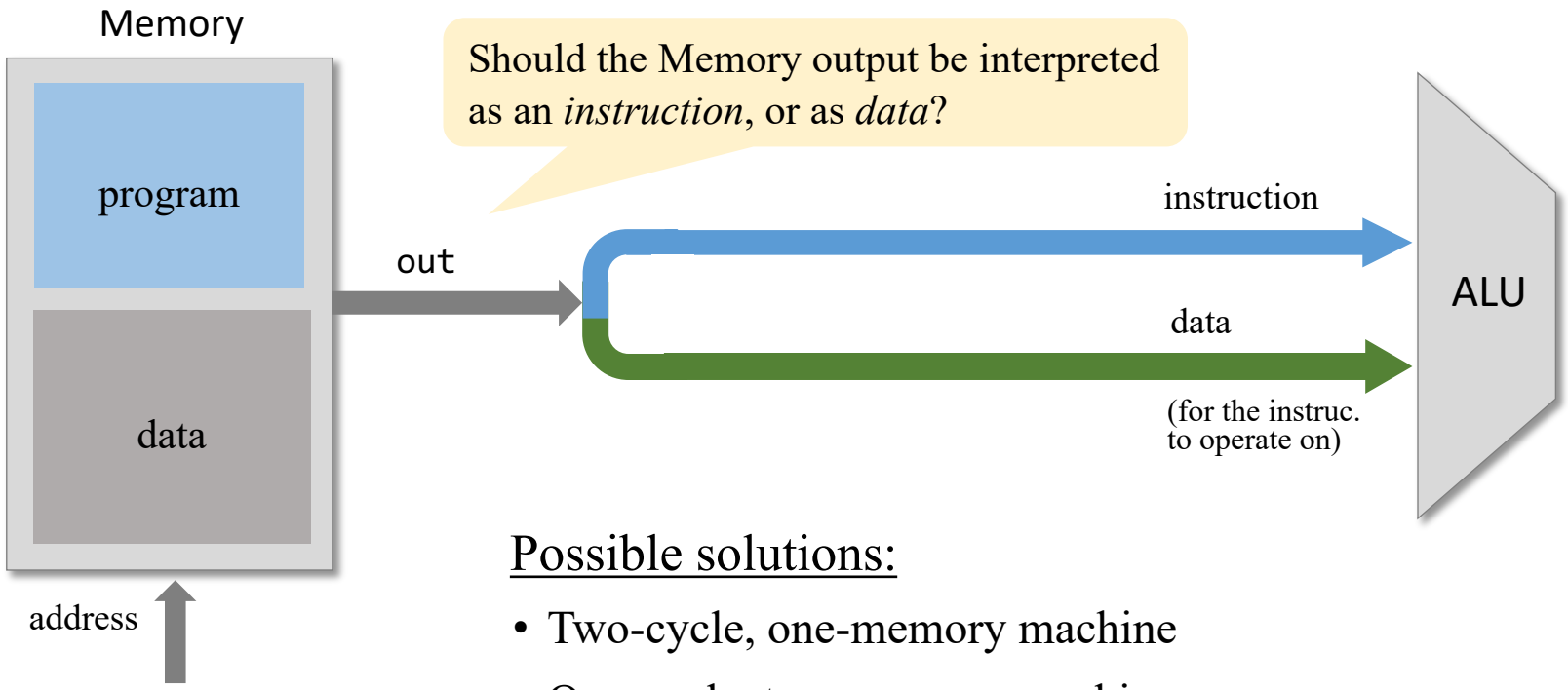


Basic loop

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

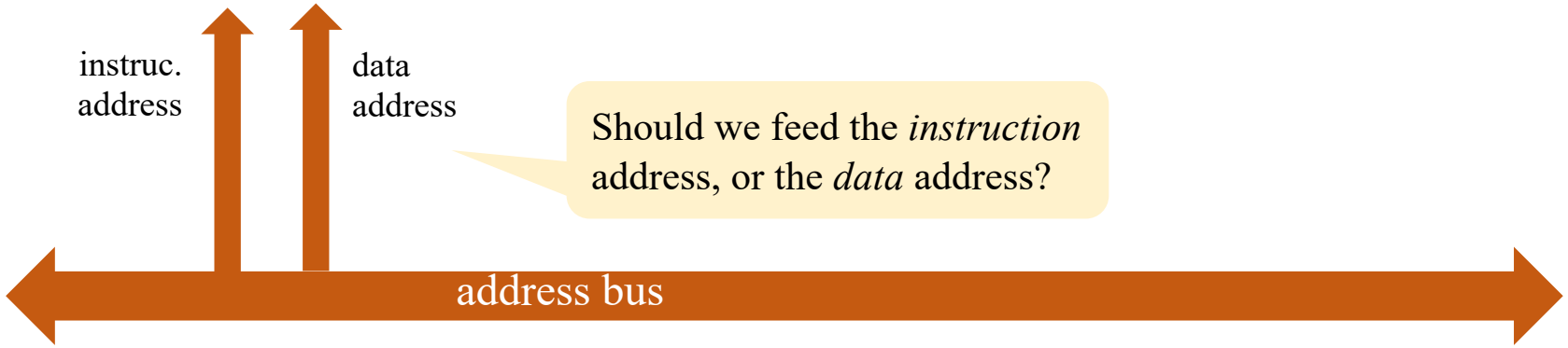
(and figure out the address of the next instruction)

Fetch – Execute issues

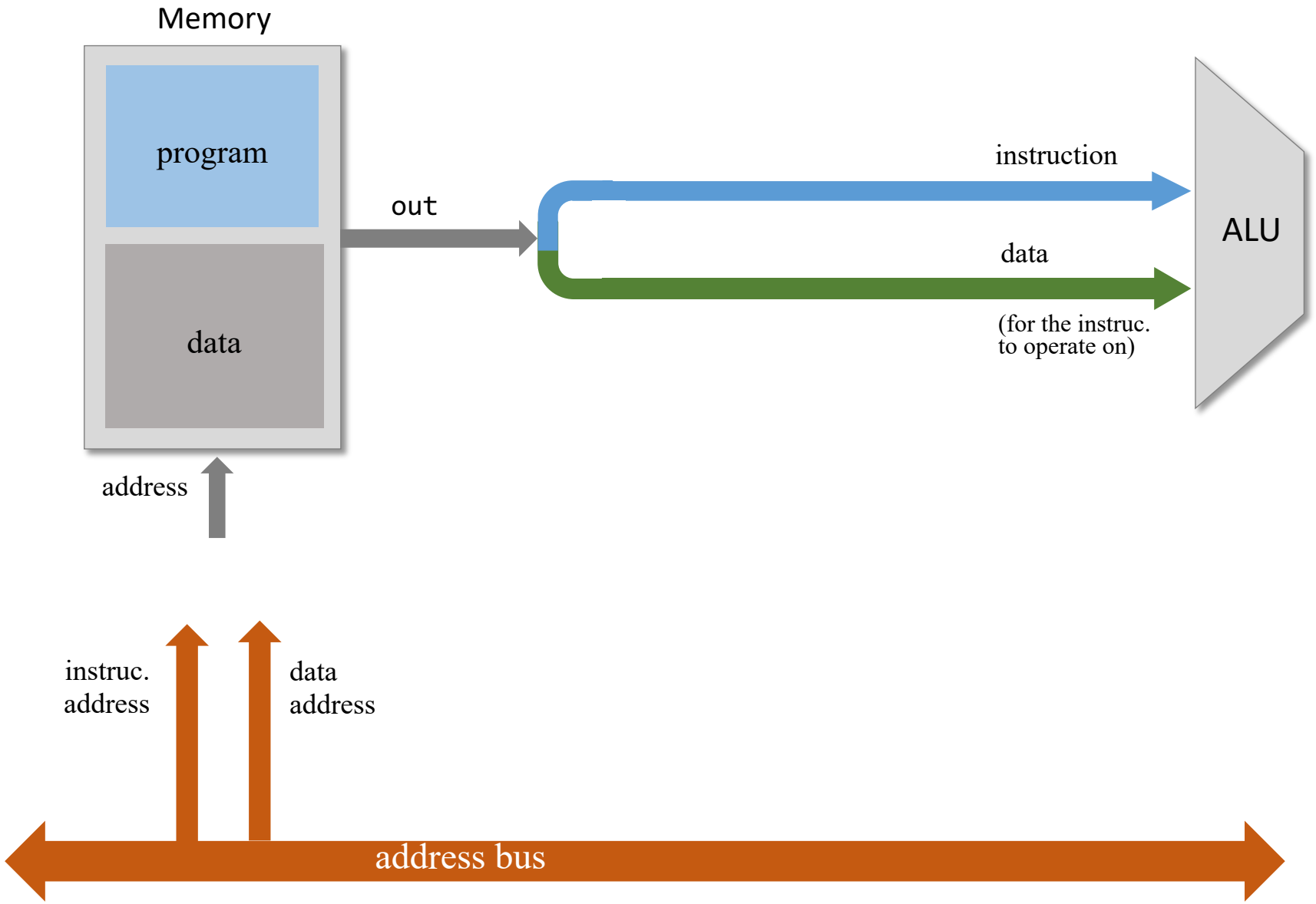


Possible solutions:

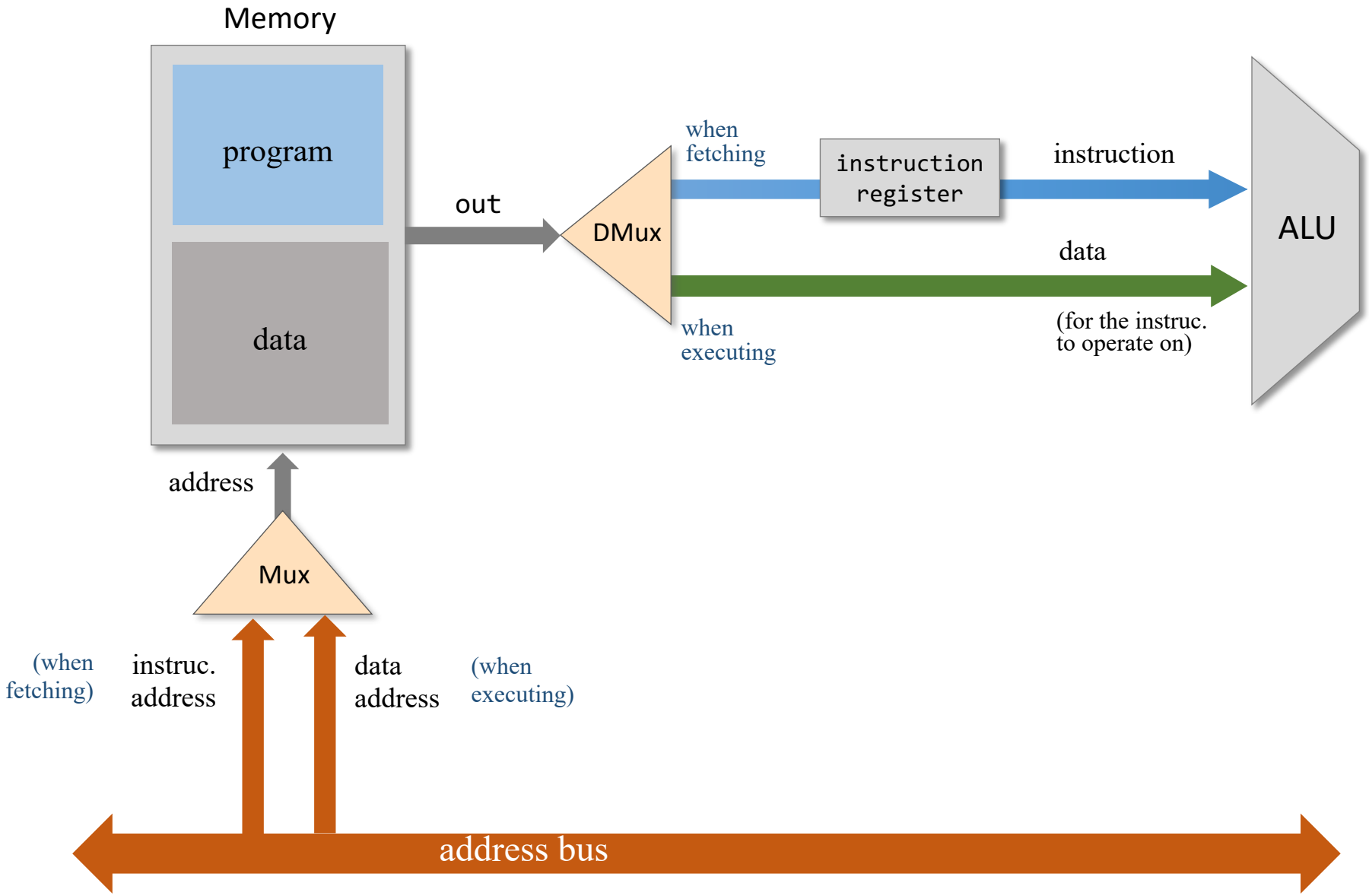
- Two-cycle, one-memory machine
- One-cycle, two-memory machine



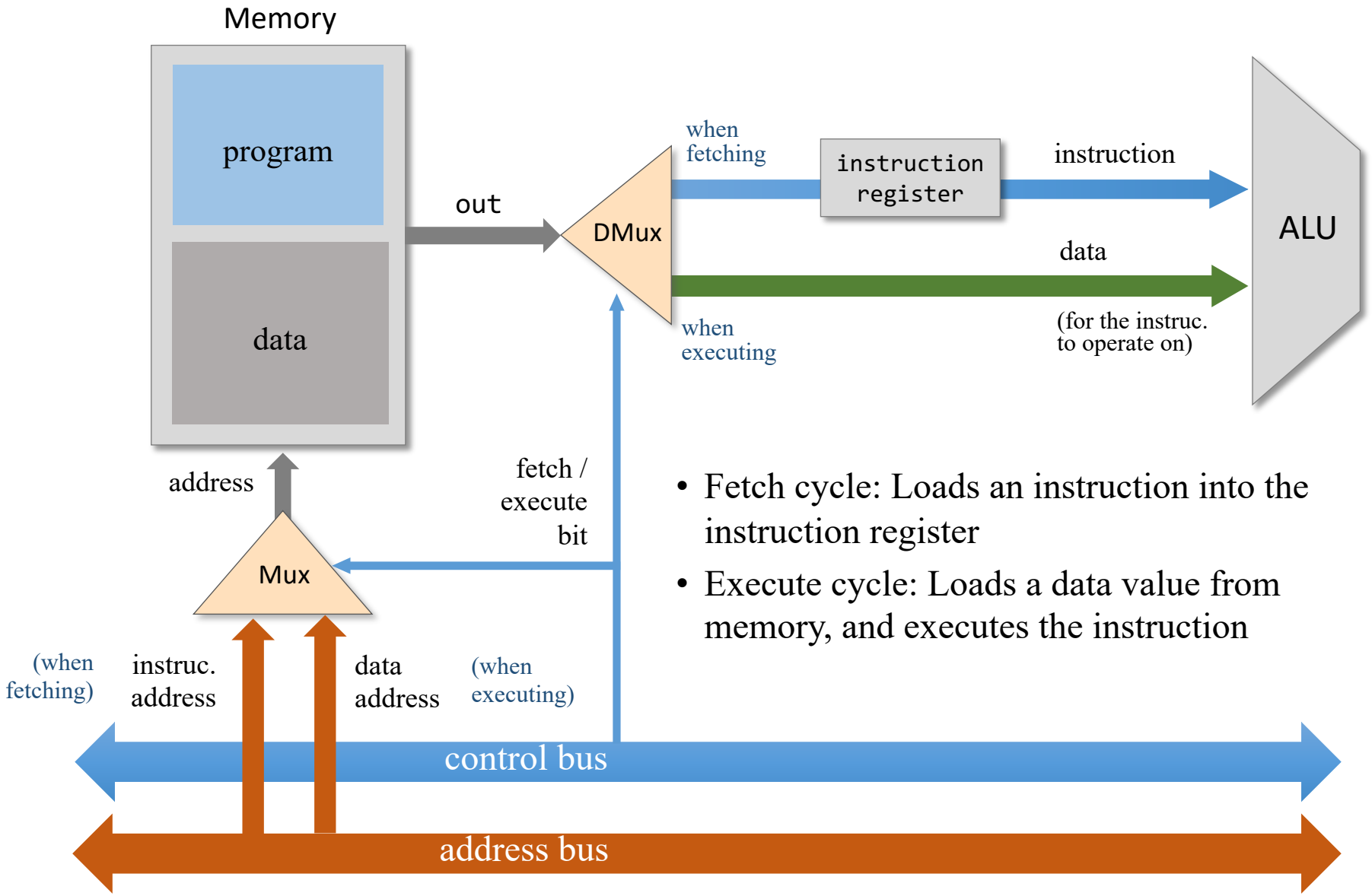
Two-cycle, one-memory machine



Two-cycle, one-memory machine

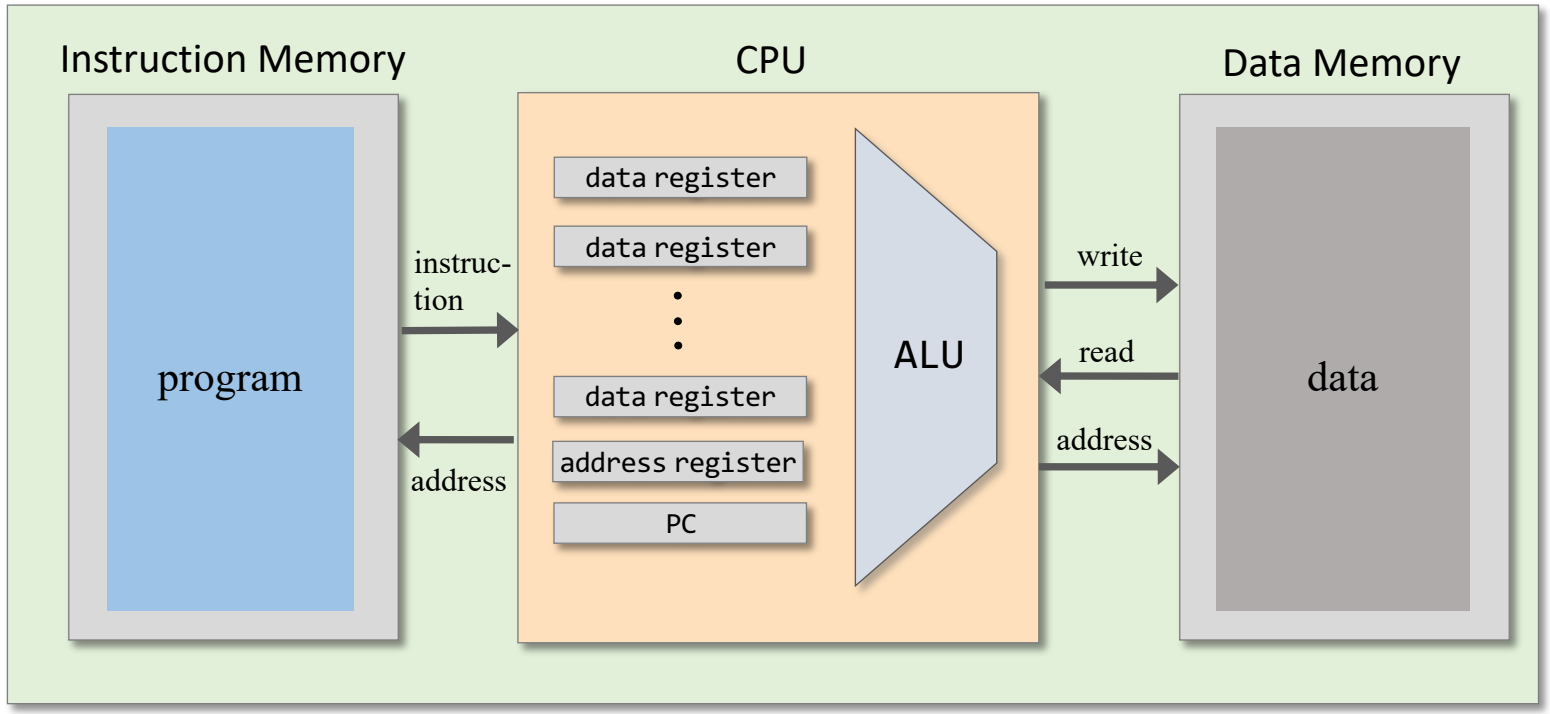


Two-cycle, one-memory machine



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

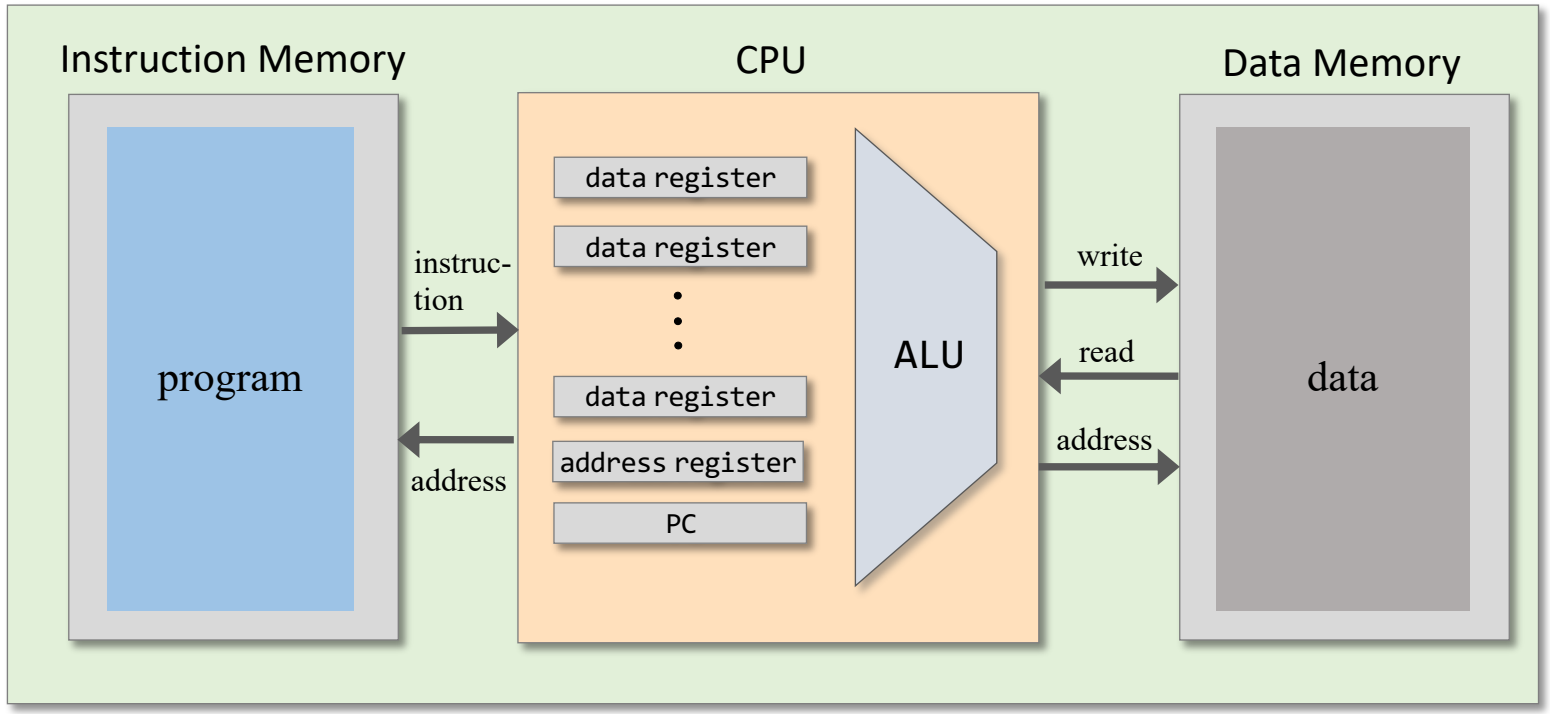
Single cycle, two-memory machine



- Instructions and data are stored in two separate physical memories
- Both memories are accessed simultaneously, in the same cycle

