Machine Architecture - Lecture 9

Ioannis Ivrissimtzis

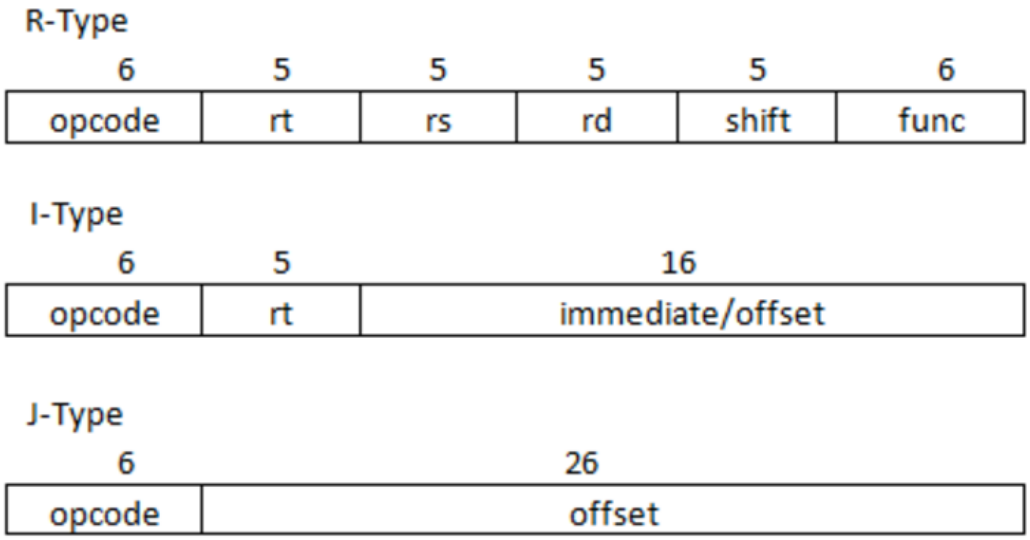
ioannis.ivrissimtzis@durham.ac.uk

CISC, RISC and pipelining

Slide material acknowledgements: Magnus Bordewich

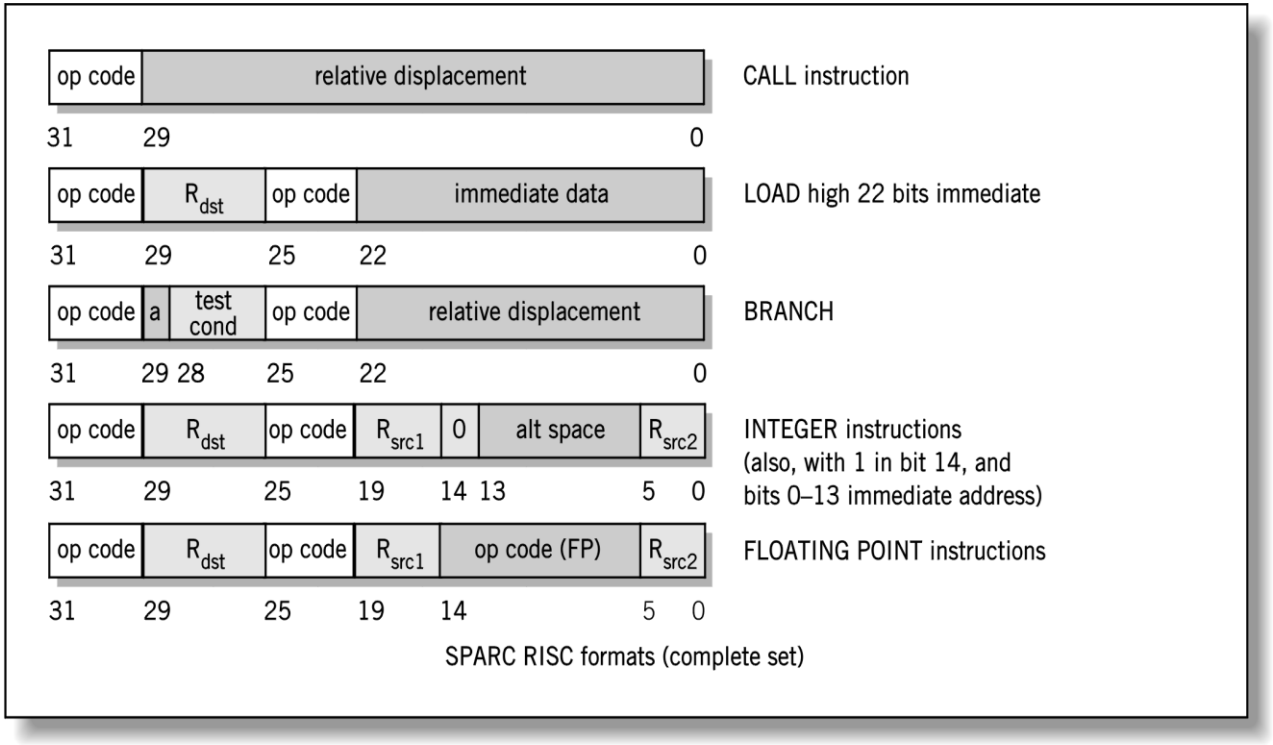
MIPS instruction formats

The MIPS model: three 32-bit instruction formats for R-type, I-type and J-type instructions.



RISC instruction formats

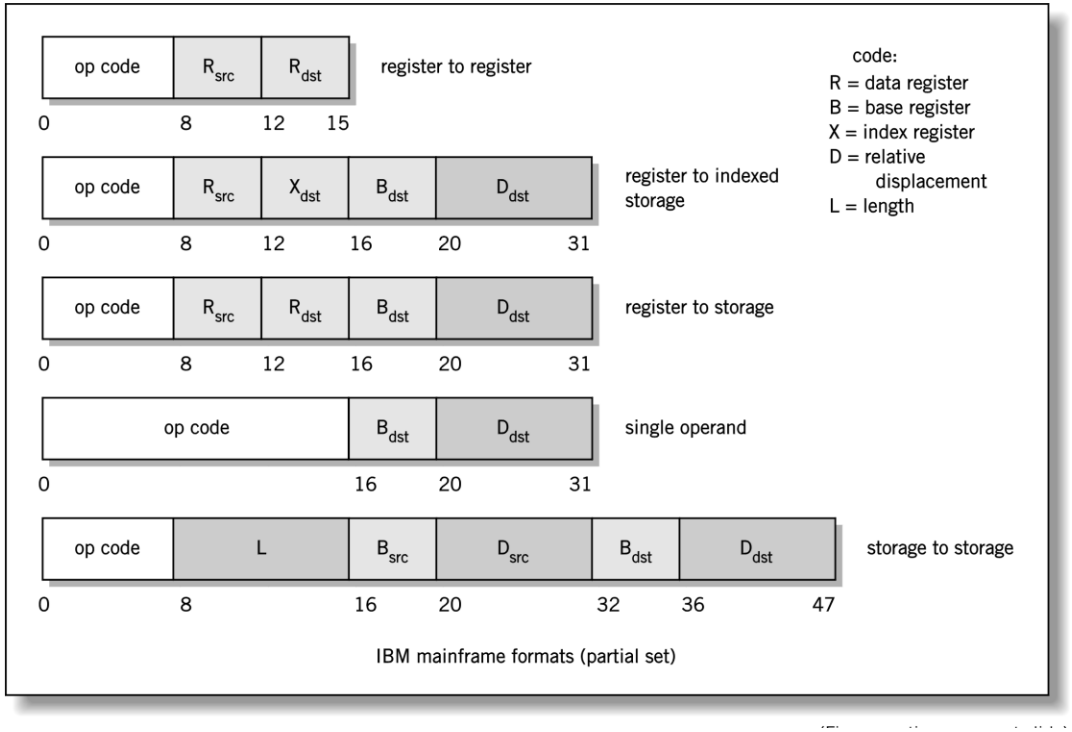
The SPARC RISC model: five different 32-bit instruction formats.



SPARC RISC formats (complete set)

CISC instruction formats

The IBM mainframe model: dozens of formats in the full set of instructions. The size of the instructions can vary.



IBM mainframe formats (partial set)

CISC architecture

Complex Instruction Set Computers (CISC)

A large number of specialised instructions.

A wide variety of addressing modes.

Instruction length varies, e.g. from 1 to 15 bytes for Pentium.

Comparatively few general purpose registers.

Prominent examples: IBM mainframe (zSeries) and Intel x86 family.

RISC architecture

Reduced Instruction Set Computers (RISC)

Limited and simplified instruction set - but instructions can be executed extremely efficiently.

Register oriented instructions - limited memory access. Essentially can only load and store to memory, most operations done on registers.

Fixed length and format of instructions - enables independent fetching and decoding of instructions.

Limited addressing modes - direct or register indirect.

A large bank of registers - intermediate results can be stored in registers to avoid load/store operations.

