CSE 390B, Spring 2023

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

- What are some exam preparation strategies that you have found helpful?
- What is one thing that is effective and ineffective about the way you study? Why?
- How might you implement some of these effective strategies for change your exam preparation strategy for this quarter?

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 you can test yourself?
  - 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:
  - Please have one person from your group email Eric when you will meet for the mock exam problem by **4/27**

Group Number	Group Members
Group 1	Guadalupe, Rodas, Vicky
Group 2	Delano, Richard
Group 3	Tai, Yohannes, David
Group 4	Nathaniel, Simon
Group 5	Benito, Ty
Group 6	Deya, Elainie
Group 7	Alex, Matthew, Said
Group 8	Kris, Binh

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

Review: What is Binary?

- A **base-n** number system is a system of number representation with n **symbols**

- Decimal system is a base-10 number system
  - Base-10 symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
  - Increase a number by moving to the next greatest symbol
  - Add another digit when we run out of symbols

- Binary is a base-2 number system
  - Often prefixed with 0b (e.g., 0b1101, 0b10)
  - Base-2 symbols: 0, 1

Hexadecimal

- Base-16 number system
  - Symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
- Commonly used for referring to memory addresses
  - Simple to convert between binary and hexadecimal
  - Hexadecimal uses fewer digits to represent a value than binary
- Uses the prefix 0x to indicate a number is written in hexadecimal
  - 32 is decimal, 0x32 is hexadecimal

Number Representation Comparison

Decimal	Hexadecimal	Binary
0	0x0	0b0000
1	0x1	0b0001
2	0x2	0b0010
3	0x3	0b0011
4	0x4	0b0100
5	0x5	0b0101
6	0x6	0b0110
7	0x7	0b0111
8	0x8	0b1000
9	0x9	0b1001
10	0xA	0b1010
11	0xB	0b1011
12	0xC	0b1100
13	0xD	0b1101
14	0xE	0b1110
15	0xF	0b1111

Binary and Hexadecimal Conversion

- One-to-one correspondence between binary and hexadecimal
- To convert from binary to hexadecimal, swap out binary bits digits for the corresponding hexadecimal digit (or vice versa)
- Example: 0x3A is 0b0011_1010
  - 0x3 == 0b0011
  - 0xA == 0b1010

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
}
```

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

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

Exercise: Implementing Multiplication

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

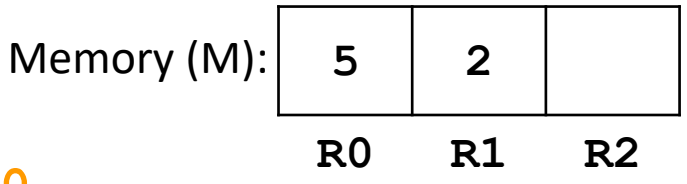
```
R2 = 0
while (R1 > 0) {
    R2 = R0 + R2
    R1 = R1 - 1
}
```

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

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

Exercise: Implementing Multiplication

- Convert to Hack Assembly

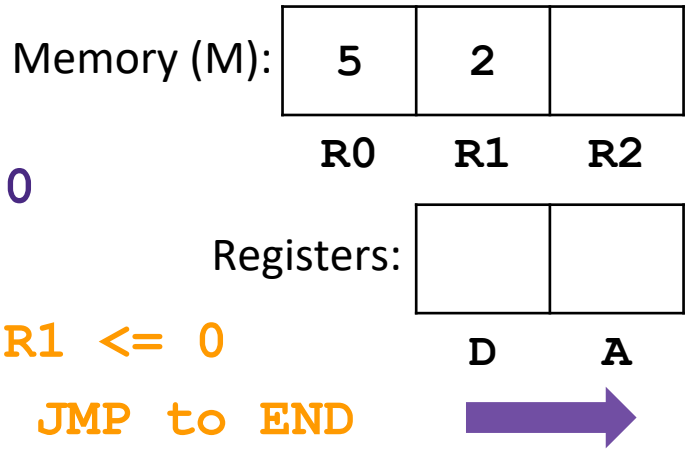



```
START:
   R2 = 0
LOOP:
      IF R1 <= 0
           JMP to END
      R2 = R0 + R2
      R1 = R1 - 1
      JMP LOOP
END:
      INFINITE LOOP
```

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

Exercise: Implementing Multiplication

- Convert to Hack Assembly

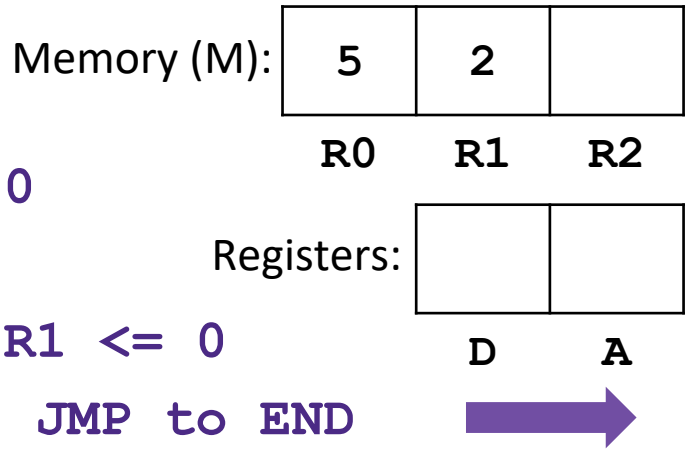



```
START:
   R2 = 0
LOOP:
     IF R1 <= 0
          JMP to END
     R2 = R0 + R2
     R1 = R1 - 1
     JMP LOOP
END:
     INFINITE LOOP
```

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

Exercise: Implementing Multiplication

- Convert to Hack Assembly

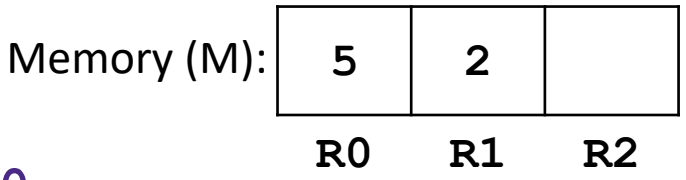



```
START:
   R2 = 0
LOOP:
      IF R1 <= 0
           JMP to END
      R2 = R0 + R2
      R1 = R1 - 1
      JMP LOOP
END:
      INFINITE LOOP
```

