CSE 390B, 2024 Winter

Building Academic Success Through Bottom-Up Computing

Exam Preparation & Building a Computer

Exam Preparation, Multiplication Implementation Exercise, Building a Computer, Hack CPU Interface

Lecture Outline

- **Exam Preparation**
  - **Study Strategies, Mock Exam Problem**
- Multiplication Implementation Exercise
  - Multiplying Two Numbers in Hack Assembly
- Building a Computer
  - Architecture, Fetch and Execute Cycle
- Hack CPU Interface
  - Implementation and Operations

Exams Preparation Discussion

- How do you usually prepare for your exams?
- What is one thing that is effective and ineffective about the way you study? Why?
- What are some effective exam preparation strategies that you would find most helpful?

Gearing Up For Exams

- Make a Study Plan
  - What key topics / concepts does your exam cover?
  - How might your study guides look different for specific classes?
  - What resources, materials, or people might you engage with?

- Create a Schedule
  - Avoid cramming
  - Office hours, review sessions, study groups
  - Reference your weekly time commitments & quarterly calendar

- Test Yourself
  - What are ways that can help address this?
  - Replicate exam-like environments

Project 6, Part I: Mock Exam Problem

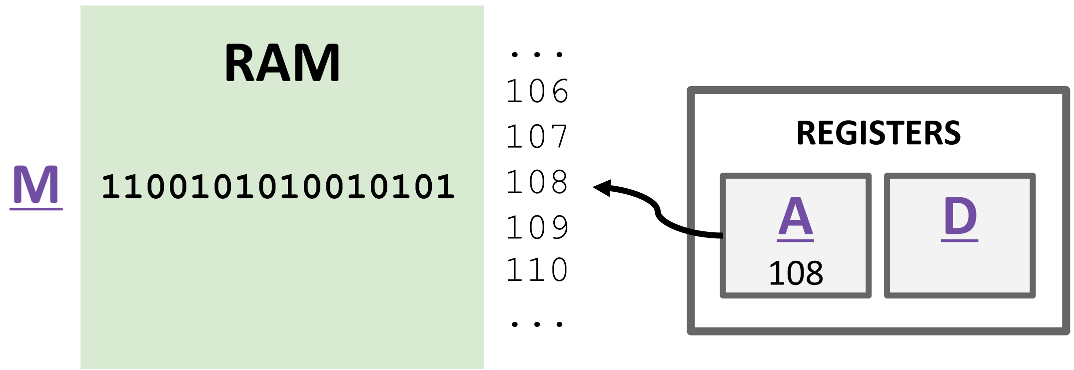
- Schedule a 30-minute session based on your group members' availability to complete one mock exam problem
- Determine how you will connect with each other and where your session will be located
- Mock exam problem groups posted on the Ed board
  - Please have one person from your group email Eric or respond on the Ed post with when your group will meet for the mock exam problem

Lecture Outline

- Exam Preparation
  - Study Strategies, Mock Exam Problem
- **Multiplication Implementation Exercise**
  - **Multiplying Two Numbers in Hack Assembly**
- Building a Computer
  - Architecture, Fetch and Execute Cycle
- Hack CPU Interface
  - Implementation and Operations

Hack: Registers

- D Register: For storing Data
- A Register: For storing data *and* Addressing memory
- M "Register": The 16-bit word in Memory currently being referenced by the address in A



Hack: A-Instructions

- Syntax: `@value`

- **`value`** can either be:
  - A decimal constant
  - A symbol referring to a constant

- Semantics:
  - Stores **`value`** in the A register

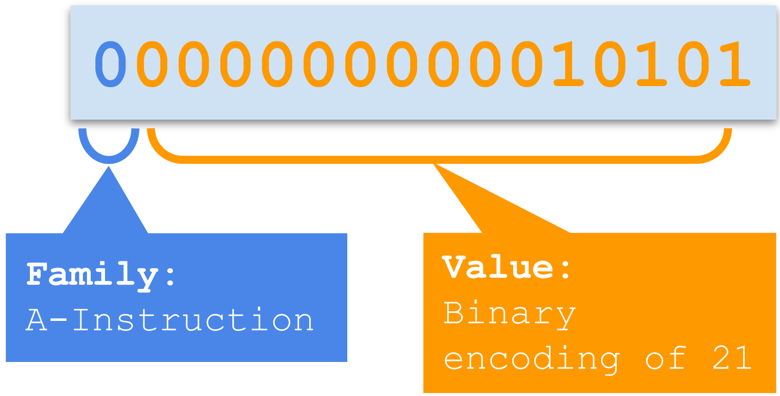
Hack: A-Instructions

- Symbolic Syntax

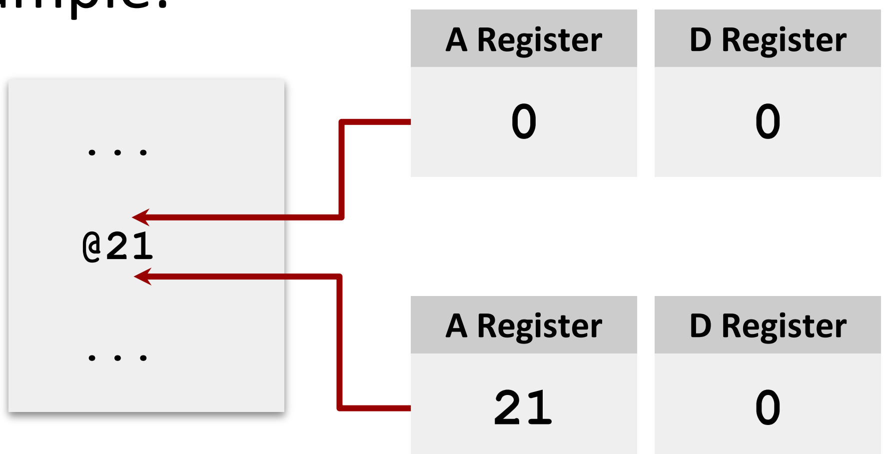
```
@value
```

  - Loads a value into the A register

- Binary Syntax



- Example:



Hack: Symbols

- Symbols are simply an <u>alias</u> for some address
  - Only in the symbolic code—don't turn into a binary instruction
  - Assembler converts use of that symbol to its value instead

- Example:

```
00      @3
01      D=0
      (LOOP)
02      @21
03      D=1
04      @LOOP

         ...
```

LOOP = 02

Assemble

```
00   0000000000000011
01   1110101010010000
02   0000000000010101
03   1110111111010000
04   0000000000000010

     ...
```

Hack: C-Instructions

- Syntax: `dest = comp ; jump` (**`dest`** and **`jump`** optional)
  - **`dest`** is a combination of destination registers:

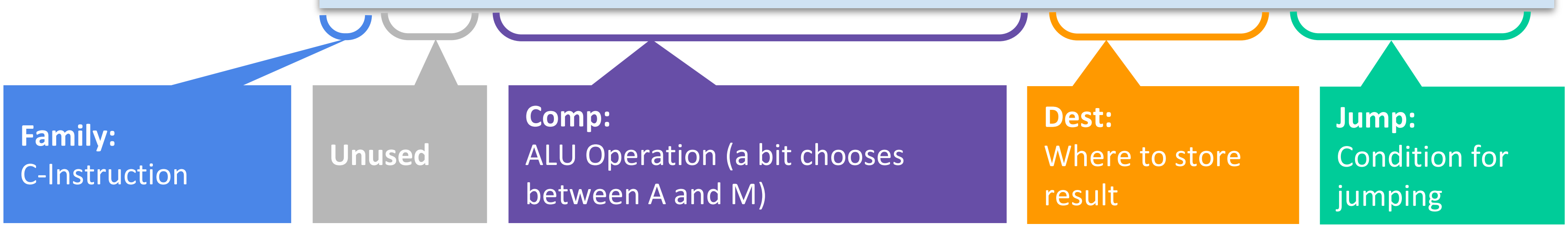
    `M, D, MD, A, AM, AD, AMD`
  - **`comp`** is a computation:

    `0, 1, -1, D, A, !D, !A, -D, -A, D+1, A+1, D-1, A-1, D+A, D-A, A-D, D&A, D|A, M, !M, -M, M+1, M-1, D+M, D-M, M-D, D&M, D|M`
  - **`jump`** is an unconditional or conditional jump:

    `JGT, JEQ, JGE, JLT, JNE, JLE, JMP`
- Semantics:
  - Computes value of **`comp`**
  - Stores results in **`dest`** (if specified)
  - If **`jump`** is specified and condition is true (by testing **`comp`** result), jump to instruction **`ROM[A]`**

Hack: C-Instructions

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



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 $out \geq 0$ jump
1	0	0	JLT	If $out < 0$ jump
1	0	1	JNE	If $out \neq 0$ jump
1	1	0	JLE	If $out \leq 0$ jump
1	1	1	JMP	Jump

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

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**"

Exercise: Implementing Multiplication

- Write a program that multiplies **`R0`** and **`R1`** and stores the result in **`R2`**
  - Remember we don't have a multiply operation
  - We will have to use add and loops to get the job done

- Roadmap
  - Start with pseudocode using if statements, loops, etc.
  - Remove conditionals and loops by using jumps in pseudocode
  - Convert pseudocode to assembly

Exercise: Implementing Multiplication

- Goal: Implement $\mathrm{R0} \times \mathrm{R1} = \mathrm{R2}$

Pseudocode	Hack Assembly

Exercise: Implementing Multiplication

- Goal: Implement $\mathrm{R0} \times \mathrm{R1} = \mathrm{R2}$

Pseudocode	Hack Assembly
- Approach: add **`R0`** to the result **`R1`** times	

Exercise: Implementing Multiplication

- Goal: Implement $R0 \times R1 = R2$

Pseudocode	Hack Assembly

Approach: add **R0** to the result **R1** times
 	
`R2 = 0`
`while (R1 > 0) {`
`    R2 = R0 + R2`
`    R1 = R1 - 1`
`}`	

Exercise: Implementing Multiplication

- Remove loops from pseudocode
- Use labels to notate important sections of the code

```
R2 = 0
while (R1 > 0) {
    R2 = R0 + R2
    R1 = R1 - 1
}
```

→

- Attempt 1: What happens when **R1** is 0? What should happen?

```
START:
    R2 = 0
LOOP:
    R2 = R0 + R2
    R1 = R1 - 1
    IF R1 > 0 JMP LOOP
END:
    INFINITE LOOP
```

Exercise: Implementing Multiplication

- Remove loops from pseudocode
- Use labels to notate important sections of the code

```
R2 = 0
while (R1 > 0) {
    R2 = R0 + R2
    R1 = R1 - 1
}
```

- Attempt 1: What happens when **R1** is 0? What should happen?

```
START:
  R2 = 0
LOOP:
    IF R1 <= 0
      JMP to END
    R2 = R0 + R2
    R1 = R1 - 1
    JMP LOOP
END:
    INFINITE LOOP
```

Exercise: Implementing Multiplication

- Convert to Hack Assembly

```
START:
  R2 = 0
LOOP:
    IF R1 <= 0
        JMP to END
    R2 = R0 + R2
    R1 = R1 - 1
    JMP LOOP
END:
    INFINITE LOOP
```

Exercise: Implementing Multiplication

- Convert to Hack Assembly

```
START:
  R2 = 0
LOOP:
    IF R1 <= 0
        JMP to END
    R2 = R0 + R2
    R1 = R1 - 1
    JMP LOOP
END:
    INFINITE LOOP
```

→

```
(START)
    @R2
    M = 0
(LOOP)
(END)
```

Exercise: Implementing Multiplication

- Convert to Hack Assembly

```
START:
  R2 = 0
LOOP:
    IF R1 <= 0
        JMP to END
    R2 = R0 + R2
    R1 = R1 - 1
    JMP LOOP
END:
    INFINITE LOOP
```

→

```
(START)
    @R2
    M = 0
(LOOP)
    @R1
    D = M
    @END
    D; JLE
(END)
```

Exercise: Implementing Multiplication

- Convert to Hack Assembly

```
START:
  R2 = 0
LOOP:
    IF R1 <= 0
        JMP to END
    R2 = R0 + R2
    R1 = R1 - 1
    JMP LOOP
END:
    INFINITE LOOP
```

→

```
(START)
    @R2
    M = 0
(LOOP)
    @R1
    D = M
    @END
    D; JLE
    @R0
    D = M
    @R2
    M = M + D
(END)
```

Exercise: Implementing Multiplication

- Convert to Hack Assembly

```
START:
  R2 = 0
LOOP:
    IF R1 <= 0
        JMP to END
    R2 = R0 + R2
    R1 = R1 - 1
    JMP LOOP
END:
    INFINITE LOOP
```

→

```
(START)
    @R2
    M = 0
(LOOP)
    @R1
    D = M
    @END
    D; JLE
    @R0
    D = M
    @R2
    M = M + D
    @R1
    M = M - 1
    @LOOP
    0; JMP
(END)
```

Exercise: Implementing Multiplication

- Convert to Hack Assembly

```
START:
   R2 = 0
LOOP:
      IF R1 <= 0
         JMP to END
      R2 = R0 + R2
      R1 = R1 - 1
      JMP LOOP
END:
      INFINITE LOOP
```

→

```
(START)
    @R2
    M = 0
(LOOP)
    @R1
    D = M
    @END
    D; JLE
    @R0
    D = M
    @R2
    M = M + D
    @R1
    M = M - 1
    @LOOP
    0; JMP
(END)
    @END
    0; JMP
```

Lecture Outline

- Exam Preparation
  - Study Strategies, Mock Exam Problem
- Multiplication Implementation Exercise
  - Multiplying Two Numbers in Hack Assembly
- **Building a Computer**
  - **Architecture, Fetch and Execute Cycle**
- Hack CPU Interface
  - Implementation and Operations