Prominent example: Motorola PowerPC

Example: MIPS vs x86

	MIPS	x86
registers	**32** general purpose	**8**, more restrictions on purpose
operands	**3** (2 source, 1 destination)	**2** (1 source, 1 source/destination)
operand location	registers, or immediates	registers, immediates, or memory
operand size	**32** bits	**8**, **16**, or **32** bits
status flags	**no**	**yes**
instruction size	fixed, **4** bytes	variable, **1-15** bytes

CISC vs RISC

Studies in the early 1980s showed that:

10 instructions accounted for 70% of all usage (branch, load, store).

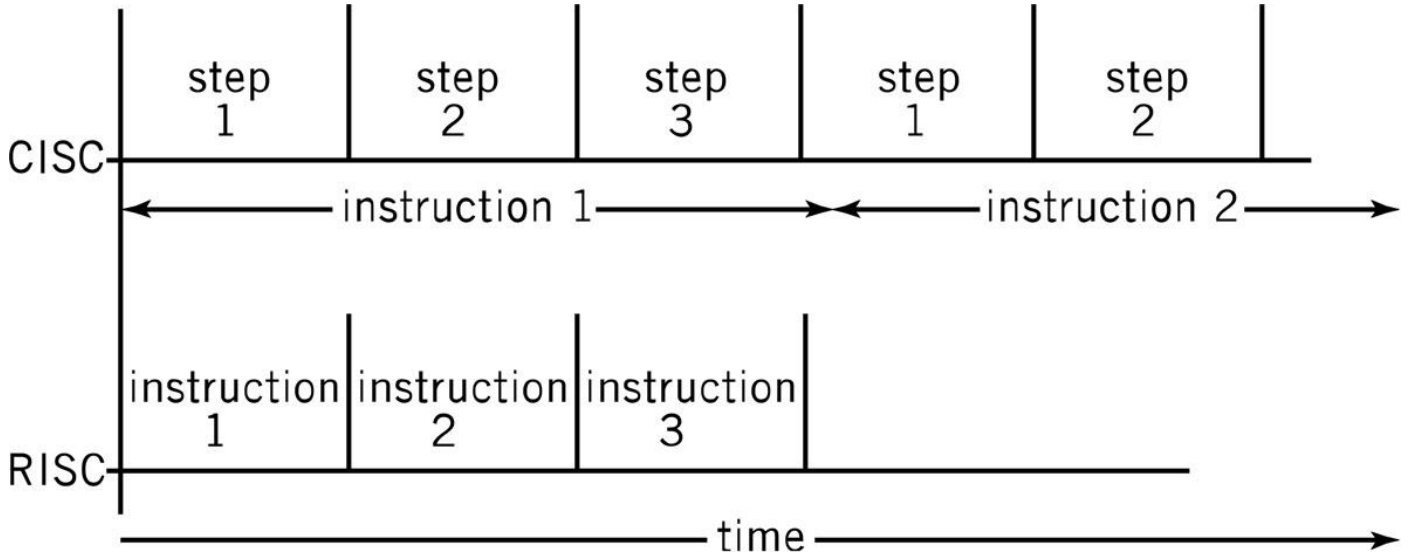
Both programmers and compilers failed to use the complex instructions available.

Procedure and function calls were a huge bottleneck, due to having to store the register contents to make the registers available for the call.

CISC vs RISC

Idea: simplified instructions can be executed more quickly than complex instructions.

Even the simple instructions in CISCs are slowed down by hardware complexity in the instruction decoder required to decode the complex instruction formats.



CISC vs RISC

Used to be a lot of debate:

Are RISC programs ‘bigger’ due to the simplified instruction set?

Does the extra memory required to store the program offset the advantage of the faster execution?

CISC vs RISC

Now the increase in chip real estate has allowed RISC chips to include more instructions and reduce program size.

Newer CISCs include greater numbers of general purpose registers.

Some CPUs have complex instruction sets but translate complex instructions into internal micro-ops of fixed length, similar to RISCs.

Nowadays the differences are smaller and performance of RISCs and CISCs is very similar.

Architecture: key characteristics

Many different architectures in widespread commercial use, but once you understand one, learning others is much easier.

Key questions to ask when approaching a new architecture:

- What is the data word length?
- What are the registers?
- How is memory organised?
- What are the instructions?

The MIPS example

What is the data word length?

MIPS is a 32-bit architecture.

What are the registers?

32 general purpose registers.

