# Computer Fundamentals
# Lecture 2

Dr Robert Harle

Michaelmas 2013

# Today's Topics

- Brief History of Computers
- Stored Program Model
- Fetch-Execute Cycle, registers, ALU etc
- Notion of Compilers and Interpreters

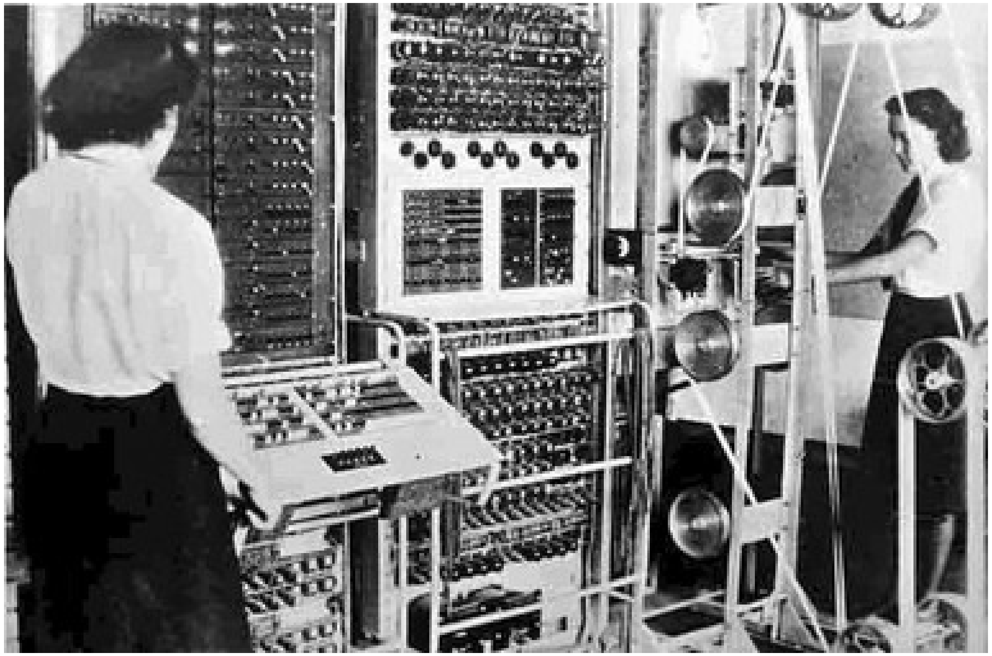
# Turing Machines

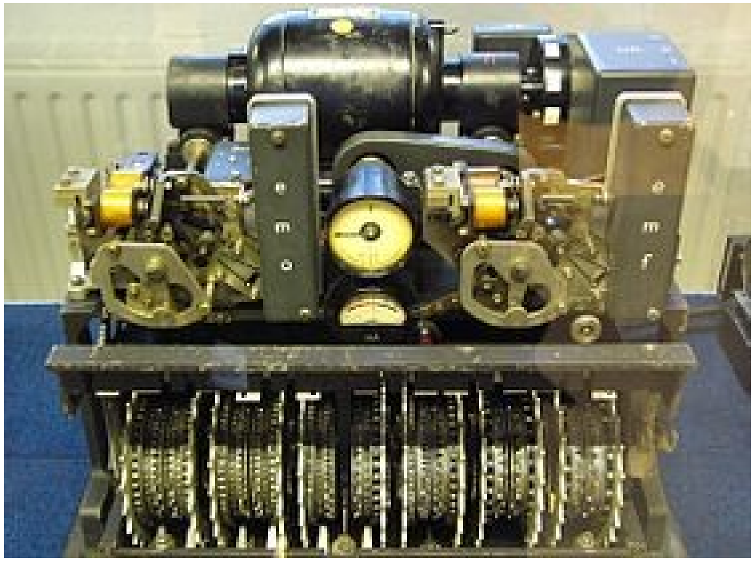
- Inspired by the typewriter (!), Alan Turing (King's) created a theoretical model of a computing machine in the 1930s. He broke the machine into:
  - A tape – infinitely long, broken up into cells, each with a symbol on them
  - A head – that could somehow read and write the current cell
  - An action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

# Colussus

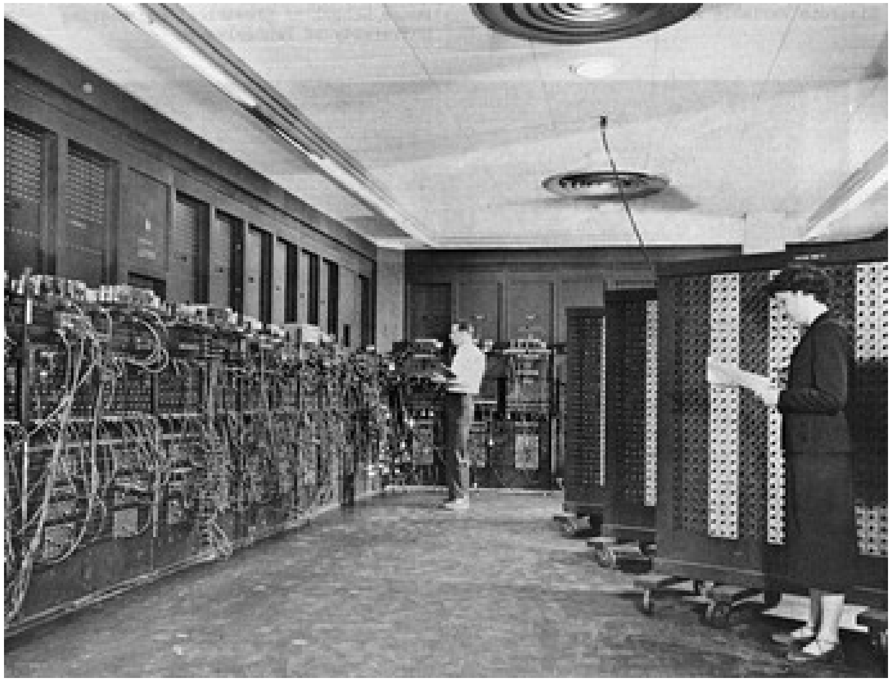




- 1944, Bletchley park
- Designed to break the German Lorenz SZ40/42 encryption machine
- Fed in encrypted messages via paper tape. Colussus then simulated the positions of the Lorenz wheels until it found a match with a high probability
- No internal program – programmed by setting switches and patching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

