

Lecture 4.1

Hardware Fundamentals

Overview

Goals:

- Understand basic hardware principles
- Explore low-level programming
- Explore the road from the low-level to the high level



Approach:

- Play with a simple computer model
- Extrapolate from the model to real computer systems.

Basic Program Elements: lecture outline



■ The Visual Computer (Vic)

- Architecture
- Commands

■ Programs

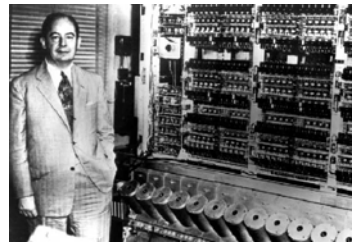
■ Branching

■ Fetch-execute cycle

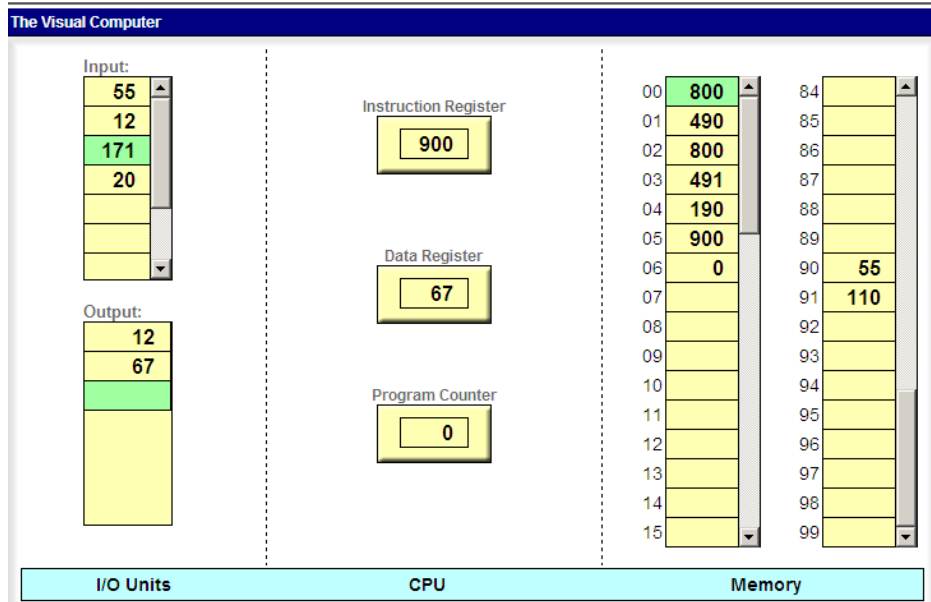
■ Looping

■ Symbolic languages

■ From Vic to a real computer

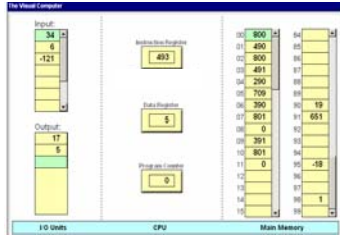


Vic: a simplified computer architecture (www.idc.ac.il/vic)

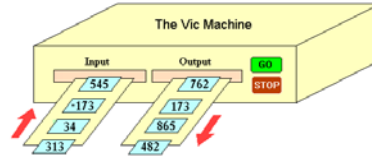


Vic: a simplified computer architecture (www.idc.ac.il/vic)

Inside view:



