CSE 390B, Winter 2022

Building Academic Success Through Bottom-Up Computing

More Hack Assembly, Project 4 Overview

Hack Assembly Language, Hack Memory Representation, Project 4 Overview

If joining virtually, please have your camera turned on if you can!

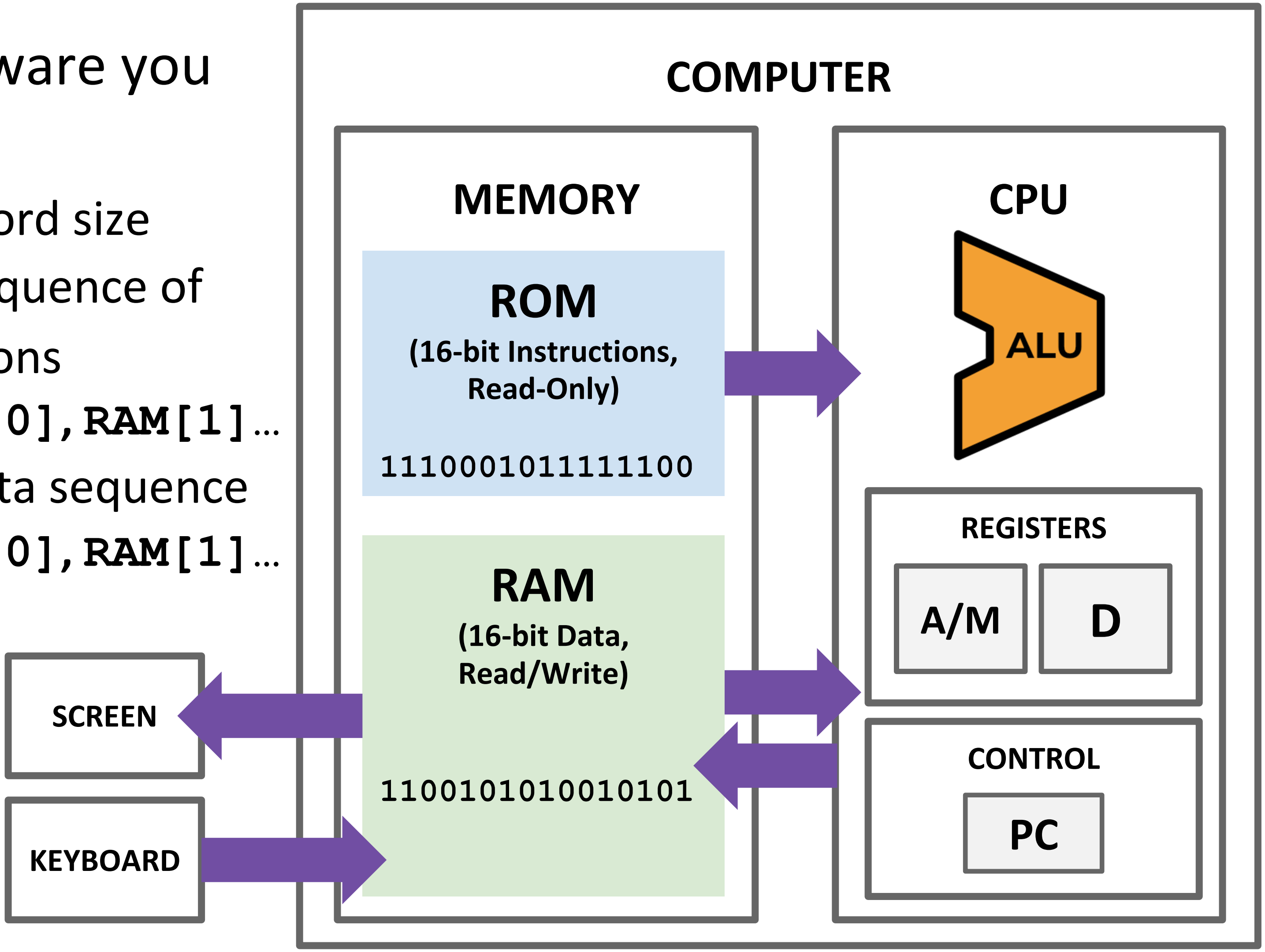
UNIVERSITY of WASHINGTON

Lecture Outline

- **Hack Assembly Language**
  - **Registers, A-Instructions, Symbols, C-Instructions**
- Hack Assembly Memory Representation
  - Input / Output, Memory Mapping, External / Internal Memory
- Multiplication Implementation Exercise
  - How do we multiply two numbers in the Hack Assembly language?
- Project 4: Machine Language and Annotation Overview
  - Annotation, Assembly Language, Building a Computer Part I, Mid-quarter Reflection

