CSE 390B, Spring 2022

Building Academic Success Through Bottom-Up Computing

Memory & Assembly in Hack

Annotation Strategies Discussion, Hack Assembly Memory Representation, Multiplication Implementation Exercise, Project 5 Overview

Lecture Outline

- **Annotation Strategies Discussion**
  - **Reflection on Annotating *Final Memory Chip* Reading**
- Hack Assembly Memory Representation
  - Input / Output, Memory Mapping, External / Internal Memory
- Multiplication Implementation Exercise
  - Multiplying Two Numbers in Hack Assembly
- Project 5 Overview
  - Specification Annotation, Machine Language, & Building Computer Memory

Annotation Strategies Discussion

- Which annotation strategies did you try for the *Final Memory Chip* reading?
- Did using annotation strategies help you comprehend the text more than usual? Why or why not?
- Any surprises or interesting observations from annotating?
- What might you change about your existing annotation strategies? What other annotation strategies might you try?

Lecture Outline

- Annotation Strategies Discussion
  - Reflection on Annotating *Final Memory Chip* Reading
- **Hack Assembly Memory Representation**
  - **Input / Output, Memory Mapping, External / Internal Memory**
- Multiplication Implementation Exercise
  - Multiplying Two Numbers in Hack Assembly
- Project 5 Overview
  - Specification Annotation, Machine Language, & Building Computer Memory

Poll Everywhere

Vote at https://pollev.com/cse390b

Which of the following statements from the *Final Memory* Chip reading is FALSE?

A. Hexadecimal is useful because it's easier for humans to read while still being interpretable by a computer

B. 0x390B in binary is 0b0011_1001_0000_1011

C. 390 in hexadecimal is 0x186

D. In Hack, the two devices the user can interact with are the screen and the keyboard

E. We're lost…

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

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 for the corresponding hexadecimal digit (or vice versa)
- Example: 0x3A is 0b0011_1010
  - 0x3 == 0b0011
  - 0xA == 0b1010

Hack Assembly: Input / Output

- Two memory maps are created for you by underlying hardware
  - **SCREEN** is a huge map where each pixel is one bit
  - **KEYBOARD** is a single 16-bit word map with code of current key

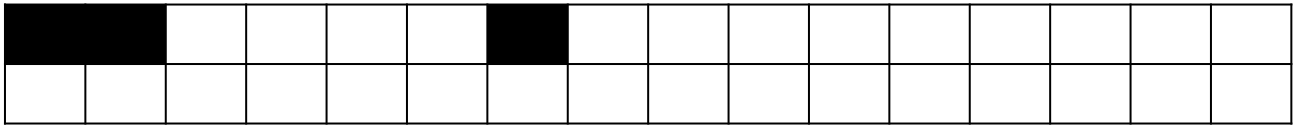
- Example:

```
@KBD
D=M

@SCREEN
M=D
```

