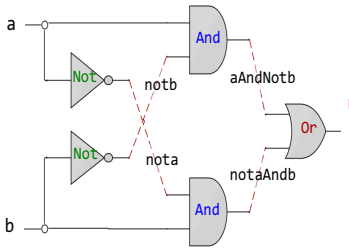
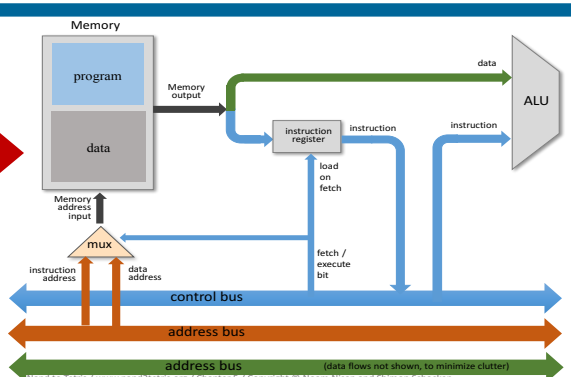
Computer Organization & Assembly Language Programming




```
CHIP Xor {
  IN a, b;
  OUT out;
  PARTS:
  Not(in=a, out=nota);
  Not(in=b, out=notb);
  And(a=nota, b=b, out=w1);
  And(a=a, b=notb, out=w2);
  Or(a=w1, b=w2, out=out);
}
```




```
@R1
D=M
@temp
M=D
```

```
0000000000000001
1111110000010000
0000000000010000
1110001100001000
```

Lecture # 21

Hack Assembly Programming - III

```
#include<stdio.h>
#include<stdlib.h>
int main(){
  printf("Learning is fun with Arif\n");
  exit(0);
}
```

```
global main
SECTION .data
   msg: db "Learning is fun with Arif", 0Ah, 0h
   len_msg: equ $ - msg
SECTION .text
   main:
      mov rax,1
      mov rdi,1
      mov rsi,msg
      mov rdx,len_msg
      syscall
      mov rax,60
      mov rdi,0
      syscall
```

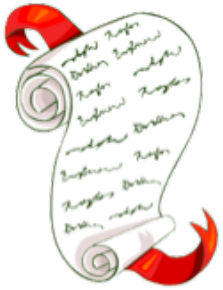
```
0:  b8 01 00 00 00
5:  bf 01 00 00 00
a:  48 be 00 00 00 00 00
11: 00 00 00
14: ba 1b 00 00 00
19: 0f 05
1b: b8 3c 00 00 00
20: bf 00 00 00 00
25: 0f 05
```

Slides of first half of the course are adapted from:
https://www.nand2tetris.org
Download s/w tools required for first half of the course from the following link:
https://drive.google.com/file/d/0B9c0BdDJz6XpZUh3X2dPR1o0MUE/view

Instructor: Muhammad Arif Butt, Ph.D.

Today's Agenda

- Review of Hack Assembly Programs
- Pointers and Arrays
- Input / Output Instructions
- Debugging



Pointers and Arrays

Pointers and Arrays: Example

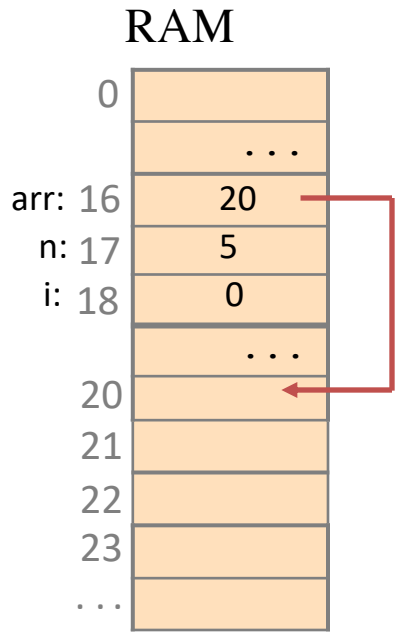
```
// for (i=0; i<n; i++)
//       arr[i] = -1;
```

Observations:

- Variables that store memory addresses like **arr** in this example are called pointers
- Abstraction of arrays exist only in high level languages. In machine language there is no abstraction of arrays. Rather array is a segment of memory of which we know the base address of this segment and the length of the array that programmer has declared
- Arrays are implemented as a block of memory registers and in order to access these memory registers one after the other, we need a variable that holds the current address
- There is nothing special about pointer variables, except that their values are interpreted as addresses

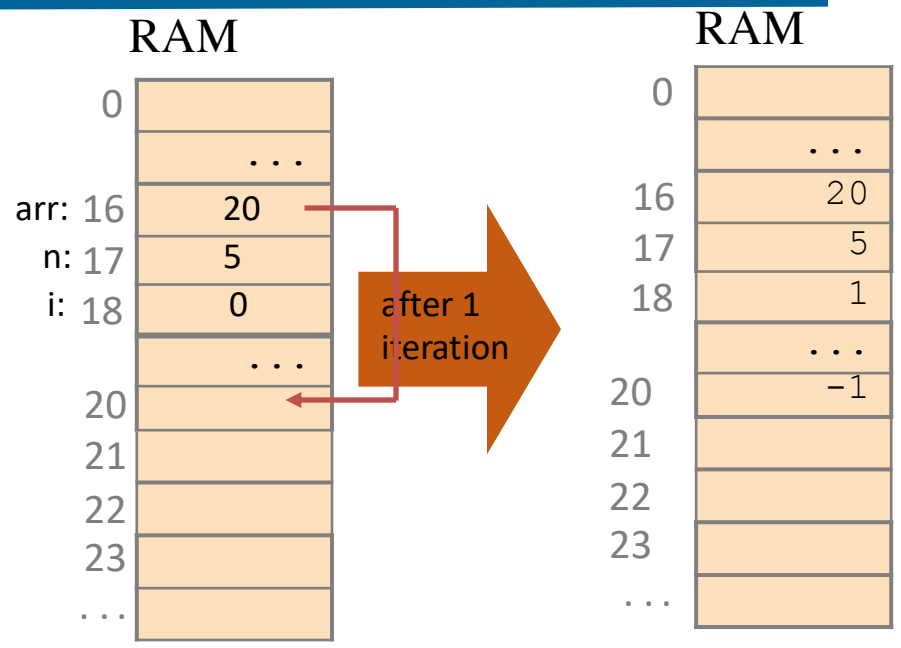
Pointers and Arrays: Example

```
// for (i=0; i<n; i++)
//      arr[i] = -1;
// Let us initialize arr=20, n=5, i=0

// arr = 20
@20
D=A
@arr
M=D

// n = 5
@5
D=A
@n
M=D

// i = 0
@i
M=0

// Loop code continues on next slide...
          (LOOP)
```



