CSCI-UA.0480-001

Special Topics: Multicore Programming

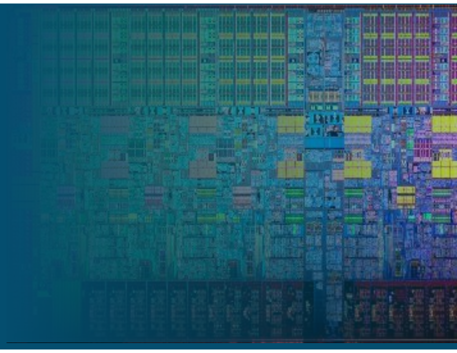
Lecture 1

The Multicore Revolution

Christopher Mitchell, Ph.D.

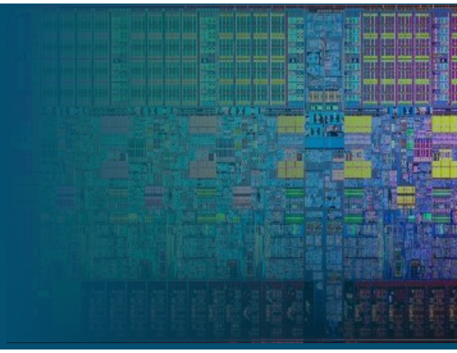
cmitchell@cs.nyu.edu || http://z80.me

Lecture 1 Outline

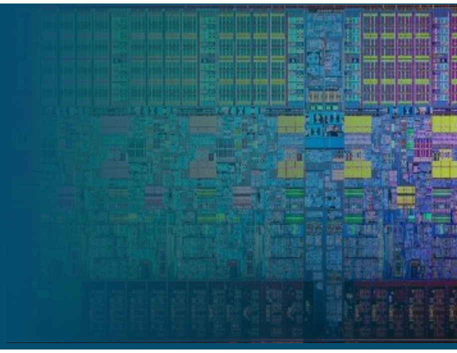


- Course Logistics and Syllabus
- History of the Multicore Processor
- “Lab 0” Assignment

Who Am I?



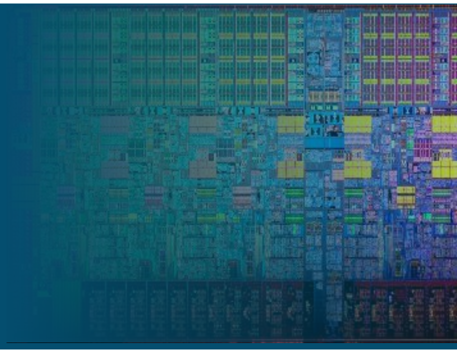
- Christopher Mitchell, Ph.D
  - http://z80.me
- Research Interests
  - Distributed Storage and Computation Systems
  - Large-Scale Automated GIS
- Office hours: Tuesdays, 4:30pm – 5:30pm
  - Subject to consensus
  - Other times by appointment, if needed
- Office: 60 Fifth Avenue, Room TBD



You’ll Learn…

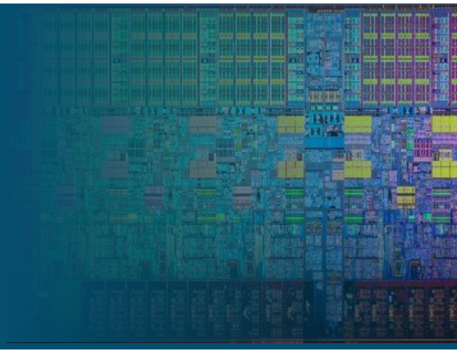
- What are multicore processors, and why do we have them?
- What are the challenges in exploiting them?
- How do we design programs for multiple cores?
- What primitives are available to simplify multicore-aware programs, and how do they work?
- How do you benchmark and debug these programs?
- How does current and future hardware simplify creating these programs?

Past curricula: Thanks to Profs. Zahran, Lerner, Herlihy, et alia for help, material, and inspiration, and to students in previous classes for helping to hone the material.



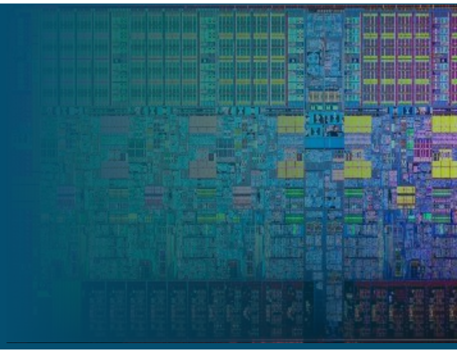
Course Goals

- Hardware and Software
  - Don’t be afraid of hardware: understand it
  - Understand the hardware/software interaction
  - Enjoy the challenge of making the best use of hardware to the benefit of software
- Understand the Theory
- Understand the Software
- Projects



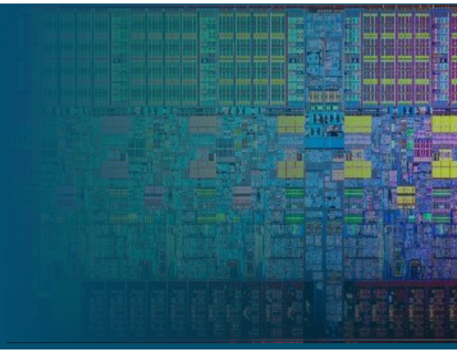
Course Goals

- Hardware and Software
- Understand the Theory
  - Parallelism and concurrency
  - Parallel programming models
- Understand the Software
- Projects



Course Goals

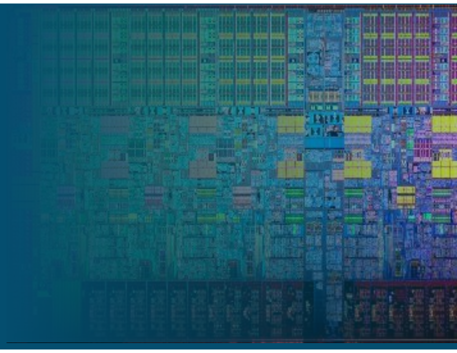
- Hardware and Software
- Understand the Theory
- Understand the Software
  - Low-level primitives
  - Data structures
  - Full programs
- Projects



Course Goals

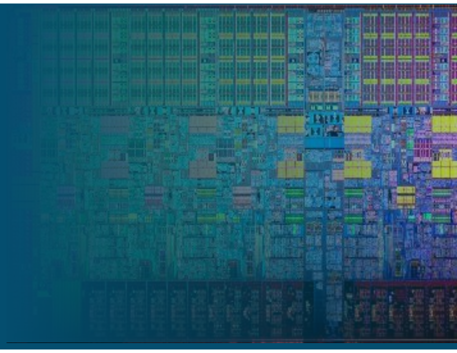
- Hardware and Software
- Understand the Theory
- Understand the Software
- Projects
  - Get more comfortable with C/C++ programming projects.
  - Build intuition about course concepts by implementing them.

