School of Science & Technology

www.city.ac.uk

Systems Architecture

IN1006

Systems Software: Operating Systems

—Dr H. Asad

Where are we?

- Components of computers
- Data representation
- Logic Gates – computer circuits
- Simple computer, assembly programming – MARIE
- Memory hierarchy
- Pipelining and parallelism
- System Software

Question?

- What do you think a C compiler does to the following code?

```
f=a+b;
```

Contents

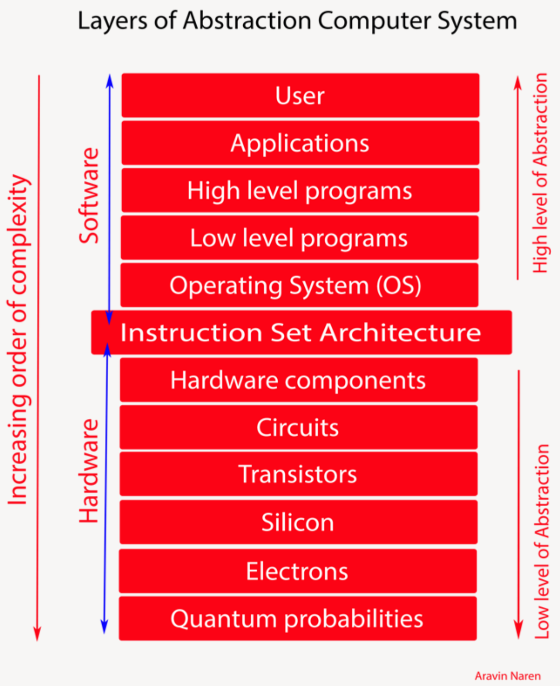
Layers of Abstraction in a Computer System

Until now:

- **Hardware interacts with Software** at the **Instruction Set Architecture**
- It should be clear how a single program can be loaded from memory and executed on the hardware
  - von Neumann architecture (ALU+CU+Regs, Memory)
  - F-D-E cycle
  - The Program Counter keeps track of things

But

- What is running on your machine?
  - Does it look like one program?
  - How is hardware shared amongst multiple programs?



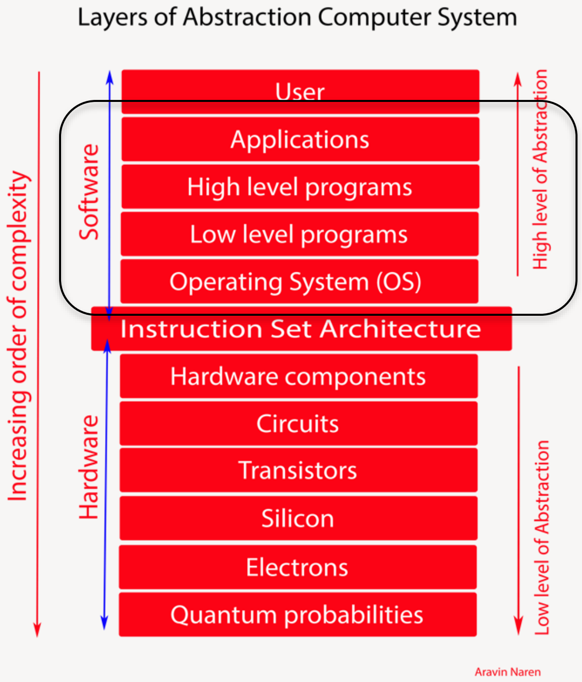
Layers of Abstraction in a Computer System

Until now:

- **Hardware interacts with Software** at the **Instruction Set Architecture**
- It should be clear how a single program can be loaded from memory and executed on the hardware
  - von Neumann architecture (ALU+CU+Regs, Memory)
  - F-D-E cycle
  - The Program Counter keeps track of things

