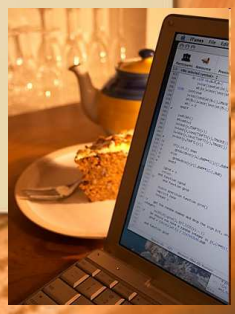
Introduction to Computer Science
Shimon Schocken
IDC Herzliya

Lecture 6-2

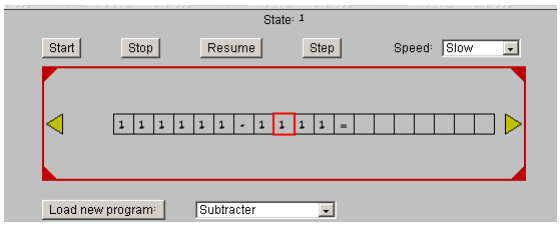
Software Fundamentals

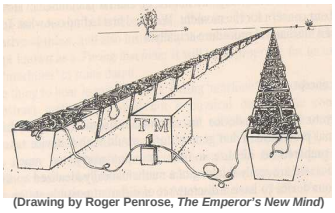


Turing machine

Alan Turing
(1912 - 1954)

"Hardware:"





(Drawing by Roger Penrose, *The Emperor's New Mind*)

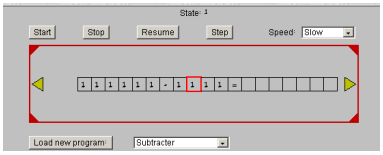
"Software:"

State	character	write	advance	goto
1	sp	-	R	1
1	1	1	R	1
1	=	sp	R	2
2	1	=	L	3
2	-	sp	L	Halt
3	1	1	L	3
3	-	-	L	4
4	sp	sp	L	4
4	1	sp	R	1

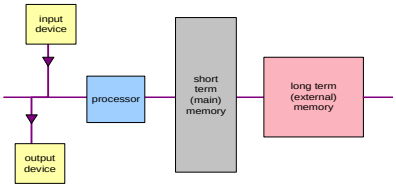
Turing machine:

A venerable computing abstraction, used mainly in theoretical computer science.

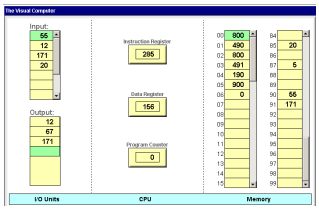
Simple computer models



Turing machine



Von Neumann machine



Vic machine

Q: Which model is more powerful?

A: They are all equivalent

Q: How powerful are they?

A: Surprisingly powerful.

On the expressive power of simple computer models (say, Vic)

We can write programs that compute:

- x + y
- x - y
- x * y
- x / y
- x^y
- $5 * 17^3 - 16 * (-9)^{12} + 11 * 27^5$
- sqrt(X), log(x), sin(x), exp(x), ...
- Etc.

We can also write programs that:

- sum up a series
- count a series
- compute average & variance
- compute *max*(*x*,*y*)
- compute the max of a series
- sort a series
- merge two series
- find an item in a series
- Etc.

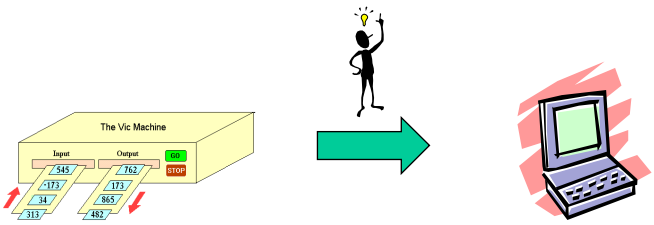
- Big Question: Can a simple computer like Vic do anything we like?
- Big Answer: For any program, running on any computer, there is an equivalent Vic program that does the same thing
- (This follows from the *Church-Turing conjecture*)
- But what about the **efficiency?**

Software fundamentals

- Simple computer models
- Low level language improvements
- High level languages
- Compilation models
- Reverse engineering
- Operating systems
- Applets
- KISS

From a simple computer model to a real computer

- Previous lecture:
  from a simple computer model to a more realistic hardware architecture
- Current lecture: from simple machine language to high level software



- We'll start at the bottom, improving the machine language:
  - More commands
  - Symbolic commands.