Almost any register can be used for any purpose. In practice, their usage follows certain conventions.

The MIPS example

How is memory organised?

Byte addressable memory with 32-bit addresses.

The memory map was described in the previous lecture.

What are the instructions?

Instructions are 32 bit long and must be word aligned.

Many commonly used MIPS instructions discussed in previous lectures and in the practicals.

Pipelining

Efficient biscuit baking

You need 100 biscuits for a party you are having.

Each tray takes 10 biscuits.

It takes you 10 minutes to roll out a tray of biscuits.

It takes 20 minutes to bake a tray of biscuits.

How long will it take you to bake 100 biscuits?

Fetch-execute cycle

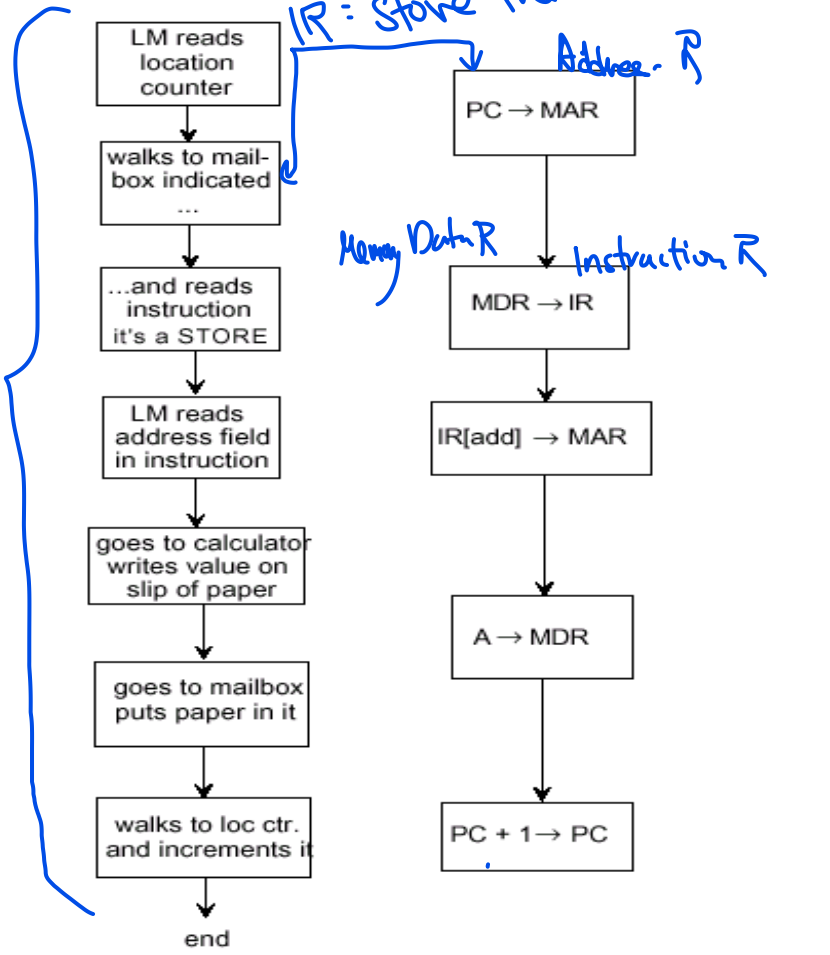
Fetch:

Find and decode instruction, load from memory into register and signal ALU.

Execute:

Performs operation that instruction requires.

Move / process data.