Course Resources: Readings



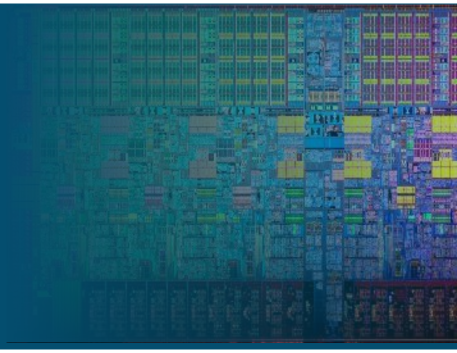
- Textbook: none
  - Readings posted on course website
  - *The Art of Multicore Programming* by Herlihy & Shavit is recommended as a good reference, but **not** required
  - *Computer Architecture* by Patterson & Hennessey is also recommended as a good reference, but **not** required
- Readings
  - Expected to read and understand before each class
  - Stuck? Ask questions on the mailing list.
  - If you're still stuck, ask again in class.

Course Resources: Other



- Course website: cs.nyu.edu/courses/summer19/CSCI-UA.0480-001/
  - You’re responsible for assignments, labs, readings
- Course mailing list: via NYU Classes
  - Let me know if you’re not on it

Grading

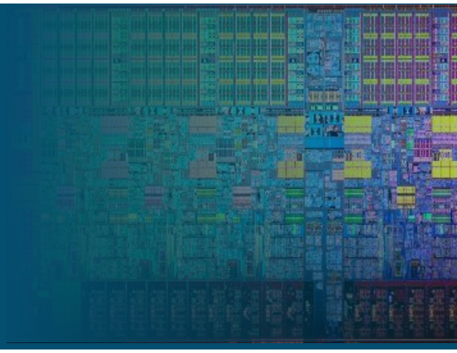


• Homework assignments	15%
• Programming assignments	30%
• Project	20%
• Final Exam (+ Midterm Quiz)	35%
• Participation	??

(Subject to change, to your advantage.)

Assignment Policies

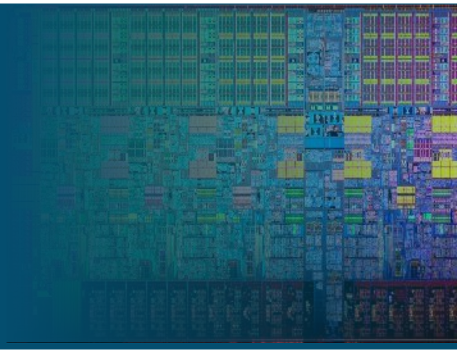
- You must work alone on all assignments
  - Post all questions on the mailing list,
  - You are encouraged to answer others' questions, but refrain from explicitly giving away solutions.
  - See course webpage for further academic integrity notes.
- Hand-ins
  - Labs due at 11:59pm on the due date
  - Homework assignments due at the end of the lecture of the due date
  - (-1) for each day of late submission (up to 3 days), then zero in the corresponding assignment



Programming Assignments

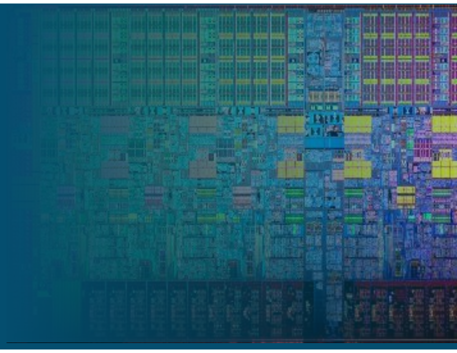
- Assignments are in C++ (not C)
- Expected to have experience with:
  - Classes and structures
  - The `std` data structures
  - Basic template usage

Programming Assignments



- Toolchain
  - Be or become comfortable g++, gdb, and git.
  - Submit assignments by committing to a private repo on your own GitHub account
    - GitHub private repos are free for everyone now
- Environment
  - SSH to CIMS Linux computers
  - *Or* your own computer
  - *Or* a VM

Course Outline



Introduction, Theory, Hardware

1. [Today] The Multicore Revolution
2. A C++ Crash Course
3. Parallelism, Concurrency, and Performance
4. Parallel Programming Models
5. Hardware & Multicore Programming

Parallel Programming Building Blocks

6. Intro to Parallel Programming Primitives
7. Mutual Exclusion
8. Semaphors and Condition Variables
9. Reader-Writer Locks
10. Barriers and Thread Pools

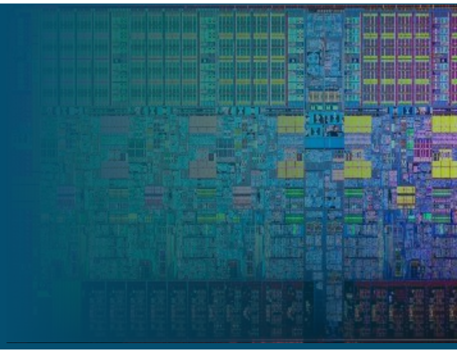
Data Structures

11. Synchronized Structures Part 1
12. Synchronized Structures Part 2
13. Synchronized Structures Part 3

Software Design & Advanced Topics

14. Multicore Correctness
15. Multicore Performance Evaluation
16. Heterogenous Multicore
17. Transactional Memory

Feedback



- Strongly encourage feedback
  - Topics (+/-)
  - Difficulty/speed
  - Readings
- Sooner is better.
- Anonymous feedback welcome.

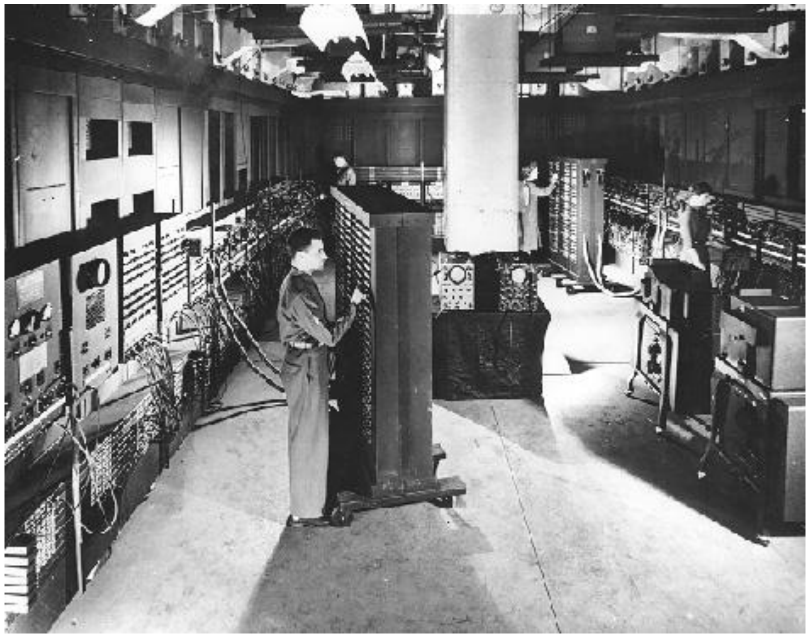
Lecture 1 Outline

- Course Logistics and Syllabus
- History of the Multicore Processor
  - Evolution of the Modern CPU
  - Computer Architecture Interlude
  - The Performance Wall
  - Multicore Processors
- "Lab 0" Assignment

