Computer Fundamentals

Dr Robert Harle

CST Part IA
NST Part IA (CS option)
PPS (CS option)

Michaelmas 2012

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.

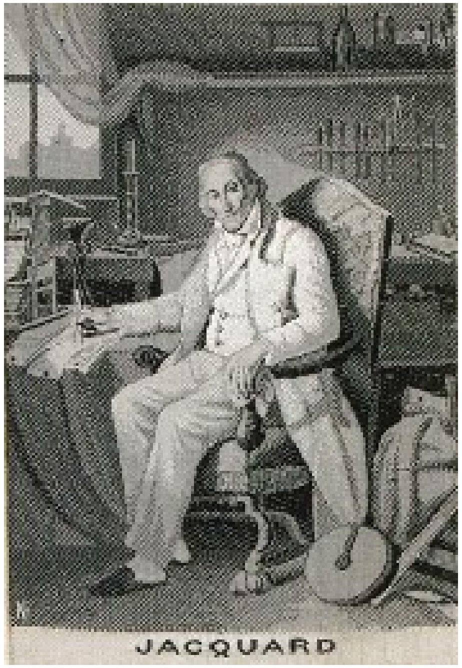
A Brief History of Computers

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



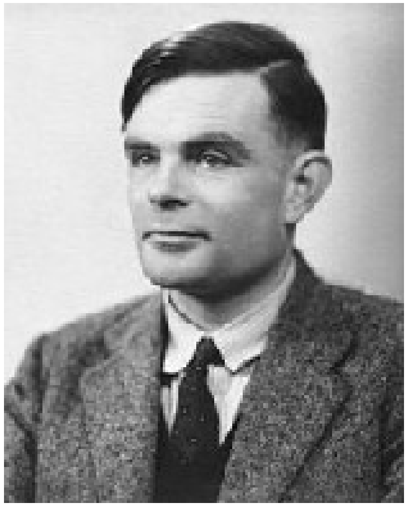
Turing Machines

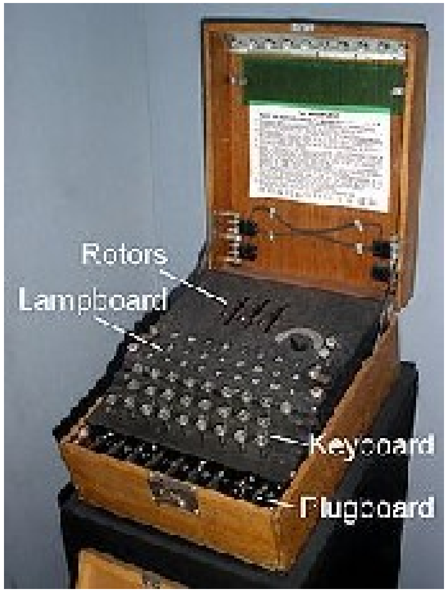
- Inspired by the typewriter (!), Alan Turing (King's) created a theoretical model of a computing machine in the 1930s. He broke the machine into:
  - A tape – infinitely long, broken up into cells, each with a symbol on them
  - A head – that could somehow read and write the current cell
  - An action 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!

Turing and the War





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)

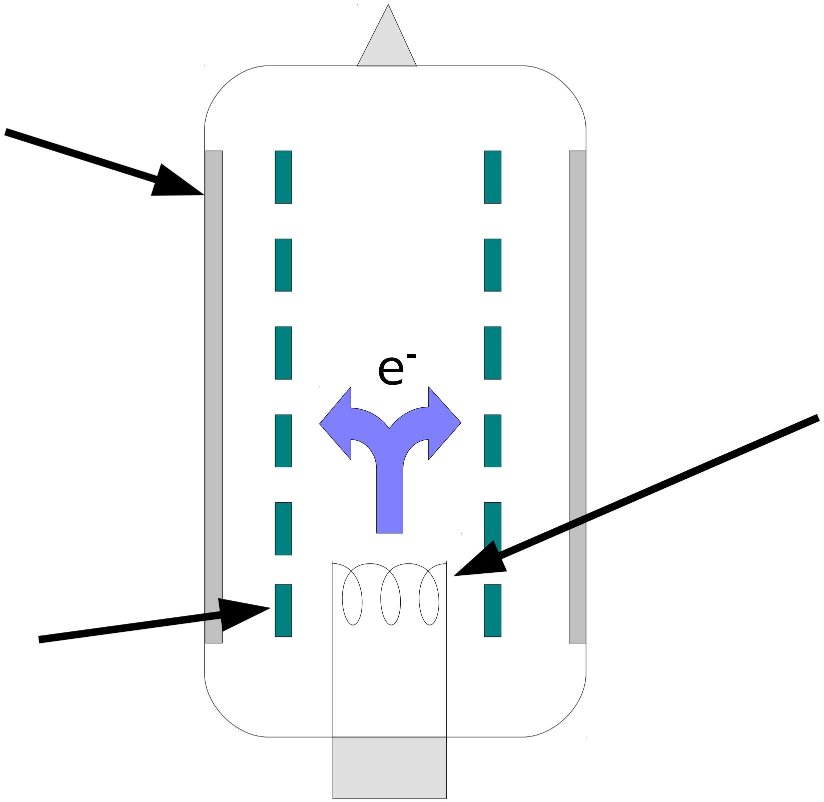
- This is useful practically too. Analogue devices:
  - have temperature-dependent behaviour
  - produce inexact answers due to component tolerances
  - are unreliable, big and power hungry

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?

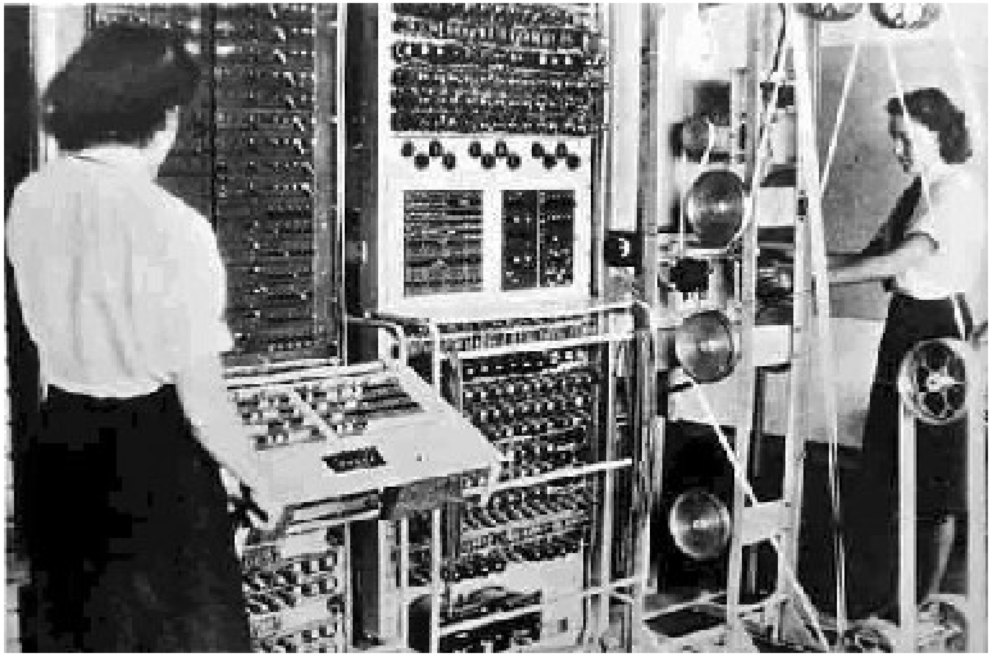
1940-58 Vacuum Tubes

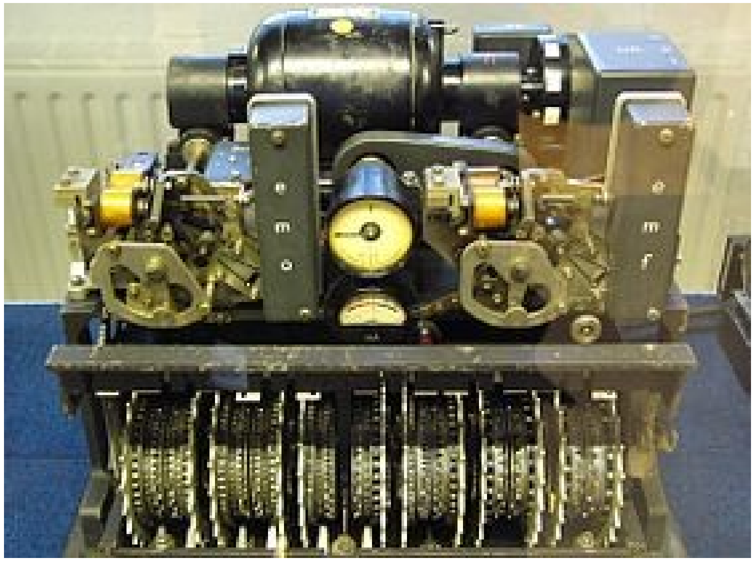
- Vacuum tubes are really just modified lightbulbs that can act as amplifiers or, crucially, switches.



- 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

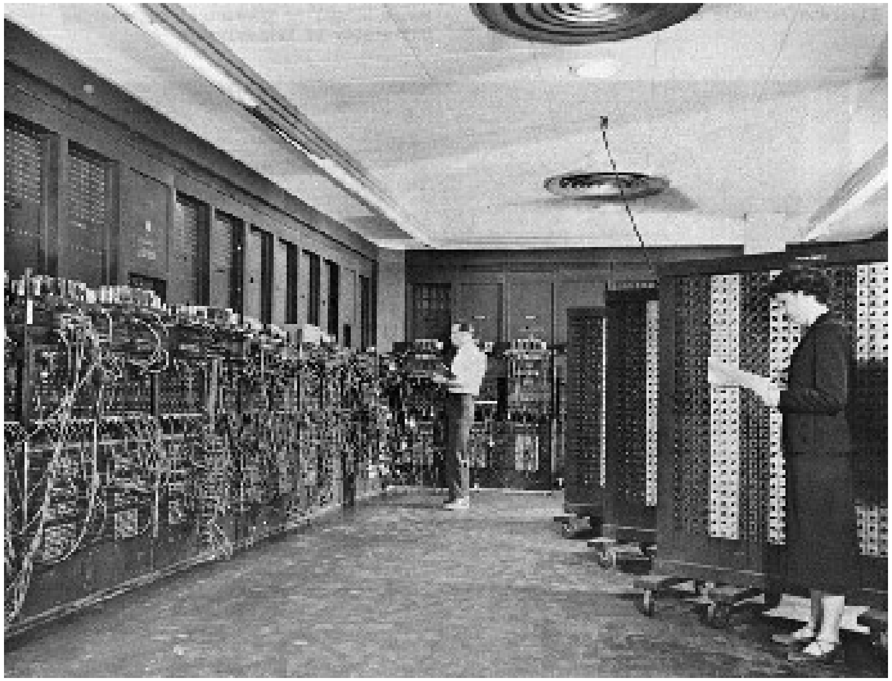




- 1944, Bletchley park
- Designed to break the German Lorenz SZ40/42 encryption machine
- Fed in encrypted messages via paper tape. Colussus then simulated the positions of the Lorenz wheels until it found a match with a high probability
- No internal program – programmed by setting switches and patching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

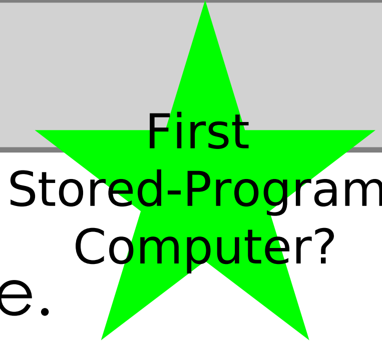
ENIAC

- 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...

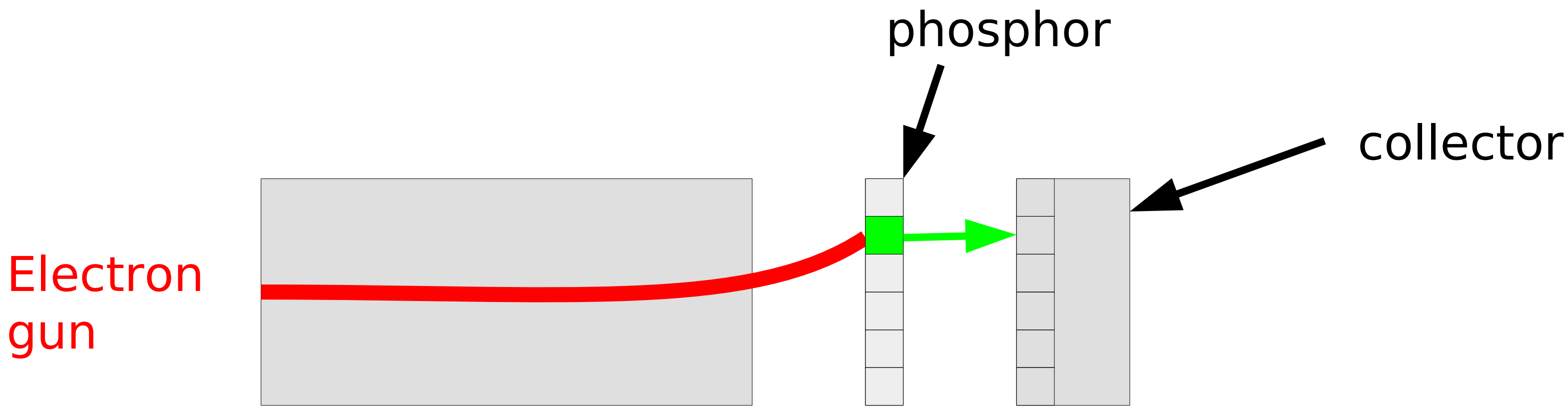


- Next step was to read in the “action table” (aka program) from tape as well as the data
- For this we needed more general purpose memory to store the program, input data and output