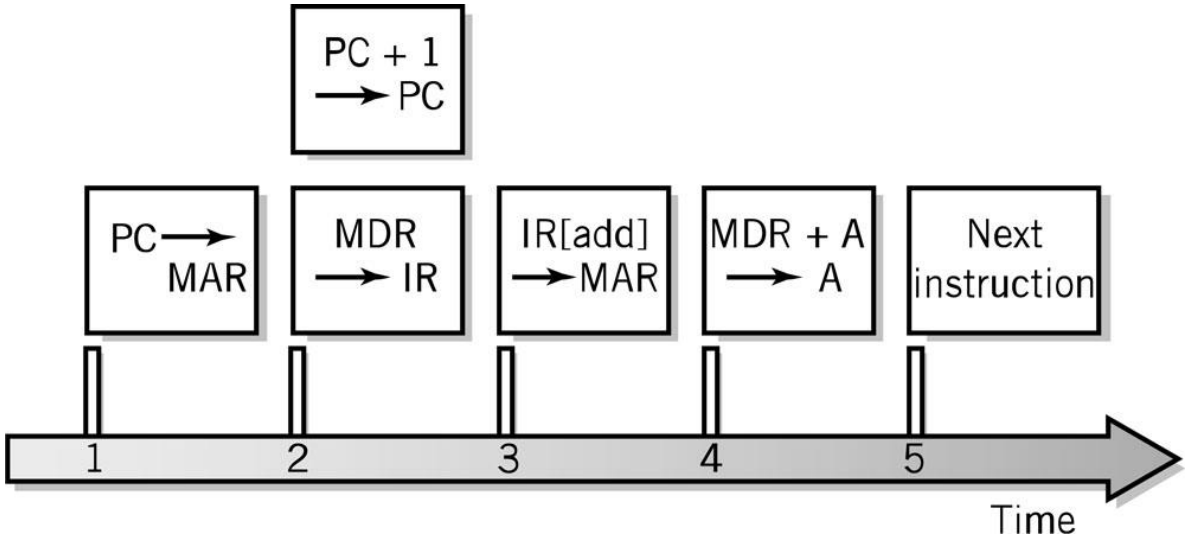
Timing

Computer clock used for timing purposes:

MHz - million steps per second

GHz - billion steps per second

Instructions can (and often) take more than one step.



Improvements

Idea: allow parts of the cycle, and consecutive cycles to be done in parallel.

- Create separate fetch and execute units to allow them to operate in parallel.
- Pipelining - assembly line approach: separate execution units for different types of instruction to allow parallel execution of unrelated instructions.

Separate fetch / execute units

Give the little minion an assistant:

> The assistant just goes back an forth between the PC and the mailboxes fetching the instructions in order.
>
> It also decodes them.
>
> The little minion executes all instructions.

If the assistant can feed decoded instructions to the little minion at the speed it can execute them, we save a lot of time.