Building a Computer

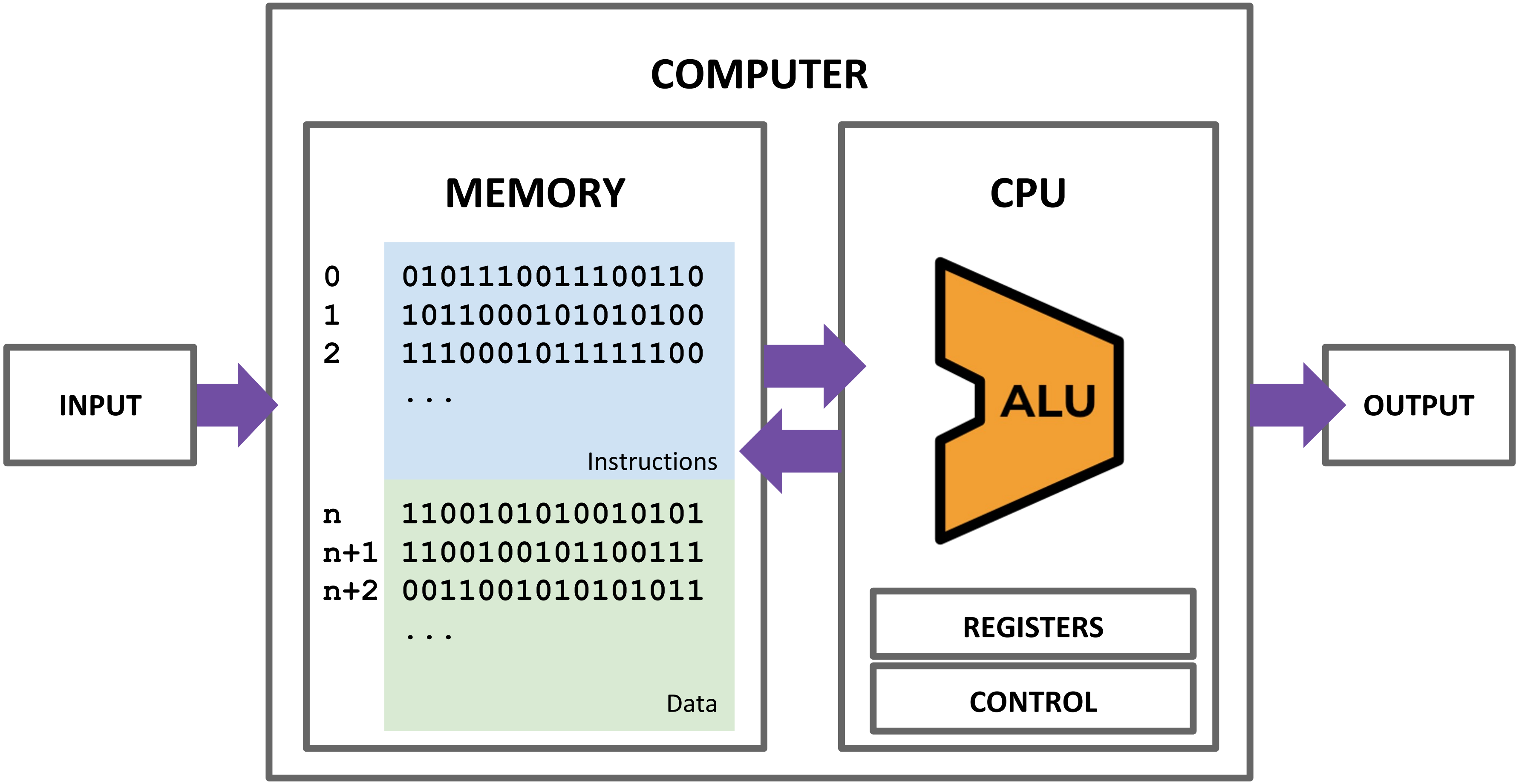
- All your hardware efforts are about to pay off!

- Perspective: **BUILDING A COMPUTER**

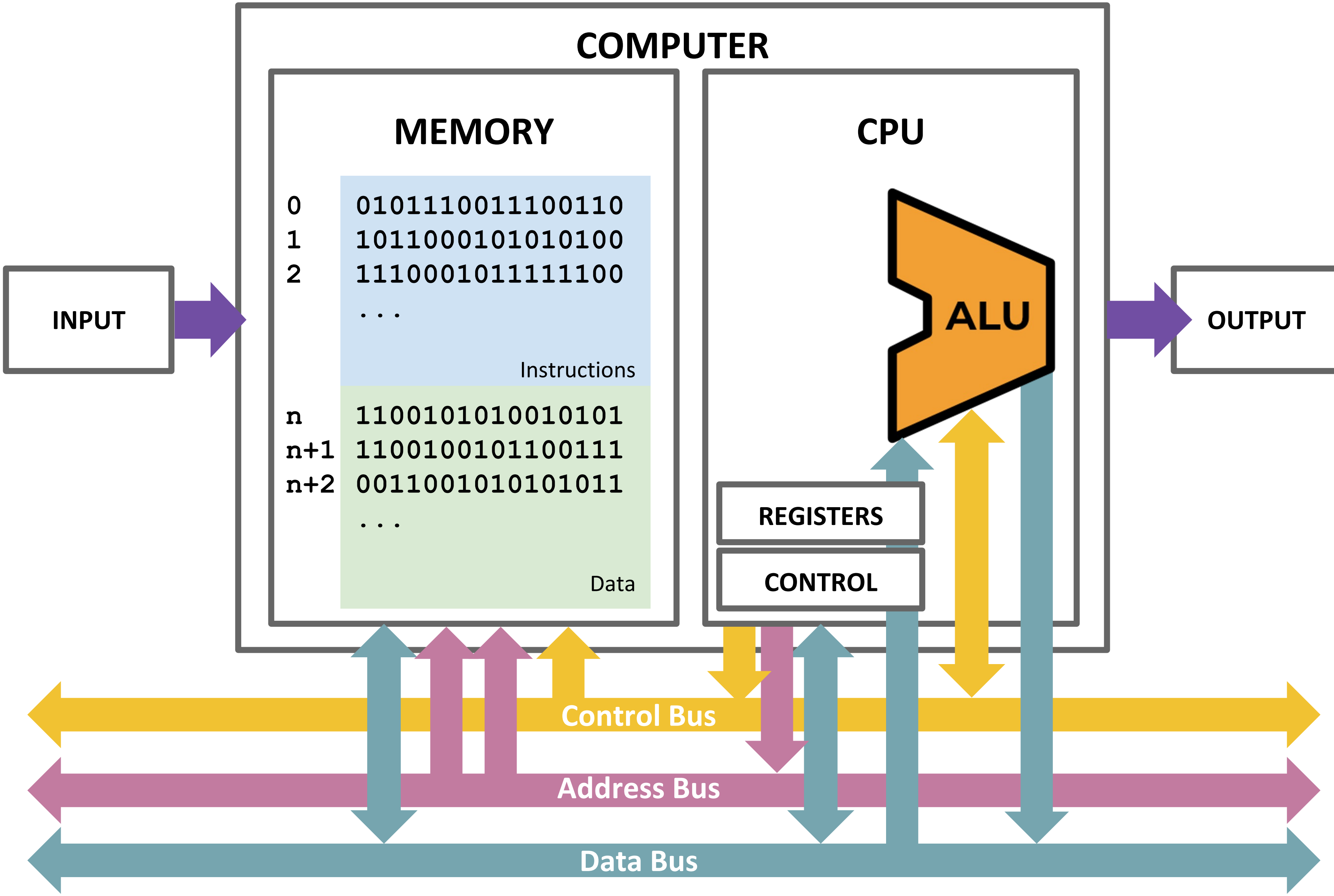
- In Project 6, you will build **`Computer.hdl`**, the final, top-level chip in this course
  - For all intents and purposes, a real computer
  - Simplified, but organization very similar to your laptop

