GEORGE MASON UNIVERSITY

GMU CS695 Advanced Computer Architecture

Spring 2023

Introduction

Lishan Yang

Reference:
Prof. Onur Mutlu, Computer Architecture (Fall 2022)
CMPE 140, Hyeran Jeon
CMPE 140, Donald Hung
CMPE 140, Haonan Wang
Computer Architecture: A Quantitative Approach

Course staff

- Instructor
  - Dr. Lishan Yang (web: https://lishanyang.github.io)
  - Email: lyang28@gmu.edu
  - Research interest: System reliability & architecture
    - GPUs, non-conventional sensors, autonomous driving systems, neural networks

Course staff

- GTA:

Getting help

- My office hours
  - Mon 8:30 am – 9:30 am, ENGR 4610 (in person)
  - Wed 7pm - 8pm, online https://gmu.zoom.us/j/9306783539?pwd=emlVSDBtcE5DMmhlVG8rOFV4VHArQT09
  - By appointment

- GTA's office hours
  - Wednesday
  - Location:

- Piazza
  - https://piazza.com/gmu/spring2023/cs695

Big picture course goals

- Demonstrate an understanding of the fundamental concepts in computer architecture
- Get an overview of the recent computer architecture design and implementations
- Have the ability to quantitatively evaluate the performance of a computer architecture
- Optimize programs and applications based on the underlying architecture features
- Demonstrate an ability to design, prototype, and implement an idea to improve computer architecture

Lectures

- (Review) + lecture
- Slides available on Blackboard (night before)
- Tentative schedule:
  - Jan - March: Fundamentals of computer architecture
  - Midterm (3/6)
  - April: Emergining computer architecture

Calendar (tentative)

- Readings, assignments, due dates
- Less concrete further out; don’t get too far ahead

Textbooks?

Deeper

- Computer Architecture: A Quantitative Approach
- Papers (required or optional) serve as reference for many topics that aren't directly covered by a text
- Slides/lecture notes
- Suggested pre-reading:

Easier

Computer Systems: a Programmer's Perspective

Collaboration Policy

- Students must work individually on all homeworks and assignments
- High-level discussions are encouraged
- Anything that you hand in, must be written in your own words
  - AI-assisted tools: prohibited (for example, chatGPT)

- **Violation of the Honor Code will result in an F.**

- The only teamwork project: course project
  - Groups must work individually
  - High-level discussions are encouraged among different groups
  - Do not look at other groups' raw code and data!
  - Anything that you hand in, must be written in your own words
    - AI-assisted tools: prohibited (for example, chatGPT)

Grading

- 35% course project
- 5% quizzes and in-class activities
- 10% homeworks
- 20% midterm exam
- 30% final exam

- Contesting of grades must be requested within one week of receiving the grade on Blackboard

Late Policy

- The deadline of assignments and course project is 11:59pm New York time of the due date.
- 10% will be deducted for late assignments each day after the due date.
    - If an assignment is late, we will grade it and scale the score by 0.9 if it is up to one day late, by 0.8 if it is up to two days late, and by 0.7 if it is up to three days late.
- Late assignments will only be accepted for 3 days after the due date.
- Assignments submitted more than 3 days late will receive a zero.
- Each student gets two "Emergency Day" tokens, which are automatically used by late submissions to avoid the degrading penalty.
- Unused emergency-tokens will be worth 0.25% bonus (+0.25 to your final grade) to a student's overall grade at the semester's end.

Late Policy

- Blackboard being unavailable is not an excuse for turning in a late assignment
- In the rare situation that the website is somehow unavailable or giving the student an error, the student MUST email their submission to the instructor or GTA before the deadline, otherwise it will be considered late.
- Catastrophic computer failure will not be cause for an extension. Use a backup service such as DropBox (or any cloud service), emailing yourself, making multiple rounds of submissions to Blackboard, whatever it takes.

Course Project

- 3/27: Project proposal presentation
- 5/1: Final project presentation
- 5/8: Written report due

- Group size: 1~4 students
- It is your responsibility to find reliable teammates. Students in the same group share the same grade of the course project.
    - In rare cases, please talk to the instructor

- (We will talk about how to do the presentation later in the semester)

Course Project

1. Find your teammate and topic
2. Start to work on the project and try to see if it is doable
   - Hardware requirement? Coding load? Too easy? Too difficult?
3. Proposal presentation
   - What's the problem? What's your goal? What's your plan?
4. Continue working on the project…
   - More debugging, discussion, …
5. Final presentation
   - What did you accomplish? What's your conclusion?
6. Written report

Office hours!
Discussion with the instructor and students

Course Project

- Written report:
  - It is a paper (We will talk about how to write it later in the semester)

Group size	Paper Length
1	6
2	7
3	8
4	9

- Some suggestions
  - Get started early
  - Communication! Regular meeting
    - Split your work and responsibility
    - Always summarize the next steps at the end of the meeting
- A shared doc/folder and github

The code and experiments of your project needs to be submitted with Github history

Get started early

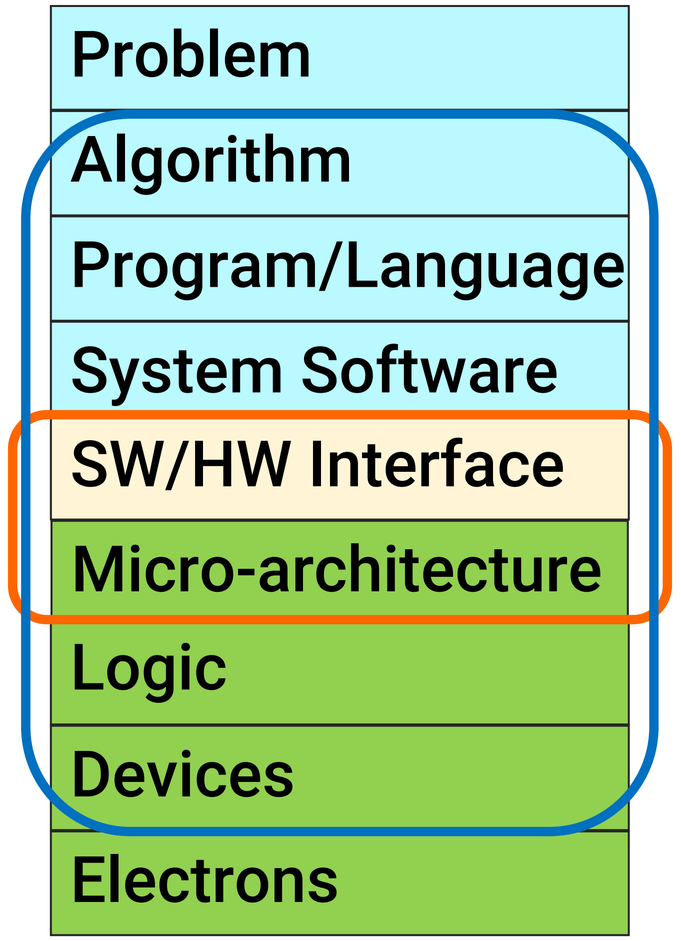
Virginia is for C Programming

Campus Resources

- Disability Services: GMU’s Office of Disability Services (http://ds.gmu.edu).
- Student Support Resources on Campus: https://stearnscenter.gmu.edu/knowledgecenter/knowing - mason-students/student-support-resources-oncampus

What is “Architecture”?

What is "Architecture"?

