CSE 390B, 2024 Spring

Building Academic Success Through Bottom-Up Computing

Building a Computer & Hack CPU Logic

Building a Computer, Hack CPU Interface, Project 6 Overview

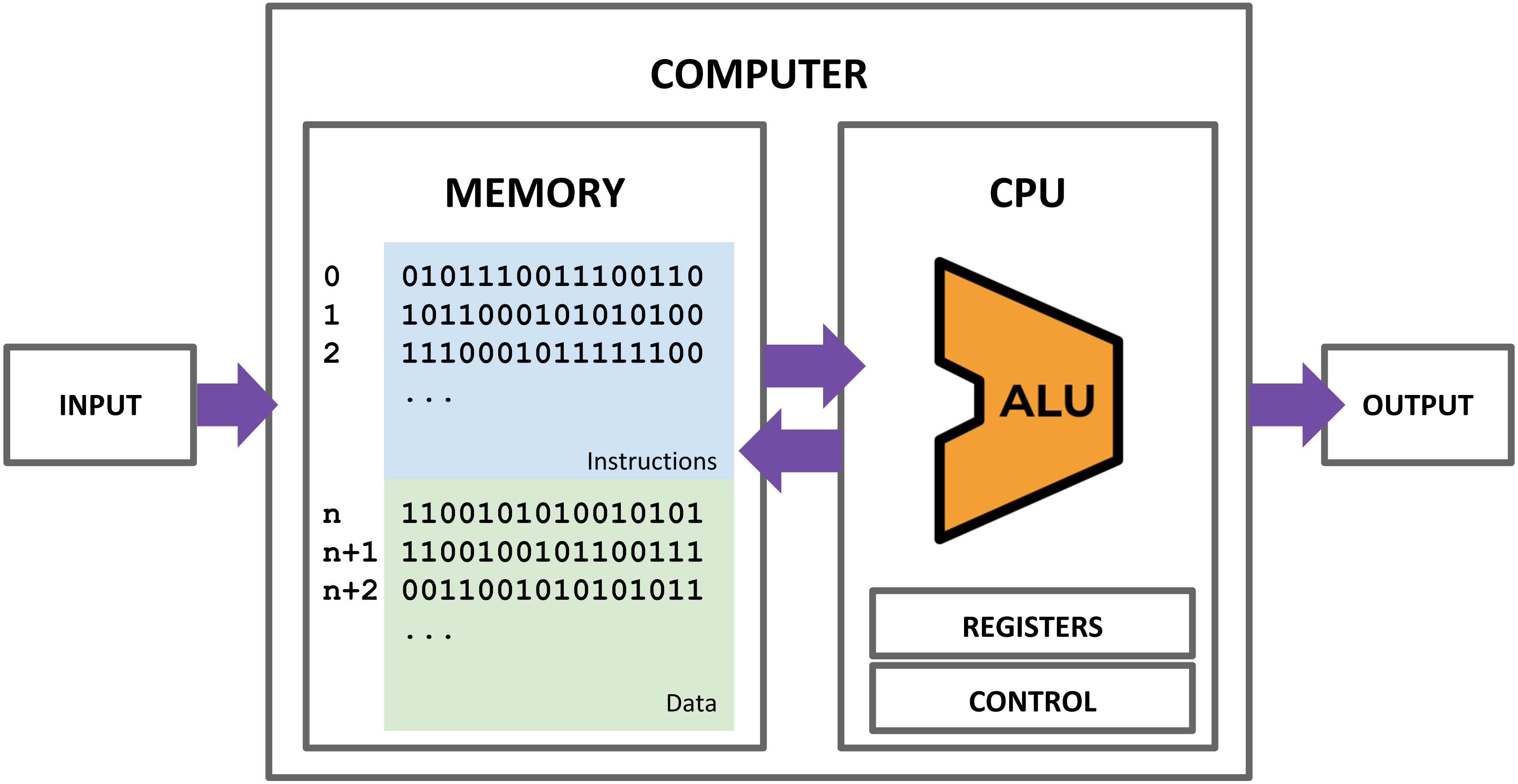
Lecture Outline

- **Building a Computer**
  - **Architecture, Fetch and Execute Cycle**
- Hack CPU Interface
  - Implementation and Operations
- Project 6 Overview
  - Mock Exam Problem and Project Tips

Building a Computer

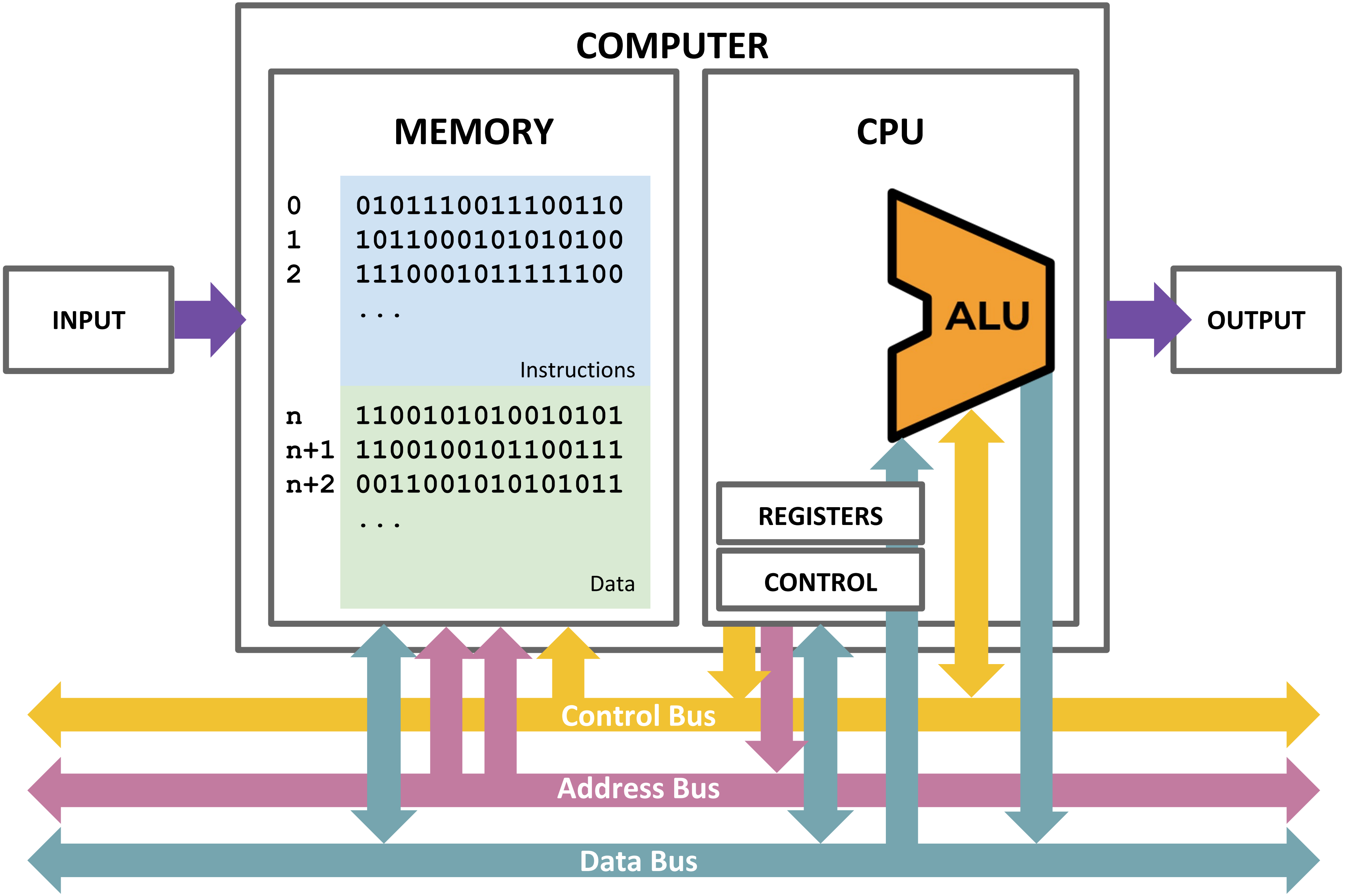
- All your hardware efforts are about to pay off!
- Perspective: **BUILDING A COMPUTER**
- In Project 6, you will build **`Computer.hdl`**, the final, top-level chip in this course
  - For all intents and purposes, a real computer
  - Simplified, but organization very similar to your laptop
- Project 7 onward, we will write software to make it useful

Von Neumann Architecture



```
0    0101110011100110
1    1011000101010100
2    1110001011111100
     ...
                 Instructions
n    1100101010010101
n+1  1100100101100111
n+2  0011001010101011
     ...
                         Data
```

Connecting the Computer: Buses



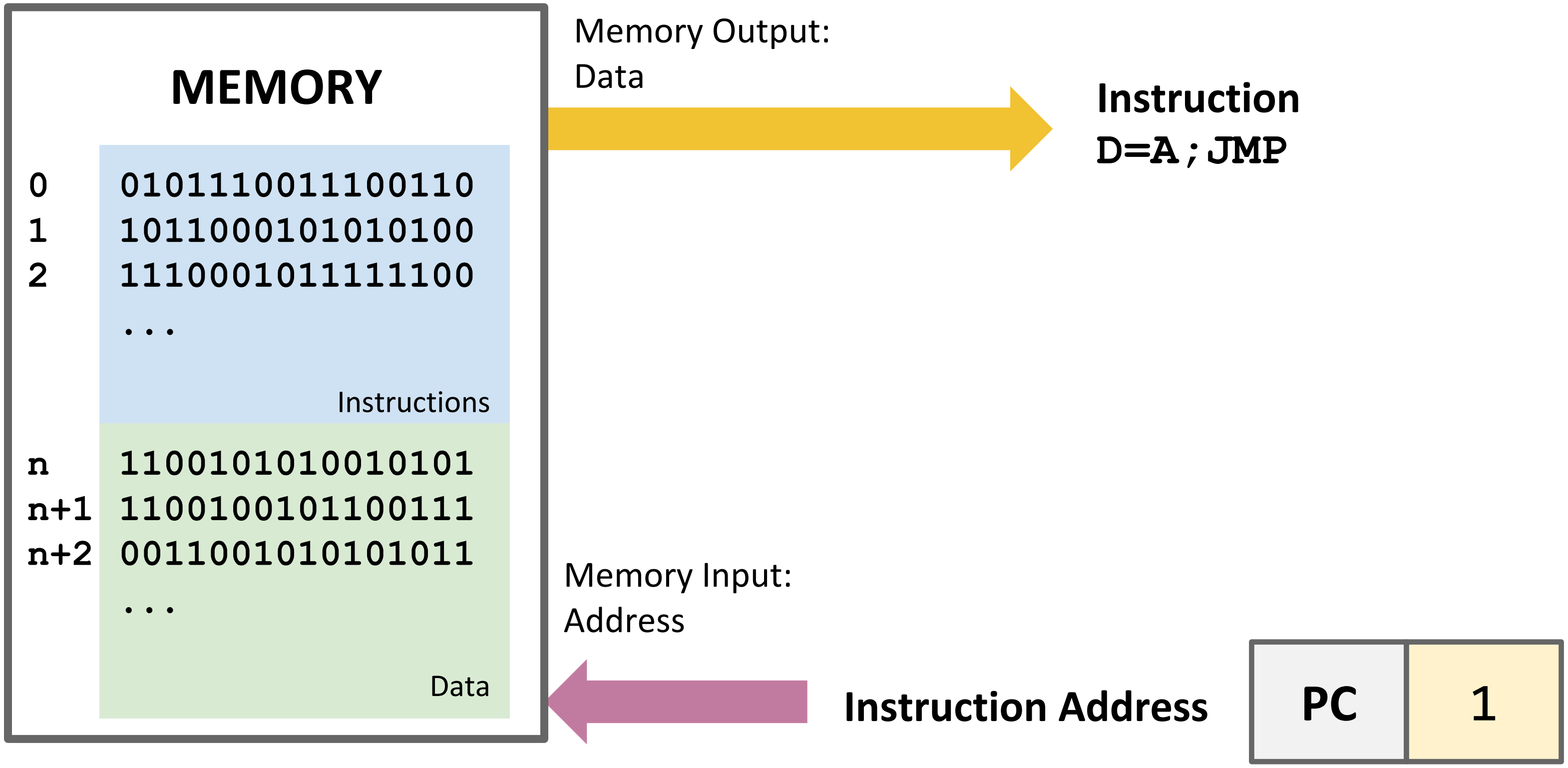
0	0101110011100110
1	1011000101010100
2	1110001011111100
	. . .
	Instructions
n	1100101010010101
n+1	1100100101100111
n+2	0011001010101011
	. . .
	Data

Basic CPU Loop

- Repeat forever:
  - **Fetch** an instruction from the program memory
  - **Execute** that instruction

Fetching

- Specify which instruction to read as the address input to our memory
- Data output: actual bits of the instruction



MEMORY

0	0101110011100110
1	1011000101010100
2	1110001011111100
	...
	Instructions
n	1100101010010101
n+1	1100100101100111
n+2	0011001010101011
	...
	Data