  (for historical reasons, sometimes called ”Harvard architecture”)

Single cycle, two-memory machine



Advantages

- Simple architecture
- Fast processing

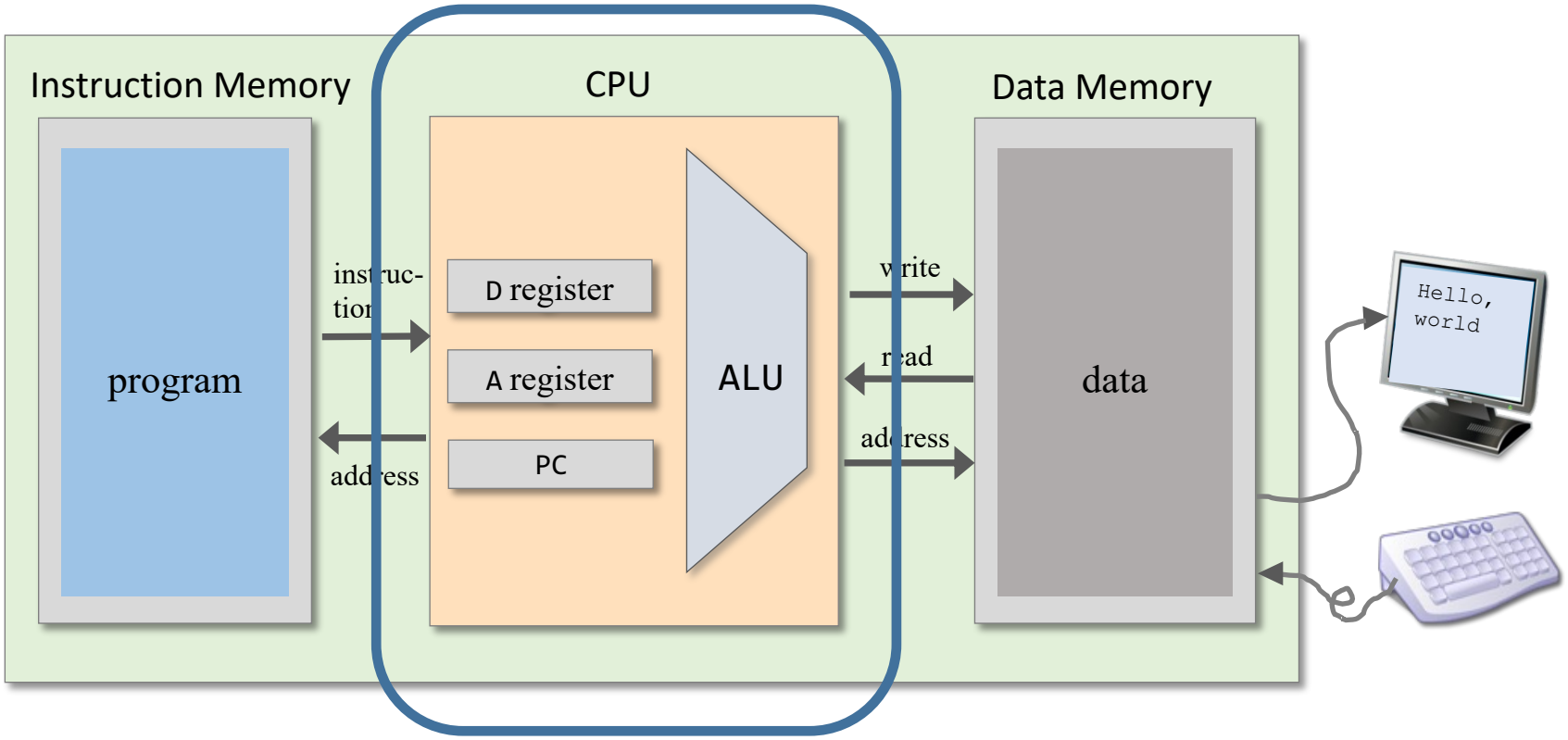
Disadvantages

- 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` *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*

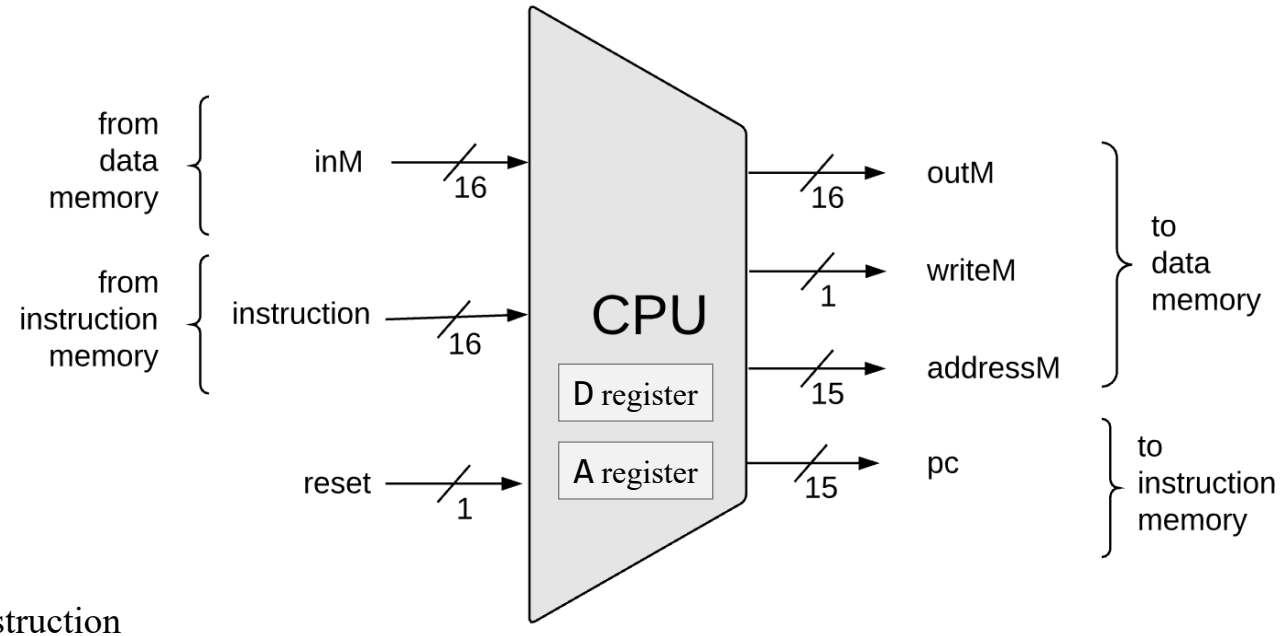
comp		*c*	*c*	*c*	*c*	*c*	*c*
0		1	0	1	0	1	0
1		1	1	1	1	1	1
-1		1	1	1	0	1	0
D		0	0	1	1	0	0
A	M	1	1	0	0	0	0
!D		0	0	1	1	0	1
!A	!M	1	1	0	0	0	1
-D		0	0	1	1	1	1
-A	-M	1	1	0	0	1	1
D+1		0	1	1	1	1	1
A+1	M+1	1	1	0	1	1	1
D-1		0	0	1	1	1	0
A-1	M-1	1	1	0	0	1	0
D+A	D+M	0	0	0	0	1	0
D-A	D-M	0	1	0	0	1	1
A-D	M-D	0	0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

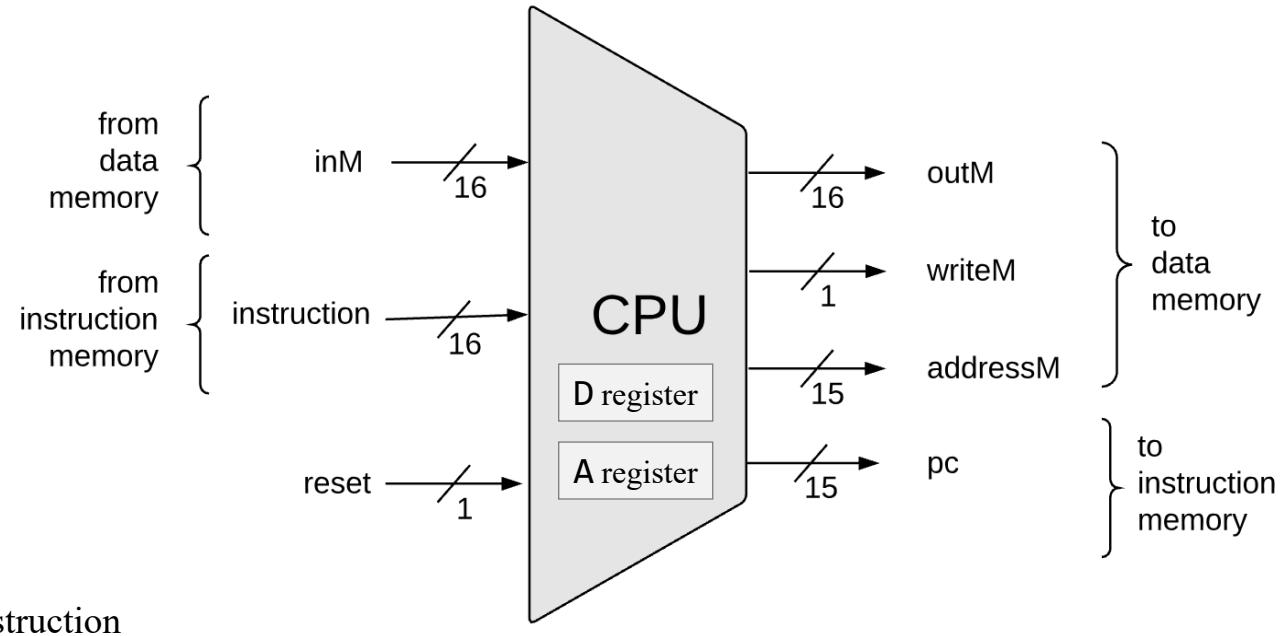


Instruction examples:

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

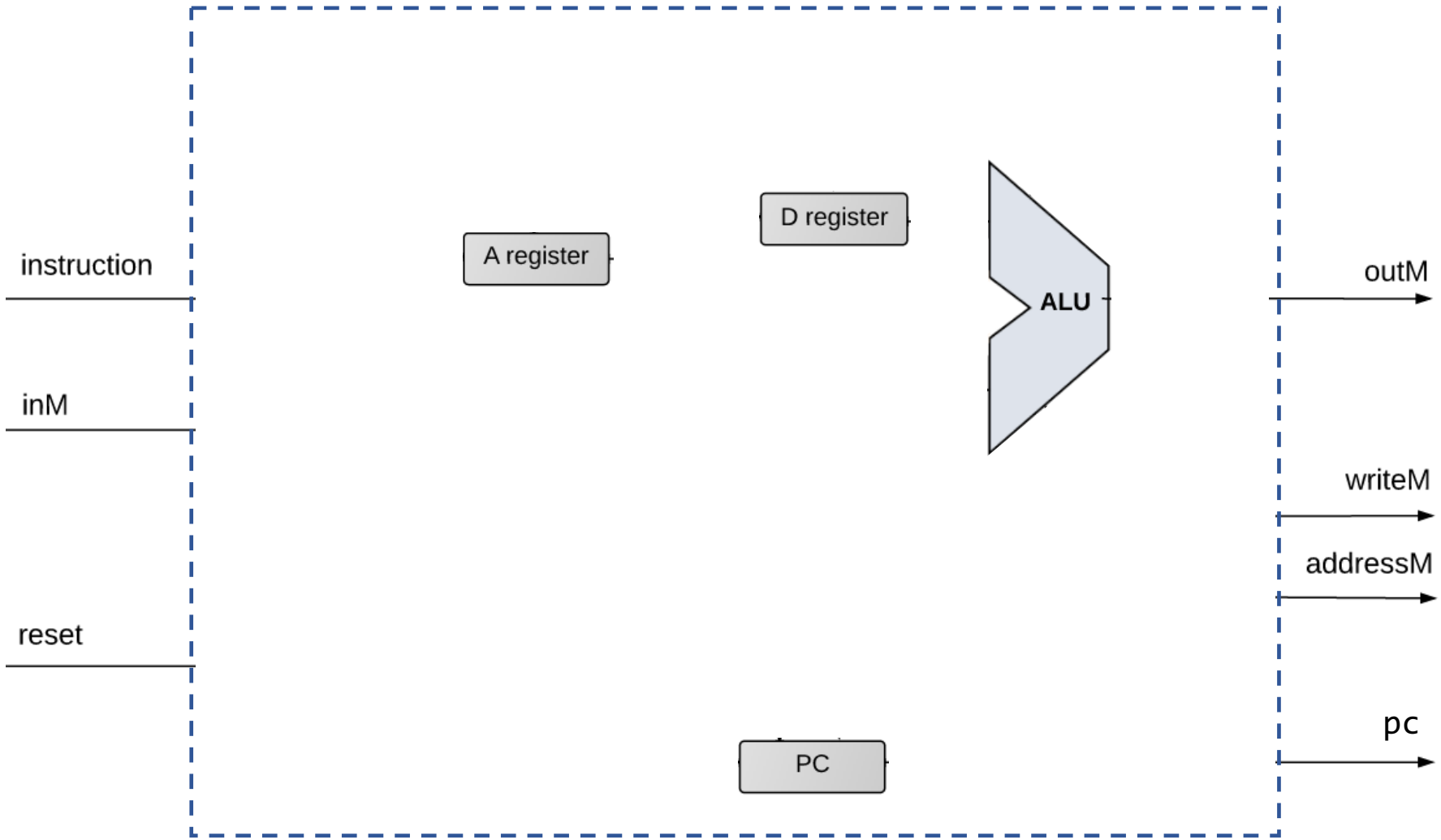
1. Executes the current instruction:

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

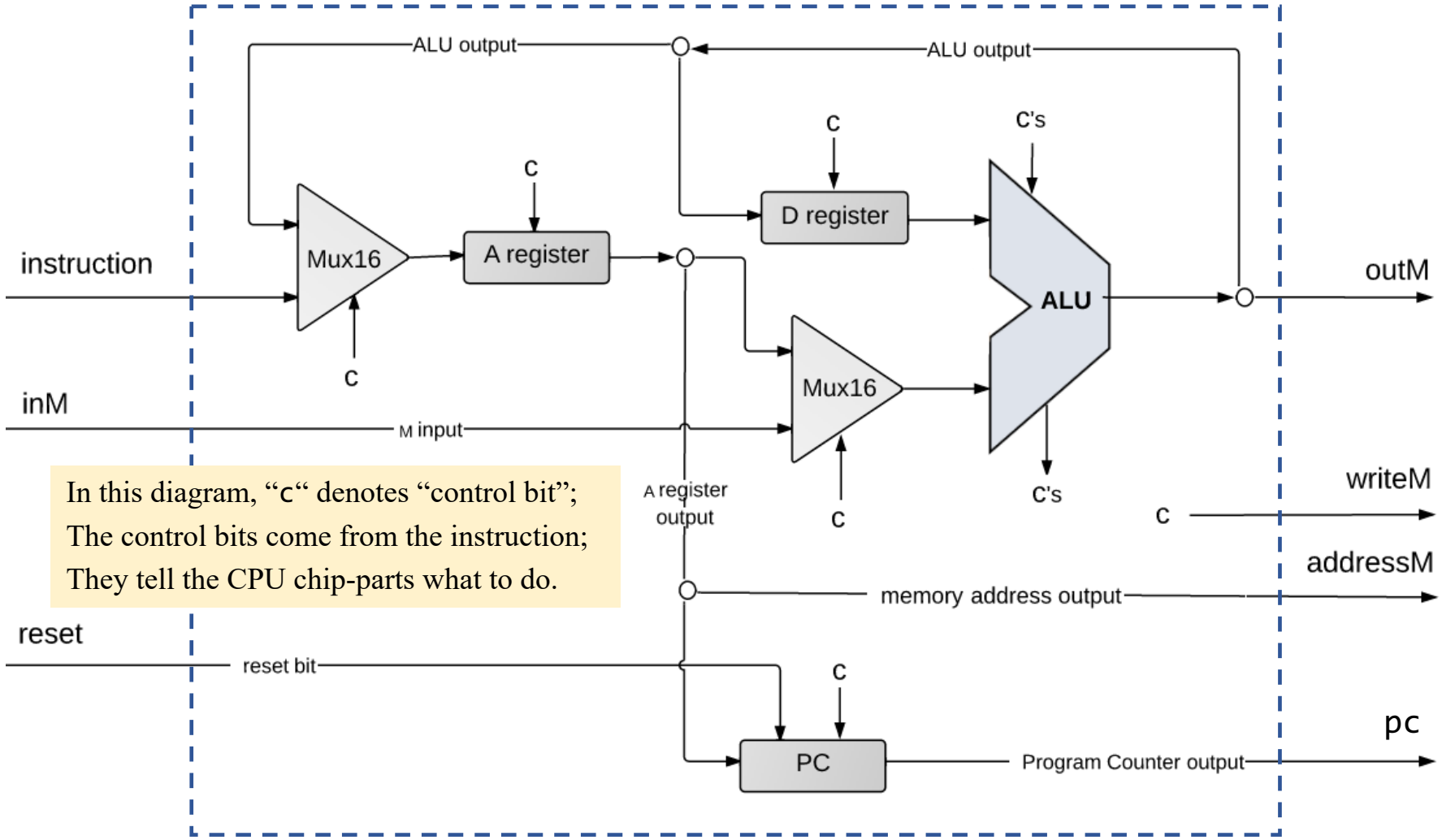
   Else (C-instruction):

   - 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

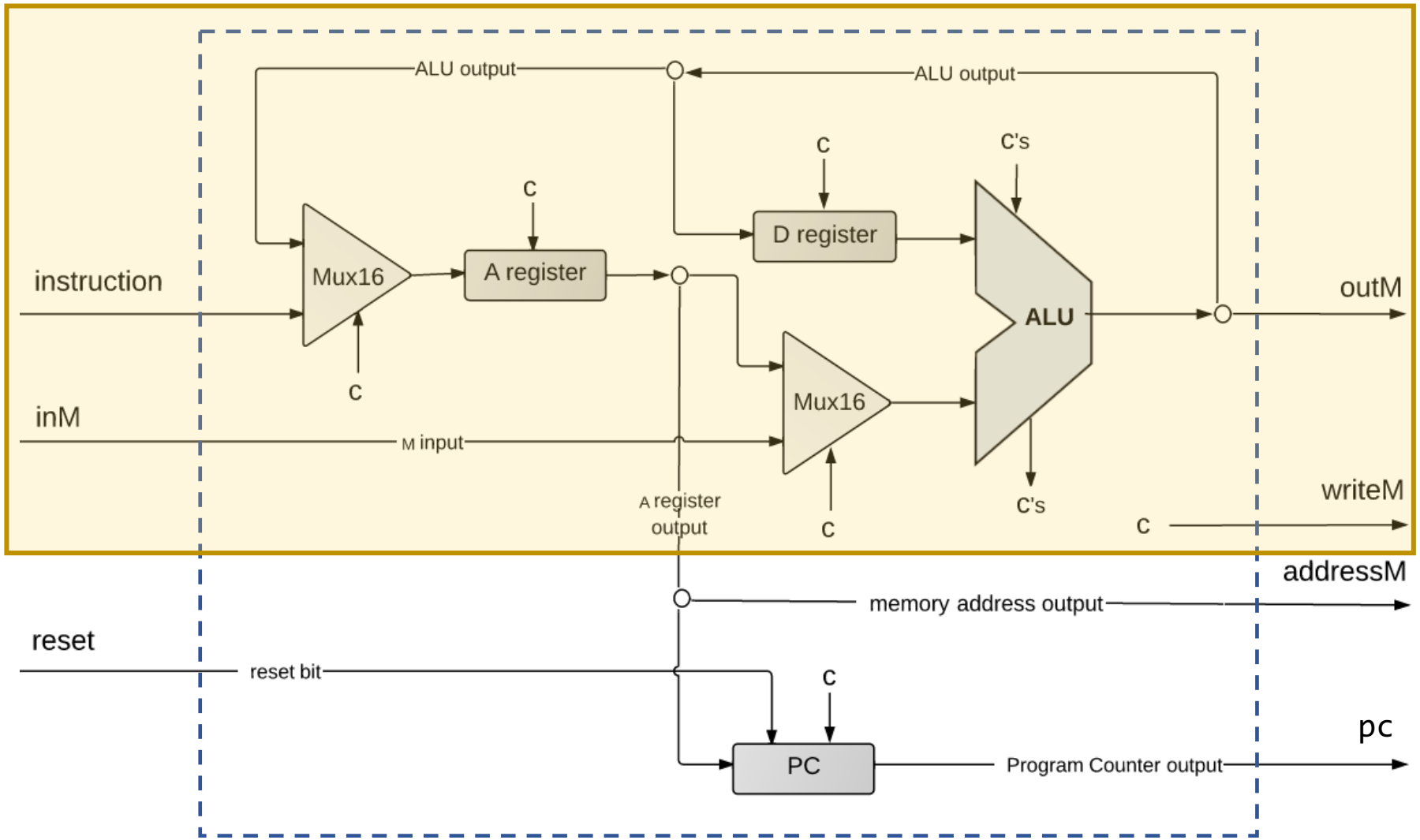


CPU implementation

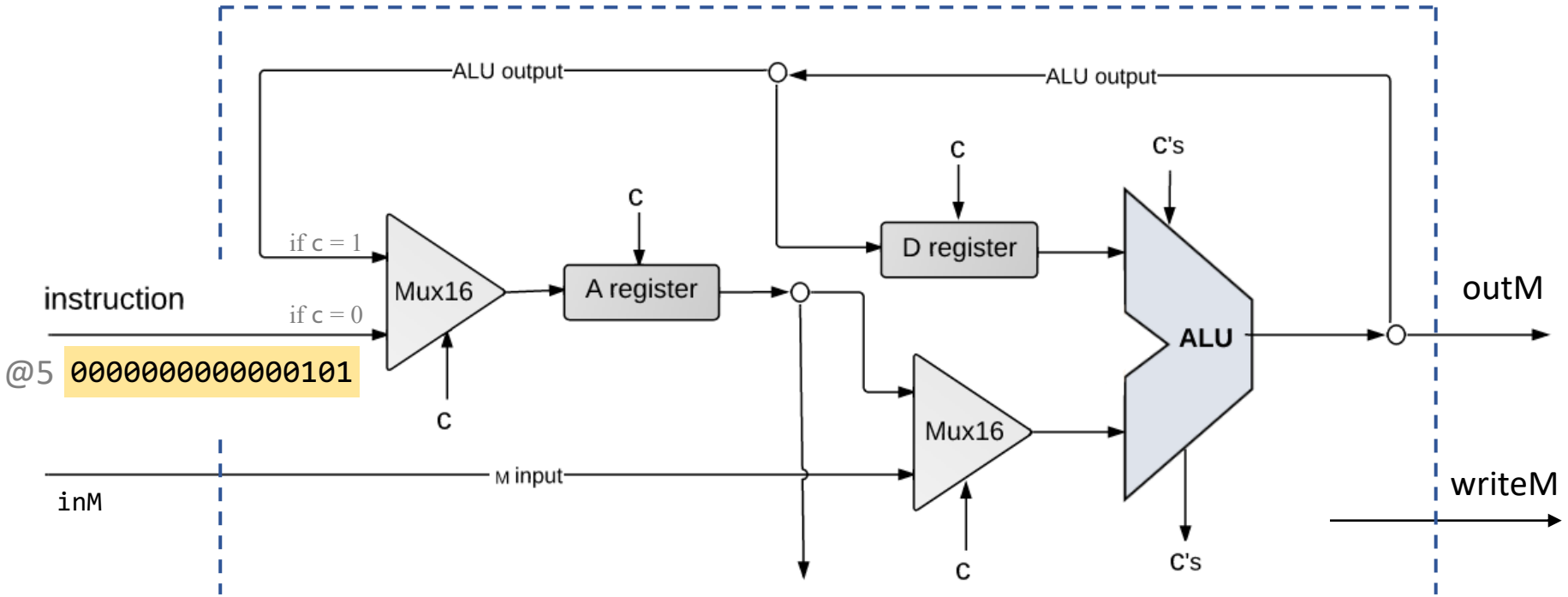


In this diagram, "c" denotes "control bit";
The control bits come from the instruction;
They tell the CPU chip-parts what to do.

