# Computer Organization and Design

Introduction to Computer Performance
(Sections 1.6 - 1.10 from Chapter 1
Plus Ancillary Information)

# Understanding Performance

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

# Performance Goals

- Designing high performance computers is an important goal of a computer architect
  - Cost vs. performance
- Assessing the performance of computer hardware is an important part of computer design
  - Affects the demand and market value of the computer
- How to quantify computer performance
  - What application to use to measure performance?
  - What component of computer to measure? (CPU, I/O, memory)
  - How do other parameters affect performance? (OS, compiler)
  - How do you define performance – faster CPU/memory or quantity of work completed per unit time?

# Executing a Program

- Load the binary executable (or a minimum portion) into memory
  - Some I/O occurs to access the stored program on disk and initiates a read of the program file
  - Operating system allocates memory space for the program
- Operating system initializes CPU for program execution
- CPU begins reading program instructions from memory and executes
- Periodic pauses in execution
  - Disk I/O – either data or more of the program is needed in memory
  - Operating System – temporarily suspends program to deal with other software or hardware operations and then resumes
- Program finishes execution
  - Operating system cleans up
  - Prepares for next program
- Time is associated with all of these operations
  - How do you narrow down what to measure?

# Basic Performance Measures

- Execution time (response time)
  - Time between start and completion of task
  - Task can  be any unit of work – program, job, procedure
  - Focus is on time
- Throughput
  - Total amount of work completed in a given time
  - Jobs/day, processes/hour, instructions/second
  - Focus is on work
- Importance of one over the other depends on perspective – ex. user vs CIO

# Abstract Example

- Consider the impact of the following changes to a computer system with regard to execution time and throughput
  - 1. Replace the processor with a faster version
  - 2. Add additional processors to a computer that uses multiple processors for separate tasks
- Option 1 would improve both execution time and throughput
  - Decreasing execution time almost always improves throughput
- Option 2 would only improve throughput
  - Individual tasks execute at the same speed but more tasks can run concurrently

# Performance Measures

- The primary measure of performance is execution time
  - Exception is I/O where throughput is important
- To maximize performance of an application, we need to minimize its execution time
- The relationship between performance and execution time is represented by the expression

$$Performance_X = \frac{1}{Execution\,Time_X}$$

# Performance Comparison

- When comparing two systems, we can say if the performance of computer X is better than computer Y, then

$$Performance_X > Performance_Y$$

$$Execution\ Time_Y > Execution\ Time_X$$

$$\frac{1}{Execution\ Time_X} > \frac{1}{Execution\ Time_Y}$$

# Performance Comparison (continued)

- The quantitative difference in performance between two computers is represented

$$\frac{Performance_x}{Performance_y} = \frac{Execution\ time_y}{Execution\ time_x} = n$$

- X is *n* times faster than Y
- If X is *n* times faster than Y, then the execution time on Y is *n* times longer than it is on X
- The value *n* is the ratio of the measured performance between two computer systems

# Performance Comparison (continued)

- Example:
  - If computer A runs a program in 10 microseconds and computer B runs the same program in 15 microseconds, how much faster is A than B?

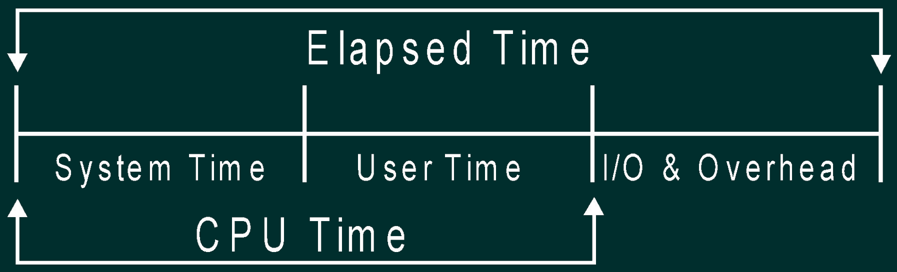
$$\frac{Execution\ time_B}{Execution\ time_A} = \frac{15}{10} = 1.5$$

# System Performance

- Total system performance depends on the performance of the individual components:
  - CPU
  - Memory
  - I/O
- CPU – execution engine
  - Performance defined by computational speed
- Memory – short term storage
  - Performance defined by speed of access
- I/O  - long term storage
  - Performance defined by data throughput

