COM1006 Devices and Networks (Autumn)
COM1090 Computer Architectures

Lecture #12

▶Accelerating performance

Dr Dirk Sudholt
Department of Computer Science
University of Sheffield

d.sudholt@sheffield.ac.uk
http://staffwww.dcs.shef.ac.uk/~dirk/campus_only/com1006/

Based on Chapter 8 in Clements, Principles of Computer Hardware

Aims of this lecture

- To briefly review techniques for quantifying computer performance.
- To show how performance can be increased by pipelining.
- To discuss potential problems ('hazards') with pipelining, and show how they can be resolved.
- To explain how throughput can be increased by using a number of processors working in parallel.
- To explain the relationship between multiprocessor systems and chips with multiple processing cores.

Measuring performance

- Before considering how to improve performance, we need to know how to **measure** performance.
- Take performance as time to execute a program, considering only the CPU. This time is determined by many factors:
  - the number of instructions in the program
  - the number of clock cycles needed to execute each instruction (some are faster than others)
  - the period of the processor's clock.

Which of these are determined by the computer architect, and which by other factors?

Benchmarking

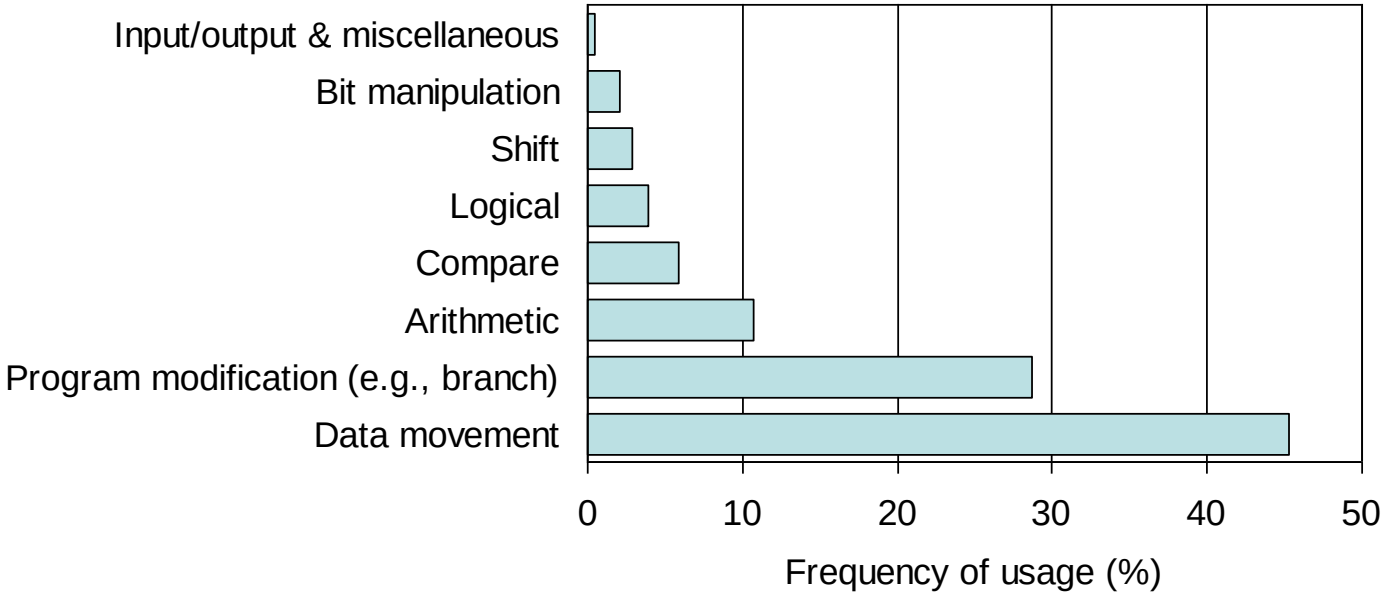
- Clearly, different computers cannot be compared simply using a single factor such as clock rate.
- A widely used technique is to compare the time taken to execute **benchmark** programs on a **reference machine**.
- Example: **SPEC benchmark** is the mean of a suite of benchmark tests that evaluate CPU, cache, memory, buses and system software.
- Standard Performance Evaluation Corporation: See

  `http://www.spec.org`

The RISC revolution

- From 1970's onwards there was a trend towards more complex computer architectures.
- Advances in chip fabrication were exploited by making processors with more complex instructions, but these required more complex decoder circuitry.
- Reaction against CISC started at IBM with the 801 architecture and continued at Berkeley where Patterson and Ditzel coined the term **RISC**.
- RISC approach informed by quantitative evaluation in late 1970's of the way in which computers execute programs.