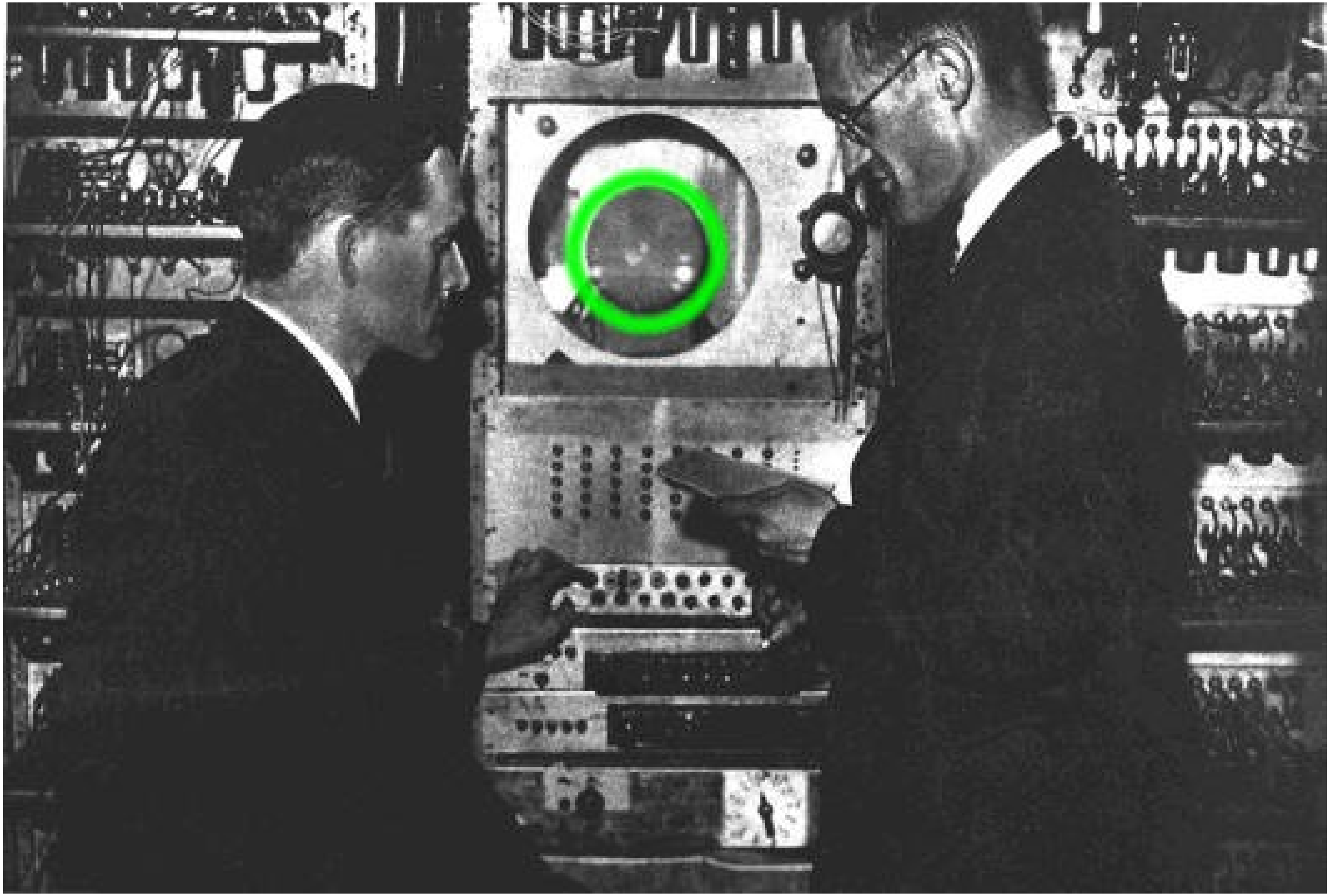
Manchester Baby



- 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.

- Memory came in the form of a mercury delay line

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

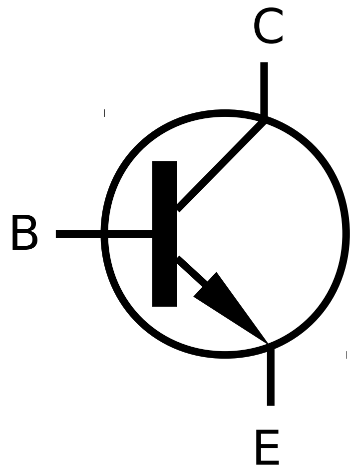
Storage: Stored-Program Machines

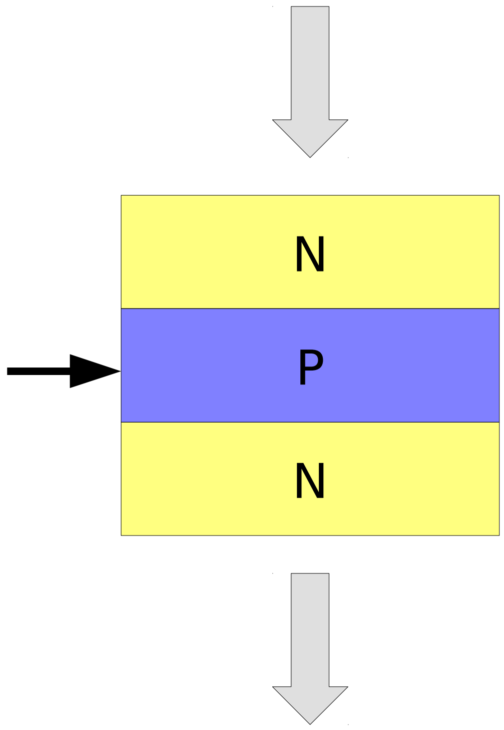
- So where do you store your programs and data?

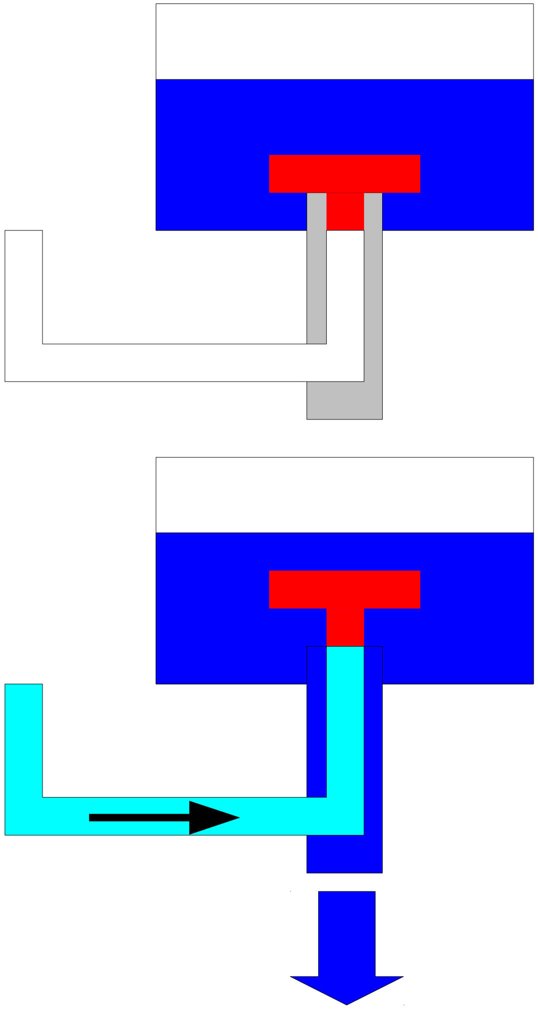
	Von-Neumann		Harvard
	Same memory for programs and data		**Separate** memories for programs and data
	+ Don't have to specify a partition so more efficient memory use		- Have to decide in advance how much to allocate to each
	+ Programs can modify themselves, giving great flexibility		+ Instruction memory 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

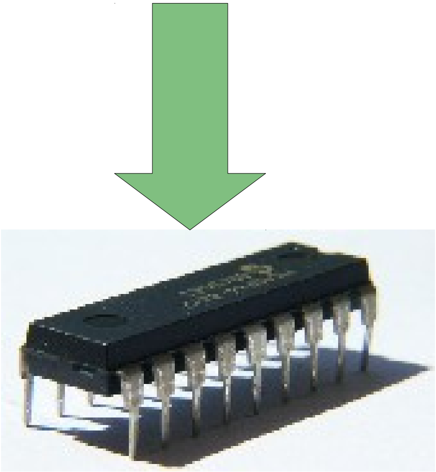
- Vacuum tubes bulky, hot and prone to failure
- Solution came from Bell labs (telecoms research)







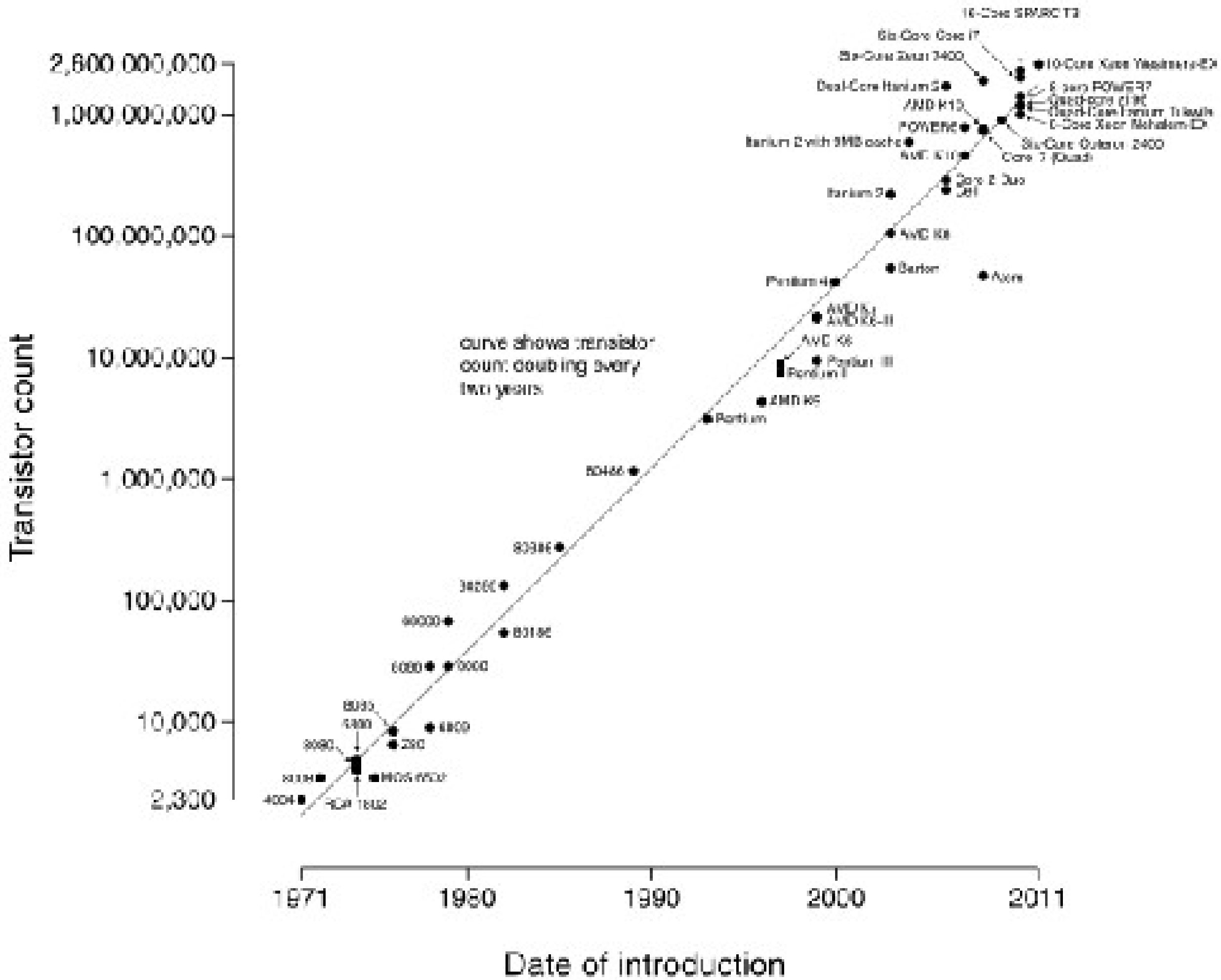
1965-70 Integrated Circuits



- Semiconductors could replace traditional electronics components → use a slice of semiconductor and 'etch' on a circuit
- End up with an Integrated Circuit (IC) a.k.a a microchip
- Much easier to pack components on an IC, and didn't suffer from the reliability issues of the soldering iron

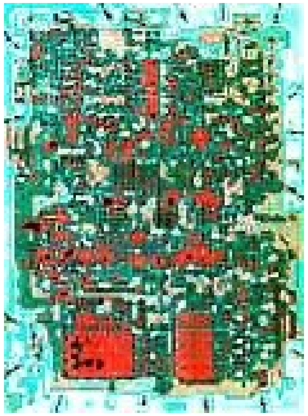
Moore's Law: the number of transistors on an IC will double every two years

Microprocessor Transistor Counts 1971-2011 & Moore's Law



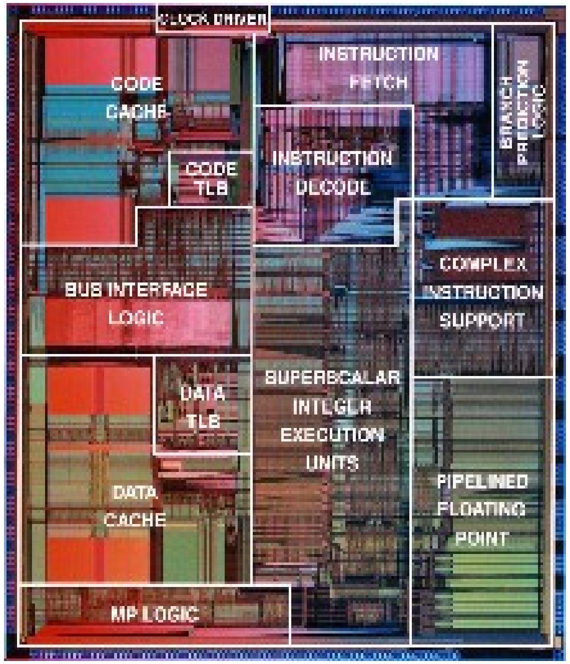
The Rise of Intel

- Intel started in 1968 manufacturing ICs, producing ICs with a particular target of memory (RAM, see later)
- 1969 – commissioned to make 12 custom chips for a calculator (one for keyboard scanning, one for display control, etc)
- Not enough resource so instead proposed a single general-purpose logic chip that could do all the tasks
- 1971 - Managed to buy the rights and sold the chip commercially as the first **microprocessor**, the Intel 4004



1971- Microprocessor Age

- The 4004 kick-started an industry and lots of competitors emerged
- Intel very savvy and began an “intel inside” branding assault with products like the 386
- Marketing to consumers, not system builders any more

