Spring 2022 (Sections 6 & 7)

CSE-332/EEE-336:
Computer Organization and Architecture

Lecture#2: Computer Performance

Mainul Hossain, Ph.D.

Assistant Professor

Department of Electrical Engineering and Electronic Engineering

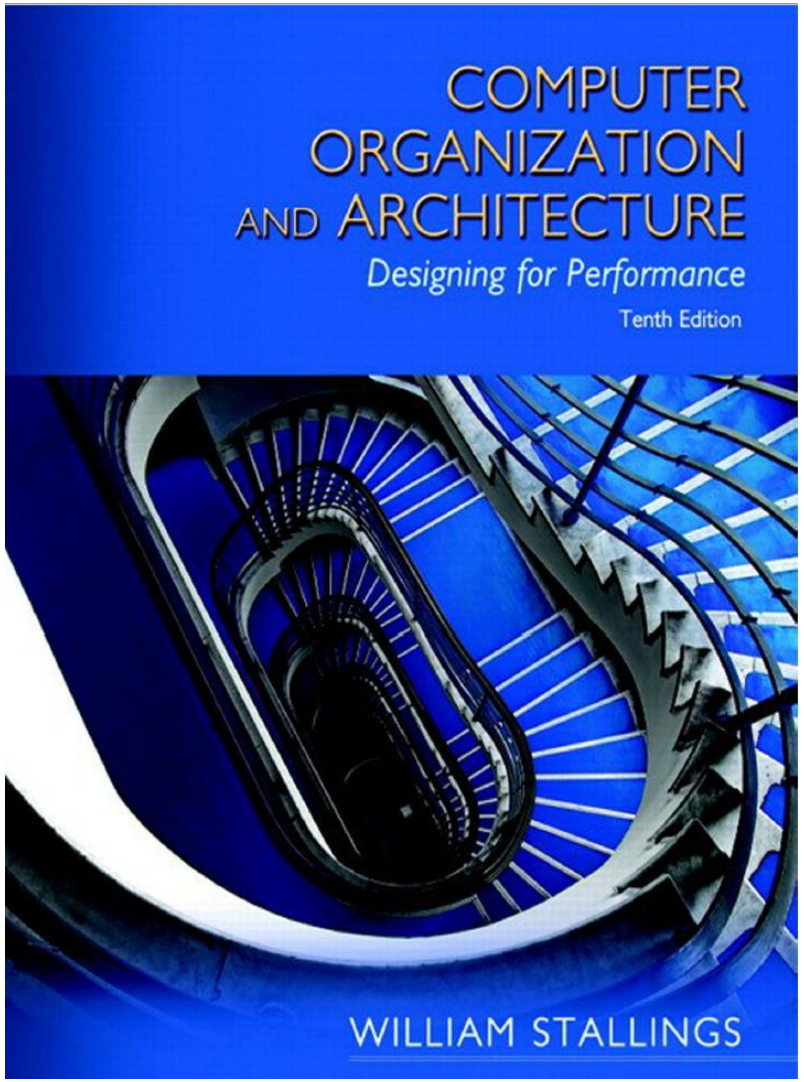
University of Dhaka

mainul.eee@du.ac.bd

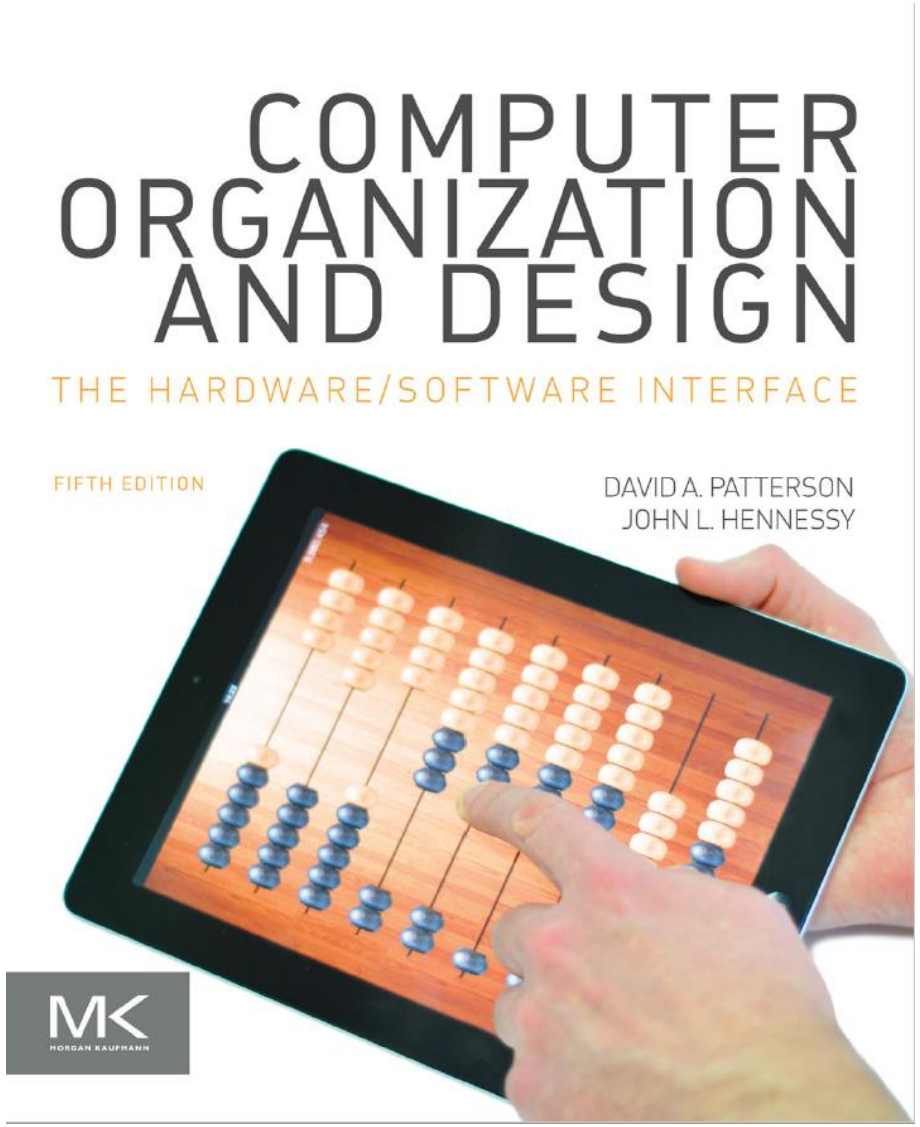
sites.google.com/du.ac.bd/mainulgroup

Reading

Chapter-2



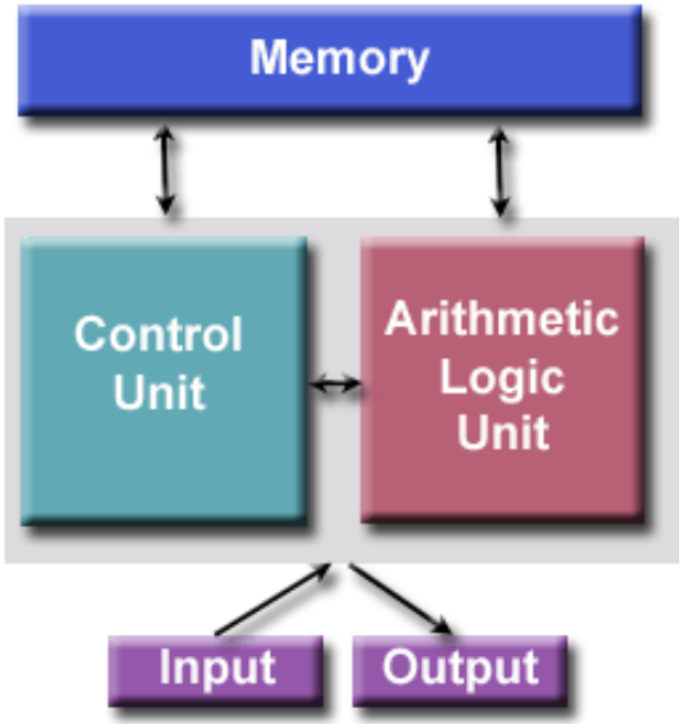
Chapter-1.6



Von Neumann Architecture: Limitations

- The Von Neumann architecture consists of a single, shared **memory** for programs and data, a single bus for **memory** access, an arithmetic unit, and a program control unit. The Von Neumann processor operates fetching and execution cycles seriously.

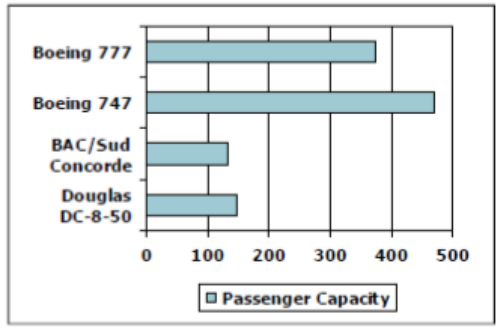
- Disadvantage of Von Neumann architecture: shared memory for instructions and data with one data bus and one address bus between processor and memory. Instructions and data have to be fetched in sequential order (known as the Von Neumann Bottleneck), limiting the operation bandwidth.

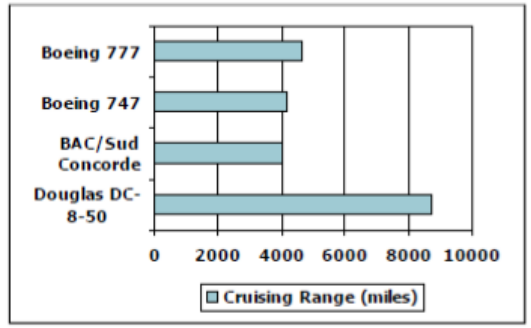


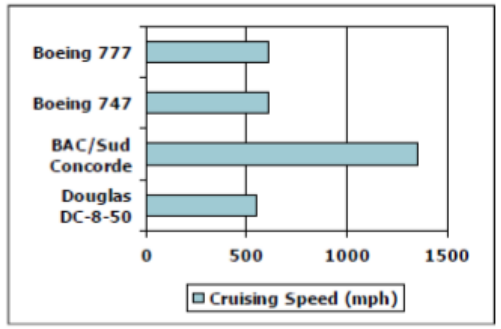
Performance

- *Performance is the key to understanding underlying motivation for the hardware and its organization*
- *Why is some hardware better than others for different programs?*
- *What factors of system performance are hardware related? (e.g., do we need a new machine, or a new operating system?)*

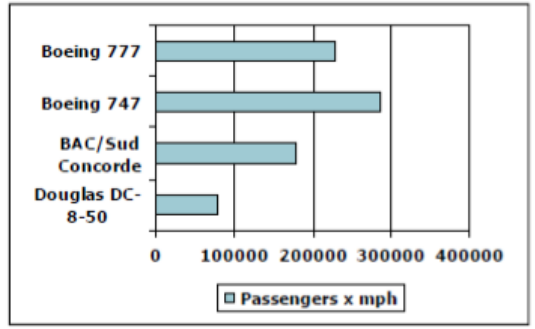
- Why is performance important?
  - For purchasers: to choose between computers
  - For designers: to make the sales pitch
- Defining performance is not straightforward!
  - An analogy with airplanes shows the difficulty







- Which airplane has the best performance?



Computer Performance: TIME! TIME! TIME !

- *Response Time* (*elapsed time*, *latency*):
  - How long does it take to complete (start to finish) *a task*?
  - Eg: how long must *I* wait for the database query?

Individual user concerns...

Individual is more interested in response time. As a user of a smart phone/laptop, the one that responds faster is better!

Response time (computer): the total time required by computer to complete a task including :

Disk access *Memory access* *I/O activities* *OS overheads* *CPU exec. time* *etc*

Computer Performance: The Enablers

Application of the following great ideas has accounted for much of the tremendous growth in computing capabilities over the past 50 years.

- Design for Moore's law
- Use abstraction to simplify design
- ✓ Make the common case fast
- ✓ Performance via parallelism
- ✓ Performance via pipelining
- ✓ Performance via prediction
- Hierarchy of memories
- ✓ Dependability via redundancy

Computer Performance: Make the Common Case Fast

- *In computer design, **favor the frequent** case over the infrequent case.*
- Improve frequent event than the rare event

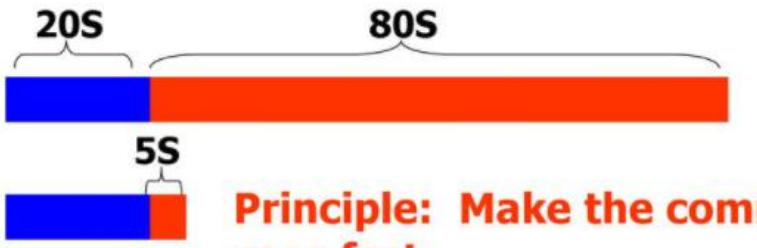
Example

- Frequent operations: Memory Read/Write
- Infrequent Operations: Floating point Multiplication

- *The most significant improvements in computer performance come from improvements to the common case, i.e. computations that are commonly executed, rather than optimizing the rare case. Rare cases are not encountered often and therefore, it does not matter even if the processor is not optimized to execute that and spends more time. Ironically, the common case is often simpler than the rare case and hence is often easier to enhance. The common case can be identified with careful experimentation and measurement.*

- Suppose a program runs in 100 seconds on a machine, with multiply responsible for 80 seconds of this time.

- How much do we have to improve the speed of multiplication if we want the program to run 4 times faster?



Computer Performance: Performance via Parallelism

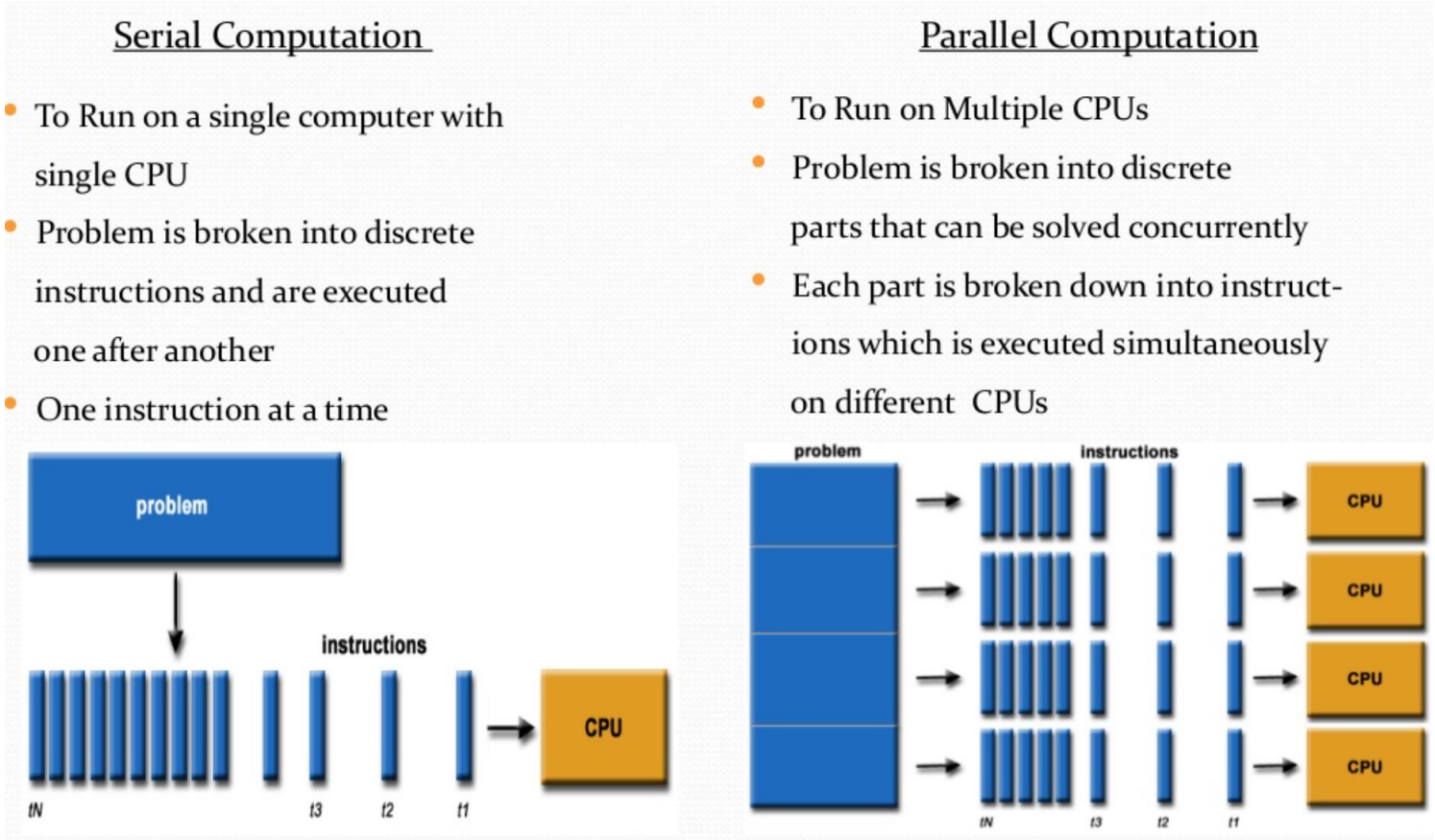
Doing different parts of a task in parallel accomplishes the task in less time than doing them sequentially.

A processor engages in several activities in the execution of an instruction. It runs faster if it can do these activities in parallel.

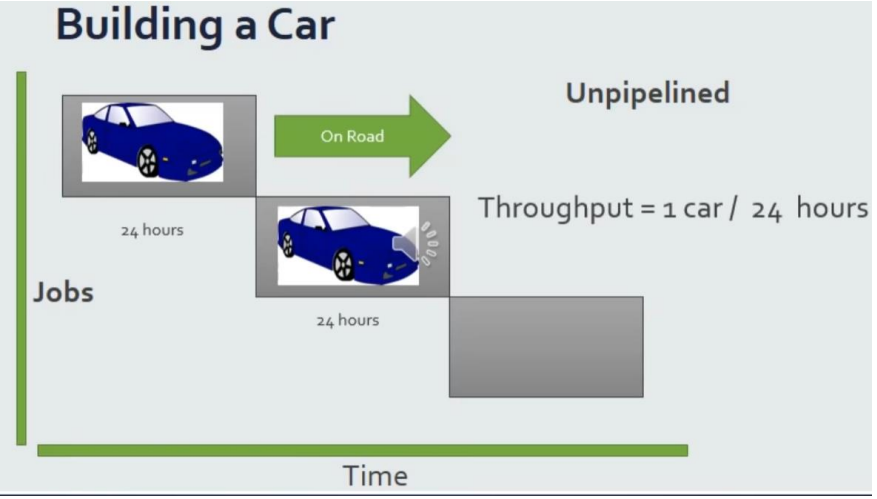
Where possible, make operations perform in parallel.

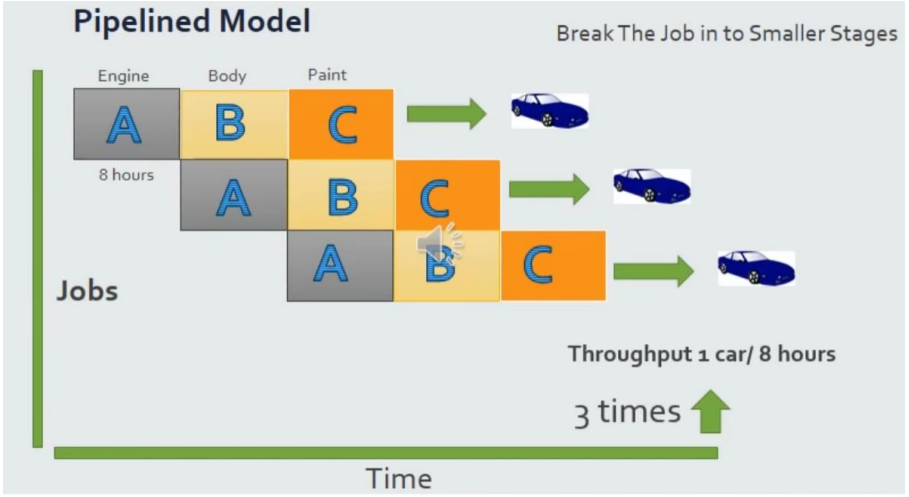
Results in multi-core processors, many-core processors (GPUs, TPUs), cluster-computing, etc.