Memory Output: Data

Instruction
`D=A;JMP`

Memory Input: Address

Instruction Address

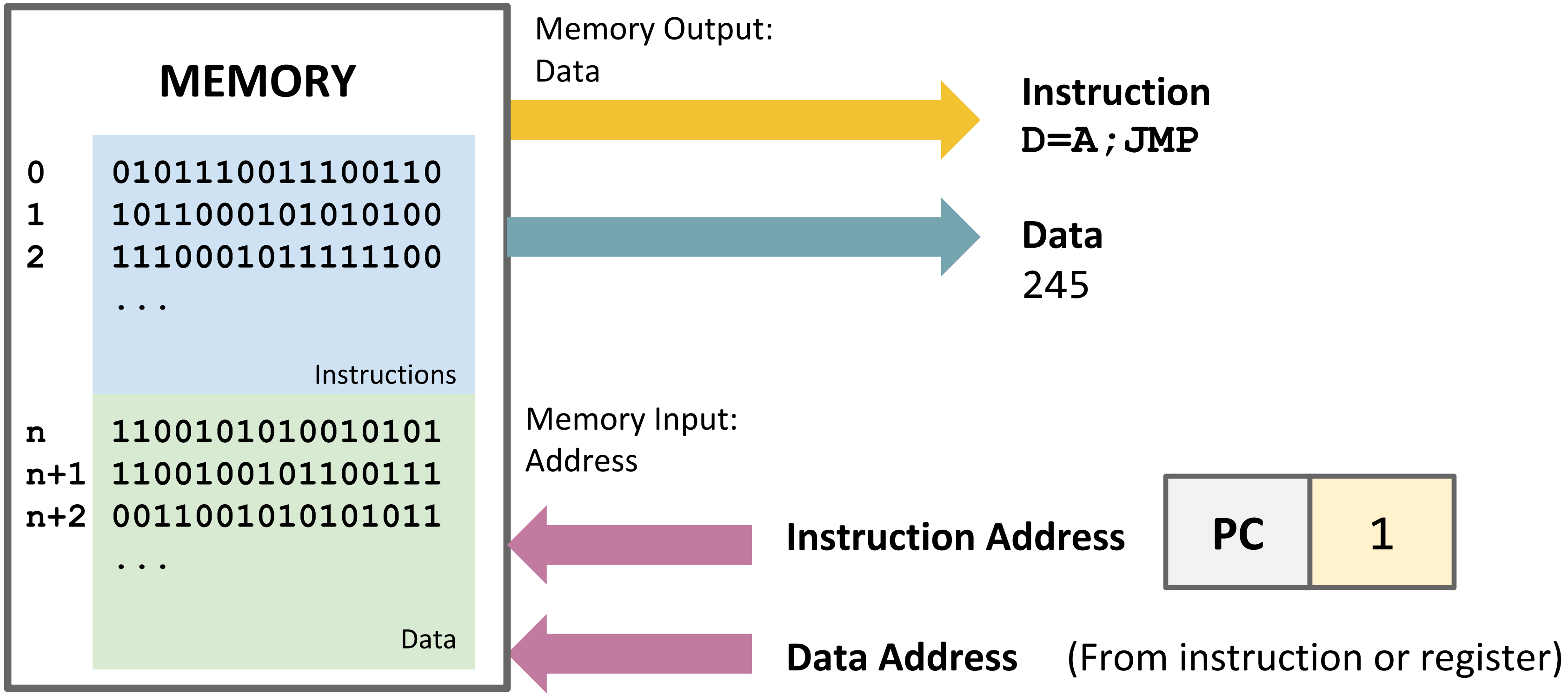
PC	1

Executing

- The instruction bits describe exactly “what to do”
  - A-instruction or C-instruction?
  - Which operation for the ALU?
  - What memory address to read? To write?
  - If I should jump after this instruction, and where?

- Executing the instruction involves data of some kind
  - Accessing registers
  - Accessing memory

Combining Fetch & Execute



MEMORY

0	0101110011100110
1	1011000101010100
2	1110001011111100
	. . .
	Instructions
n	1100101010010101
n+1	1100100101100111
n+2	0011001010101011
	. . .
	Data

Memory Output: Data

Instruction
`D=A;JMP`

Data
245

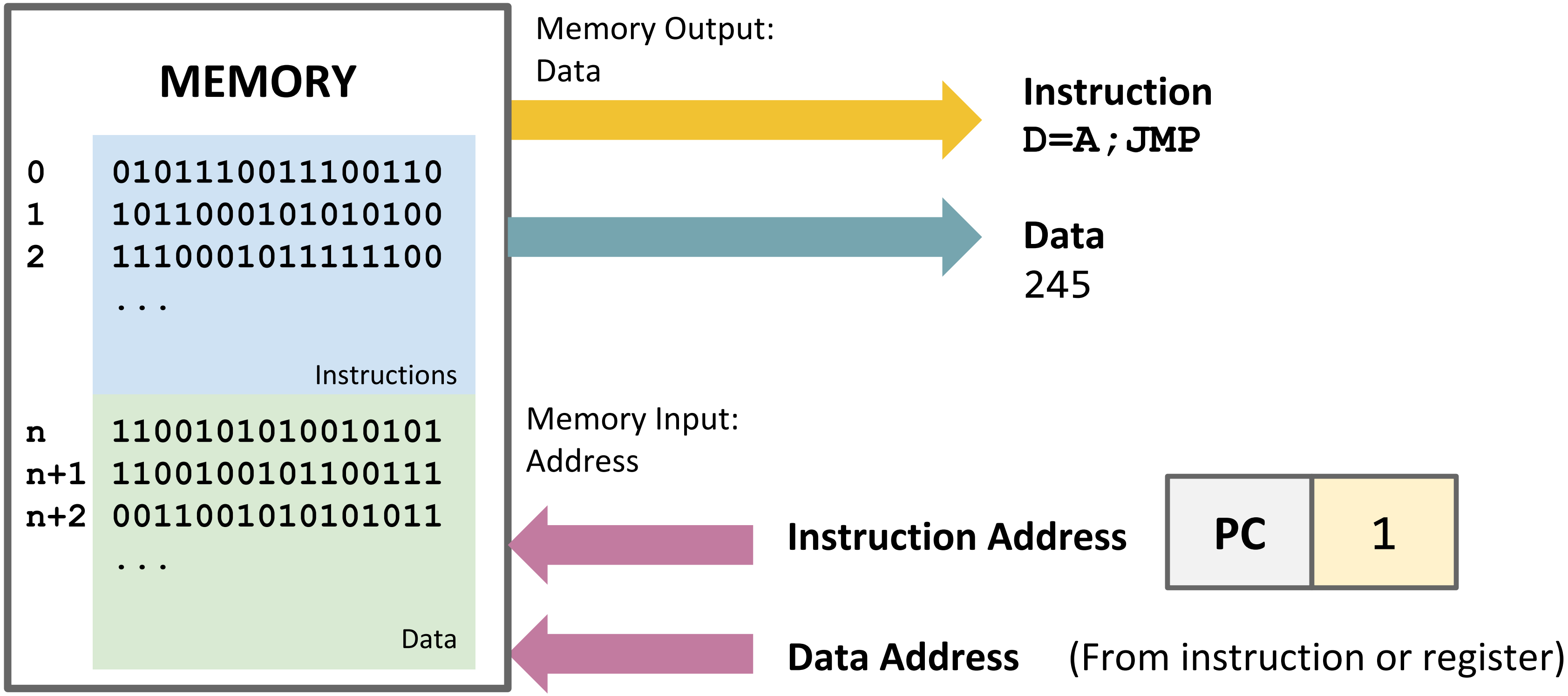
Memory Input: Address

Instruction Address

PC	1

Data Address (From instruction or register)

Combining Fetch & Execute



MEMORY

0	0101110011100110
1	1011000101010100
2	1110001011111100
	...
	Instructions
n	1100101010010101
n+1	1100100101100111
n+2	0011001010101011
	...
	Data

Memory Output: Data

Instruction
`D=A;JMP`

Data
245

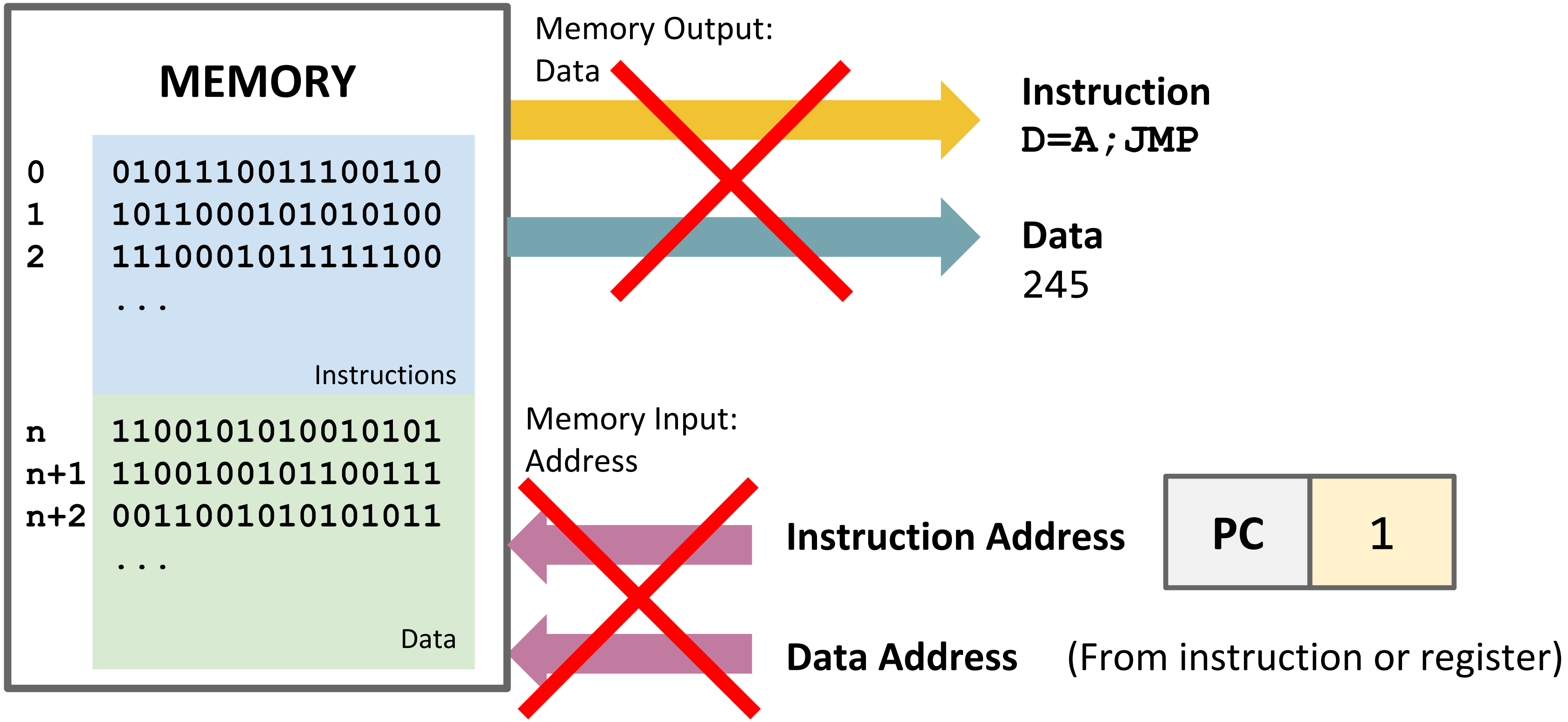
Memory Input: Address

Instruction Address PC 1

Data Address (From instruction or register)