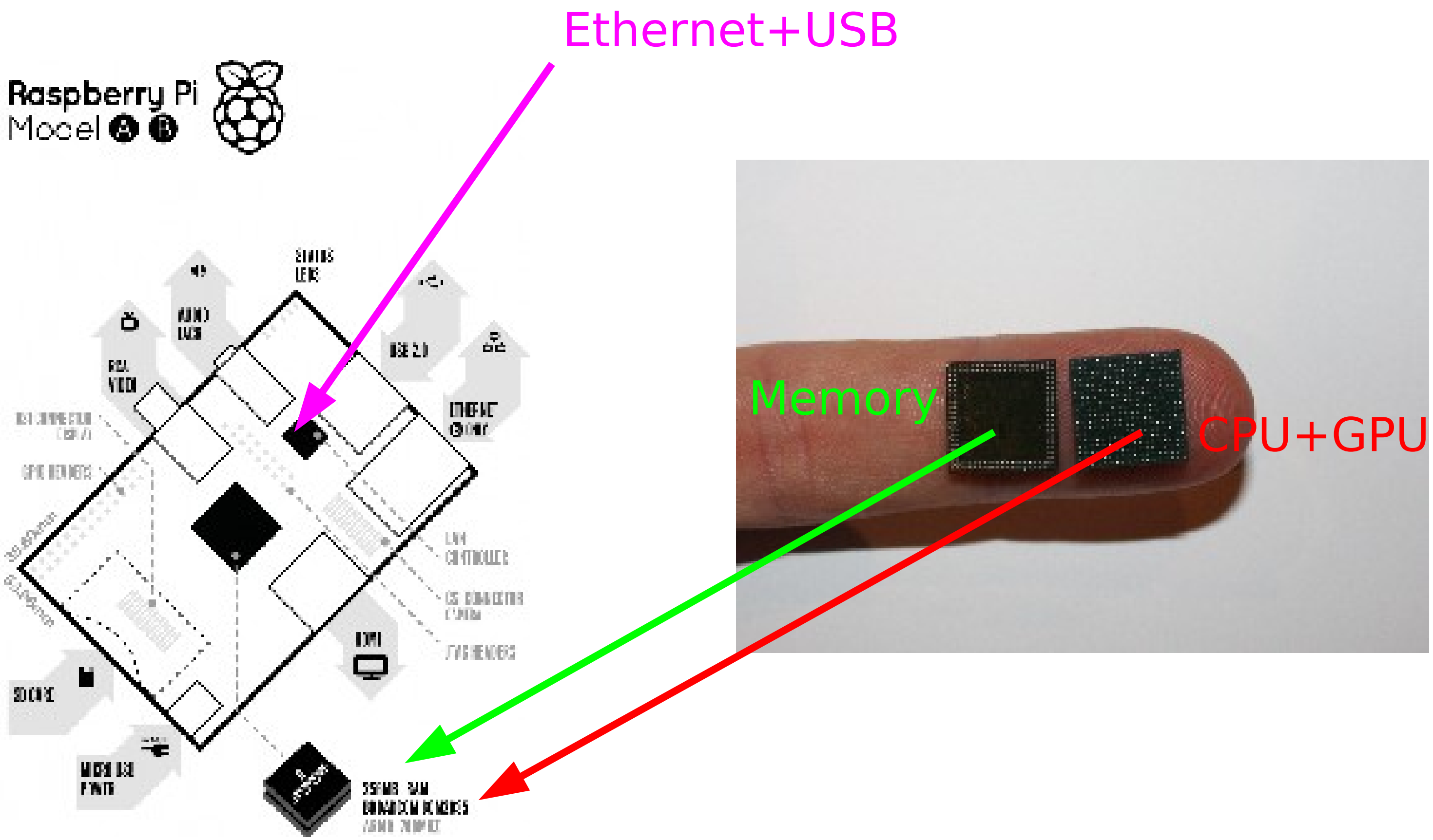


The Rise of ARM

- After the BBC micro, Acorn wanted a new processor and set about designing a cheap, simple-to-implement CPU (using a RISC approach – see later)
- Apple was interested and started to work with them. Eventually the project unit was spun out to form Advanced RISC Machines (ARM) Ltd.
- Chips were very low power and cheap, but struggled against the might of intel's more complex chips
- BUT then the PDA/smartphone/mobile revolution came along and suddenly ARM had the perfect product – cheap, low power, simple and reasonable performance
- Now accounts for the majority of all 32-bit processor produced – **and Cambridge-based too...**

System-on-Chip (SoC)

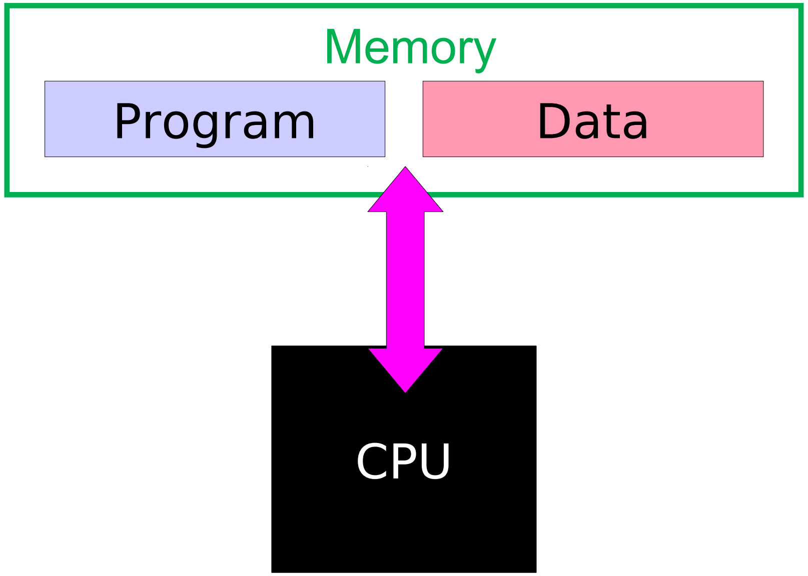
- For smaller systems, often see hardware elements bundled together to form an SoC e.g. R-Pi



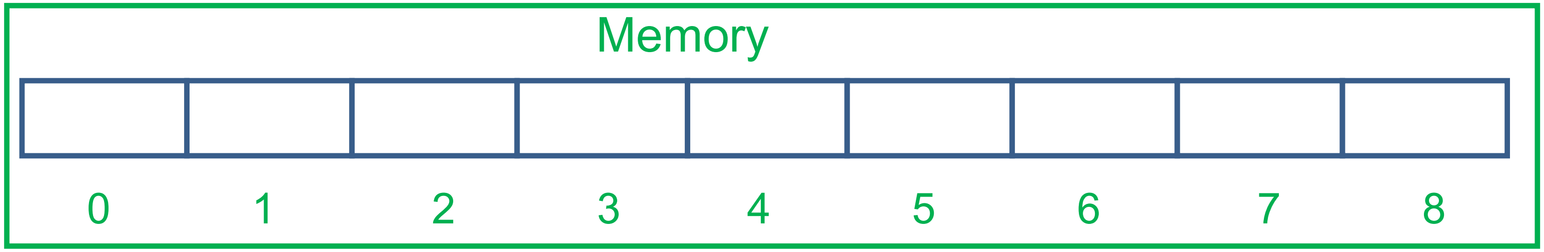
The CPU in more Detail

Programs, Instructions and Data

- Recall: Turing's universal machine reads in an action 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.

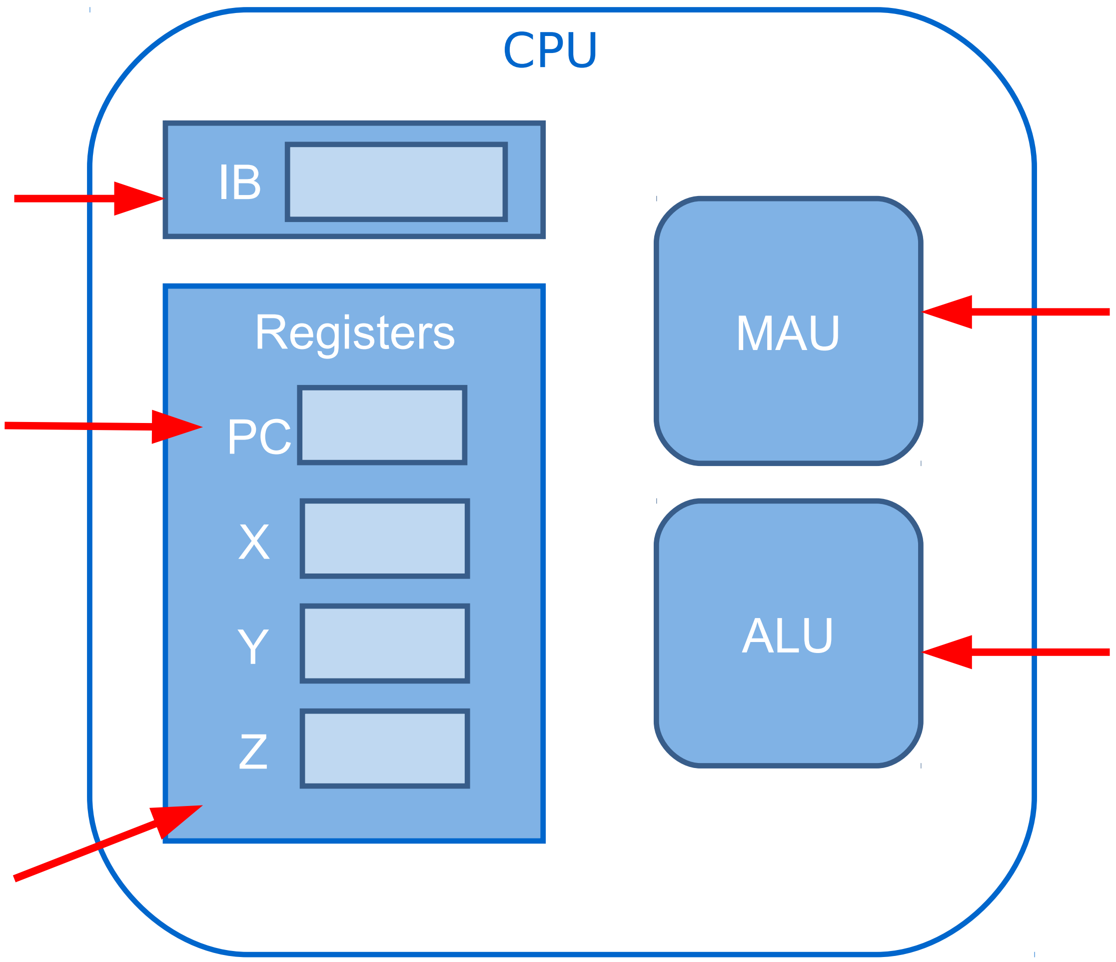


Simple Model of Memory

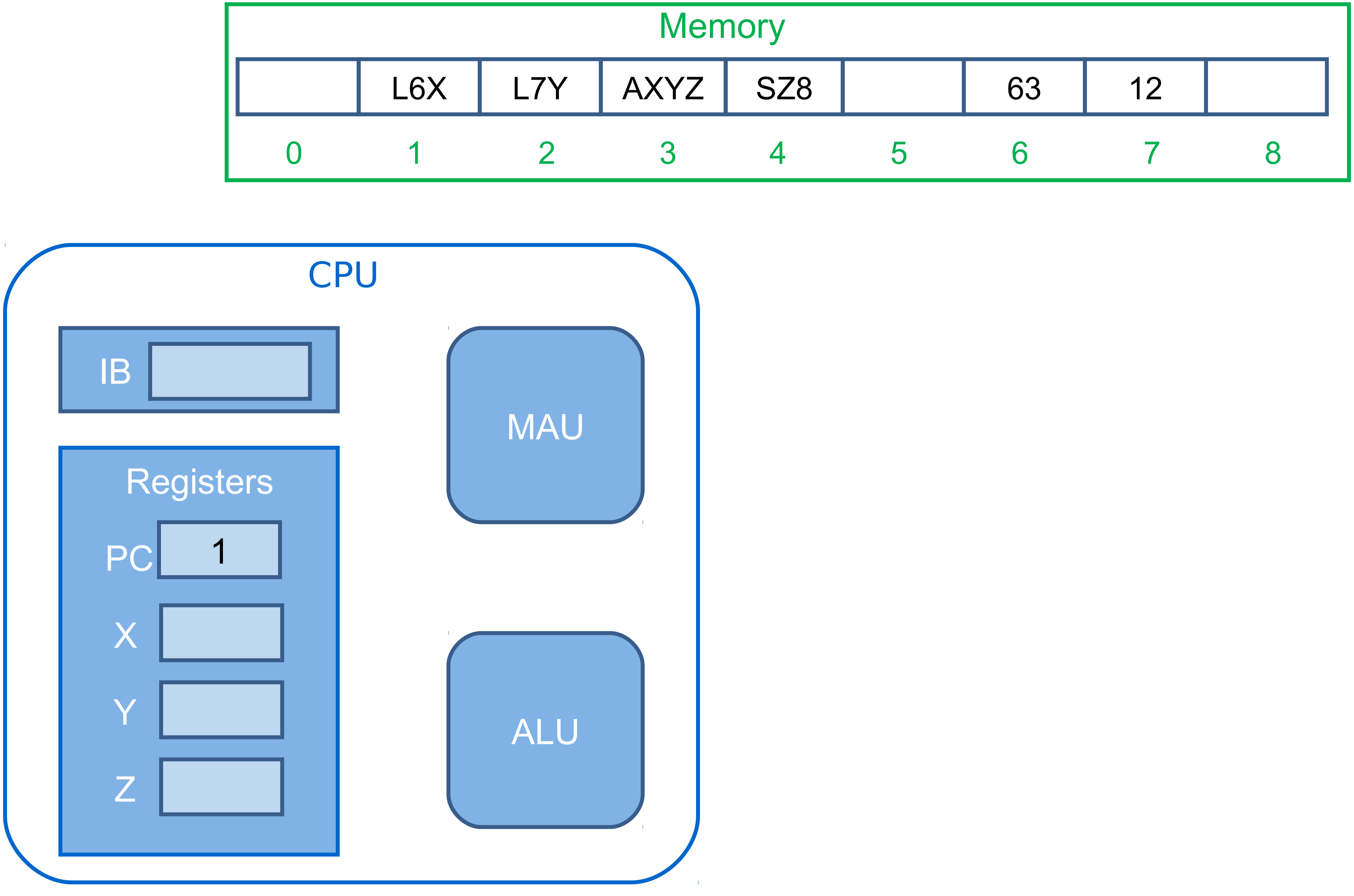


- We think of memory abstractly, as being split into discrete chunks, each given a unique *address*
- We can read or write in whole chunks
- Modern memory is <u>big</u>