Instruction usage (from 1970s)



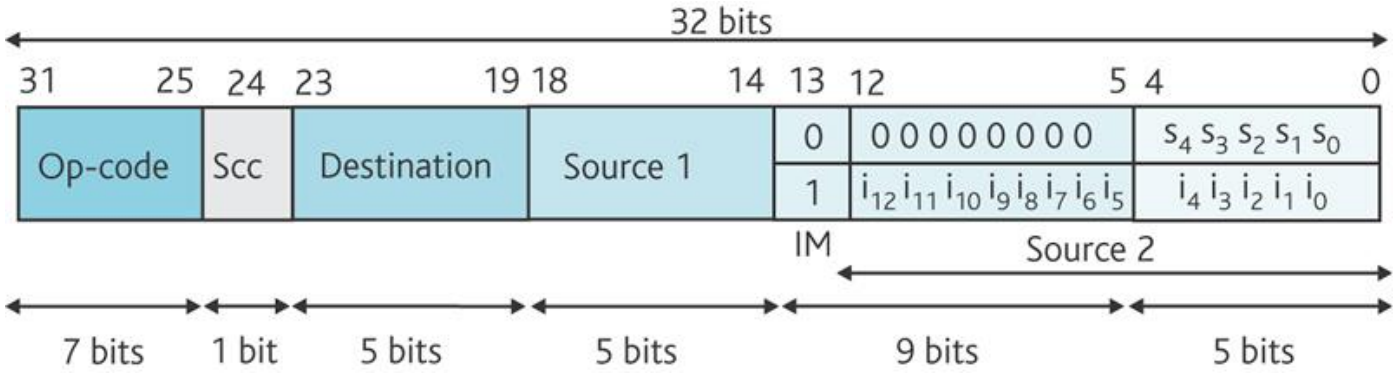
- Data movement (e.g., MOVE) and program modification (e.g. BEQ) together count for about 75% of instruction usage.
- **Conclusion**: make frequently used instructions fast. Don't spend design effort on complex instructions that are seldom used anyway.

Characteristics of RISC architectures

- Lots of **on-chip memory** to counter the effect of the von Neumann bottleneck.
- Three-address, **register-to-register** architectures.
- Subroutines and flow control are used frequently, so make parameter passing and branching fast.
- Don't implement infrequently used instructions.
- Aim to execute **one instruction per clock cycle** on average. Hence complexity of instructions must be limited.
- **Fixed-length instructions** (CISC processors have variable-length instructions, which complicates decoding).

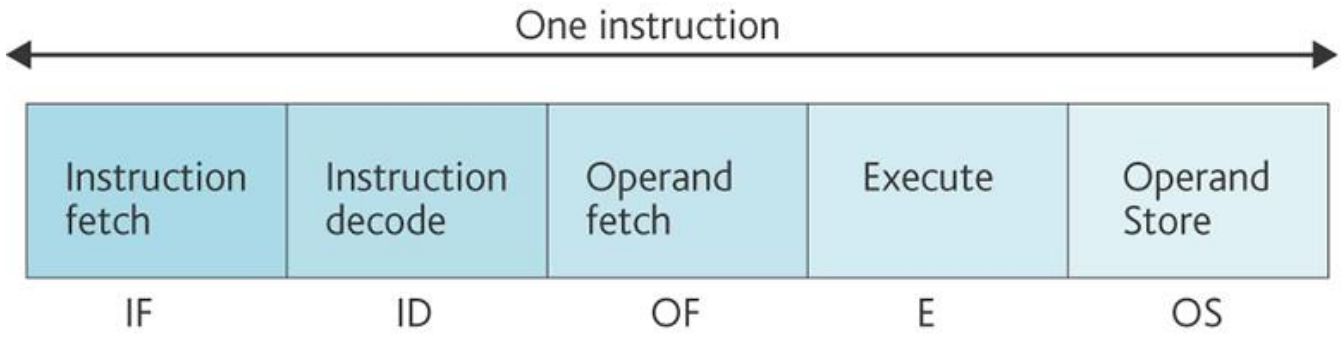
The Berkeley RISC machine

- One of first RISC processors developed at University of California at Berkeley. Not commercial, but influential.
- Fixed 32-bit instruction format:



- If Scc=1 status register updated after execution of instruction.
- When IM=1 (immediate mode) source 2 is a 13-bit constant value. When IM=0 bits 5-12 are zero and bits 0-4 provide the second source operand register.
- 32 internal registers, but can switch between 138 registers.

