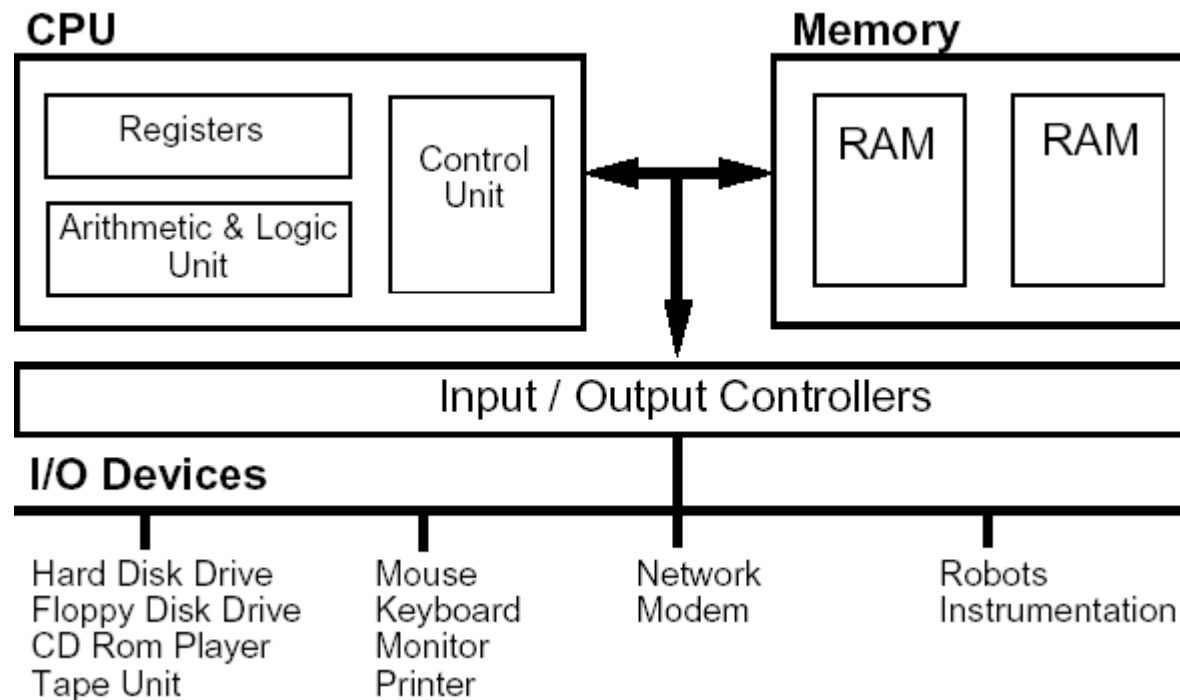


Introduction to computing

- Computers have become increasingly powerful, computing power increased exponentially in the last decades (Moore's law).
- Computers have become indispensable tools in science: simulations, modelling, numerically solving equations, image processing, data analysis.
- Many jobs require computer proficiency that goes beyond word processing and excel tables.