Pointers and Arrays: Example

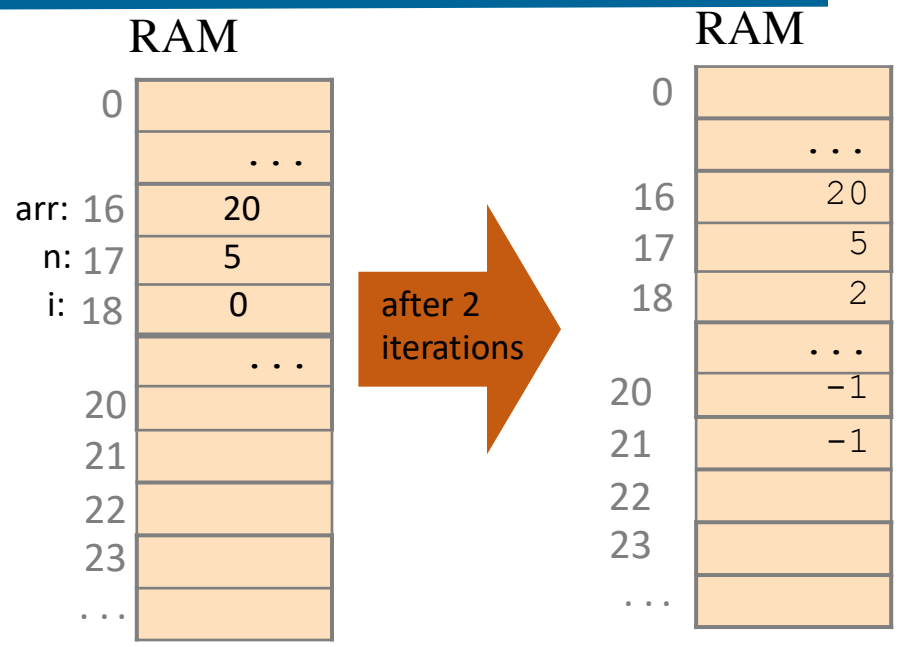
```
// Code continues from previous slide
(LOOP)
// if (i==n) goto END
   @i
   D=M
   @n
   D=D-M
   @END
   D;JEQ
// RAM[arr+i] = -1
   @arr
   D=M
   @i
   A=D+M
   M=-1
// i++
   @i
   M=M+1
   @LOOP
   0;JMP
(END)
   @END
   0;JMP
```



- Pointers in Hack: Whenever we have to access memory using a pointer, we need an instruction like A=expression
- Typical Pointer Semantics: Set the address register to the contents of some memory register

Pointers and Arrays: Example

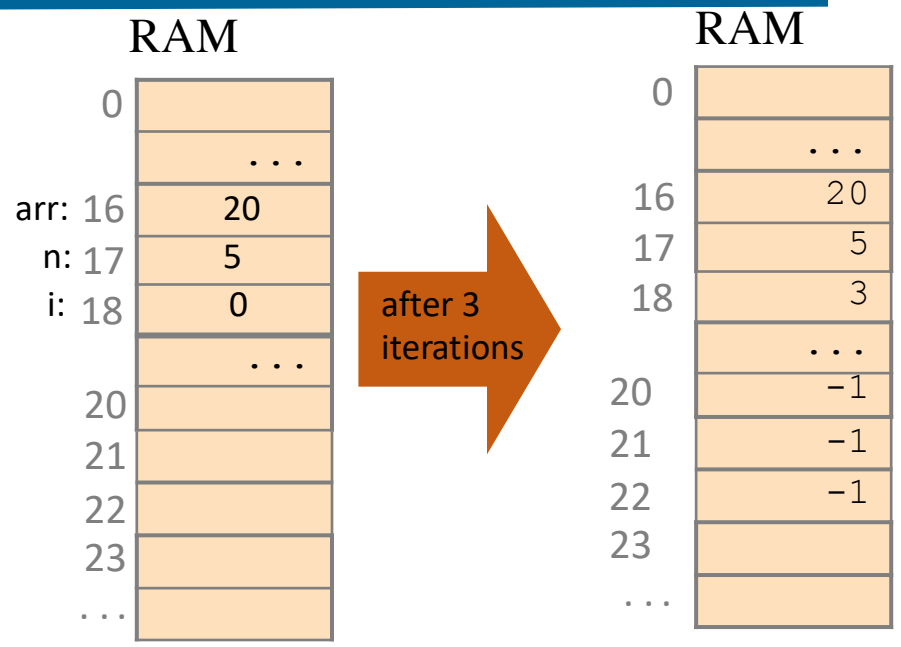
```
// Code continues from previous slide
(LOOP)
// if (i==n) goto END
   @i
   D=M
   @n
   D=D-M
   @END
   D;JEQ
// RAM[arr+i] = -1
   @arr
   D=M
   @i
   A=D+M
   M=-1
// i++
   @i
   M=M+1
   @LOOP
   0;JMP
(END)
   @END
   0;JMP
```



- Pointers in Hack: Whenever we have to access memory using a pointer, we need an instruction like A=expression
- Typical Pointer Semantics: Set the address register to the contents of some memory register

Pointers and Arrays: Example

```
// Code continues from previous slide
(LOOP)
// if (i==n) goto END
   @i
   D=M
   @n
   D=D-M
   @END
   D;JEQ
// RAM[arr+i] = -1
   @arr
   D=M
   @i
   A=D+M
   M=-1
// i++
   @i
   M=M+1
   @LOOP
   0;JMP
(END)
   @END
   0;JMP
```