# Measuring Performance

- Time is the measure of computer performance
- Two kinds of time
  - Wall-clock time (response time, elapsed time)
    - used to measure the total time to complete a task or job
      - includes I/O time, memory accesses, and OS overhead
  - CPU execution time (CPU time)
    - used to measure the time the processor spends working on a single task or job
    - not including I/O time or time spent on other tasks
- CPU time can be further divided
  - User CPU time – time spent working on a program
  - System CPU time – time spent by OS doing things for the program



# Measuring Performance (continued)

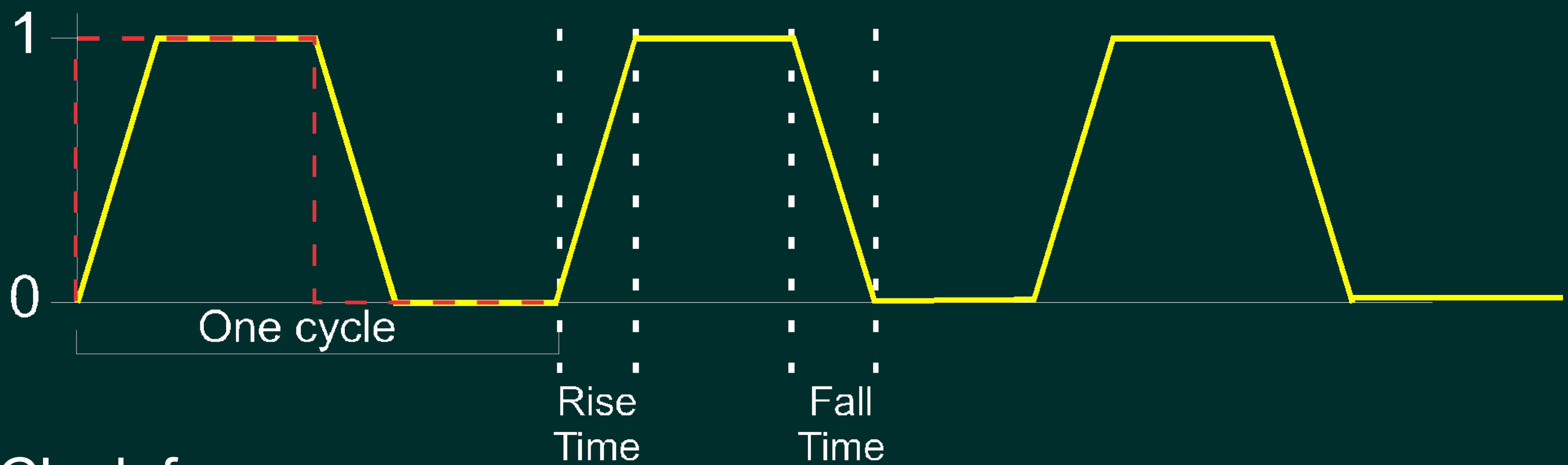
- CPU performance measurements
  - use CPU time rather than response time because it is more accurate
- User CPU time is preferred
  - because of its independence of the operating system and other factors
- The time required for a computer to perform a task (run a program) is dependent on the speed of the hardware it runs on
  - Execution time is measured in seconds per program

# Finite State Machine

- A mathematical model of computation used to design both computer programs and sequential logic circuits
- A computer is an implementation of a finite state machine
  - At any given point in time, the state of the machine is defined by the individual values present on the inputs and outputs of its components
- These values are combinations of 0s and 1s
- The operation of the machine consists of transitions from one state to the next
- The change in state is initiated by a triggering event
  - The clock