First: A Bit of History

The First Computers: ENIAC

1940
1950
1960
1970
1980
1990
2000
2010
2020

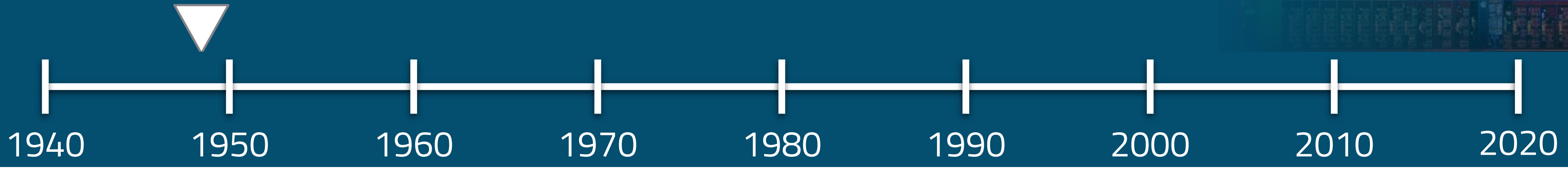


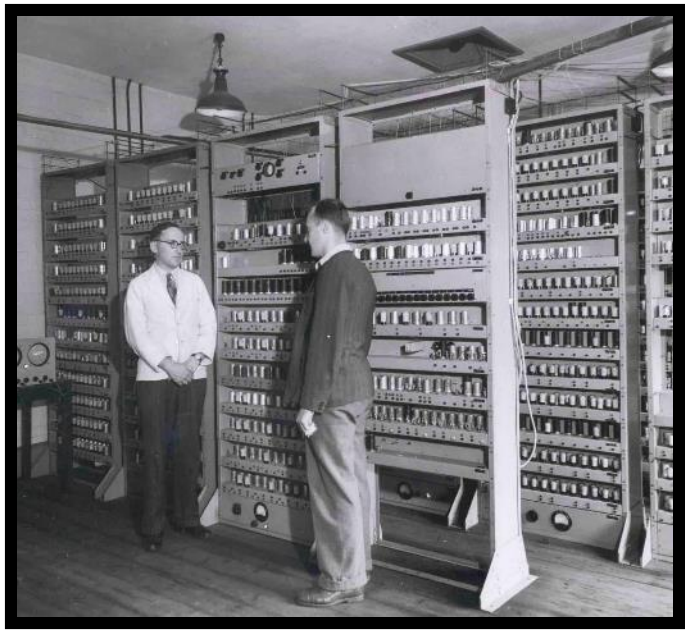
ENIAC

- Eckert and Mauchly

- 1st working electronic computer (1946)
- 18,000 Vacuum tubes
- 1,800 instructions/sec
- 3,000 ft^3

The First Computers: EDSAC 1



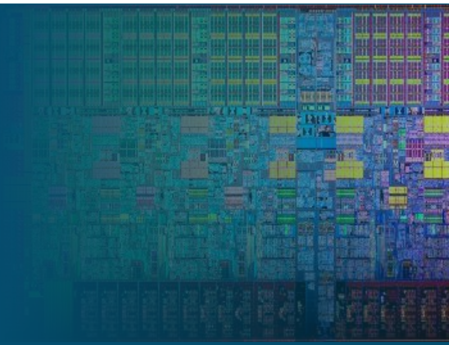

EDSAC 1 (1949)

- Maurice Wilkes

- First computer using stored programs
- 650 instructions/sec
- 1,400 ft^3

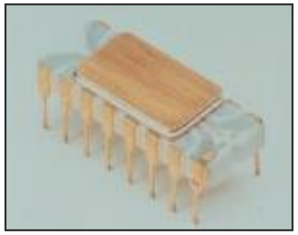
Source: http://www.cl.cam.ac.uk/UoCCL/misc/EDSAC99/

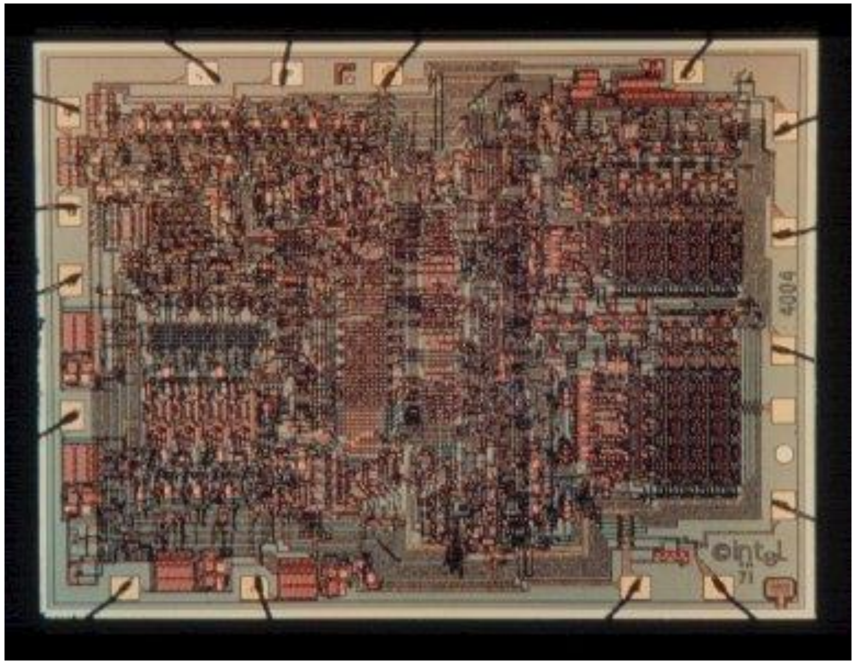
The Microprocessor Age: The Intel 4004



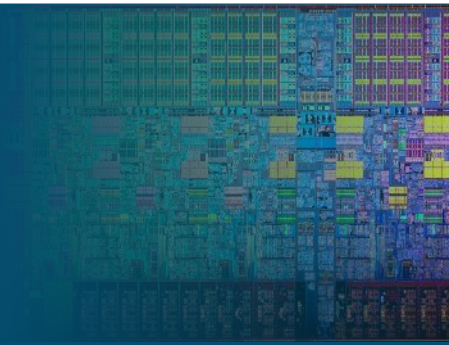
1940
1950
1960
1970
1980
1990
2000
2010
2020

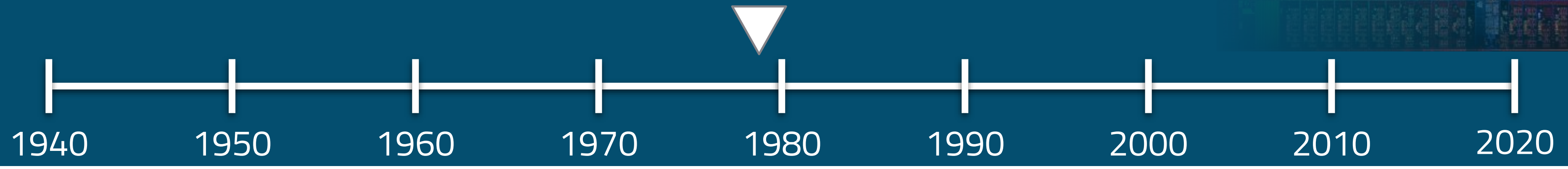
- Introduced in 1970
  - First microprocessor
- 2,250 transistors
- 12 mm^2
- 108 KHz