CPU implementation: Instruction handling



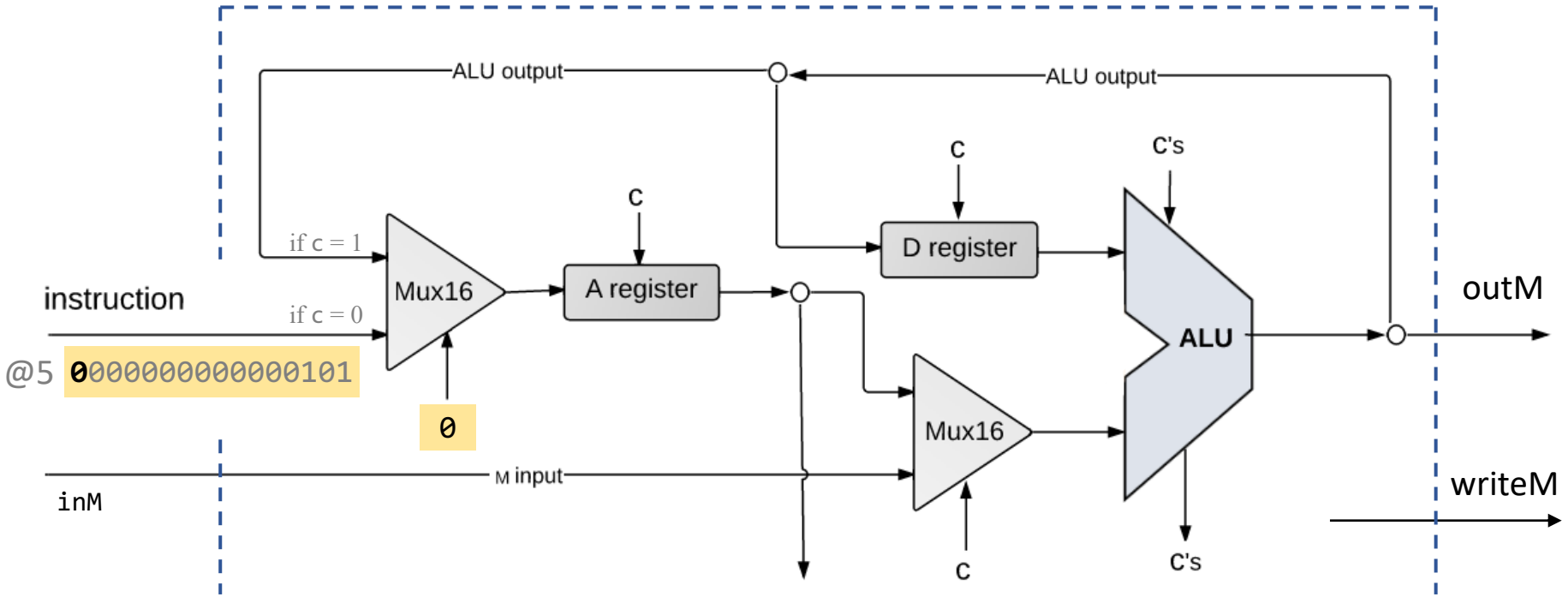
CPU implementation: Instruction handling



Handling A-instructions

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

CPU implementation: Instruction handling



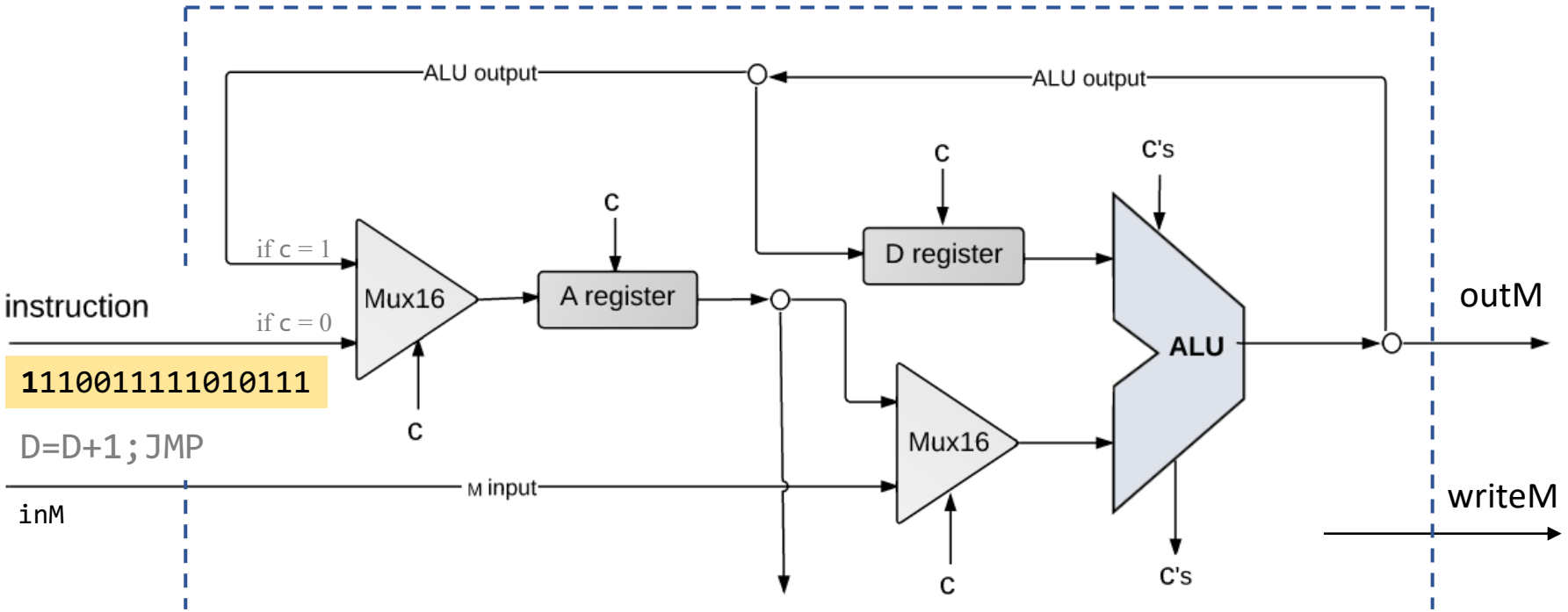
Handling A-instructions

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

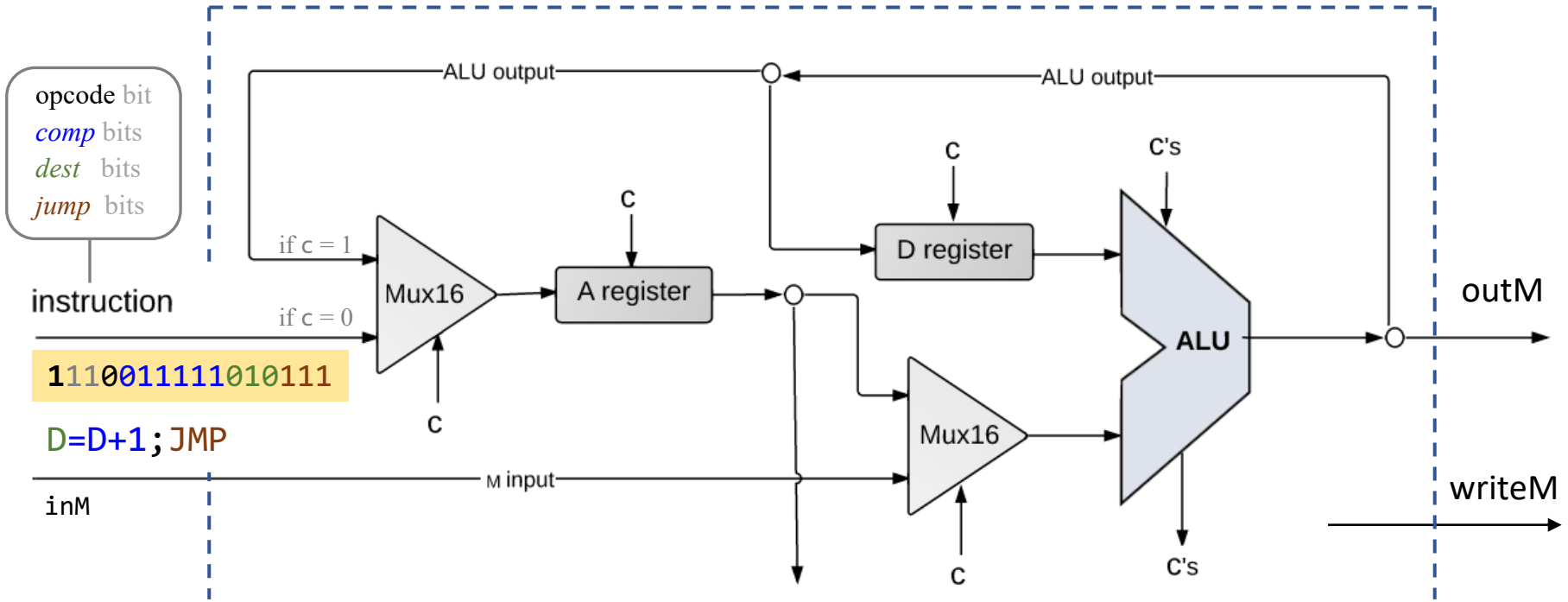
Effect: A-register ← instruction (treated as a value)

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

CPU implementation: Instruction handling



CPU implementation: Instruction handling



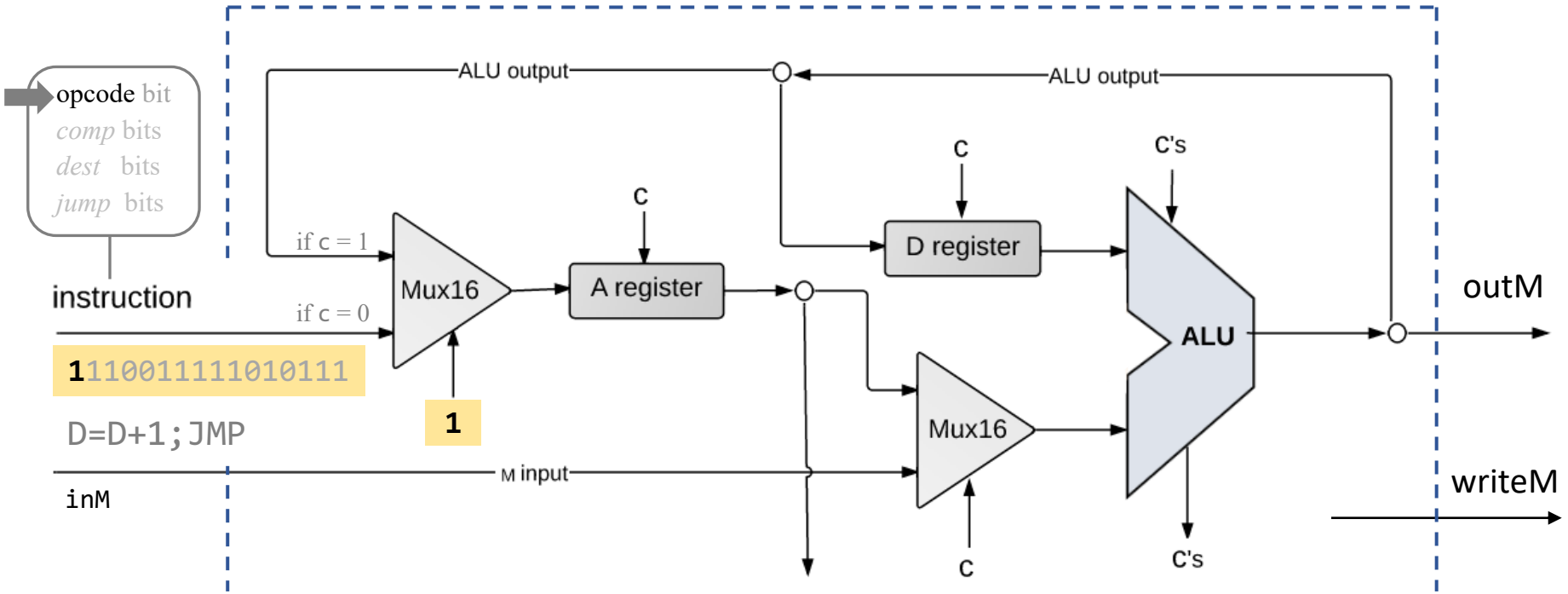
Handling C-instructions

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

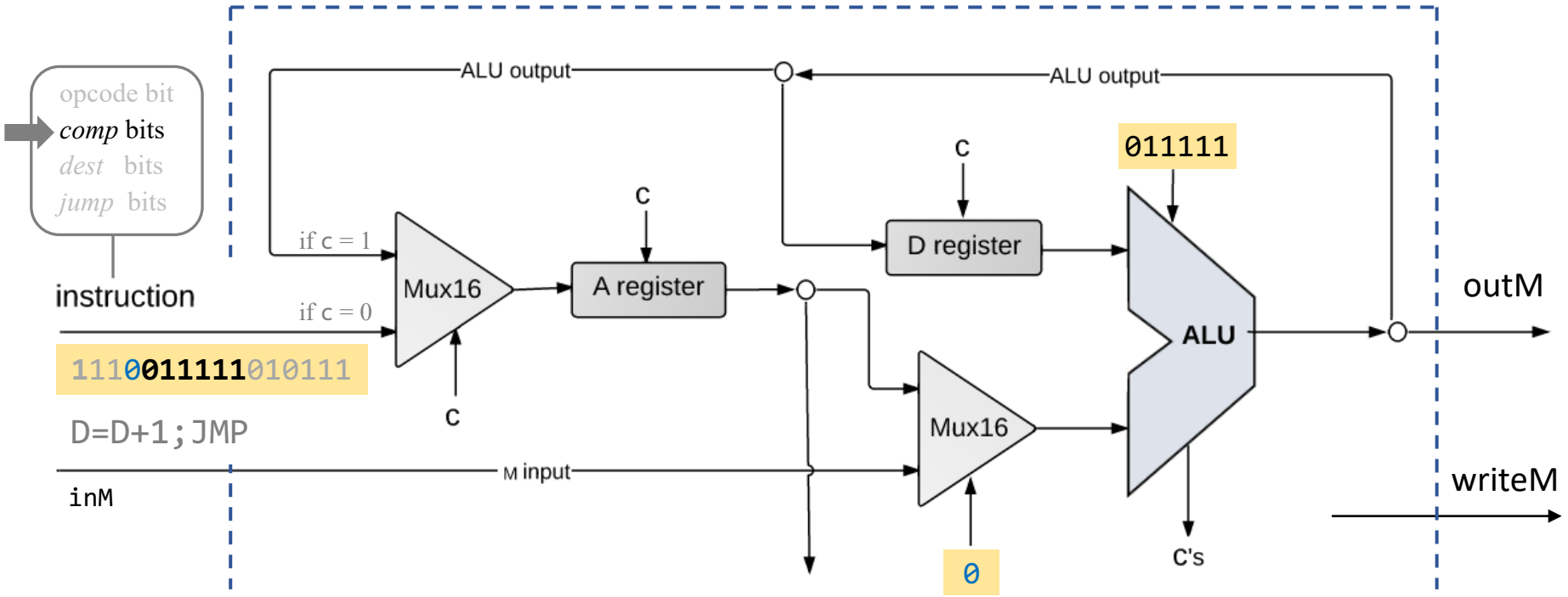


Handling C-instructions: The *opcode* bit

Routes the instruction's MSB to the Mux16

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

CPU implementation: Instruction handling



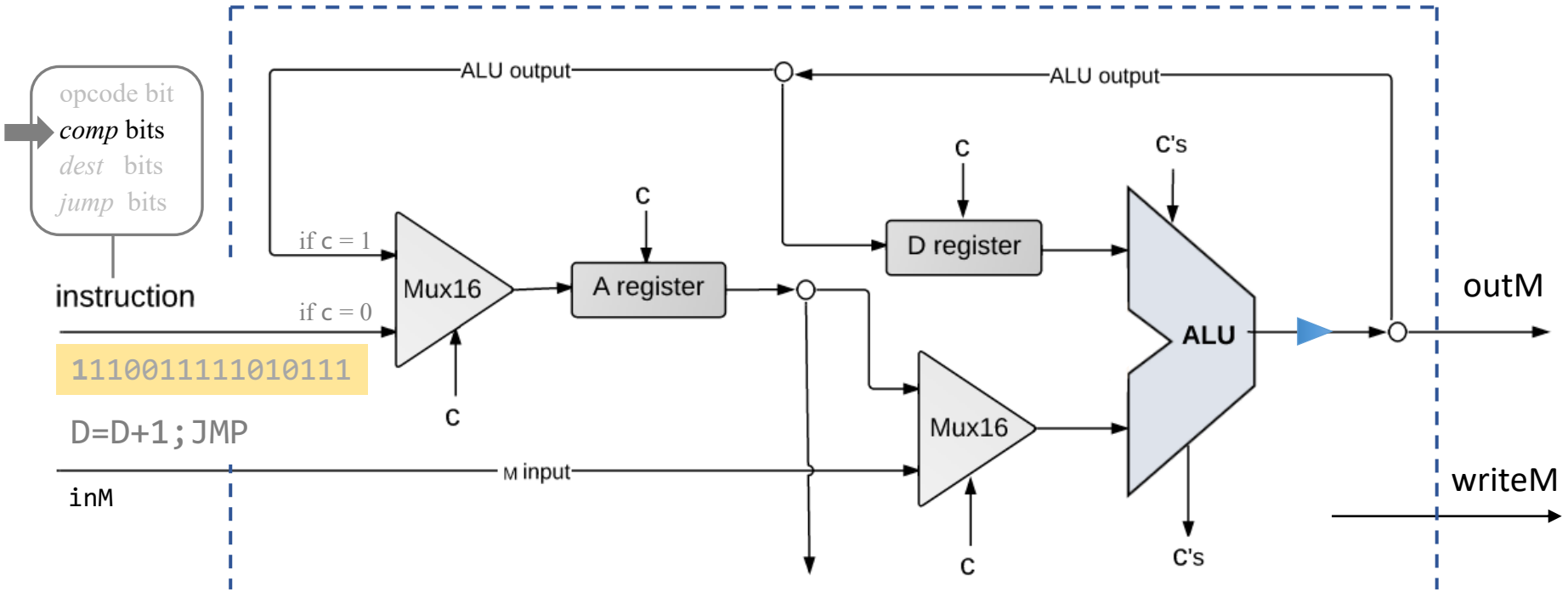
Handling C-instructions: The *comp*utation bits