Software improvement: more commands

Vic has 10 commands

The instruction set of a typical CPU (e.g. from the MIPS or X86 families) has 100+ commands, supporting at least the following operations:

- Add, subtract
- Multiply, divide

(On both integer and floating point operands)

- And, Or, Not
- Shift operations

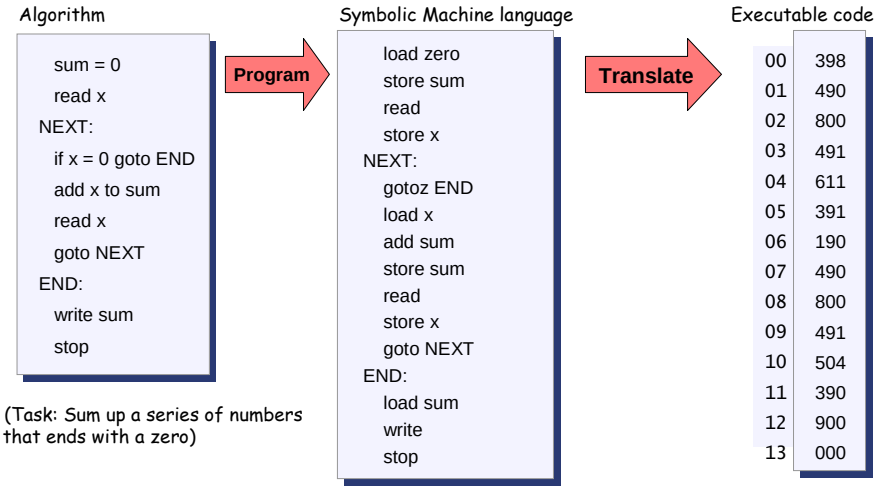
(Bit-wise operations)

- Improved memory access

Modern machine languages also support:

- Subroutines
- Modular programming techniques.

Software improvement: symbolic commands

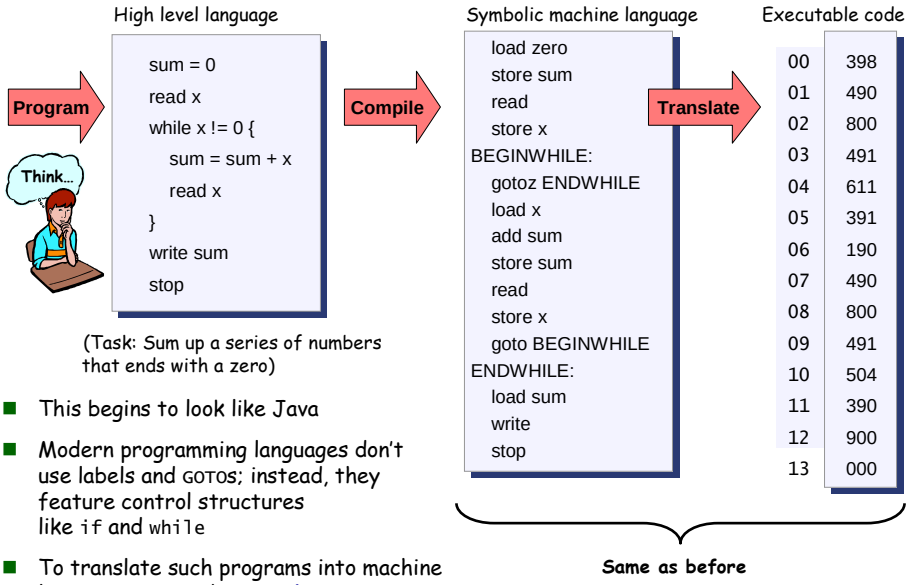


(Task: Sum up a series of numbers that ends with a zero)

Once we move to symbolic commands we no longer have to worry about command numbers and physical memory addresses

But … can we do better than that?

High level language

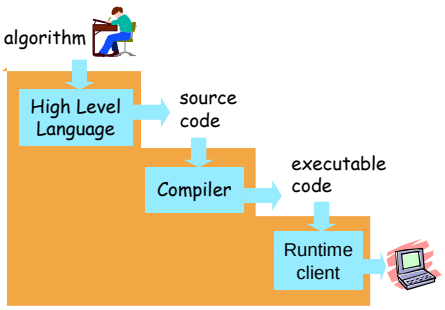


(Task: Sum up a series of numbers that ends with a zero)

- This begins to look like Java
- Modern programming languages don't use labels and GOTOs; instead, they feature control structures like `if` and `while`
- To translate such programs into machine language, we need a **compiler**.

High-level language: a crucial abstraction

- HL programs are easy to read, maintain and extend
- HL programs make no assumptions about the underlying hardware
- The same HL program can be compiled for different computer platforms
  (implications for users, SW companies, HW companies, competition, innovation)



- Users don't have to see the HL source code; They get only the executable code. This way, software companies can protect their intellectual property. Important exception: open source
- Computer scientists can invent all sorts of HL languages, designed for different purposes; As long as we can translate the HL code to machine language, the language can be as abstract as we want (what about English?)