But

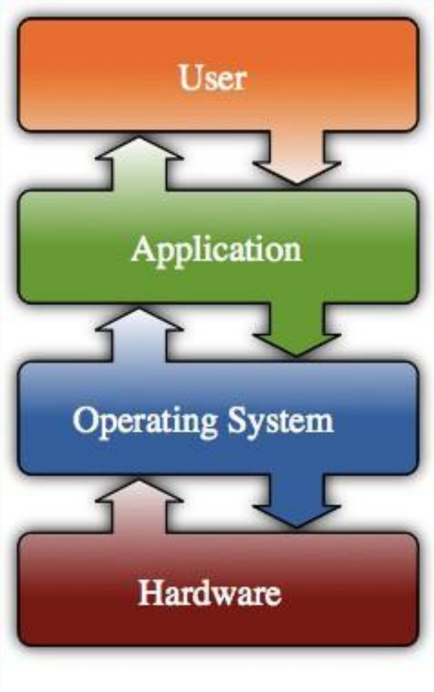
- What is running on your machine?
  - Does it look like one program?
  - How is hardware shared amongst multiple programs?



Our focus tody

Going Up in the Abstraction Hierarchy: the Operating System

- Application-level programs do not directly interface to the “metal”
- There is a software abstraction layer between applications and the hardware
- This abstraction layer is called the **Operating System (OS)**

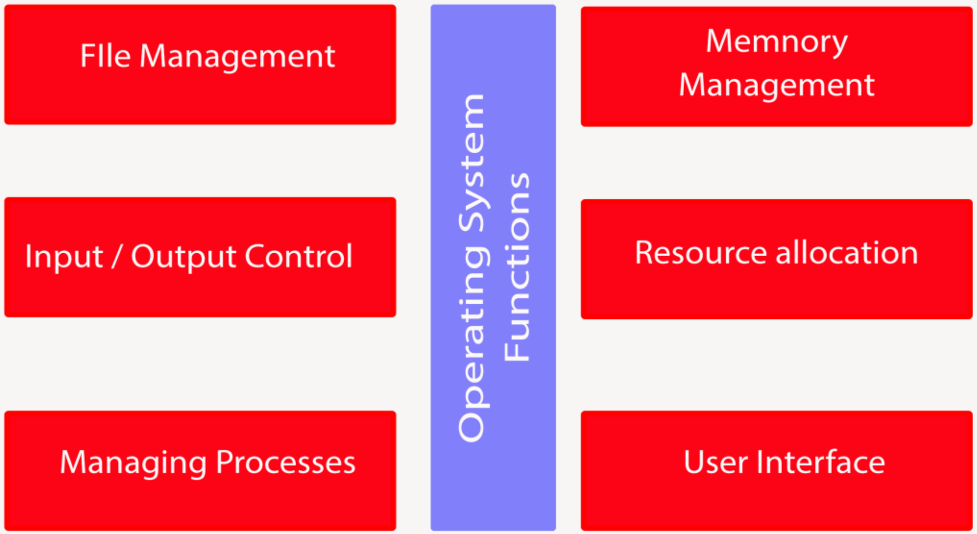


Current (and widely used) Operating Systems

- Operating systems are found on many devices that contain a computer – from cellular phones and video game consoles to web servers and supercomputers.
- Popular OSs include:
    - Windows
    - Mac OSX
    - Linux
    - Unix
- So, what do they do for us, what are the common features?

