CSE 390 B Spring 2021

Machine Languages & Annotation

Annotation Strategies, Machine Language Considerations, Hack Assembly Language & Project 4 Tools Practice

Agenda

- Check-In Survey
- 24-Hour Time Audit Debrief
- Annotation Strategies
- Reading Review and Q&A
- Hack Assembly Language

Agenda

- **Check-In Survey**
- 24-Hour Time Audit Debrief
- Annotation Strategies
- Reading Review and Q&A
- Hack Assembly Language

pollev.com/cse390b

- Check-in survey related to how the lecture readings and time spent in lecture is going
  - Specifically for the technical portions of lecture
- Completely anonymous!!!
  - It may ask you to log in but it won't record your info
- A few multiple choice questions and an open-feedback section at the end

Agenda

- Check-In Survey
- **24-Hour Time Audit Debrief**
- Annotation Strategies
- Reading Review and Q&A
- Hack Assembly Language

24-Hour Time Audit Debrief

- What was the most surprising to you about your day?
- # of hours you spent sleeping…
- # of hours you spend sitting…
- What is one thing that you want to change in how you spend your time? How will you implement this change?

Agenda

- Check-In Survey
- 24-Hour Time Audit Debrief
- **Annotation Strategies**
- Reading Review and Q&A
- Hack Assembly Language

Annotating Your Texts

- **WHAT**
  Intentionality of interacting with a text to enhance the reader's understanding of, recall of, and reaction to the text

Annotating Your Texts

- **WHAT**

  Intentionality of interacting with a text to enhance the reader's understanding of, recall of, and reaction to the text

- **HOW**
  - Highlighting, underlining or using [brackets] to note key points or ideas

Annotating Your Texts

WHAT

Intentionality of interacting with a text to enhance the reader's understanding of, recall of, and reaction to the text

HOW

- Highlighting, underlining or using [brackets] to note key points or ideas
- Circling unfamiliar words or confusing parts of the text

Annotating Your Texts

WHAT

Intentionality of interacting with a text to enhance the reader's understanding of, recall of, and reaction to the text

HOW

- Highlighting, underlining or using [brackets] to note key points or ideas
- Circling unfamiliar words or confusing parts of the text
- *Paraphrasing or summarizing passages/chapters/sections*

Annotating Your Texts

WHAT

Intentionality of interacting with a text to enhance the reader's understanding of, recall of, and reaction to the text

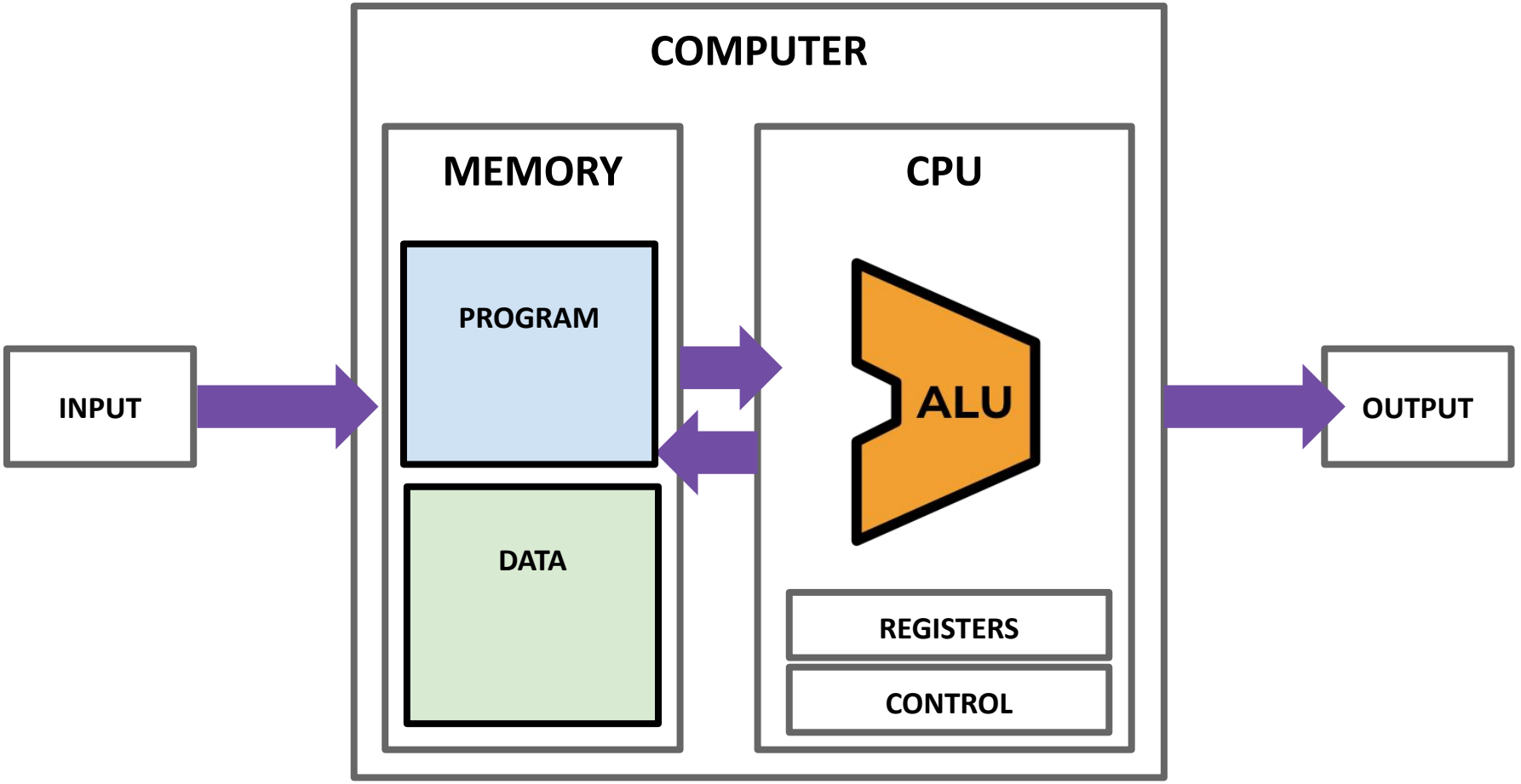
HOW

- Highlighting, underlining or using [brackets] to note key points or ideas
- Circling unfamiliar words or confusing parts of the text
- *Paraphrasing or summarizing passages or chapters*
- Commenting or reacting to the text 🤯

Agenda

- Check-In Survey
- 24-Hour Time Audit Debrief
- Annotation Strategies
- **Reading Review and Q&A**
- Hack Assembly Language