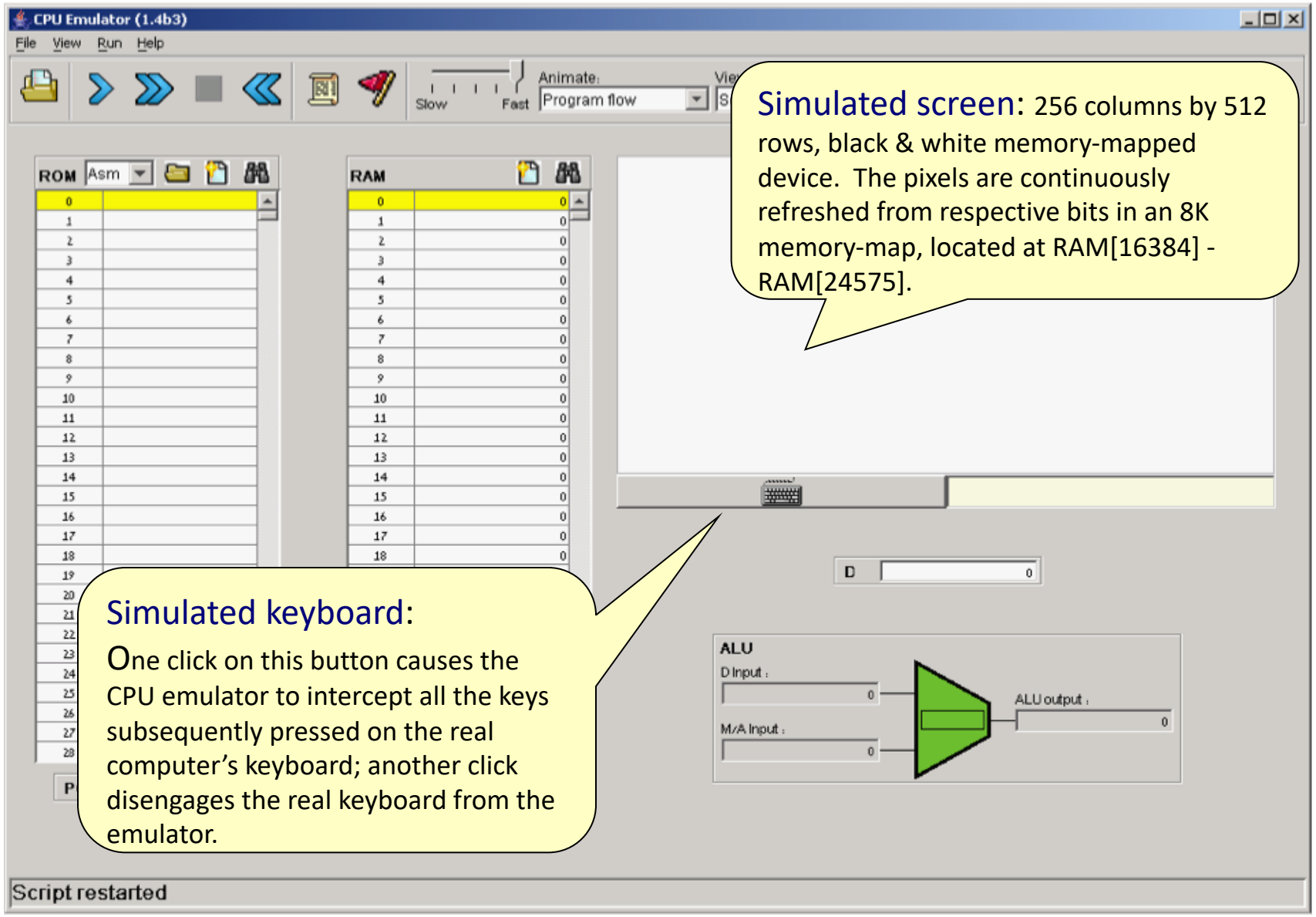


- Pointers in Hack: Whenever we have to access memory using a pointer, we need an instruction like A=expression
- Typical Pointer Semantics: Set the address register to the contents of some memory register

Manipulating Arrays using Pointers

Input & Output

I/O Devices: Screen And Keyboard

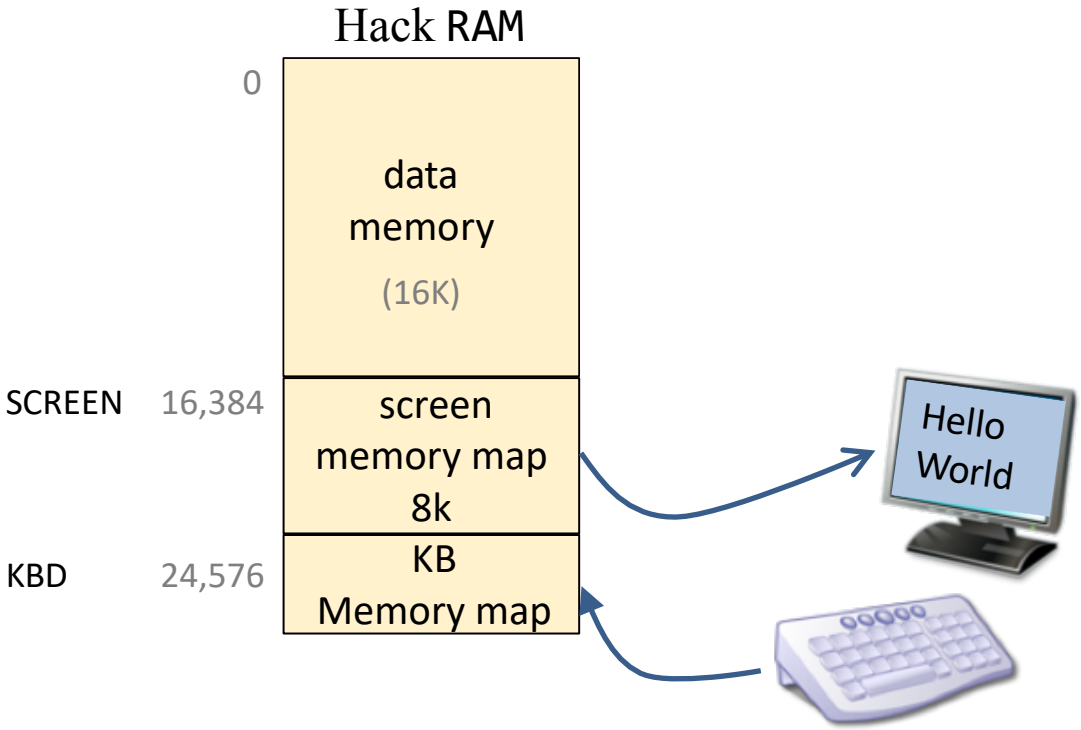


Simulated screen: 256 columns by 512 rows, black & white memory-mapped device. The pixels are continuously refreshed from respective bits in an 8K memory-map, located at RAM[16384] - RAM[24575].

Simulated keyboard:

One click on this button causes the CPU emulator to intercept all the keys subsequently pressed on the real computer's keyboard; another click disengages the real keyboard from the emulator.

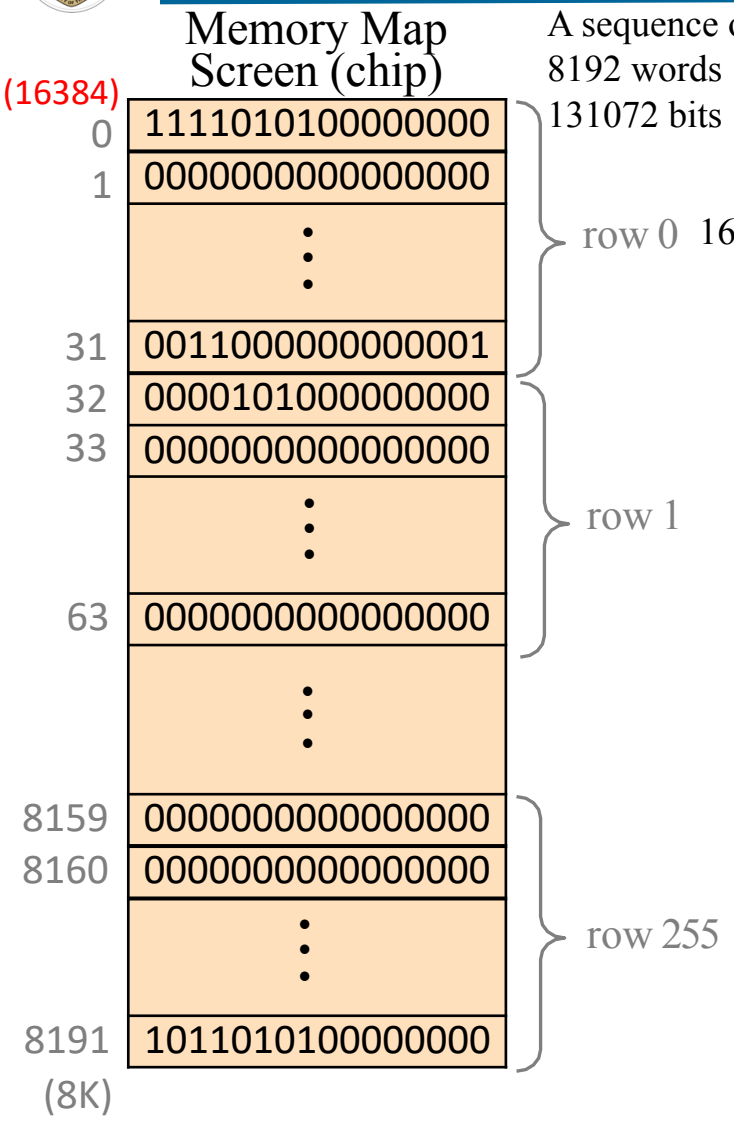
Memory Map of Screen and Keyboard



Hack language convention:

- SCREEN: base address of the screen memory map
- KBD: address of the keyboard memory map

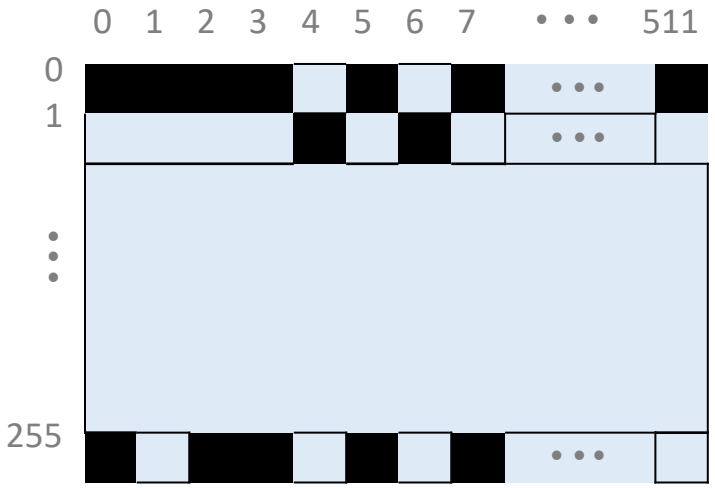
Screen Memory Map



refresh

Black & White Display Unit

- A matrix of 256 rows x 512 columns
- 131072 pixels

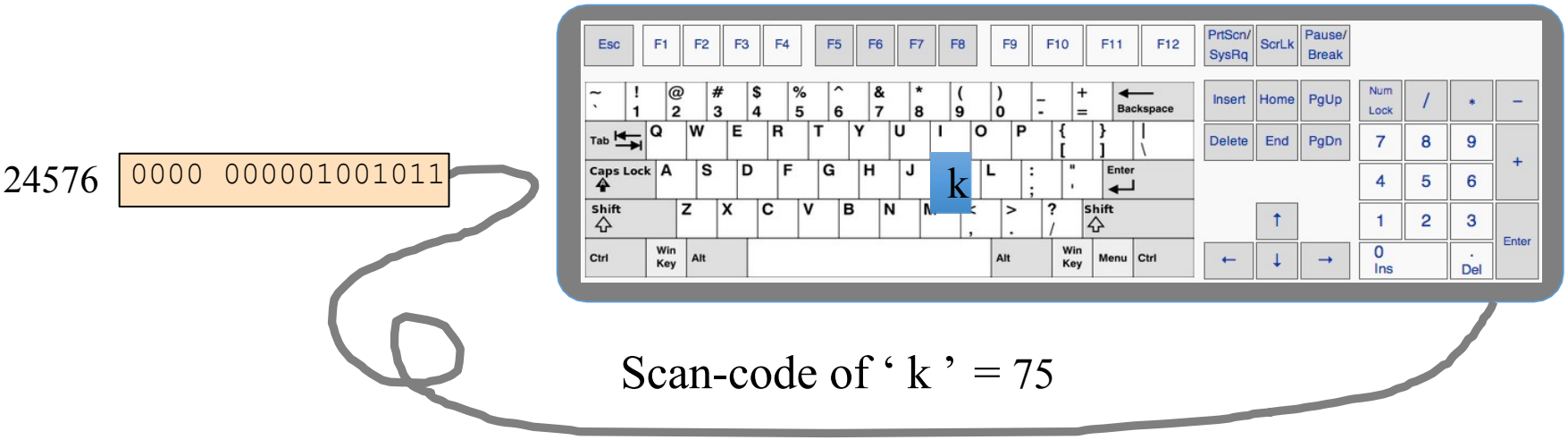


To set pixel (row,col) on/off

```
word = Screen[32*row + col/16]
word = RAM[16384 + 32*row + col/16]
Set (col%16)th bit of word to 0 or 1
RAM[i] = word
```

Handling The Keyboard

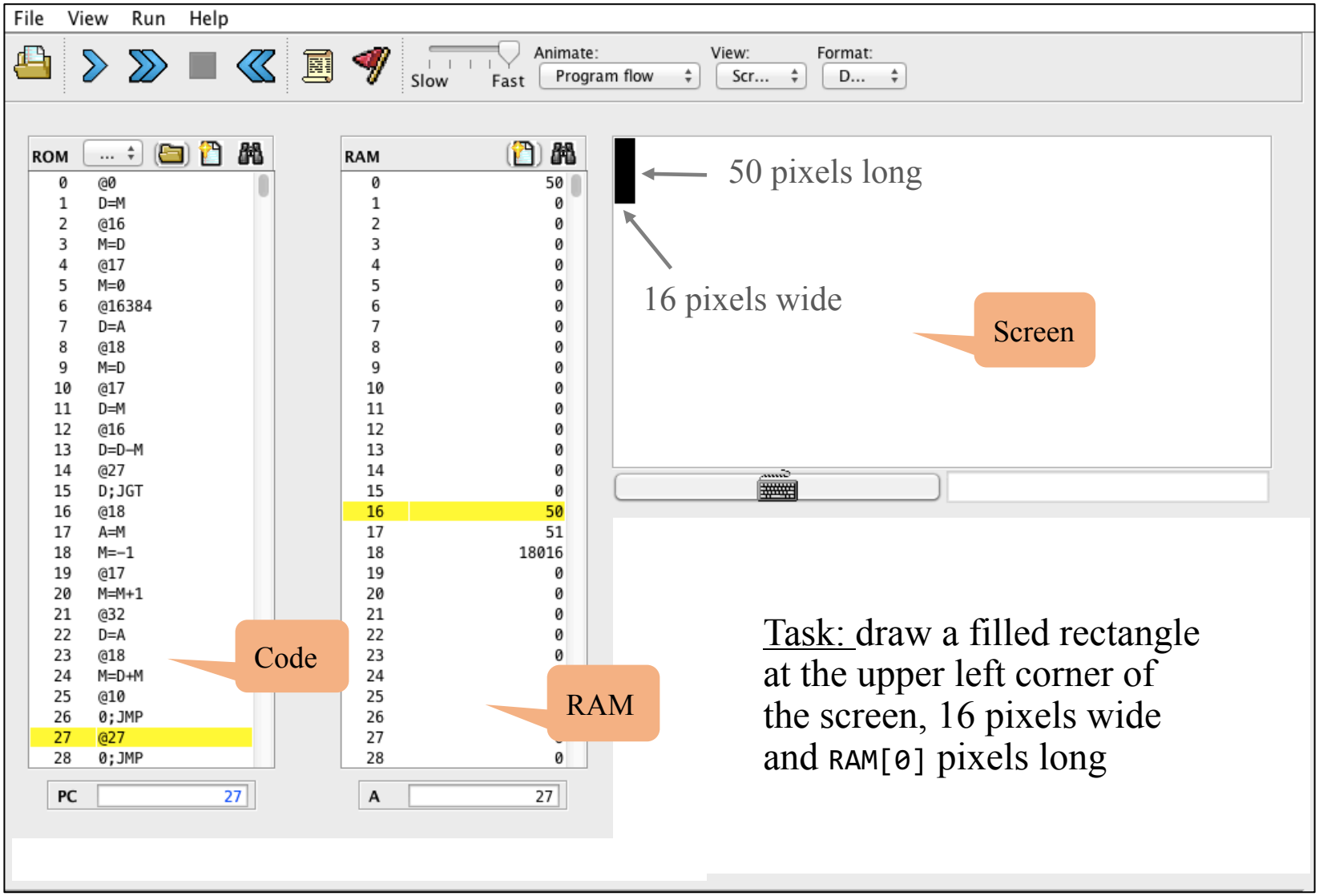
Hack RAM



To check which key is currently pressed:

- Read the contents of RAM[24576] (address KBD)
- If the register contains 0, no key is pressed
- Otherwise, the register contains the scan code of the currently pressed key

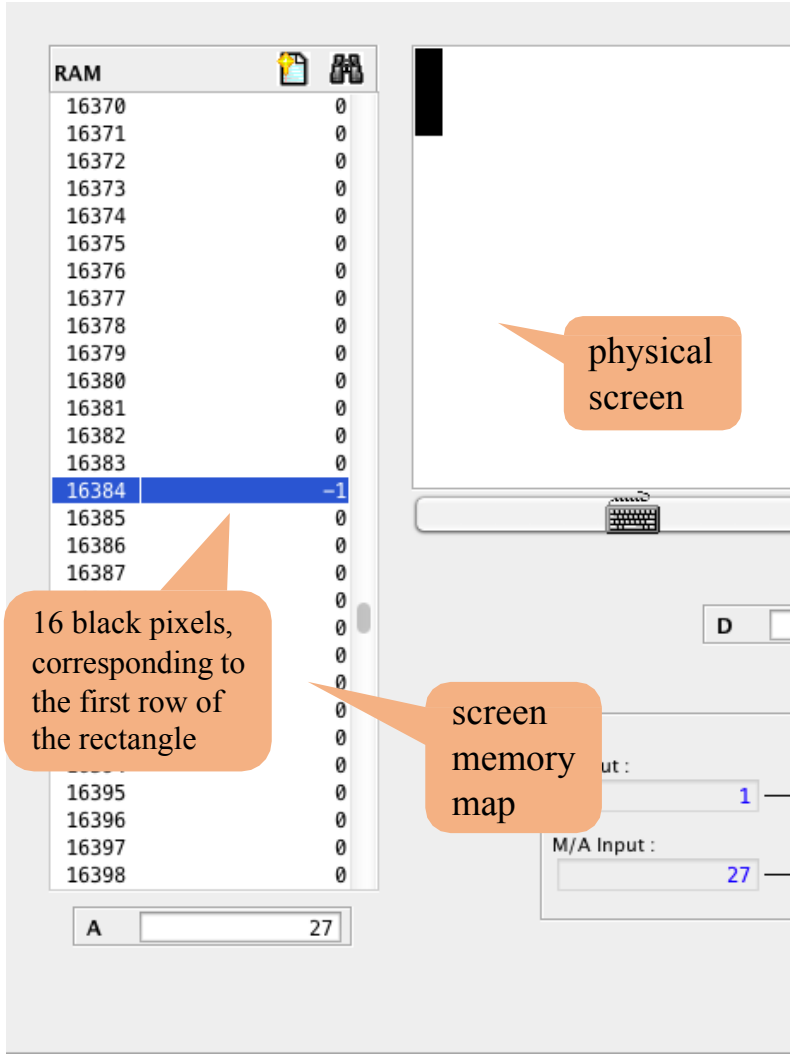
Drawing a Rectangle on The Screen



Drawing a Rectangle on The Screen

Pseudo code

```
for (i=0; i<n; i++)
   draw 16 black pixels at the beginning of row i


addr = SCREEN
n = RAM[0]
i = 0


LOOP:
   if i > n goto END
   RAM[addr] = -1 //1111111111111111
   // advances to the next row
   addr = addr + 32
   i = i + 1
   goto LOOP


END:
goto END
```



Drawing a Rectangle on The Screen

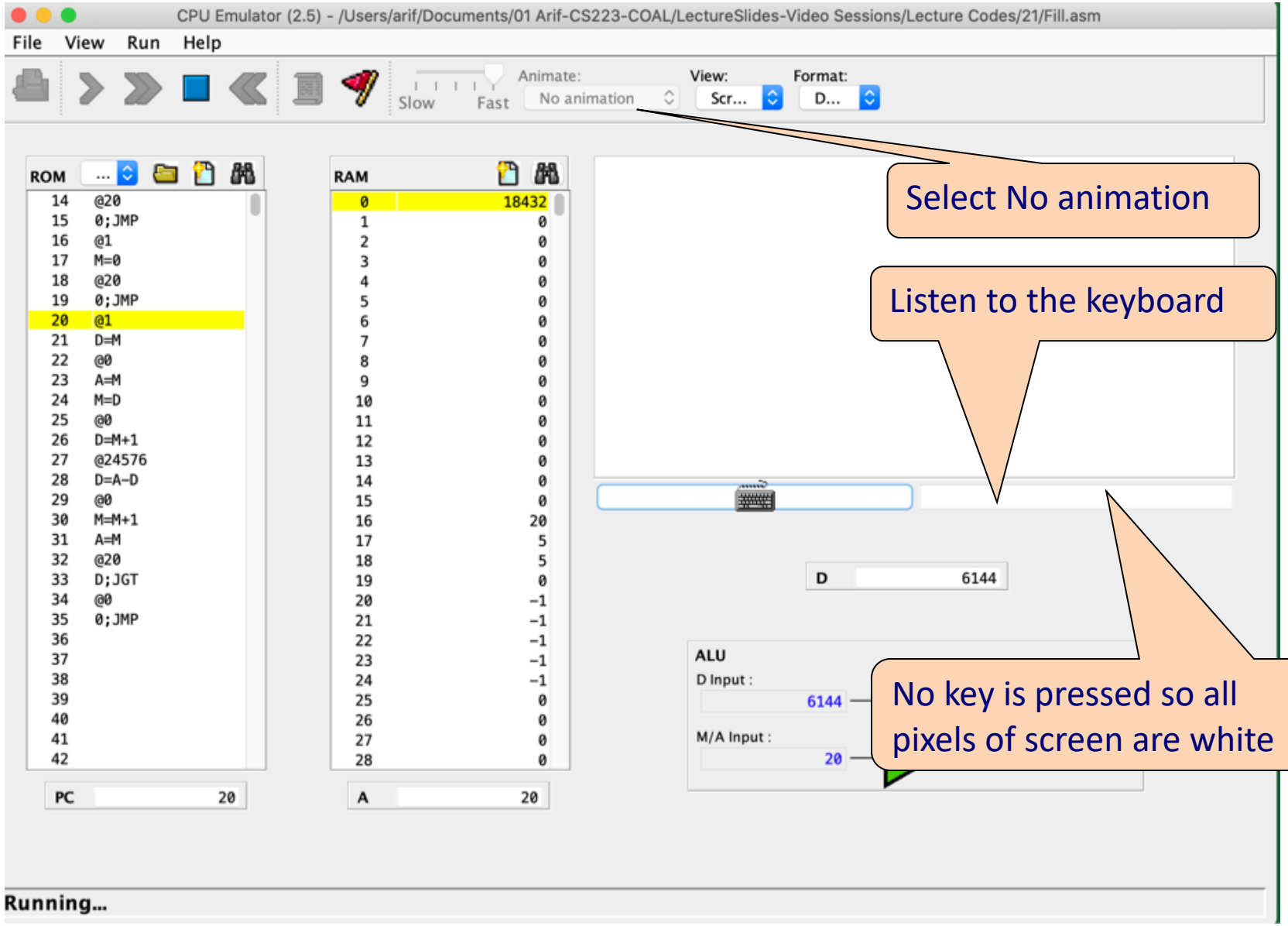
```
/* Program: Rectangle.asm

Draws a filled rectangle at the screen's
top left corner, with width of 16 pixels
and height of RAM[0] pixels.

Usage: put a non-negative number

(rectangle's height) in RAM[0] */

@R0
D=M
@n
M=D    // n = RAM[0]
@i
M=0    // i = 0
@SCREEN
D=A
@addr
M=D    // addr = 16384 (screen's base
address)


(LOOP)
// ...
```

```
//...
(LOOP)
   @i
   D=M
   @n
   D=D-M
   @END
   D;JGT // if i>n goto END
   @addr
   A=M
   M=-1 //RAM[addr]=1111111111111111
   @i
   M=M+1 // i = i + 1
   @32
   D=A
   @addr
   M=D+M // addr = addr + 32
   @LOOP
   0;JMP // goto LOOP
(END)
   @END  // program's end
   0;JMP // infinite loop
```