Phases of execution for `ADD R1,R2,R3`



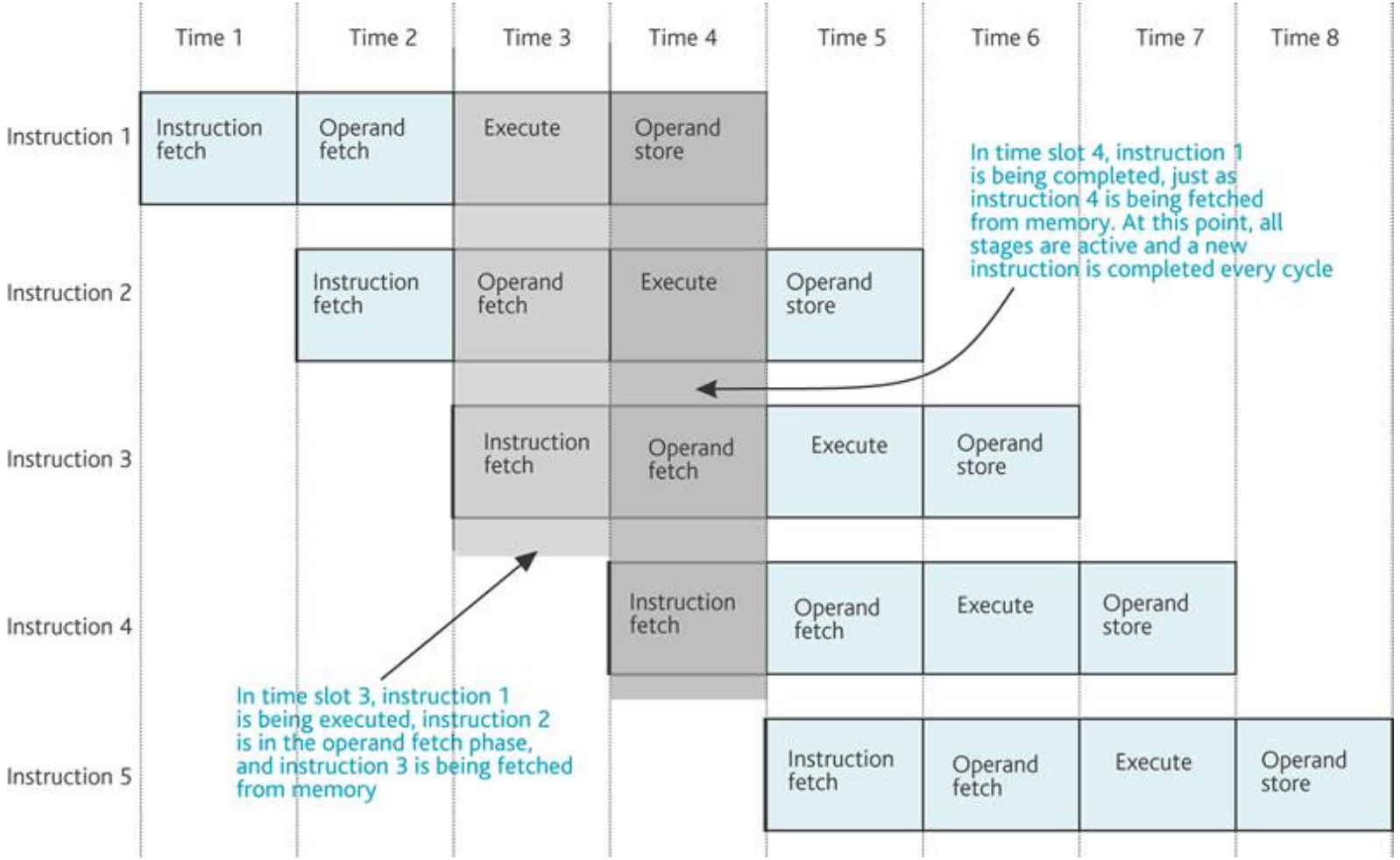
IF	**`ADD R1,R2,R3`**	Read this instruction from memory
ID	`ADD R1,R2,R3`	Decode this instruction into "ADD" and registers 1, 2, 3
OF	`ADD R1,`**`R2,R3`**	Read operands R2 and R3 from the register file
E	**`ADD`** `R1,R2,R3`	Add the two operands together
OS	`ADD` **`R1`**`,R2,R3`	Store the result in R1

- Note the inefficiency: each phase of instruction execution is active only 20% of the time, lies idle for remaining 80%.

Pipelining

- Pipelining increases effective speed of the processor by **overlapping** the various stages of instruction execution.
- Example: Fetch instruction *i*+1 while fetching the operand of instruction *i*.
- Aim to keep the pipeline full (all stages active).
- Following examples show a four-stage pipeline; in RISC processor instruction decode phase is minimal because instruction encoding is very regular.

Pipelining and instruction execution



Speedup due to pipelining

- Consider execution of ***n* instructions** in ***m*-stage** pipeline.
- Takes *m* cycles for first instruction to be completed, then *n*-1 instructions at rate of one per clock cycle.
- Total time is *m*+*n*-1 cycles.
- Without pipeline, takes *n* x *m* cycles to execute *n* instructions (if each is executed in *m* phases).