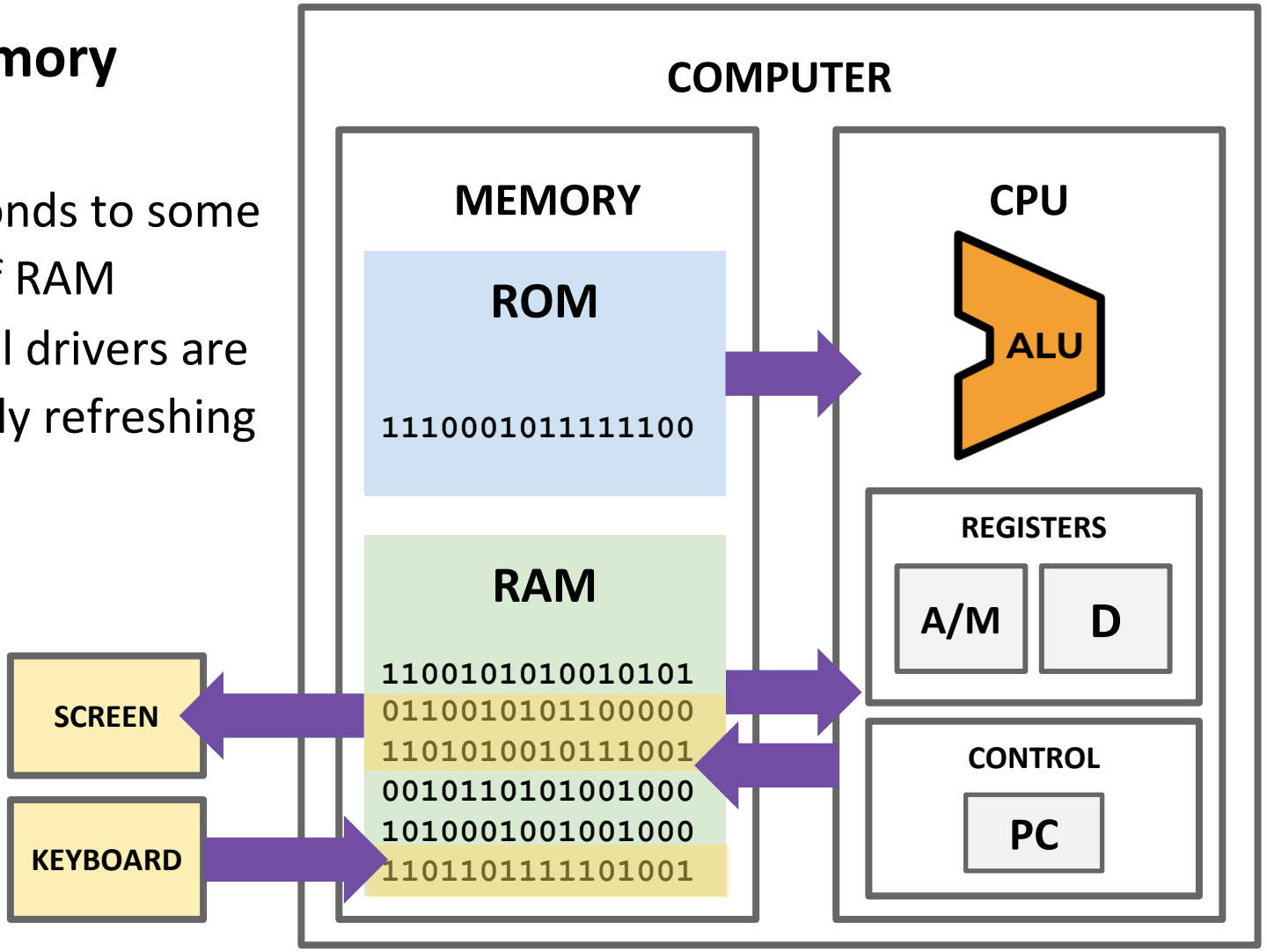
D contains code of current key (e.g., 67 for "C")

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



Hack: Input / Output (I/O)

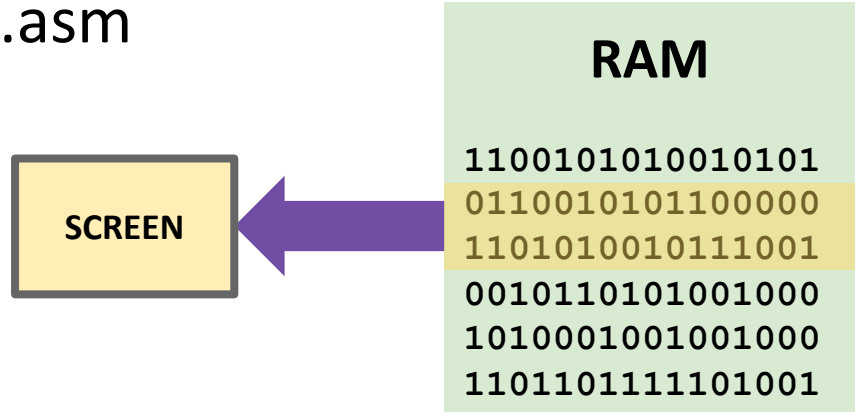
- I/O is **memory mapped**
  - Corresponds to some region of RAM
  - Low-level drivers are constantly refreshing

