Introduction to Computer Science
RUNI

Lecture 6-1

Computer Fundamentals

Part I

Overview

Goals: Explore...

- How computers work
- Low-level programming
- The road from the low-level to the high level

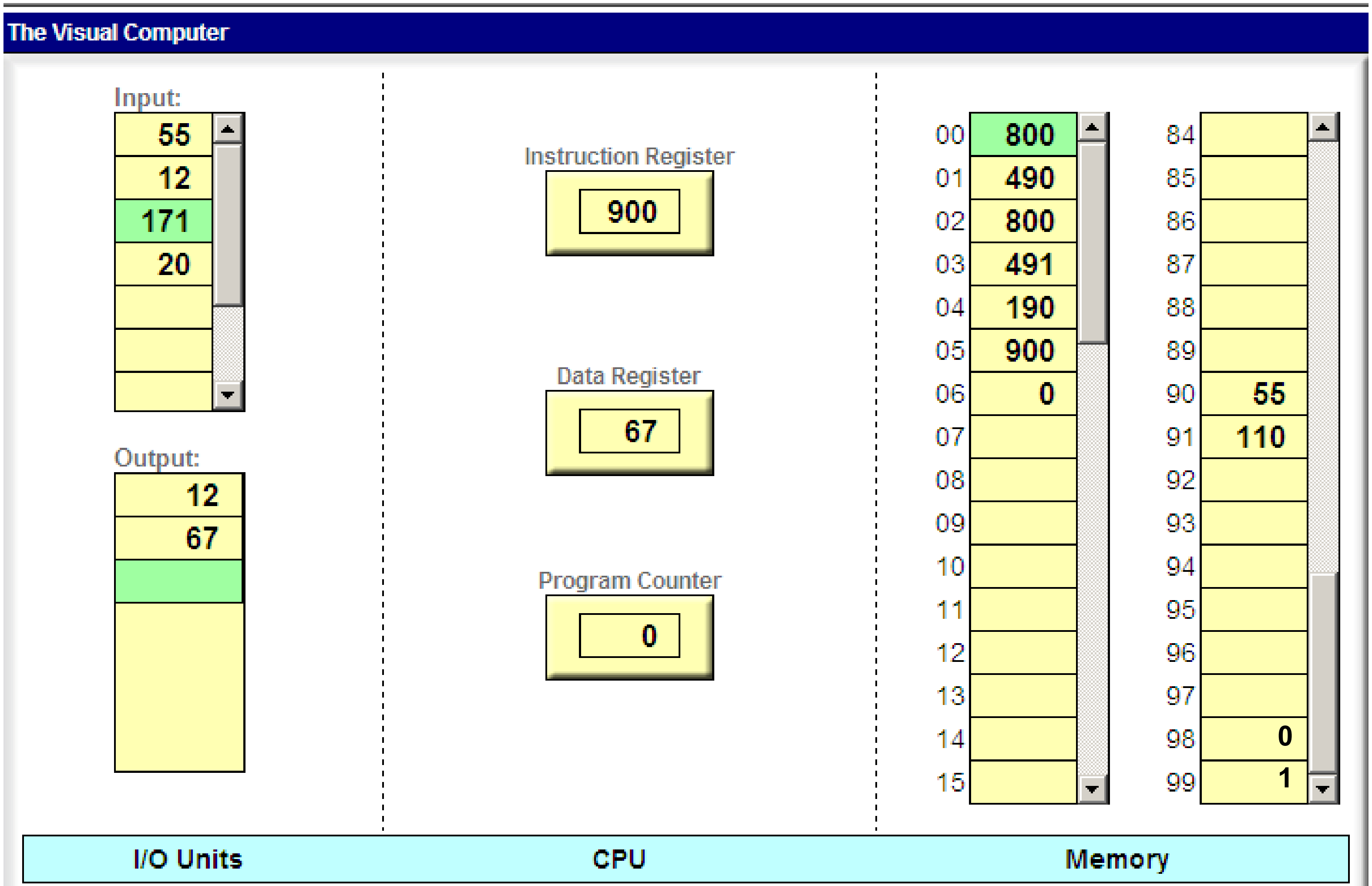
Approach

- Introduce a simple computer
- Write some machine language programs
- Understand the hardware / software interplay, hands-on.

Lecture plan

- Computer (Vic)
  - Architecture
  - Instructions
- Low-level programming
  - Basic
  - Branching
- Control
- Program translation
- From Vic to a real computer

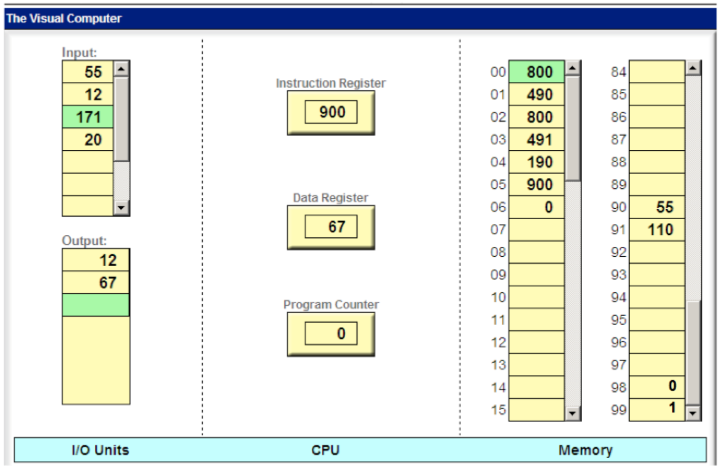
Vic: a simple computer architecture



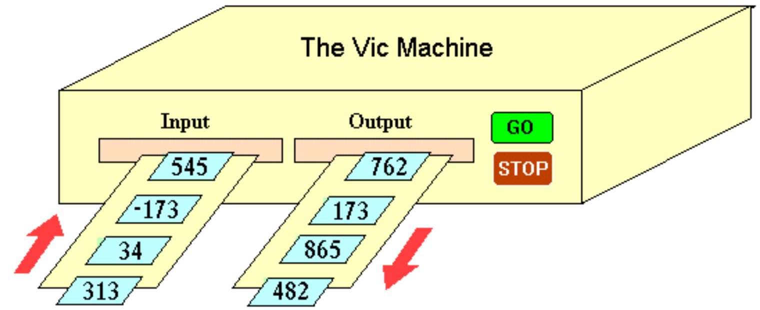
https://faculty.runi.ac.il/vic/software/computer/?lang=en

Vic: a simple computer architecture

inside view



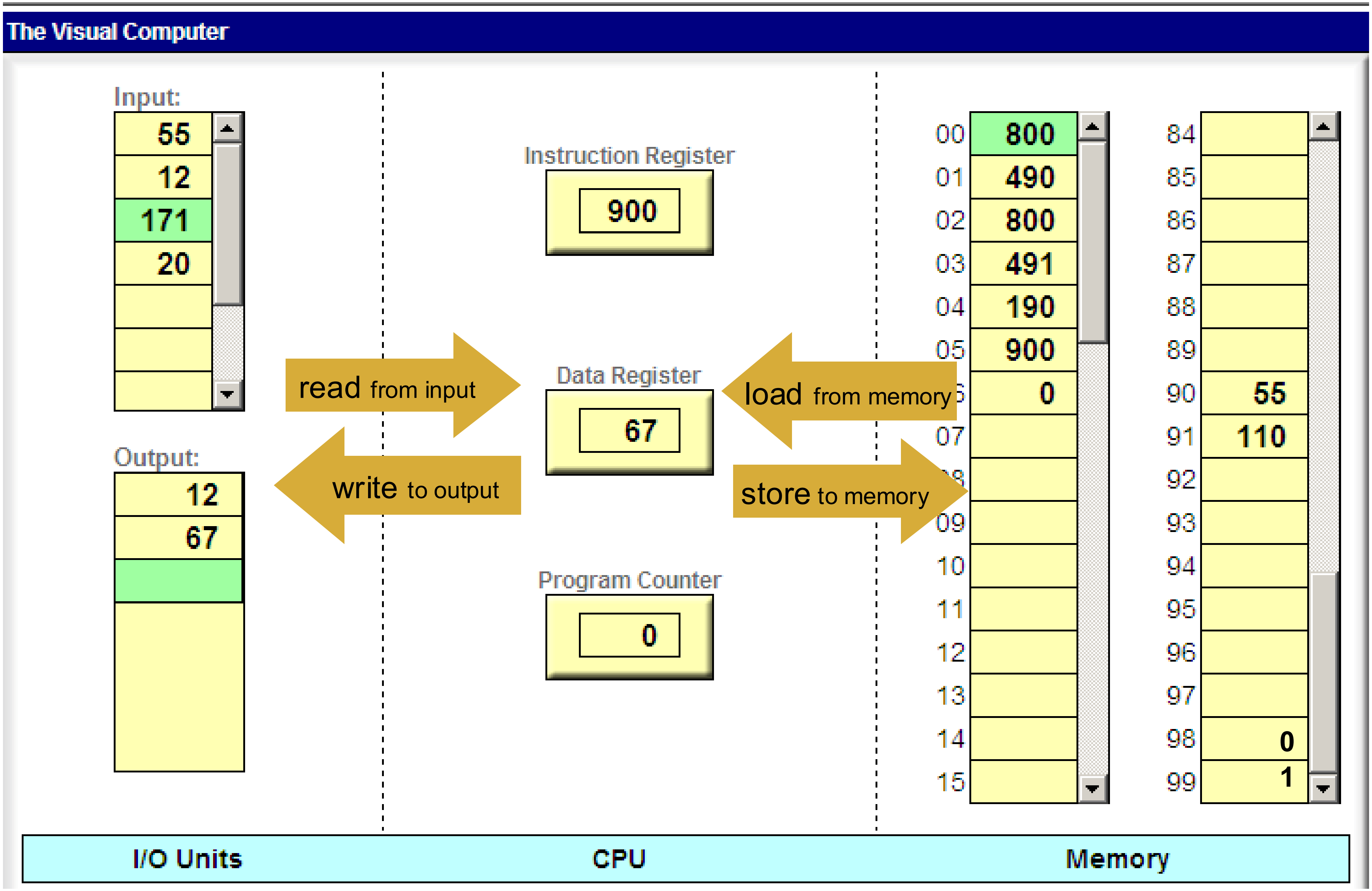
outside view



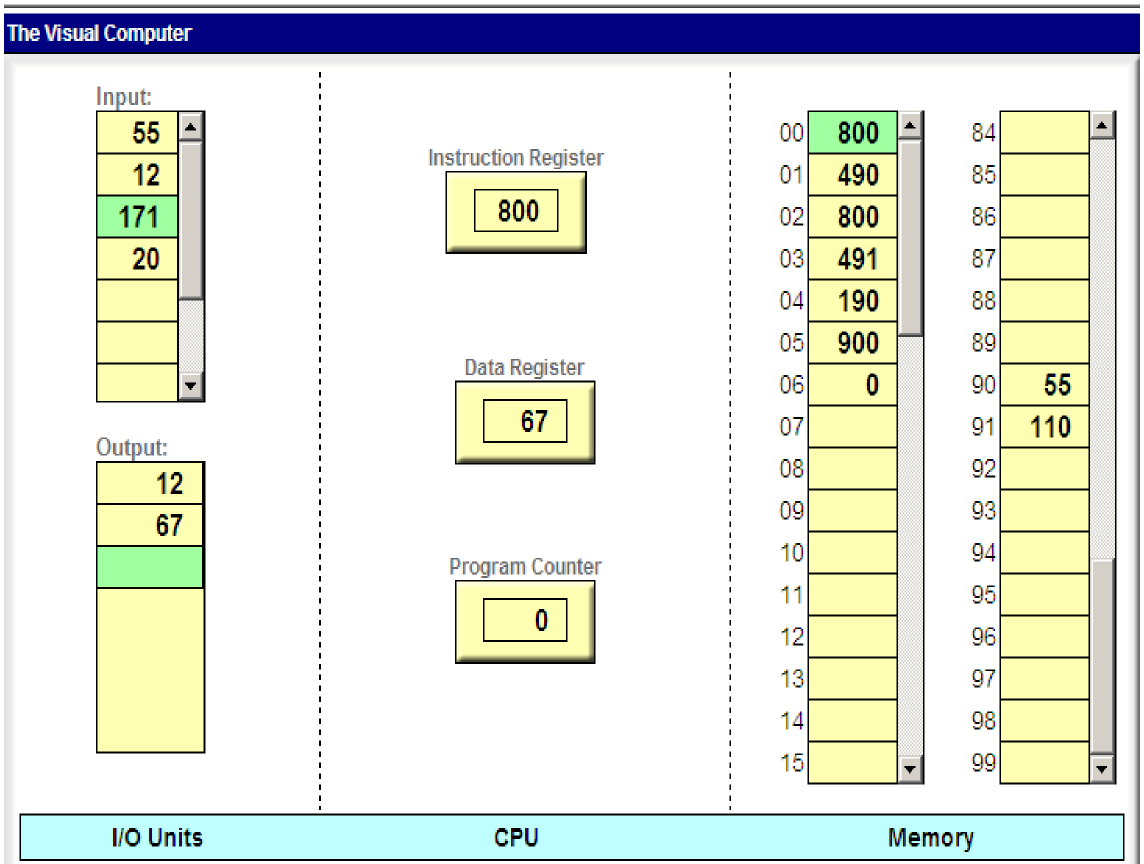
instruction set

```
read
write
load
store
add
sub
...
```