Speedup due to pipelining is

$$\frac{n \times m}{m + n - 1}$$

n	3-stage	6-stage	12-stage
4	2.000	2.667	3.200
8	2.400	3.692	5.053
20	2.727	4.800	7.742
100	2.941	5.714	10.810
1000	2.994	5.970	11.869
∞	3.000	6.000	12.000

Pipeline hazards

- A pipeline is an ordered structure that exploits regularity.
- In practice machine code is divided into blocks with breaks between them called **hazards** or **pipeline stalls**.
- **Data dependency**
  - An instruction cannot be executed because it requires the result from an operation that is still in the pipeline.
- **Branch penalty**
  - When branch occurs, program counter is loaded with a new address.
  - Partially completed instructions in the pipeline must be thrown away, causing a **bubble.**

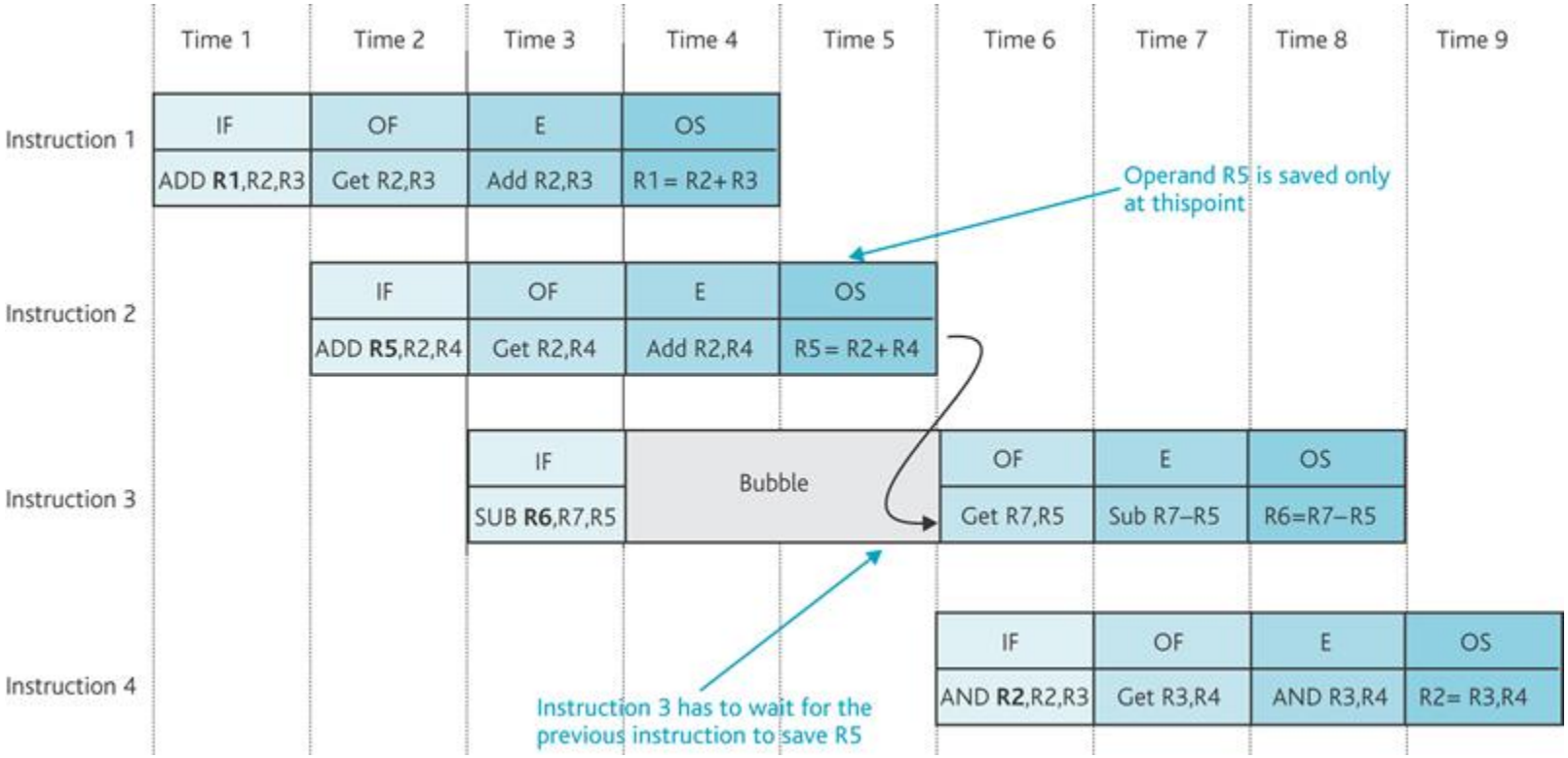
Data dependency

- Consider the following sequence of instructions:

```
ADD R1, R2, R3      [R1] ← [R2] + [R3]
ADD R5, R2, R4      [R5] ← [R2] + [R4]
SUB R6, R7, R5      [R6] ← [R7] - [R5]
AND R2, R3, R4      [R2] ← [R3] AND [R4]
```