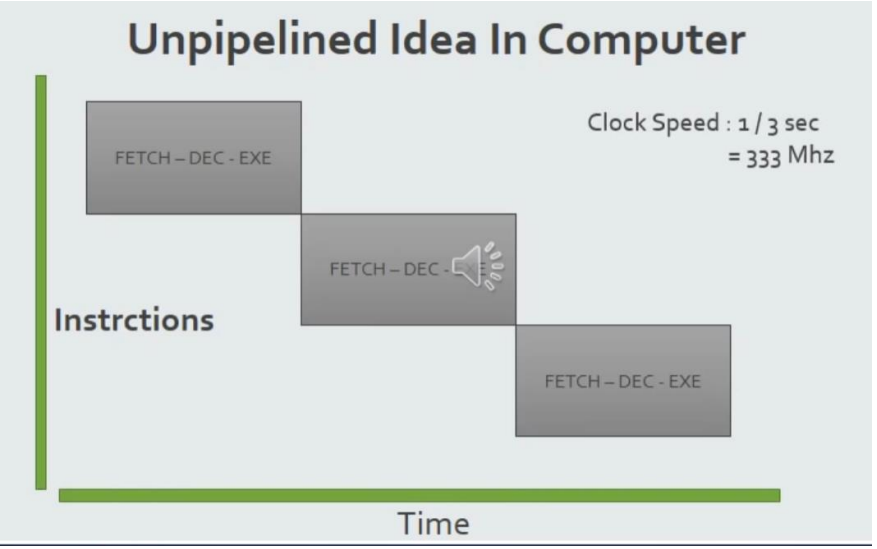
- One way to improve performance is to do more processing at once.
- There were several examples of this in our CPU designs.
  - Multiple functional units can be included in a datapath to let single instructions execute faster. For example, we can calculate a branch target while reading the register file.
  - Pipelining allows us to overlap the executions of several instructions.
  - SIMD performance operations on multiple data items simultaneously.
  - Multi-core processors enable thread-level parallel processing.
- Memory and I/O systems also provide many good examples.
  - A wider bus can transfer more data per clock cycle.
  - Memory can be split into banks that are accessed simultaneously. Similar ideas may be applied to hard disks, as with RAID systems.
  - A direct memory access (DMA) controller performs I/O operations while the CPU does compute-intensive tasks instead.

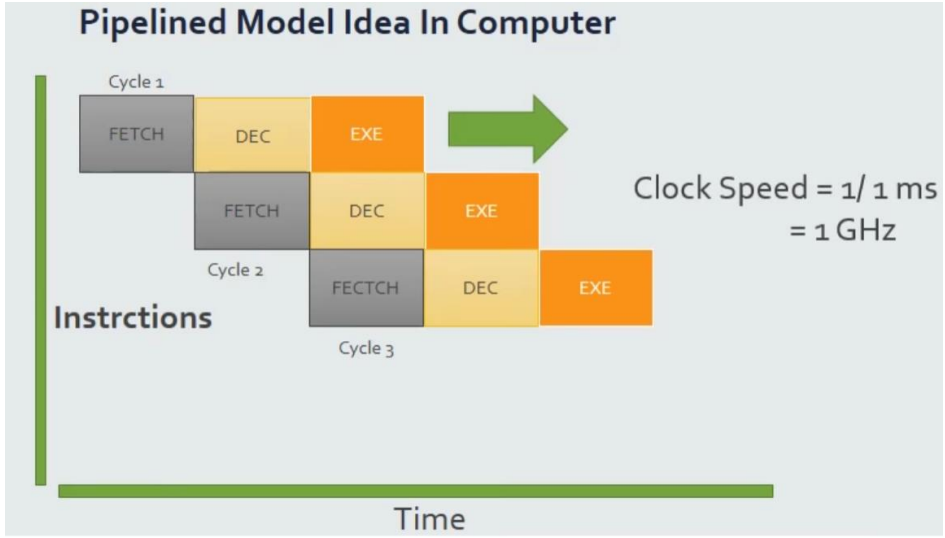


Computer Performance: Performance via Pipelining





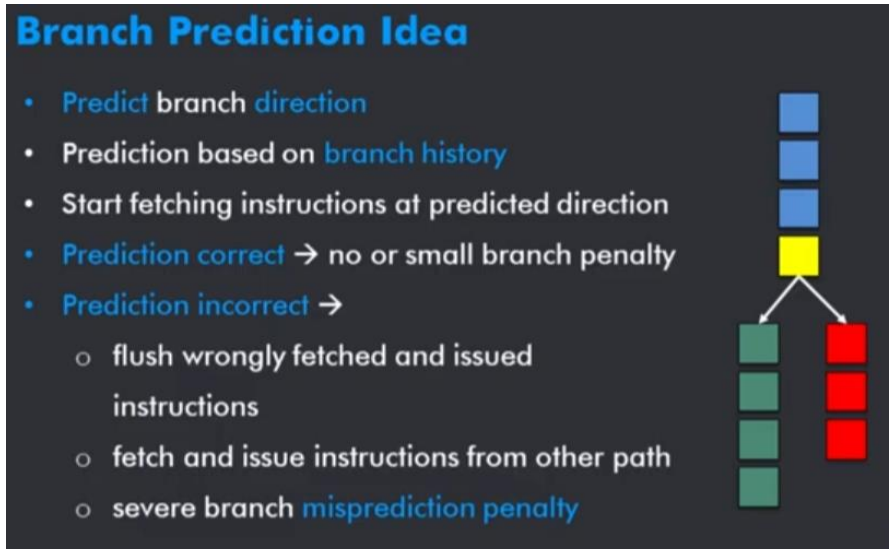




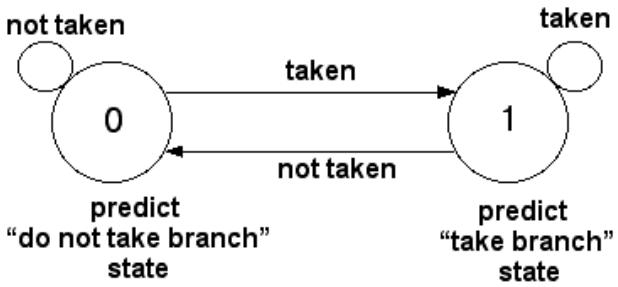
This idea is an extension of the idea of parallelism. It is essentially handling the activities involved in instruction execution as an assembly line. As soon as the first activity of an instruction is done you move it to the second activity and start the first activity of a new instruction. This results in executing more instructions per unit time compared to waiting for all activities of the first instruction to complete before starting the second instruction.

Computer Performance: Performance via Branch Prediction

- A conditional branch is a type of instruction determines the next instruction to be executed based on a condition test. Conditional branches are essential for implementing high-level language if statements and loops.
- Unfortunately, conditional branches interfere with the smooth operation of a pipeline — the processor does not know where to fetch the next instruction until after the condition has been tested.
- Many modern processors reduce the impact of branches with speculative execution: make an informed guess about the outcome of the condition test and start executing the indicated instruction. Performance is improved if the guesses are reasonably accurate and the penalty of wrong guesses is not too severe.



1-bit Branch Prediction: Example




```
for (i = 0; i<5; i++) {
        a+ =1
}
```

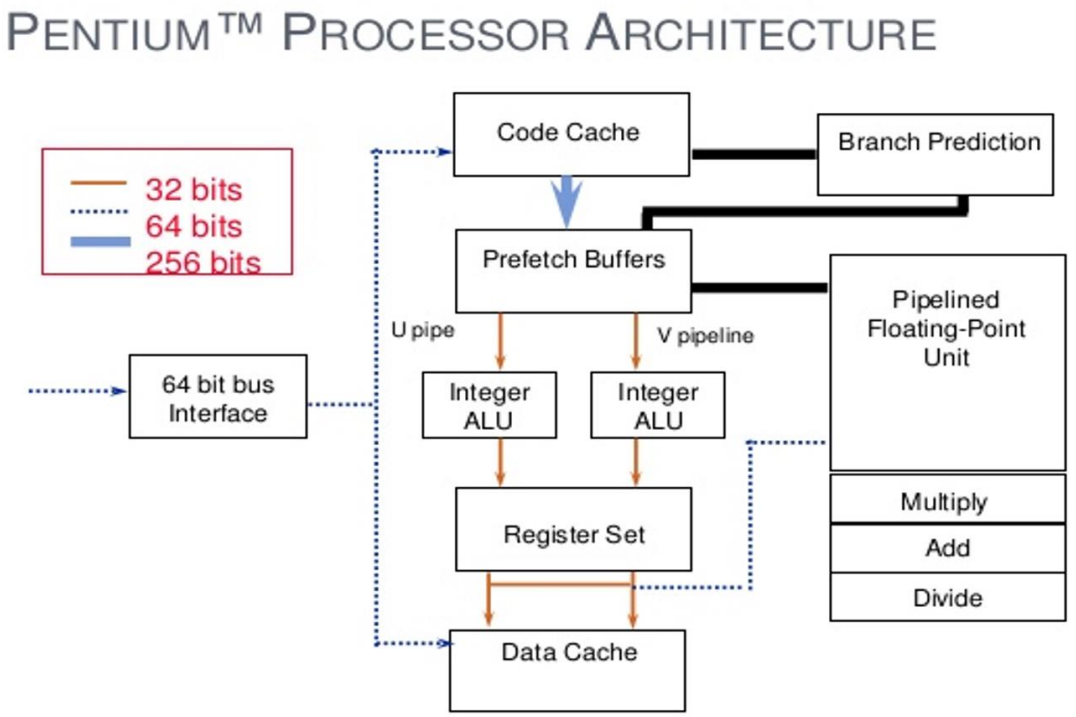
i: 0 1 2 3 4 5 0 1

T T T T T N T T

✓ ✓ ✓ ✓ ✓ ✗ ✗

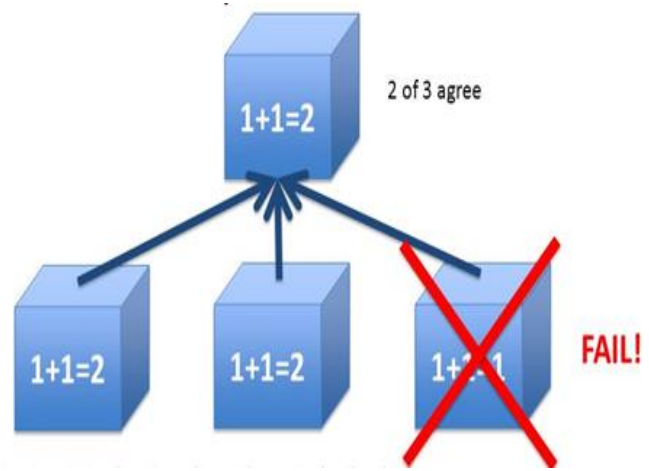
Computer Performance: Performance via Branch Prediction

- Branch prediction is another new feature of the Pentium.
- The penalty for jumping is very high for a high-performance pipelined microprocessor such as the Pentium.
- For example, in the case of the JNZ instruction, if it jumps, the pipeline must be flushed and refilled with instructions from the target location.
- This takes time. In contrast, the instruction immediately below the JNZ is already in the pipeline and is advancing without delay.
- The Pentium processor has the capability to predict and prefetch code from both possible locations and have them advanced through the pipeline without waiting (installing) for the outcome of the zero flag.
- The ability to predict branches and avoid the branch penalty combined with the instruction pairing can result in a substantial reduction in the clock count for a given program.



Dependability via Redundancy

One of the most important ideas in data storage is the Redundant Array of Inexpensive Disks (RAID) concept. In most versions of RAID, data is stored redundantly on multiple disks. The redundancy insures that if one disk fails the data can be recovered from other disks.



Dependability Measures

- Reliability: Mean Time To Failure (MTTF)
- Service interruption: Mean Time To Repair (MTTR)
- Mean time between failures (MTBF)
  - MTBF = MTTF + MTTR
- Availability = MTTF / (MTTF + MTTR) = MTTF / MTBF
- Improving Availability
  - Increase MTTF: More reliable hardware/software + Fault Tolerance
  - Reduce MTTR: improved tools and processes for diagnosis and repair

Reliability Measures

- MTTF, MTBF usually measured in hours
  - E.g., average MTTF is 100,000 hours
- Another measure is average number of failures per year
  - E.g., 1000 disks with 100,000 hour MTTF
  - 365 days * 24 hours = 8760 hours
  - (1000 disks * 8760 hrs/year) / 100,000 hrs = 87.6 failed disks per year on average
  - 87.6/1000 = 8.76% annualized failure rate (AFR)

Availability Measures

- Availability = MTTF / (MTTF + MTTR) as %
- Since hope rarely down, shorthand is "number of 9s of availability per year"
- 1 nine: 90% => 36 days of repair/year
- 2 nines: 99% => 3.6 days of repair/year
- 3 nines: 99.9% => 526 minutes of repair/year
- 4 nines: 99.99% => 53 minutes of repair/year
- 5 nines: 99.999% => 5 minutes of repair/year

Computer Performance: Throughput and Response Time

- *Throughput*:
  - Total work done per unit time……(per hr,day etc)
  - how *many* jobs can the machine run at once?
  - what is the *average* execution rate?
  - how *much* work is getting done?

Systems manager concerns…

Response Time (*latency*): how long it takes to complete a task

Throughput: total # of tasks completed per unit time

For now, we will focus on response time… and define

$$\text{Performance} = \frac{1}{\text{Execution Time}}$$

- *If we upgrade a machine with a new processor what do we increase?*

- How are response time and throughput affected by
  - Replacing the processor with a faster version?
  - Adding more processors?

Computer Performance: Relative Performance/Performance Ratio

- Define Performance = 1/Execution Time
- "X is n time faster than Y"

$$\text{Performance}_X / \text{Performance}_Y = \text{Execution time}_Y / \text{Execution time}_X = n$$

- Example: time taken to run a program
  - 10s on A, 15s on B
  - Execution Time_B / Execution Time_A = 15s / 10s = 1.5
  - So A is 1.5 times faster than B

Computer Performance: Execution Time

Execution Time

- ***Elapsed Time***
  - counts everything (*disk and memory accesses, waiting for I/O, running other programs, etc.*) from start to finish
  - **a useful number, but often not good for comparison purposes**

    Elapsed time = CPU time + wait time (I/O, other programs, etc.)

- ***CPU time***
  - **doesn't count waiting for I/O or time spent running other programs**
  - can be divided into *user CPU time* and *system CPU time* (OS calls)

    CPU time = user CPU time + system CPU time

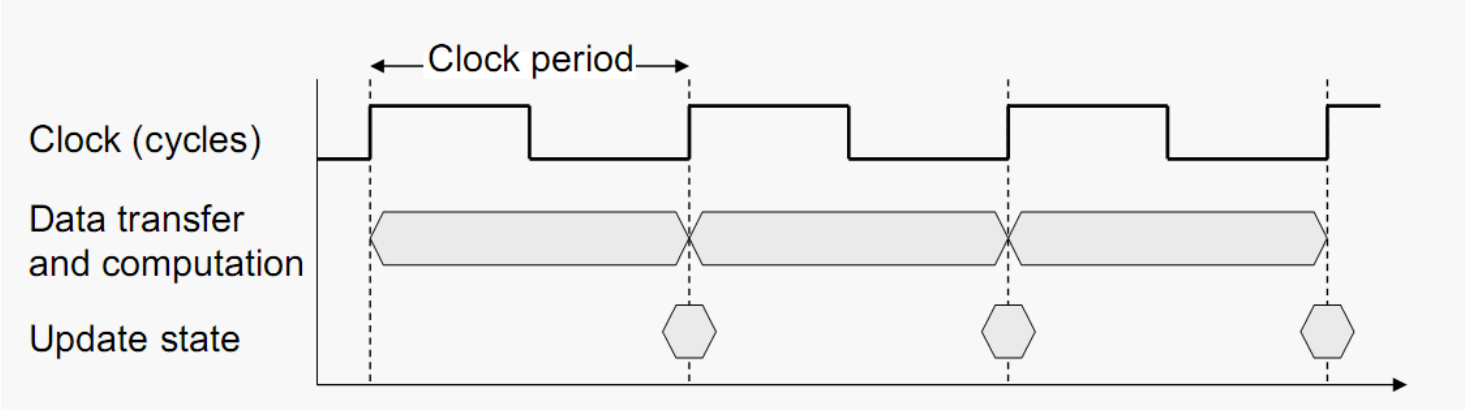
- Our focus:
- *user CPU time* (*CPU execution time* or, simply, *execution time*)
  - **time spent executing the lines of code that are *in our program***
  - *For easier writing, **user CPU time** has been termed simply as **CPU time** in rest of the studies.*

Let's formulize execution time!

