Computer Fundamentals

Dr Robert Harle

CST Part IA

NST Part IA (CS option)

PPS (CS option)

Michaelmas 2011

What is Computer Science?

- Surprisingly hard to answer definitively
  - Gets confused with IT, which is merely the *use* of present day technology

- We're trying to teach theory and practice that will defined *future* technology
  - CS has strong theoretical underpinnings that stem from maths

- This short course is introductory material that touches on the absolute basics
  - Examined **indirectly –** no specific exam question but the topics surface in later courses throughout the year

Topics

- Computer Components
  - Brief history. Main components: CPU, memory, peripherals (displays, graphics cards, hard drives, flash drives, simple input devices), motherboard, buses.
- Data Representation and Operations
  - Simple model of memory. Bits and bytes. Binary, hex, octal, decimal numbers. Character and numeric arrays. Data as instructions: von-Neumann architecture, fetch-execute cycle, program counter (PC)
- Low- and High- level Computing
  - Pointers. The stack and heap? Box and Pointer Diagrams. Levels of abstraction: machine code, assembly, high-level languages. Compilers and interpreters. Read-eval-print loop.
- Platforms and Multitasking
  - The need for operating systems. Multicore systems, time-slicing. Virtual machines. The Java bytecode/VM approach to portability. ML as a high-level language emphasing mathematical expressivity over input-output.

A Brief History of Computers

Analogue Computers

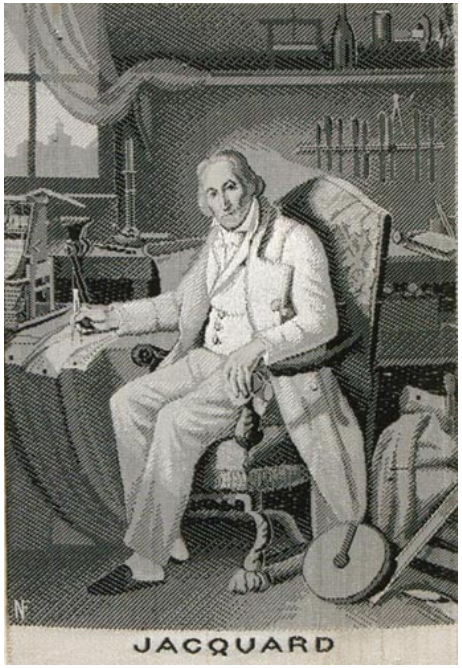
- You've probably been taught various electrical phenomena by analogy with mechanical systems
  - Voltage ↔ water flow
  - Electrical resistance ↔ mechanical resistance
  - Capacitance ↔ compressed spring

- Works the other way: simulate mechanical systems using electrical components
  - This is then an analogue computer
  - Cheaper, easier to build and easier to measure than mechanical system
  - Can be run faster than 'real time'
  - BUT each computer has a specialised function

- Very good for solving differential equations. Used extensively for physics, esp. artillery calculations!

Input: Jacquard's Loom

- Not a computer per-se, but very important in the history of them. Jacquard wanted to create a textile loom that could remember how to create specific textiles
- Used many needles and realised he could create a series of template cards with holes to let through only some needles. Running a series of templates through in a specific order produced the garment.
- Basic idea for punch cards



Turing Machines

- Inspired by the typewriter (!), Alan Turing (King's) created a theoretical model of a computing machine in the 1930s. He broke the machine into:
  - A tape – infinitely long, broken up into cells, each with a symbol on them
  - A head – that could somehow read and write the current cell
  - An action table – a table of actions to perform for each machine state and symbol. E.g. move tape left
  - A state register – a piece of memory that stored the current state

Universal Turing Machines

- Alan argued that a Turing machine could be made for any computable task (e.g. sqrt etc)
- But he also realised that the action table for a given turing machine could be written out as a string, which could then be written to a tape.
- So he came up with a **Universal Turing Machine**. This is a special Turing Machine that reads in the action table from the tape
  - A UTM can hence simulate any TM if the tape provides the same action table
- This was all theoretical – he used the models to prove various theories. But he had inadvertently set the scene for what we now think of as a computer!

Note...

- ...A Turing machine made a shift from the analogue to the discrete domain (we are reading explicit symbols and not analogue voltages)
  - In part this is because Turing needed it to be able to represent things exactly, even infinite numbers (hence the infinite tape)

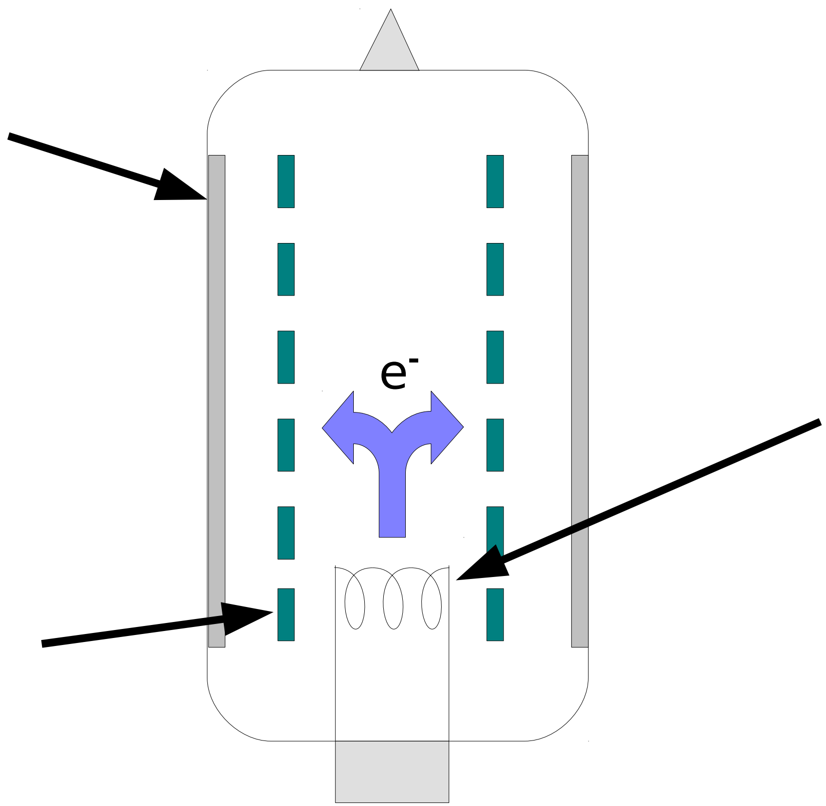
- This is useful practically too. Analogue devices:
  - have temperature-dependent behaviour
  - produce inexact answers due to component tolerances
  - are unreliable, big and power hungry

The Digital World

- When we have discrete states, the simplest hardware representation is a switch → digital world
- Going **digital** gives us:
  - Higher precision (same answer if you repeat)
  - Calculable accuracy (the answer is of known quality)
  - The possibility of using cheaper, lower-quality components since we just need to distinguish between two states (on/off)
- One problem: no switches?

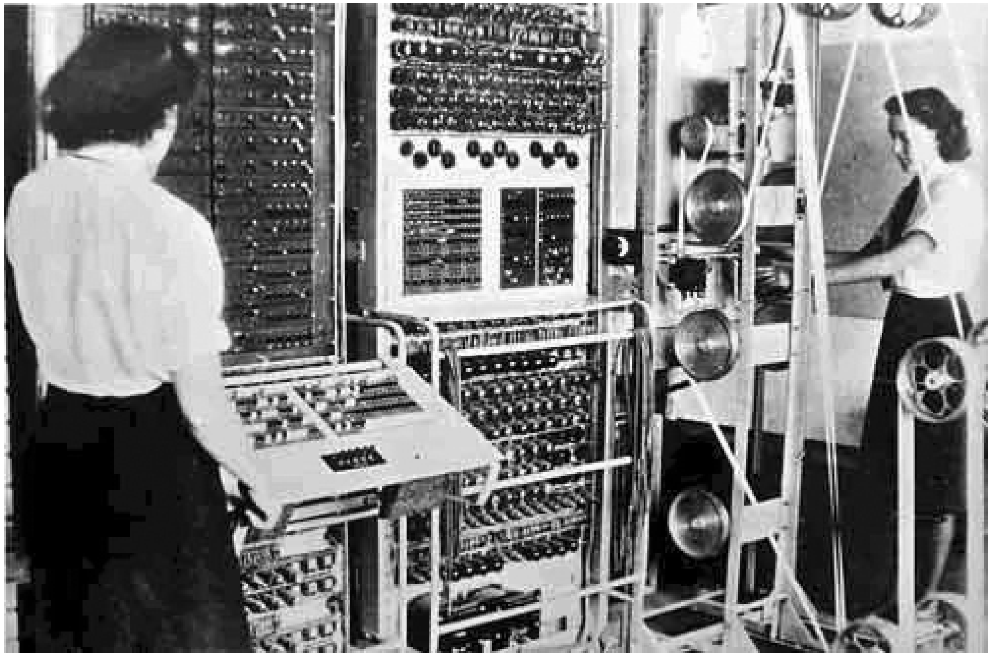
1946-58 Vacuum Tubes

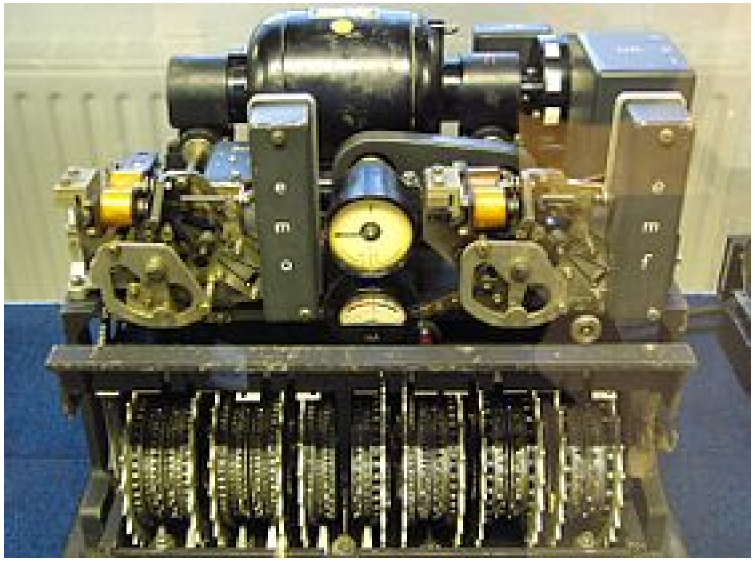
- Vacuum tubes are really just modified lightbulbs that can act as amplifiers or, crucially, switches.



- By the 1940s we had all we needed to develop a useful computer: vacuum tubes for switches; punch cards for input; theories of computation; and (sadly) war for innovation

Colussus

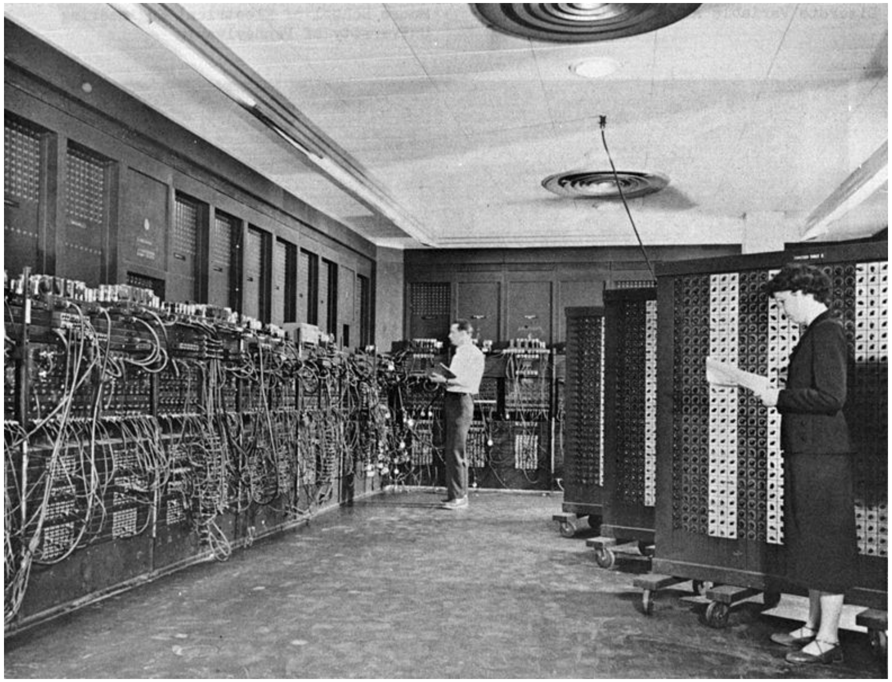




- 1944, Bletchley park
- Designed to break the German Lorenz SZ40/42 encryption machine
- Fed in encrypted messages via paper tape. Colussus then simulated the positions of the Lorenz wheels until it found a match with a high probability
- No internal program – programmed by setting switches and patching leads
- Highly specific use, not a general purpose computer
- Turing machine, but not universal

ENIAC

- Electronic Numerical Integrator and Computer
  - 1946, “Giant brain” to compute artillery tables for US military
  - First machine designed to be turing complete in the sense that it could be adapted to simulate other turing machines
  - But still programmed by setting switches manually…

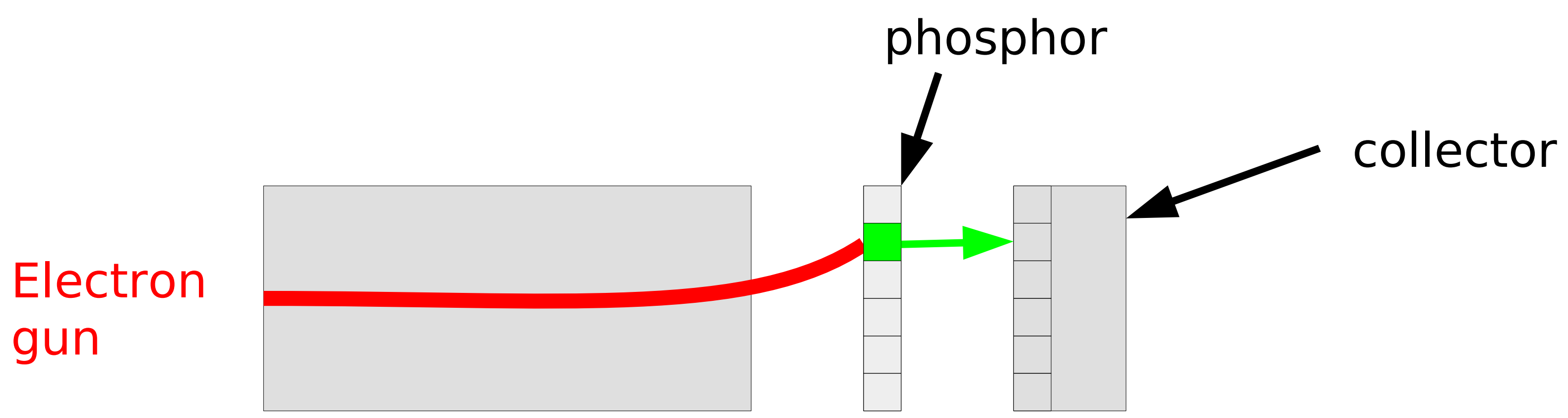


- Next step was to read in the “action table” (aka program) from tape as well as the data
- For this we needed more general purpose memory to store the program, input data and output

Manchester Baby

First Stored-Program Computer?

