CSE 390B, 2024 Winter

Building Academic Success Through Bottom-Up Computing

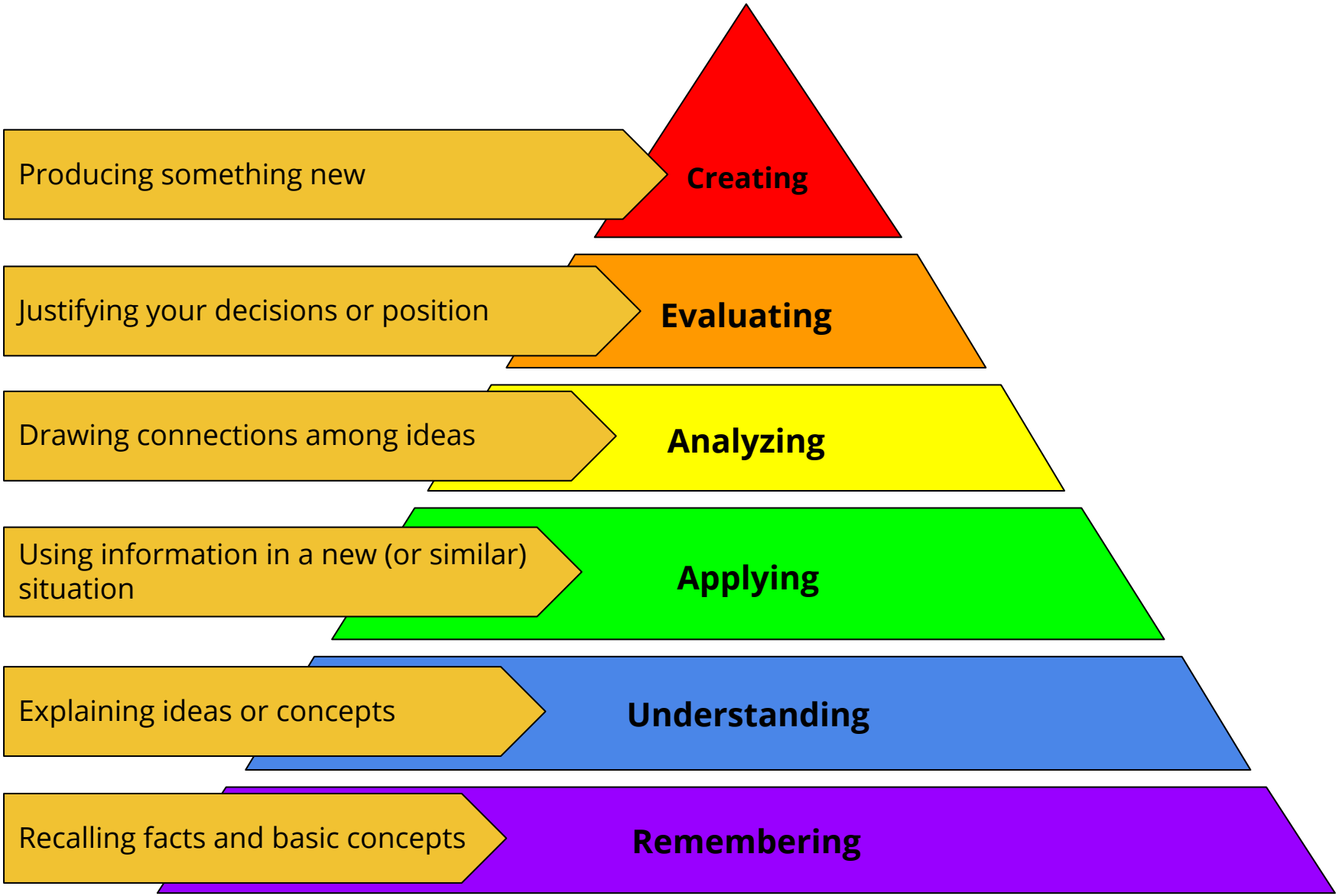
Cornell Note-taking & Machine Language

Cornell Note-taking Method, Machine Languages, Control Flow of Computer Instructions, The Hack Assembly Language

Lecture Outline

- **Cornell Note-taking Method**
  - **System for Taking, Organizing, and Reviewing Notes**
- Machine Languages
  - Assembly Languages, Producing Machine Code
- Control Flow of Computer Instructions
  - Jumps in Assembly, The Program Counter
- The Hack Assembly Language
  - Registers, A-Instructions, Symbols, & C-Instructions

Recap: Bloom's Taxonomy



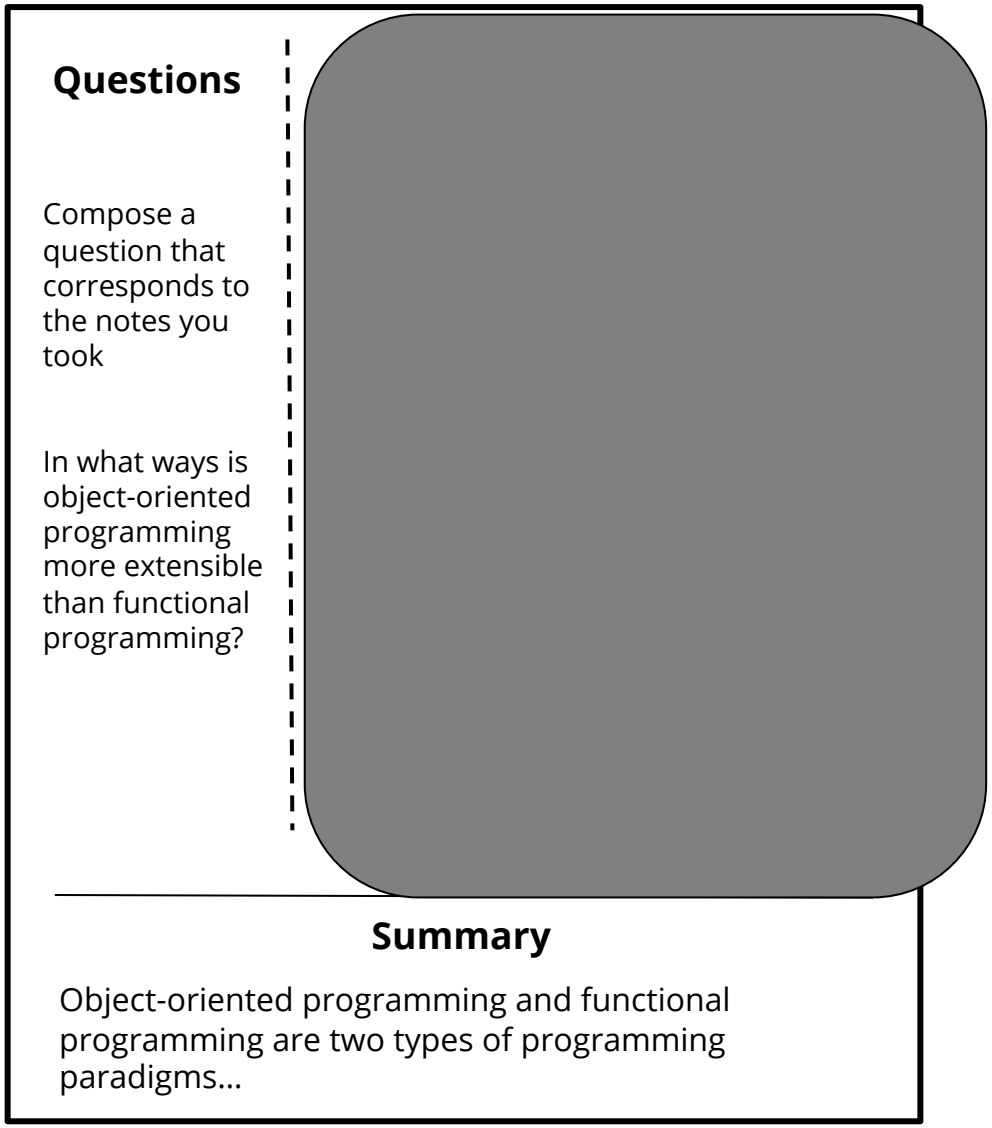
Cornell Note Taking Method

Questions	Notes
Compose a question that corresponds to the notes you took In what ways is object-oriented programming more extensible than functional programming?	I. Main Topic ○ Sub point ○ definition ○ example ** II. Object-Oriented Programming ○ Encapsulates the data and the operations for a given data type ○ Provides abstractions - you don't need to know how a car is implemented in order to use it ○ Extensibility - easier to add new data types III. Functional Programming ○ Extensibility - easier to add new operations

Summary

Object-oriented programming and functional programming are two types of programming paradigms…