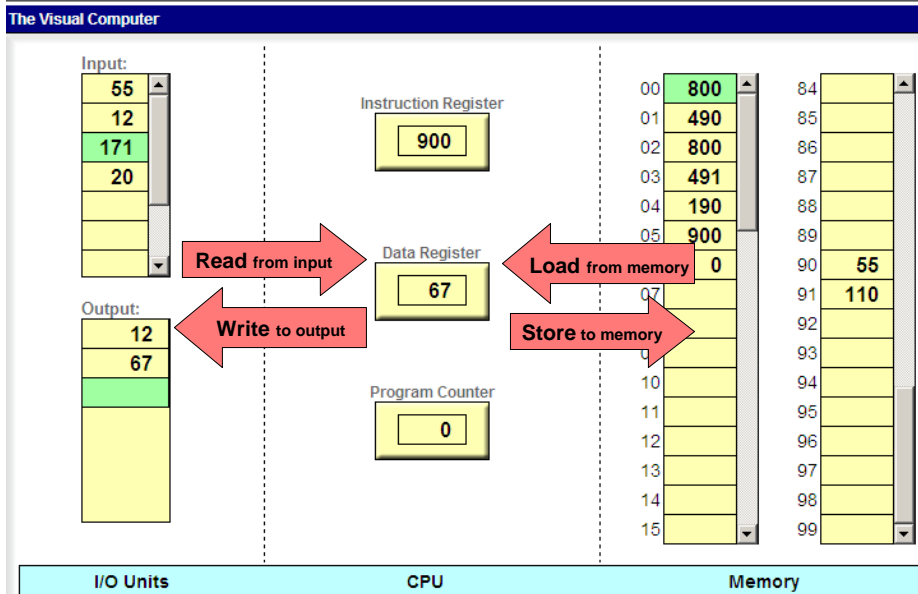
Outside view:



Commands:

- READ
- WRITE
- LOAD
- STORE
- ADD
- SUB
- ...

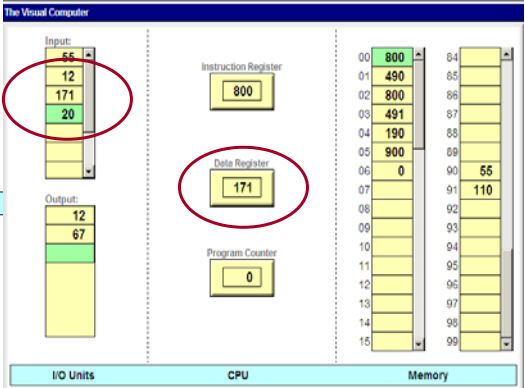
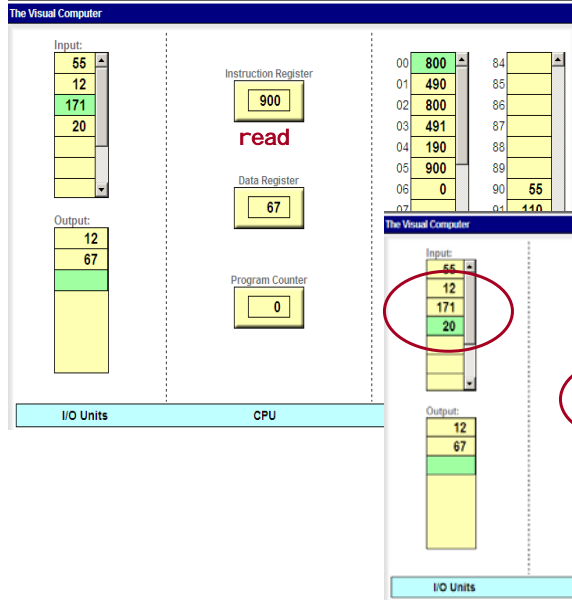
Data bus



Read operation

Symbolic Syntax Numeric Syntax Semantics (meaning)