- 1948 a.k.a. mark I computer
- Cunning memory based on cathode ray tube. Used the electron gun to charge the phosphor on a screen, writing dots and dashes to the tiny screen

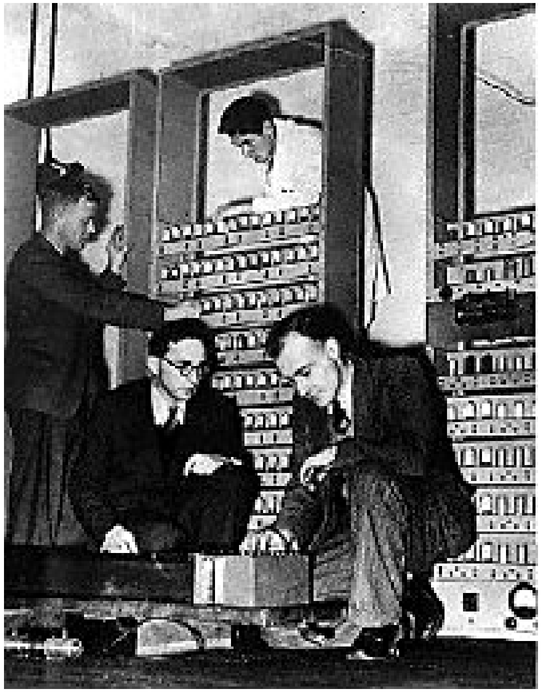


- A light-sensitive collector plate read the screen
- But the charge would leak away within 1s so they had to develop a cycle of read-refresh
- Gave a huge 2048 bits of memory!

EDSAC

- Electronic Delay Storage Automatic Calculator
- First practical stored-program computer, built **here** by Maurice Wilkes et al.

First Stored-Program Computer?



- Memory came in the form of a mercury delay line

- Used immediately for research here.
- Although they did have to invent programming....

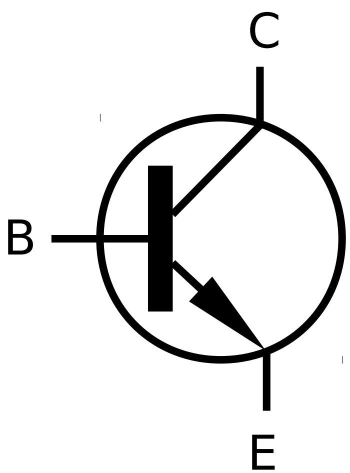
Storage: Stored-Program Machines

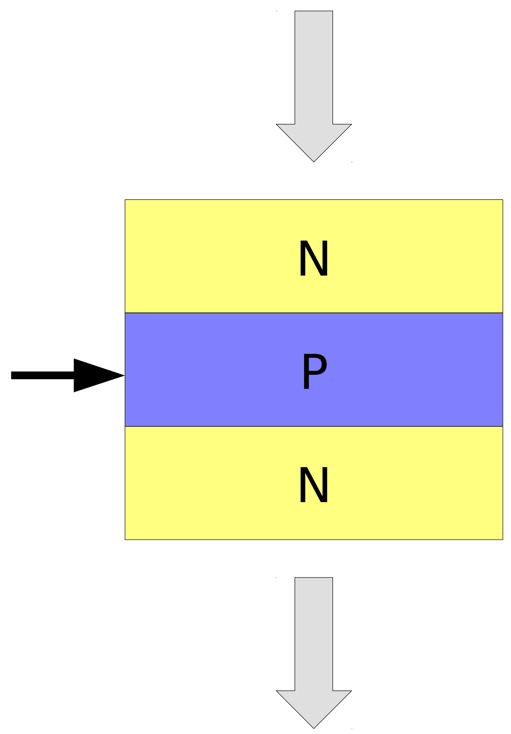
- So where do you store your programs and data?

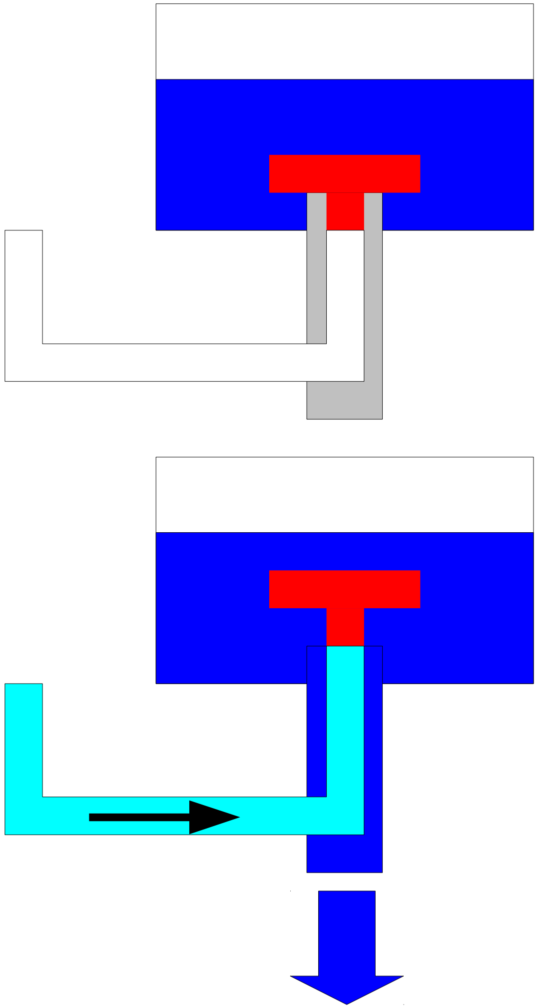
	Von-Neumann		Harvard
	Same memory for programs and data		**Separate** memories for programs and data
	+ Don't have to specify a partition so more efficient memory use		- Have to decide in advance how much to allocate to each
	+ Programs can modify themselves, giving great flexibility		+ Instruction memory can be declared read only to prevent viruses etc writing new instructions
	- Programs can modify themselves, leaving us open to malicious modification		
	- Can't get instructions and data simultaneously (therefore slower)		+ Can fetch instructions and data simultaneously

1959-64 Transistors

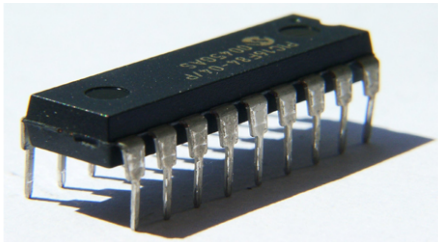
- Vacuum tubes bulky, hot and prone to failure
- Solution came from Bell labs (telecoms research)

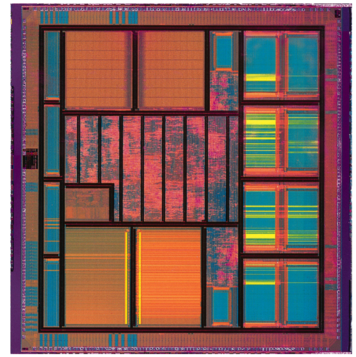






1965-70 Integrated Circuits





- Shift from separate transistors to a monolithic (formed from a single crystal) IC
- Essentially a miniature electronic circuit etched onto a sliver of semiconductor (usually silicon these days, but originally germanium)
- *Moore's law: the number of transistors that can be placed on an IC doubles approximately every two years*

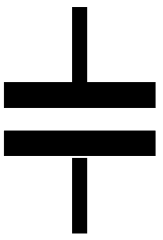
1971- Microprocessors

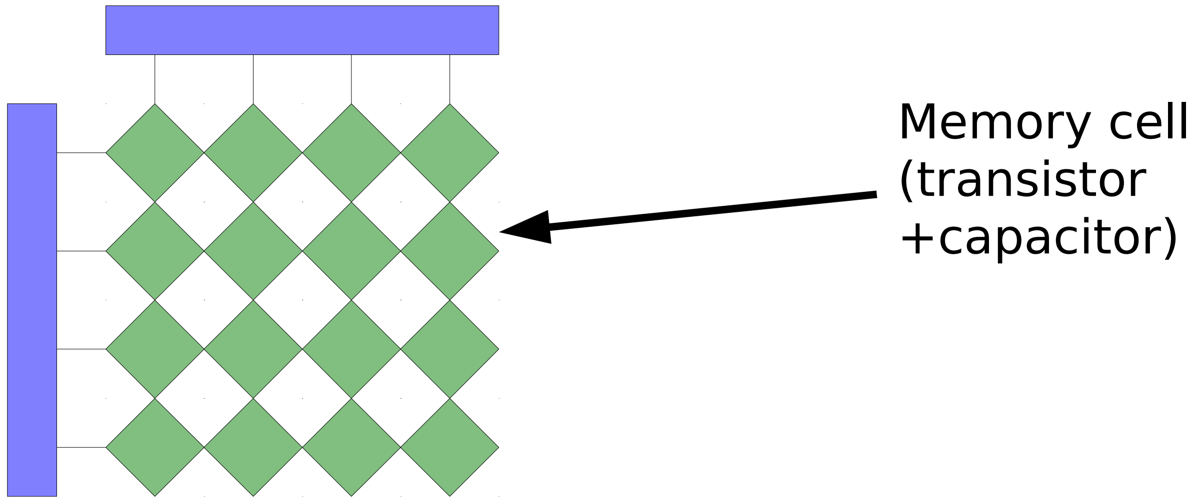
- a.k.a. a Central Processing Unit (CPU)
- A complete computer on an IC

Modern Systems

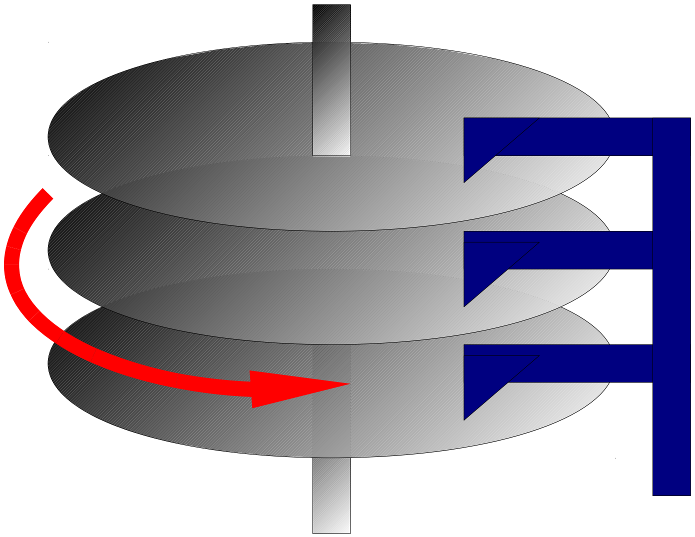
Main Memory (RAM)

- The alternative to mercury delay lines is essentially a capacitor. A charged capacitor is a “1” and a discharged capacitor is a ”0”
- Problem: capacitors leak charge over time, so a “1” becomes a “0”. Therefore we must refresh the capacitor regularly
- Cunningly we combine a transistor and a capacitor to store each bit and arrange them in a grid so we can just jump around in memory (Random Access Memory – RAM)





Hard Drives (Magnetic Media)

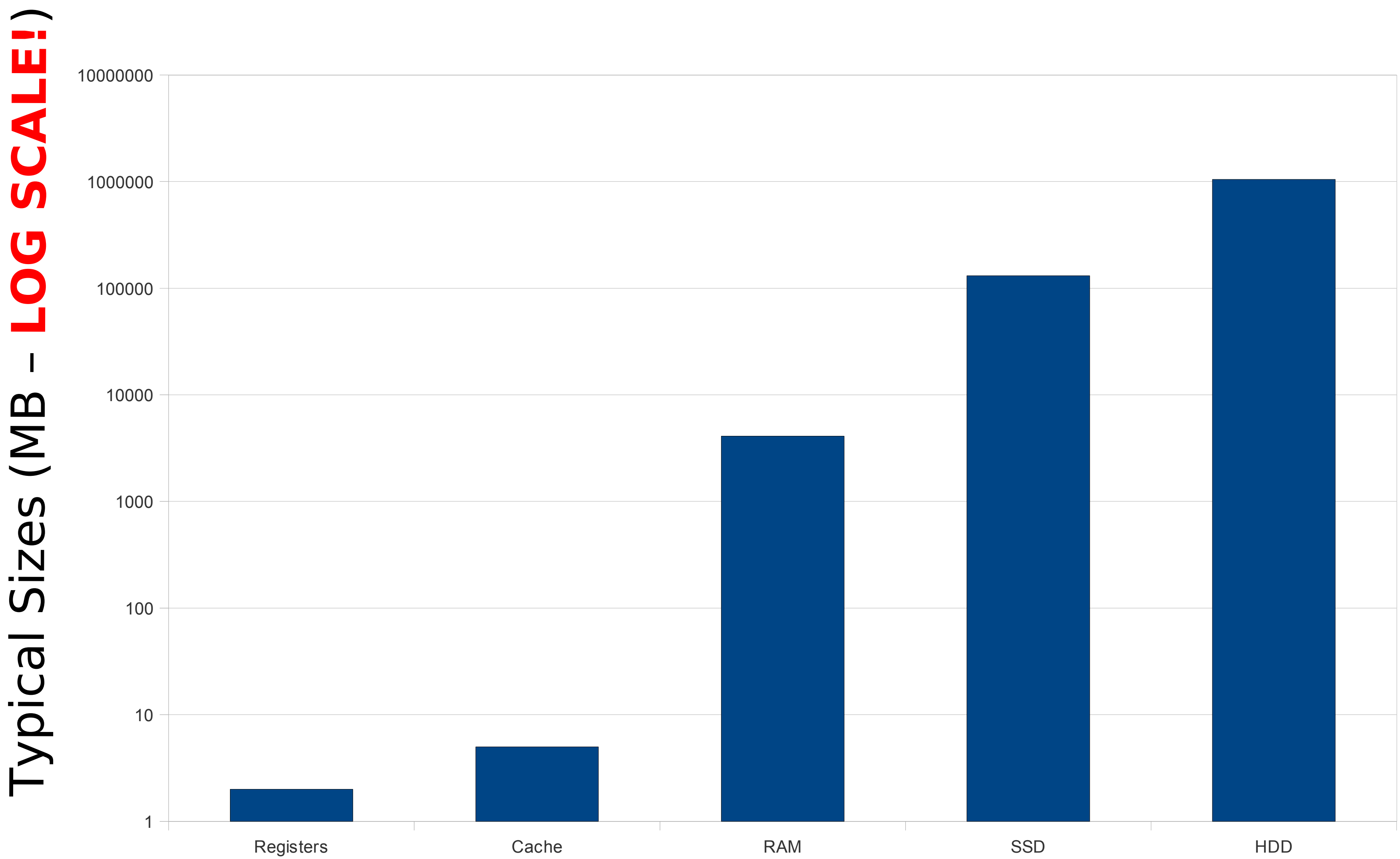


- Lots of tiny magnetic patches on a series of spinning discs
- Can easily magnetise or demagnetise a patch, allowing us to represent bits
- Similar to an old cassette tape only more advanced
- Read and write heads move above each disc, reading or writing data as it goes by
- Remarkable pieces of engineering that can store terabytes (TB, 1,000,000MB) or more.
- Cheap mass storage
- Non-volatile (the data's still there when you next turn it on)
- But much slower than RAM (because it takes time to seek to the bit of the disc we want)

Flash and SSDs

- Magnetic storage is great but moving parts mean many limitations, not least speed and size. RAM is volatile – turn off the power and it is lost
- Toshiba came up with Flash memory in the 1980s
  - Non-volatile memory that works essentially by trapping charge in a non-conducting layer between two transistors (much more complex than this, but out of scope here)
  - Slower than RAM and a limited number of writes, but still extremely useful
    - No moving parts
    - Used in USB flash drives, camera memory and now Solid State Discs

Modern Memories



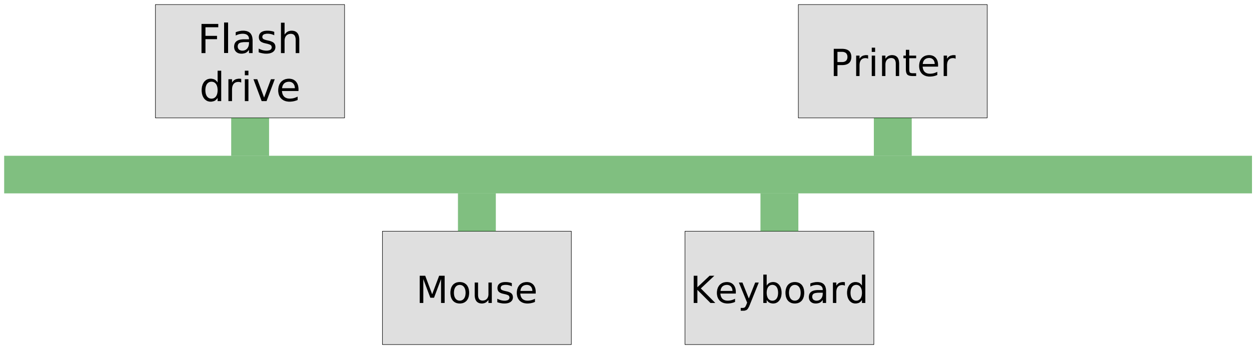
Graphics Cards

- Started life as Digital to Analogue Convertors (DACs) that took in a digital signal and spat out a voltage that could be used for a cathode ray screen
- Have become powerful computing devices of their own, transforming the input signal to provide fast, rich graphics.
  - Driven primarily by games and a need for 3D, graphics cards now contain Graphical Processing Units which you can think of as containing many (hundreds) of CPUs working in parallel.
  - Current trend is to exploit the powerful parallel processing capabilities of GPUs to do scientific simulations.