- 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

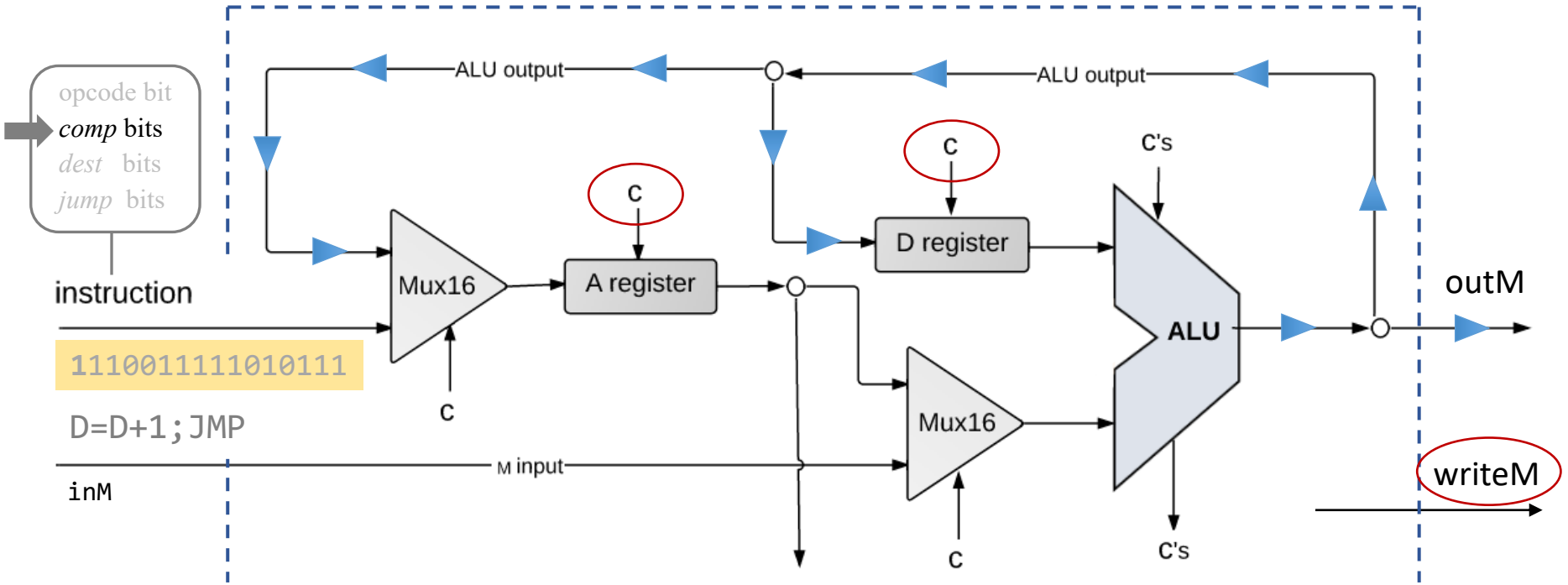
CPU implementation: Instruction handling



ALU output:

- Result of ALU calculation

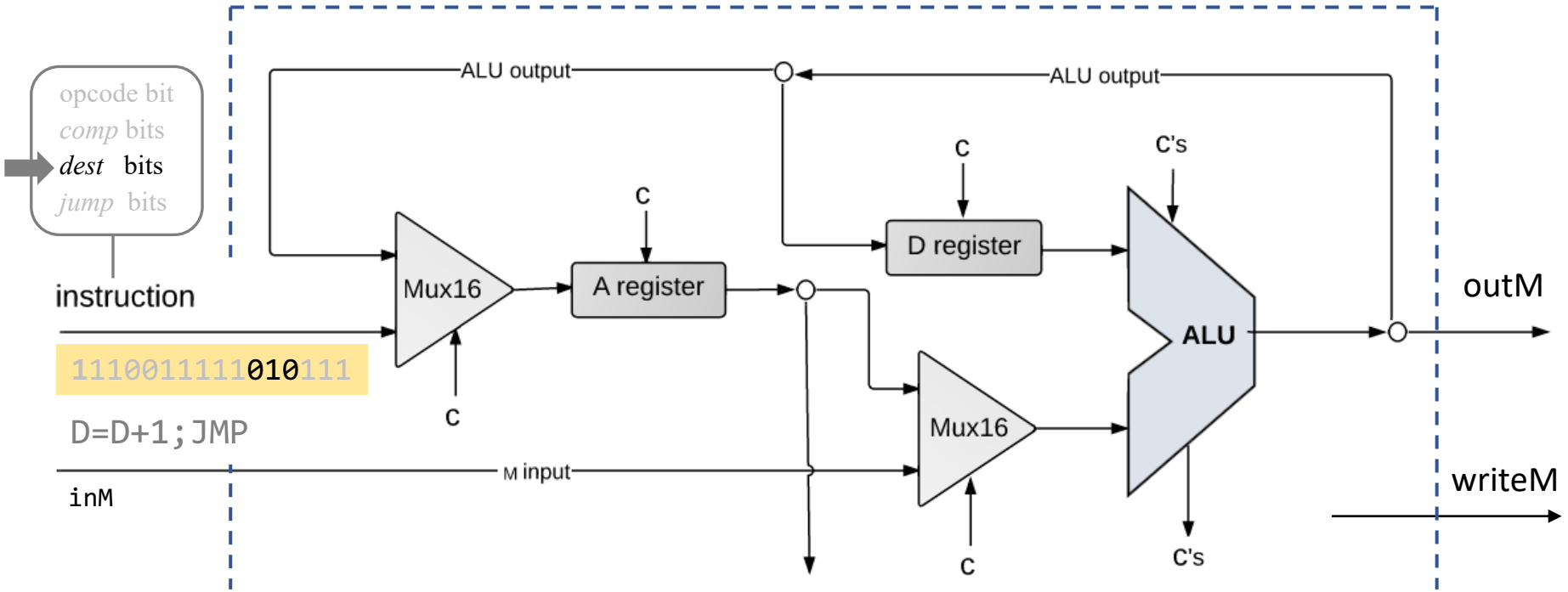
CPU implementation: Instruction handling



ALU output:

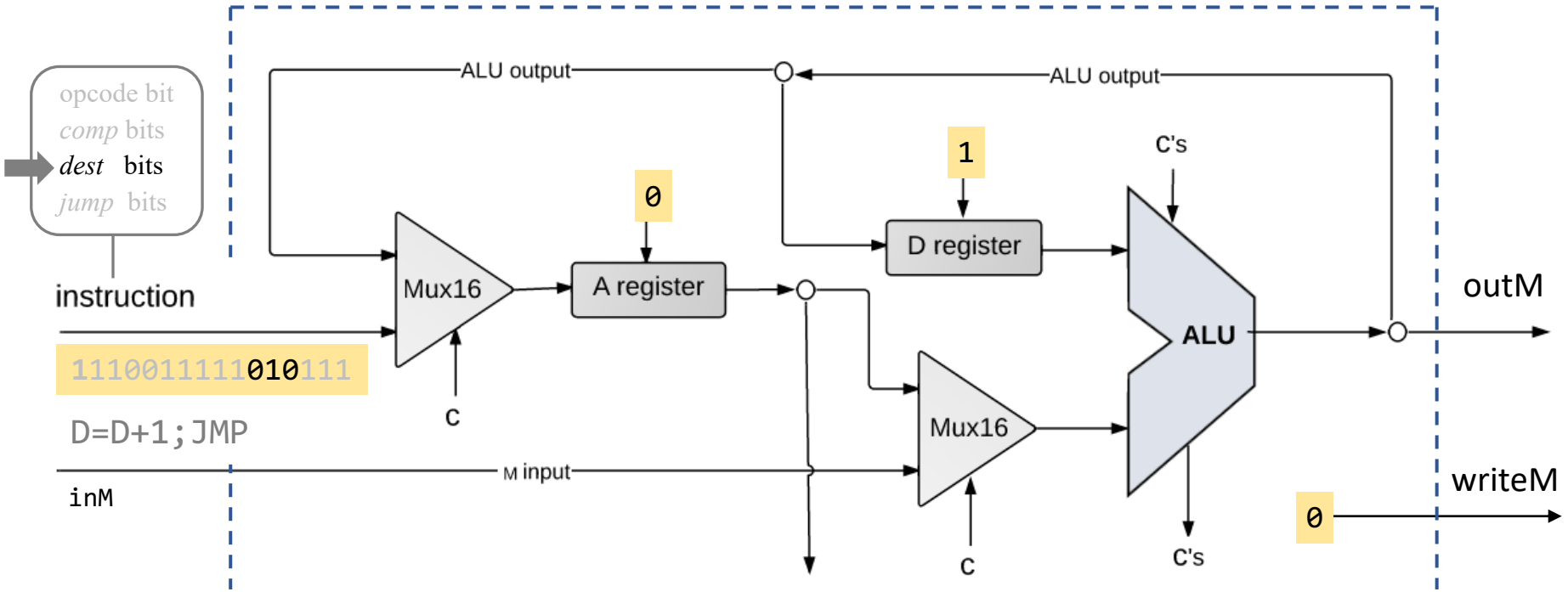
- Result of ALU calculation
- Fed simultaneously to D-register, A-register, data memory
- Each enabled/disabled by its control bit

CPU implementation: Instruction handling



Handling C-instructions: The *dest*ination bits

CPU implementation: Instruction handling

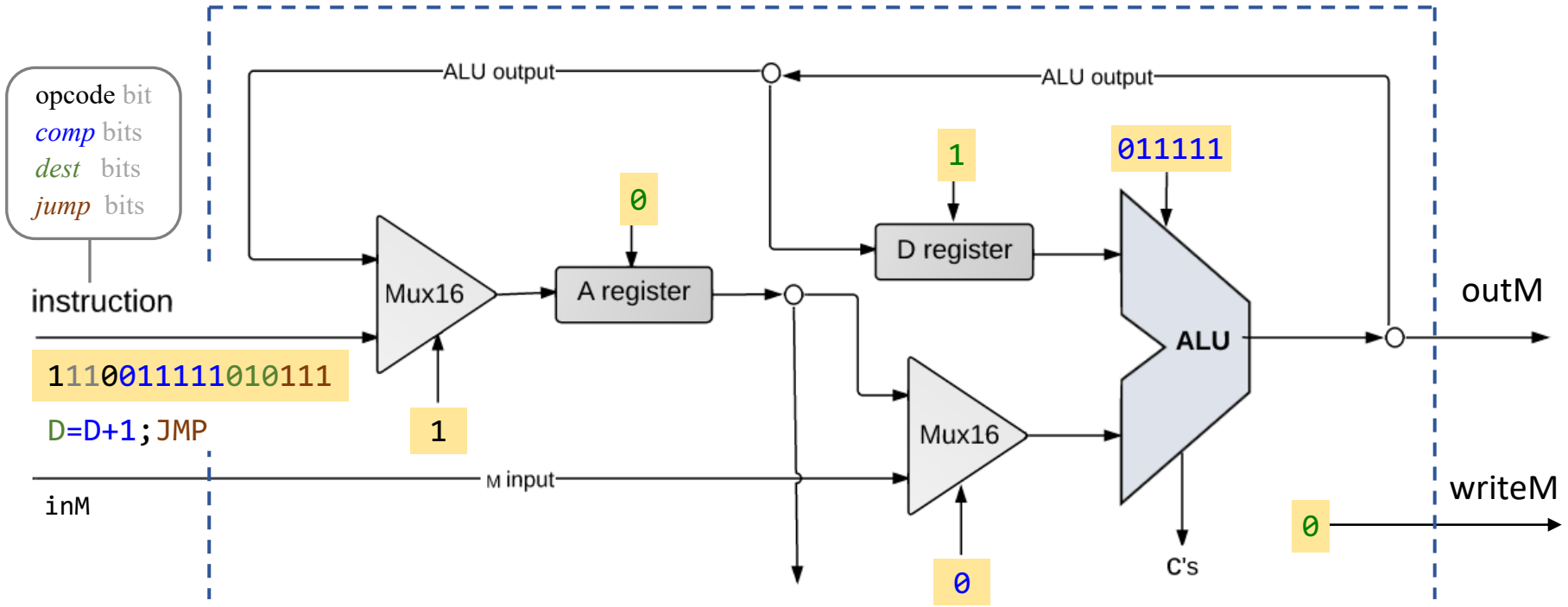


Handling C-instructions: The *dest*ination bits

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

Effect: Only the enabled destinations get the ALU output

CPU implementation: Instruction handling

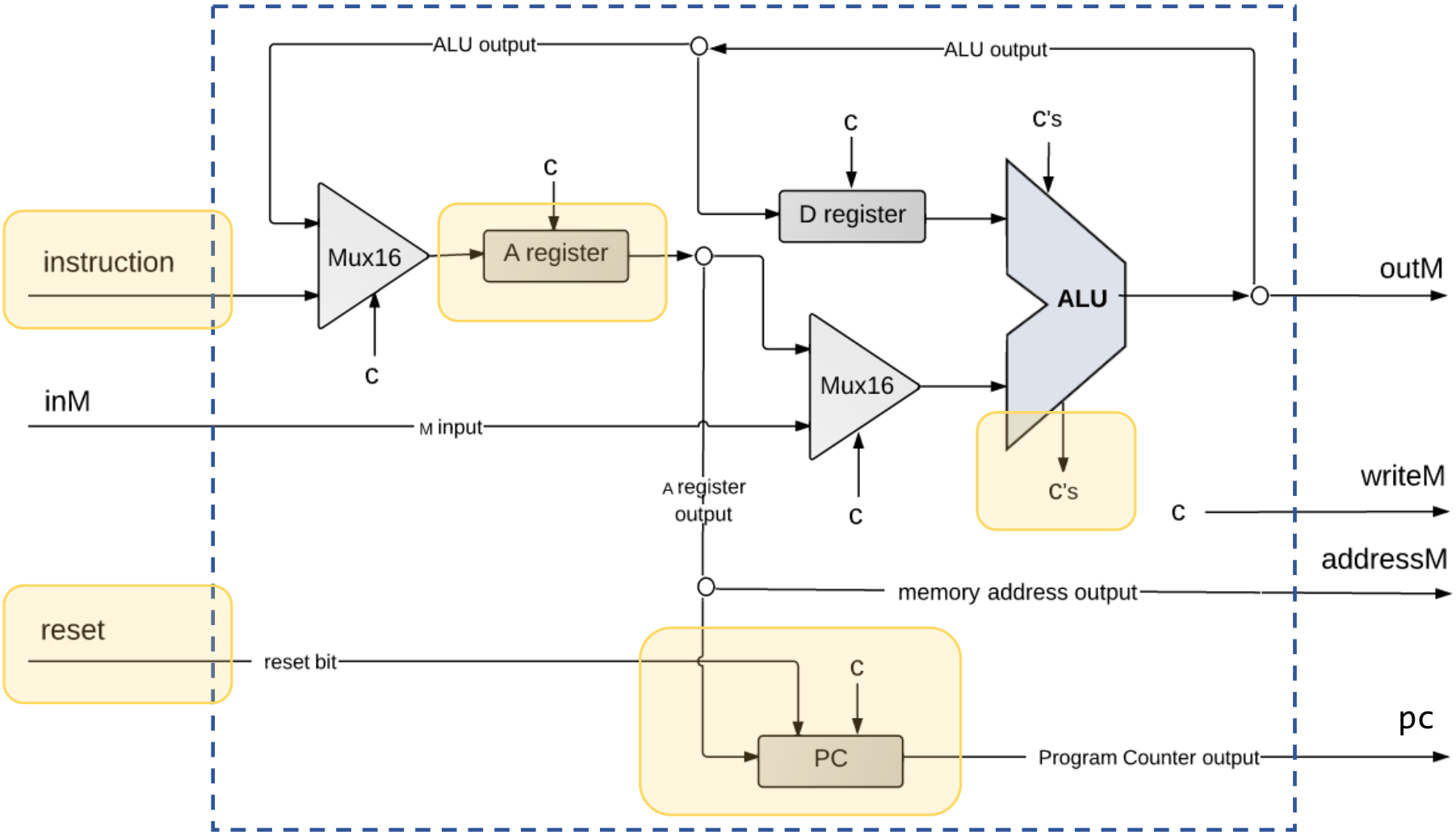


Handling C-instructions: Recap

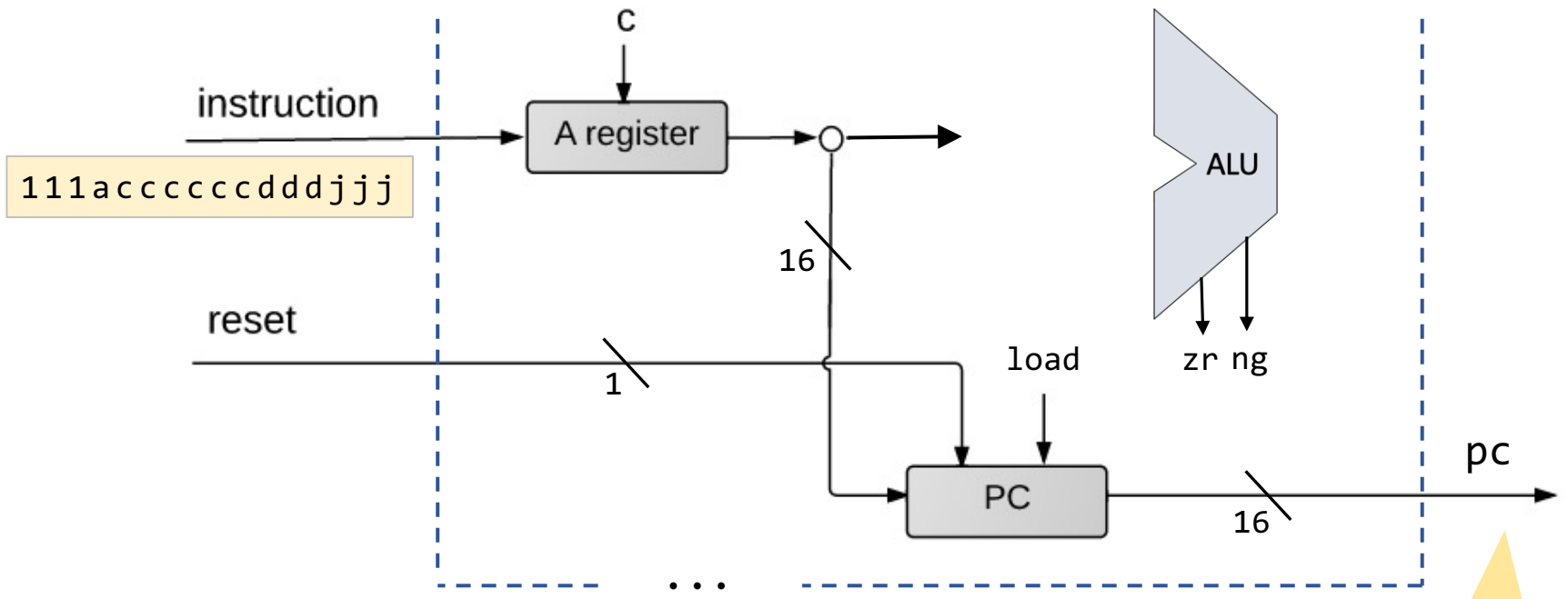
✓ Executes *dest* = *comp*

➡ Figures out which instruction to execute next

CPU implementation: Control



CPU implementation: Control



PC abstraction

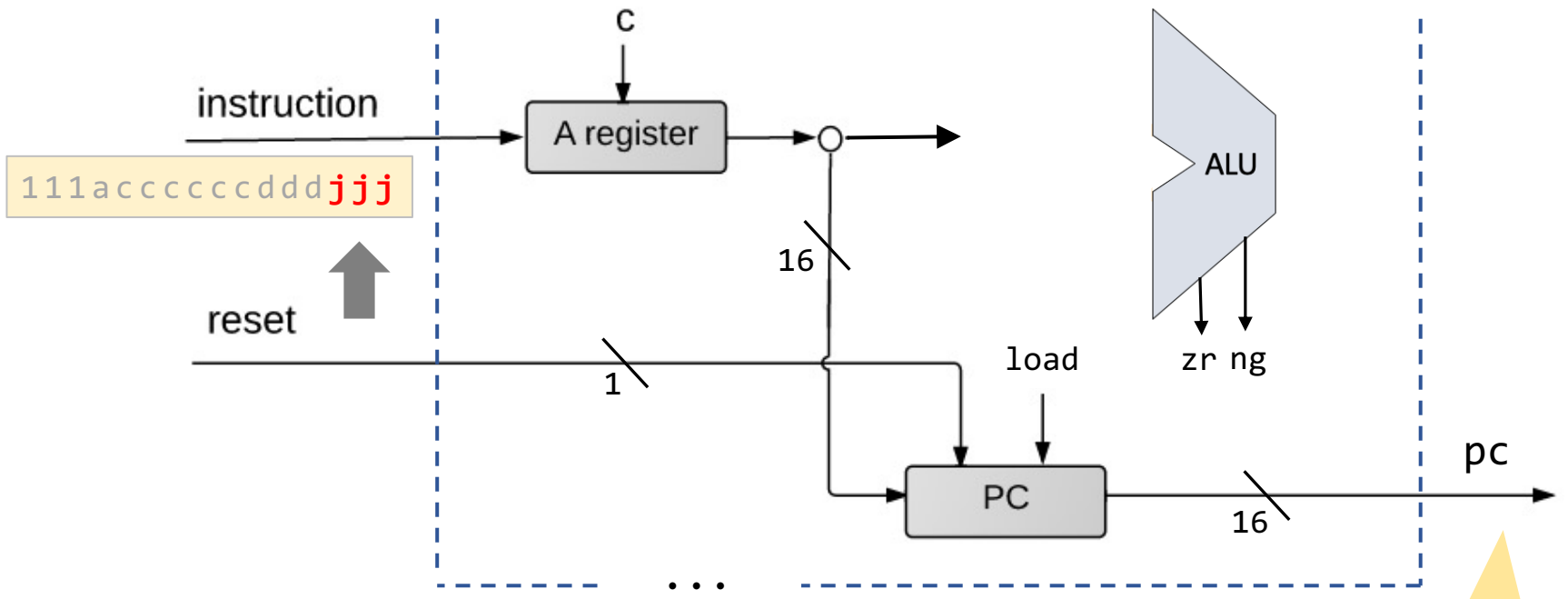
Outputs the address of the next instruction, has three states:

reset: `PC ← 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 ← 0`

no jump: `PC++`

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

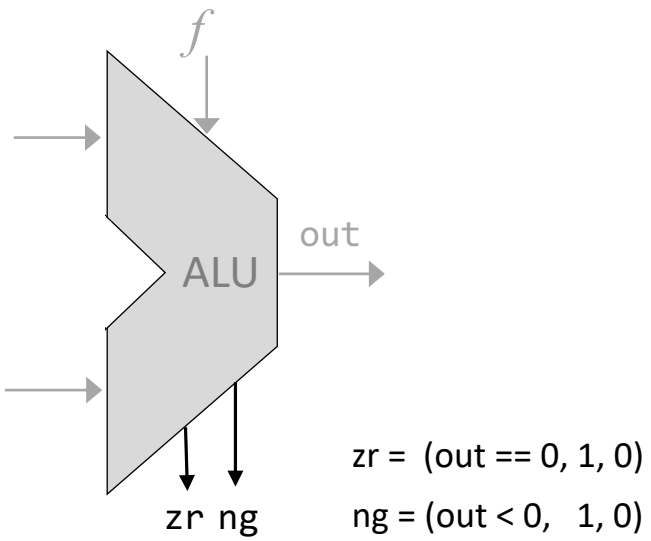
CPU implementation: Control

Symbolic syntax: *dest* = *comp* ; *jump*

Binary syntax:

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

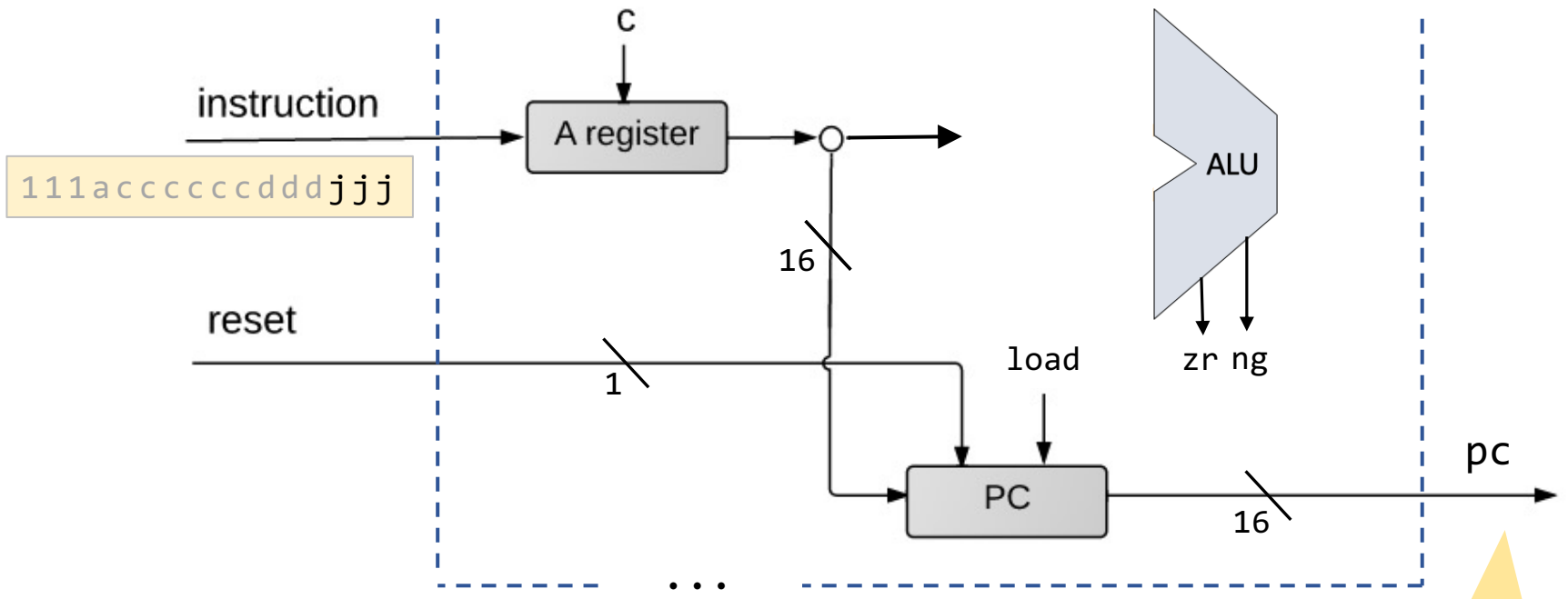
jump	j1	j2	j3	*condition*
null	0	0	0	no jump
JGT	0	0	1	if (ALU out > 0) jump
JEQ	0	1	0	if (ALU out = 0) jump
JGE	0	1	1	if (ALU out ≥ 0) jump
JLT	1	0	0	if (ALU out < 0) jump
JNE	1	0	1	if (ALU out ≠ 0) jump
JLE	1	1	0	if (ALU out ≤ 0) jump
JMP	1	1	1	Unconditional jump



We can use logic gates to compute the following function:

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

CPU implementation: Control



PC implementation

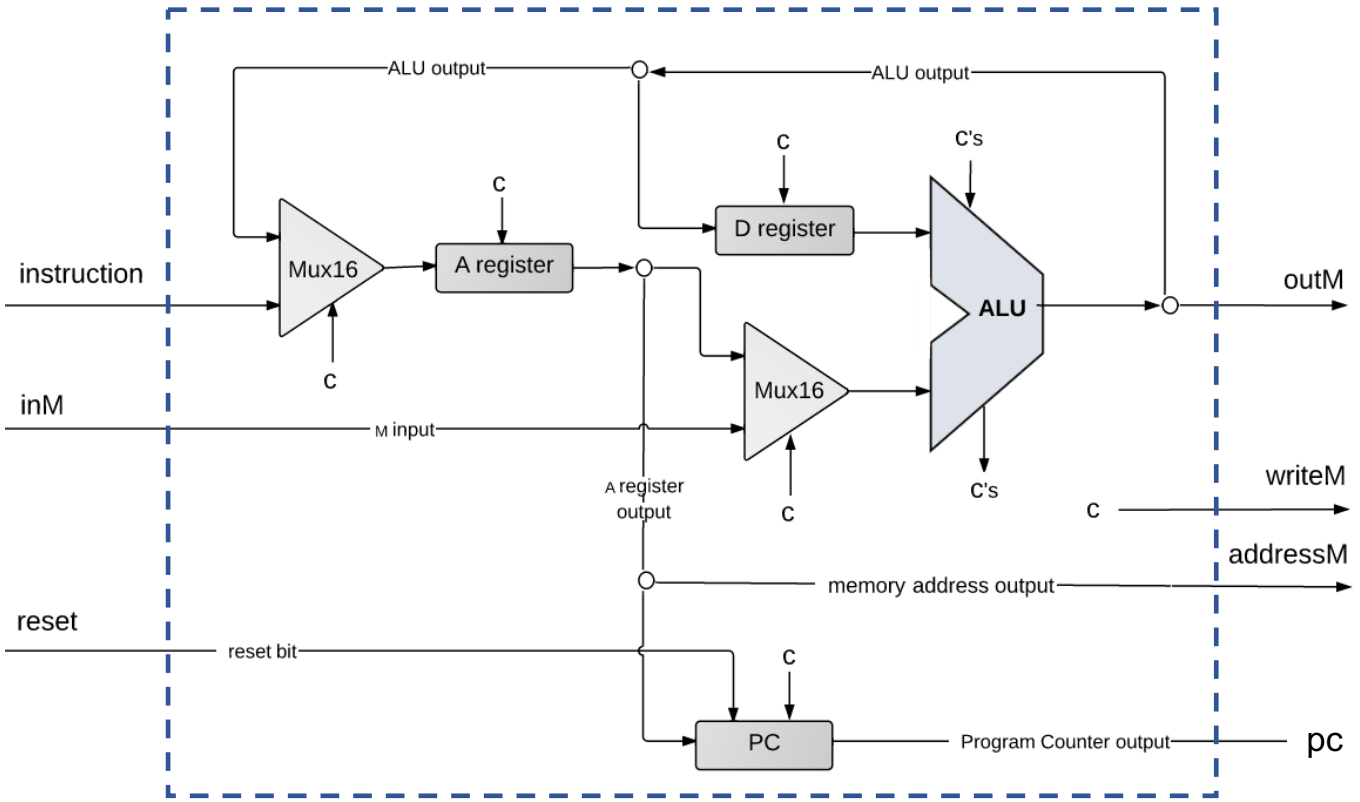
if (reset = 1) PC ← 0 // reset

else

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

else PC++ // next instruction

CPU implementation

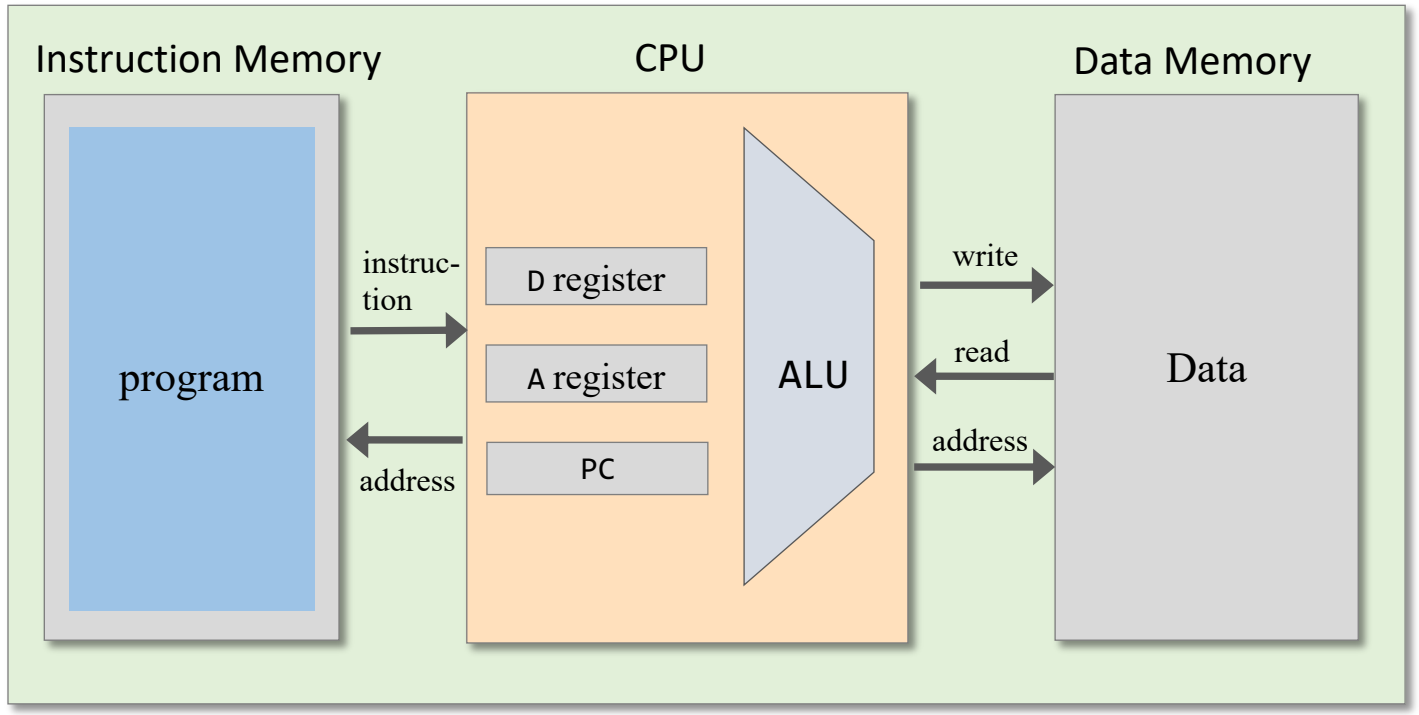


- ✓ Executes the current instruction
- ✓ Figures out which instruction to execute next.

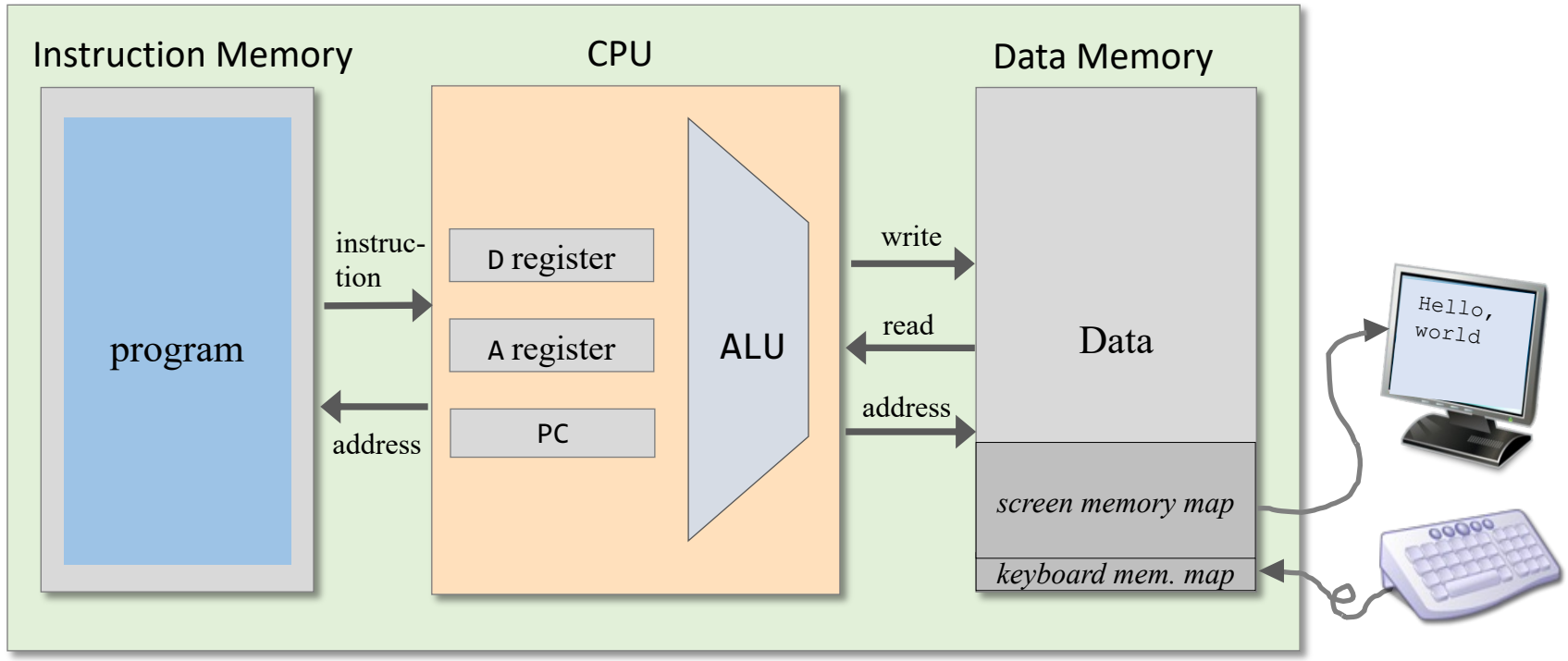
Computer Architecture

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

Hack computer



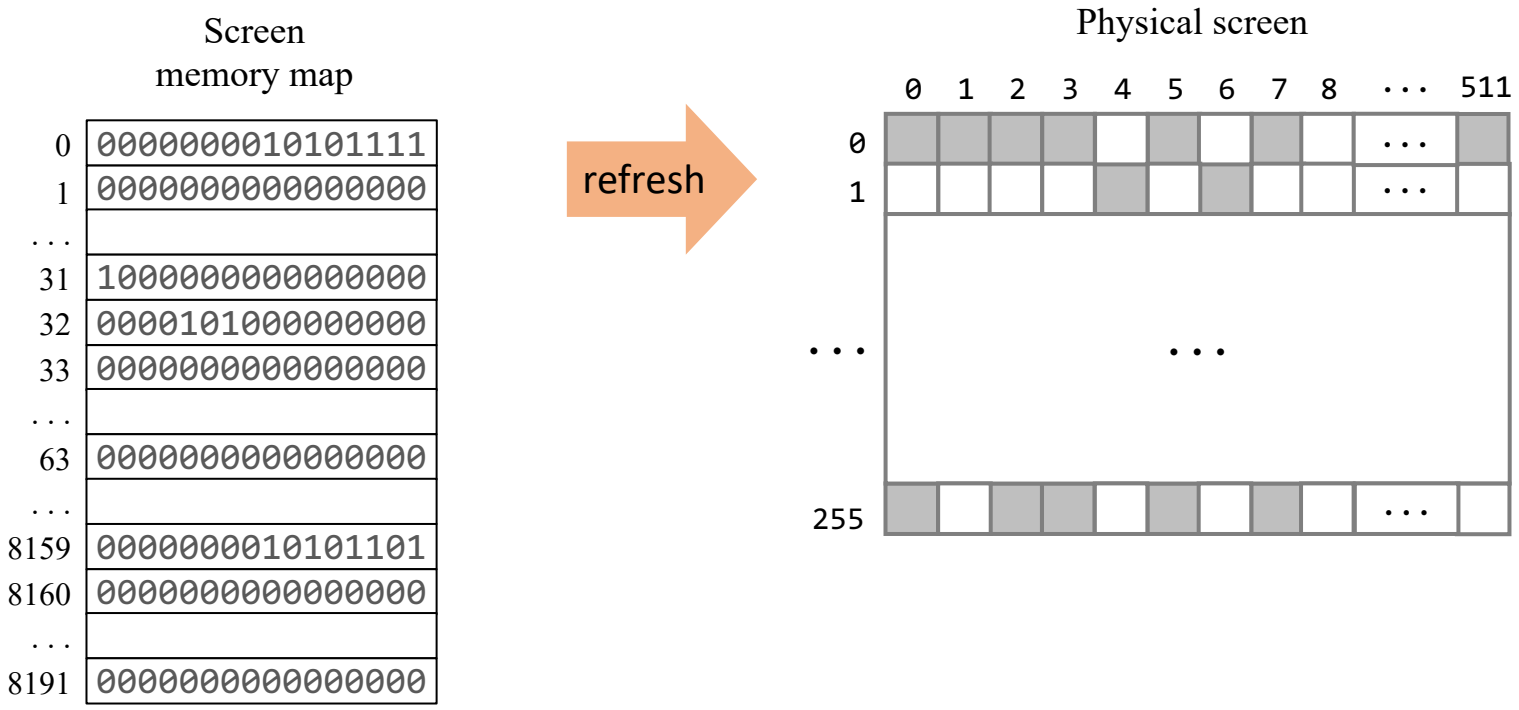
Hack computer



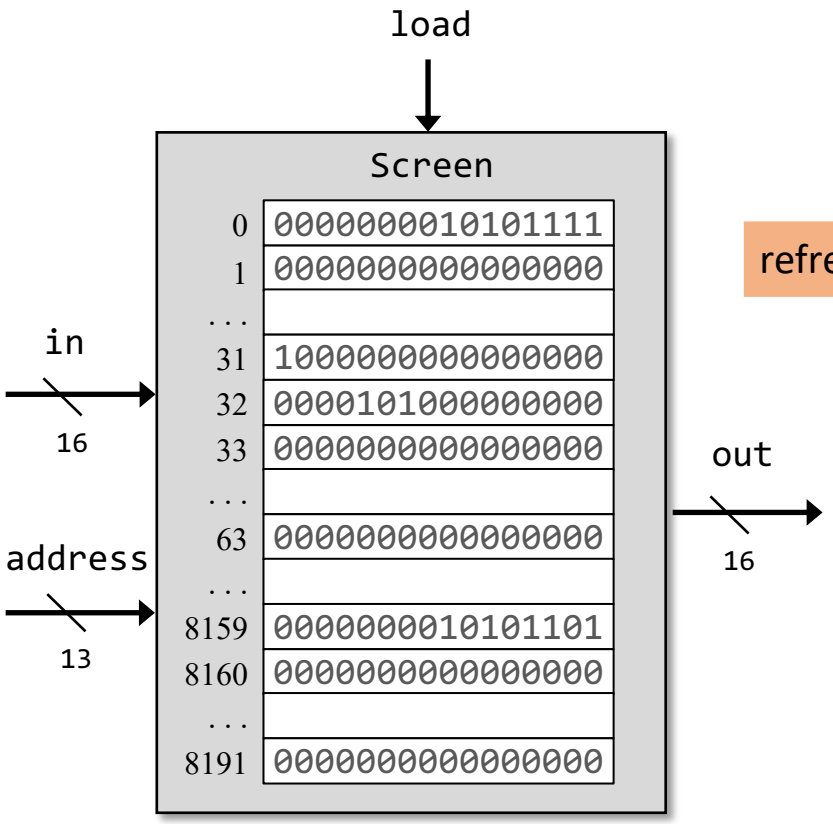
I/O devices

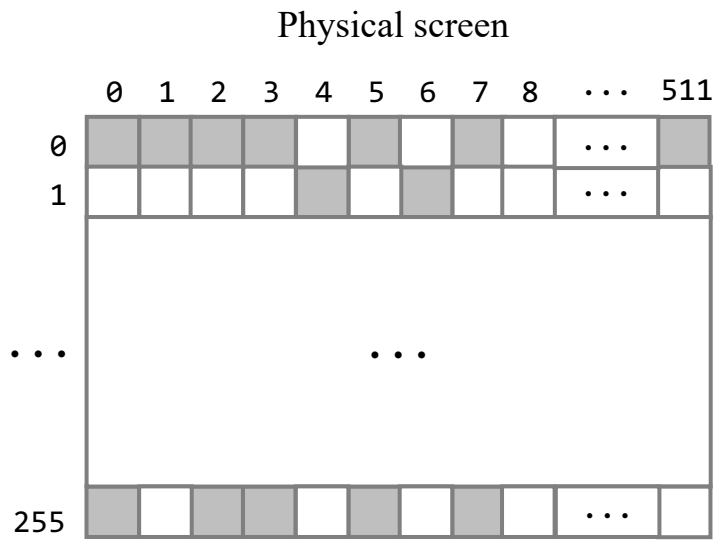
- Screen
- Keyboard

Screen


Screen
memory map
0 0000000010101111
1 0000000000000000
31 1000000000000000
32 0000101000000000
33 0000000000000000
63 0000000000000000
8159 0000000010101101
8160 0000000000000000
8191 0000000000000000
refresh
Physical screen
0 1 2 3 4 5 6 7 8 ... 511
0
1
255