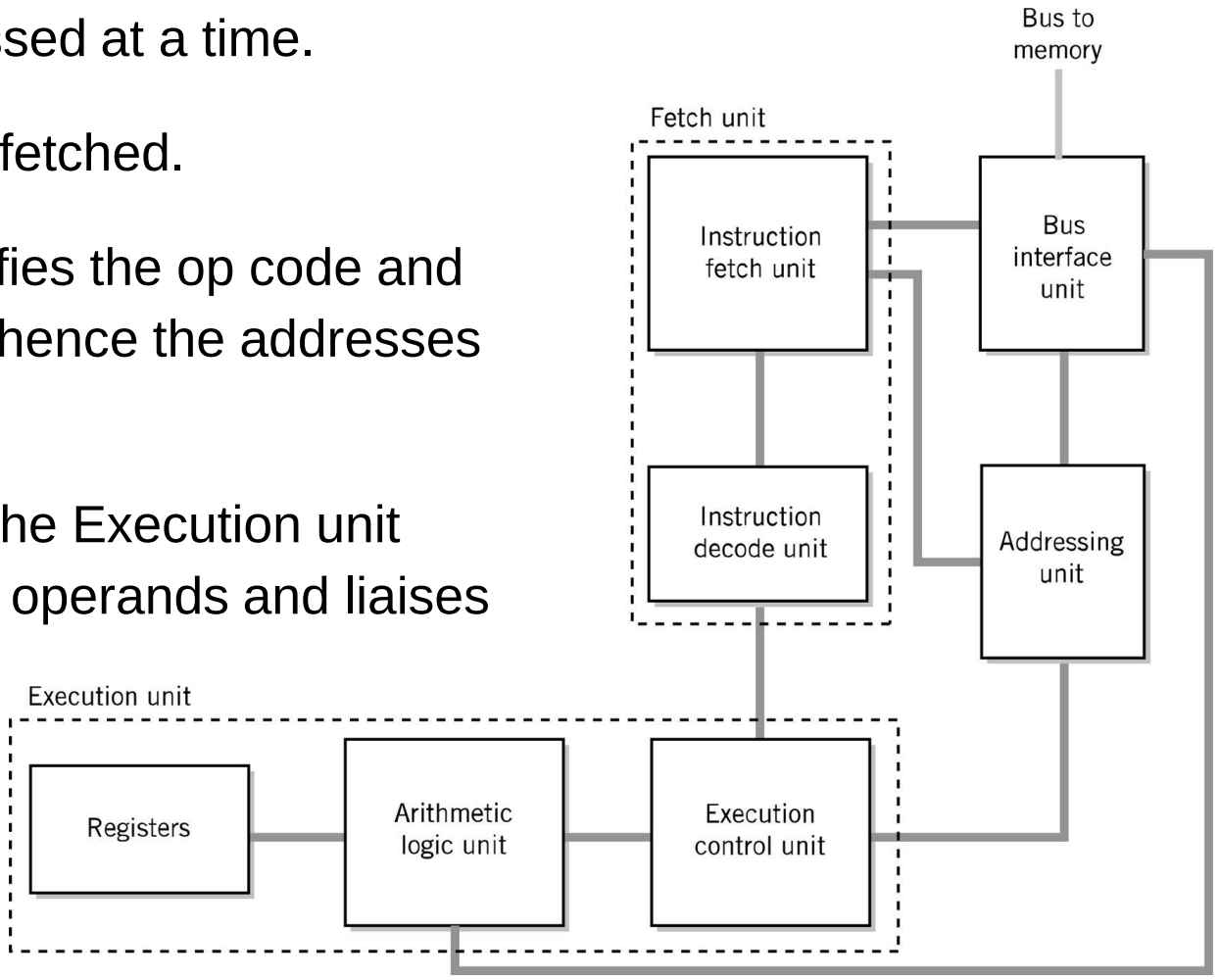
Separate fetch / execute units

One instruction processed at a time.

The next (few) can be fetched.

The decode unit identifies the op code and instruction format and hence the addresses of the operands.

This info is passed to the Execution unit which reads/writes the operands and liaises with the ALU.



Pipelining

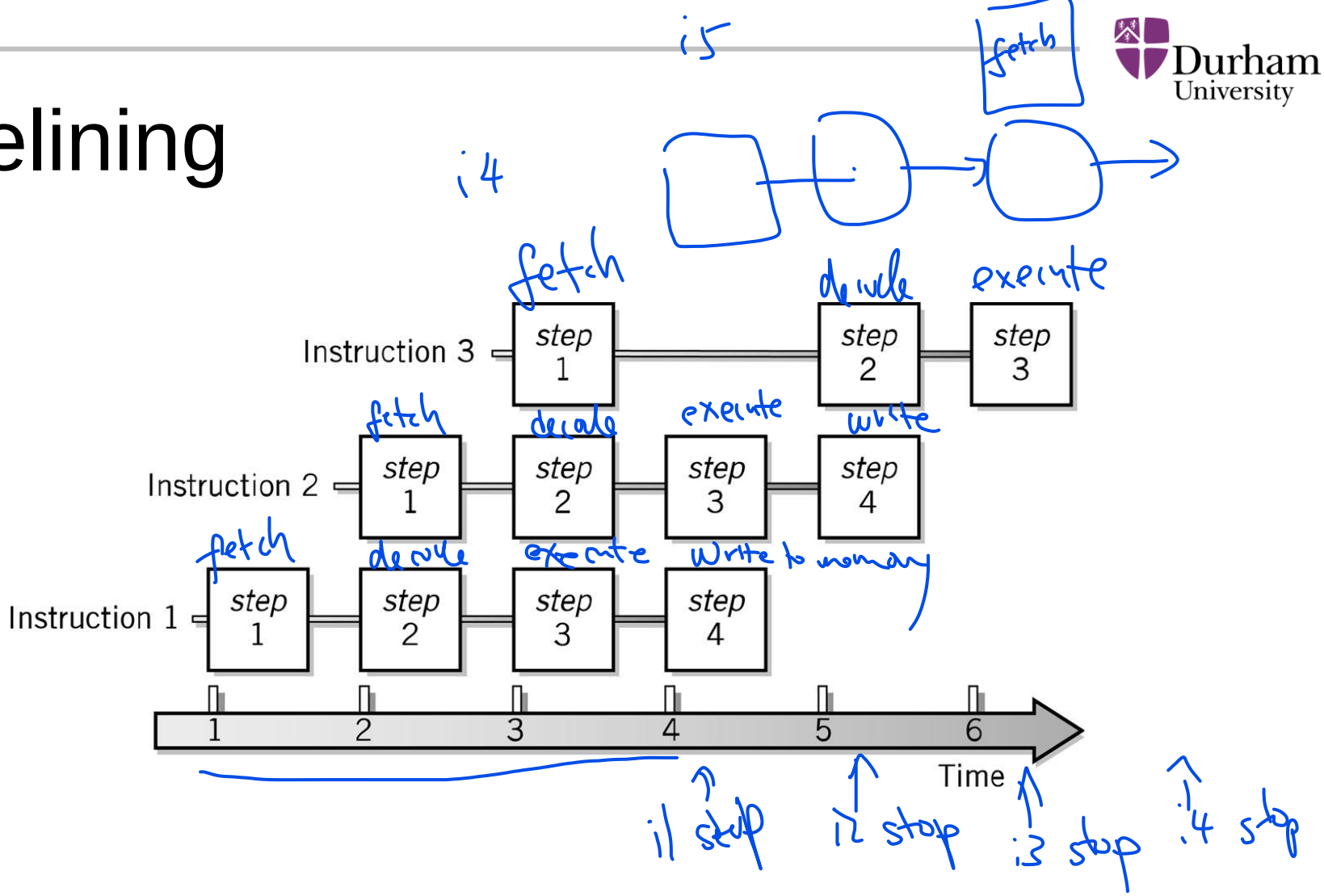
We have separated the fetch and execute.

