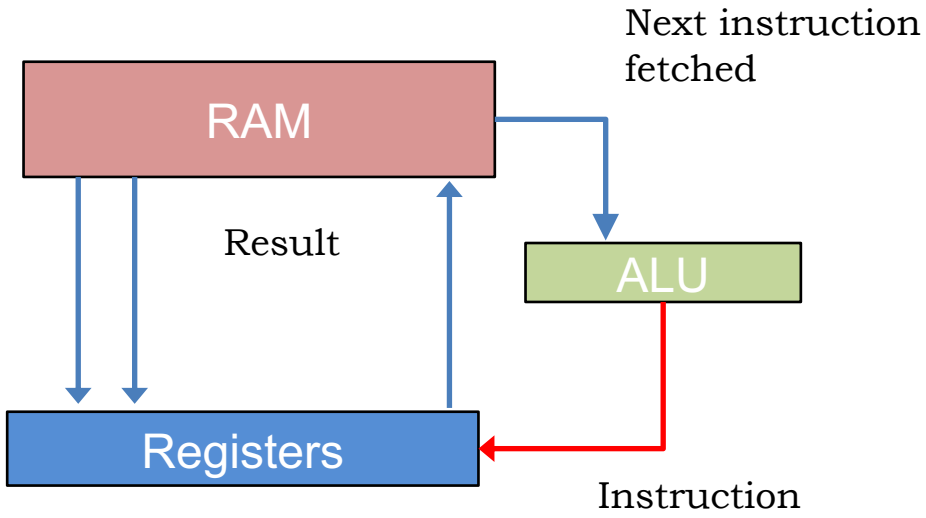
Pipelining: the key to performance

How a computer works in theory

Operands fetched & placed in the registers
Instruction fetched from memory
Operation on registers
Result stored in RAM

A system working like that would be 1000s slower

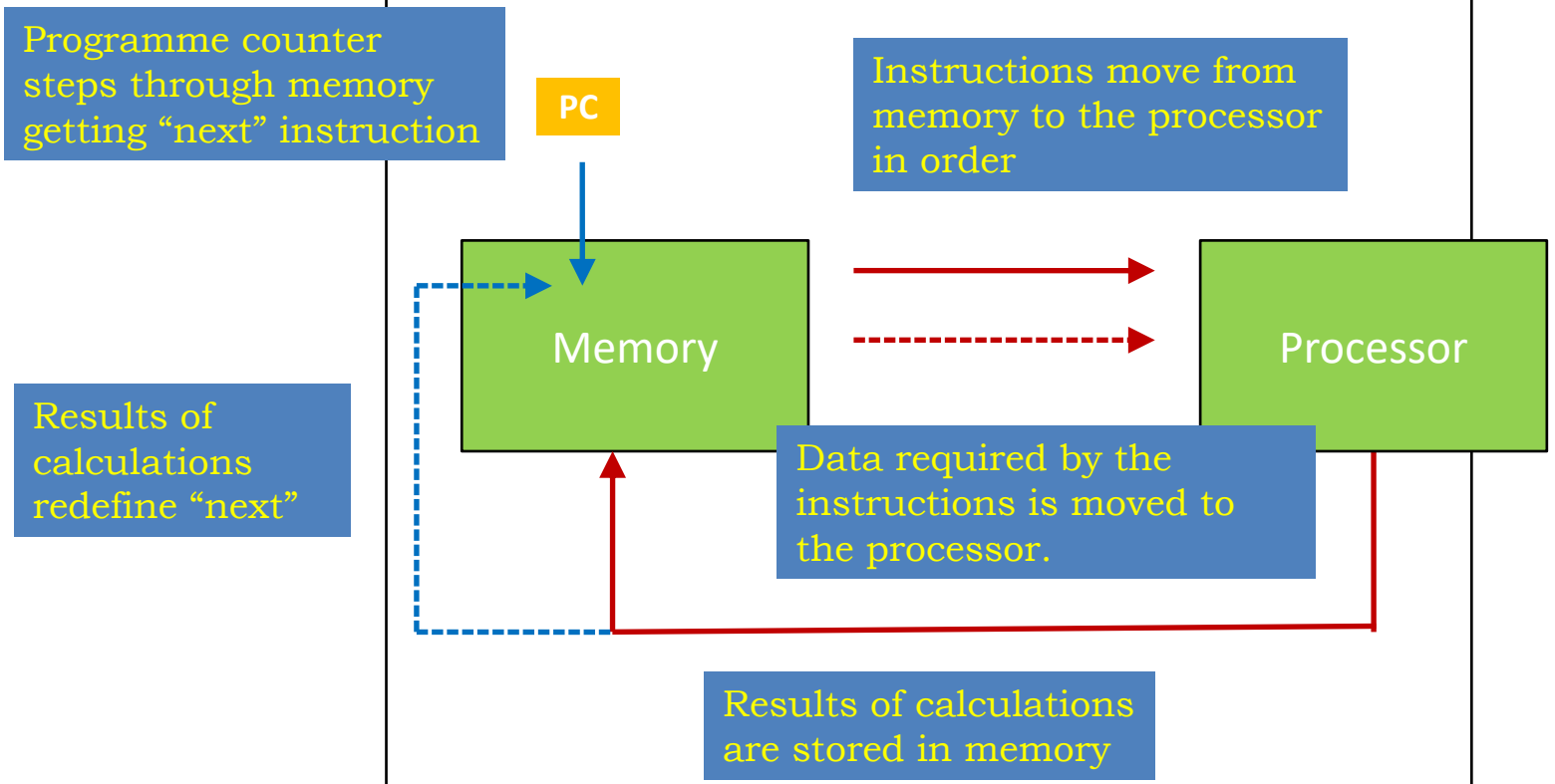
The optimisations for a real system, make it harder to write parallel computations – and we need to understand them to understand parallel computations.