```
(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

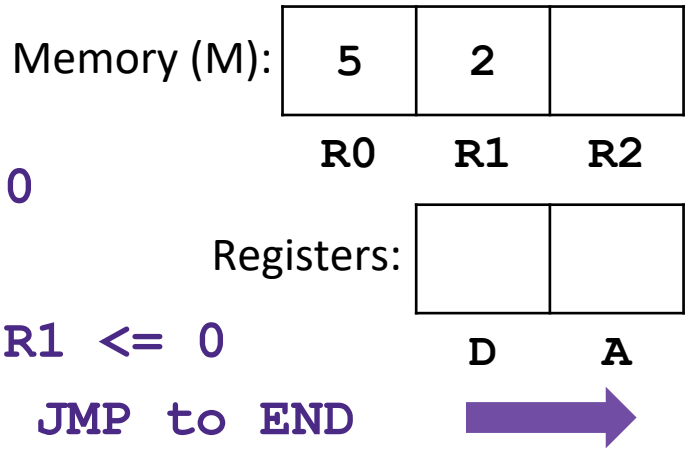



```
START:
   R2 = 0
LOOP:
      IF R1 <= 0
           JMP to END
      R2 = R0 + R2
      R1 = R1 - 1
      JMP LOOP
END:
      INFINITE LOOP
```

```
(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

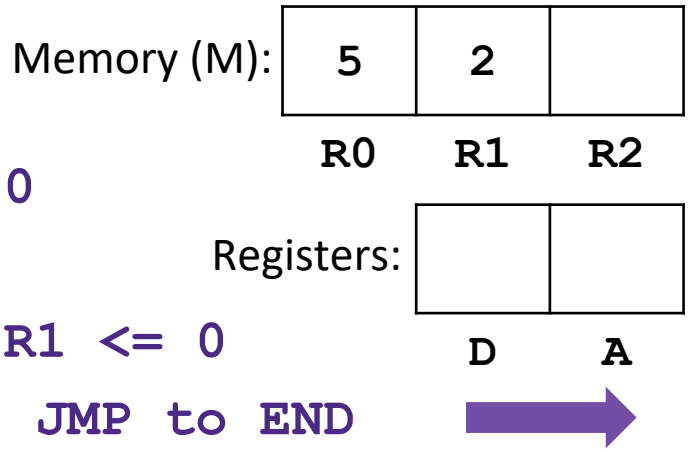



```
START:
  R2 = 0
LOOP:
     IF R1 <= 0
          JMP to END
     R2 = R0 + R2
     R1 = R1 - 1
     JMP LOOP
END:
     INFINITE LOOP
```