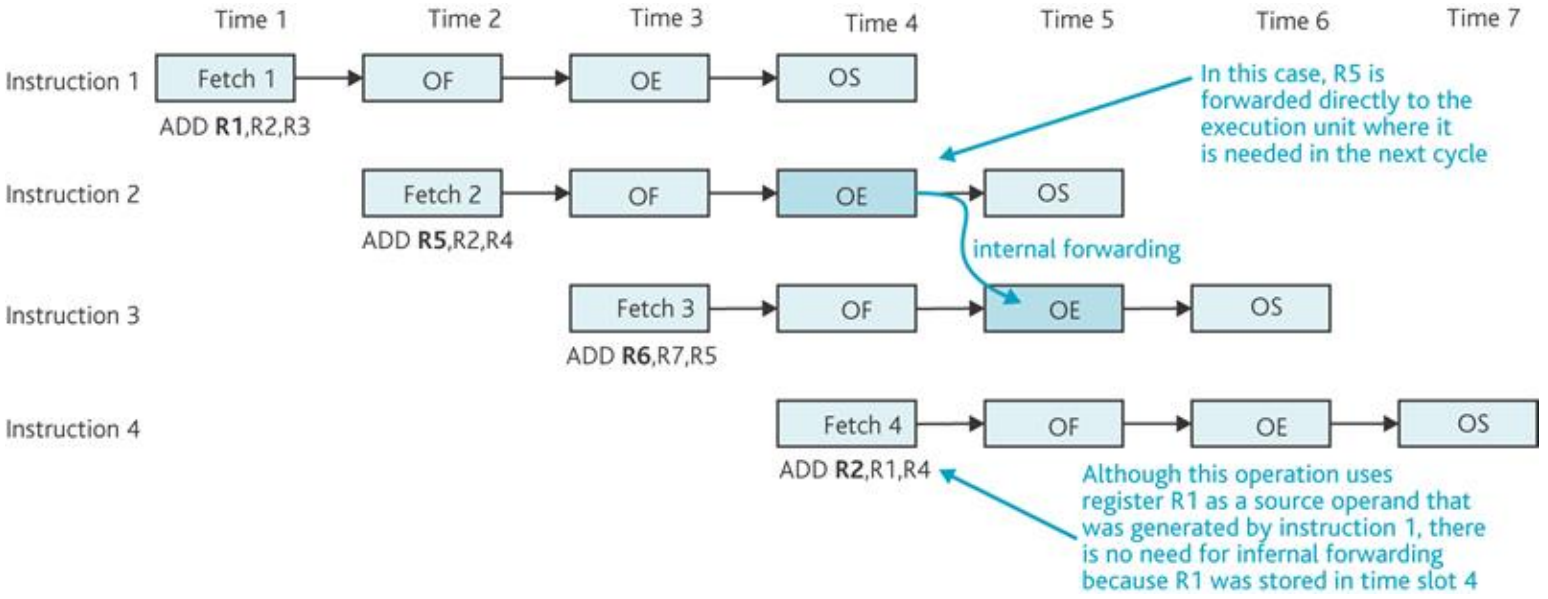
- The value of R5 will not have been stored by the second instruction by the time it is needed for the third instruction.
- The pipeline must be stalled (causing a bubble) until R5 has been stored.
- Bubbles only occur when variables are being **stored** (e.g. there is no data dependency involving R2 as R2 is only being read in steps 1-3).

Data dependency (cont'd)