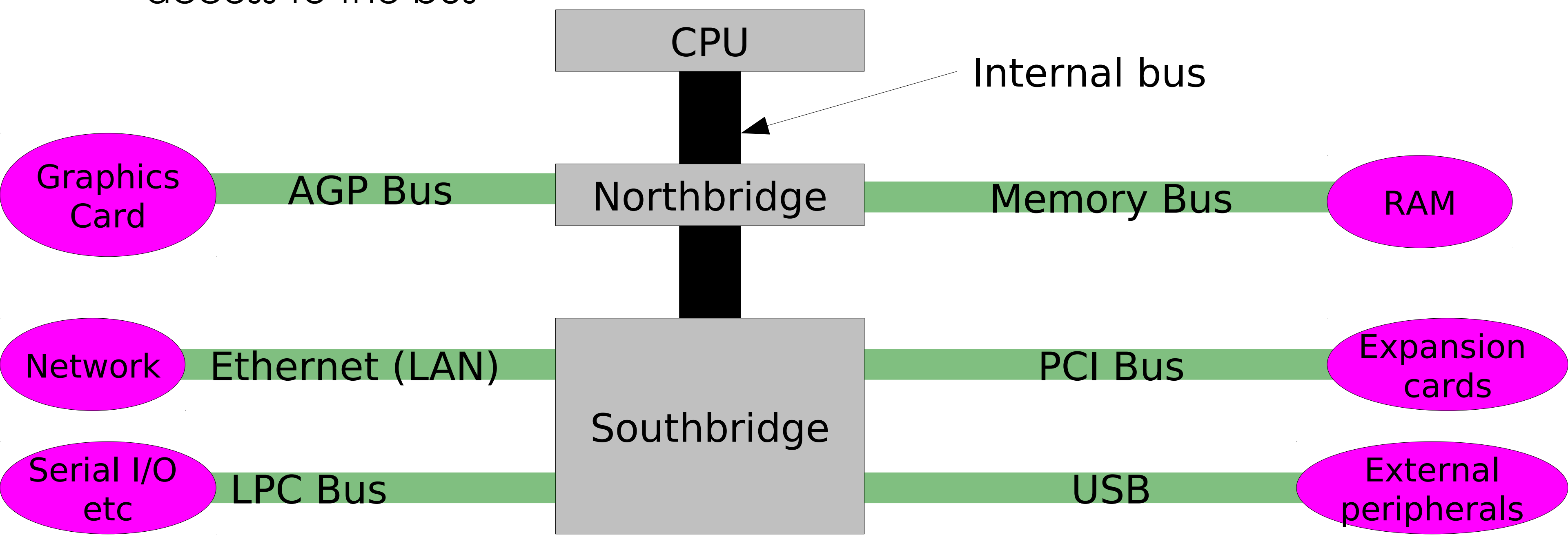
Peripherals

- Modern computers have a range of peripherals that they support:
  - Input (mouse, keyboard, etc)
  - Output (printer, display)
  - Network adapters
  - Graphics cards
- It's not particularly efficient to have dedicated cables/connects for each peripheral
  - How would we cope with future developments?
- Instead we have general purpose **buses** that provide communications pathways that can be shared amongst peripherals...

Buses

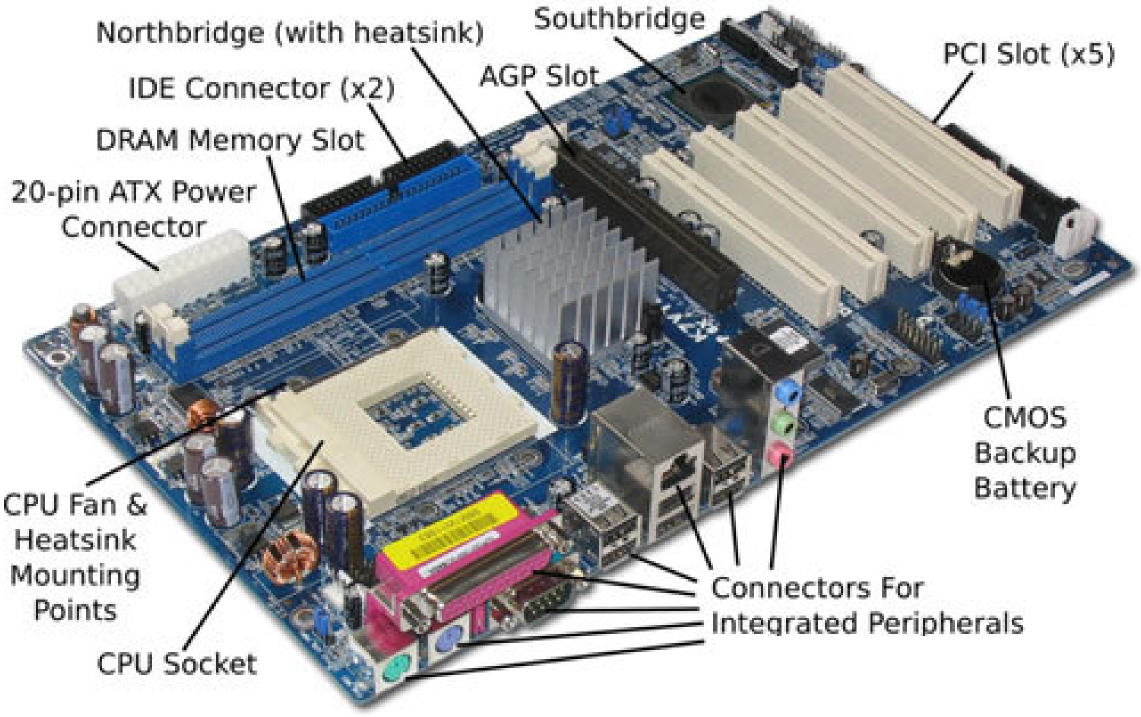


- Think of a bus as a data highway
- To prevent conflicts, buses have control lines (wires) that govern access to the bus



The Motherboard

- An evolution of the circuitry between the CPU and memory to include general purpose buses (and later to integrate some peripherals directly!)
- Internal Buses
  - ISA, PCI, PCIe, SATA, AGP
- External buses
  - USB,Firewire, eSATA, PC card

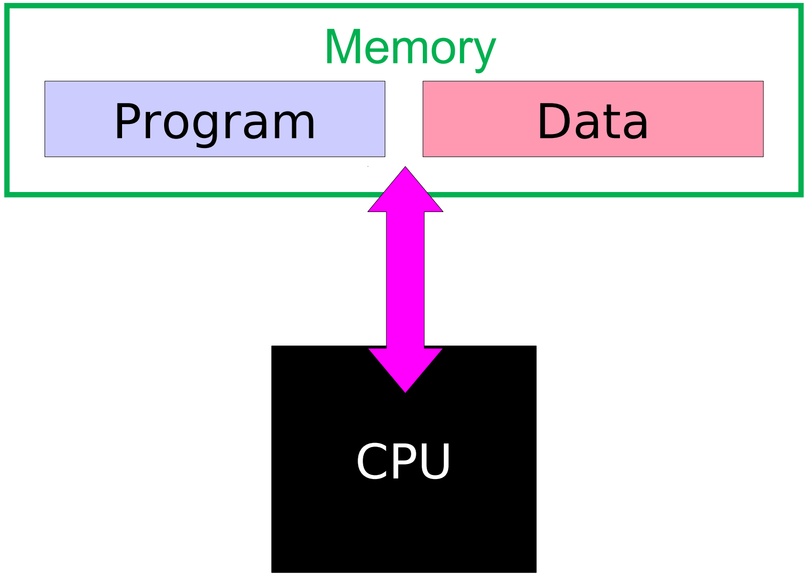


Computer Architectures

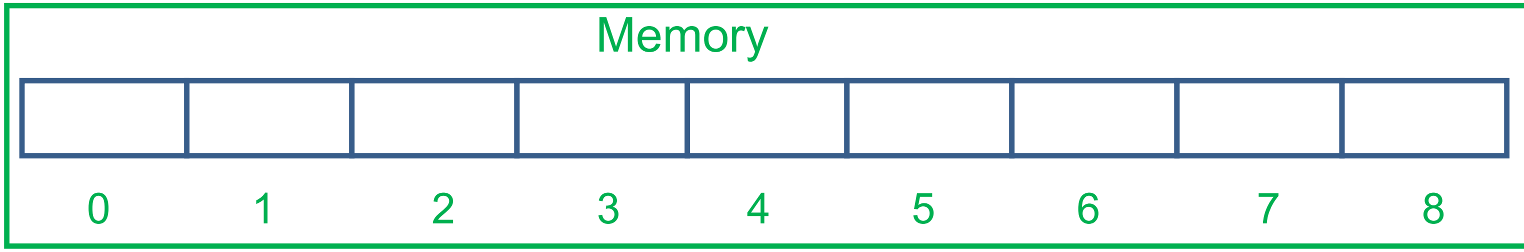
(What the CPU really does)

Programs, Instructions and Data

- Recall: Turing's universal machine reads in a table (=program) of instructions, which it then applies to a tape (=data) We will assume a Von-Neumann architecture since this is most common in CPUs today.

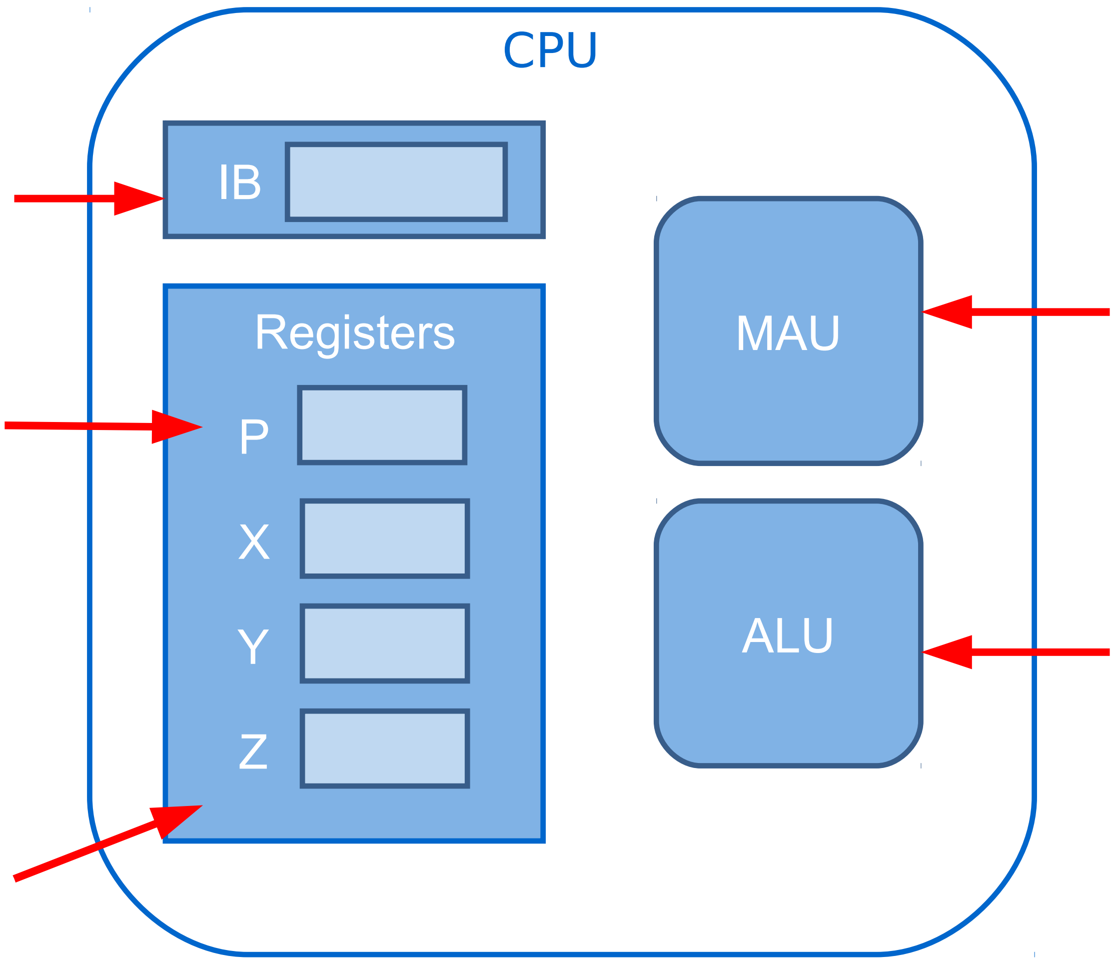


Simple Model of Memory

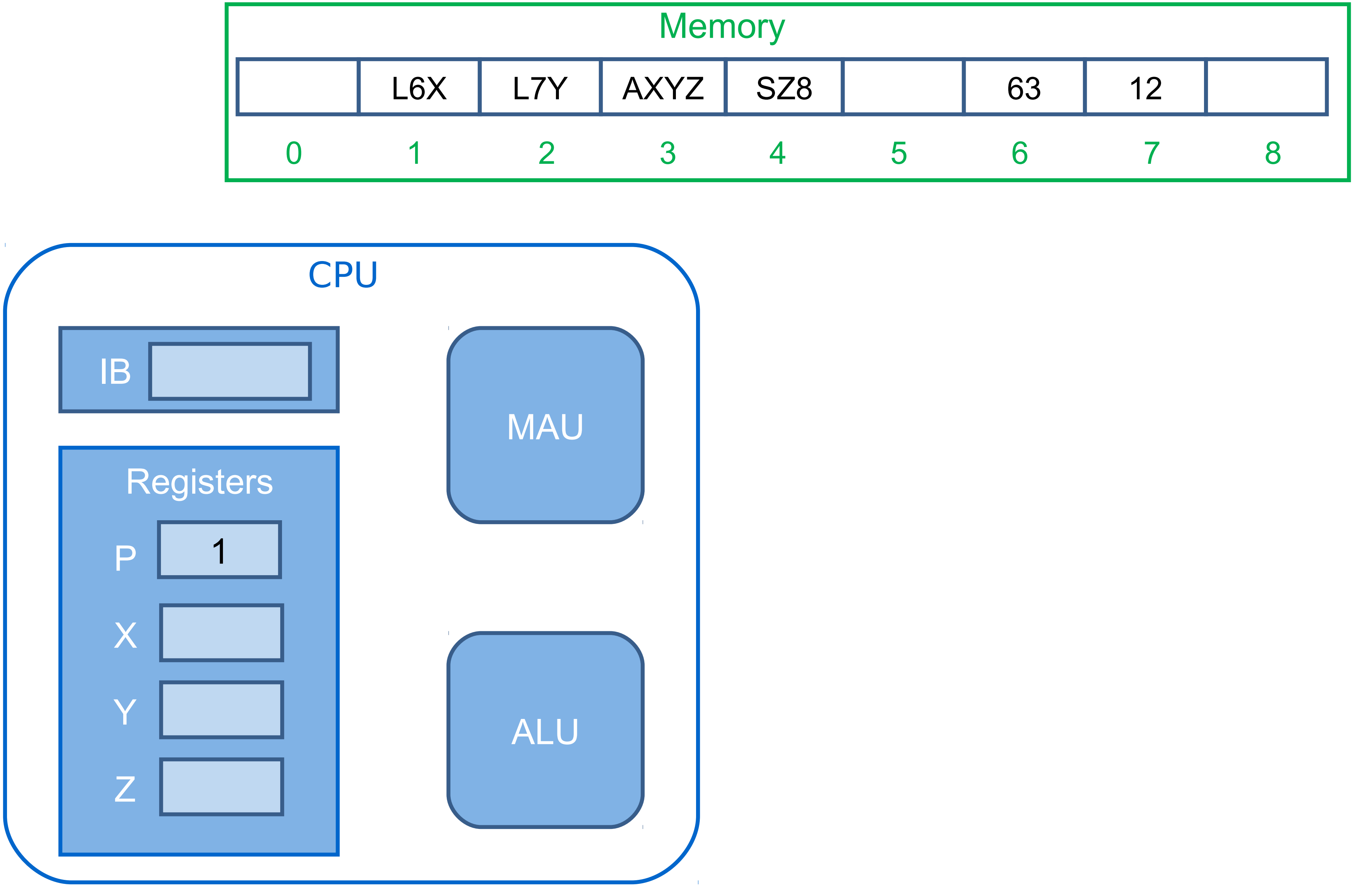


- We think of memory abstractly, as being split into discrete chunks, each given a unique *address*
- We can read or write in whole chunks
- Modern memory is <u>big</u>