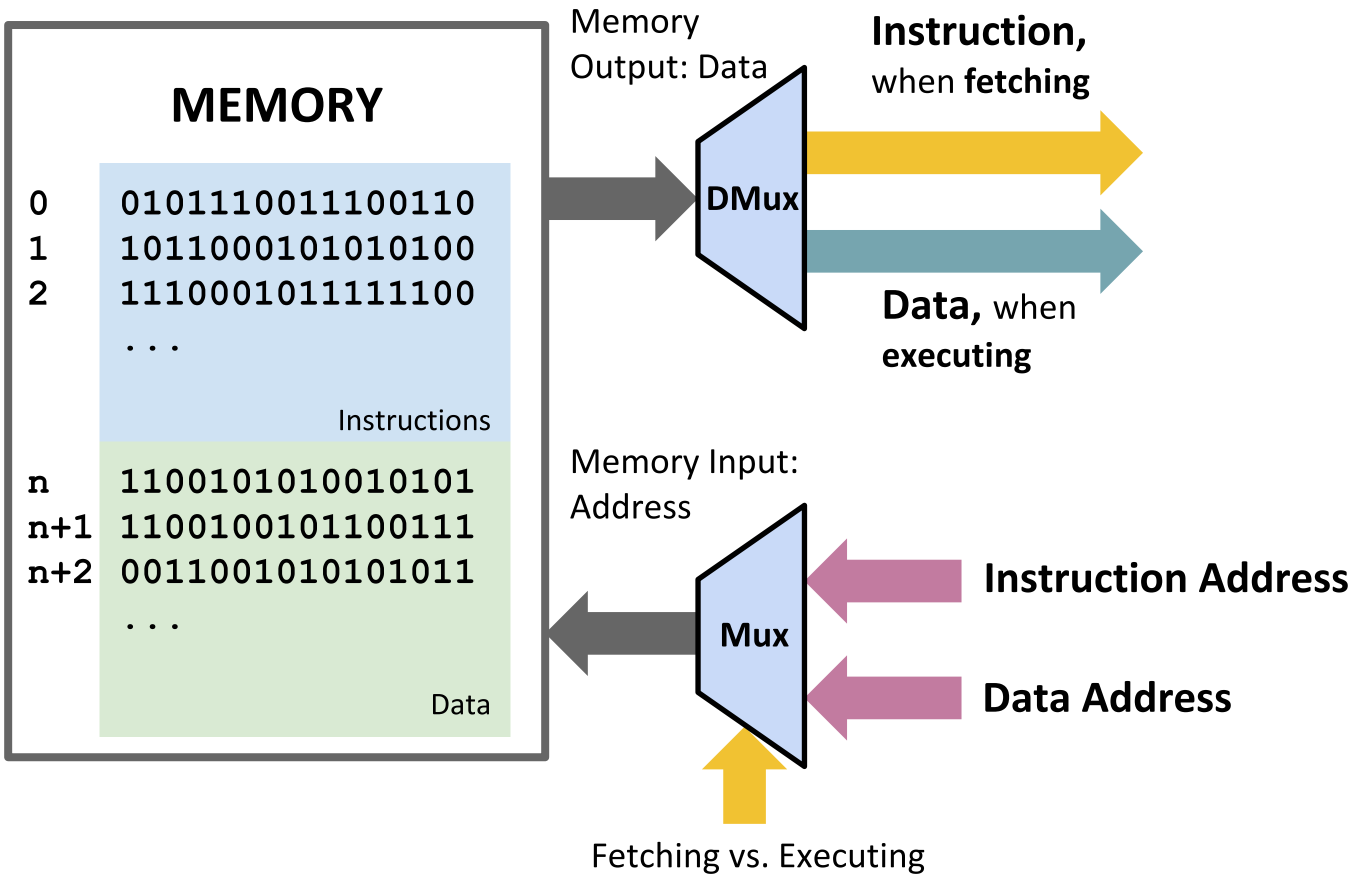
- Could we implement with **`RAM16K.hdl`**?
  - (Hint: Think about the I/O of RAM)

Combining Fetch & Execute



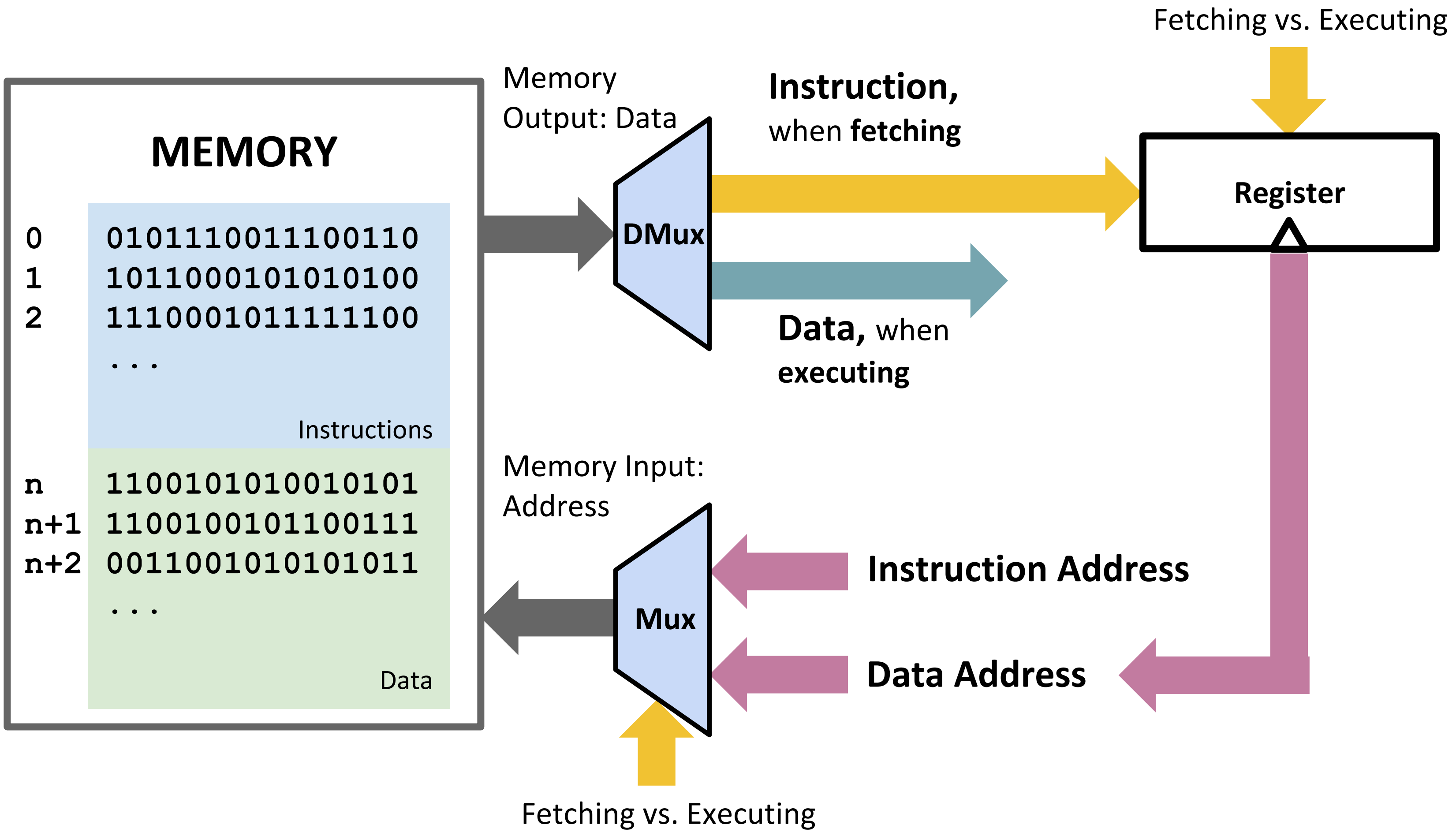
- Could we implement with **`RAM16K.hdl`**?
  - **No!** Our memory chips only have one input and one output

Solution 1: Handling Single Input / Output



- Can use multiplexing to share a single input or output

Solution 1: Fetching / Executing Separately



Fetching vs. Executing

MEMORY

```
0    0101110011100110
1    1011000101010100
2    1110001011111100
     . . .
                 Instructions
n    1100101010010101
n+1  1100100101100111
n+2  0011001010101011
     . . .
                         Data
```

Memory Output: Data

Memory Input: Address

Instruction, when **fetching**

Data, when **executing**

Instruction Address

Data Address

Fetching vs. Executing

- Need to store fetched instruction so it's available during execution phase

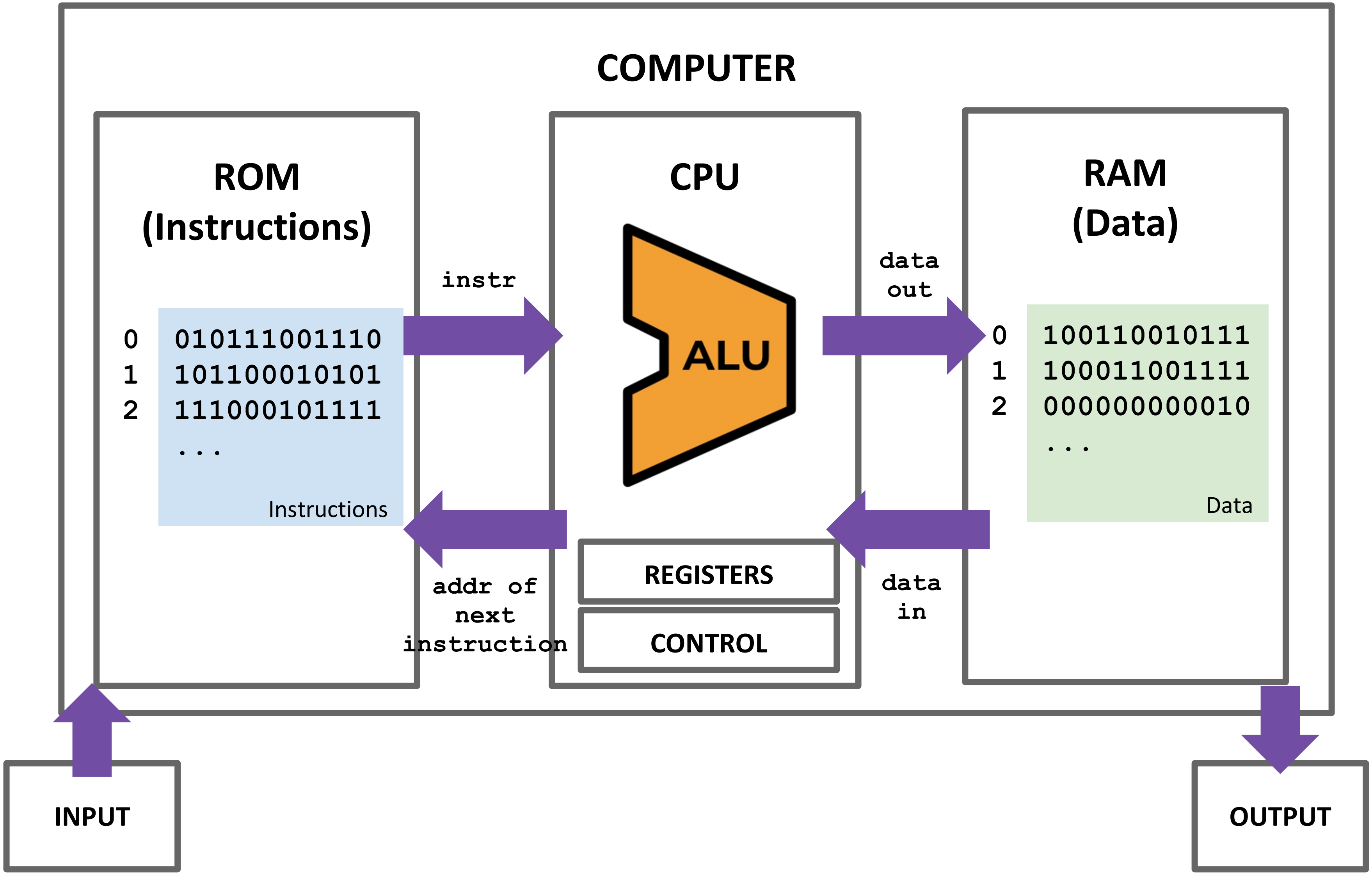
Solution 2: Separate Memory Units

- Separate instruction memory and data memory into two different chips
  - Each can be independently addressed, read from, written to
- Pros:
  - Simpler to implement
- Cons:
  - Fixed size of each partition, rather than flexible storage
  - Two chips → redundant circuitry

Lecture Outline

- Building a Computer
  - Architecture, Fetch and Execute Cycle
- **Hack CPU Interface**
  - **Implementation and Operations**
- Project 6 Overview
  - Mock Exam Problem and Project Tips

Hack CPU



COMPUTER

ROM
(Instructions)

```
0  010111001110
1  101100010101
2  111000101111
   ...
```