Cornell Note Taking Method



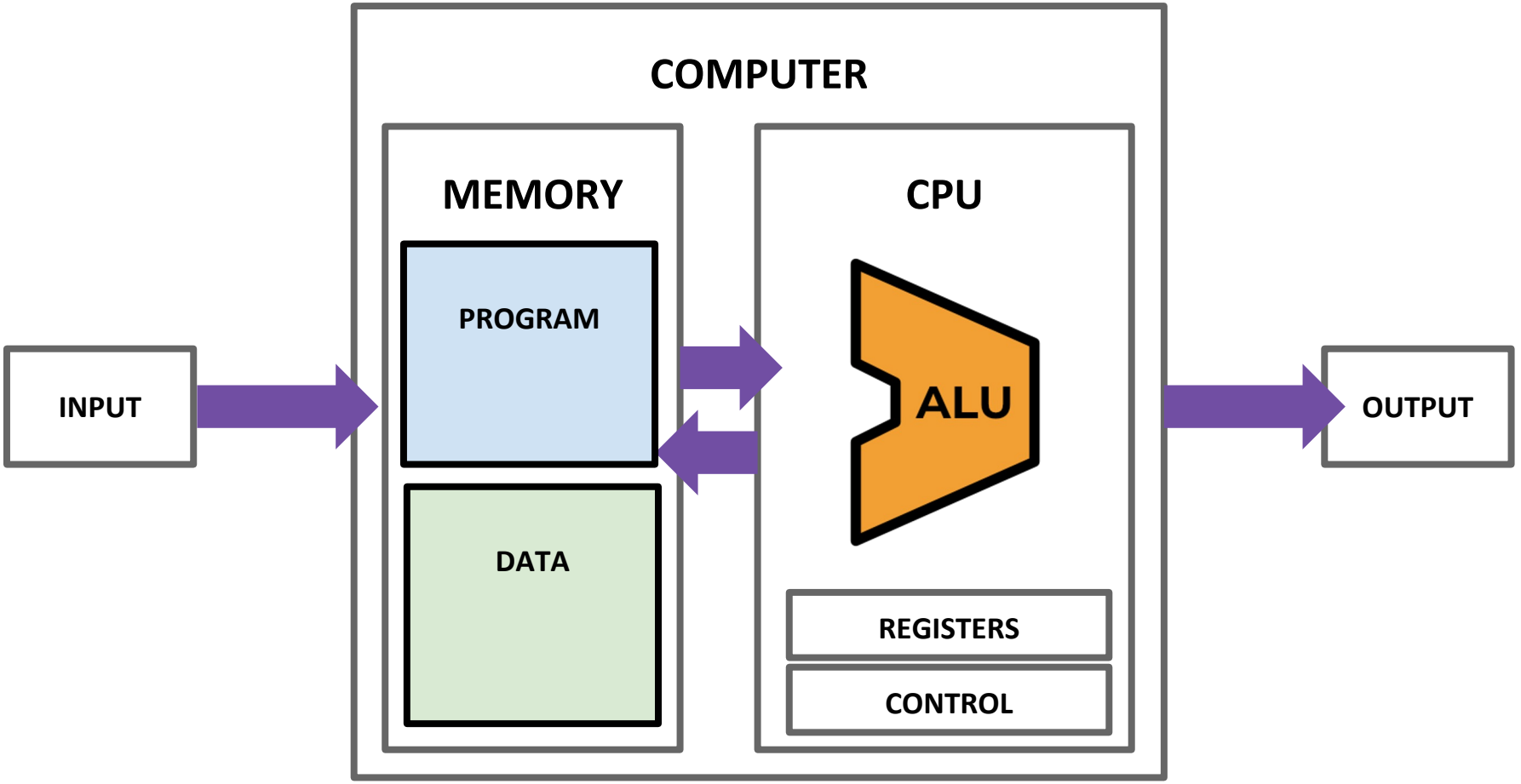
Applying the Cornell Note-Taking Method

- Try it during today's technical lecture!
- You will have a chance to reflect on your experience taking Cornell Notes in Project 4

Lecture Outline

- Cornell Note-taking Method
  - System for Taking, Organizing, and Reviewing Notes
- **Machine Languages**
  - **Assembly Languages, Producing Machine Code**
- Control Flow of Computer Instructions
  - Jumps in Assembly, The Program Counter
- The Hack Assembly Language
  - Registers, A-Instructions, Symbols, & C-Instructions

Revisiting The Von Neumann Architecture



(This picture will get more detailed as we go!)

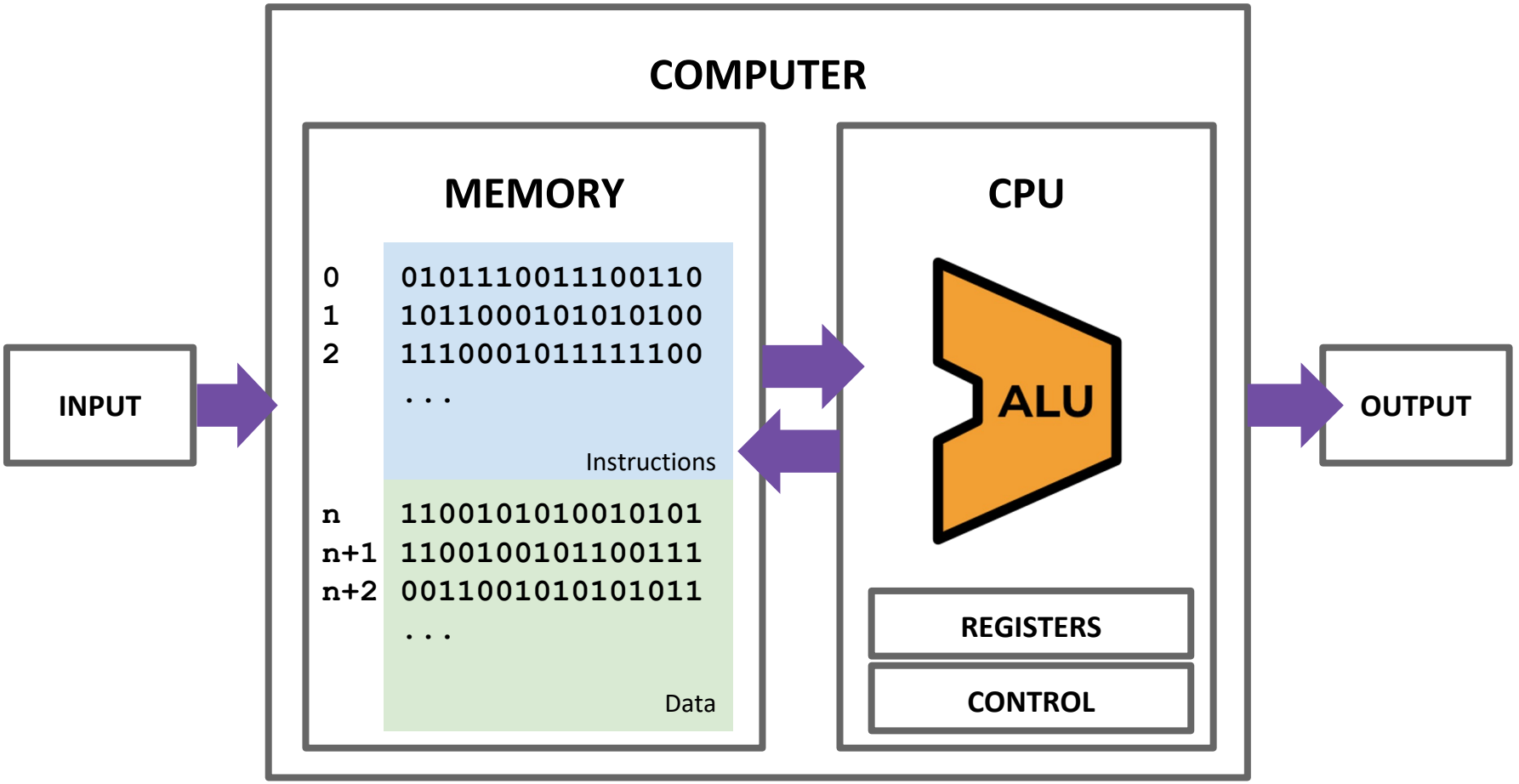
Machine Code

- Instructions are stored in memory, so they must be able to be encoded in binary

- When we refer to **machine code**, we are typically talking about this binary representation of code

- Each instruction is a sequence of 0s and 1s
  - Our computer / hardware specification is what gives meaning to each part of this sequence
  - "Is this an add or subtract instruction? What are the inputs?"

Storing the Program



(This picture will get more detailed as we go!)