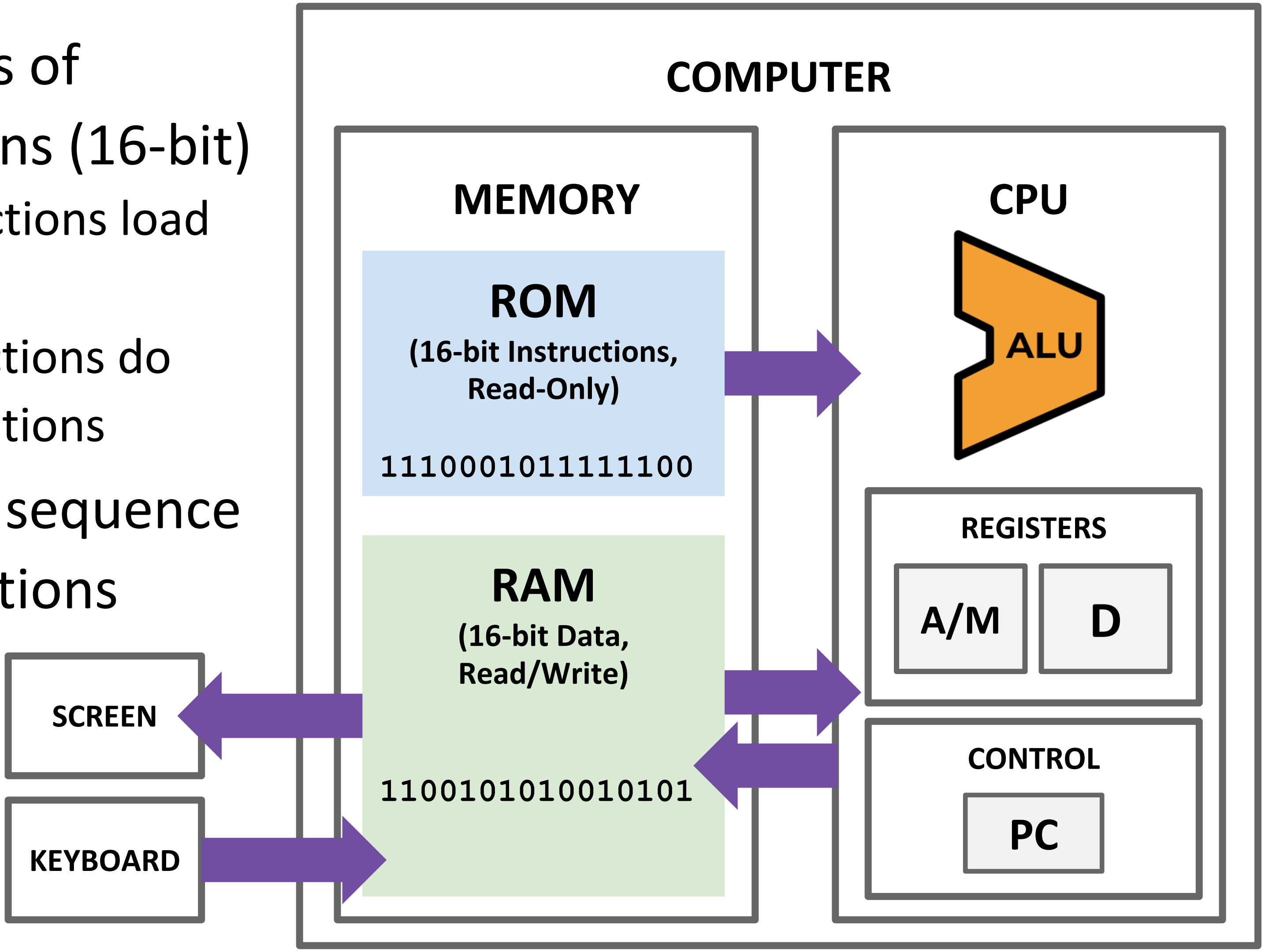
The Hack Computer

- 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

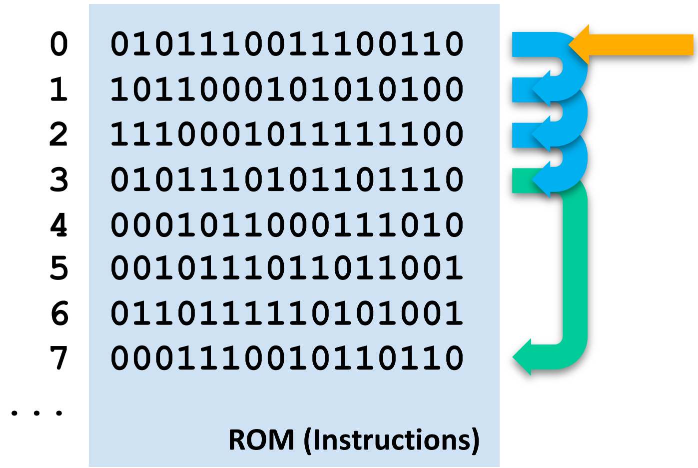
- Two types of instructions (16-bit)
  - A-instructions load data
  - C-instructions do computations
- Program: sequence of instructions



Hack: Control Flow

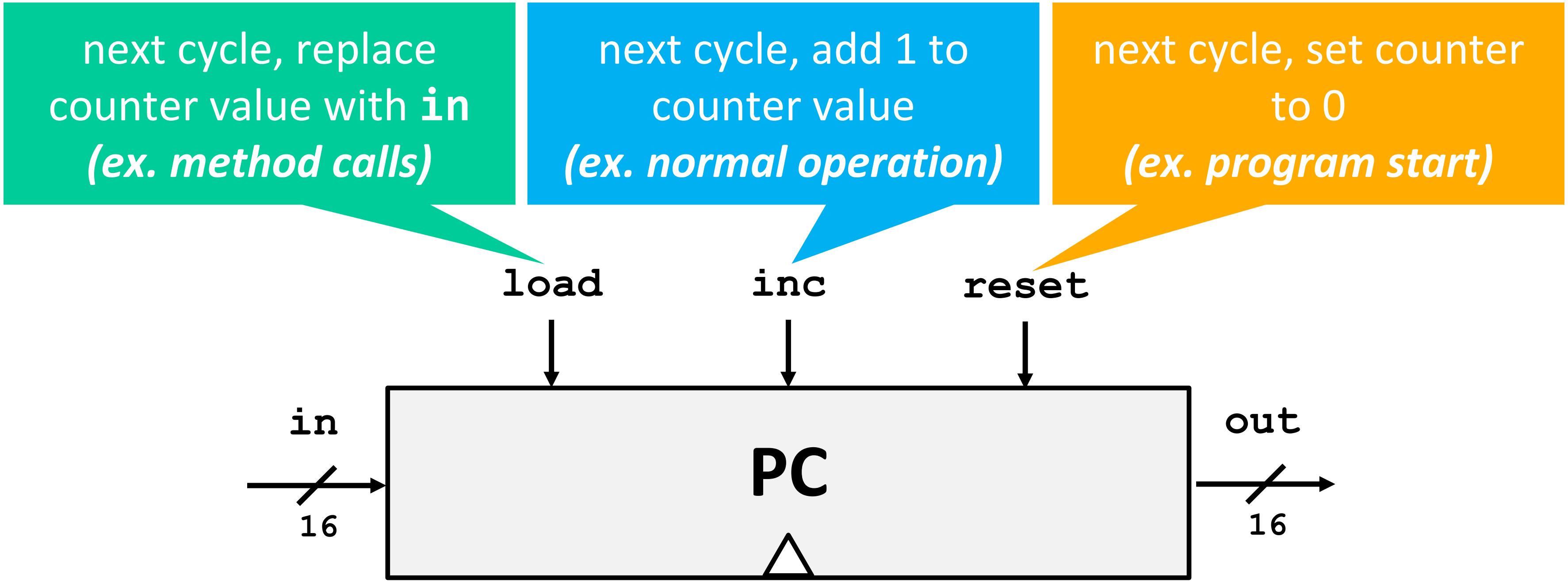
- Startup
  - Hack instructions loaded into ROM
  - Reset signal initializes computer state (instruction 0)

