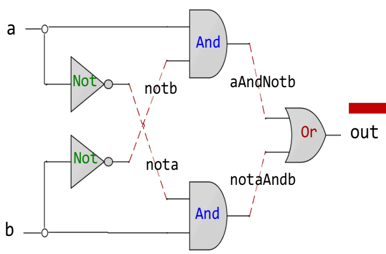
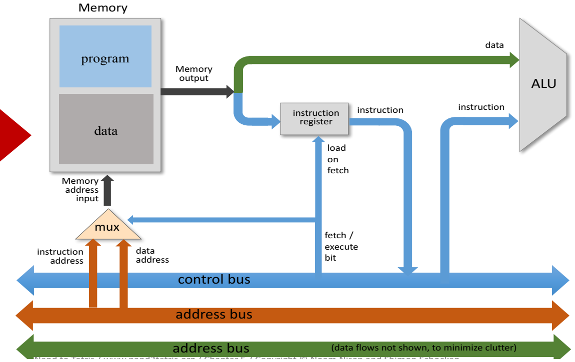
Digital Logic Design




```
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 # 25

Interfacing I/O Devices

```
#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
```

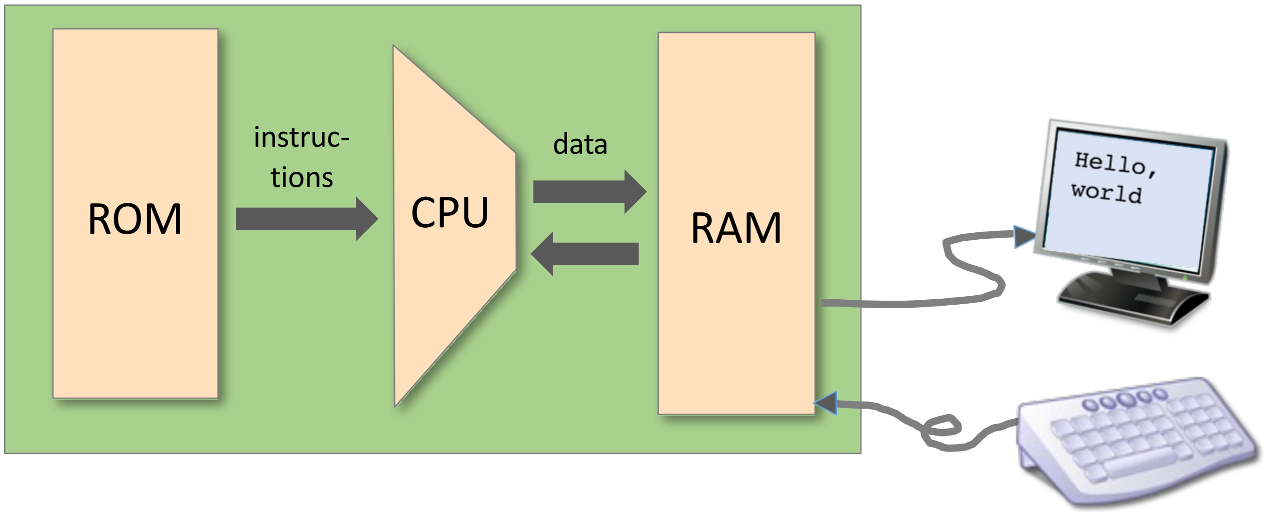
Slides of first half of the course are adapted from:
https://www.nand2tetris.org
Download s/w tools required for first half of the course from the following link:
https://drive.google.com/file/d/0B9c0BdDJz6XpZUh3X2dPR1o0MUE/view

Instructor: Muhammad Arif Butt, Ph.D.

Today's Agenda

- How to interface I/O devices with computer
- Interfacing Screen with Hack computer
  - Demo of built-in Screen chip on h/w Simulator
- Interfacing Keyboard with Hack computer
  - Demo of built-in Keyboard chip on h/w Simulator
- Assembly Programming with Screen using CPU Emulator
- Assembly Programming with KBD using CPU Emulator

Input / Output



Screen: used to display outputs

Keyboard: used to get data from user

I/O Handling

- **High Level Approach:** Sophisticated software library functions are used to display text/graphics on the monitor, read the keyboard, read voice notes from mic and play the audio on speakers etc
- **Low Level:** Bits Manipulation

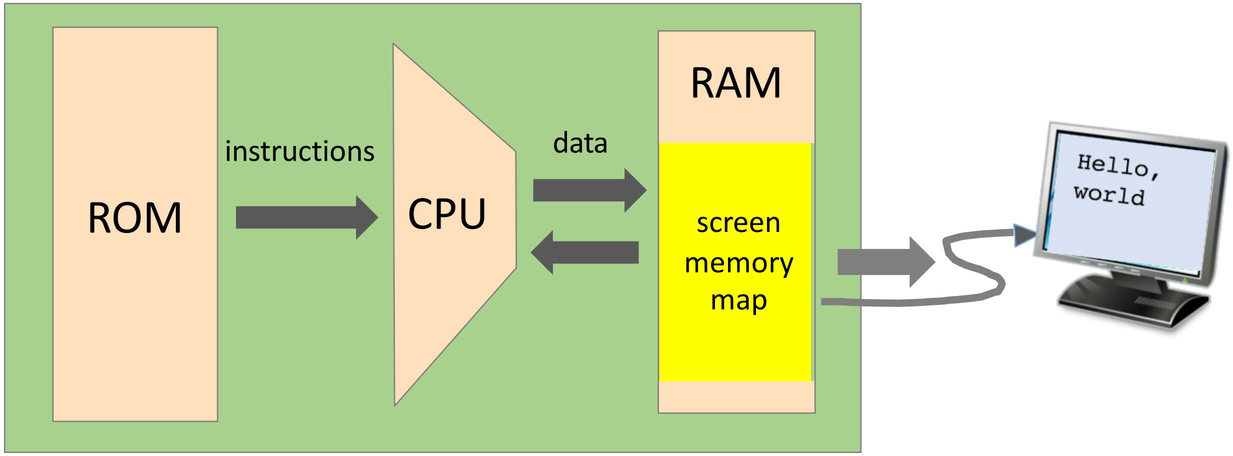
Interfacing I/O Devices with a Computer

- The way a microprocessor need to read/write different memory locations, similarly the microprocessor also need to read/write different I/O devices like the keyboard, mouse, monitor, printer, etc. This linking is also be called I/O Interfacing. An I/O interface acts as a communication channel between the processor and the externally interfaced device. The interfacing of the I/O devices can be done in two ways
  - **Memory Mapped I/O Interfacing:** Both memory and I/O devices have same address space. So addressing capability of memory become less because some part is occupied by the I/O. In memory mapped I/O, there are same read-write instructions for memory and I/O devices, so CPUs are cheaper, faster and easier to build. Example is Hack CPU