We can now start the next cycle before the current one has finished.

Can we do the same at the instruction level?

If instructions take several steps, can we move the next instruction into that step as soon as the instruction before has finished it?

Pipelining



Notice that here Instruction 3 is delayed by waiting for the result of step 3 in Instruction 2.

Pipelining

Assembly-line technique to allow overlapping between fetch-execute cycles of sequences of instructions.

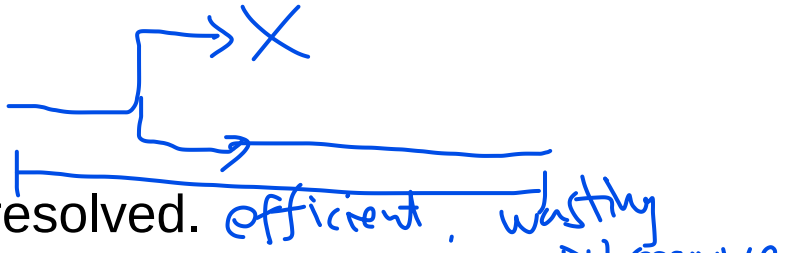
Scalar processing:

- An individual instruction takes the same amount of time as before.
- Average instruction throughput is approximately equal to the clock speed of the CPU.

Problems from stalling and branching.

- Waiting for prior results to become available.
- At a branch, we don’t know what the next instruction is.

Branch Problem Solutions



Separate pipelines for both possibilities:

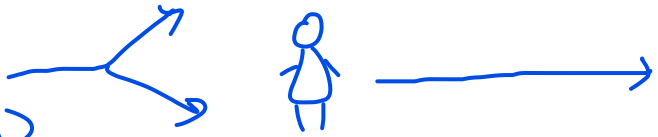
Work on both outcomes until the branch is resolved.

Probabilistic approach:

Predict the branch outcome based on previous visits or programmer hint.

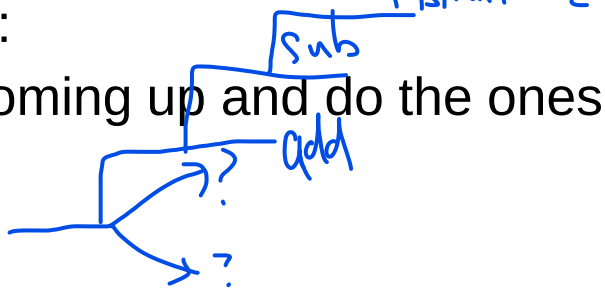
Requiring the following instruction to not be dependent on the branch:

Can also be used to avoid stalling: following a write to memory do not allow read from that location for a few steps.



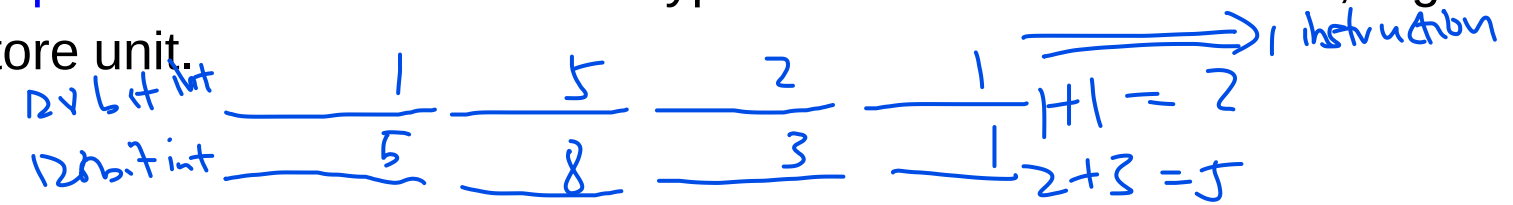
Instruction Reordering:

Analyse instructions coming up and do the ones that do not depend on the current pipeline first.