Operating Systems: Overview of key functions

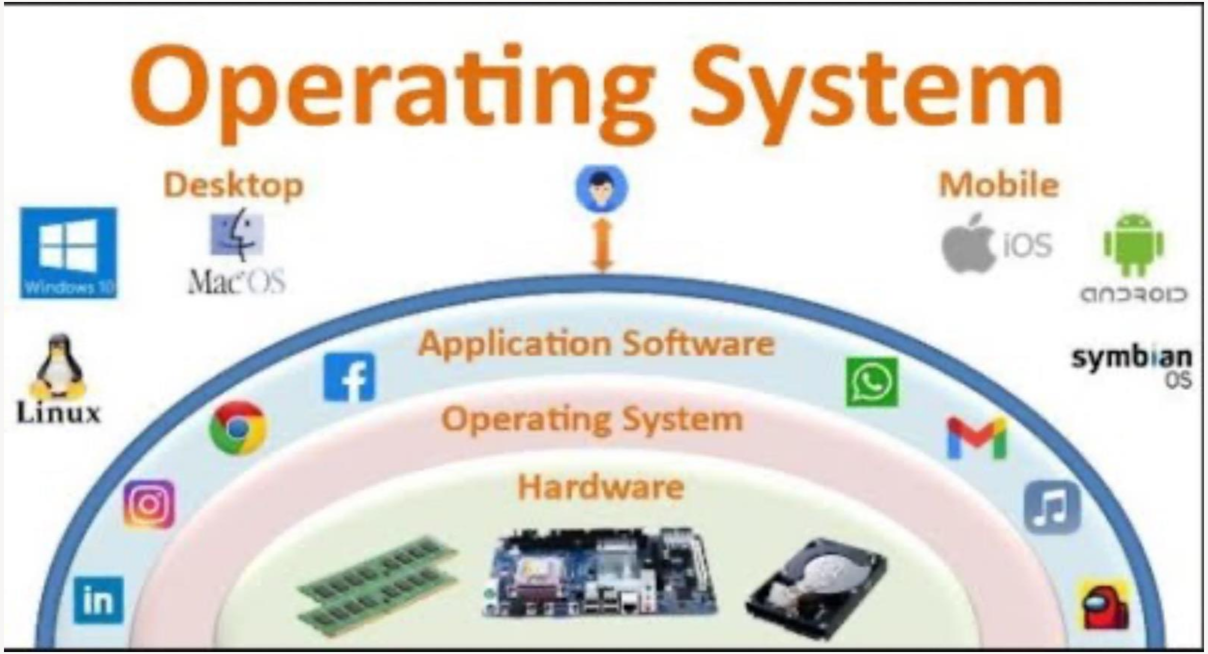
- It **provides a platform** for all computer programs **to execute** on a computer system - It is the hardware interface to the software.
- It **controls the hardware** for the software applications.
- It is a **resource manager** that multiplexes hardware resources for applications
- It **manages files** and **processes**
- It **controls inputs** and **outputs**



Operating Systems: Evolution

- The evolution of operating systems has paralleled the evolution of computer hardware.
- As hardware becomes more powerful, operating systems allowed people to more easily manage the power of the machine.
- In the early days, operating systems were simple **resident monitor programs**.
  - The **resident monitor** could only
    - **Load a program**
    - **Execute** a program
    - **Terminate a program**

Operating Systems: Functions



Operating Systems in the Personal Computer Revolution Era

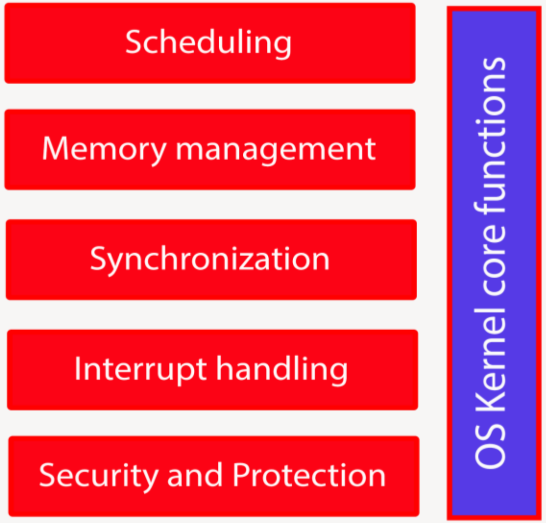
- Personal computer operating systems are designed for ease of use
- The idea that revolutionized small computer operating systems was the **BIOS (basic input-output operating system)**
  - BIOS takes care of the details involved in handling peripheral devices and gets the OS started using a **Boot Loader**
- The Boot Loader starts up automatically and loads the rest of the OS

The (OS) Kernel

As the core of the operating system, the kernel performs some fundamental functions

- Deciding which process gets the CPU and when
- Ensuring that programs don't overwrite each other & virtual memory handling
- Arbitrating shared resources
- Dealing with events outside the simple flow
- Only allowing legal access

These features allow multiple programs to share the same hardware resources.