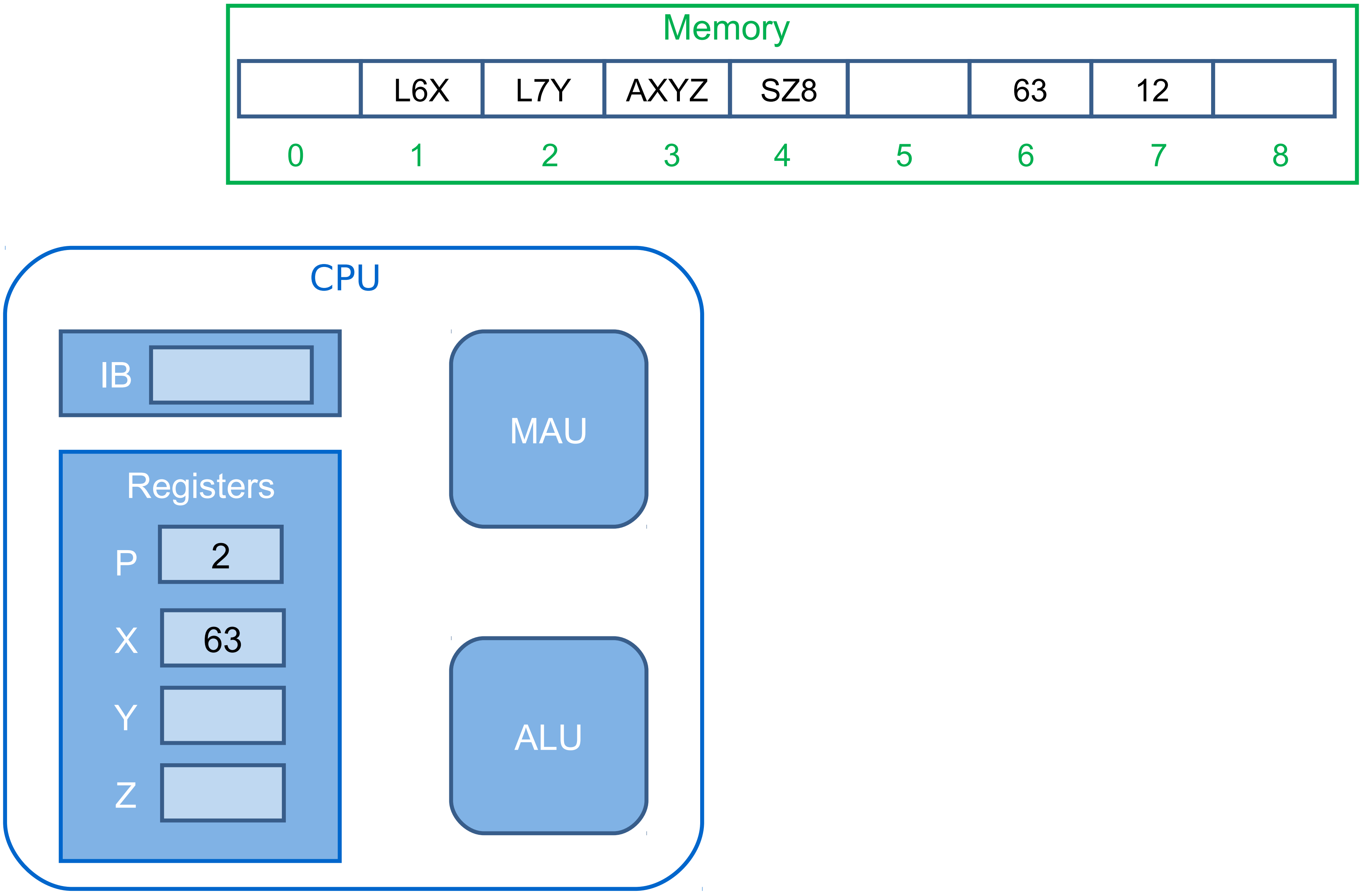
Simple Model of a CPU



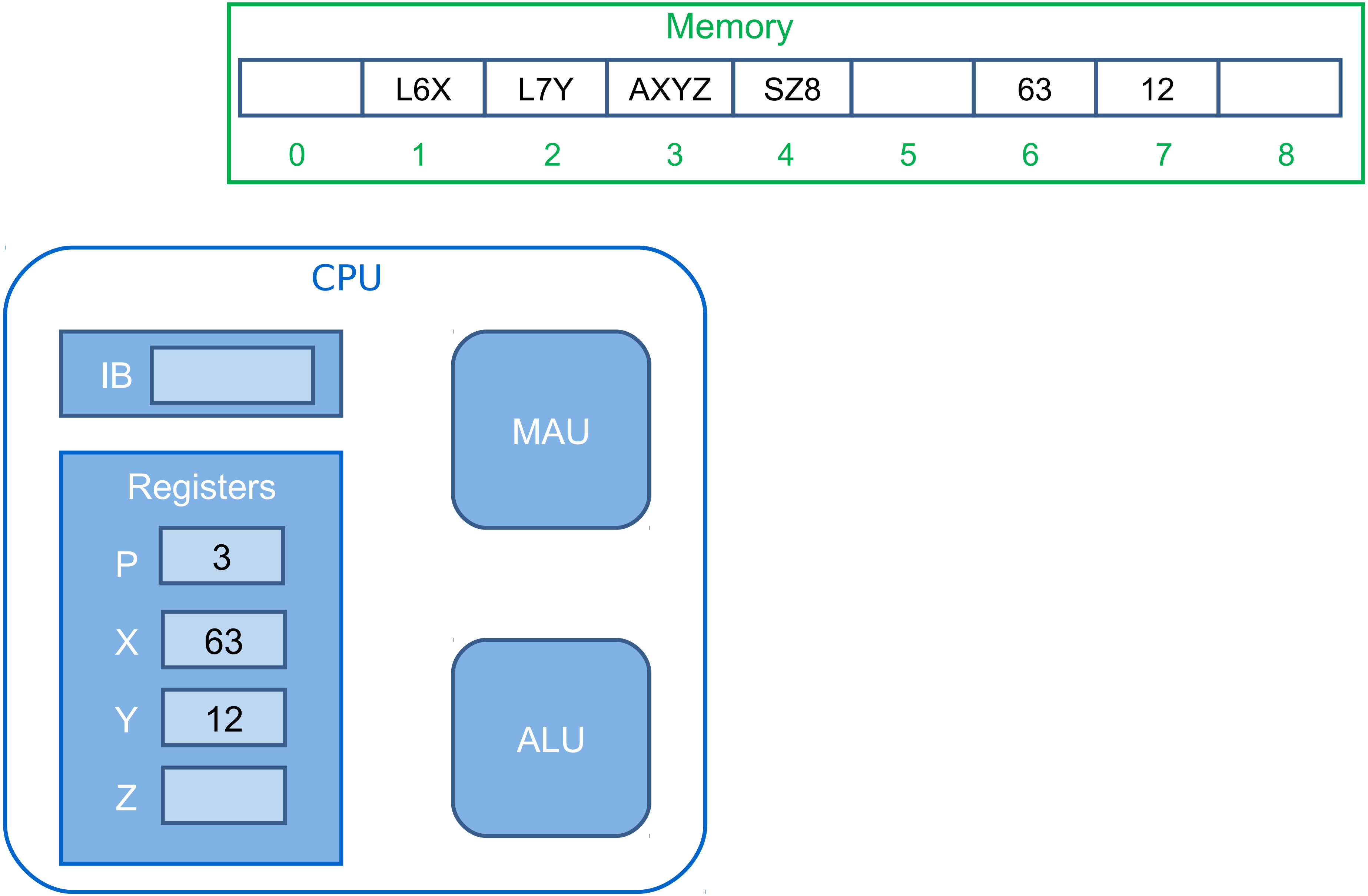
Fetch-Execute Cycle I



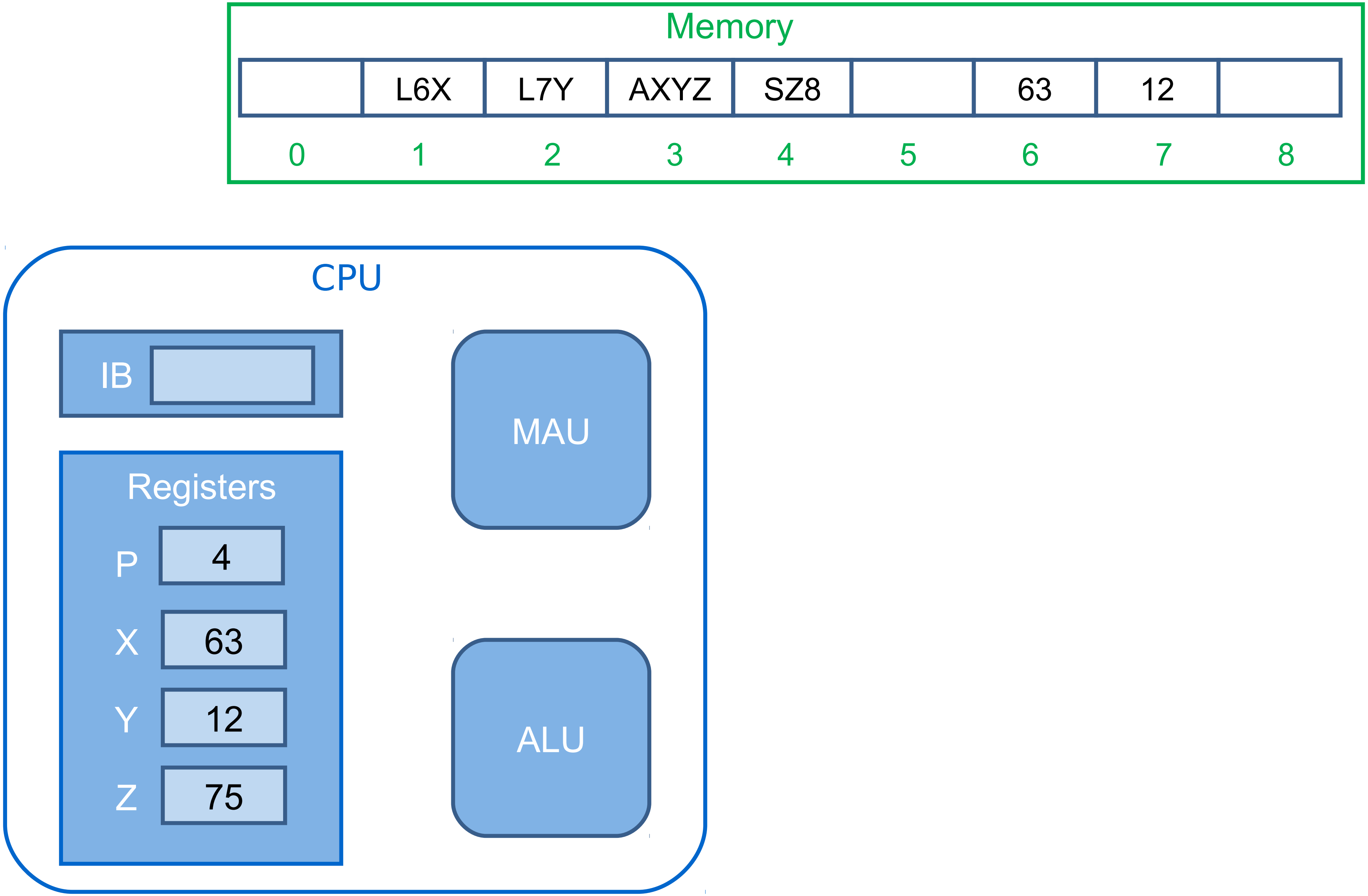
Fetch-Execute Cycle II



Fetch-Execute Cycle III



Fetch-Execute Cycle IV



CPU Processing Units

- Other common units
  - MAU – Memory Access Unit
  - ALU – Arithmetic Logic Unit
  - FPU – Floating Point Unit
  - BU – Branching Unit

CPU Architecture Sizes

- The registers are fixed sized, super-fast on-chip memory.
- When we build them we have to decide how big to make them
  - Bigger registers
    - Allow the CPU to do more per cycle
    - Mean we can have more memory
    - Too big and we might waste the electronics
  - Smaller registers
    - Less electronics (smaller, cooler CPUs)
    - Too small and it takes multiple cycles for simple operations

Aside: bits, bytes, words

- A bit is either 0 or 1, represented by a 'b'
- A byte is (usually) eight bits, represented by a 'B'
- A word is the natural unit of data for a given architecture (i.e. register size)

- Larger collections
  - SI units are based on powers of ten, but computers use powers of two, which causes confusion
  - 1 kilobyte (kB) might be 1,000B or 1024B (nearest power of two)
  - Technically, there is now 1 kibibyte = 1 kiB = 1024B etc but no-one really uses these..!

Instruction Sets

- At first, every CPU that came out had its own, special instruction set. This meant that any program for machine X would be useless on machine Y
- We needed a standardised set of instructions
  - Intel drew up the 8086 specification in 1978
  - Manufacturers produced CPUs that understood 8086 instructions and the so-called PC was born
- Modern PCs **still** support this instruction set, albeit manufacturers add their own special commands to take advantage of special features (MMX, etc).
- Each set of instructions is referred to as an **architecture**

Representing Data

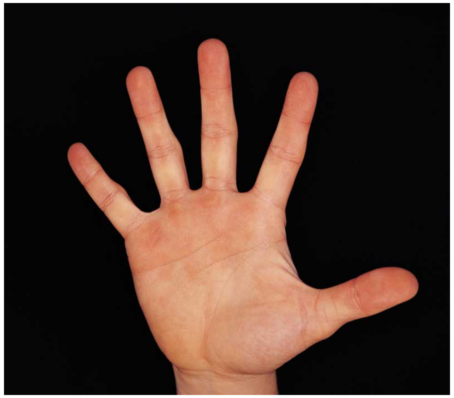
(How a computer sees the world)

Decimal

- We're all happy with decimal ("base-10") numbers.

$$512_{10} = 5\text{x}10^2+1\text{x}10^1+2\text{x}10^0$$

- To represent a number we use a series of digits, each of which can adopt one of ten states, 0-9.



Binary

- As we know, most computers store data using a huge bank of switches; the natural counting unit is hence **two** (on or off)
- Therefore we use base-2 numbers, formed from binary digits ("bits" – 0s or 1s)

$$1101_2 = 1x2^3+1x2^2+0x2^1+1x2^0$$

$$= 13_{10}$$

Binary Integers

- n *decimal digits* can label 10^n things
- n *bits* allow us to label 2^n things
- This allows us to represent the number range $0, 1, \ldots, 2^n-1$

Note!