Screen

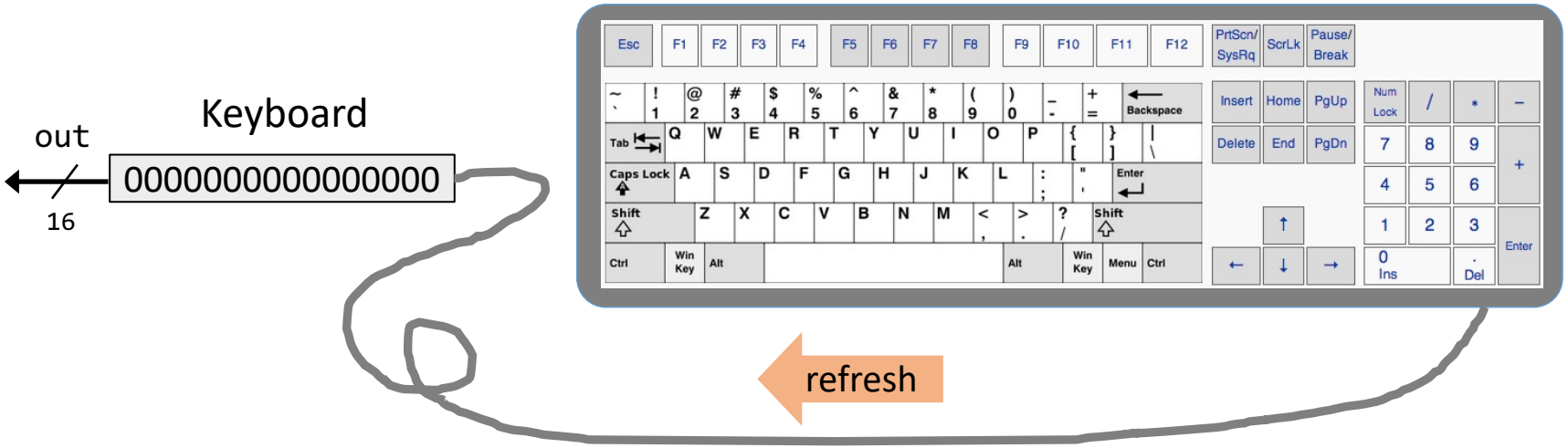




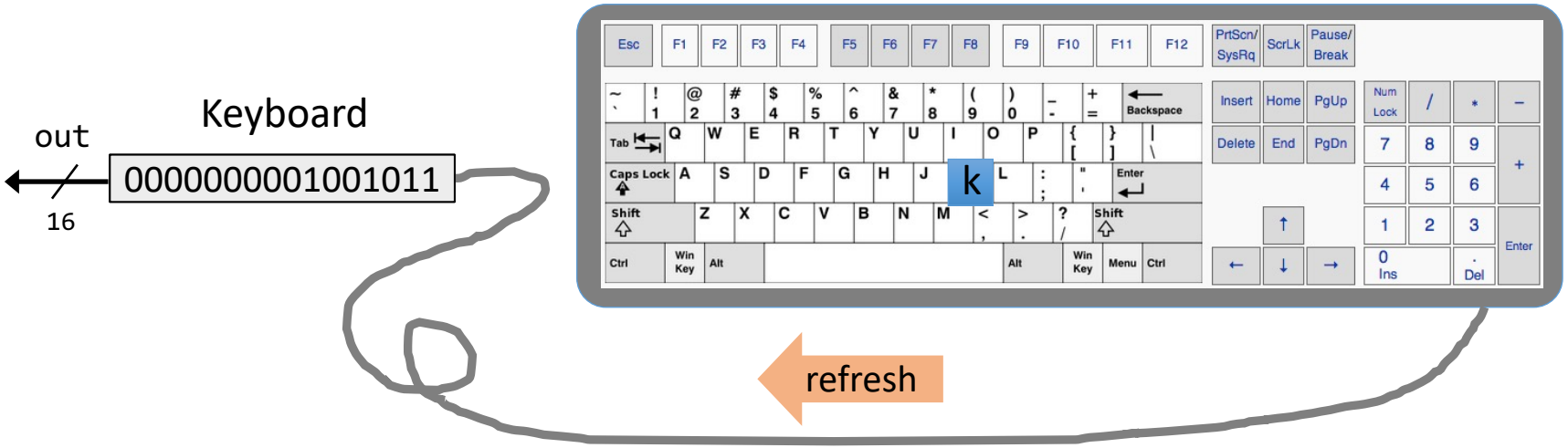
Implemented as a built-in 8K memory chip named `Screen`

```
/** 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

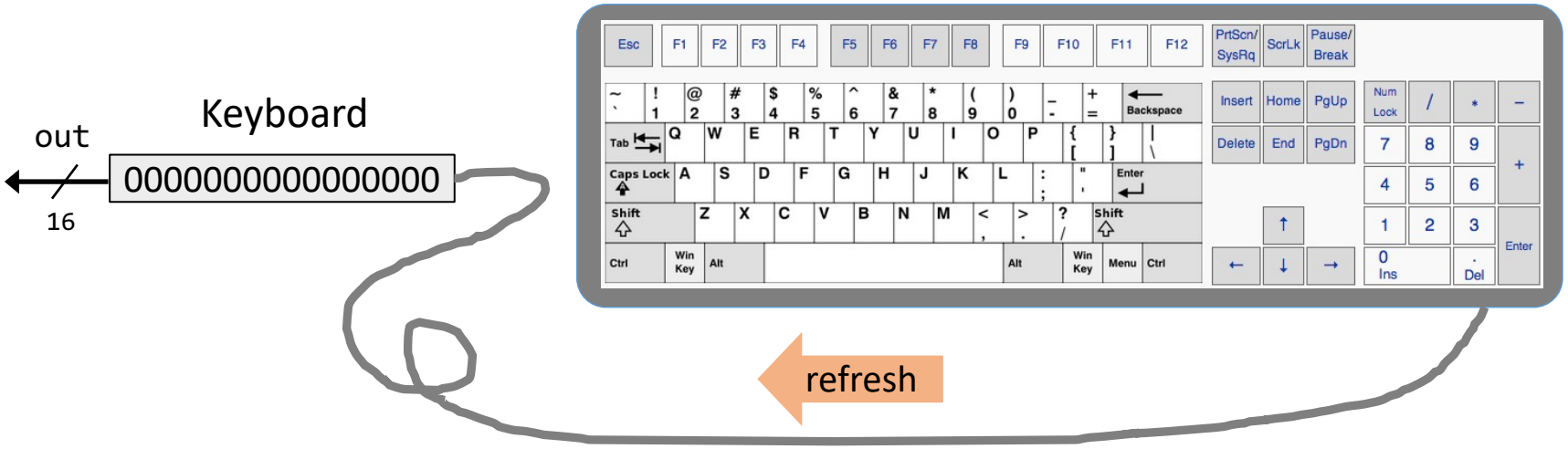


Keyboard



code('k') = 75

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

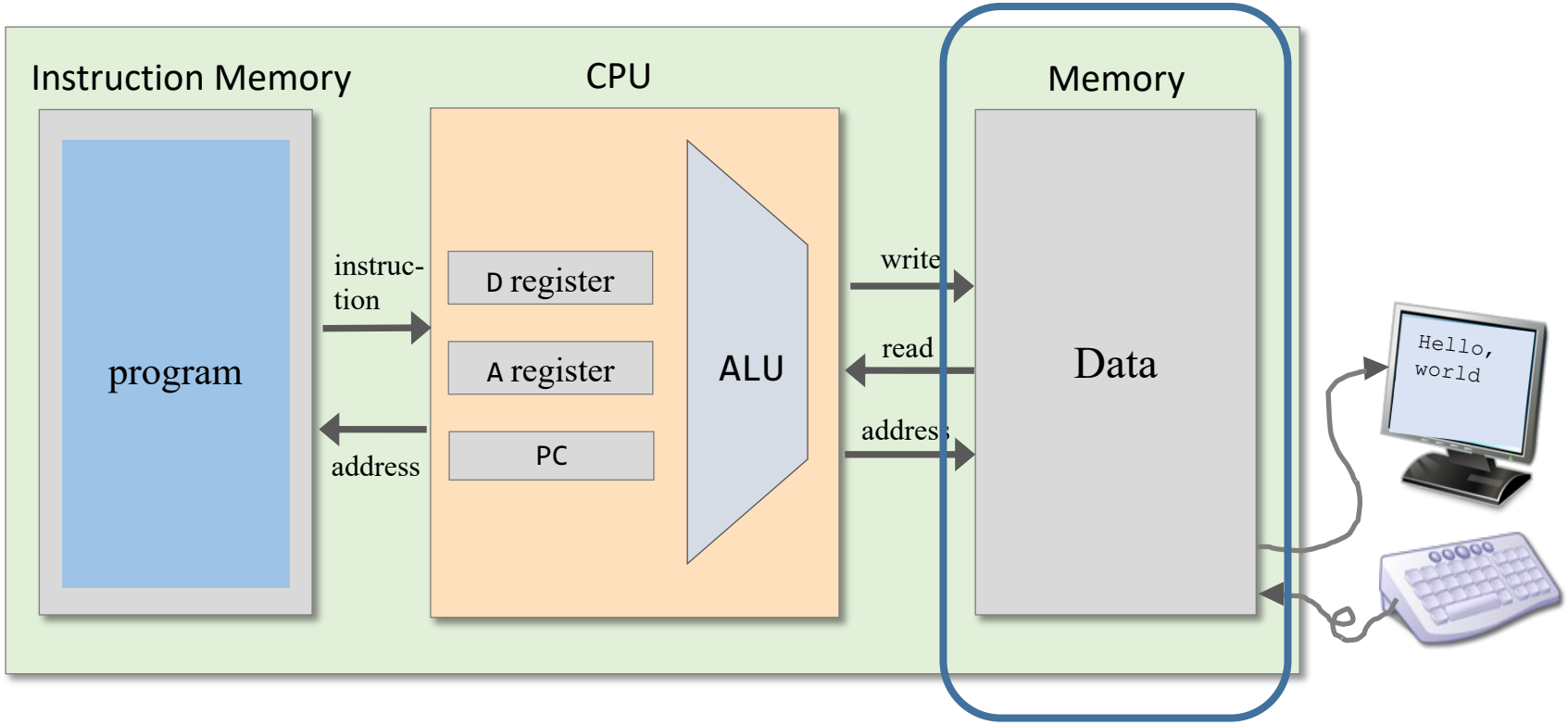
- ✓ Basic architecture
- ✓ Fetch-Execute cycle
- ✓ The Hack CPU
- ✓ Input / output
- Memory
- Computer
- Project 5: Chips
- Project 5: Guidelines

Chapter 5: Computer Architecture

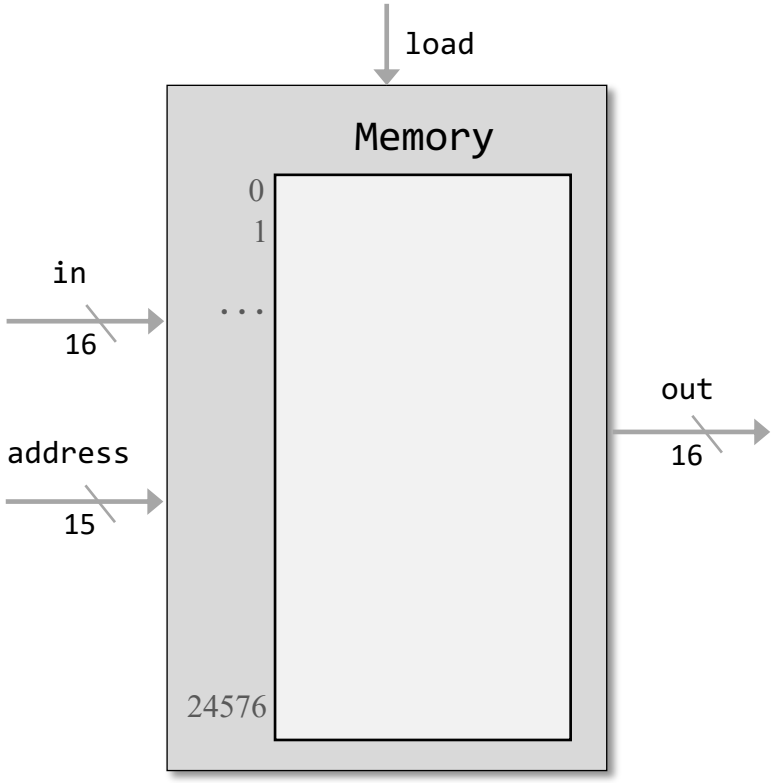
- Basic architecture
- Fetch-Execute cycle
- The Hack CPU
- Input / output

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

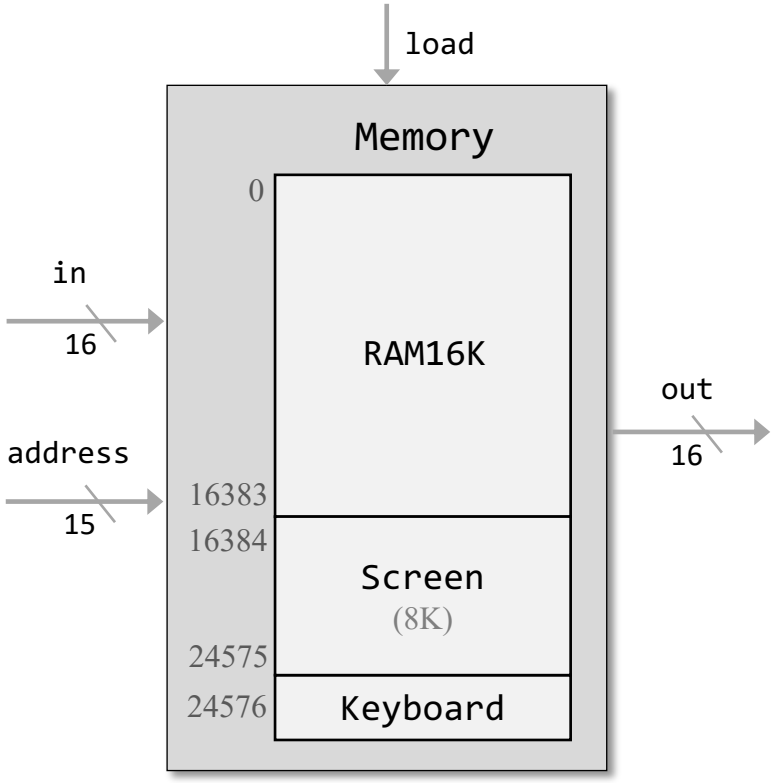
Memory


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

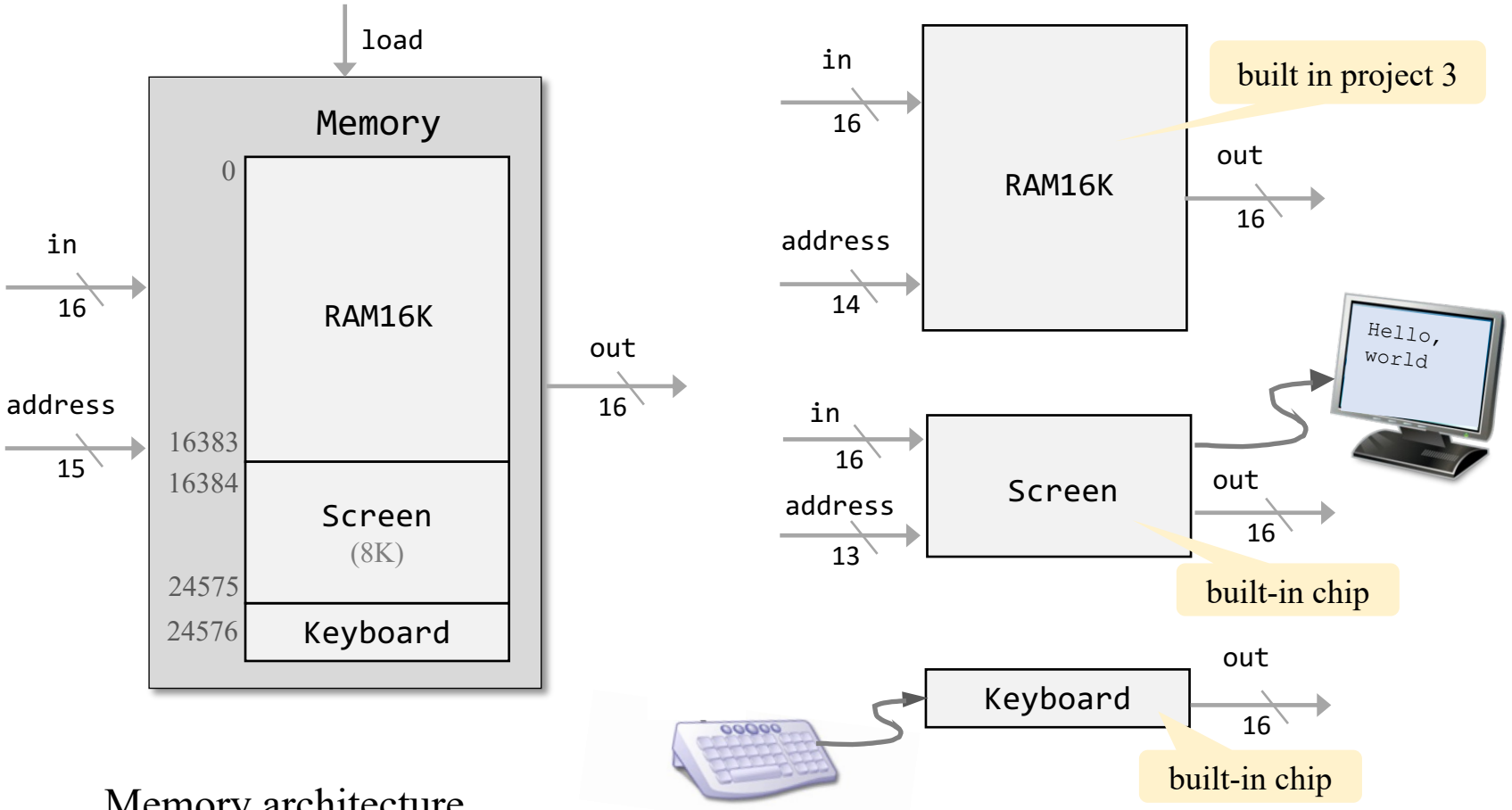
Memory: Implementation



Memory: Implementation



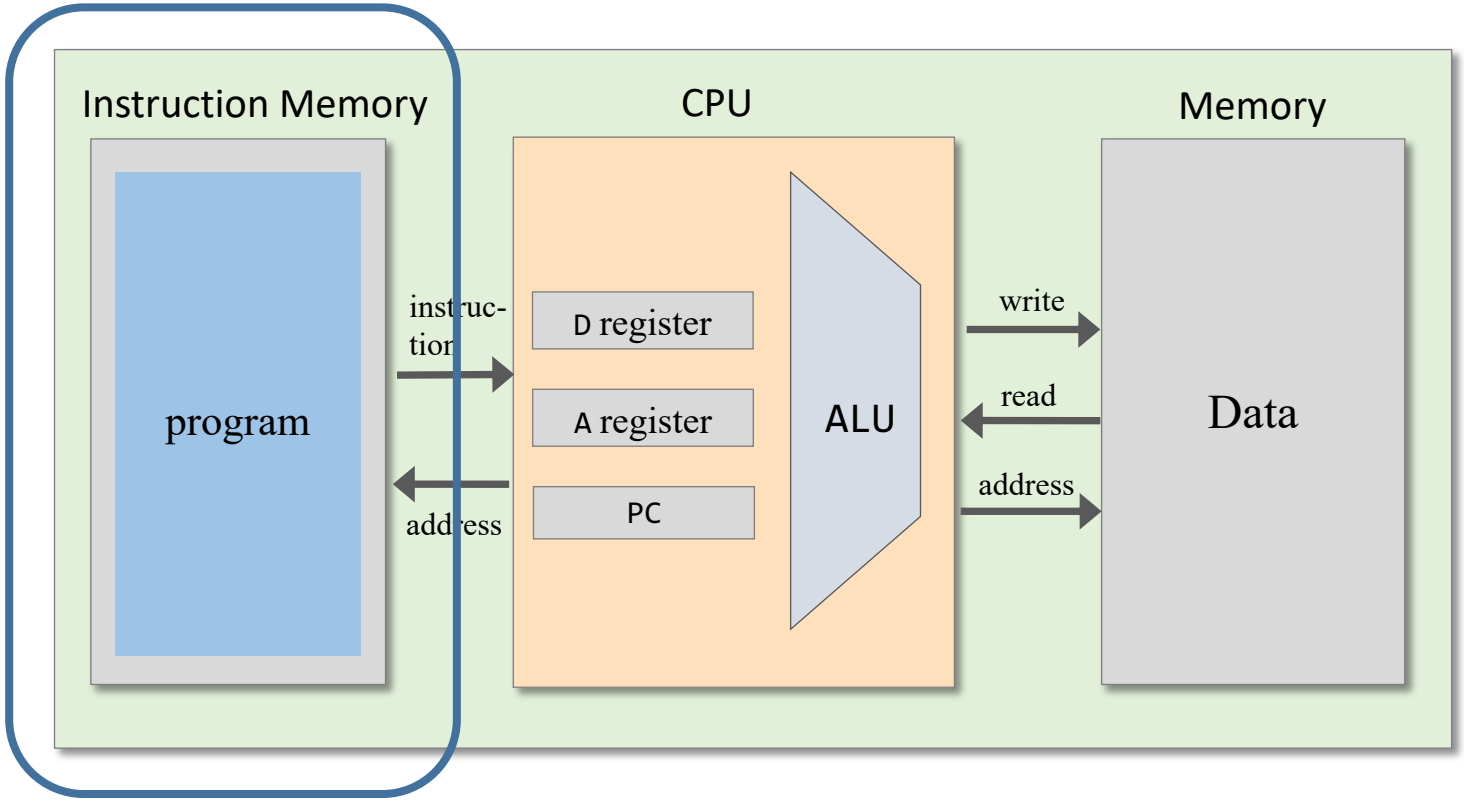
Memory: Implementation



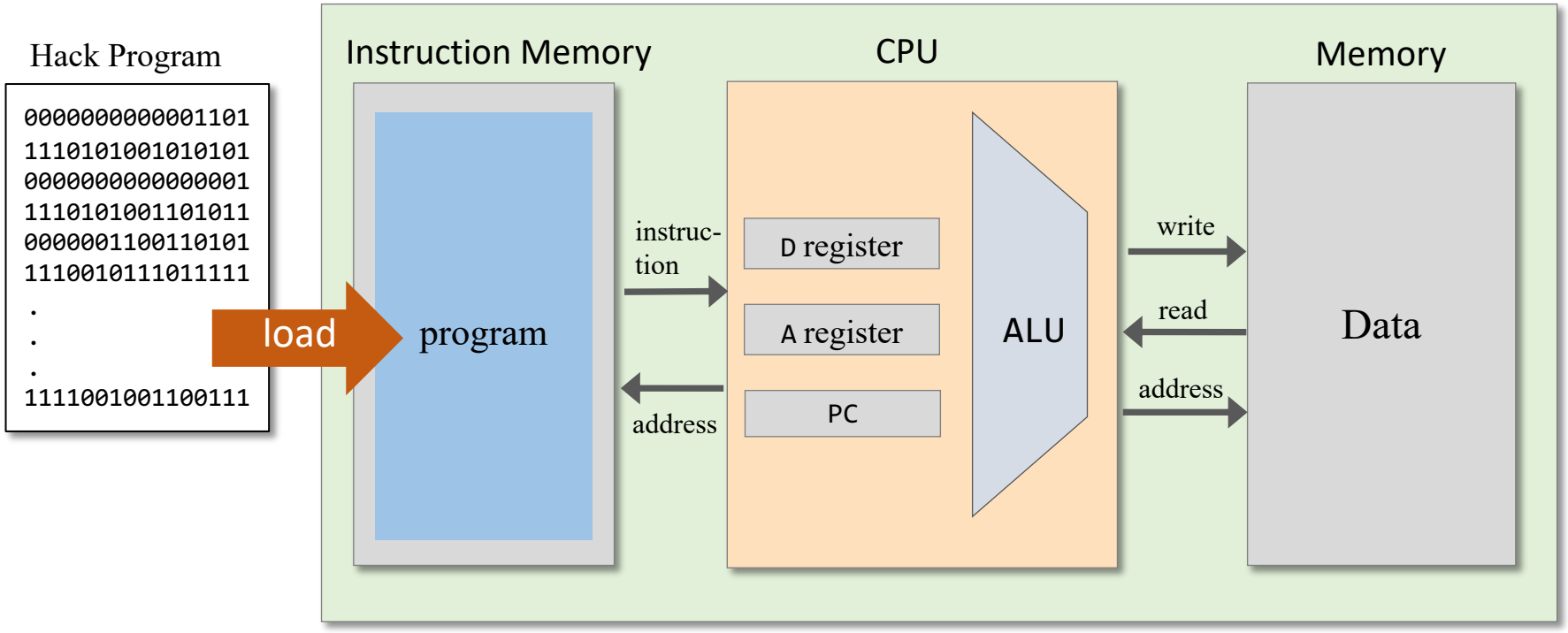
Memory architecture

- 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

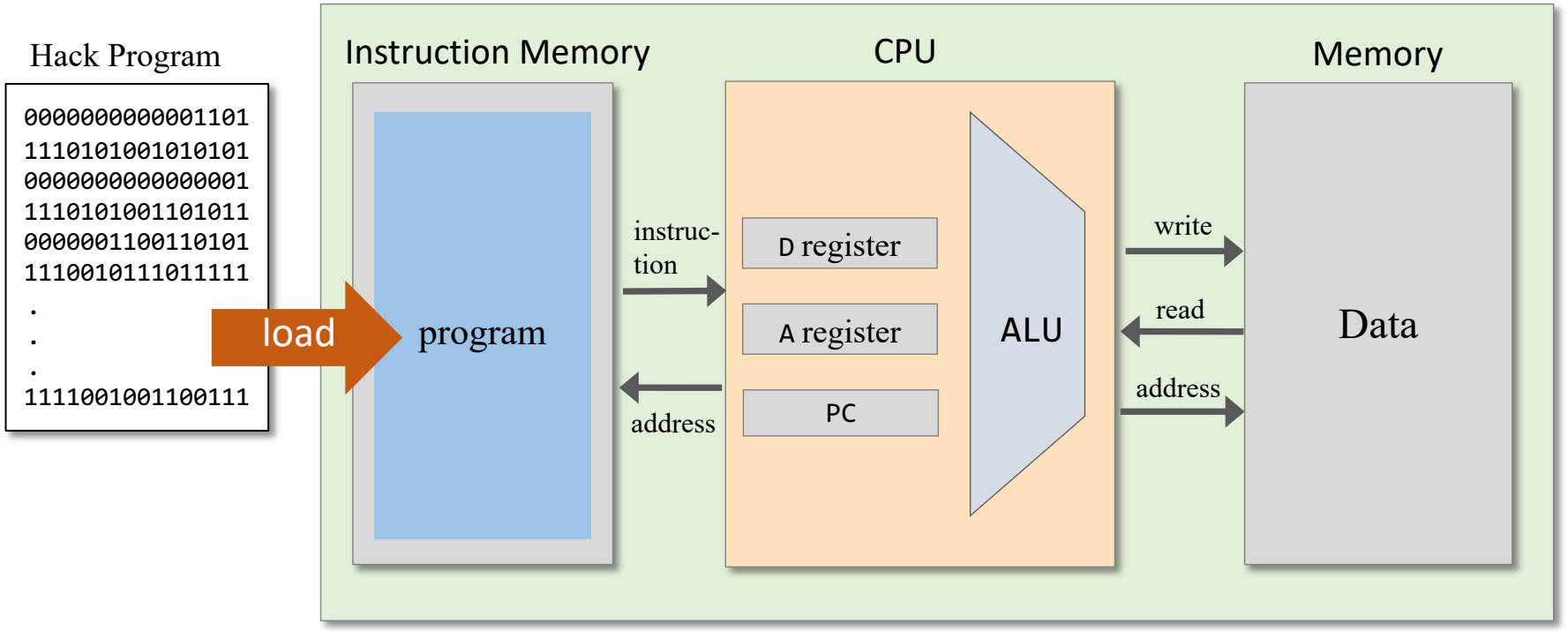
Implemented as a built-in plug-and-play chip named `ROM32K`

(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

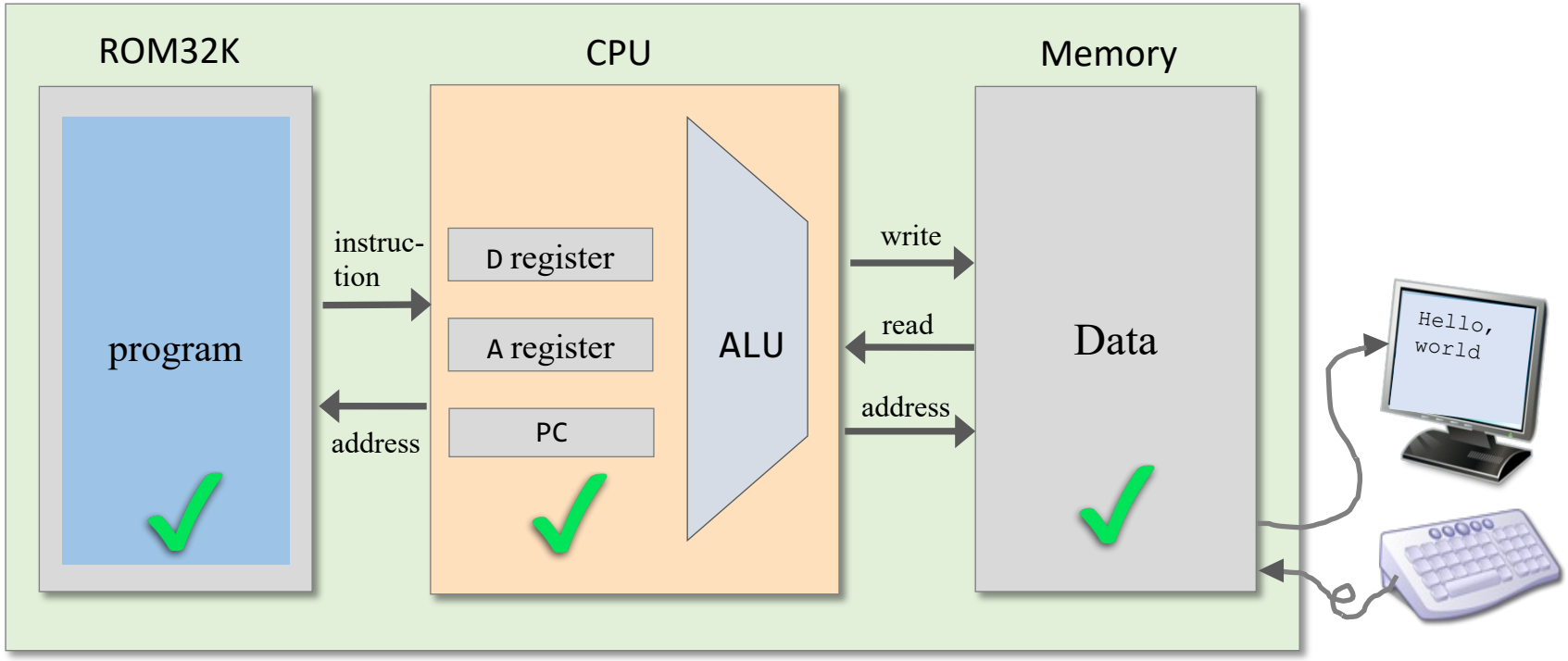
- Basic architecture
- Fetch-Execute cycle
- The Hack CPU
- Input / output