How to increase computer performance

We want to do as much work as possible in a given time.

Seems simple; drive the clock faster and complete more instructions per second.
Increase in clock speed 1940-2000 was certainly useful but from the very early days the clock speed was **never** the determining factor in processor performance.



In fact this is gross simplification
The PC is pointing at location
000A000540130500

- The instruction needs to be transferred from that location to the execution register
  - The PC needs to be incremented to the next location 000A000540130504 (an adder is required)
- The instruction needs to be decoded
- The address of any operands needs to be generated.
- The operands required to be transferred to the input of the Arithmetic Logic Unit (ALU)
- The operation will be executed – (even in the case of addition this is in effect many operations
  - . . . . . .
- The value of the PC may be modified – (a branch instruction)
- The result of any arithmetic or logical operation needs to be stored.

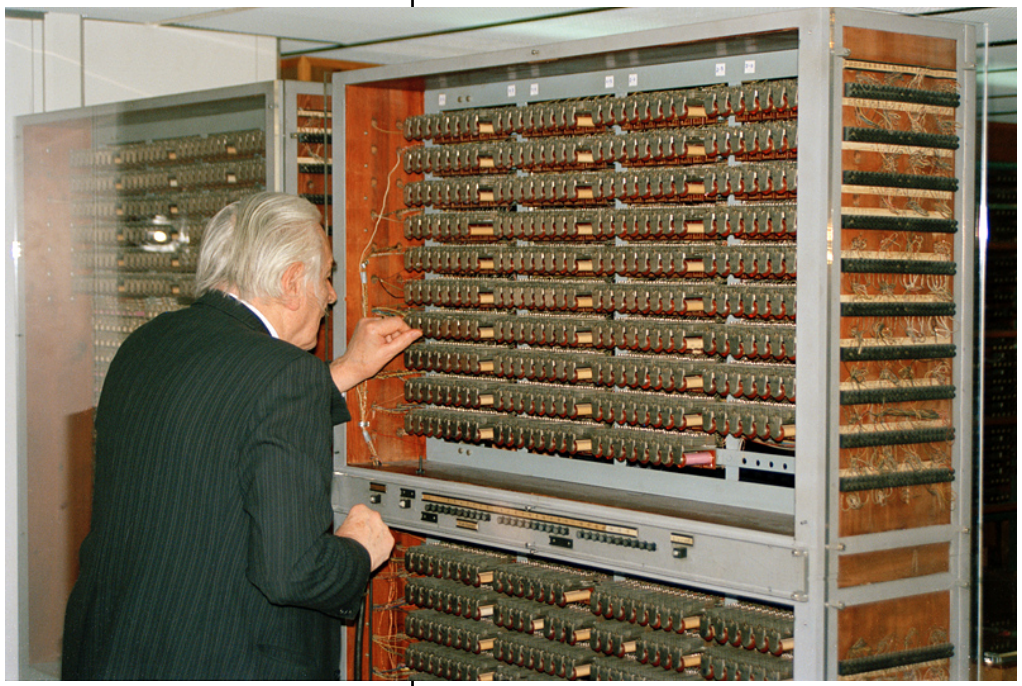
What is thought of as a single action turns out to be a multi-part action

The idea of performing different parts of the instruction in parallel goes back to the Z1 and Z3.