```
(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)
```

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

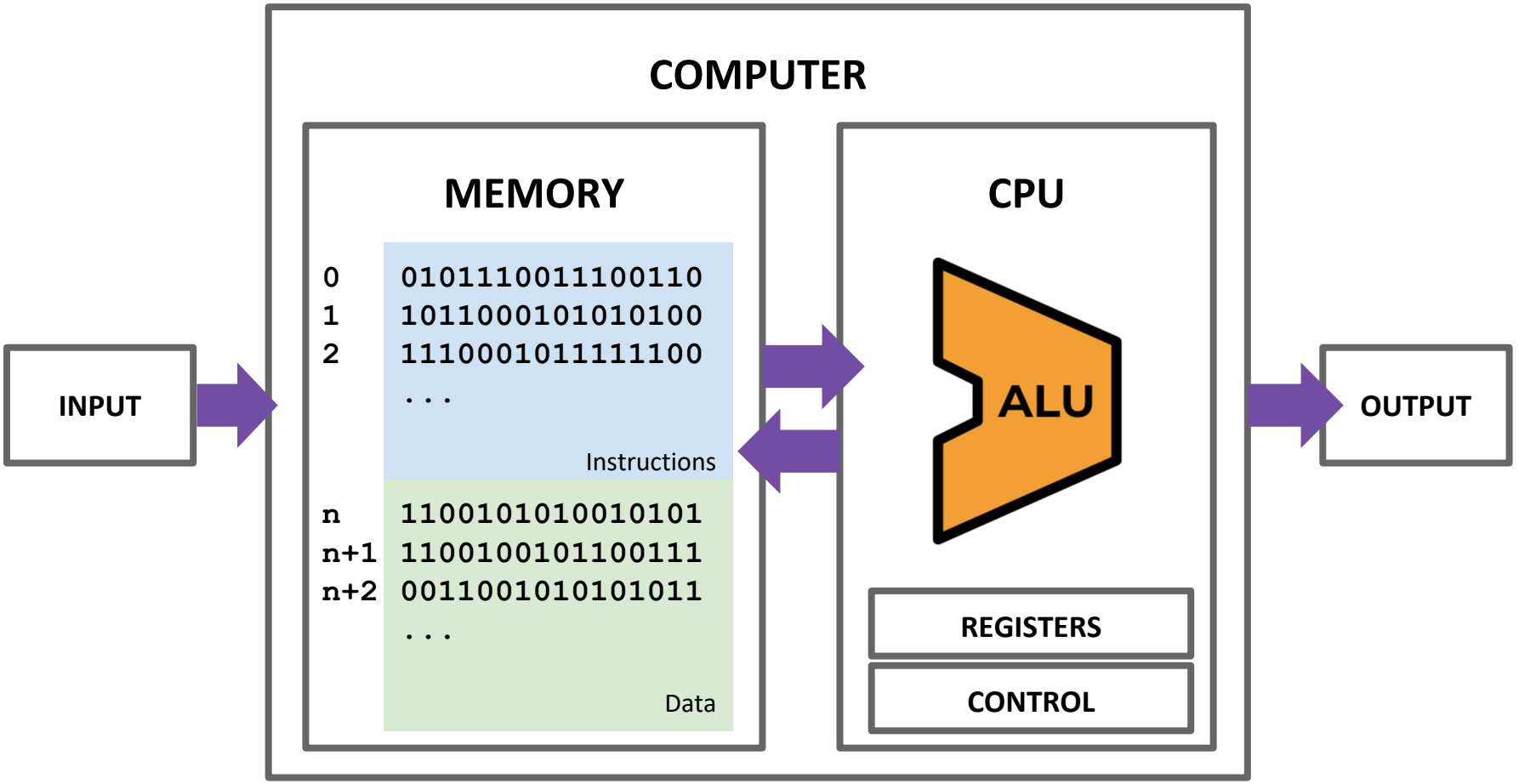
- All your hardware efforts are about to pay off!

- Perspective: **BUILDING A COMPUTER**

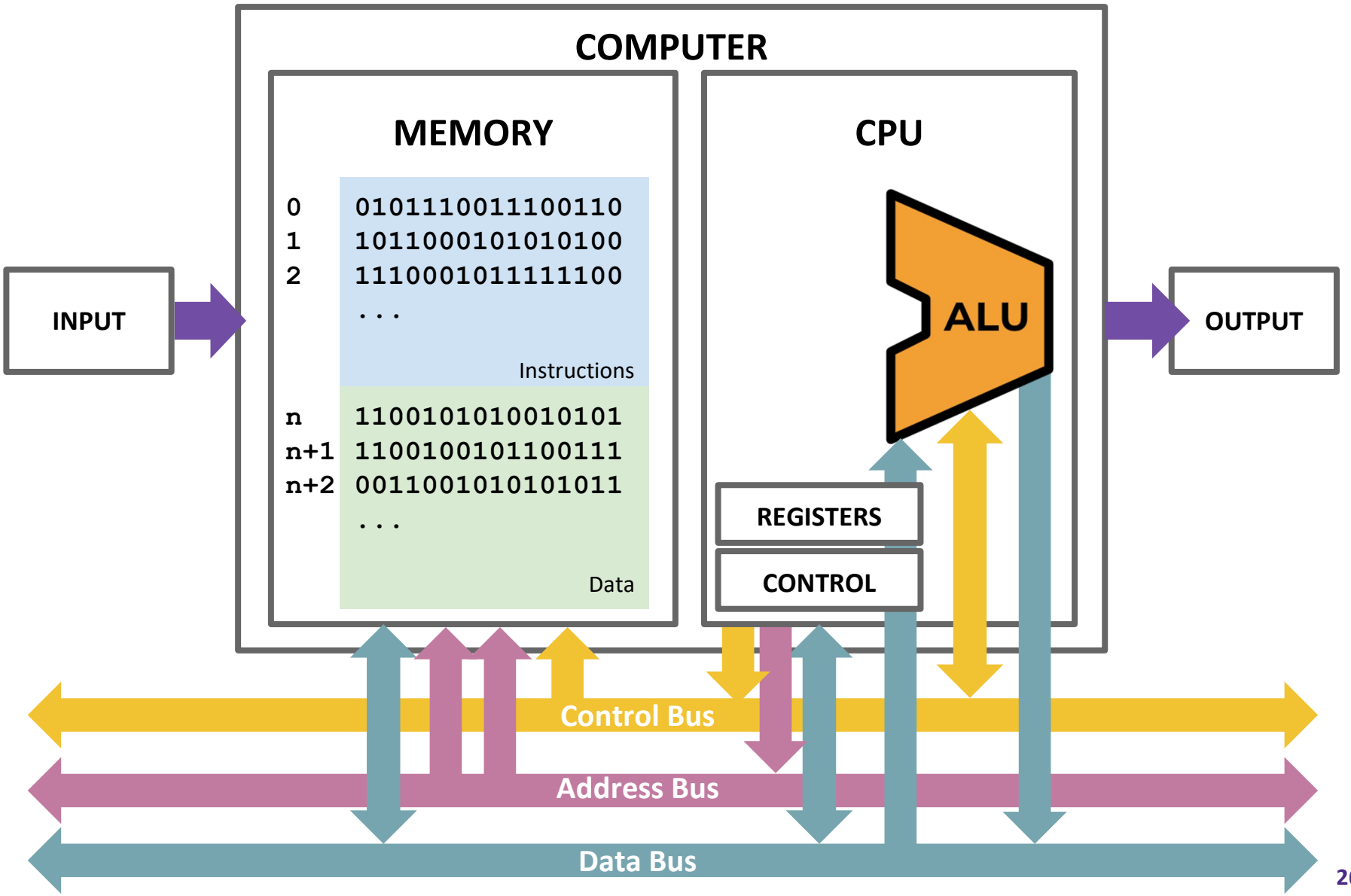
- In Project 6, you will build **`Computer.hdl`**, the final, top-level chip in this course
  - For all intents and purposes, a real computer