Revisiting: The Von Neumann Architecture

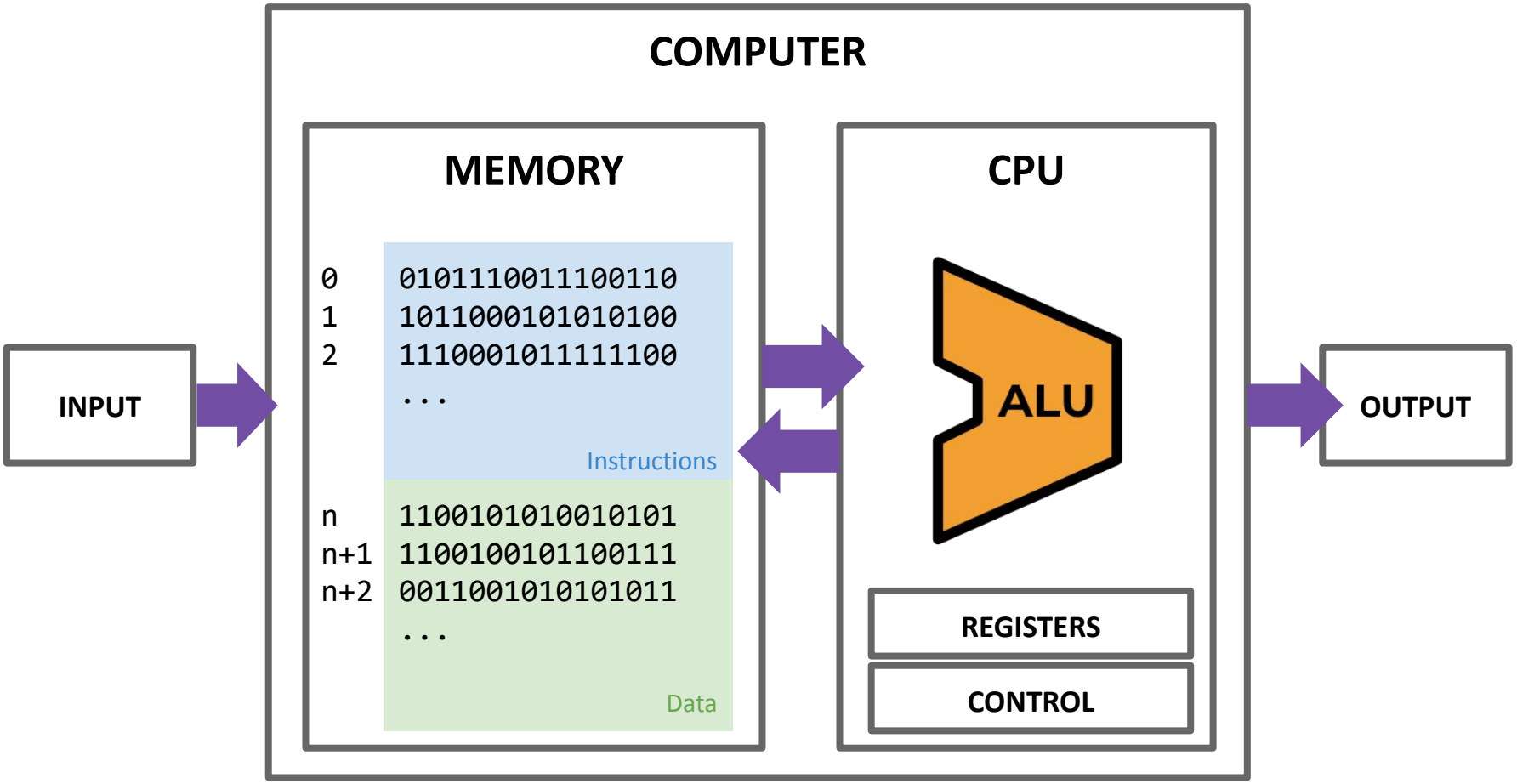


(This picture will get more detailed as we go!)

Machine Code

- Instructions are stored in memory, so they have to be able to be encoded in binary
- When we refer to **machine code** we typically are talking about this binary representation of code
- Each instruction ends up being a sequence of 1s and 0s, 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, etc.
  - More on this later!

Storing the Program



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

Assembly Languages

- Writing code as a bunch of 1s and 0s seems tedious and error prone right?

- Assembly languages are a more human readable format of the binary instructions your CPU runs
  - Developed to help humans write programs!

- Each human-readable assembly instruction has a corresponding binary machine code instruction
  - `addq reg1, reg2 == 1011000101010100`

- Often assembly is used as an intermediary between a high level language (e.g. Java) and machine code

Producing Machine Code

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

Java

Compile

Compile

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

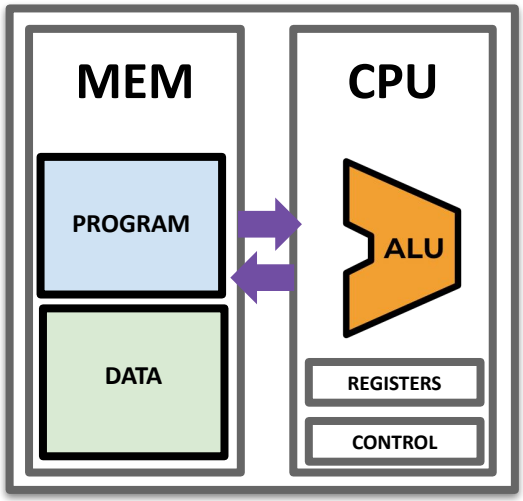
Assembly Language

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/Performance Tradeoffs
  - Silicon area/complexity
  - Time to complete instruction
  - Power consumption

Machine Operations

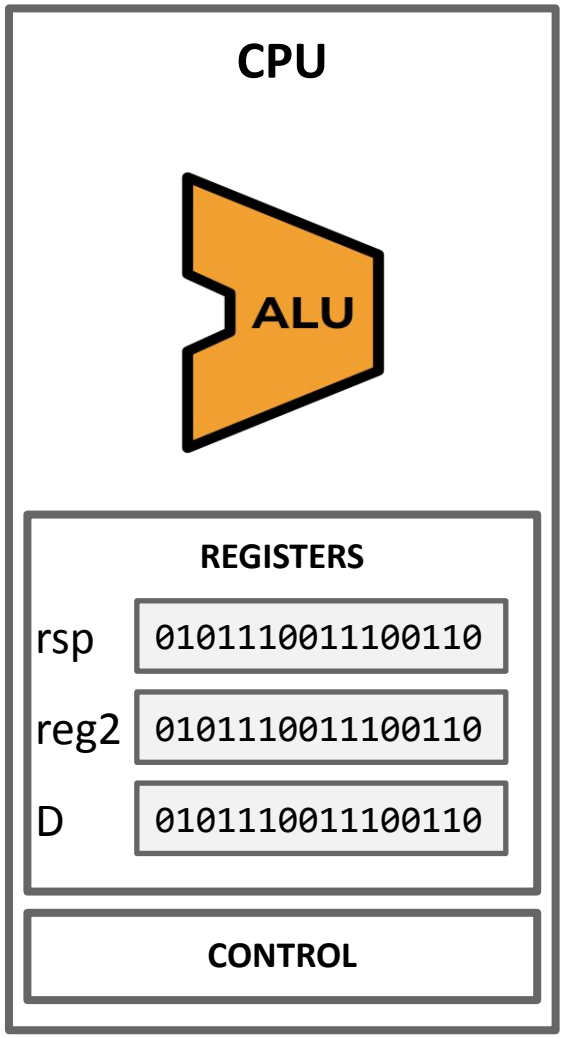
- Correspond to the operations supported by hardware:
  - Arithmetic (+, -)
  - Logical (And, Or)
  - Flow Control ("goto instruction *n*", "if (*condition*) then goto instruction *n*")