  - **Isolated I/O Interfacing:** The I/O devices are given a separate addressing region (separate from the memory). These separate address spaces are known as 'Ports'. In isolated I/O, there are different read-write instructions for memory and I/O devices. x86-64 use Isolated I/O

Note: Data can be transferred between CPU and I/O devices in three modes, namely Program controlled I/O, Interrupt initiated I/O, and Direct Memory Access

Interfacing Screen with Hack Computer

Memory Mapped Output

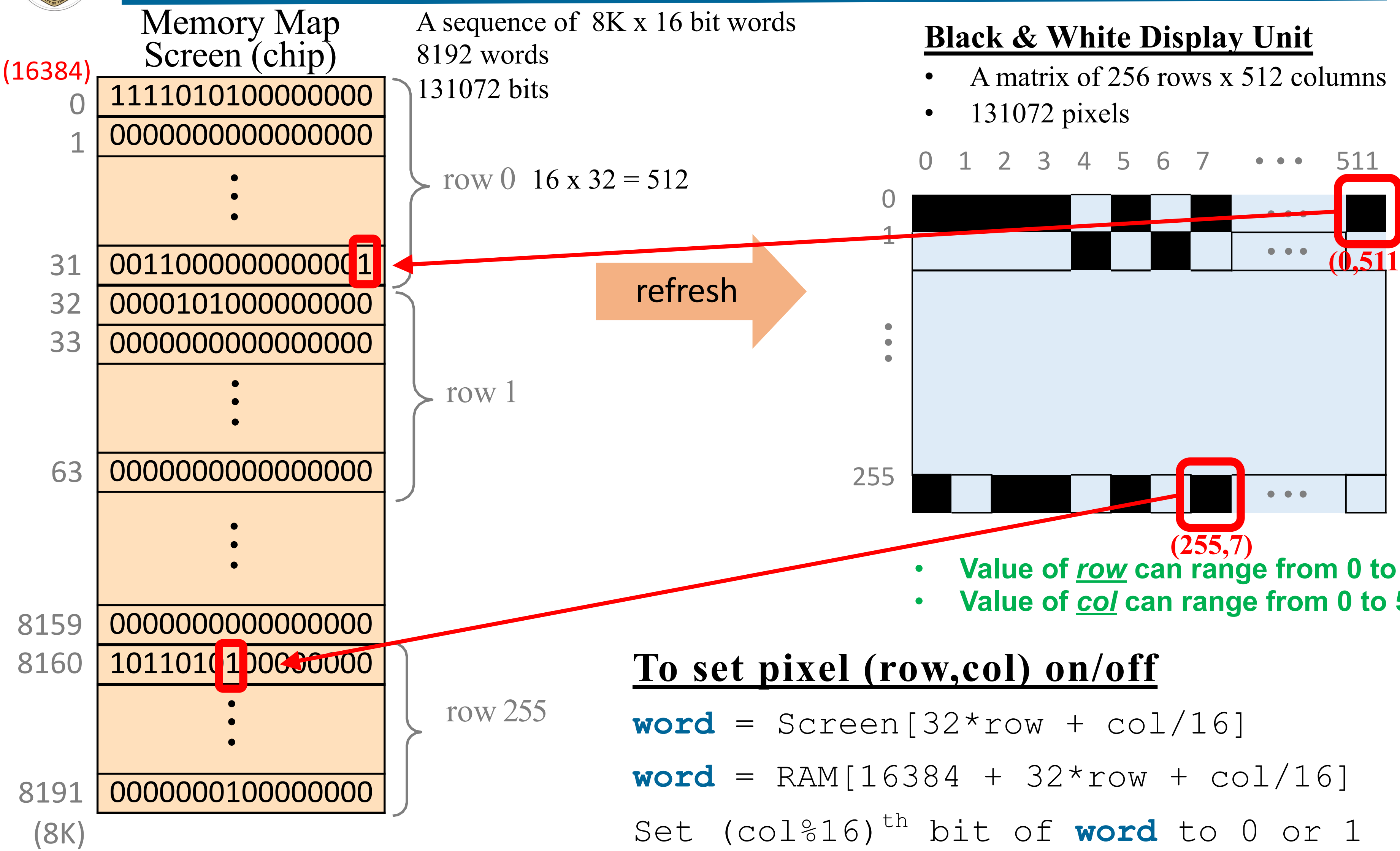


Screen Memory Map:

- Screen memory map is a designated memory area, dedicated to manage a display unit
- To write something on the display unit, write some bits in the designated memory area (zero to make a pixel off/white and one to make a pixel on/black)
- The physical display is continuously *refreshed* from the contents of memory map, many times per second
- Whatever, we write in the memory map makes the corresponding pixels of screen black and white in the next refresh cycle
- This is how we can write "Hello World" message on the screen

Screen Memory Map

Screen Memory Map (1D MM to 2D Screen)

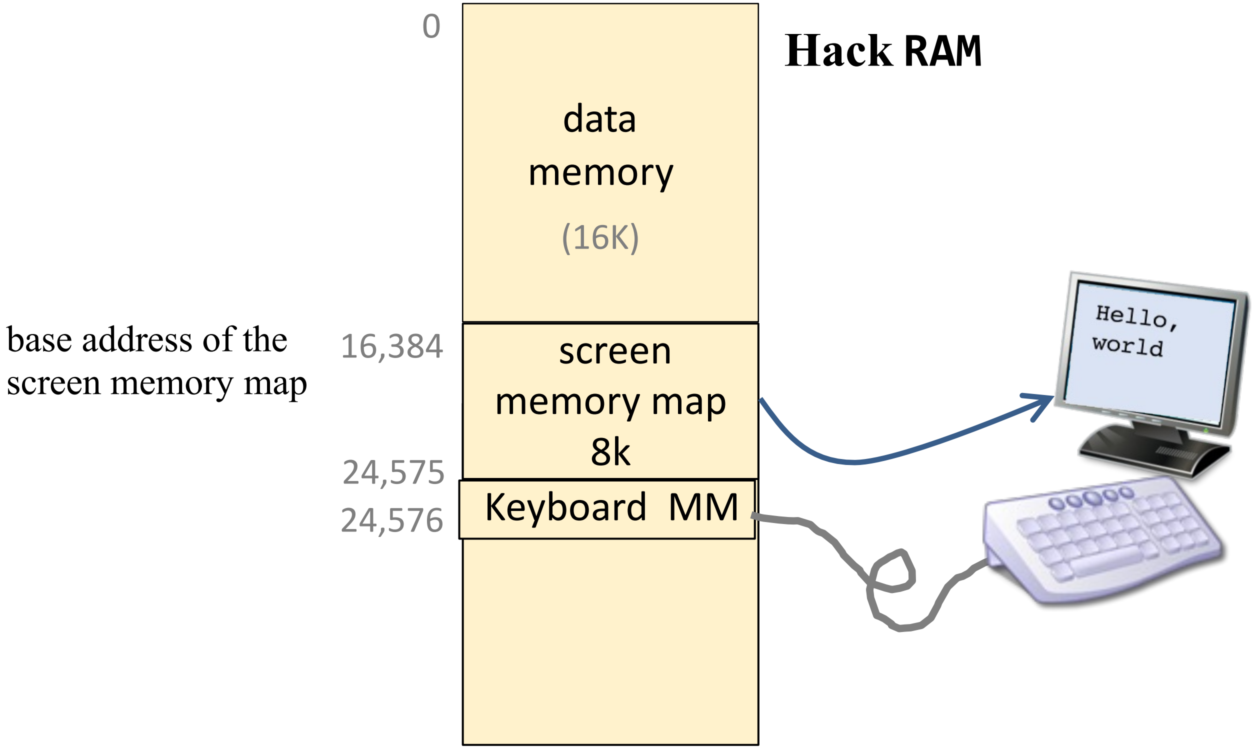


- Value of ***row*** can range from 0 to 255
- Value of ***col*** can range from 0 to 511

To set pixel (row,col) on/off

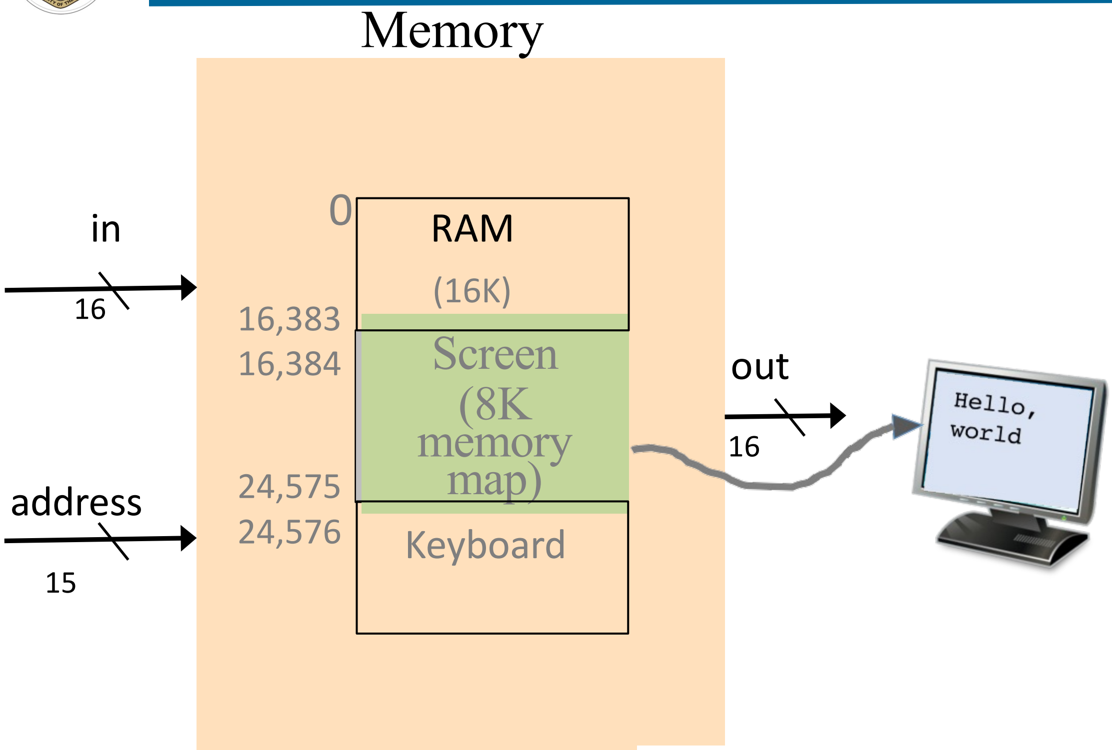
```
word = Screen[32*row + col/16]
word = RAM[16384 + 32*row + col/16]
Set (col%16)th bit of word to 0 or 1
RAM[i] = word
```

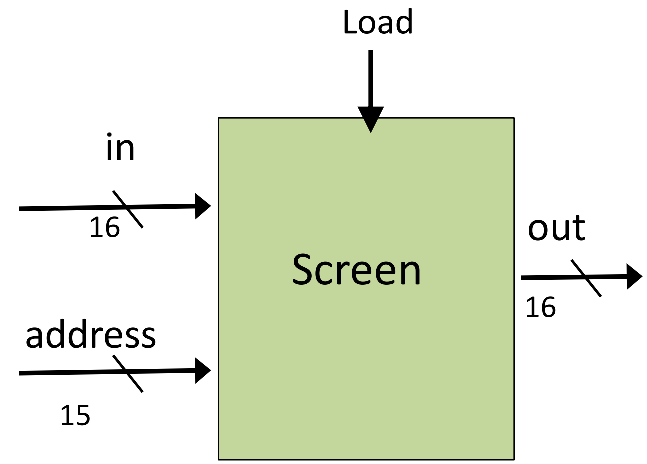
Output



- The physical screen is of 256 rows and 512 columns which makes 256 x 512 = 131072 pixels
- To map each pixel of screen on a single bit, the Screen memory map must contain 8K, 16 bits words, which makes 8192 x 16 = 131072 bits
- The built-in chip implementation has the side effect of continuously refreshing a visual 256 by 512 black-and-white screen, simulated by the simulator. Each row in the visual screen is represented by 32 consecutive 16-bit words, starting at the top left corner of the visual screen.

Screen Built-in Chip





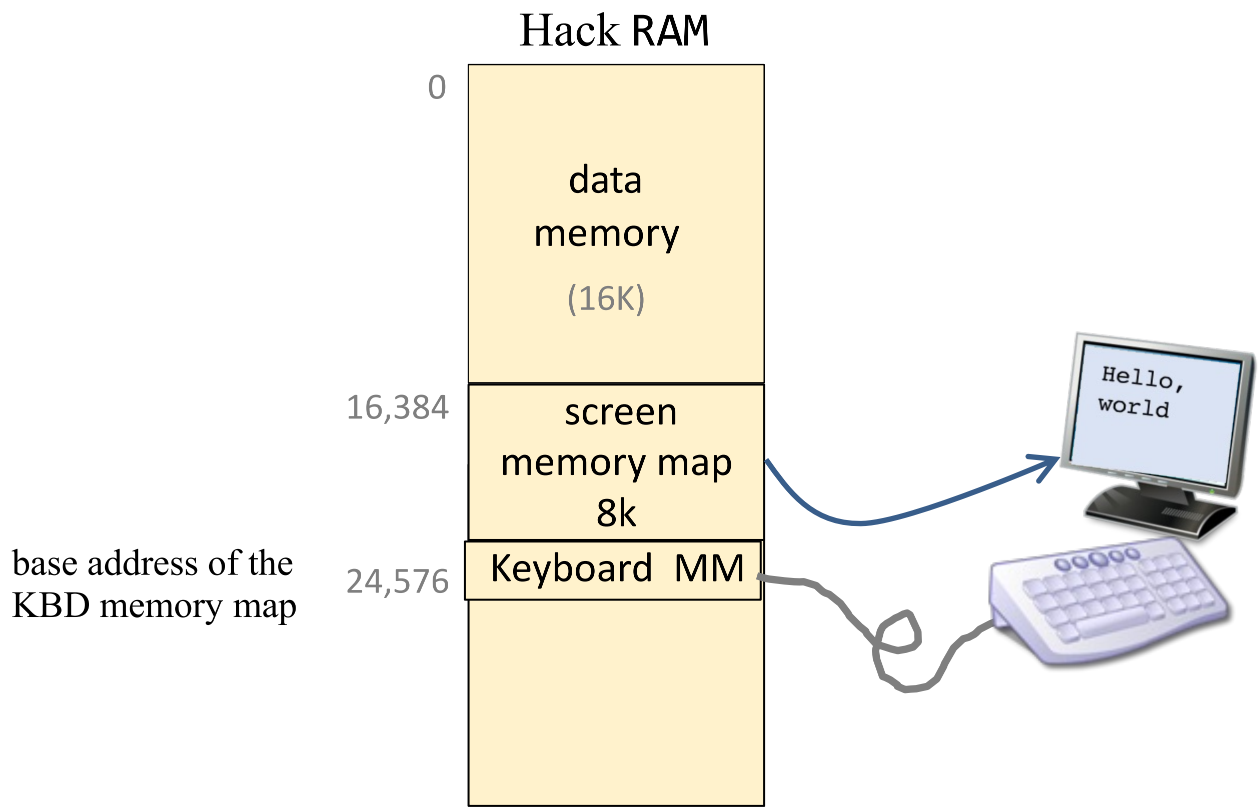
The built-in chip implementation has the side effect of continuously refreshing a visual 256 by 512 black-and-white screen, simulated by the simulator