  - Simplified, but organization very similar to your laptop

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

Von Neumann Architecture



Connecting the Computer: Buses

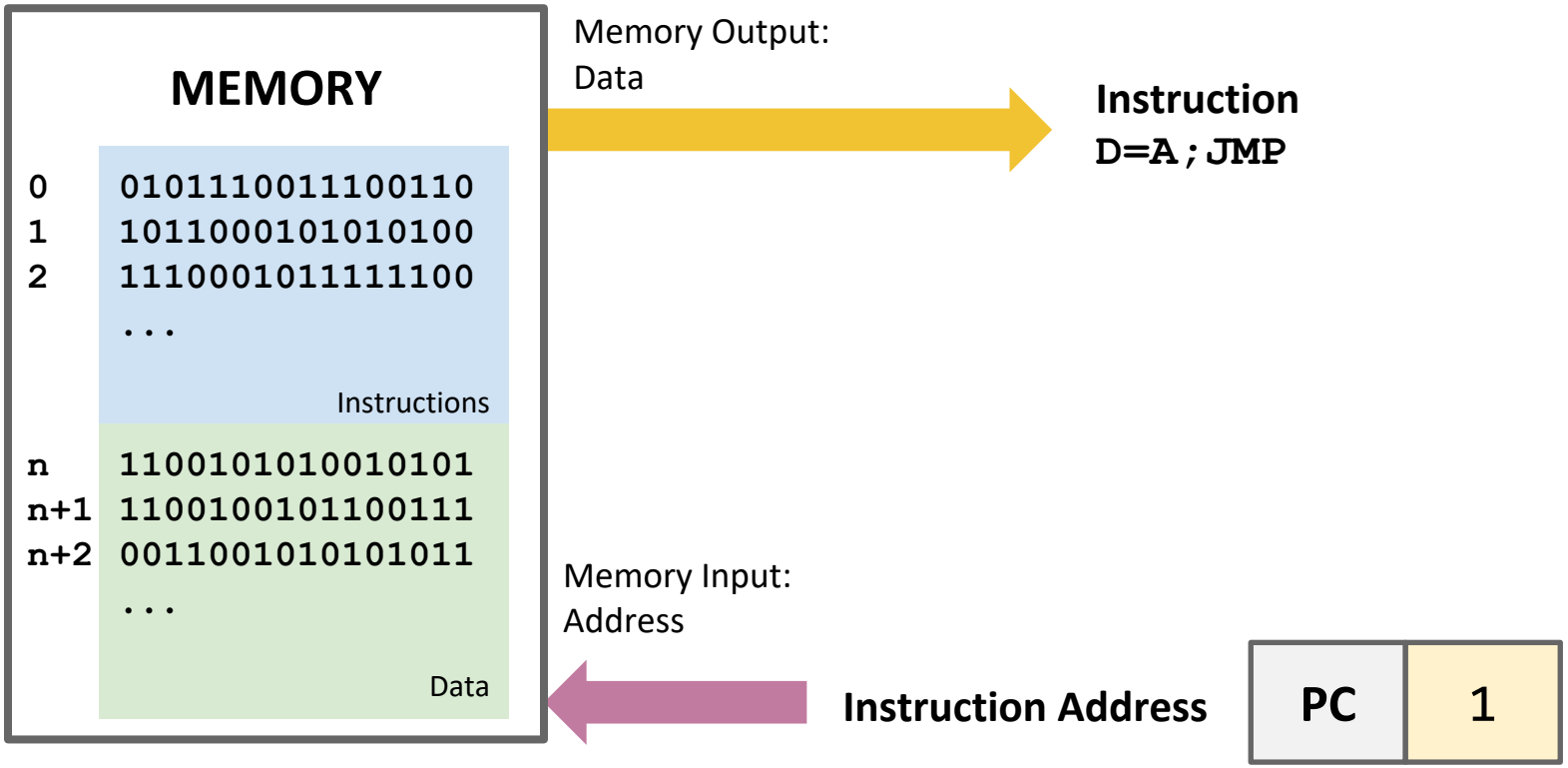


Basic CPU Loop

- 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

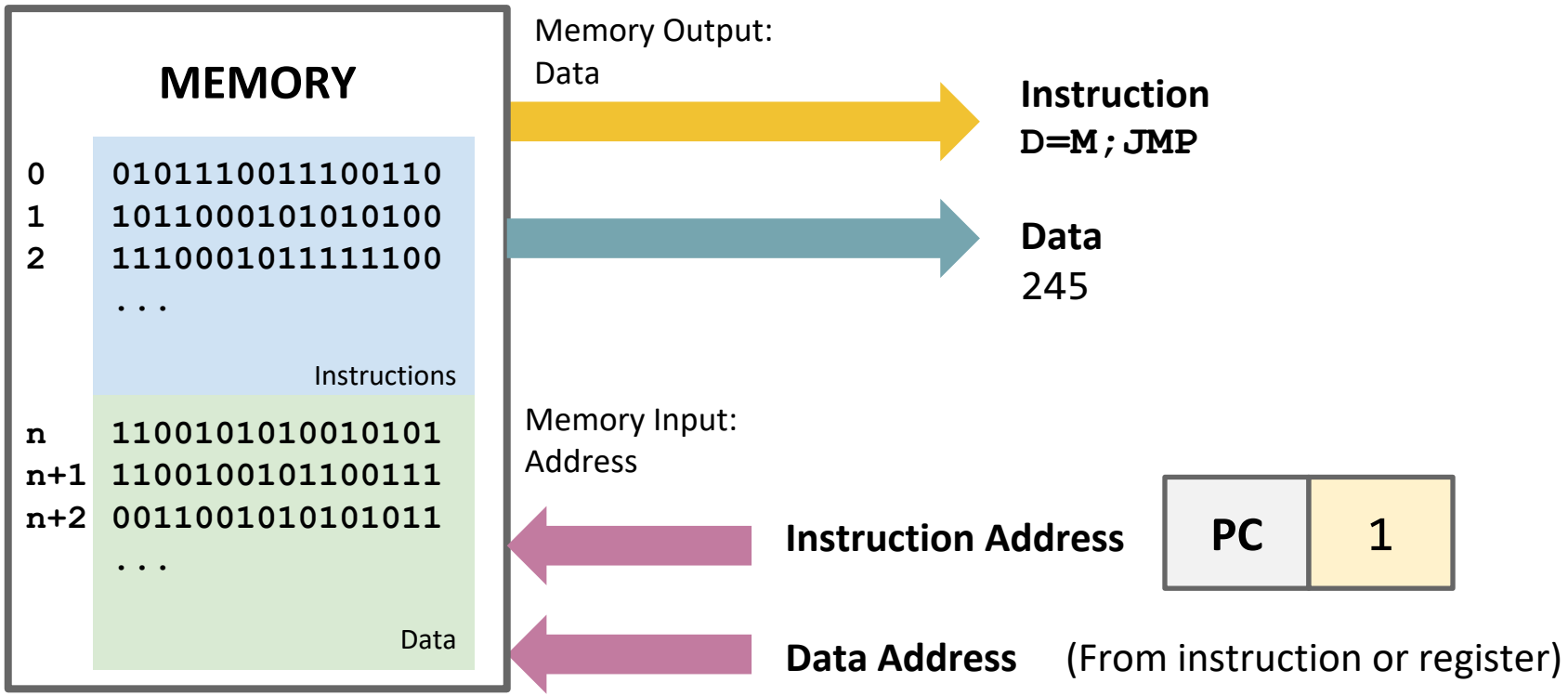


Executing

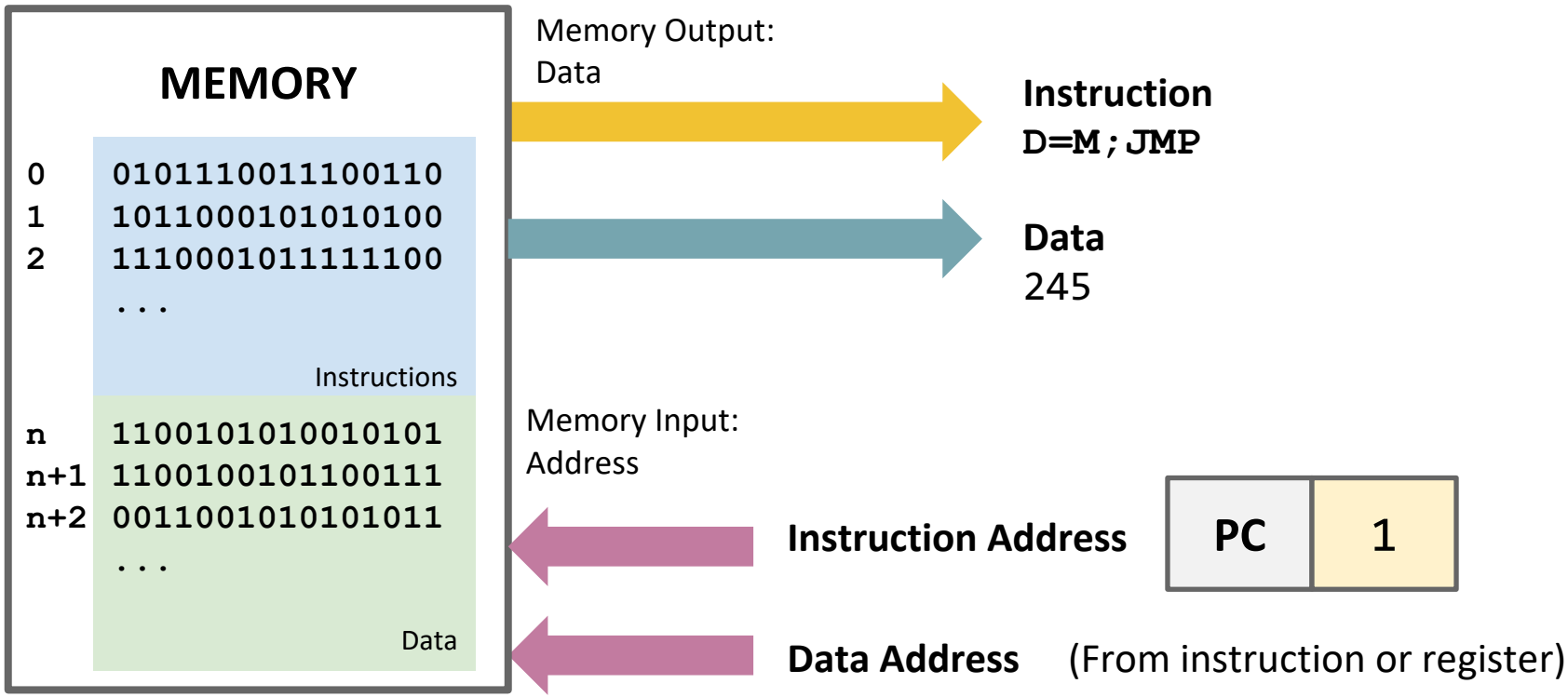
- The instruction bits describe exactly "what to do"
  - A-instruction or C-instruction?