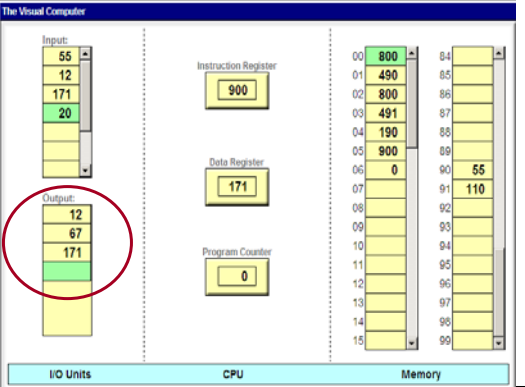
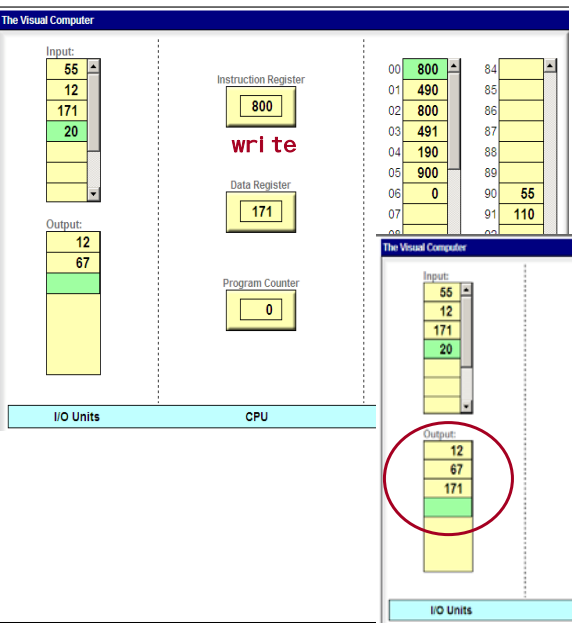
read 800 D=i input



Write operation

Symbolic Syntax Numeric Syntax Semantics (meaning)

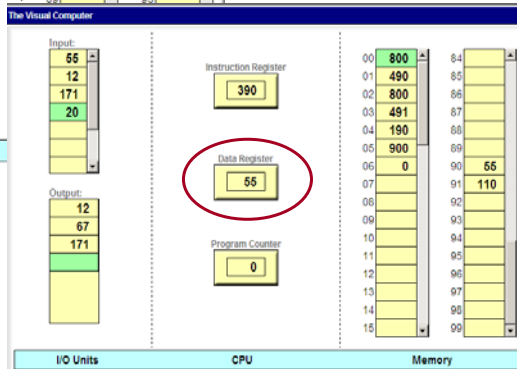
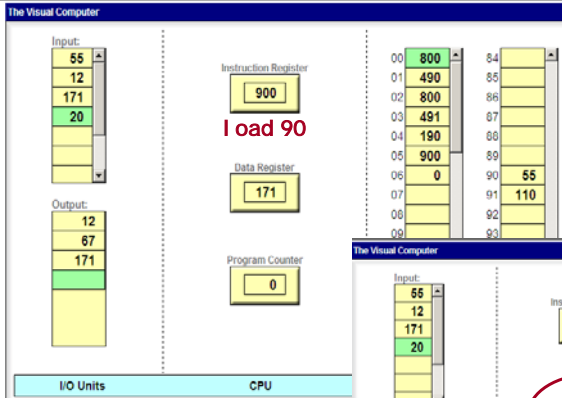
read 800 D=i input
write 900 output=D



Load operation

Symbolic Syntax Numeric Syntax Semantics (meaning)

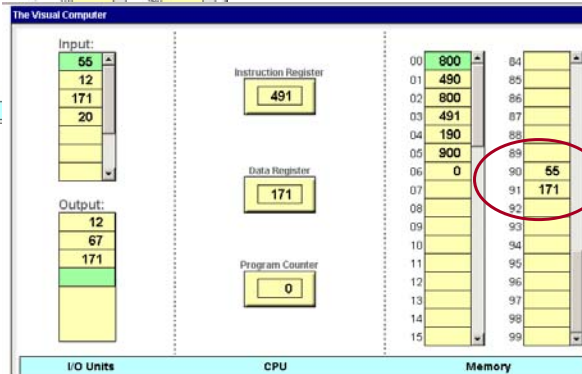
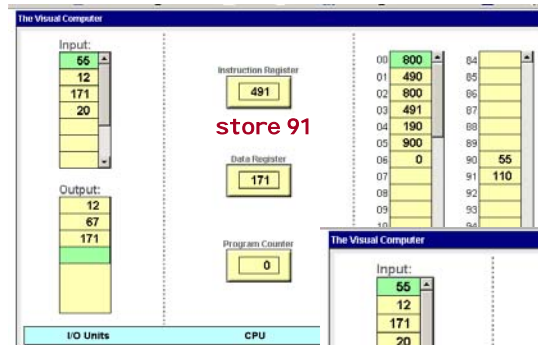
read	800	D=input
write	900	output=D
load xx	3xx	D = M[xx]