One of the most important ideas in CS: The notion of a HL language abstracts the hardware away from the programmer and the user.

High level programming languages: a Tower of Babel

Programming Style

- Procedural: Machine languages, C, Pascal, …
- Object-oriented: C++, Java, C# , …
- Object-based: JavaScript, Visual Basic, …
- Special paradigms: functional languages, logical languages, …

Purpose:

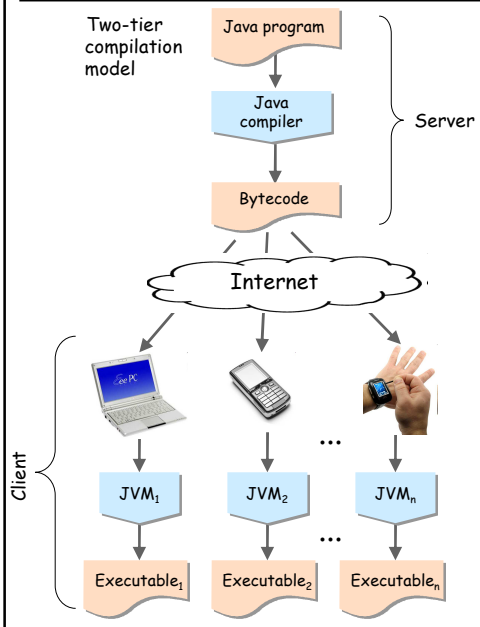
- General purpose: C, Java, C#, Python, …
- Special purpose:
  - Data: SQL, …
  - Text: Latex, …
  - Presentation: HTML, …
  - Scripting: Flash, …
  - Educational: Pascal, …

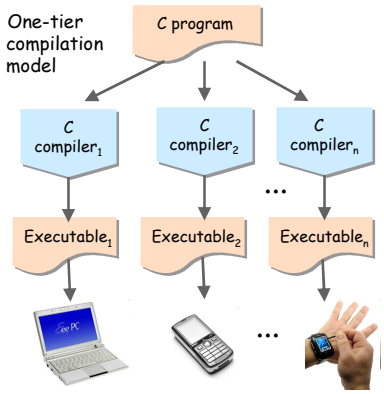
"Java is simple, object-oriented, distributed, interpreted, robust, secure, architecture-neutral, portable, high-performance, multithreaded, and dynamic language." (Sun literature)

Software fundamentals

- Simple computer models
- Low level language improvements
- High level languages
- Compilation models
- Reverse engineering
- Operating systems
- Applets
- KISS

Compilation models

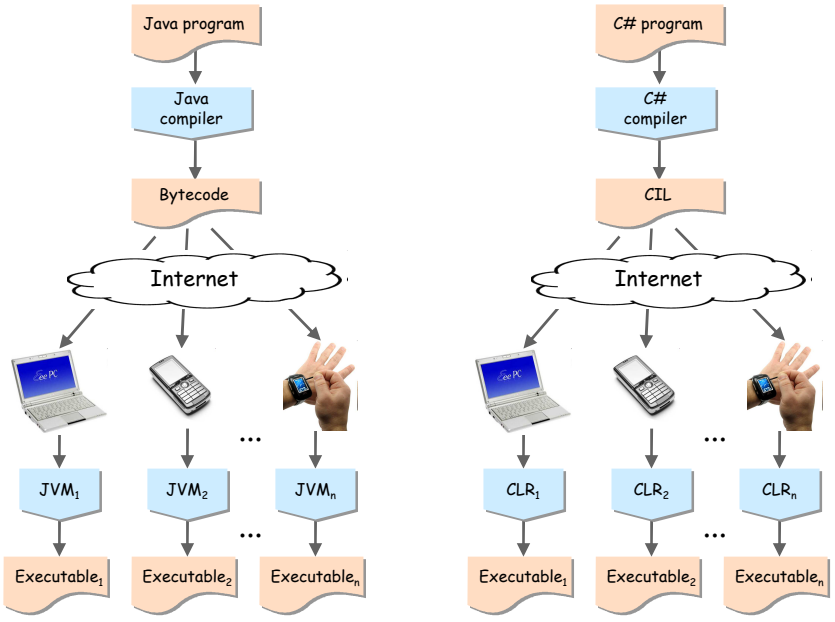




Major characteristics of the resulting code:

- One-tier:
  efficient, not portable, can be unsafe
- Two-tier:
  less efficient, portable, safe.