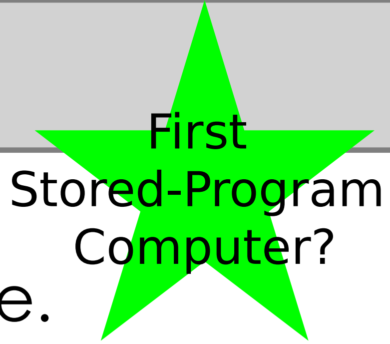
# ENIAC

- 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…

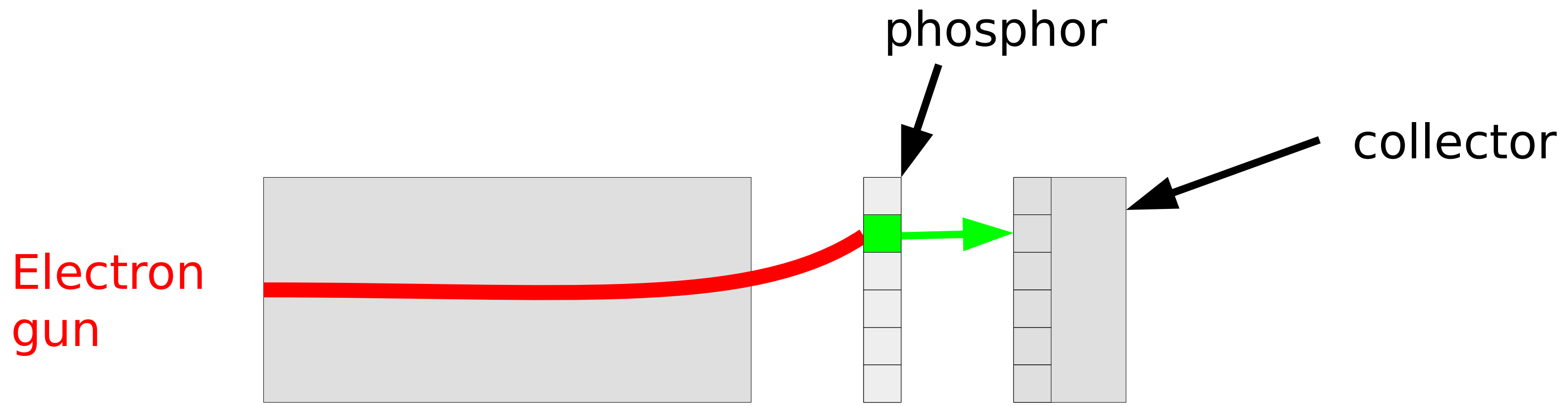


- Next step was to read in the "action table" (aka program) from tape as well as the data
- For this we needed more general purpose memory to store the program, input data and output

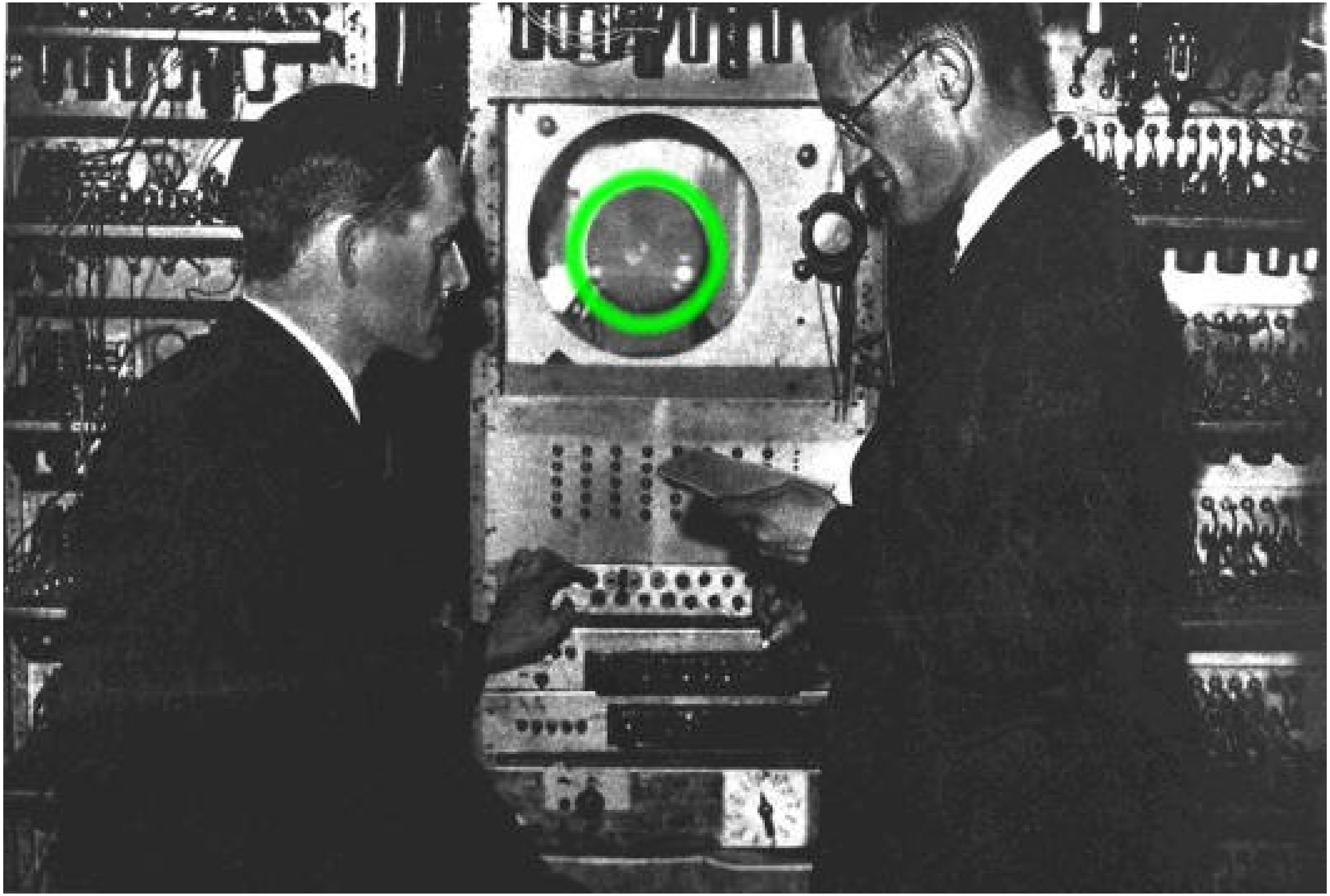
# Manchester Baby




- 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

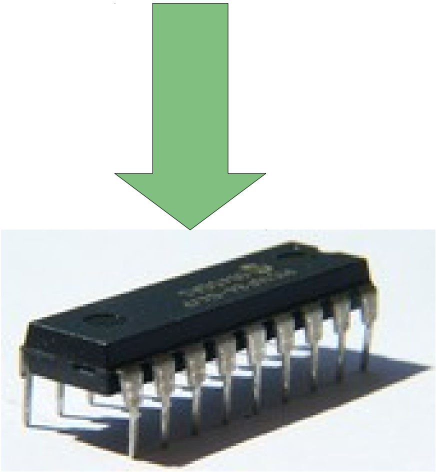
- Electronic Delay Storage Automatic Calculator
- First practical stored-program computer, built **here** by Maurice Wilkes et al.

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....