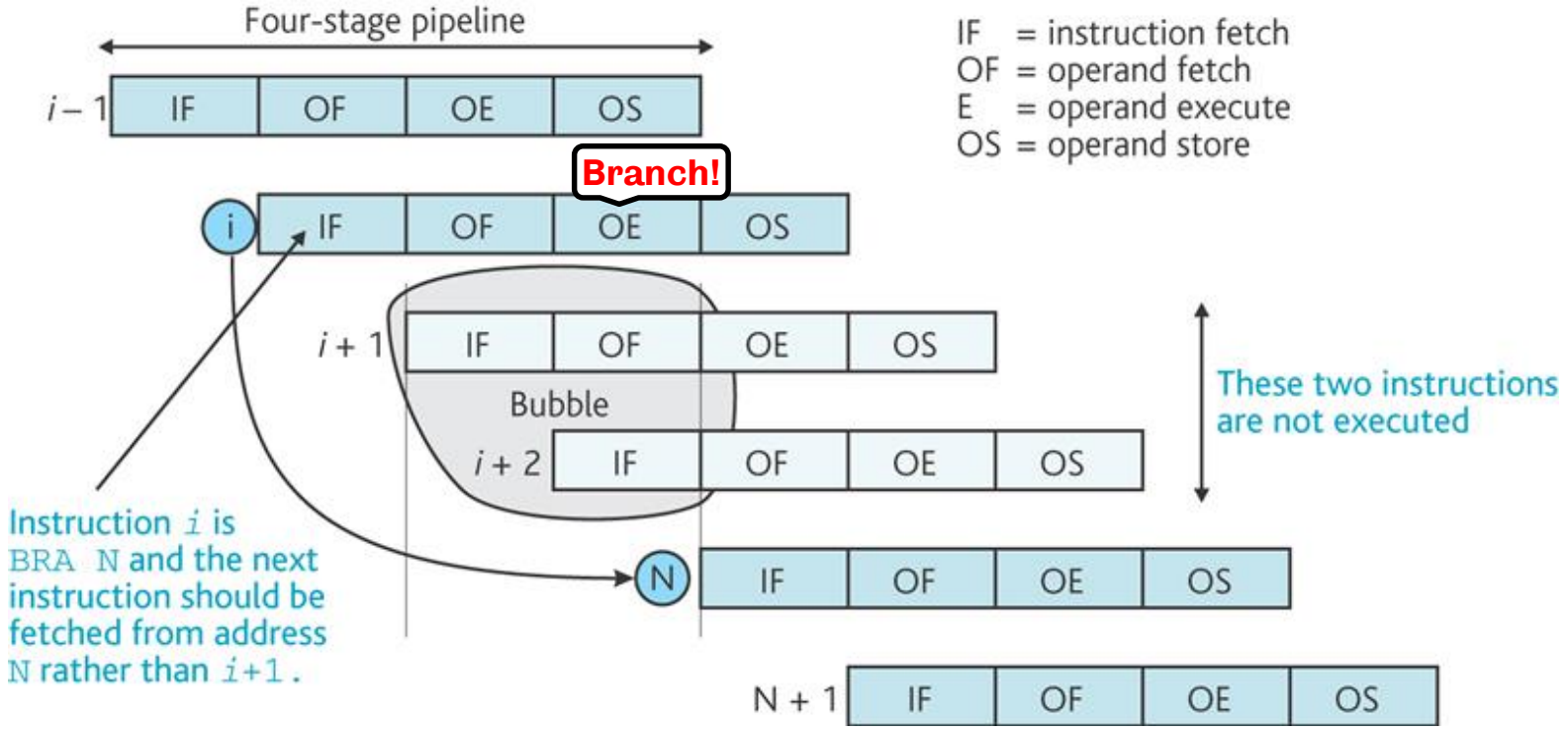


How to deal with data dependencies

- Compilers can **rearrange code** to minimise bubbles.
- **Internal forwarding:** Value of R5 is moved **directly** from execution unit of second instruction into the execution unit of the third instruction.



Branch penalty



- Branch to N is detected only during its **execution** phase.
- Next two partially executed instructions need to be discarded.
- Continue filling the pipeline from instruction N.

Reducing the branch penalty

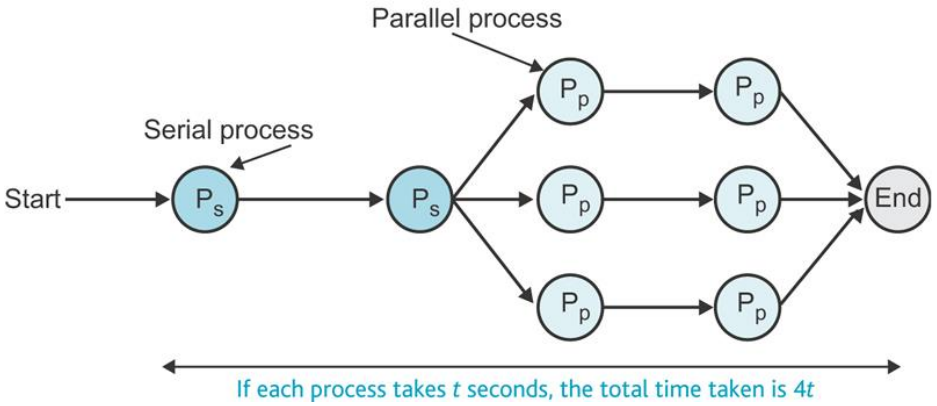
- Predict outcome of branch instruction before it is executed, and start **filling pipeline with instructions from target address.**
- Simple for unconditional branches. Need a more sophisticated approach for conditional branches.
- **Static branch prediction**:
  - assume that branches are always taken or never taken.
  - code analysis suggests that branches are taken more than 50% of the time, so best approach is to assume branch always taken.
- **Dynamic branch prediction**:
  - At run-time, use past behaviour of program to predict its future behaviour. Outcome of branch instructions stored in table.