  - Which operation for the ALU?
  - What memory address to read? To write?
  - If I should jump after this instruction, and where?

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

Combining Fetch & Execute

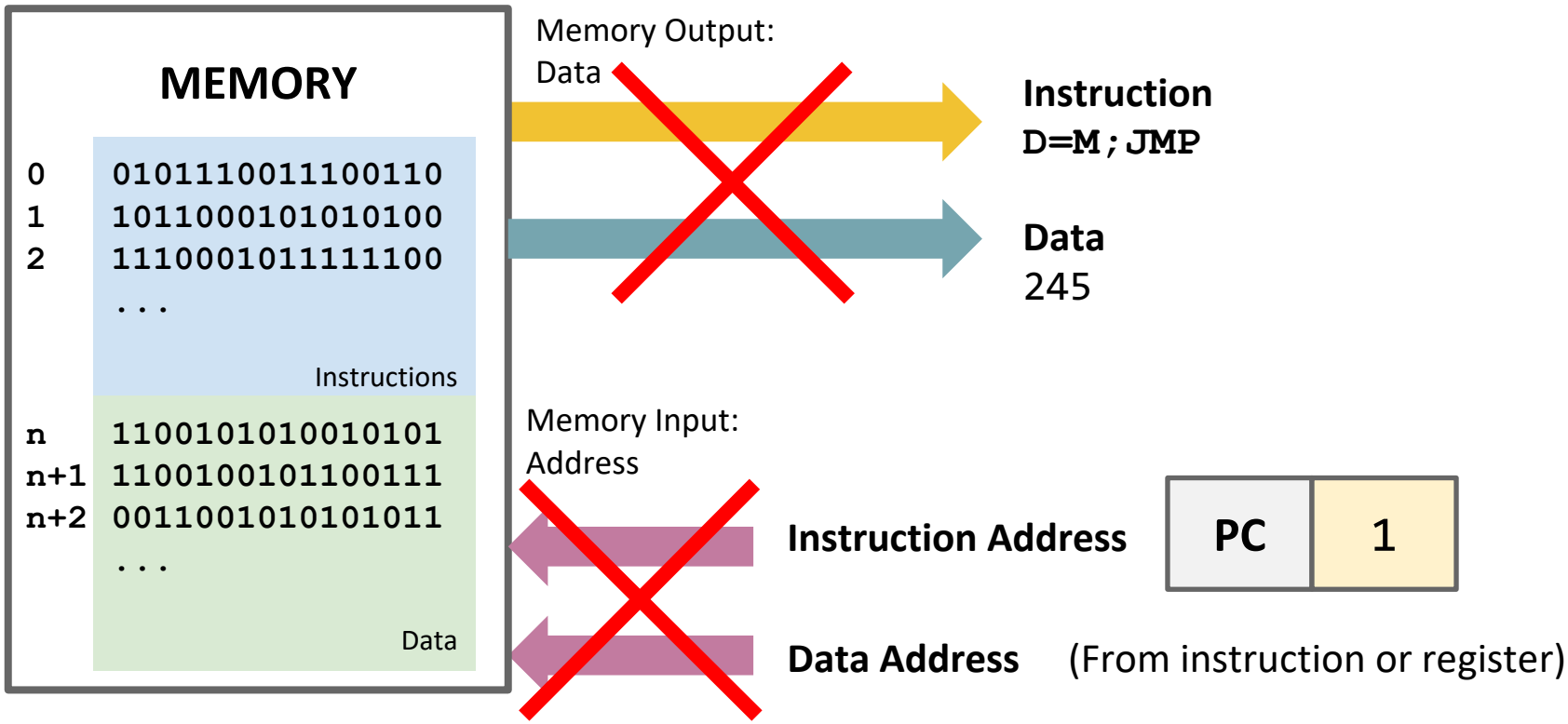


Combining Fetch & Execute



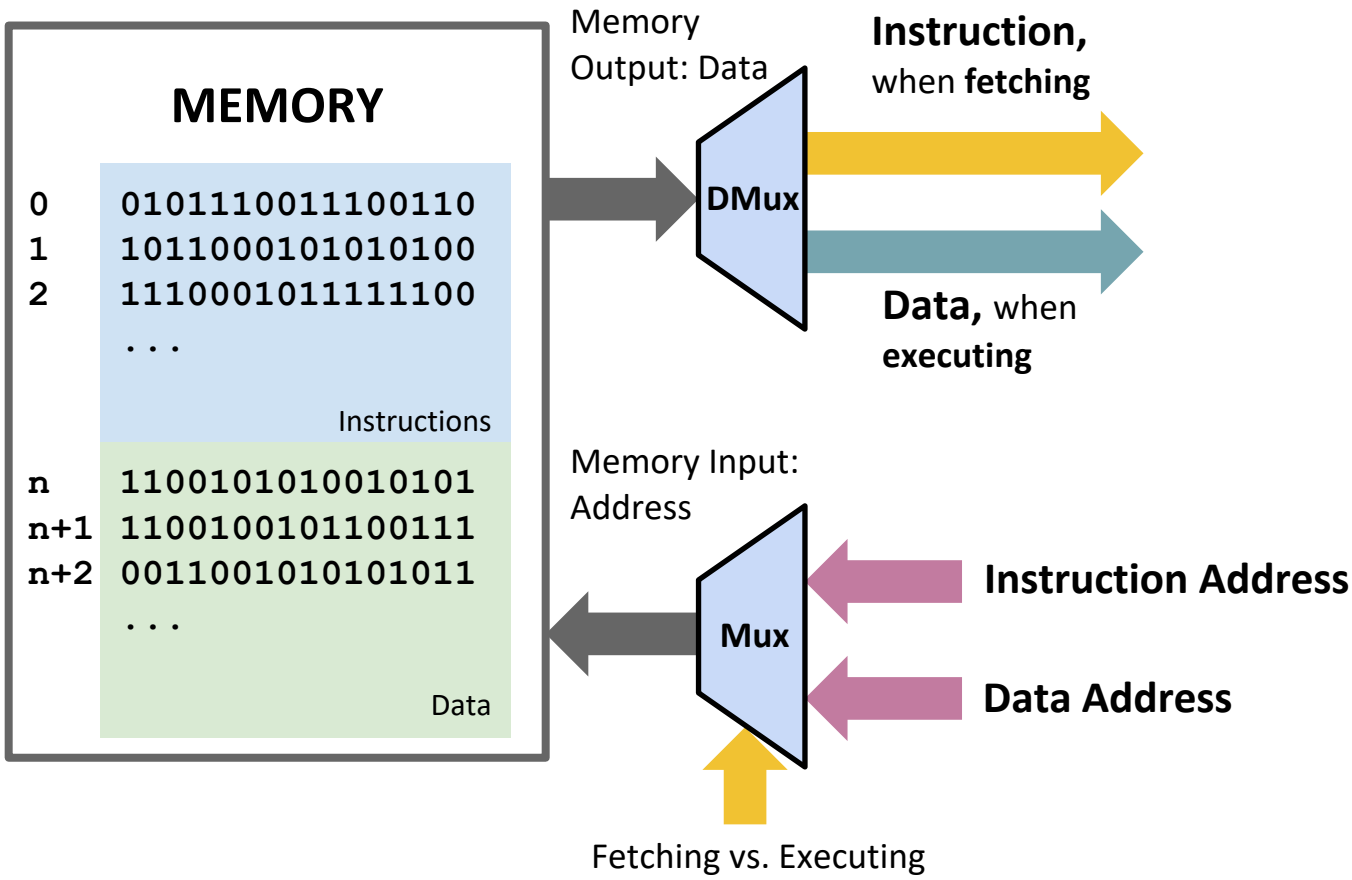
- Could we implement with **`RAM16K.hdl`**?
  - (Hint: Think about the I/O of RAM)

Combining Fetch & Execute



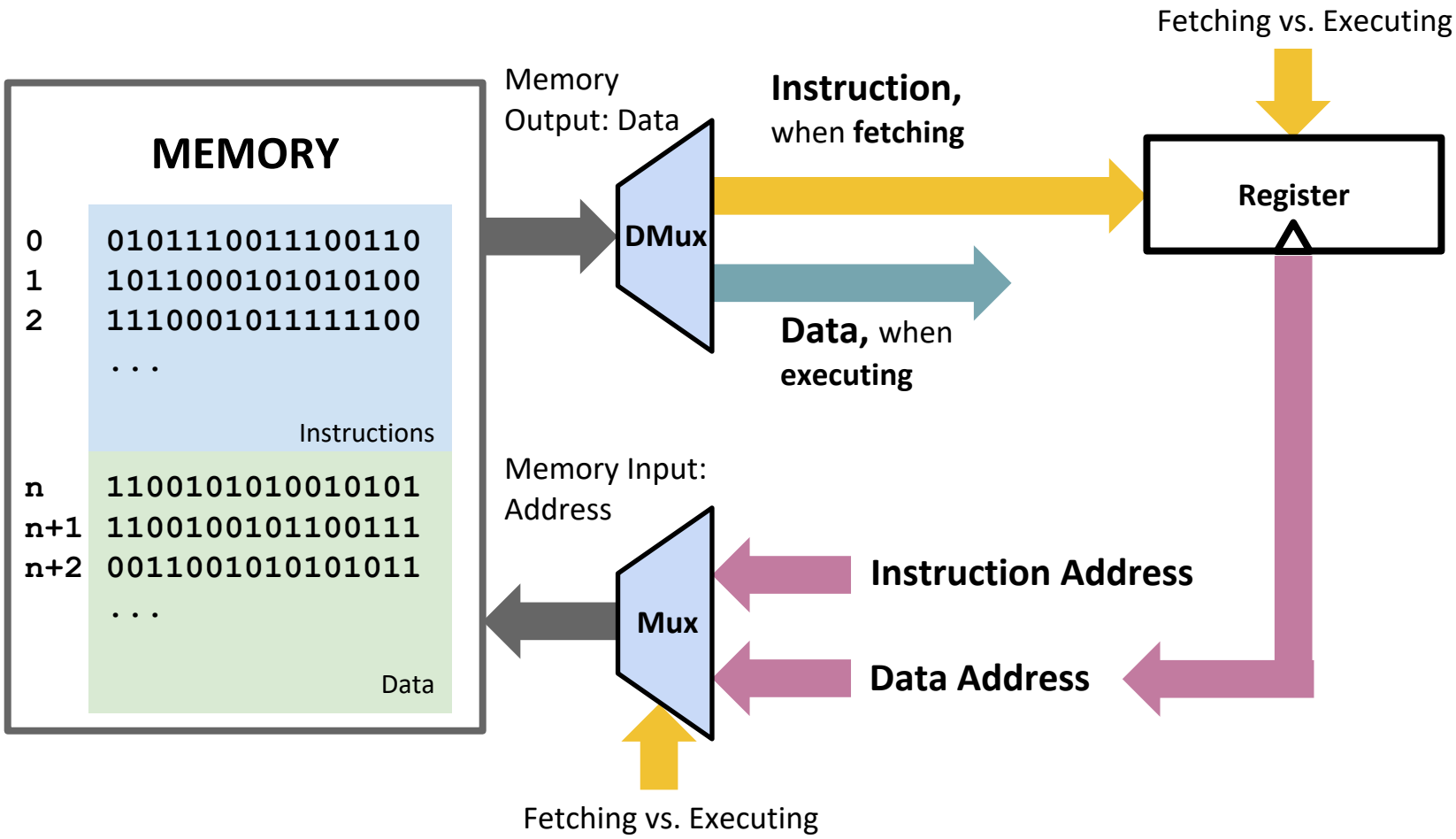