Java and C#



Java Bytecode

Java source code (`PrintSomeNumbers.java`)

```
// prints the numbers 0 to 5
public class PrintSomeNumbers {
  public static void main(String[] args){
    int i = 0;
    while (i < 6) {
      // print the current value of i
      System.out.println(i);
      i = i + 1;
    }
    System.out.println("Done");
  }
}
```

Compiled bytecode (`PrintSomeNumbers.class`)

```
Compiled from "PrintSomeNumbers.java"
public class PrintSomeNumbers
       extends java.lang.Object{
public PrintSomeNumbers();
  Code:
   0:     aload_0
   1:     invokespecial       #1;
   4:     return

public static void main(java.lang.String[]);
  Code:
   0:     iconst_0
   1:     istore_1
   2:     iload_1
   3:     bipush    6
   5:     if_icmpge     22
   8:     getstatic     #2;
   11:    iload_1
   12:    invokevirtual
   15:    iload_1
   16:    iconst_1
   17:    iadd
   18:    istore_1
   19:    goto      2
   22:    getstatic     #2;
   25:    ldc           #4;
   27:    invokevirtual #5;
   30:    return
}
```

the bytecode of any `FileName.class` file can be viewed via

`javap -c FileName.java`

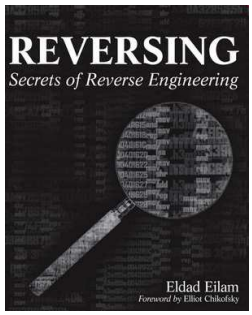
The Java code is translated by the compiler into an equivalent bytecode designed to run on an abstract virtual machine

The bytecode and the VM abstraction are then translated into machine language by the JVM

- The bytecode is CPU-agnostic
- The JVM is CPU-specific.

Reverse engineering

- Decompiler: a program that gets low-level code (e.g. bytecode) and constructs from it high-level code (e.g. Java code)
- Obfuscator: a program that gets a source file as input and obfuscates it to make decompilation hard
- Reverse engineering: trying to build something "backwards":
  - Decompilation
  - From the program's actions and UI
- Reverse engineering is an intricate art that may involve violation of intellectual property.



Compilers VS interpreters

Compiler: Translates from a high-level language into a a lower-level language.

Examples:

- Java compiler (from Java to bytecode)
- C compiler (from C to machine language)

Interpreter: Translates and executes a given code. Each statement is translated and executed separately, in real time. Examples:

- Python interpreter
- Lisp interpreter
- JavaScript interpreter

- Compiled code: translated once and executed for ever (there is no need to have access to the source code);
- Interpreted code: re-translated each time it runs.

Software fundamentals

- Simple computer models
- Low level language improvements
- High level languages
- Compilation models
- Reverse engineering
- Operating systems
- Applets
- KISS

Operating Systems

An operating system (OS) is a software host that runs in the background all the time, providing many services to application programs that run on top of it

User's view of the OS

- GUI
- Files / directories management
- Security
- Communications
- Etc.

Linux

Windows

Programmer view of the OS:

- Disk access
- Device drivers
- Memory management
- Process management
- Etc.

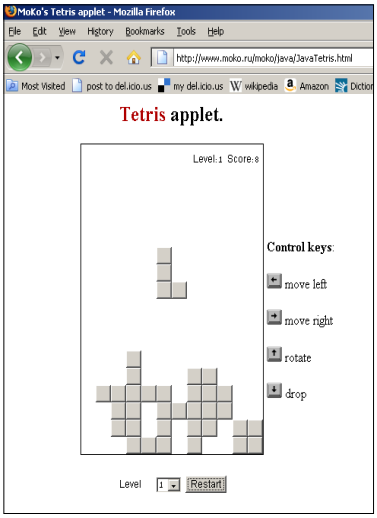
Mac OS

Android

High level languages hide the gory OS details from the programmer.

Java Applets

- Java application: a stand-alone program. The resulting bytecode executes by the JVM residing on a local client computer. Runs in the OS environment
- Java applet: a bytecode file is linked to in an HTML document, transported over the Internet, and executed by a web browser
- How applets work:
  - All major web browsers come with a built-in JVM
  - Therefore, every device that has a web browser can run Java applets
- Realizing Java's vision of portable code, running over the web on multiple clients.