```
CHIP Screen {
   IN in[16],    // what to write
   load,          // write-enable bit
   address[13]; // where to read/write
   OUT out[16]; // Screen value at the given address
   BUILTIN Screen;
   CLOCKED in, load;
}
```

Screen Output Demo

Interfacing Keyboard with Hack Computer

Input



- The physical keyboard requires just one word inside the Hack Memory, as it will contain the ASCII code of the character pressed on keyboard.
- So the 16 bit word of Hack RAM at address 24576 is where the keyboard is mapped.

The Hacker Character Set

key	code
(space)	32
!	33
“	34
#	35
$	36
%	37
&	38
‘	39
(	40
)	41
*	42
+	43
,	44
-	45
.	46
/	47

key	code
0	48
1	49
…	…
9	57

:	58
;	59
<	60
=	61
>	62
?	63
@	64

key	code
A	65
B	66
C	…
…	…
Z	90

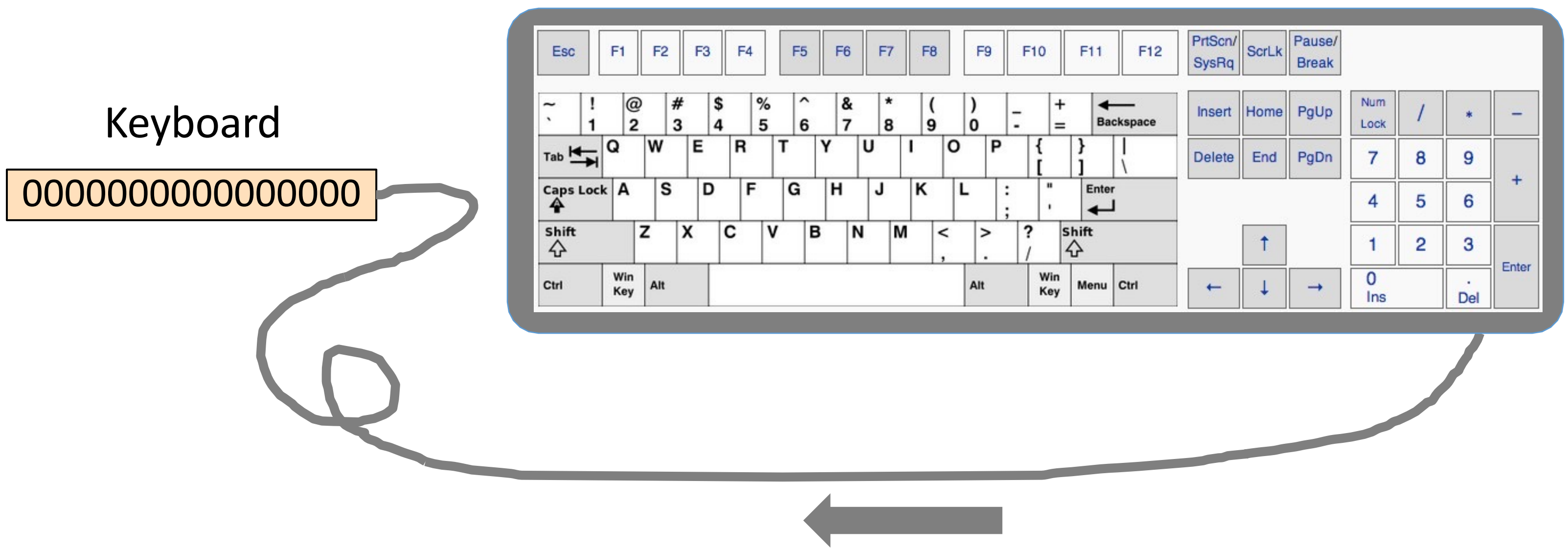
[	91
/	92
]	93
^	94
_	95
`	96

key	code
a	97
b	98
c	99
…	…
z	122

{	123
|	124
}	125
~	126

key	code
newline	128
backspace	129
left arrow	130
up arrow	131
right arrow	132
down arrow	133
home	134
end	135
Page up	136
Page down	137
insert	138
delete	139
esc	140
f1	141
…	…
f12	152

Memory Mapped Input

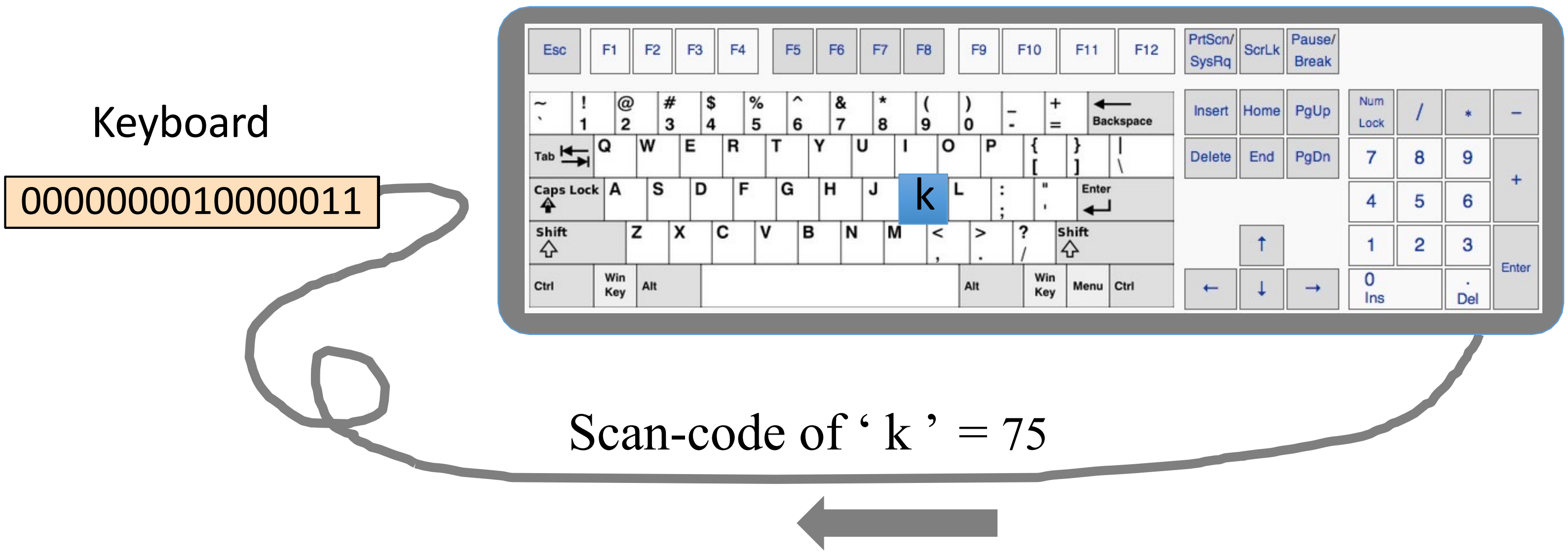