- *Note that we often count from zero and not one. Out-by-one errors are very common programming mistakes when the programmer counts from one*

Hexadecimal

- Decimal is nice for humans because you can represent big numbers using relatively few digits
  - E.g. "123456" vs "11110001001000000"

- Programmers usually consider small groups of bits:
  - E.g. 0001-1110-0010-0100-0000
  - There are 4 bits per group, so 16 possibilities. We label them using decimal digits until we run out:

0 1 2 3 4 5 6 7 8 9 A B C D E F

  - Making the example 1-E-2-4-0 or 0x1E240
  - This is just base-16 numbering or hexadecimal

Octal

- Another (less common) alternative is octal, which is groupings of 3 bits, or base-8
  - 000-011-110-001-001-000-000 becomes 0-3-6-1-1-0-0

- Note that there isn't a convenient grouping for our beloved base-10 decimal. Three bits isn't enough and four bits is too much.
  - We can see this using the log function. We want to solve 2^b-1=9, where b is the number of bits:
    - 2^b=10
    - b=$\log_2$10=3.322 bits
    - So 3.3 bits map to a decimal digit – yuk! It's much easier if the bases are a power-of-two.

Binary Addition and Subtraction

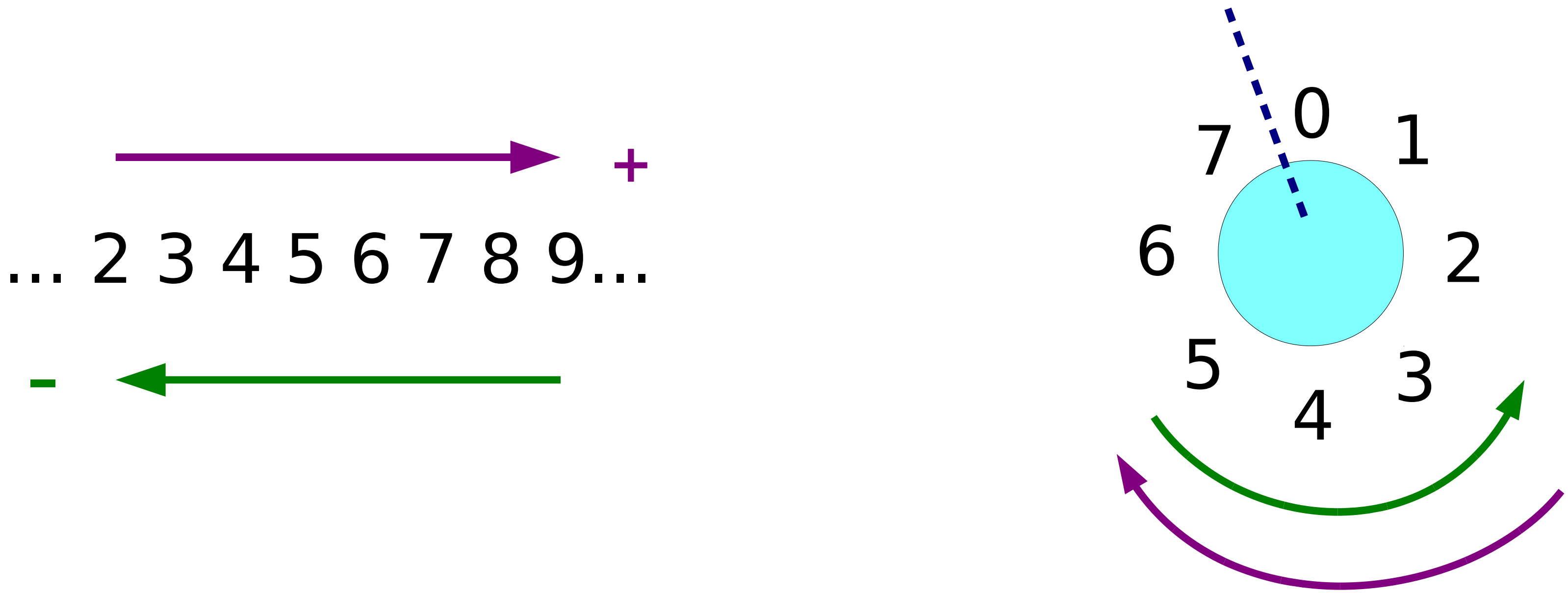
- Really easy. A simplified version of what you do in decimal, with carries and borrows etc

```
  0101          0101
+ 0011        - 0011
```

- **Except**: this all has to be done from the registers, which have a set size. What happens if the number gets too big (overflow) or too small (underflow)?

```
  1111          0000
+ 0001        - 0001
```

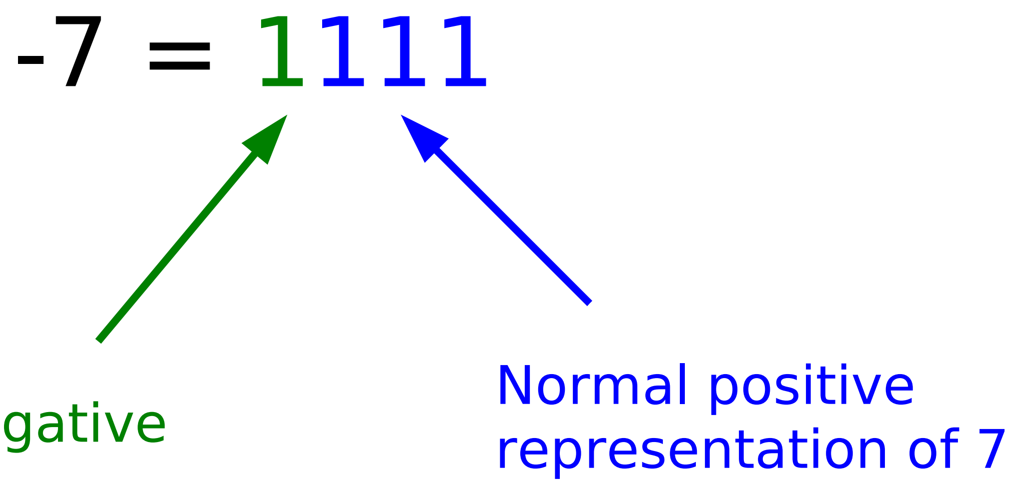
Modulo Arithmetic



- Overflow takes us across the dotted boundary
  - So 3+6=1 (overflow)
  - We say this is (3+6) mod 8

Negative Numbers

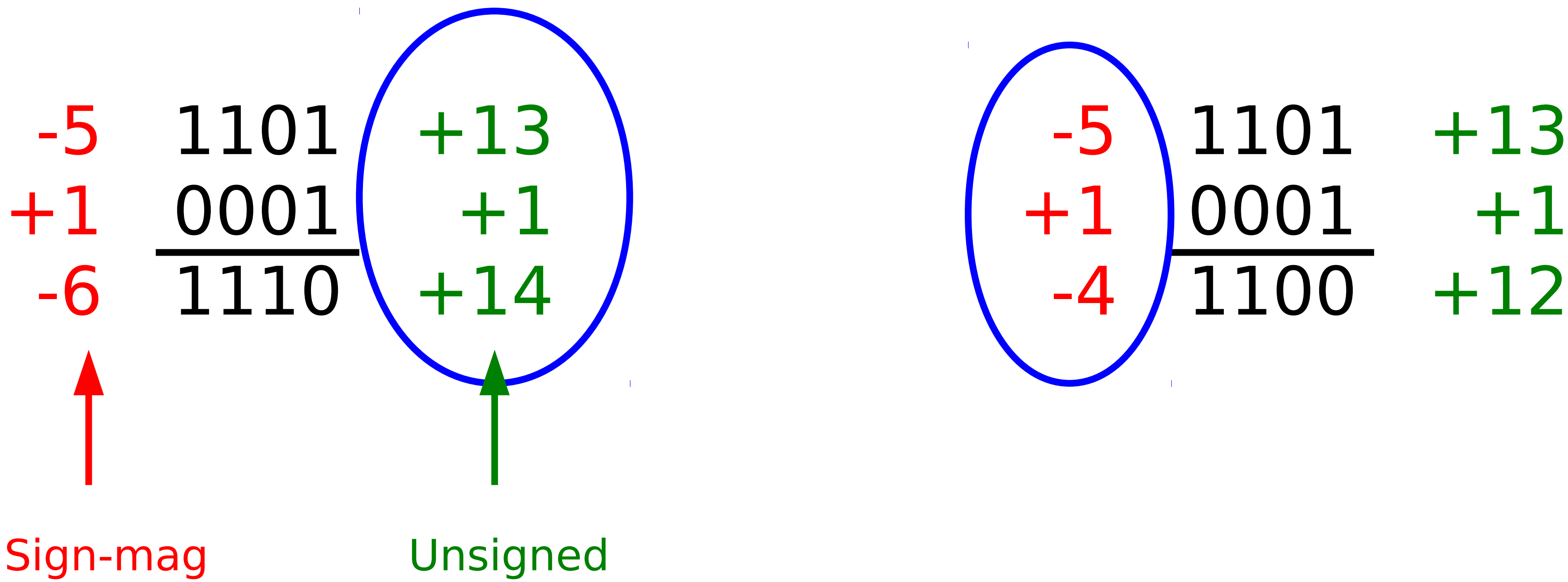
- All of this skipped over the need to represent negatives.
- The naïve choice is to use the MSB to indicate +/-
  - A 1 in the MSB is negative
  - A 0 in the MSB is positive



- This is the **sign-magnitude** technique

Difficulties with Sign-Magnitude

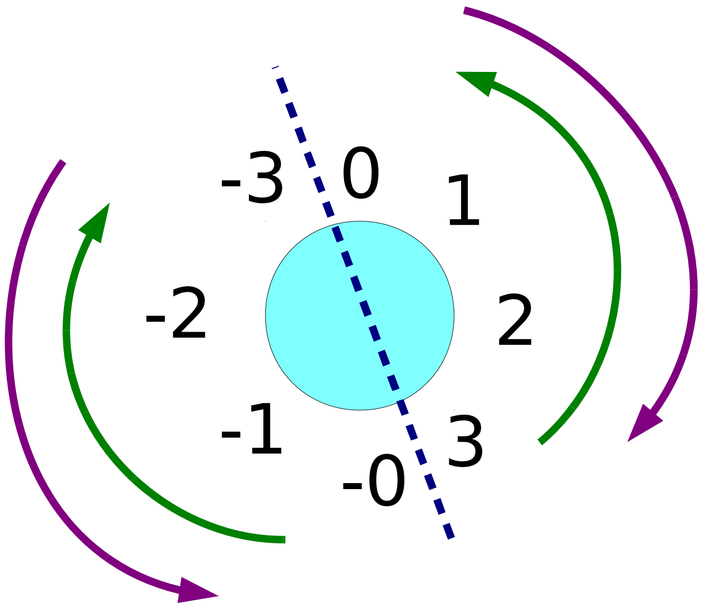
- Has a representation of minus zero (1000_2=-0) so wastes one of our 2^n labels
- Addition/subtraction circuitry is not pretty



“normal” addition

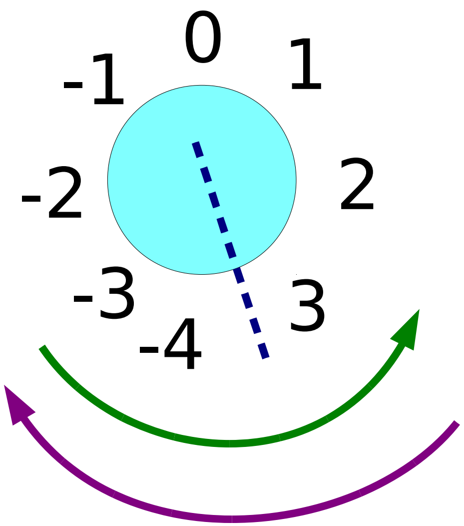
“sign-mag” addition

Alternatively...



- Gives us two discontinuities and a reversal of direction using normal addition circuitry!!

Two's complement



- How about this?
- One discontinuity again
- Efficient (no minus zero!)
- Crucially we can use normal addition/subtraction circuits!!
- "Two's complement"

- Positive to negative: Invert all the bits and add 1

  0101 (+5) → 1010 → 1011 (-5)

- Negative to positive: Same procedure!!

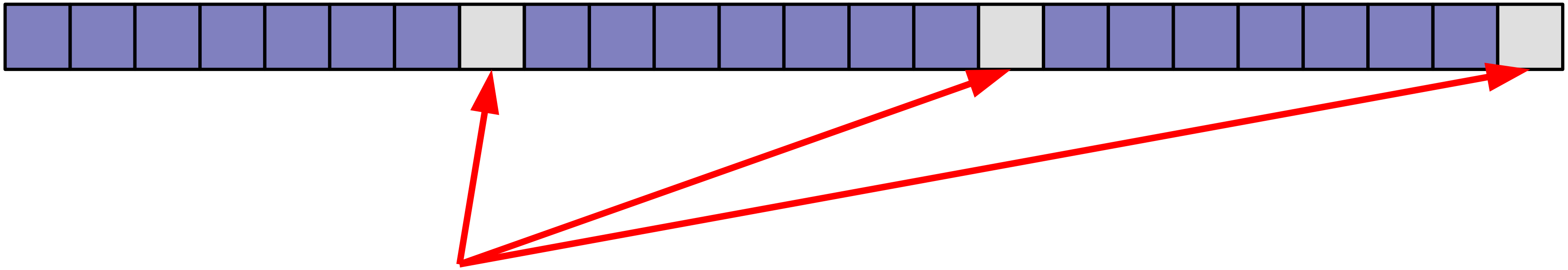
  1011 (-5) → 0100 → 0101 (+5)

Fractional Numbers

- Scientific apps rarely survive on integers alone, but representing fractional parts efficiently is complicated.
- Option one: **fixed point**
  - Set the point at a known location. Anything to the left represents the integer part; anything to the right the fractional part
  - But where do we set it??
- Option two: **floating point**
  - Let the point 'float' to give more capacity on its left or right as needed
  - Much more efficient, but harder to work with
  - Very important: dedicated course on it later this year.