Scheduling

- The operating system **schedules** process execution
- The operating system determines which process will be granted access to the CPU (**long-term scheduling**)
- For a number of active processes, the operating system determines which one will have access to the CPU at any particular time (**short-term scheduling**)
- **Context switches** occur when the process using the CPU is halted and the CPU is switched to another process
  - The state of the process is preserved during a context switch

Scheduling (cont'd)

- Some **simple approaches to CPU scheduling** are:
  - **First-come, first-served** where jobs are serviced in arrival sequence and run to completion if they have all of the resources they need
  - **Shortest job first** where the shortest job in duration get scheduled first
  - **Round robin** scheduling where each job is allotted a certain amount of CPU time and a context switch occurs when the time expires
  - **Priority** scheduling pre-empts a job with a lower priority when a higher- priority job needs the CPU

Can any of the above approaches cause **process starvation?**

CITY ST GEORGE'S UNIVERSITY OF LONDON

School of Science & Technology

www.city.ac.uk

Systems Architecture

IN1006

Systems Software: Assemblers

—Dr H. Asad

Assemblers

- **Assemblers** are the simplest of all programming tools they **translate mnemonic instructions to machine code**

Assemblers (cont'd)

- Assemblers **create an object program** file **from** mnemonic **source code**
- They do so in **two passes** (of the source code)

1) **First pass:** The assembler assembles as much of the program as it can, while it **builds a symbol table** that contains **memory references for all symbols in the program**

2) **Second pass:** The assembler **completes instructions** using the values from the symbol table

How do Assemblers work: First Pass

It creates

- a **symbol table** storing translations of memory address labels to actual addresses

- A table of **partially-assembled instructions** in which mnemonic instruction names are replaced by instruction codes

Labels instead of instruction codes

Labels instead of memory addresses

Address	Instruction
100	Load X
101	Add Y
102	Store Z
103	Halt
104 X,	DEC 35
105 Y,	DEC -23
106 Z,	HEX 0000

Label, Physical Address

X	104
Y	105
Z	106

Symbol Table

Opcode, operand

1	X
3	Y
2	Z
7 0 0 0	

Partially Assembled Inst

How do Assemblers work: Second Pass

- It replaces operands in partially assembled instruction table with their actual memory addresses as recorded in symbol table

Label, Physical Address

X	104
Y	105
Z	106

Symbol Table

Opcode, operand

1	X
3	Y
2	Z
7 0 0 0	

Partially

Assembled Inst

1 1 0 4
3 1 0 5
2 1 0 6
7 0 0 0
0 0 2 3
F F E 9
0 0 0 0

machine code: opcode, memory location

of operand

How do Assemblers work: First and Second Pass

Address	Instruction
100	Load X
101	Add Y
102	Store Z
103	Halt
104 X,	DEC 35
105 Y,	DEC -23
106 Z,	HEX 0000

1^{st} pass

Label, Physical Address

X	104
Y	105
Z	106

Symbol Table

Opcode, operand

1	X
3	Y
2	Z
7 0 0 0	

Partially

Assembled Inst

2^{nd} pass

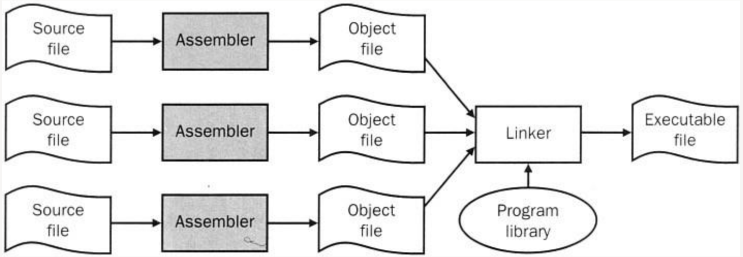
1 1 0 4
3 1 0 5
2 1 0 6
7 0 0 0
0 0 2 3
F F E 9
0 0 0 0

machine code: opcode, memory location

of operand

The Assembler Flow

1. Programmer writes text
2. Creates a `foo.asm` file
3. Calls the assembler to produce a `foo.bin` file
    a. First pass assembles and builds the symbol table
    b. Second pass resolves the labels to actual addresses
4. Calls a **loader** to relocate the program into absolute memory addresses and places it at an appropriate place in memory
5. Control is passed to the start of the program
    a. The PC is loaded with the start address
6. The program runs



Example MARIE Code

- Assembled version is on top right.
- The "+" indicates relative offset

```
    Load x        1+004
    Add y         3+005
    Store z       2+006
    Halt          7000
 x, DEC 35        0023
 y, DEC -23       FFE9
 z, HEX 0000      0000
```

