07-Performance Issues

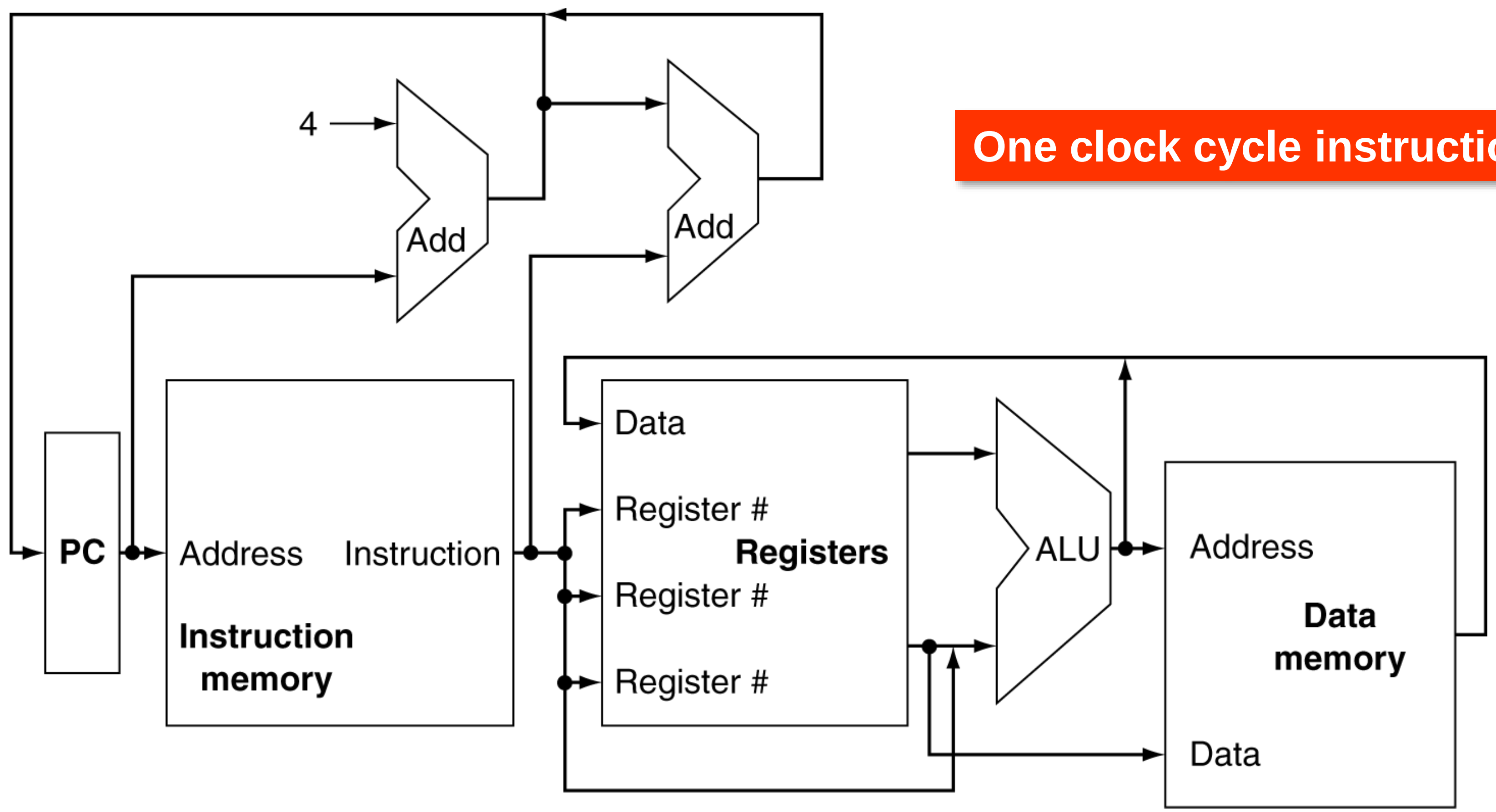
Ahmad Abdullah, PhD
abdullah@uvic.ca
https://web.uvic.ca/~abdullah/csc230

Lectures: MR 10:00 – 11:20 am
Location: ECS 125

Outline

- Pipelining
- Multicore
- Multithreading
- Amdahl’s Law
- Measures of performance