Multiple execution units

Include specialist units for different types of instruction execution, e.g. Load/Store unit.



Have several units for the most common instructions.

Each unit can have a pipeline, and if they are all filled, the overall processor can execute more than one instruction per clock cycle.

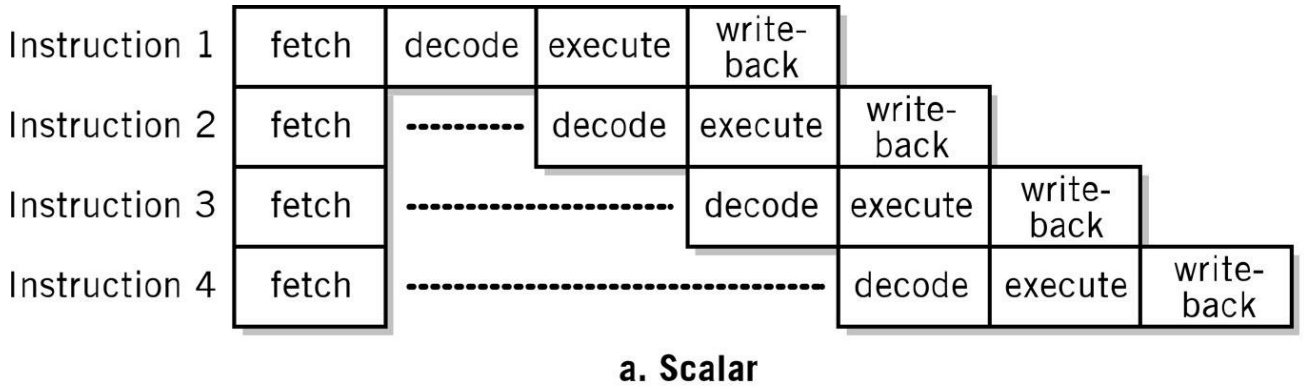
This is called superscalar processing.

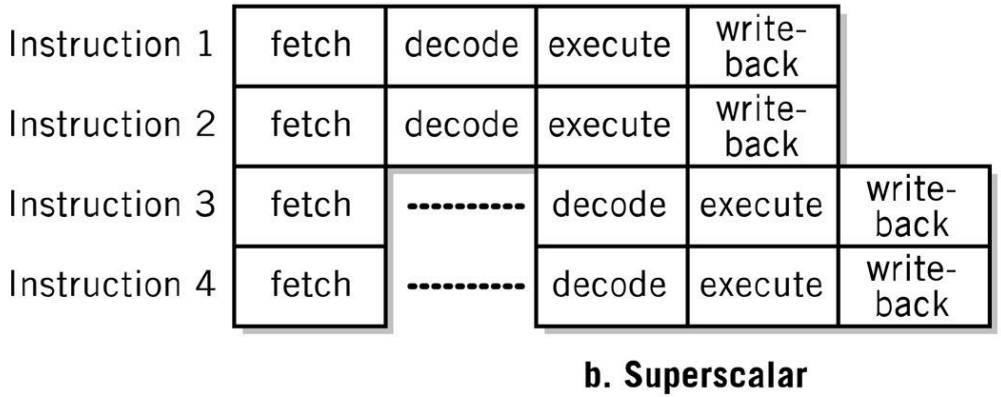
SIMD

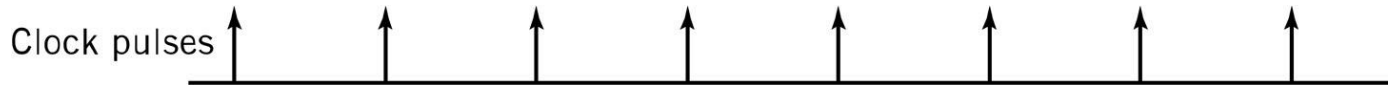
Single instruction Multiple Data

Current CPUs achieve up to 5 times scalar processing speeds.

Superscalar processing







Superscalar processing

Complications:

- Instructions may complete in the wrong order.
- Dependencies on prior results may be lost.
- Branch instructions must be resolved correctly.
- Conflicts for CPU resources, e.g. registers, arise.

Getting it all to work requires a lot of additional complexity in the CPU and a burden on the assembly programmer.