Assembly Languages

- Writing code using 0s and 1s is tedious and error prone
- Assembly languages are a human-readable format of binary instructions that a CPU runs
- Each human-readable assembly instruction has a corresponding binary machine code instruction
  - Example: **`addq reg1, reg2 == 0b1011000101010100`**
- Assembly is often used as an intermediary between a high-level programming language and machine code

Producing Machine Code

```
while (i < 100)
{
  sum += arr[i];
  i++;
}
```

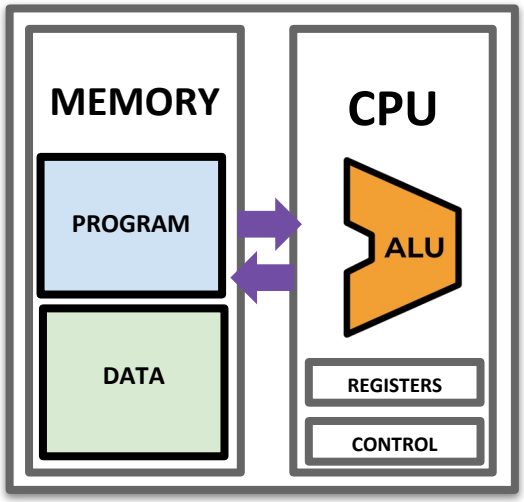
Java

Compile

```
0101110011100110
1011000101010100
1110001011111100
...
```

Machine Code Instructions

Load & Execute



Producing Machine Code

```
while (i < 100)
{
  sum += arr[i];
  i++;
}
```

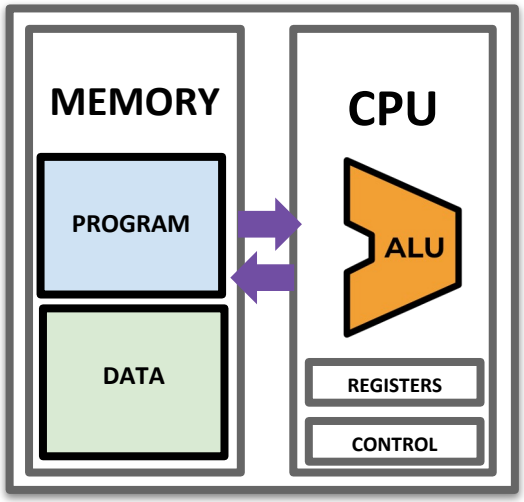
Java

Compile

```
0101110011100110
1011000101010100
1110001011111100
...
```

Machine Code Instructions

Load & Execute




```
movq $5, %rdx
addq %rsx, %rdx
movq %rdx, %rax
ret
```

Assembly Language

Assemble

Producing Machine Code

```
while (i < 100)
{
  sum += arr[i];
  i++;
}
```

Java

Compile

Compile

```
movq $5, %rdx
addq %rsx, %rdx
movq %rdx, %rax
ret
```

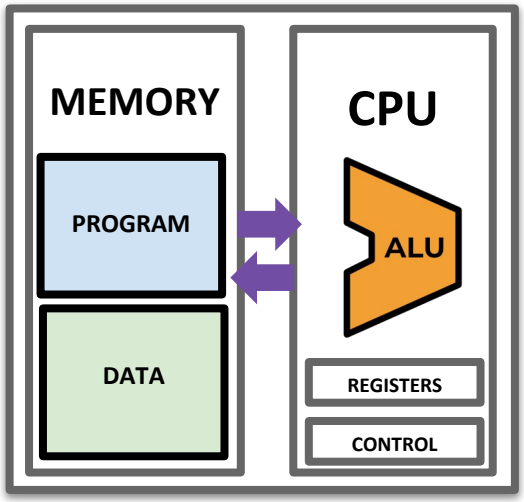
Assembly Language

Assemble

```
0101110011100110
1011000101010100
1110001011111100
...
```

Machine Code Instructions

Load & Execute



Machine Language

- Specification of the Hardware / Software interface
  - What operations are supported?
  - What do they operate on?
  - How is the program controlled?
- Usually in close correspondence with the hardware architecture
  - Different specification for different hardware platforms
- Cost and Performance Tradeoffs
  - Silicon area and complexity
  - Time to complete instruction
  - Power consumption

Machine Operations

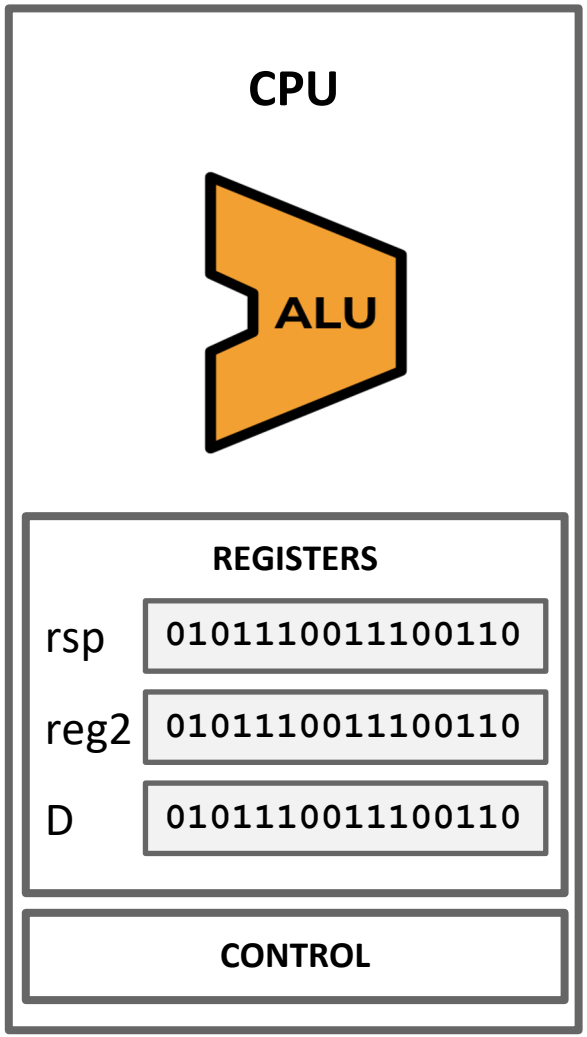
- Correspond to the operations supported by hardware:
  - Arithmetic (+, –)
  - Logical (And, Or)
  - Flow Control (“go to instruction n”, “if (condition) then go to instruction n”)

- Differences between machine languages:
  - Instruction set richness (e.g., division? bulk copy?)
  - Data types (e.g., word size, floating point)

Registers

- CPU typically has a small number of **registers**
  - Very efficient to access
  - Used for intermediate, short-term "scratch work"

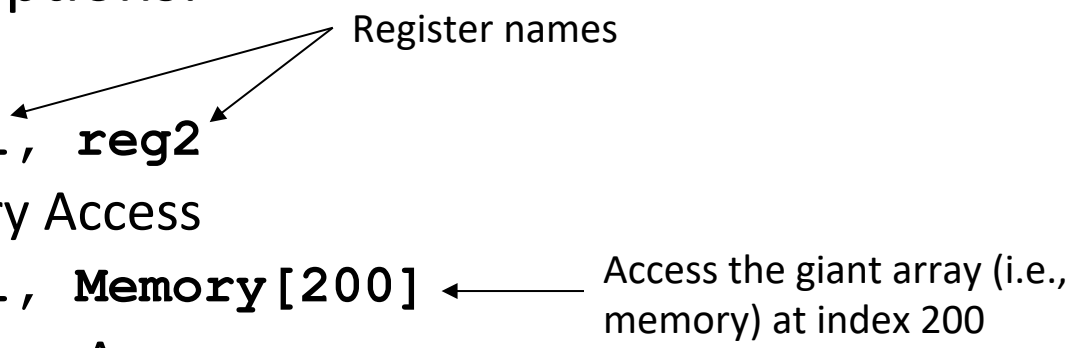
- Number and use of registers is a central part of any machine language



Addressing Modes

- "What locations can I specify in my assembly code?"