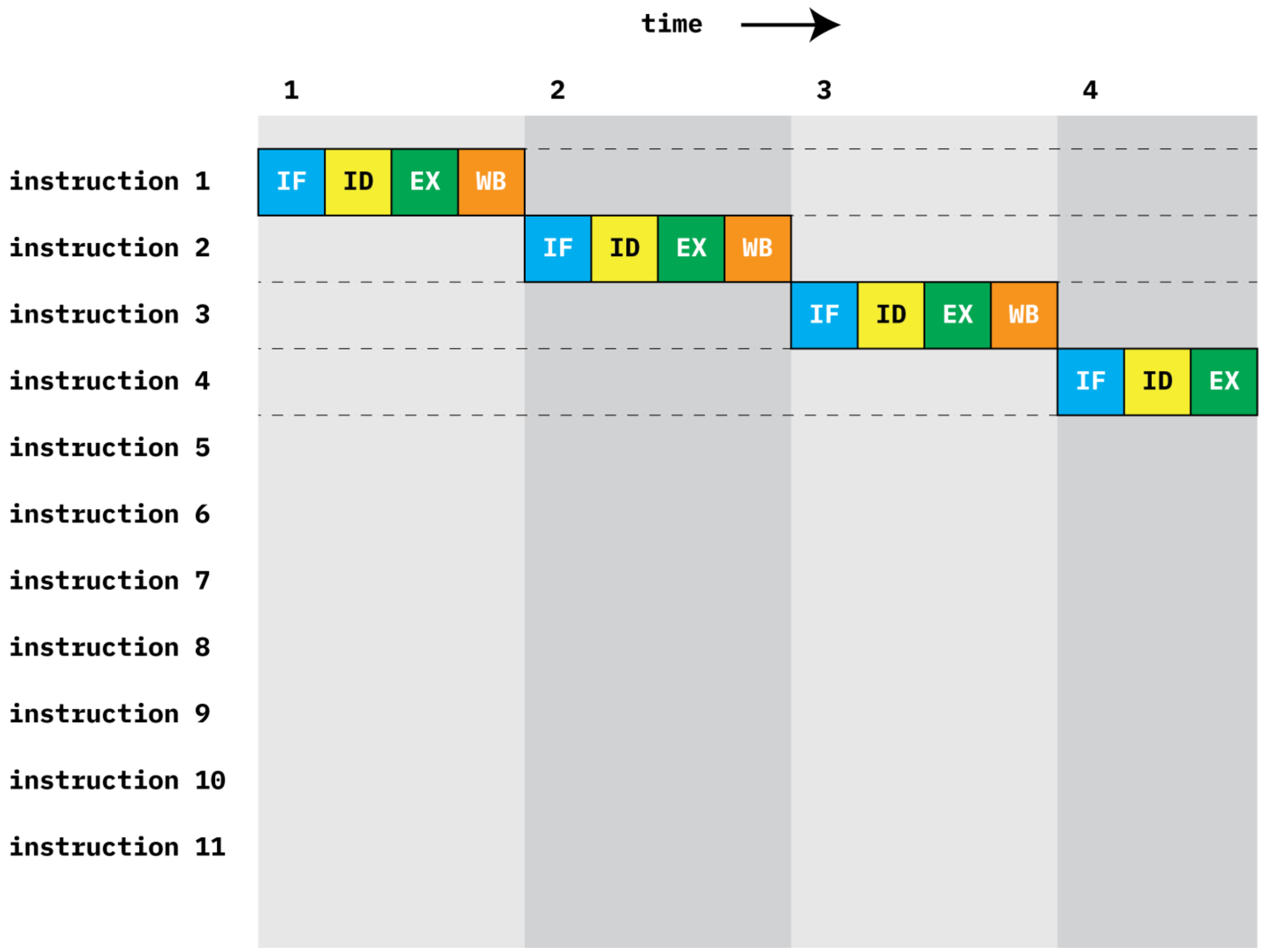
Instruction Execution Cycle



One clock cycle instruction phases

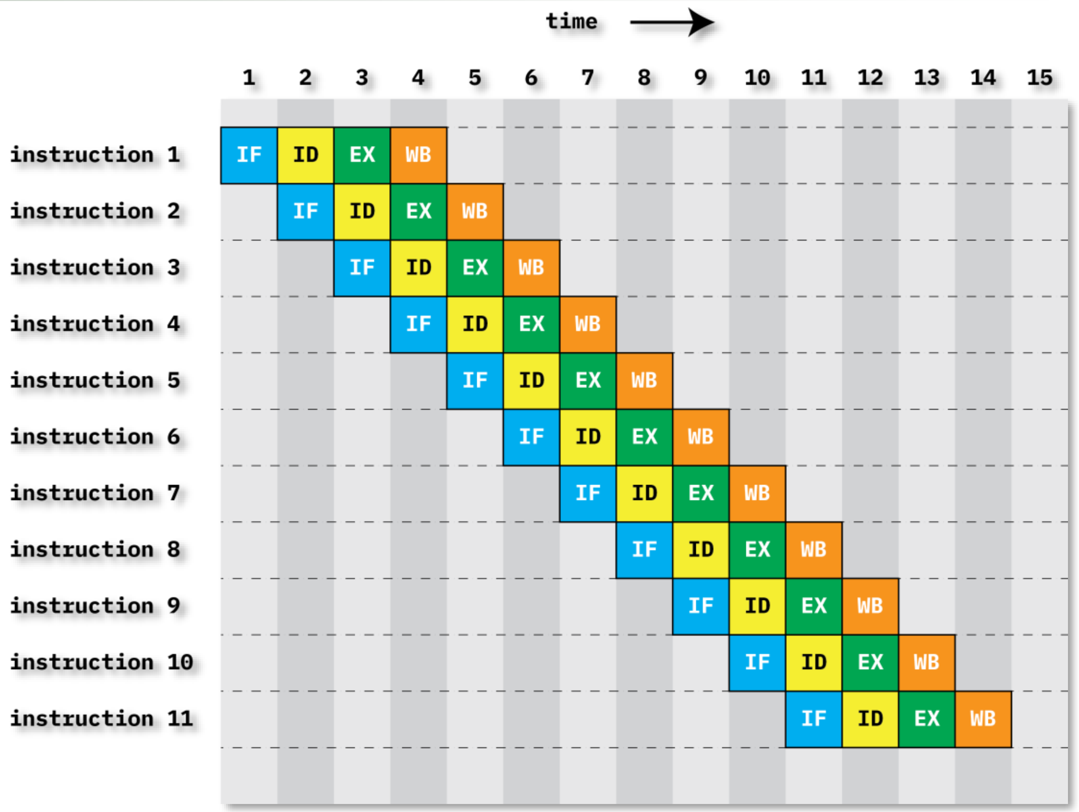
Instruction Fetch | Instruction Decode | Execute | Writeback