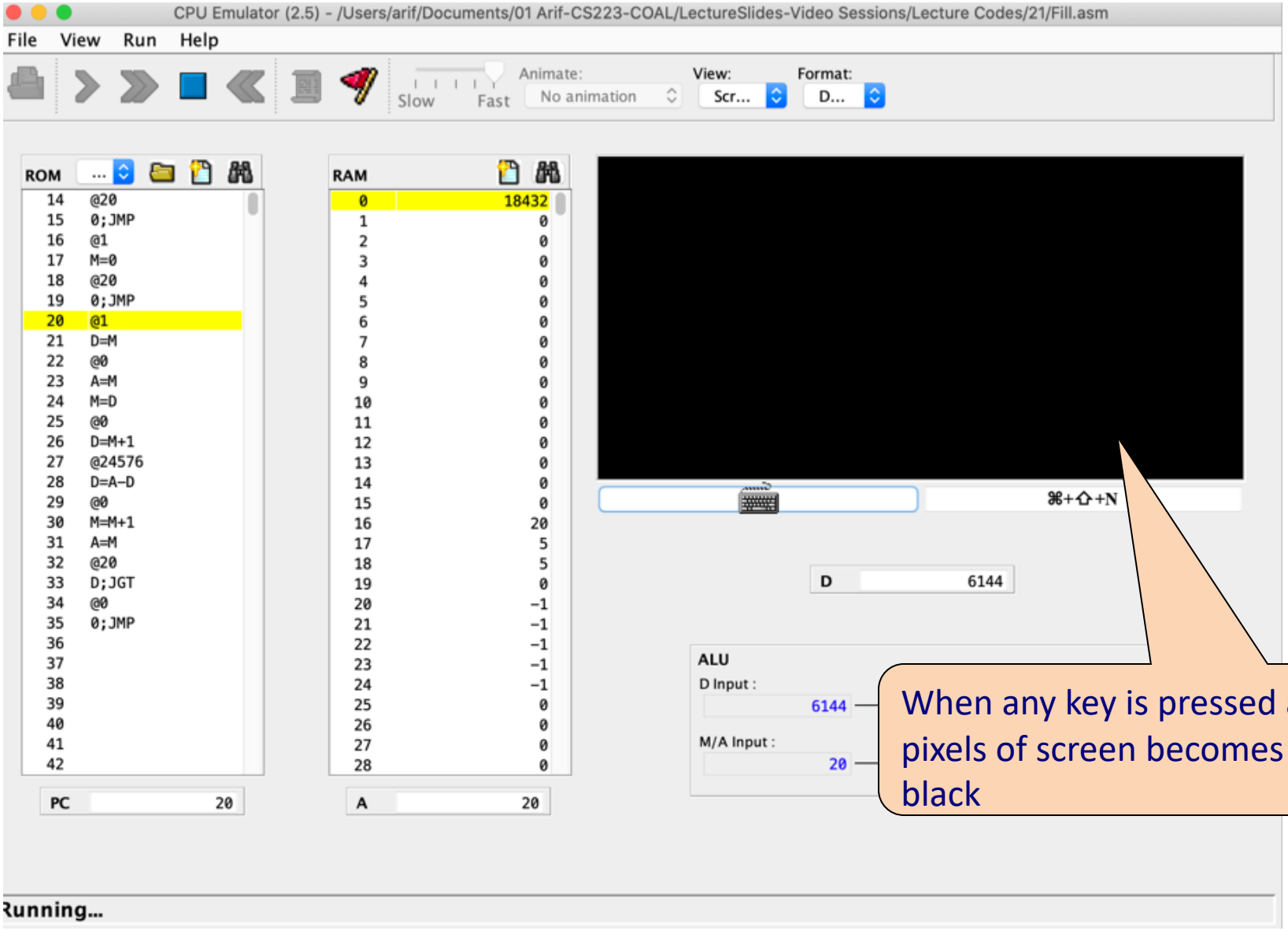
Drawing a Rectangle on The Screen

An Interactive Program

Fill: A Simple Interactive Program



Fill: A Simple Interactive Program



Fill: A Simple Interactive Program

Debugging

Breakpoints: A Powerful Debugging Tool

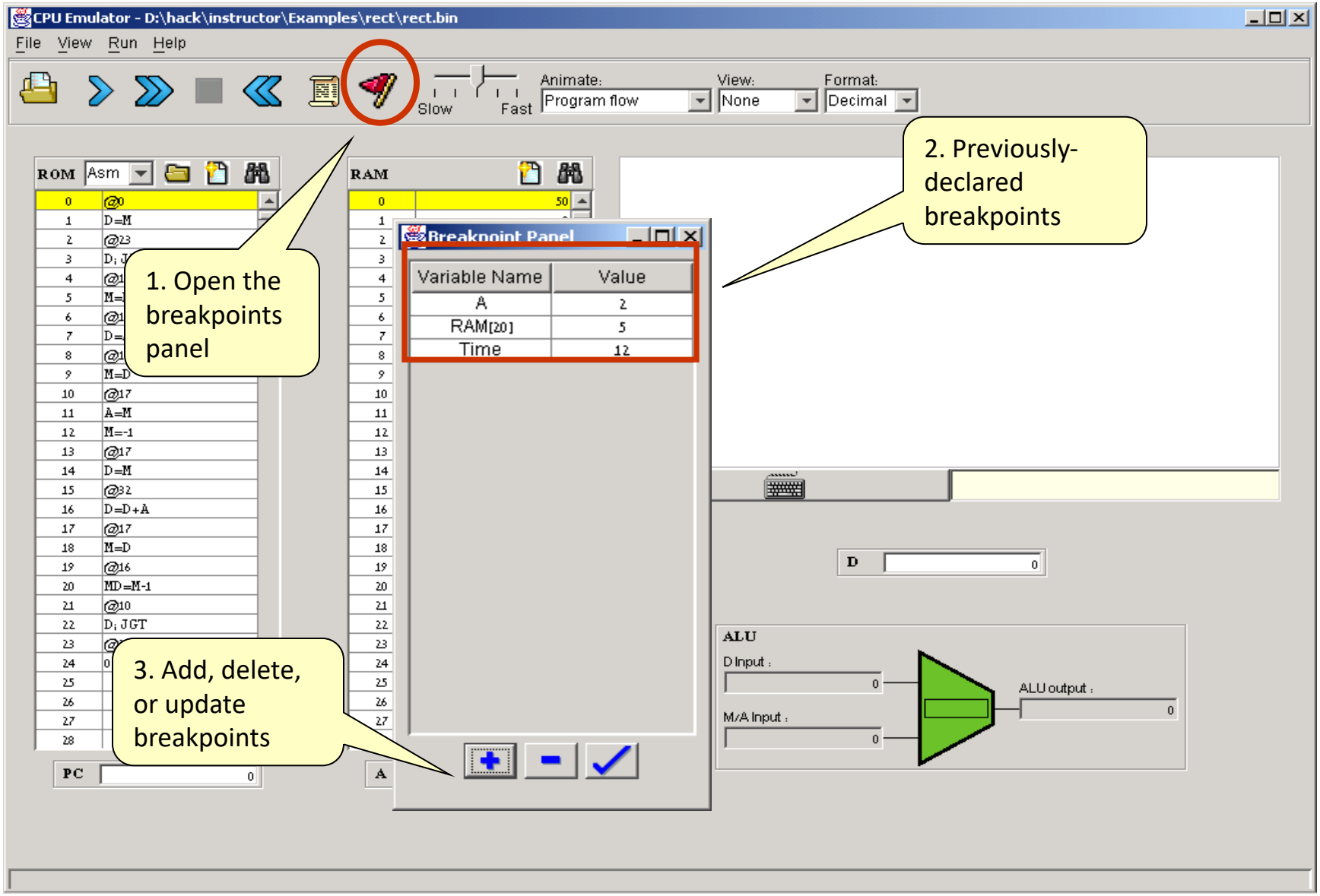
The CPU emulator continuously keeps track of:

- A: value of the A register
- D: value of the D register
- PC: value of the Program Counter