When a key is pressed on the keyboard, the key's scan code appears in the keyboard memory map. Since no key is being pressed on the keyboard in this figure, so the keyboard memory map contains all zeros

<u>To check which key is currently pressed:</u>

- Probe the contents of the Keyboard chip
- In the Hack computer: probe the contents of RAM[24576]

Memory Mapped Input



When a key is pressed on the keyboard, the key’s scan code appears in the keyboard memory map

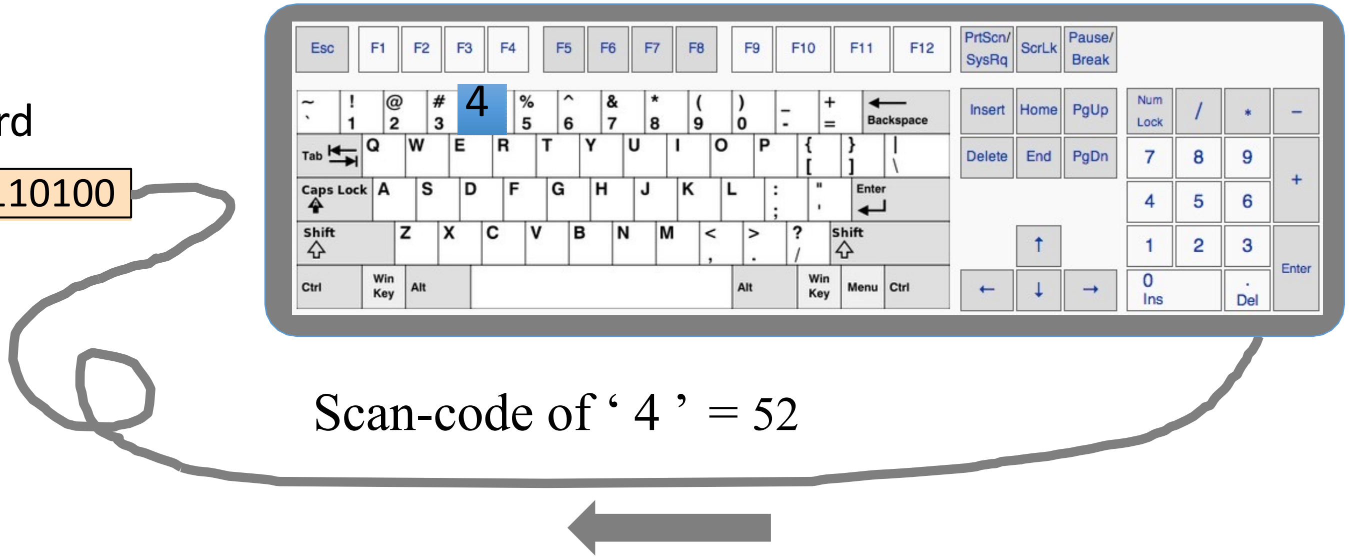
<u>To check which key is currently pressed:</u>

- Probe the contents of the Keyboard chip
- In the Hack computer: probe the contents of RAM[24576]

Memory Mapped Input

Keyboard

0000000000110100



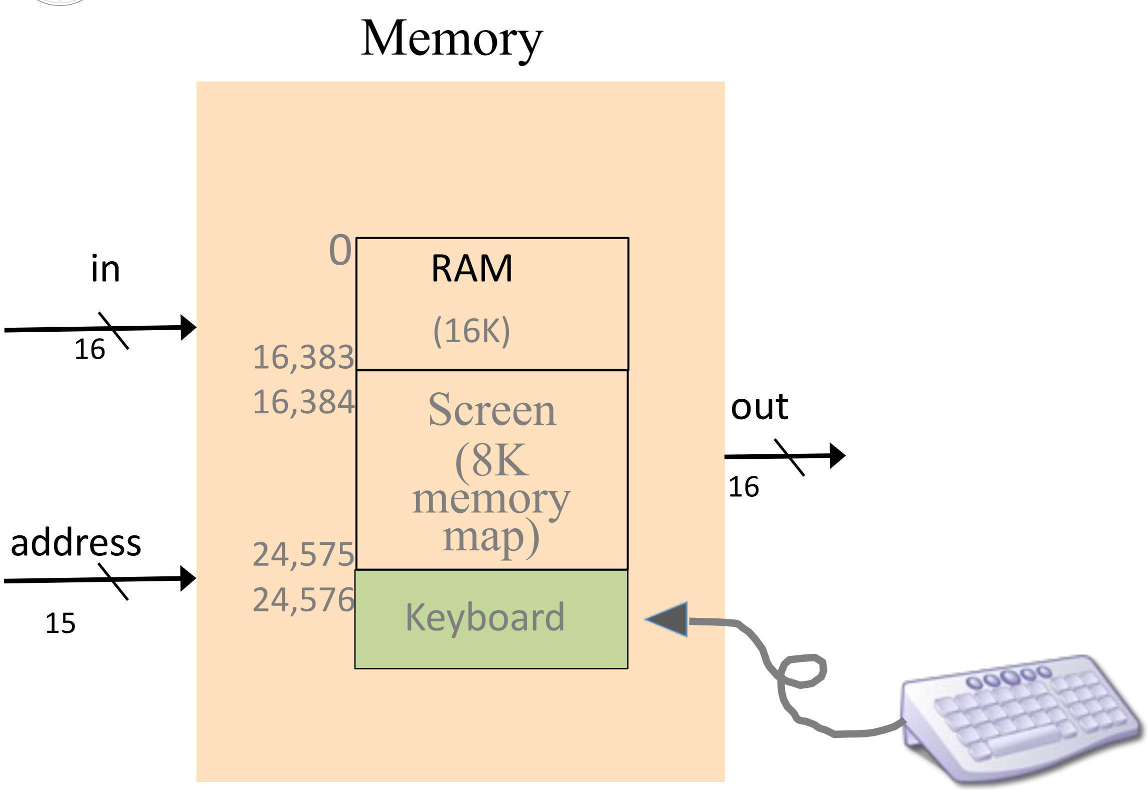
Scan-code of ‘ 4 ’ = 52

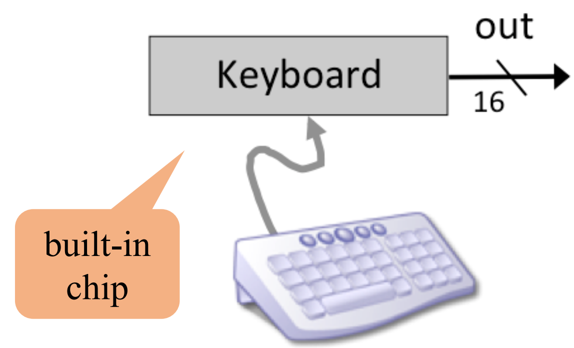
When a key is pressed on the keyboard, the key’s scan code appears in the keyboard memory map

To check which key is currently pressed:

- Probe the contents of the Keyboard chip
- In the Hack computer: probe the contents of RAM[24576]

Keyboard Built-in Chip



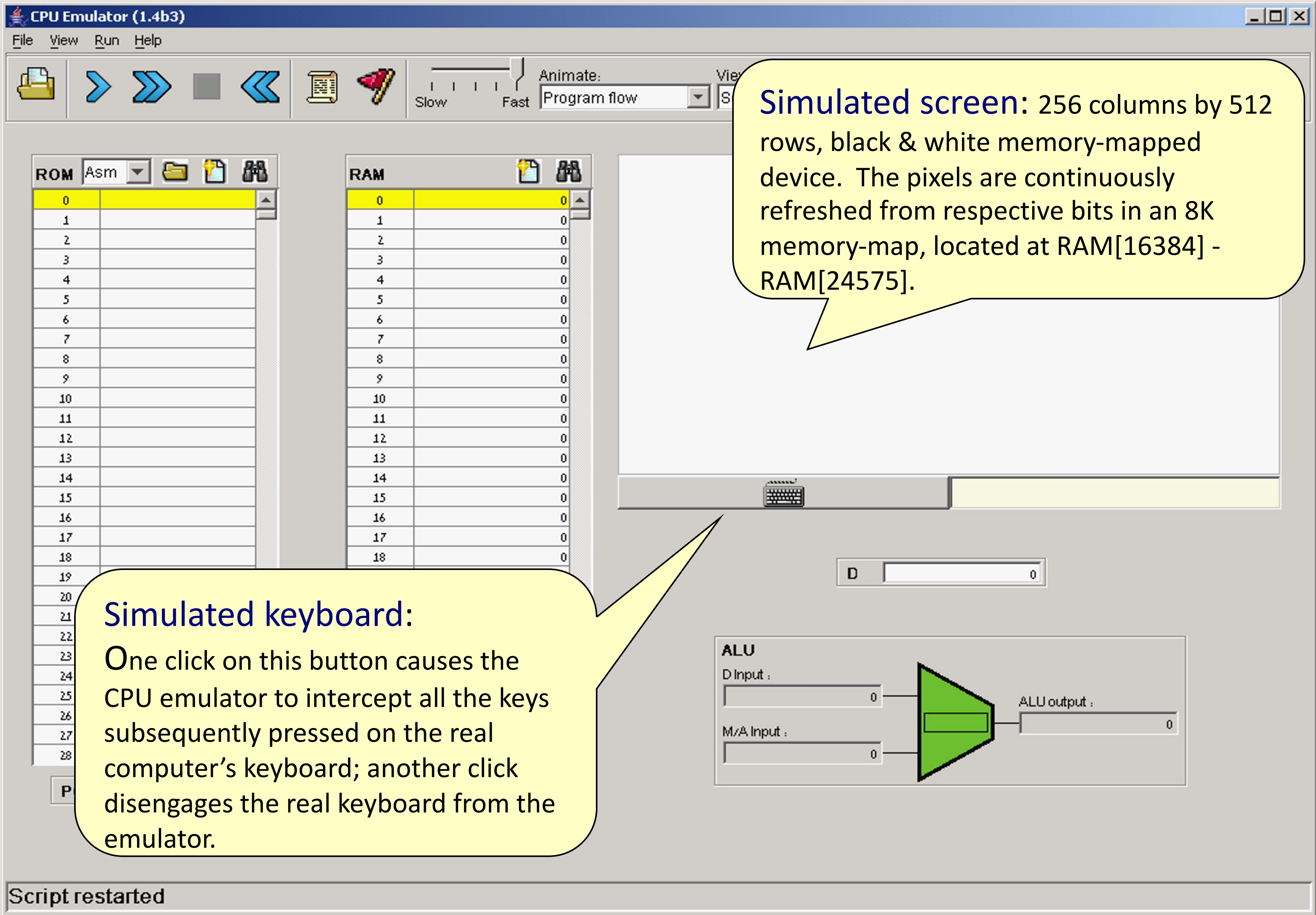