Data bus


The Visual Computer
Input:
55
12
171
20
Output:
12
67
read from input
write to output
Instruction Register
900
Data Register
67
Program Counter
0
load from memory
store to memory
00 800
01 490
02 800
03 491
04 190
05 900
0
07
09
10
11
12
13
14
15
84
85
86
87
88
89
90 55
91 110
92
93
94
95
96
97
98 0
99 1
I/O Units
CPU
Memory

Instruction set



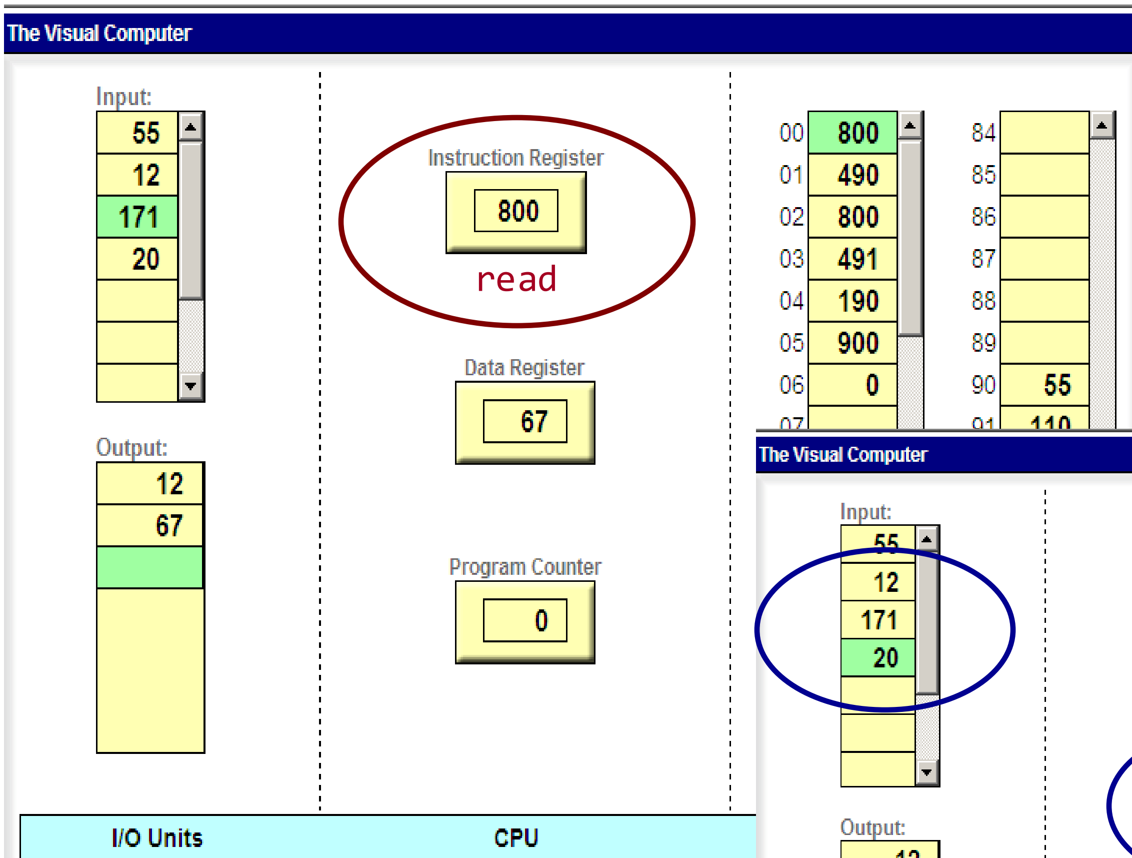
symbolic syntax	numeric syntax	semantics (meaning)
read	800	D = input
write	900	output = D
load xx	3xx	D = M[xx]
store xx	4xx	M[xx] = D
add xx	1xx	D = D + M[xx]
sub xx	2xx	D = D - M[xx]

(D = the Data register)

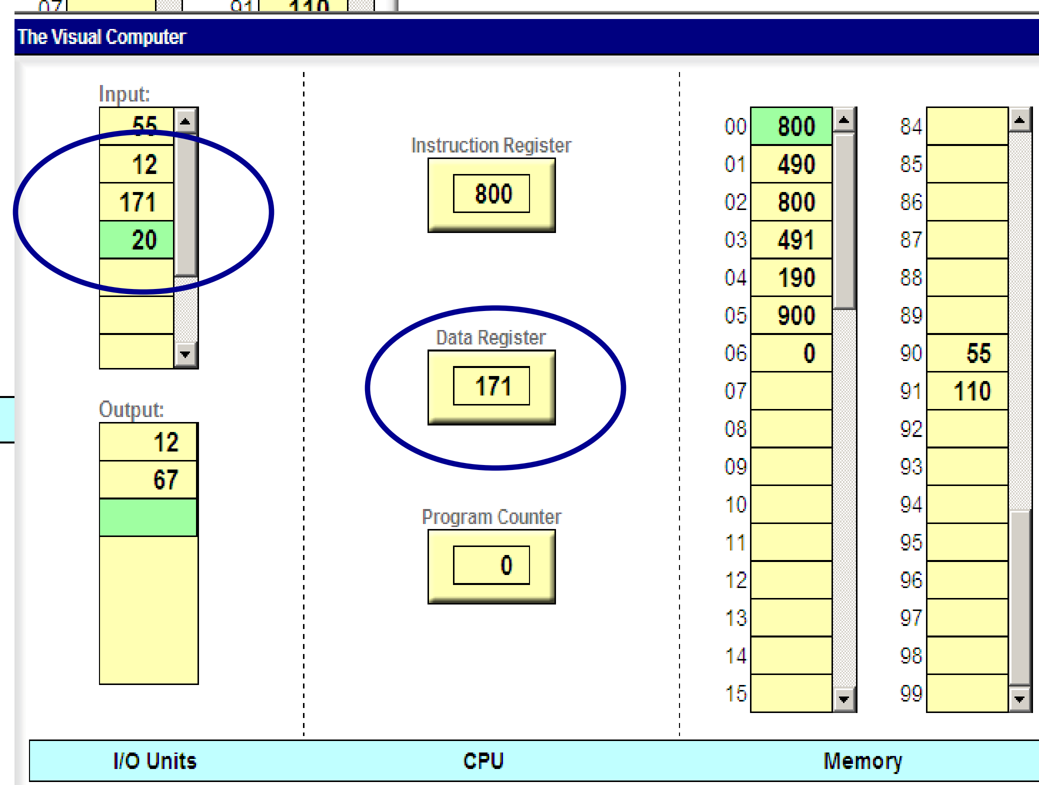
Read operation

symbolic syntax	numeric syntax	semantics (meaning)
read	800	D = input
write	900	output = D
load xx	3xx	D = M[xx]
store xx	4xx	M[xx] = D
add xx	1xx	D = D + M[xx]
sub xx	2xx	D = D - M[xx]

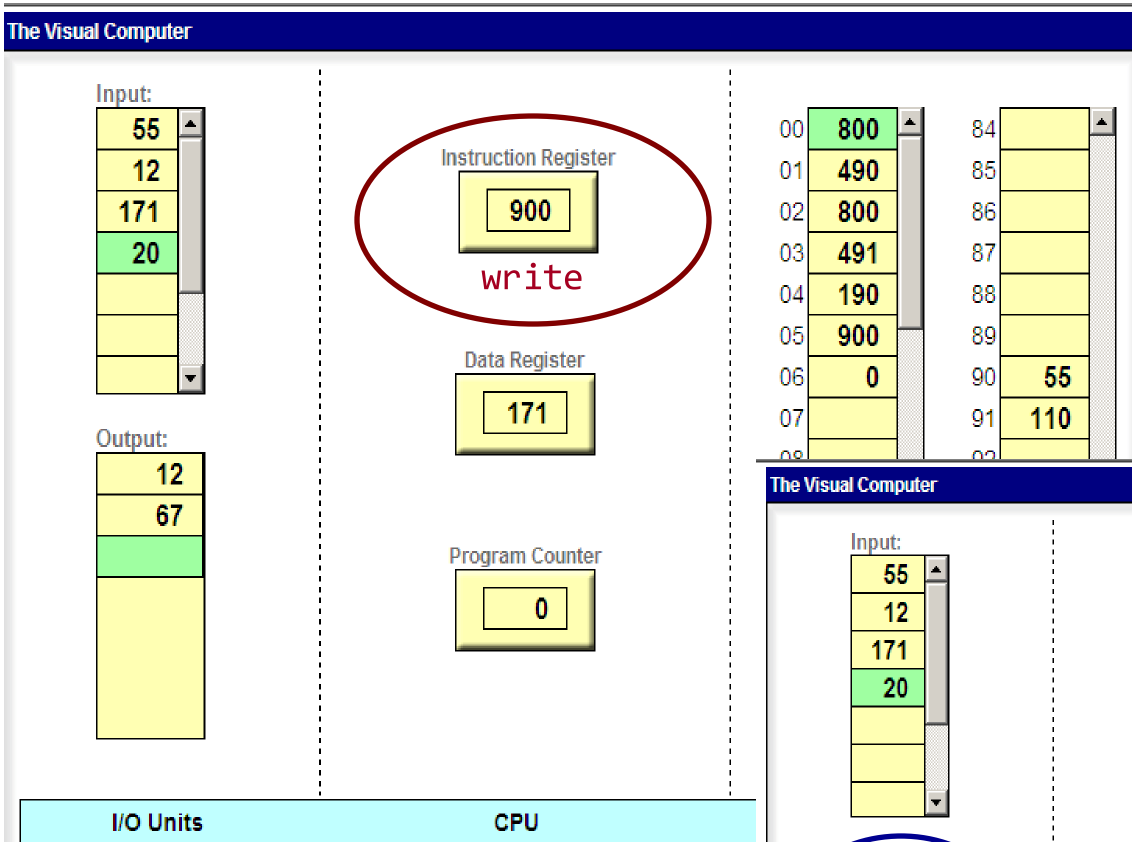
(D = the Data register)



following execution:



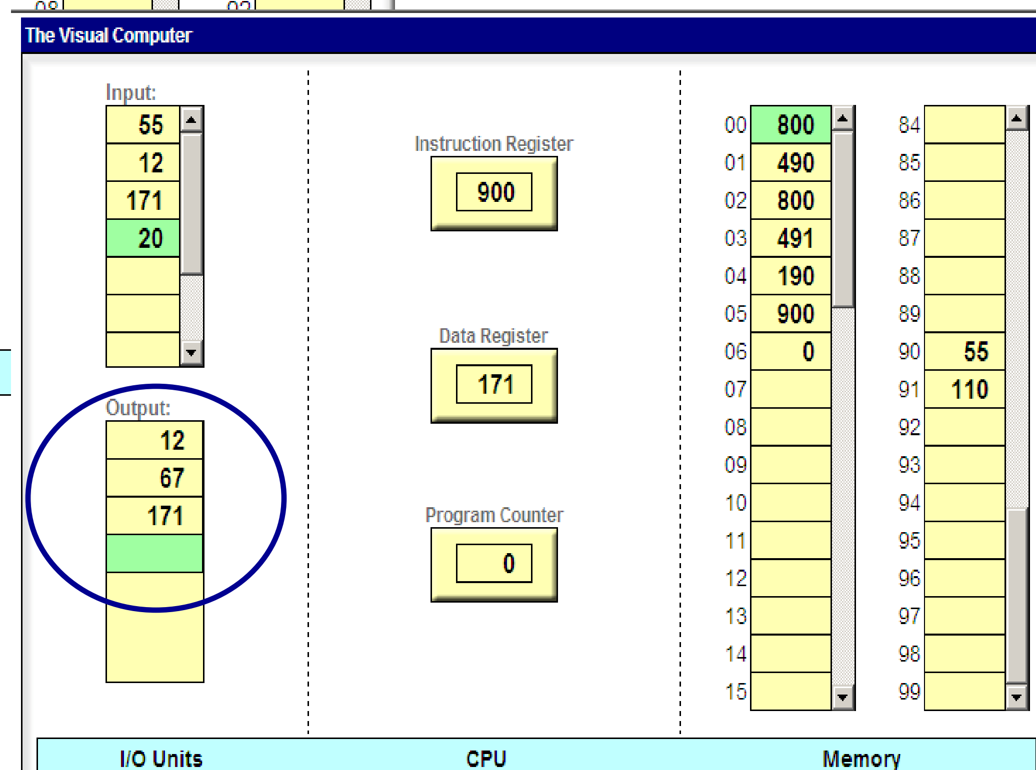
Write operation



symbolic syntax	numeric syntax	semantics (meaning)
read	800	D = input
write	900	output = D
load xx	3xx	D = M[xx]
store xx	4xx	M[xx] = D
add xx	1xx	D = D + M[xx]
sub xx	2xx	D = D - M[xx]

(D = the Data register)