- Project 7 onward, we will write software to make it useful

Von Neumann Architecture



Connecting the Computer: Buses

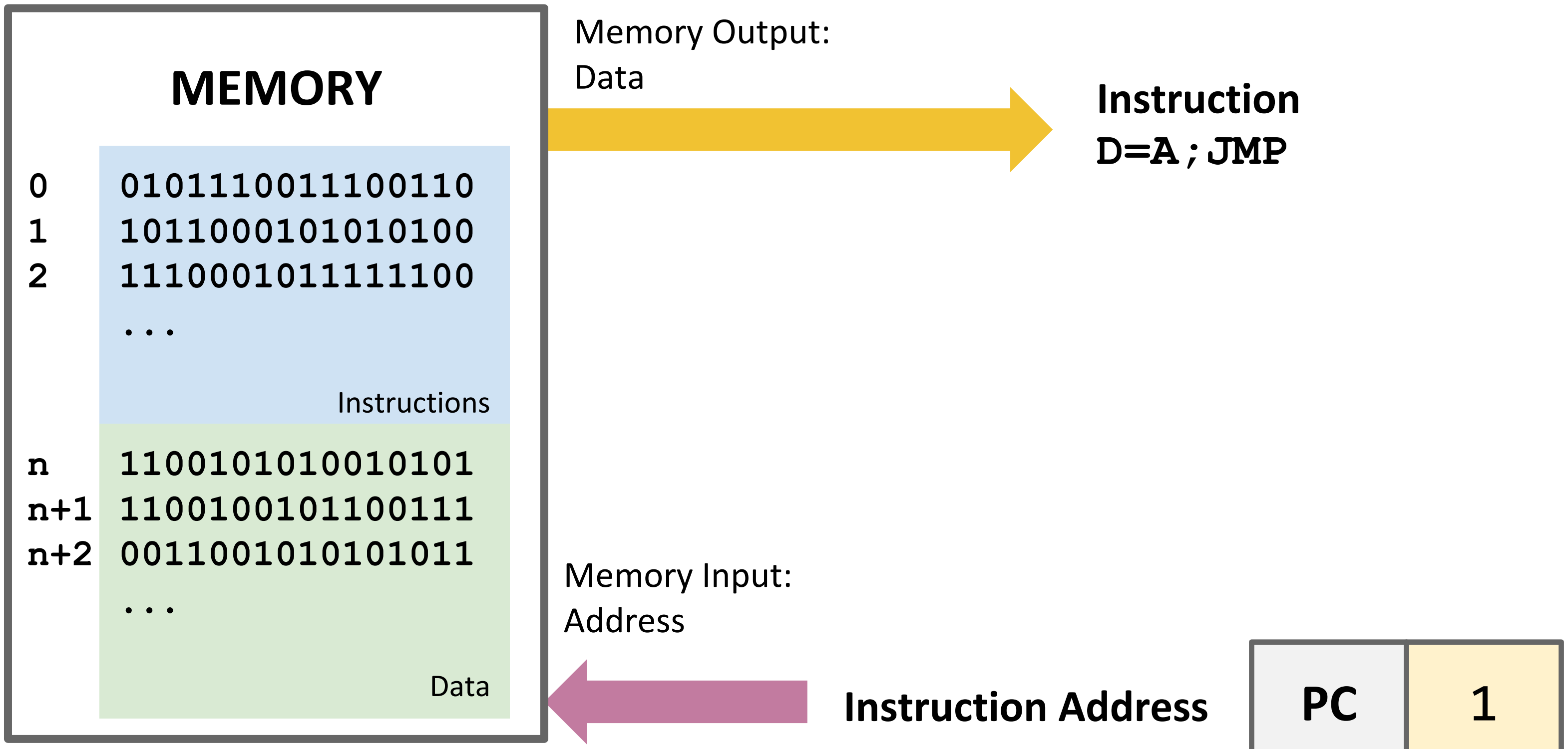


Basic CPU Loop

- 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

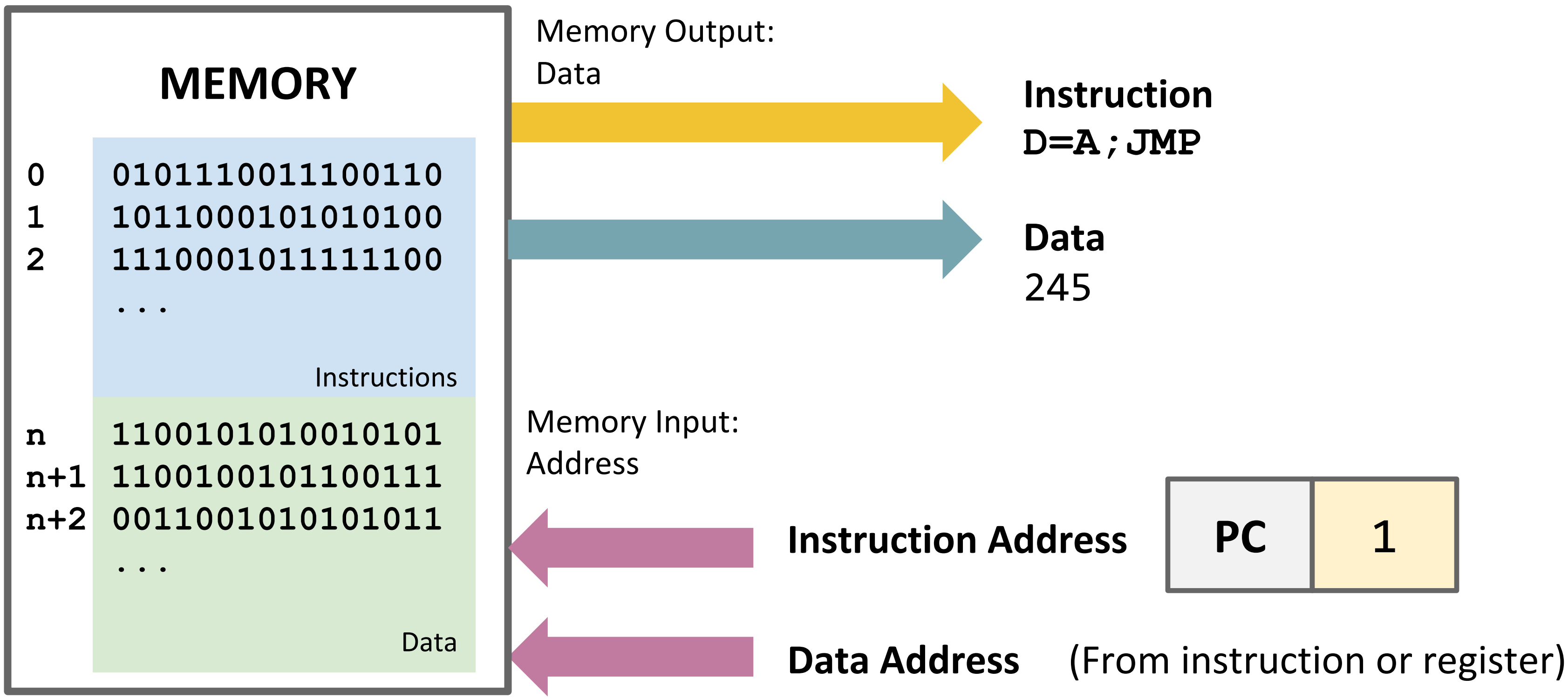


Executing

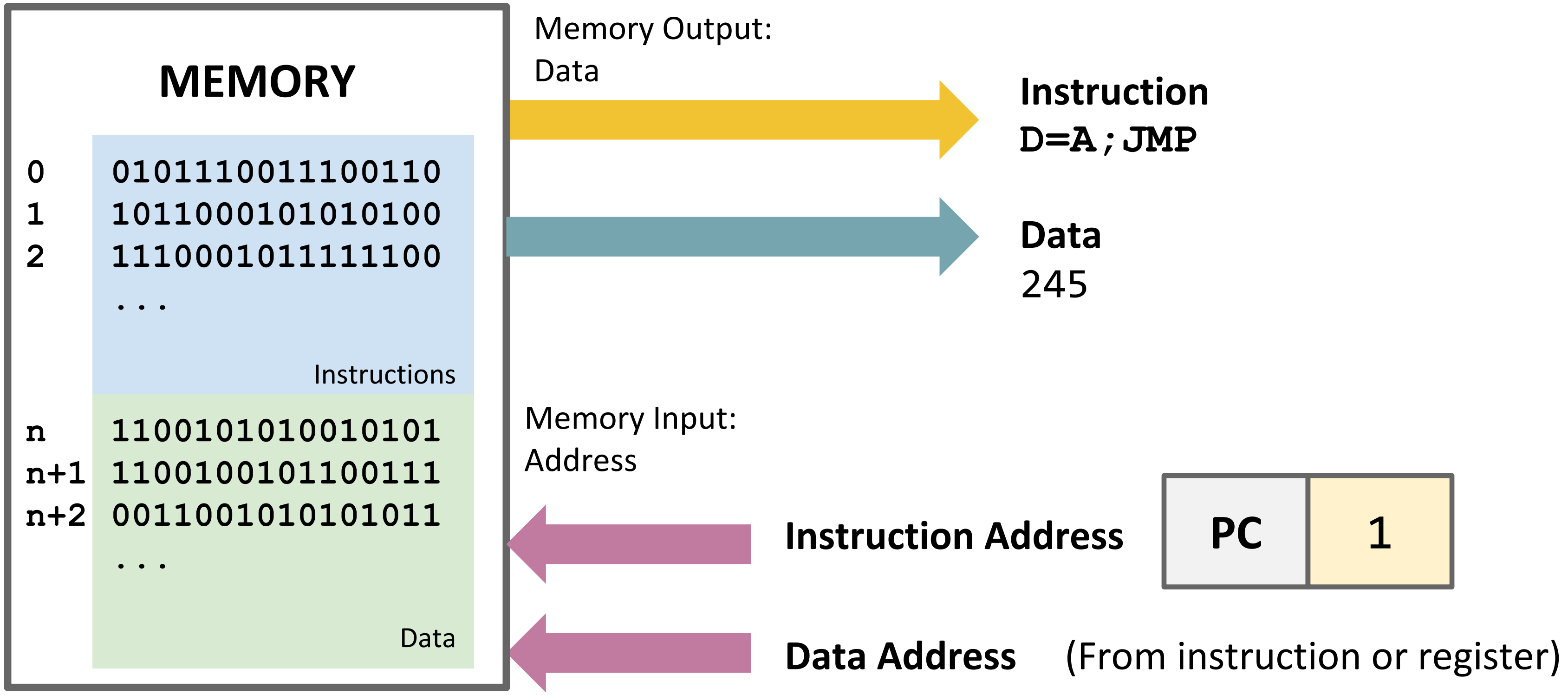
- The instruction bits describe exactly “what to do”
  - A-instruction or C-instruction?
  - Which operation for the ALU?
  - What memory address to read? To write?
  - If I should jump after this instruction, and where?