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



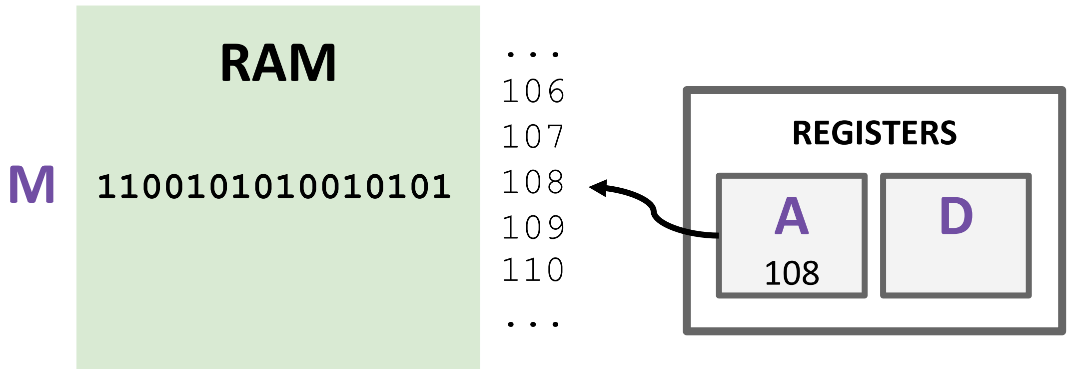
Program Counter (PC)

- 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



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

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

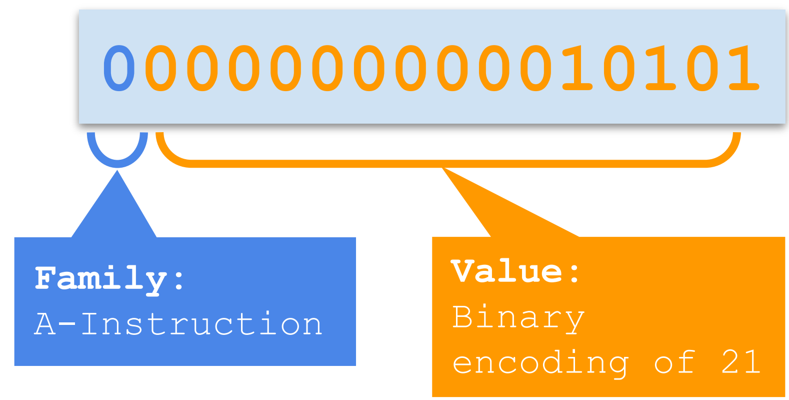
Hack: A-Instructions

- Symbolic Syntax

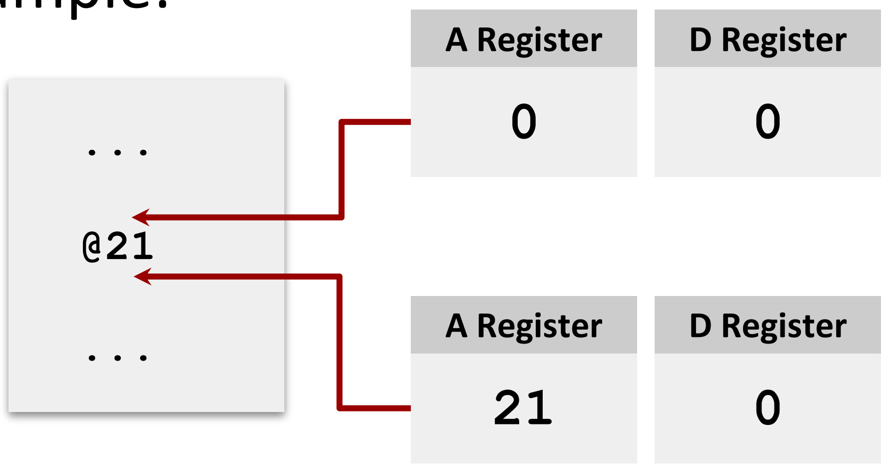
```
@value
```

  - Loads a value into the A register

- Binary Syntax



- Example:



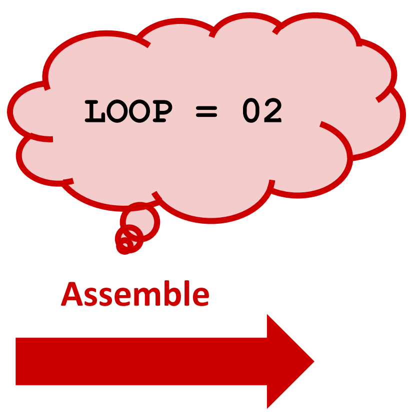
Hack: Symbols

- Symbols are simply an <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

       ...
```

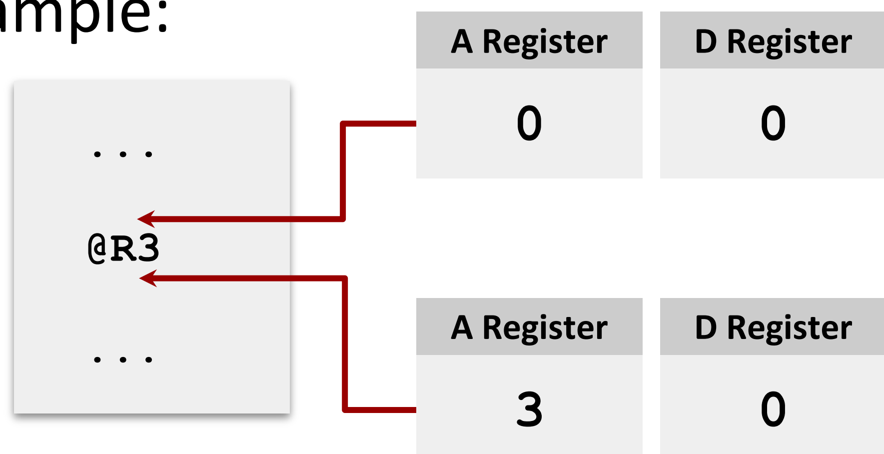



```
00  0000000000000011
01  1110101010010000
02  0000000000010101
03  1110111111010000
04  0000000000000010