Character Arrays

- To represent text, we simply have an encoding from an integer to a letter or character
- The classic choice is **ASCII**
  - Takes one byte per character but actually only uses 7 bits of it so can represent 2^7=128 characters



Other encodings

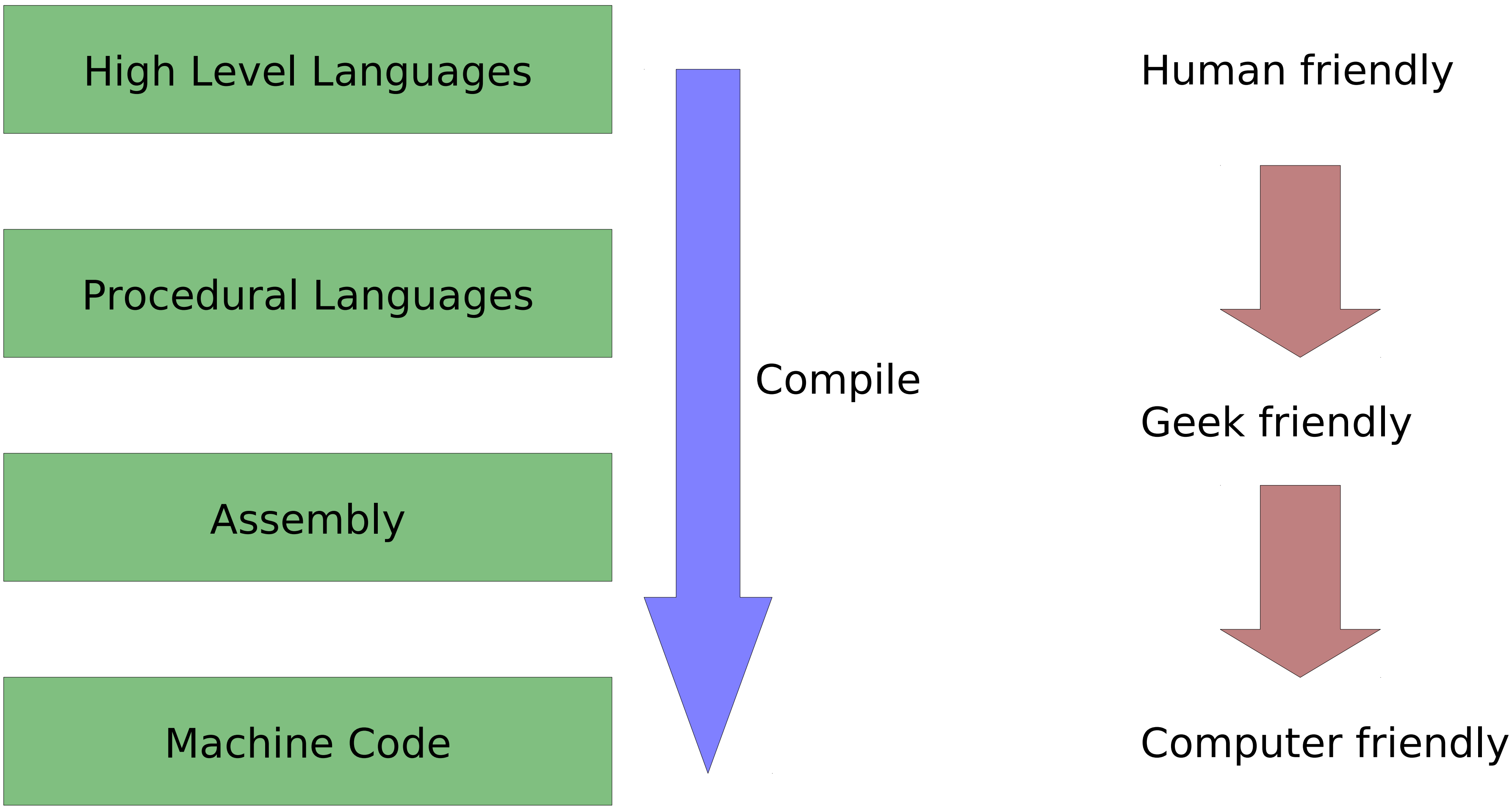
- 128 letters is fine for English alphabet
  - Turns out there are other alphabets (who knew?!)
  - So we have unicode and other representations that typically take two bytes to represent each character

  - *Remember this when we come to look at Java next term, which uses 2-byte unicode as standard...*

Levels of Abstraction

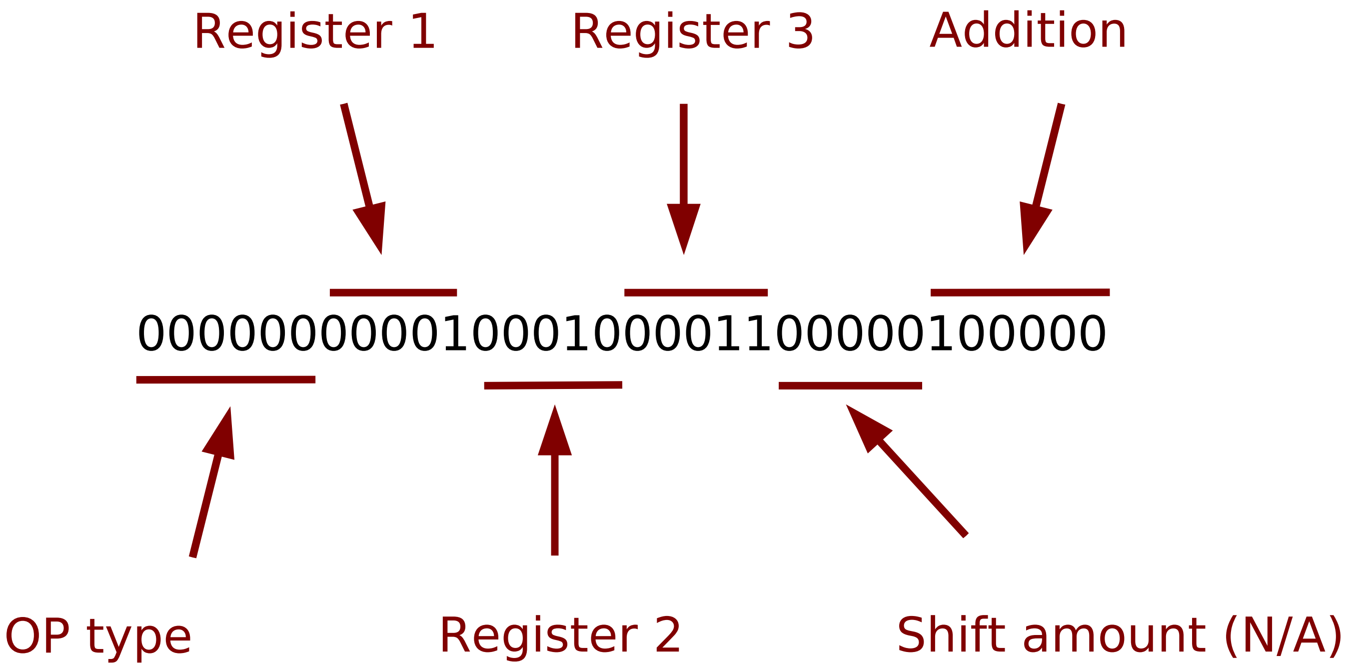
(How humans can program computers)

Levels of Abstraction for Programming



Machine Code

- What the CPU 'understands': a series of instructions that it processes using the the fetch-execute technique
- E.g. to add registers 1 and 2, putting the result in register 3 using the MIPS architecture:



Assembly

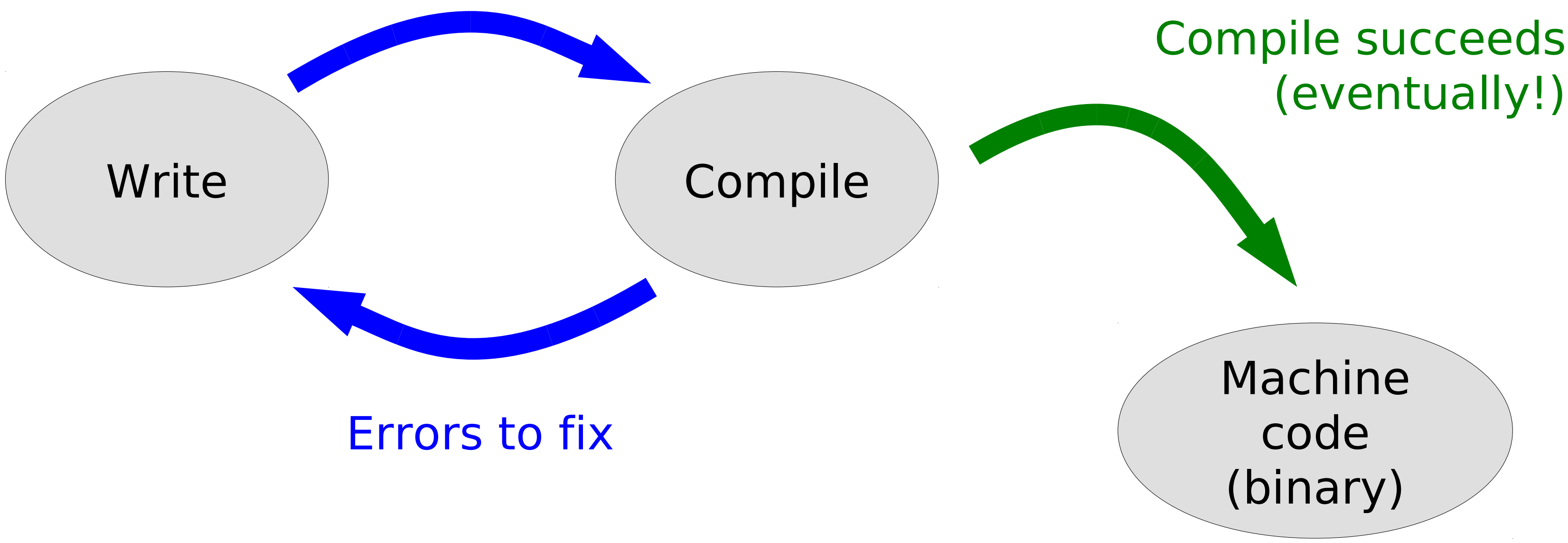
- Essentially machine code, except we replace binary sequences with text that is easier for humans
- E.g. add registers 1 and 2, storing in 3:

```
add $s3, $s1, $s2
```

- Produces small, efficient machine code when **assembled**
- Almost as tedious to write as machine code
- Becoming a specialised skill...
- Ends up being architecture-specific if you want the most efficient results :-(

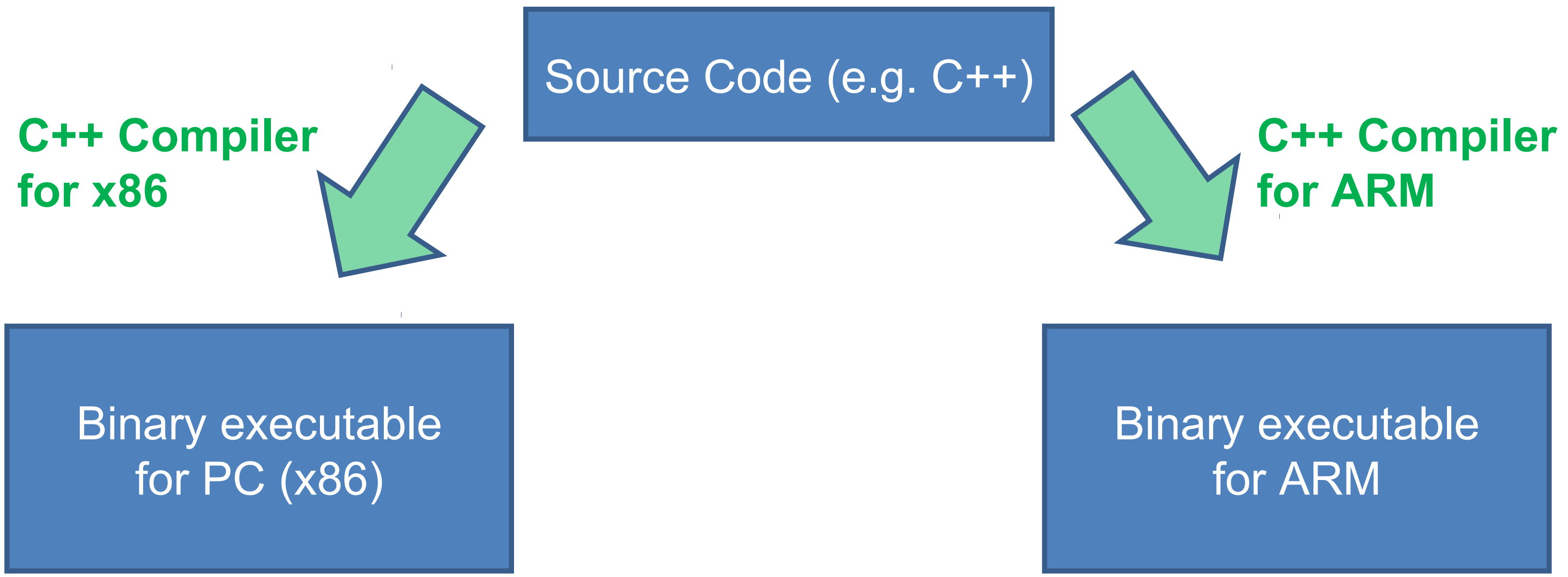
Compilers

- A compiler effectively acts as a translator, from source to machine code (or some intermediary)
- Writing one is tricky and we require strict rules on the input (i.e. on the programming language). Unlike English, ambiguities cannot be tolerated!



Avoiding Architecture Lock-In

- Different CPUs have different instruction sets
- We write high level code
- We compile the code to a specific architecture (i.e. machine code for that processor)



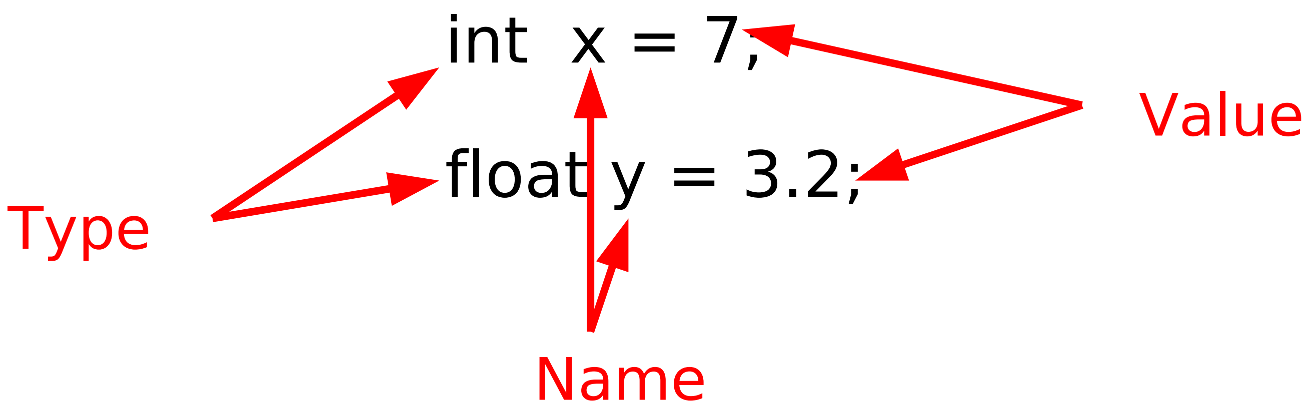
Interpreters

- The end result is a compiled program that can be run on **one** CPU architecture.
- As computers got faster, it became apparent that we could potentially compile 'on-the-fly'. i.e. translate high-level code to machine code as we go
- Call programs that do this ***interpreters***

Architecture agnostic – distribute the code and have a dedicated interpreter on each machine	Have to distribute the code
Easier development loop	Errors only appear at runtime
	Performance hit – always compiling

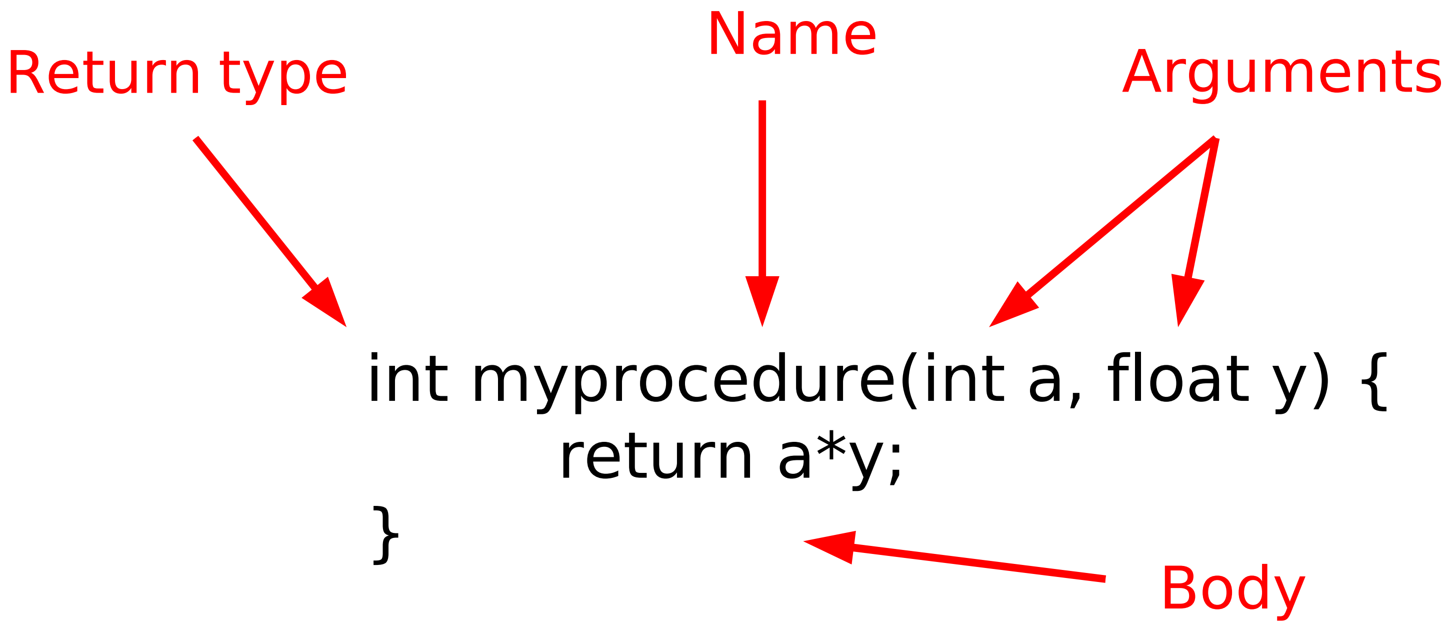
Procedural Languages

- The next logical step up from assembly is a procedural language, which relies on **procedures** (aka methods, subroutines, functions*) and provides an architecture-agnostic specification that is closer to natural language
- Represent state by **declaring variables.** E.g.