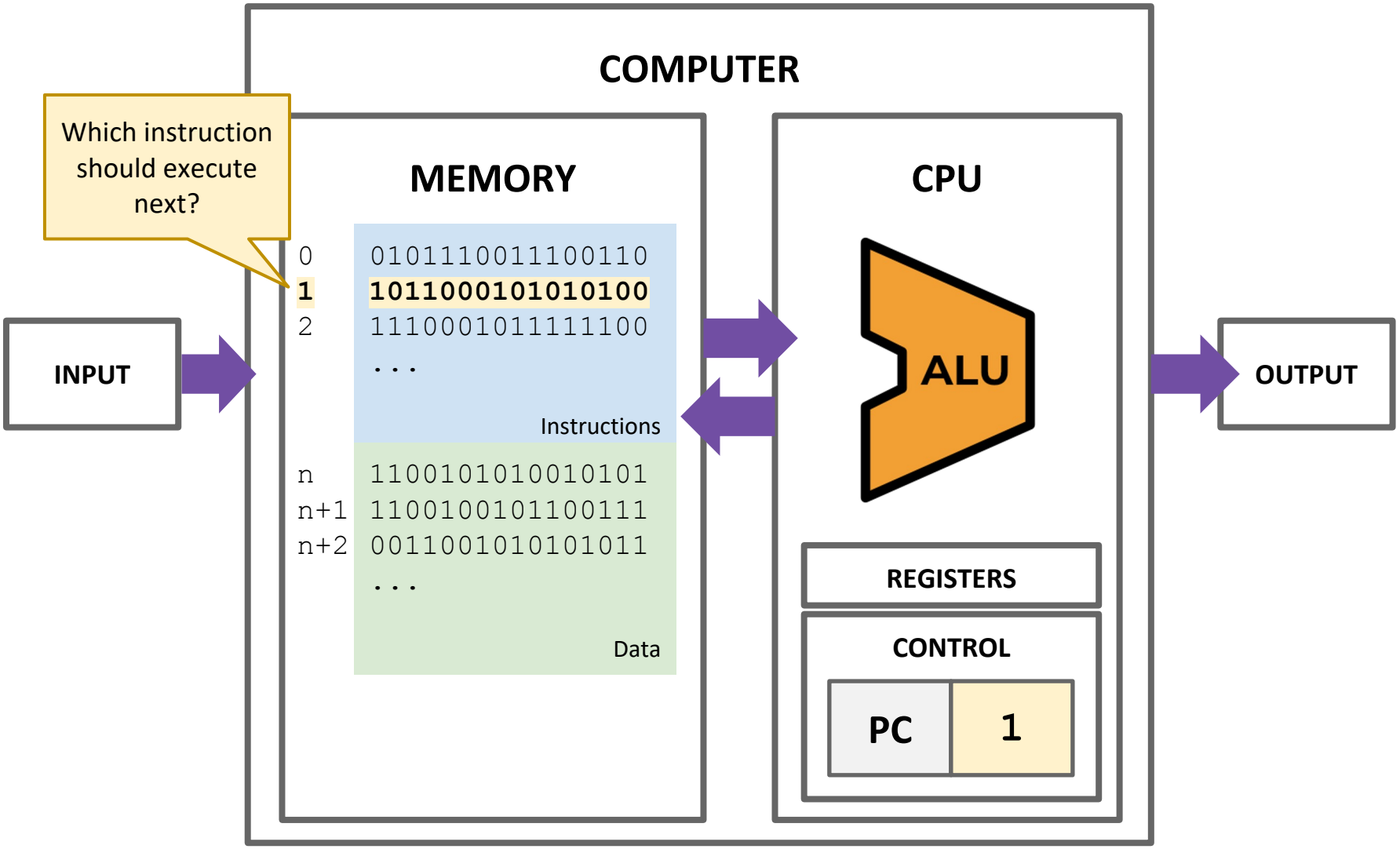
- Some useful options:
  - Register
    - `add reg1, reg2`
  - Direct Memory Access
    - `add reg1, Memory[200]`
  - Indirect Memory Access
    - `add reg1, Memory[reg2]`
  - Immediate
    - `add 100, reg2`



Lecture Outline

- Cornell Note-taking Method
  - System for Taking, Organizing, and Reviewing Notes
- Machine Languages
  - Assembly Languages, Producing Machine Code
- **Control Flow of Computer Instructions**
  - **Jumps in Assembly, The Program Counter**
- The Hack Assembly Language
  - Registers, A-Instructions, Symbols, & C-Instructions

Flow Control



Flow Control: Unconditional Jumps

- Usually, the CPU just executes machine instructions in a sequence
  - Typically moves to the instruction with the next highest address

- Sometimes we want to **always** "jump" to another location
  - Example: At the end of an infinite loop

High Level Code (similar to Java)	Assembly Code

`while (true) {`
`  reg1++;`
`  <more loop body>`
`}`
`<code after loop>`
	`TOP:`
	`  add 1, reg1`
	`  <more loop body>`
	`  jmp TOP`
	`  <code after loop>`

Flow Control: Conditional Jumps

- Usually, the CPU just executes machine instructions in a sequence
  - Typically moves to the instruction with the next highest address

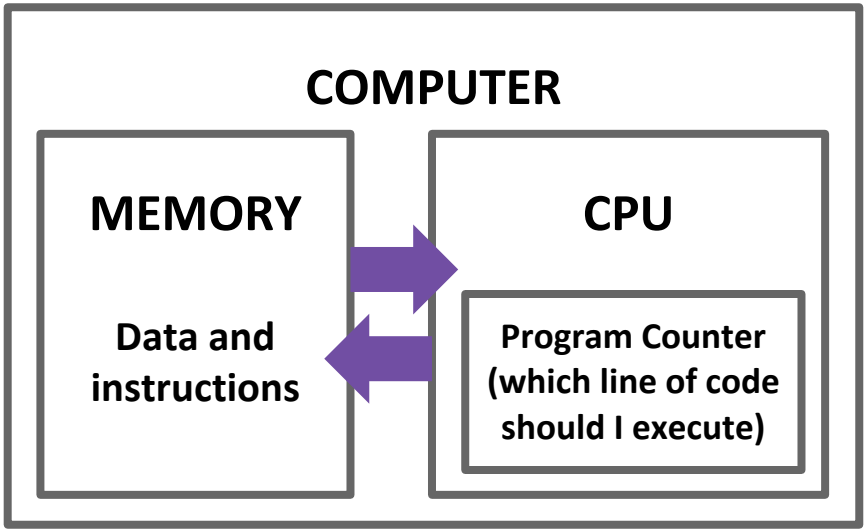
- Sometimes we want to "jump" **only if a condition is met**
  - Example: At the condition of an if statement

High Level Code (similar to Java)	Assembly Code

`if (reg1 < reg2) {`
`  reg1++;`
`}`
`reg2++;`
	`  cmp reg1, reg2`
	`  jge SKIP`
	`  add 1, reg1`
	`SKIP:`
	`  add 1, reg2`

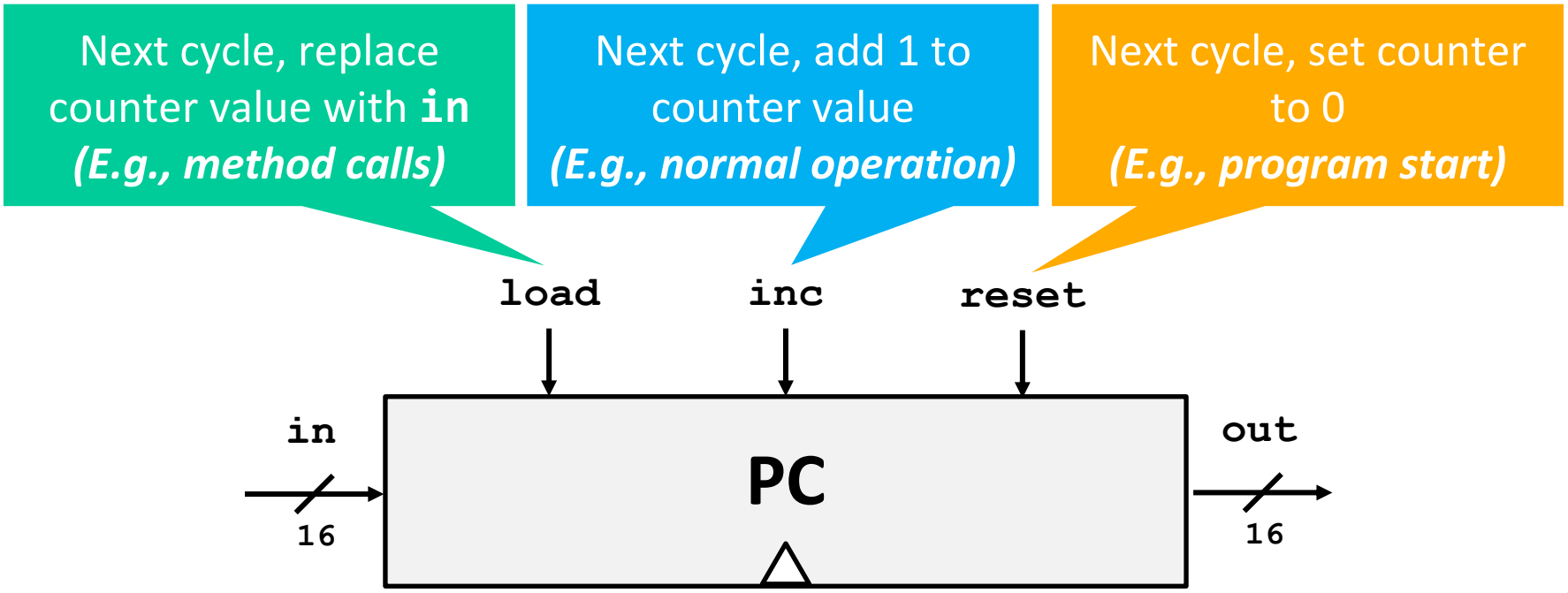
Program Counter (PC)