It become popular with the rise of the supercomputer in the late 60's early '70s

It is referred to as a **pipelining** more formally as **Instruction Level Parallelism (ILP)**

The instruction stages are said to move down a pipeline.



Konrad Zuse in front of a replica of the Z3. Deutches Museum Munich

The names for the stages which the instructions is split are referred to as

Fetch
Decode
Execute

The engineering design of the stretch computer
Erich Bloch
IRE-AIEE-ACM '59 (Eastern) Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference

The IBM stretch computer.
IBM's first transistorised supercomputer. Aimed for two orders of magnitude improvement over any existing machine

Abstract
"*The Stretch Computer project was started in order to achieve two orders of magnitude of improvement in performance over the then existing 704. Although this computer, like the 704, is aimed at scientific problems such as reactor design, hydrodynamics problems, partial differential equations etc., its instruction set and organization are such that it can handle with ease data-processing problems normally associated with commercial applications, such as processing of alphanumeric fields, sorting, and decimal arithmetic.*"

The RISC pipeline – as described in Patterson and Hennessey is

Instruction Fetch
Instruction Decode
Execute
Memory Access
Writeback

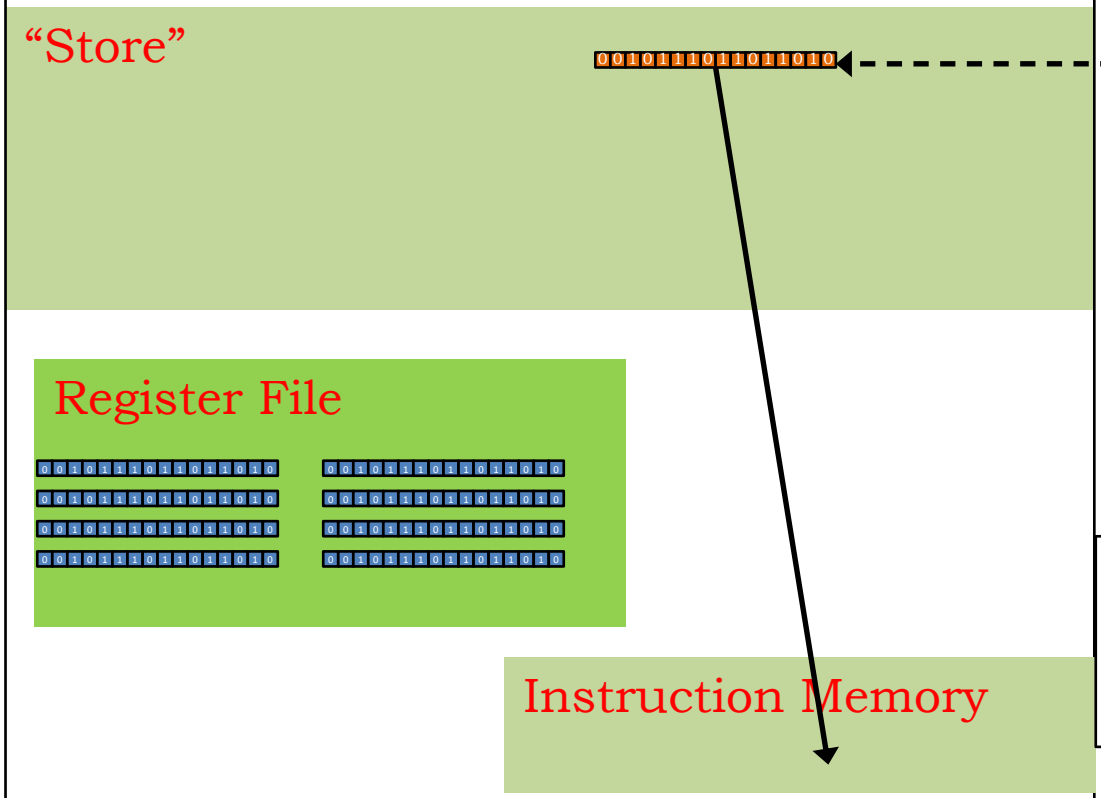
And the standard diagram which displays its operation is

The RISC pipeline – as described in Patterson and Hennessey.

The operands for the CPU to work on must be held in **registers** a special place in the chip called the **register file**. The results must be written out to a register.

More about this later –
for the moment we assume that the data is in the registers.