# 1965-70 Integrated Circuits

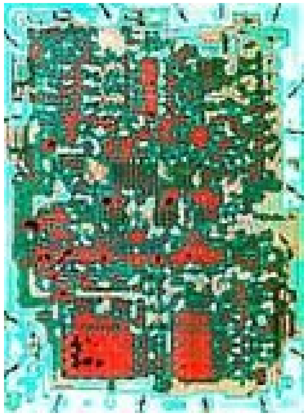


- Semiconductors could replace traditional electronics components → use a slice of semiconductor and 'etch' on a circuit
- End up with an Integrated Circuit (IC) a.k.a a microchip
- Much easier to pack components on an IC, and didn't suffer from the reliability issues of the soldering iron

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

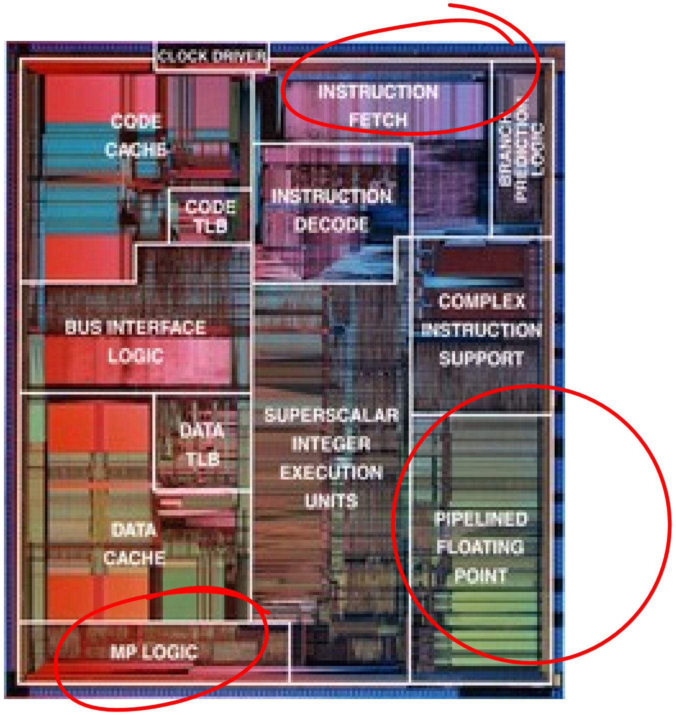
# The Rise of Intel

- 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

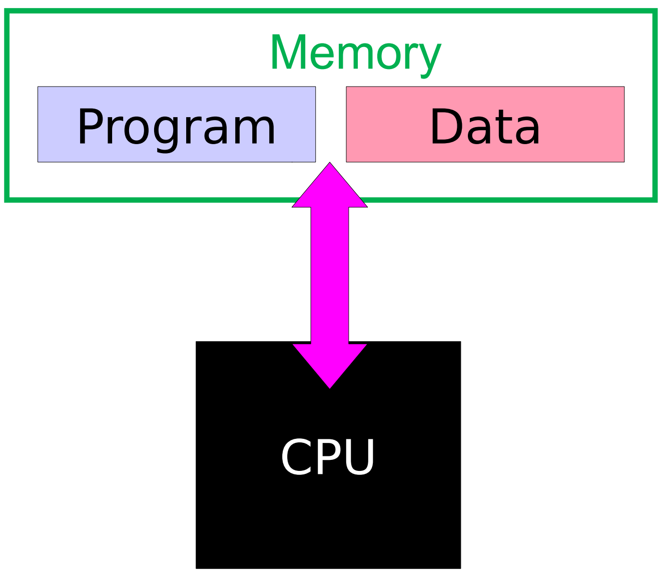
- The 4004 kick-started an industry and lots of competitors emerged
- Intel very savvy and began an "intel inside" branding assault with products like the 386
- Marketing to consumers, not system builders any more

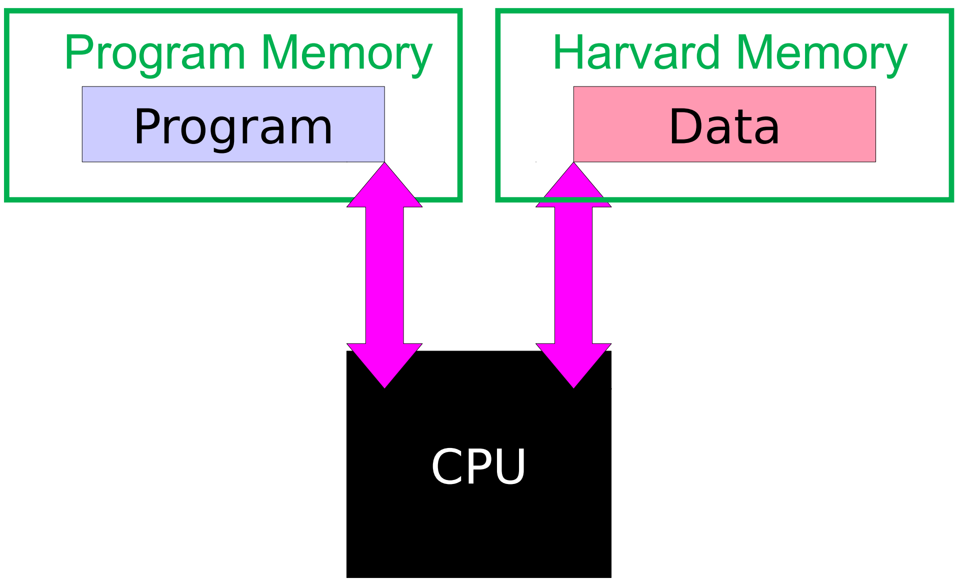
# The CPU in more Detail

# 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). Two options for our program storage in a modern machine




Von-Neumann Architecture



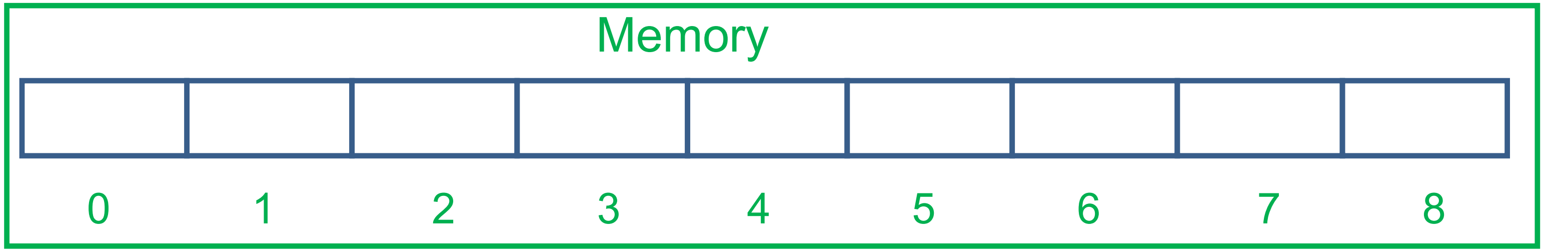

Harvard Architecture

# Storage: Stored-Program Machines

- So where do you store your programs and data?