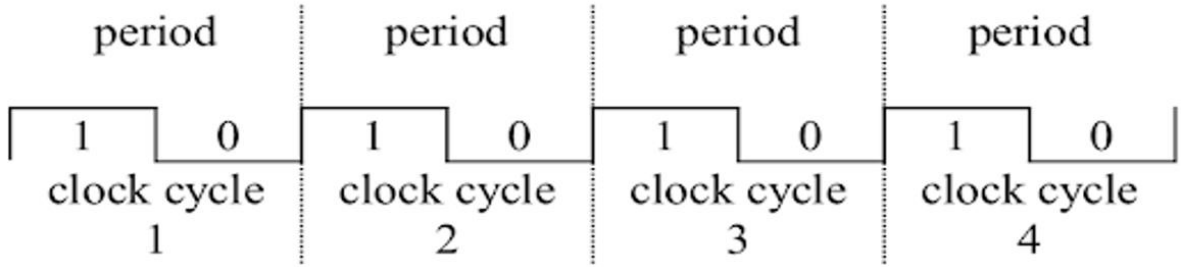
- ET: execution time (the time required to execute a program that is written in a given high level language)
- N: the total number of instructions executed including repeated instructions. Complete execution of the program requires the execution of N machine language instructions.
- S: the average number of basic steps (clocks) per instruction (technically called cycle per instruction (CPI)).
- R: the clock rate in cycles per second
- Then,

$$ET = \frac{\text{\# instructions X cycles per inst}}{\text{clock rate}} = \frac{N \times S}{R}$$

CPU Clocking



- Operation of digital hardware governed by a constant-rate clock
  - Clock period: duration of a clock cycle
    - e.g., 250ps = 0.25ns = 250×10^{-12}s
  - Clock frequency (rate): cycles per second
    - e.g., 4.0 GHz = 4000 MHz = 4.0×10^{9}Hz

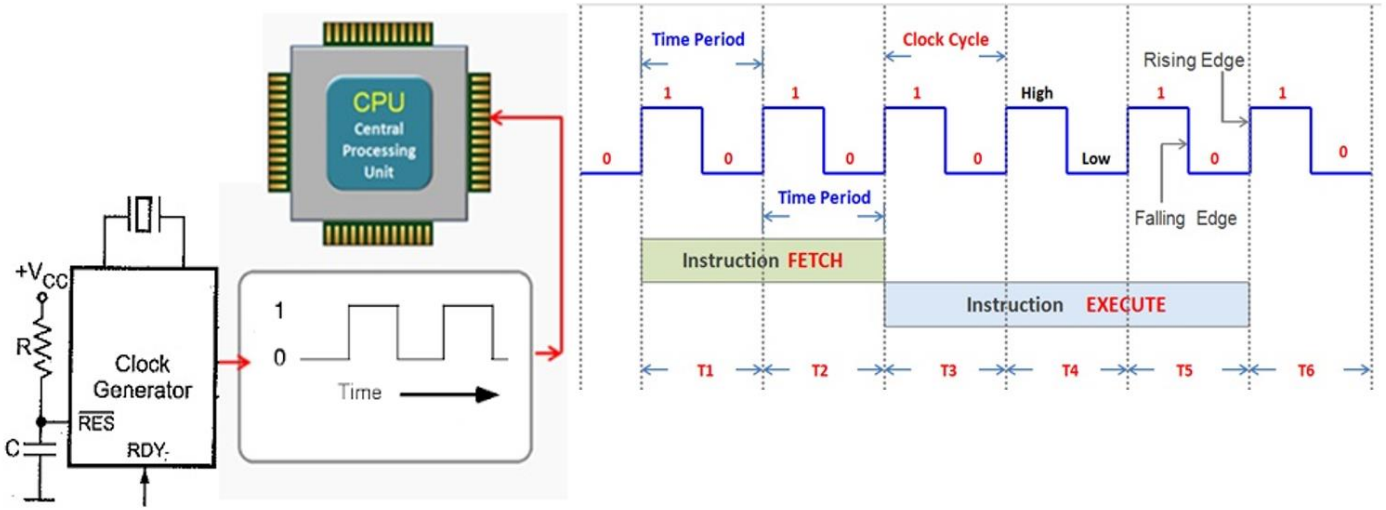


- Computers use a clock to determine when events take place within hardware
- *Clock cycles:* discrete time intervals
  - a.k.a. clocks, cycles, clock periods, clock ticks
- *Clock rate* or *clock frequency:* clock cycles per second (inverse of clock cycle time)
- **Example:** 3 GigaHertz clock rate means
  clock cycle time = $1/(3\times10^{9})$ seconds
  = 333 picoseconds (ps)

CPU Time: Performance Equation

CPU Clocking

Operation of CPU hardware governed by a constant-rate clock



$$\text{CPU Time} = \text{CPU Clock Cycles} \times \text{Clock Cycle Time}$$

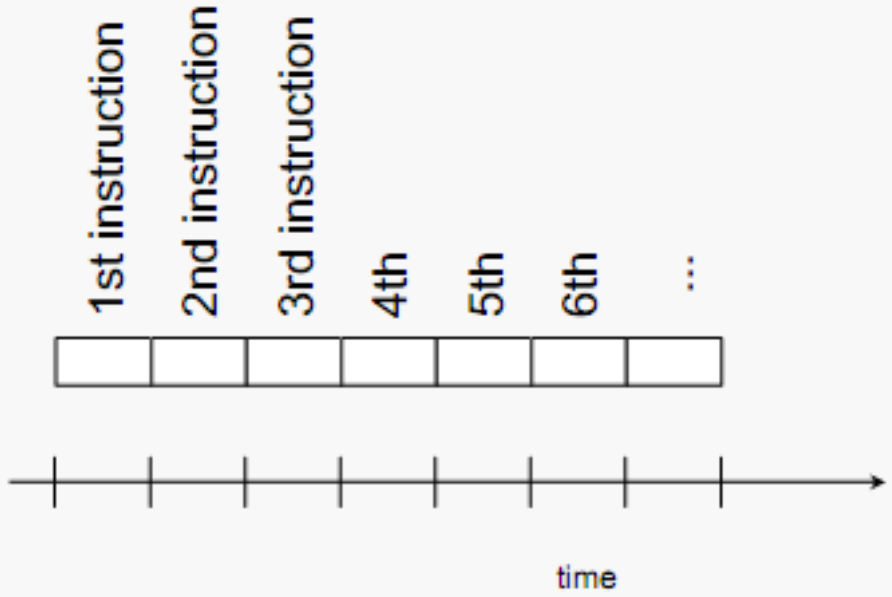
$$= \frac{\text{CPU Clock Cycles}}{\text{Clock Rate}}$$

- Performance improved by
  - Reducing number of clock cycles
  - Increasing clock rate
  - Hardware designer must often trade off clock rate against cycle count

- Clock period: duration of a clock cycle
  - e.g., 1ns = 1×10^{-9} s
- Clock frequency (rate): cycles per second
  - e.g., 1 GHz = 1000MHz = 1.0×10^{9}Hz

of Clock Cycles

Could assume that number of cycles equals number of instructions:



This assumption is incorrect,

- different instructions may take different numbers of cycles on same machine
- same instruction may take different number of cycles on different machines

Why? hint: remember that these are machine instructions, not lines of C code

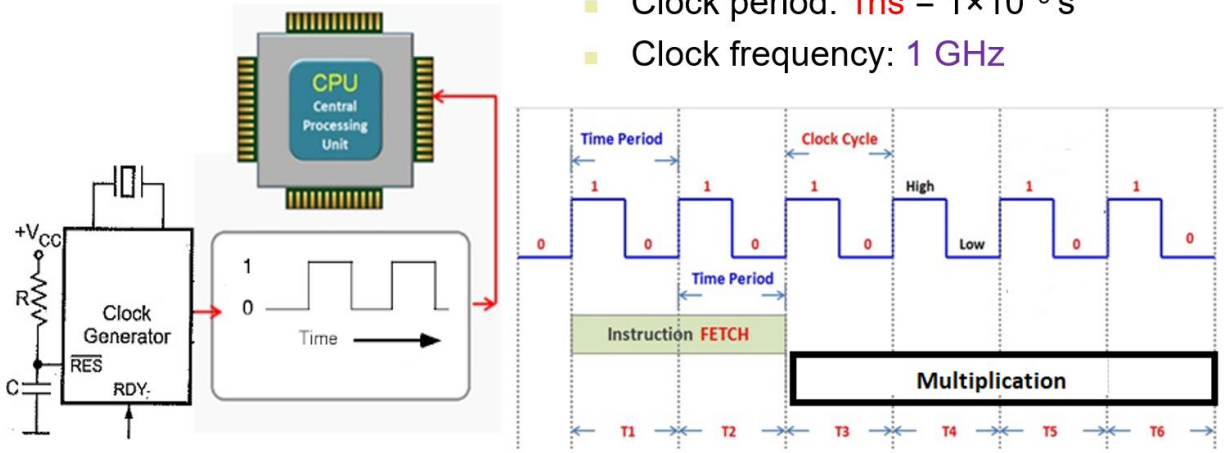
CPU Time: Different Instructions

$$\text{CPU Time} = \text{CPU Clock Cycles} \times \text{Clock Cycle Time}$$

$$= \frac{\text{CPU Clock Cycles}}{\text{Clock Rate}}$$

- Performance improved by
  - Reducing number of clock cycles
  - Increasing clock rate
  - Hardware designer must often trade off clock rate against cycle count

CPU Time for an Instruction

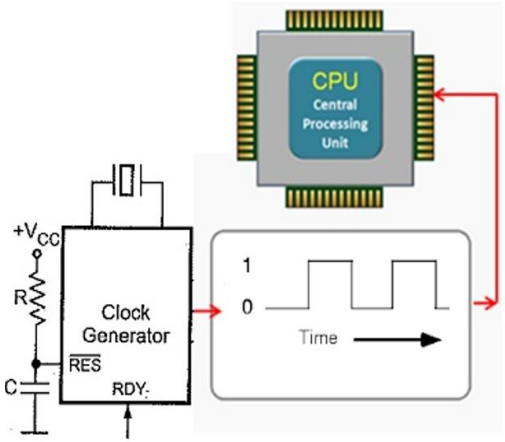


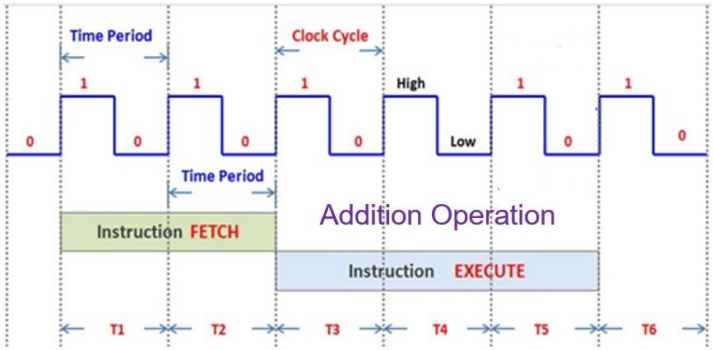
- Clock period: 1ns = 1×10^{-9} s
- Clock frequency: 1 GHz

Example: Multiplication Operation

- Time = 4ns = 4×10^{-9} s
- Time: 4 CPU Clock cycles = 4 CPI for Multiplication
- Time = CPI x Clock period = 4 x 1ns = 4ns

CPU Time for an Instruction





- Clock period: 1ns = 1×10^{-9} s
- Clock frequency: 1 GHz

Example: Addition Operation

- Time = 3ns = 3×10^{-9} s
- Time: 3 CPU Clock cycles = 3 Cycles per Addition = 3 CPI for Addition
- Time = CPI x Clock period = 3 x 1ns = 3ns

CPU Time: Example Problem

- Our favorite program runs in 10 seconds on computer A, which has a 2Ghz. clock.
- We are trying to help a computer designer build a new machine B, that will run this program in 6 seconds.
- The designer can use new (or perhaps more expensive) technology to substantially increase the clock rate, but has informed us that this increase will affect the rest of the CPU design, causing machine B to require 1.2 times as many clock cycles as machine A for the same program.

- *What clock rate should we tell the designer to target?*

- Computer A: 2GHz clock, 10s CPU time
- Designing Computer B
  - Aim for 6s CPU time
  - Can do faster clock, but causes 1.2 × clock cycles compared to A
- How fast must Computer B clock be i.e., what is the clock rate for Computer B?

$$\text{CPU Time}_B = \frac{\text{CPU Clock Cycles}_B}{\text{Clock Rate}_B}$$

CPU Time: Example Problem Solution

- Computer A: 2GHz clock, 10s CPU time
- Designing Computer B
  - Aim for 6s CPU time
  - Can do faster clock, but causes 1.2 × clock cycles compared to A
- How fast must Computer B clock be i.e., what is the clock rate for Computer B?

$$\text{CPU Time}_B = \frac{\text{CPU Clock Cycles}_B}{\text{Clock Rate}_B}$$

Let's first find the number of clock cycles required for the program on A:

$$\text{CPU time}_A = \frac{\text{CPU clock cycles}_A}{\text{Clock rate}_A}$$

$$10 \text{ seconds} = \frac{\text{CPU clock cycles}_A}{2 \times 10^9 \frac{\text{cycles}}{\text{second}}}$$

$$\text{CPU clock cycles}_A = 10 \text{ seconds} \times 2 \times 10^9 \frac{\text{cycles}}{\text{second}} = 20 \times 10^9 \text{ cycles}$$

CPU time for B can be found using this equation:

$$\text{CPU time}_B = \frac{1.2 \times \text{CPU clock cycles}_A}{\text{Clock rate}_B}$$

$$6 \text{ seconds} = \frac{1.2 \times 20 \times 10^9 \text{ cycles}}{\text{Clock rate}_B}$$

$$\text{Clock rate}_B = \frac{1.2 \times 20 \times 10^9 \text{ cycles}}{6 \text{ seconds}} = \frac{0.2 \times 20 \times 10^9 \text{ cycles}}{\text{second}} = \frac{4 \times 10^9 \text{ cycles}}{\text{second}} = 4 \text{ GHz}$$

To run the program in 6 seconds, B must have twice the clock rate of A.

Instruction Performance: CPI

$$\text{Clock Cycles} = \text{Instruction Count} \times \text{Cycles per Instruction}$$

$$\text{CPU Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}$$

$$= \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$$

- Instruction Count for a program
  - Determined by program, ISA and compiler
- Average cycles per instruction
  - Determined by CPU hardware
  - If different instructions have different CPI
    - Average CPI affected by instruction mix

- *Execution time also equals the number of instructions executed multiplied by the average time per instruction*
- *The term **clock cycles per instruction**, which is the average number of clock cycles each instruction takes to execute, is often abbreviated as **CPI**.*
- *Since different instructions may take different amounts of time depending on what they do, CPI is an average of all the instructions executed in the program*
- *CPI provides one way of comparing two different implementations of the same instruction set architecture, since the number of instructions executed for a program will, of course, be the same.*

Examples: CPI

Average CPI

For a given program executed on a given CPU

$$\text{Avg CPI} = \frac{\text{Total CPU Clock cycles for all instruction of program}}{\text{Instructions count in the program}}$$

Example: A program contains following instructions. CPI for different types of instructions are indicated. Calculate the Average CPI of the CPU for the program.

Instruction type	Instruction Count	CPI
ALU	500	2
Load	200	4
Store	200	4
Branch	100	6

Answer: Avg CPI = (500x2 + 200x4 + 200x4 + 100x6)/1000

- Alternative compiled code sequences using instructions in classes A, B, C

Class	A	B	C	
CPI for class	1	2	3	
IC in sequence 1	2	1	2	2+1+2=5 inst.
IC in sequence 2	4	1	1	4+1+1=6 inst.

- Sequence 1: IC = 5
  - Clock Cycles
    = 2×1 + 1×2 + 2×3
    = 10
  - Avg. CPI = 10/5 = 2.0
- Sequence 2: IC = 6
  - Clock Cycles
    = 4×1 + 1×2 + 1×3
    = 9
  - Avg. CPI = 9/6 = 1.5

Instruction Performance: CPI

$$\text{Clock Cycles} = \text{Instruction Count} \times \text{Cycles per Instruction}$$

$$\text{CPU Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}$$

$$= \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$$

- Execution Time = Seconds / Program

$$\frac{\text{Instructions}}{\textbf{program}} \times \frac{\text{cycles}}{\text{Instruction}} \times \frac{\textbf{seconds}}{\text{cycle}}$$

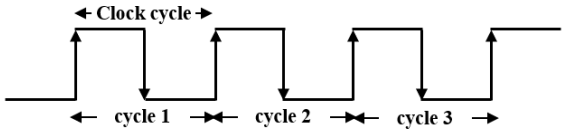
- Programmer
- Algorithms
- ISA
- Compilers

- Microarchitecture
- System architecture

- Microarchitecture, pipeline depth
- Circuit design
- Technology

T = I x CPI x C

Example: Cycles Per Instruction (CPI)



CPU Clock rate, f =1MHz; Clock Cycle, C = 1 micro second

- **If CPU takes 1 micro second to complete an Instruction, then CPI =1 for that Instruction.**
- **If CPU takes 3 micro seconds to complete another Instruction, then CPI =3 for that Instruction.**

If CPU Clock rate, f =2MHz; Clock Cycle, C = 0.5 micro second

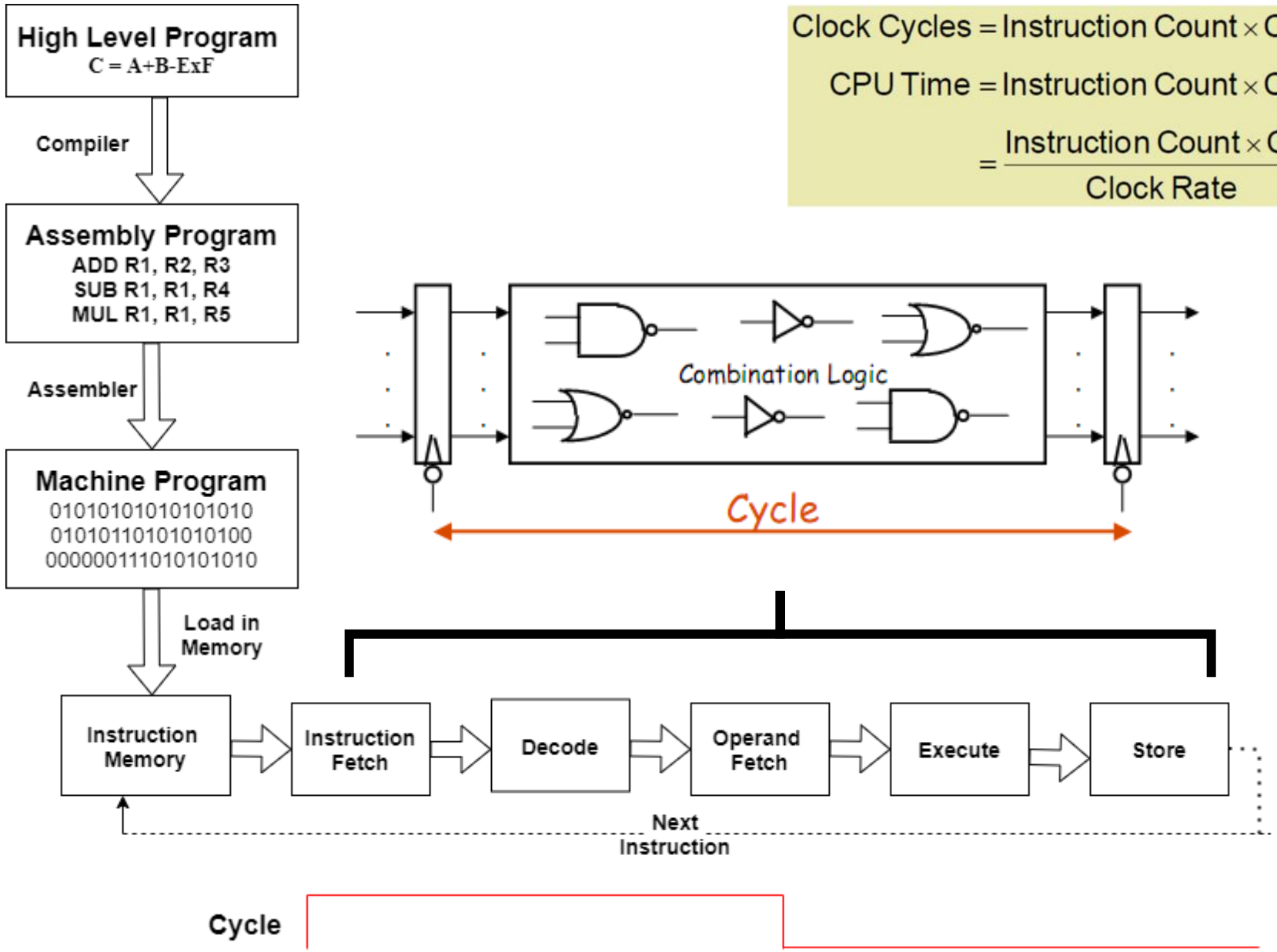
- **If CPU takes 1 micro second to complete an Instruction, then CPI =2 for that Instruction.**
- **If CPU takes 3 micro seconds to complete another Instruction, then CPI =6 for that Instruction.**