RISC and CISC in practice

- In practice, there is now fuzzy line between RISC and CISC.
  - CISC architectures apply RISC techniques such as pipelining and fast cache memory.
  - RISC architectures have complex instructions in their instruction sets (e.g., for multiplying floating point numbers).
- It's no good increasing processor throughput unless memory can keep up.
- Developments in cache memory have been key to the success of RISC - see next lecture.

Multiprocessor systems

- Accelerate performance by using two or more processors working in **parallel**, dividing the work between them.
- Speedup limited by ability to distribute tasks to processors, which in turn depends on nature of problem.
- Performance increase limited if parts of problem can only be done serially.



- Classify architectures as SISD, SIMD, MISD or MIMD based on topology and inter-processor communication (due to Michael J. Flynn, Stanford University).

Multiprocessor architectures

SISD (single instruction single data stream)

- Conventional single processor, executes one instruction at a time.

SIMD (single instruction multiple data stream)

- Execute instructions sequentially on parallel data streams.
- E.g., vector mathematics: compute inner product
  $\mathbf{A} \bullet \mathbf{B} = a_1b_1 + a_2b_2 + a_3b_3 + \ldots + a_nb_n$
  by computing each multiplication a_ib_i on one of n processors.
- Also called **array processor** or **vector processor**.
- Important applications in graphics and image processing.

Multiprocessor architectures (cont'd)

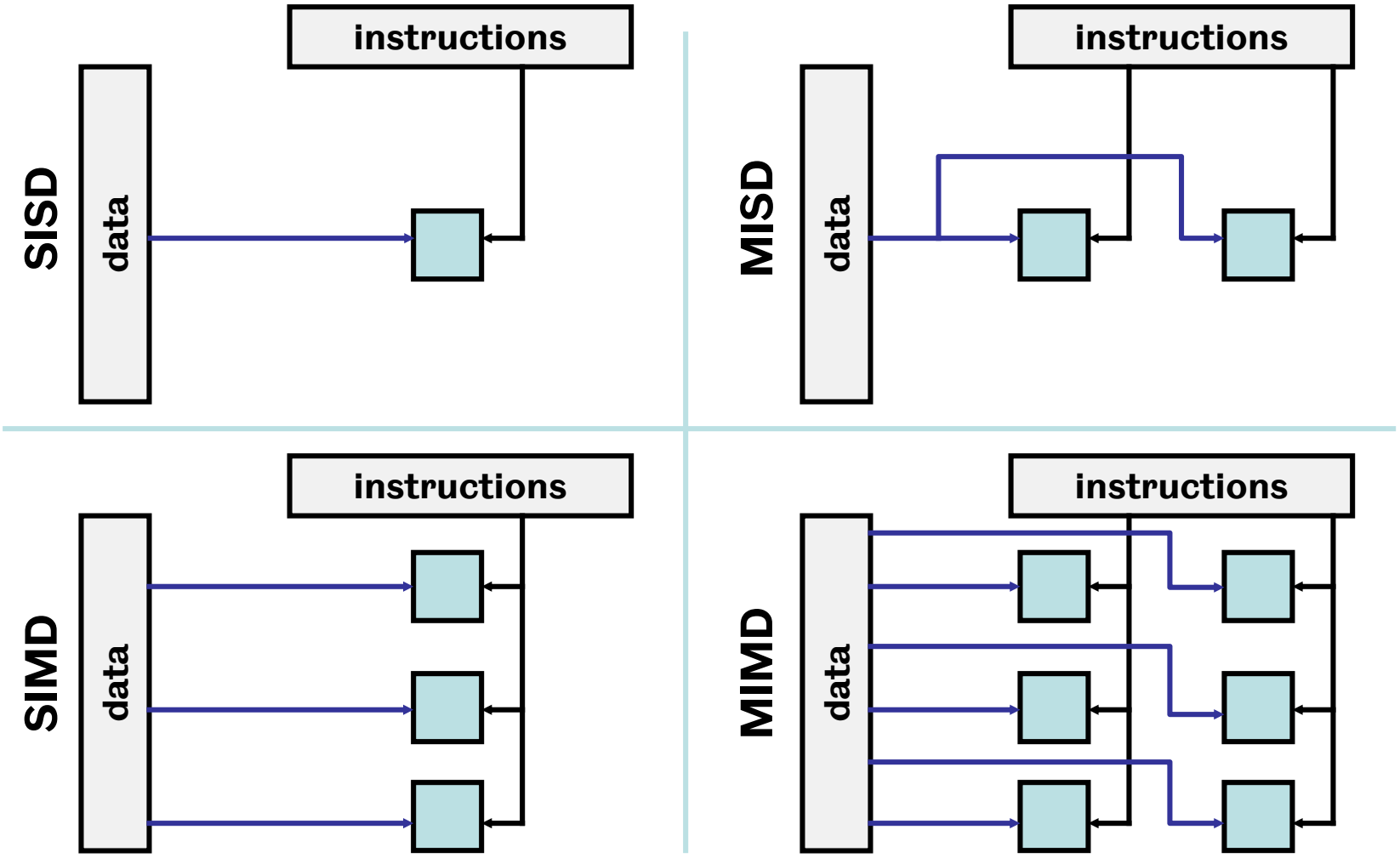
MISD (multiple instruction single data stream)