Program Execution



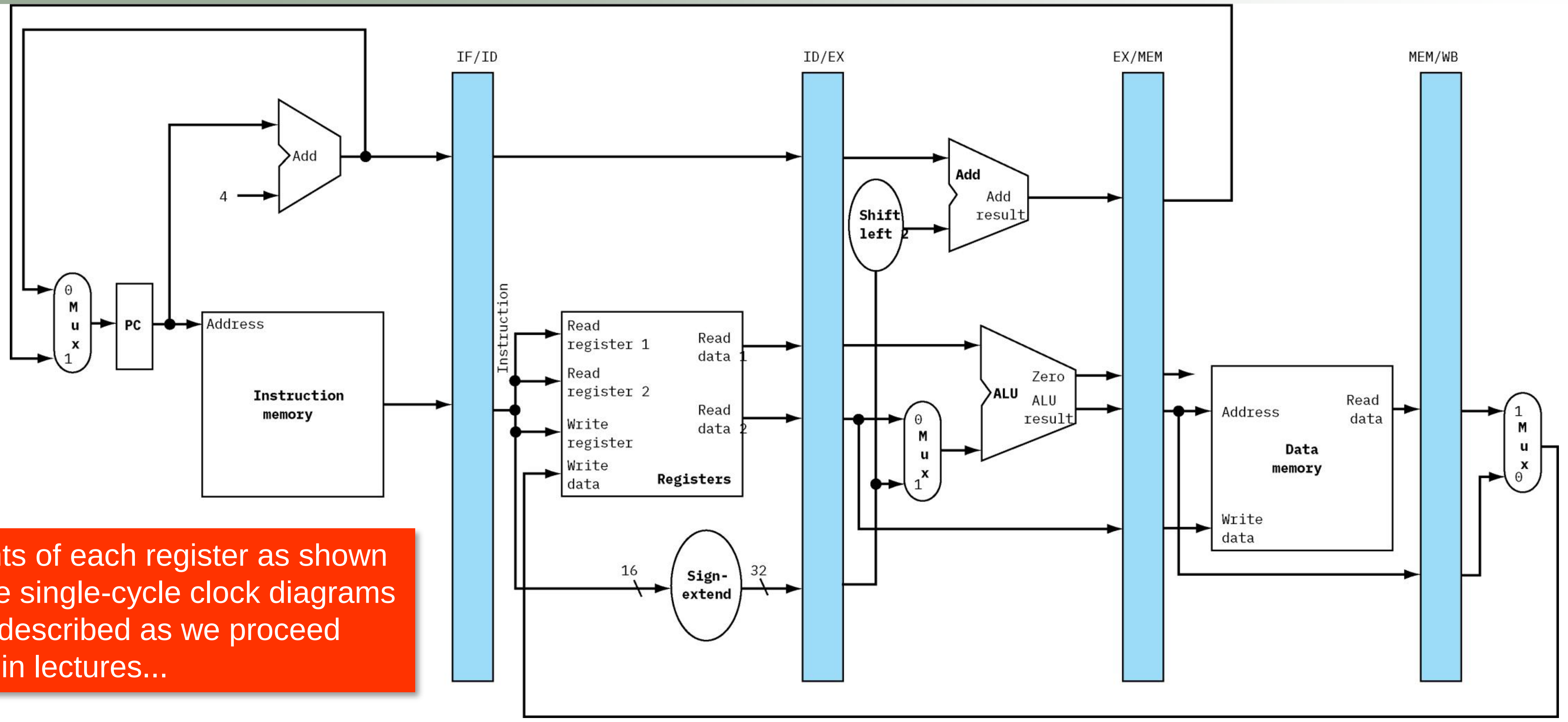
Pipelining

- Intuition:
  - like an assembly line
- Shrink cycle time (i.e., increase clock frequency)
- Now each phase fits within a clock cycle
- Different instructions at different stages can be worked on simultaneously (“instruction-level parallelism”)


time
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
instruction 1: IF ID EX WB
instruction 2: IF ID EX WB
instruction 3: IF ID EX WB
instruction 4: IF ID EX WB
instruction 5: IF ID EX WB
instruction 6: IF ID EX WB
instruction 7: IF ID EX WB
instruction 8: IF ID EX WB
instruction 9: IF ID EX WB
instruction 10: IF ID EX WB
instruction 11: IF ID EX WB

During cycle 8:
– instruction 5 is in its final phase
– instruction 6 is in its execute phase
– instruction 7 is in its decode phase
– instruction 8 has just been fetched

Pipeline Stages



Contents of each register as shown in these single-cycle clock diagrams will be described as we proceed further in lectures...

Fetch

Decode

Execute

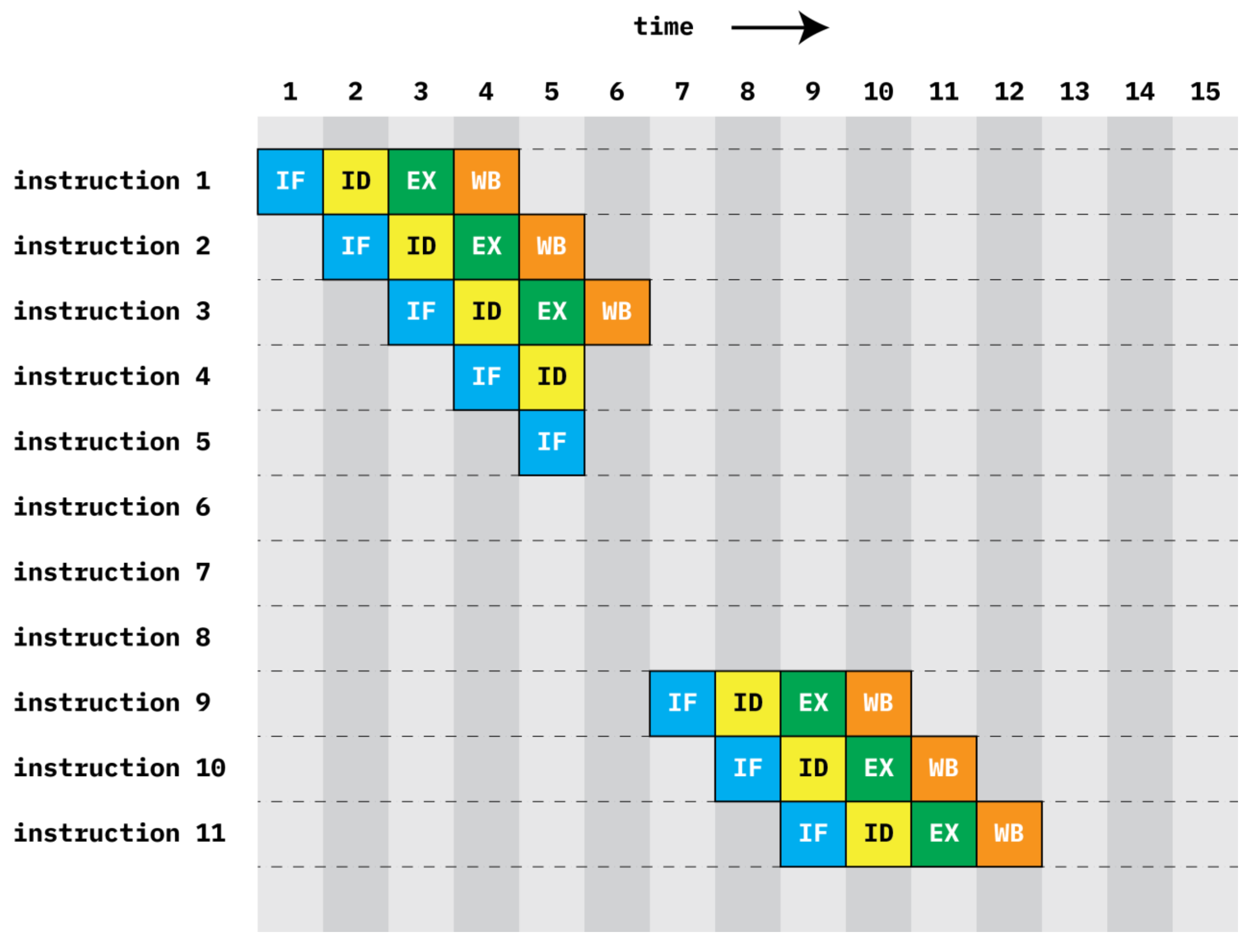
Writeback

Pipeline Hazards

- Control hazard:
  - branching issue
- Structural hazard:
  - more than one instruction in the pipeline needs the CPU resource in the same cycle
- Data hazard:
  - instruction in earlier stage requires data that will be the result of a instruction in later stage (i.e., results only available after writeback phase)