- RAM[i]: value of any RAM location
- time: number of elapsed machine cycles

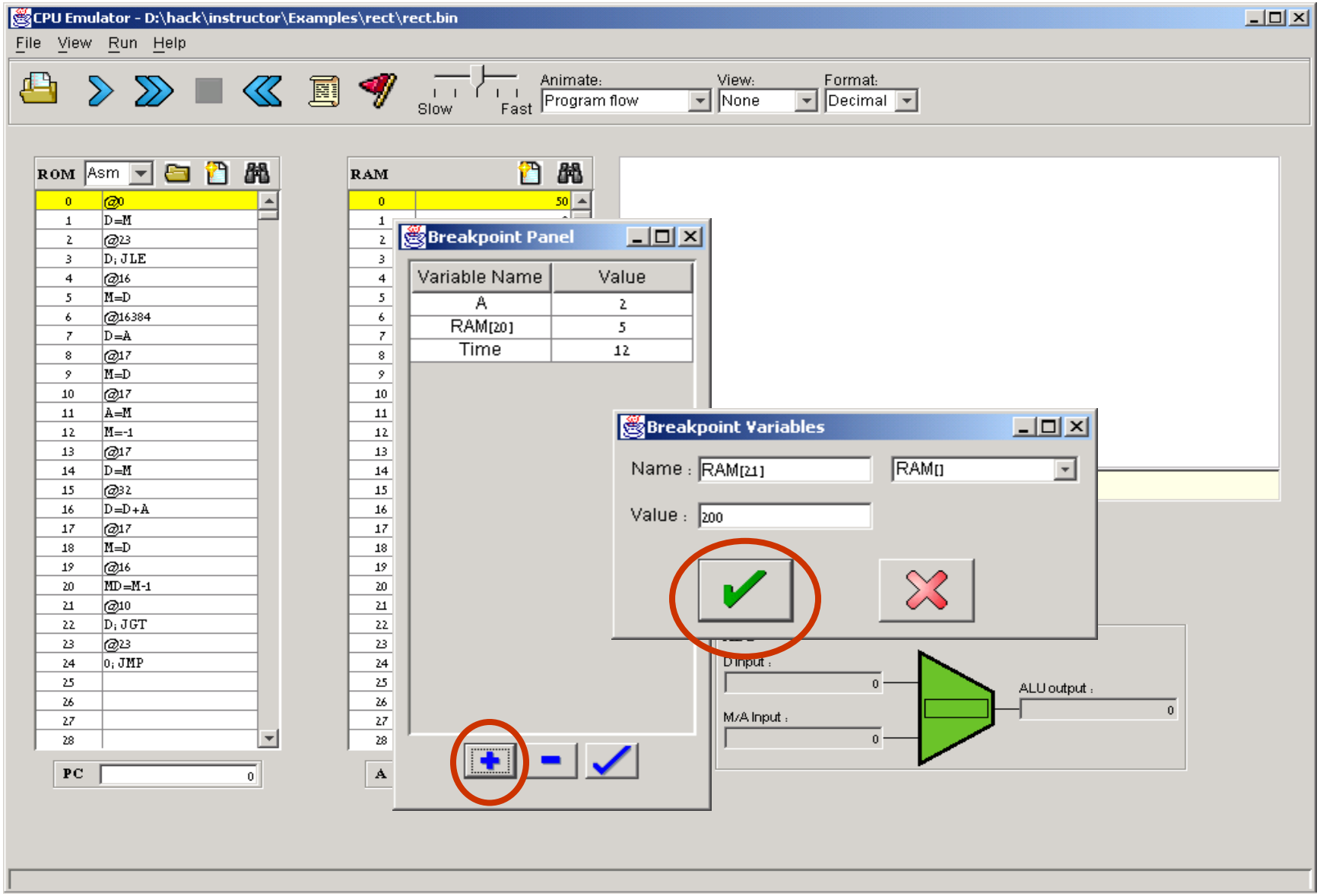
Breakpoints:

- A breakpoint is a pair <variable, value> where variable is one of {A, D, PC, RAM[i], time} and i is between 0 and 32K.
- Breakpoints can be declared either interactively, or via script commands.
- For each declared breakpoint, when the variable reaches the value, the emulator pauses the program's execution with a proper message.