Address	Memory Contents
0x250	1254
0x251	3255
0x252	2256
0x253	7000
0x254	0023
0x255	FFE9
0x256	0000

Loaded Starting at Address 0x250

Address	Memory Contents
0x400	1404
0x401	3405
0x402	2406
0x403	7000
0x404	0023
0x405	FFE9
0x406	0000

Loaded Starting at Address 0x400

School of Science
& Technology

www.city.ac.uk

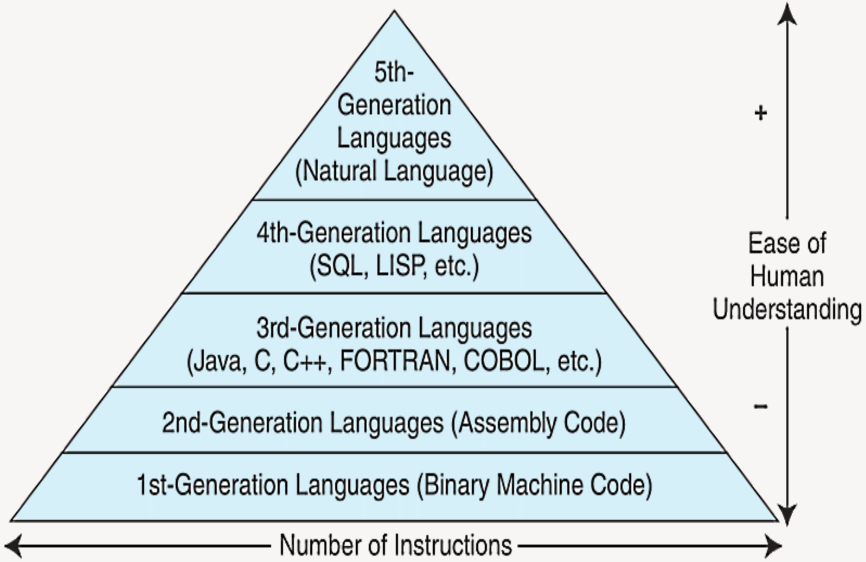
Systems Architecture

IN1006

System Software: Compilers

—Dr H. Asad

The Hierarchy of Programming Languages



Each language generation presents problem solving tools that are closer to how people think and farther away from how the machine implements the solution

The hardware only speaks the lowest level language!

Computer hardware and programming language evolution: a saga of close correlation

Computer hardware generations	Generation of computer language
First-generation: Vacuum tubes and relays. No OS, human operators performed task management	**1st generation languages:** low-level languages that were machine languages
Second-generation: Transistors Batch processing and monitors	**2nd generation languages:** Low-level assembly languages. Used in kernels and hardware drives, commonly used for video editing and video games.
Third-generation: Integrated circuits Timesharing, virtual memory, multiprogramming, modern OSes	**3rd generation languages:** High-level languages, such as C, C++, Java, JavaScript, and Visual Basic.
Fourth-generation: VLSI Network operating systems, distributed systems, virtualization, cloud	**4th generation languages:** Similar to statements in a human language. Database programming and scripting languages (e.g., Perl, PHP, Python, Ruby, and SQL)
	5th generation languages: Visual tools to help develop a program, visual programming. Examples of fifth generation languages include Mercury, OPS5

More types of system software

(due to the increasing abstraction in level of programming)

- Compilers
- Virtual machines

Compilers

- Compilers are for high level languages (HLL) what assemblers are for low level languages: they **translate from the HLL to machine code**
- Compilers bridge the semantic gap between the higher level language and the machine's binary instructions
- Most compilers effect this translation in a **six-phase process.**

  The first three are

  analysis phases:

  1) **Lexical analysis** extracts tokens, e.g., reserved words and variables
  2) **Syntax analysis** (parsing) checks statement construction
  3) **Semantic analysis** checks data types and the validity of operators

Compilers (cont'd)

Consider the program:

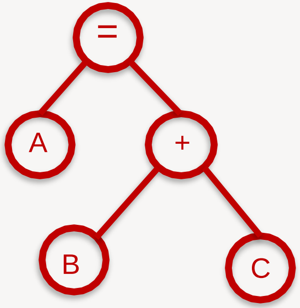
Int A; Int B; Int C; A=B + C;

(1) Lexical analysis extracts tokens, e.g., reserved words and variables

Tokens: A, =, B, +, C

(2) Syntax analysis (parsing) checks statement construction

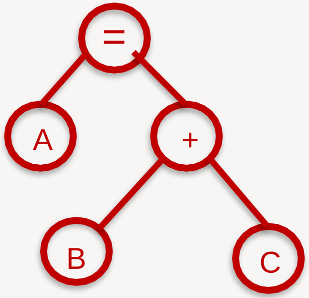
Abstract Syntax Tree:



(3) Semantic analysis checks data types and the validity of operators

IF

Int A;
Int B;
Int C;



Ok ✔

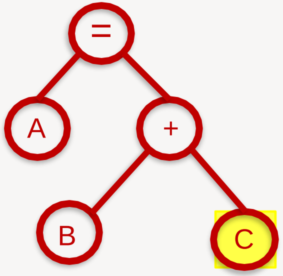
BUT IF

Int A;

Int B;

String C;

ERROR !



Semantic analysis

Compilers (cont’d)

The last three compiler phases are synthesis phases:

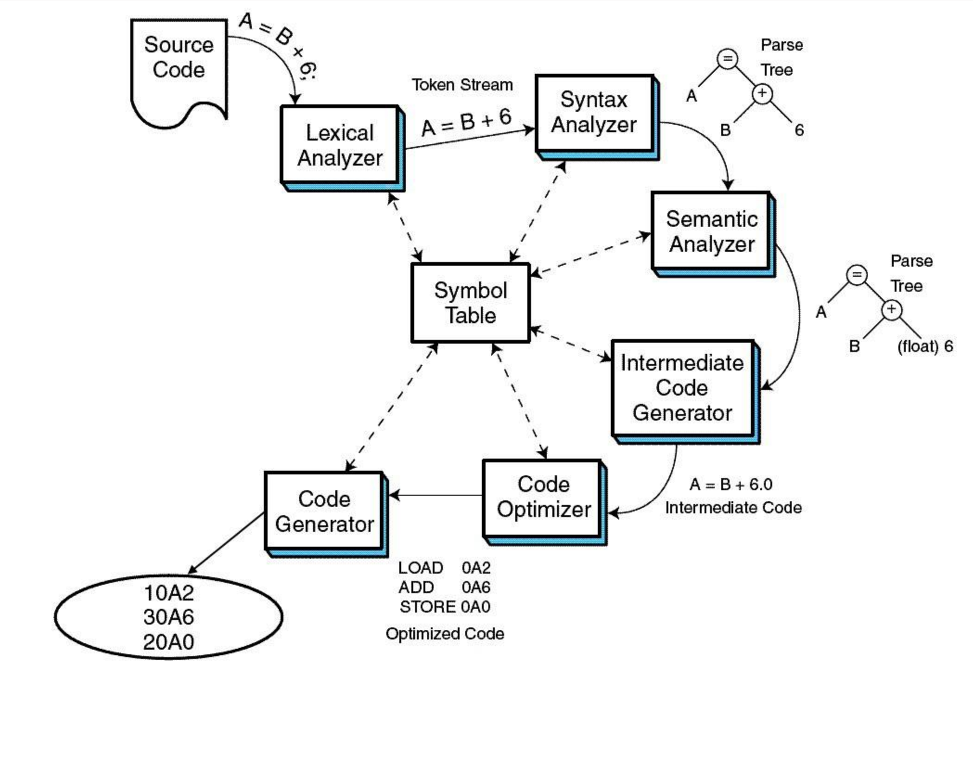
(4) **Intermediate code generation** creates **assembly level code** to facilitate optimisation and translation.