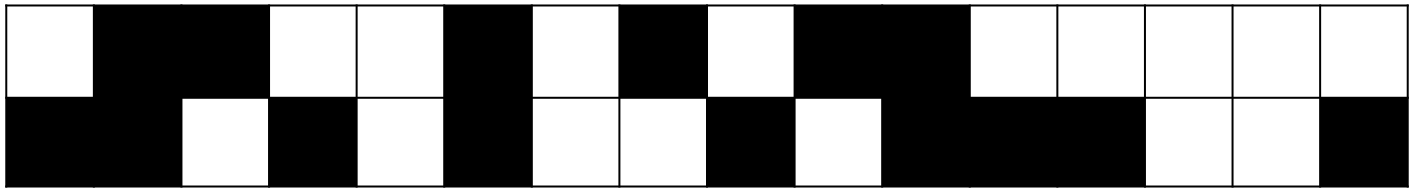


Hack: Memory Mapped Output

- 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

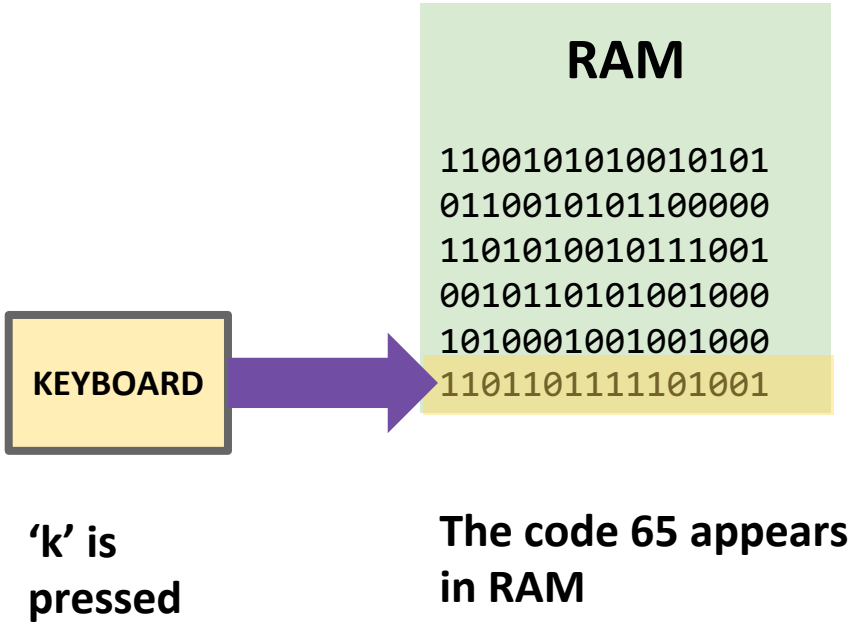




Hack: Memory Mapped Output

- A single 16-bit word in memory is constantly refreshed with the scan code of the keyboard button being pressed