Instruction Performance: Behind the Scenes (1)

$$\text{Clock Cycles} = \text{Instruction Count} \times \text{Cycles per Instruction}$$

$$\text{CPU Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}$$

$$= \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$$

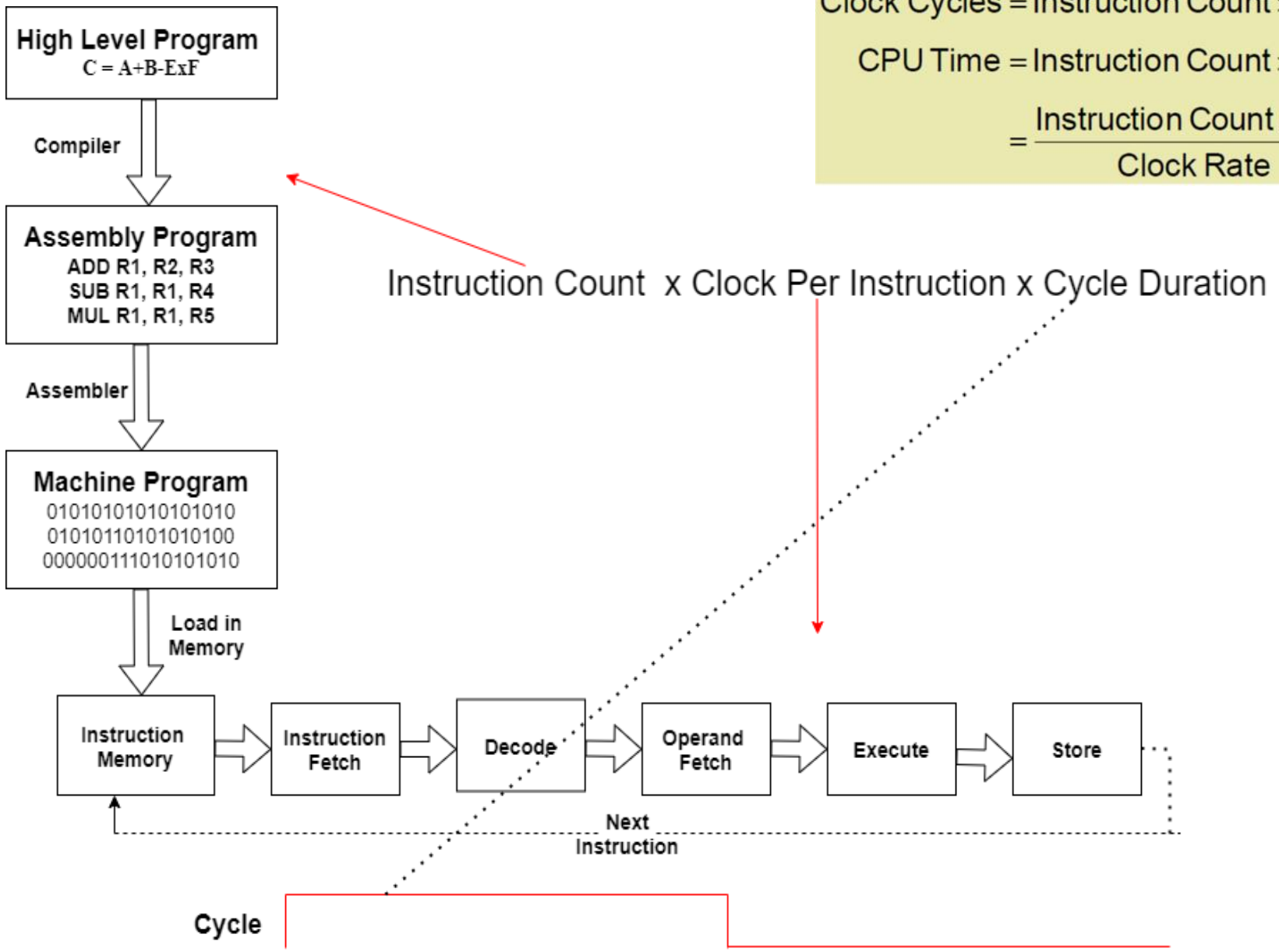


Instruction Performance: Behind the Scenes (2)

$$\text{Clock Cycles} = \text{Instruction Count} \times \text{Cycles per Instruction}$$

$$\text{CPU Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}$$

$$= \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$$

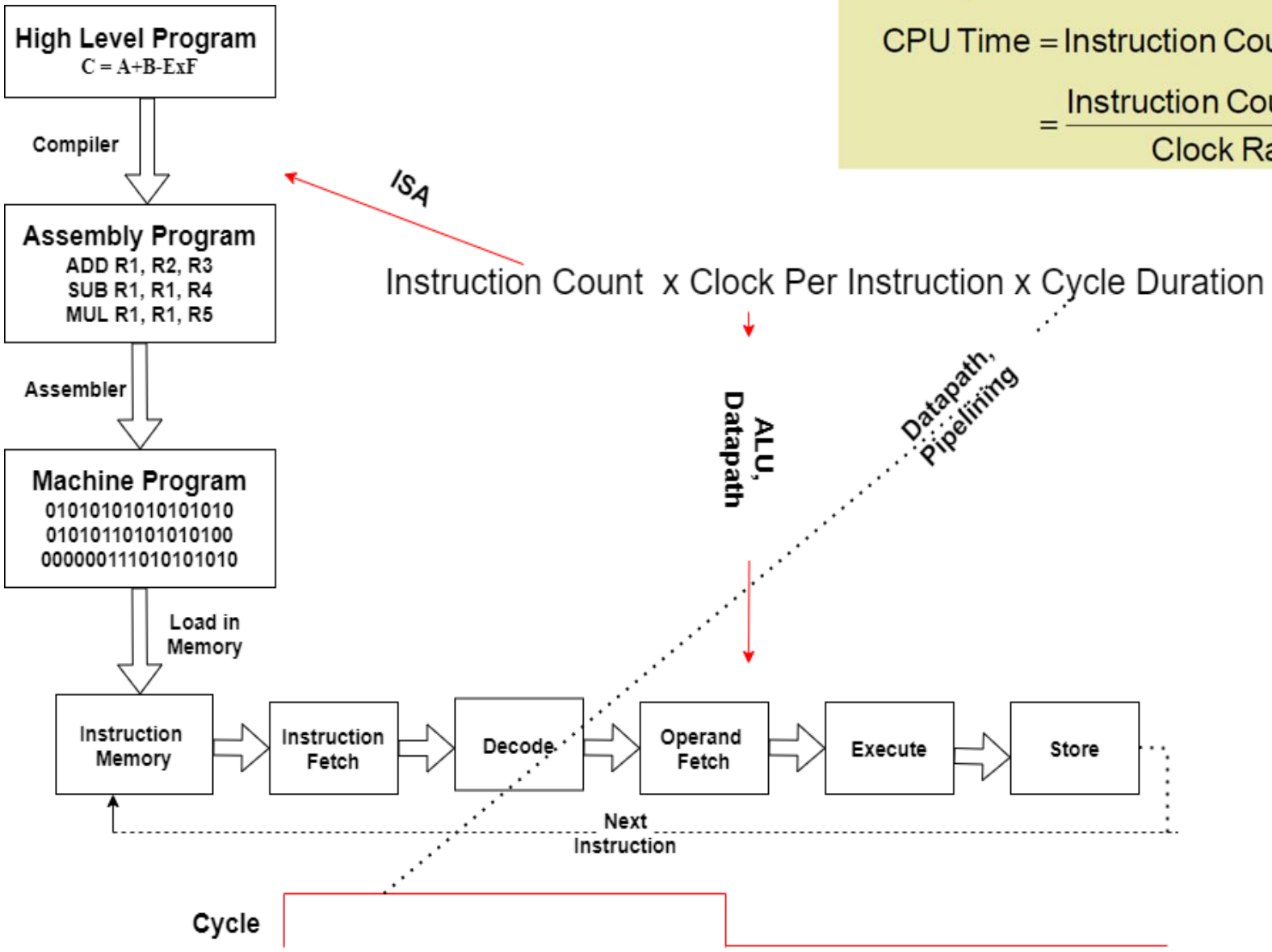


Instruction Performance: Behind the Scenes (3)

$$\text{Clock Cycles} = \text{Instruction Count} \times \text{Cycles per Instruction}$$

$$\text{CPU Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}$$

$$= \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$$



Factors Influencing Performance (1)

Components of performance	Units of measure
CPU execution time for a program	Seconds for the program
Instruction count	Instructions executed for the program
Clock cycles per instruction (CPI)	Average number of clock cycles per instruction
Clock cycle time	Seconds per clock cycle

Performance is determined by execution time

Do any of the other variables equal performance?

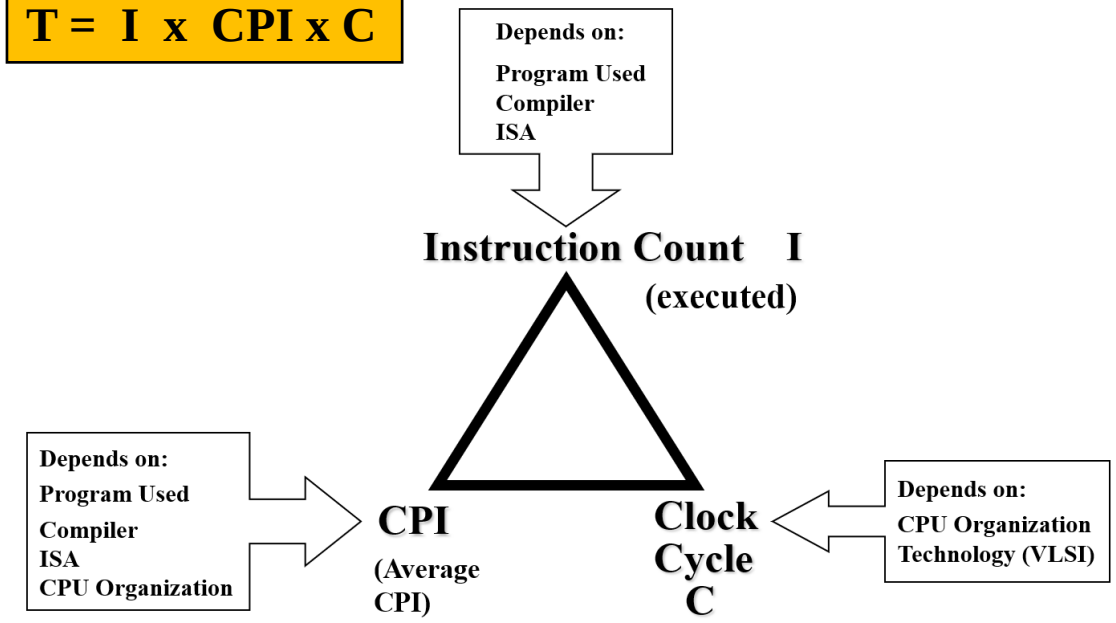
- # of cycles to execute program?
- # of instructions in program?
- # of cycles per second?
- average # of cycles per instruction?
- average # of instructions per second?

Common pitfall: thinking one of these variables (alone) is indicative of performance, when none really is

Aspects of CPU Execution Time

CPU Time = Instruction count executed x CPI x Clock cycle

T = I x CPI x C



$$\text{Clock Cycles} = \text{Instruction Count} \times \text{Cycles per Instruction}$$

$$\text{CPU Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}$$

$$= \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$$

Factors Influencing Performance (2)

$$CPUtime = \frac{Seconds}{Program} = \frac{Cycles}{Program} \cdot \frac{Seconds}{Cycle}$$

$$= \frac{Instructions}{Program} \cdot \frac{Cycles}{Instruction} \cdot \frac{Seconds}{Cycle}$$

$$\text{Clock Cycles} = \text{Instruction Count} \times \text{Cycles per Instruction}$$

$$\text{CPU Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}$$

$$= \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$$

	IC	CPI	Clock Cycle
Program	√		
Compiler	√	(√)	
Instruction Set	√	√	
Organization		√	√
Technology			√

Factors Influencing Program Performance

The performance of a program depends on the algorithm, the language, the compiler, the architecture, and the actual hardware. The following table summarizes how these components affect the factors in the CPU performance equation.

Hardware or software component	Affects what?	How?
Algorithm	Instruction count, possibly CPI	The algorithm determines the number of source program instructions executed and hence the number of processor instructions executed. The algorithm may also affect the CPI, by favoring slower or faster instructions. For example, if the algorithm uses more divides, it will tend to have a higher CPI.
Programming language	Instruction count, CPI	The programming language certainly affects the instruction count, since statements in the language are translated to processor instructions, which determine instruction count. The language may also affect the CPI because of its features; for example, a language with heavy support for data abstraction (e.g., Java) will require indirect calls, which will use higher CPI instructions.
Compiler	Instruction count, CPI	The efficiency of the compiler affects both the instruction count and average cycles per instruction, since the compiler determines the translation of the source language instructions into computer instructions. The compiler's role can be very complex and affect the CPI in complex ways.
Instruction set architecture	Instruction count, clock rate, CPI	The instruction set architecture affects all three aspects of CPU performance, since it affects the instructions needed for a function, the cost in cycles of each instruction, and the overall clock rate of the processor.

Performance Comparison: Example

$T = I \times CPI \times C$

A Program is running on a specific machine (CPU) with the following parameters:

- Total executed instruction count, I: 10,000,000 instructions
- Average CPI for the program: 2.5 cycles/instruction.
- CPU clock rate: 200 MHz.

Thus: $C = 1/(200 \times 10^6) = 5 \times 10^{-9}$ seconds

- Using the same program with these changes:
  - A new compiler used: New executed instruction count, I: 9,500,000
    New CPI: 3.0
  - Faster CPU implementation: New clock rate = 300 MHz
- What is the speedup with the changes?

Thus: $C = 1/(300 \times 10^6) = 3.33 \times 10^{-9}$ seconds

$$\text{Speedup} = \frac{\text{Old Execution Time}}{\text{New Execution Time}} = \frac{I_{old} \times CPI_{old} \times \text{Clock cycle}_{old}}{I_{new} \times CPI_{new} \times \text{Clock Cycle}_{new}}$$

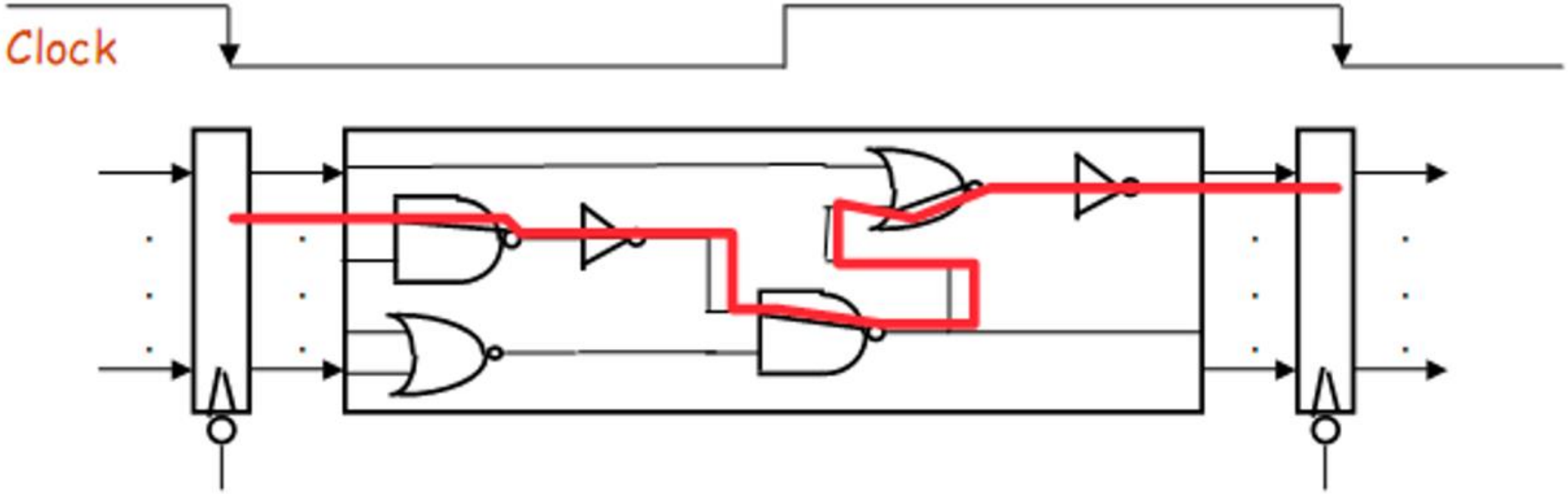
$$\text{Speedup} = (10{,}000{,}000 \times 2.5 \times 5 \times 10^{-9}) / (9{,}500{,}000 \times 3 \times 3.33 \times 10^{-9})$$

$$= .125 / .095 = 1.32$$

or 32 % faster after changes.

Factors Influencing Performance: Example (Hardware)

Critical Path & Cycle Time

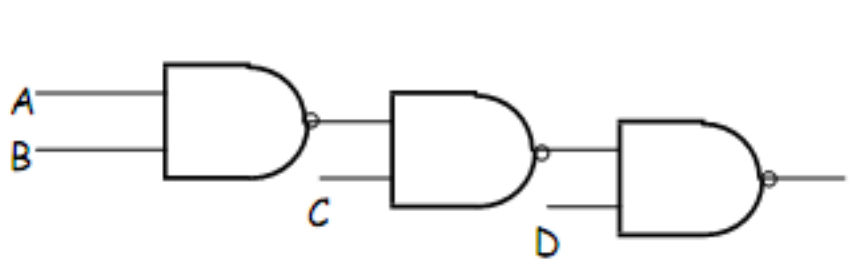


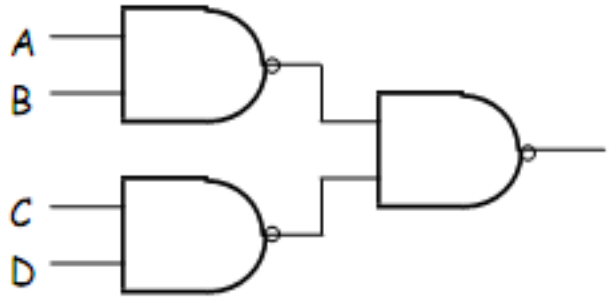
Critical path: the slowest path between any two storage devices

Cycle time is a function of the critical path

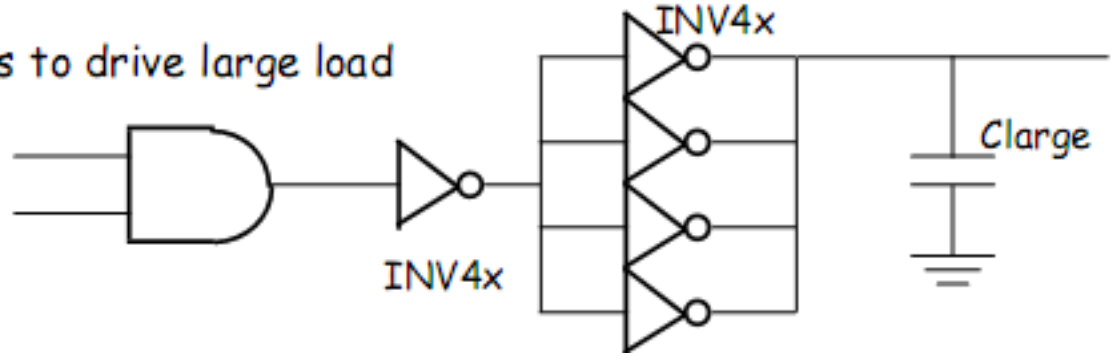
Factors Influencing Performance: Example (Hardware)

Reduce the number of gate levels





- Pay attention to loading
  - One gate driving many gates is a bad idea
  - Avoid using a small gate to drive a long wire
- Use multiple stages to drive large load
- Revise design



More to follow in ALU and Pipeline chapter

Instruction Performance: Example Problem and Solution

EXAMPLE

Suppose we have two implementations of the same instruction set architecture. Computer A has a clock cycle time of 250 ps and a CPI of 2.0 for some program, and computer B has a clock cycle time of 500 ps and a CPI of 1.2 for the same program. Which computer is faster for this program and by how much?

ANSWER

$$\text{Clock Cycles} = \text{Instruction Count} \times \text{Cycles per Instruction}$$

$$\text{CPU Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}$$

$$= \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$$

We know that each computer executes the same number of instructions for the program; let's call this number I. First, find the number of processor clock cycles for each computer:

$$\text{CPU clock cycles}_A = I \times 2.0$$
$$\text{CPU clock cycles}_B = I \times 1.2$$

Now we can compute the CPU time for each computer:

$$\text{CPU time}_A = \text{CPU clock cycles}_A \times \text{Clock cycle time}$$
$$= I \times 2.0 \times 250 \text{ ps} = 500 \times I \text{ ps}$$

Likewise, for B:

$$\text{CPU time}_B = I \times 1.2 \times 500 \text{ ps} = 600 \times I \text{ ps}$$

Clearly, computer A is faster. The amount faster is given by the ratio of the execution times:

$$\frac{\text{CPU performance}_A}{\text{CPU performance}_B} = \frac{\text{Execution time}_B}{\text{Execution time}_A} = \frac{600 \times I \text{ ps}}{500 \times I \text{ ps}} = 1.2$$

We can conclude that computer A is 1.2 times as fast as computer B for this program.