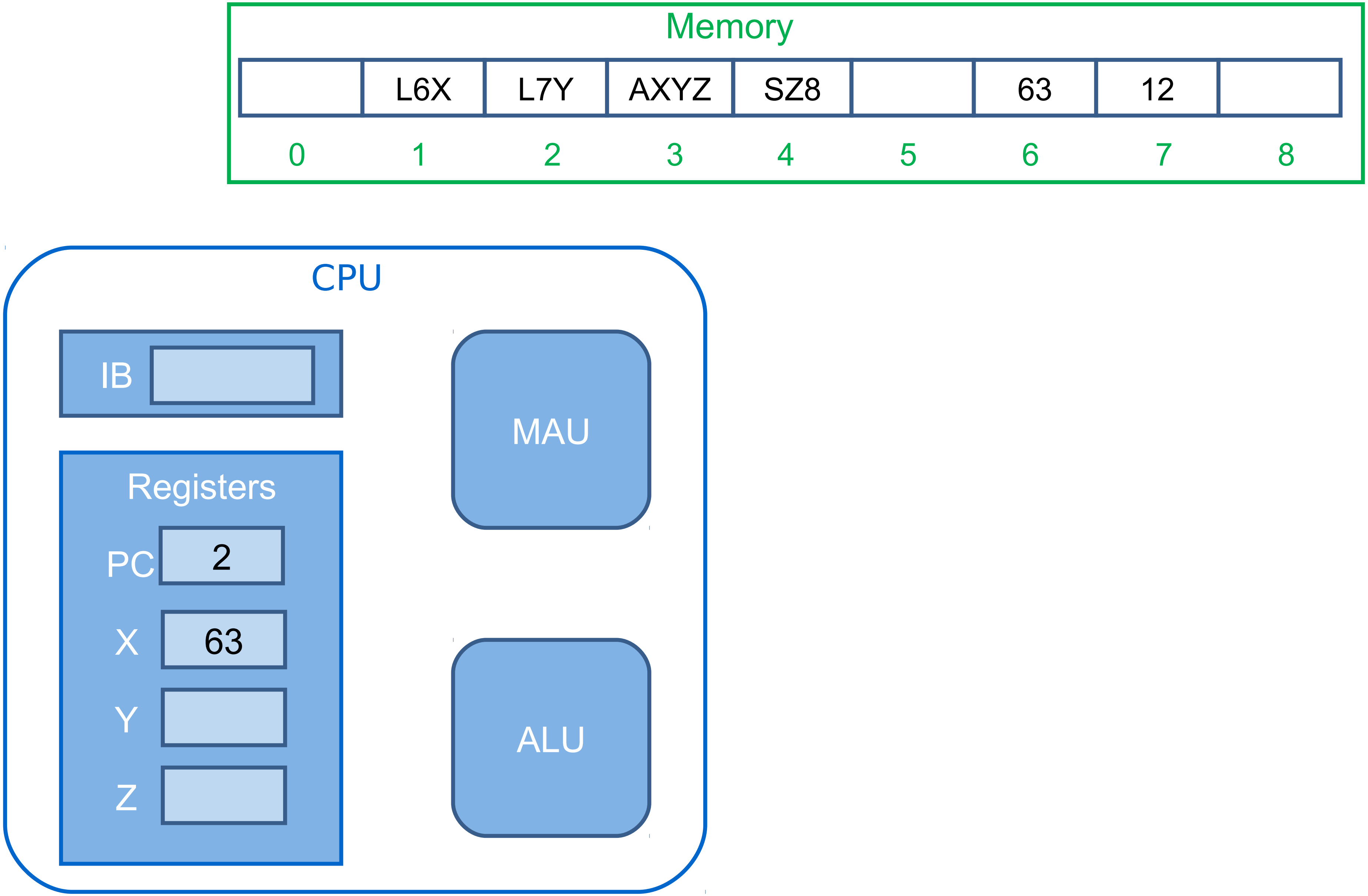
Simple Model of a CPU



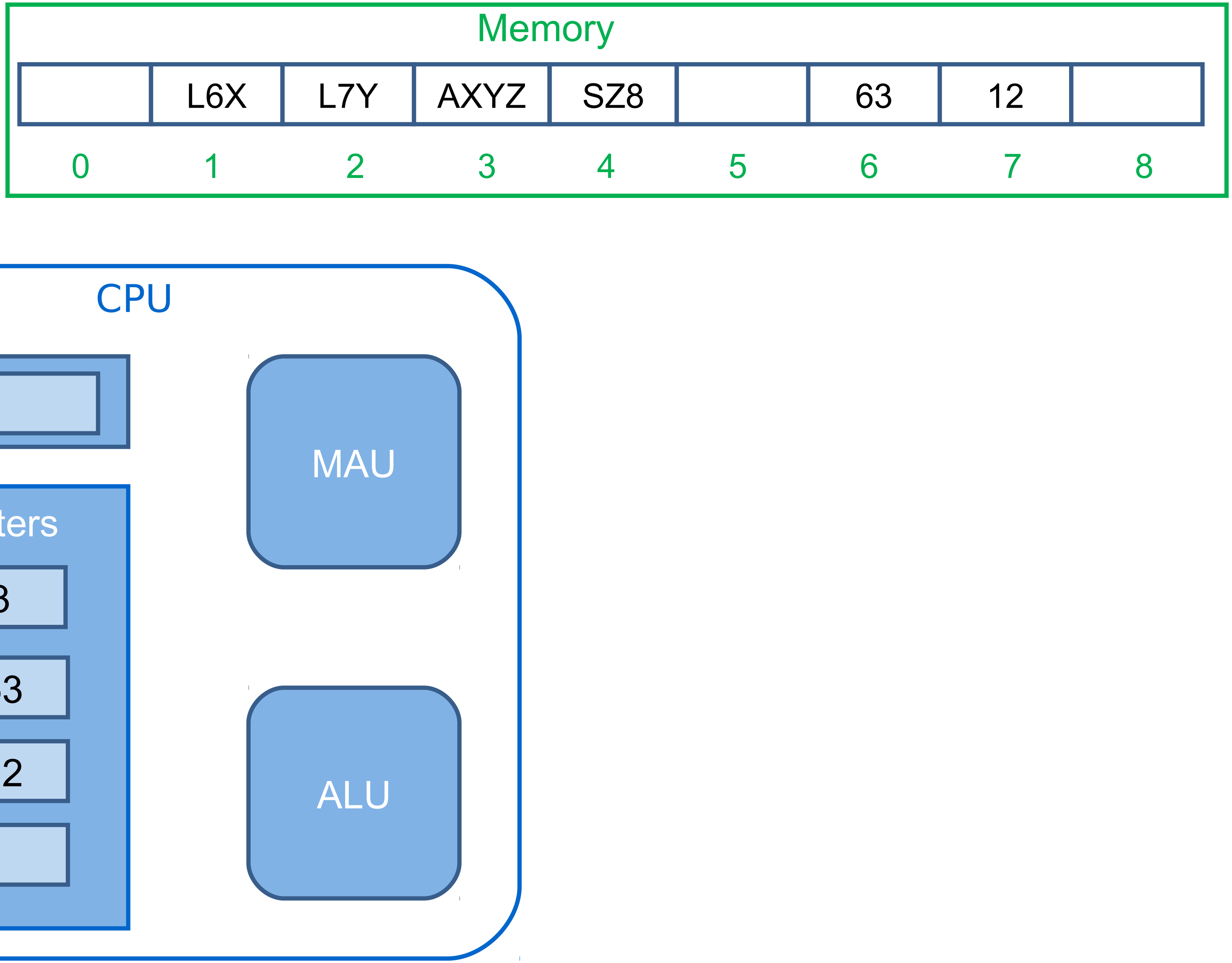
Fetch-Execute Cycle I



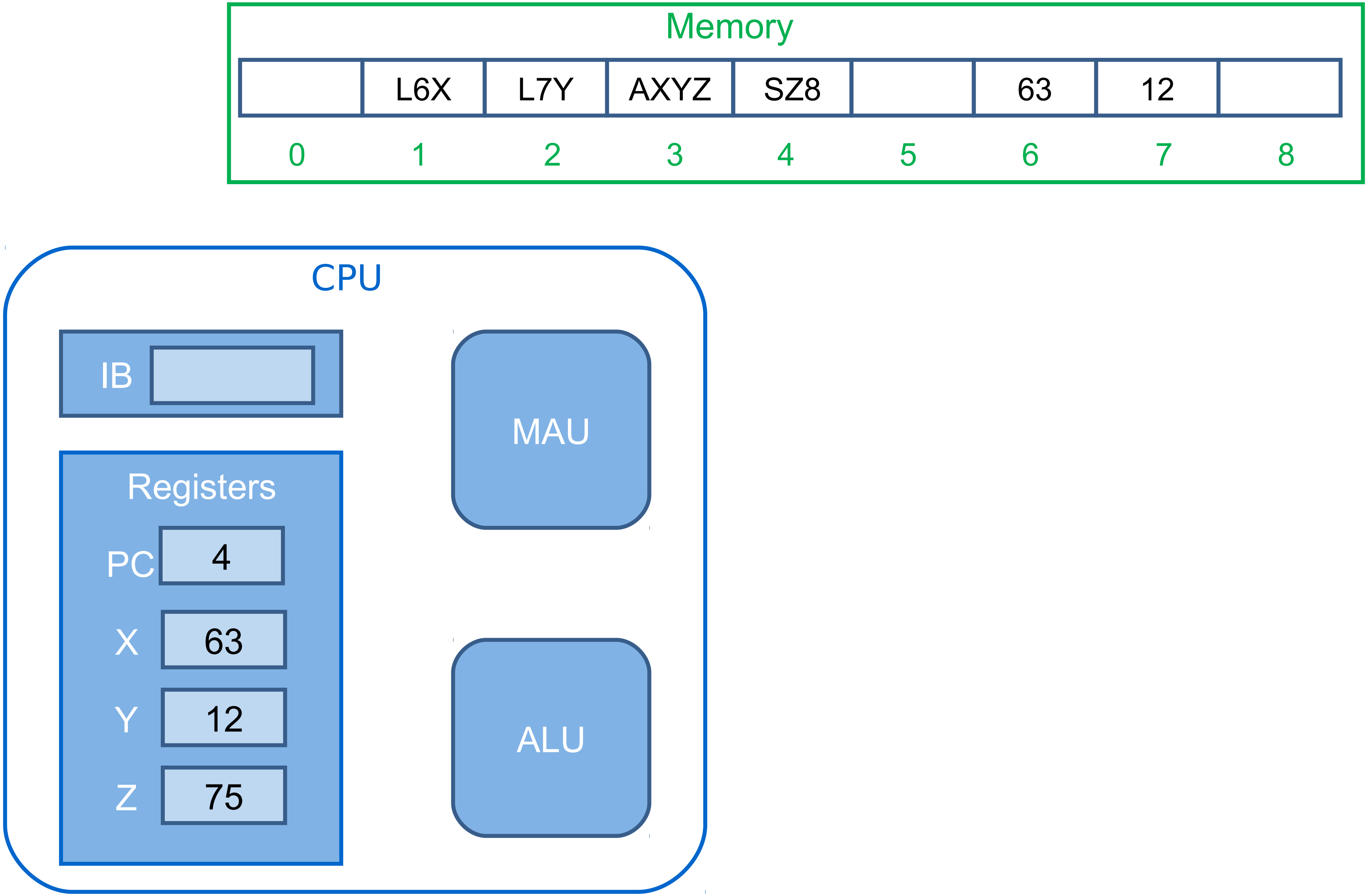
Fetch-Execute Cycle II



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

Handling Numbers in the CPU

ALU Circuitry

- The ALU in the CPU is responsible for arithmetic operations. Exactly *what* is supported directly is CPU manufacturer-dependent
- Integer arithmetic is always supported, but there are issues:
  - Overflow
  - Representing fractional numbers
  - Underflow (see floating point course)
  - Negative numbers

Unsigned Integer Addition

- You should be happy that binary addition can use the same algorithm as decimal addition as taught in junior school.
- CPUs (or rather ALUs in them) implement this algorithm, but there is a practical issues: there is a set number of bits in the register that we can unintentionally exceed (overflow)
- The ALU has a carry flag (a single bit in a special register) that is switched on if the addition has a carry left after processing the most significant bit

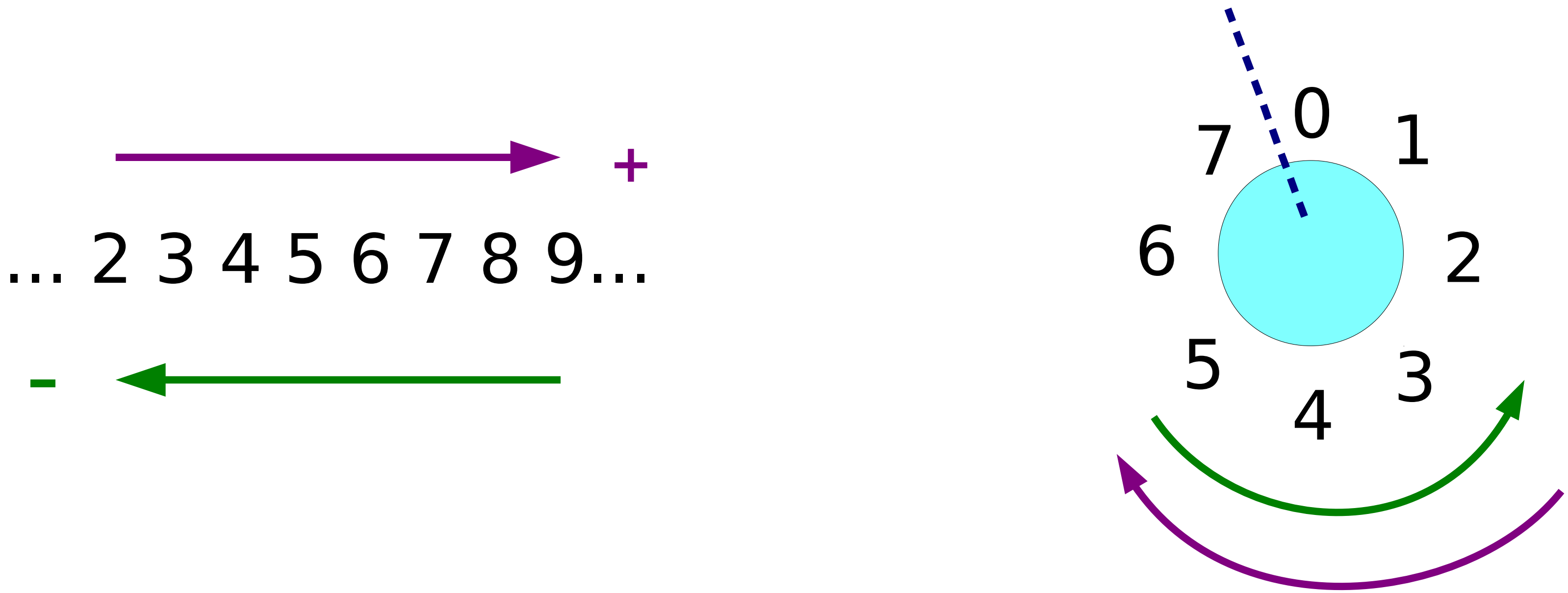
```
  001
+ 001
```

Carry flag:

```
  111
+ 001
```

Carry flag:

Modulo Arithmetic



- Overflow takes us across the dotted boundary
  - So 7+1=0 (overflow)
  - We say this is (7+1) mod 8

Unsigned Integer Subtraction

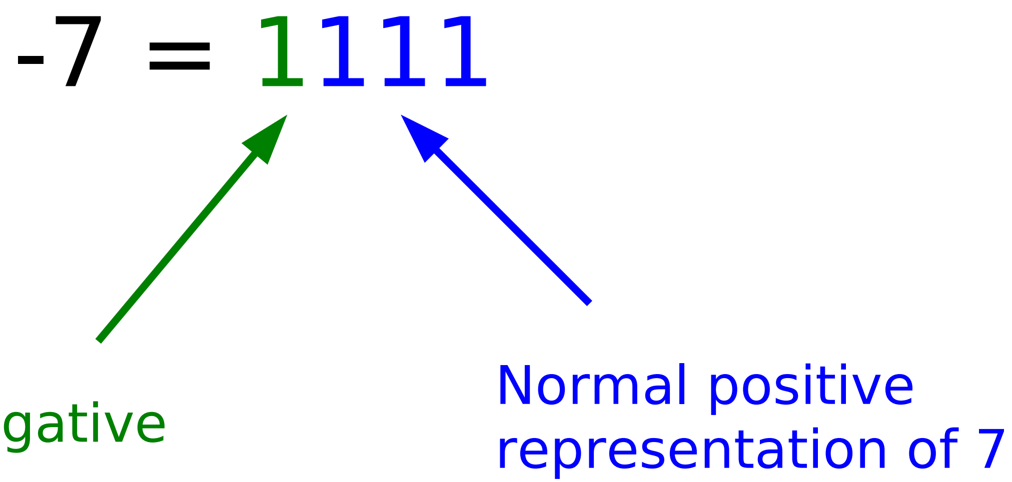
- Integer subtraction can proceed as decimal subtraction, 'borrowing' from the left if necessary
- If we still need to borrow after the left-most bit, this signifies an error and the carry flag is set.

```
  011
- 001

Carry flag:
```

```
  001
- 010

Carry flag:
```

Negative Numbers

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

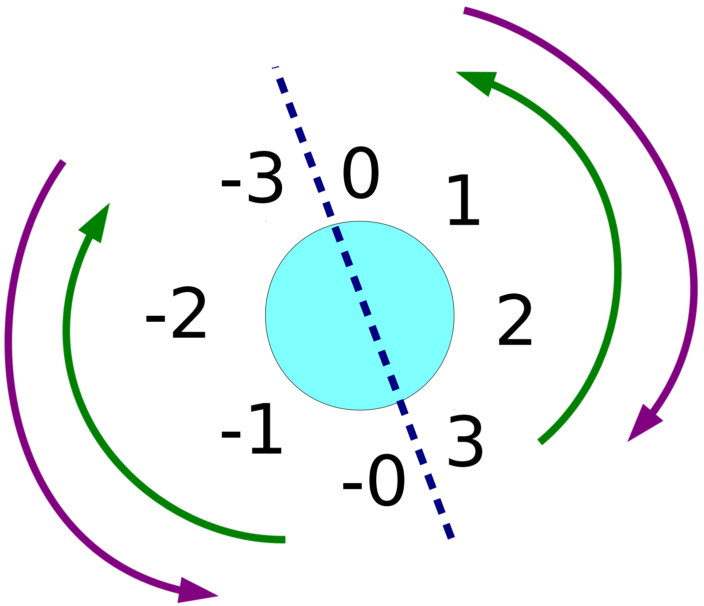
Sign-mag interpretation		Unsigned interpretation
-5	1101	+13
+1	+ 0001	+1
-6	1110	+14

Our unsigned addition alg.