- This spot in memory accessible via the **KBD** constant in Hack.asm



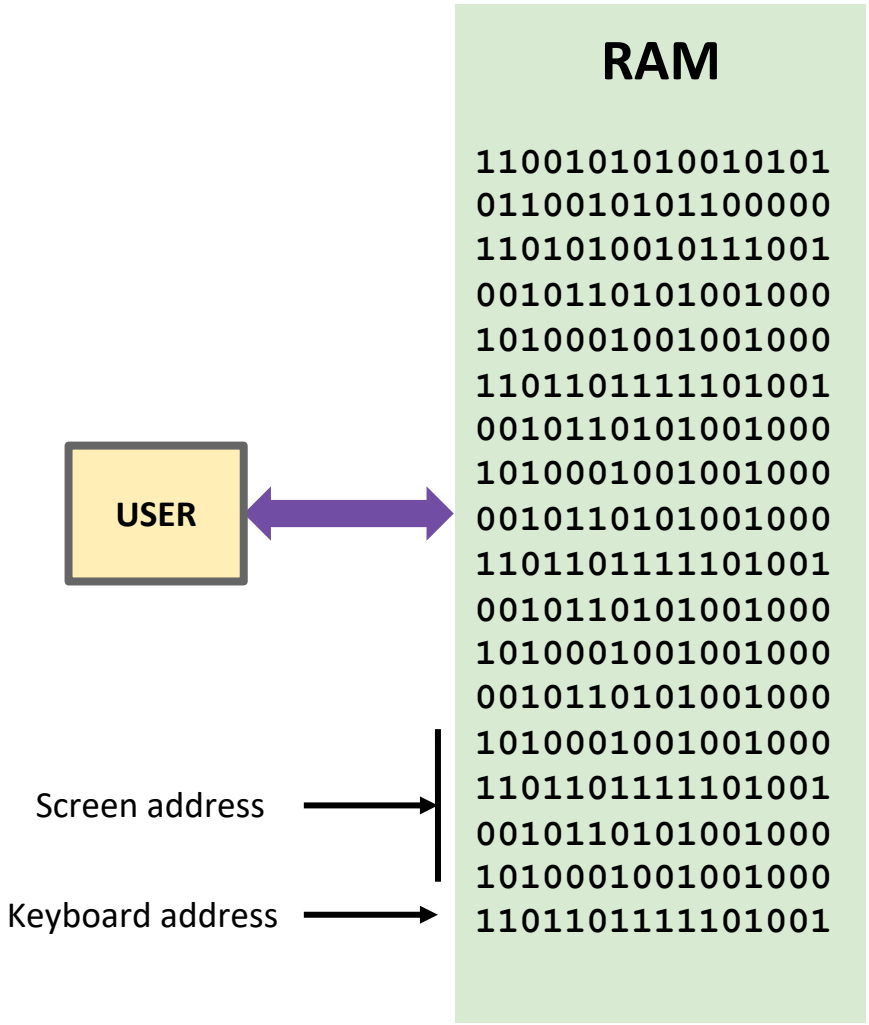
Hack: External Memory Abstraction

- 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

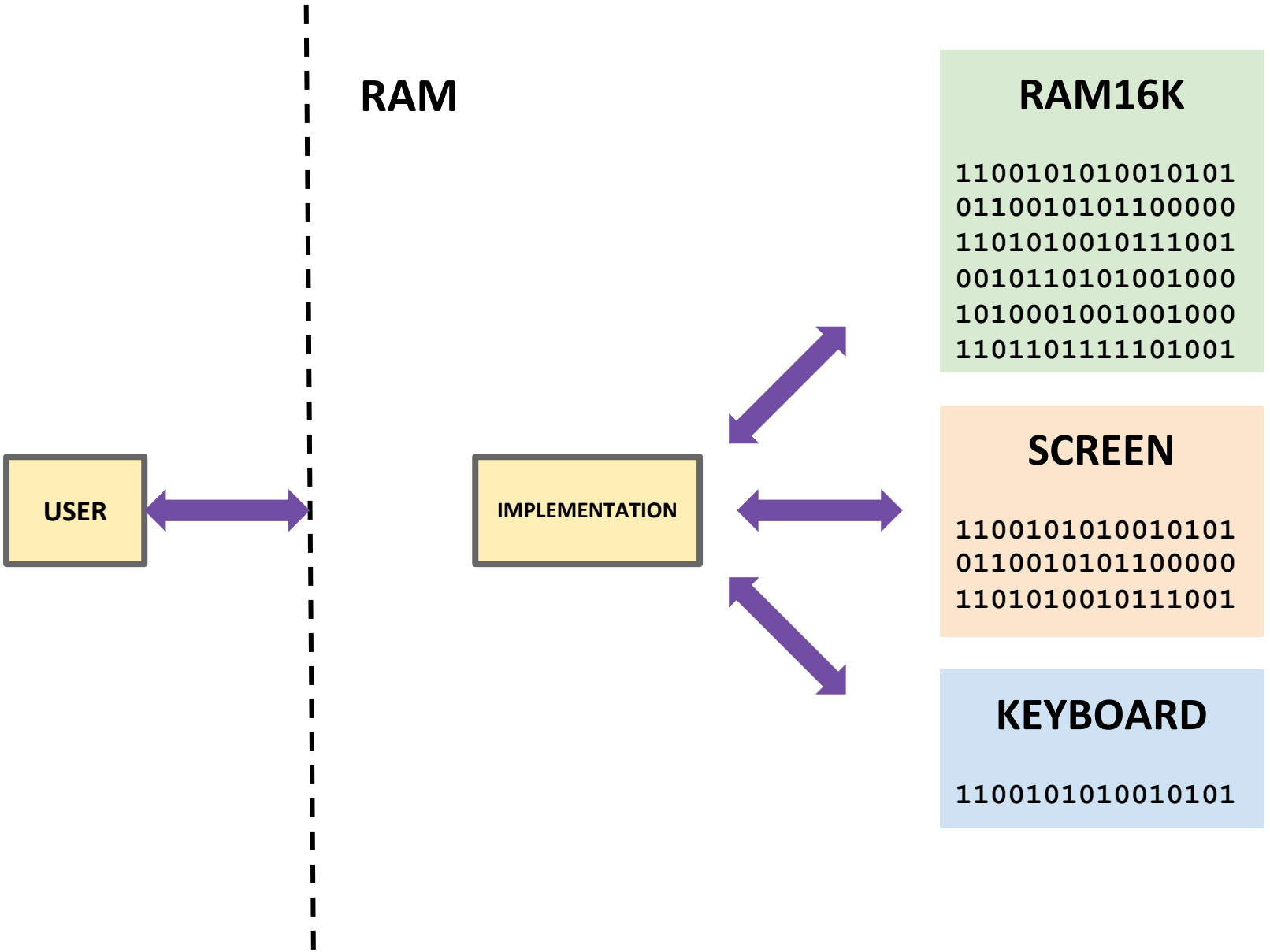
Hack: Internal Memory Implementation

- 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 **`SCREEN`** and **`KEYBOARD`** chips so our virtual computer can detect and show changes in the screen and keyboard
- 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

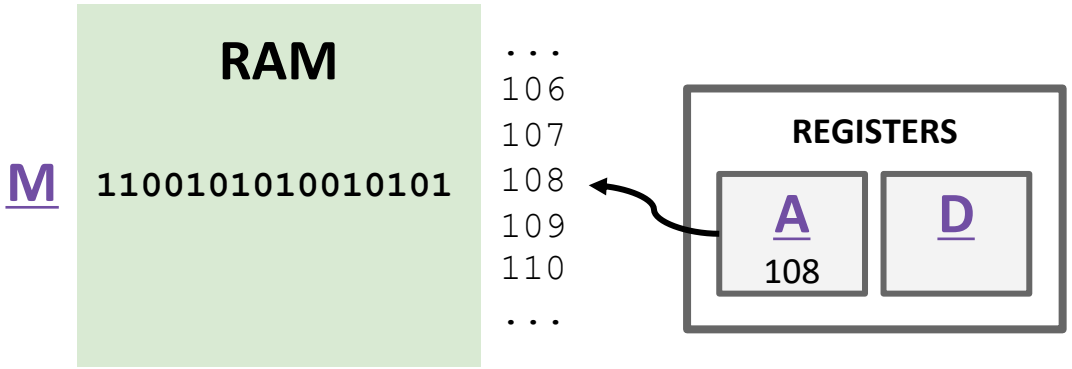


Lecture Outline

- Annotation Strategies Discussion