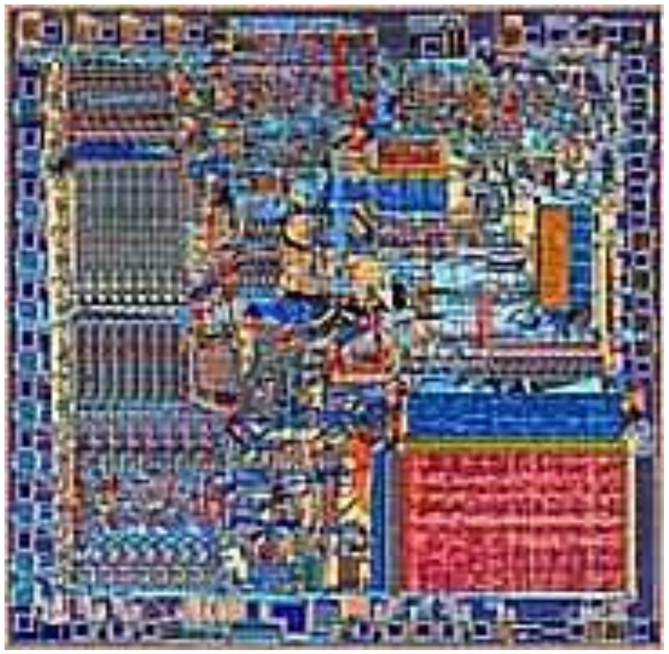




The Microprocessor Age: The Intel 8086

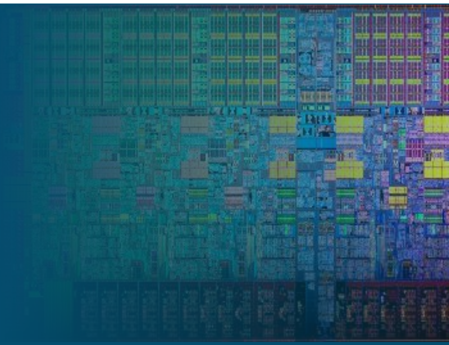


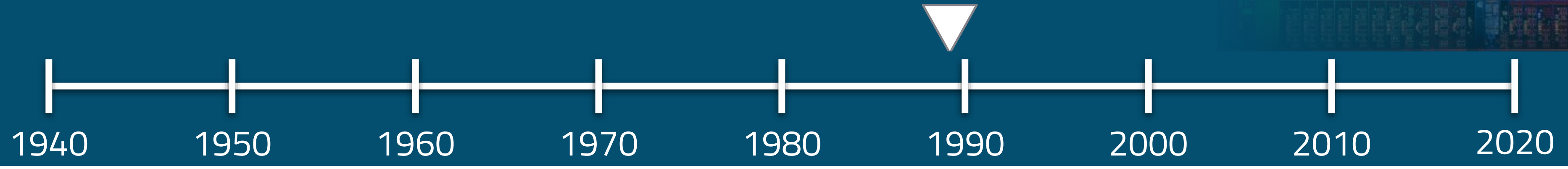


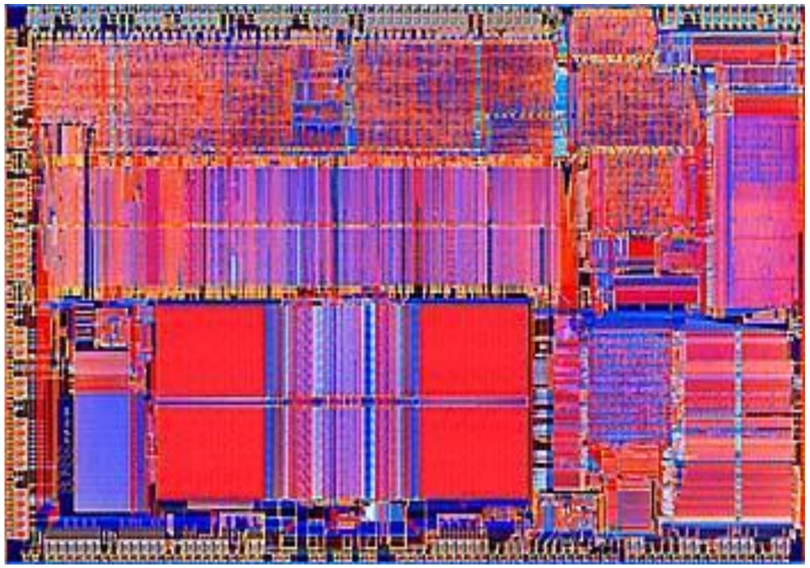


- Introduced in 1979
  - Direct ancestor of the x86 PC
- 29,000 transistors
- 33 mm^2
- 5 MHz