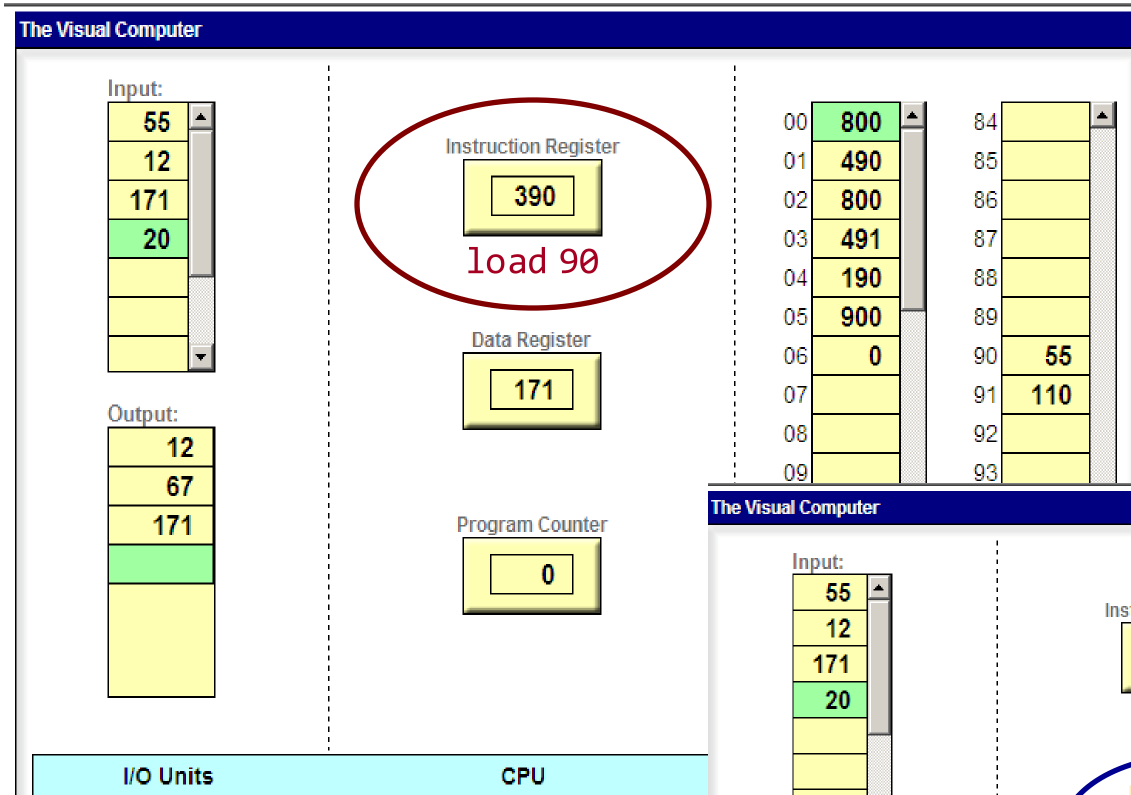
following execution:



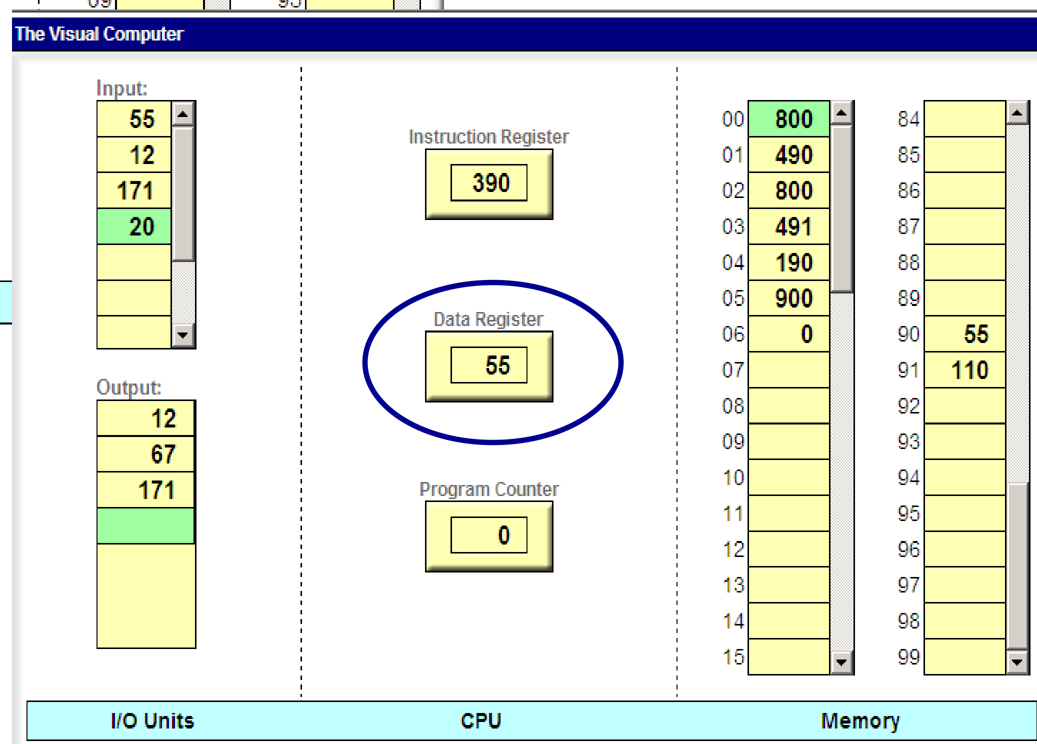
Load operation

symbolic syntax	numeric syntax	semantics (meaning)
`read`	800	D = input
`write`	900	output = D
`load xx`	3xx	D = M[xx]
`store xx`	4xx	M[xx] = D
`add xx`	1xx	D = D + M[xx]
`sub xx`	2xx	D = D - M[xx]

(D = the Data register)



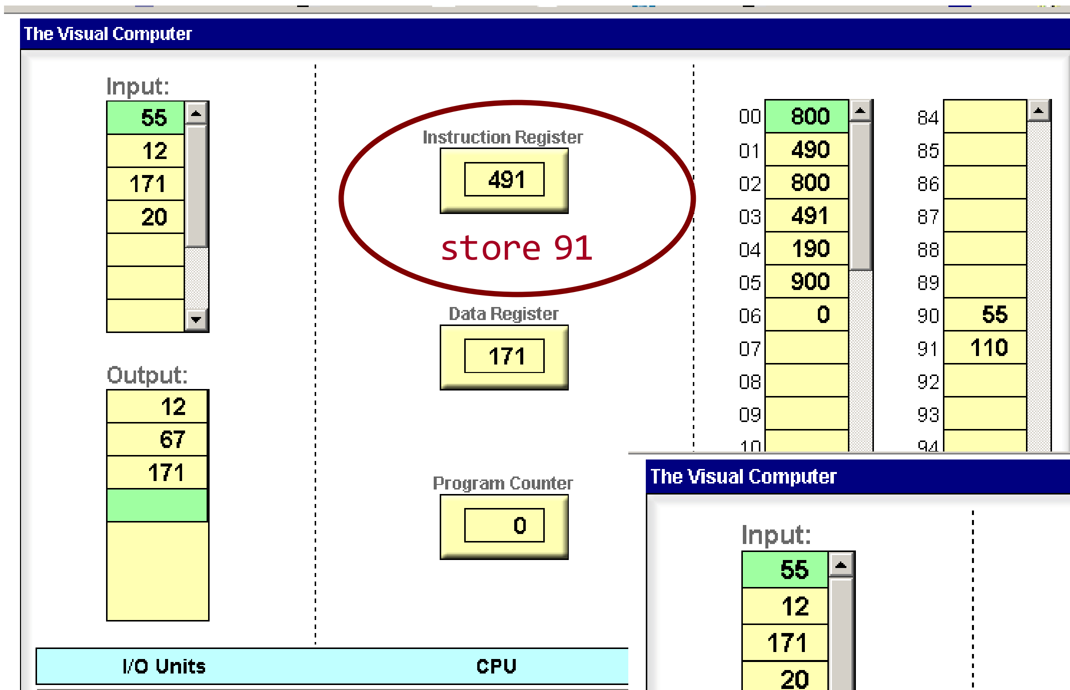
following execution:



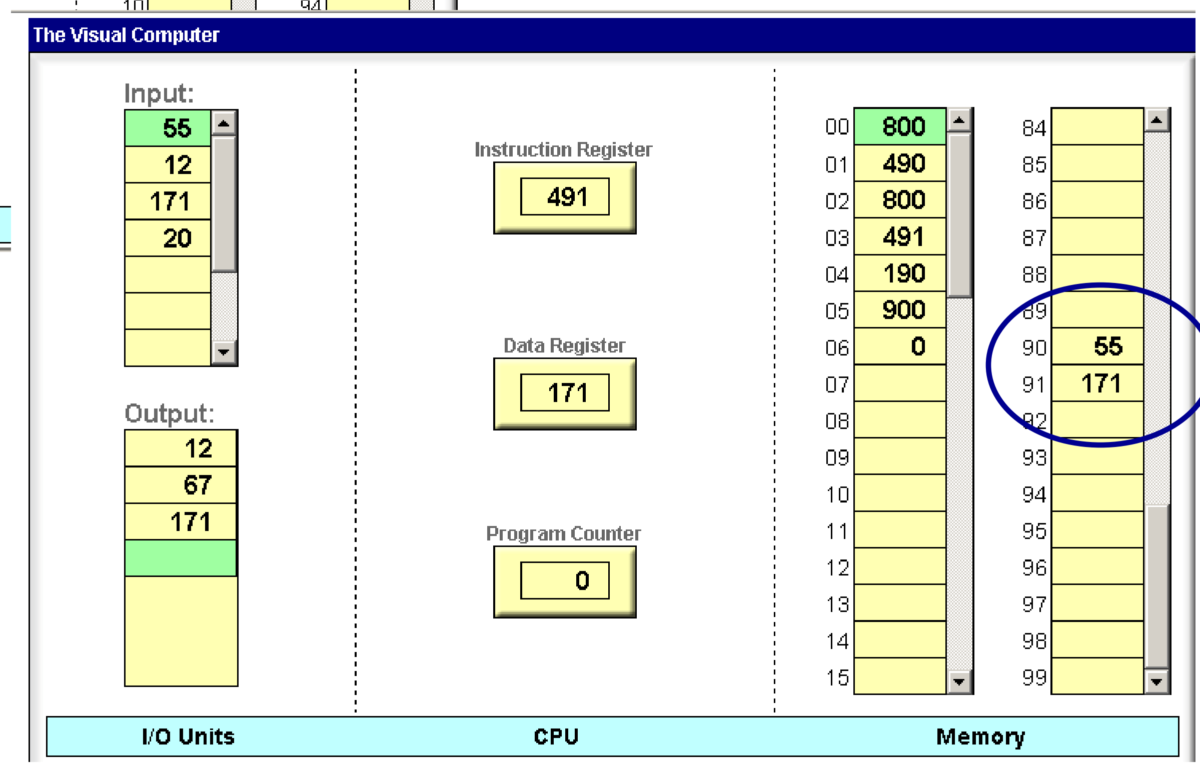
Store operation

symbolic syntax	numeric syntax	semantics (meaning)
read	800	D = input
write	900	output = D
load xx	3xx	D = M[xx]
store xx	4xx	M[xx] = D
add xx	1xx	D = D + M[xx]
sub xx	2xx	D = D - M[xx]

(D = the Data register)



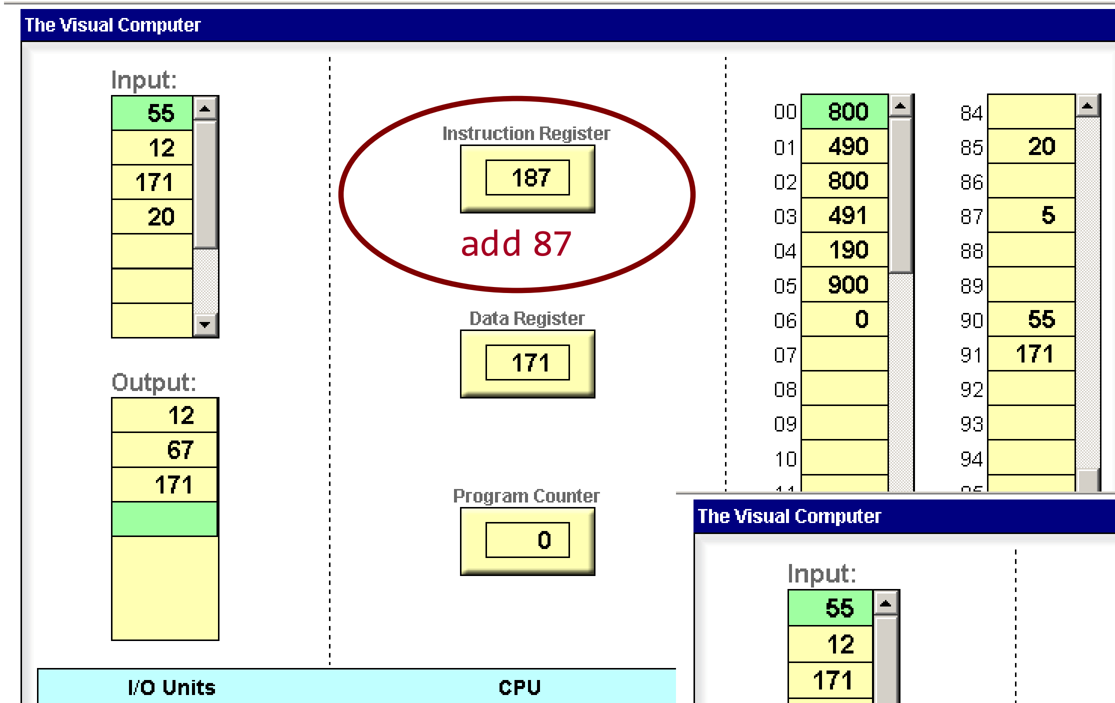
following execution:



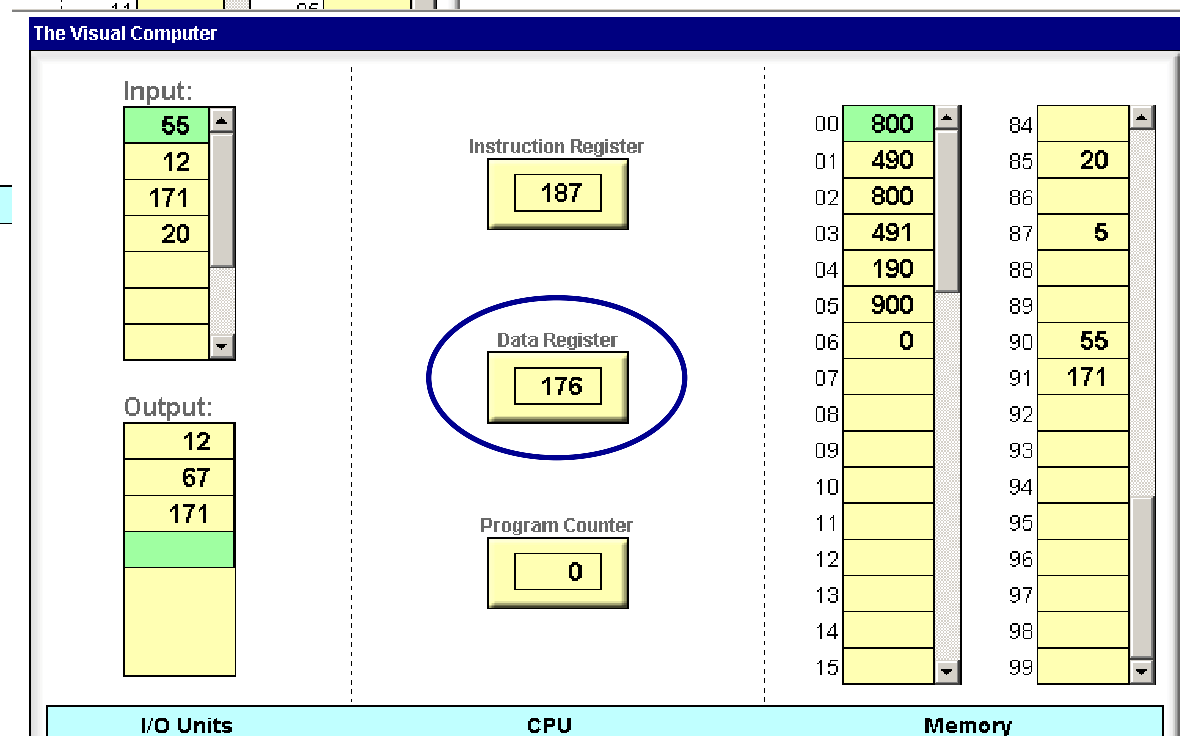
Add operation

symbolic syntax	numeric syntax	semantics (meaning)
read	800	D = input
write	900	output = D
load xx	3xx	D = M[xx]
store xx	4xx	M[xx] = D
add xx	1xx	D = D + M[xx]
sub xx	2xx	D = D - M[xx]

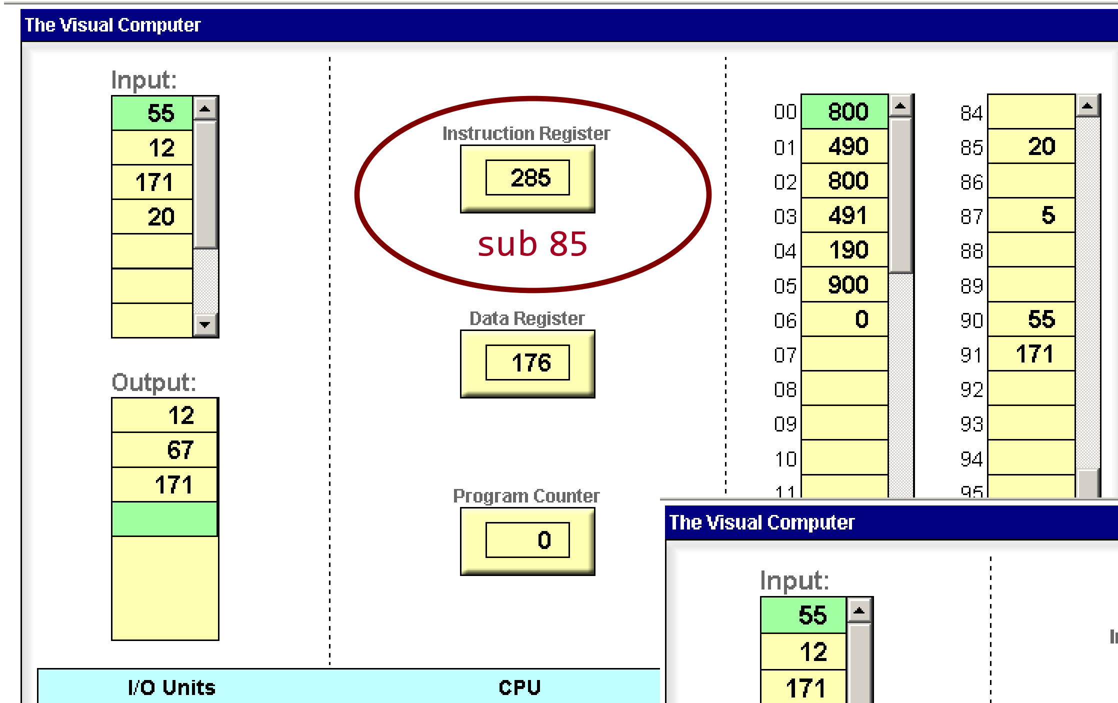
(D = the Data register)



following execution:



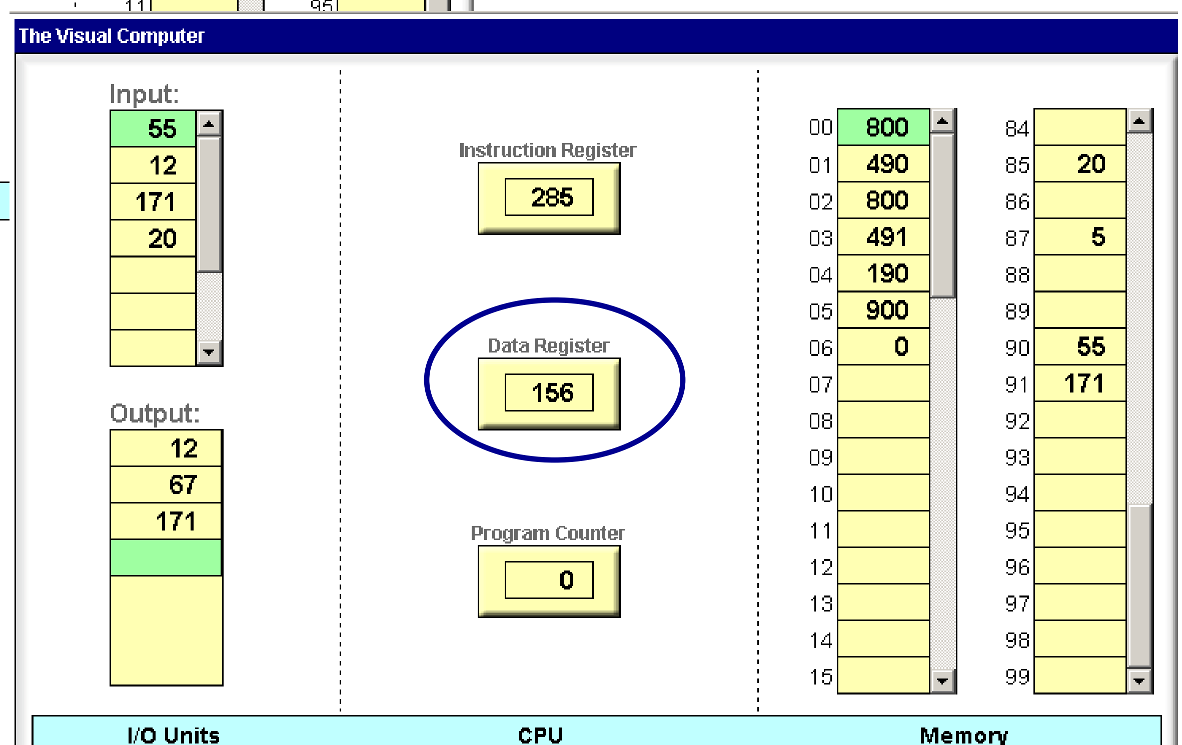
Subtract operation



symbolic syntax	numeric syntax	semantics (meaning)
read	800	D = input
write	900	output = D
load xx	3xx	D = M[xx]
store xx	4xx	M[xx] = D
add xx	1xx	D = D + M[xx]
sub xx	2xx	D = D - M[xx]

(D = the Data register)

following execution:



Lecture plan

- Computer (Vic)
    - Architecture
    - Instructions
- Low-level programming
    - Basic
    - Branching
- Control
- Program translation
- From Vic to a real computer

Low-level programming

Task: read two numbers and write their sum.

pseudocode

High-level programming

You think about the code *abstractly*, with Java or Python in mind,

without worrying about how the machine handles the abstraction.

Low-level programming

<u>Task:</u> read two numbers and write their sum.

pseudocode

```
int x = read()
int y = read()
print(x + y)
```

<u>High-level programming</u>

You think about the code *abstractly*, with Java or Python in mind,

without worrying about how the machine handles the abstraction.

Low-level programming

Task: read two numbers and write their sum.

pseudocode

Low-level programming

You think about the code *mechanically*, in terms of machine-level operations.

symbolic syntax	numeric syntax	semantics (meaning)
read	800	D = input
write	900	output = D
load xx	3xx	D = M[xx]
store xx	4xx	M[xx] = D
add xx	1xx	D = D + M[xx]
sub xx	2xx	D = D - M[xx]

Low-level programming

<u>Task:</u> read two numbers and write their sum.

pseudocode

read a number
store it somewhere
read a number
store it somewhere
add the first number
write
stop

implement →

symbolic program

00	`read`
01	`store 90`
02	`read`
03	`store 91`
04	`add 90`
05	`write`
06	`stop`

translate →

executable code

00	800
01	490
02	800
03	491
04	190
05	900
06	000

symbolic syntax	numeric syntax	semantics (meaning)
`read`	800	D = input
`write`	900	output = D
`load` xx	3xx	D = M[xx]
`store` xx	4xx	M[xx] = D
`add` xx	1xx	D = D + M[xx]
`sub` xx	2xx	D = D - M[xx]

<u>Low-level programming</u>

You think about the code *mechanically*, in terms of machine-level operations.

Low-level programming

<u>Task:</u> read two numbers and write their sum.

pseudocode

```
read a number
store it somewhere
read a number
store it somewhere
add the first number
write
stop
```

implement →

symbolic program

```
00  read
01  store 90
02  read
03  store 91
04  add 90
05  write
06  stop
```

translate →

executable code

```
00  800
01  490
02  800
03  491
04  190
05  900
06  000
```

load →

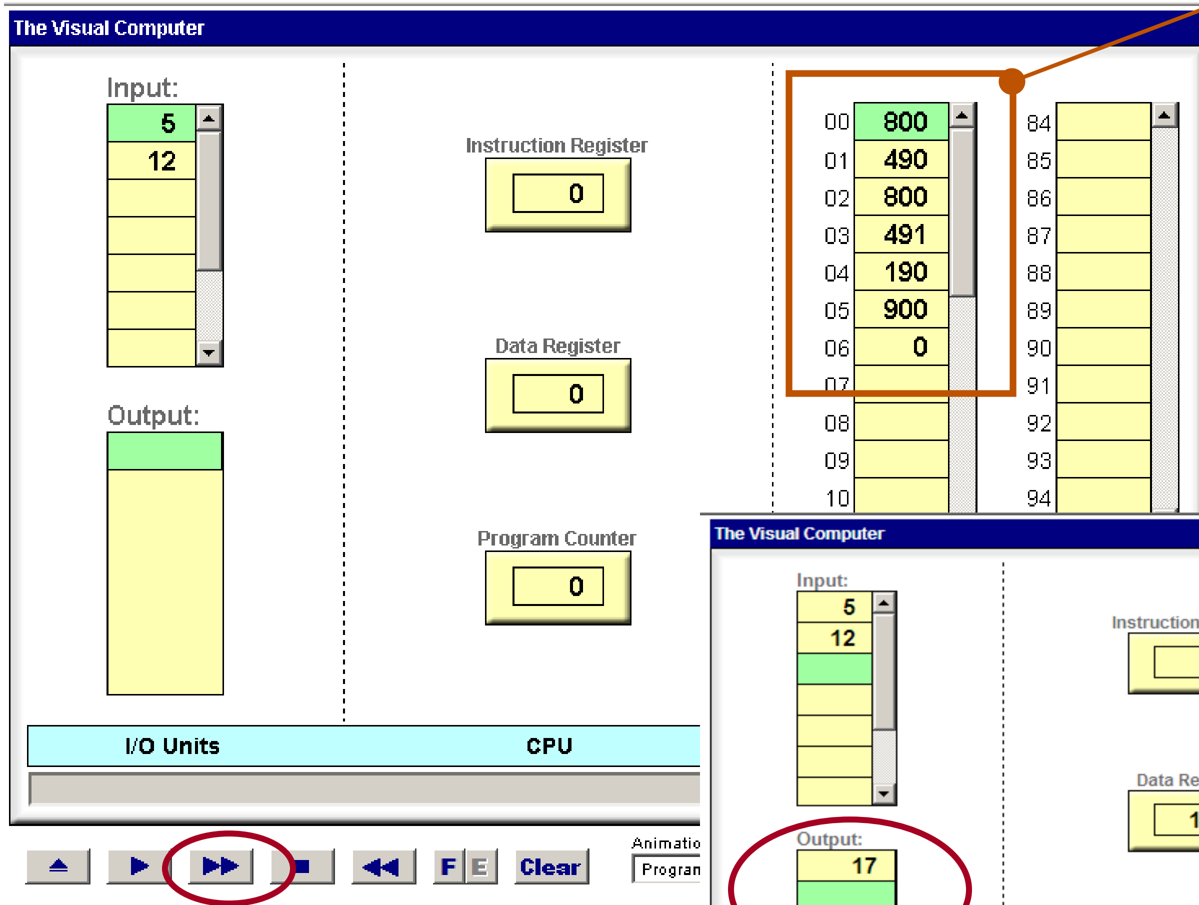
Since instructions can be coded as *numbers*, we can load them into the machine, and have the machine execute the program's semantics.