    ...
```

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 and jump are 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`

Family: C-Instruction

Unused

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

Dest: Where to store result

Jump: Condition for jumping

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)

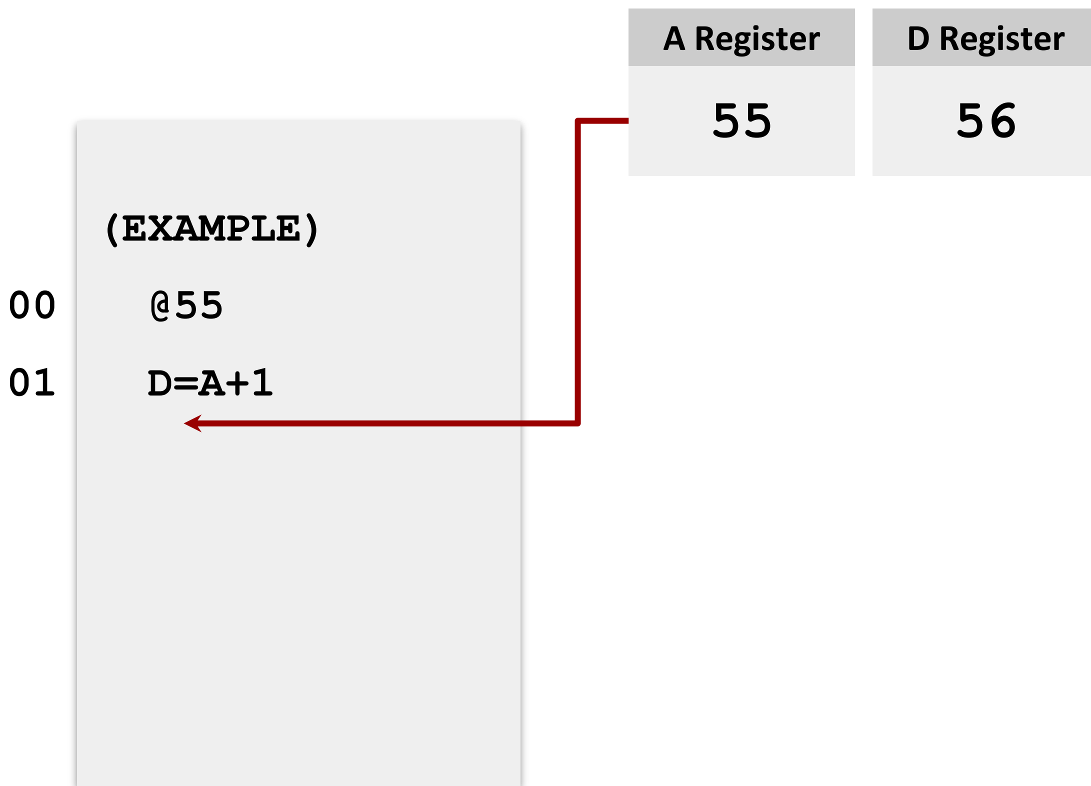
Chapter 4

(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

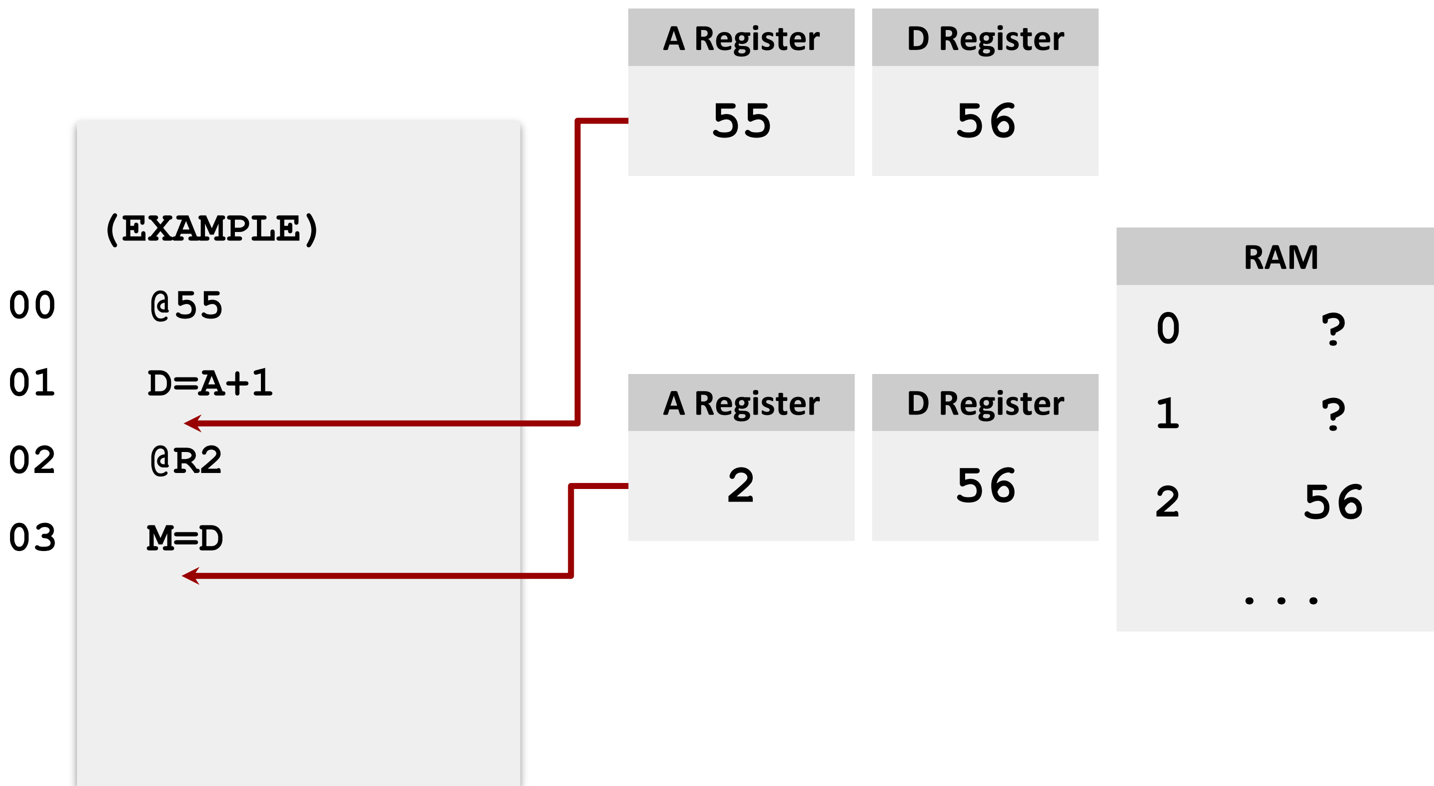
Important: just pattern matching text!

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

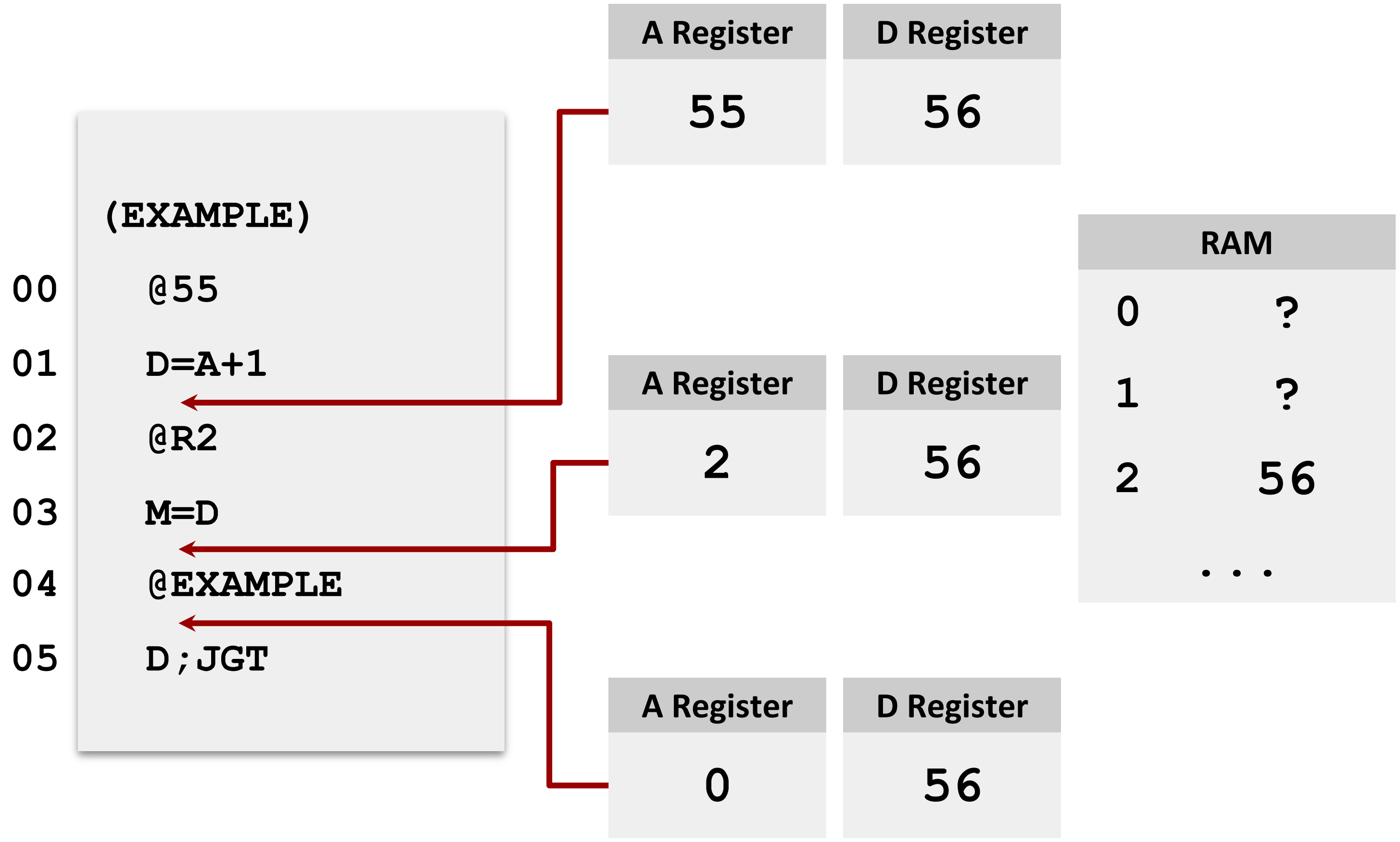
Hack: C-Instructions Example



Hack: C-Instructions



Hack: C-Instructions



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

Lecture Outline

- Hack Assembly Language
  - Registers, A-Instructions, Symbols, C-Instructions