# The Clock

- Clock signals

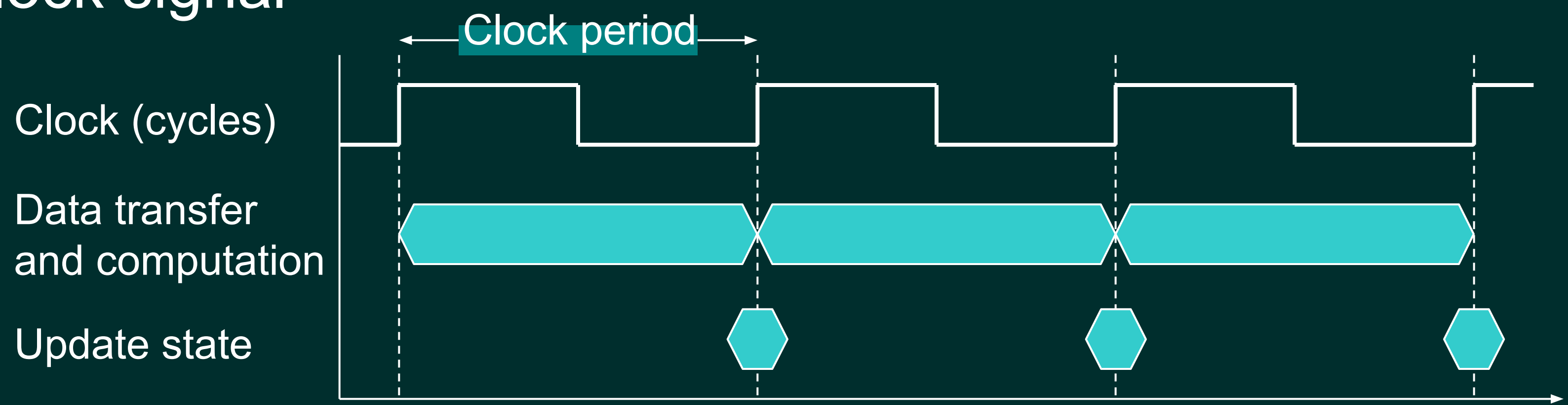



- Clock frequency
  - Cycles per second (CPS) or Hertz (Hz)        [kHz, MHz, GHz]
- Clock period
  - Seconds or fraction of (millisecond, microsecond, nanosecond)
  - Calculated as inverse of frequency ( 1/Hz)
- Rise time/Fall time
  - Time to go from 0 to 1 or from 1 to 0

# Actual Clock Waveform

# Hardware and Timing

- A circuit called a *clock* generates timing signals that control all the other circuits in the computer
- Clock signal




- A computer's clock speed is described using two terms
  - Frequency - cycles per second or Hertz (Hz)
    - **clock rate**
  - Period - time for one cycle – inverse of frequency - seconds
    - **clock cycle time**

# Calculating CPU Time

- CPU time is calculated using the following two formulas

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

$$CPU_{Time} = \frac{CPU\ Clock\ Cycles}{Clock\ Rate}$$

- CPU performance is improved
  - By reducing the length of a clock cycle or
  - By reducing the number of clock cycles required for a program

# Example

- CPU A
  - Runs a program in 10 seconds
  - Has a 4 GHz clock
- CPU B
  - Being designed
  - Want to run the same program in 6 seconds
  - The clock rate can be made to run faster
  - BUT it will affect the CPU design such that the new computer will need 1.2 times as many clock cycles as CPU A.
- What clock rate do we need to achieve on the new CPU?

# Example (continued)

- First, calculate the number of clock cycles required for the program on CPU A

$$CPU\ time_A = \frac{CPU\ clock\ cycles_A}{Clock\ rate_A}$$

$$\boxed{\begin{aligned} CPU\ clock\ cycles_A &= CPU\ time_A \times Clock\ rate_A \\ &= 10\ seconds\ \times (4 \times 10^9)\ cycles\ per\ second \\ &= 40\ \times\ 10^9\ cycles \end{aligned}}$$

# Example (continued)

- The CPU time for CPU B can now be calculated as follows

$$CPU\ time_B = \frac{1.2 \times CPU\ clock\ cycles_A}{Clock\ rate_B}$$

$$6\ seconds = \frac{1.2 \times 40 \times 10^9\ cycles}{Clock\ rate_B}$$

$$Clock\ rate_B = \frac{1.2 \times 40 \times 10^9\ cycles}{6\ seconds}$$

$$Clock\ rate_B = 8\ GHz$$

- CPU B needs a clock rate twice that of CPU A
  - But can we actually achieve that clock rate?
  - Are there other issues that need to be considered?