- 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



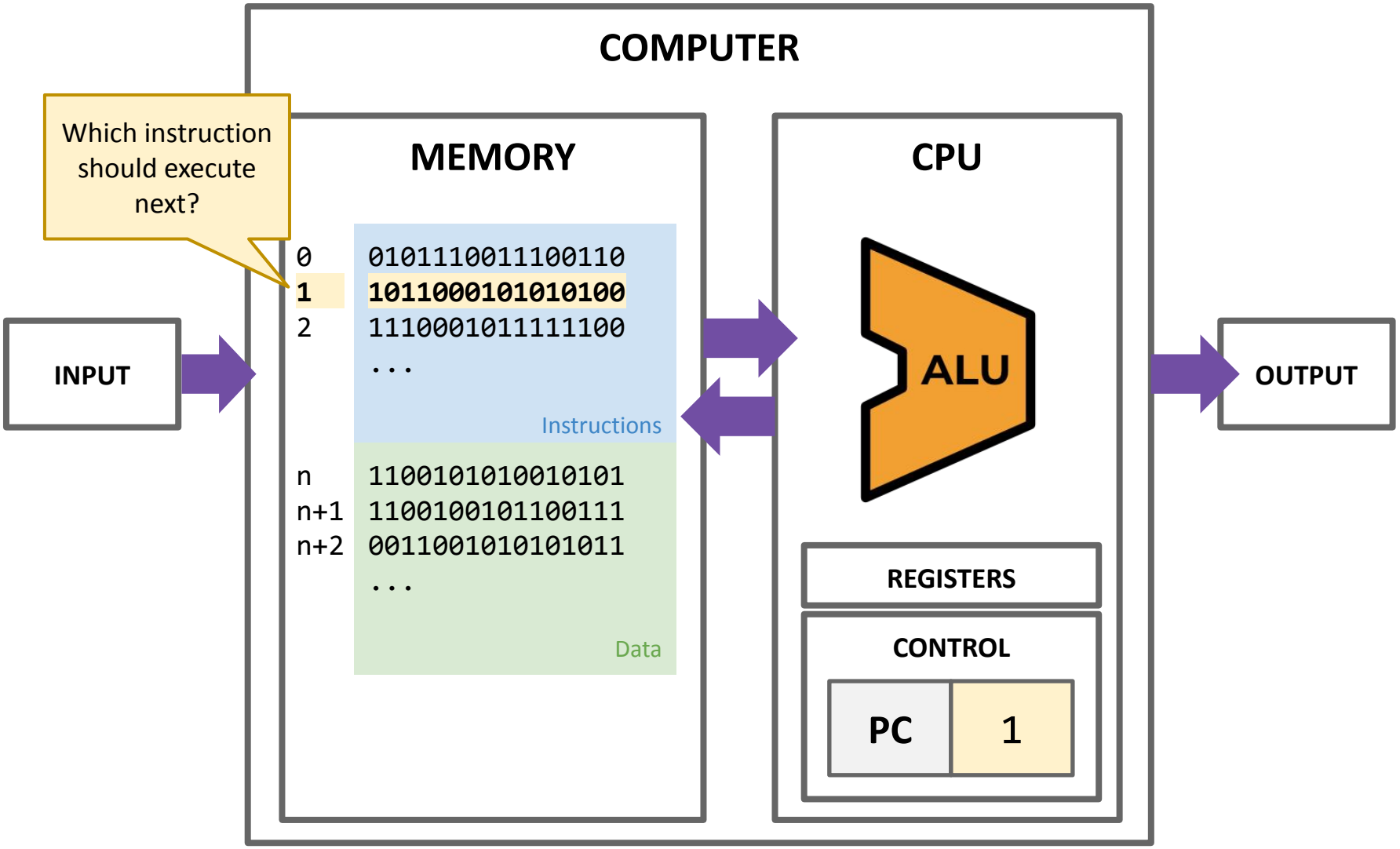
Addressing Modes

- A fancy way of saying 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`

`reg1, reg2` are the names of registers

`Memory[N]` means "access the giant array that is memory, at index N"

Flow Control



Flow Control

- 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" to another location no matter what
  - E.g. 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

- 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
  - E.g. to implement an if statement!
- We will focus more on this on Thursday!