- Could we implement with **`RAM16K.hdl`**?
  - No! Our memory chips only have one input and one output

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

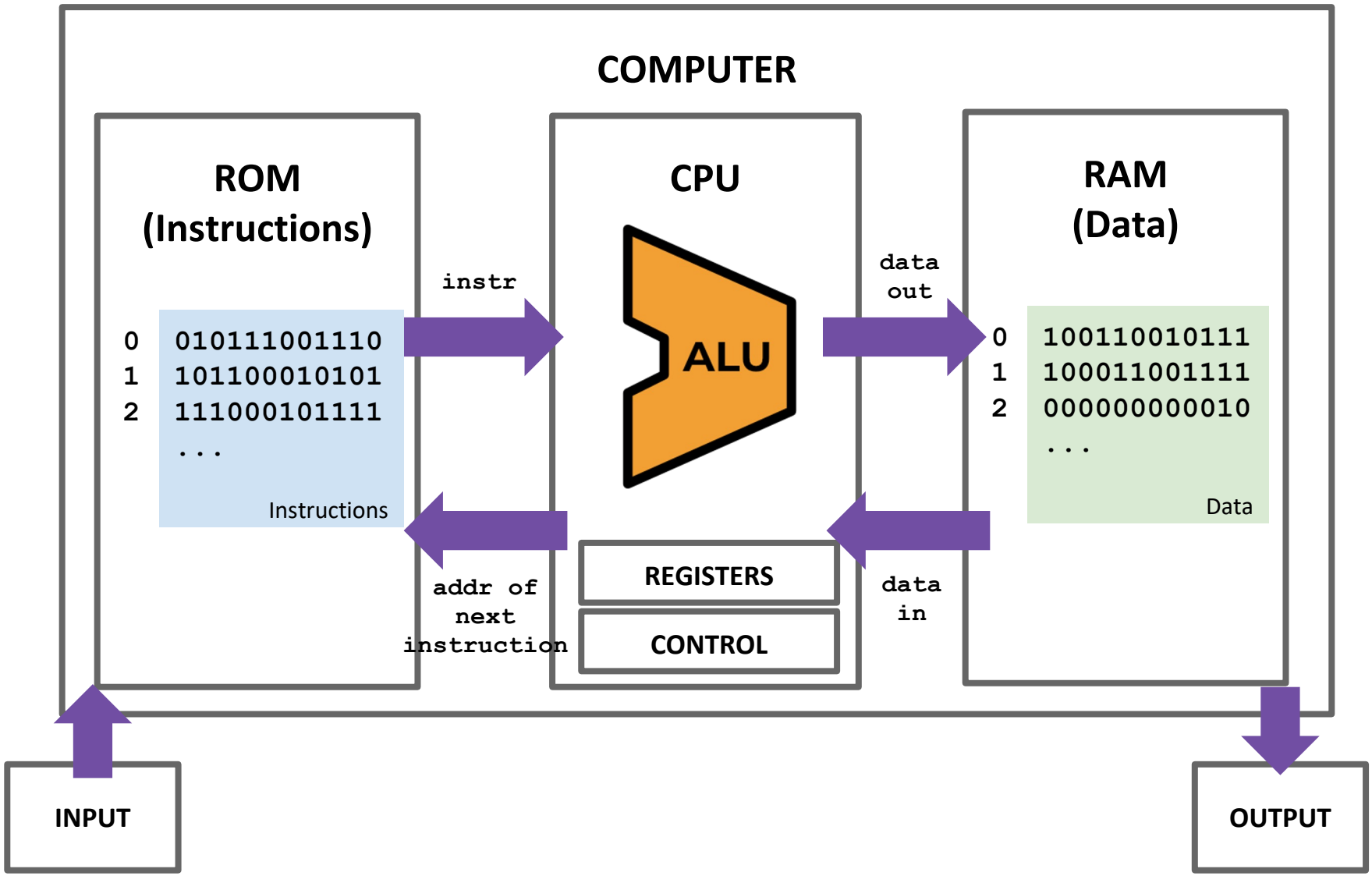
Solution 2: Separate Memory Units

- 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

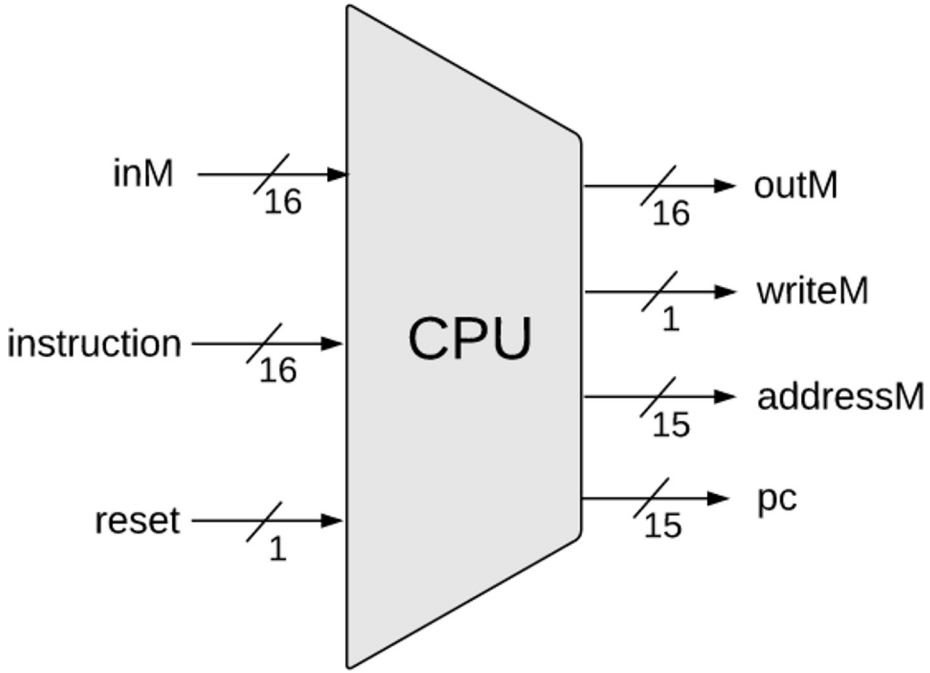
- Exam Preparation
  - Study Strategies, Mock Exam Problem
- Multiplication Implementation Exercise