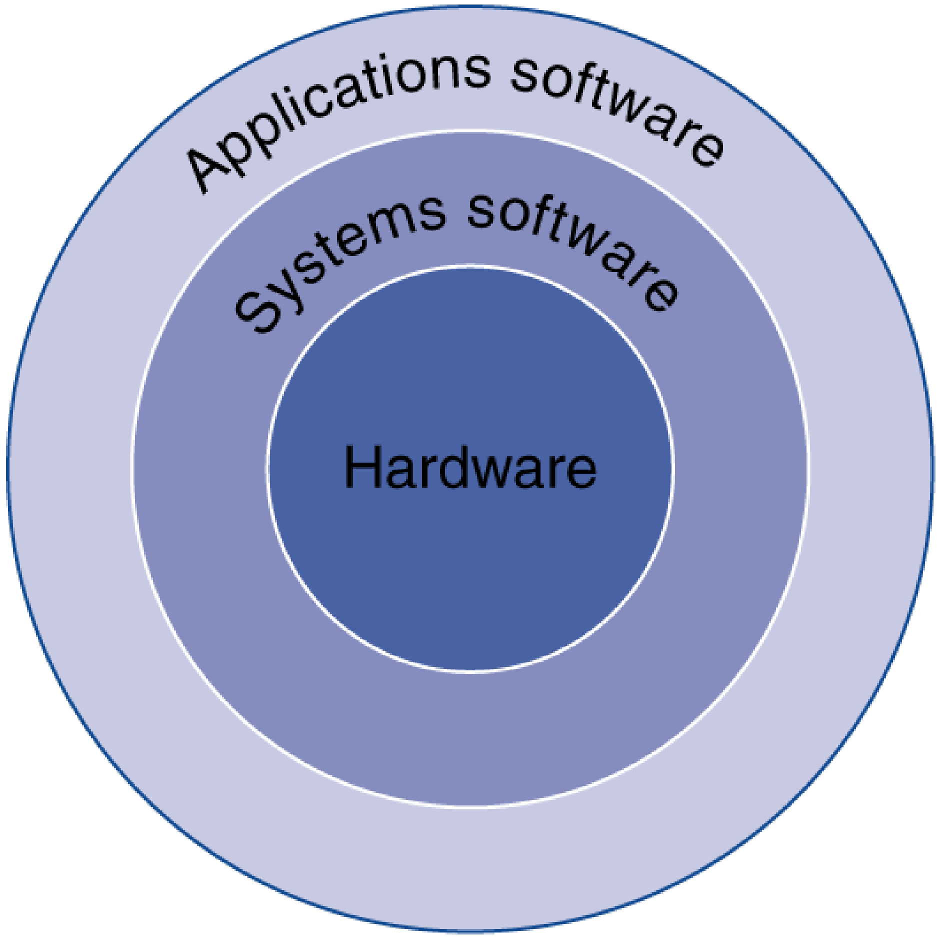


Computer Architecture (expanded view)

Computer Architecture (narrow view)

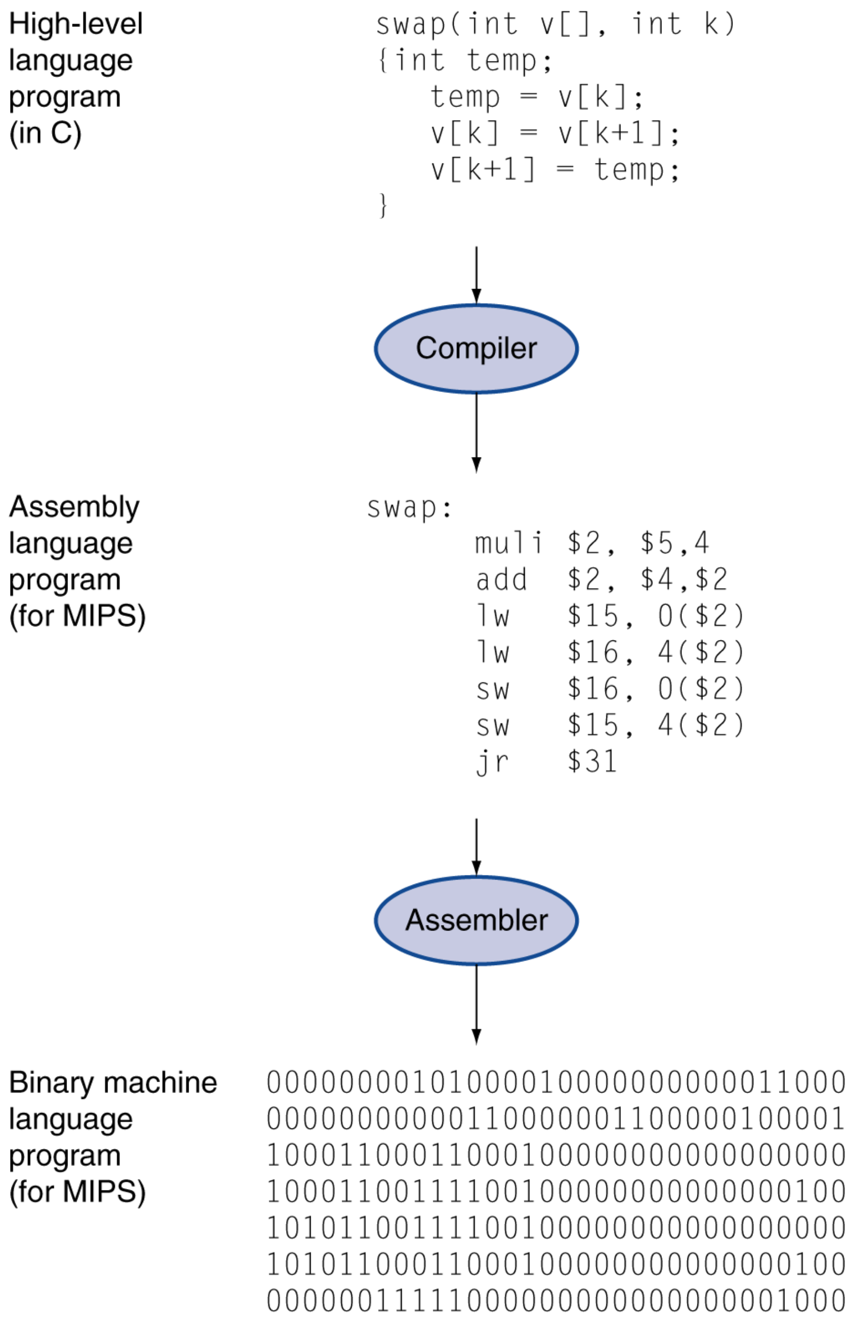
The Software's Point of View

- **Application software**
  - Written in high-level language
- **System software**
  - **Compiler:** translates HLL code to machine code
  - **Operating System:** service code
    - Handling input/output
    - Managing memory and storage
    - Scheduling tasks & sharing resources
- **Hardware**
  - Processor, memory, I/O controllers, …



The Codes' Point of View

- **High-level language**
  - Abstraction Level closer to problem domain
  - Provides for productivity and portability
- **Assembly language**
  - Textual representation of instructions
- **Hardware representation**
  - Binary digits (bits)
  - Encoded instructions and data


High-level language program (in C)
Compiler
Assembly language program (for MIPS)
Assembler
Binary machine language program (for MIPS)

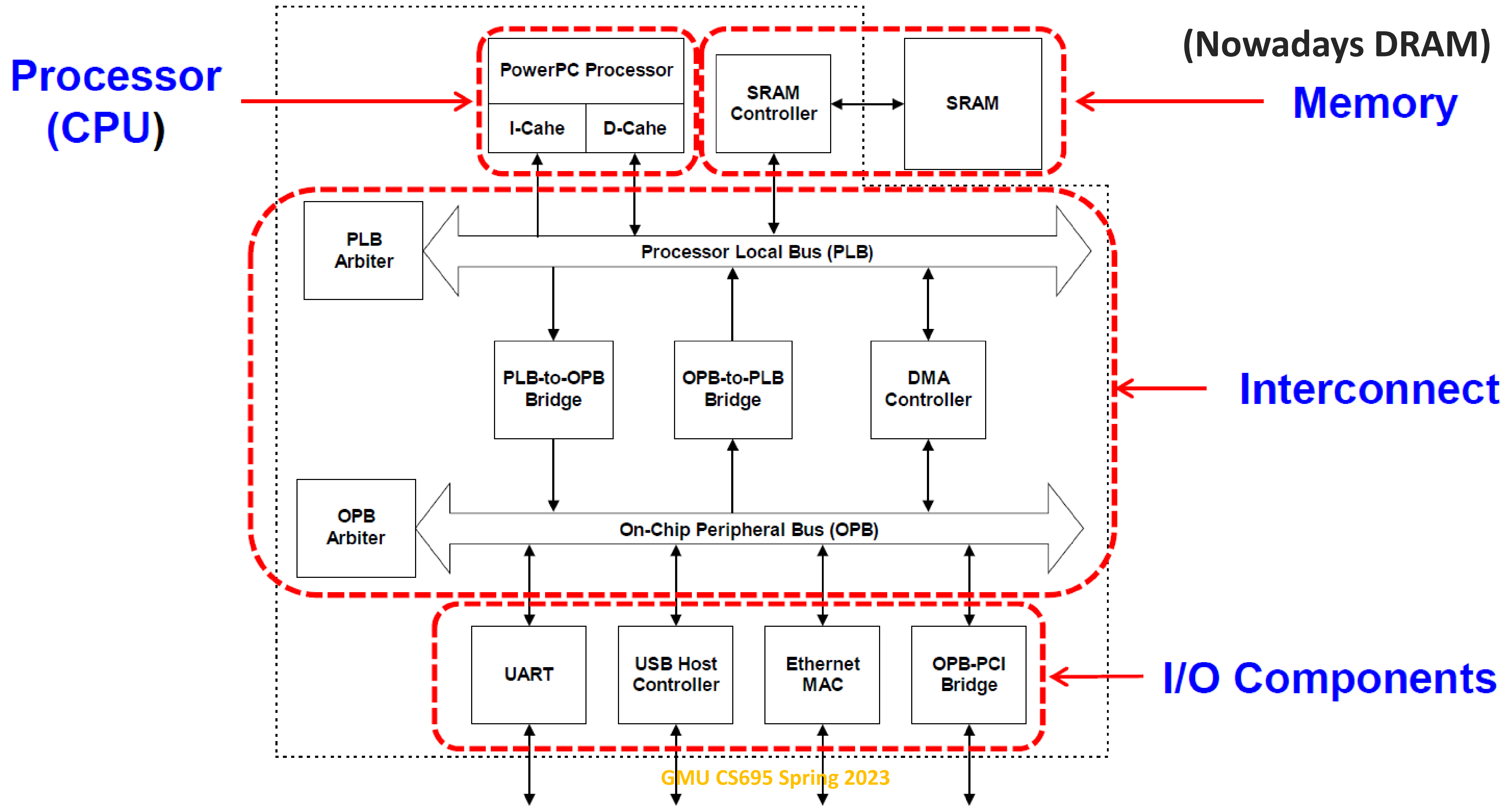
```
swap(int v[], int k)
{int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