Instruction Performance: Practice Problem

- Suppose we have two implementations of the same instruction set architecture (ISA). For some program:
  - machine A has a clock cycle time of 10 ns. and a CPI of 2.0
  - machine B has a clock cycle time of 20 ns. and a CPI of 1.2

- *Which machine is faster for this program, and by how much?*
- *If two machines have the same ISA, which of our quantities (e.g., clock rate, CPI, execution time, # of instructions, MIPS) will always be identical?*

Computer Performance: MIPS

Million instructions per second (MIPS)
A measurement of program execution speed based on the number of millions of instructions. MIPS is computed as the instruction count divided by the product of the execution time and 10^6.

$$\text{MIPS} = \frac{\text{Instruction count}}{\text{Execution time} \times 10^6}$$

$$\text{Clock Cycles} = \text{Instruction Count} \times \text{Cycles per Instruction}$$

$$\text{CPU Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}$$

$$= \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$$

There are three problems with using MIPS as a measure for comparing computers. First, MIPS specifies the instruction execution rate but does not take into account the capabilities of the instructions. We cannot compare computers with different instruction sets using MIPS, since the instruction counts will certainly differ. Second, MIPS varies between programs on the same computer; thus, a computer cannot have a single MIPS rating. For example, by substituting for execution time, we see the relationship between MIPS, clock rate, and CPI:

$$\text{MIPS} = \frac{\text{Instruction count}}{\frac{\text{Instruction count} \times \text{CPI}}{\text{Clock rate}} \times 10^6} = \frac{\text{Clock rate}}{\text{CPI} \times 10^6}$$

Finally, and most importantly, if a new program executes more instructions but each instruction is faster, MIPS can vary independently from performance! Consider the following performance measurements for a program:

Measurement	Computer A	Computer B
Instruction count	10 billion	8 billion
Clock rate	4 GHz	4 GHz
CPI	1.0	1.1

a. Which computer has the higher MIPS rating?

b. Which computer is faster?

Computer Performance: MIPS

- **Under what conditions can the MIPS rating be used to compare performance of different CPUs?**

- **The MIPS rating is only valid to compare the performance of different CPUs provided that the following conditions are satisfied:**

  1 **The same program is used**

  (actually this applies to all performance metrics)

  2 **The same ISA is used**

  3 **The same compiler is used**

  ⇒ (Thus the resulting programs used to run on the CPUs and obtain the MIPS rating are identical at the machine code level including the same instruction count) **(binary)**

Problem with MIPS: An Example

Compiler Variations, MIPS & Performance: An Example

- For a machine (CPU) with instruction classes:

Instruction class	CPI
A	1
B	2
C	3

- For a given high-level language program, two compilers produced the following executed instruction counts:

	Instruction counts (in millions) for each instruction class		
Code from:	A	B	C
Compiler 1	5	1	1
Compiler 2	10	1	1

- The machine is assumed to run at a clock rate of 100 MHz.

$$\text{Clock Cycles} = \text{Instruction Count} \times \text{Cycles per Instruction}$$

$$\text{CPU Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}$$

$$= \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$$

Compiler Variations, MIPS & Performance: An Example (Continued)

MIPS = Clock rate / (CPI x 10^6) = 100 MHz / (CPI x 10^6)

CPI = CPU execution cycles / Instructions count

$$CPU\ clock\ cycles = \sum_{i=1}^{n} (CPI_i \times C_i)$$

CPU time = Instruction count x CPI / Clock rate

- For compiler 1:
  - CPI_1 = (5 x 1 + 1 x 2 + 1 x 3) / (5 + 1 + 1) = 10 / 7 = 1.43
  - MIPS $Rating_1$ = 100 / (1.428 x 10^6) = 70.0 MIPS
  - CPU $time_1$ = ((5 + 1 + 1) x 10^6 x 1.43) / (100 x 10^6) = 0.10 seconds
- For compiler 2:
  - CPI_2 = (10 x 1 + 1 x 2 + 1 x 3) / (10 + 1 + 1) = 15 / 12 = 1.25
  - MIPS $Rating_2$ = 100 / (1.25 x 10^6) = 80.0 MIPS
  - CPU $time_2$ = ((10 + 1 + 1) x 10^6 x 1.25) / (100 x 10^6) = 0.15 seconds

MIPS rating indicates that compiler 2 is better while in reality the code produced by compiler 1 is faster

Instructions with Varying Numbers of Cycles

- **Given a program with *n* types or classes of instructions executed on a given CPU with the following characteristics:**

C_i = Count of instructions of type$_i$ executed

CPI_i = Cycles per instruction for type$_i$ **i = 1, 2, …. n**

Depends on CPU Design

Then:

CPI = CPU Clock Cycles / Instruction Count I

i.e average or effective CPI

Executed

Where:

$$CPU\ clock\ cycles = \sum_{i=1}^{n} (CPI_i \times C_i)$$

$$\text{Executed Instruction Count } I = \Sigma C_i$$

If different instruction classes take different numbers of cycles

$$\text{Clock Cycles} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{Instruction Count}_i)$$

Weighted average CPI

$$\text{CPI} = \frac{\text{Clock Cycles}}{\text{Instruction Count}} = \sum_{i=1}^{n} \left(\text{CPI}_i \times \underbrace{\frac{\text{Instruction Count}_i}{\text{Instruction Count}}}_{\text{Relative frequency}} \right)$$

Instructions with Varying Numbers of Cycles: Example Problem

A compiler designer is trying to decide between two code sequences for a particular computer. The hardware designers have supplied the following facts:

	CPI for each instruction class		
	A	B	C
CPI	1	2	3

For a particular high-level language statement, the compiler writer is considering two code sequences that require the following instruction counts:

Code sequence	Instruction counts for each instruction class		
	A	B	C
1	2	1	2
2	4	1	1

Which code sequence executes the most instructions? Which will be faster? What is the CPI for each sequence?

Instructions with Varying Numbers of Cycles: Example Solution

Sequence 1 executes 2 + 1 + 2 = 5 instructions. Sequence 2 executes 4 + 1 + 1 = 6 instructions. Therefore, sequence 1 executes fewer instructions.

We can use the equation for CPU clock cycles based on instruction count and CPI to find the total number of clock cycles for each sequence:

$$\text{CPU clock cycles} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{C}_i)$$

This yields

$$\text{CPU clock cycles}_1 = (2 \times 1) + (1 \times 2) + (2 \times 3) = 2 + 2 + 6 = 10 \text{ cycles}$$

$$\text{CPU clock cycles}_2 = (4 \times 1) + (1 \times 2) + (1 \times 3) = 4 + 2 + 3 = 9 \text{ cycles}$$

So code sequence 2 is faster, even though it executes one extra instruction. Since code sequence 2 takes fewer overall clock cycles but has more instructions, it must have a lower CPI. The CPI values can be computed by

$$\text{CPI} = \frac{\text{CPU clock cycles}}{\text{Instruction count}}$$

$$\text{CPI}_1 = \frac{\text{CPU clock cycles}_1}{\text{Instruction count}_1} = \frac{10}{5} = 2.0$$

$$\text{CPI}_2 = \frac{\text{CPU clock cycles}_2}{\text{Instruction count}_2} = \frac{9}{6} = 1.5$$

ANSWER

	CPI for each instruction class		
	A	B	C
CPI	1	2	3

For a particular high-level language statement, the compiler writer is considering two code sequences that require the following instruction counts:

Code sequence	Instruction counts for each instruction class		
	A	B	C
1	2	1	2
2	4	1	1

T = I x CPI x C

$$\text{Clock Cycles} = \text{Instruction Count} \times \text{Cycles per Instruction}$$

$$\text{CPU Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}$$

$$= \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$$

Instruction Frequency & CPI

- Given a program with n types or classes of instructions with the following characteristics:

C_i = Count of instructions of type$_i$ executed

CPI_i = Average cycles per instruction of type$_i$

F_i = Frequency or fraction of instruction type$_i$ executed

= C_i/ total executed instruction count = C_i/ I

i = 1, 2, …. n

Where: Executed Instruction Count I = ΣC_i

Then:

$$CPI = \sum_{i=1}^{n} (CPI_i \times F_i)$$

i.e average or effective CPI

Fraction of total execution time for instructions of type i = $\frac{CPI_i \times F_i}{CPI}$

T = I x CPI x C

Instruction Type Frequency & CPI: A RISC Example

Program Profile or Executed Instructions Mix

Base Machine (Reg / Reg)

Depends on CPU Design

$\frac{CPI_i \times F_i}{CPI}$

Op	Freq, F_i	CPI_i	$CPI_i \times F_i$	% Time
ALU	50%	1	.5	23% = .5/2.2
Load	20%	5	1.0	45% = 1/2.2
Store	10%	3	.3	14% = .3/2.2
Branch	20%	2	.4	18% = .4/2.2
			Sum = 2.2	

Given

Typical Mix

$$CPI = \sum_{i=1}^{n} (CPI_i \times F_i)$$

i.e average or effective CPI

CPI = .5 x 1 + .2 x 5 + .1 x 3 + .2 x 2 = 2.2

= .5 + 1 + .3 + .4

T = I x CPI x C

Instructions with Varying Numbers of Cycles: Practice Problem

A given application written in Java runs 15 seconds on a desktop processor. A new Java compiler is released that requires only 0.6 as many instructions as the old compiler. Unfortunately, it increases the CPI by 1.1. How fast can we expect the application to run using this new compiler? Pick the right answer from the three choices below:

a. $\frac{15 \times 0.6}{1.1} = 8.2 \text{ sec}$

b. $15 \times 0.6 \times 1.1 = 9.9 \text{ sec}$

c. $\frac{15 \times 1.1}{0.6} = 27.5 \text{ sec}$

$$\text{Clock Cycles} = \text{Instruction Count} \times \text{Cycles per Instruction}$$

$$\text{CPU Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}$$

$$= \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$$

Instructions with Varying Numbers of Cycles: Practice Problem

- Two different compilers are being tested for a 500 MHz. machine with three different classes of instructions: Class A, Class B, and Class C, which require 1, 2 and 3 cycles (respectively). Both compilers are used to produce code for a large piece of software.
- Compiler 1 generates code with 5 billion Class A instructions, 1 billion Class B instructions, and 1 billion Class C instructions.
- Compiler 2 generates code with 10 billion Class A instructions, 1 billion Class B instructions, and 1 billion Class C instructions.

- *Which sequence will be faster according to MIPS?*
- *Which sequence will be faster according to execution time?*

Quantitative Principles of Computer Design

- **Amdahl's Law:**

The performance gain from improving some portion of a computer is calculated by: i.e using some enhancement

$$\text{Speedup} = \frac{\text{Performance for entire task using the enhancement}}{\text{Performance for the entire task without using the enhancement}}$$

$$\text{or Speedup} = \frac{\text{Execution time without the enhancement}}{\text{Execution time for entire task using the enhancement}}$$

Here: Task = Program

Recall: Performance = 1 /Execution Time

Performance Enhancement Calculations: Amdahl's Law

- The performance enhancement possible due to a given design improvement is limited by the amount that the improved feature is used
- Amdahl's Law:

Performance improvement or speedup due to enhancement E:

$$\text{Speedup(E)} = \frac{\text{Execution Time without E}}{\text{Execution Time with E}} = \frac{\text{Performance with E}}{\text{Performance without E}}$$

original

– Suppose that enhancement E accelerates a fraction F of the execution time by a factor S and the remainder of the time is unaffected then:

Execution Time with E = ((1-F) + F/S) X Execution Time without E

Hence speedup is given by:

$$\text{Speedup(E)} = \frac{\text{Execution Time without E}}{((1 - F) + F/S) \times \text{Execution Time without E}} = \frac{1}{(1 - F) + F/S}$$

F (Fraction of execution time enhanced) refers to original execution time before the enhancement is applied

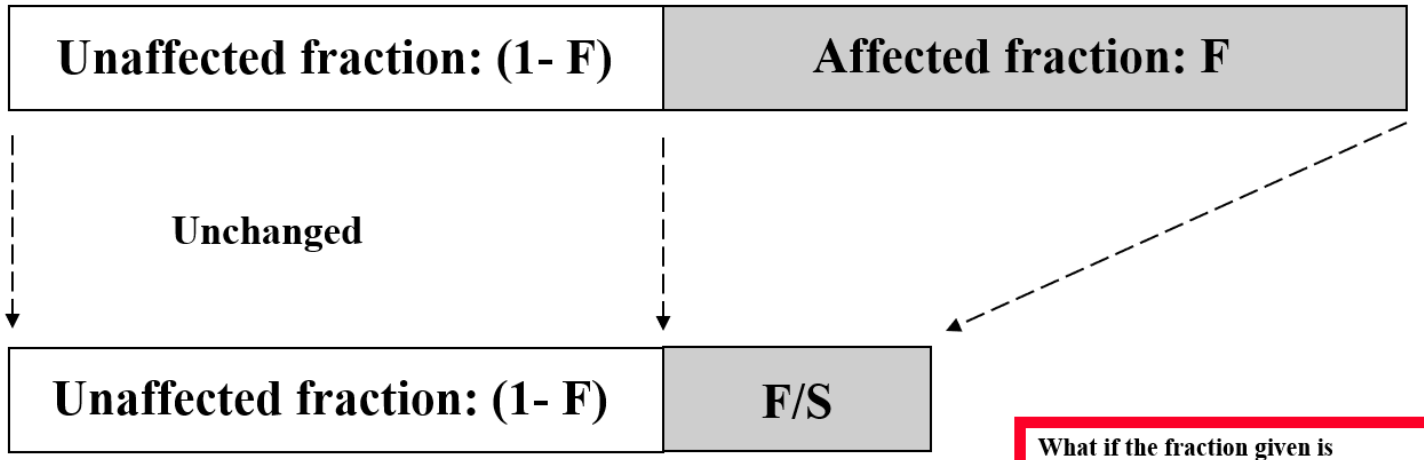
Pictorial Depiction of Amdahl's Law

Enhancement E accelerates fraction F of original execution time by a factor of S

Before:

Execution Time without enhancement E: (Before enhancement is applied)

- **shown normalized to 1 = (1-F) + F =1**



What if the fraction given is after the enhancement has been applied? How would you solve the problem? (i.e find expression for speedup)

After:

Execution Time with enhancement E:

$$\text{Speedup(E)} = \frac{\text{Execution Time without enhancement E}}{\text{Execution Time with enhancement E}} = \frac{1}{(1 - F) + F/S}$$

Performance Enhancement Example

- **For the RISC machine with the following instruction mix given earlier:**

Op	Freq	Cycles	CPI(i)	% Time
ALU	50%	1	.5	23%
Load	20%	5	1.0	45%
Store	10%	3	.3	14%
Branch	20%	2	.4	18%

CPI = 2.2

- **If a CPU design enhancement improves the CPI of load instructions from 5 to 2, what is the resulting performance improvement from this enhancement:**

Fraction enhanced = F = 45% or .45

Unaffected fraction = 1- F = 100% - 45% = 55% or .55

Factor of enhancement = S = 5/2 = 2.5

Using Amdahl's Law:

$$\text{Speedup(E)} = \frac{1}{(1 - F) + F/S} = \frac{1}{.55 + .45/2.5} = 1.37$$

Amdahl`s Law: Alternate Solution using CPU Equation

An Alternative Solution Using CPU Equation

Op	Freq	Cycles	CPI(i)	% Time
ALU	50%	1	.5	23%
Load	20%	5	1.0	45%
Store	10%	3	.3	14%
Branch	20%	2	.4	18%

CPI = 2.2

- **If a CPU design enhancement improves the CPI of load instructions from 5 to 2, what is the resulting performance improvement from this enhancement:**

Old CPI = 2.2

New CPI of load is now 2 instead of 5

New CPI = .5 x 1 + .2 x 2 + .1 x 3 + .2 x 2 = 1.6

$$\text{Speedup(E)} = \frac{\text{Original Execution Time}}{\text{New Execution Time}} = \frac{\text{Instruction count} \times \text{old CPI} \times \text{clock cycle}}{\text{Instruction count} \times \text{new CPI} \times \text{clock cycle}}$$

$$= \frac{\text{old CPI}}{\text{new CPI}} = \frac{2.2}{1.6} = 1.37$$

Which is the same speedup obtained from Amdahl's Law in the first solution.

$$T = I \times CPI \times C$$

Performance Enhancement using Amdahl`s Law: Example-2

Performance Enhancement Example

- A program runs in 100 seconds on a machine with multiply operations responsible for 80 seconds of this time. By how much must the speed of multiplication be improved to make the program four times faster?

$$\text{Desired speedup} = 4 = \frac{100}{\text{Execution Time with enhancement}}$$

→ Execution time with enhancement = 100/4 = 25 seconds

25 seconds = (100 - 80 seconds) + 80 seconds / S

25 seconds = 20 seconds + 80 seconds / S

→ 5 = 80 seconds / S

→ S = 80/5 = 16

Alternatively, it can also be solved by finding enhanced fraction of execution time:

F = 80/100 = .8

and then solving Amdahl's speedup equation for desired enhancement factor S

$$\text{Speedup(E)} = \frac{1}{(1 - F) + F/S} = 4 = \frac{1}{(1 - .8) + .8/S} = \frac{1}{.2 + .8/s}$$

Solving for S gives S= 16

Hence multiplication should be 16 times faster to get an overall speedup of 4.

Machine = CPU

Remember: Slide #7 ?

- Suppose a program runs in 100 seconds on a machine, with multiply responsible for 80 seconds of this time.
- How much do we have to improve the speed of multiplication if we want the program to run 4 times faster?

Performance Enhancement Example

- For the previous example with a program running in 100 seconds on a machine with multiply operations responsible for 80 seconds of this time. By how much must the speed of multiplication be improved to make the program five times faster?

$$\text{Desired speedup} = 5 = \frac{100}{\text{Execution Time with enhancement}}$$

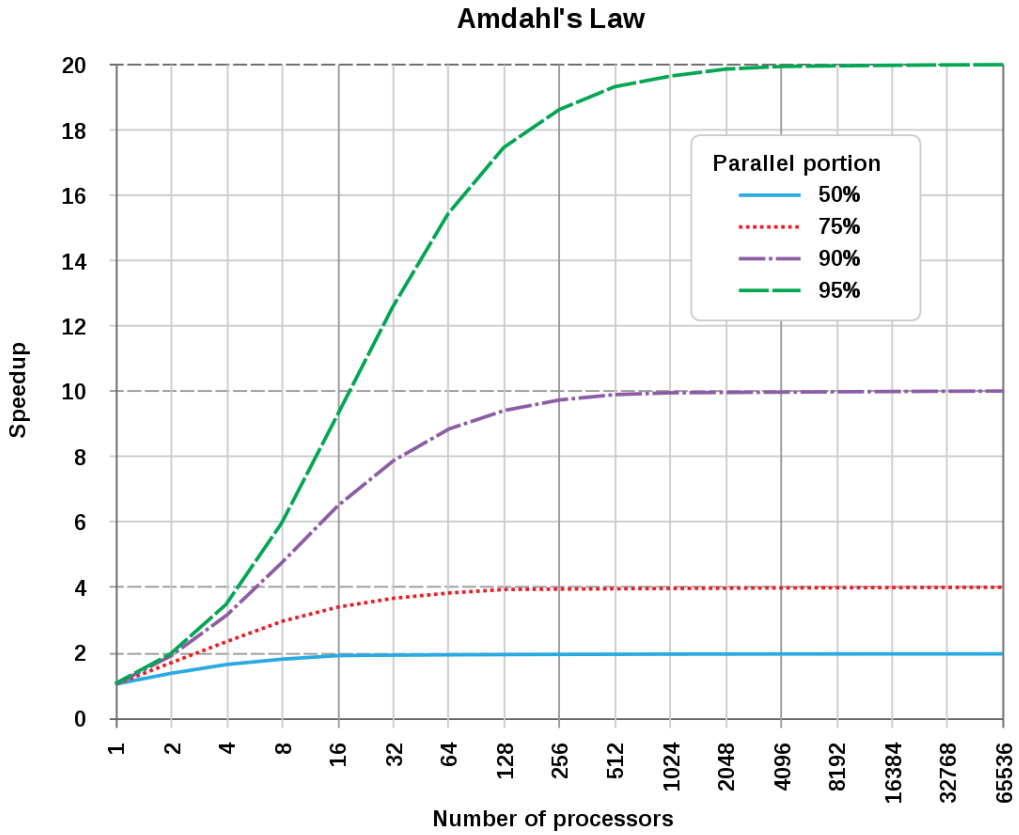
→ Execution time with enhancement = 100/5 = 20 seconds

20 seconds = (100 - 80 seconds) + 80 seconds / s

20 seconds = 20 seconds + 80 seconds / s

→ 0 = 80 seconds / s

No amount of multiplication speed improvement can achieve this.