# Program Execution Time

- The CPU time for a program directly depends on the number of instructions in that program

| CPU clock cycles | = | # instructions for a program | x | Average clock cycles per instruction |
|---|---|---|---|---|

- The term *clock cycles per instruction* is often abbreviated CPI
- The CPI of a program depends on the instruction set of the computer and the compiler
- In other words, the CPI depends on the architecture of the system

# Example

- Suppose we have two implementations of the same instruction set architecture.
  - Example:  Intel x86 vs. AMD x86
  - CPU A has a clock cycle time of 10 ns and a CPI of 2.0 for a given program.
  - CPU B has a clock cycle time of 20 ns and a CPI of 1.2 for the same program.
- Which machine is faster for this program?
- Assume the program requires $I$ instructions to be executed.
  - $I$ is the variable representing the unknown number of instructions

# Example (continued)

- First calculate the clock cycles for each CPU
  - CPU clock cycles A = $I$ x 2.0
  - CPU clock cycles B = $I$ x 1.2

- Next calculate CPU time
  - CPU time A = 2.0$I$ x 10 ns = 20$I$ ns
  - CPU time B = 1.2$I$ x 20 ns = 24$I$ ns

- CPU A is 24 / 20 = 1.2 times faster than CPU B

# Program CPU Time

- The CPU time for a program, which is our main measure of performance, can be written as

$$\text{CPU time} = \begin{array}{c}\text{Instruction}\\ \text{count}\end{array} \times \text{CPI} \times \begin{array}{c}\text{Clock}\\ \text{Cycle time}\end{array}$$

$$\text{CPU time} = \frac{\text{Instruction count} \times \text{CPI}}{\text{Clock rate}}$$

- The performance of the CPU is directly dependent on
  - the clock speed
  - the number of cycles per instruction
  - the number of instructions per program - instruction count (IC)

# CPI (Cycles Per Instruction)

- An instruction is a unit of work in the CPU
- Different types of instructions require more or less time to execute in terms of number of cycles
  - Ex. An add instruction may require fewer clock cycles to execute than a memory access instruction
- 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

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

# Calculating CPI

- A RISC-V microprocessor has been designed to achieve a target clock frequency.  The various instruction classes require varying clock cycles as shown in the following table:

| Intstruction Class | Instruction Latency |
|---|---|
| Loads | 5 cycles |
| Stores | 4 cycles |
| Branches | 3 cycles |
| Jumps | 2 cycles |
| ALU/Logic | 4 cycles |

- What is the average CPI for the microprocessor?
  - Simply average the latencies for all instruction classes
  - CPI = ( 5 + 4 + 3 + 2 + 4) / 5 = 3.6 cycles

# Calculating CPI

- A program is executed on the previously defined RISC-V microprocessor. The relative frequency of each class of instruction, which is program dependent, is added to the table:

| Intstruction Class | Instruction Latency | Relative Frequency |
|---|---|---|
| Loads | 5 cycles | 30% |
| Stores | 4 cycles | 15% |
| Branches | 3 cycles | 10% |
| Jumps | 2 cycles | 5% |
| ALU/Logic | 4 cycles | 40% |

- What is the average CPI for the <u>program</u>?
  - Factor the instruction frequency with instruction latency
  - CPI = (5 * 0.3) + (4 * 0.15) + (3 * 0.1) + (2 * 0.05) + (4 * 0.4)
    = 4.1 cycles

# Components of Performance

| Performance Component | Unit of Measure |
|---|---|
| CPU execution time for a program | Seconds for the program to run |
| 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 |

# Contributors to Performance

| Hardware or Software Component | Affects What? |
|---|---|
| Algorithm | Instruction count and possibly the CPI |
| Programming language | Instruction count and CPI |
| Compiler | Instruction count and CPI |
| Instruction set architecture | Instruction count, clock rate and CPI |