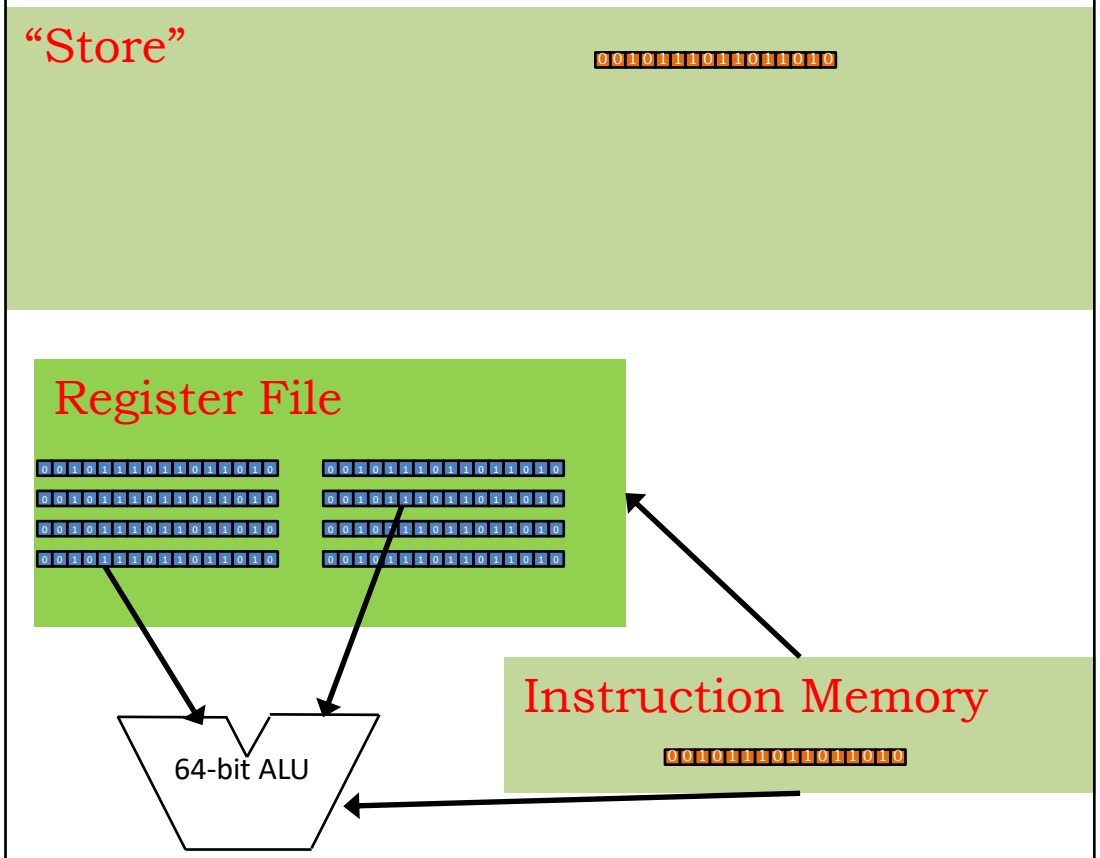
Instruction Fetch



There is an assumption that the instruction is in the instruction cache and can be fetched in 1 cycle

The first stage is to place the next instruction into the location in the CPU where it can be used to drive computation.
Assumes the Programme Counter PC – contains the next address of the next instruction
"Store" – deliberately ambiguous

Instruction Decode



PC will need to be updated

Suppose the instruction is
add $R3, $R4, $R5

For a modern computer it is normal that arithmetical operations can only affect the register files. So values from two of the registers must be transferred to the inputs to the ALU.
The instruction must assert suitable control lines to the ALU

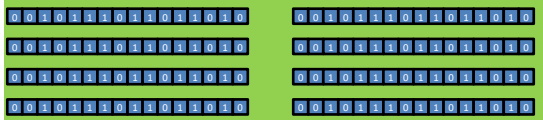
Execute

"Store"

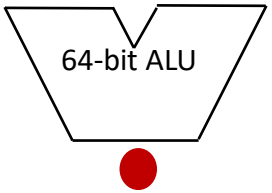
PC

PC will need to be updated

Register File



Instruction Memory



Suppose the instruction is
add $R3, $R4, $R5

The ALU performs the requested operation and the result will appear at the output of the ALU.
Here we may run into trouble with the time it takes for the operation to complete.
Increment/compare etc. can clearly be completed in one operation others may take longer.