```
swap:
      muli $2, $5,4
      add  $2, $4,$2
      lw   $15, 0($2)
      lw   $16, 4($2)
      sw   $16, 0($2)
      sw   $15, 4($2)
      jr   $31
```

```
00000000101000010000000000011000
00000000000110000001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

Hardware's Point of View

Computer Organization:



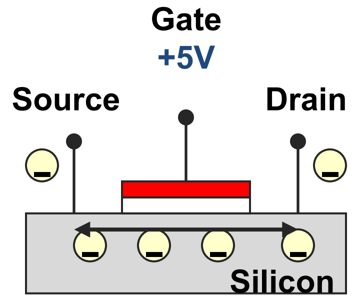
Computer Organization vs. Computer Architecture

– What is the key difference?

Connection vs. interface

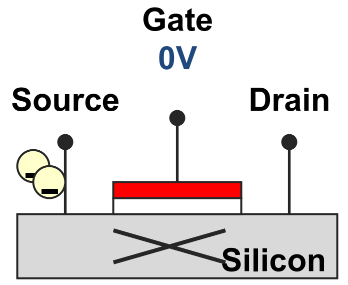
Electronic Components' Point of view

- **Transistors**
  - 3-terminal device
  - Gate input: the control input; its voltage determines whether current can flow
  - Source & Drain: terminals that current flows from/to



Transistor is 'on'

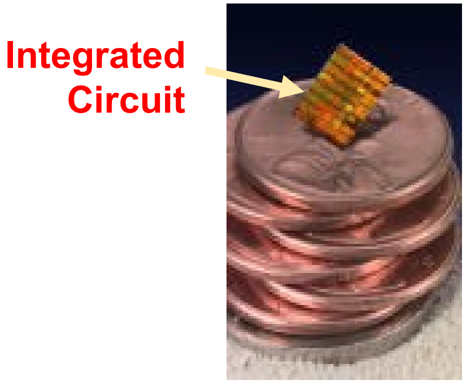
High voltage at gate allows current to flow between source and drain



Transistor is 'off'

Low voltage at gate prevents current from flowing between drain and source

- **Many transistors can be fabricated on one piece of silicon** (i.e. an integrated chip, IC)



Actual silicon wafer is quite small but can contain several billion transistors

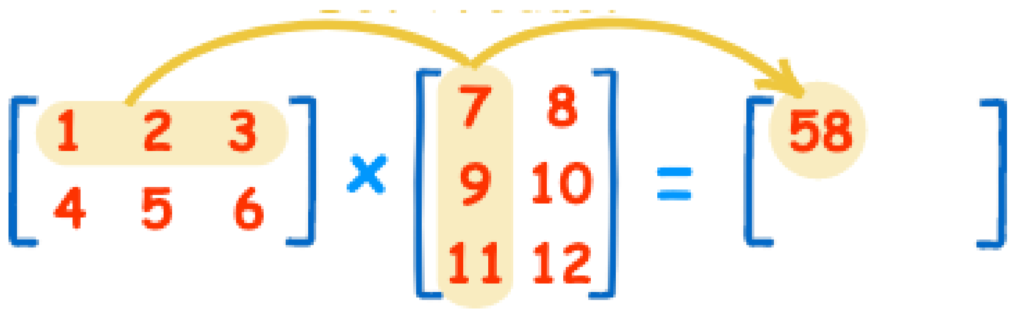
Silicon wafer is then packaged to form the chips we are familiar with

More transistors = faster core?

Software & Hardware Optimizations

Multiplying Two 4096-by-4096 Matrices

```
for i in xrange(4096):
  for j in xrange(4096):
    for k in xrange(4096):
      C[i][j] += A[i][k] * B[k][j]
```



Implementation	Running time (s)	Absolute speedup
Python	25,552.48	1x
Java	2,372.68	11x
C	542.67	47x
Parallel loops	69.80	366x
Parallel divide and conquer	3.80	6,727x
plus vectorization	1.10	23,224x
plus AVX intrinsics	0.41	62,806x

Leiserson+, "There's plenty of room at the Top: What will drive computer performance after Moore's law?", Science, 2020

Computer Architecture

- is the science and art of designing computing platforms (hardware, interface, system SW, and programming model)

- to achieve a set of design goals
    - E.g., highest performance on earth on workloads X, Y, Z
    - E.g., longest battery life at a form factor that fits in your pocket with cost < $$$ CHF
    - E.g., best average performance across all known workloads at the best performance/cost ratio
    - …

    - Designing a supercomputer is different from designing a smartphone → But, many fundamental principles are similar

Where is Architecture?

intel

arm

IBM

NVIDIA

AMD

Where is Architecture?

Argonne NATIONAL LABORATORY

OAK RIDGE National Laboratory

Why Architecture?

Moore’s Law (1965)

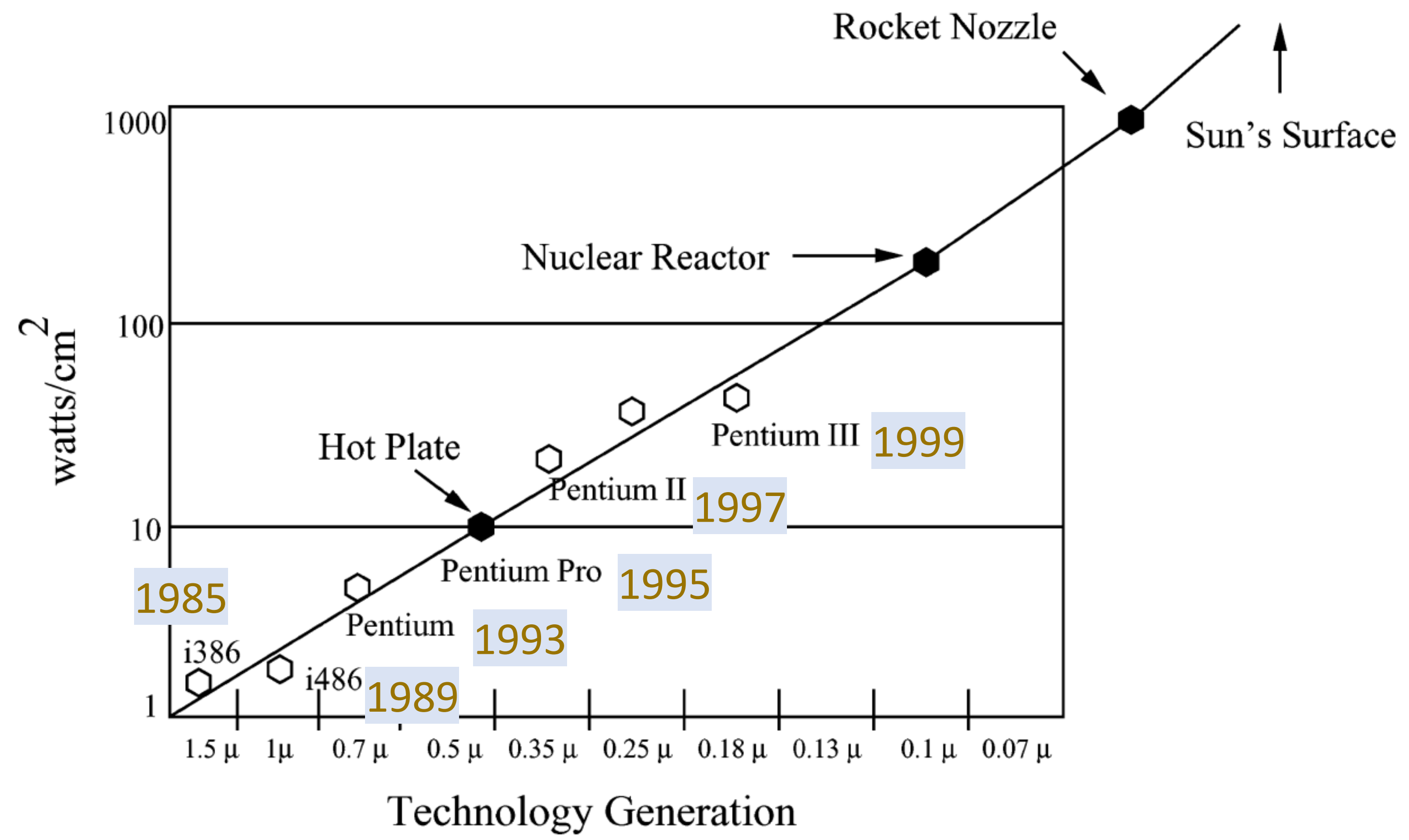
“The number of transistors per square inch on integrated circuits had doubled every year since the integrated circuit was invented and this trend would continue for the foreseeable future.”