# Performance Tradeoffs

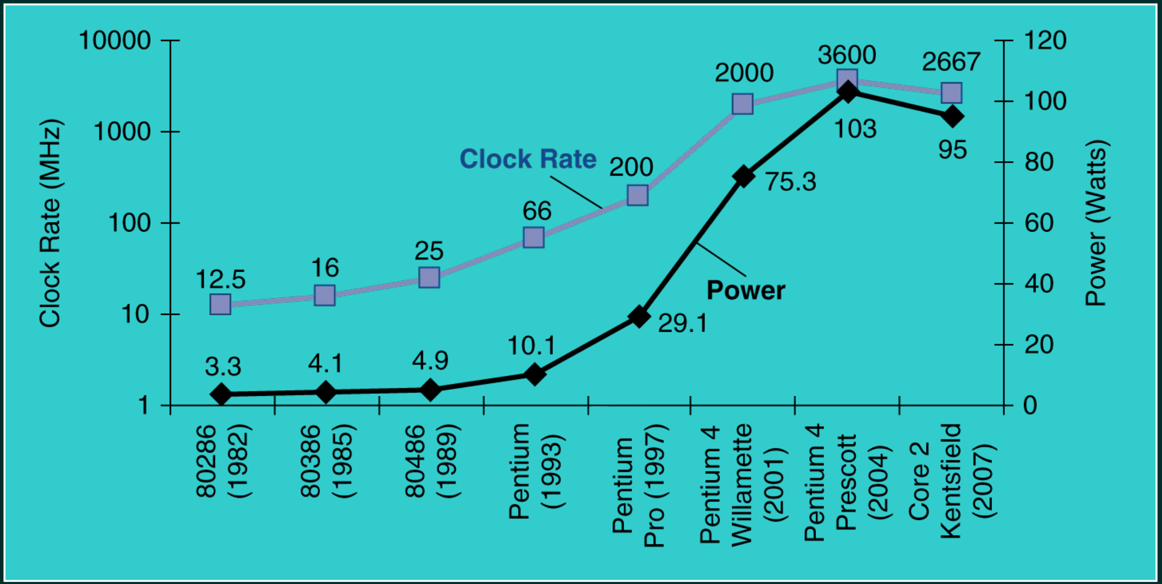
- Performance is equally dependent on each of the three terms: IC, CPI, and clock cycle time
- When comparing machines, you must consider all three components
- There are trade-offs between the factors that affect performance
  - optimizing one of them can lead to worsening the performance of another
  - degrade overall performance

# Benchmarks

- Programs that are used to measure performance
- Execute a specific set of programs that represents the typical real world workload
- Most widely used benchmark is SPEC (Standard Performance Evaluation Corporation)
  - CPU
  - Graphics/Workstations
  - High-Performance Computing, MPI/OMP
  - Java Client/Server
  - Mail Servers
  - Network File System
  - Power
  - SIP (Session Initiation Protocol)
  - Virtualization
  - Web Servers

# Power Trends

- Dominant technology for integrated circuits today is CMOS (Complimentary Metal Oxide Semiconductor)
  - Primary power consumed during switching (dynamic power)
- Dynamic power is measured in watts based on the following equation:
  - $\text{Power}_{\text{Dynamic}} = 1/2 \times \text{Capacitive load} \times \text{Voltage}^2 \times \text{Switching frequency}$




Clock rate and power for Intel x86 microprocessors over 8 generations

# Power Problems

- Want to build faster processors?
  - Natural tendency is to increase frequency which makes power increase.
  - So, now reduce voltage to help reduce power but find that more power is consumed by leaky transistors (even the ones that are turned off and not switching).
  - So, increase insulation properties of transistors but it only helps a little and there's not much room to add a lot of insulation around the transistors and wires and still get everything to fit on the chip.
- Ultimately, you get to the point where you can't go forward without employing more expensive cooling methods that are not practical for consumer systems.
- This limitation is referred to as the power wall
  - Can't improve performance with current designs by increasing frequency, lowering voltage and reducing capacitive load
  - Limit for cooling commodity general purpose processors

# The Shift to Multiprocessors

- Small increases in performance in uniprocessors (single core) will continue to be made but not at the same rate as in the past.
  - There is still a market for uniprocessors (embedded systems/IoT)
- New general purpose CPU designs are based on multicore processors
  - More than one CPU (core) on a single chip
  - Newer designs are hybrid core (big.LITTLE or other)
- New Moore's Law?
  - What will define and guide future development of microprocessors when doubling transistor counts no longer can be done?
  - Domain specific architectures, more hybrid designs
- Not feasible to continue increasing core counts when most applications cannot utilize more processors
  - There is a need to program differently to take advantage of multiple hardware