- **Hack Assembly Memory Representation**
  - **Input / Output, Memory Mapping, External / Internal Memory**
- Multiplication Implementation Exercise
  - How do we multiply two numbers in the Hack Assembly language?
- Project 4: Machine Language and Annotation Overview
  - Annotation, Assembly Language, Building a Computer Part I, Mid-quarter Reflection

Lecture 3 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

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 $0b\ 0011\ 1010$
  - $0x3 == 0b0011$
  - $0xA == 0b1010$

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

Hack Assembly: Input/Output

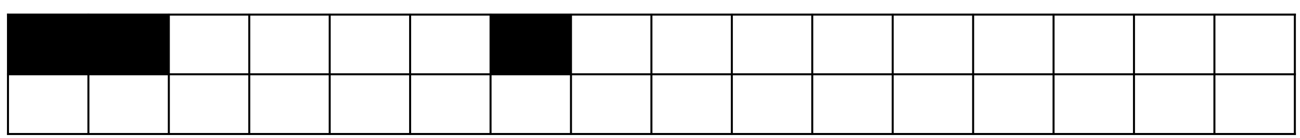
- Two memory maps are created for you by underlying hardware (all you have to do is use them)
  - Screen is a huge map where each pixel is one bit
  - Keyboard is a single 16-bit word map with code of current key
- Example:

```
@KBD
D=M

@SCREEN
M=D
```

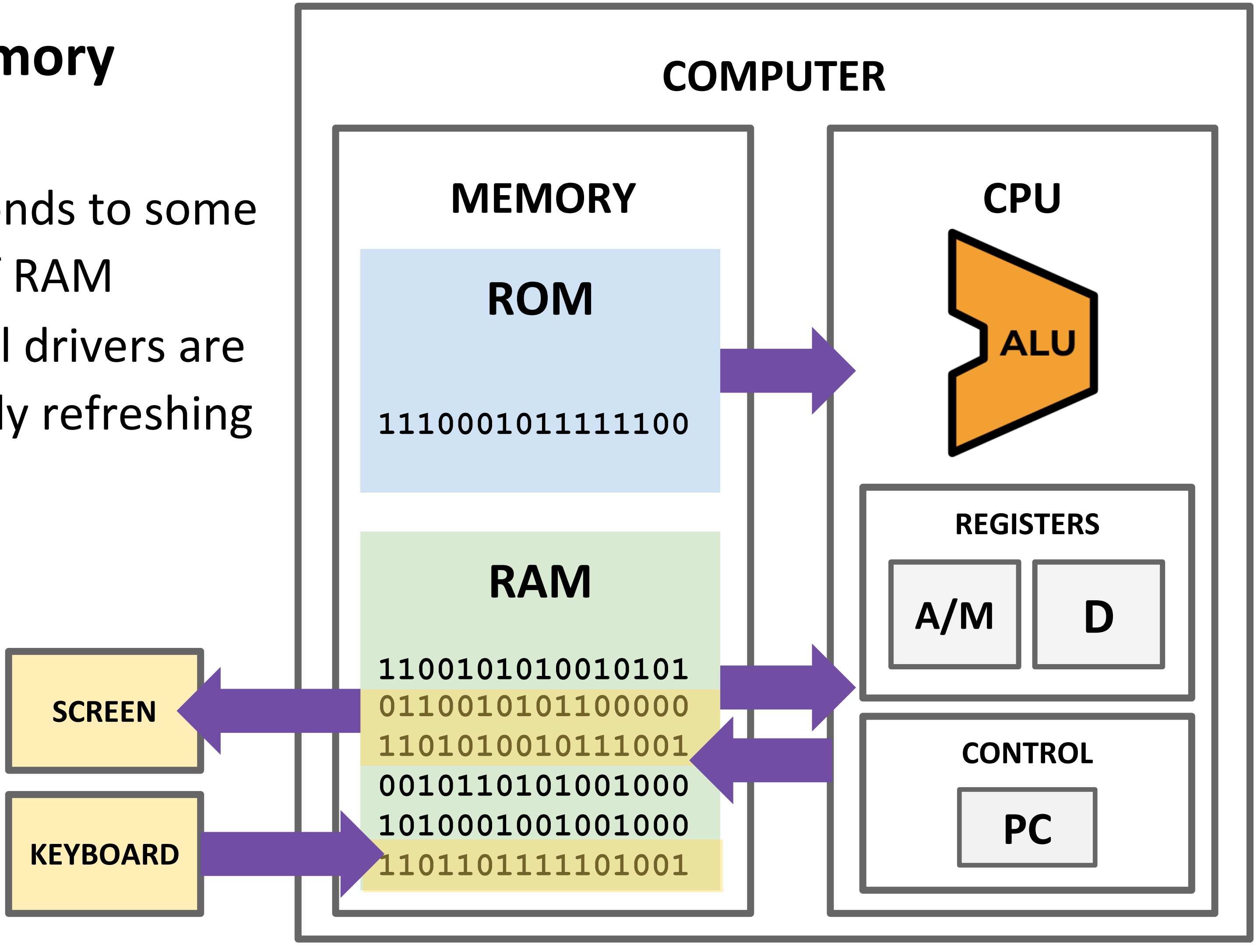
D contains code of current key (e.g. 67 for "C")

First 16 bits of screen (top left) show binary for 67



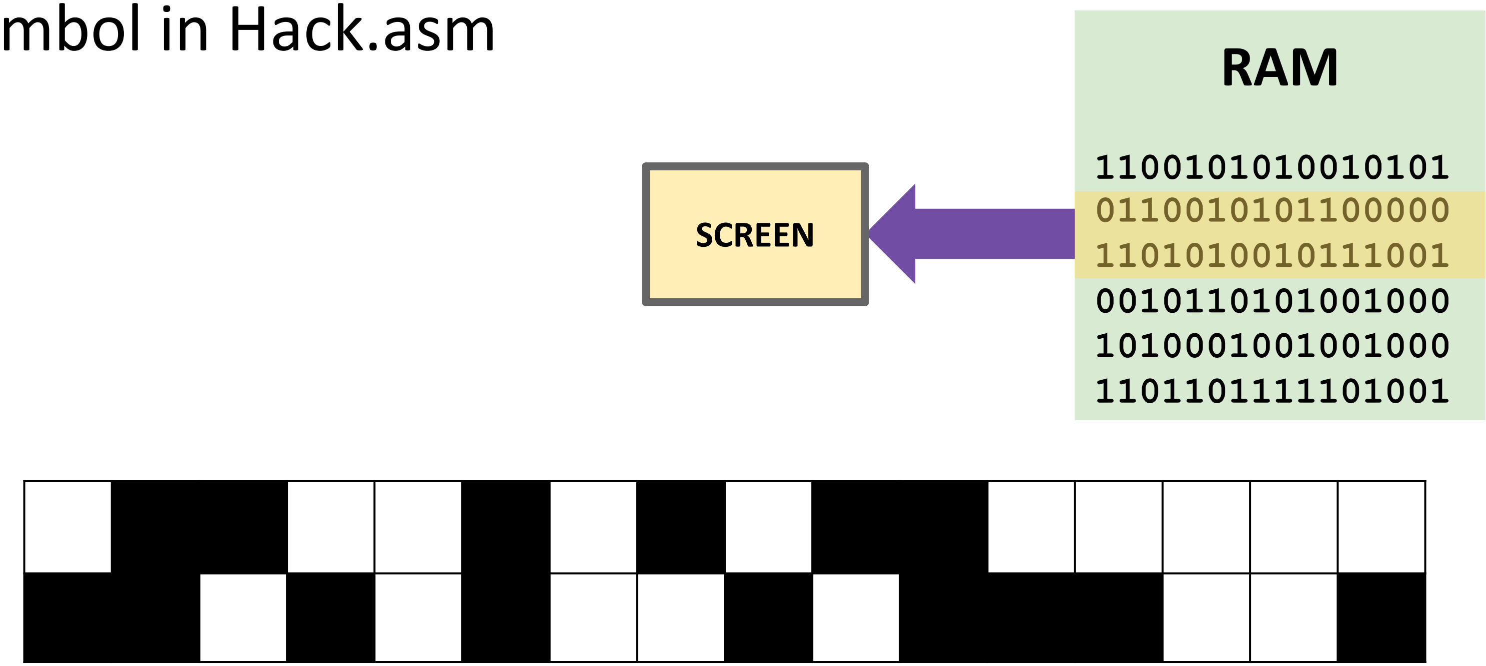
Hack: Input/Output

- I/O is **memory mapped**
  - Corresponds to some region of RAM
  - Low-level drivers are constantly refreshing



Hack: Memory Mapped Output

- Each bit of the screen memory map corresponds to one pixel (1 = black, 0 = white)
- The start of the memory map is accessible via the SCREEN symbol in Hack.asm



Hack: External Memory Abstraction

- Programmer sees one RAM32K memory region
  - Only 16K + 8K + 1 registers are being used
- Split into three parts: Screen, Keyboard, and the rest
  - Screen: 8K registers
  - Keyboard: 1 register
  - The rest: 16K registers (used for data and instructions)
- Programmer can use the same interface to interact with the Screen, Keyboard, or normal RAM
  - Just specify address, value, and other inputs
  - Address determines what part we are interacting with

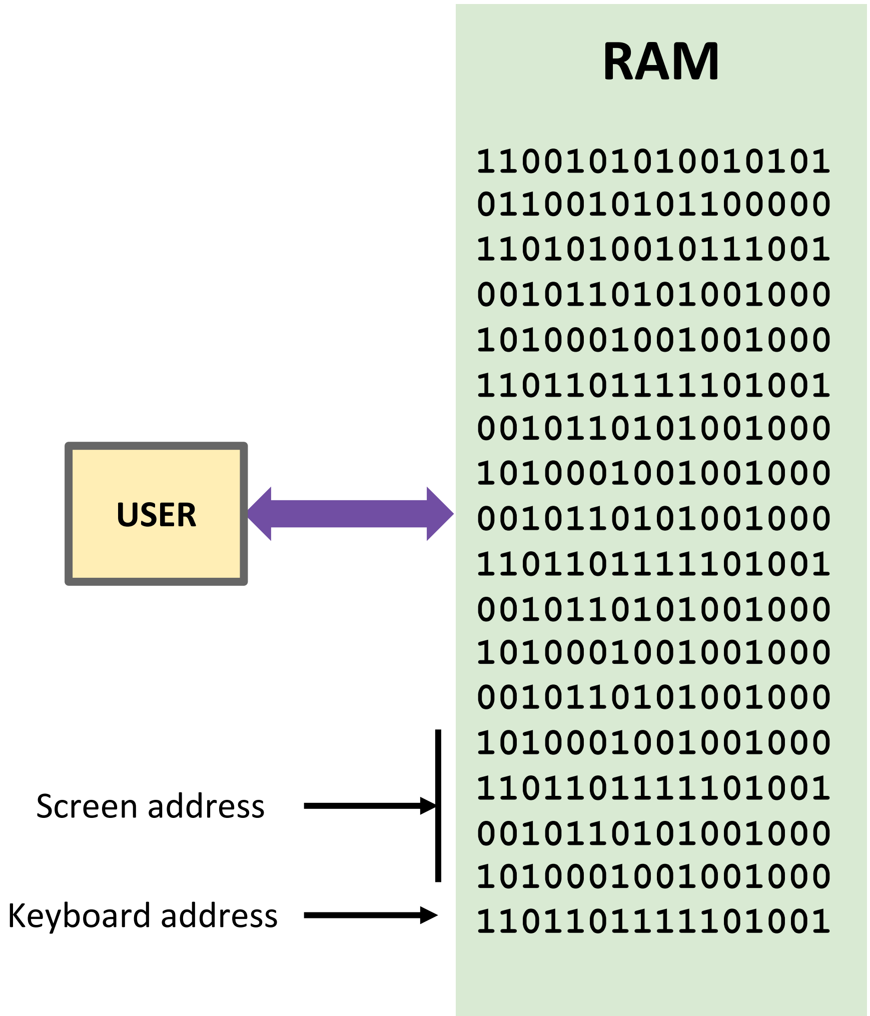
Hack: Internal Memory Implementation

- In reality, separate memory chips for memory devices is unnecessary
  - "Drivers" are code relaying changes in memory values to the device

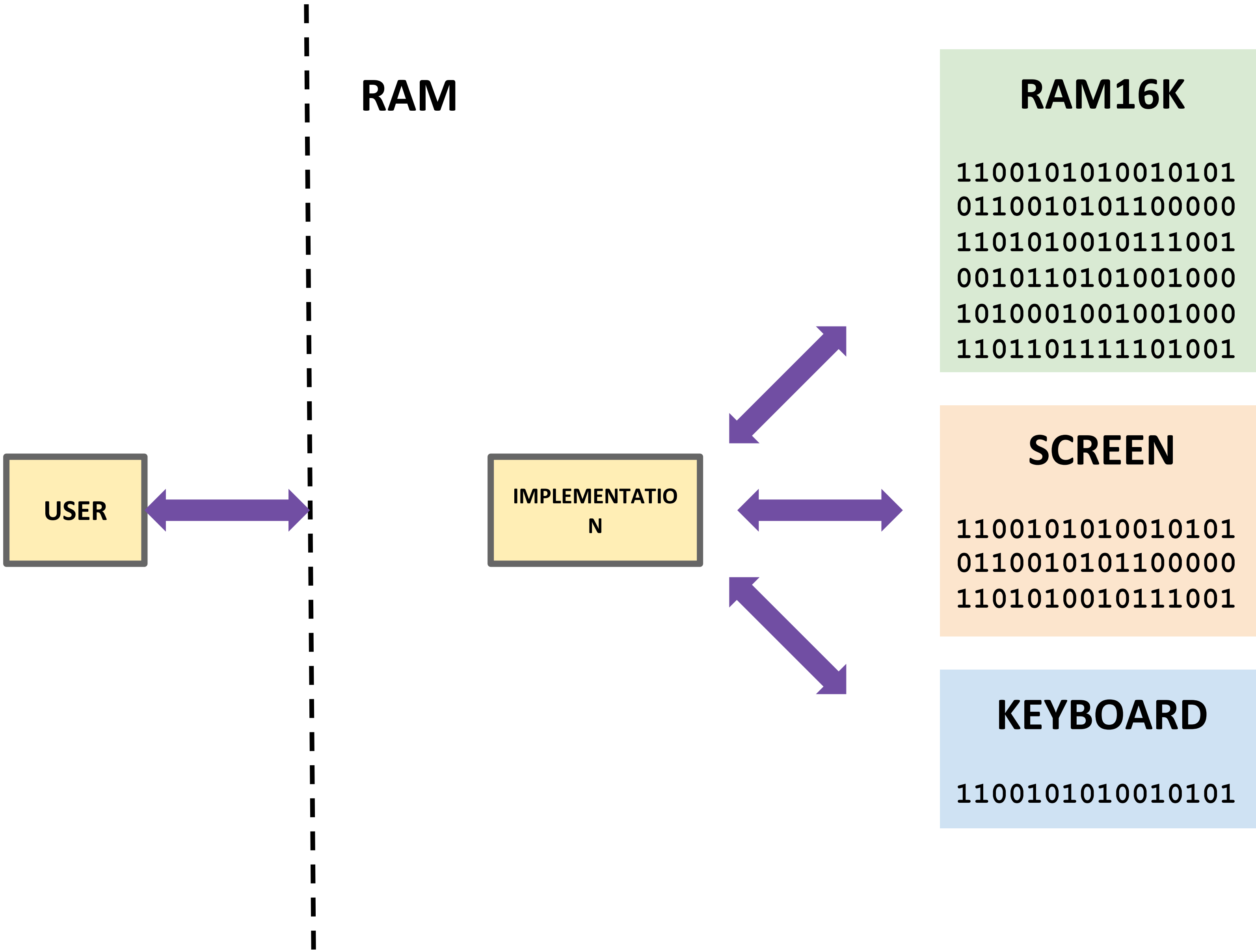
- In Hack, it's not as simple as one RAM32K chip
  - Use internal Keyboard and Screen chips so our virtual computer can detect/show changes in the keyboard and screen