- Memory is used to store data as well as code
- Instructions and operations are stored at different addresses in memory
- Program Counter in the CPU keeps track of which address contains the instruction that should be executed next



Program Counter (PC)

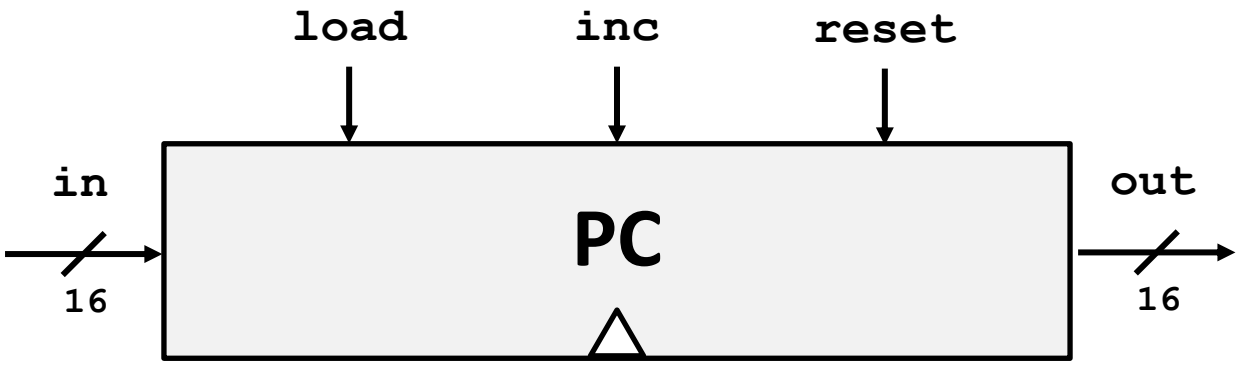
- Keeps track of what instruction we are executing
  - If the PC outputs 24, on the next clock cycle the computer runs the instruction at address 24 in the code segment



Program Counter (PC)

- Keeps track of what instruction we are executing
  - If the PC outputs 24, on the next clock cycle the computer runs the instruction at address 24 in the code segment
- Program counter specification:

```
if      (reset[t] == 1) out[t+1] = 0
else if (load[t] == 1)  out[t+1] = in[t]
else if (inc[t] == 1)   out[t+1] = out[t] + 1
else                    out[t+1] = out[t]
```

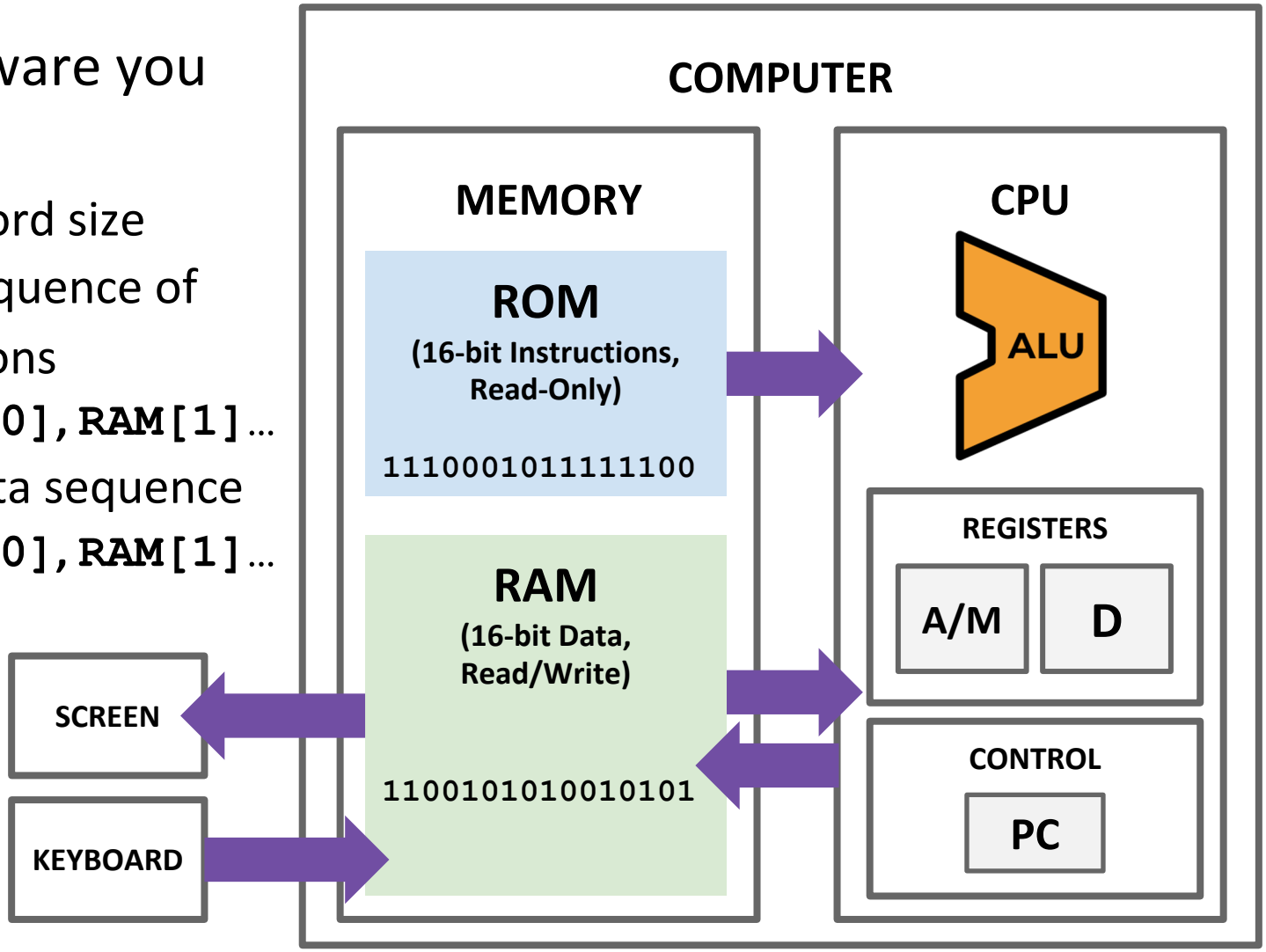


Lecture Outline

- Cornell Note-taking Method
  - System for Taking, Organizing, and Reviewing Notes
- Machine Languages
  - Assembly Languages, Producing Machine Code
- Control Flow of Computer Instructions
  - Jumps in Assembly, The Program Counter
- **The Hack Assembly Language**
  - **Registers, A-Instructions, Symbols, & C-Instructions**