* see OOP course in Lent for more careful definitions of these

Procedures



- In procedural programming you call a series of procedures in a specific order to alter the state

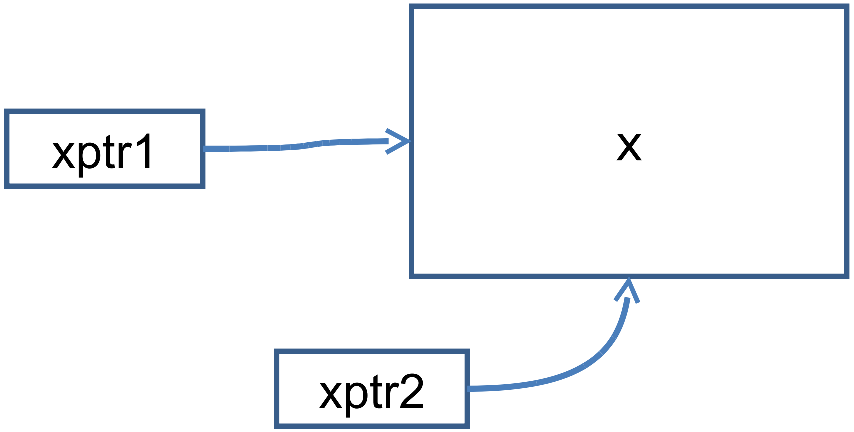
Memory and Pointers

- In reality the compiler stores a mapping from variable name to a specific memory address, along with the type so it knows how to interpret the memory (e.g. "*x is an int so it spans 4 bytes starting at memory address 43526*").
- Lower level languages often let us work with memory addresses directly. Variables that store memory addresses are called **pointers** or sometimes **references**
- Manipulating memory directly allows us to write fast, efficient code, but also exposes us to bigger risks
  - Get it wrong and the program 'crashes' .

Pointers: Box and Arrow Model

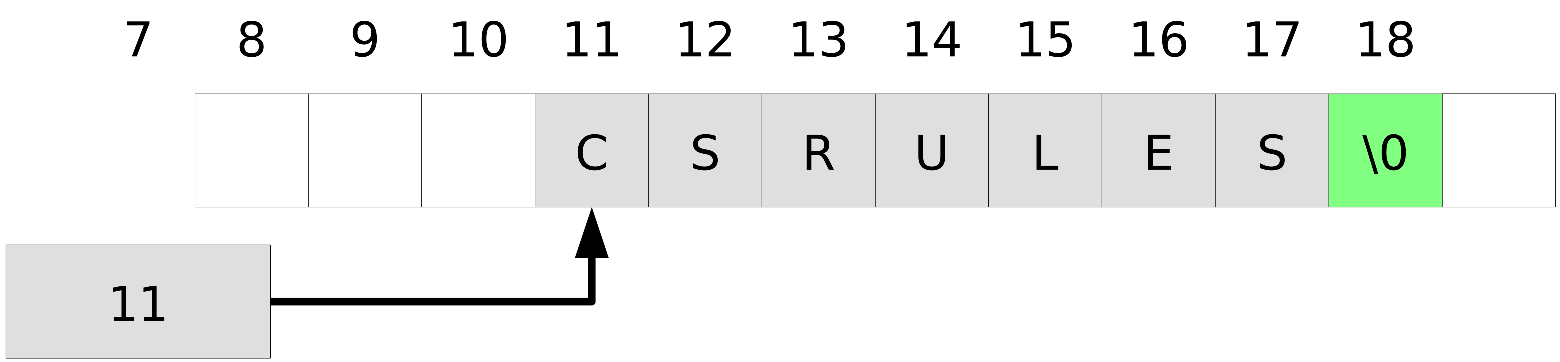
- A pointer is just the memory address of the first memory slot used by the variable
- The pointer type tells the compiler how many slots the whole object uses

```
int x = 72;
int *xptr1 = &x;
int *xptr2 = xptr1;
```



Example: Representing Strings I

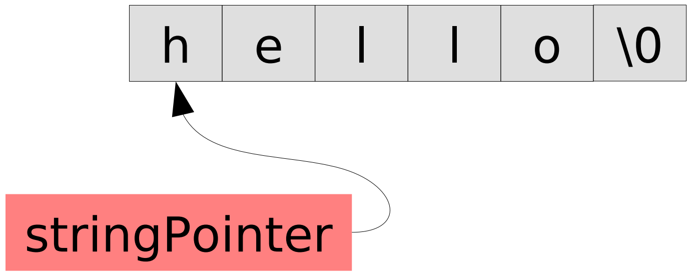
- A single character is fine, but a text string is of variable length – how can we cope with that?
- We simply store the start of the string in memory and require it to finish with a special character (the NULL or terminating character, aka '\0')
- So now we need to be able to store memory addresses → use pointers



- We think of there being an array of characters (single letters) in memory, with the string pointer pointing to the first element of that array

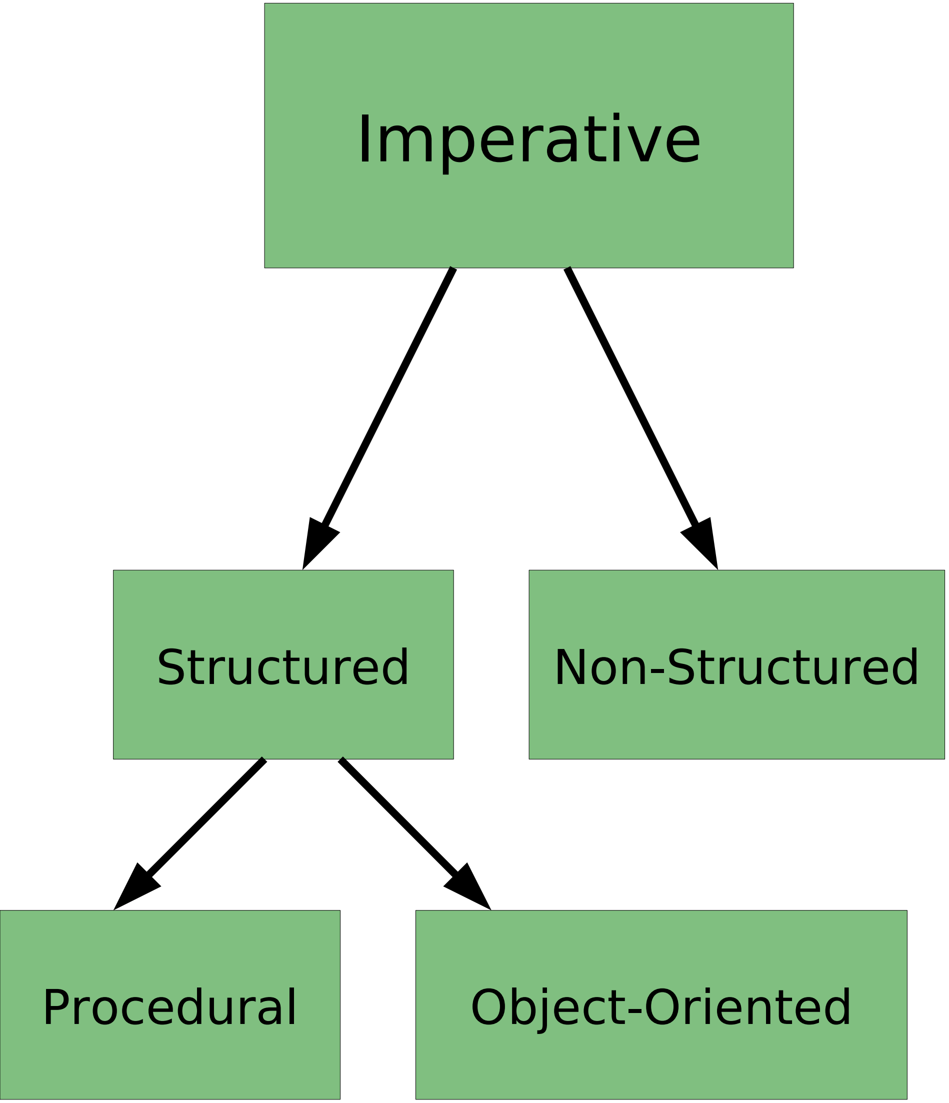
Example: Representing Strings II

```
char letterArray[] = {'h','e','l','l','o','\0'};

char *stringPointer = &(letterArray[0]);

printf("%s\n",stringPointer);

letterArray[3]='\0';

printf("%s\n",stringPointer);
```



Imperative and Functional Programming

Imperative Programming



- Procedural languages belong to a larger class of **imperative** languages
- This class of language describes a program in terms of **state** (variables etc)
- Each instruction manipulates explicit state
- E.g. Java, C, C++, python, Basic, etc.
- This is probably what you're familiar with, if you've done any programming before

Imperative Example

```
float delivery = 1.50;
float vatrate = 1.20;

float getFullPrice(float price) {
        return (price + delivery)*vatrate;
}

float labelprice =7.50;
Float salesprice = getFullPrice(labelprice)
```

- How would we represent this algebraically?
- Problem: the getFullPrice() function depends on state outside the arguments (i.e. delivery, vatrate)
- This is like having a function f(x) that can give different values for the same input!!

Imperative Example

- Could instead have made a 'proper' function:

```
float getFullPrice(float price,float delivery, float vatrate) {
        return (price + delivery)*vatrate;
}

float delivery = 1.50;
float vatrate = 1.20;
float labelprice =7.50;
float salesprice = getFullPrice(labelprice, delivery, vatrate)
```

- Now we have a function that always returns the same answer for a set of inputs
- Maps to the maths directly

Functional Programming

- This is an extreme of what we just did, forcing you not to use state but to use lots of well-defined proper functions
  - You can *never* change the value of any piece of state
  - Functions can only depend on their arguments
  - There are no for loops, while loops – basically nothing that isn't done in the algebra you know so well
- This type of programming was a natural way to go in the Turing era, when actual computers didn't exist

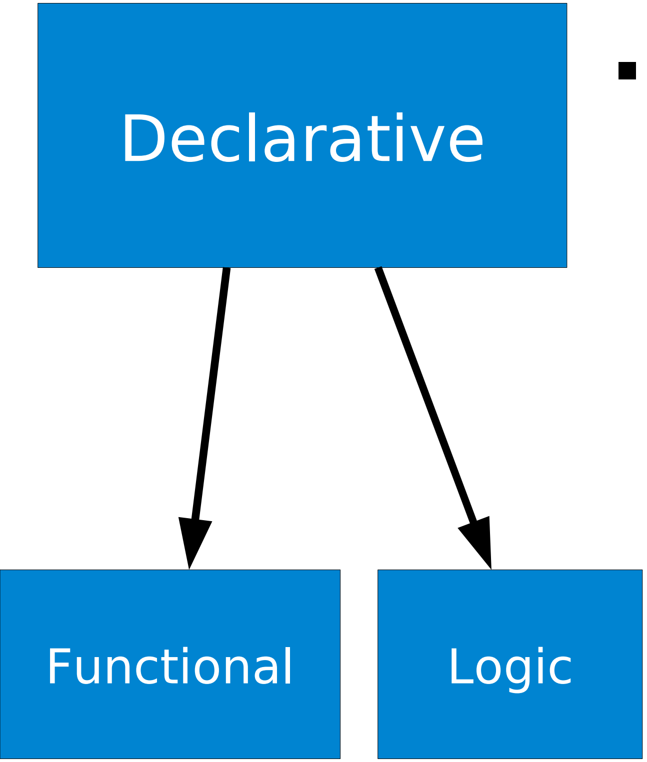
Example: ML

- In a week or so you will be introduced to **ML**, a functional language. We start with this because:
  - It is closer to maths in form
  - almost no-one in the room knows it
  - It allows you to focus quickly on the interesting parts of computer science rather than first learning the minutiae of a programming language

- E.g. computing a^b

Maths	ML
p(a,b) = a x p(a,b-1)	fun p(a,b) = a*p(a,b-1)

Declarative Programming



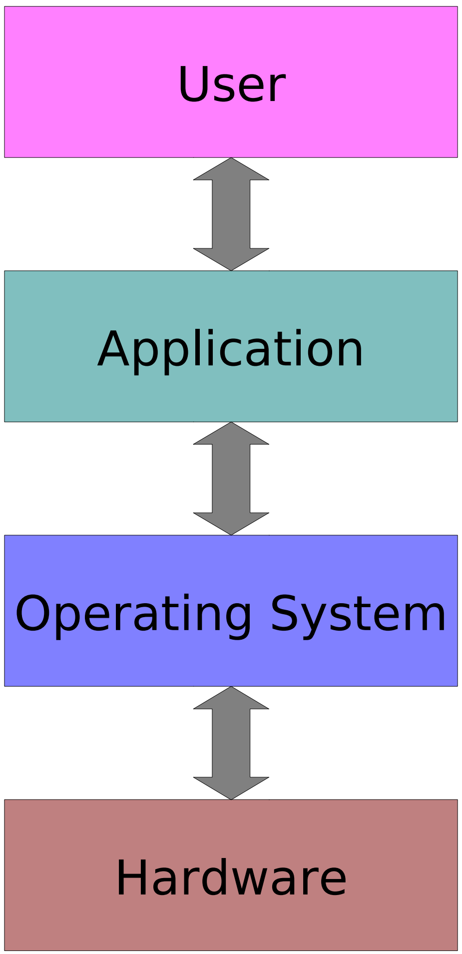
- Functional Programming is a subset of **declarative** programming
- Turns out to be very powerful
- Essentially the programming language is a mathematical description of **what** to do and not a low level description of **how** to do it
  - A compiler can completely rewrite a function so long as the overall effect is unchanged.
  - Because the compiler does the low level stuff, silly programmer errors relating to state can be avoided

Platforms and Operating Systems
(Software to control your hardware)

The Origins of the OS

- A lot of the initial computer programs covered the same ground – they all needed routines to handle, say, floating point numbers, differential equations, etc.
  - Therefore systems soon shipped with libraries: built-in chunks of programs that could be used by other programs rather than re-invented.
- Then we started to add new peripherals (screens, keyboards, etc).
  - To avoid having to write the control code (“drivers”) for each peripheral in each program the libraries expanded to include this functionality
- Then we needed multiple simultaneous users
  - Need something to control access to resources...