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`

Reading Q&A

Agenda

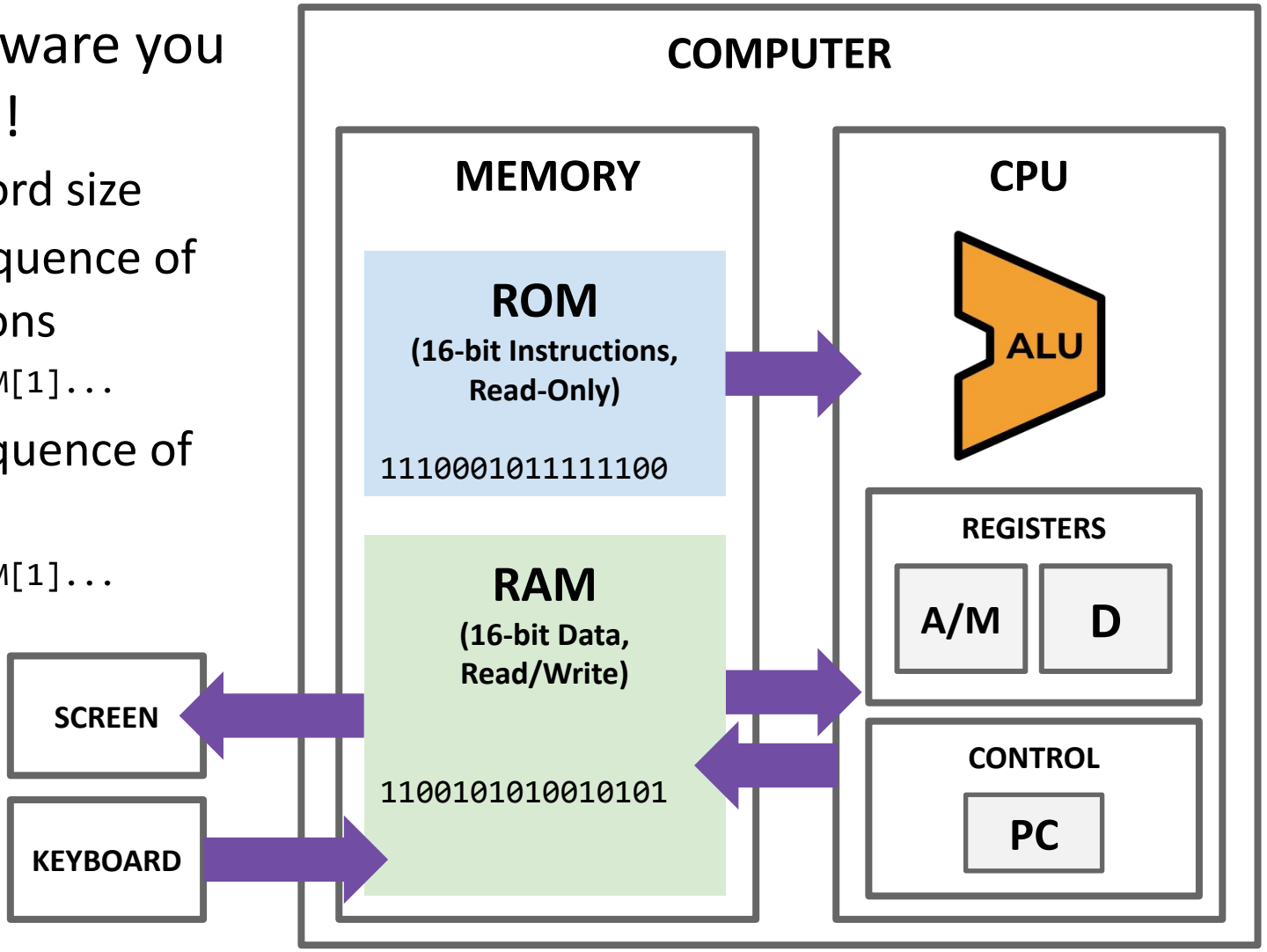
- Check-In Survey
- 24-Hour Time Audit Debrief
- Annotation Strategies
- Reading Review and Q&A
- **Hack Assembly Language**

The Hack Computer

- The hardware you will build!
  - 16-bit word size
  - ROM: sequence of instructions

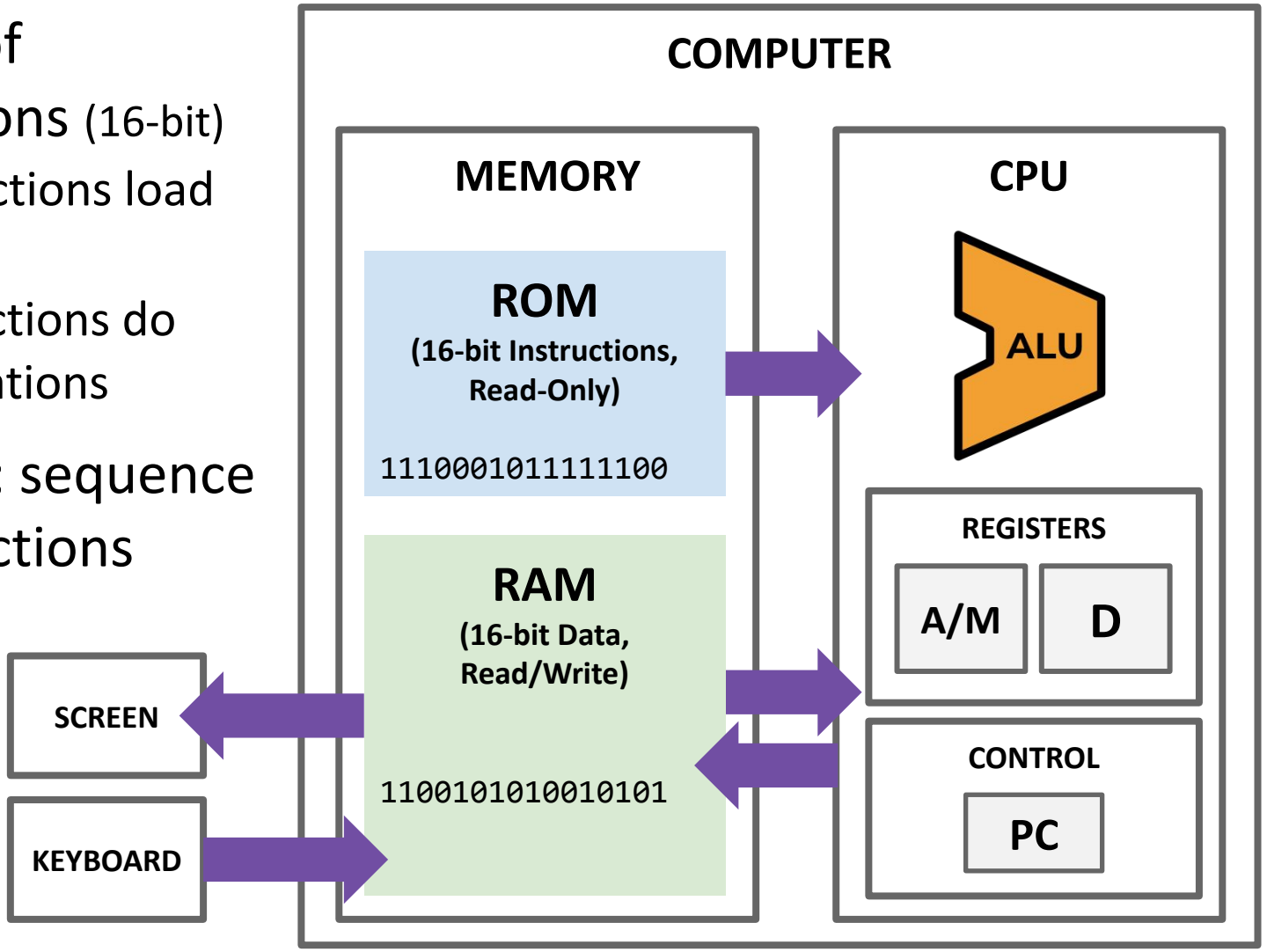
    `ROM[0], ROM[1]...`
  - RAM: sequence of data