Store operation

Symbolic Syntax Numeric Syntax Semantics (meaning)

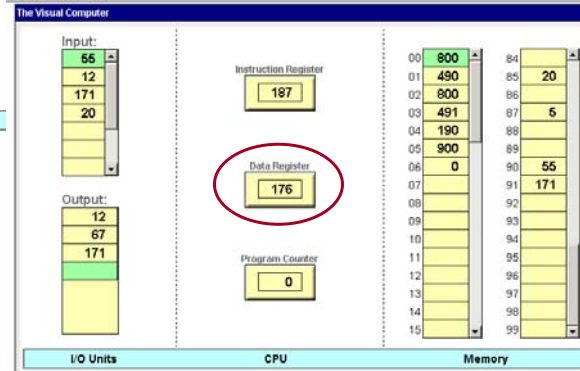
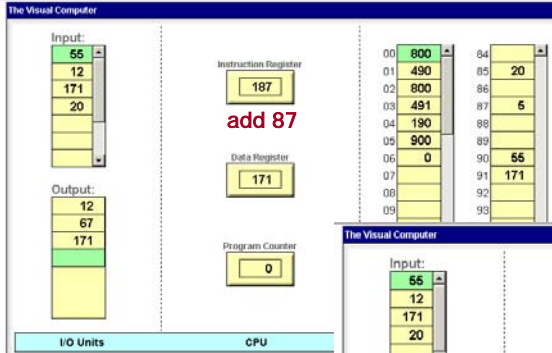
read	800	D=input
write	900	output=D
load xx	3xx	D = M[xx]
store xx	4xx	M[xx] = D



Add operation

Symbolic Syntax Numeric Syntax Semantics (meaning)

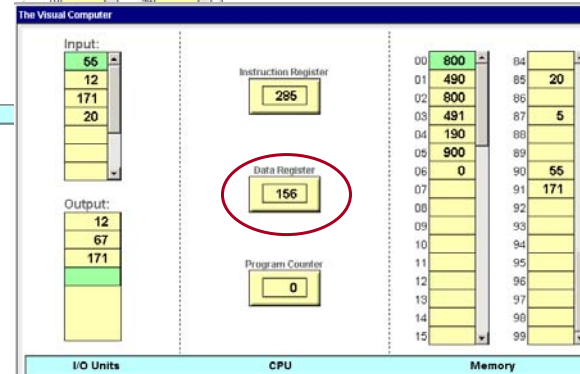
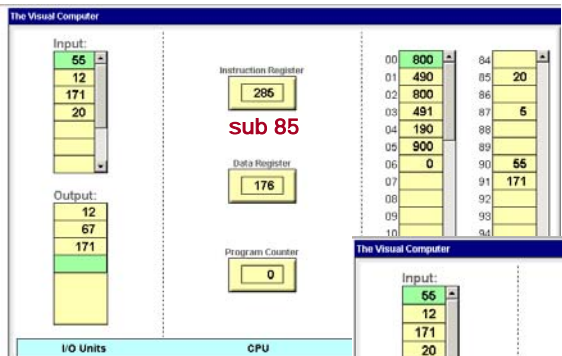
read	800	D=input
write	900	output=D
load xx	3xx	D = M[xx]
store xx	4xx	M[xx] = D
add xx	1xx	D=D+M[xx]



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]



Basic Program Elements: lecture outline

- The Visual Computer (Vic)
 - Architecture
 - Commands
- ➔ ■ Programs
- Branching
- Fetch-execute cycle
- Looping
- Symbolic languages
- From Vic to a real computer

Program: add two numbers

Task: Read two numbers from the input and write their sum.