-- Gordon E. Moore, “Cramming More Components onto Integrated Circuits,” Electronics, pp. 114–117, April 19, 1965.

In 1975, he recalibrated it to every **two** years.
Later, it is recalibrated again to **18 months,** as is widely known as Moore’s Law.

Exciting time

Moore's law ending: Power wall



Source: V. Venkatachalam and M. Franz. "Power reduction techniques for microprocessor systems", ACM Computing Surveys, 37(3):195–237, 2005. Original figure adapted from Pollack 1999.

Why Study Computer Architecture?

- Enable better systems: make computers faster, cheaper, smaller, more reliable, …
  - By exploiting advances and changes in underlying technology/circuits
- Enable new applications
  - Life-like 3D visualization 20 years ago? Virtual reality?
  - Self-driving cars?
  - Personalized genomics? Personalized medicine?
- Enable better solutions to problems
  - Software innovation is built on trends and changes in computer architecture
    - > 50% performance improvement per year has enabled this innovation
- Understand why computers work the way they do

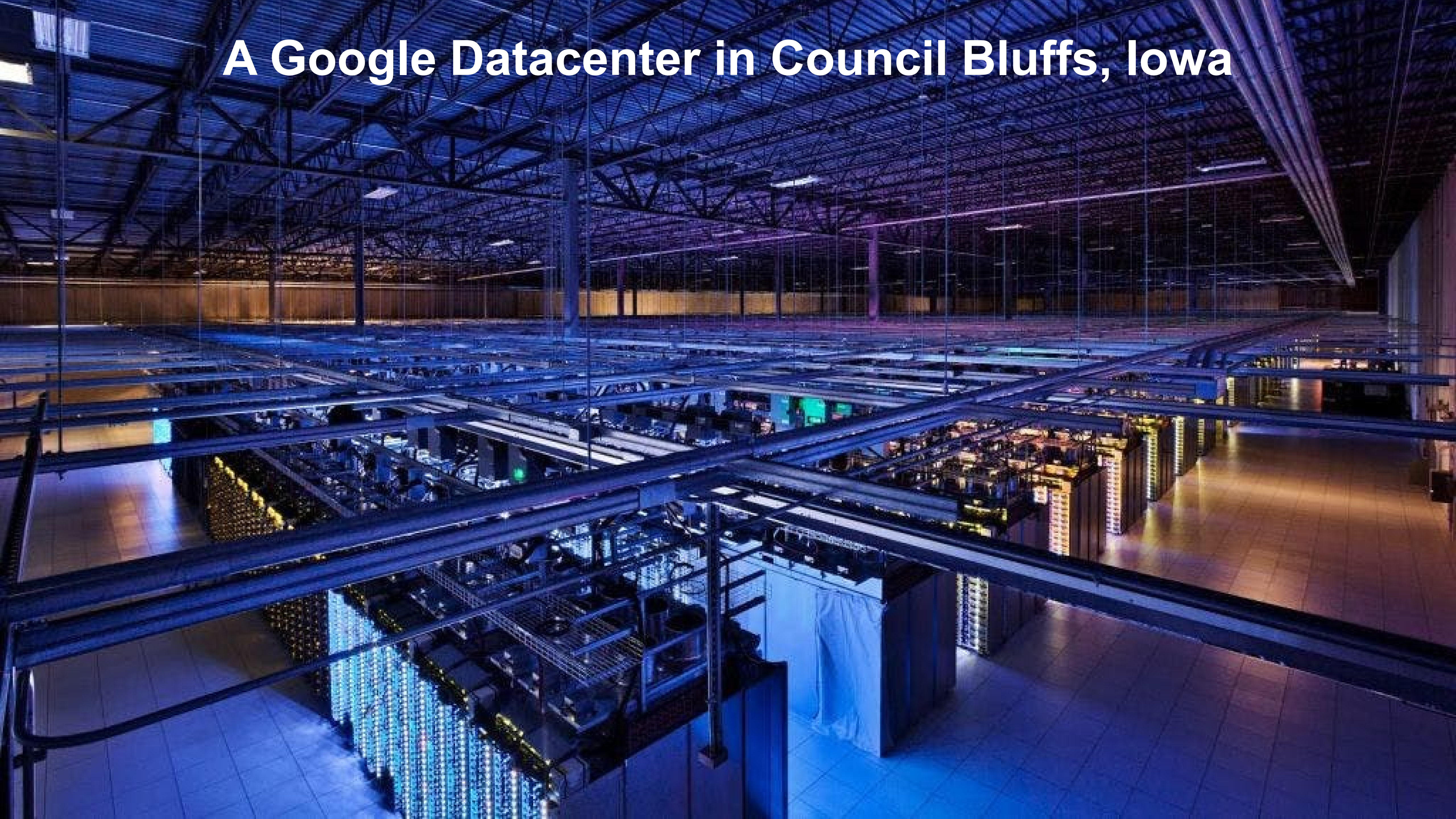
Computer Architecture Today

- Today is a very exciting time to study computer architecture
- Industry is in a large paradigm shift (to novel architectures) – many different potential system designs possible
- Many difficult problems *motivating* and *caused by* the shift
  - Huge hunger for data and new data-intensive applications
  - Power/energy/thermal constraints
  - Complexity of design
  - Difficulties in technology scaling
  - Memory bottleneck
  - Reliability problems
  - Programmability problems
  - Security and privacy issues
- No clear, definitive answers to these problems

Different platforms, different goals.

summit
IBM
EXIT

A Google Datacenter in Council Bluffs, Iowa



Different platforms, different goals.

- Accelerators
- _PU?

How to design and implement Architecture?