  - Multiplying Two Numbers in Hack Assembly
- Building a Computer
  - Architecture, Fetch and Execute Cycle
- **Hack CPU Interface**
  - **Implementation and Operations**

Hack CPU


COMPUTER
ROM
(Instructions)
0 010111001110
1 101100010101
2 111000101111
...
Instructions
instr
CPU
ALU
REGISTERS
CONTROL
addr of
next
instruction
data
out
data
in
RAM
(Data)
0 100110010111
1 100011001111
2 000000000010
...
Data
INPUT
OUTPUT

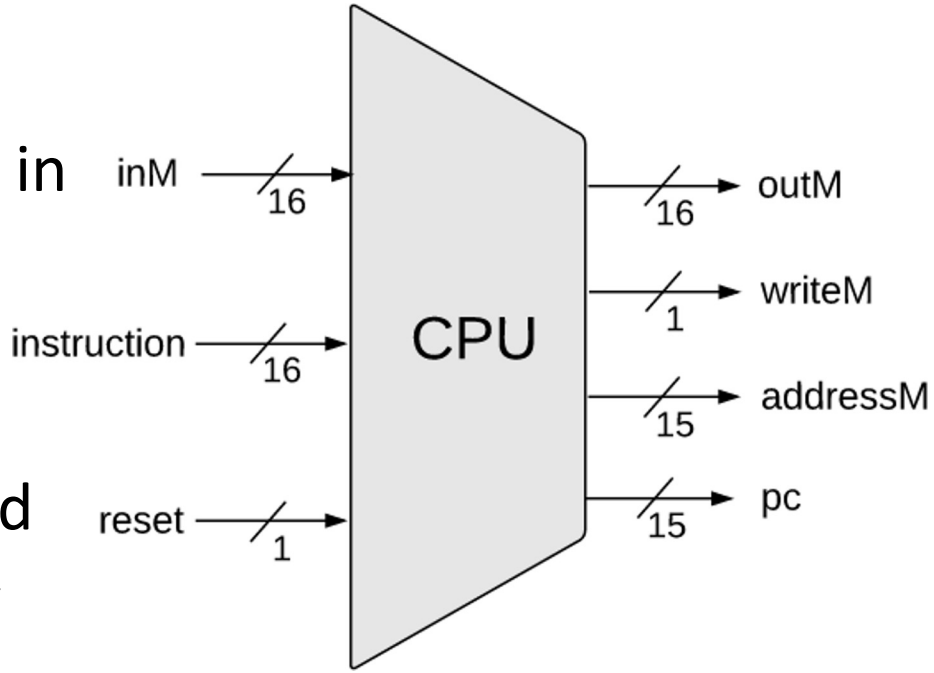