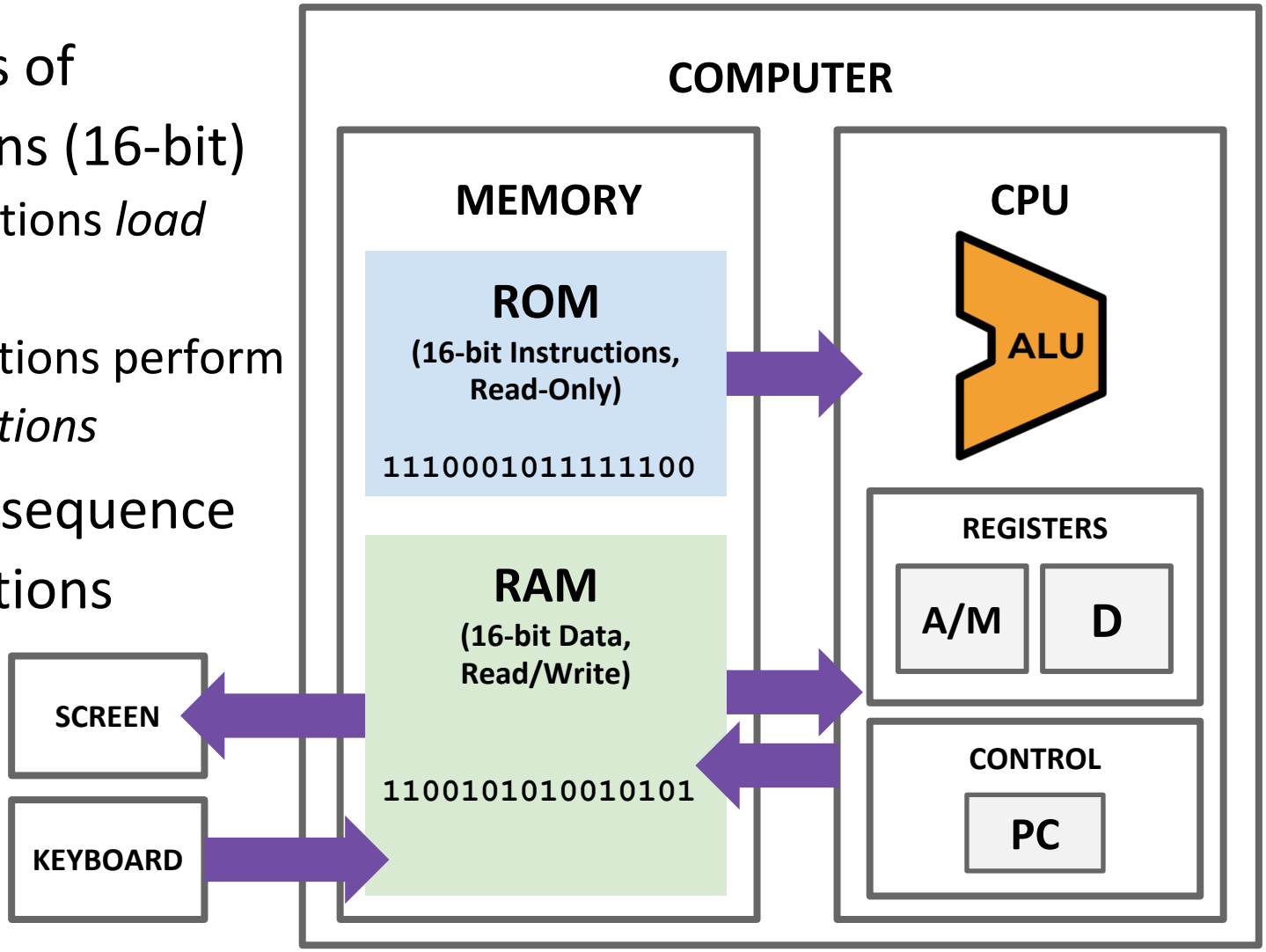
The Hack Computer

- The hardware you will build
  - 16-bit word size
  - ROM: sequence of instructions
    - `ROM[0], RAM[1]…`
  - RAM: data sequence
    - `RAM[0], RAM[1]…`



The Hack Machine Language

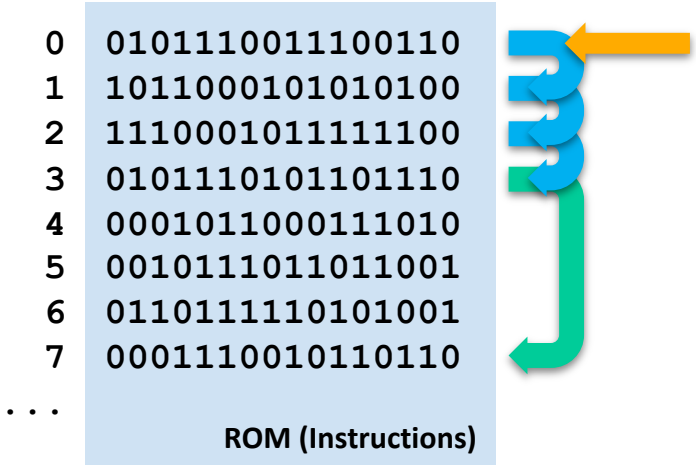
- Two types of instructions (16-bit)
  - A-instructions *load data*
  - C-instructions perform *computations*
- Program: sequence of instructions



Hack: Control Flow

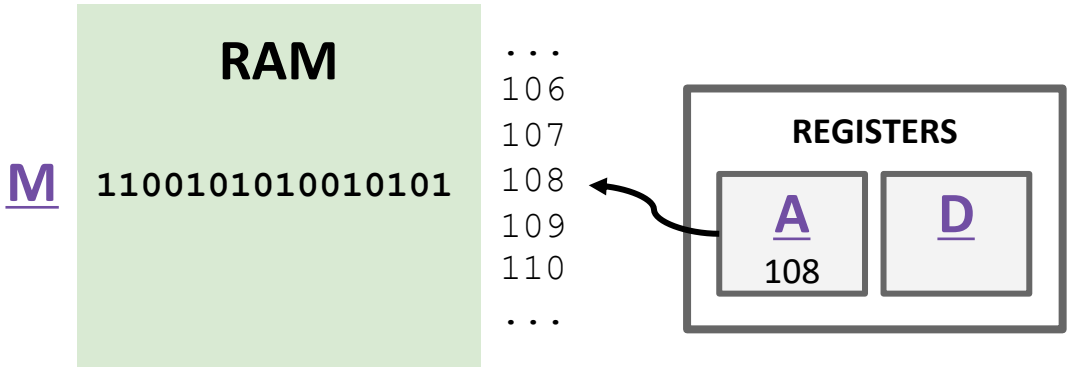
- Startup
  - Hack instructions loaded into ROM
  - Reset signal initializes computer state (instruction 0)

- Execution
  - Usually, advance to next instruction each cycle
  - On jump instruction, write a different address into the PC



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 non-negative decimal constant
  - A symbol referring to a constant