```
CHIP Keyboard {

   OUT out[16]; // The ASCII code of the pressed key,
                // or 0 if no key is currently pressed,
                // or one the special codes
   BUILTIN Keyboard;
}
```

Keyboard Input Demo

Hack Assembly Programming involving I/O on Hack CPU Emulator

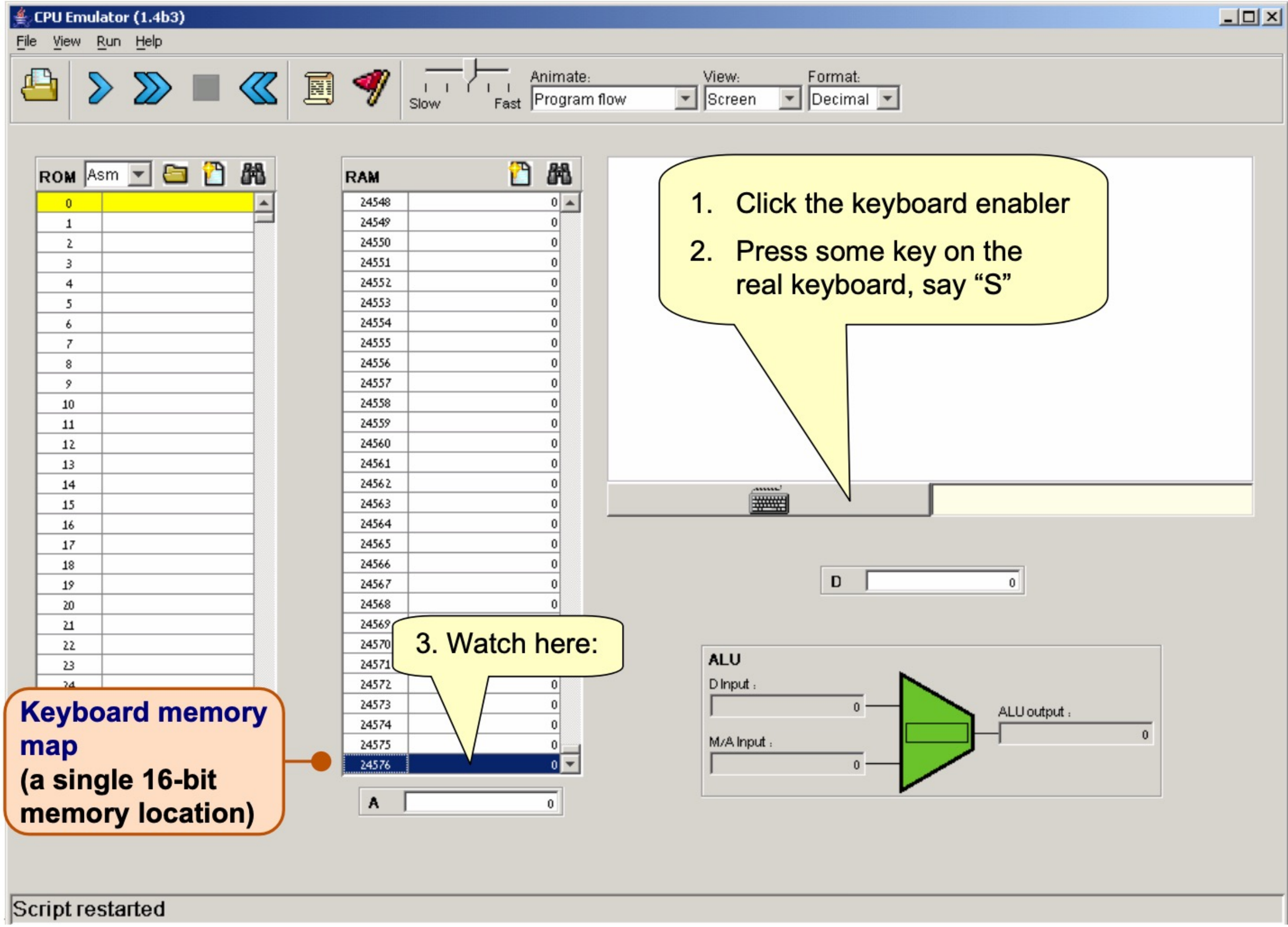
I/O Devices: Screen And Keyboard



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.

I/O Devices: Keyboard in Action

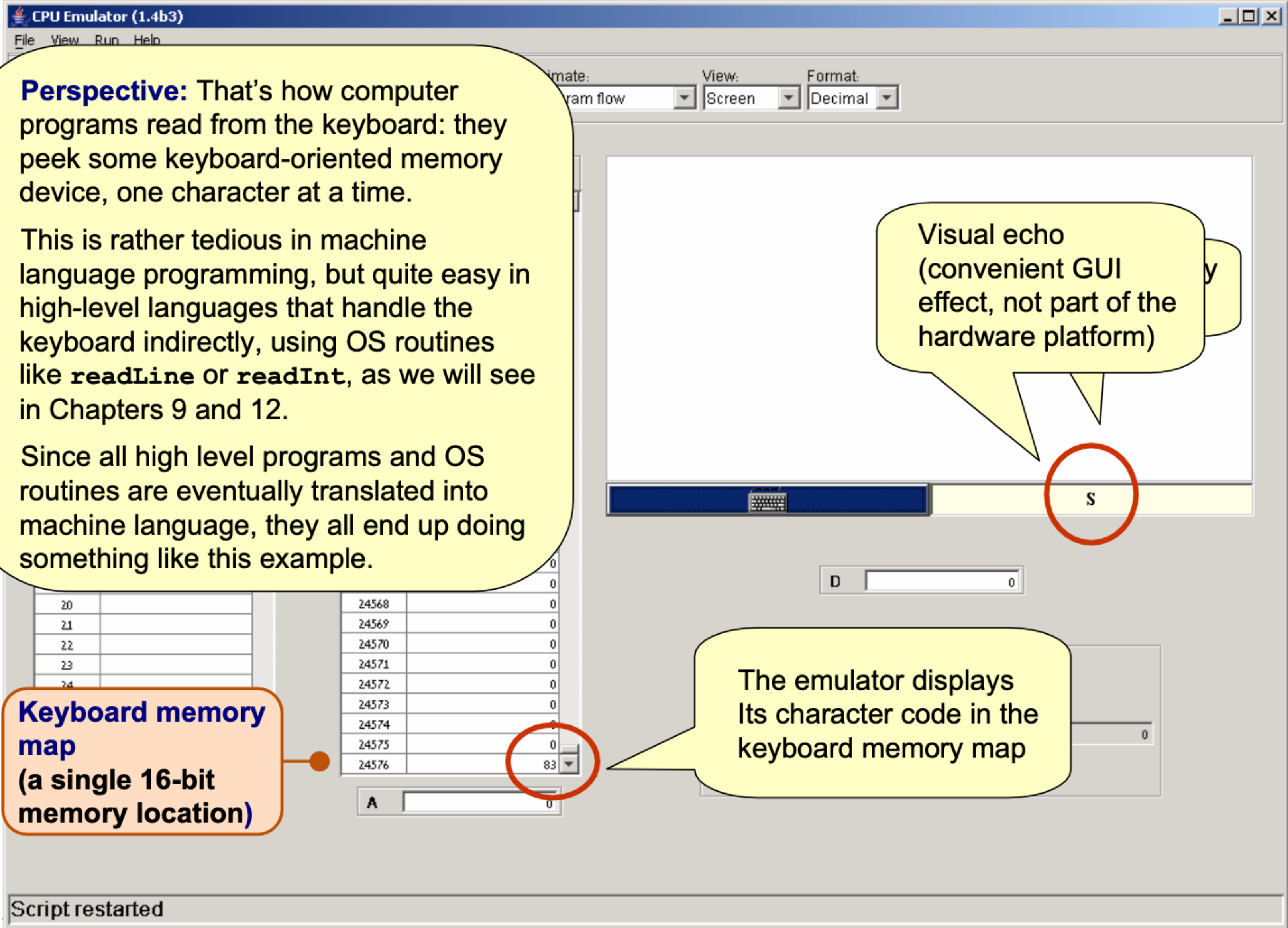


I/O Devices: Keyboard in Action

Perspective: That's how computer programs read from the keyboard: they peek some keyboard-oriented memory device, one character at a time.

This is rather tedious in machine language programming, but quite easy in high-level languages that handle the keyboard indirectly, using OS routines like **`readLine`** or **`readInt`**, as we will see in Chapters 9 and 12.

Since all high level programs and OS routines are eventually translated into machine language, they all end up doing something like this example.

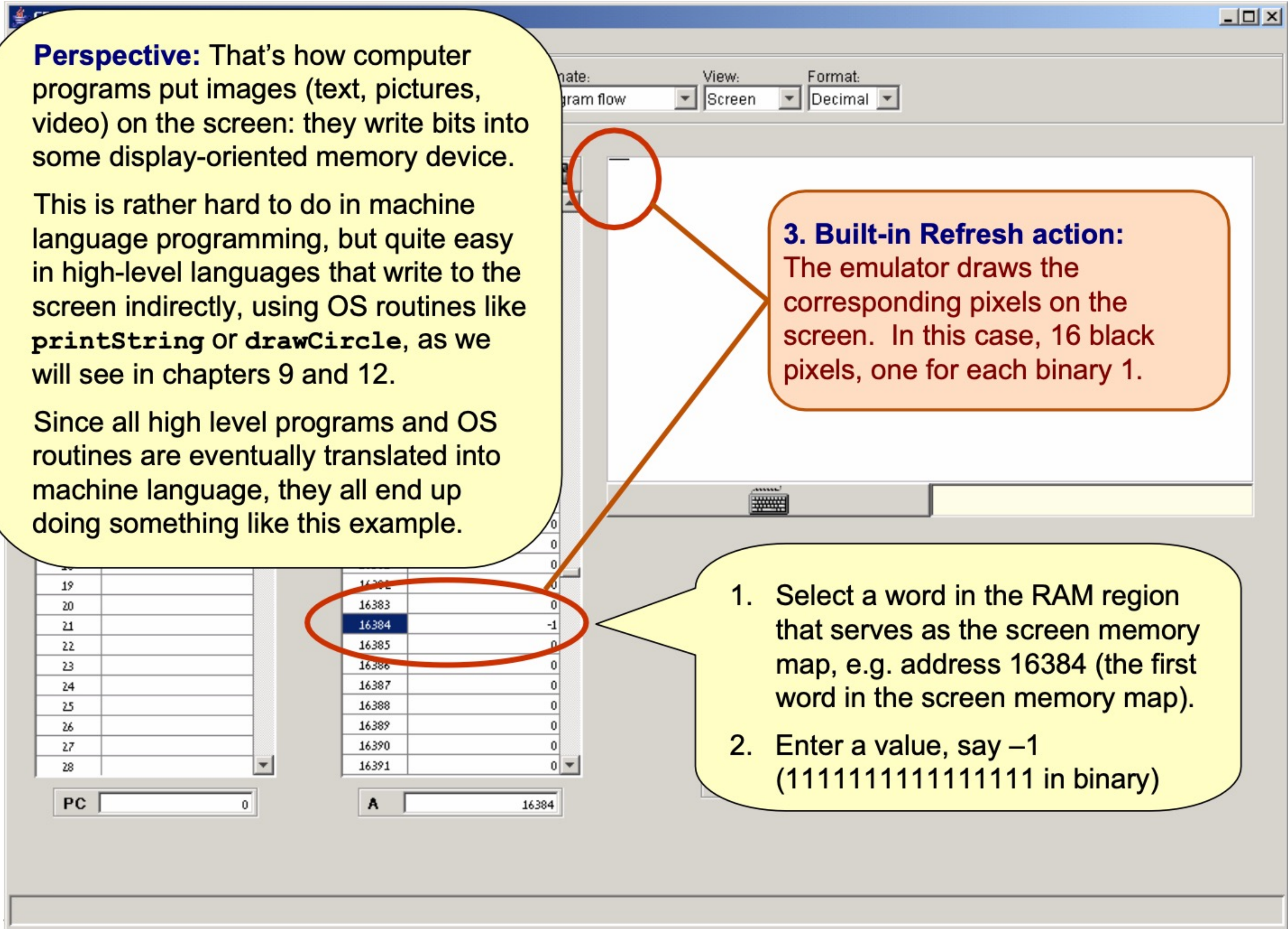


Visual echo (convenient GUI effect, not part of the hardware platform)

Keyboard memory map (a single 16-bit memory location)