Hack CPU Interface Inputs

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

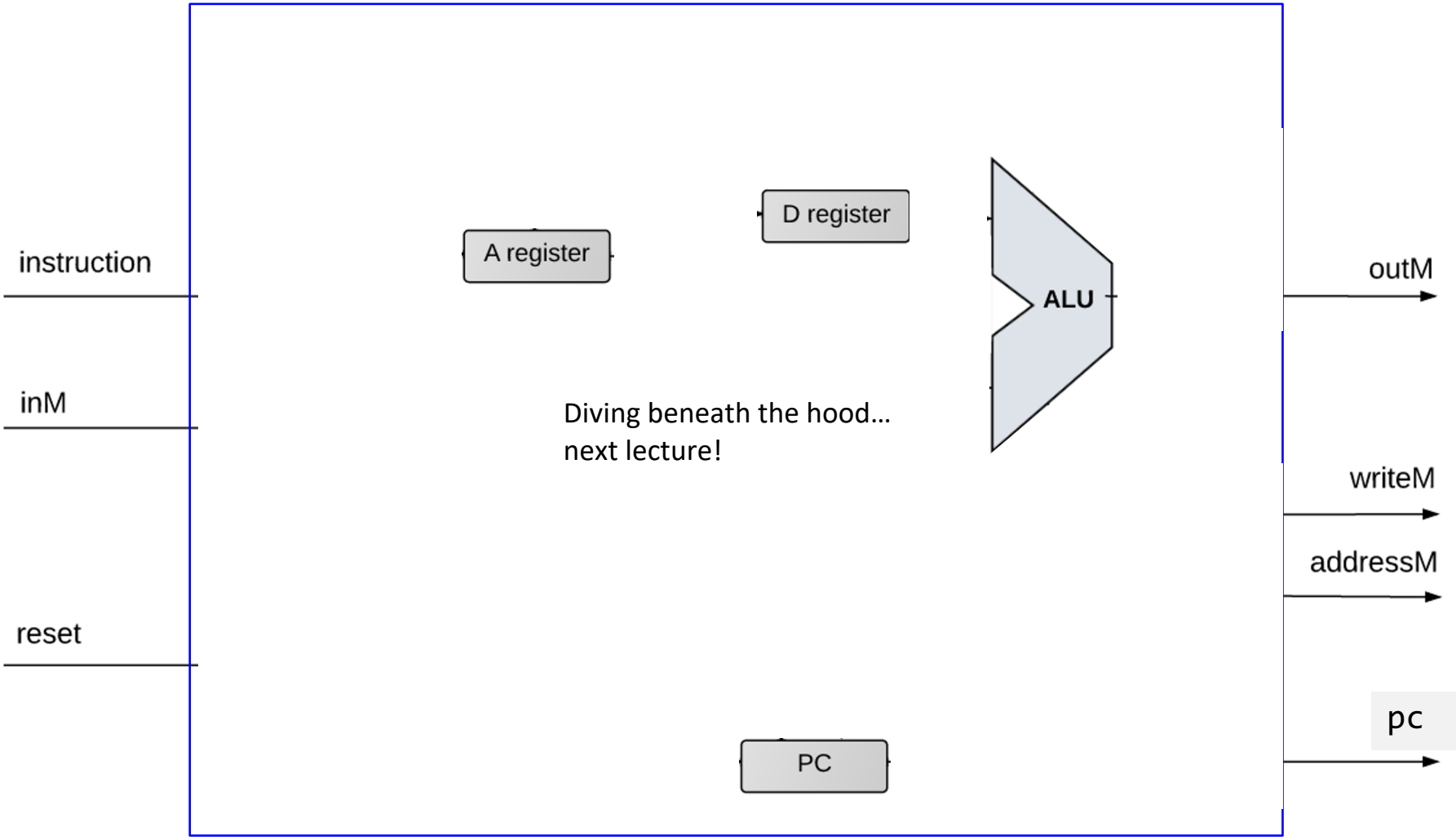


Hack CPU Interface Outputs

- **`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



Post-Lecture 9 Reminders

- **Project 5 due this Thursday (4/27) at 11:59pm**
- **Midterm exam coming up on 5/4 during lecture time**
- Preston has office hours after class in CSE2 152
  - Feel free to post your questions on the Ed board as well