Intel 80486

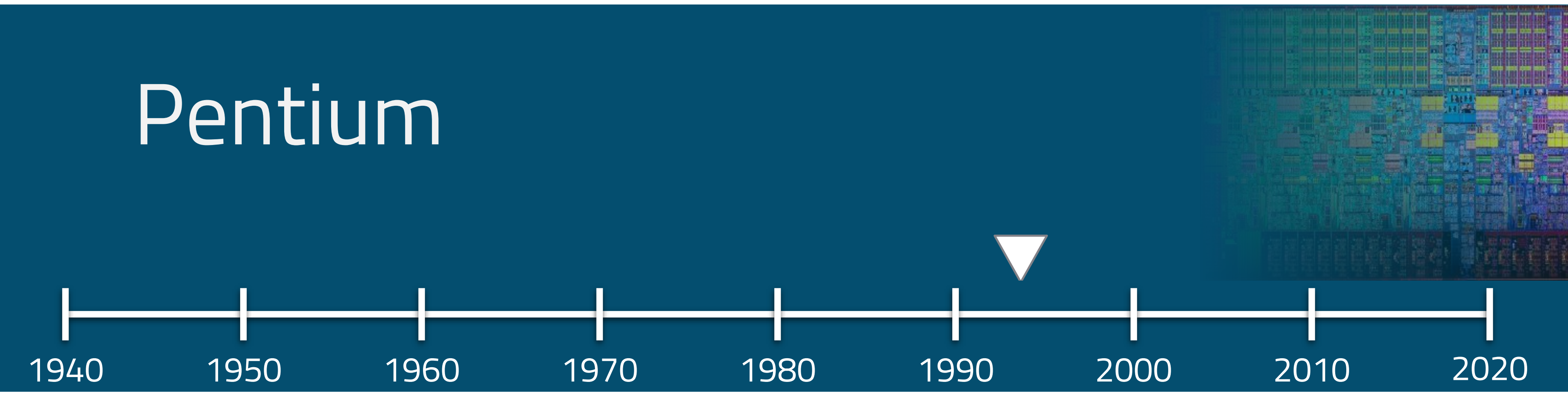


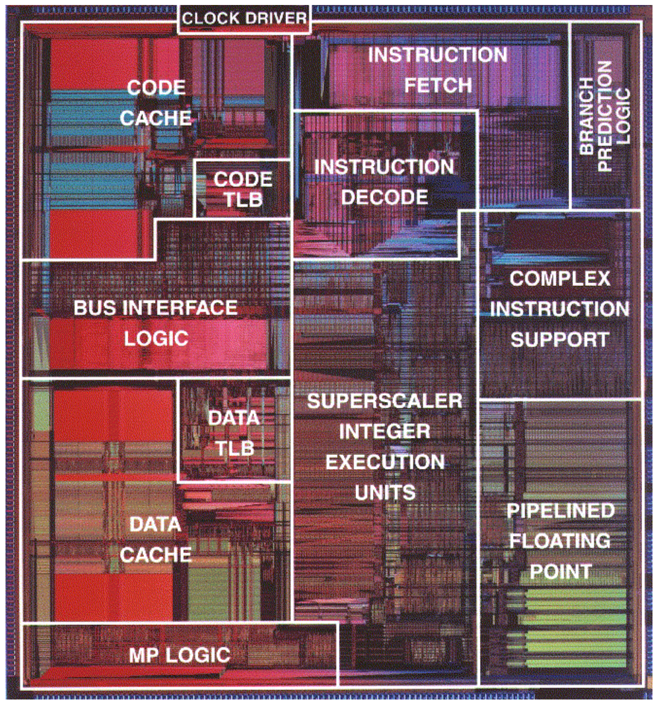




- Introduced in 1989
  - 1st pipelined implementation of x86
- 1,200,000 transistors
- 81 mm^2
- 25 MHz

Pentium

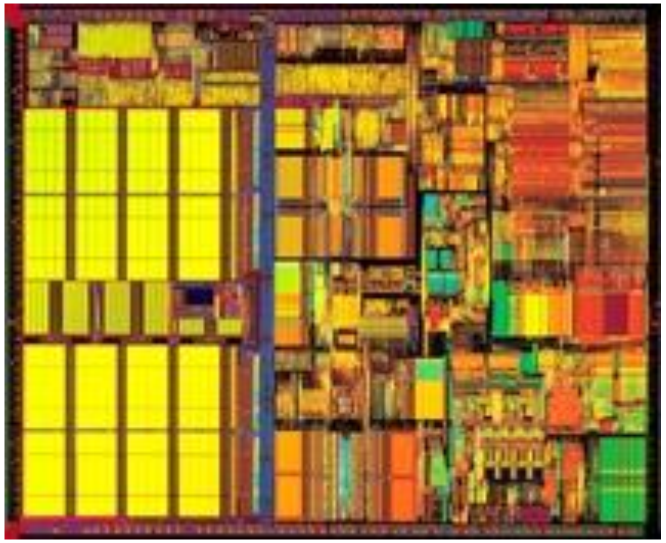




- Introduced in 1993
  - 1st superscalar implementation of IA32
- 3,100,000 transistors
- 296 mm^2
- 60 MHz

Pentium III

1940
1950
1960
1970
1980
1990
2000
2010
2020

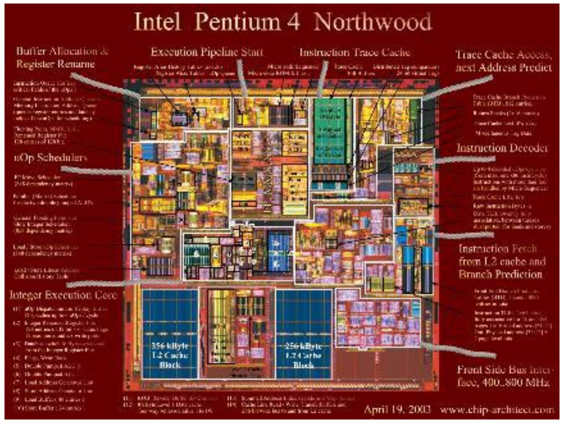


- Introduced in 1999
- 9,500,000 transistors
- 125 mm^2
- 450 MHz

Source: http://www.intel.com/intel/museum/25anniv/hof/hof_main.htm

Pentium 4

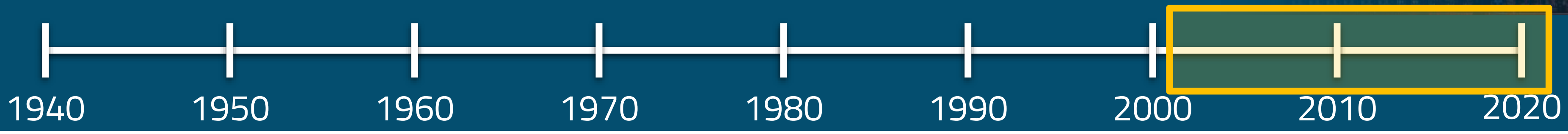
1940
1950
1960
1970
1980
1990
2000
2010
2020

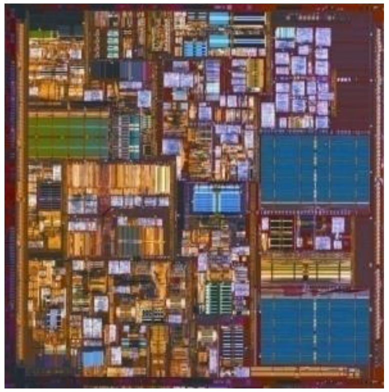

Intel Pentium 4 Northwood
Buffer Allocation & Register Rename
Execution Pipeline Start
Instruction Trace Cache
Trace Cache Access, next Address Predict
uOp Schedulers
Instruction Decoder
Instruction Fetch from L2 cache and Branch Prediction
Integer Execution Core
256 kByte L2 Cache Block
256 kByte L2 Cache Block
Front Side Bus Interface, 400..800 MHz
April 19, 2003
www.chip-architect.com

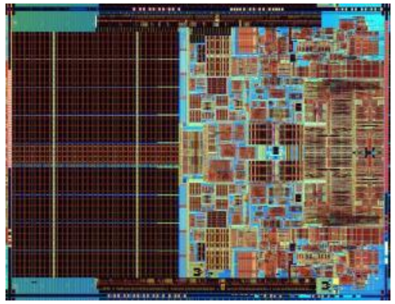
- Introduced in 2000
- 55,000,000 transistors
- 146 mm^2
- 3 GHz