The emulator displays Its character code in the keyboard memory map

I/O Devices: Screen in Action



Perspective: That's how computer programs put images (text, pictures, video) on the screen: they write bits into some display-oriented memory device.

This is rather hard to do in machine language programming, but quite easy in high-level languages that write to the screen indirectly, using OS routines like `printString` or `drawCircle`, as we will see in chapters 9 and 12.

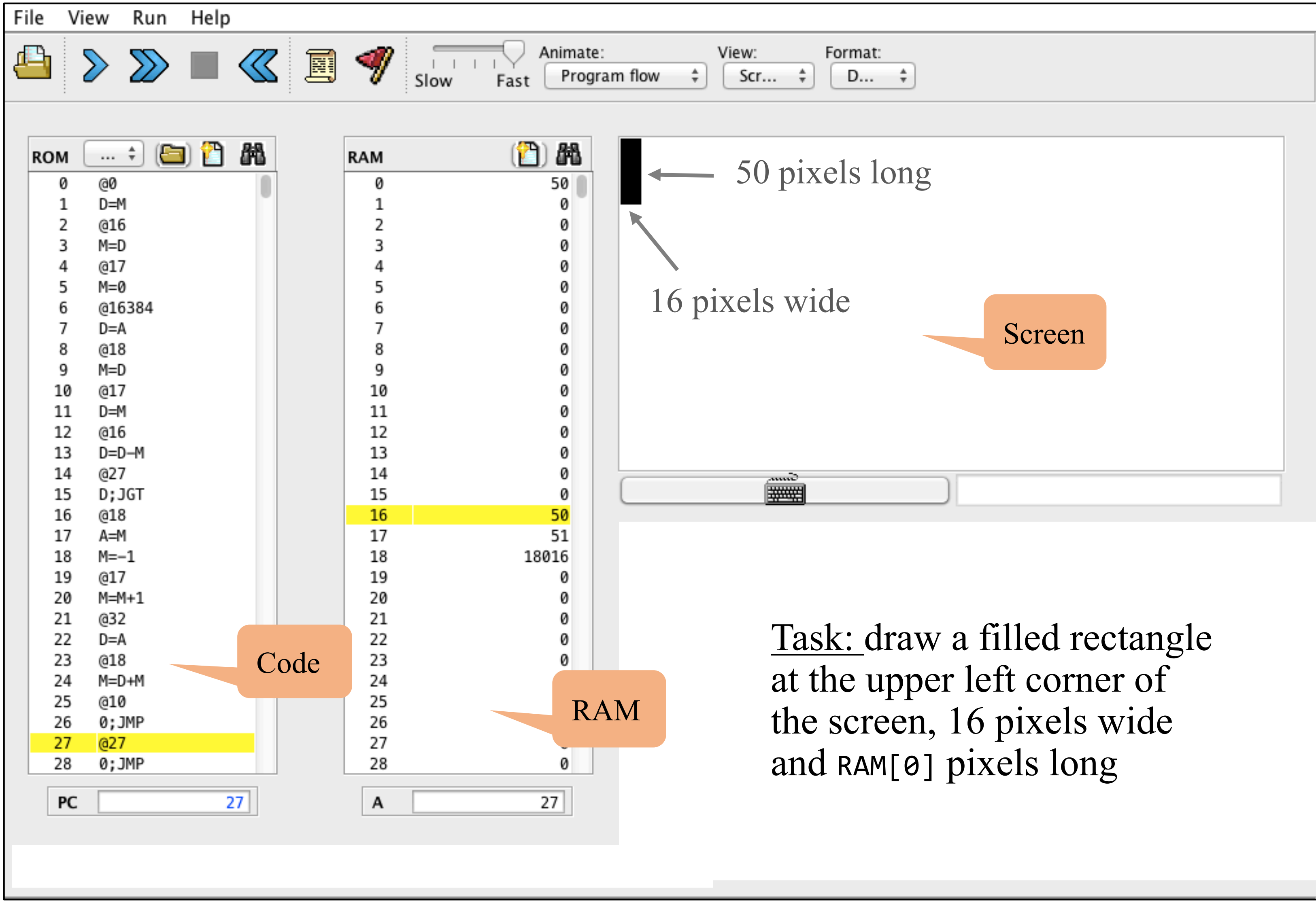
Since all high level programs and OS routines are eventually translated into machine language, they all end up doing something like this example.

1. Select a word in the RAM region that serves as the screen memory map, e.g. address 16384 (the first word in the screen memory map).
2. Enter a value, say –1 (1111111111111111 in binary)

3. Built-in Refresh action: The emulator draws the corresponding pixels on the screen. In this case, 16 black pixels, one for each binary 1.

Hack Assembly for Input & Output

Drawing a Rectangle on The Screen

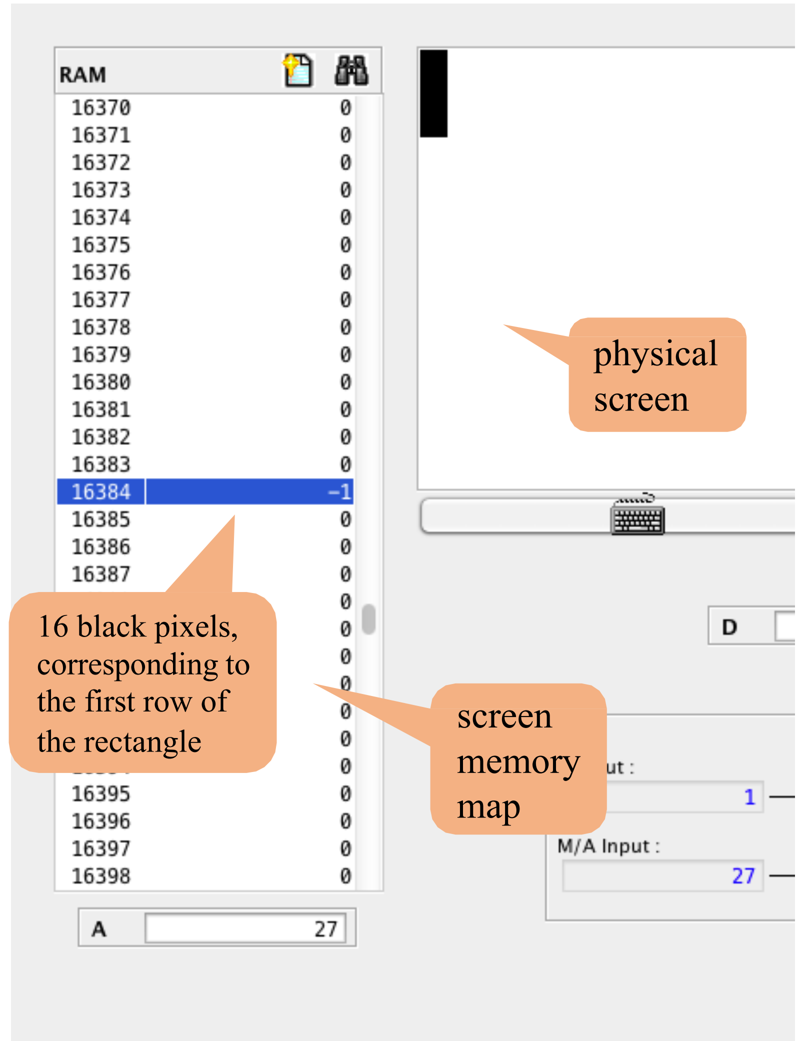


Task: draw a filled rectangle at the upper left corner of the screen, 16 pixels wide and `RAM[0]` pixels long

Drawing a Rectangle on The Screen

Pseudo code