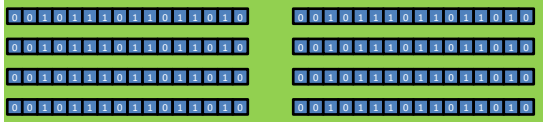
Memory Access

"Store"

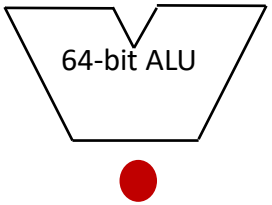
PC

PC will need to be updated

Register File



Instruction Memory



Suppose the instruction is
add $R3, $R4, $R5

This is the stage where memory is accessed if it is required. For instance a load for an array involves using the ALU to calculate an address using the address of the start of the array and the offset of the required element.

Write Back

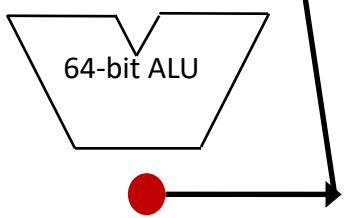
Memory

PC

PC will need to be updated

Register File

Instruction Memory



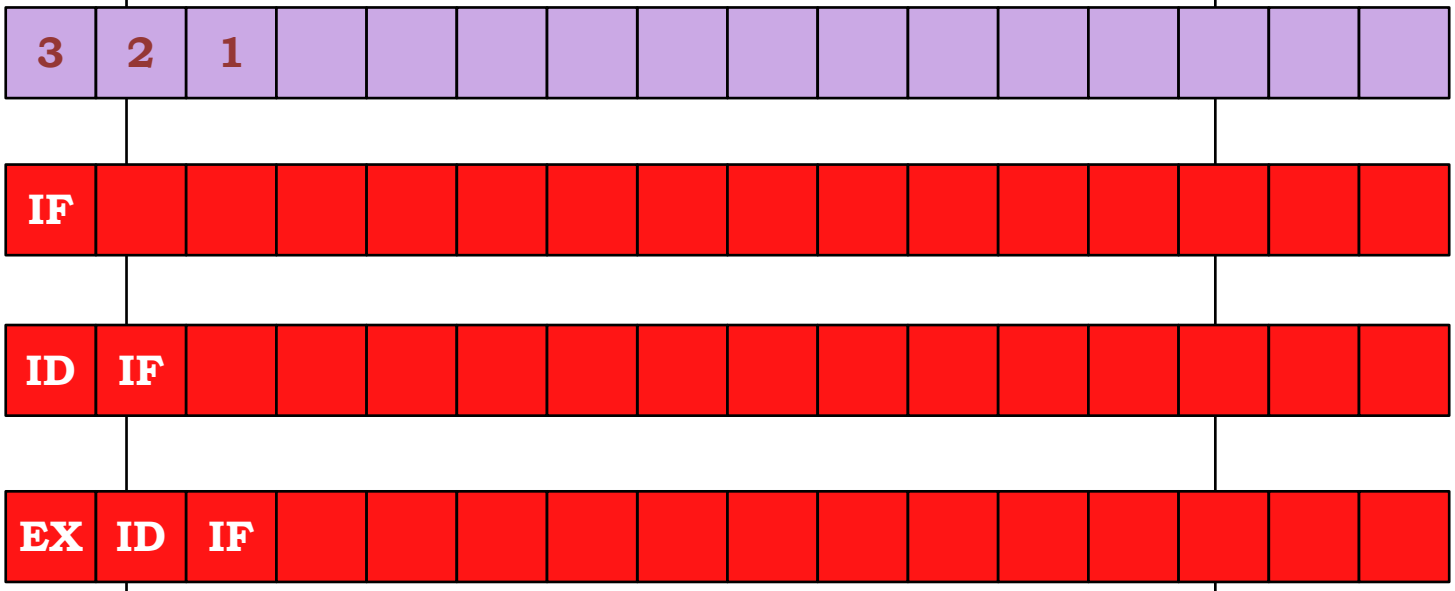
Output of the ALU – transferred back to the register file

Overlapping instructions

Instructions are divided into a number of parts.

Idea is to fetch the second instruction while decoding the first instruction.