- Our memory chip has three subchips: Screen, Keyboard, and RAM16K
  - Process the address given by the programmer and relay the request to the appropriate subchip

Hack: Memory Abstraction User View



Hack: Memory Abstraction Internal View

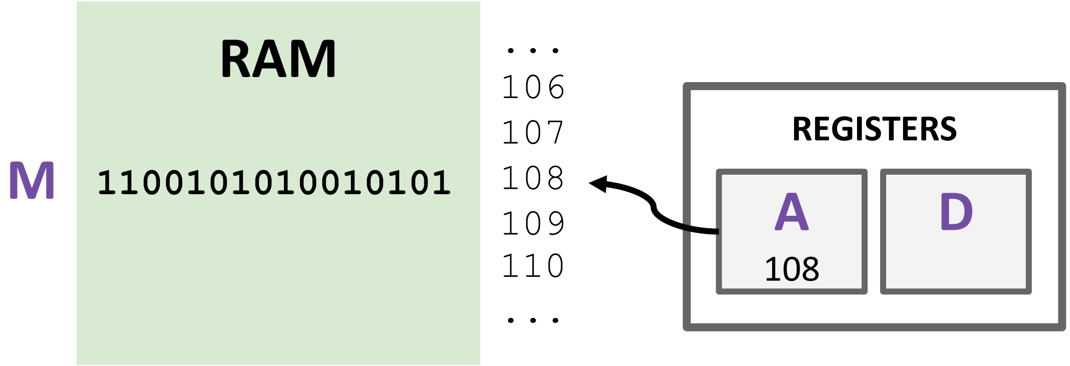


Lecture Outline

- Hack Assembly Language
  - Registers, A-Instructions, Symbols, C-Instructions
- Hack Assembly Memory Representation
  - Input / Output, Memory Mapping, External / Internal Memory
- **Multiplication Implementation Exercise**
  - **How do we multiply two numbers in the Hack Assembly language?**
- Project 4: Machine Language and Annotation Overview
  - Annotation, Assembly Language, Building a Computer Part I, Mid-quarter Reflection

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

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

Hack: C-Instructions

- Syntax: `dest = comp ; jump` (dest and jump are 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]`**

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

Example: Implementing Multiplication

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

- Pseudocode, add R0 to the result R1 times:

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

Example: Implementing Multiplication

- Remove loops from pseudocode
- Uses 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
```