Computer architecture



CPU: Central Processing unit

RAM: Random Access Memory

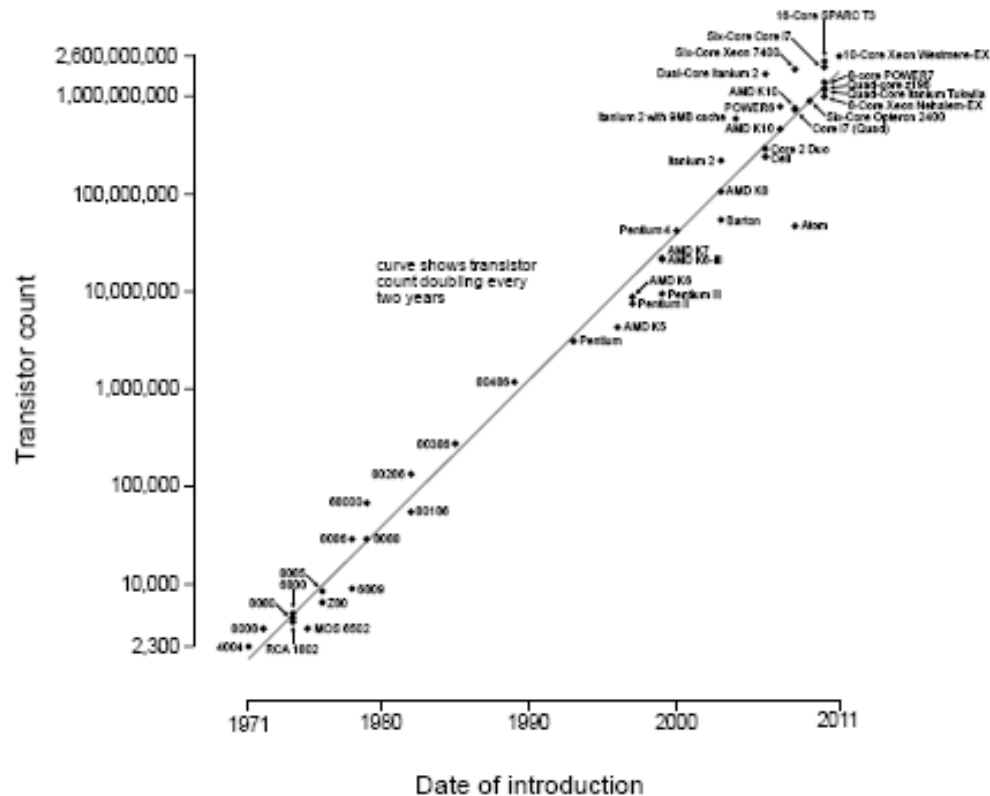
I/O: Input/Output

Nowadays, also have powerful GPU's (Graphical Processing Unit) that is not only used for games, but also parallel computing (NVIDIA CUDA)

Moore's law

- (Self-fulfilling) prediction by Gordon Moore (Intel co-founder)
- Transistors count on chips doubles every 2 years.
- Soon will reach its physical limit (~10 nm width)

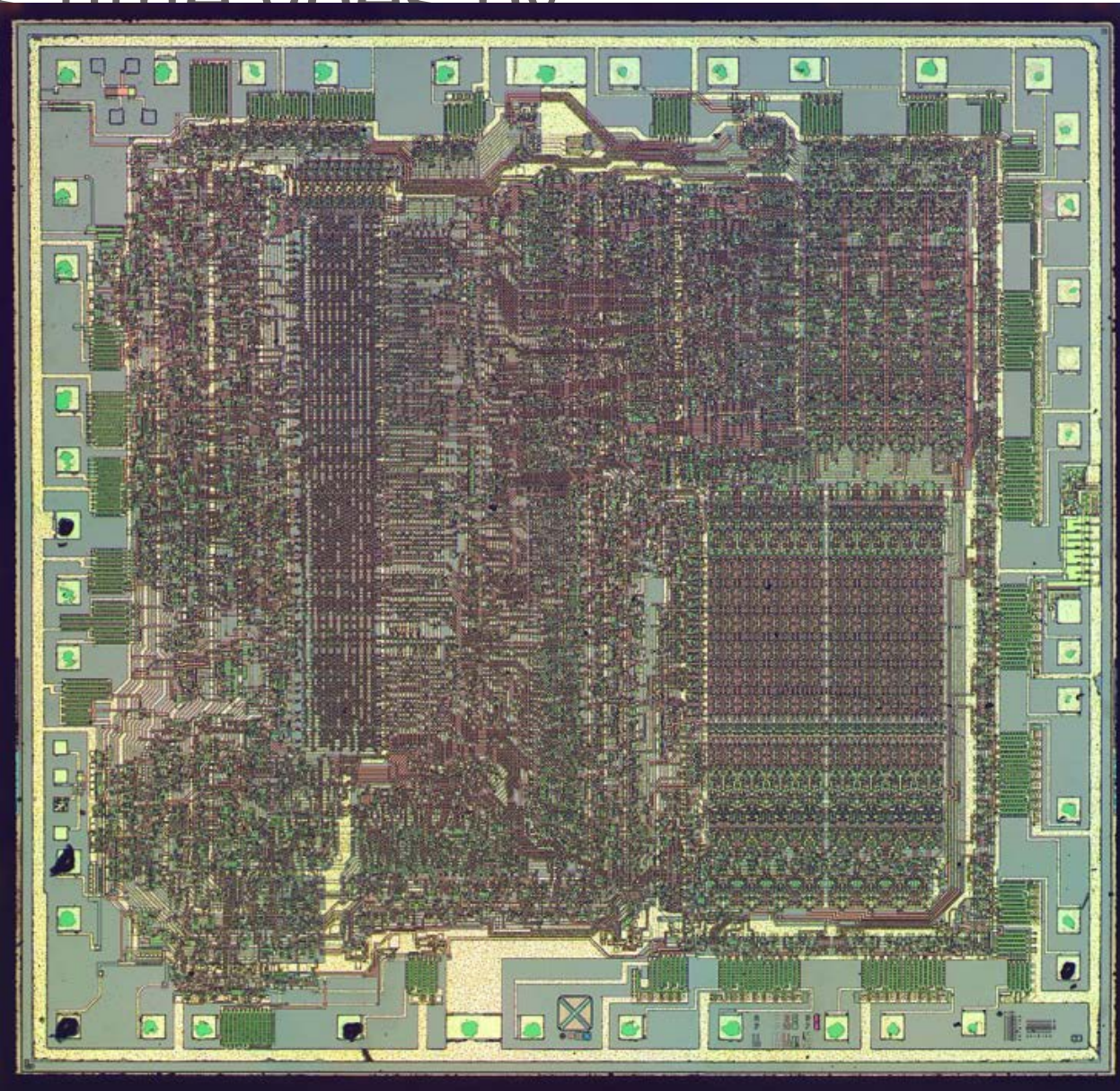
Microprocessor Transistor Counts 1971-2011 & Moore's Law