  - Reflection on Annotating *Final Memory Chip* Reading
- Hack Assembly Memory Representation
  - Input / Output, Memory Mapping, External / Internal Memory
- **Multiplication Implementation Exercise**
  - **Multiplying Two Numbers in Hack Assembly**
- Project 5 Overview
  - Specification Annotation, Machine Language, & Building Computer Memory

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: C-Instructions

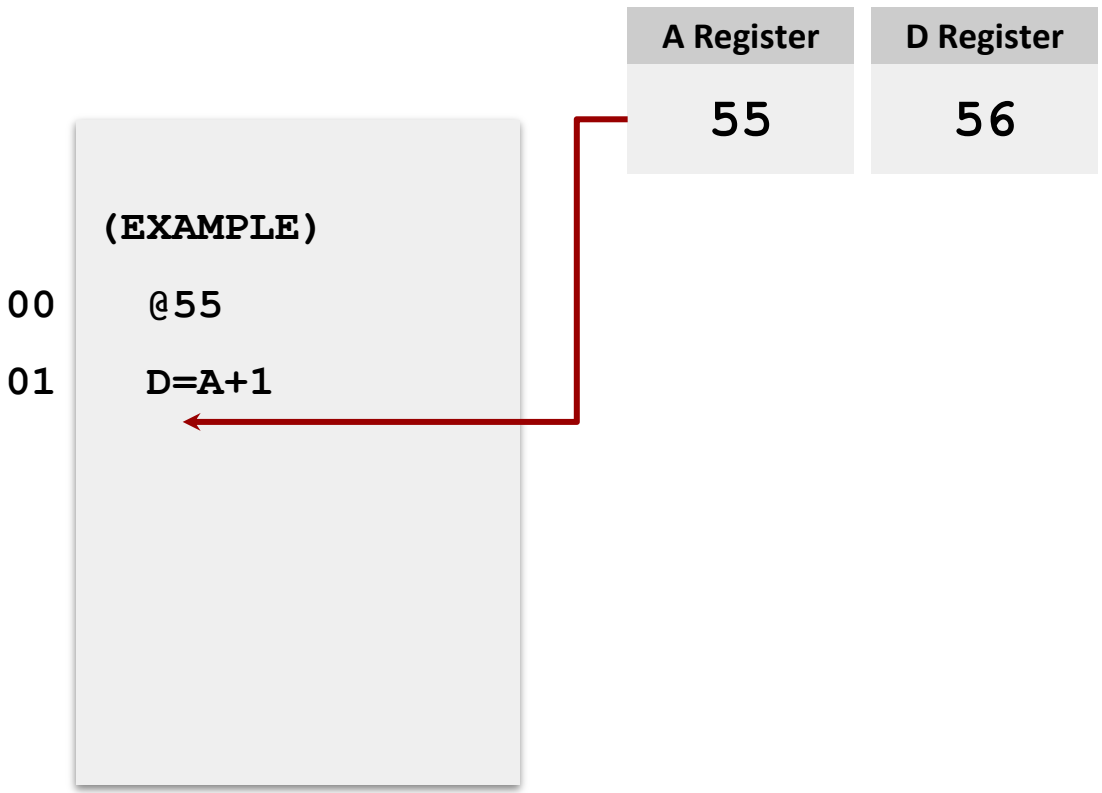
- Syntax: `dest = comp ; jump` (**`dest`** and **`jump`** optional)
  - **`dest`** is a combination of destination registers:

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

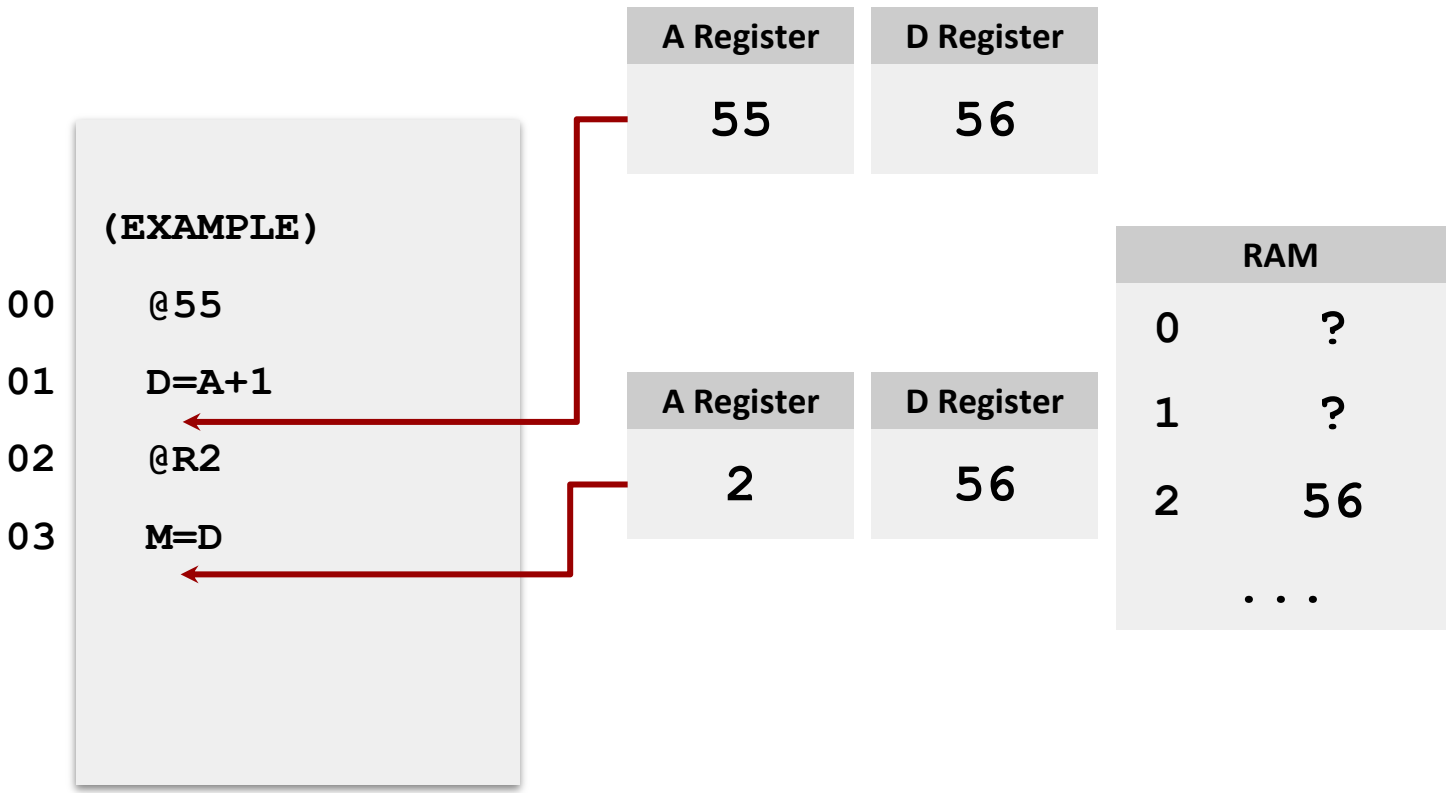
    `0, 1, -1, D, A, !D, !A, -D, -A, D+1, A+1, D-1, A-1, D+A, D-A, A-D, D&A, D|A, M, !M, -M, M+1, M-1, D+M, D-M, M-D, D&M, D|M`
  - **`jump`** is an unconditional or conditional jump:

    `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]`**

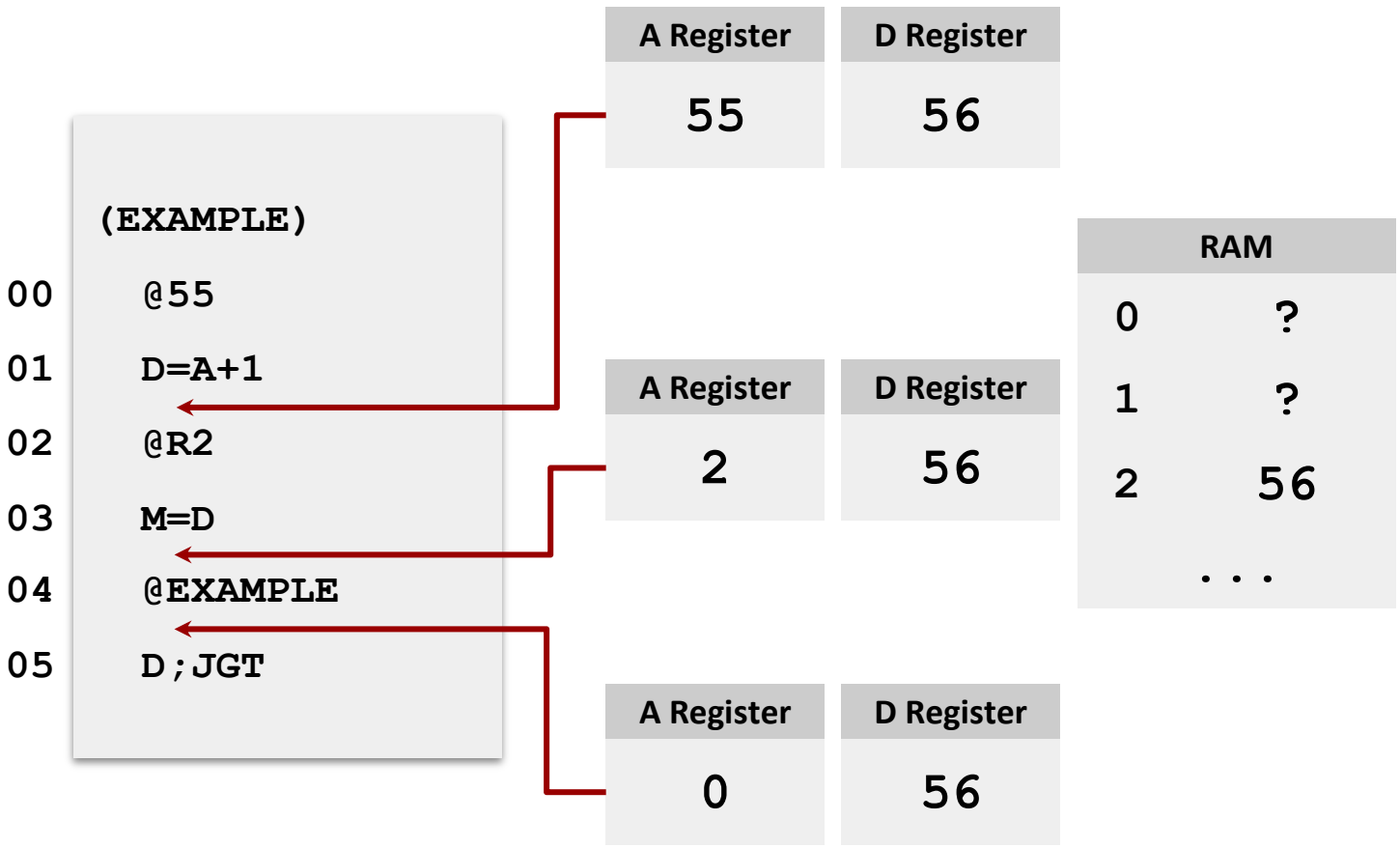
Hack: C-Instructions Example



Hack: C-Instructions Example



Hack: C-Instructions Example



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

Five-minute Break!

- Feel free to stand up, stretch, use the restroom, drink some water, review your notes, or ask questions
- We'll be back at:
- Research shows mid-lecture breaks reduce the decline of attention in the middle of lecture (Olmsted, 1999)

Olmsted III, John. "The Mid-Lecture Break: When Less Is More." *Journal of Chemical Education* (1999). https://pubs.acs.org/doi/abs/10.1021/ed076p525.

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, add **`R0`** to the result **`R1`** times:

Exercise: Implementing Multiplication

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

- Pseudocode, 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
- Uses 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
- 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
```

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 = A
    @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
(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

- Annotation Strategies Discussion
  - Reflection on Annotating *Final Memory Chip* Reading
- Hack Assembly Memory Representation
  - Input / Output, Memory Mapping, External / Internal Memory
- Multiplication Implementation Exercise