Instructions

instr

addr of
next
instruction

CPU

ALU

REGISTERS

CONTROL

data
out

data
in

RAM
(Data)

```
0  100110010111
1  100011001111
2  000000000010
   ...
```

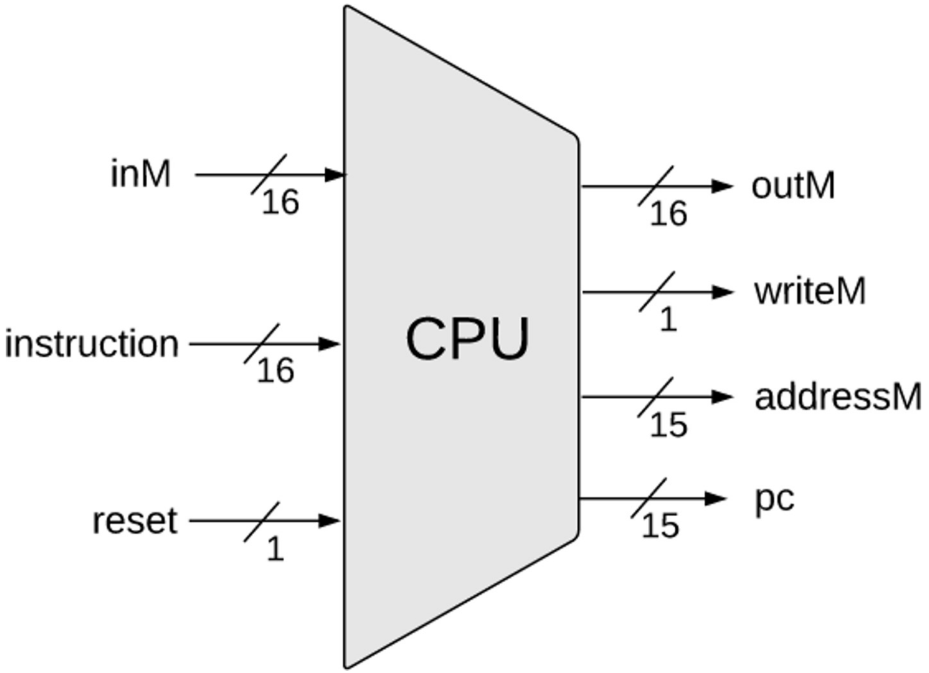
Data

INPUT

OUTPUT

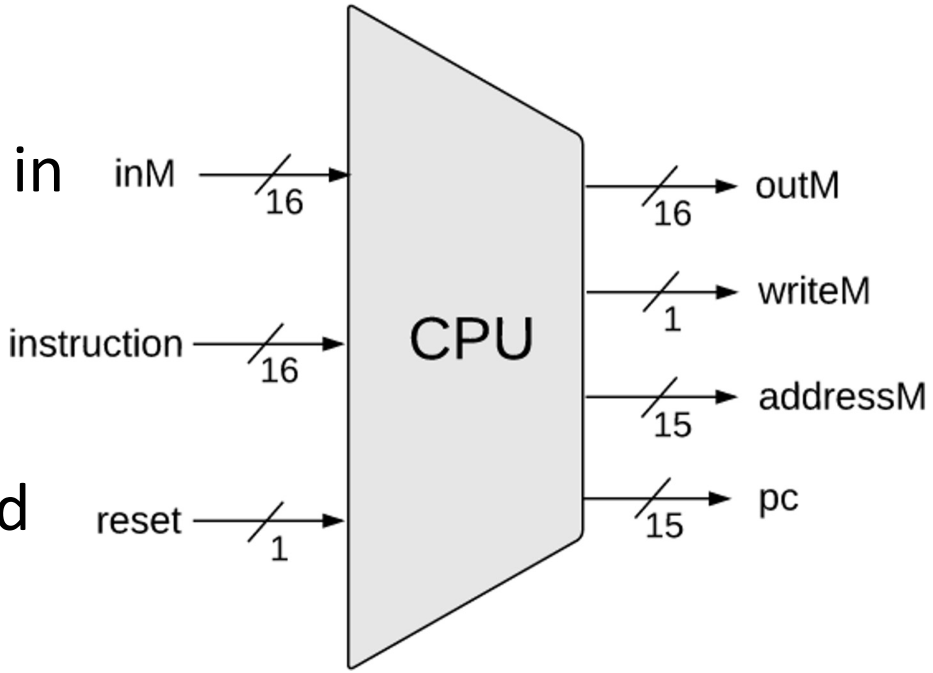
Hack CPU Interface Inputs

- **`inM`**: Value coming from memory
- **`instruction`**: 16-bit instruction
- **`reset`**: if 1, reset the program

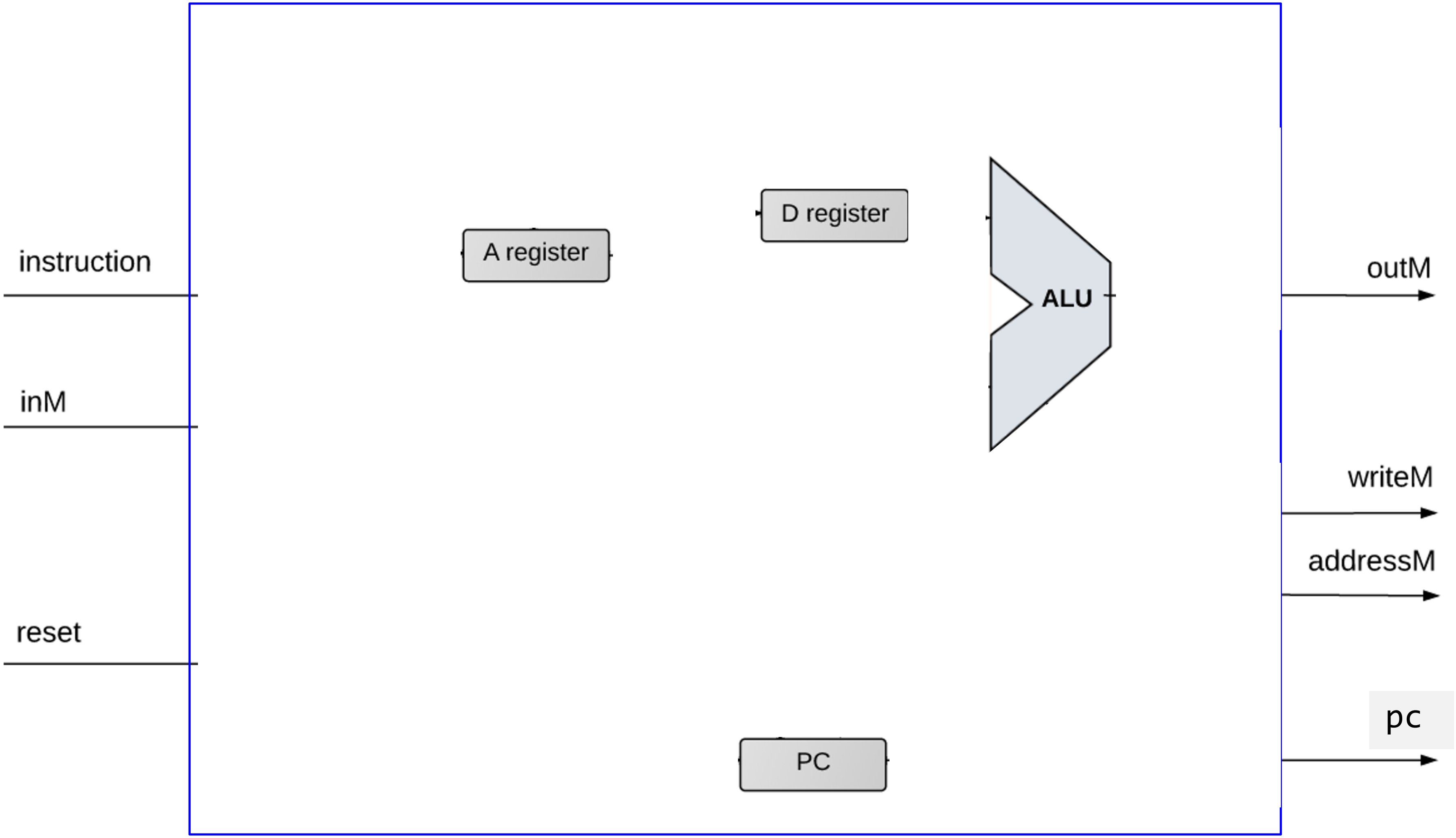


Hack CPU Interface Outputs

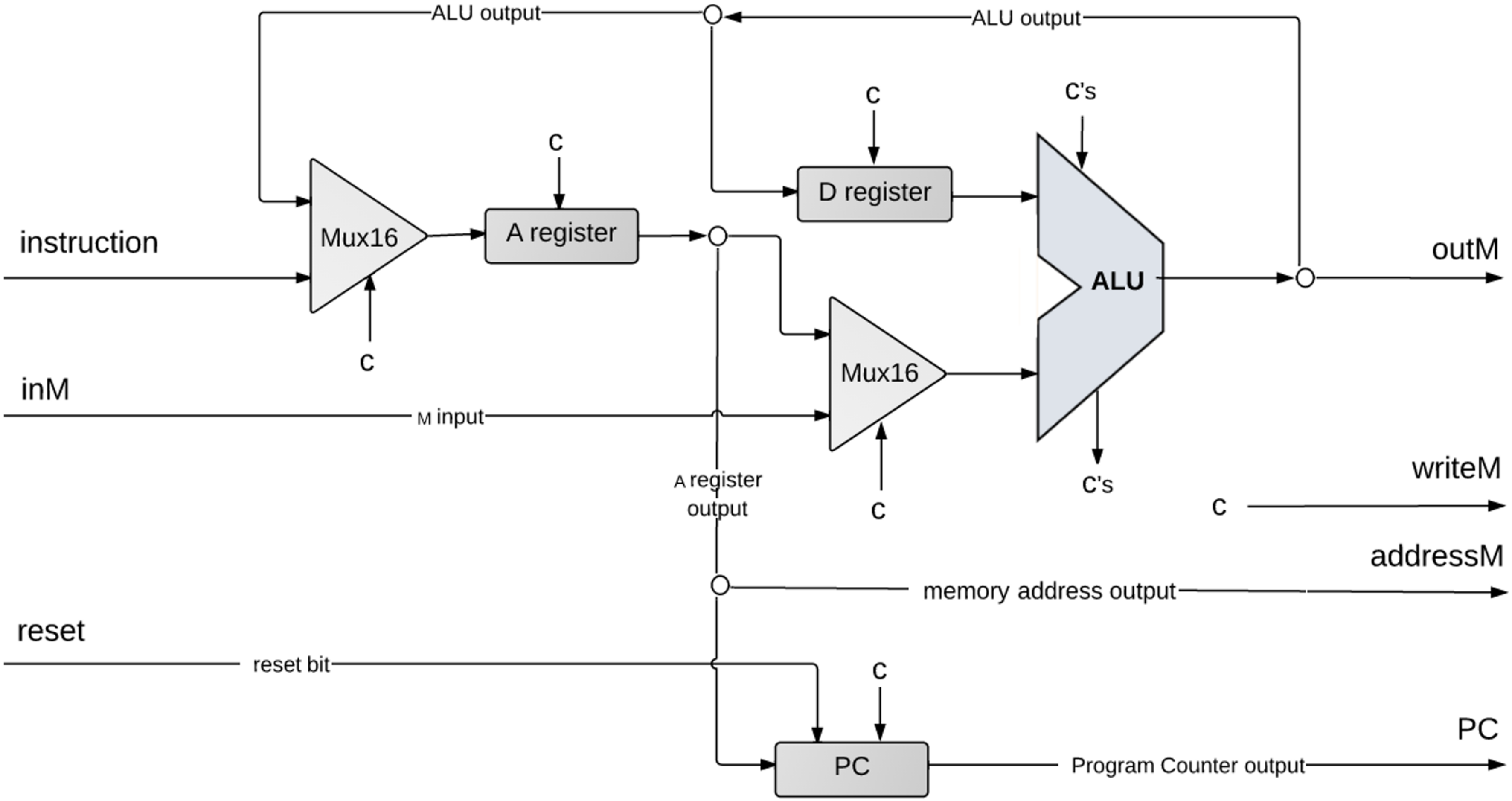
- **outM**: value used to update memory if writeM is 1
- **writeM**: if 1, update value in memory at addressM with outM
- **addressM**: address to read from or write to in memory
- **pc**: address of next instruction to be fetched from memory