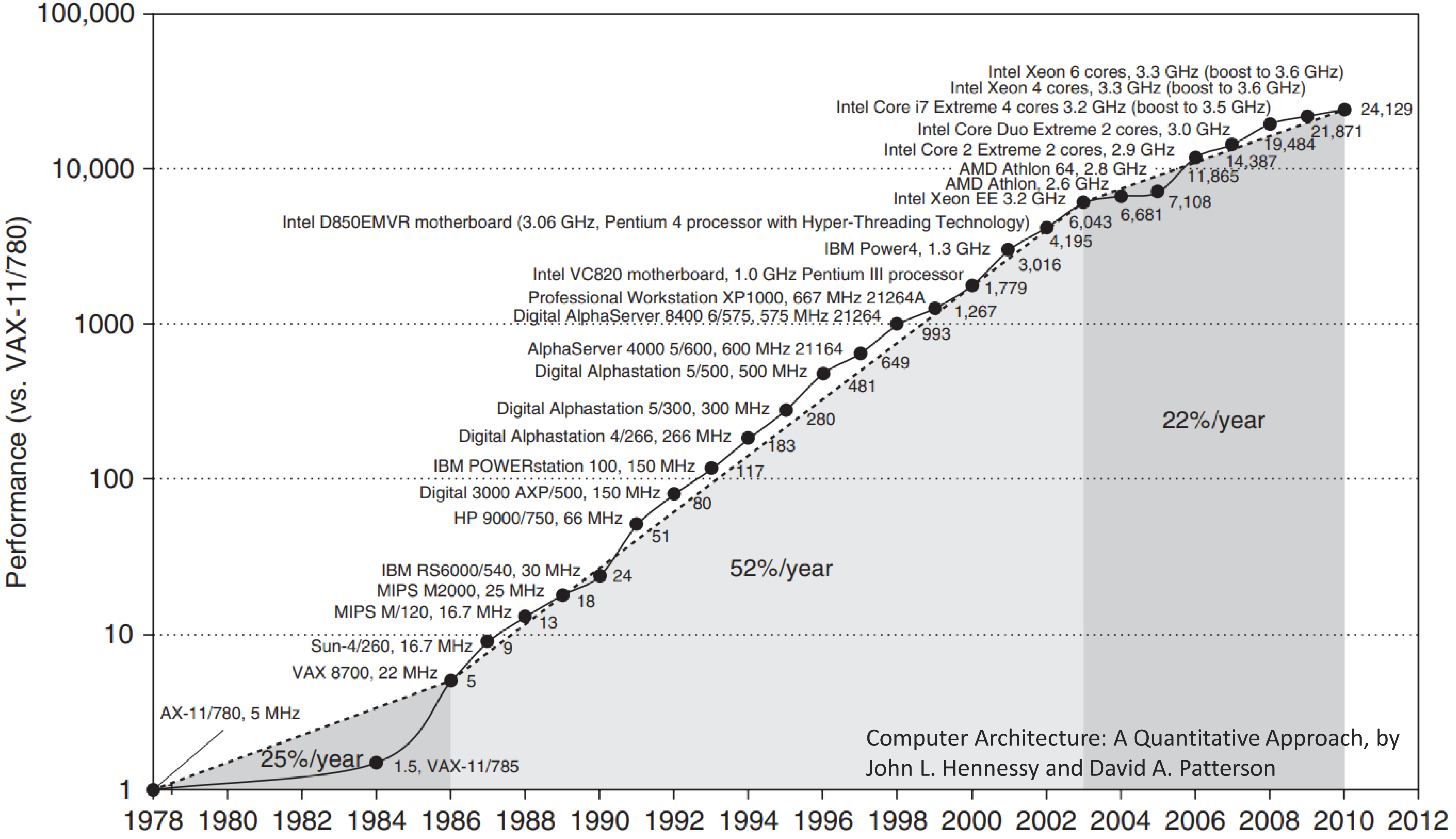
Course Project: Decide the problem

- Bad problems (as a course project for CS695)
  - Build an Intel Xeon CPU from scratch
  - Build an A100 GPU from scratch
  - …
  - Check the size of Cache for a CPU
  - Check the DRAM size of a GPU
  - …
- Possible problems (these still need further clarification):
  - Evaluate a memory system, find out the bottle neck, and improve it
  - Evaluate the reliability of different hardware structures and improve system reliability
  - Which scheduling mechanism works the best? Under what condition?

Example

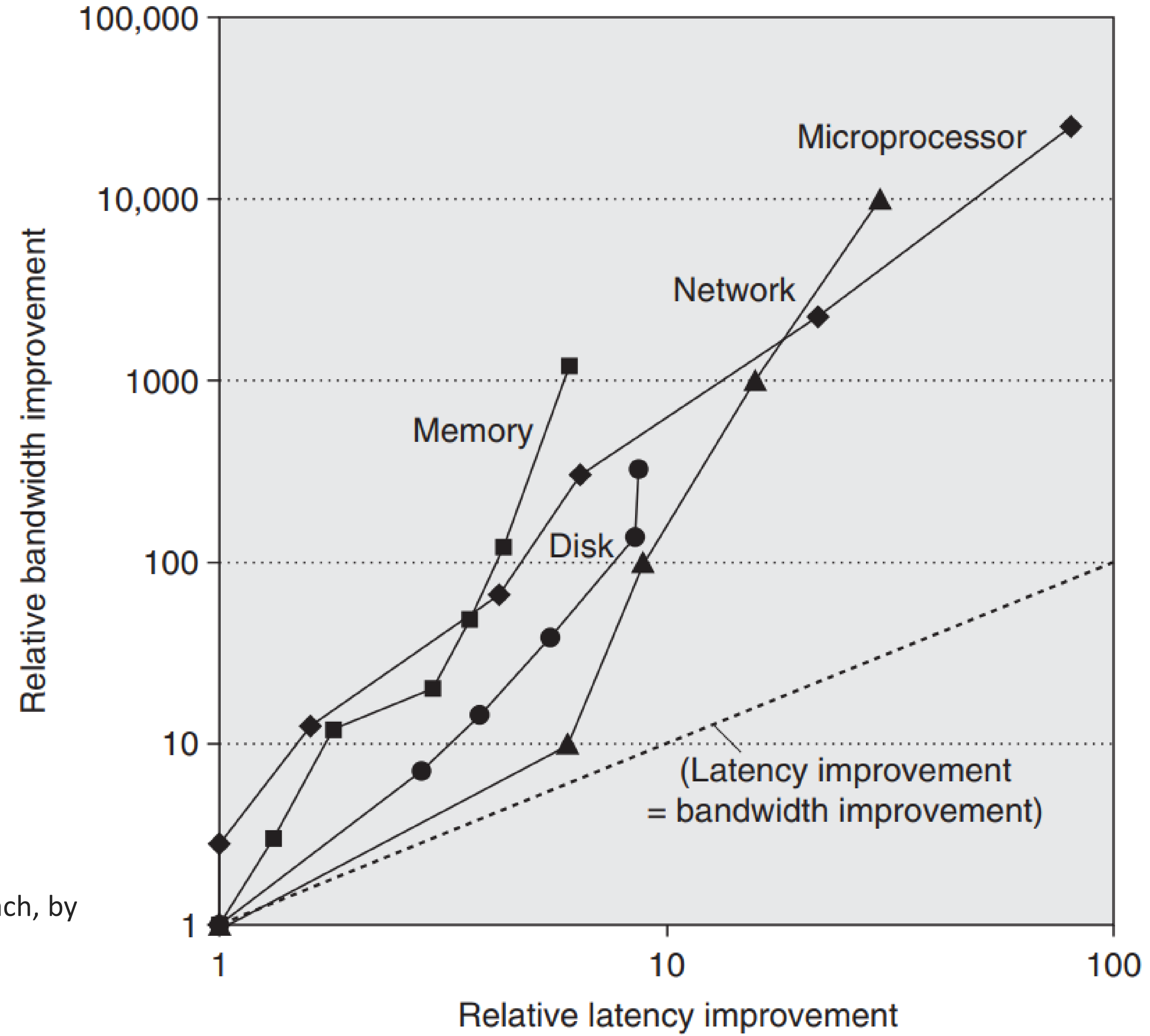
- Evaluate the reliability of different hardware structures and improve system reliability

How to measure performance?