(5) **Optimisation** creates more efficient assembly code (**example from previous lectures**)

(6) **Code generation** creates binary code from the optimised assembly code (this might be using the assembler)

Through this modularity, compilers can be written for various platforms by **rewriting only the last two phases**, making HLLs machine independent

The Six Phases of Program Compilation



An Example of Compilation

- If we have a statement in a high level language

  F = (a + b) – (c + d)
- Then we would render that in a (simple) assembly language as

```
LOAD a
ADD  b
STORE tmp1
LOAD c
ADD d
STORE tmp2
LOAD tmp1
SUBTR tmp2
STORE F
```

- Which would then go through the Link and Load process to run

CITY
ST GEORGE'S
UNIVERSITY OF LONDON

School of Science & Technology

www.city.ac.uk

Systems Architecture

IN1006

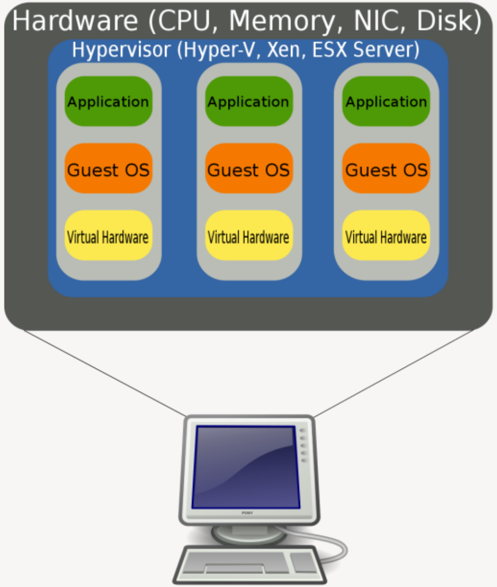
Systems Software
Java Virtual Machine

—Dr H. Asad

Virtual Machines

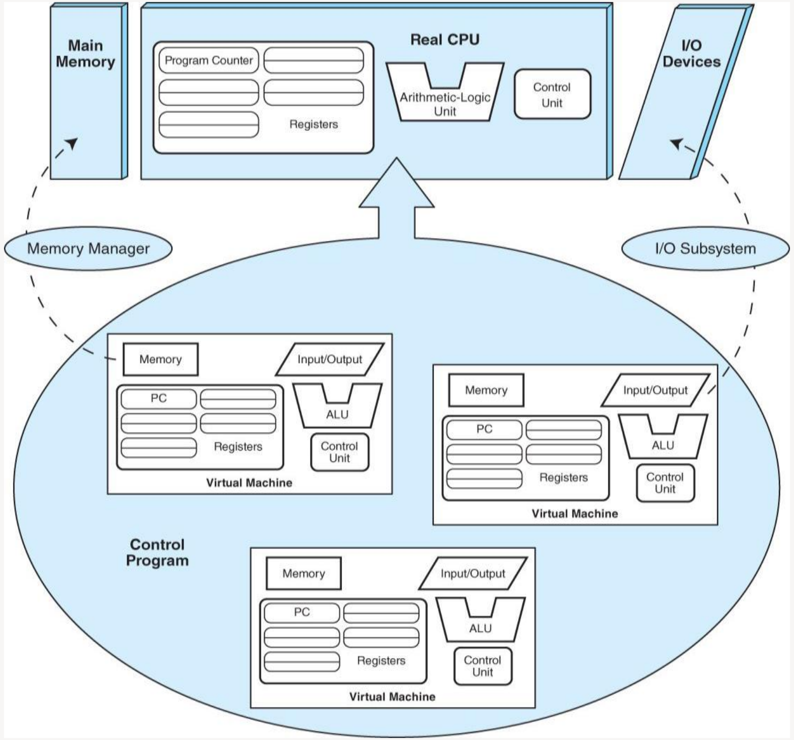
- **Virtual machine** is the **virtualization** or **emulation** of a computer system.

- A virtual machine is exactly that: **an imaginary computer**.
- The underlying real machine is under the control of the kernel – which receives and manages all resource



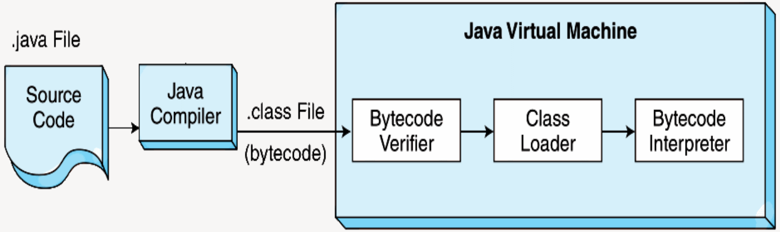
https://en.wikipedia.org/wiki/Virtual_machine#/media/File:Hardware_Virtualization_(copy).svg

Virtual Machines (cont’d)



What about the world you program in: JVM

- The **Java Virtual Machine (JVM)** is an operating system in miniature.
  - It loads programs, links them, starts execution threads, manages program resources, and deallocates resources when the programs terminate.