Sign-mag interpretation		Unsigned interpretation
-5	1101	+13
-1	- 0001	-1
-4	1100	+12

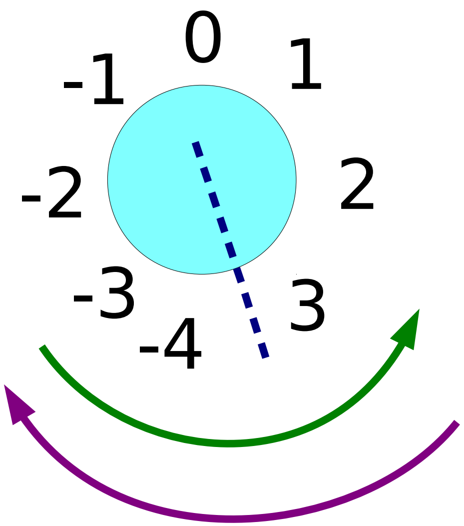
Our unsigned subtraction alg.

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)

Signed Addition

2's-comp		Unsigned
-3	1101	+13
+1	+0001	+1
-2	1110	+14

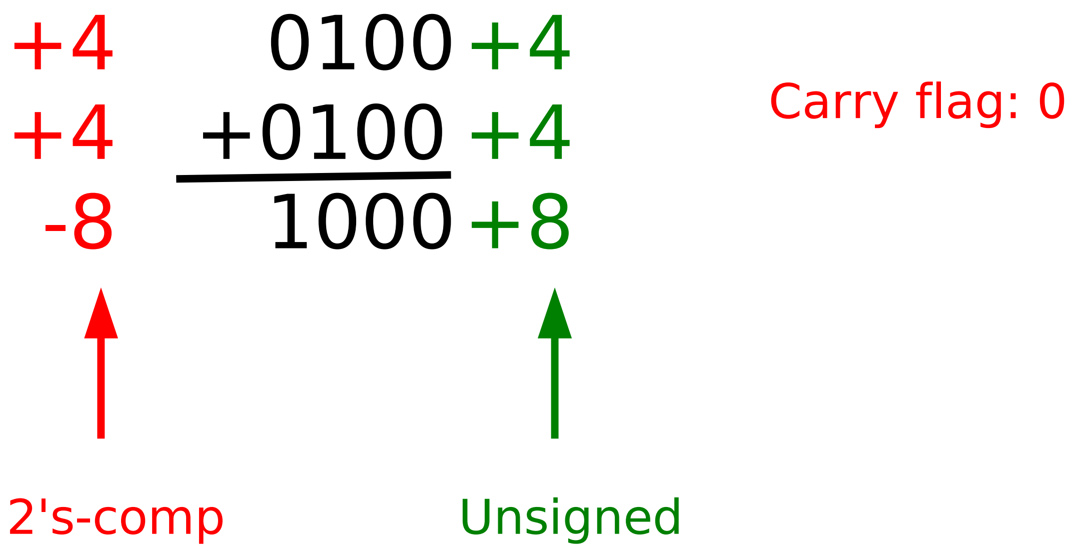
Our unsigned addition alg.

2's-comp		Unsigned
-3	1101	+13
- +1	- 0001	- +1
-4	1100	+12

Our unsigned subtraction alg.

Signed Addition

- So we can use the same circuitry for unsigned and 2s-complement addition :-)
- Well, almost.



- The problem is our MSB is now signifying the sign and our carry should really be testing the bit to its right :-(
- So we introduce an overflow flag that indicates this problem

Flags Summary

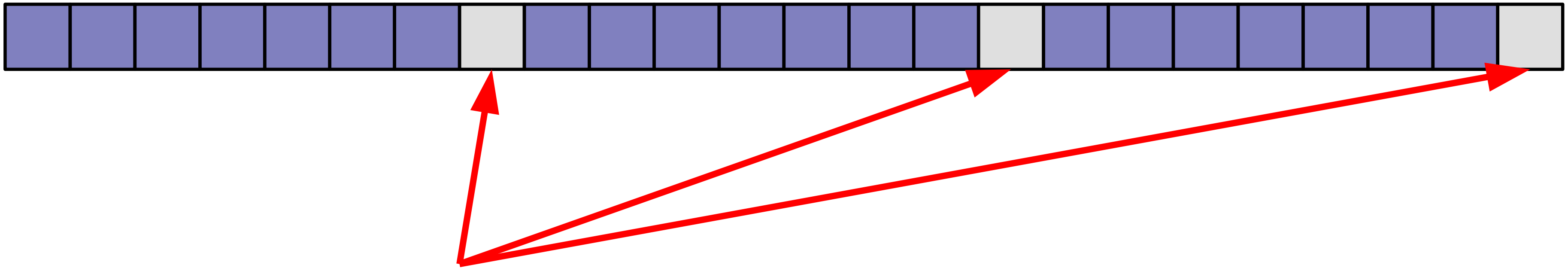
- When adding/subtracting
  - The **carry** flag indicates an overflow for **unsigned** numbers
  - The **overflow** flag indicates an overflow for **signed** integers

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



ASCII Codes

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

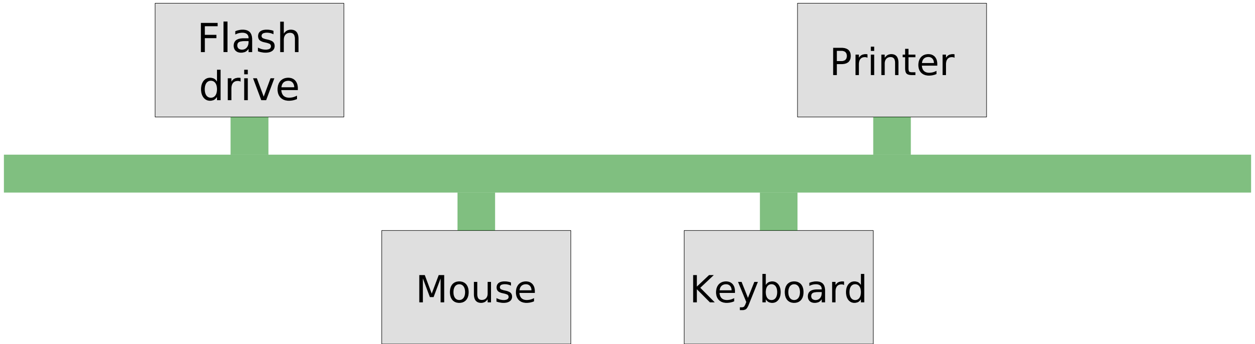
Other encodings

- 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...*

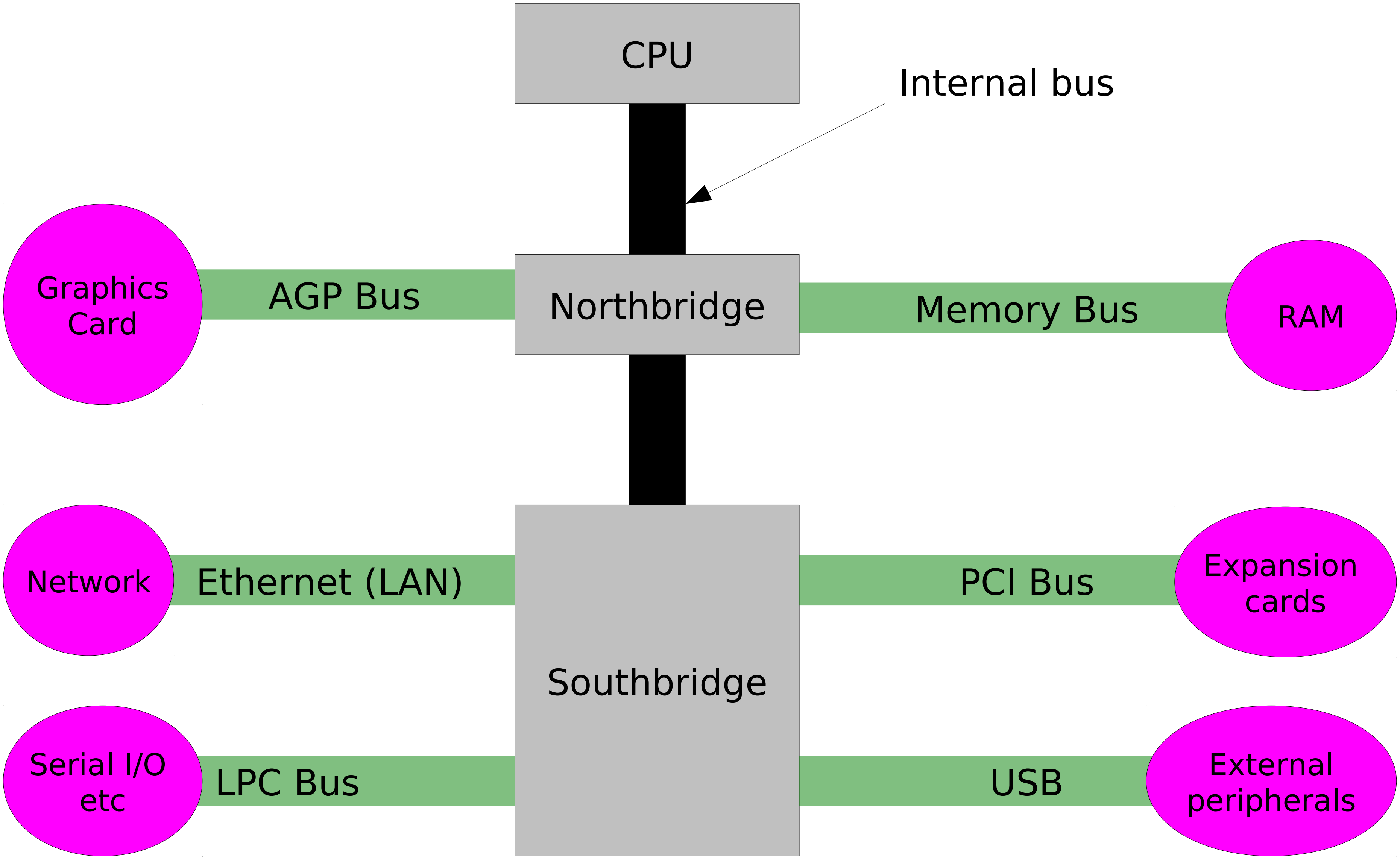
The Guts of Modern Systems

Desktop Systems Today

- Based on a core system plus peripherals:
  - Input (mouse, keyboard, etc)
  - Output (printer, display)
  - Network adapter, etc
- Peripherals connect to **buses** in order to communicate with the core system
  - A bus is just a set of wires that can be used by multiple peripherals. Special control wires are used to ensure the data from two or more connected peripherals do not clash.