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

Combining Fetch & Execute

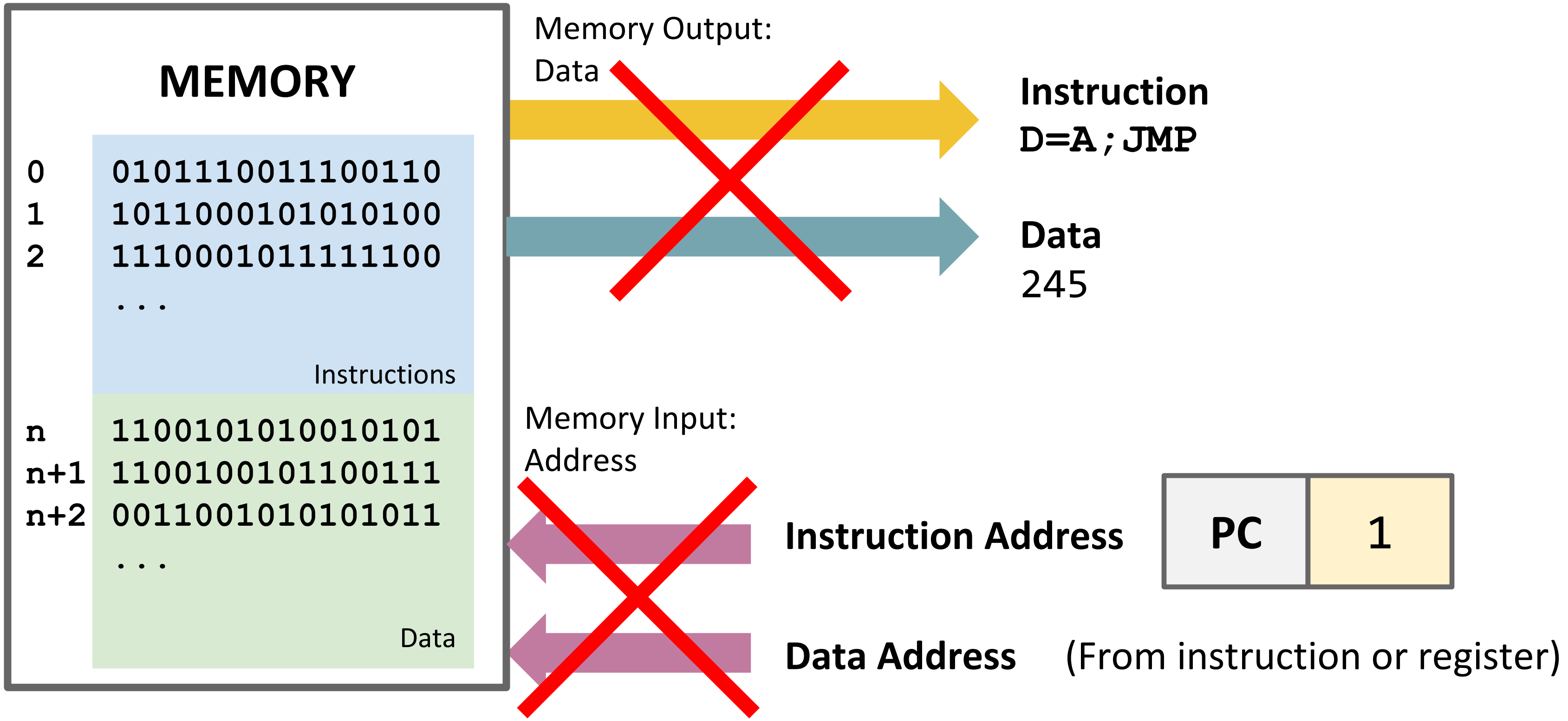


Combining Fetch & Execute



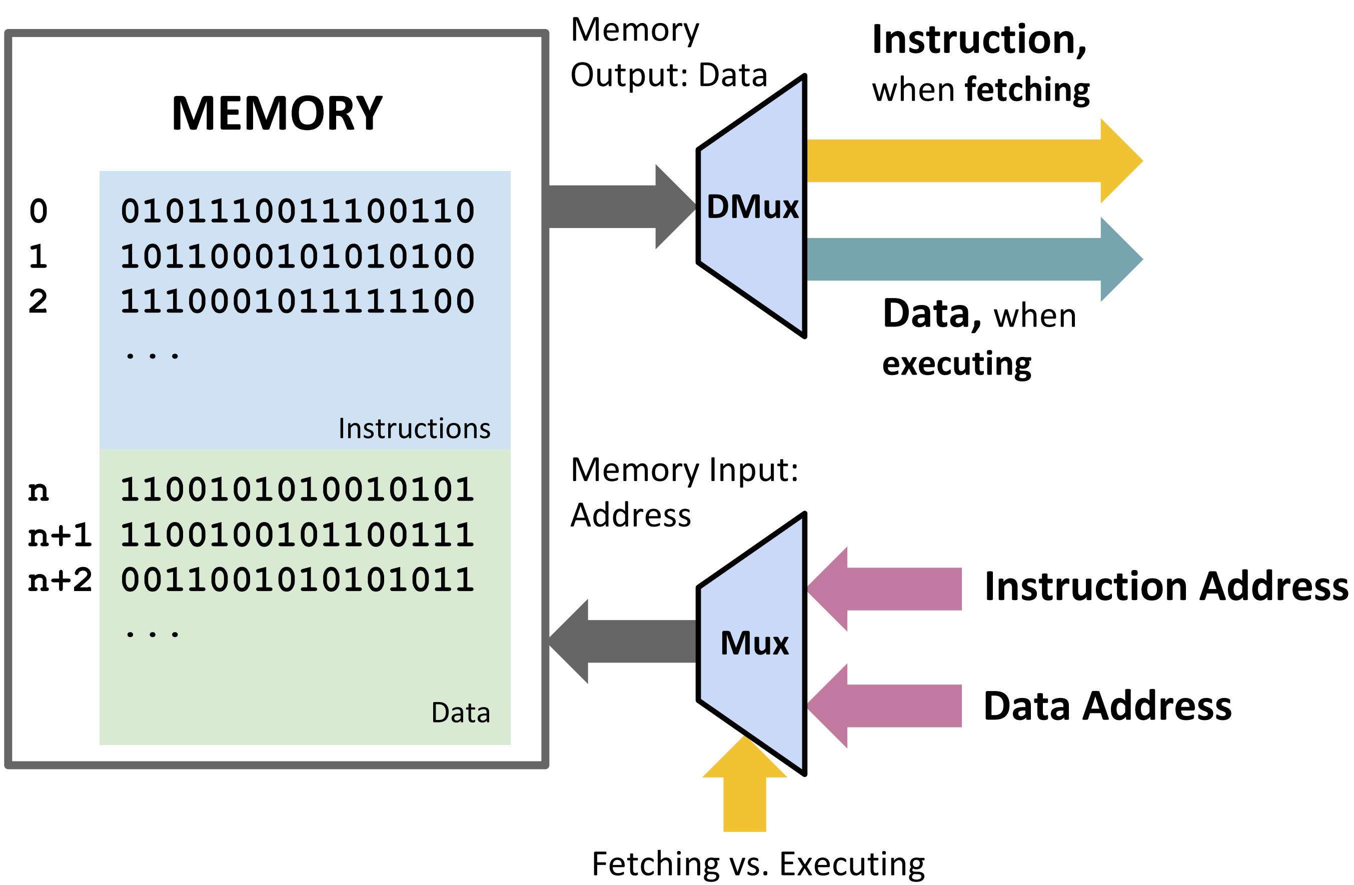
- Could we implement with **`RAM16K.hdl`**?