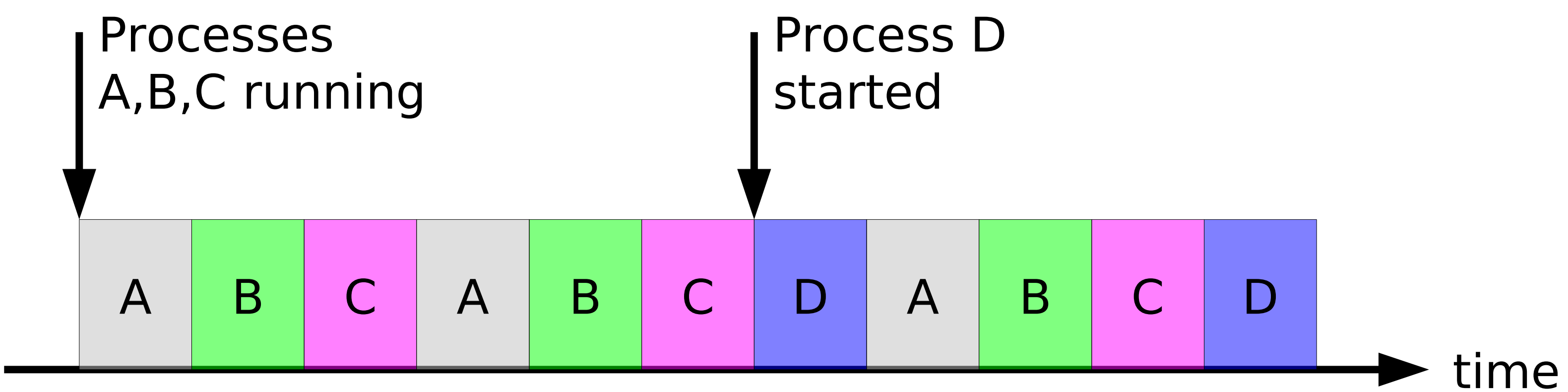
Operating System



- Now sits between the application and the hardware
- Today's examples include MS Windows, GNU Linux, Apple OSX and iOS, Google Android, etc.
- Today's applications depend on **huge** pieces of code that are in the OS and not the actual program code
- The OS provides a common interface to applications
  - Provides common things such as memory access, USB access, networking, etc, etc.

Timeslicing

- Modern OSes allow us to run many programs at once. Or so it seems. In reality a CPU time-slices:
  - Each running program (or "process") gets a certain slot of time on the CPU
  - We rotate between the running processes with each timeslot
  - This is all handled by the OS, which schedules the processes. It is invisible to the running program.



Context Switching

- Every time the OS decides to switch the running task, it has to perform a **context switch**
- It saves all the program's context (the program counter, register values, etc) to main memory
- It loads in the context for the next program
- Obviously there is a time cost associated with doing this...

What Time Slice is Best?

- Longer
  - The computer is more efficient: it spends more time doing useful stuff and less time context switching
  - The illusion of running multiple programs simultaneously is broken
- Shorter
  - Appears more responsive
  - More time context switching means the overall efficiency drops
- Sensible to adapt to the machine's intended usage. Desktops have shorter slices (responsiveness important); servers have longer slices (efficiency important)

The Kernel

- The kernel is the part of the OS that runs the system
  - Just software
  - Handles process scheduling (what gets what timeslice and when)
  - Access to hardware
  - Memory management
- Very complex software – when it breaks... game over.

The Importance of APIs

- API = Application Programming Interface

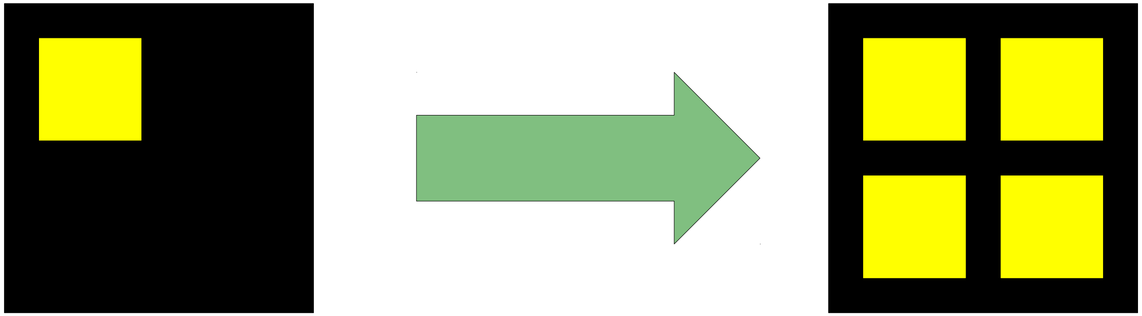
- Software vendors ship their libraries with APIs, which describes only what is need for a programmer to use the library in their own program.
  - The library itself is a black box – shipped in binary form.
- Operating systems are packed with APIs for e.g. window drawing, memory access, USB, sound, video, etc.
  - By ensuring new versions of the software support the same API (even if the result is different), legacy software can run on it.

Platforms

- A typical program today will be compiled for a specific architecture, a specific operating system and possibly some extra third party libraries.
  - So PC software compiled for linux does not work under Windows for example.
- We call the {architecture, OS} combination a *platform*
- The platforms you are likely to encounter here:
  - Intel/Linux
  - Intel/Windows
  - Intel/OSX
  - ARM/iOS
  - ARM/Android

Multicore Systems

- Ten years ago, each generation of CPUs packed more in and ran faster. But:
  - The more you pack stuff in, the hotter it gets
  - The faster you run it, the hotter it gets
  - And we got down to physical limits anyway!!
- We have seen a shift to multi-core CPUs
  - Multiple CPU cores on a single CPU package (each runs a separate fetch-execute cycle)
  - All share the same memory and resources!

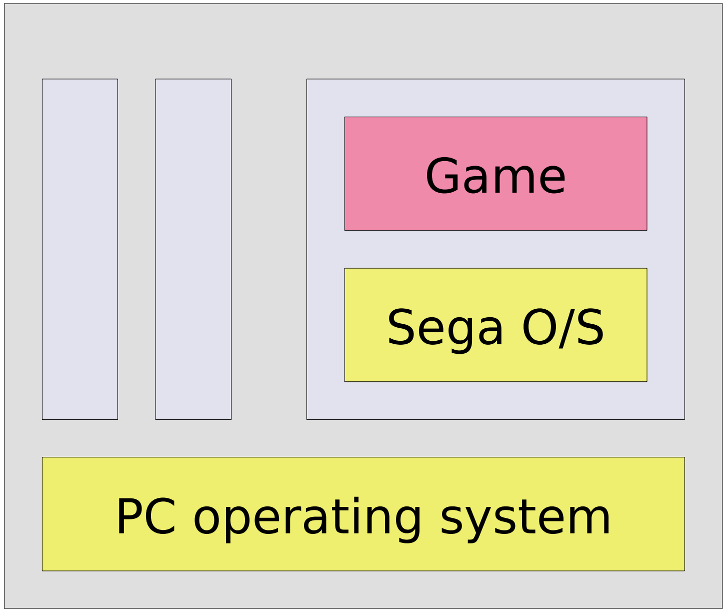


The New Challenge

- Two cores run completely independently, so a single machine really *can* run two or more applications simultaneously
- BUT the real interest is how we write programs that use **more** than one core
  - This is hard because they use the same resources, and they can then interfere with each other
  - Those sticking around for IB CST will start to look at such '**concurrency**' issues in far more detail

Virtual Machines

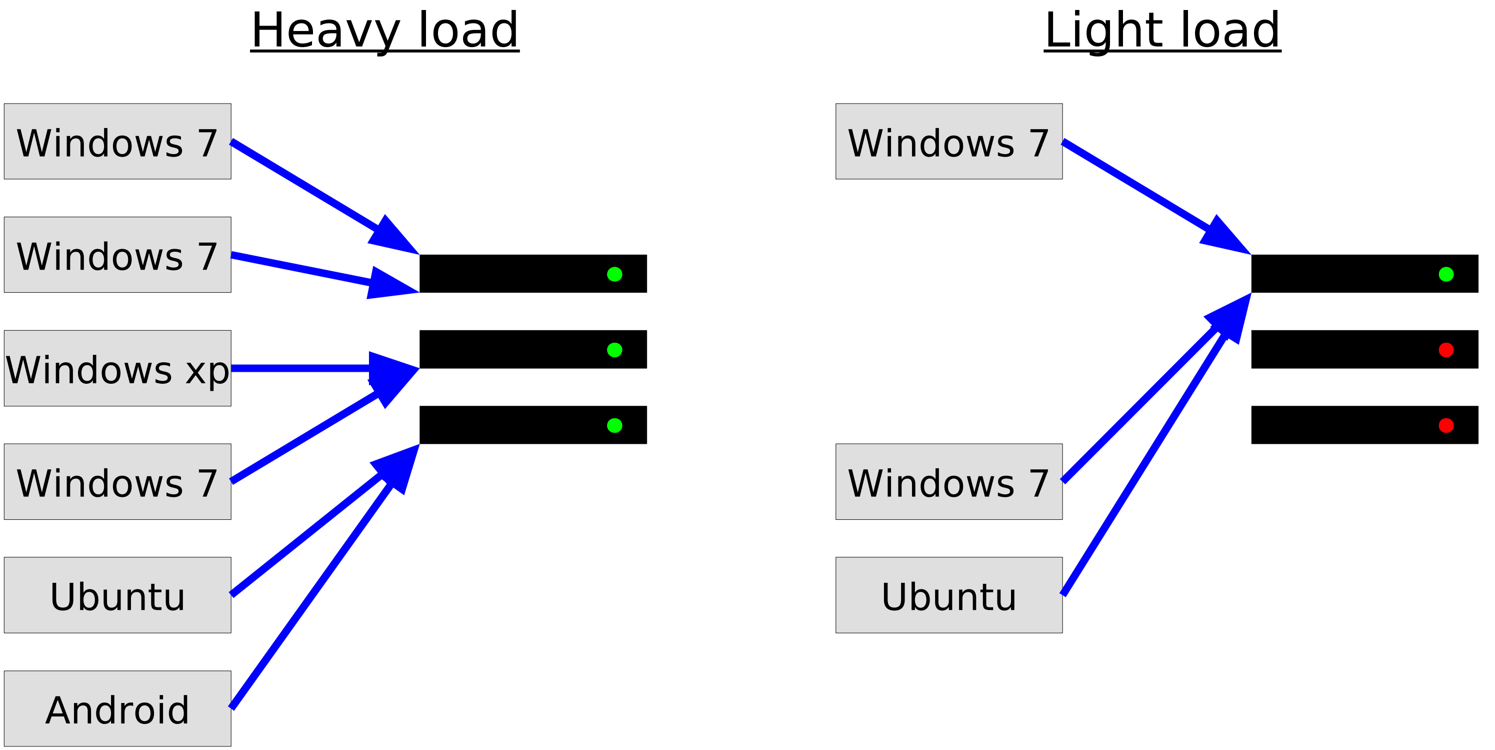
- Go back 20 years and emulators were all the rage: programs on architecture X that simulated architecture Y so that programs for Y could run on X
- Essentially interpreters, except they had to recreate the entire system. So, for example, they had to run the operating system on which to run the program.



- Now computers are so fast we can run multiple *virtual machines* on them
- Allows us to run multiple operating systems simultaneously!

Virtualisation

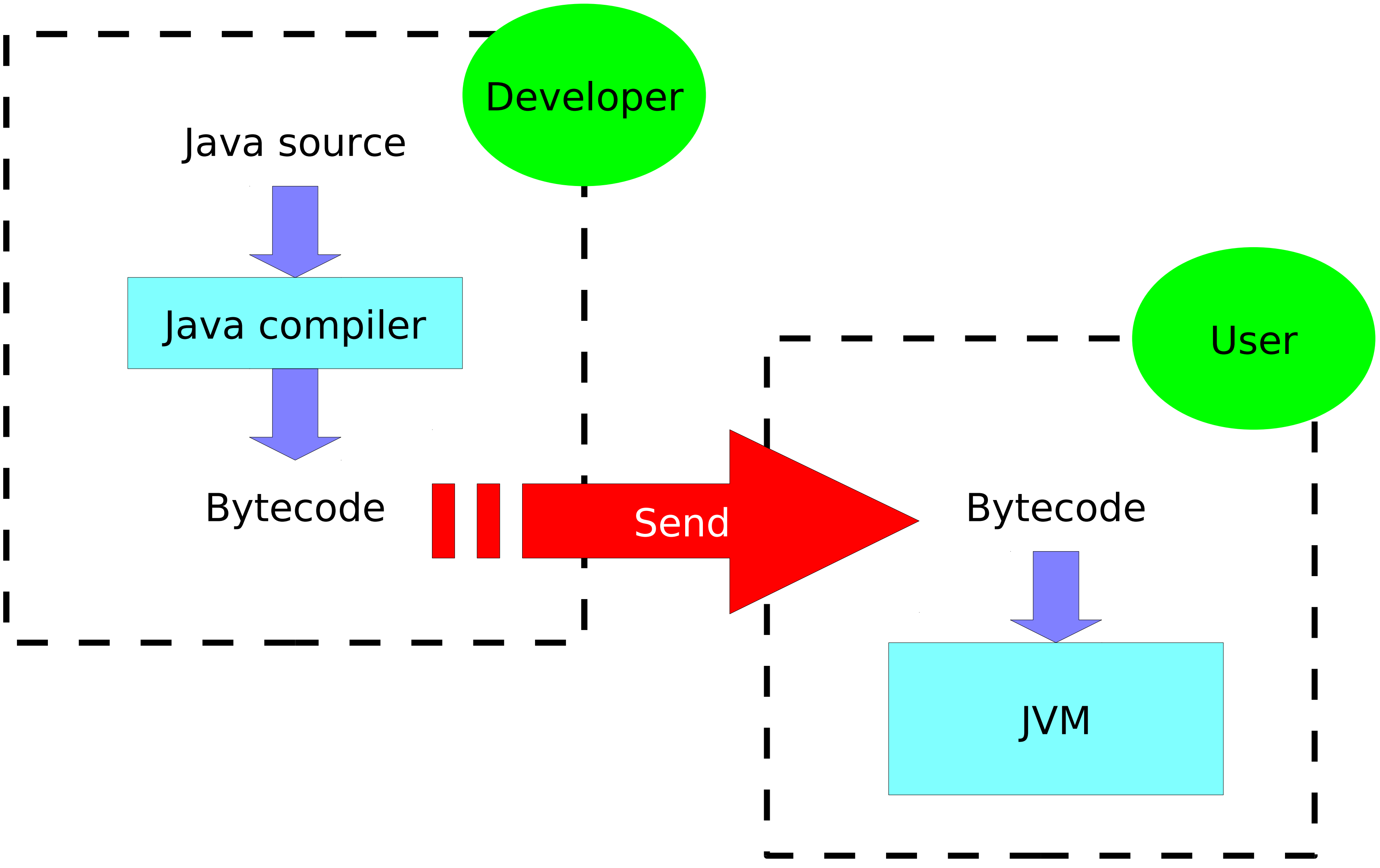
- Virtualisation is the new big thing in business. Essentially the same idea: emulate entire systems on some host server
- But because they are virtual, you can swap them between servers by copying state
- And can dynamically load your server room!



The Java Approach

- *Java* was born in an era of internet connectivity. SUN wanted to distribute programs to internet machines
  - But many architectures were attached to the internet – how do you write one program for them all?
  - And how do you keep the size of the program small (for quick download)?
- Could use an interpreter (→ Javascript). But:
  - High level languages not very space-efficient
  - The source code would implicitly be there for anyone to see, which hinders commercial viability.
- Went for a clever hybrid interpreter/compiler

Java Bytecode I



Java Bytecode I

- SUN envisaged a hypothetical Java Virtual Machine (JVM). Java is compiled into machine code (called bytecode) for that (imaginary) machine. The bytecode is then distributed.
- To use the bytecode, the user must have a JVM that has been specially compiled for their architecture.
- The JVM takes in bytecode and spits out the correct machine code for the machine. i.e. is a bytecode interpreter

Java Bytecode II

+ Bytecode is compiled so not easy to reverse engineer

+ The JVM ships with tons of libraries which makes the bytecode small

+ The toughest part of the compile (from human-readable to computer readable) is done by the compiler, leaving the computer-readable bytecode to be translated by the JVM (→ easier job → faster job)

- Still a performance hit compared to fully compiled ("native") code

Where Do You Go From Here?

- Paper 1
  - FoCS: look at the fundamentals of CS whilst learning ML
  - Discrete Maths: build up your knowledge of the maths needed for good CS
  - OOP/Java: look at imperative programming as it is used in the 'real world'
  - Floating Point: learn how to use computers for floating point computations (and when not to trust them..!)
  - Algorithms: The core of CS: learn how to do things efficiently/optimally
- Paper 2
  - Digital Electronics: hardware in detail
  - Operating Systems: an in-depth look at their workings
  - Probability: learn how to model systems
  - Software Design: good practice for large projects
  - RLFA: an intro to describing computer systems mathematically