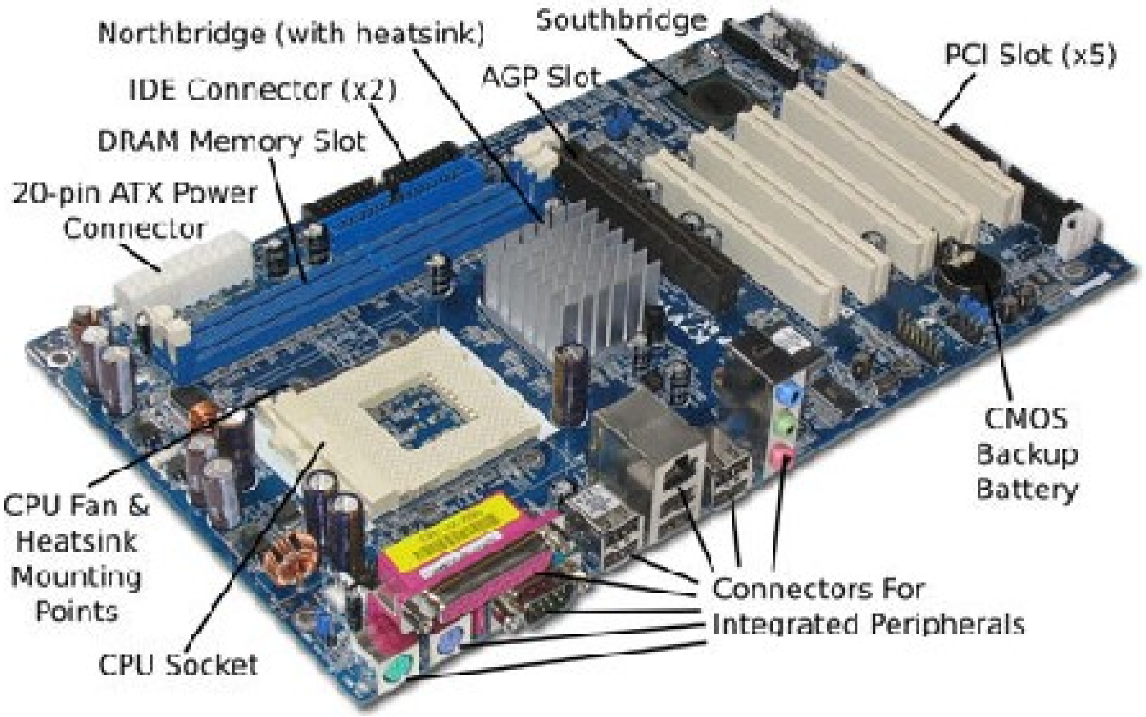


Typical Desktop Architecture

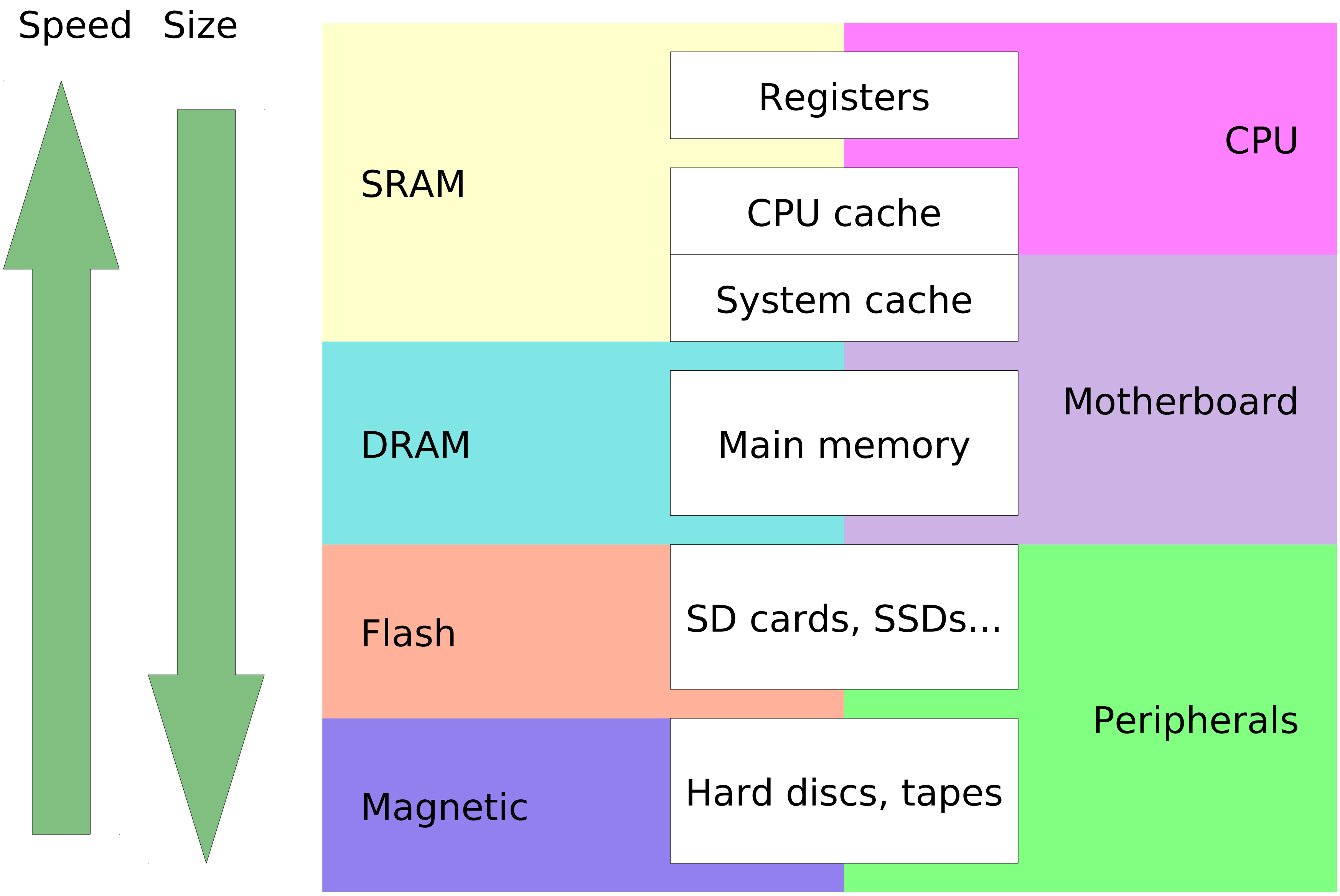


The Motherboard

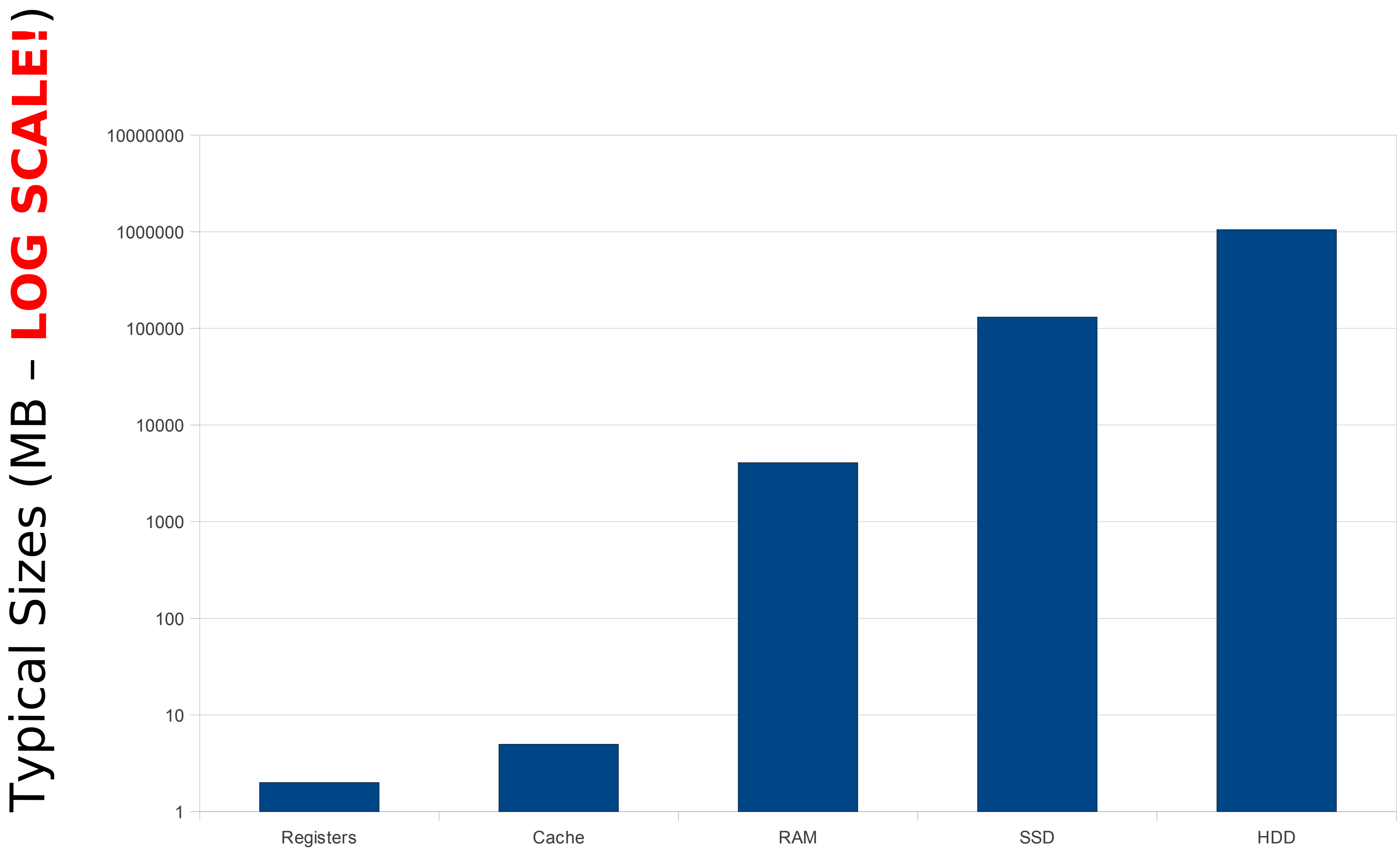
- An evolution of the circuitry between the CPU and memory to include general purpose buses (and later to integrate some peripherals directly!)
- Internal Buses
  - ISA, PCI, PCIe, SATA, AGP
- External buses
  - USB,Firewire, eSATA, PC card



Memory Hierarchy (Typical)

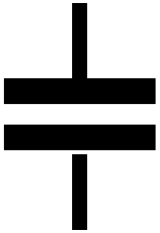


Typical Memory Capacities

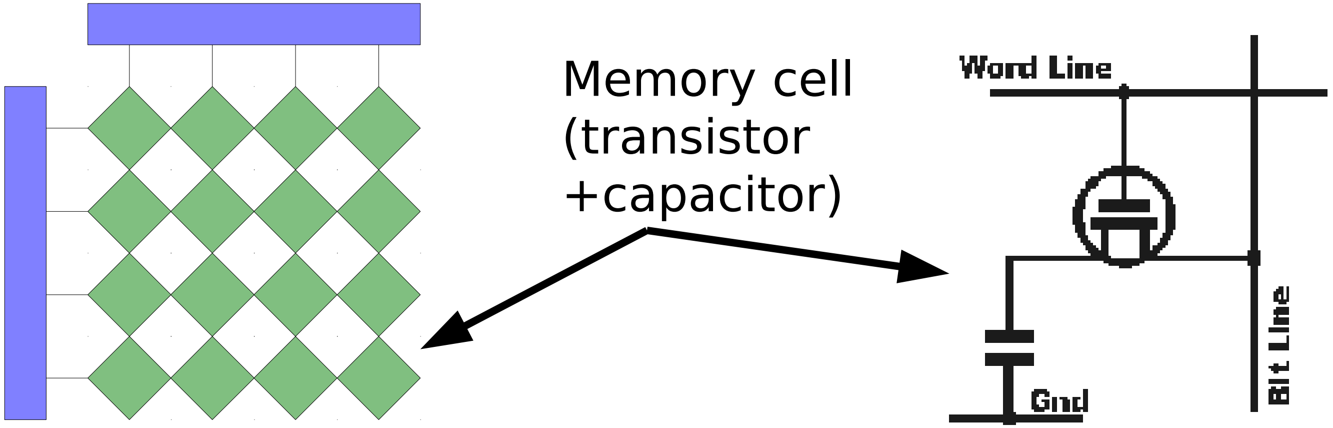


Random Access Memory (RAM)

- The alternative to mercury delay lines is essentially a capacitor. A charged capacitor is a “1” and a discharged capacitor is a ”0”



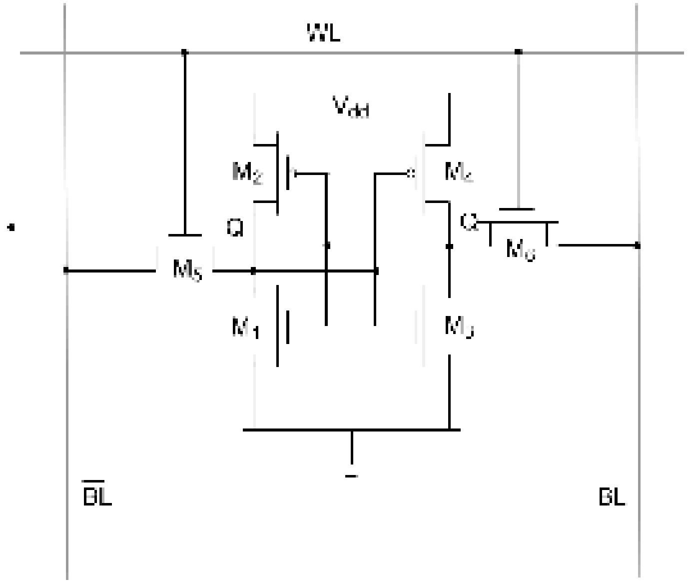
- If we stick a capacitor together with a transistor we create a memory cell. Put lots of cells in a matrix and we can use the transistor to 'activate' a specific cell and ignore all others. In doing so, we can randomly jump around in the data (random access)



- This is Dynamic RAM (DRAM) and it is cheap because each cell is simple
- BUT: capacitors leak charge over time, so a “1” becomes a “0”. Therefore we must <u>refresh</u> the capacitor regularly and this slows everything down plus it drains power...

Static RAM (SRAM)

- We can avoid the need to refresh by using Static RAM (SRAM) cells. These use more electronics (typically 6 transistors per cell) to effectively self-refresh.



SRAM Memory Cell

- This is 8-16x faster than DRAM
- But each cell costs more and takes more space so it's also about 8-16x more costly!
- And both DRAM and SRAM are volatile (lose power = lose data)

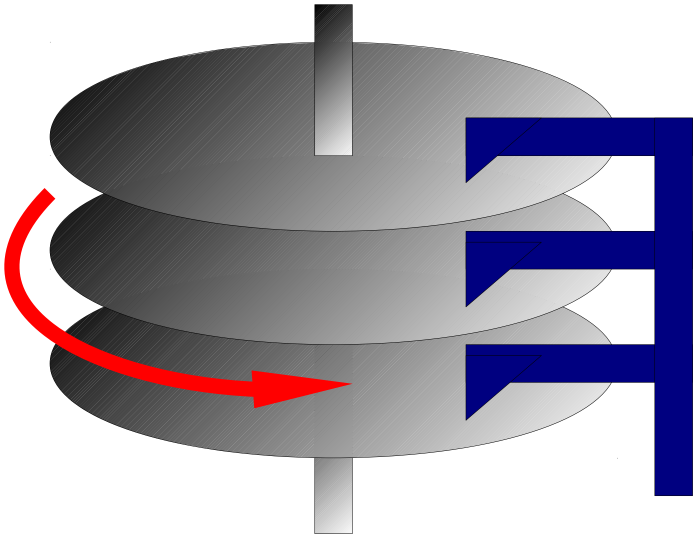
Register Sizes

- Registers are fixed size, super-fast on-chip memory usually made from SRAM.
- When we build the CPU we have to decide how big to make them
  - Bigger registers
    - Allow the CPU to do more per cycle
    - Mean we can have more main RAM (longer addresses can be supported)
    - Too big and we might never use the whole length (waste of electronics)
  - Smaller registers
    - Less electronics (smaller, cooler CPUs)
    - Too small and it takes more cycles to complete simple operations

Flash and SSDs

- Toshiba came up with Flash memory in the 1980s as a non-volatile storage without moving parts
  - 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, small
    - Used in USB flash drives, camera memory and now Solid State Discs.