Amdahl`s Law: Multiple Enhancements

Extending Amdahl's Law To Multiple Enhancements

n enhancements each affecting a different portion of execution time

- Suppose that enhancement E_i accelerates a fraction F_i of the original execution time by a factor S_i and the remainder of the time is unaffected then:

$i = 1, 2, \ldots. n$

$$Speedup = \frac{\text{Original Execution Time}}{\left((1 - \sum_i F_i) + \sum_i \frac{F_i}{S_i}\right) X \text{Original Execution Time}}$$

Unaffected fraction

$$Speedup = \frac{1}{\left((1 - \sum_i F_i) + \sum_i \frac{F_i}{S_i}\right)}$$

What if the fractions given are after the enhancements were applied? How would you solve the problem? (i.e find expression for speedup)

Note: All fractions F_i refer to original execution time before the enhancements are applied.

Amdahl's Law With Multiple Enhancements: Example

- Three CPU performance enhancements are proposed with the following speedups and percentage of the code execution time affected:

$\text{Speedup}_1 = S_1 = 10$	$\text{Percentage}_1 = F_1 = 20\%$
$\text{Speedup}_2 = S_2 = 15$	$\text{Percentage}_1 = F_2 = 15\%$
$\text{Speedup}_3 = S_3 = 30$	$\text{Percentage}_1 = F_3 = 10\%$

- While all three enhancements are in place in the new design, each enhancement affects a different portion of the code and only one enhancement can be used at a time.
- What is the resulting overall speedup?

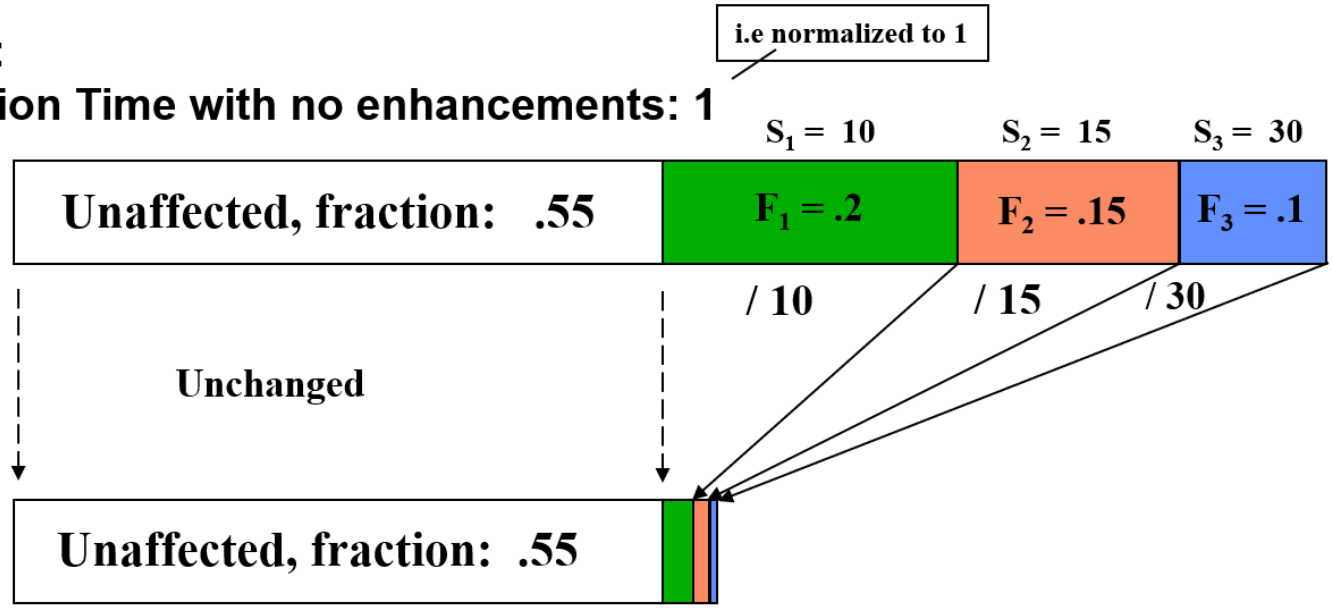
$$Speedup = \frac{1}{\left((1 - \sum_i F_i) + \sum_i \frac{F_i}{S_i}\right)}$$

- Speedup = 1 / [(1 - .2 - .15 - .1) + .2/10 + .15/15 + .1/30)]
  = 1 / [.55 + .0333]
  = 1 / .5833 = 1.71

Amdahl`s Law: Multiple Enhancements Example-1 (Pictorial Depiction)

Pictorial Depiction of Example

Before:
Execution Time with no enhancements: 1



After:
Execution Time with enhancements: .55 + .02 + .01 + .00333 = .5833

Speedup = 1 / .5833 = 1.71

What if the fractions given are after the enhancements were applied? How would you solve the problem?

Note: All fractions F_i refer to original execution time.

Factors Influencing CPU Performance

- Algorithm
  - Determines number of operations executed
- Programming language, compiler, architecture
  - Determine number of machine instructions executed per operation
- Processor and memory system
  - Determine how fast instructions are executed
- I/O system (including OS)
  - Determines how fast I/O operations are executed

CPU Speed vs Computer Performance

CPU Speed Vs Computer Performance

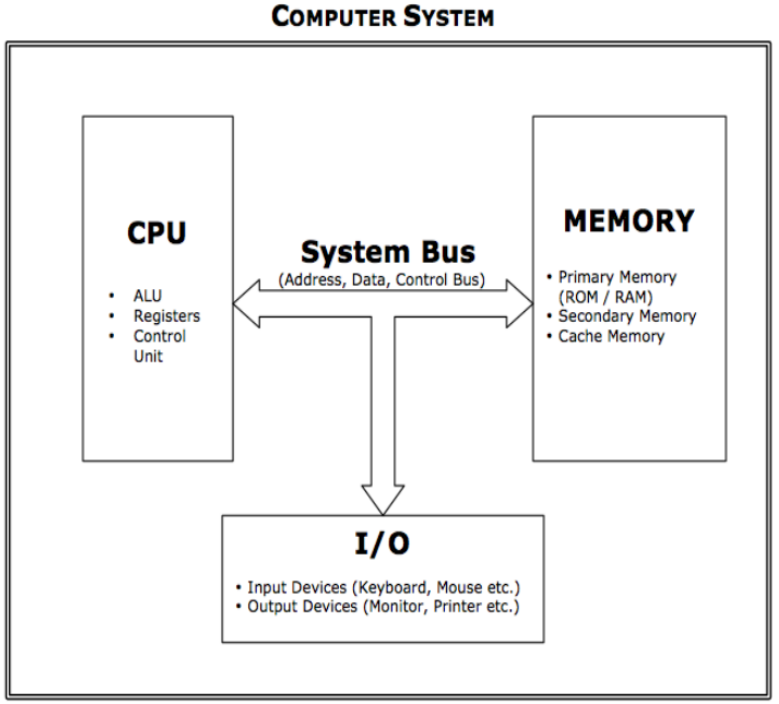
Processing of a single instruction

Instruction Fetch — Requires to access **Memory** through **System Bus**

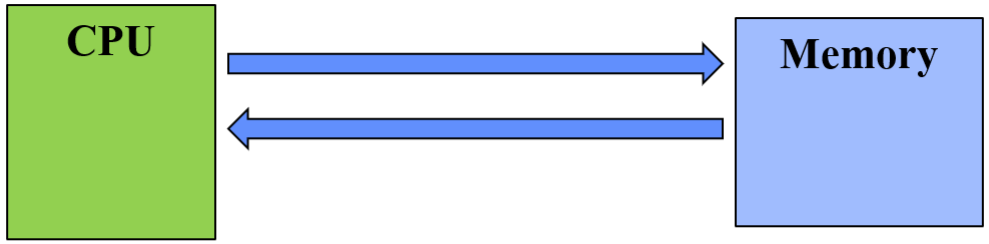
Instruction Execution — Happens within the CPU On-Chip Operation

Instruction fetch: Read machine code of Instruction from Memory and store in a register within CPU/Microprocessor

Instruction Execution: Decode instructions and perform ALU/data transfer operations etc.



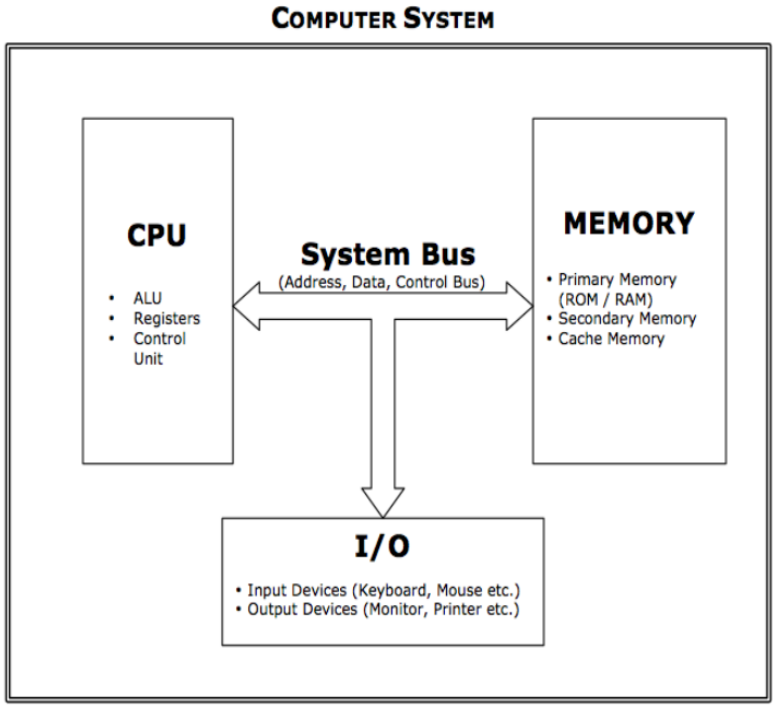
CPU Speed vs Computer Performance



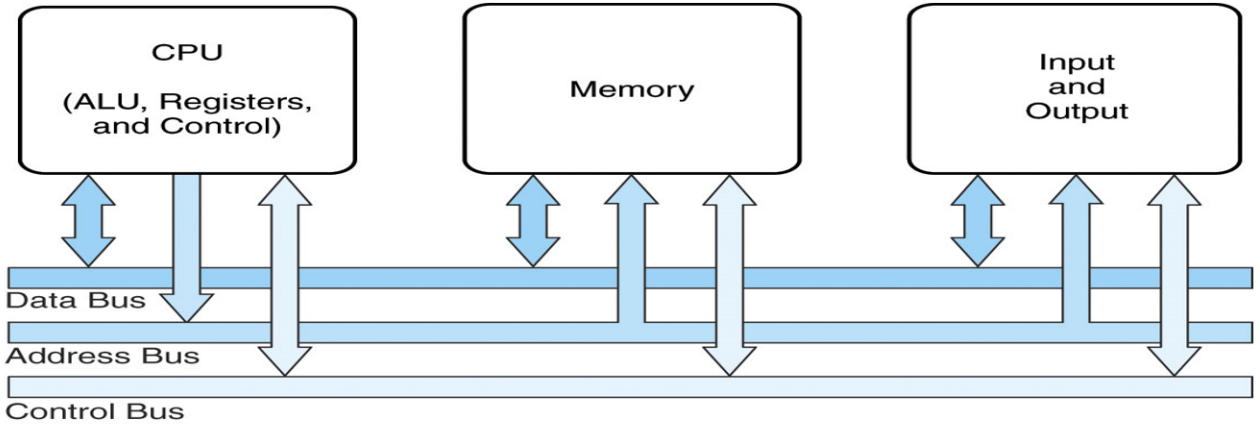
Both BUS speed and speed of memory device are much slower than the speed of CPU/Microprocessor.

During Instruction Fetch, the CPU/Microprocessor has to WAIT for the Machine code to be transferred from memory to CPU through system bus.

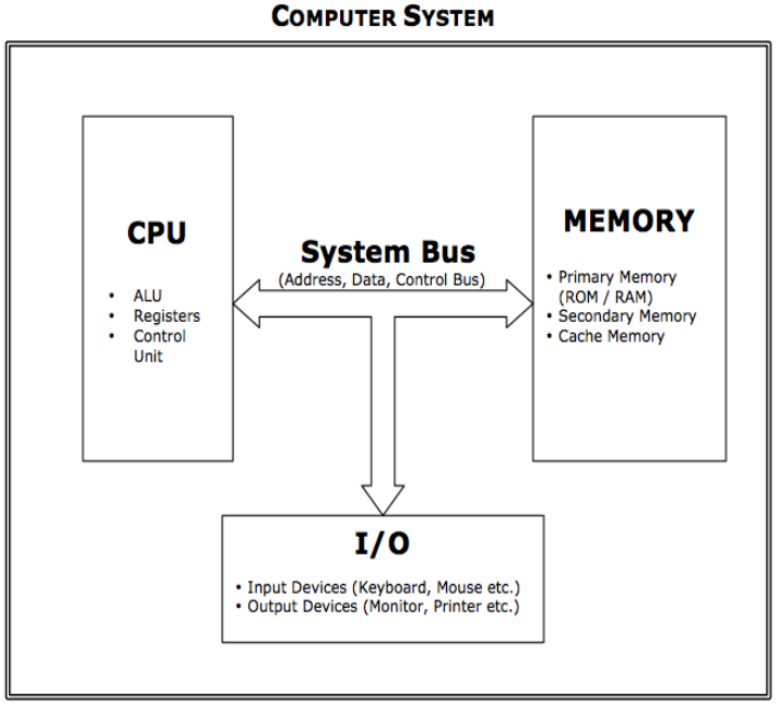
Instruction Execution: mostly ON-CHIP operation, requires much less time than Instruction fetch.



CPU Speed vs Computer Performance

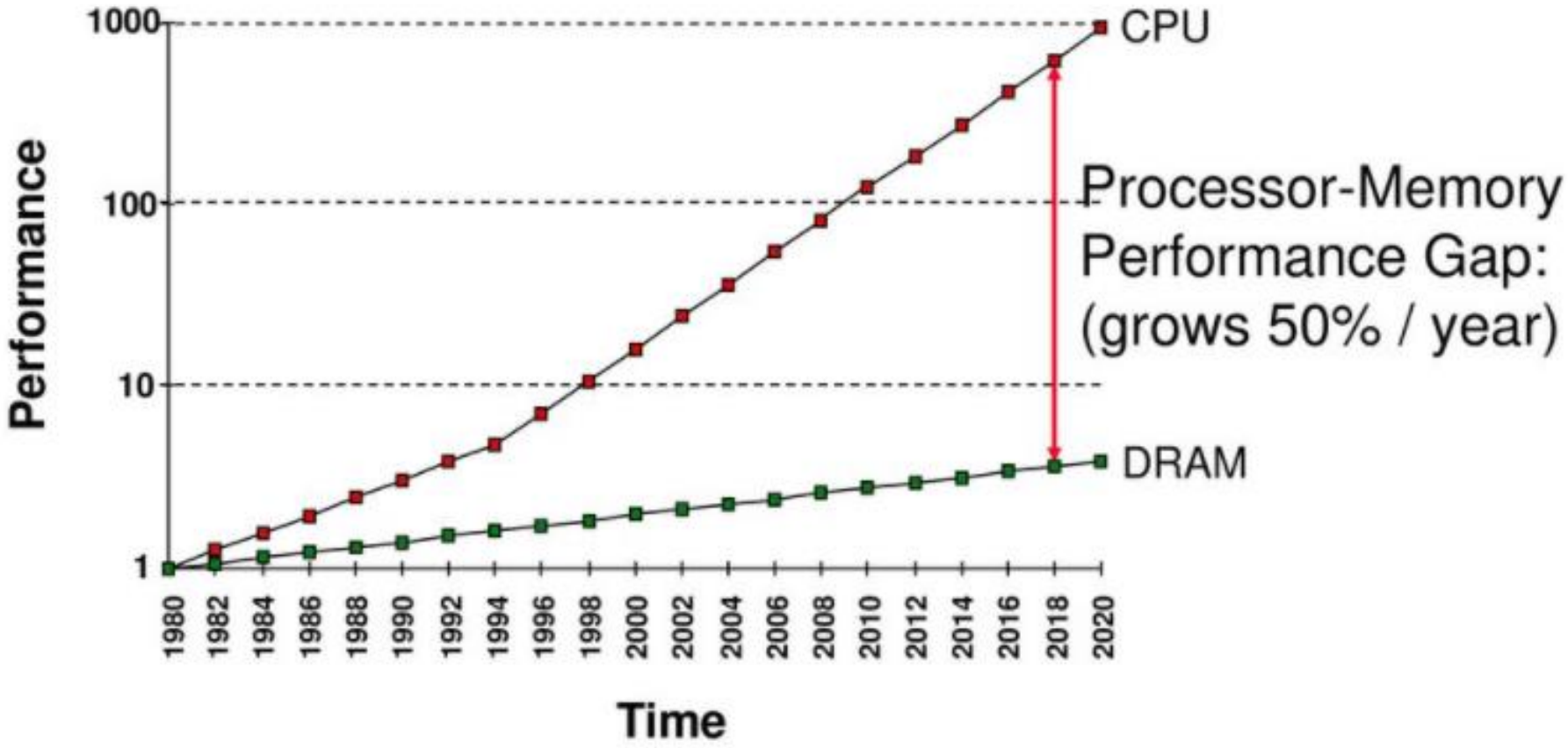


- The interface between processor and main memory is the most crucial pathway in the entire computer because it is responsible for carrying a constant flow of program instructions and data between memory chips and the processor.
- If memory or the pathway fails to keep pace with the processor's insistent demands, the processor stalls in a wait state, and valuable processing time is lost.



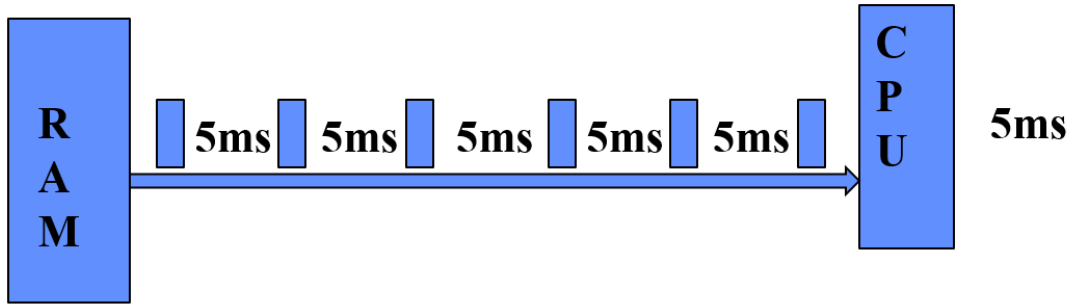
Challenges: Processor vs Memory Performance Gap

- **Processor speed increased**
- **Memory capacity increased**
- **Memory speed lags behind processor speed**



Challenges: Process vs Memory Performance Gap

Processor Speed or Utilization?