✓ Memory

Computer

- Project 5: Chips
- Project 5: Guidelines

Hack computer architecture



Remaining challenge

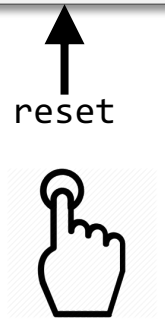
Integrate into a single chip, named `Computer`

Computer abstraction

Assumption:

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

Computer



To execute the stored program:

set `reset` ← `1`, then

set `reset` ← `0`

Computer abstraction

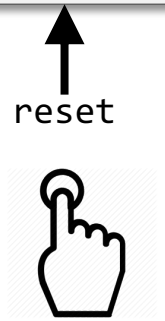
Assumption:

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

Computer

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

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

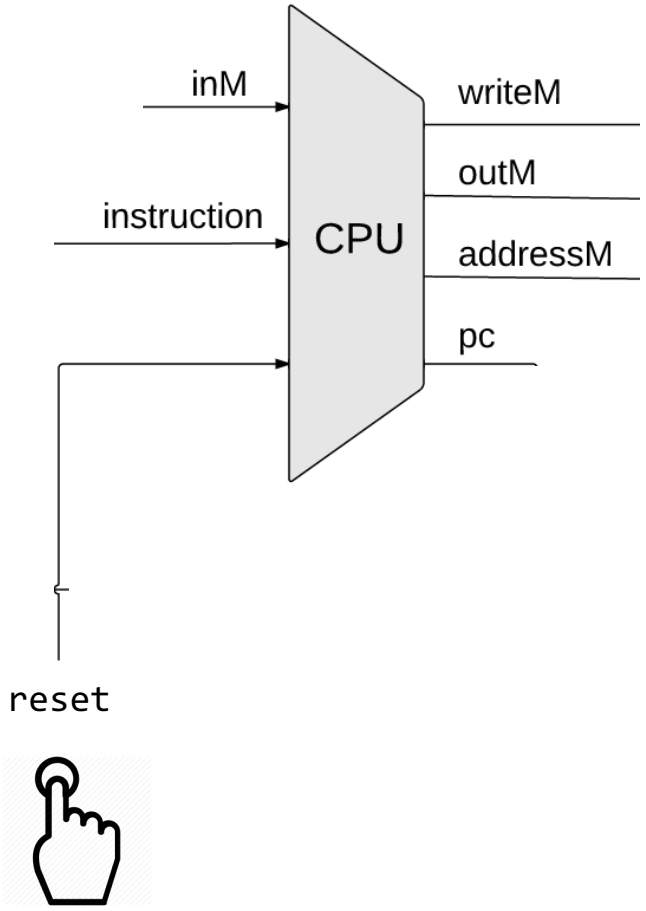


To execute the stored program:

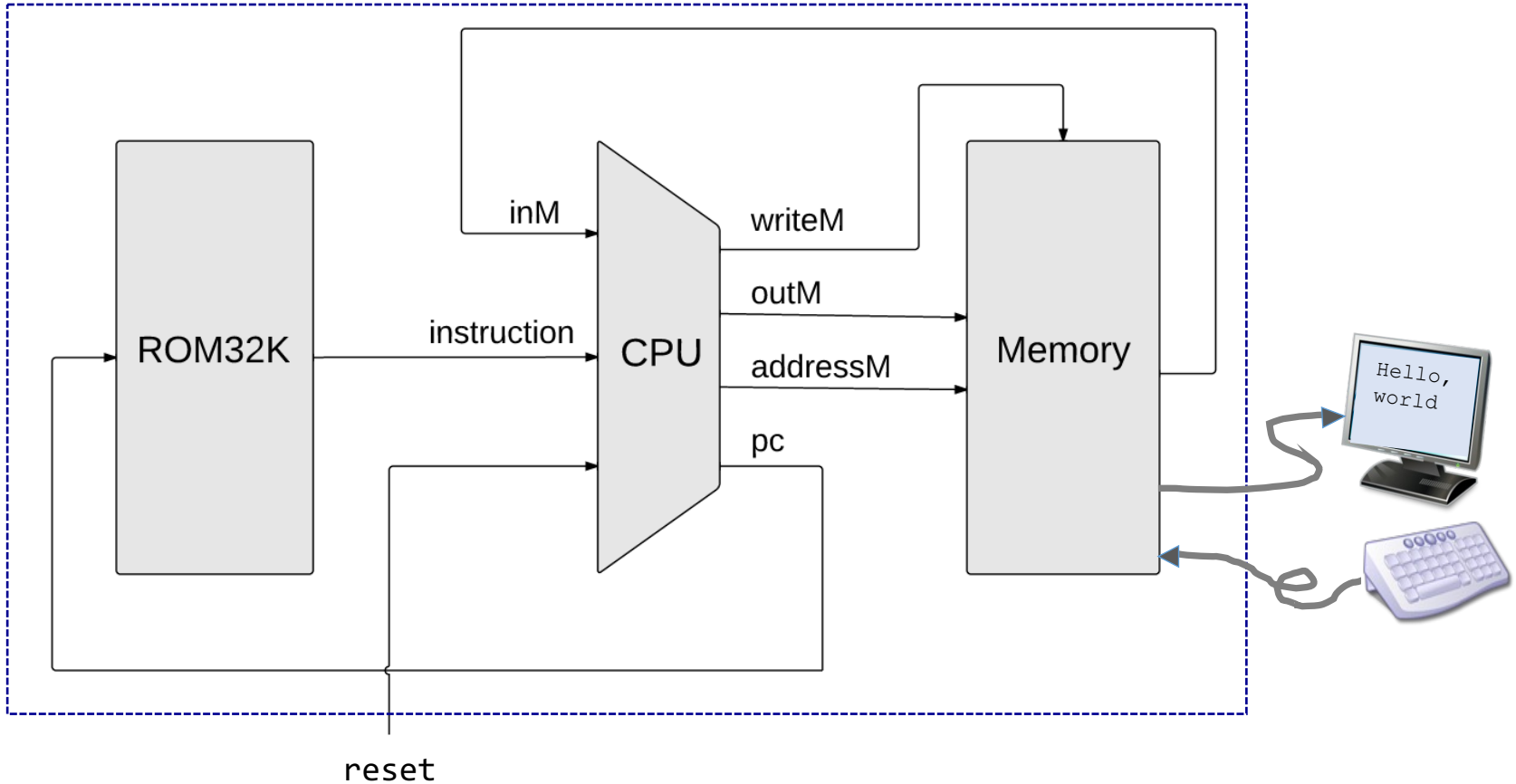
set `reset ← 1`, then

set `reset ← 0`

Computer implementation



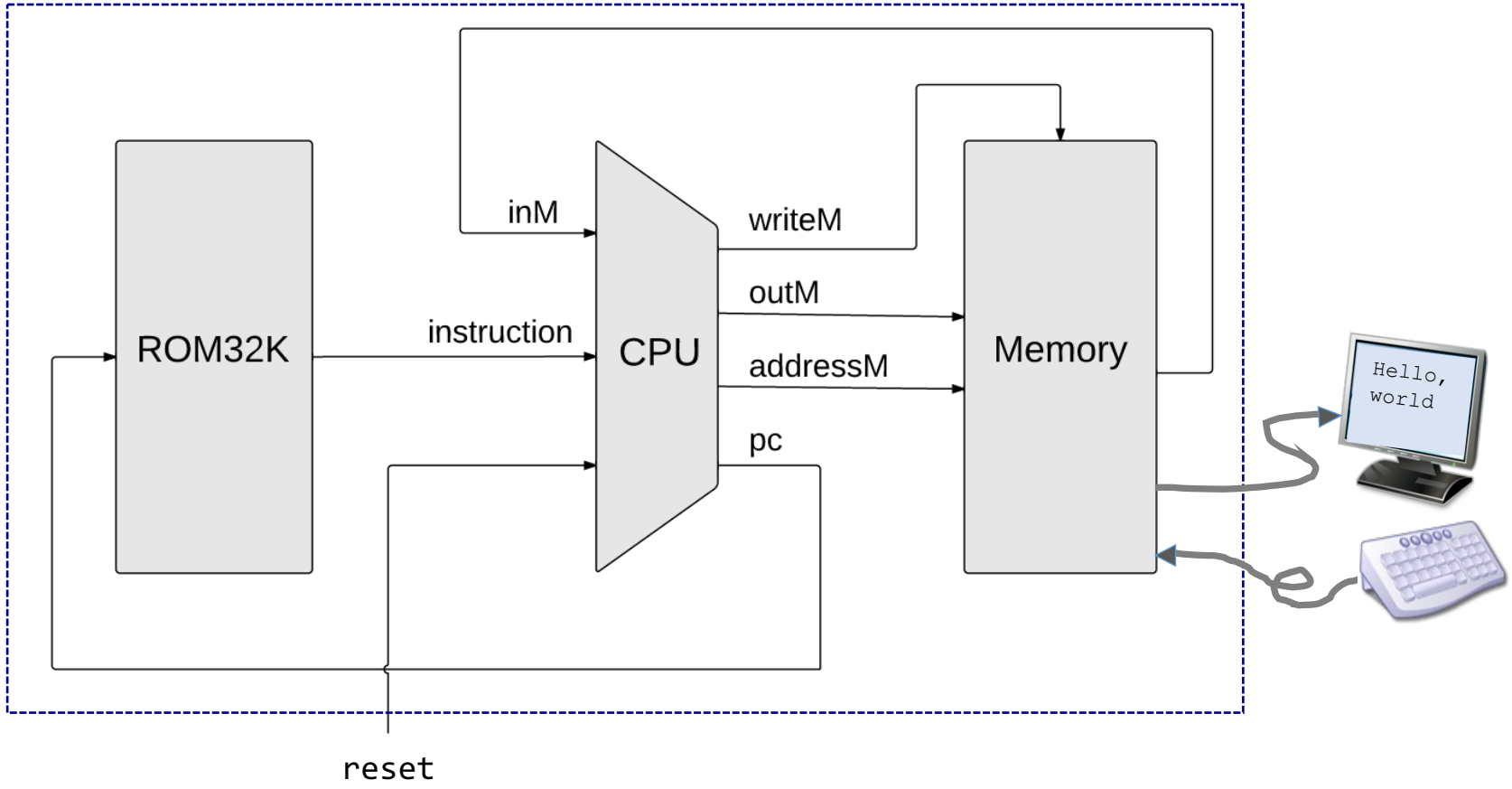
Computer implementation



Implementation

Connect the three chip-parts along these lines.

Computer implementation



Computer Architecture

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

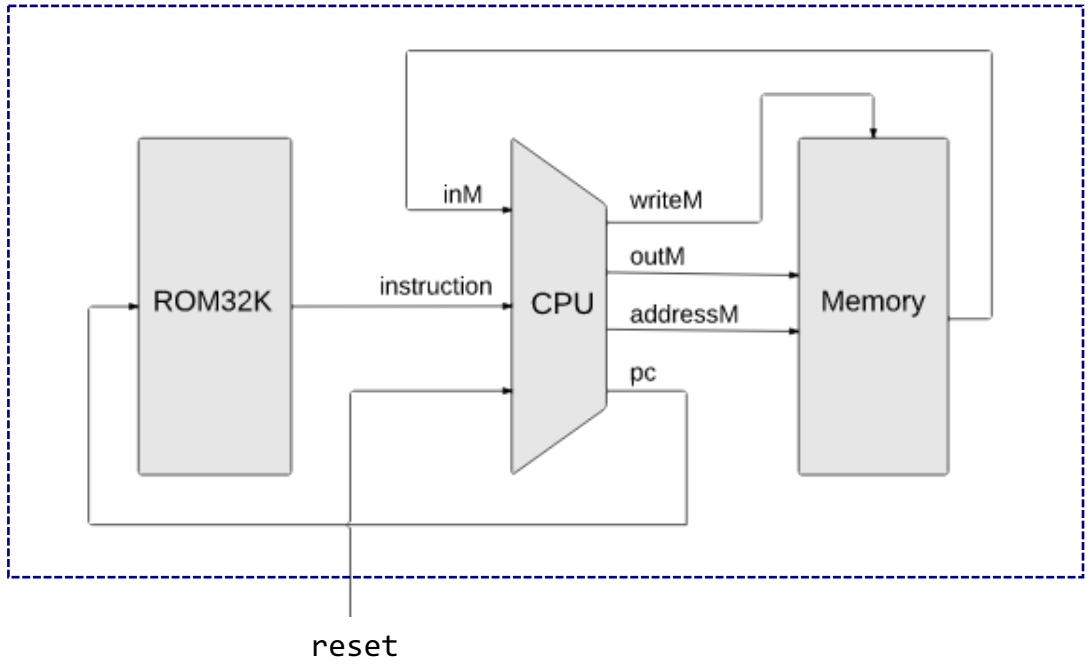
✓ Memory

✓ Computer

Project 5: Chips

- Project 5: Guidelines

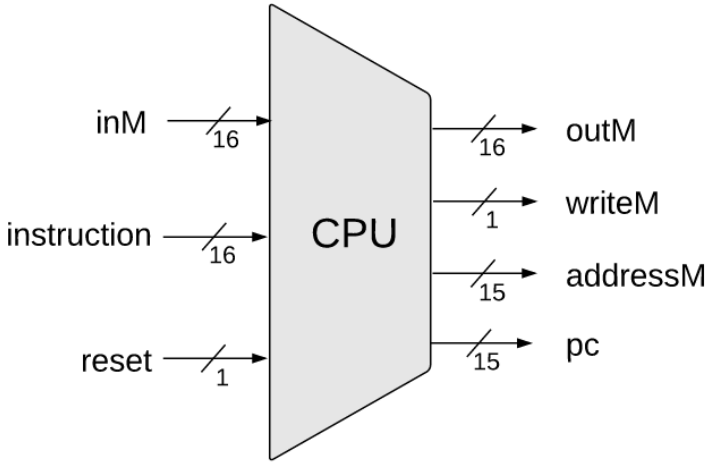
Computer Architecture



Project 5:

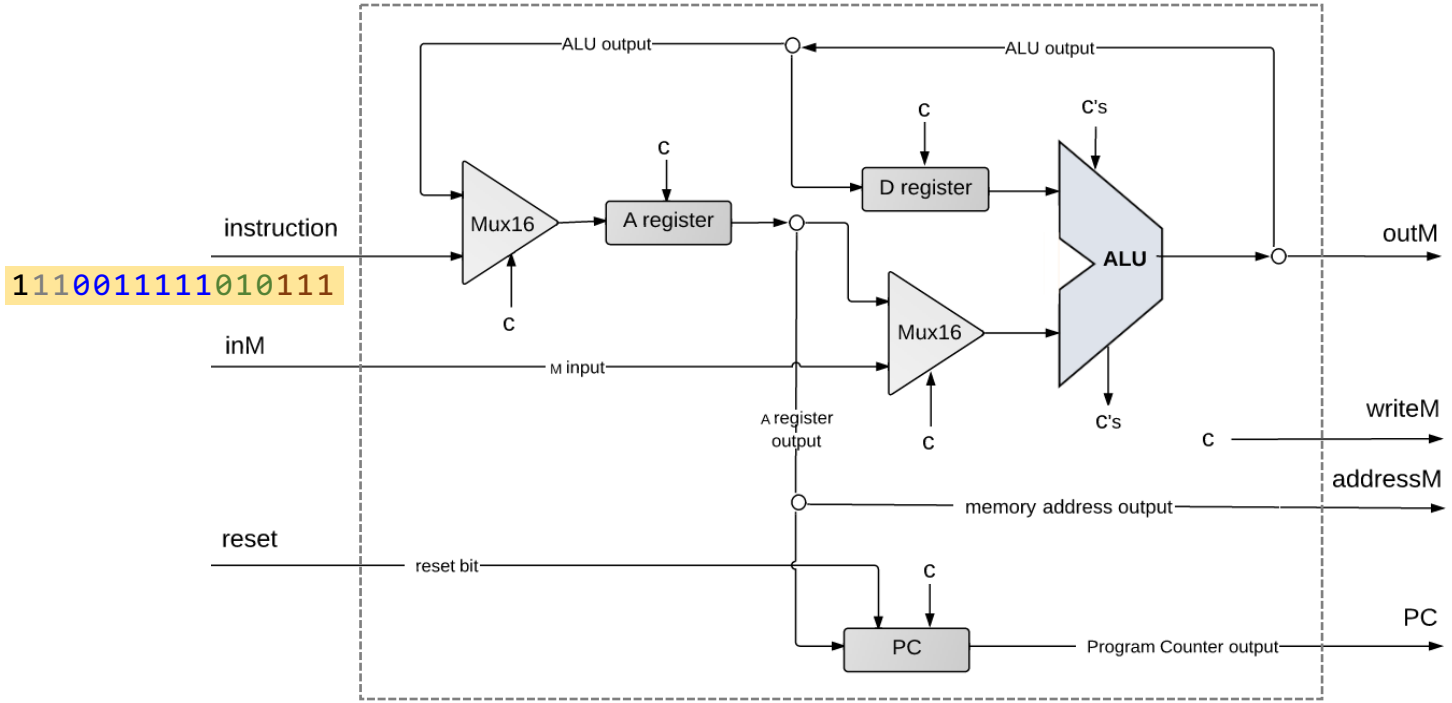
- CPU
- Memory
- Computer

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 you code here
}
```