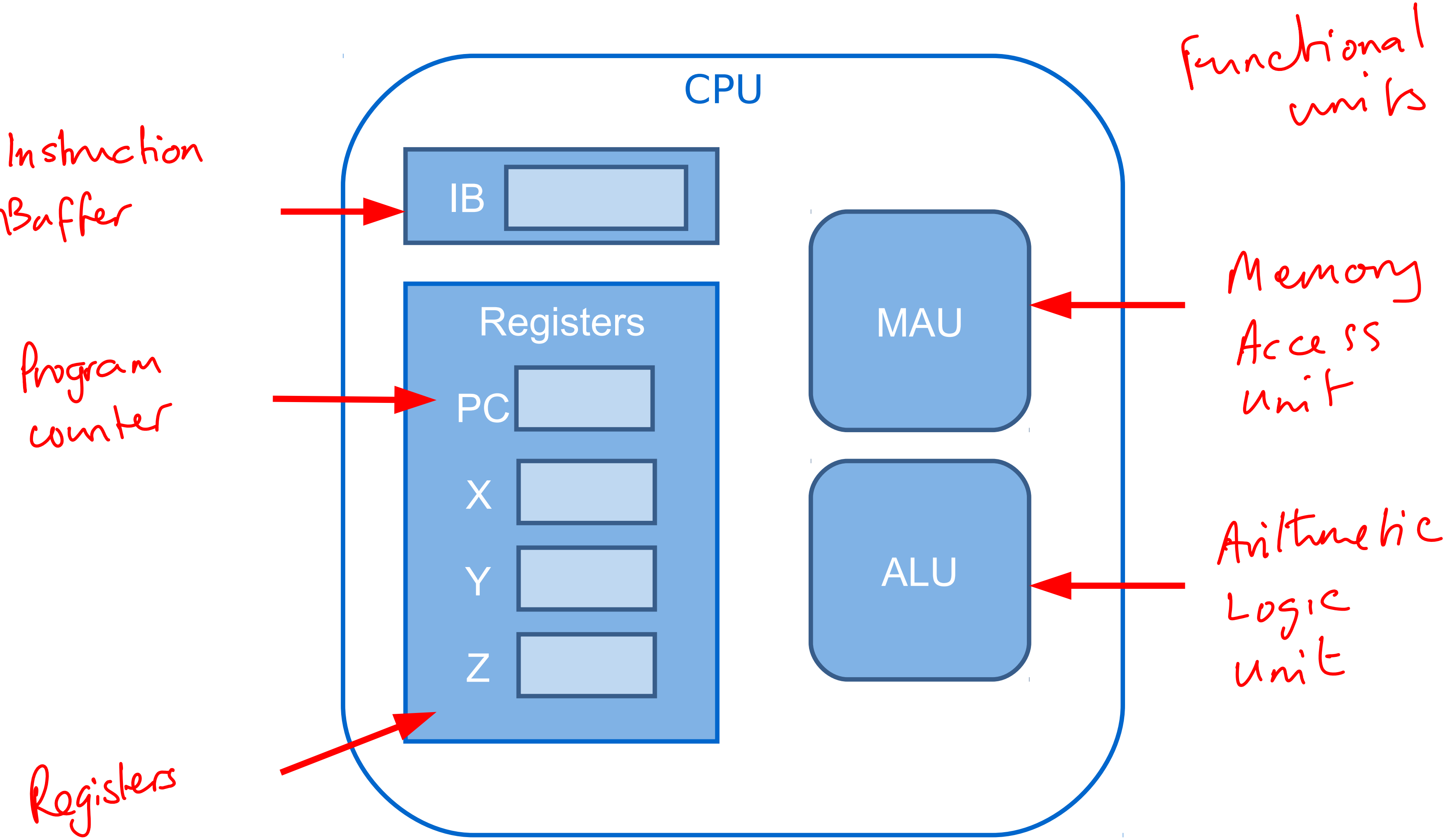
| | Von-Neumann | | Harvard |
|---|---|---|---|
| | **<u>Same</u>** memory for programs and data | | **<u>Separate</u>** memories for programs and data |
| | + Don't have to specify a partition so more efficient memory use | | - Have to decide in advance how much to allocate to each |
| | + Programs 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 |