Algorithm

read a number (let's call it x)
read a number (let's call it y)
add x to y
write the result
stop

Think ...



Symbolic program

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

Executable code

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



Design



Implement

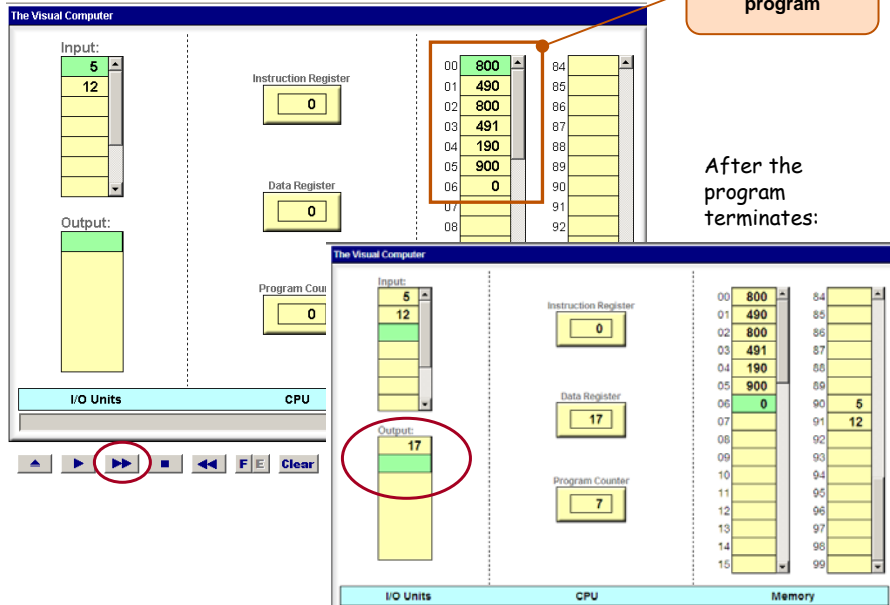


Translate

- We can write programs in some agreed-upon symbolic language, and then translate them into executable code
- Since commands can be expressed as numbers, the code can be loaded into the machine and cause it to do what we wish.

Program: add two numbers

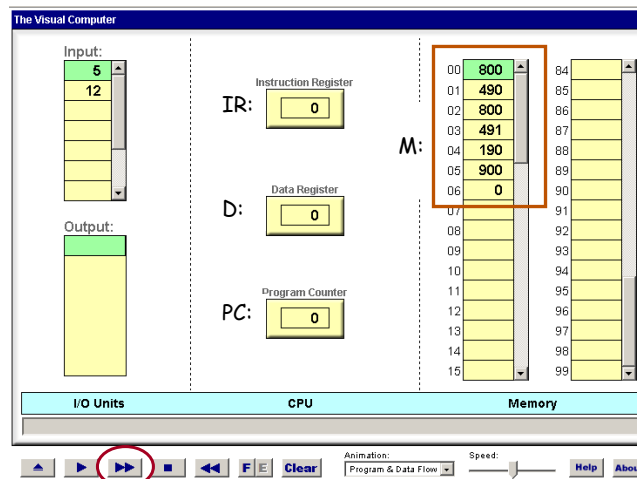
"add 2 numbers" program



Fetch-execute logic

```

PC = 0
fetch:
  IR = M[PC]
  if IR = 0 goto end
  execute IR
  PC++
  goto fetch
end:
  do nothing
    
```



Where is the fetch-execute logic implemented?

It is hard-wired into the computer hardware.

Basic Program Elements: lecture outline

- The Visual Computer (Vic)
 - Architecture
 - Commands
- Programs
- ■ Branching
- Fetch-execute cycle
- Looping
- Symbolic languages
- From Vic to a real computer

Branching

Branching commands:

- In low level programming languages, IF and WHILE logic is implemented using goto label commands
- Labels are used to mark various program locations
- goto label commands are used to transfer the flow of control to labeled program locations
- The Vic language features three goto commands:

Symbolic Syntax	Numeric Syntax	Semantic (meaning)
goto xx	5xx	goto xx
gotoz xx	6xx	if D=0 goto xx
gotop xx	7xx	if D>0 goto xx

Task: read and write until 0 is read.

```
LOOP:
  read
  gotoz END
  write
  goto LOOP
END:
  stop
```

Branching

Task: Read two inputs and write their maximum.