Branching Issue

- If a branch is taken, then work for instructions still in the pipeline following the branch instruction is discarded
- Instruction 3 was a branch...
- ... and the branch is taken (i.e., to instruction 9)



Branching Prediction

- **Idea:** use actual program behavior to predict future behavior
- Processor keeps track of branch history
  - If CPU reaches branch instruction...
  - ... it looks up the instruction address in a table
  - ... then fetches the state of the branch
- If the state is “branch normally taken”
  - then target address fetched

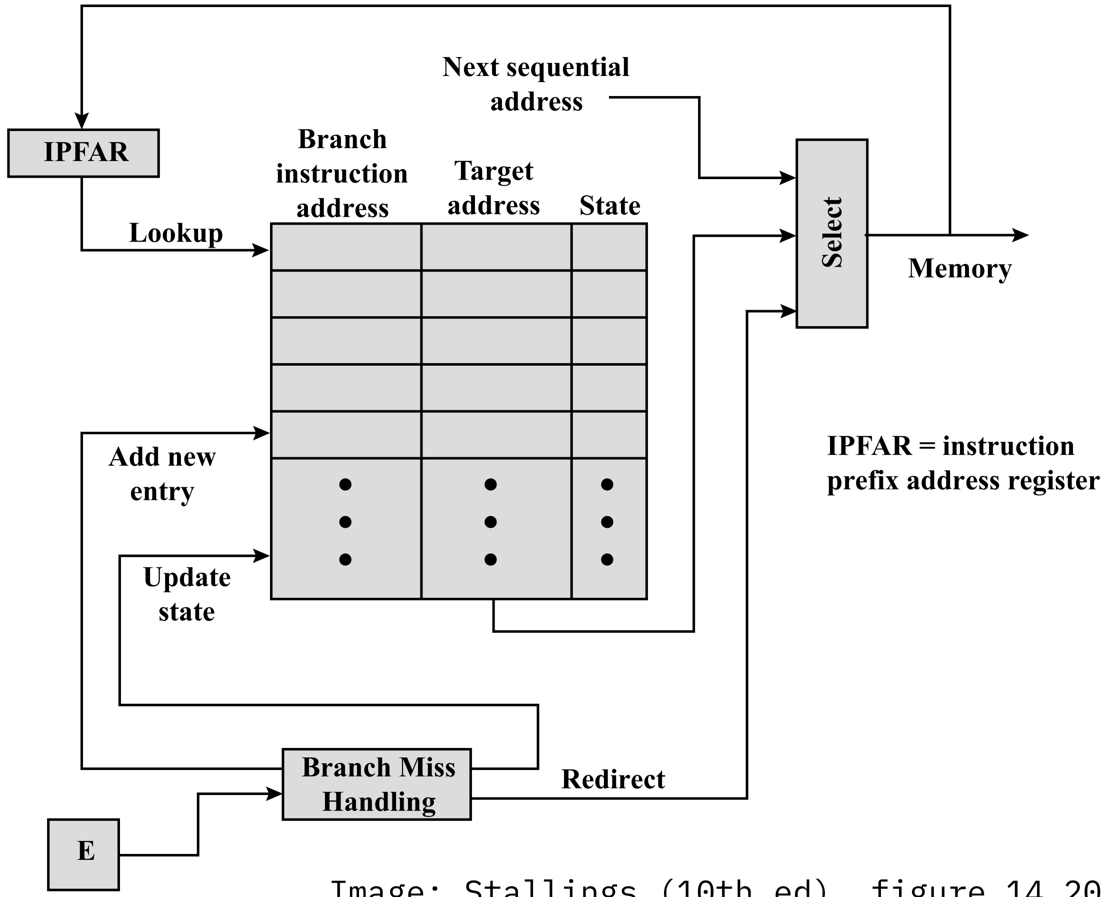


Image: Stallings (10th ed), figure 14.20

Branching Prediction (Cont.)

- If the branch is correctly predicted:
  - ... then the instructions fetched from memory based on that prediction (i.e., taken or not taken)...
  - ... and fed into the pipeline ...
  - ... will not result in wasted work.
- Of course, sometimes the prediction will be wrong
  - Much work has been done on increasing prediction accuracy
  - Example: using more and more history to generated branch prediction
  - (At some point, predictions must be wrong, otherwise we wouldn't need to have branches in the first place!)

Superscalar Execution

- This departs from basic pipelining in an important way.
- More than one instruction is fetched on the same clock cycle
  - In essence, multiple parallel pipelines

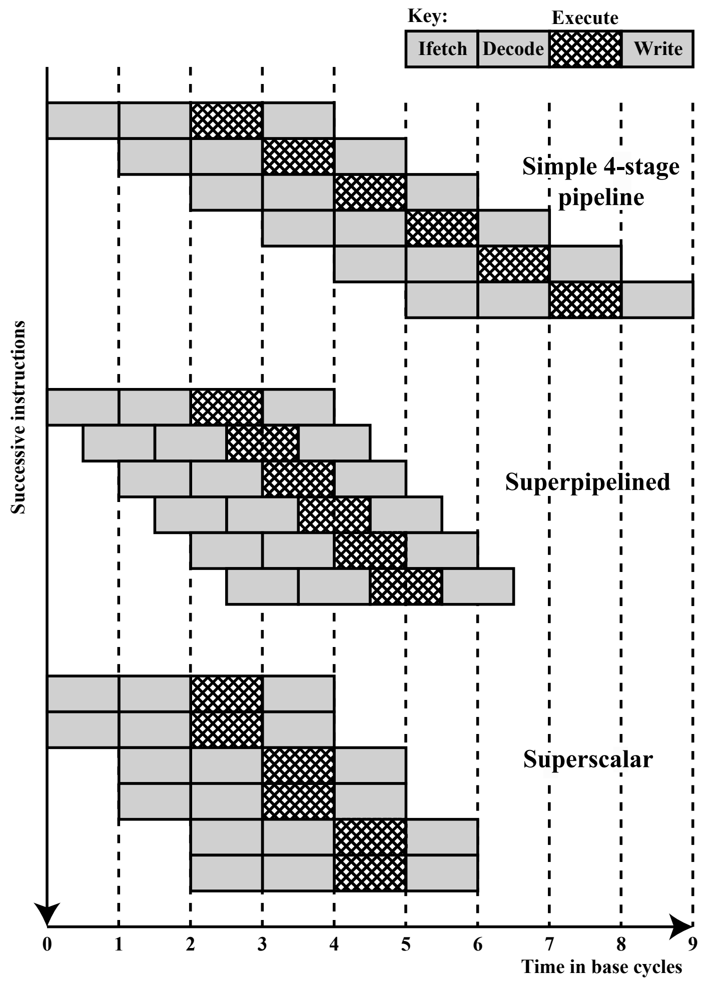


Image: Stallings (10th ed), figure 16.2

Data flow Analysis

- Processor determines possible equivalent orderings of instructions:
  - Ultimate goal: choose an order that executes the fastest
- **Data flow**: analyzing code to discover how the results of computations are used as inputs/operands for later computations