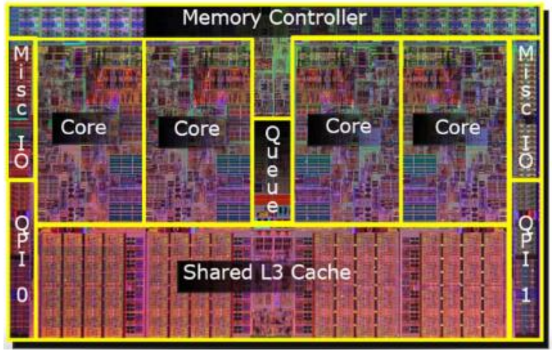
Source: http://www.chip-architect.com

Modern CPUs

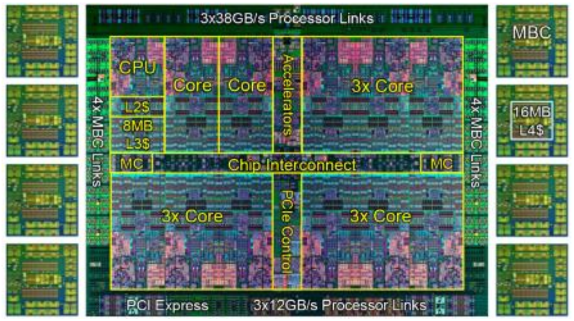



Pentium 4

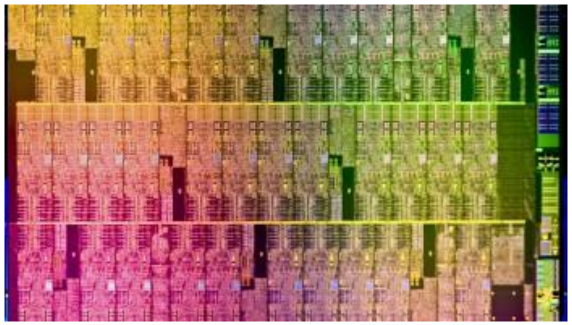

Core 2 Duo (Merom)

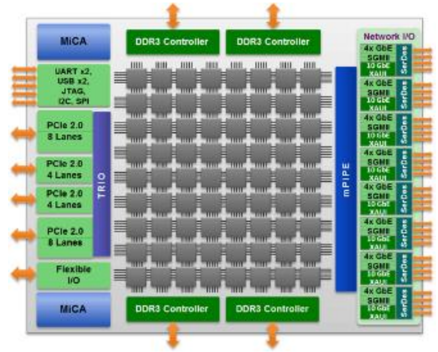


Intel Core i7 (Nehalem)



IBM Power 8

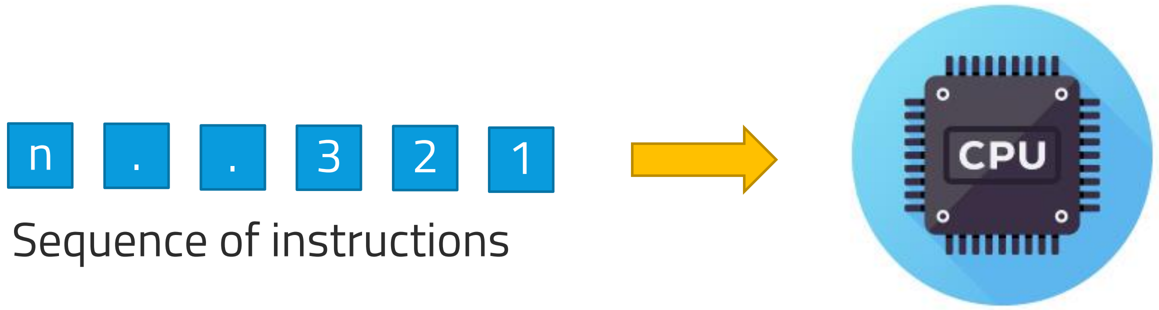

Intel Xeon Phi (50 cores)



Tilera (72 cores)

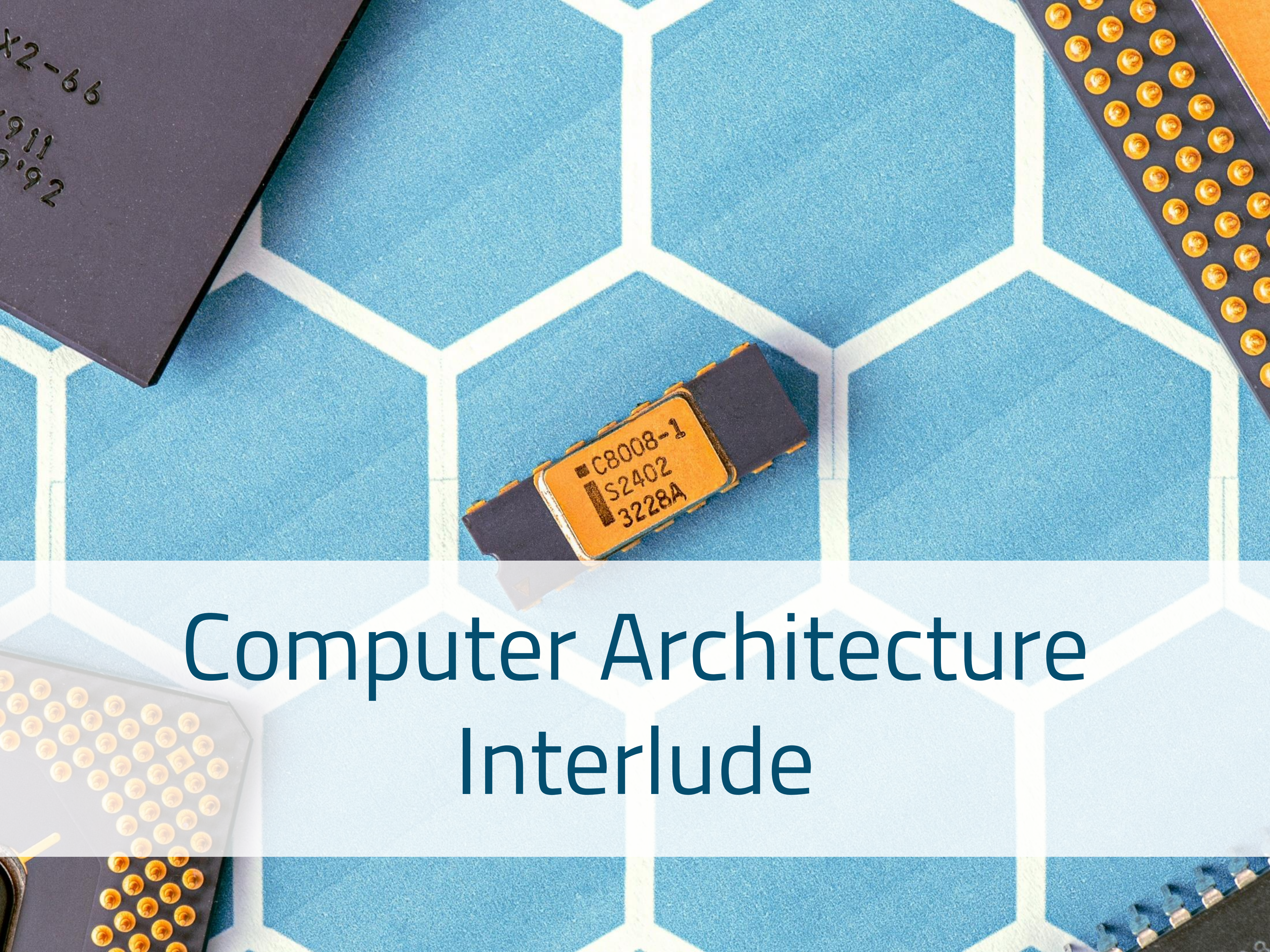
Architecture Advances

- How did we get to multicore processors?
  - The quest for performance
- What is a program?