Simple.java

```
public class Simple {
  public static void main (String[ ] args) {
    int i = 0;
    double j = 0;
    while (i < 10) {
     i = i + 1;
     j = j + 1.0;
    } // while
  } // main()
} // Simple()
```

To run a Java Program

- At execution time, a Java Virtual Machine must be running on the host system
- It loads and executes the bytecode class file (i.e., the “machine code”)
- Steps in the JVM:
  1) **Bytecode verifier:** the JVM verifies the integrity of the bytecode
  2) **Class loader:** Loads the bytecode (of classes) in memory and whilst doing so it performs a number of run-time checks
  3) The loader invokes the **bytecode interpreter** for execution

Binary Image of Simple.class

	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+A	+B	+C	+D	+E	+F	Characters
000	CA	FE	BA	BE	00	03	00	2D	00	0F	0A	00	03	00	0C	07	-
010	00	0D	07	00	0E	01	00	06	3C	69	6E	69	74	3E	01	00	<init>
020	03	28	29	56	01	00	04	43	6F	64	65	01	00	0F	4C	69	()V Code Li
030	6E	65	4E	75	6D	62	65	72	54	61	62	6C	65	01	00	04	neNumberTable
040	6D	61	69	6E	01	00	16	28	5B	4C	6A	61	76	61	2F	6C	main ([Ljava/l
050	61	6E	67	2F	53	74	72	69	6E	67	3B	29	56	01	00	0A	ang/String;)V
060	53	6F	75	72	63	65	46	69	6C	65	01	00	0B	53	69	6D	SourceFile Sim
070	70	6C	65	2E	6A	61	76	61	0C	00	04	00	05	01	00	06	ple.java
080	53	69	6D	70	6C	65	01	00	10	6A	61	76	61	2F	6C	61	Simple java/la
090	6E	67	2F	4F	62	6A	65	63	74	00	21	00	02	00	03	00	ng/Object !
0A0	00	00	00	00	02	00	01	00	04	00	05	00	01	00	06	00	
0B0	00	00	1D	00	01	00	01	00	00	00	05	2A	B7	00	01	B1	*
0C0	00	00	00	01	00	07	00	00	00	06	00	01	00	00	00	01	
0D0	00	09	00	08	00	09	00	01	00	06	00	00	00	46	00	04	F
0E0	00	04	00	00	00	16	03	3C	0E	49	A7	00	0B	1B	04	60	< I `
0F0	3C	28	0F	63	49	1B	10	0A	A1	FF	F5	B1	00	00	00	01	<(cI
100	00	07	00	00	00	1E	00	07	00	00	00	03	00	02	00	04	
110	00	04	00	05	00	07	00	06	00	0B	00	07	00	0F	00	05	
120	00	15	00	09	00	01	00	0A	00	00	00	02	00	0B	00	3D	=

Summary: The Complete Journey from Java to transistors

- By this point in the module, you now know enough to understand how a statement you write in Java actually runs as electrical values on transistors (or at least logic values on gates)
- You should understand, at the conceptual level, everything from the silicon up through to programs you write at the application level, and how they build on lower levels
- This abstraction stack is no longer mysterious to you and you really do understand how a computer works

Summary

- Operating system as an abstraction of the hardware
- Operating system features
  - Multiprocessing
  - Scheduling
  - Protection
- Assemblers
- High Level Languages
- Compilers and other Tools
- A look at Java

School of Science & Technology
City, University of London
Northampton Square
London
EC1V 0HB
United Kingdom

T: +44 (0)20 7040 5060
E: SST-ug@city.ac.uk
www.city.ac.uk/department

Acknowledgement:
Prof George Spanoudakis