Algorithm

```

read x
read y
subtract x
if D>0 goto END
write x
stop
END:
write y
stop
    
```

Program →

Machine language

```

00 read
01 store 90 // x is in M[90]
02 read
03 store 91 // y is in M[91]
04 sub 90 // D = y - x
05 gotop 09 // if D>0 goto 09
06 load 90 // write x
07 write
08 stop
09 load 91 // write y
10 write
11 stop
    
```

→ **Translate**

Executable code

```

00 800
01 490
02 800
03 491
04 290
05 709
06 390
07 900
08 000
09 391
10 900
11 000
    
```

Vic goto commands:

```

goto xx    5xx    goto xx
gotoz xx   6xx    if D=0 goto xx
gotop xx   7xx    if D>0 goto xx
    
```

- A symbolic machine language is also called "assembly language"
- The assembly-to-executable code translator is called "assembler".

Branching

"max of two numbers" program

The Visual Computer

Input: 164, 38

Output:

Instruction Register: 0

Data Register: 0

Program Counter: 0

Memory:

```

00 800
01 490
02 800
03 491
04 290
05 709
06 390
07 900
08 0
09 391
10 900
    
```

After the program terminates:

Input: 164, 38

Output: 164

Instruction Register: 0

Data Register: 154

Program Counter: 0

Memory:

```

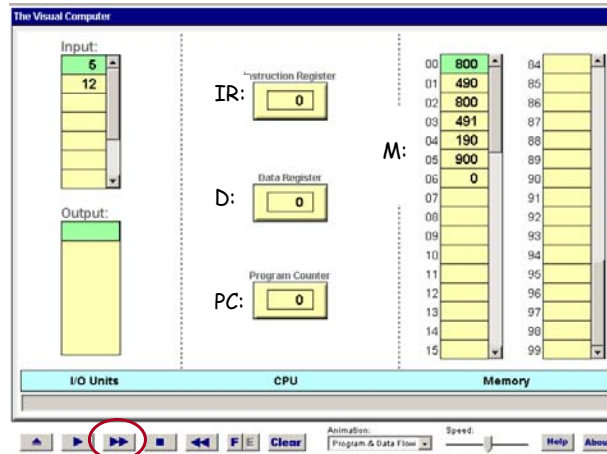
00 800
01 490
02 800
03 491
04 290
05 709
06 390
07 900
08 0
09 391
10 900
11 0
    
```

Basic Program Elements: lecture outline

- The Visual Computer (Vic)
 - Architecture
 - Commands
- Programs
- Branching
- ➔ ■ Fetch-execute cycle
- Looping
- Symbolic languages
- From Vic to a real computer

Fetch-execute logic (modified, to accommodate branching commands)

```
PC = 0
fetch:
  IR = M[PC]
  if IR = 0 goto end
  if (IR = 5xx or
      IR = 6xx and D=0 or
      IR = 7xx and D>0)
    PC = xx
  else
    execute IR
    PC++
  goto fetch
end:
goto end
```



And that's it! The description of the Vic computer is now complete.

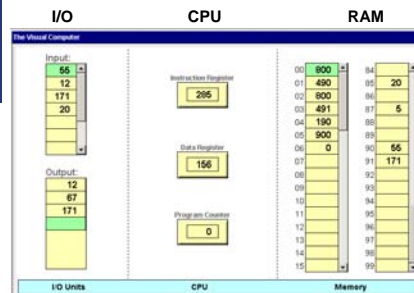
The Vic computer architecture: complete description