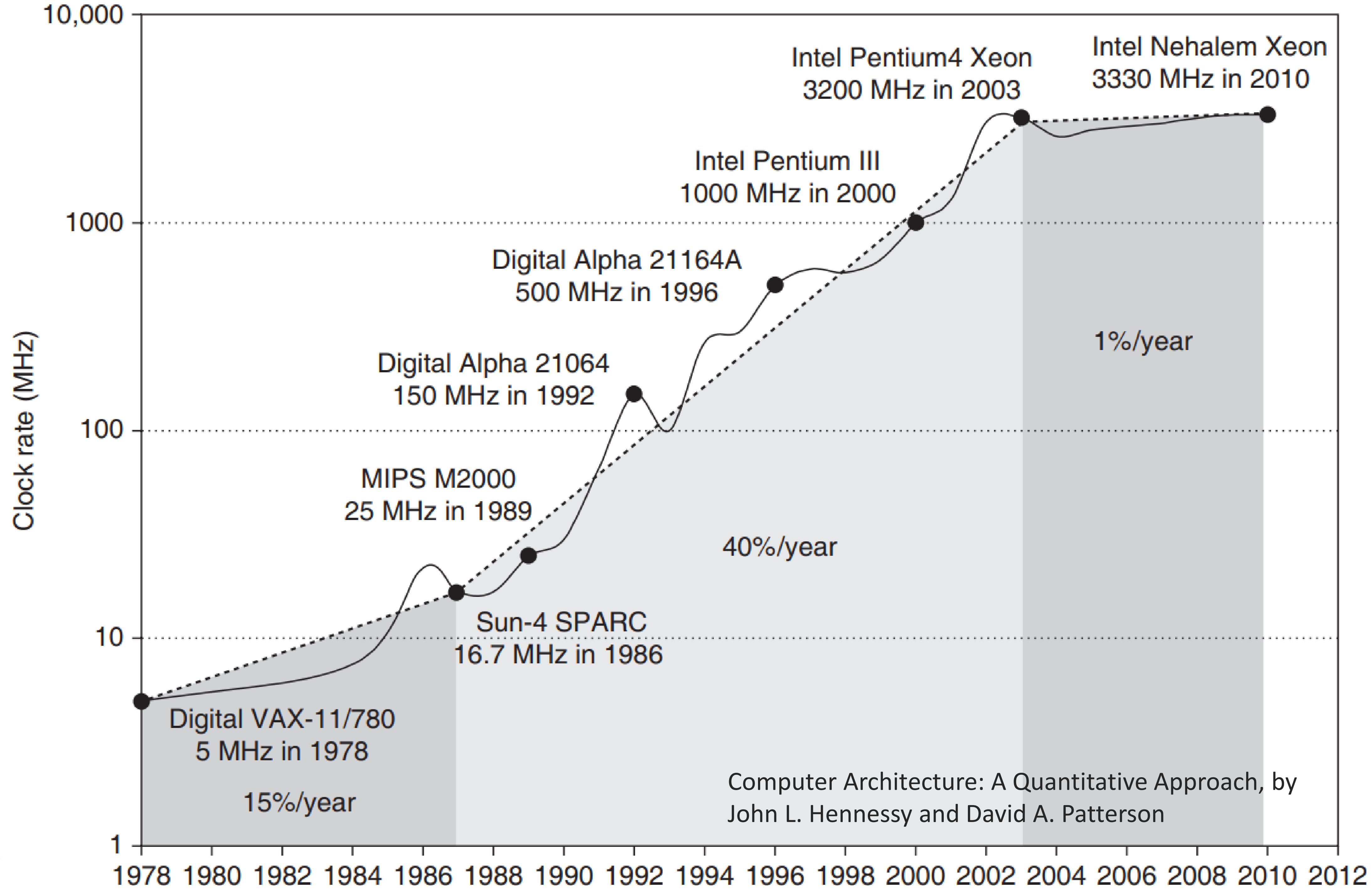


Computer Architecture: A Quantitative Approach, by John L. Hennessy and David A. Patterson

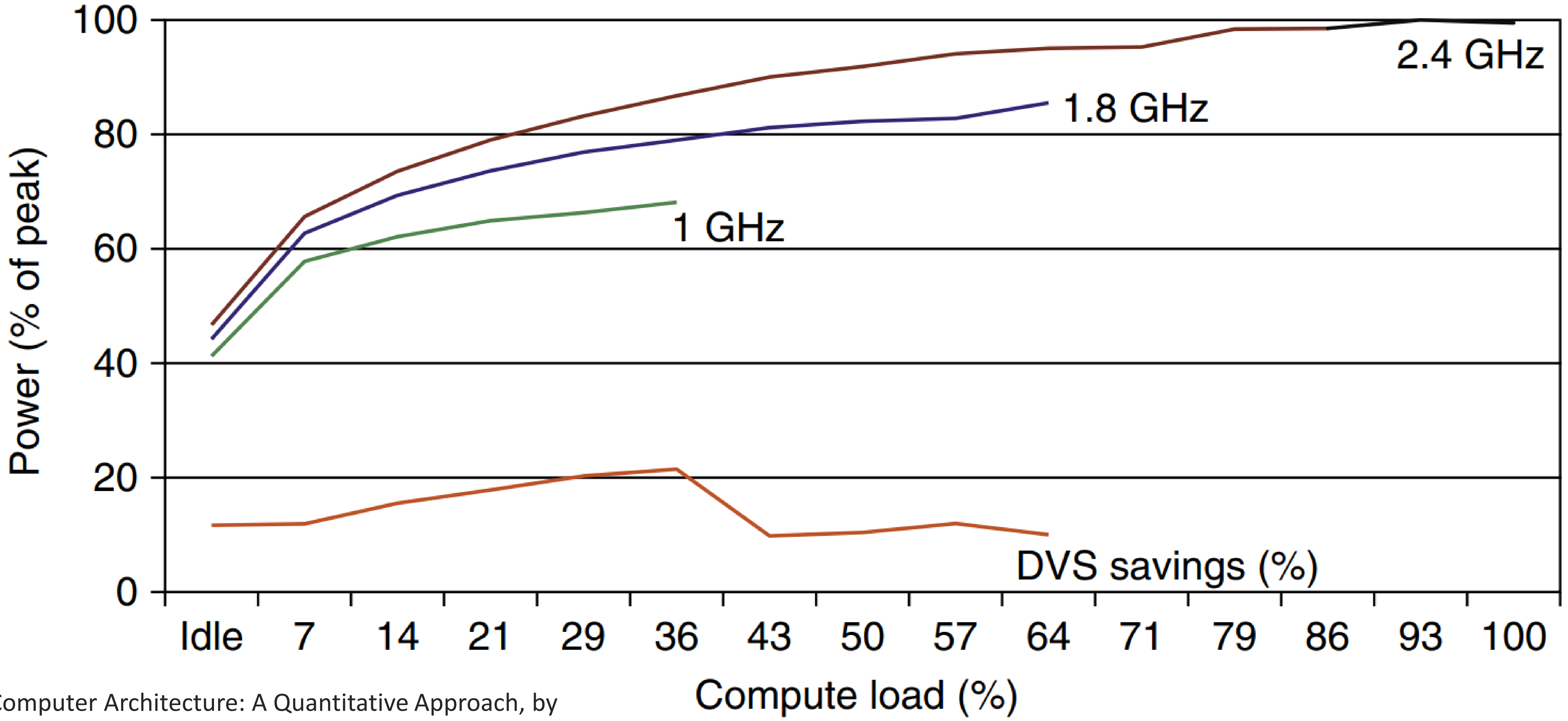
Exciting time



Computer Architecture: A Quantitative Approach, by John L. Hennessy and David A. Patterson



Computer Architecture: A Quantitative Approach, by John L. Hennessy and David A. Patterson



Computer Architecture: A Quantitative Approach, by John L. Hennessy and David A. Patterson

What Determines 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

Relative Performance

- **Define** $\text{Performance} = \dfrac{1}{\text{Execution time}}$

- **"X is n times faster than Y"**

$$\frac{\text{Performance}_\text{X}}{\text{Performance}_\text{Y}} = \frac{\text{Execution time}_\text{Y}}{\text{Execution time}_\text{X}} = n$$

Example: time taken to run a program -- 10s on a computer A, 15s on computer B

Execution Time$_B$ / Execution Time$_A$ = 15s / 10s = 1.5

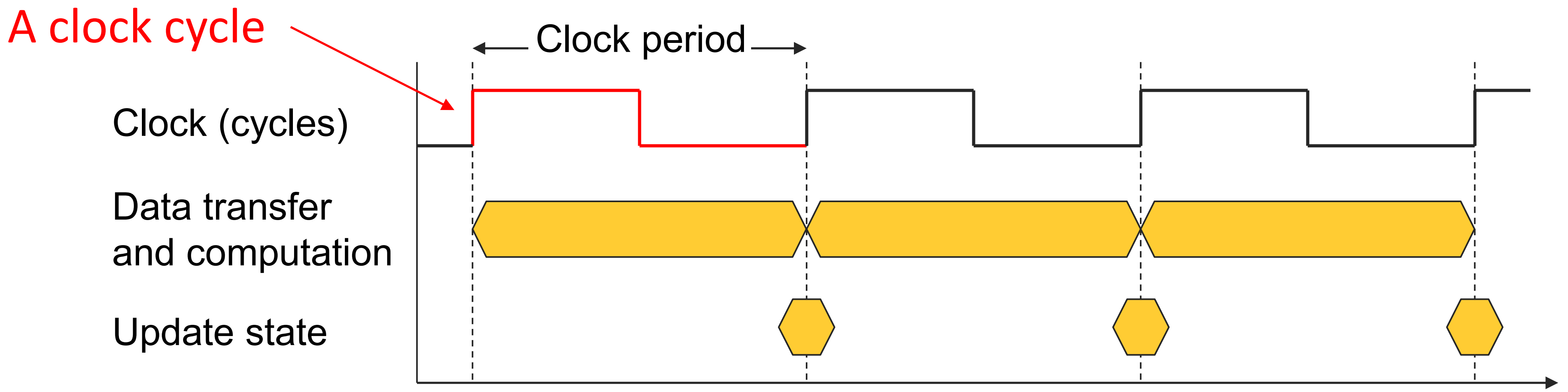
A is 1.5 times faster than B.