As time goes by



1984
CPU:
Featu
Clock
RAM
Data



Bits and bytes

- Transistors on computer chips act like on and off switches.
- On the lowest level, chips operate using Boolean logic

A	B	A AND B	A OR B	NOT A
False	False	False	False	True
False	True	False	True	True
True	False	False	True	False
True	True	True	True	False

- These on/off (True/False) states can be represented by bits, which are binary numbers: 0110011
- Information is stored in bits, and calculations on the chip are performed using Boolean logic.

Bits and bytes

- 8 bits: 01001101, there are $2^8 = 256$ different 8 bit numbers.
- 8 bits = 1 byte, 1 kilobyte=1024(= 2^{10}) bytes, 1 Megabyte = 1024 kilobyte, etc.
- In computing numbers are often expressed in hexadecimals. This allows 8 bit numbers to be expressed in 2 digits (0-FF) rather than 3 (0-255).
- Hexadecimals: Digits run from 0 to F:
0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
- 16 digits, therefore a 2 digit hexadecimal number can express $16^2 = 256$ different numbers.

Types of computer languages

```
MONITOR FOR 6802 1.4          9-14-80  TSC ASSEMBLER  PAGE    2

C000                ORG      ROM+$0000 BEGIN MONITOR
C000 8E 00 70  START  LDS      #STACK

                *****
                * FUNCTION: INITA - Initialize ACIA
                * INPUT: none
                * OUTPUT: none
                * CALLS: none
                * DESTROYS: acc A

0013                RESETA EQU    %00010011
0011                CTLREG EQU    %00010001

C003 86 13          INITA  LDA A  #RESETA  RESET ACIA
C005 B7 80 04                STA A  ACIA
C008 86 11                LDA A  #CTLREG   SET 8 BITS AND 2 STOP
C00A B7 80 04                STA A  ACIA
```

Assembly (CPU dependent):
Imperative, low-level language. Program CPU directly. Commands have direct correspondence with machine code.

Functional programming (C, Fortran, Python):

Data is manipulated by functions that have well defined inputs and outputs. Functions are independent of the data (like a black box). There is a main program that defines the data structures and has flow control and calls the functions.

Types of computer languages

Object orientated (C++, Java, Python)

Data structures and functions are merged into objects.

e.g. windows are objects that have attributes (size, title, scrollbar etc.) and methods (functions) acting on them (window closing, window opening, window resizing)

Note: There is no “best” language. Each language has its own advantages and disadvantages and the choice depends on the task at hand.

For scientific computing, functional programming is appropriate and sufficient in many cases.

Example of an object - circle

```
from numpy import pi,sqrt
```

```
class circle:
```

```
    def __init__(self,cx,cy,r):
```

```
        self.cx = cx
```

```
        self.cy = cy
```

```
        self.r = r
```

```
    def area(self):
```

```
        return self.r*self.r*pi
```

```
    def inside(self,x,y):
```

```
        dist = sqrt((x-self.cx)*(x-self.cx)+(y-self.cy)*(y-self.cy))
```


```
        if dist<= self.r:
```

```
            return True
```

```
        else:
```


```
            return False
```

3 attributes: x,y center
position and radius



Two methods: calculate area of
circle

Find out if a certain point in the
xy plane is inside the circle



Creating objects, using methods

```
>>> from circle import *
>>> C=circle(1,-1,3)
>>> C.cx
1
>>> C.cy
-1
>>> D=circle(2,0,2)
>>> C.area()
28.274333882308138
>>> D.area()
12.566370614359172
>>> C.inside(-1,0)
True
>>> D.inside(-1,0)
False
```

What computer languages do scientists use?