# 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 big

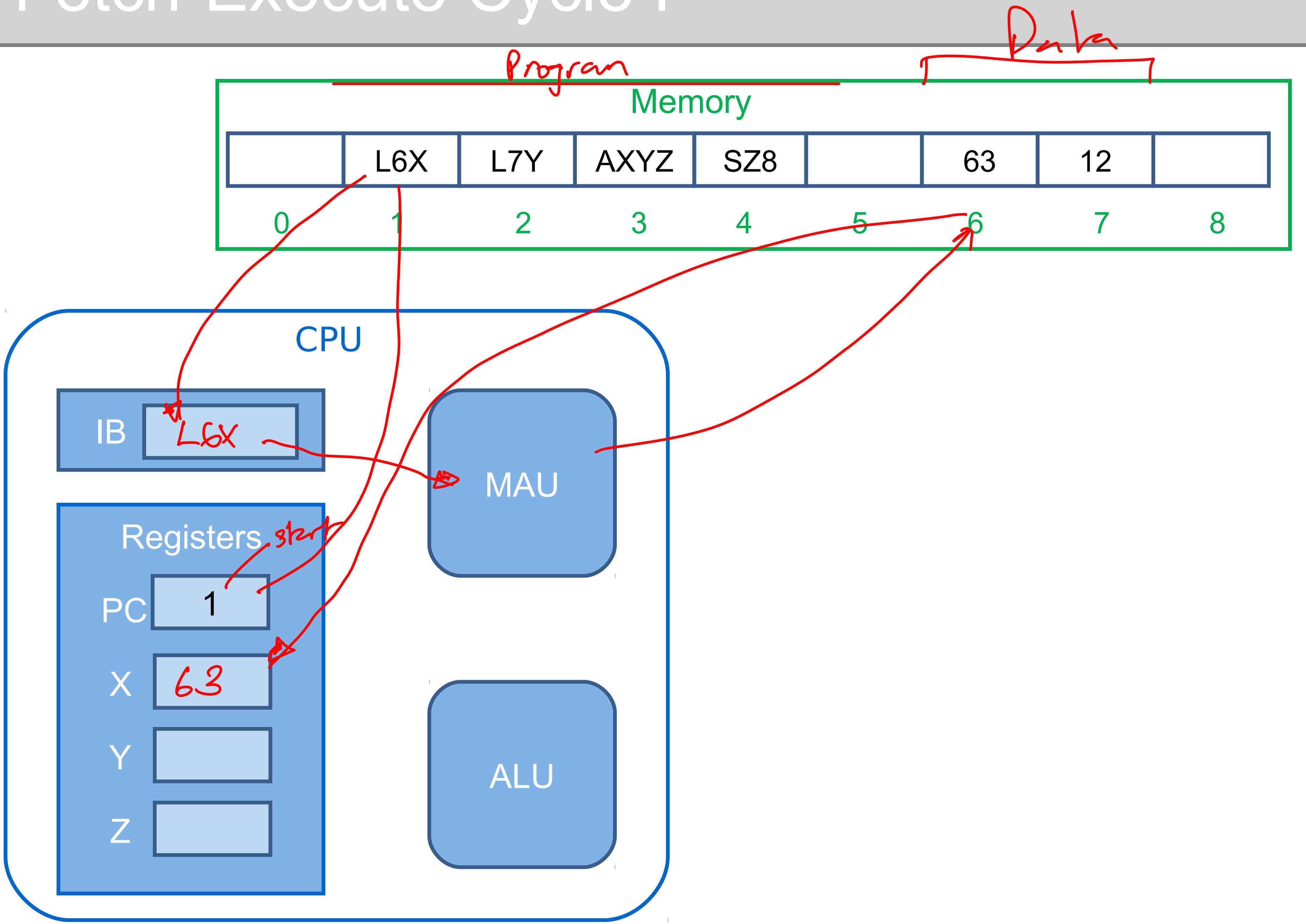
# Simple Model of a CPU



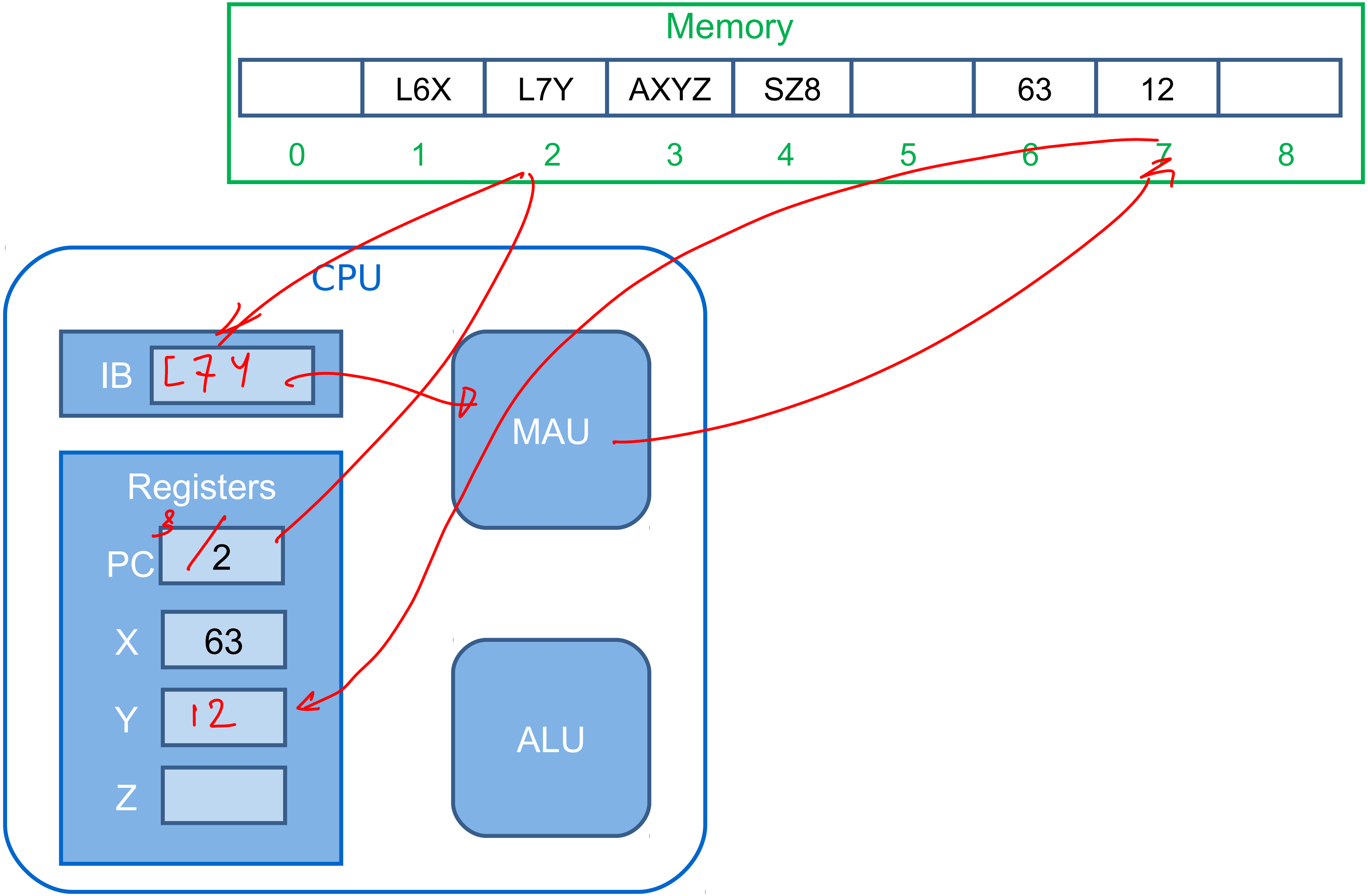
# A Simple Command Set

- A program is just a sequence of instructions. The instructions available depend on the CPU manufacturer
- We will make up some very simple instruction labels
  - **L**IJ: **Load** value at memory address I into register J
  - **A**IJK: **Add** register I to J and put the result in K
  - **S**IJ: **Store** register I in memory address J