The raw speed of the microprocessor will not achieve its potential unless it is fed a constant stream of work to do in the form of computer instructions.

Solution: Either Memory and BUS speed should be same as Processor speed…consequence…EXPENSIVE!

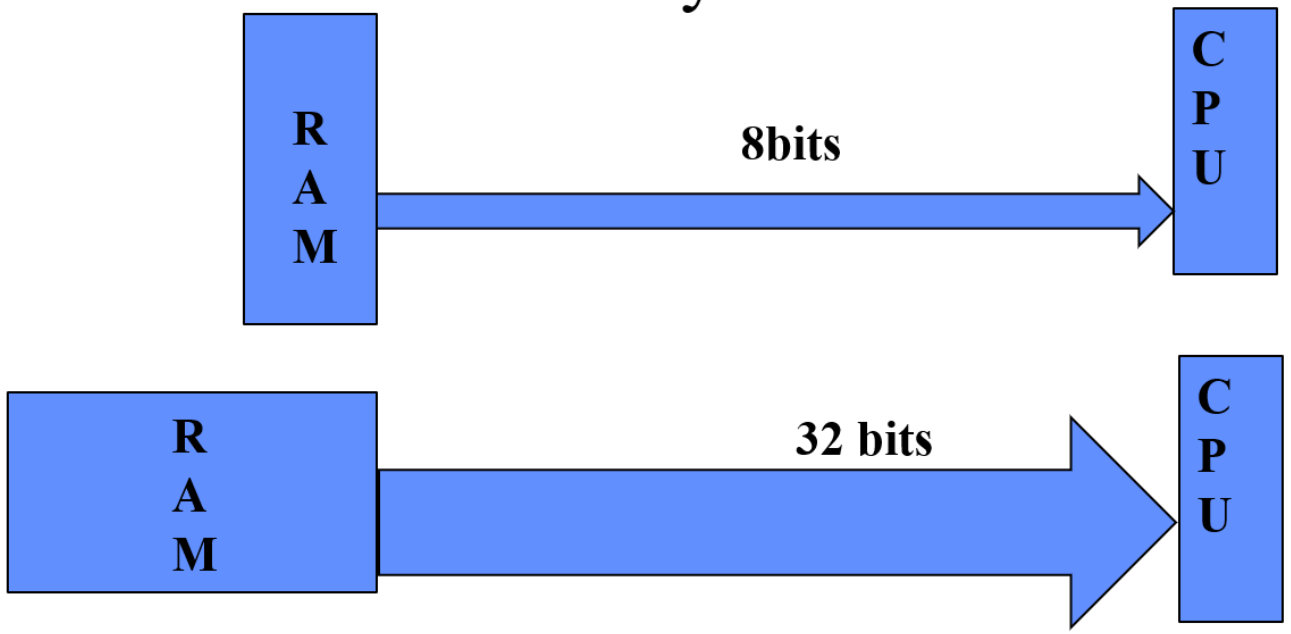
Techniques for maintaining a continuous stream of Instructions to the Microprocessor/CPU.

Processor vs Memory Performance Gap: Solution

Solution: Improved architecture

- Making RAM wider
- Making data bus wider

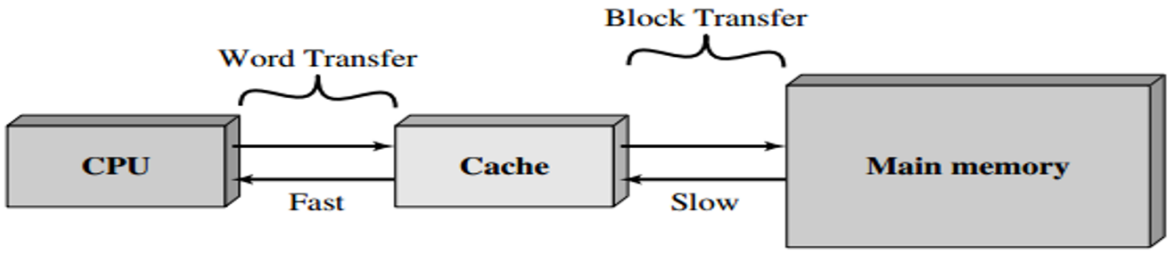
To allow processor to read more data and instruction in one bus cycle



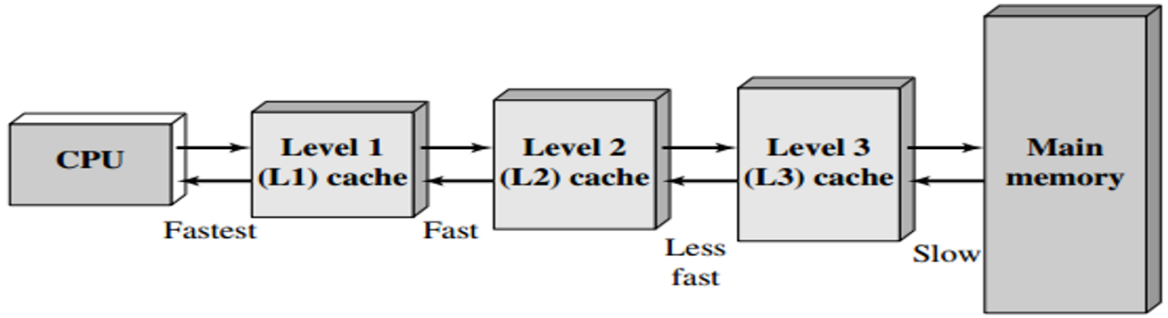
Process vs Memory Performance Gap: Solution

Solution: Cache memory

Reduce the frequency of memory access by incorporating cache memory.



(a) Single cache

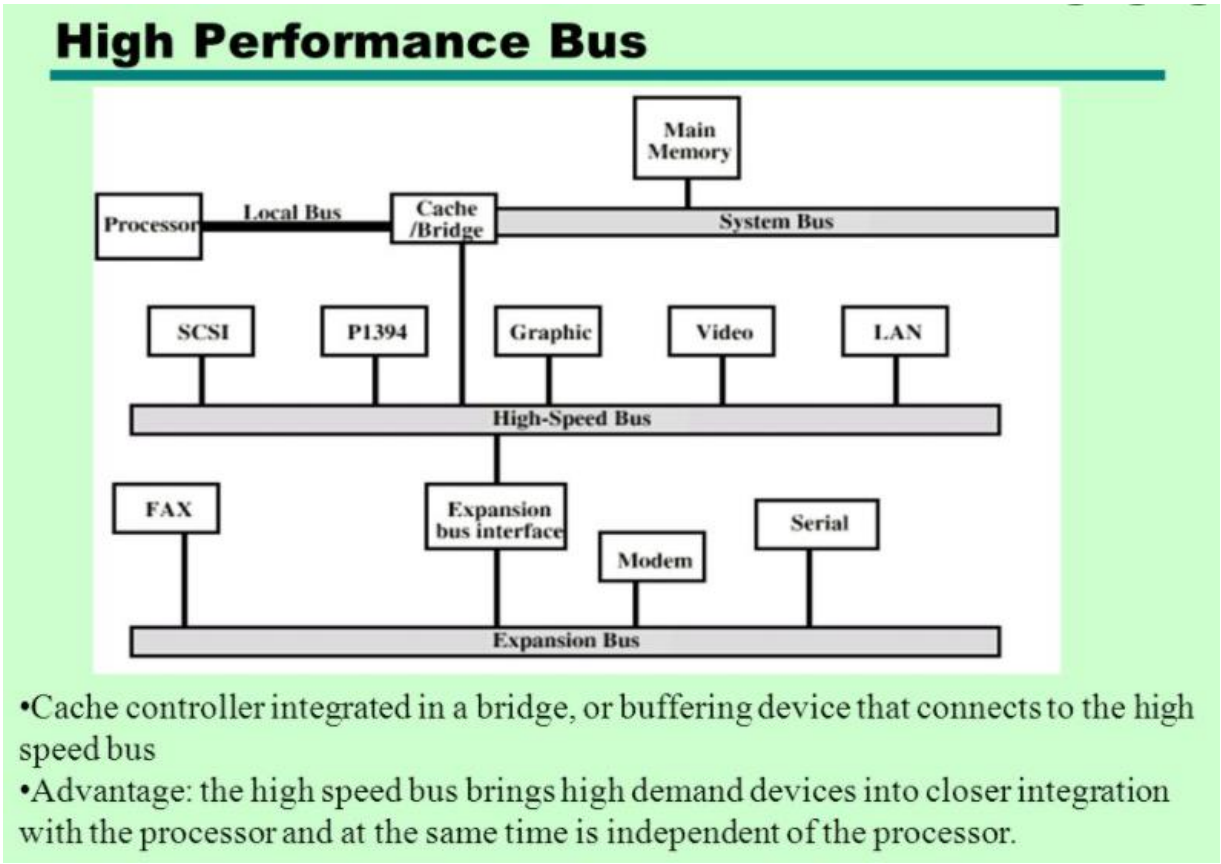


(b) Three-level cache organization

Process vs Memory Performance Gap: Solution

Solution: High-speed buses

Increase the interconnect bandwidth between processors and memory.



•Cache controller integrated in a bridge, or buffering device that connects to the high speed bus

•Advantage: the high speed bus brings high demand devices into closer integration with the processor and at the same time is independent of the processor.

Achieving High Processor Speed

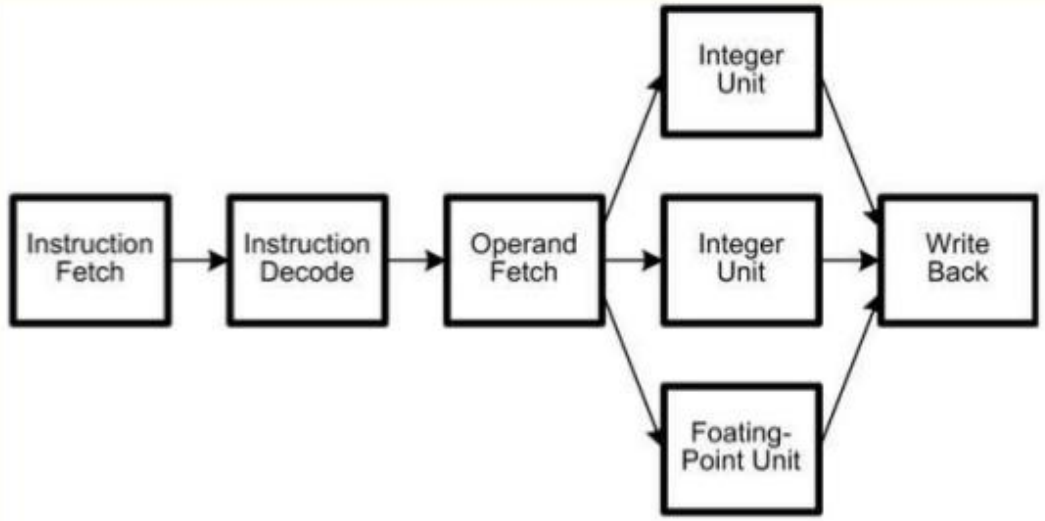
- Increase the hardware speed of the processor. This increase is fundamentally due to shrinking the size of the logic gates on the processor chip, so that more gates can be packed together more tightly and to increasing the clock rate. With gates closer together, the propagation time for signals is significantly reduced, enabling a speeding up of the processor. An increase in clock rate means that individual operations are executed more rapidly.

- Increase the size and speed of caches that are interposed between the processor and main memory. In particular, by dedicating a portion of the processor chip itself to the cache, cache access times drop significantly.

- Make changes to the processor organization and architecture that increase the effective speed of instruction execution. Typically, this involves using parallelism in one form or another.

Achieving High Processor Speed: Superscalar Architecture

- Superscalar Architecture (SSA) describes a microprocessor design that execute more than one instruction at a time during a single clock cycle .
- In a SSA design, the processor or the instruction compiler is able to determine whether an instruction can be carried out independently of other sequential instructions, or whether it has a dependency on another instruction and must be executed sequentially.
- The design is sometimes called "Second Generation RISC". Another term used to describe superscalar processors is multiple instruction issue processors.

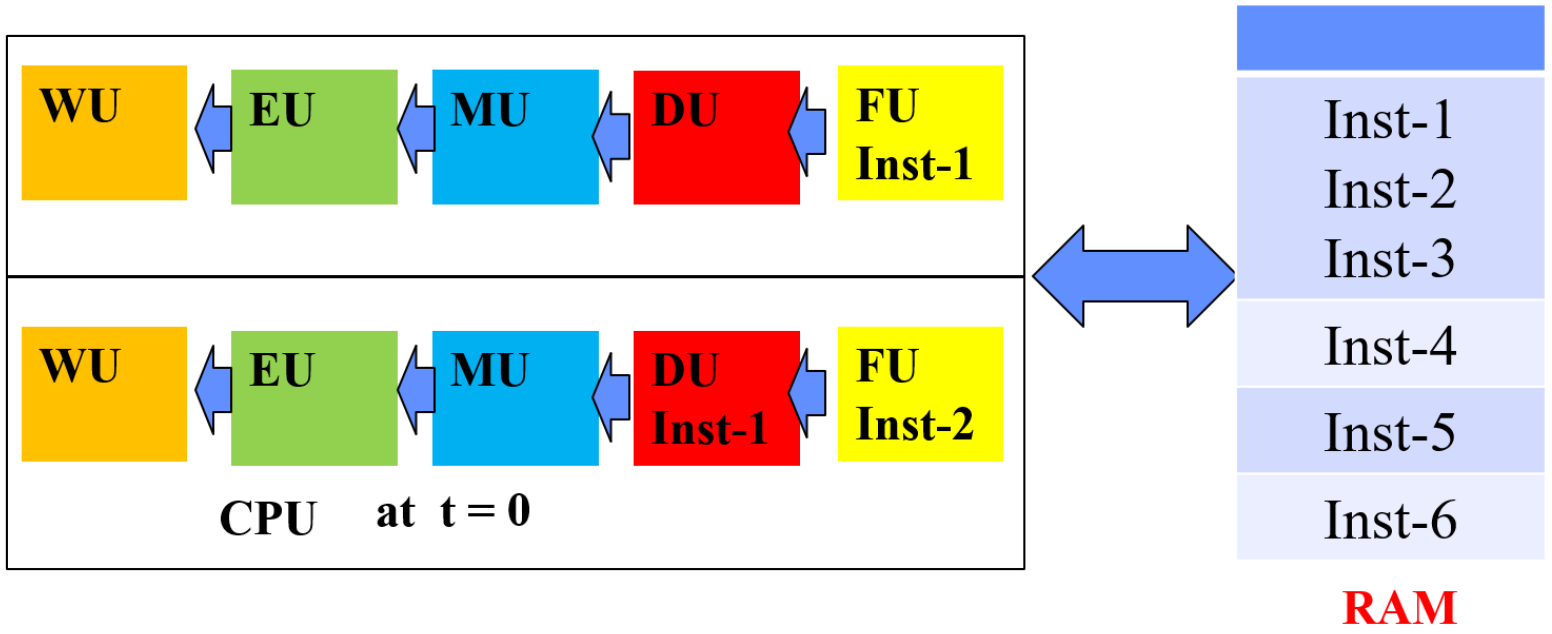


A Superscalar Processor with 3 Functional Units

- In a SSA, several scalar instructions can be initiated simultaneously and executed independently.
- A long series of innovations aimed at producing ever-faster microprocessors.
- Includes all features of pipelining but, in addition, there can be several instructions executing simultaneously in the same pipeline stage.
- SSA introduces a new level of parallelism, called instruction-level parallelism.

- The simplest processors are scalar processors. Each instruction executed by a scalar processor typically manipulates one or two data items at a time.
- In a superscalar CPU the dispatcher reads instructions from memory and decides which one can be run in parallel.
- Therefore a superscalar processor can be proposed having multiple parallel pipelines, each of which is processing instructions simultaneously from a single instruction thread.

Achieving High Processor Speed: Superscalar Architecture



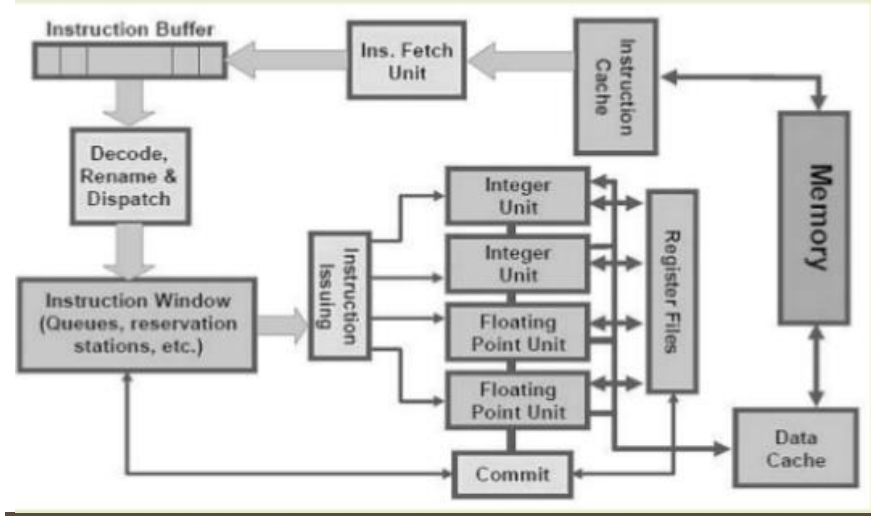
A superscalar approach in essence allows multiple pipelines within a single processor so that instructions that do not depend on one another can be executed in parallel.

Achieving High Processor Speed: Superscalar Architecture

- In Superscalar CPU Architecture implementation of Instruction Level Parallelism (ILP) within a single processor allows faster CPU at a given clock rate.
- A superscalar processor executes more than one instruction during a clock cycle by simultaneously dispatching multiple instructions to functional units.
- Each functional unit is not a separate CPU core but an execution resource within a single CPU such as an arithmetic logic unit, a bit shifter, or a multiplier.

- SimpleScalar is an open source computer architecture simulator which is written using 'C' programming language.
- A set of tools that model a virtual computer system with CPU, Cache and Memory Hierarchy.
- Using the tool, users can model applications that simulate programs running on a range of modern processors and systems.
- The tool set includes sample simulators ranging from a fast functional simulator to a detailed.

Instruction Flow in Superscalar Architecture



Limitations of Superscalar

- Instruction-fetch inefficiencies caused by both branch delays and instruction misalignment
- not worthwhile to explore highly- concurrent execution hardware, rather, it is more appropriate to explore economical execution hardware
- degree of intrinsic parallelism in the instruction stream (instructions requiring the same computational resources from the CPU)
- complexity and time cost of the dispatcher and associated dependency checking logic
- branch instruction processing.