… fetch the third instruction while decoding the second instruction and executing the third instruction

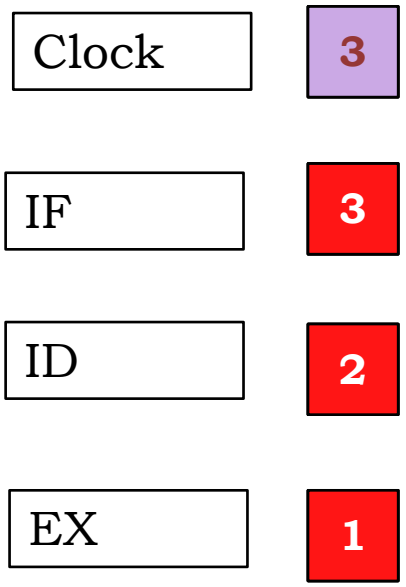


Overlapping instructions

Clock cycle 1: Instruction 1 Fetch

Clock cycle 2: Instruction 2 Fetch
Instruction 1 Decode

Clock cycle 3: Instruction 3 Fetch
Instruction 2 Decode
Instruction 1 Execute



Pipeline balance

Each stage of the pipeline needs to take the same length of time.
The advance through the pipeline will proceed at the speed of the slowest stage.

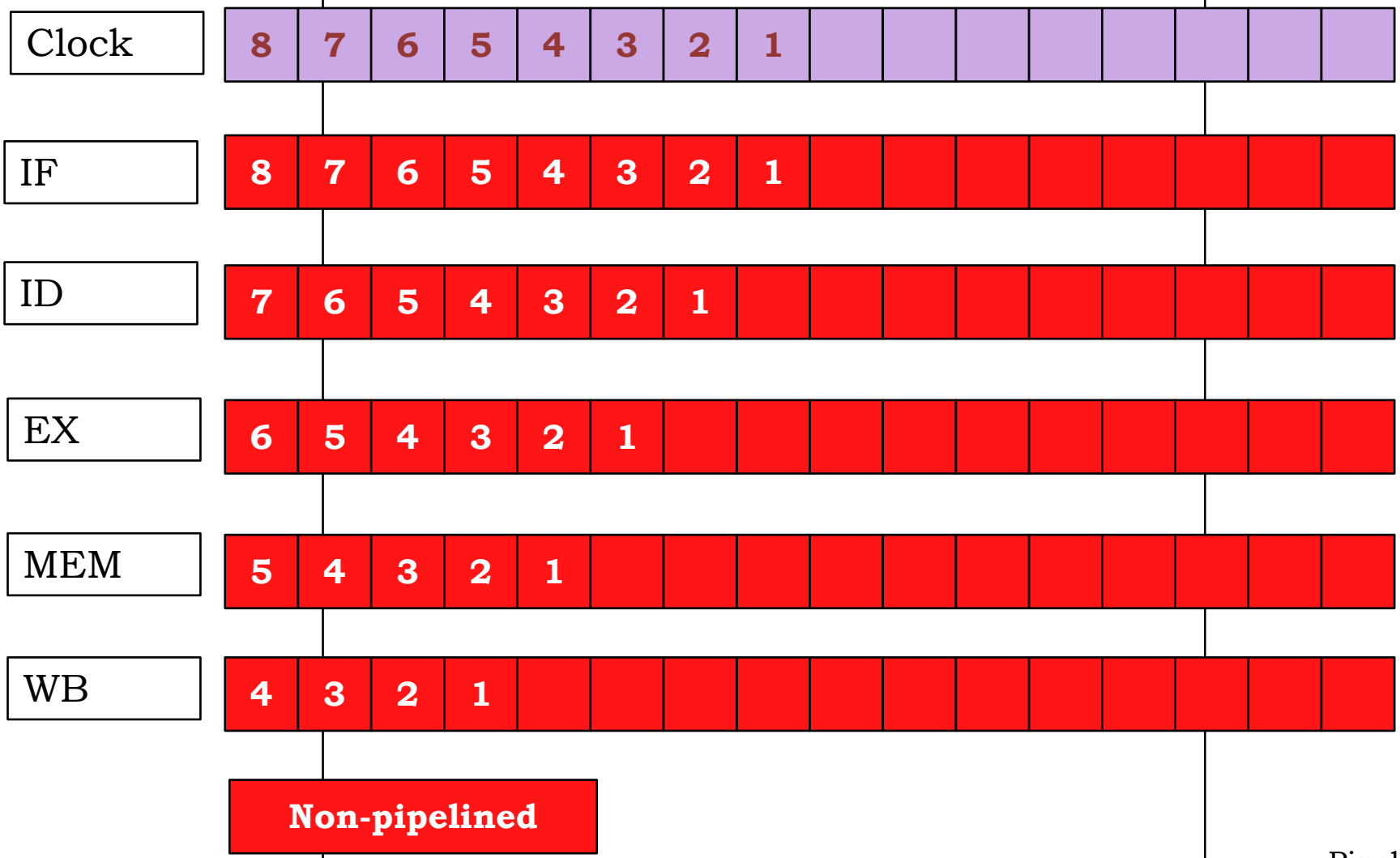
In pipelining the latency does not improve the time for the first instruction to complete does not change.