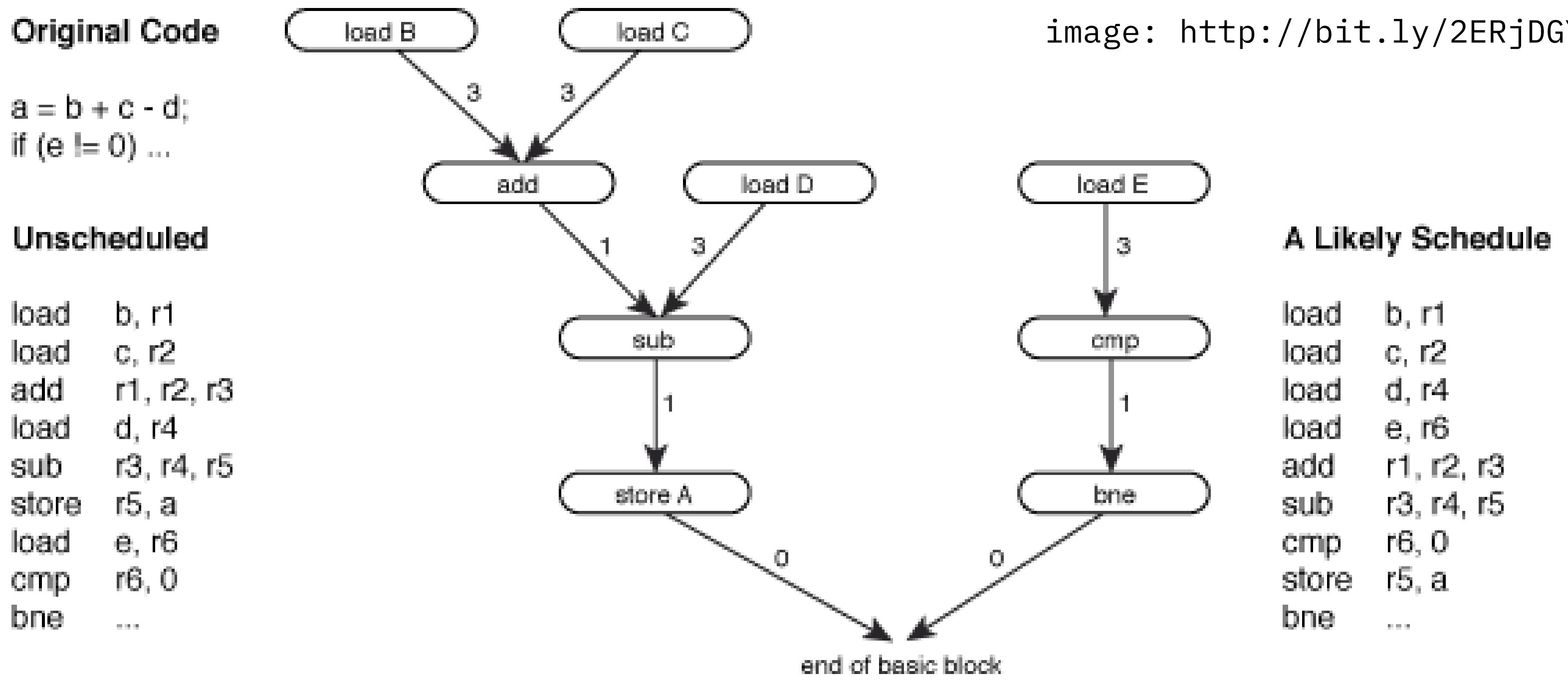


image: http://bit.ly/2ERjDGY

Original Code

```
a = b + c - d;
if (e != 0) ...
```

Unscheduled

```
load    b, r1
load    c, r2
add     r1, r2, r3
load    d, r4
sub     r3, r4, r5
store   r5, a
load    e, r6
cmp     r6, 0
bne     ...
```

A Likely Schedule

```
load    b, r1
load    c, r2
load    d, r4
load    e, r6
add     r1, r2, r3
sub     r3, r4, r5
cmp     r6, 0
store   r5, a
bne     ...
```

Speculative Execution

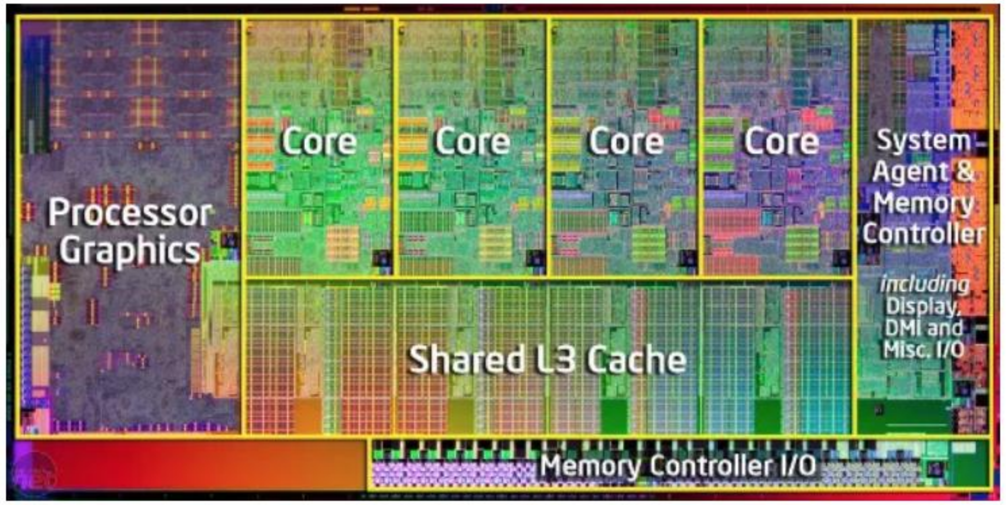
- This is a combination of:
  - branch prediction; plus
  - data-flow analysis
- Idea:
  - Perform some computations far ahead of time (i.e., speculatively)
  - This is done before we know they will be used...
  - ... although branch prediction helps increase the odds.
  - If results are not needed, then they are simply discarded.
- The observation here is that:
  - mainline code + speculative code can be evaluated in parallel