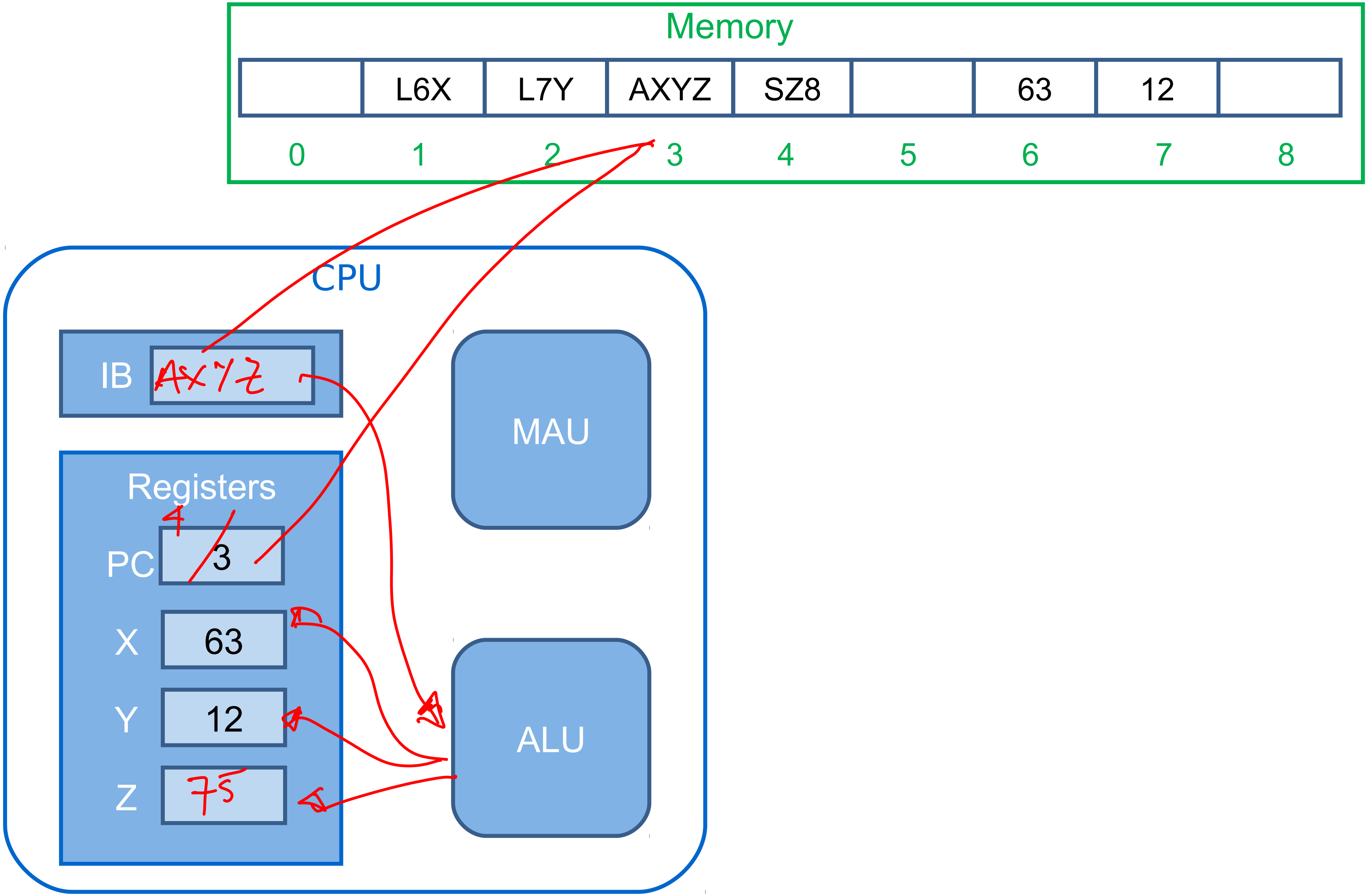
# Fetch-Execute Cycle I



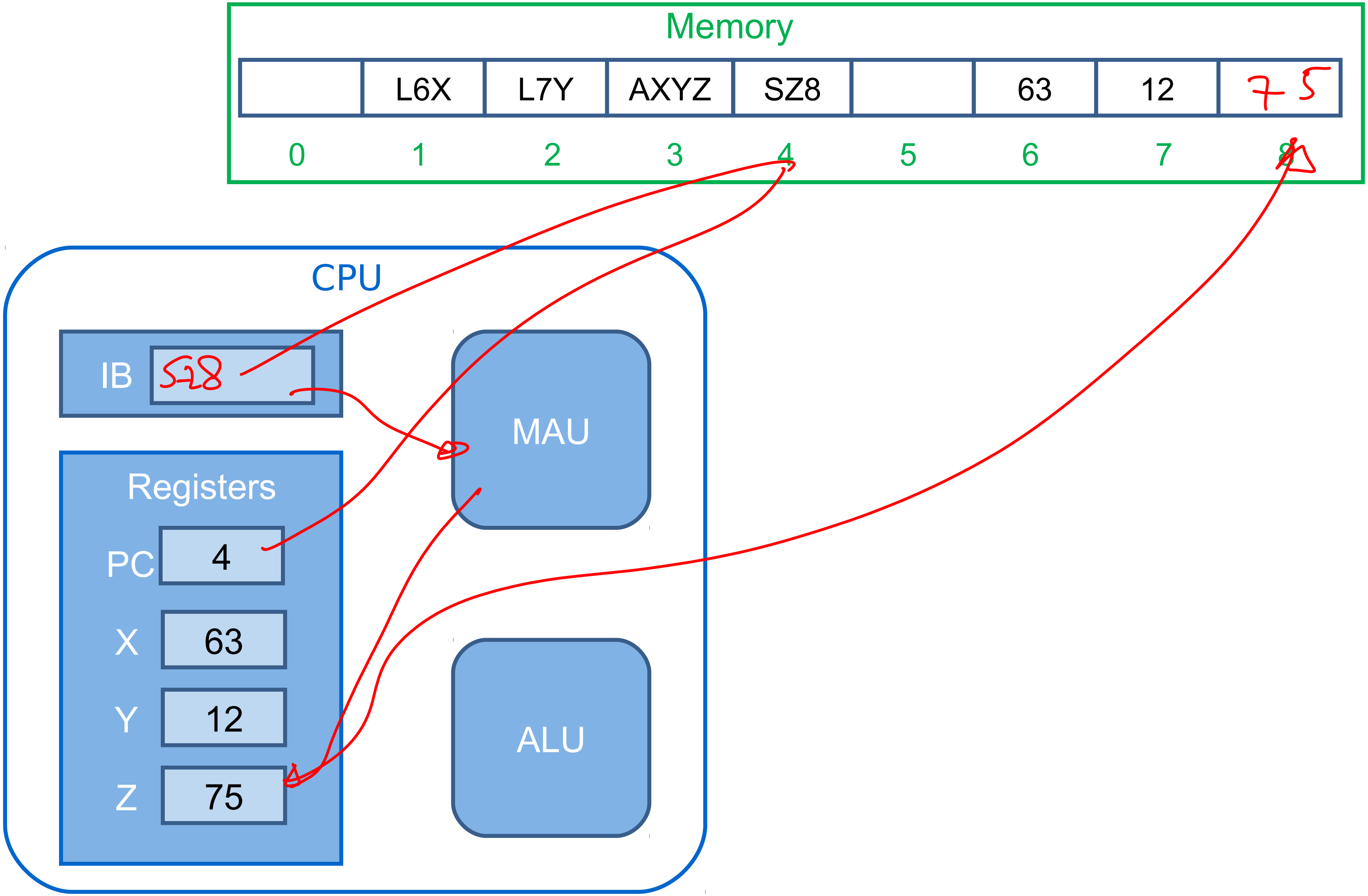
# Fetch-Execute Cycle II



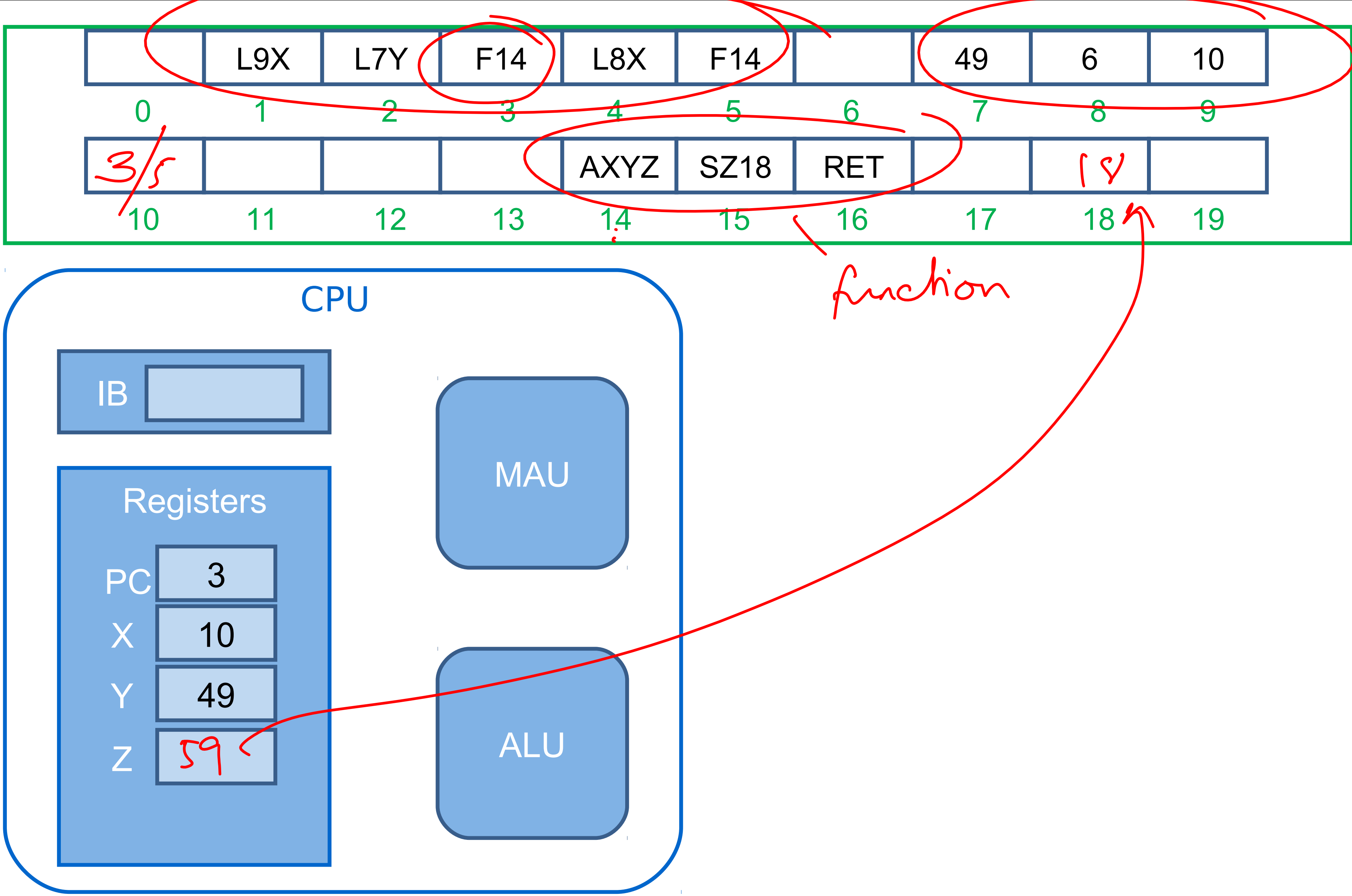
# Fetch-Execute Cycle III



# Fetch-Execute Cycle IV



# Functions


Prog
Data
L9X
L7Y
F14
L8X
F14
49
6
10
0
1
2
3
4
5
6
7
8
9
3/5
AXYZ
SZ18
RET
18
10
11
12
13
14
15
16
17
18
19
function
CPU
IB
MAU
Registers
PC
3
X
10
Y
49
Z
59
ALU

# Instruction Sets

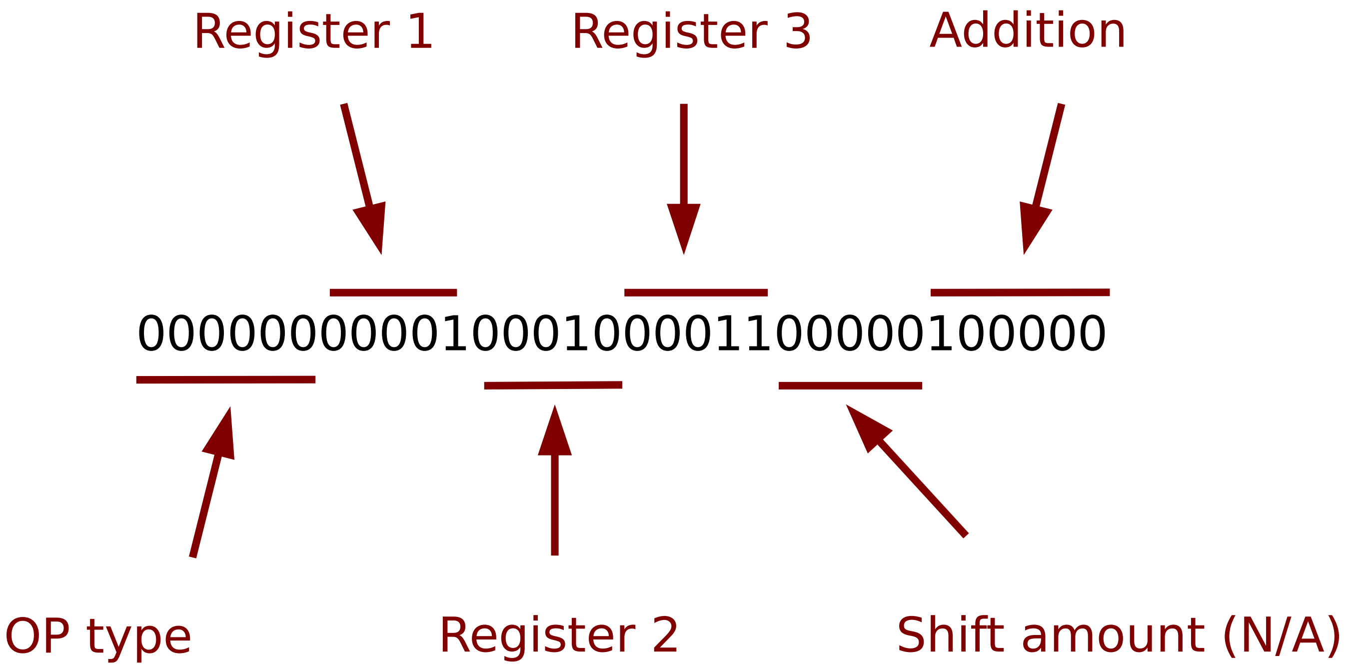
- The list of instructions a CPU supports is its Instruction Set Architecture (ISA)
  - Initially all used different instructions but there is clearly an advantage to using the same instruction sets
  - Intel's x86 set is a de-facto standard for PCs
  - ARM's v6 and v7 specifications are used for lower power applications (phones etc)

# Writing Programs

- Computers don't store text instructions like L6X, but rather a binary code for each instruction
- Called machine code