Hack CPU Implementation

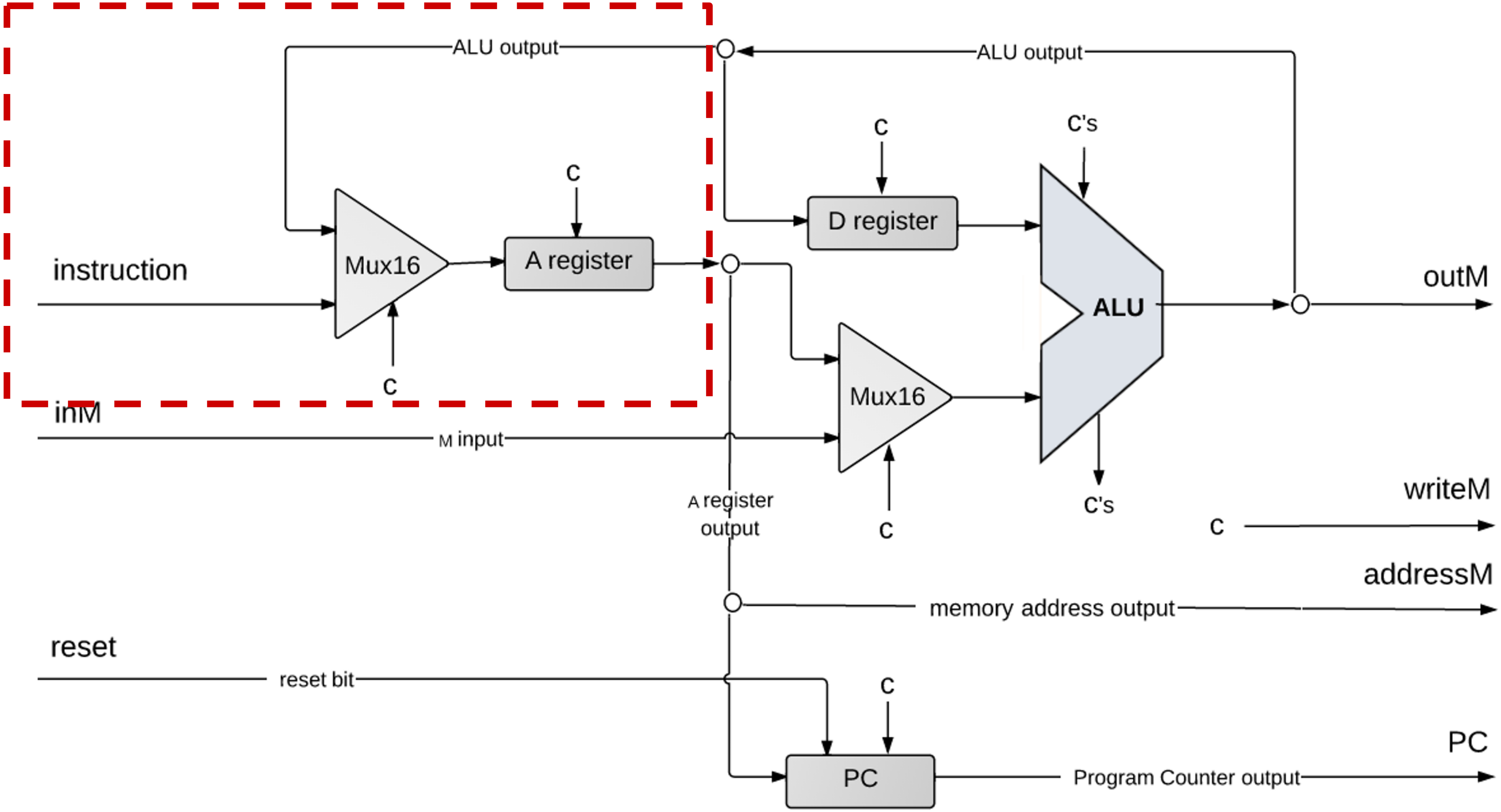


Hack CPU Implementation



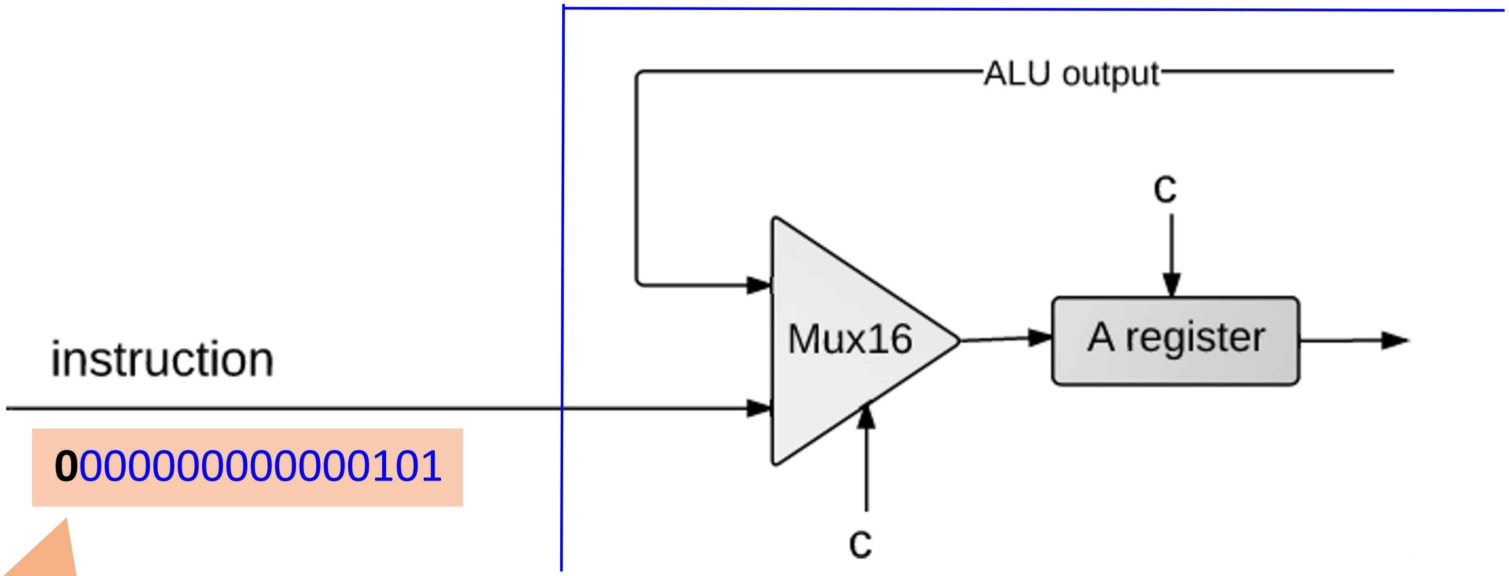
(each "C" symbol represents a control bit)

CPU Operation: Instruction Handling



(each "c" symbol represents a control bit)

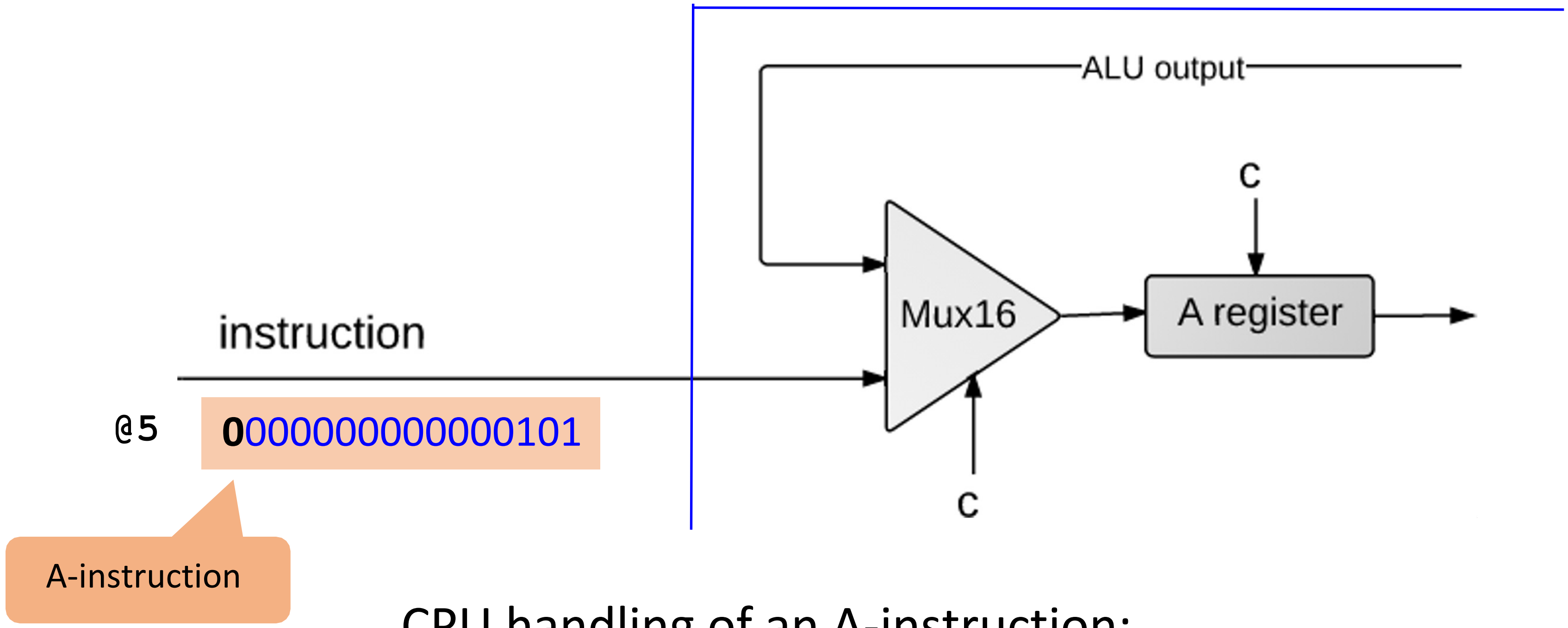
CPU Operation: Instruction Handling



@5 0000000000000101

A-instruction

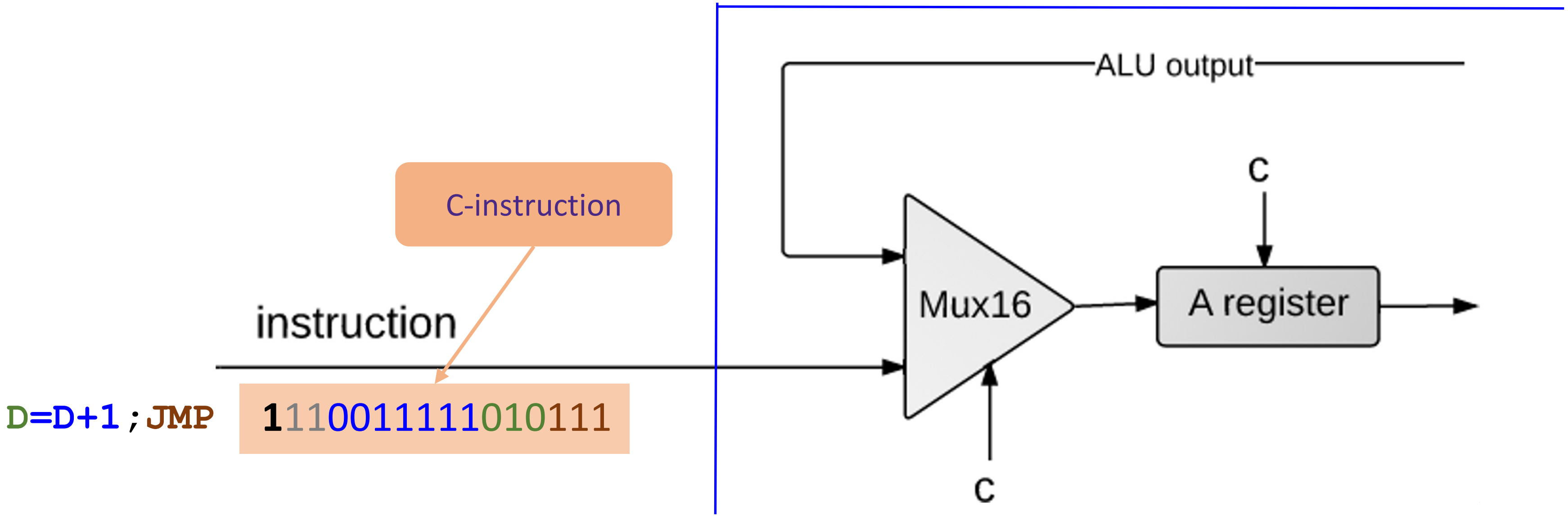
CPU Operation: Instruction Handling



<u>CPU handling of an A-instruction:</u>

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

CPU Operation: Instruction Handling



C-instruction

instruction

D=D+1;JMP 1110011111010111

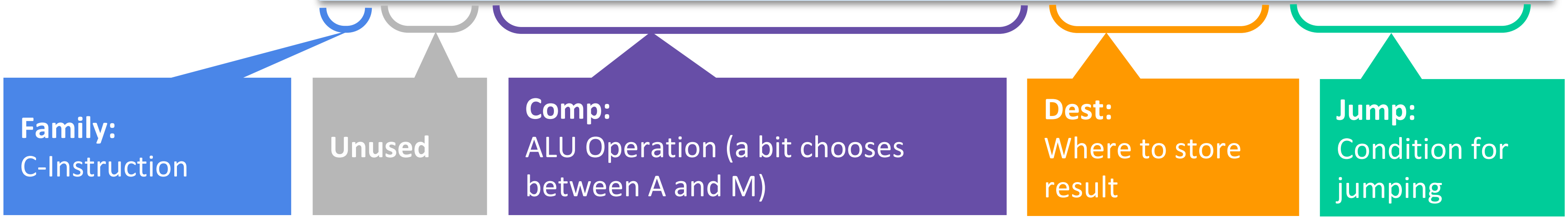
Hack: C-Instructions

- Symbolic:

```
dest = comp ; jump
```

- Binary:

```
1 1 1 a c1 c2 c3 c4 c5 c6 d1 d2 d3 j1 j2 j3
```



Family: C-Instruction

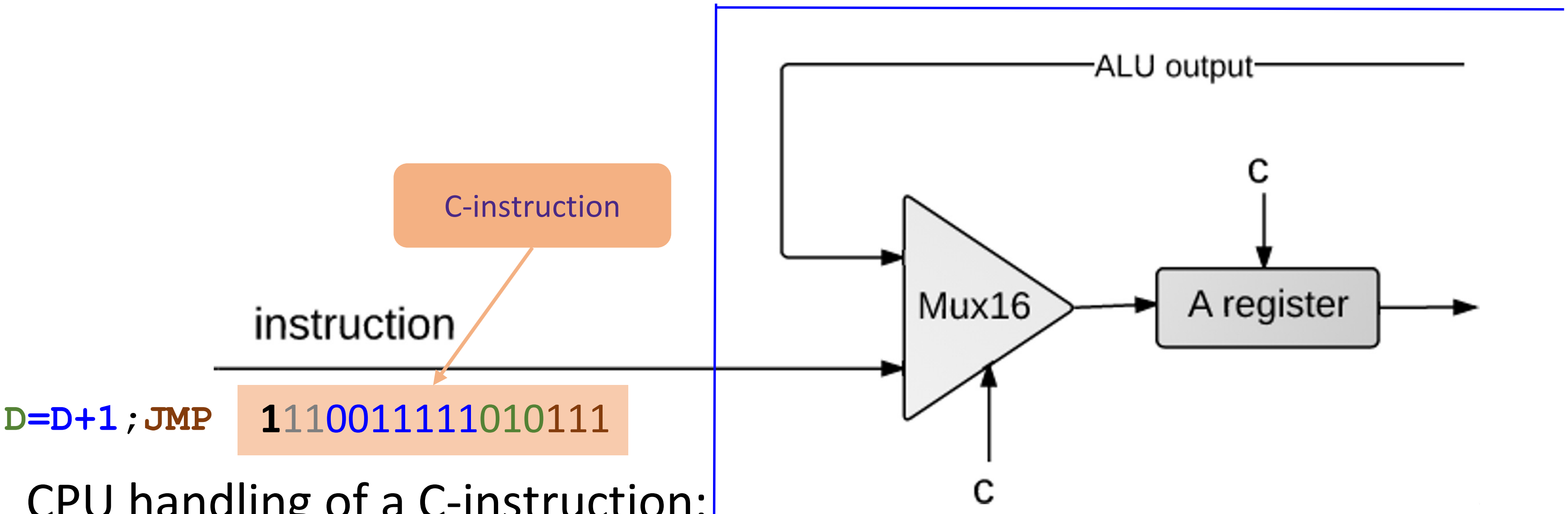
Unused

Comp: ALU Operation (a bit chooses between A and M)