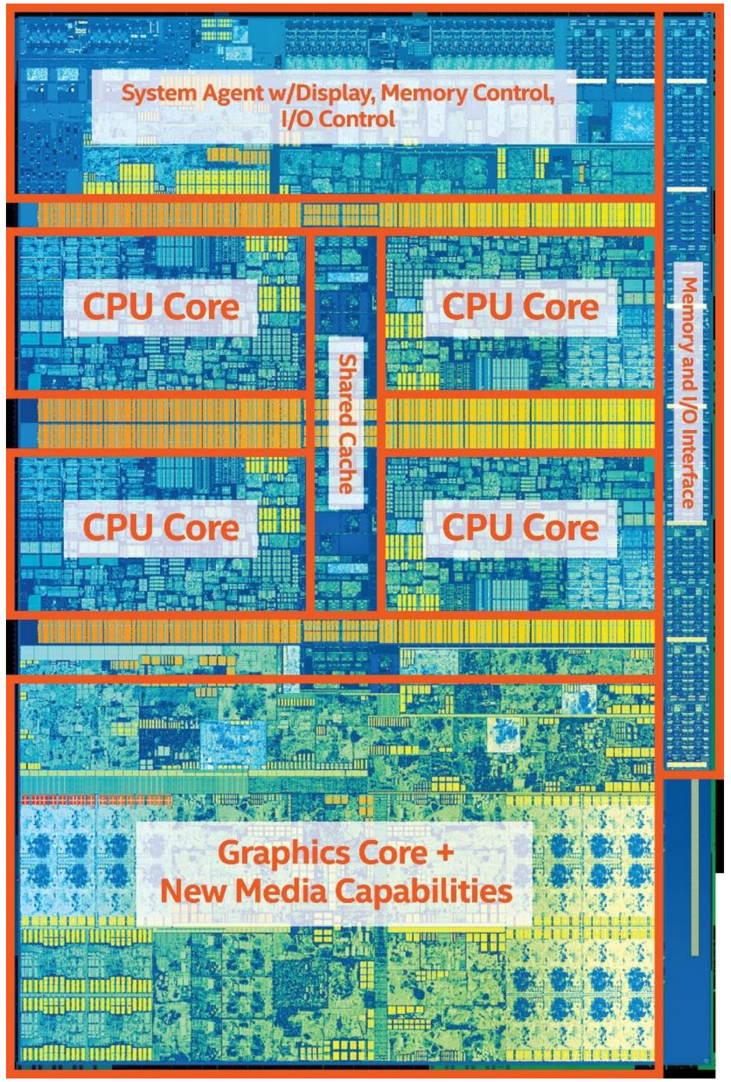
Spectre!

Meltdown!

Multicore

- Multiple cores == multicore
  - simply put more CPUs onto the same die!
- Different tasks can be executed on a different CPU cores (Multitasking)
- Real parallel execution
- One mechanism for this: **multithreaded code**

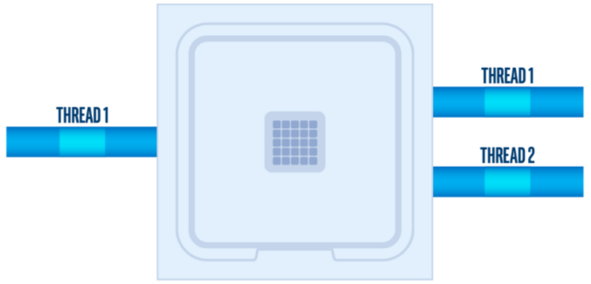



System Agent w/Display, Memory Control, I/O Control
CPU Core
CPU Core
CPU Core
CPU Core
Shared Cache
Memory and I/O Interface
Graphics Core + New Media Capabilities

Images: http://cnet.co/2EHdIRu, http://bit.ly/2BEzJ4V

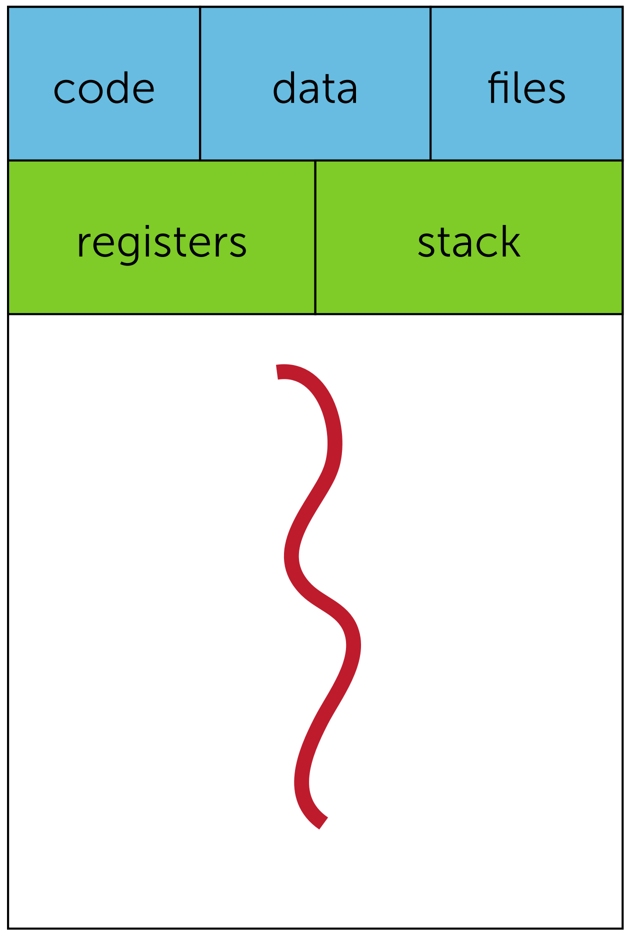
Multicore (Cont.)