Ideally after the pipeline is full an instruction completes every tick of the clock

Anything that means an instruction cannot enter the pipeline OR cannot proceed to the next step reduces the speed of the processor.

Pipeline

Each stage of the pipeline needs to take the same time



Balanced pipeline

If it is possible to split the full instruction into n equal sections each of which take 1/n of the time – then in the steady state the speed up is n.

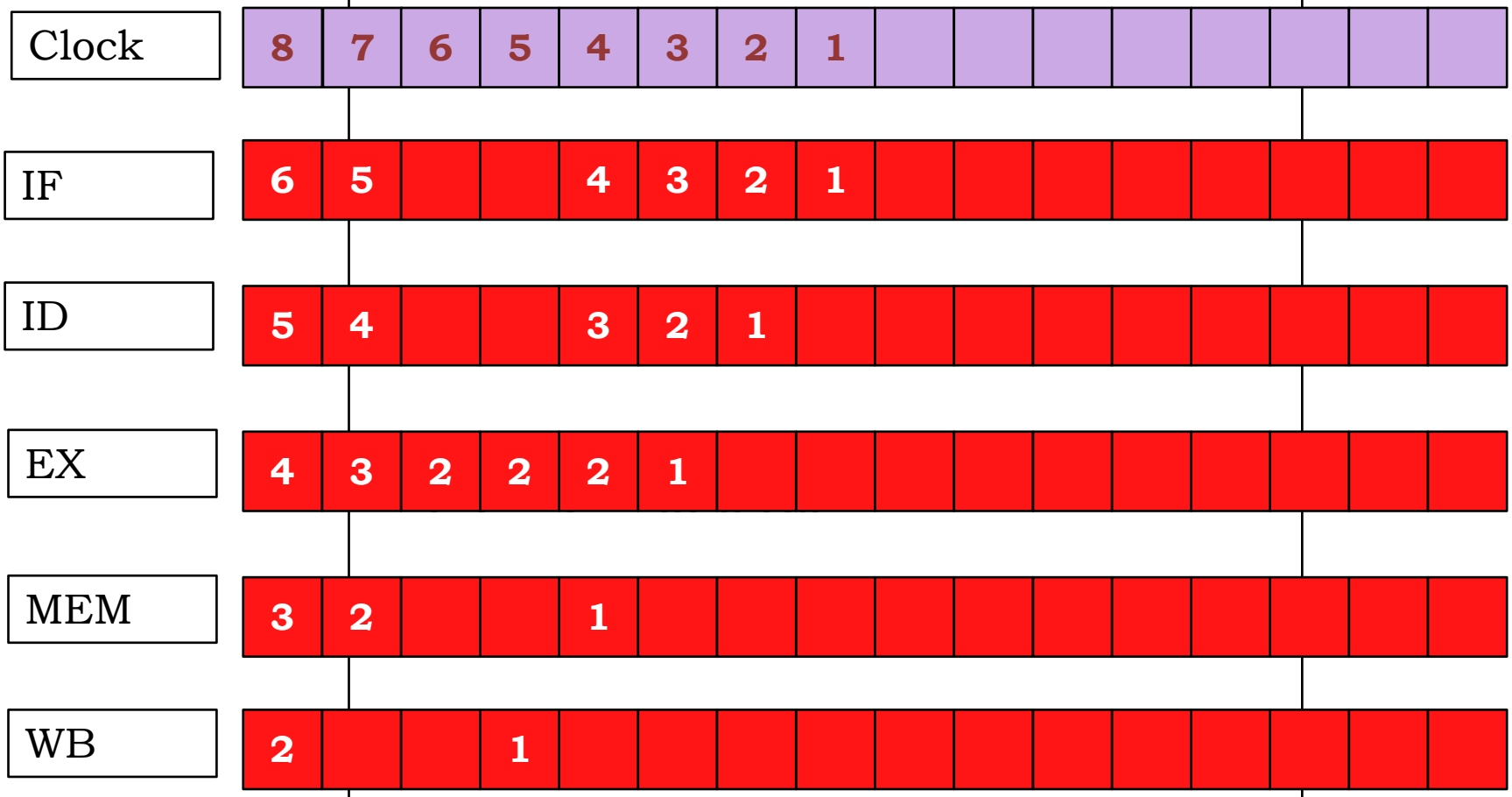
If for example we have 8 instructions.