<u>Low-level programming</u>

You think about the code *mechanically*, in terms of machine-level operations.

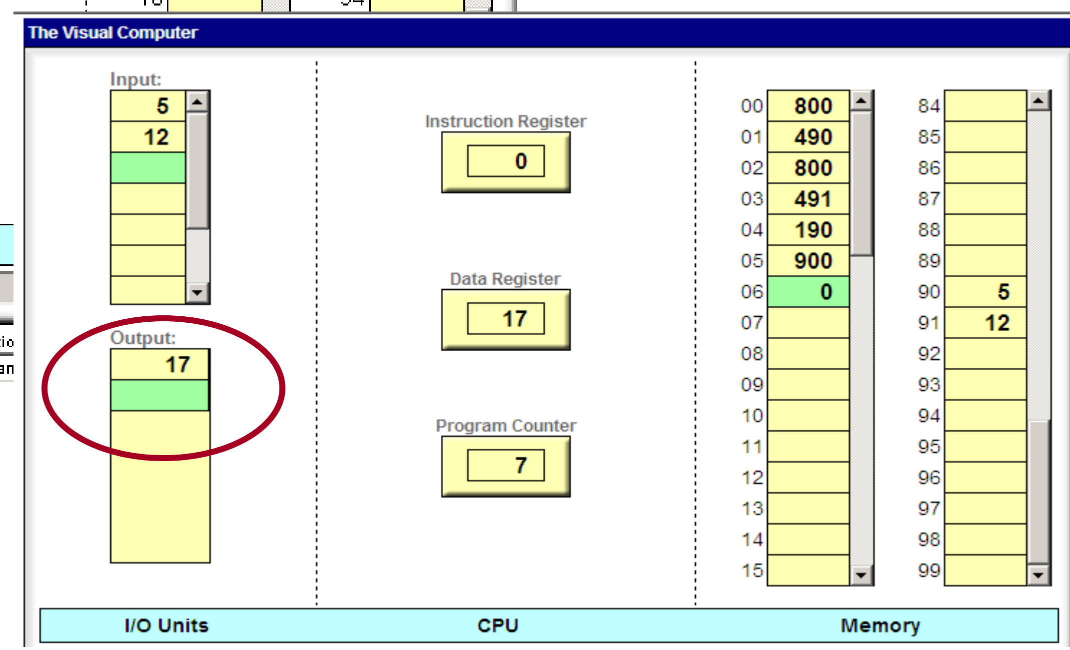
Loading and executing a program

Load the program's code

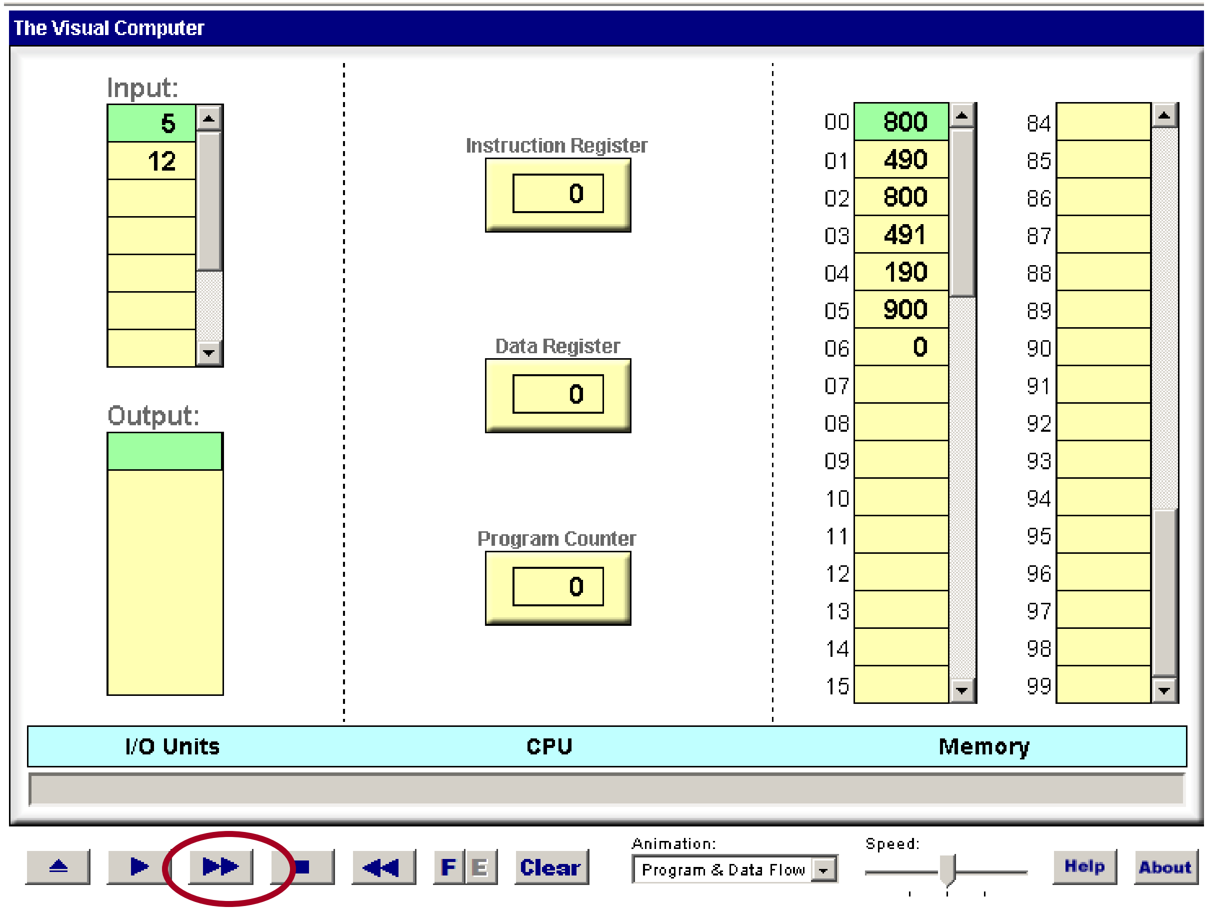


execute

following execution:



Loading and executing a program



Fetch-execute cycle:
(basic version)

```
   PC = 0
fetch:
   IR = M[PC]
   if (IR == 0) stop
   execute IR  (read, write, load, ...)
   PC++
   goto fetch
```

How is the fetch-execute logic implemented?

It is hard-wired into the computer hardware

(not shown in this lecture).

Who controls the program execution magic?

Lecture plan

- Computer (Vic)
  - Architecture
  - Instructions
- Low-level programming
  - Basic
  - Branching
- Control
- Program translation
- From Vic to a real computer

Branching

Default flow of control: linear

command
command
command
command
command
command
command
command
command
command
command
command
command
command

Typical flow of control: branching

command
`label` *a*
command
`if` *condition* `goto` *b*
command
command
`label` *b*
command
command
`if` *condition* `goto` *c*
command
`label` *c*
command
`goto` *a*

Branching

Low-level branching instruction: `goto` *addr*

addr: An address in memory (also, a line number in the program)

`goto`: Transfers control to the instruction stored in *addr*

Vic features three branching instructions:

Symbolic	Exec.	Semantics
`goto` xx	5xx	goto xx
`gotoz` xx	6xx	if (D == 0) goto xx
`gotop` xx	7xx	if (D > 0) goto xx

Task (example): Read and write until `0` is read

```
0  read
1  gotoz 4
2  write
3  goto 0
4  stop
```

translate

```
0  800
1  604
2  900
3  500
4  000
```

test

Branching

Task: Read two numbers and write the greater one

	machine language
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
...	

Branching instructions:

Symbolic	Exec.	Semantics
goto xx	5xx	goto xx
gotoz xx	6xx	if (D == 0) goto xx
gotop xx	7xx	if (D > 0) goto xx

Branching

Task: Read two numbers and write the greater one

machine language

```
0   read
1   store x
2   read
3   store y
4   sub x
5   gotop YISMAX
6   load x
7   write
8   stop
9   load y
10  write
11  stop
...
```

Branching instructions:

Symbolic	Exec.	Semantics
goto xx	5xx	goto xx
gotoz xx	6xx	if (D == 0) goto xx
gotop xx	7xx	if (D > 0) goto xx

Branching

<u>Task:</u> Read two numbers and write the greater one

machine language

0 read
1 store x
2 read
3 store y
4 sub x
5 gotop ~~YISMAX~~ 9
6 load x
7 write
8 stop
9 load y
10 write
11 stop
...

<u>Branching instructions:</u>

Symbolic	Exec.	Semantics
goto xx	5xx	goto xx
gotoz xx	6xx	if (D == 0) goto xx
gotop xx	7xx	if (D > 0) goto xx

Branching

Task: Read two numbers and write the greater one

machine language

```
0   read
1   store x
2   read
3   store y
4   sub x
5   gotop 9
6   load x
7   write
8   stop
9   load y
10  write
11  stop
...
```

Branching instructions:

Symbolic	Exec.	Semantics
goto xx	5xx	goto xx
gotoz xx	6xx	if (D == 0) goto xx
gotop xx	7xx	if (D > 0) goto xx

Branching

Task: Read two numbers and write the greater one

	machine language	executable code
0	`read`	`800`
1	`store x`	`490`
2	`read`	`800`
3	`store y`	`491`
4	`sub x`	`290`
5	`gotop 9`	`709`
6	`load x`	`390`
7	`write`	`900`
8	`stop`	`000`
9	`load y`	`391`
10	`write`	`900`
11	`stop`	`000`
...		

Convention: Allocate variables to memory, from address 90 onward:

x: represented by `M[90]`

y: represented by `M[91]`

translate

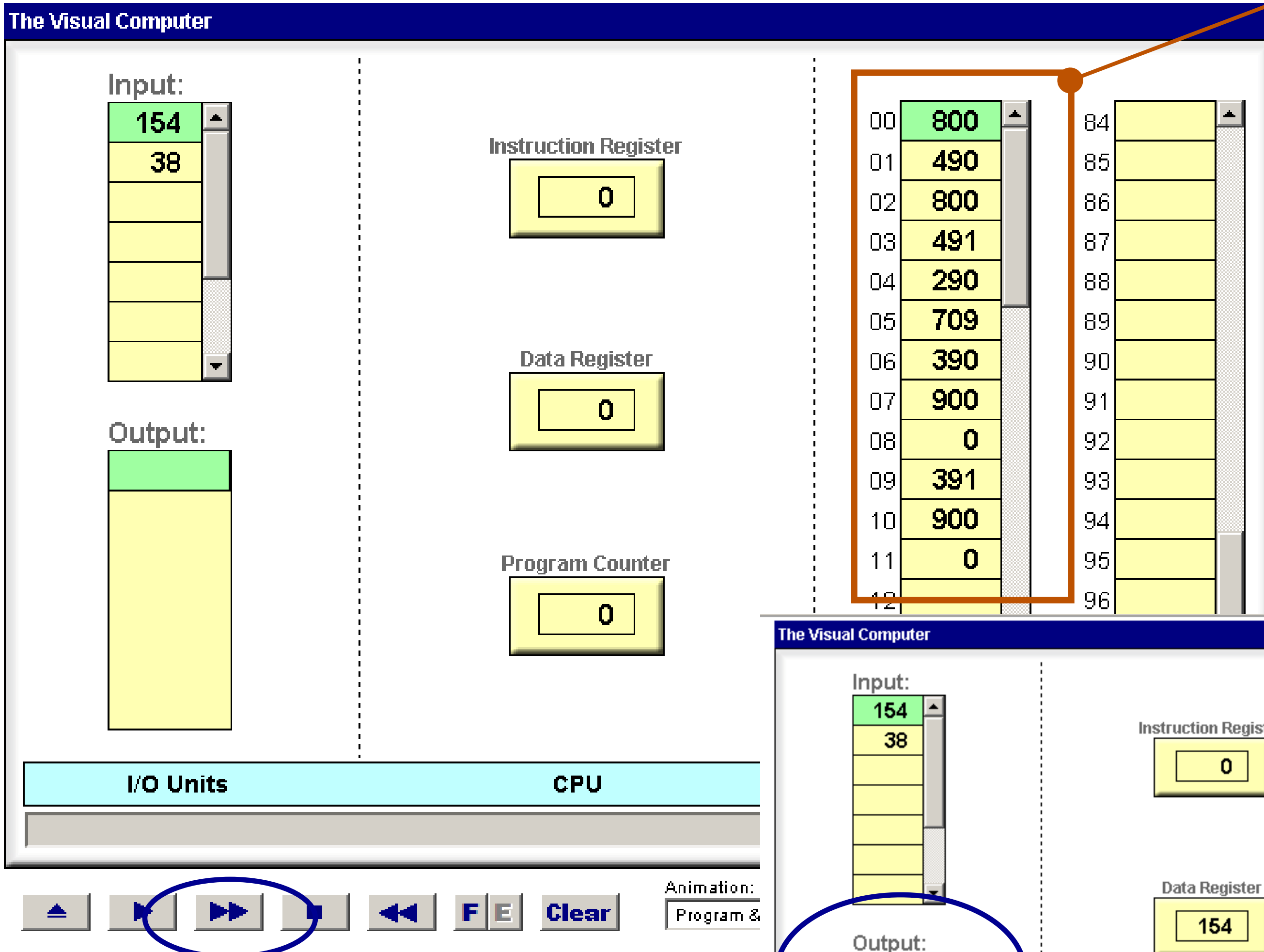
load

Branching instructions:

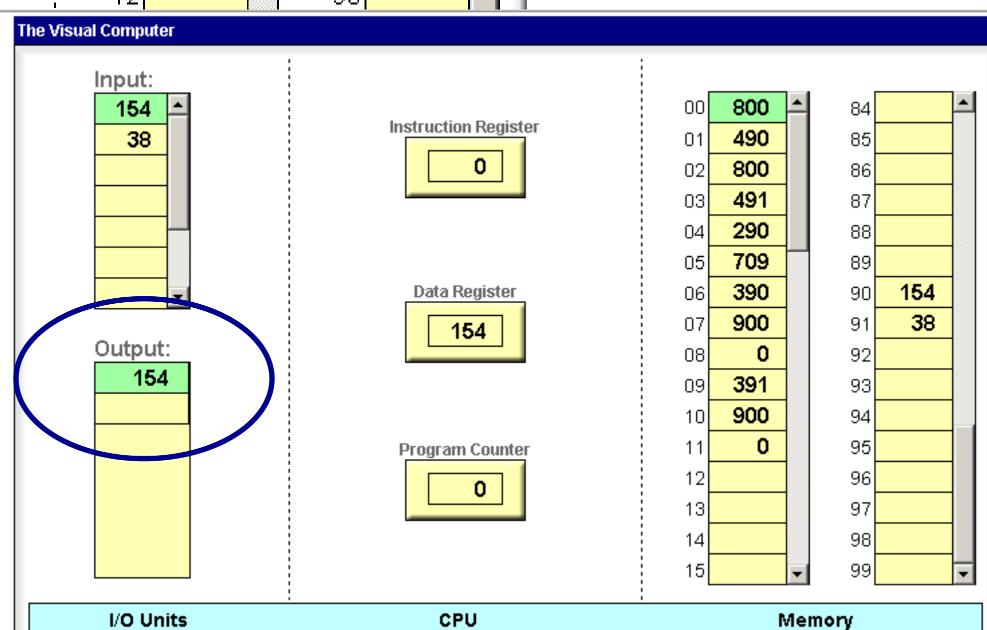
Symbolic	Exec.	Semantics
`goto` xx	5xx	goto xx
`gotoz` xx	6xx	if (D == 0) goto xx
`gotop` xx	7xx	if (D > 0) goto xx

Branching

“max of two numbers” program



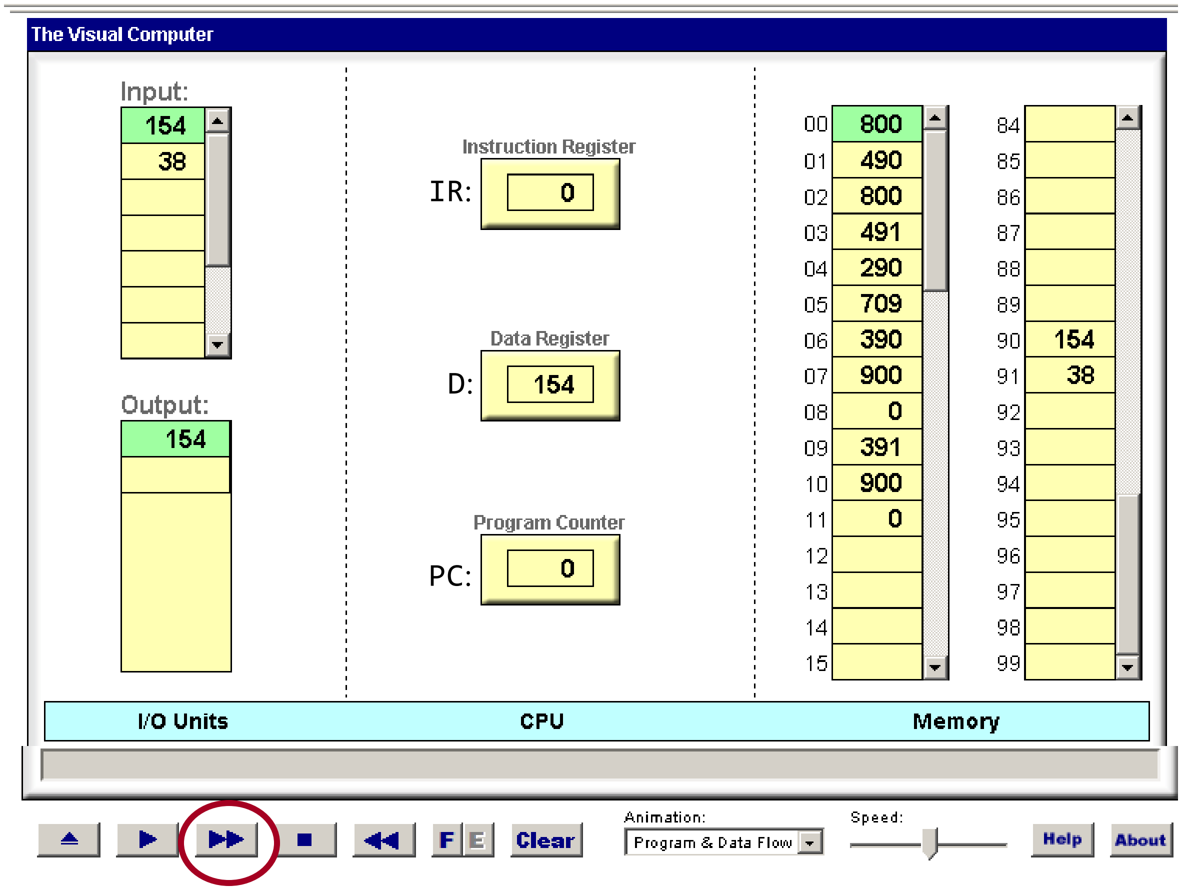
Following execution:



Lecture plan

- Computer (Vic)
  - Architecture
  - Instructions
- Low-level programming
  - Basic
  - Branching
- Control
- Program translation
- From Vic to a real computer

Fetch–execute cycle

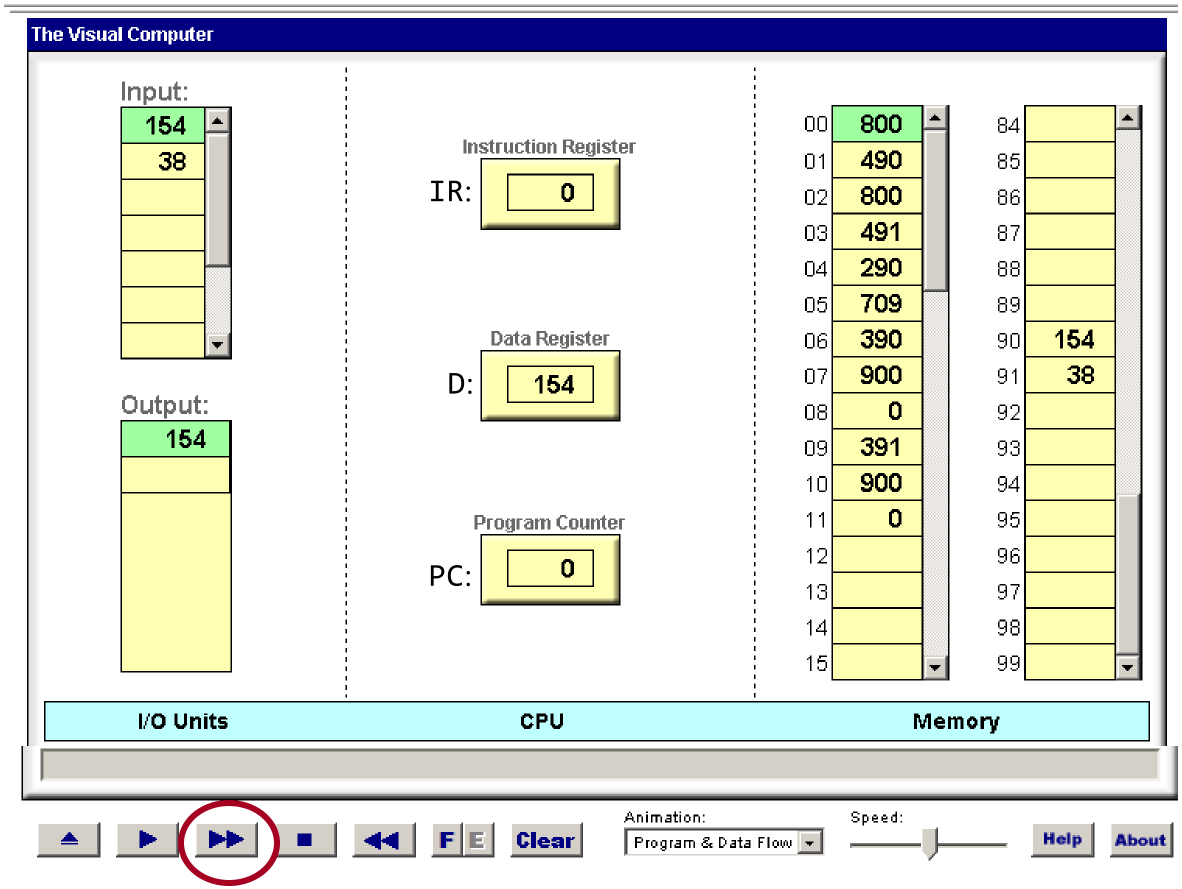



```
PC = 0

fetch:
   IR = M[PC]
   if (IR == 0) stop
```

Who controls the program execution magic?

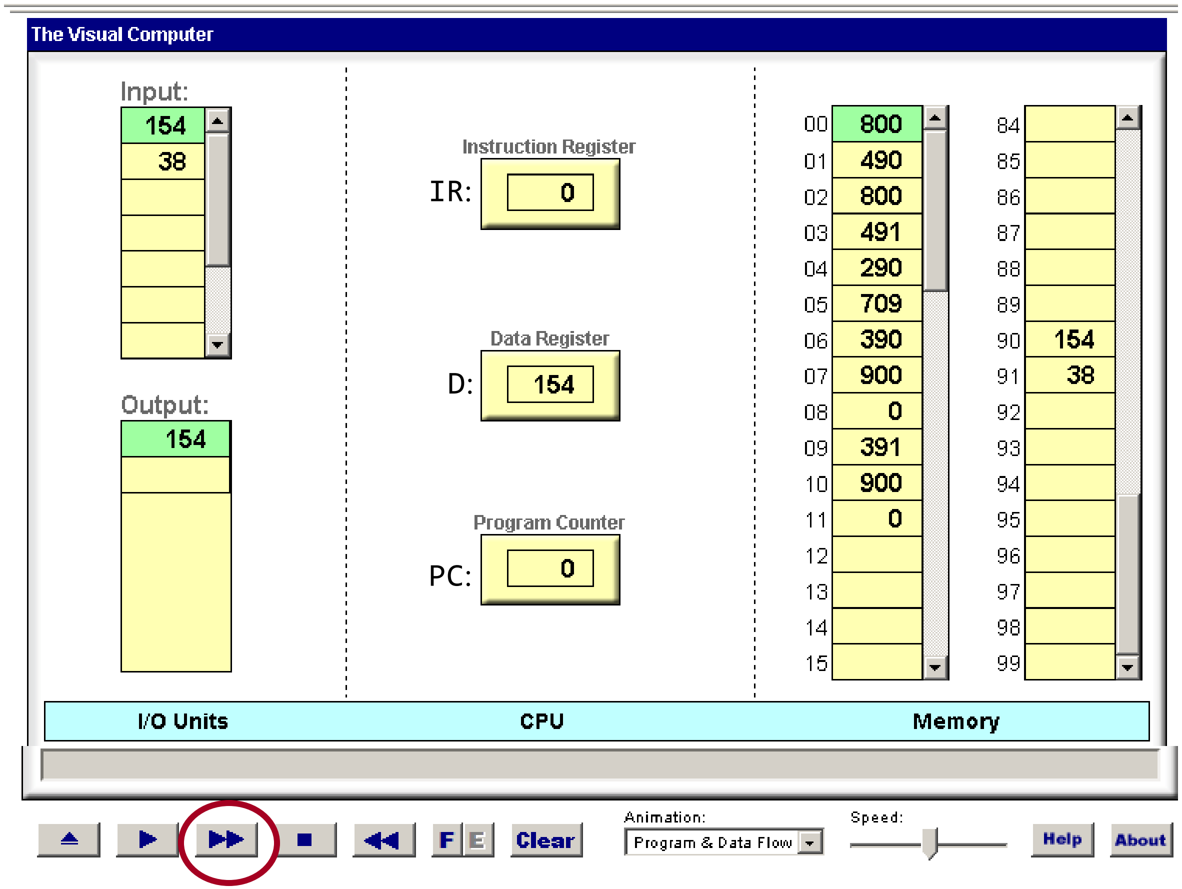
Fetch–execute cycle




```
PC = 0
fetch:
   IR = M[PC]
   if (IR == 0) stop
   if ((IR == 5xx) or
        (IR == 6xx and D == 0) or
        (IR == 7xx and D > 0))
        PC = xx
   else
        execute IR (read, write, load, store, add, sub)
        PC++
   goto fetch
```

Who controls the program execution magic?