	Symbolic Syntax	Numeric Syntax	Semantics (meaning)
I/O commands	read	800	D = i nput
	wri te	900	output = D
Memory commands	l oad xx	3xx	D = M[xx]
	store xx	4xx	M[xx] = D
Arithmetic commands	add xx	1xx	D = D + M[xx]
	sub xx	2xx	D = D - M[xx]
Control commands	goto xx	5xx	goto xx
	gotoz xx	6xx	i f D=0 goto xx
	gotop xx	7xx	i f D>0 goto xx
	stop	000	stop

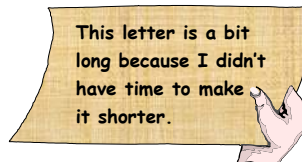
Conventions:

- Each command is represented as a 3-digit number
- D: data register
- xx: a two digit number
- M[xx]: contents of RAM cell whose address is xx.

Why is the Vic IS (instruction set) so simple?
Because we couldn't make it any simpler.



Side note: Pascal on brevity



Blaise Pascal, 1623-1662

The "Pascaline":
a calculator
built by Pascal.



He also developed the theory of hydrostatics, invented the binomial coefficients (Pascal's triangle), laid the foundations of probability theory, and wrote a fundamental philosophical treatise, called *Pensées*

The programming language Pascal is named in his honor

Died at the age of 39.

Basic Program Elements: lecture outline

- The Visual Computer (Vic)
 - Architecture
 - Commands
- Programs
- Branching
- Fetch-execute cycle
- ■ Looping
- Symbolic languages
- From Vic to a real computer

Looping

Task: sum up a series of inputs that ends with a zero.

Algorithm

```
sum = 0
read x
NEXT:
  if x = 0 goto END
  add x to sum
  read x
  goto NEXT
END:
write sum
stop
```

Program

Machine language

```
00: load 98
01: store 90 // sum in M[90]
02: read
03: store 91 // x in M[91]
04: gotoz 10 // if x=0 goto 10
05: add
06: read
07: goto 02
08:
09: goto
10: load sum // write sum
11: write
12: stop
```

Translate

Executable code

```
00 398
01 490
02 800
03 491
04 611
05 391
06 190
07 490
08 800
09 491
10 504
11 390
12 900
13 000
```

Yucc!!!

- Writing programs in machine language is not fun
- Can we express the algorithm more directly, in some higher-level language?

Symbolic language

Task: sum up a series of inputs that ends with a zero.

Algorithm

```

sum = 0
read x
NEXT:
if x = 0 goto END
add x to sum
read x
goto NEXT
END:
write sum
stop
    
```

Program →

Symbolic Machine language

```

load zero
store sum
read
store x
NEXT:
gotoz END
load x
add sum // sum = sum + x
store sum
read
store x
goto NEXT
END:
load sum
write
stop
    
```

What's new?

- Symbolic variables
- Symbolic labels
- No command numbers.

Translate →

(Assume that the code starts at address 0 and variables are allocated to memory address 90 and onward)

Symbol table

zero	98
one	99
sum	90
x	91
NEXT	04
END	11

Executable code

00	398
01	490
02	800
03	491
04	611
05	391
06	190
07	490
08	800
09	491
10	504
11	390
12	900
13	000

Translation process

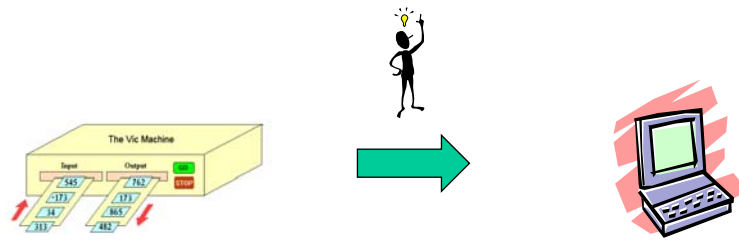
- First pass: resolve labels; allocate memory to variables; create a symbol table
 - Second pass: use the symbol table to translate the program to executable code
- Mazel Tov!** No more command numbers; no more physical memory addresses.