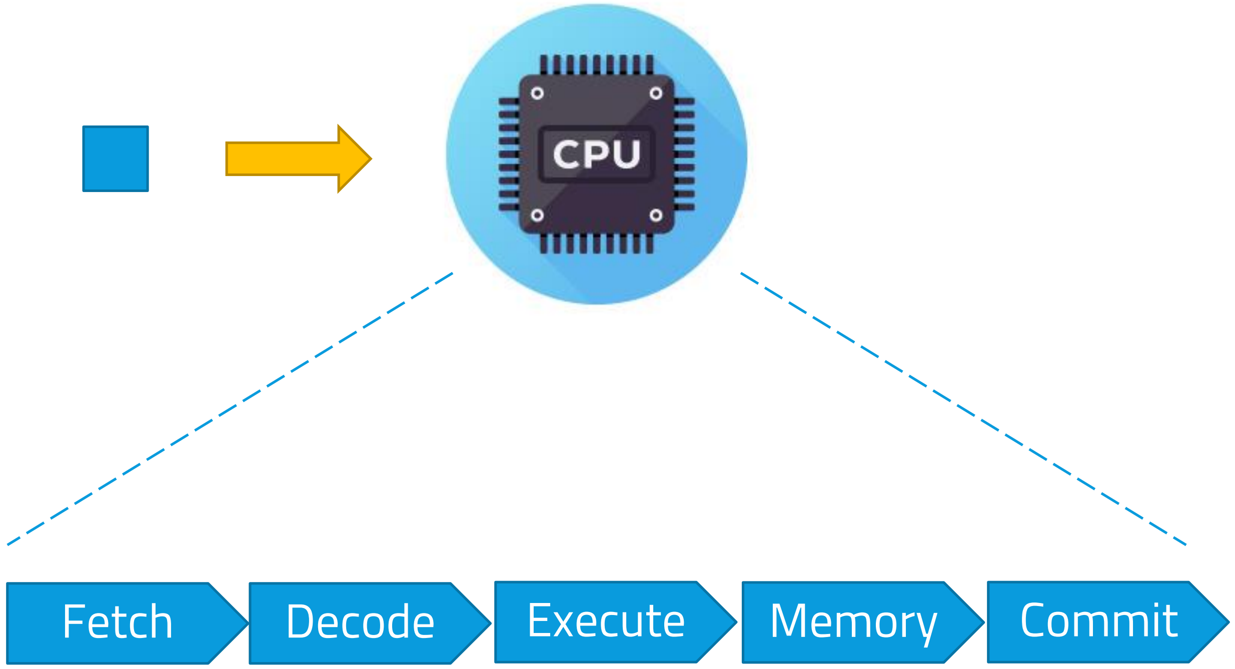
Sequence of instructions

- How can we make this faster?