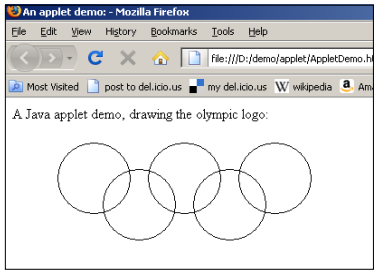


Java Applet anatomy

AppletDemo.html

```
<Html>
  <Head>
    <Title>An applet demo:</Title>
  </Head>
  <Body>
    A Java applet demo, drawing the olympic logo:<br>
    <br>
    <Applet Code="Olympics.class" width=500 Height=250>
    </Applet>
  </Body>
</Html>
```

- This applet uses Java's Graphics classes
- Since the applet runs inside a larger program (browser), it does not require a `Main` method.

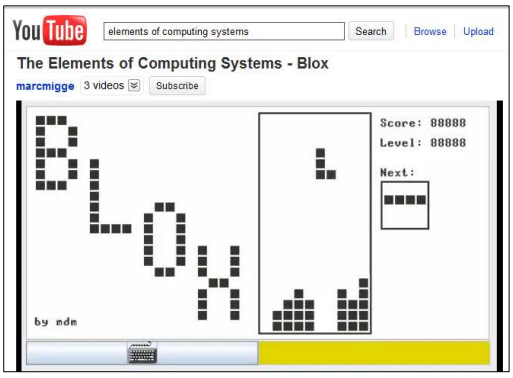


Olympics.java

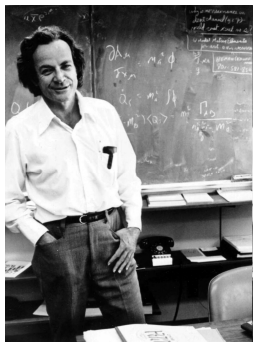
```
import javax.swing.JApplet;
import java.awt.*;
public class Olympics extends JApplet {
  public void paint (Graphics page) {
    page.drawOval( 50, 50, 80, 80);
    page.drawOval(150, 50, 80, 80);
    page.drawOval(250, 50, 80, 80);
    page.drawOval(100, 80, 80, 80);
    page.drawOval(200, 80, 80, 80);
  }
}
```


And if you REALLY want to understand how computers work …

Then the best way to do it is to build a computer from scratch – hardware and software, and then write some computer game on it.



"What I cannot create, I do not understand."
(Richard Feynman)



More details: www.idc.ac.il/tecs

Software fundamentals

- Simple computer models
- Low level language improvements
- High level languages
- Compilation models
- Reverse engineering
- Operating systems
- Applets
- End notes

The road from low level to high level

Looking back:

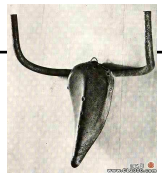
- We started with a very primitive machine language
- Then we got rid of numeric commands, using symbols instead
- Then we got rid of physical addresses, line numbers, and register names
- Then we got rid of labels and GOTO commands
- We ended up with an elegant high level language

In each step we created a new abstraction;
Once we show that the abstraction can be implemented,
we no longer care about the implementation

Computer science consists of thousands of layered abstractions;
The bottom layer is based on transistors and logic gates;
the upper layer is human intelligence

Computer scientists often invent an abstraction (like Vic) and then start playing with it. Good abstractions are simple, expressive, and scalable.

The case for simplicity



The logical principles underlying hardware are simple

The logical principles underlying software are simple

Good science:
Explaining maximum phenomena with minimum rules

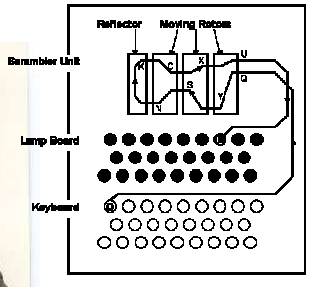
Good engineering:
Creating maximum functionality with a few simple components

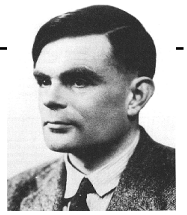
Hardware-software architectures are a prime example of good science leading to good engineering.

"Designers know they have achieved perfection not when there is nothing left to add, but when there is nothing left to take away."
(Antoine de Saint-Exupry)

"All things being equal, simpler solutions tend to be better"
(Occam's Razor)

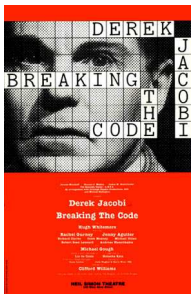
Endnote: Alan Turing and the Enigma





Alan Turing
1912 - 1954

Bletchley Park



- *Great biography:*
  "Alan Turing: The Enigma",
  by Andrew Hodges, Walker & Co., 2000.