    `RAM[0], RAM[1]...`



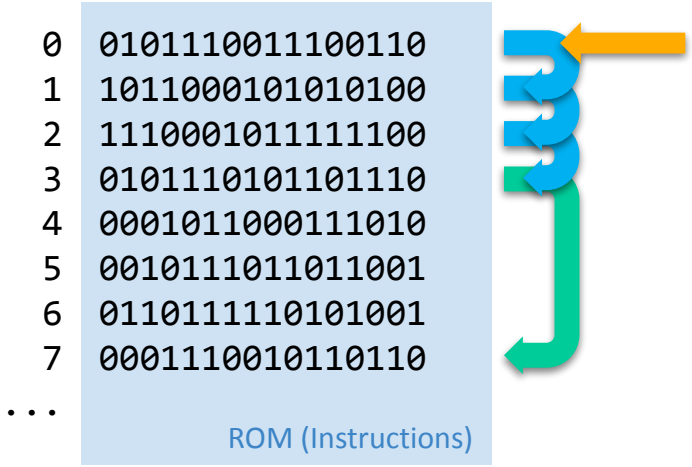
The Hack Machine Language

- 2 types of instructions (16-bit)
  - A-instructions load data
  - C-instructions do computations
- Program: sequence of instructions



Hack: Control Flow

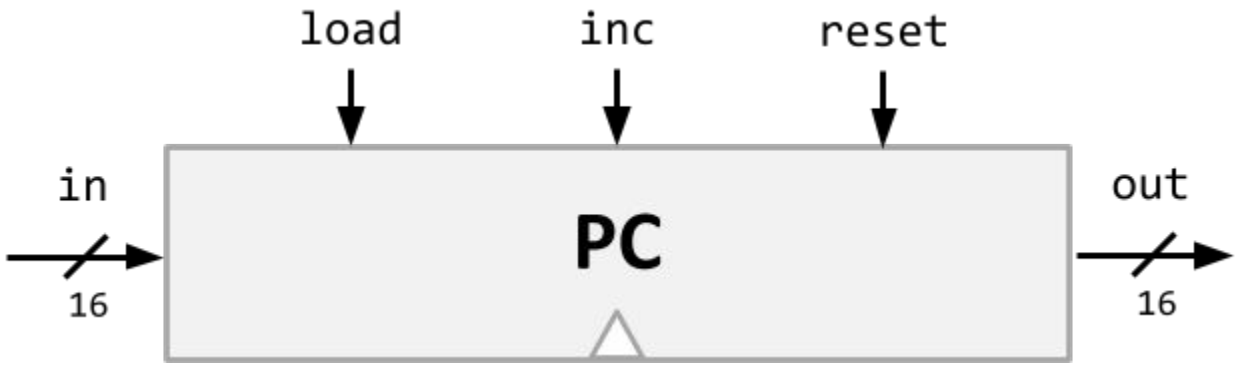
- Startup
  - Hack instructions loaded into ROM
  - Reset signal initializes computer state (instruction 0)
- Execution
  - Usually, advance to next instruction each cycle
  - On jump instruction, write a different address into the PC



Program Counter

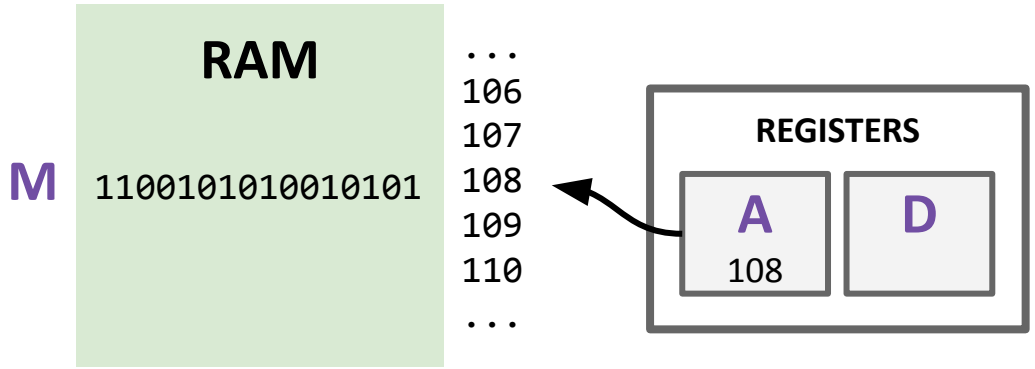
- Keeps track of what instruction we're executing
  - If it outputs 24, on the next clock cycle the computer runs the instruction at address 24 in the code segment
- Program counter interface:

```
if      (reset[t] == 1) out[t+1] = 0          // restart!
else if (load[t] == 1)  out[t+1] = in[t]      // jump!
else if (inc[t] == 1)   out[t+1] = out[t] + 1 // next!
else                    out[t+1] = out[t]
```



Hack: Registers

- **D Register**: For storing data
- **A Register**: For storing data AND addressing memory
- **M "Register"**: The 16-bit word of memory currently being referenced by the address in A



Hack: A-Instructions

Syntax: `@value`

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

Semantics:

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

Hack: A-Instructions

Symbolic Syntax:

```
@value
```

Simply loads a value into the A register

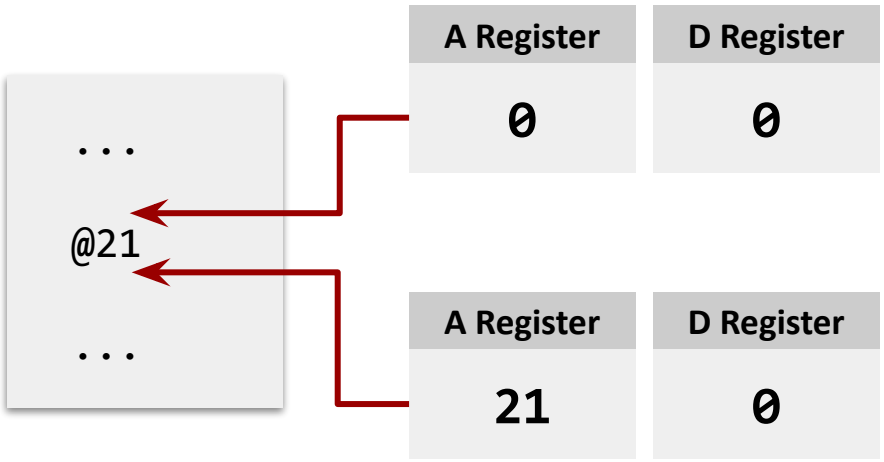
Binary Syntax:

```
0000000000010101
```

Family: A-Instruction

Value: Binary encoding of 21

Example:



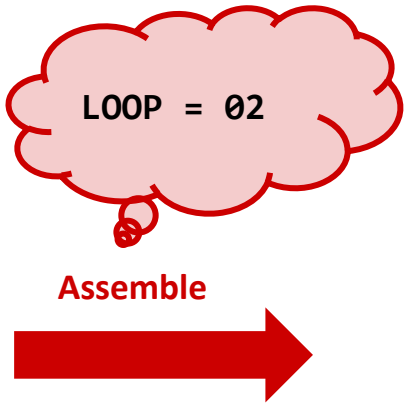
Hack: Symbols

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

Example:

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

      ...
```



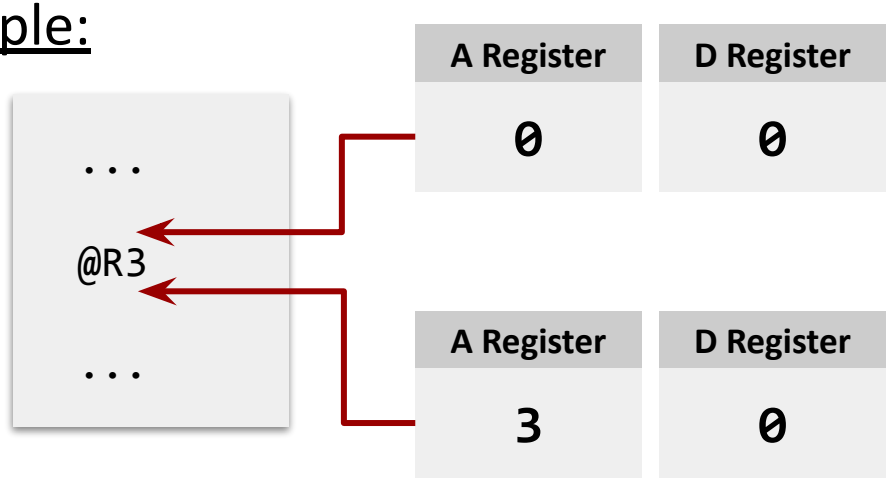

```
00  0000000000000011
01  1110101010010000
02  0000000000010101
03  1110111111010000
04  0000000000000010