Non pipelined they take 5*8 cycles – 40.
Pipelined they the first instruction does not complete for 5 cycles, and then they complete every cycle – 12 cycles.
Speed up is 40/12 = 3.3

Thus even an “ideal” pipeline does not reach the theoretical speed up for short programmes.

Pipeline

Pipeline stall in one stage looses an instruction at every stage



Effect of a stall

The rest of the pipeline is unable to work.

A bubble propagates downstream from the stall.

Instructions unable to proceed upstream of the stall.

The result is that performance is lost.

For 8 instructions with a 2 cycle stall –
12 instructions becomes 14
a loss of around 15%

Causes of a stall

There are a number of things which cause a pipeline stall.

The **hardware is not available** for the operation to proceed.

To fetch the next operation we need to generate a new PC (programme counter) That means adding 4 to the current counter. If the EX stage involves an addition then either the EX will have to wait a cycle or the next.

Information from earlier instruction not available

C = A + B
if C > 10 jump <target>

OR

C = A + B
D = C + E

In both cases the value of C is not available to following instruction, until it has been written to registers.
That delays the next instruction by two cycles.

Some part of the instruction takes too long.

Multiplication takes longer than addition so if the EX stage involves multiplying two numbers it make take longer than adding two numbers.

Accessing instruction or data takes more than one cycle.

Retrieving a data item, an instruction or a value from disk, may take hundreds of thousands of cycles.

A disk spins at 10,000 revolutions per minute.
If you just miss reading a location and have to wait for the disk to revolve one more cycle.
1 revolution takes 60/10000 = 0.0006s
For a 3 GHz machine that is 200,000 cycles.

Ignoring any other causes of latency.

Preserving performance

A large part of ISA is concerned with the prevention of stalls in the processor pipeline or the mitigation of their effects.

In what follows we will spend a lot of time talking about preventing stalls or minimising their effects.

Complete one instruction per cycle

Mitigating stalls

One data transfer takes 200,000 cycles.

Start a transfer every cycle. After 200,000 cycles a transfer finishes every cycle.
400,000 cycles for 200,000 data words is 2 cycles/ word.
So 1 stall per word – if we can **overlap** the transfers.

There is no machine that can perform 200,000 simultaneous reads/writes to disk: but the principle of overlapping latencies so that their total effect is mitigated is a good one.

Transferring data in blocks.

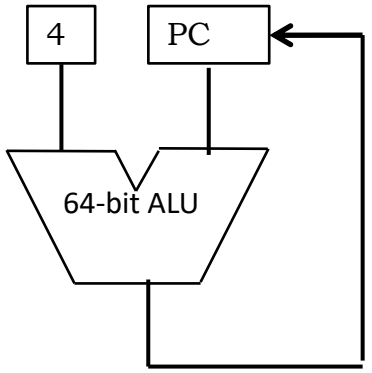
It took 200,000 cycles for the disk to be ready to transfer the word from disk, but the word that immediately follows is available immediately.

Transferring blocks of data quite often means that after the latency to the first word, subsequent words can be transferred with far less latency.

So rather than 200000 transfers, getting 1 word each, we have the more plausible 1 transfer fetching 200000 words, with the same average time.

Duplicating hardware

If we want to advance the PC on every clock cycle we can introduce a dedicated PC adder.



Now we always have a fresh PC, because while the instruction is being fetched the PC can be incremented.

This principle can be extended in other ways as we shall see.
The PC must remain stable during IF – so we might clock the PC register so it only transitions on the clock pulse.

Executing out of order

We need the result of instruction in order to execute the subsequent instruction.
But we may be able to find an instruction further on in the code which does not suffer from this problem.
So

C = A + B
D = C + E
G = X + Y

But execute as

C = A + B
G = X + Y
D = C + E

Covering up a one cycle stall is relatively easy – but you have to be sure that executing out of order still preserves the correct answer.

Summary

Instruction Level Parallelism (ILP) also called **pipelining**, drives performance.

We want to break the instruction into parts (as many as possible) – such that each part is independent and takes the same amount of time. **Balanced pipeline**

In such a machine the main limitations to performance are **pipeline stalls**.

Pipeline stalls have a number of sources

Insufficient hardware

Long latency steps – CPU or memory

Missing information

Removal or mitigation of those stalls is a major topic in ISA design.