```
 for (i=0; i<50; i++)
    draw 16 black pixels at the beginning of row i


 addr = 16384

 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
```



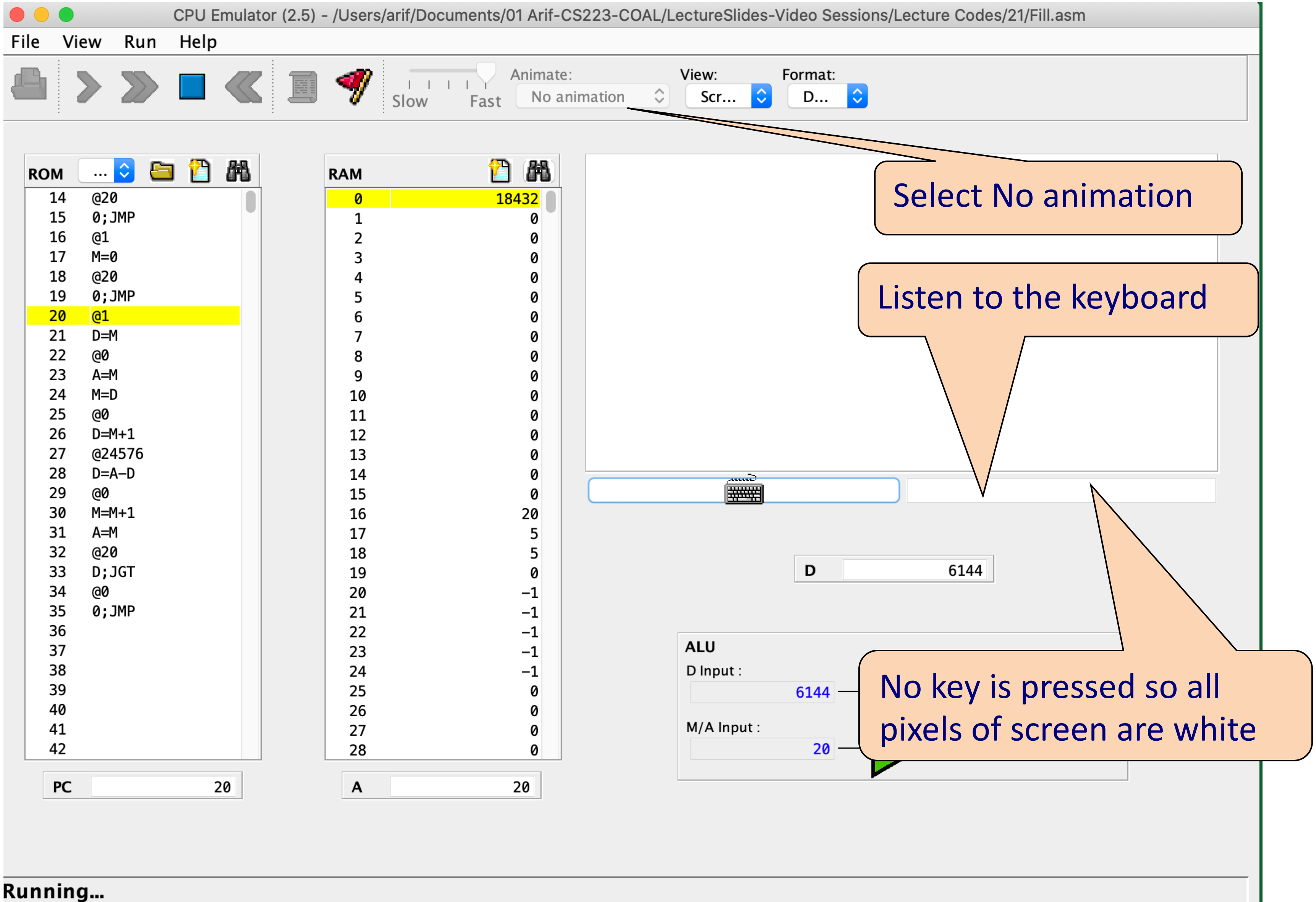
Drawing a Rectangle on The Screen