Achieving High Processor Speed

Pipelining

Processor moves data or instructions into a conceptual pipe with all stages of the pipe processing simultaneously

Branch prediction

Processor looks ahead in the instruction code fetched from memory and predicts which branches, or groups of instructions, are likely to be processed next

Data flow analysis

Processor analyzes which instructions are dependent on each other's results, or data, to create an optimized schedule of instructions

Speculative execution

Using branch prediction and data flow analysis, some processors speculatively execute instructions ahead of their actual appearance in the program execution, holding the results in temporary locations, keeping execution engines as busy as possible

Increasing Clock Speed and Chip Density

Problems with Increase clock speed and logic gate density in Processor

- **Power density increases with density of logic gates and clock speed as a result more heat dissipates**
- **May cause Delay in signal transmission within CPU due to increase in Resistance and Capacitance (RC Delay)**
  - **wire interconnects become thinner as logic gate density increases, as a result resistance increases**
  - **Wires get closer increases capacitance**

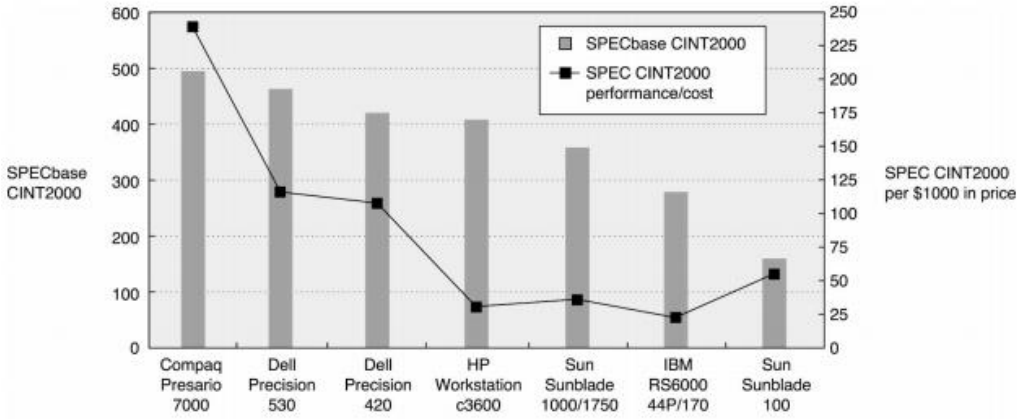
SPEC: System Performance Evaluation Corporation

- Sponsored by industry but independent and self-managed – trusted by code developers and machine vendors
- Clear guides for testing, see www.spec.org
- Regular updates (benchmarks are dropped and new ones added periodically according to relevance)
- Specialized benchmarks for particular classes of applications
- The 2006 version includes 12 integer and 17 floating-point applications
- The SPEC rating specifies how much faster a system is, compared to
- a baseline machine – a system with SPEC rating 600 is 1.5 times
- faster than a system with SPEC rating 400
- Note that this rating incorporates the behavior of all 29 programs – this
- may not necessarily predict performance for your favorite program!

Description	Name	Instruction Count x 10^9	CPI	Clock cycle time (seconds x 10^{-9})	Execution Time (seconds)	Reference Time (seconds)	SPECratio
Interpreted string processing	perl	2252	0.60	0.376	508	9770	19.2
Block-sorting compression	bzip2	2390	0.70	0.376	629	9650	15.4
GNU C compiler	gcc	794	1.20	0.376	358	8050	22.5
Combinatorial optimization	mcf	221	2.66	0.376	221	9120	41.2
Go game (AI)	go	1274	1.10	0.376	527	10490	19.9
Search gene sequence	hmmer	2616	0.60	0.376	590	9330	15.8
Chess game (AI)	sjeng	1948	0.80	0.376	586	12100	20.7
Quantum computer simulation	libquantum	659	0.44	0.376	109	20720	190.0
Video compression	h264avc	3793	0.50	0.376	713	22130	31.0
Discrete event simulation library	omnetpp	367	2.10	0.376	290	6250	21.5
Games/path finding	astar	1250	1.00	0.376	470	7020	14.9
XML parsing	xalancbmk	1045	0.70	0.376	275	6900	25.1
Geometric mean	–	–	–	–	–	–	25.7

FIGURE 1.18 SPECINTC2006 benchmarks running on a 2.66 GHz Intel Core i7 920. As the equation on page 35 explains, execution time is the product of the three factors in this table: instruction count in billions, clocks per instruction (CPI), and clock cycle time in nanoseconds. SPECratio is simply the reference time, which is supplied by SPEC, divided by the measured execution time. The single number

SPEC CINT2000



Summary: Performance

- Performance is specific to a particular program

  total execution time is a consistent summary of performance
- For a given architecture performance increases come from:
  - ❖ increases in clock rate (without adverse CPI affects)
  - ❖ improvements in processor organization that lower CPI
  - ❖ compiler enhancements that lower CPI and/or instruction count

Power Wall

The power wall refers to the electric energy consumption of a chip as a limiting factor for processor frequency increase

- Running out of ideas to improve single thread performance
- Power wall makes it harder to add complex features
- Power wall makes it harder to increase frequency

The energy of a pulse during the logic transition of 0 → 1 → 0 or 1 → 0 → 1

$Energy \propto Capacitive\ load \times Voltage^2$

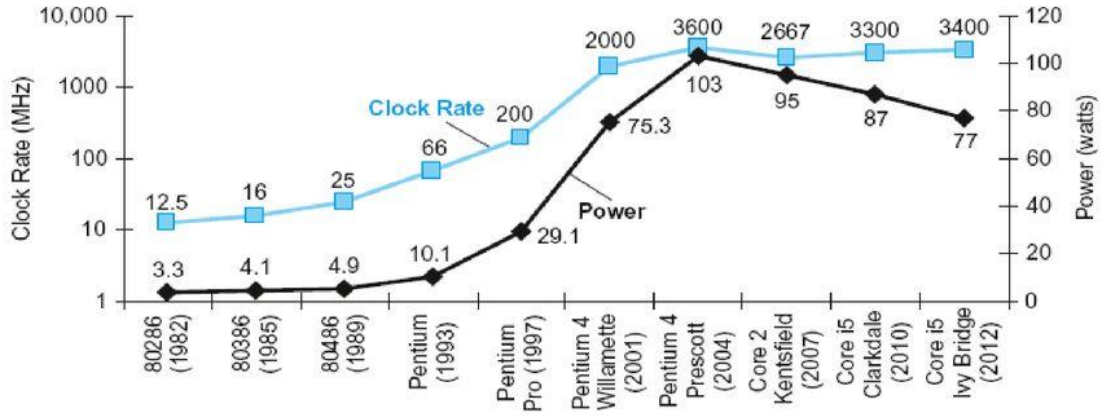
The energy of a single transition

$Energy \propto \frac{1}{2} \times Capacitive\ load \times Voltage^2$

The power required per transistor

$Power \propto \frac{1}{2} \times Capacitive\ load \times Voltage^2 \times Frequency\ switched$

Trends in Clock Rates and Power



❖ Power Wall: Cannot Increase the Clock Rate

- ✧ Heat must be dissipated from a 1.5 × 1.5 cm chip
- ✧ Intel 80386 (1985) consumed about 2 Watts
- ✧ Intel Core i7 running at 3.3 GHz consumes 130 Watts
- ✧ This is the limit of what can be cooled by air

Power Wall: Example Problem and Solution

The energy of a pulse during the logic transition of 0 → 1 → 0 or 1 → 0 → 1

$$Energy \propto Capacitive\ load\ X\ Voltage^2$$

The energy of a single transition

$$Energy \propto \frac{1}{2}\ X\ Capacitive\ load\ X\ Voltage^2$$

The power required per transistor

$$Power \propto \frac{1}{2}\ X\ Capacitive\ load\ X Voltage^2\ X\ Frequency\ switched$$

Suppose we developed a new, simpler processor that has 85% of the capacitive load of the more complex older processor. Further, assume that it has adjustable voltage so that it can reduce voltage 15% compared to processor B, which results in a 15% shrink in frequency. What is the impact on dynamic power?

$$\frac{\text{Power}_{\text{new}}}{\text{Power}_{\text{old}}} = \frac{\langle \text{Capacitive load} \times 0.85 \rangle \times \langle \text{Voltage} \times 0.85 \rangle^2 \times \langle \text{Frequency switched} \times 0.85 \rangle}{\text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency switched}}$$

Thus the power ratio is

$$0.85^4 = 0.52$$

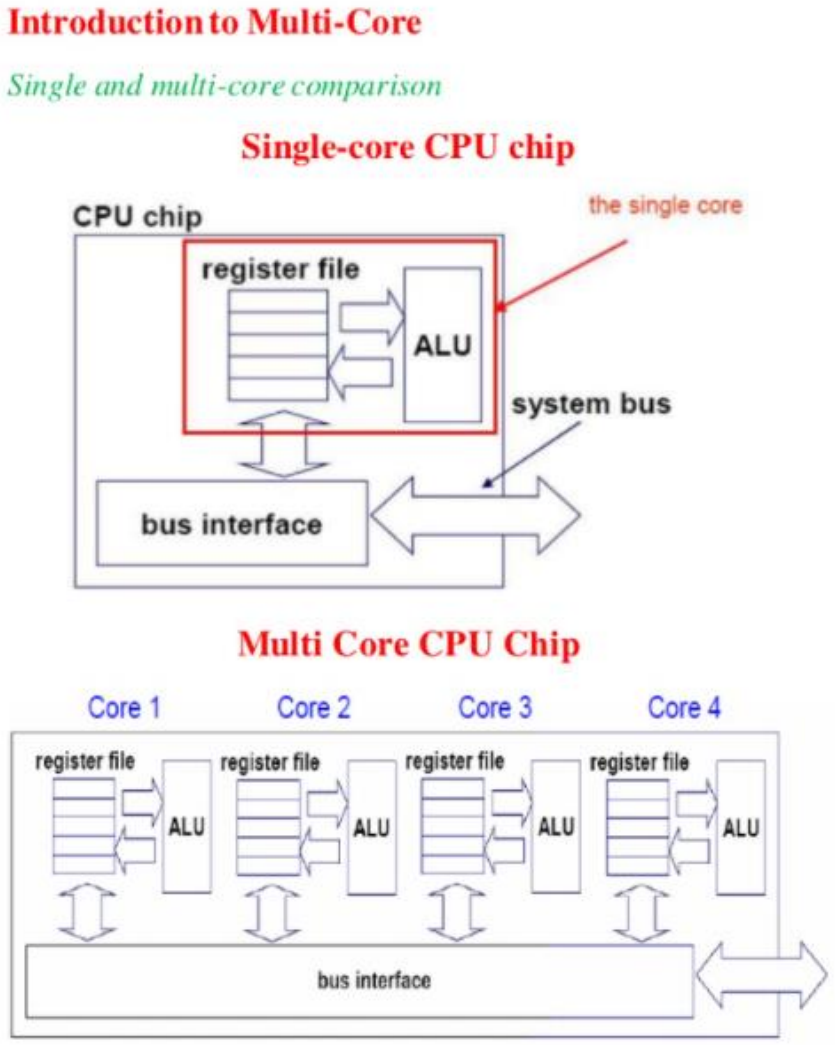
Hence, the new processor uses about half the power of the old processor.

New Approach: Multi-Core Processors

- However, simply relying on increasing clock rate for increased performance runs into the power dissipation problem.
- The faster the clock rate, the greater the amount of power to be dissipated.

$$\text{Power} = 0.5 \times \text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency}$$

- With all of these difficulties in mind, designers have turned to a fundamentally new approach to improving performance: placing multiple processors on the same chip, with a large shared cache.
- The use of multiple processors on the same chip, also referred to as multiple cores, or multicore, provides the potential to increase performance without increasing the clock rate.



New Approach: Multi-Core Processors

- Studies indicate that, within a processor, **the increase in performance is roughly proportional to the square root of the increase in complexity**. But if the software can support the effective use of multiple processors, **then doubling the number of processors almost doubles performance**. Thus, the strategy is to use two simpler processors on the chip rather than one more complex processor.
- In addition, with two processors, larger caches are justified. This is important because **the power consumption of memory logic on a chip is much less than that of processing logic**. In coming years, we can expect that most new processor chips will have multiple processors.

New Approach – Multiple Cores

Multiple processors on single chip with Large shared cache

- **Within a processor, increase in performance proportional to square root of increase in design complexity**

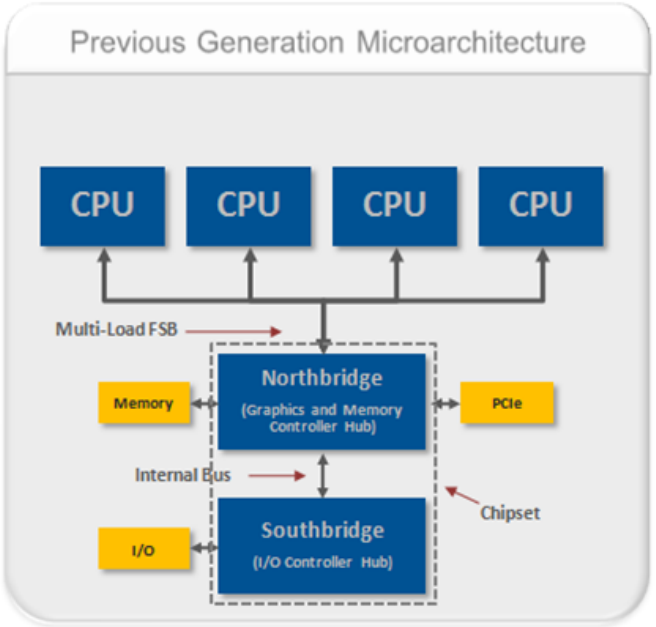
$$performance \propto \sqrt{complexity}$$

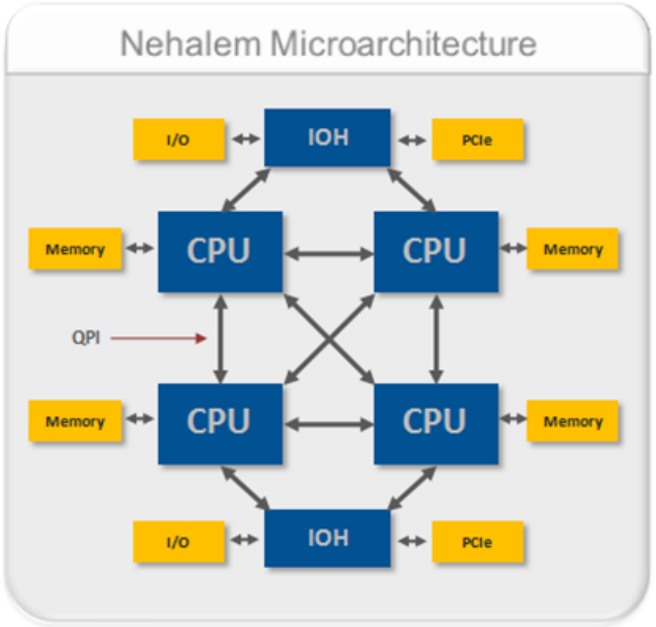
- **If software can use multiple processors, doubling number of processors almost doubles performance**

$$performance \propto no\ of\ cores$$

Example: Intel Core i3, i5 and i7

Core i3 processors have two cores, Core i5 CPUs have four and Core i7 models also have four. Some Core i7 Extreme processors have six or eight cores. Generally speaking, we find that most applications can't take full advantage of six or eight cores.





	Core i3	Core i5	Core i7
1	Entry level processor.	Mid range processor.	High end processor.
2	2-4 Cores	2-4 Cores	4 Cores
3	4 Threads	4 Threads	8 Threads
4	Hyper-Threading (efficient use of processor resources)	Hyper-Threading (efficient use of processor resources)	Hyper-Threading (efficient use of processor resources)
5	3-4 MB Cache	3-8 MB Cache	4-8 MB Cache
6	32 nm Silicon (less heat and energy)	32-45 nm Silicon (less heat and energy)	32-45 nm Silicon (less heat and energy)
7		Turbo Mode (turn off core if not used)	Turbo Mode (turn off core if not used)

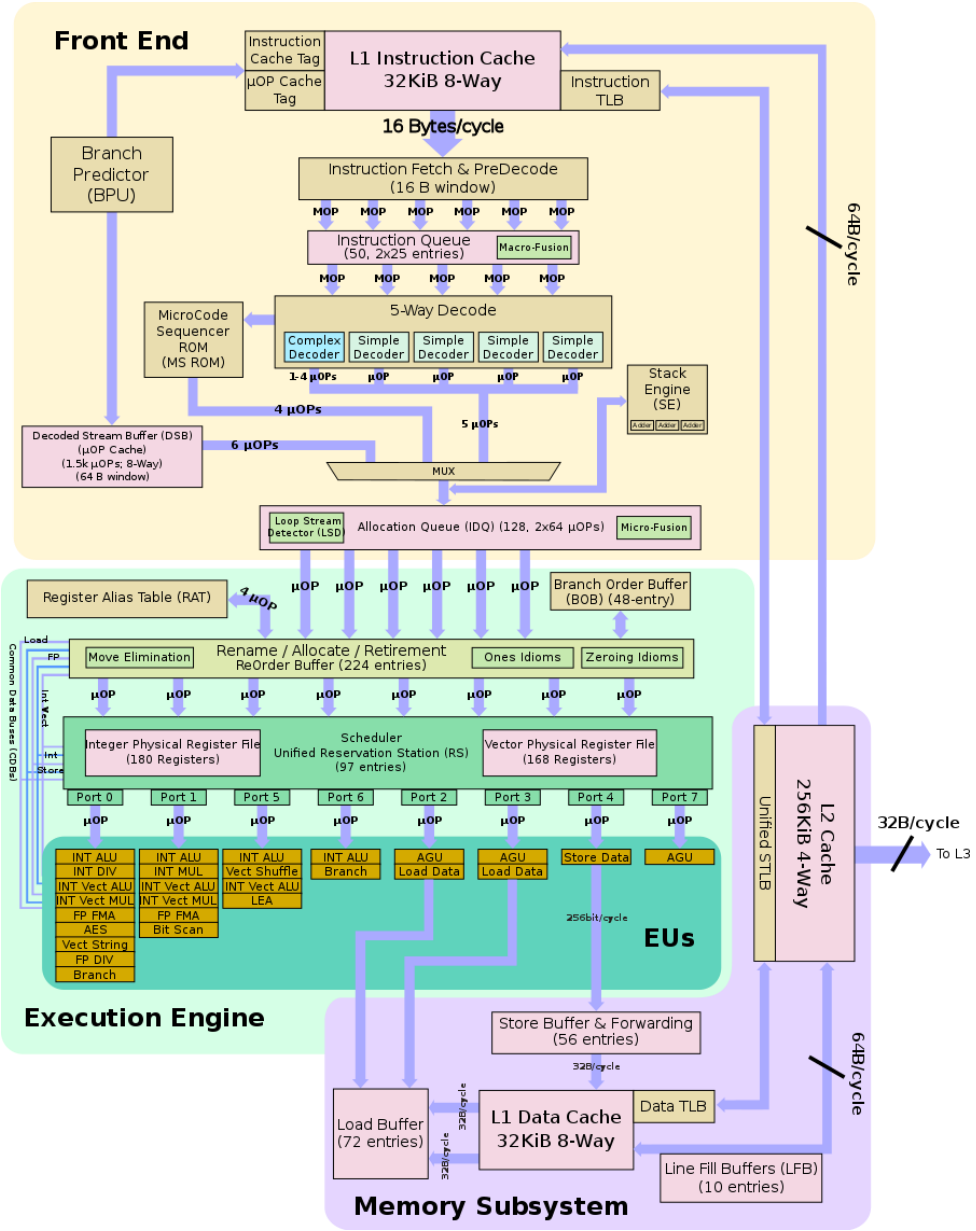
Core i3, i5 and i7

Intel Core i9 Processor Architecture

Core i9-9900K *is a 64-bit octa-core high-end performance x86 desktop microprocessor introduced by Intel in late 2018. This processor, which is based on the Coffee Lake microarchitecture, is manufactured on Intel's 3rd generation enhanced 14nm++ process. The i9-9900K operates at 3.6 GHz with a TDP of 95 W and a Turbo Boost frequency of up to 5 GHz. This chip supports up to 128 GiB of dual-channel DDR4-2666 memory and incorporates Intel's UHD Graphics 630 IGP operating at 350 MHz with a burst frequency of 1.2 GHz.*

https://en.wikichip.org/wiki/intel/core_i9/i9-9900k

https://en.wikichip.org/wiki/intel/microarchitectures/coffee_lake#Memory_Hierarchy




See you next class!