- Not so widely used. Performs multiple operations on a single data stream.
- Like a pipeline in which each stage is a separate processor.

MIMD (multiple instruction multiple data stream)

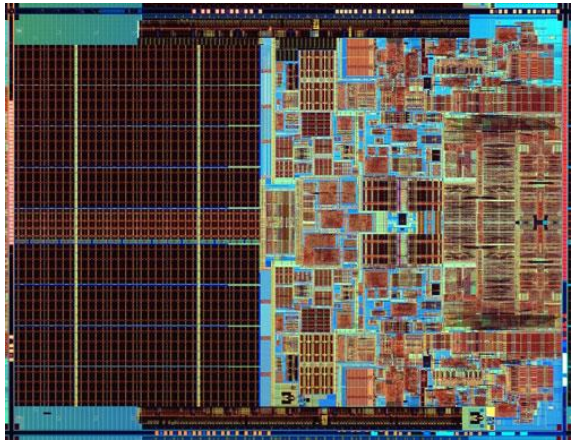
- Each processor executes own instructions and has own data stream.
- Most general and most widely used multiprocessor architecture.
- Many possible topologies (bus, ring, star, hypercube etc.)

Summary: multiprocessor architectures



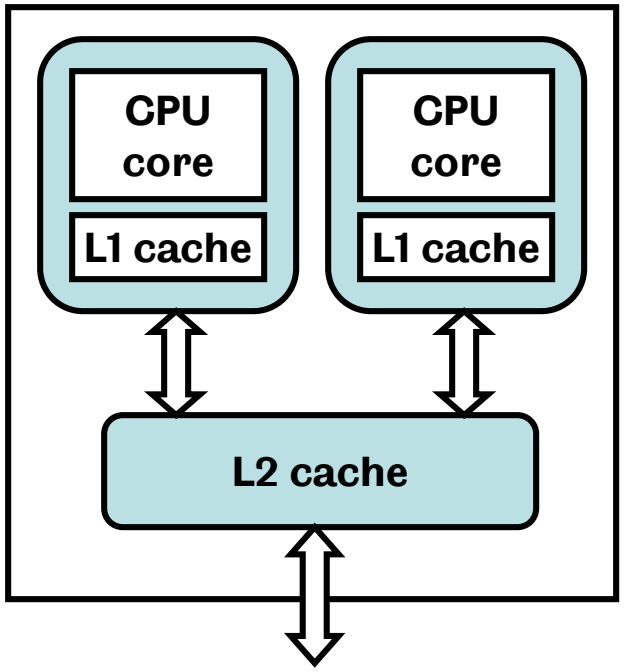
Multiple processing cores

- Current trend is away from "GHz-oriented" approach to multiple processing cores, because:
  - High clock rates use more power;
  - No point having very fast clock rate if memory can't keep up.
- Today's commercial processors use 2, 4, 6, or 8 processing cores within the same integrated circuit.
- The operating system regards each CPU core as a logical processor.



Multiple processing cores (cont'd)

- Each core has cache memory (L1 cache) and shares cache with other cores (L2 cache).
- **Thread-level parallelism**.
- Performance improvement only if software spreads workload across multiple cores by running different threads on each core.
- In practice many applications can be multi-threaded (e.g., graphics, multimedia, multi-tasking operating systems etc).



❓ **Is multi-core processor SISD, SIMD, MISD or MIMD?**

Summary

- Techniques for quantifying computer performance have been instrumental in driving the development of techniques such as RISC.
- RISC and pipelining are good ideas, now widely adopted.
- Pipelining is very simple in principle, but requires sophisticated fixes to avoid branch and data dependency hazards.
- Multiprocessor systems can lead to large performance improvement if the problem is amenable to parallelism.
- Current trend towards multi-core, not high clock rate.