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

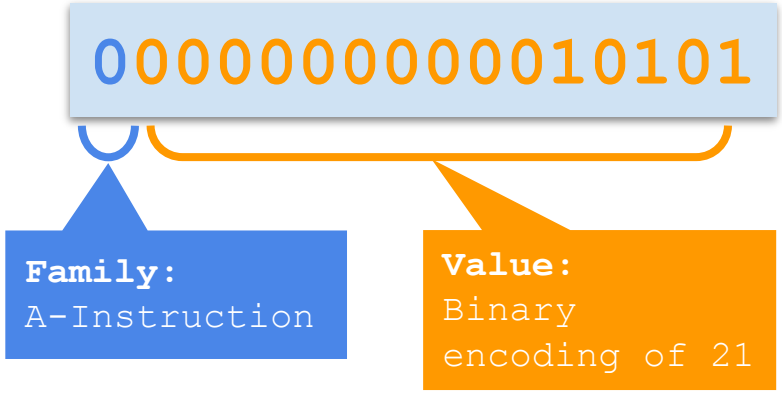
Hack: A-Instructions

- Symbolic Syntax

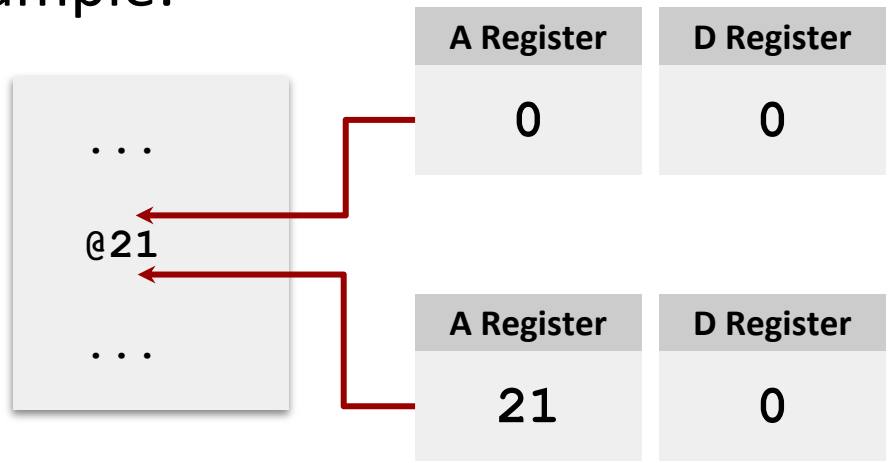
```
@value
```

  - Loads a value into the A register

- Binary Syntax



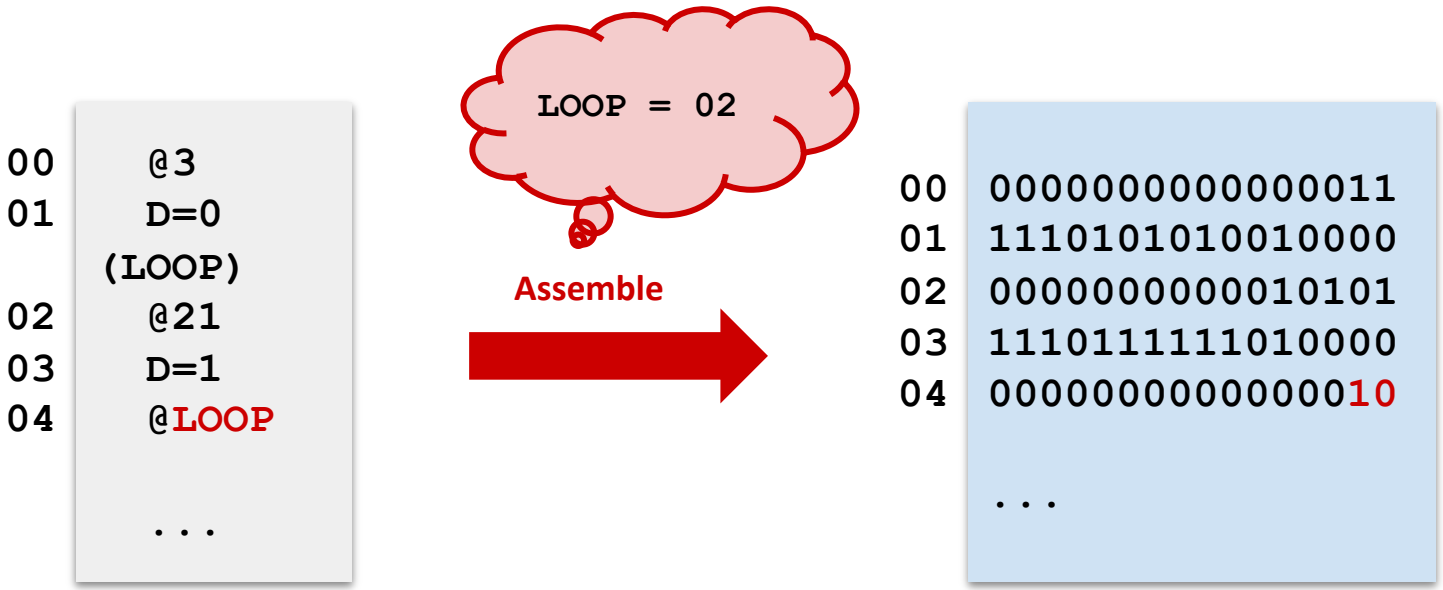
- Example:



Hack: Symbols

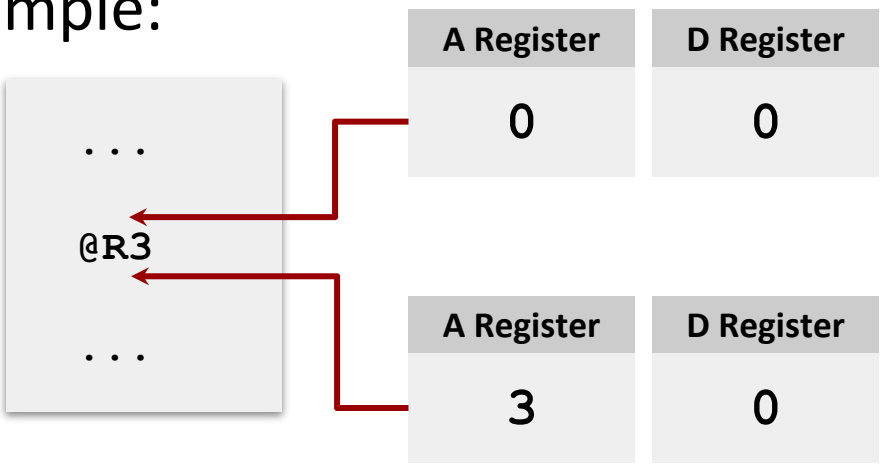
- Symbols are simply an alias for some address
  - Only in the symbolic code—don't turn into a binary instruction
  - Assembler converts use of that symbol to its value instead

- Example:



Hack: Built-In Symbols

- Using `(  )` defines a symbol in ROM / Instructions
- Assembler knows a few built-in symbols in RAM / Data
- **`R0, R1, ..., R15`**: Correspond to addresses at the very beginning of RAM (0, 1, …, 15)
  - "Virtual registers," Useful to store variables
- **`SCREEN, KBD`**: Base of I/O Memory Maps
- Example:



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
    ```

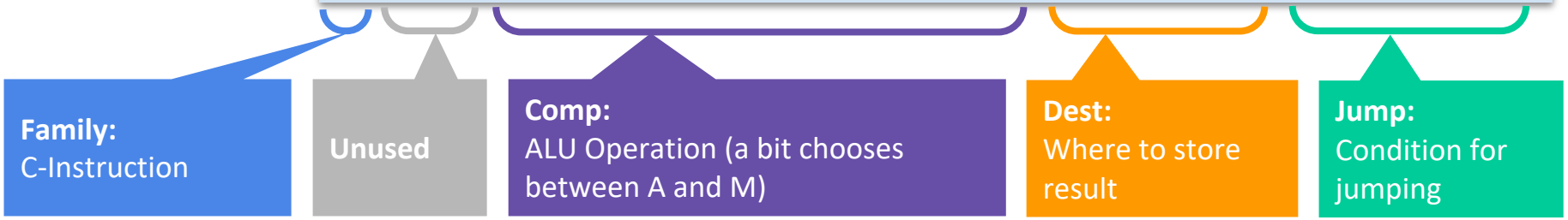
  - **`jump`** is an unconditional or conditional jump:

    ```
    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)

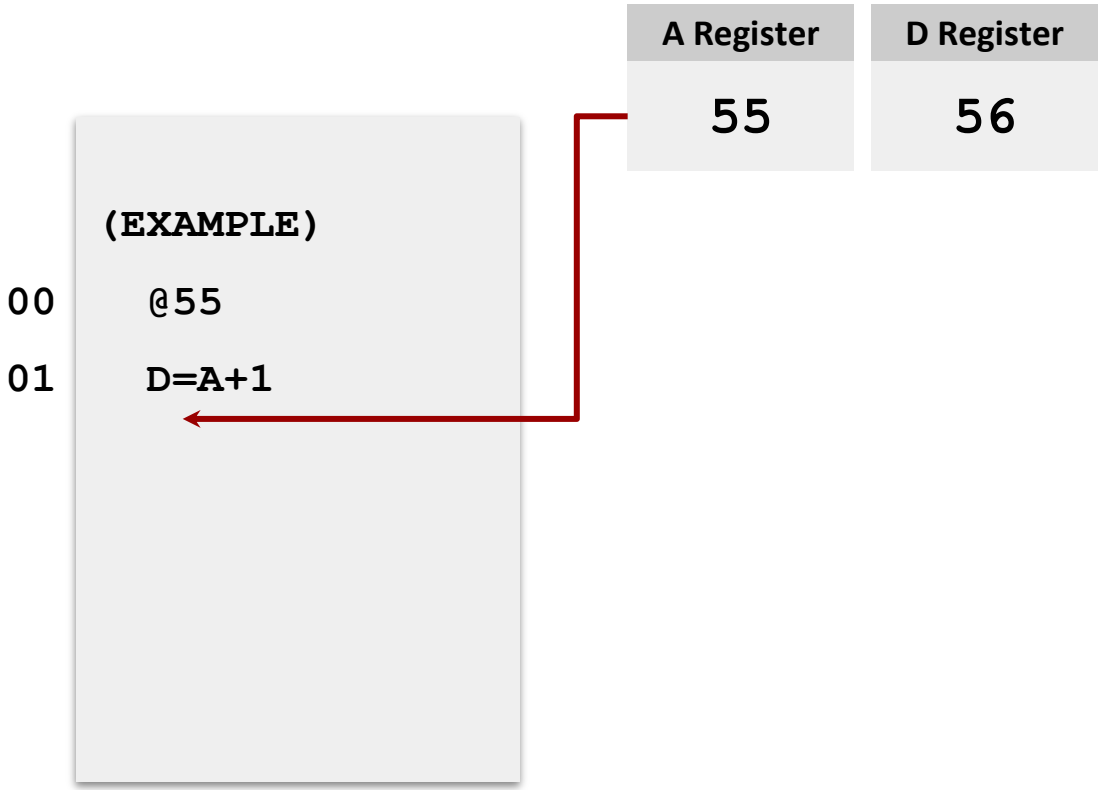
(when a=0) *comp mnemonic*	c1	c2	c3	c4	c5	c6	(when a=1) *comp mnemonic*
0	1	0	1	0	1	0	
1	1	1	1	1	1	1	
-1	1	1	1	0	1	0	
D	0	0	1	1	0	0	
A	1	1	0	0	0	0	M
!D	0	0	1	1	0	1	
!A	1	1	0	0	0	1	!M
-D	0	0	1	1	1	1	
-A	1	1	0	0	1	1	-M
D+1	0	1	1	1	1	1	
A+1	1	1	0	1	1	1	M+1
D-1	0	0	1	1	1	0	
A-1	1	1	0	0	1	0	M-1
D+A	0	0	0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