  - Multiplying Two Numbers in Hack Assembly
- **Project 5 Overview**
  - **Specification Annotation, Machine Language, & Building Computer Memory**

Project 5 Overview

- Part I: Annotation
  - Come prepared to your upcoming Student-TA 1:1 meeting to work on Project 5 (e.g., specification reading and identifying annotation strategies you would want to use)
- Part II: Machine Language
  - Implement Max.asm in Hack Assembly
- Part III: Building Computer Memory
  - Implement Memory.hdl in HDL
- Part IV: Project 5 Reflection

Project 5, Part I: Annotation

- **Fill out the Assignment Timeline**
  - Divide up Project 5 into doable chunks for the days you plan to work on the assignment
  - Describe each day's task in as much detail as possible

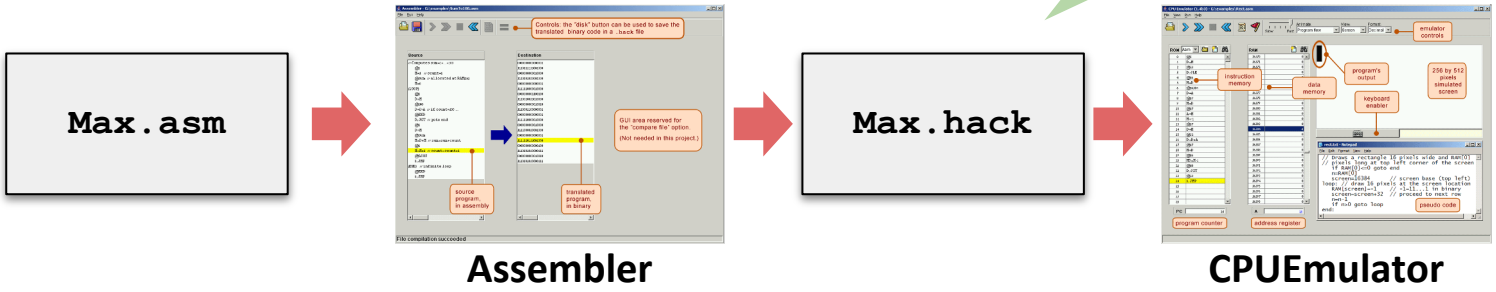
- **Annotate the Project 5 Specification**
  - Identify five annotation strategies that you want to try
  - Practice these strategies on the Project 5 specification

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

Project 5: 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 directly from the .asm file

Lecture 8 Wrap-up

- Thrilled for these Week 5 topics!
  - Metacognitive Subject: Exam Preparation
  - Technical Subject: Building a Computer
- Project Reminders
  - Project 3 grades and feedback released
  - **Project 4 due tonight (4/21) at 11:59pm PDT**
  - Project 5 released, due next Thursday (4/28) at 11:59pm PDT
- Midterm Exam coming up in two weeks (5/5) during lecture time