Measuring Execution Time

- **Elapsed time**
  - Total response time, including all aspects
    - Processing, I/O, OS overhead, idle time
  - Determines system performance
- **CPU time**
  - Time spent processing a given job on CPU
    - Discounts I/O time, other jobs' shares
  - Estimating CPU time is relatively easy based on the number of lines of code and processing speed of the CPU

CPU Clocking

- Operation of digital hardware governed by a constant-rate clock
- Different system components may have different clock rates



Clock period: duration of a clock cycle

e.g., 250ps = 0.25ns = 250×10^{-12}s

Clock frequency (rate): cycles per second

e.g., 4.0GHz = 4000MHz = 4.0×10^{9}Hz

CPU Time Estimation

- A code line of high-level language program can be translated to **one or multiple** assembly code lines
- CPU processes a piece of machine code for each assembly code line, one by one

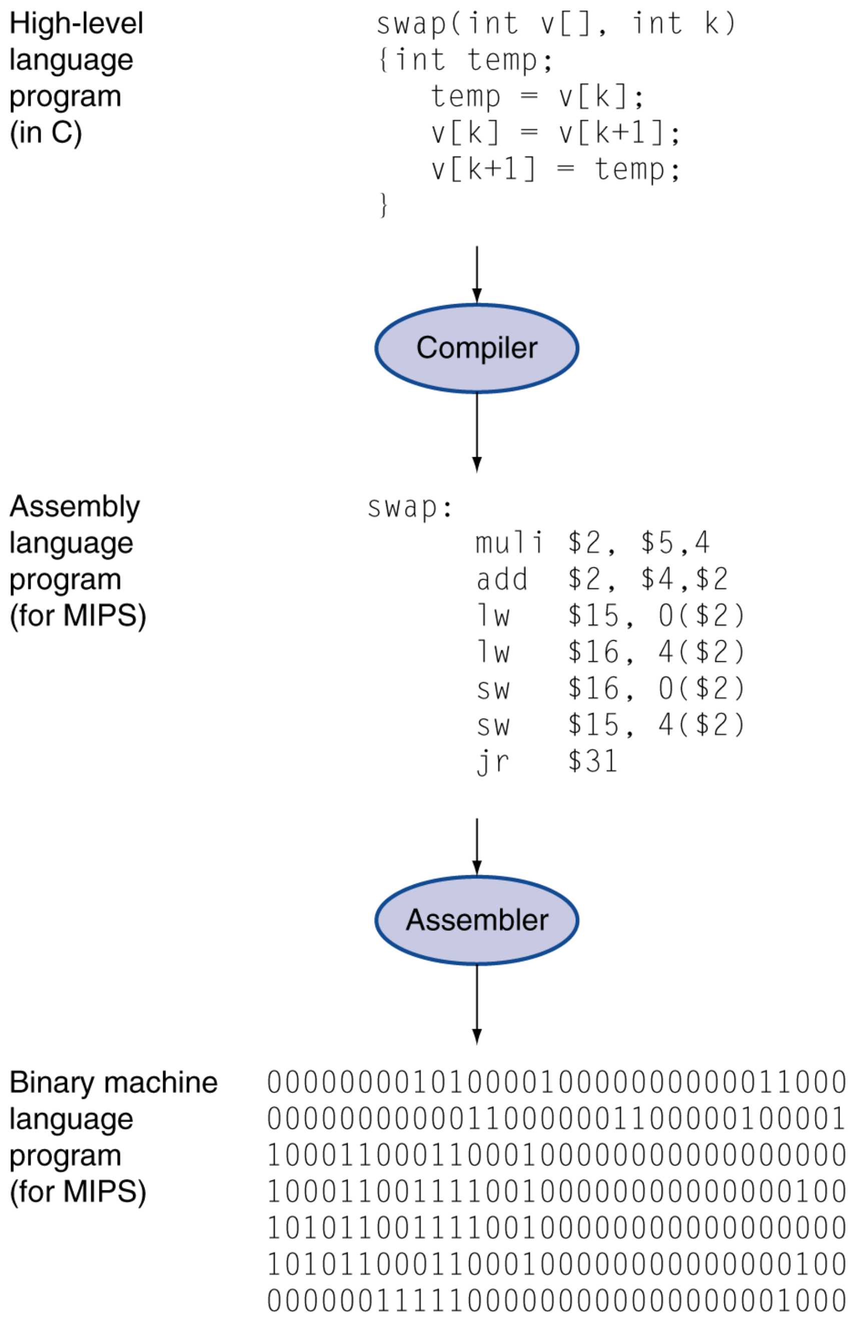
- **Example:** Application A has 1000 assembly lines. CPU has 250 ps clock period.

Clock cycles for Application A

= 1000 cycles

CPU time for Application A

= 1000 cycles x 250 ps = 250×10^{-9}s


High-level language program (in C)
Compiler
Assembly language program (for MIPS)
Assembler
Binary machine language program (for MIPS)

```
swap(int v[], int k)
{int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

```
swap:
      muli $2, $5,4
      add  $2, $4,$2
      lw   $15, 0($2)
      lw   $16, 4($2)
      sw   $16, 0($2)
      sw   $15, 4($2)
      jr   $31
```

```
00000000101000010000000000011000
00000000000110000001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

CPU Time $= \text{Clock Cycles} \times \text{Clock Period} = \frac{\text{Clock Cycles}}{\text{Clock Frequency}}$

- Computer A: 2GHz clock, 10s CPU time to run an Application
- Designing Computer B
  - Aim for 6s CPU time for the same Application
  - Can do faster clock, but causes 1.2 × clock cycles
- How fast must Computer B clock be?

CPU Time_B = 6 seconds
= Clock Cycles_B x Clock Period_B
= 1.2 x Clock Cycles_A x Clock Period_B
= 1.2 x 20 x 10^9 x Clock Period_B
Therefore, Clock Period_B = 6 second / (24 x 10^9)
Clock Frequency_B = 4 x 10^9 = 4GHz

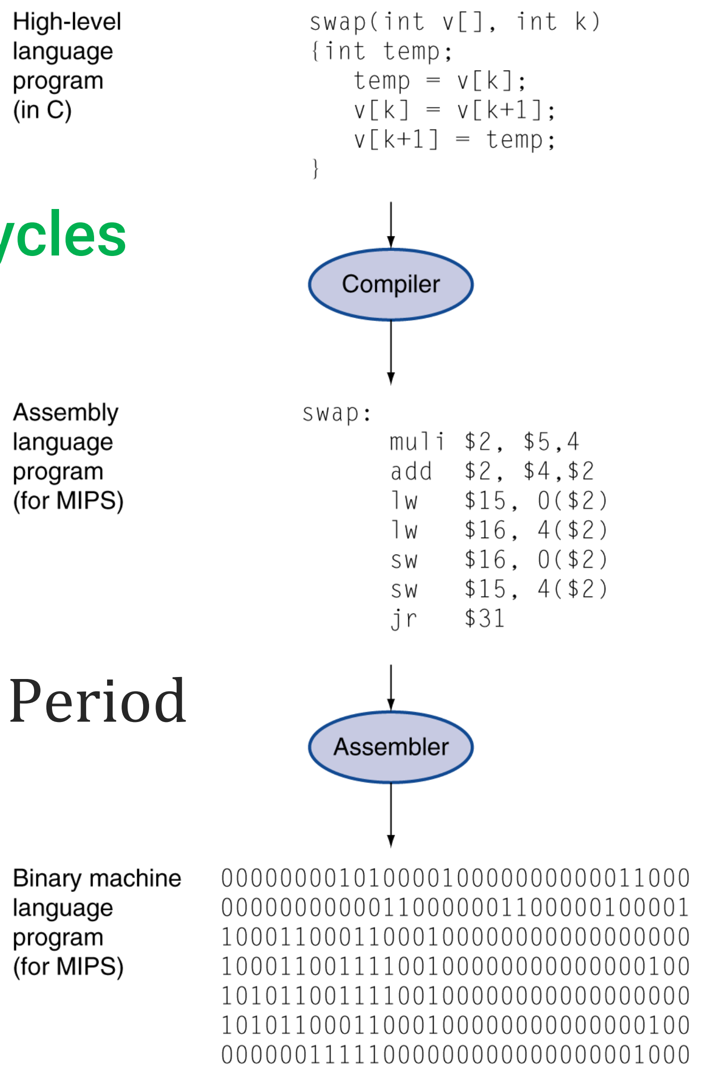
CPU Time_A = Clock Cycles_A / Clock Frequency_A
10 seconds = Clock Cycles_A / 2GHz
Clock Cycles_A = 10 seconds x 2GHz = 20 x 10^9 cycles

Instruction Count and CPI

- An assembly line is also called one **Instruction**
- In some CPUs, an instruction may take **multiple clock cycles**
- CPU Time equation can be rewritten

$$\begin{aligned}\text{CPU Time} &= \text{Clock Cycles} \times \text{Clock Period}\\ &= \text{Instruction Count} \times \text{Cycles Per Instruction} \times \text{Clock Period}\end{aligned}$$

- **Cycles Per Instruction** is also called **CPI**


High-level language program (in C)
Compiler
Assembly language program (for MIPS)
Assembler
Binary machine language program (for MIPS)

```
swap(int v[], int k)
{int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

```
swap:
      muli $2, $5,4
      add  $2, $4,$2
      lw   $15, 0($2)
      lw   $16, 4($2)
      sw   $16, 0($2)
      sw   $15, 4($2)
      jr   $31
```

```
00000000101000010000000000011000
00000000000110000001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

CPI Example

- A program takes 33 billion instructions to run
- CPU processes instructions at 2 cycles per instruction
- Clock speed is 3GHz

What is estimated CPU Time for this program?