Chapter 4

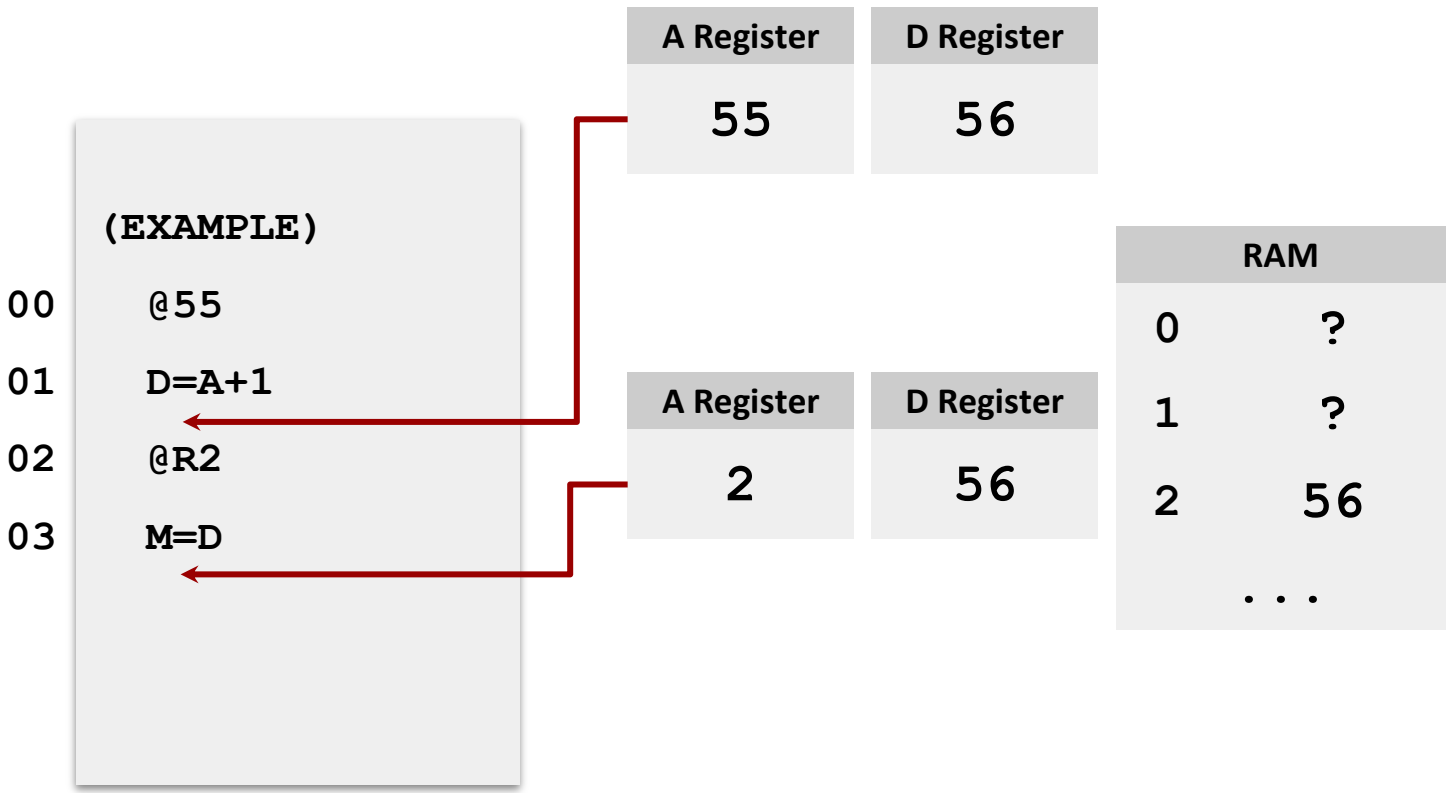
Important: just pattern matching text!

Cannot have "**1+M**"

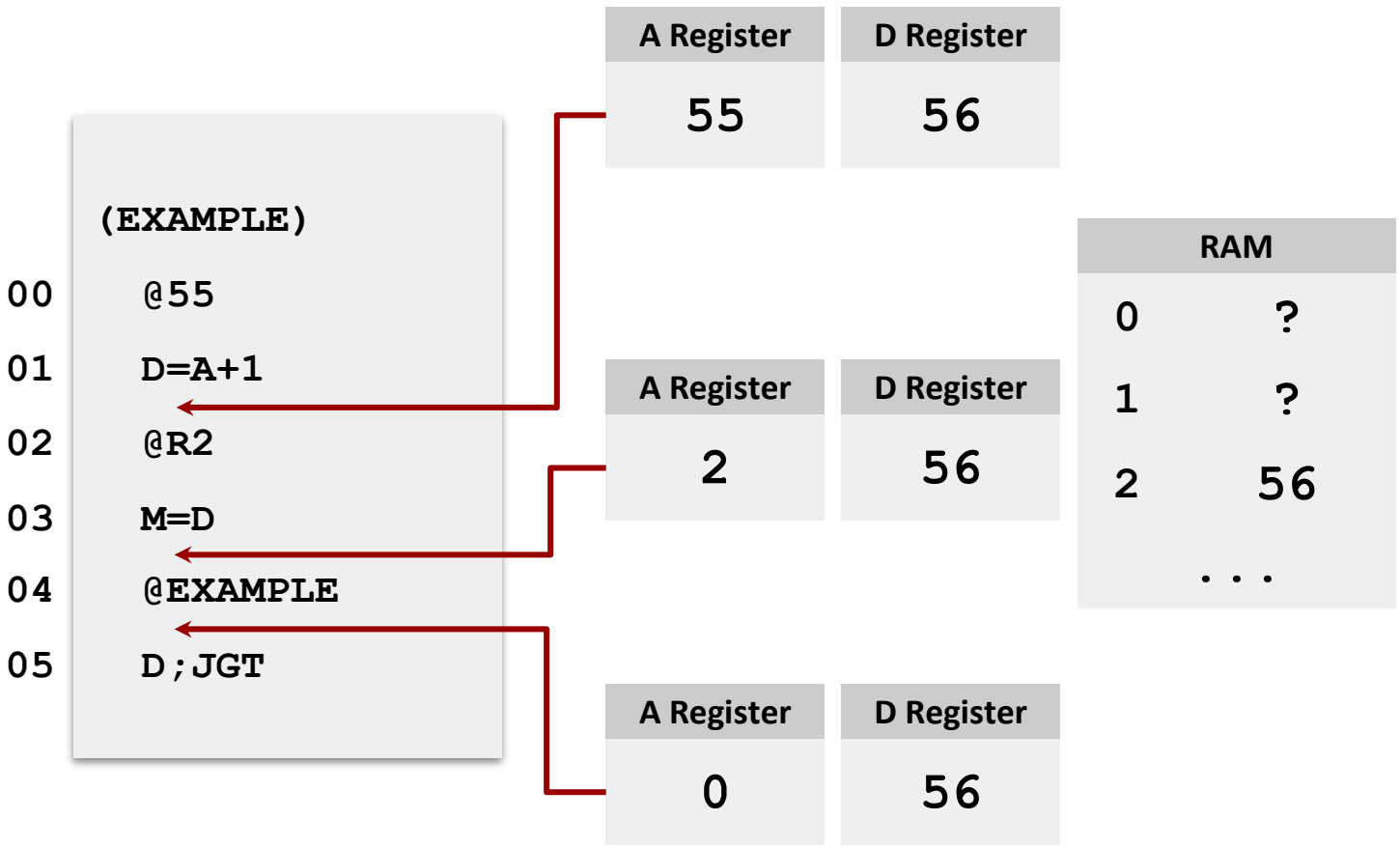
Hack: C-Instructions Example



Hack: C-Instructions Example



Hack: C-Instructions Example



(Will jump to instruction 0, since D > 0)

Lecture 7 Reminders

- Lectures will be in CSE2 271 starting this Friday until the rest of the quarter
- **Project 4 due this Friday (1/26) at 11:59pm**
- Amy has office hours tomorrow at 1:30pm in CSE2 151
  - Feel free to post your questions on the Ed board as well
- Midterm exam coming up in around two weeks (2/9) during lecture time
  - More details to come, along with metacognitive strategies for preparing for exams