Basic Program Elements: lecture outline

- The Visual Computer (Vic)
 - Architecture
 - Commands
- Programs
- Branching
- Fetch-execute cycle
- Looping
- Symbolic languages
- ➔ ■ From Vic to a real computer

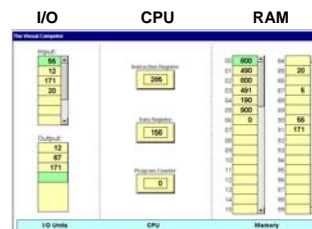
From Vic to the real thing

- Suppose we have unlimited supplies of
 - Money
 - Time
 - Creativity
- How can we evolve Vic into a real computer?



From Vic to a real computer system: hardware

<u>Vic:</u>	<u>Typical PC:</u>
1 processor	several processors, working simultaneously
1 instruction per cycle	several instructions execute in parallel
1 data register	32 data registers
100 memory cells	Billions of memory cells
1 memory unit	RAM, ROM, cache
No secondary storage	Disk
1 input port	Keyboard, mouse, disk, modem, ...
1 output port.....	Screen, speakers, disk, modem, ...
1 "computer".....	Each I/O device has a dedicated processor and memory space
1 program	several programs running "simultaneously".

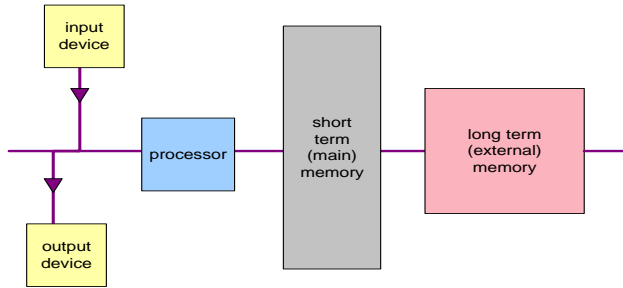


implementing any one of these improvements is a straightforward extension of the basic Vic architecture.

Von Neumann machine (1945)



John V. Neumann
1903 - 1957



Other people who contributed to this model no less than Von Neumann:

- Alan Turing
- Konrad Zuse
- Presper Eckert
- John Mauchly

All (practical) modern computers are Von Neumann machines!



Processor:
RISC ARM 946E



Processor:
Intel Xeon



Processor:
TI OMAP 1710

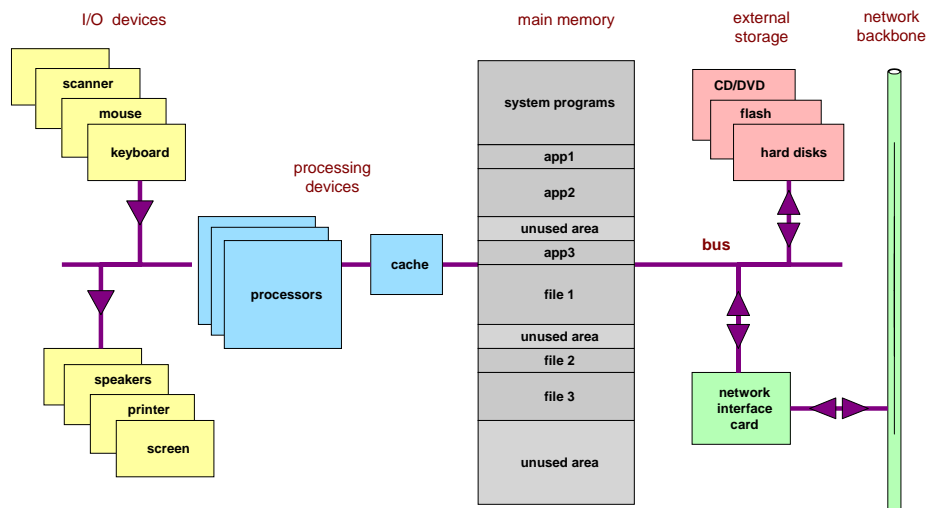


Processor:
IBM PowerPC



Vic

Extending the basic architecture



How it actually looks (thank goodness for abstractions!)