Dest: Where to store result

Jump: Condition for jumping

CPU Operation: Instruction Handling



`D=D+1;JMP` 1110011111010111

CPU handling of a C-instruction:

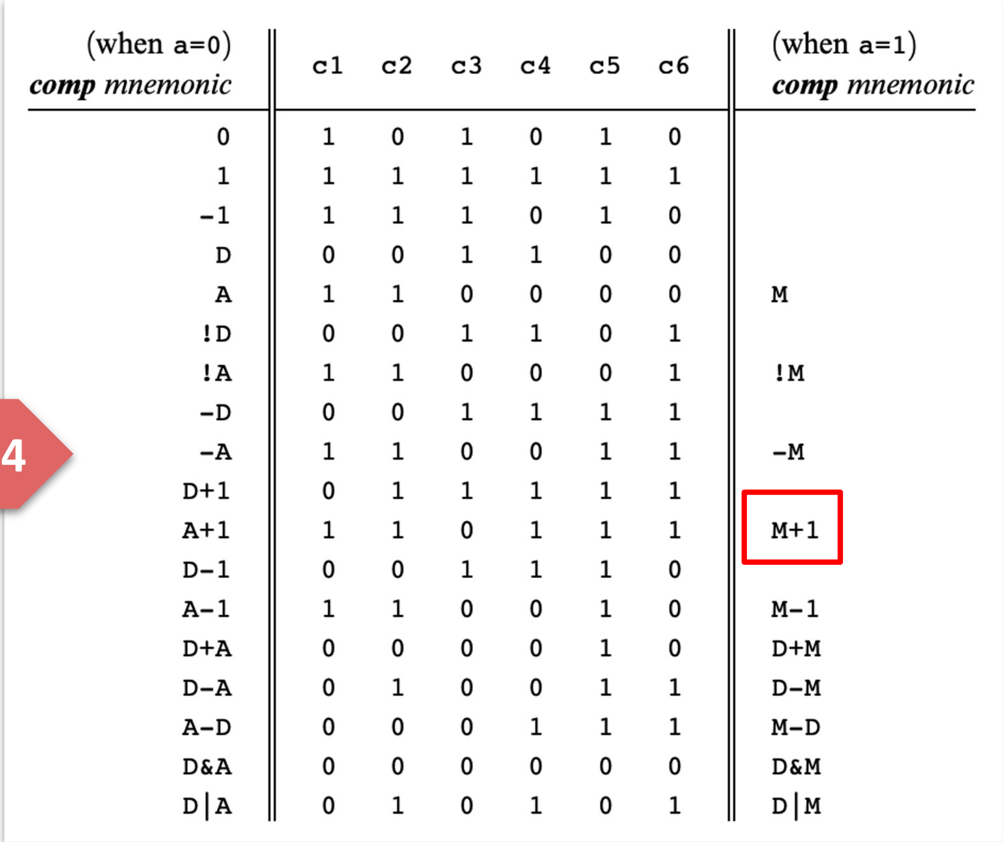
- Decodes the instruction bits into:
  - Op-code
  - ALU control bits
  - Destination load bits
  - Jump bits
- Routes these bits to their chip-part destinations
- The chip-parts (most notably, the ALU) execute the instruction

Hack: C-Instructions

- Symbolic: `dest = comp ; jump`
- Binary: `1 1 1 a c1 c2 c3 c4 c5 c6 d1 d2 d3 j1 j2 j3`

Comp:
ALU Operation (a bit chooses between A and M)

Chapter 4

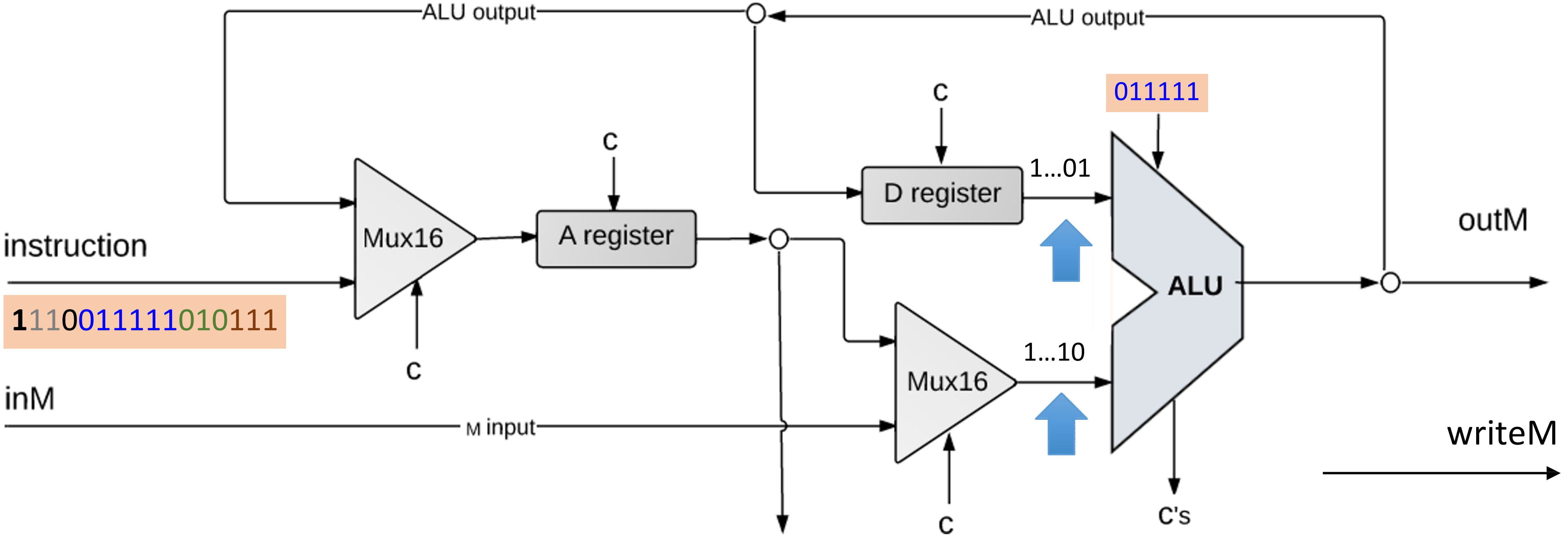


(when a=0) *comp mnemonic*	c1	c2	c3	c4	c5	c6	(when a=1) *comp mnemonic*
0	1	0	1	0	1	0	
1	1	1	1	1	1	1	
-1	1	1	1	0	1	0	
D	0	0	1	1	0	0	
A	1	1	0	0	0	0	M
!D	0	0	1	1	0	1	
!A	1	1	0	0	0	1	!M
-D	0	0	1	1	1	1	
-A	1	1	0	0	1	1	-M
D+1	0	1	1	1	1	1	
A+1	1	1	0	1	1	1	M+1
D-1	0	0	1	1	1	0	
A-1	1	1	0	0	1	0	M-1
D+A	0	0	0	0	1	0	D+M
D-A	0	1	0	0	1	1	D-M
A-D	0	0	0	1	1	1	M-D
D&A	0	0	0	0	0	0	D&M
D|A	0	1	0	1	0	1	D|M

Important: just pattern matching text!

Cannot have “**1+M**”

CPU Operation: Handling C-Instructions



<u>ALU data inputs:</u>

- Input 1: from the D-register
- Input 2: from either:
  - A-register, or
  - data memory

<u>ALU control inputs:</u>

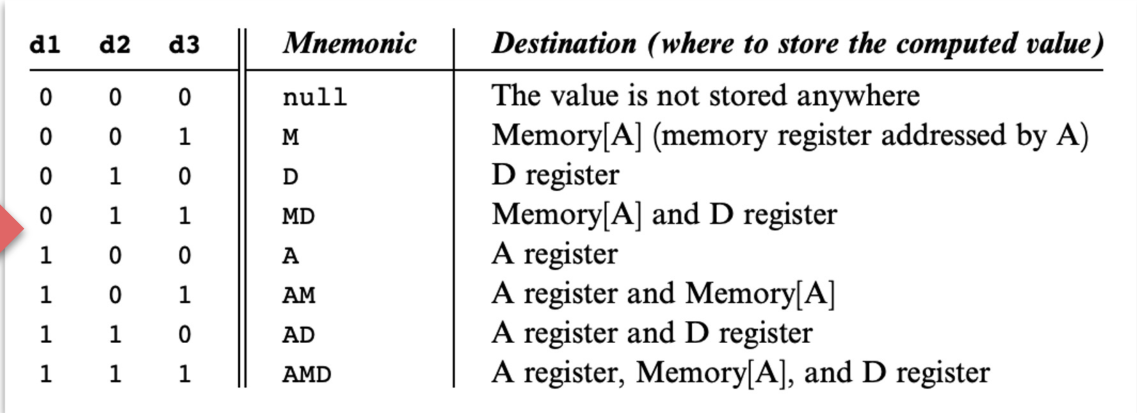
- Control bits (from the instruction)

Hack: C-Instructions

- Symbolic: `dest = comp ; jump`
- Binary: `1 1 1 a c1 c2 c3 c4 c5 c6 d1 d2 d3 j1 j2 j3`

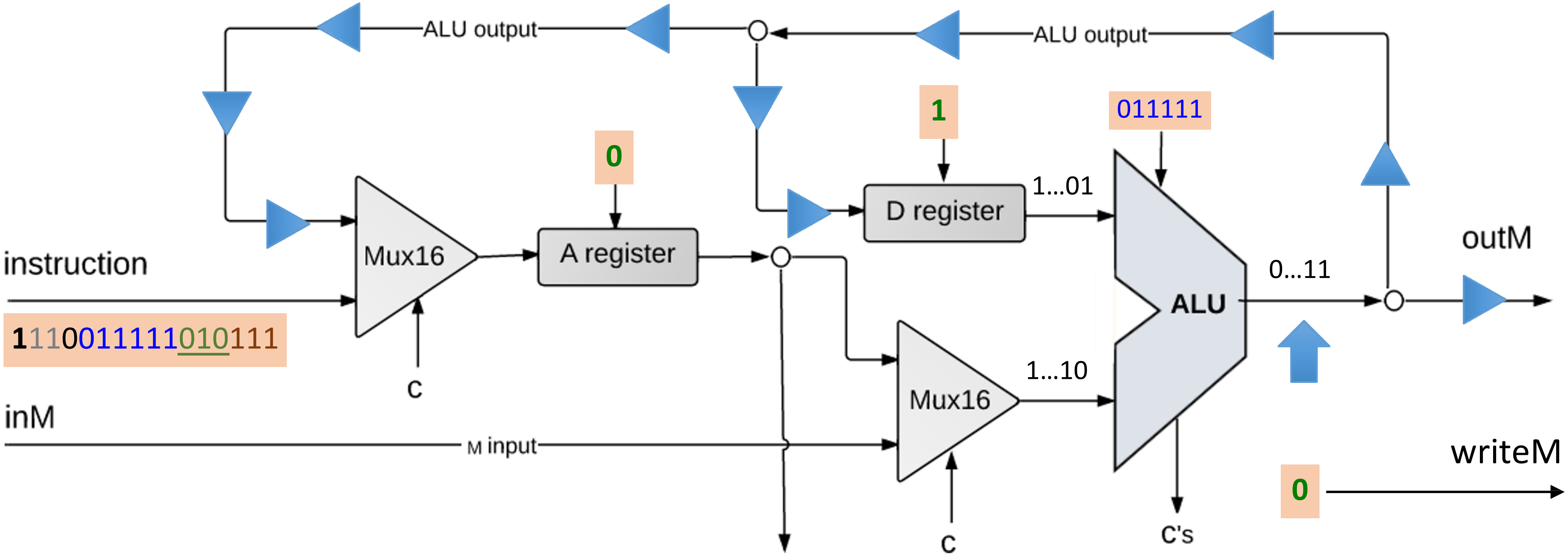
Dest:
Where to store result

Chapter 4



d1	d2	d3	*Mnemonic*	*Destination (where to store the computed value)*
0	0	0	null	The value is not stored anywhere
0	0	1	M	Memory[A] (memory register addressed by A)
0	1	0	D	D register
0	1	1	MD	Memory[A] and D register
1	0	0	A	A register
1	0	1	AM	A register and Memory[A]
1	1	0	AD	A register and D register
1	1	1	AMD	A register, Memory[A], and D register