Fetch–execute cycle

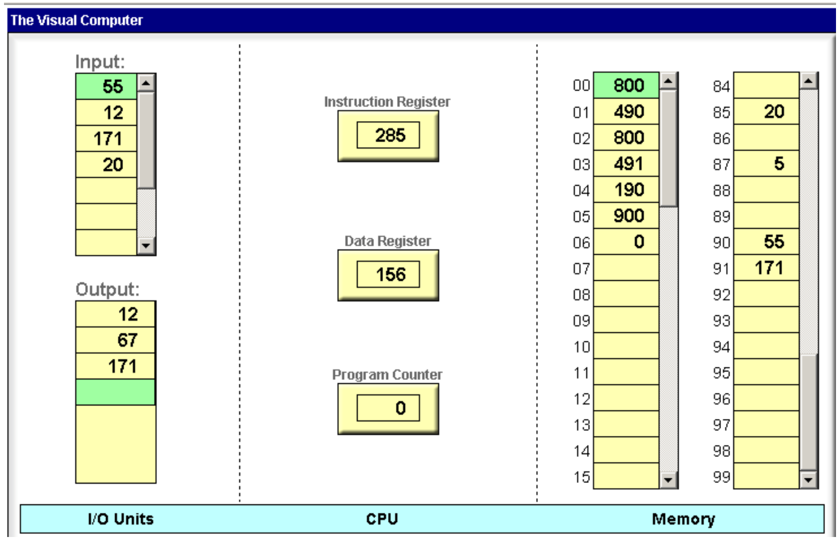



```
PC = 0
fetch:
   IR = M[PC]
   if (IR == 0) stop
   if ((IR == 5xx) or
       (IR == 6xx and D == 0) or
       (IR == 7xx and D > 0))
      PC = xx
   else
      execute IR (read, write, load, store, add, sub)
      PC++
   goto fetch
```

<u>That's it!</u> the description of the Vic computer is now complete.

The Vic computer specification (recap)

I/O CPU RAM



	symbolic syntax	numeric syntax	semantics (meaning)
I/O Instructions	`read`	`800`	D = input
	`write`	`900`	output = D
memory Instructions	`load` xx	3xx	D = M[xx]
	`store` xx	4xx	M[xx] = D
arithmetic Instructions	`add` xx	1xx	D = D + M[xx]
	`sub` xx	2xx	D = D - M[xx]
control Instructions	`goto` xx	5xx	goto xx
	`gotoz` xx	6xx	if (D == 0) goto xx
	`gotop` xx	7xx	if (D > 0) goto xx

Conventions

instruction = 3-digit number

`D`: data register

xx: 2-digit number

`M[xx]`: contents of the RAM at address xx

Lecture plan

- Computer (Vic)
  - Architecture
  - Instructions
- Low-level programming
  - Basic
  - Branching
- Control
- Program translation
- From Vic to a real computer

Symbolic programming

<u>Task:</u> sum up a series of numbers that ends with a zero

Pseudocode

```
    sum = 0
LOOP:
    read x
    if (x == 0) goto END
    sum = sum + x
    goto LOOP
END:
    write sum
    stop
```

Use symbolic variables and symbolic addresses, as needed

write

Write the code using the *symbolic Vic language*

Symbolic program

```
    // sum = 0
    load zero
    store sum
LOOP:
    // read x
    read
    store x
    // if (x == 0) goto END
    gotoz END
    // sum = sum + x
    load sum
    add x
    store sum
    goto LOOP
END:
    // write sum
    load sum
    write
    stop
```

Symbolic programming

<u>Task:</u> sum up a series of numbers that ends with a zero

Pseudocode

```
   sum = 0
LOOP:
   read x
   if (x == 0) goto END
   sum = sum + x
   goto LOOP
END:
   write sum
   stop
```

Use symbolic variables and symbolic addresses, as needed

write

Write the code using the *symbolic Vic language*

Symbolic program

```
   // sum = 0
   load zero
   store sum
LOOP:
   // read x
   read
   store x
   // if (x == 0) goto END
   gotoz END
   // sum = sum + x
   load sum
   add x
   store sum
   goto LOOP
END:
   // write sum
   load sum
   write
   stop
```

Let's remove the white space, and add line numbers

Program translation

<u>Task:</u> sum up a series of numbers that ends with a zero

Pseudocode

```
   sum = 0
LOOP:
   read x
   if (x == 0) goto END
   sum = sum + x
   goto LOOP
END:
   write sum
   stop
```

write

Symbolic program

```
 0      load zero
 1      store sum
      LOOP:
 2      read
 3      store x
 4      gotoz END
 5      load sum
 6      add x
 7      store sum
 8      goto LOOP
      END:
 9      load sum
10      write
11      stop
```

Program translation

Task: sum up a series of numbers that ends with a zero

Pseudocode

```
   sum = 0
LOOP:
   read x
   if (x == 0) goto END
   sum = sum + x
   goto LOOP
END:
   write sum
   stop
```

write

Symbolic program

```
0     load zero
1     store sum
   LOOP:
2     read
3     store x
4     gotoz END
5     load sum
6     add x
7     store sum
8     goto LOOP
   END:
9     load sum
10    write
11    stop
```

Symbol table:

```
zero:  98  (predefined)
one:   99  (predefined)
sum:   90
x:     91
LOOP:   2
END:    9
```

Construct a symbol table: Maps every symbol in the code to an (agreed-upon) memory address

Program translation

Task: sum up a series of numbers that ends with a zero

Pseudocode

```
    sum = 0
LOOP:
    read x
    if (x == 0) goto END
    sum = sum + x
    goto LOOP
END:
    write sum
    stop
```

write

Symbolic program

```
   load zero
   store sum
LOOP:
   read
   store x
   gotoz END
   load sum
   add x
   store sum
   goto LOOP
END:
   load sum
   write
   stop
```

translate

Symbol table:

```
zero: 98
one:  99
sum:  90
x:    91
LOOP:  2
END:   9
```

Executable code

0	398
1	490
2	800
3	491
4	609
5	390
6	191
7	490
8	502
9	390
10	900
11	000

Use the symbol table to translate symbolic variables and symbolic goto destinations to numeric addresses.

Program translation

Task: sum up a series of numbers that ends with a zero

Pseudocode

```
   sum = 0
LOOP:
   read x
   if (x == 0) goto END
   sum = sum + x
   goto LOOP
END:
   write sum
   stop
```

write

Symbolic program

```
   load zero
   store sum
LOOP:
   read
   store x
   gotoz END
   load sum
   add x
   store sum
   goto LOOP
END:
   load sum
   write
   stop
```

translate

Symbol table:

zero:	98
one:	99
sum:	90
x:	91
LOOP:	2
END:	9

Executable code

0	398
1	490
2	800
3	491
4	609
5	390
6	191
7	490
8	502
9	390
10	900
11	000

Observations

- Compared to executable code, **symbolic code** is much easier to write / debug / maintain.
- The translation process can be automated.

Program translation

<u>Task:</u> sum up a series of numbers that ends with a zero

Pseudocode

```
   sum = 0
LOOP:
   read x
   if (x == 0) goto END
   sum = sum + x
   goto LOOP
END:
   write sum
   stop
```

write

Use an **<u>editor</u>**

to write a program using the Vic assembly language

`Sum.asm`

```
   load zero
   store sum
LOOP:
   read
   store x
   gotoz END
   load sum
   add x
   store sum
   goto LOOP
END:
   load sum
   write
   stop
```

translate

Use an **<u>assembler</u>**

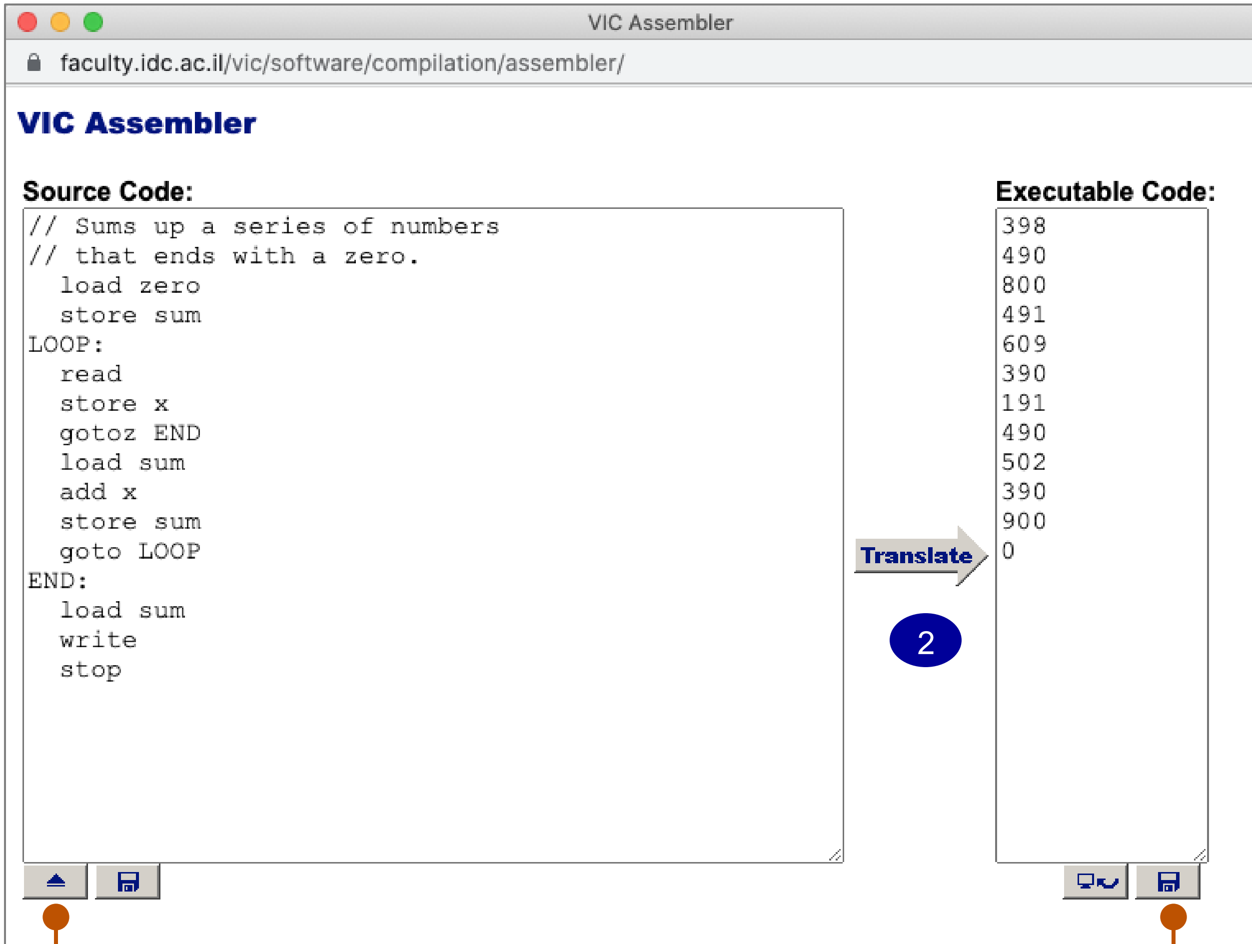
to translate the program into executable code

`Sum.vic`

```
398
490
800
491
609
390
191
490
502
390
900
000
```

Program translation: Vic https://faculty.runi.ac.il/vic/software/compilation/assembler/

Task: sum up a series of numbers that ends with a zero



1 Load an `.asm` file

3 Load the resulting `.vic` file into the Vic computer

Program translation: C

```
int main() {

        int image [4][4];
        for (int i = 0; i < 4; ++i) {
                for (int j = 0; j < 4; ++j) {
                        image[i][j] = rand();

         return 0;
    }
```

C program

(typical 2D array processing)

Program translation: C

```
000000d0 <main>:
int main() {
  d0:   ff010113            addi   sp,sp,-16
  d4:   00812423            sw     s0,8(sp)
  d8:   00112623            sw     ra,12(sp)
  dc:   00400413            li     s0,4
        int image [4][4];
        for (int i = 0; i < 4; ++i) {
                for (int j = 0; j < 4; ++j) {
                        image[i][j] = rand();
  e0:   00000097            auipc  ra,0x0
  e4:   054080e7            jalr   84(ra) # 134 <rand>
  e8:   00000097            auipc  ra,0x0
  ec:   04c080e7            jalr   76(ra) # 134 <rand>
  f0:   00000097            auipc  ra,0x0
  f4:   044080e7            jalr   68(ra) # 134 <rand>
  f8:   fff40413            addi   s0,s0,-1
  fc:   00000097            auipc  ra,0x0
 100:   038080e7            jalr   56(ra) # 134 <rand>

        . . .

 12c:   0a07a623            sw     zero,172(a5)
 130:   00008067            ret
```

1 2 3

Translated to RISC-V,
a machine language used in
many modern computers

Shows the compiled code generated by a C compiler:

1: RAM address (in hexa)

2: Binary instructions (in hexa)

3: Symbolic instructions (op code, followed by parameters)