# Parallelism

- Parallelism is a concept that describes the ability to run more than one thing at a time.
  - Multiprocessor mainframes have existed since the 1970s
  - Each processor (a uniprocessor) could be assigned a program or a part of a program to run under operating system control
  - Goal was to increase throughput
- Instruction level parallelism (ILP) refers to the abstract parallelism among instructions in a program
  - If two instructions are not dependent on each other, then they could be executed in parallel providing adequate hardware resources
  - Pipelining (Chapter 4) is a form of ILP
  - Hardware executes multiple instructions at once
  - Details are hidden from programmer
    - Don’t have to change or recompile programs for ILP

# Multiprocessor Performance

- Gains in performance for multicore processors requires programming for performance
  - Programmers have to design their programs to take advantage of parallelism
  - Explicitly Parallel Programming (EPP)
    - Can’t rely on hardware alone to increase performance
    - Load balancing becomes programmer responsibility
    - Inter-processor communication and synchronization must be optimized
    - New programming paradigm
      - Programmers have to learn new ways of writing algorithms to achieve higher parallelism

# Amdahl’s Law

- Used to calculate improvement in performance based on isolating the feature improved.
- A rule stating that the performance enhancement possible with a given improvement is limited by the amount that the improved feature is used.
- Also known as the law of diminishing returns.
- Is used to calculate improvements in performance for both software and hardware

# Amdahl's Law (Example 1)

- Suppose we are considering an enhancement that runs 10 times faster than the original system but is used only 40% of the time. What is the overall speedup gained by incorporating the enhancement?

- Amdahl's Law formula:

$$S_O = \frac{1}{(1 - F) + \frac{F}{S_E}}$$

  - $S_E$ = speedup of enhancement
  - F = fraction enhancement is used
  - $S_O$ = overall speedup

$$S_0 = \frac{1}{(1 - 0.4) + \frac{0.4}{10}} = 1.5625$$

# Amdahl's Law (Example 2)

- A program takes 100 seconds to run; 80 seconds are consumed by multiply instructions. How much do I have to improve the multiplication speed so the program will run 5 times faster?

Execution time after improvement =

$$\frac{\text{Execution time affected by improvement}}{\text{Amount of improvement}}$$

$$+ \text{ Execution time unaffected}$$

# Amdahl's Law Example (cont)

$$Execution\ time\ after\ improvement = \frac{80\ seconds}{n} + (100 - 80\ seconds)$$

$$20\ seconds = \frac{80\ seconds}{n} + 20 seconds$$

$$0 = \frac{80\ seconds}{n}$$

***Can't be done!***

# Another Example

- If 30% of the execution time of a program can be enhanced, the speedup ***p*** will be 0.3.
- If the improvement makes the affected part twice as fast, ***s*** will be 2. Amdahl's law states that the overall speedup of applying the improvement will be:

$$S_{latency} = \frac{1}{1 - p + \frac{p}{s}} = \frac{1}{1 - 0.3 + \frac{0.3}{2}} = 1.18$$

# Another Example

- Assume that we are given a serial task which is split into four consecutive parts
- The percentages of execution time are
  - p1 = 0.11,   p2 = 0.18,   p3 = 0.23,   p4 = 0.48
- The speedups for each part is as follows:
  - s1 = 1,   s2 = 5,   s3 = 20,   s4 = 1.6
- By using Amdahl's law, the overall speedup is

$$S_{latency} = \frac{1}{\frac{p1}{s1} + \frac{p2}{s2} + \frac{p3}{s3} + \frac{p4}{s4}} = \frac{1}{\frac{011}{1} + \frac{0.18}{5} + \frac{0.23}{20} + \frac{0.48}{1.6}}$$

$$S_{latency} = 2.19$$

Notice how the 5 times and 20 times speedup on the 2nd and 3rd parts respectively don't have much effect on the overall speedup when the 4th part (48% of the execution time) is accelerated by only 1.6 times.

# Terminology / Vocabulary

- Execution time (response time)
- Throughput
- User CPU time vs. System CPU time
- Clock
- Frequency (clock rate)
- Period (cycle time)
- CPI (cycles per instruction)
- Benchmarks
- Power
- ILP (Instruction Level Parallelism)
- Amdahl's Law