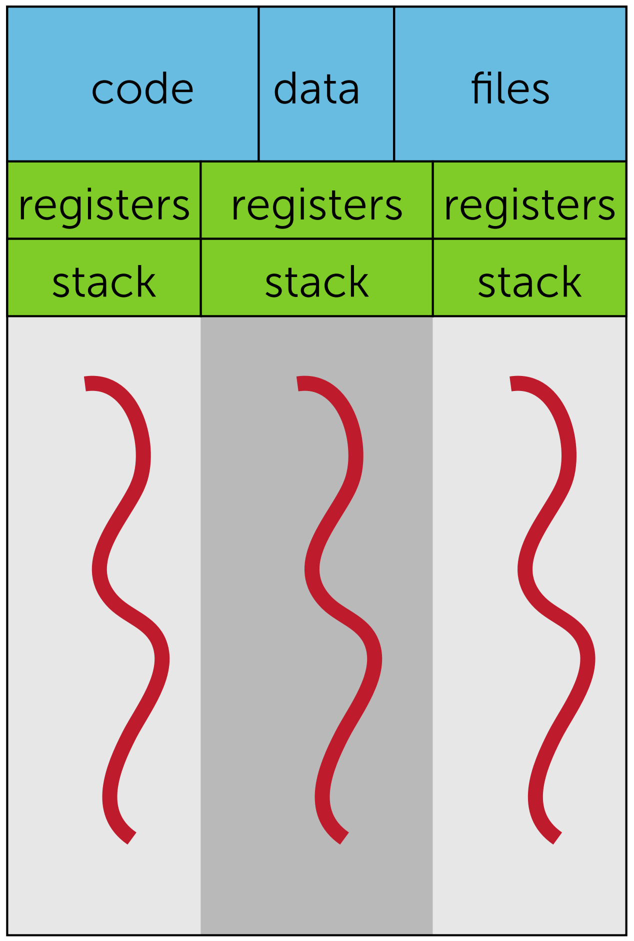
- Each core is able to run an independent sequence of instructions from other cores
  - Some of the onchip DRAM cache is shared amongst cores (Level 2)
  - ... but some cache is not (Level 1)
  - More about this later in the course when we examine the memory hierarchy
- Intel also introduced **hyperthreading**
  - Term is a bit confusing as it is not under programmer control (like regular threads)
  - Processor itself is able to make a single core look like more than one
  - **This is done without the intervention from the software or from a software developer!**



Hyper-Threading Technology allows the CPU exposes two execution contexts per physical core.

Single- vs. Multi-threaded





Speedup Analysis

- Consider a simpler example: parallel (i.e., multiple cores)
- Question:
  - Given some program P...
  - ... that takes time T to execute on a single core ...
  - ... what is the **greatest speedup possible** when using N processors/cores
- Assumption:
  - We have programs for which some fraction of the code can be into a parallel form
  - We name this fraction f such that $0 \le f \le 1$
  - Therefore, the fraction of the code which **cannot be parallelized** is $(1-f)$

Amdahl’s Law

- This can be applied to evaluate any design or technical improvement

$$\text{speedup} = \frac{\text{execution time } \textbf{before} \text{ enhancement}}{\text{execution time } \textbf{after} \text{ enhancement}}$$

- Assumption here is that we can always characterize the fraction of code that can be enhanced (i.e., f)

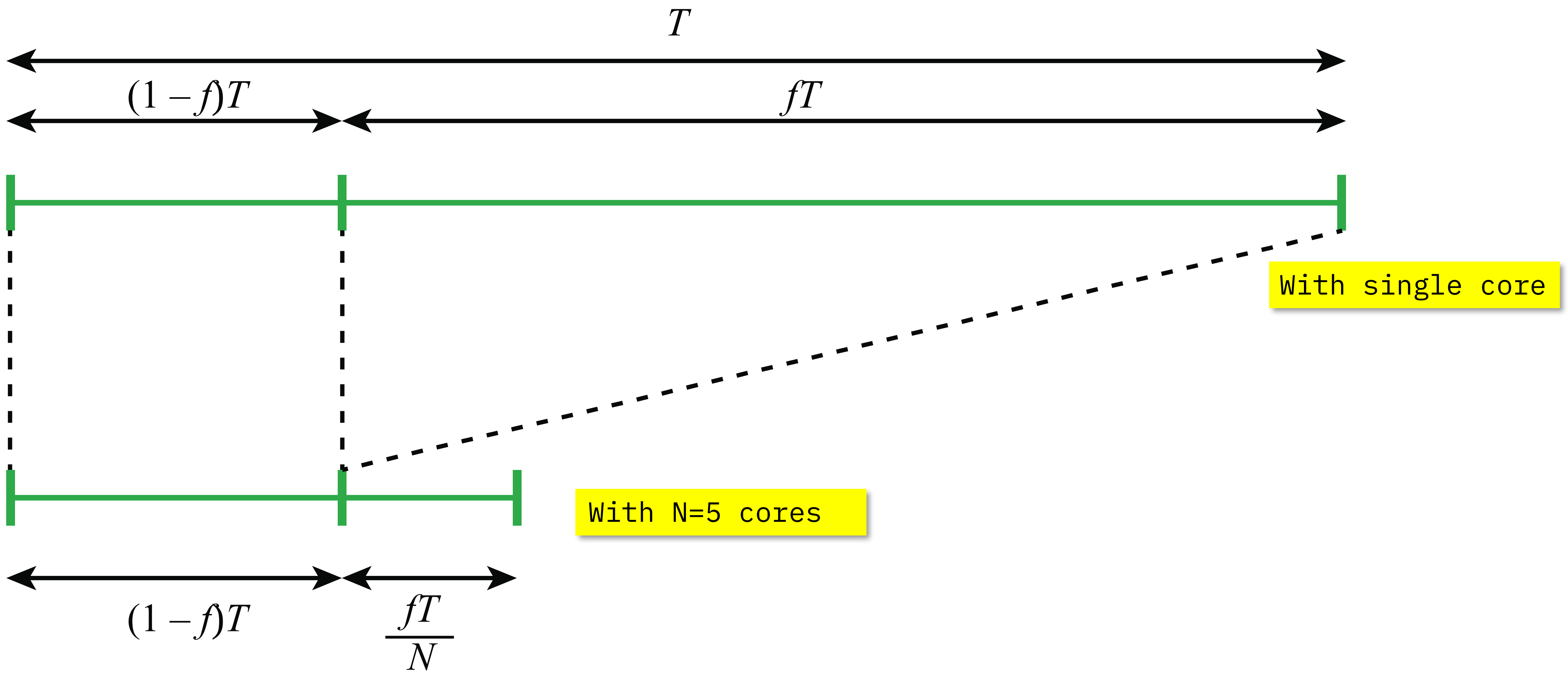
Possible benefits from multiple cores

- If we have a single core:
  - Time to execute: T

- If the whole program is parallelizable (i.e., $f = 1.0$) and we have N cores:
  - New time to execute: $\frac{T}{N}$

- If part of the program is parallelizable ($0 < f < 1.0$), and we have N cores:
  - New time to execute: $T(1 - f) + \frac{Tf}{N}$

Possible benefits from multiple cores (Cont.)



Possible benefits from multiple parallel

- The ratio of **pre-improvement time** to **post-improvement time** is the **speedup** of the improvement
- Speedup = $\frac{T}{T(1-f)+\frac{Tf}{N}} = \frac{1}{(1-f)+\frac{f}{N}}$
- Note:
  - When f is small, speedup is negligible
  - As $N \to \infty$, speedup is bound by $1/(1-f)$
- These equations and observations are collectively known as **Amdahl's Law**

Possible benefits from multiple parallel (Cont.)