# 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:



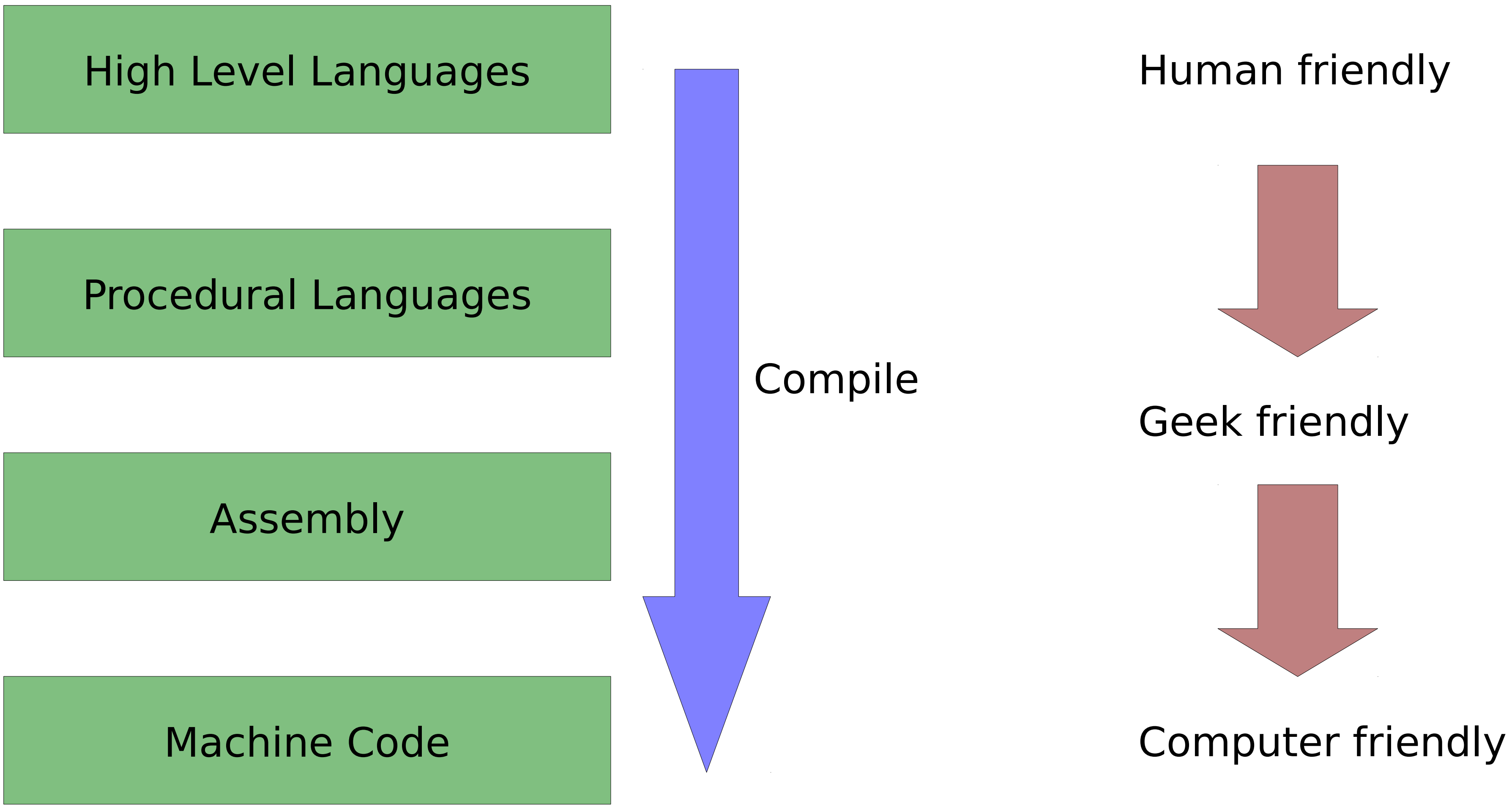
# Assembly

- 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
```

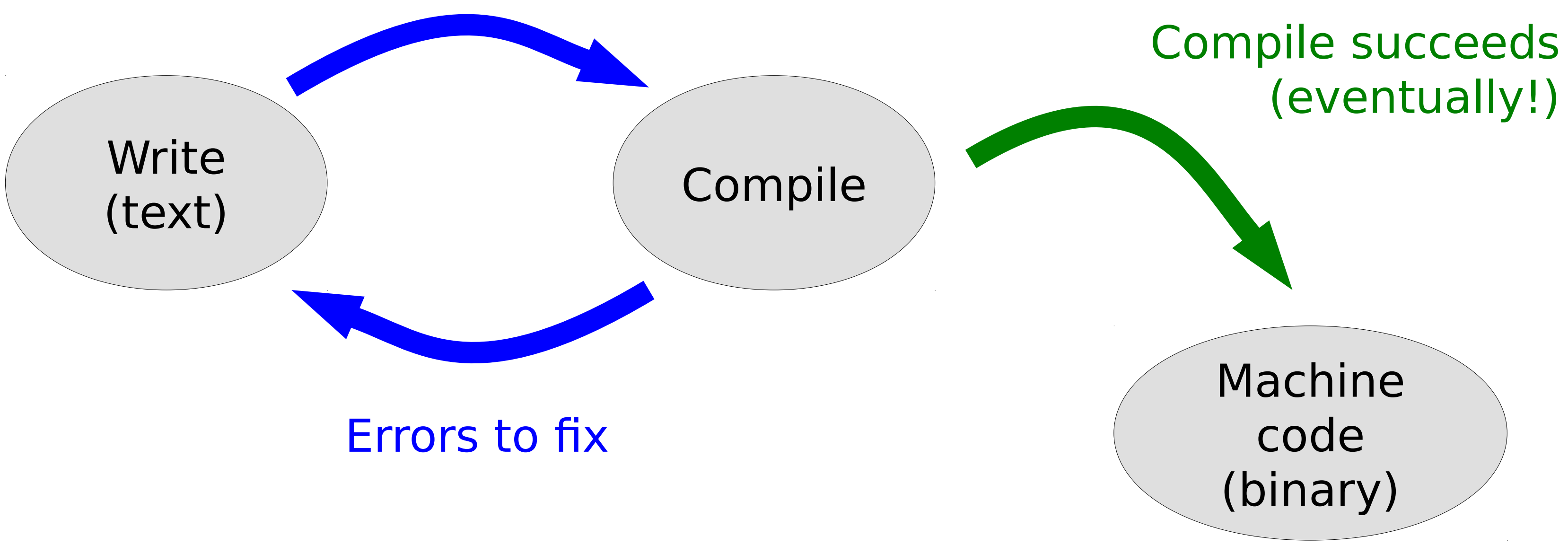
- Produces small, efficient machine code when **assembled**
- Almost as tedious to write as machine code
- Becoming a specialised skill...
- Ends up being architecture-specific if you want the most efficient results :-(

# Levels of Abstraction for Programming

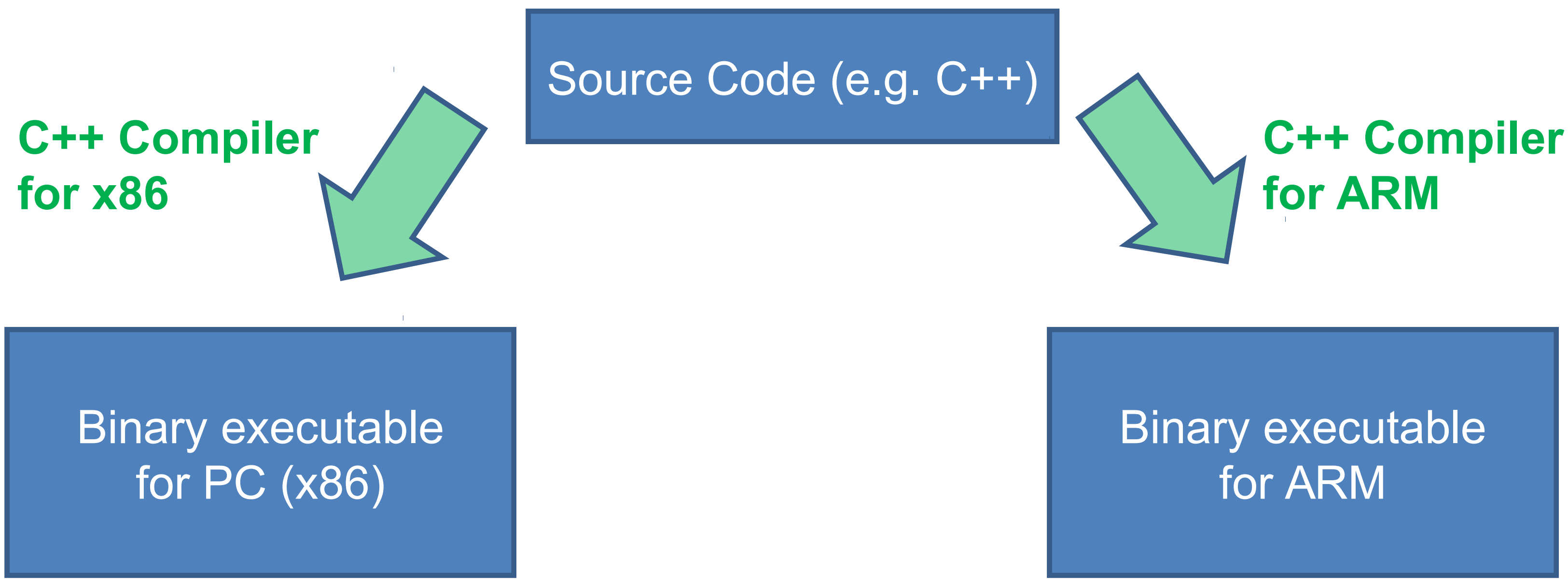


# Compilers

- A compiler is just a software program that converts high-level code to machine code for a particular archistecture (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 |