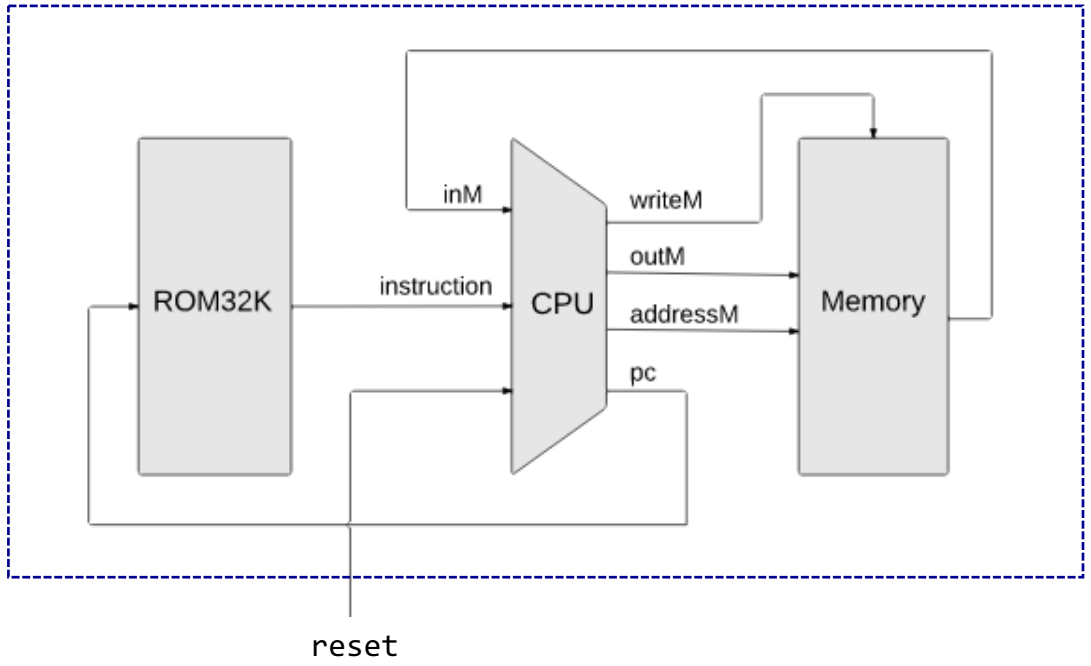
CPU



Implementation tips

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

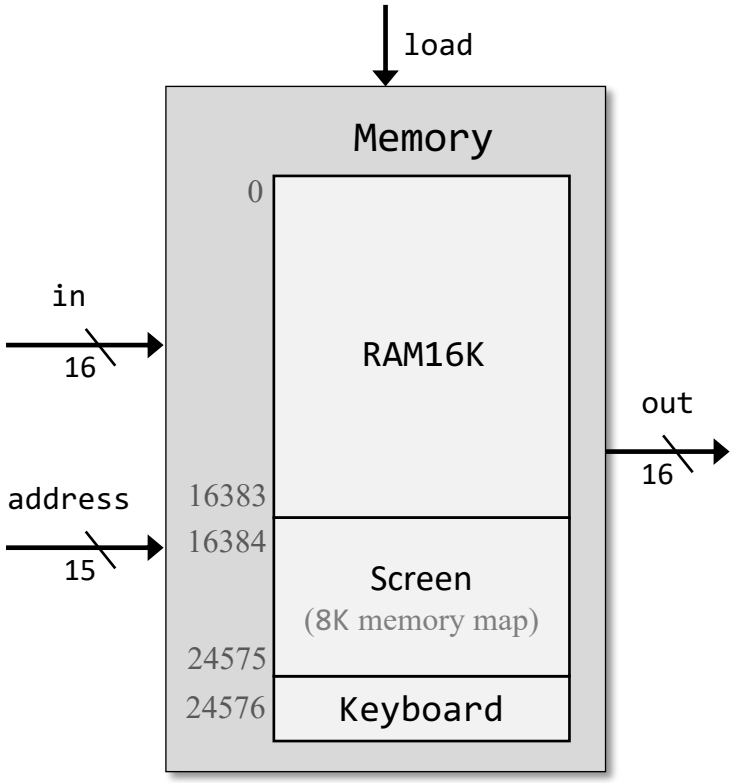
Computer Architecture



Project 5:

- ✓ CPU
- ➡ Memory
- Computer

Memory



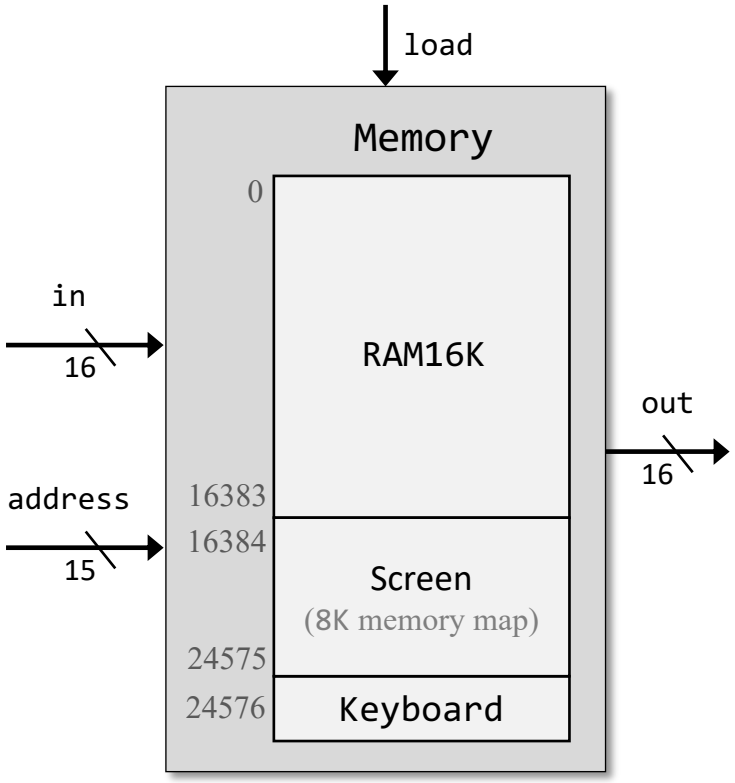
Memory.hdl

```
/** 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 address within the target memory-part
- Do the read/write, and output the memory-part[*address*] value to the `out` output.

Memory



Memory.hdl

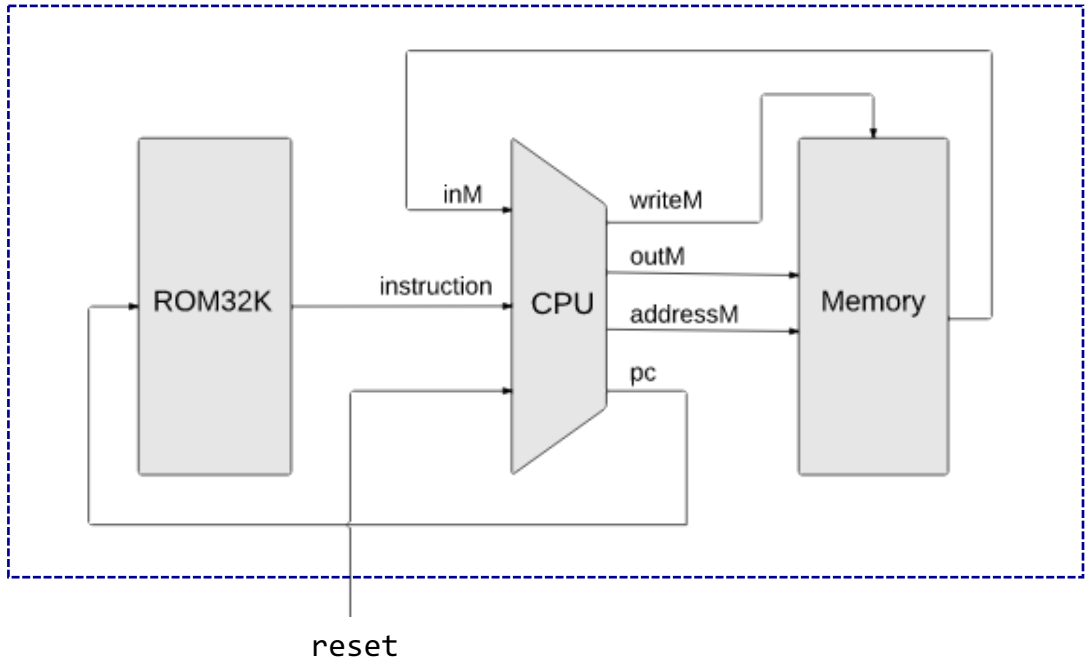
```
/** Complete address space of the computer's data memory,
    including RAM, screen memory map, and keyboard memory map.
    Outputs the value of the memory location specified by address.
    If (load==1), the in value is loaded into the memory location specified by address.

    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.

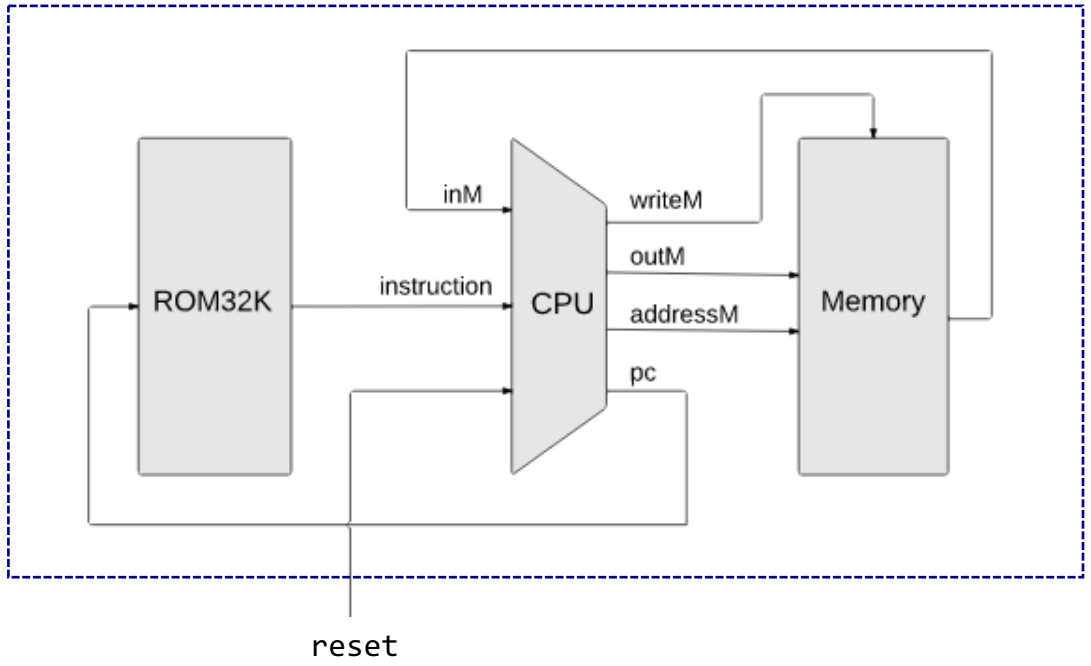
Computer



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

}
```

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:
  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 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

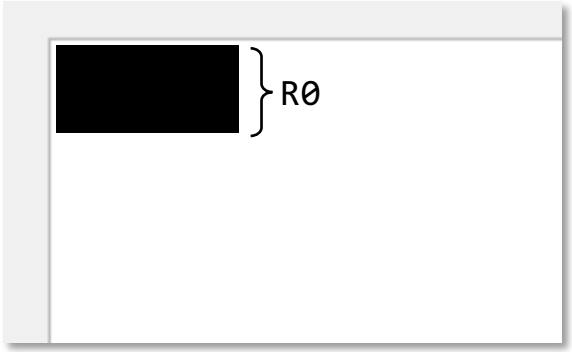
Testing logic

- 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:
  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.

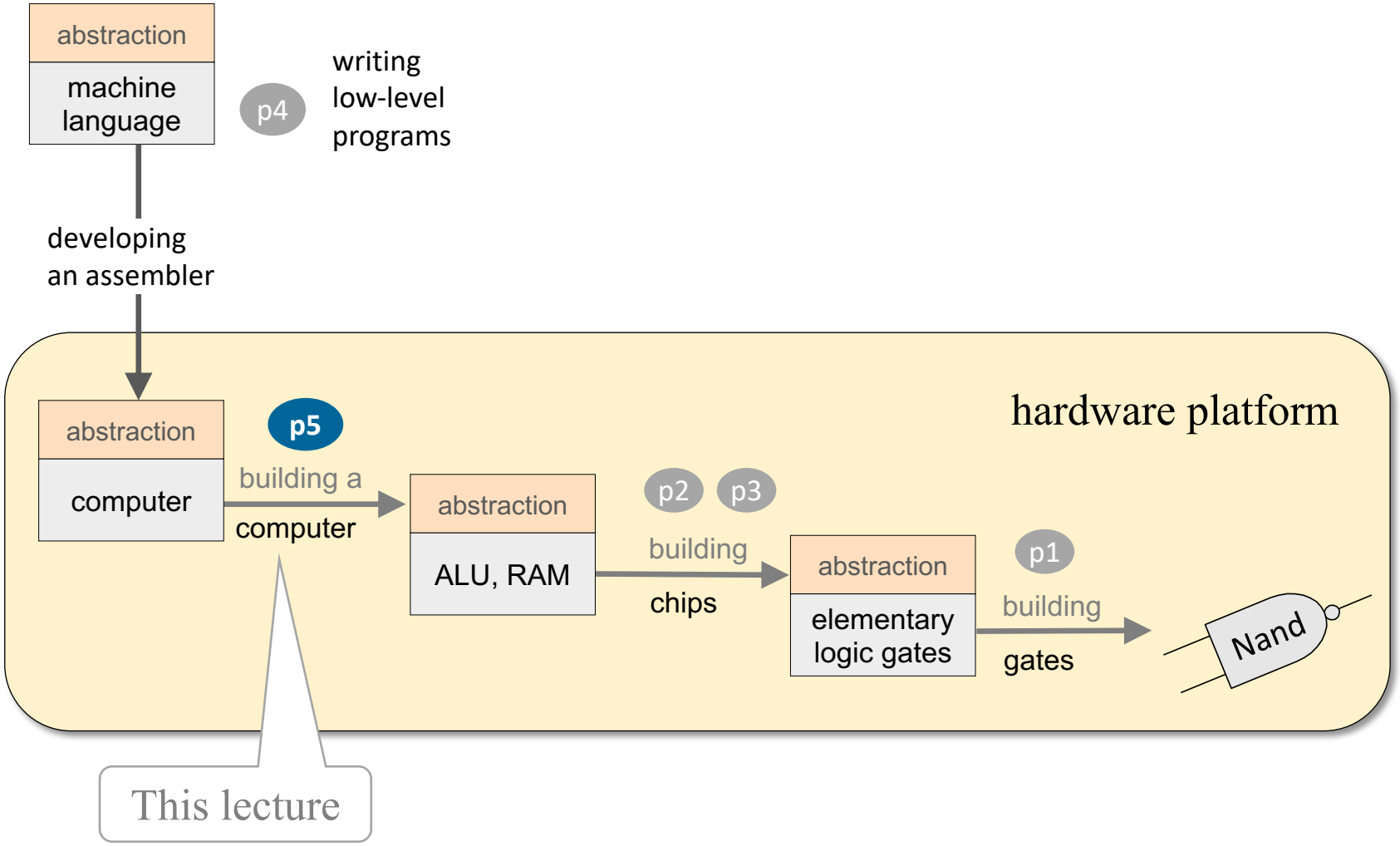
Rect.hack output:



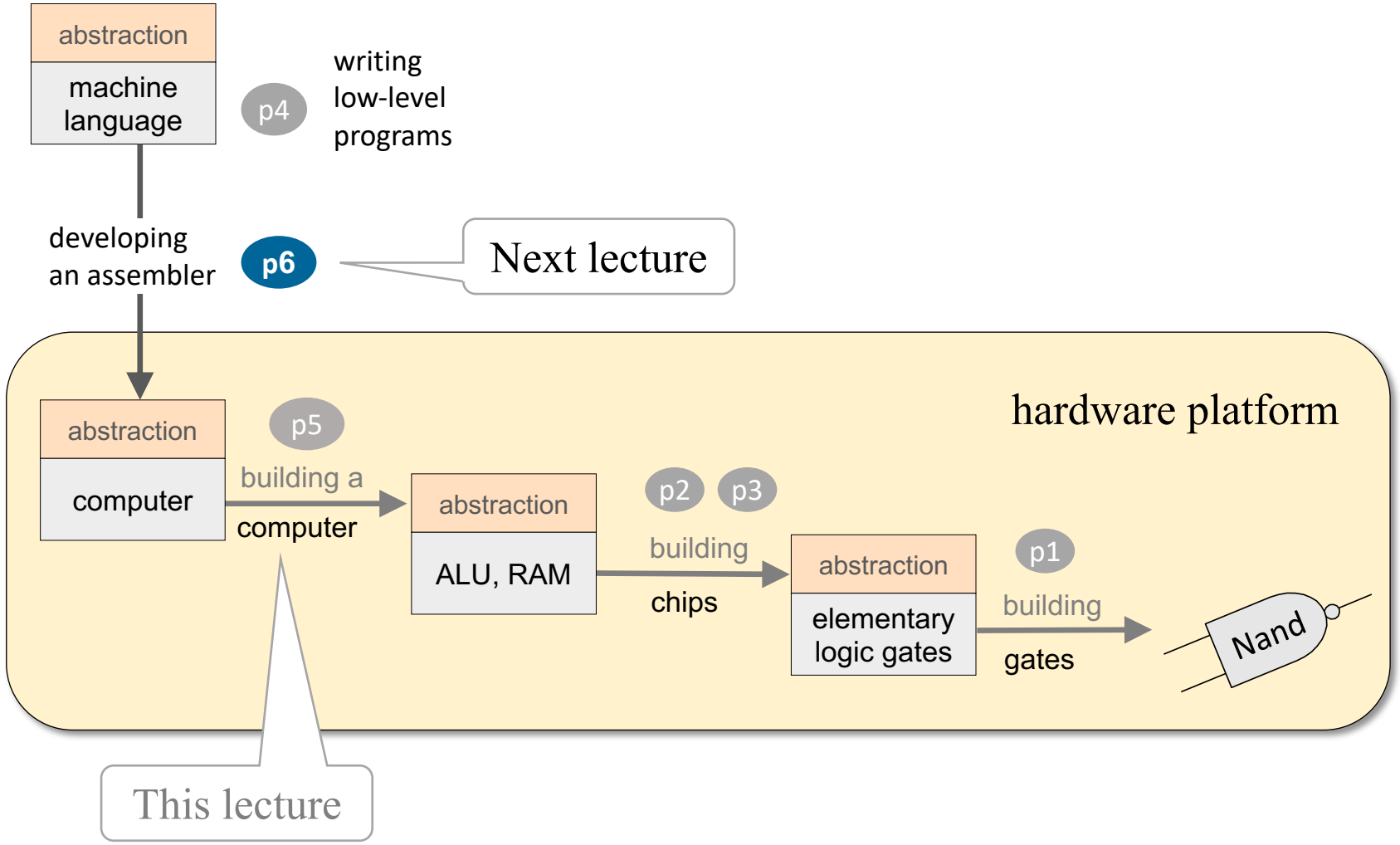
Test script

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

What's next?


abstraction
machine language
p4
writing low-level programs
developing an assembler
abstraction
computer
p5
building a computer
abstraction
ALU, RAM
p2
p3
building chips
abstraction
elementary logic gates
p1
building gates
Nand
hardware platform
This lecture

What's next?


abstraction
machine language
p4
writing low-level programs
developing an assembler
p6
Next lecture
hardware platform
abstraction
computer
p5
building a computer
abstraction
ALU, RAM
p2
p3
building chips
abstraction
elementary logic gates
p1
building gates
Nand
This lecture