...
```

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** or **jump** is 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]

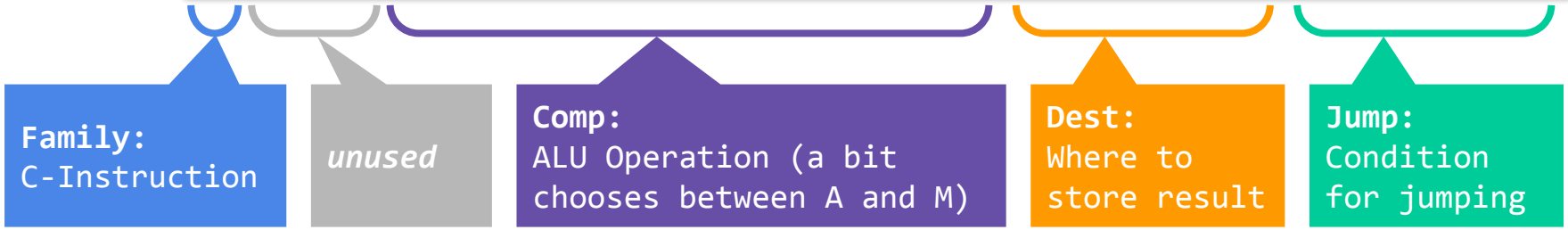
Hack: C-Instructions

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* ≥ 0 jump
1	0	0	JLT	If *out* < 0 jump
1	0	1	JNE	If *out* ≠ 0 jump
1	1	0	JLE	If *out* ≤ 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

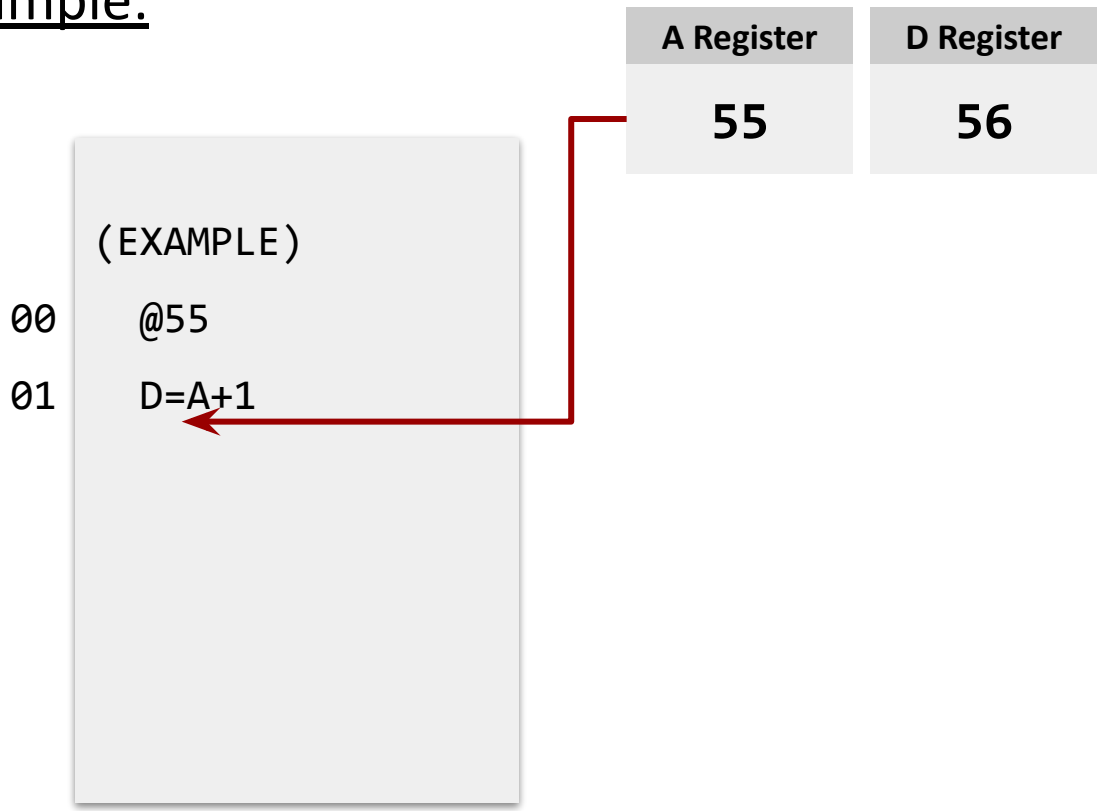
(when a=0) *comp* mnemonic	c1	c2	c3	c4	c5	c6	(when a=1) *comp* mnemonic
0	1	0	1	0	1	0	
1	1	1	1	1	1	1	
-1	1	1	1	0	1	0	
D	0	0	1	1	0	0	
A	1	1	0	0	0	0	M
!D	0	0	1	1	0	1	
!A	1	1	0	0	0	1	!M
-D	0	0	1	1	1	1	
-A	1	1	0	0	1	1	-M
D+1	0	1	1	1	1	1	
A+1	1	1	0	1	1	1	M+1
D-1	0	0	1	1	1	0	
A-1	1	1	0	0	1	0	M-1
D+A	0	0	0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

Can't do "**1+M**"

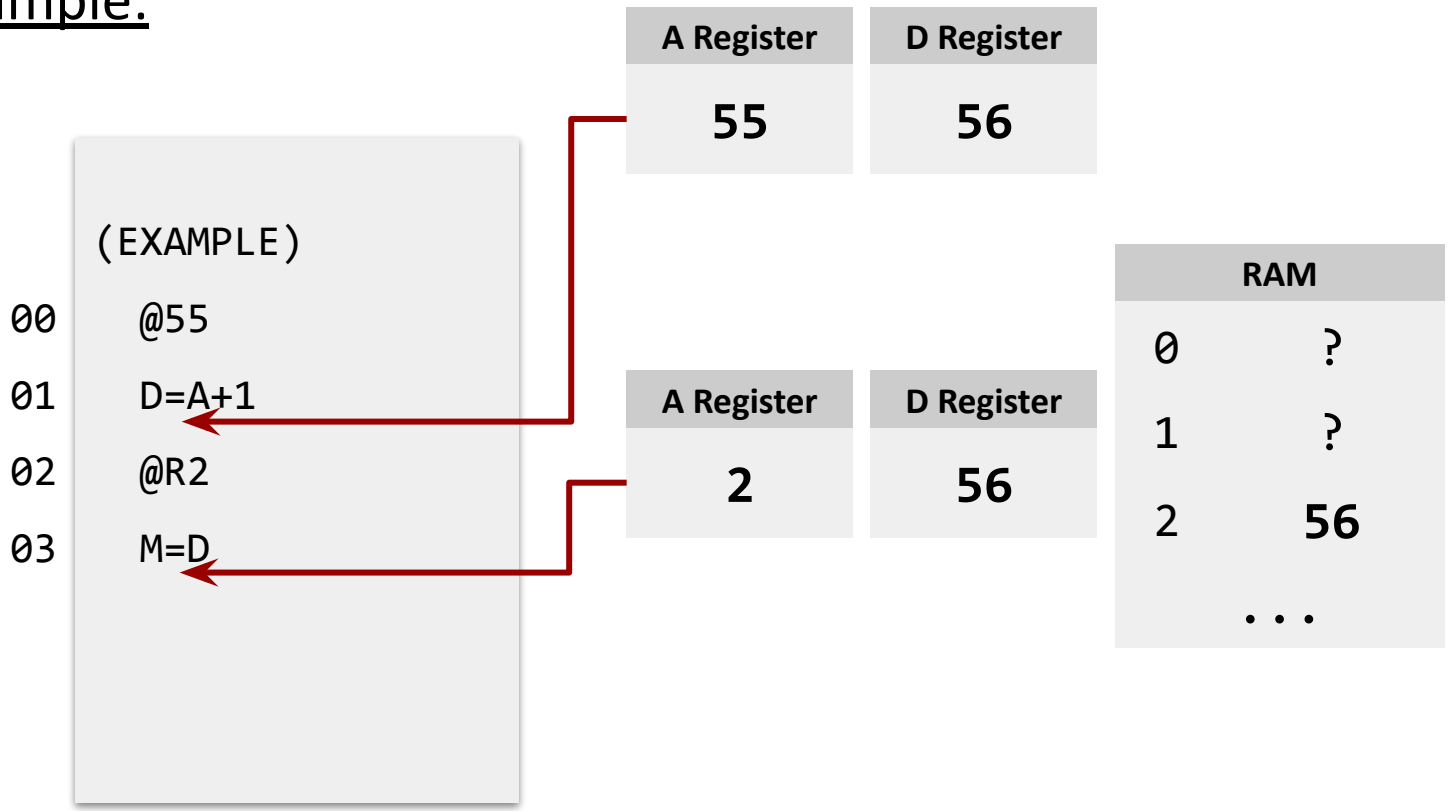
Hack: C-Instructions

Example:



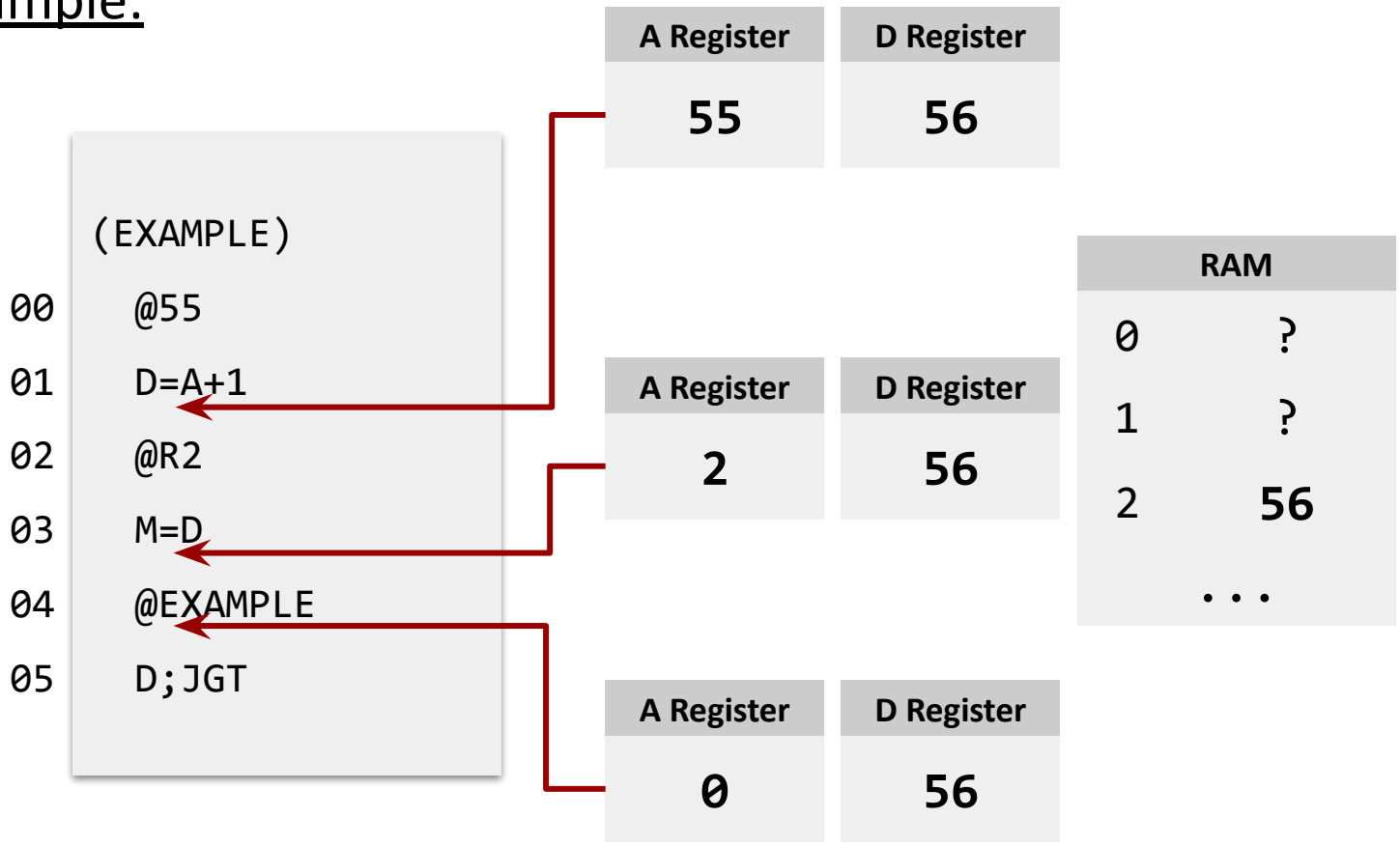
Hack: C-Instructions

Example:



Hack: C-Instructions

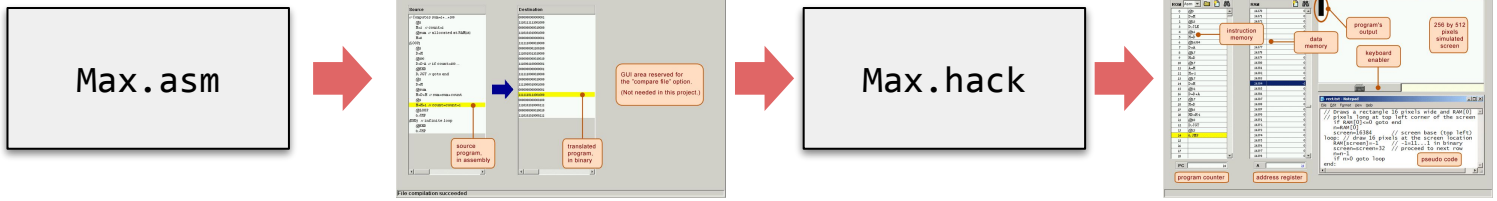
Example:



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

Project 4: Tools

- Running a Test Script (recommended flow):



The test scripts use the .hack files directly! Don't let your .asm and .hack get out of sync!

- Quickly Iterating or Experimenting:

Can still "run" the program, even without a script

Tools Exercise: Run SimpleAdd.tst

- In gitlab, a `projects/lecture/` folder was added with a SimpleAdd example
- Open up the assembly file SimpleAdd.asm. What does this program do?
- Assemble SimpleAdd.asm using the Assembler to produce SimpleAdd.hack