- For simulations with heavy computing requirements use compiled languages such as C, C++, Fortran as they are the fastest.
- For data analysis, image processing and simulations with moderate computing requirements use interpreted languages such as Python, Matlab or IDL (popular with astrophysicists). Slower, but huge built-in functionality/libraries.
- Python is open source and free and has similar capabilities as commercial software such as Matlab or IDL. However, documentation is not as good and also contains more bugs.
- For simple image processing tasks can also use ImageJ (popular with biologists)

Interpreted versus compiled languages

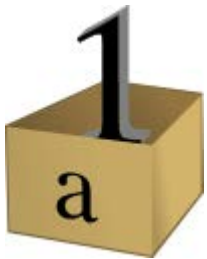
- Compiled languages translate the source code into a binary executable (machine code) via a compiler.
- Common compiled languages are C, C++, Fortran.
- Advantages: very fast, compiler optimises code execution (e.g. memory allocation).
- Disadvantages: executable not platform/processor independent.
- Interpreted languages parse the code for its meaning and execute precompiled code.
- Common interpreted languages: Python, Java, Matlab, IDL
- Advantages: Platform independence - source code will run on any platform. Much more built-in functionality for scientific computing.
- Disadvantages: Slower compared to compiled languages.

Variables versus objects

Variables such as in C

```
int a = 1;
```

Assigning a number to a variable in C, puts the value in a box (i.e. a location in the memory). 'a' refers to a specific location in the memory



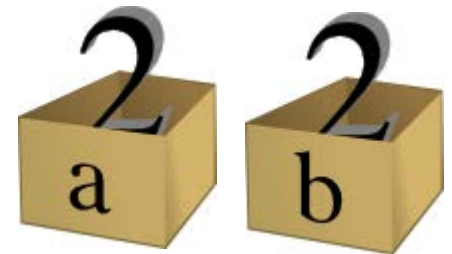
```
a = 2;
```

Reassigning the variable with a different value, replaces that value in the memory



```
int b = a;
```

Assigning one variable to another makes a copy of the value and puts it in the new box (i.e. a new location in the memory)



Variables versus objects

Objects in Python

In Python, a "name" or "identifier" is like a parcel tag (or nametag) attached to an object.

`a = 1`

Assigning a name to an integer object with value 1.



`a = 2`

Reassigning the name 'a' attaches it to a different integer object with value 2.



`b = a`

This attaches another "name" to the same object. Both 'a' and 'b' refer to the same object.



Why does this matter ?

In Python:

```
a=[5]
```

```
b=a
```

```
b.append(1)
```

```
print a
```

Output: [5,1]

In most other languages such as C, 'a' and 'b' would be two separate entities in different memory locations. 'a' would just contain [5]

Although we commonly refer to “variables” in Python, they are different compared to variables in most other languages. They are names (identifiers) of objects.

How are variables passed in to functions?

In C, variable(s) can be passed into functions via two methods:

- Call by value: A copy of the variable is created in memory. Any changes that are made to the variable inside the function will not carry over to the main program.
- Call by reference: Only the memory address (Pointer) of the variable is passed into the function – not its value. Any changes to the variable inside the function will carry over to the main program.

How are variables passed in to functions?

In Python, everything is an object, so we have

Call by object reference:


Name ('identifier') of the object is passed. If name is reassigned to a different object inside the function, then object outside the function remains unchanged.

Example:

```
def add2list_wrong(list):  
    list=list+[1]
```

```
list=[0]  
add2list_wrong(list)  
print list
```

```
>>> [0]
```



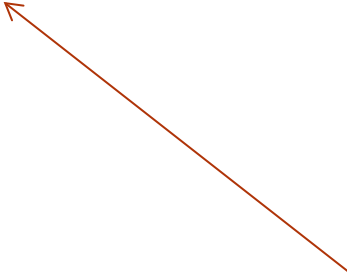
`list+[1]` is a new object. The name 'list' is now referring to this new object

Correct way

```
def add2list(list):  
    list.append(1)
```

```
list=[0]  
add2list(list)  
print list
```

```
>>> [0,1]
```



`list.append(1)` is a method acting on the 'list' object. `list` is not reassigned to a new object inside the function

Alternative (less elegant) way

```
def add2list(list):  
    list=list+[1]  
    return list
```

```
list=[0]  
list=add2list(list)  
print list
```

```
>>> [0,1]
```

Methods vs. functions

Example: Sum elements in an array

```
List=np.array([1,2,3])
```

```
# use function to compute sum
```

```
print np.sum(List)
```

```
# Alternatively, use method associated with array object
```

```
print List.sum()
```

For a full list of methods for numpy arrays see

<http://docs.scipy.org/doc/numpy/reference/arrays.ndarray.html#array-methods>

Good programming practice/tips

Choose descriptive variable names

- Often code is shared with other people. Short, cryptic variable names make it difficult to read the code for everybody – including yourself.
- If you resume working on your code after a few days or week, you will not remember what the variable name stands for.
- E.g. variable names such as 'm2' are not very illuminating. Try to use 'mass_partcl2' instead.
- While it makes the code more verbose and requires more typing, you will save time in the long term. Many compilers/editors also support autofill.