CPU Time = Instruction Count x CPI x Clock Period

$= 33 \times 10^9 \times 2 \times 1/(3 \times 10^9)$

= 22 seconds

CPI and Average CPI

- In some CPUs, different types of instructions may take different number of cycles

- **CPU Time equation can be rewritten**

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

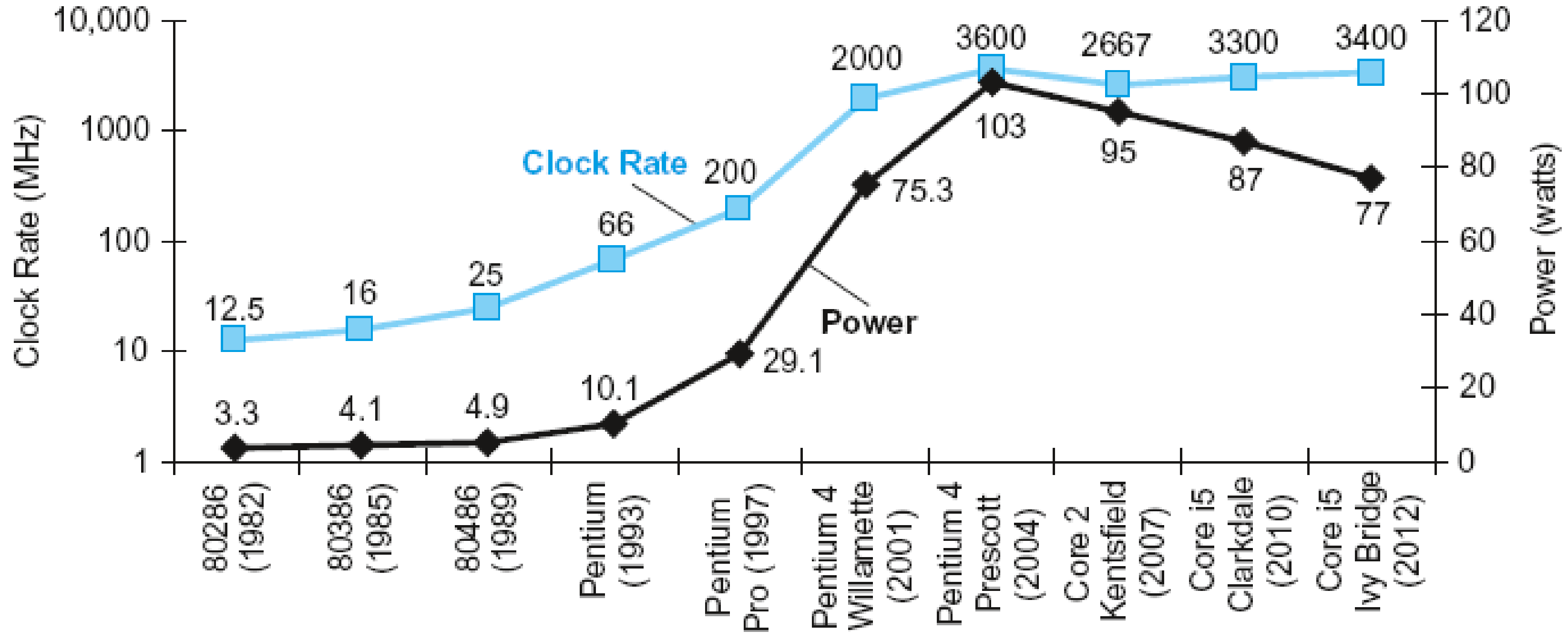
$$= \boxed{\sum (\text{IC}_\text{i} \times \text{CPI}_\text{i})} \times \text{ClockPeriod}$$

(IC = instruction count)

Sum of (IC of each type instruction x CPI of the type)

$$\text{Average CPI} = \frac{\text{Total Clock Cycles}}{\text{Total Instruction Count}} = \frac{\sum (\text{IC}_\text{i} \times \text{CPI}_\text{i})}{\text{Total Instruction Count}}$$

Power and Performance

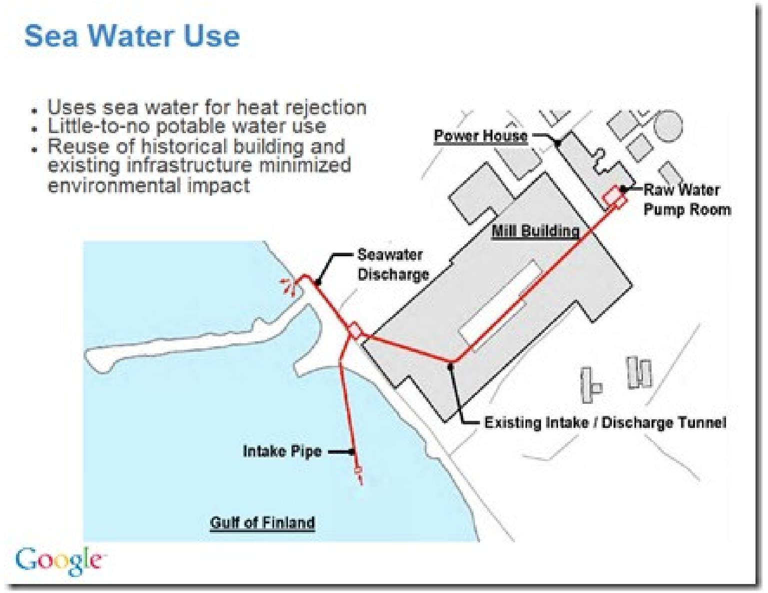


$$Power = Capacitive\ load\ \times Voltage^2 \times Frequency$$

Issue with Power Scaling

- **Increasing core frequency may burn your processor!**
  - Almost all energy consumption turns into heat

- **Cooling is costly**


Sea Water Use

- Uses sea water for heat rejection
- Little-to-no potable water use
- Reuse of historical building and existing infrastructure minimized environmental impact

- **Design more efficient system instead**
  - Multicore, etc..

Benchmarks

- SPEC, Rodinia, Polybench, …
- Reproducibility
- Standard

- SPECRatio: normalize the execution time to a reference computer
  - $SPECRatio = \frac{Execution\ time_{reference}}{Execution\ time_{A}}$

- Geometric mean vs arithmetic mean
  - $\text{Geometric}\ mean = \sqrt[n]{\prod_{i=1}^{n} sample_i}$
  - $Arithmetic\ mean = \frac{1}{n}\sum_{i=1}^{n} sample_i$

What do we really improve?

Amdahl's Law

- The performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used.
- Defines the Speedup that can be gained by using a particular feature

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

Same:

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

- Speedup depends on:
  - The fraction of the computation time in the original computer that can be converted to take advantage of the enhancement
  - The improvement gained by the enhanced execution mode

Amdahl's Law

- The performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used.
- Speedup depends on:
    - The fraction of the computation time in the original computer that can be converted to take advantage of the enhancement
    - The improvement gained by the enhanced execution mode

$$\text{Execution time}_{\text{new}} = \text{Execution time}_{\text{old}} \times \left(\left(1 - \text{Fraction}_{\text{enhanced}}\right) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}} \right)$$

$$\text{Speedup}_{\text{overall}} = \frac{Execution\ time_{old}}{Execution\ time_{new}} = \frac{1}{\left(1 - \text{Fraction}_{\text{enhanced}}\right) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

Peak performance vs observed performance