- Run SimpleAdd.tst using the CPUEmulator and verify there are no errors

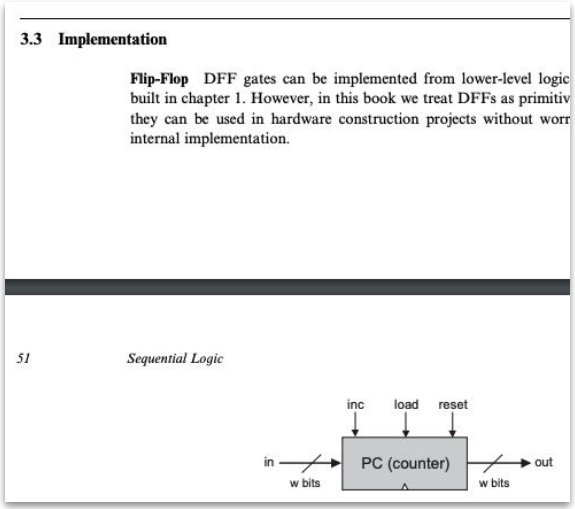
Reminders

- **Office Hours**
  - Eric & Margot's office hours happening right after class!

- **Project 3: Memory & Cornell Note-Taking**
  - Due Thursday 11:59PM PDT
  - Reminder of Project 3 Resources..

Reminder: Project 3 Resources

- Take advantage of hints in project 3 readings

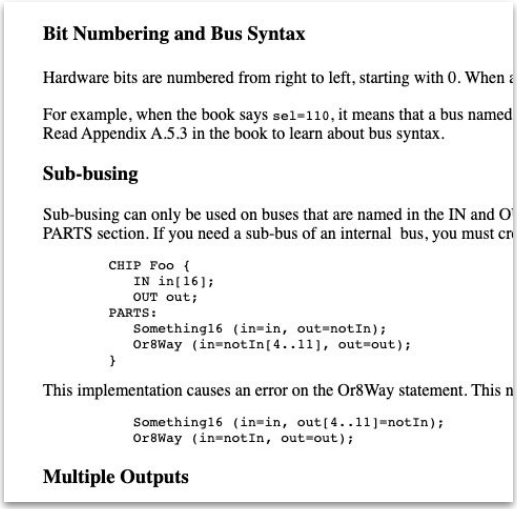


3.3 **Implementation**

Flip-Flop DFF gates can be implemented from lower-level logic built in chapter 1. However, in this book we treat DFFs as primitiv they can be used in hardware construction projects without worr internal implementation.

51 *Sequential Logic*

Chapter 3



Bit Numbering and Bus Syntax

Hardware bits are numbered from right to left, starting with 0. When a

For example, when the book says `sel=110`, it means that a bus named Read Appendix A.5.3 in the book to learn about bus syntax.

Sub-busing

Sub-busing can only be used on buses that are named in the IN and O PARTS section. If you need a sub-bus of an internal bus, you must cr

```
CHIP Foo {
   IN in[16];
   OUT out;
PARTS:
   Something16 (in=in, out=notIn);
   Or8Way (in=notIn[4..11], out=out);
}
```

This implementation causes an error on the Or8Way statement. This n

```
   Something16 (in=in, out[4..11]=notIn);
   Or8Way (in=notIn, out=out);
```

Multiple Outputs

HDL Survival Guide

- Reach out to your classmates and course staff
  - **Post on discussion board**: https://us.edstem.org
  - **Come to office hours!**