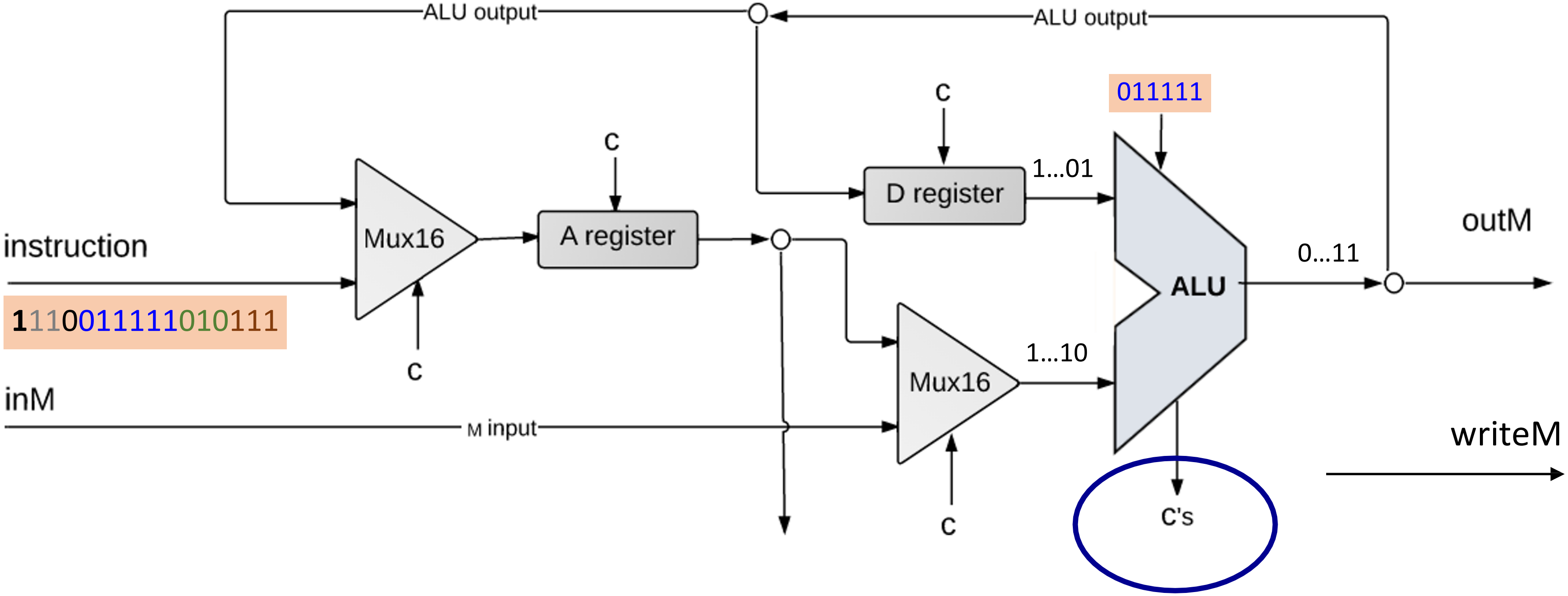
CPU Operation: Handling C-Instructions



ALU data output:

- Result of ALU calculation
- Fed simultaneously to: D-register, A-register, data memory
- Which destination *actually* commits to the ALU output is determined by the instruction's destination bits

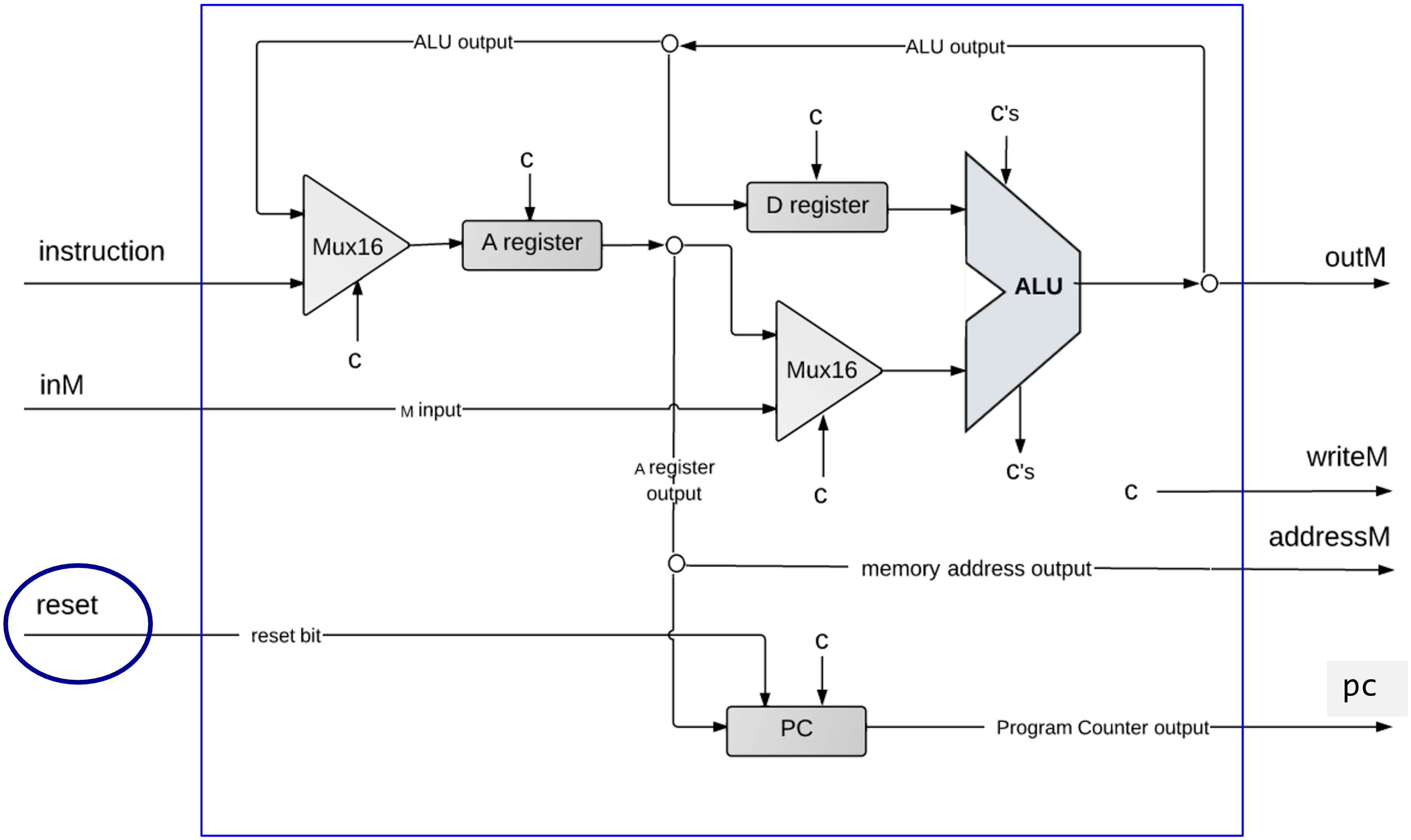
CPU Operation: Handling C-Instructions



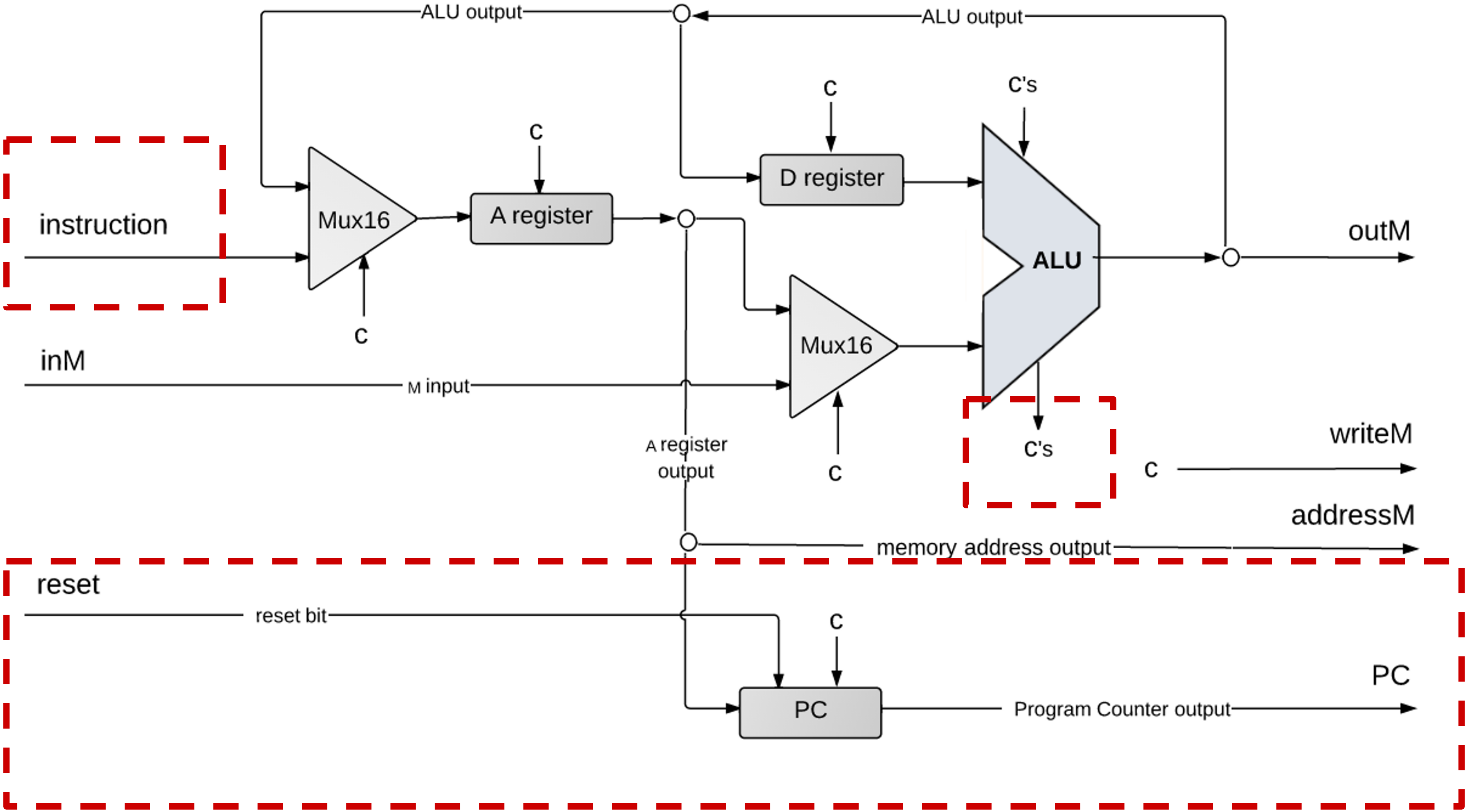
ALU control outputs:

- Is the output negative?
- Is the output zero?