Magnetic Media (Hard Discs)



- 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 – sequential access, not random access)

Graphics Cards

- Started life as simple 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.
- Todays GCs are based around GPUs with lots of tiny processors (cores) sharing some memory. The notion is one of SIMD – Single Instruction Multiple Data
  - Every instruction is copied to each core, which applies it to a different (set of) pixel(s)
  - Thus we get parallel computation → fast
  - Very useful for scientific computing
  - CPUs better for serial tasks

Memory Manipulation

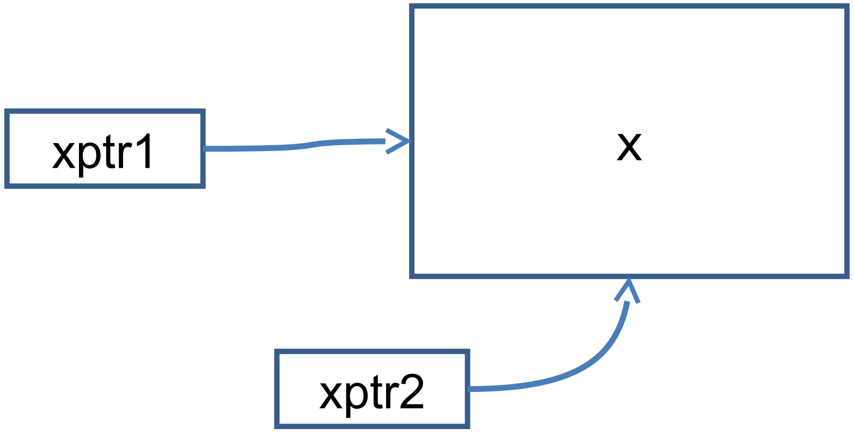
Memory and Pointers

- 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

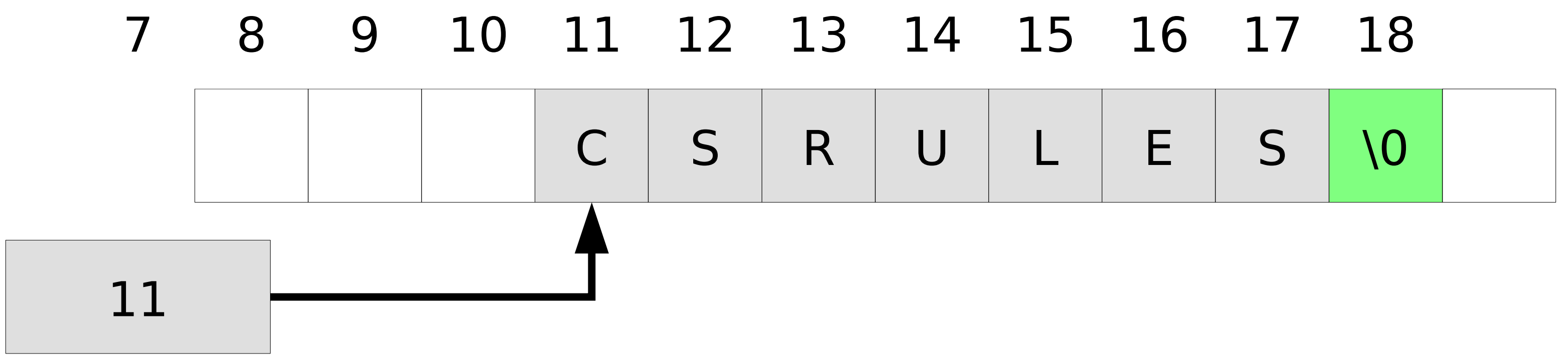
- A pointer is just the memory address of the first memory slot used by the variable
- The pointer type tells the compiler how many slots the whole object uses

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



Example: Representing Strings I

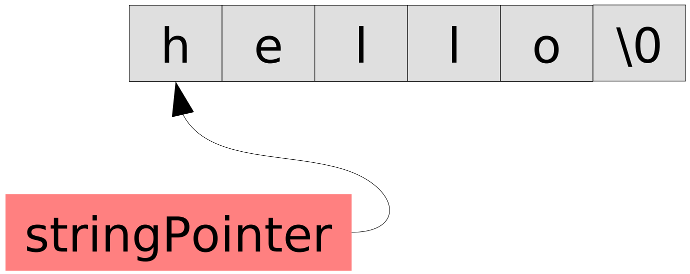
- A single character is fine, but a text string is of variable length – how can we cope with that?
- We simply store the start of the string in memory and require it to finish with a special character (the NULL or terminating character, aka '\0')
- So now we need to be able to store memory addresses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

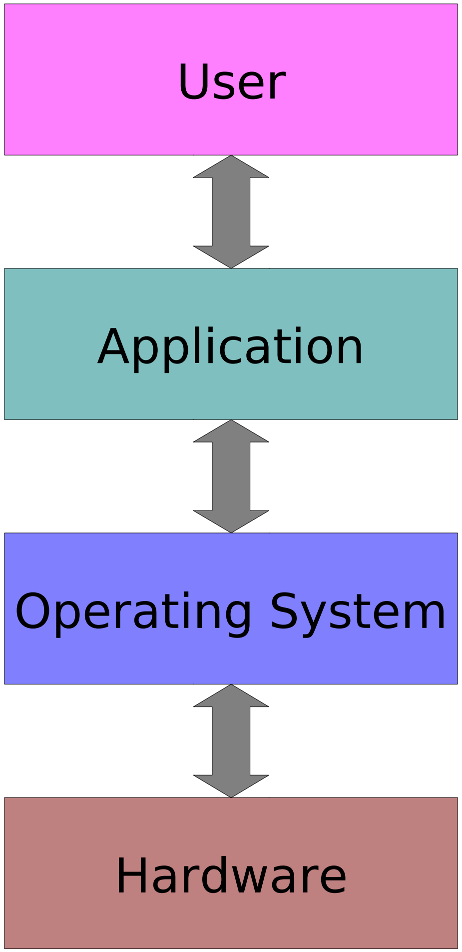


Platforms and Operating Systems
(Software to control your hardware)

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 apps and users
  - Need something to control access to resources...

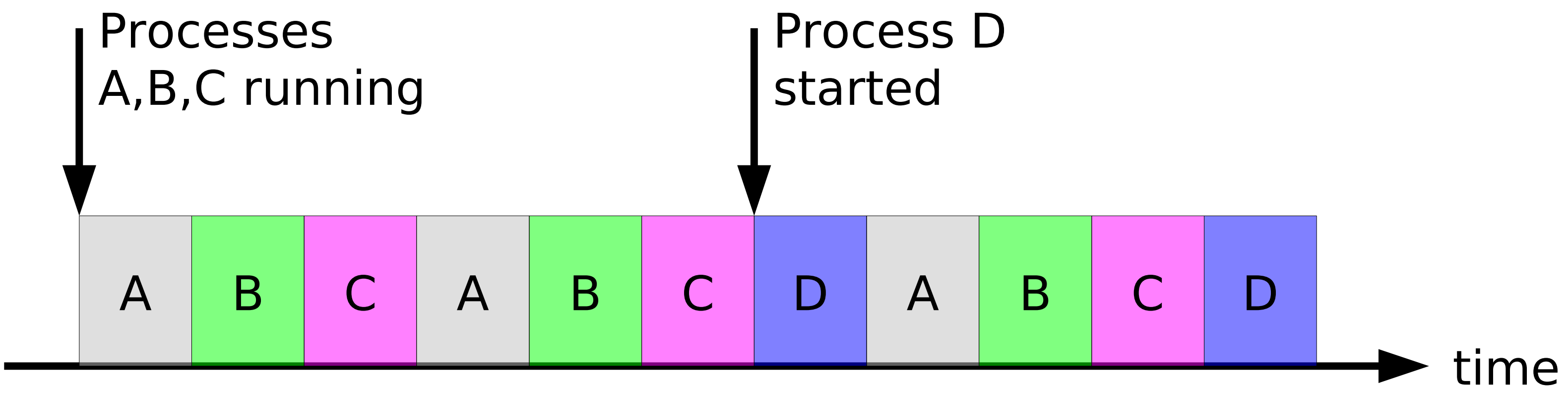
Operating System



- 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

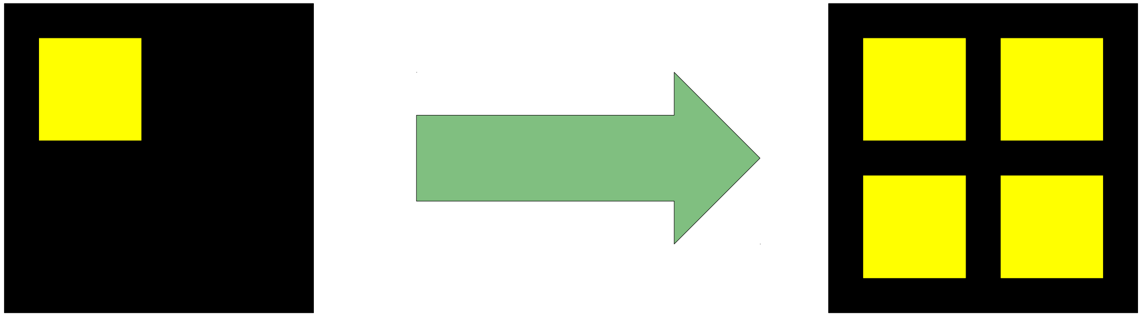
- API = Application Programming Interface
- Software vendors ship their libraries with APIs, which describes only what is need for a programmer to use the library in their own program.
  - The library itself is a black box – shipped in binary form.
- Operating systems are packed with APIs for e.g. window 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
  - Apple/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!

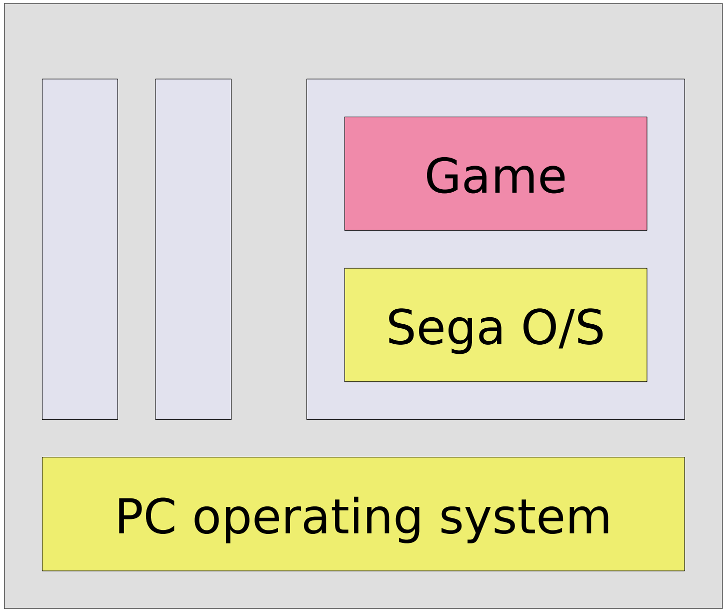


The New Challenge

- 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

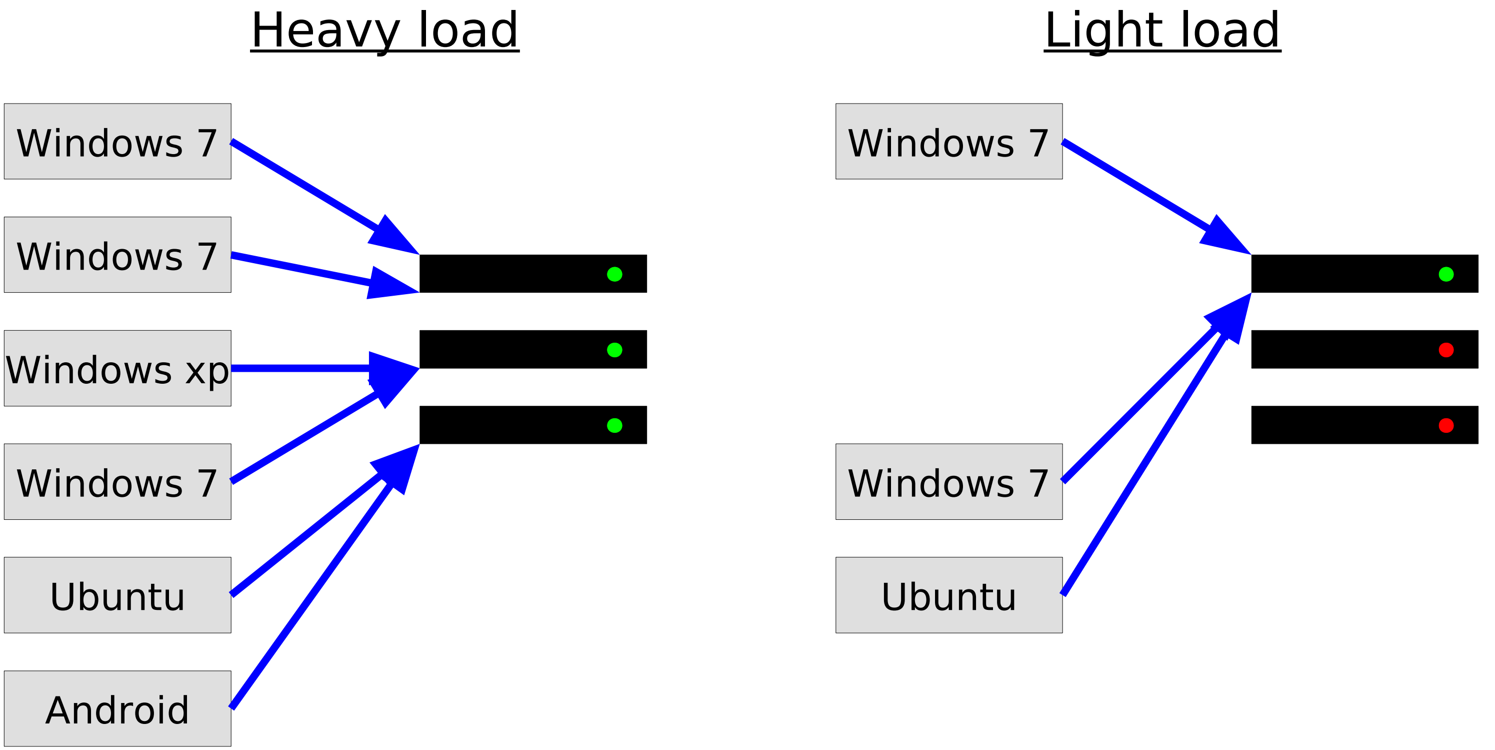
- Go back 20 years and emulators were all the rage: programs on architecture X that simulated architecture Y so that programs for Y could run on X
- Essentially interpreters, except they had to recreate the entire system. So, for example, they had to run the operating system on which to run the program.