```
/* 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)
// ...
```

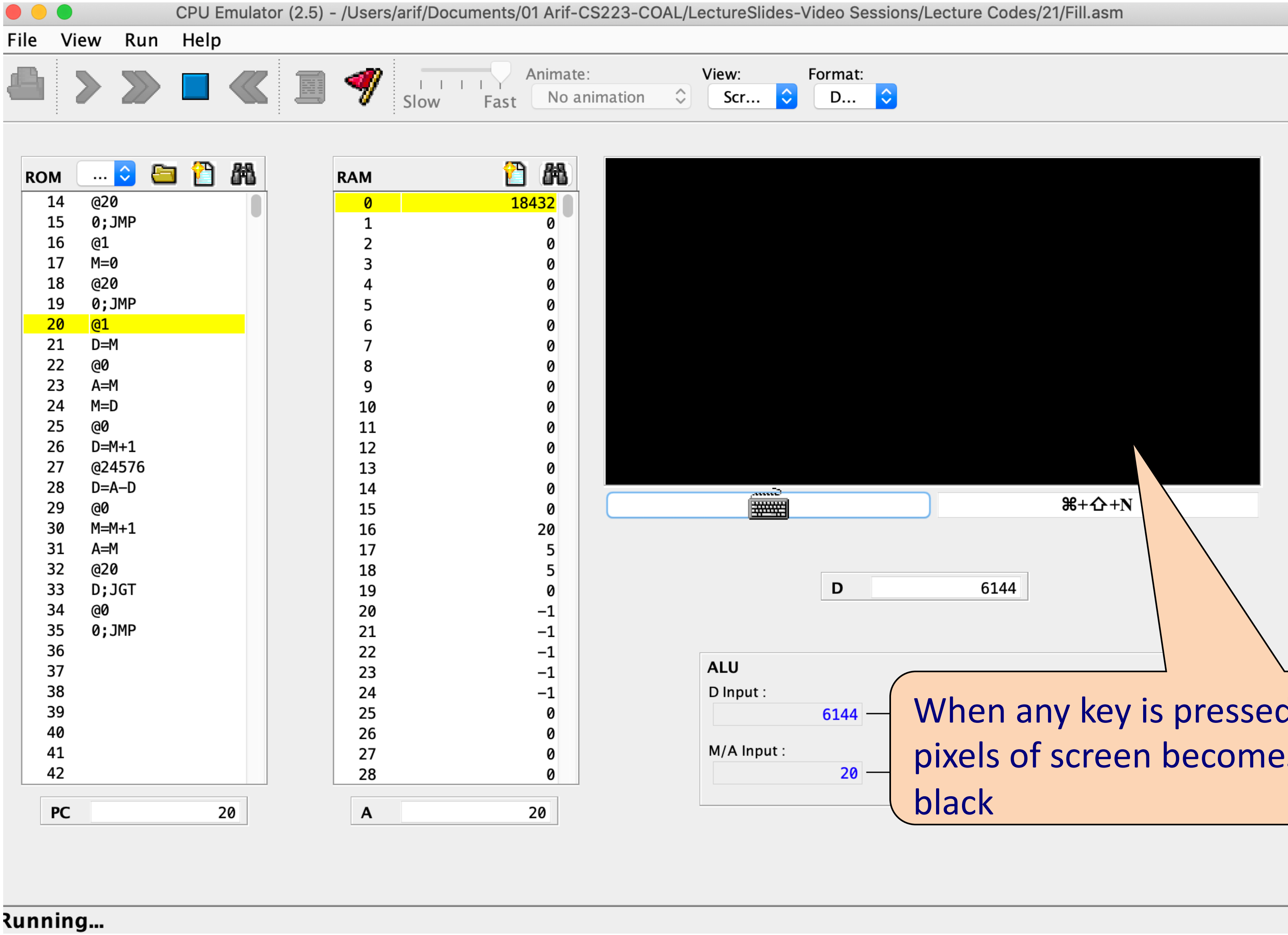
```
//...
(LOOP)
   @i
   D=M
   @n
   D=D-M
   @END
   D;JGT // if i>n goto END
   @addr
   A=M
   M=-1 //RAM[addr]=1111111111111111
   @i
   M=M+1 // i = i + 1
   @32
   D=A
   @addr
   M=D+M // addr = addr + 32
   @LOOP
   0;JMP // goto LOOP
(END)
   @END  // program's end
   0;JMP // infinite loop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

Drawing a Rectangle on The Screen

Example 2: `fill.asm`



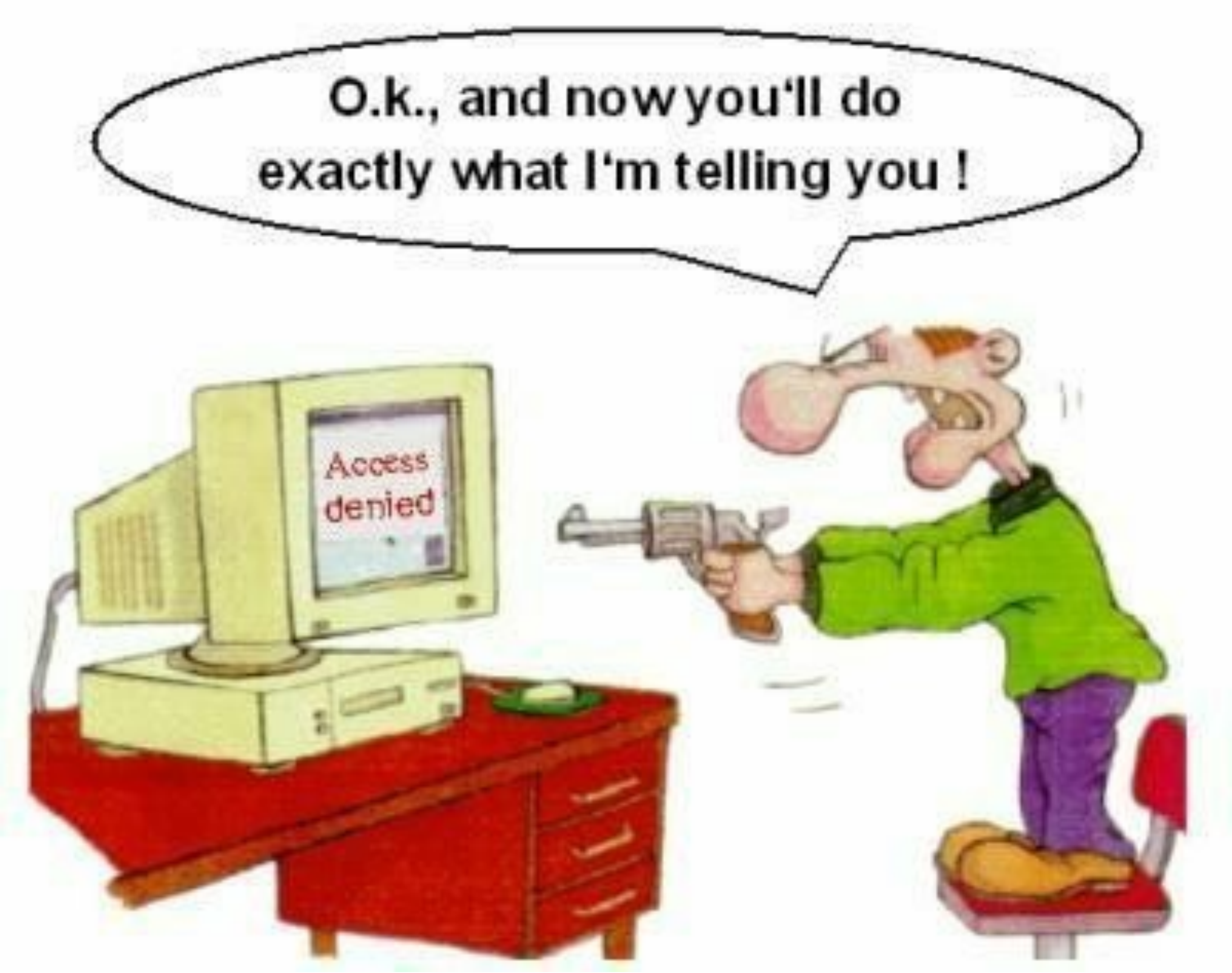
Example 2: `fill.asm`



When any key is pressed all pixels of screen becomes black

Fill: A Simple Interactive Program

Things To Do


O.k., and now you'll do exactly what I'm telling you !
Access denied