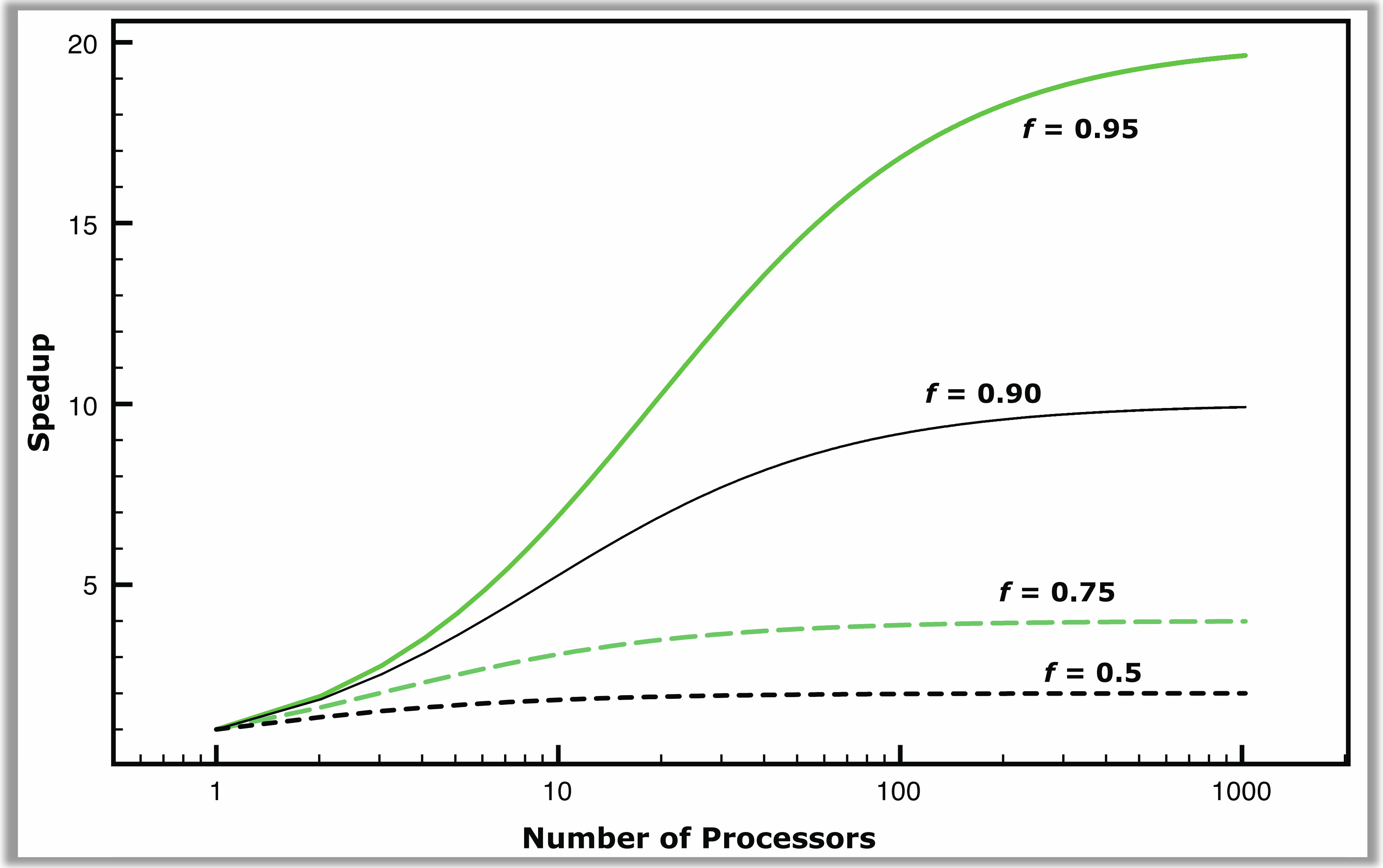


Image: Stallings (10th ed), figure 2.4

Measures of computer performance

- Warning:
  - In the previous slides we used the symbol f to mean a fraction of code.
  - We re-use the symbol f in what follows, but to mean a different kind of quantity
  - Sorry, but that's the nature of the field
- Recall:
  - Processor's operations driven by the clock that runs at certain frequency f
  - Time between clock pulses (cycle time): $\tau = \frac{1}{f}$
  - Each processor instruction requires a fixed number of cycles
  - Different types of instruction require different numbers of cycles

Given a specific program & architecture...

- I_C is the **instruction count**
  - number of machine instructions executed for that whole program until it runs to completion
- I_i is the count of instructions executed of type i
- CPI_{i_i} is the number of cycles needed to execute an instruction of type i

- Overall $CPI = \frac{\sum_{i=1}^{n}(CPI_i \times I_i)}{I_c}$

- Time to execute a given program:

$$T = I_C \times CPI \times \tau$$

Given a specific program & architecture...

- Suppose we had four instruction classes:
  - ALU
  - Memory (load, store)
  - Branches (conditionals)
  - Jumps (unconditional)

- Overall CPI $= \dfrac{(CPI_{ALU} \times I_{ALU}) + (CPI_{MEM} \times I_{MEM}) + (CPI_{BR} \times I_{BR}) + (CPI_{J} \times I_{J})}{I_C}$

Slight re-writing of CPI

- Our formulation of CPI does not make explicit the effects of memory latency
- We can re-write the equation to bring out this detail:
  - cycles to decode and execute instruction: p
  - number of memory references: m
  - ratio of memory-cycle time to processor-cycle time (i.e., how much slower or faster memory is compared to CPU): k
- $T = I_C \times [p + (m \times k)] \times \tau$

MIPS

- **Millions of instructions per second**
  - Common measure
  - (It is, however, a bit misleading)
- $\text{MIPS rate} = \frac{I_c}{T \times 10^6} = \frac{f}{CPI \times 10^6}$
- Example:
  - 400 MHz processor
  - Instruction type 1: Arithmetic & Logic (CPI = 1, 60% of instructions)
  - Instruction type 2: Load/store from cache (CPI = 2, 18% of instructions)
  - Instruction type 3: Branch (CPI = 4, 12% of instructions)
  - Instruction type 4: Load/store from outside cache (CPI=8, 10% of instructions)
  - $CPI = 0.6 + (2 \times 0.18) + (4 \times 0.12) + (8 \times 0.1) = 2.24$
  - MIPS rate = $\frac{400 \times 10^6}{2.24 \times 10^6} \cong 178$

Any Questions?