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

Virtualisation

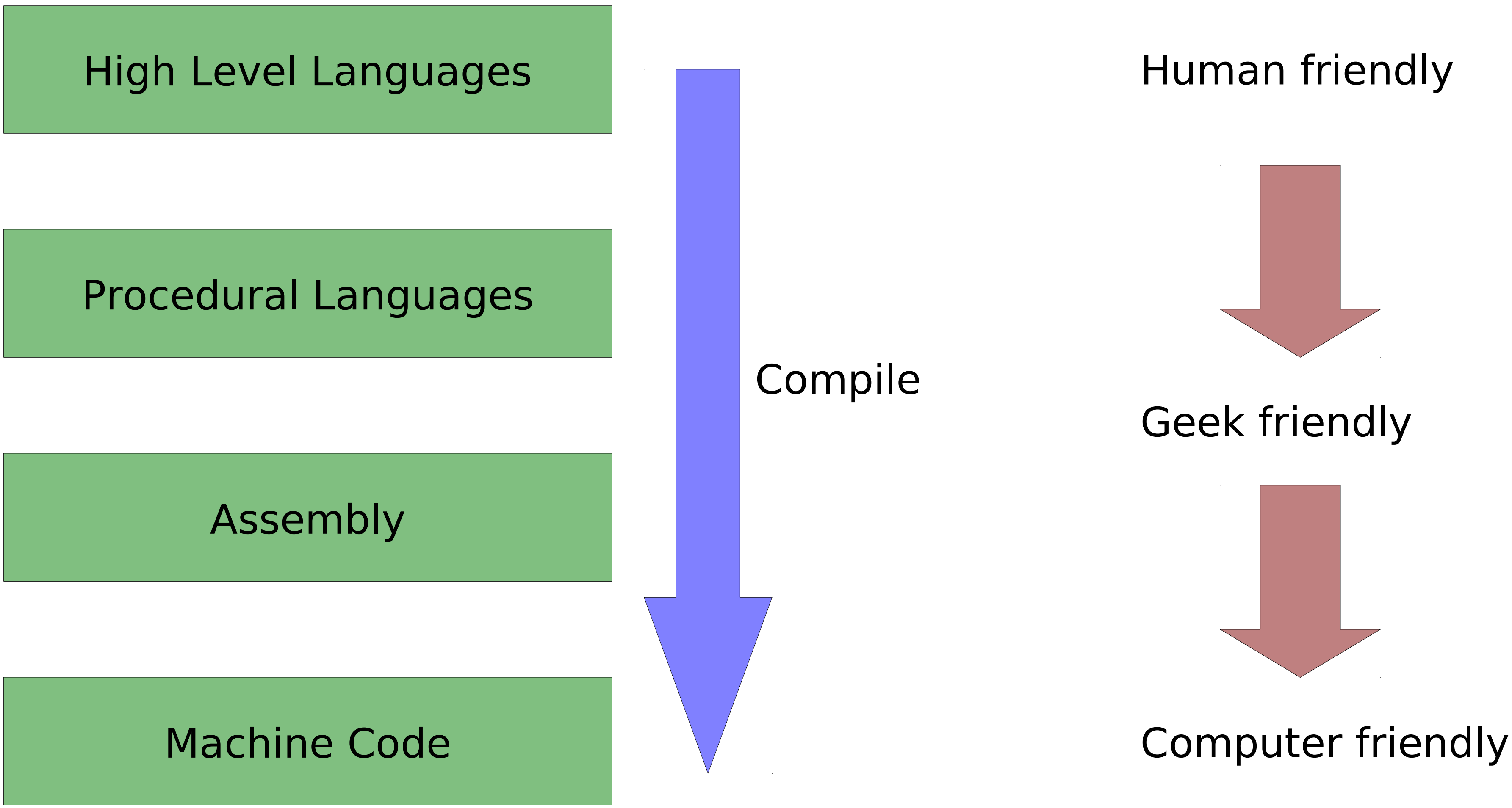
- Virtualisation is the new big thing in business. Essentially the same idea: emulate entire systems on some host server
- But because they are virtual, you can swap them between servers by copying state
- And can dynamically load your server room!



Levels of Abstraction

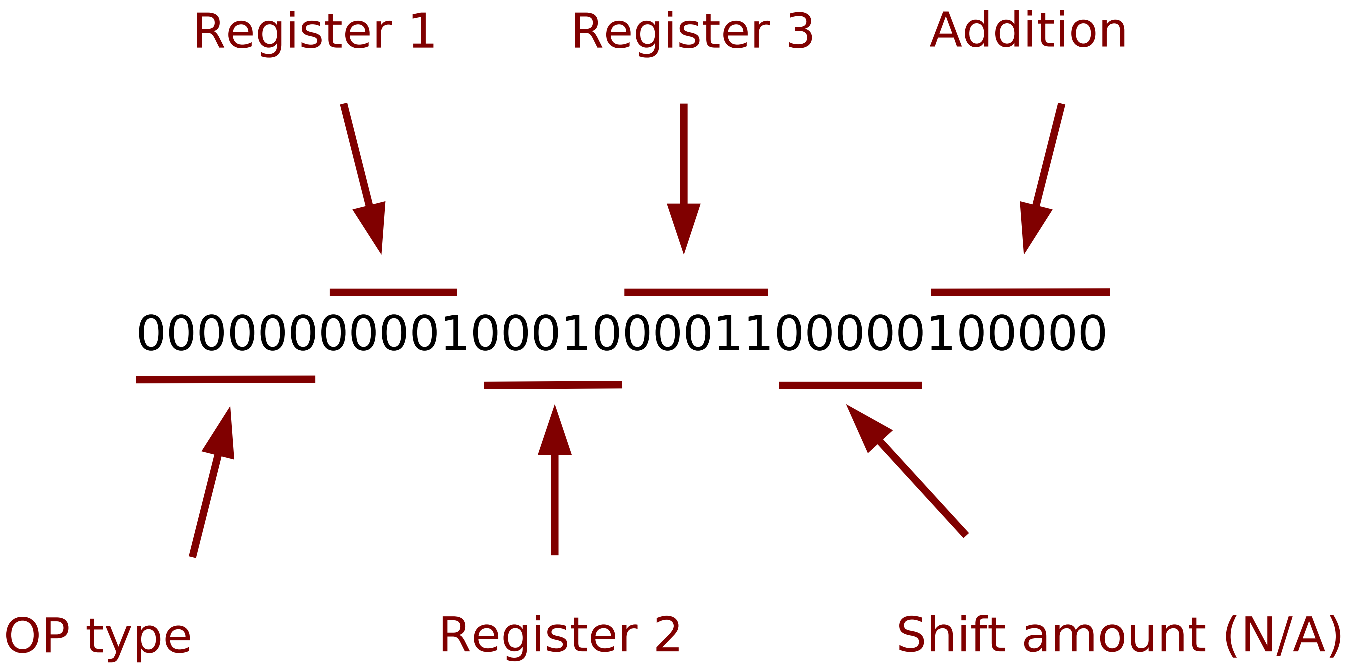
(How humans can program computers)

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:



RISC

- The simplest way to create a CPU is to have a small number of simple instructions that allow you to do very small unit tasks
  - E.g. load a value to a register, add two registers
  - If you want more complicated things to happen (e.g. multiplication) you use just use multiple instructions
  - This is a **RISC** approach (Reduced Instruction Set arChitecture) and we see it in the ARM CPUs

CISC

- Actually, two problems emerged
  - People were coding at a low level and got sick of having to repeatedly write multiple lines for common tasks
  - Programs were large with all the tiny instructions. But memory was limited…
- Obvious soln: add "composite" instructions to the CPU that carry out multiple RISC instructions for us
  - This is a **CISC** (Complex Instruction Set arChitecture) and we see it in the Intel chips
  - Instructions can even be variable length

RISC vs CISC

RISC

- Every instruction takes one cycle
- Smaller, simpler CPUs
- Lower power consumption
- Fixed length instructions

CISC

- Multiple cycles per instruction
- Smaller programs
- Hotter, complex CPUs
- Variable length instructions

RISC vs CISC

- CISC has traditionally dominated (for backwards compatibility and political reasons) e.g. Intel PCs
- RISC was the route taken by Acorn, and resulted in the ARM processors e.g. smartphones

Practicalities: Microcode

- An easy way to create a CISC processor is to use a RISC processor at the core, and then have an interface layer that converts each composite instruction to a set of RISC instructions
- It was quickly realised that this interface could be in software if the hardware could execute it very fast
  - Very high speed ROM on the CPU to store this
- This has led to the notion of **microcode**, which specifies the translations.
  - Microcode is set by the CPU manufacturer and not something the end-user or developer is expected to fiddle with!

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**

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
```

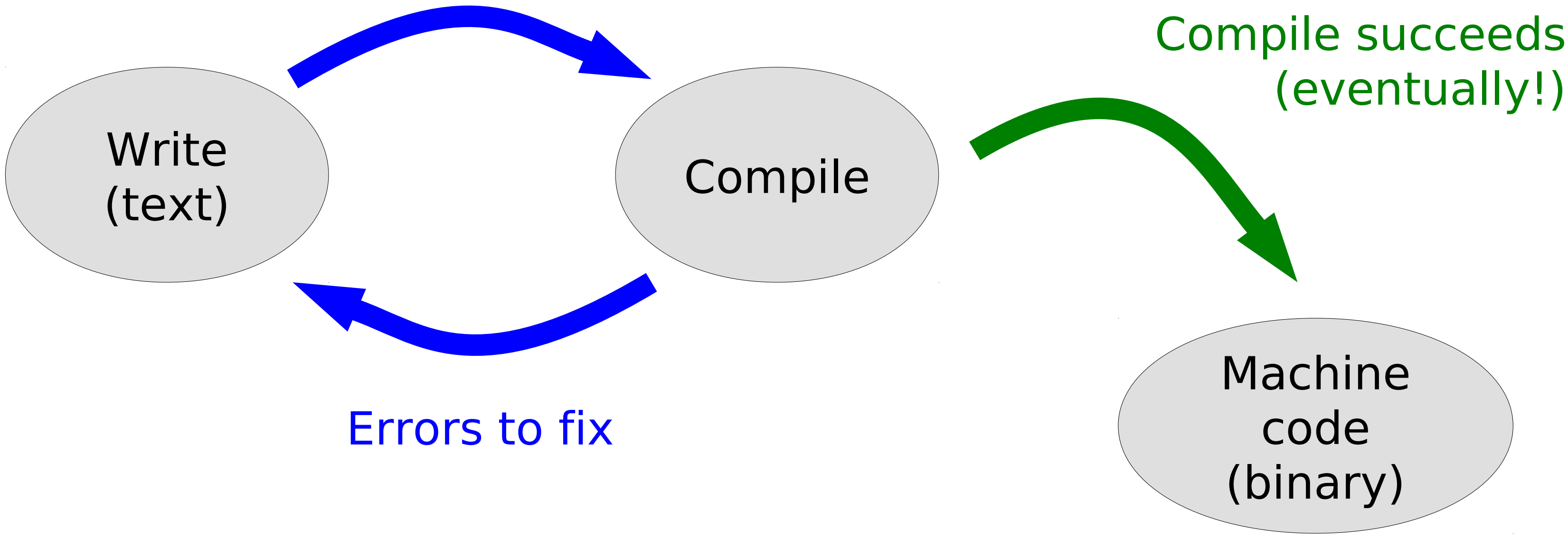
- Produces small, efficient machine code when **assembled**
- Almost as tedious to write as machine code
- Becoming a specialised skill...
- Ends up being architecture-specific if you want the most efficient results :-(

Instruction Set Architectures (ISAs)

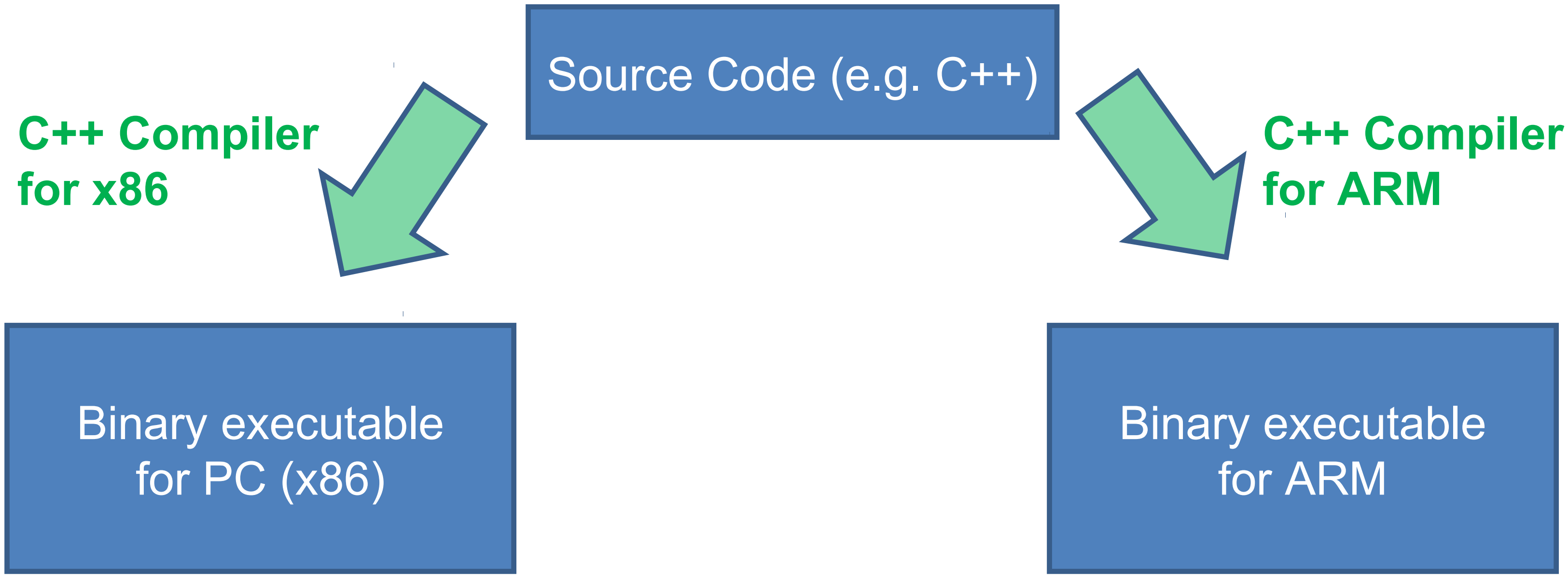
- When you create a CPU, you decide the instructions it will work with. A given the set is an (instruction set) architecture (ISA)
- Initially all architectures different → no software compatibility
- A few have emerged as de-facto standards
  - x86 – the intel line of CPUs right back to the 1980s (a.k.a PC arch)
  - PowerPC – Apple/IBM/Motorola's RISC competitor to x86
  - ARM – RISC-based ISA based on Acorn's processors
  - MIPS – RISC-based ISA used in embedded designs

Compilers

- A compiler is just a software program that converts the high-level code to machine code for a particular ISA (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!



Handling Architectures



Interpreters

- The final binary 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

Types of Languages

- Declarative - specify what to do, not how to do it. i.e.
  - E.g. HTML describes what should appear on a web page, and not how it should be drawn to the screen
  - E.g. SQL statements such as “select * from table” tell a program to get information from a database, but not how to do so
- Imperative – specify both what and how
  - E.g. “double x“ might be a declarative instruction that you want the variable x doubled somehow. Imperatively we could have “x=x*2” or “x=x+x”

Functional vs Imperative

- **Functional** languages are a subset of declarative languages
  - You will be learning a functional language this term: ML
  - This may look like the imperative languages you have seen before, but it is a little different
  - Specifically, functions can't have side-effects. i.e. the output can <u>only</u> depend on the inputs
  - Example of side-effect:

```
t=3
f(y) = y*t
f(2)  ←6
t=1
f(2) ←3
```

Functional vs Imperative

- We'll look more closely at this when you do the Object-Oriented Programming (OOP) course next term
- For now, just appreciate that the new language you're about to meet has some advantages
  - Fewer opportunities for error
  - Closer to maths
  - All of you start at the same level

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