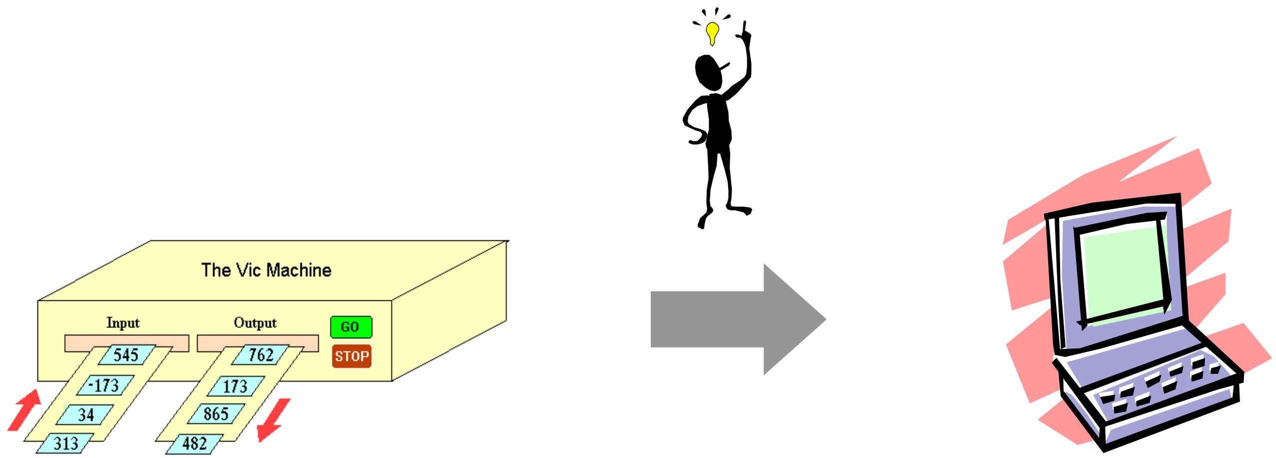
Lecture plan

- Computer (Vic)
  - Architecture
  - Instructions
- Low-level programming
  - Basic
  - Branching
- Control
- Program translation
- From Vic to a real computer

From Vic to the real thing

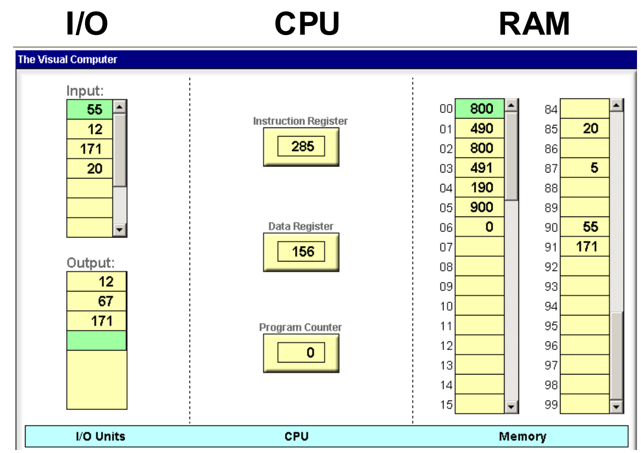
Suppose we have unlimited supplies of money, time, and creativity;

How can we evolve Vic into a real computer?



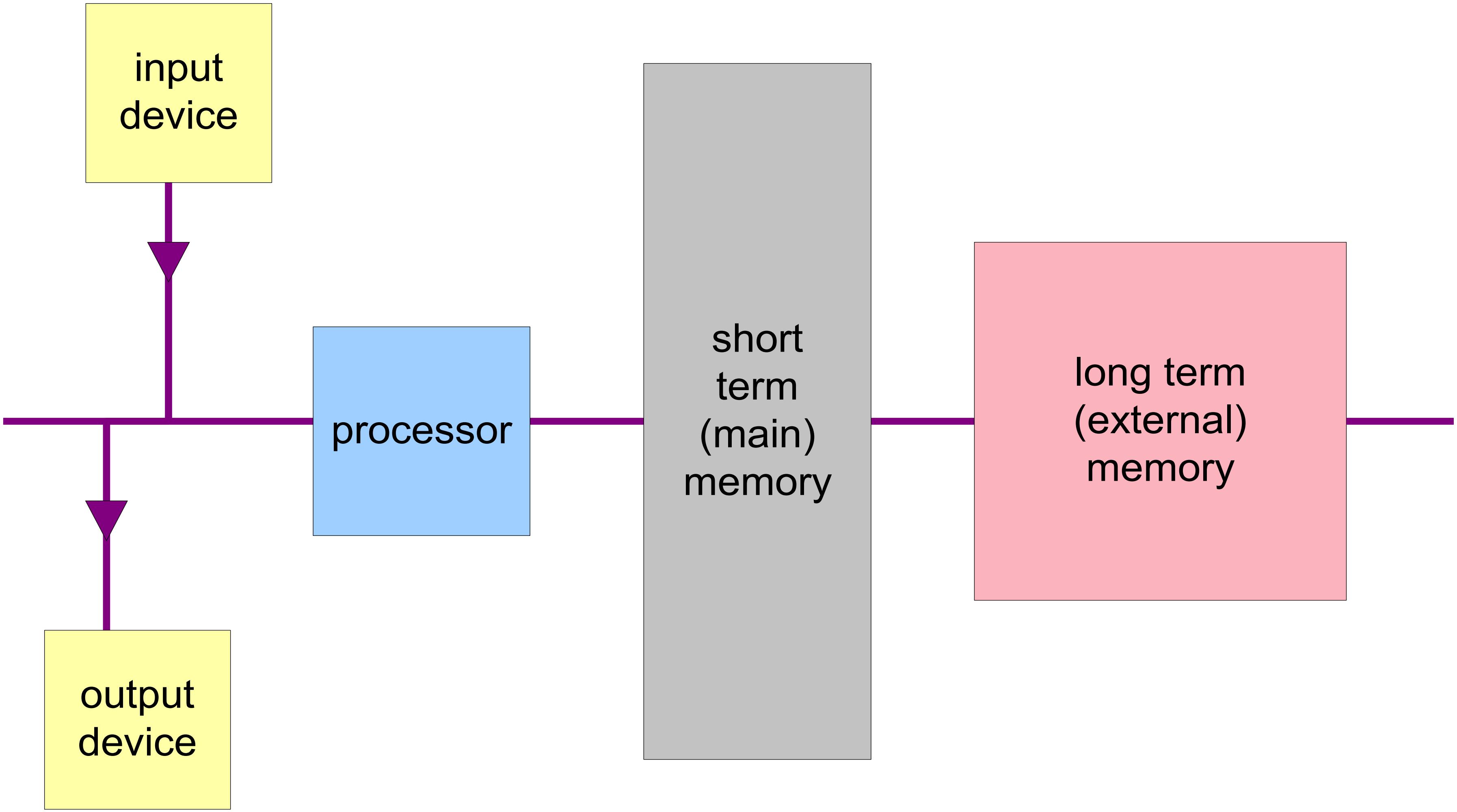
Computer architecture

Vic	Typical PC
1 data register	32 data registers
3-digit words	64-bit words
100 memory cells	billions of memory cells
1 memory unit	RAM, ROM, cache
No secondary storage	disk
1 input unit	keyboard, mouse, disk, modem, …
1 output unit	screen, speakers, disk, network, …
1 “computer”	CPU + various I/O processors, one for each I/O device
10 instructions	100-300 instructions
1 program	several programs running “simultaneously”
1 instruction per cycle	parallel processing.



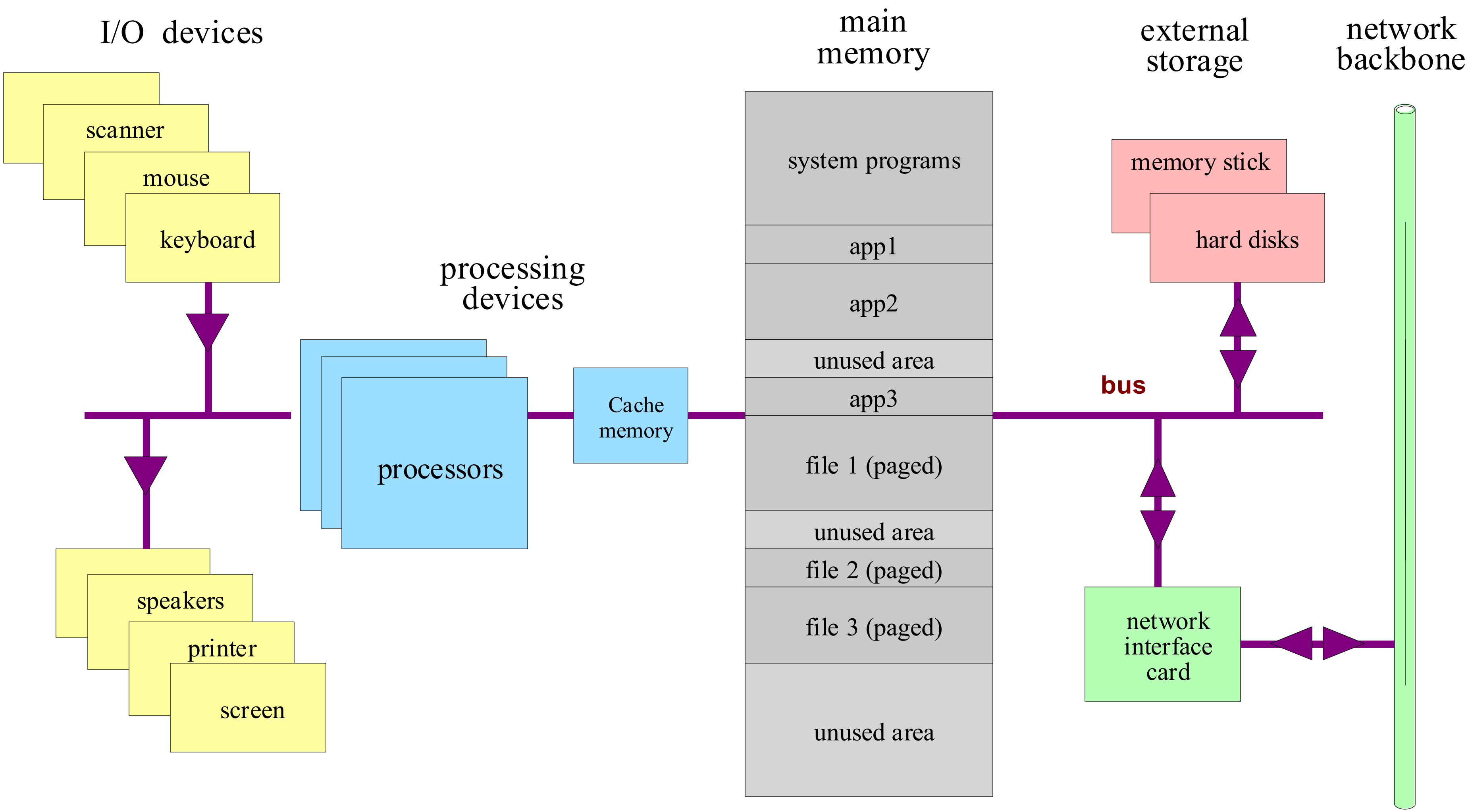
Implementing any one of these improvements is a straightforward extension of the basic Vic architecture.

Computer architecture



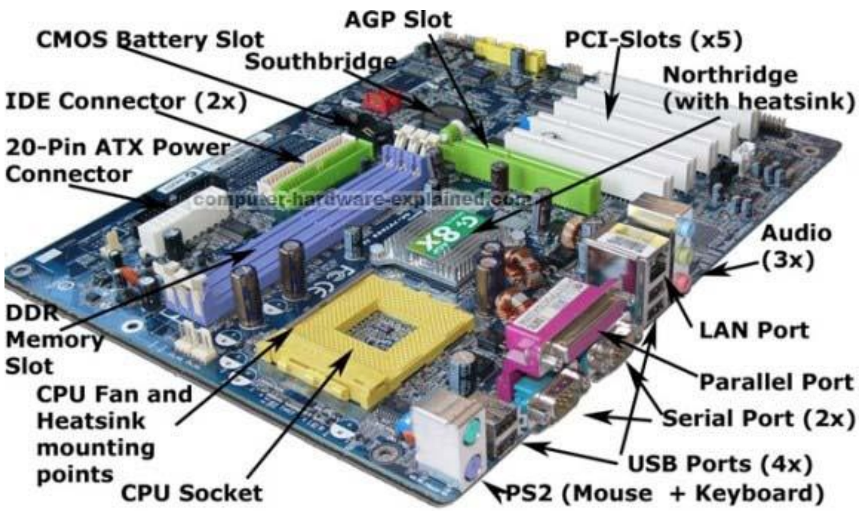
All computers are based on variations of this basic model

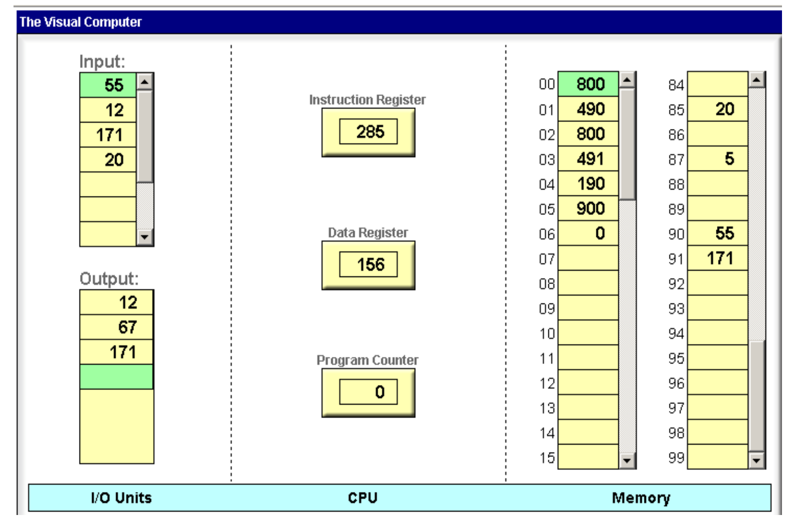
Computer architecture



All computers are based on variations of this basic model

Computer architecture





Thanks goodness for abstractions!

Computer Science (CS): Focuses on logical operations

Electrical engineering (EE): Focuses on physical implementations

<u>Remember:</u> Vic is just a simple metaphor