  - (Hint: Think about the I/O of RAM)

Combining Fetch & Execute



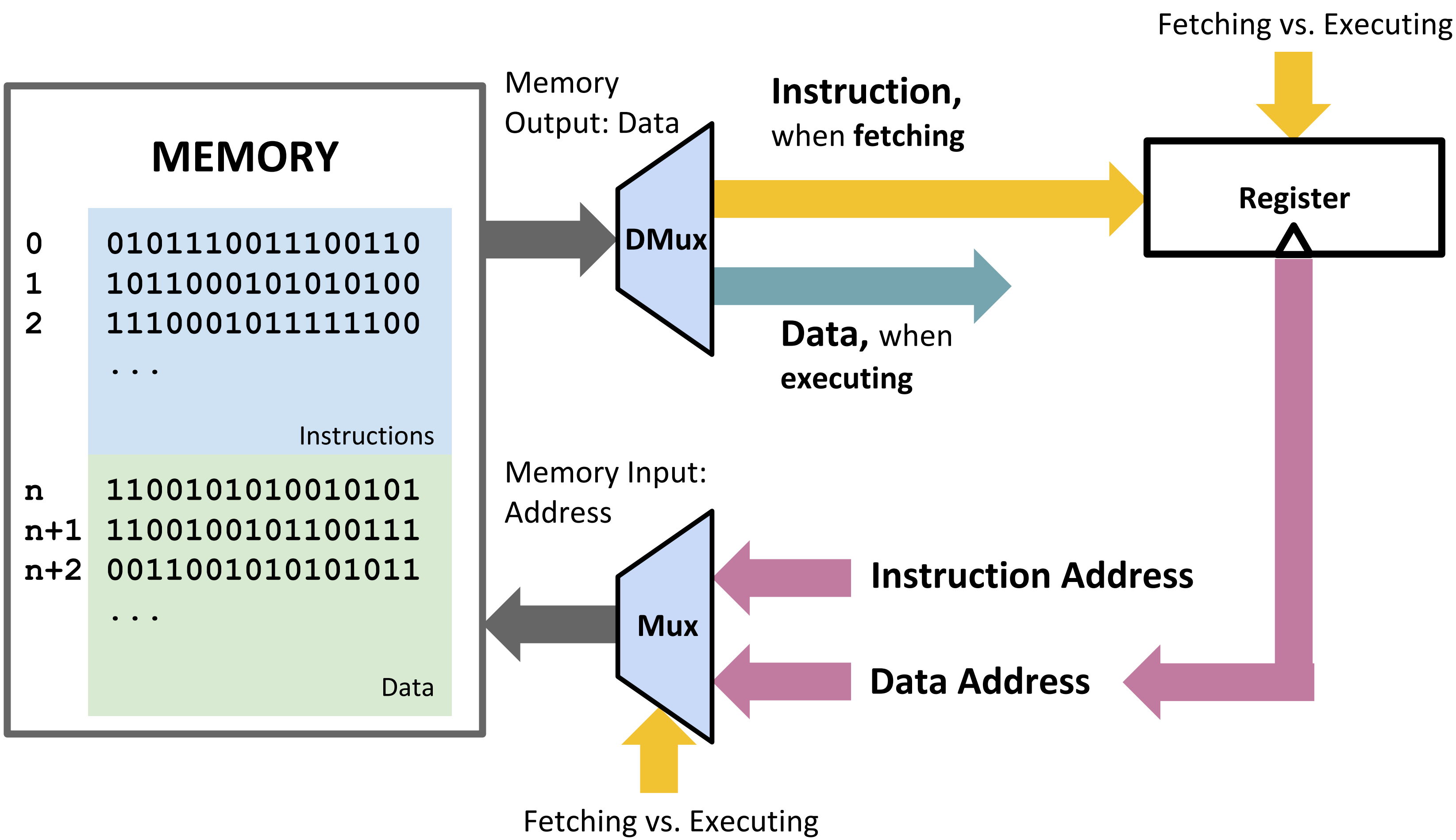
- Could we implement with **`RAM16K.hdl`**?
  - **No!** Our memory chips only have one input and one output