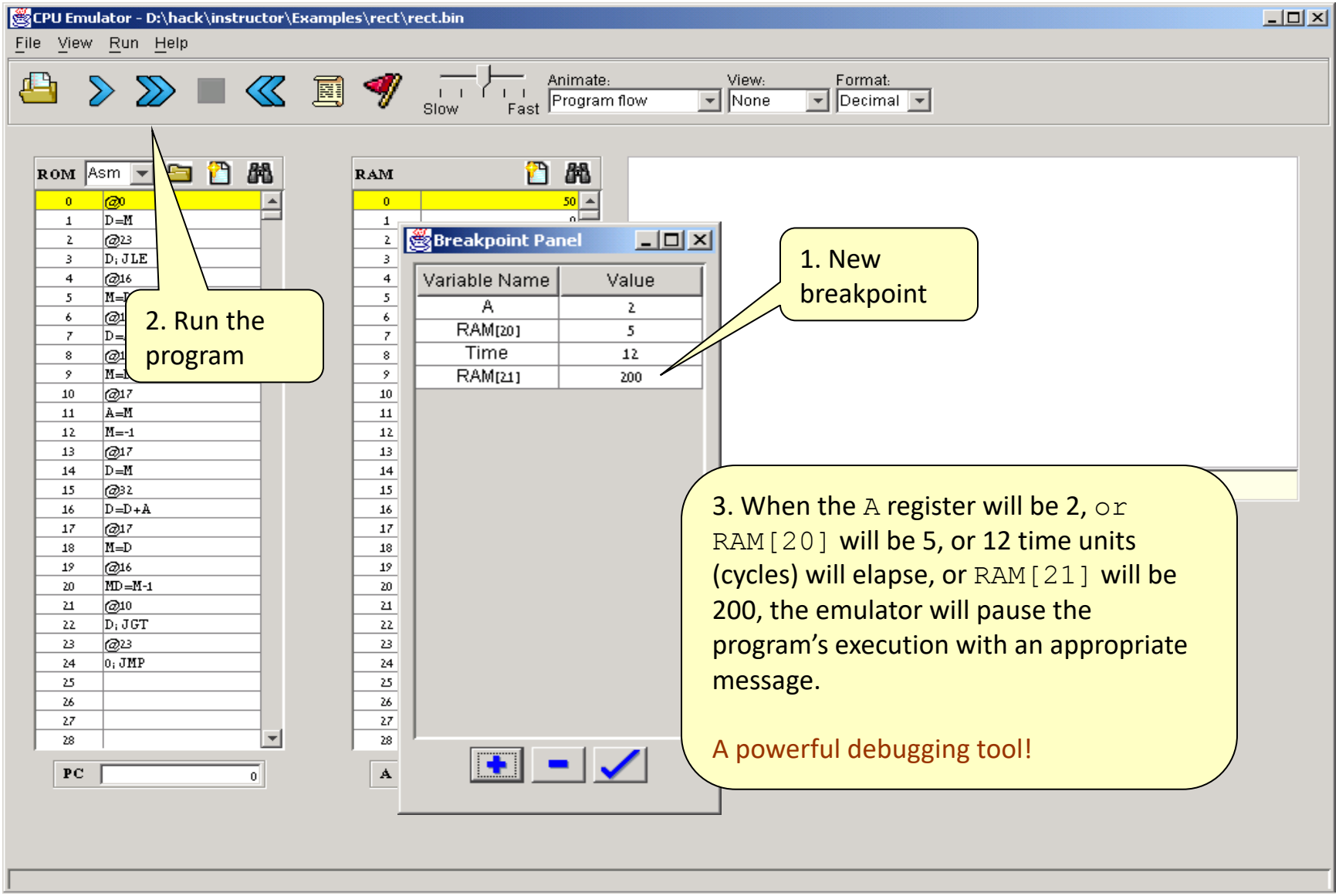
Breakpoints Declaration



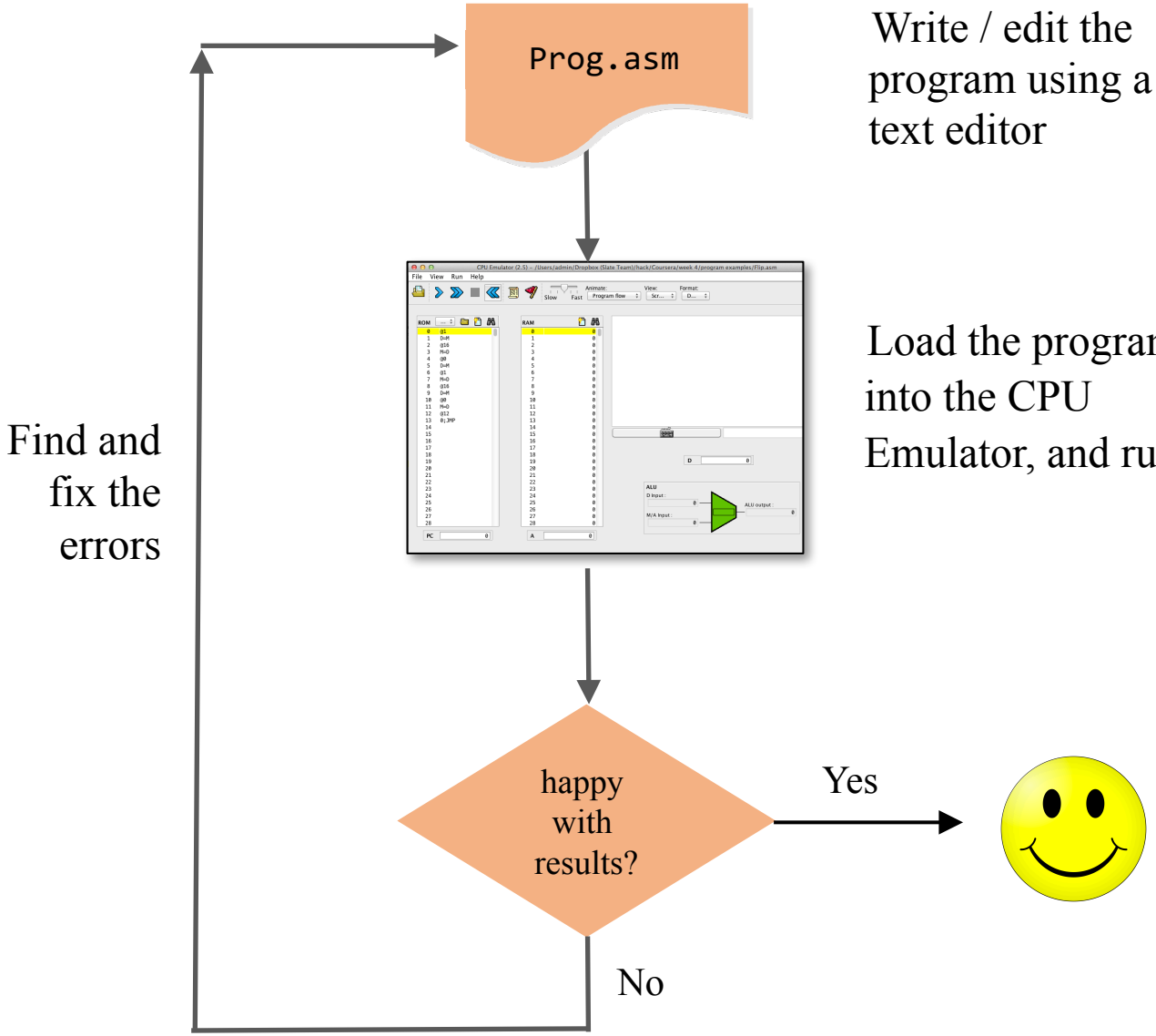
Breakpoints Declaration



Breakpoints Usage



Program Development Process



Best Practice

Well-written low-level code is

- Short
- Efficient
- Elegant
- Self-describing

Technical tips

- Use symbolic variables and labels
- Use sensible variable and label names
- Variables: lower-case
- Labels: upper-case
- Use indentation
- Start with pseudo code

Things To Do

- Download all the assembly program from the course bitbucket repository

https://bitbucket.org/arifpucit/coal/

make changes to them and execute them in the CPU Emulator

- Run the programs, one instruction at a time, do the working in your head or on a piece of paper, while executing the programs one instruction at a time

Coming to office hours does NOT mean you are academically week!