Good programming practice/tips

No hardcoding

- Always use variables to store numerical parameters that are used throughout the program. Define at the beginning of the code if possible.
- E.g. `mass=10.5` . If you want to run the program with a different value for the mass, you only need to change it in one place.
- Also makes code much easier to read.

Comment your code

- Always comment your code.
- When defining functions, comment what the inputs and outputs are.
- Again, makes it easy to understand the code for others and yourself.

Good programming practice/tips

For larger programs, plan ahead, make a flow diagram if necessary

- Define the scope of your code – what should it do and how?
- Make a flow diagram to illustrate what.
- Modularise your code into functions that perform specific tasks that are performed many times.

Common mistakes

- Unlike C and many other languages, variables do not have to be declared explicitly.
- E.g. in C, integer variables are declared as `'int a =10;'`
- In Python, the data type is determined when a value is assigned to a variable. E.g. `'a=10'` is an integer, while `'a=10.'` is a floating point (Note, that by default, floating point numbers have 64 bit precision.)
- This can lead to mistakes, especially integer division:

`'10/20=0'`, but `'10./20=0.5'`

Explicitly convert integers to floating point numbers if necessary:

`float(a)`

Good programming practice/tips

Don't reinvent the wheel and make use of built-in functions/methods

- Python has a lot of in-built functionality for manipulating data structures.
- It is worth browsing the Python documentation (<http://docs.scipy.org/doc/>), especially numpy as it contains all the array manipulation methods/functions that are commonly used in scientific computing

Speed

- Python is an interpreted language. Some commands can be much slower than others.
- In general, avoid loops whenever you can – especially if the number of loops is large.

Example: Numerical integration with the trapezoidal rule

$$\begin{aligned}\int_a^b f(x)dx &\approx \frac{h}{2} \left[f(a) + f(b) + 2 \sum_{i=1}^{n-1} f(x_i) \right] \\ &= \frac{h}{2} \left[f(a) + f(b) + 2 \sum_{i=0}^n f(x_i) \right]\end{aligned}$$

where $a = x_0$ and $b = x_n$. $f(x)$ is discretised with n equally spaced points x_i between a and b .

Let's write a function that evaluates that sum.

Version 1 – for loop

```
def fct(x):  
    return x**2 #  $f(x)=x^2$   
  
def trapezoid(fct,a,b,n):  
    h = (float(b)-float(a))/float(n)  
    total = 0.0  
    for i in np.arange(n+1):  
        total += fct(a+i*h)  
    total *= 2.0  
    total -= fct(a)+fct(b)  
    return total*h/2.0
```

Version 2 – use vectorisation

```
def arraytrapezoid(fct,a,b,n):  
    h = (float(b)-float(a))/float(n)  
    x = np.arange(a,b+h,h)  
    y = 2.0*fct(x)  
    y[0] = y[0]/2.0  
    y[n] = y[n]/2.0  
    return np.sum(y)*h/2.0
```

Both functions perform the same operation. Input is a function, limits of integration and number of discretisation points.

However, for $n = 10^6$, trapezoid takes 11-12 seconds, while arraytrapezoid needs only 0.5 seconds (but needs more memory).

20x faster! When dealing with large data sets, avoiding for loops becomes much more important.

What software do scientists use?

Writing articles/reports, giving presentations

Word, Powerpoint

Advantages: WYSIWYG – what you see is what you get, freedom in layout and style

Disadvantages: Entering formulae requires several mouse clicks and can be cumbersome if there are many equations to type in.

Latex (Typesetting languages)

Advantages: Easy to type in formulae

e.g. $\frac{\alpha}{\beta}$,
`\frac{\alpha}{\beta}`

easy referencing and labelling of equations and figures.

Disadvantage: Need to compile text to view, limited freedom in layout.

Plotting/curve fitting

- On Windows machines, most common plotting program is *Origin*. Unfortunately, it is not free, but TCD has site license (used in SF physics onwards).
- On Linux, have *xmgrace* and *gnuplot*
- On Macs, have *Abcissa*, *Plot2*
- On all platforms, can use *matplotlib* package from Python
- Here at TCD, we have a license for Logger Pro (Windows and Mac, no Linux version)