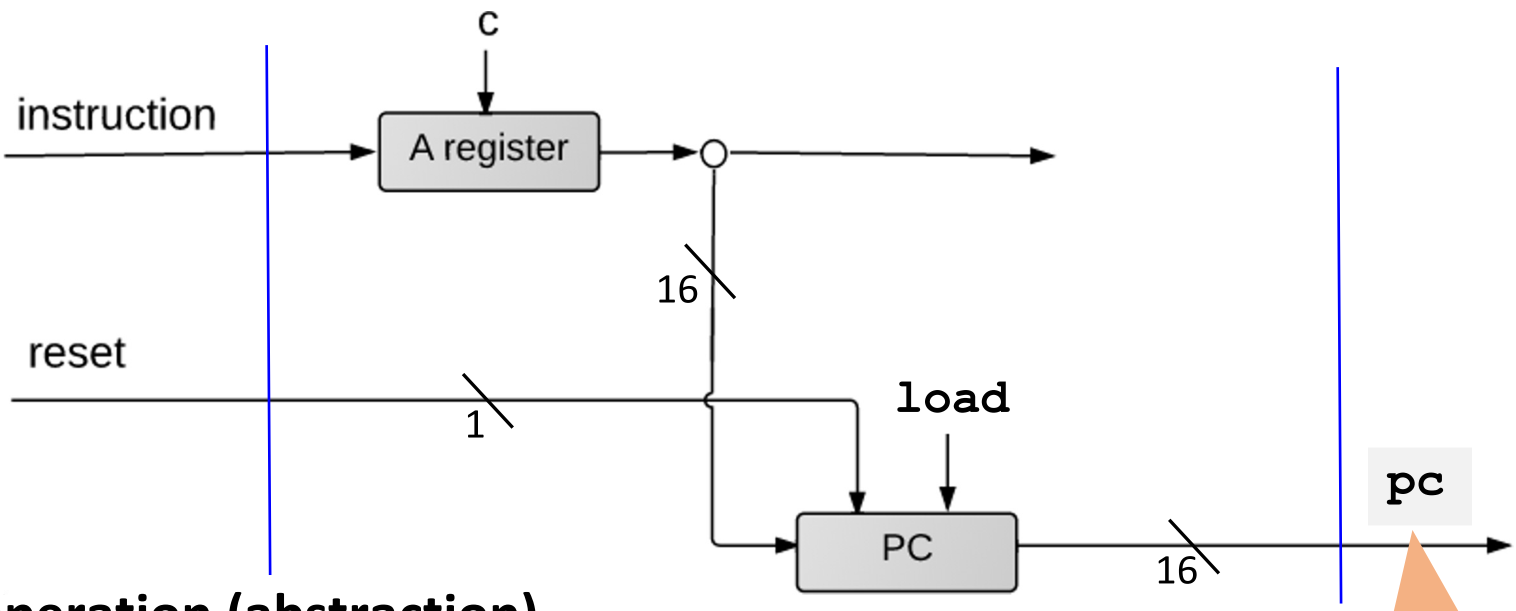
CPU Operation: Control



CPU Operation: Control



CPU Operation: Control



Address of next instruction

PC operation (abstraction)

Outputs the address of the next instruction:

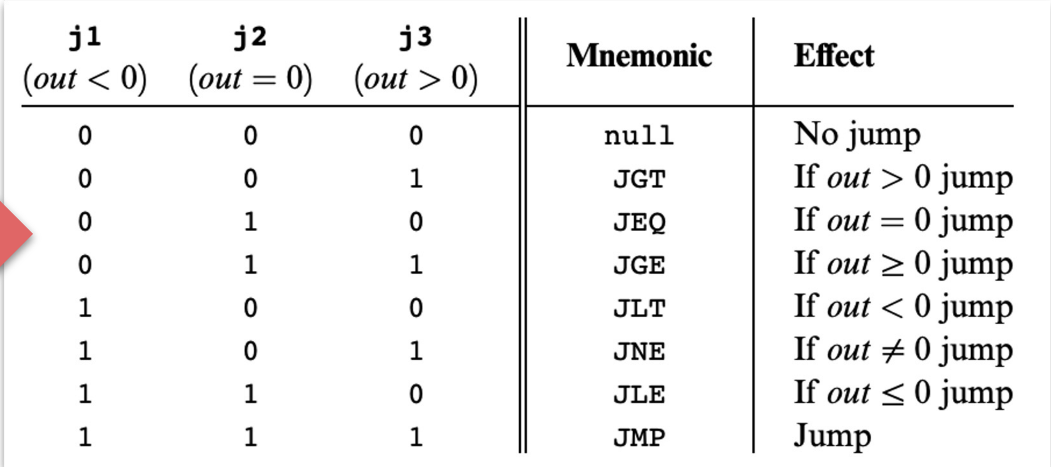
- Restart: PC = 0
- No jump: PC++
- Go to: PC = A
- Conditional go to: if (condition) PC = A
  else PC ++

Hack: C-Instructions

- Symbolic: `dest = comp ; jump`
- Binary: `1 1 1 a c1 c2 c3 c4 c5 c6 d1 d2 d3 j1 j2 j3`

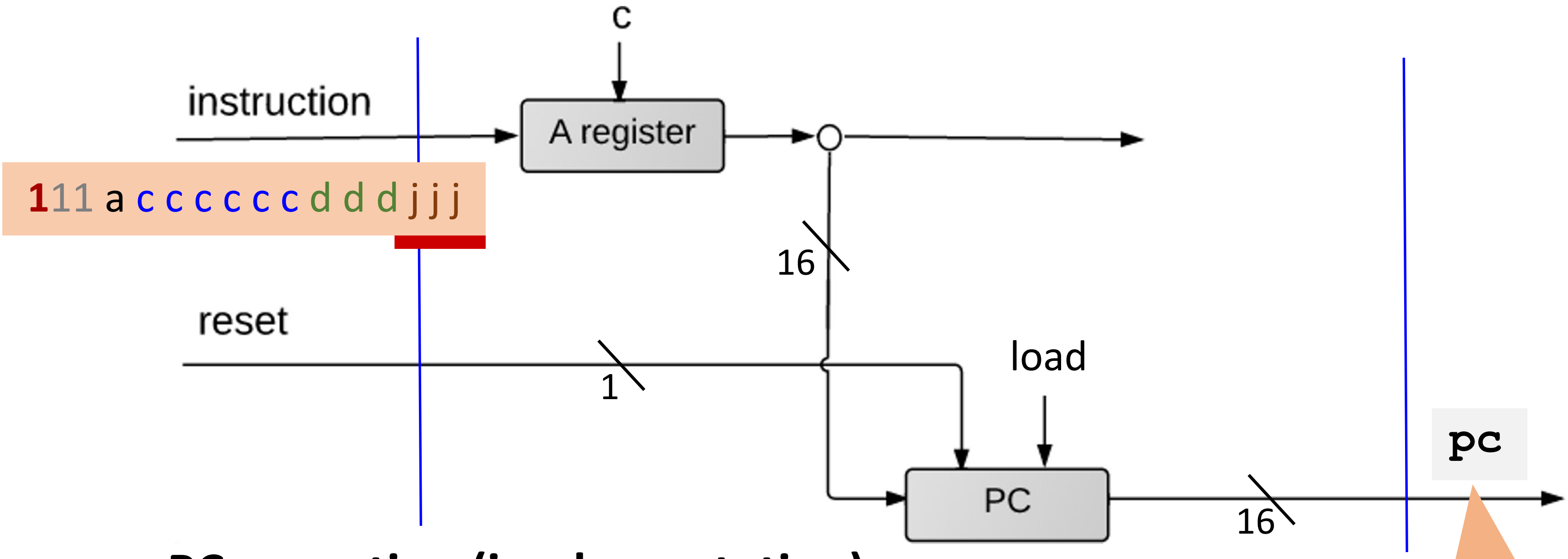
Jump: Condition for jumping

Chapter 4



j1 ($out < 0$)	j2 ($out = 0$)	j3 ($out > 0$)	Mnemonic	Effect
0	0	0	null	No jump
0	0	1	JGT	If *out* > 0 jump
0	1	0	JEQ	If *out* $= 0$ jump
0	1	1	JGE	If *out* ≥ 0 jump
1	0	0	JLT	If *out* < 0 jump
1	0	1	JNE	If *out* $\neq 0$ jump
1	1	0	JLE	If *out* ≤ 0 jump
1	1	1	JMP	Jump

CPU Operation: Control

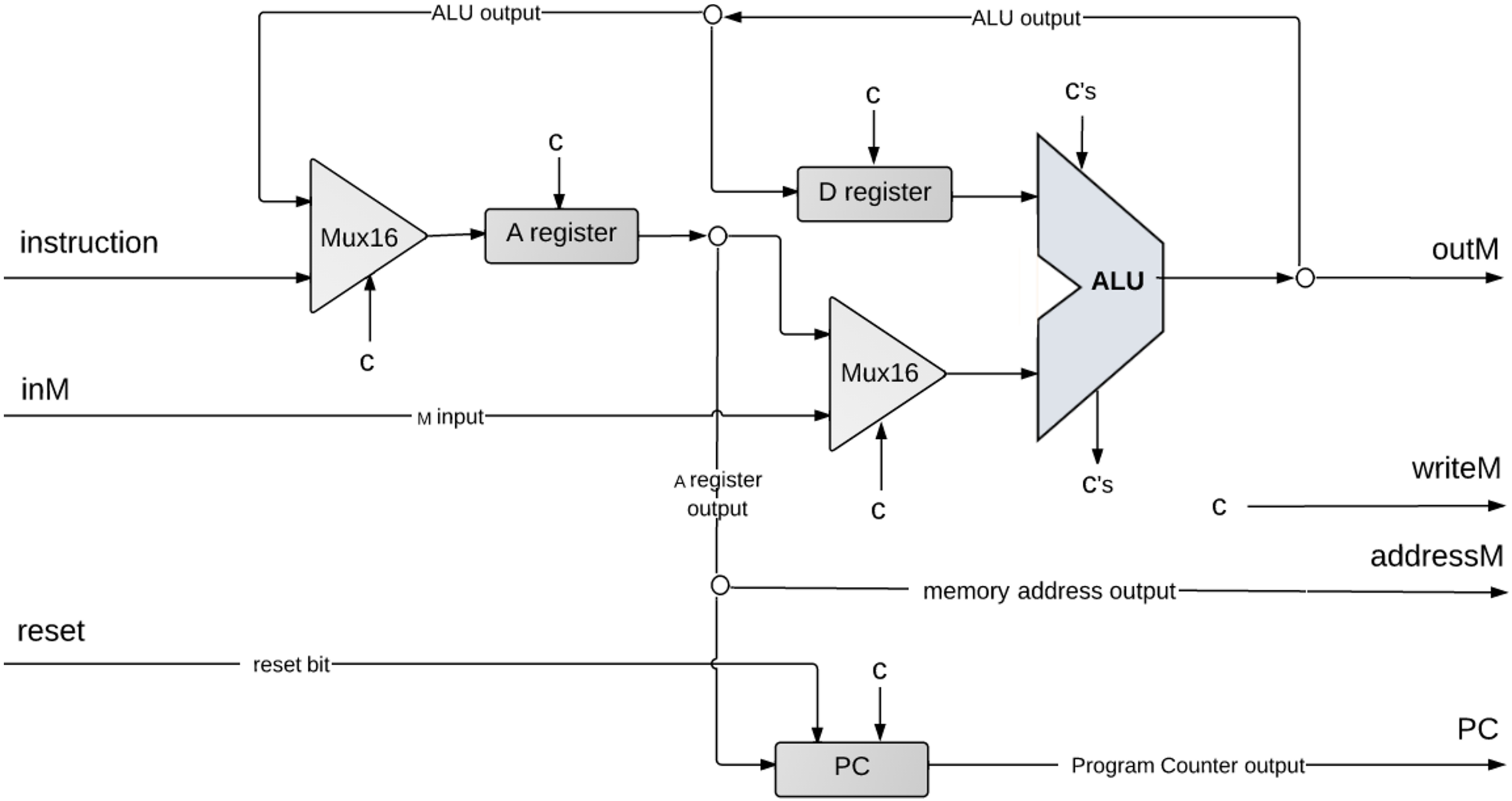


Address of next instruction

PC operation (implementation)

```
if (reset==1) PC = 0
else
    // In the course of handling the current instruction:
    load = f (jump bits, ALU control outputs)
    if (load == 1) PC = A    // jump
    else           PC++      // next instruction
```

Hack CPU Implementation: That's It!



Lecture Outline

- Building a Computer
  - Architecture, Fetch and Execute Cycle
- Hack CPU Interface
  - Implementation and Operations
- **Project 6 Overview**
  - **Mock Exam Problem and Project Tips**

Project 6: Overview

- Part I: Mock Exam Problem

- Part II: Building a Computer
  - **`LoadAReg.hdl`**, **`LoadDReg.hdl`** (Easier)
  - **`JumpLogic.hdl`** (Medium)
  - **`CPU.hdl`** (Harder)
  - **`Computer.hdl`** (Easier)

- Part III: Project 6 Reflection

Project 6, Part I: Mock Exam Problem

- Your group will meet for a 30-minute session to do one mock exam problem
  - Your group's mock exam problem will be emailed right before your session

- Your 30-minute session will include:
  - Set up: 5 minutes
  - Mock Exam Problem: 10 minutes
  - Debrief & Reflection: 15 minutes

- Part I task: Submit the completed mock exam problem and complete the reflection questions

Project 6, Part II Tips

- `CPU.hdl`: We provide an overview diagram, but there are details to fill in, especially control
  - Draw your own detailed diagram first
  - Handling jumps will require a lot of logic—sketch out the cases
  - Textbook chapter 4 and 5 helpful for Project 6

- Multi-Bit Buses: MSB to the left, LSB to the right
  - Important to keep in mind when taking apart the instruction

- Debugging: Consult `.out` and `.cmp` files to debug, then look at internal wires in simulator
  - See also the "Debugging tips" section of the specification

Lecture 10 Reminders

- **Project 5: Annotation, Machine Language, Computer Memory due tonight (4/26) at 11:59pm**
- **CSE 390B midterm next Friday (5/3) during lecture**
- Project 6 (Mock Exam Problem & Building a Computer) released today, due in two Fridays (5/10) at 11:59pm
- Eric has office hours after class in CSE2 153
  - Feel free to post your questions on the Ed board as well