Solution 1: Handling Single Input / Output



- Can use multiplexing to share a single input or output

Solution 1: Fetching / Executing Separately



- 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

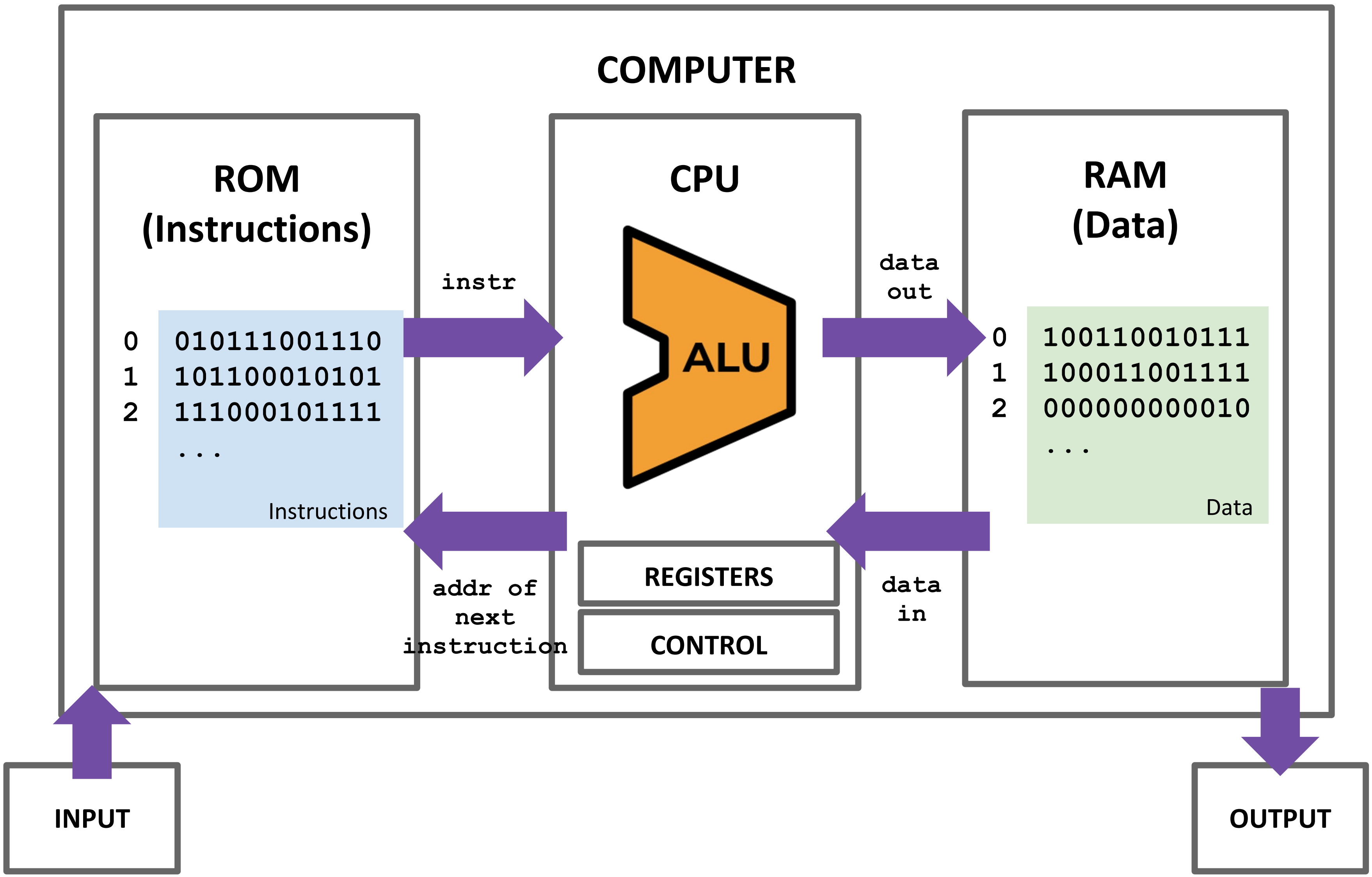
- Pros:
  - Simpler to implement

- Cons:
  - Fixed size of each partition, rather than flexible storage
  - Two chips → redundant circuitry

Lecture Outline

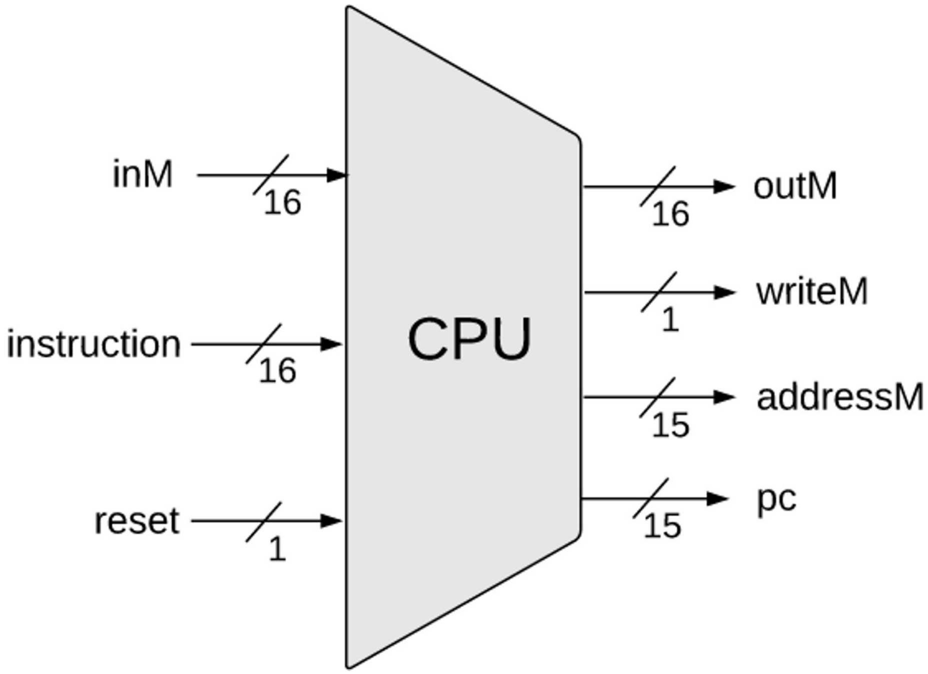
- Exam Preparation
  - Study Strategies, Mock Exam Problem
- Multiplication Implementation Exercise
  - Multiplying Two Numbers in Hack Assembly
- Building a Computer
  - Architecture, Fetch and Execute Cycle
- **Hack CPU Interface**
  - **Implementation and Operations**