Example: Implementing Multiplication

- Remove loops from pseudocode
- Uses 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
```

Example: 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)
```

Example: 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 = A
    @END
    D; JLE
(END)
```

Example: 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)
```

Example: 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)
```

Example: 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)
```

Example: 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

- Hack Assembly Language
  - Registers, A-Instructions, Symbols, C-Instructions
- Hack Assembly Memory Representation
  - Input / Output, Memory Mapping, External / Internal Memory
- Multiplication Implementation Exercise
  - How do we multiply two numbers in the Hack Assembly language?
- **Project 4: Machine Language and Annotation Overview**
  - **Annotation, Assembly Language, Building a Computer Part I, Mid-quarter Reflection**

Project 4 Overview

- Part I: Annotation
  - Come prepared to your upcoming Student-TA 1:1 meeting to work on Project 4 (e.g., spec reading and identifying annotation strategies you would want to use)
- Part II: Assembly Language
- Part III: Building a Computer Part I (Memory)

Project 4: Annotation Specs

- Annotate Project 4 Spec
  - Identify 5 annotation strategies that you want to try
  - Practice these strategies on the P4 Spec

- Fill out the Assignment Timeline
  - Divide up Project 4 into doable chunks for the days you plan to work on the assignment
  - Describe each day's task in as much detail as possible

Project 4: Annotation Specs

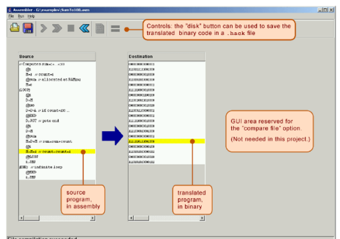
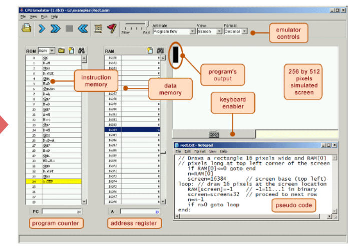
- Complete Annotation Reflection
  - Reflect on the strategies you used and why or why not they were effective

- Submit a copy of your annotations along with the Assignment Timeline document and the Annotation Reflection document

Project 4: Tools

- Running a Test Script (recommended flow):

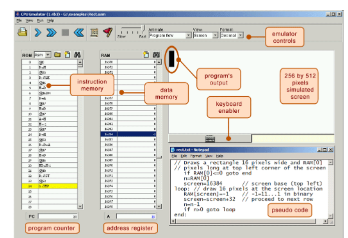
The test scripts use the .hack files directly! Don't let your .asm and .hack get out of sync!

`Max.asm` →  → `Max.hack` → 

Assembler

CPUEmulator

- Quickly Iterating or Experimenting:

`Max.asm` → 

CPUEmulator

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

Post-Lecture 8 Reminders

- What's in store for Week 5?
  - Technical Subject: Building a Computer
  - Metacognitive Subject: Exam Preparation
  - Project 5: Building a Computer Released

- Project Reminders
  - Project 2 grades released on Gradescope
  - **Project 3 due tonight (1/27) at 11:59pm PST**
  - Project 4: Machine Language and Annotation to be released today
    - Due next Thursday (2/3) at 11:59pm PST