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

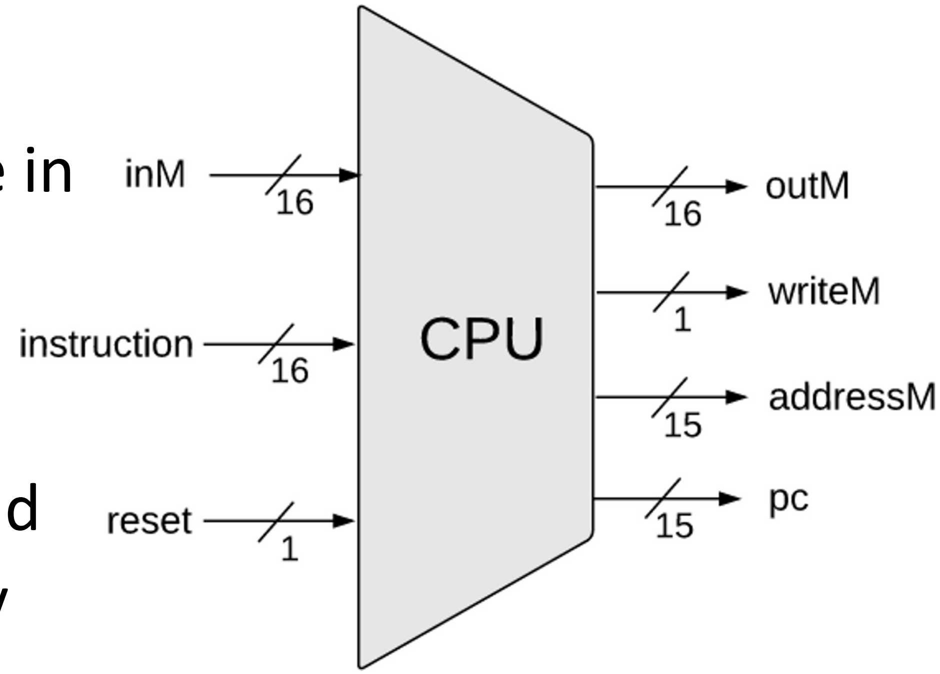
Hack CPU Interface Inputs

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

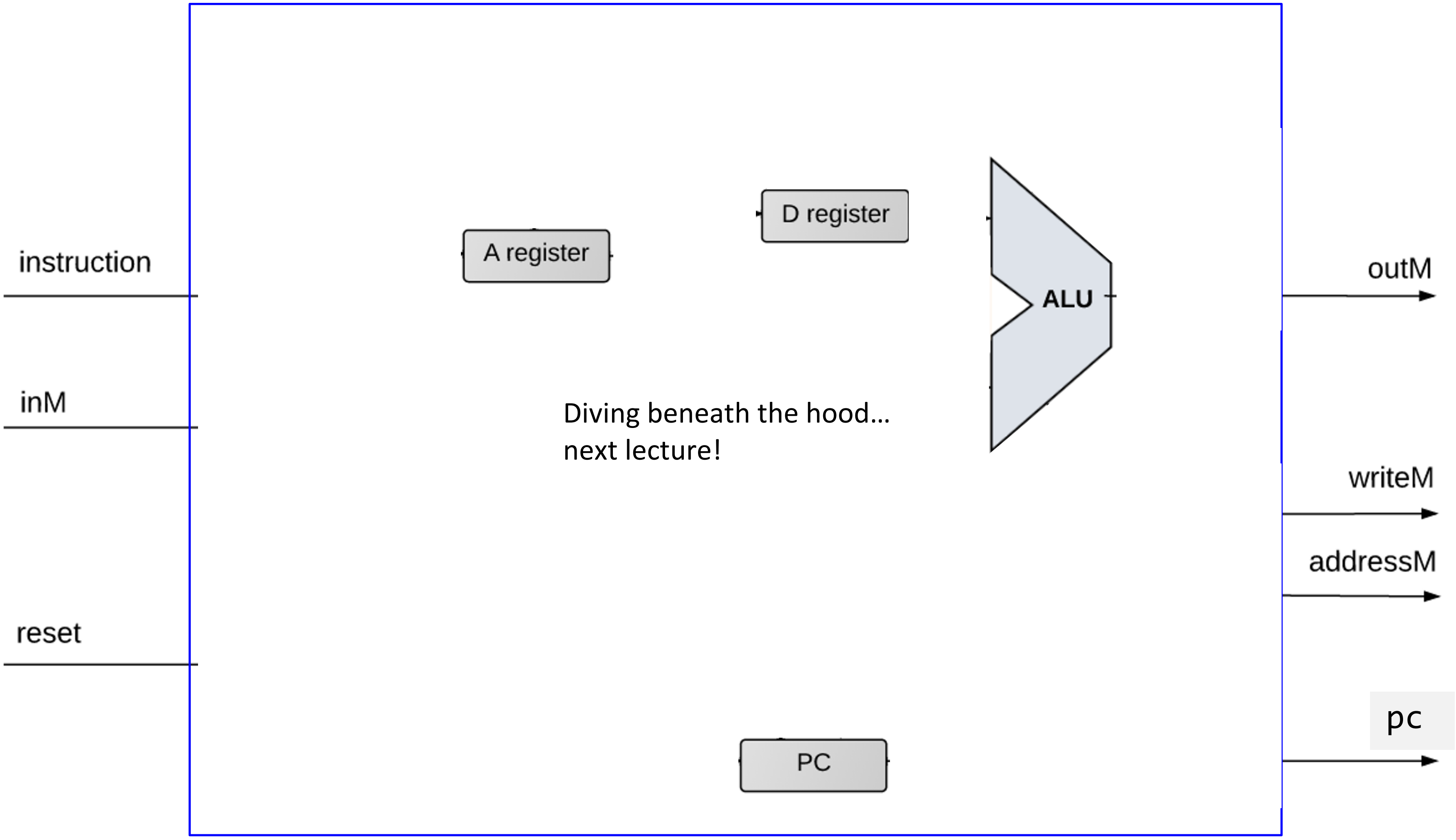


Hack CPU Interface Outputs

- **`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



Lecture 9 Reminders

- **Project 5 due this Friday (2/2) at 11:59pm**

- **Midterm exam coming up on 2/9 during lecture time**

- Amy has office hours tomorrow at 1:30pm in CSE2 151
  - Feel free to post your questions on the Ed board as well