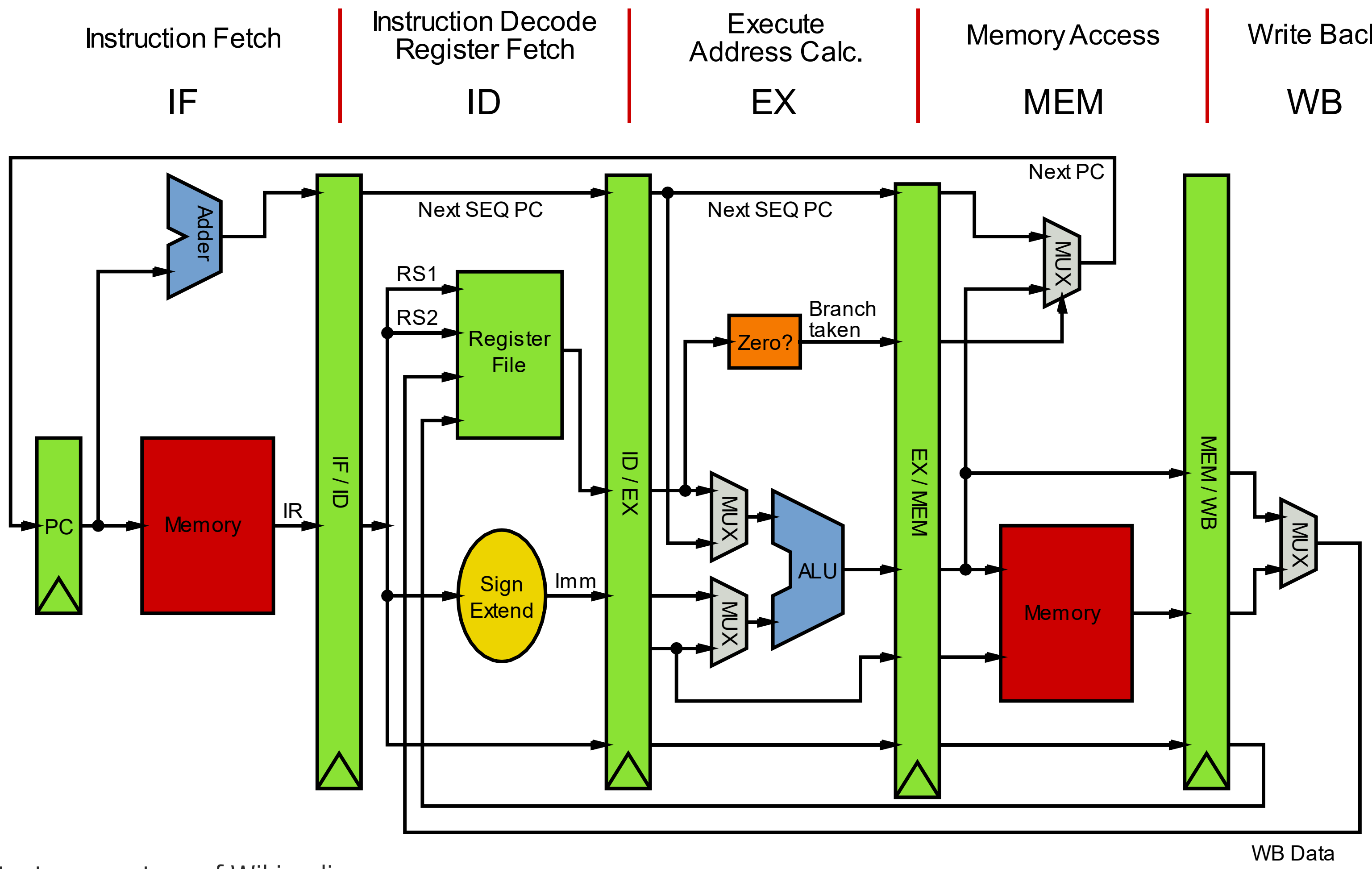


Computer Architecture Interlude

CPU Instructions Are Divisible



Architecture Example: MIPS

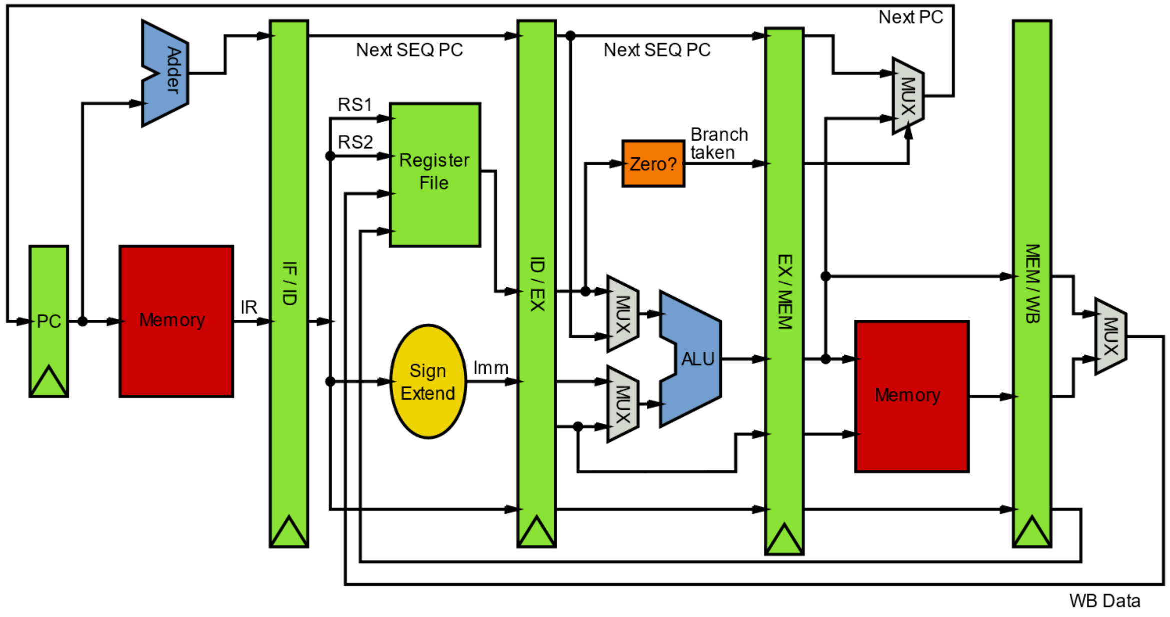


MIPS architecture courtesy of Wikipedia

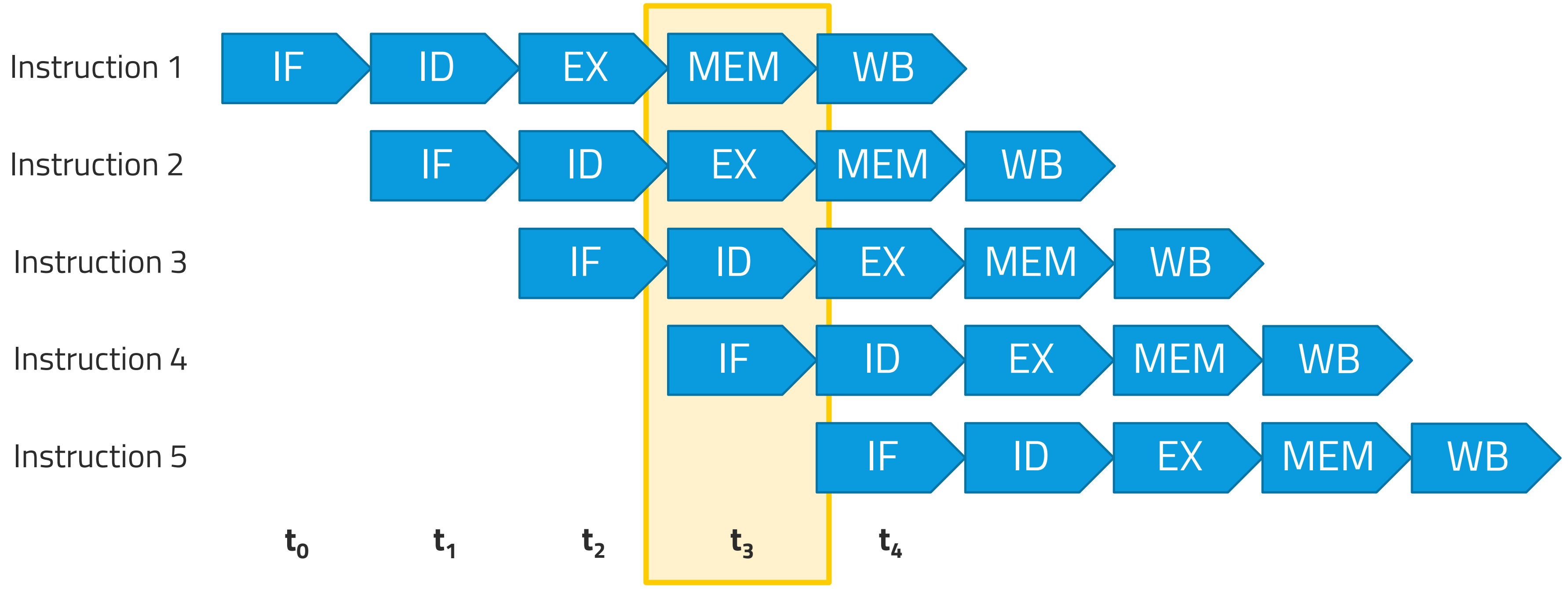
CPU Instructions Are Divisible

Fetch → Decode → Execute → Memory → Commit

	Fetch	Decode	Execute	Memory	Commit
Load	Get opcode	Parse opcode	Compute load source address	Access memory	Write register
Add	Get opcode	Parse opcode	Compute sum	(None)	Write register
Conditional Branch	Get opcode	Parse opcode	Compute condition, adjust PC	(None)	(None)

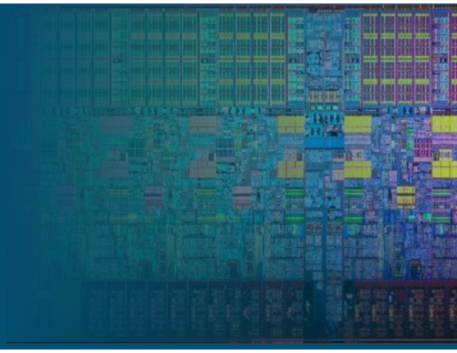


Faster Cycles: Pipelining



- Pipelinining: temporal parallelism
- Number of stages increase with each generation
- Minimum (ideal) CPI = 1
- No programmer effort required

Hazards



- Dependencies between instructions
- eg: Branch instruction changes PC when taken

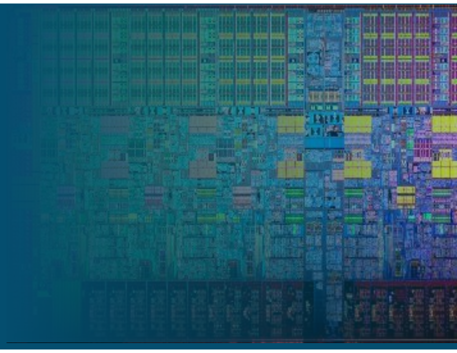
```
a:    lea 12(%eax),%ebx
      test %ebx
      jp nz,b
      inc %ebx
b:    ...
```

- Hazards reduce the efficiency of the pipeline

Improving Pipelines

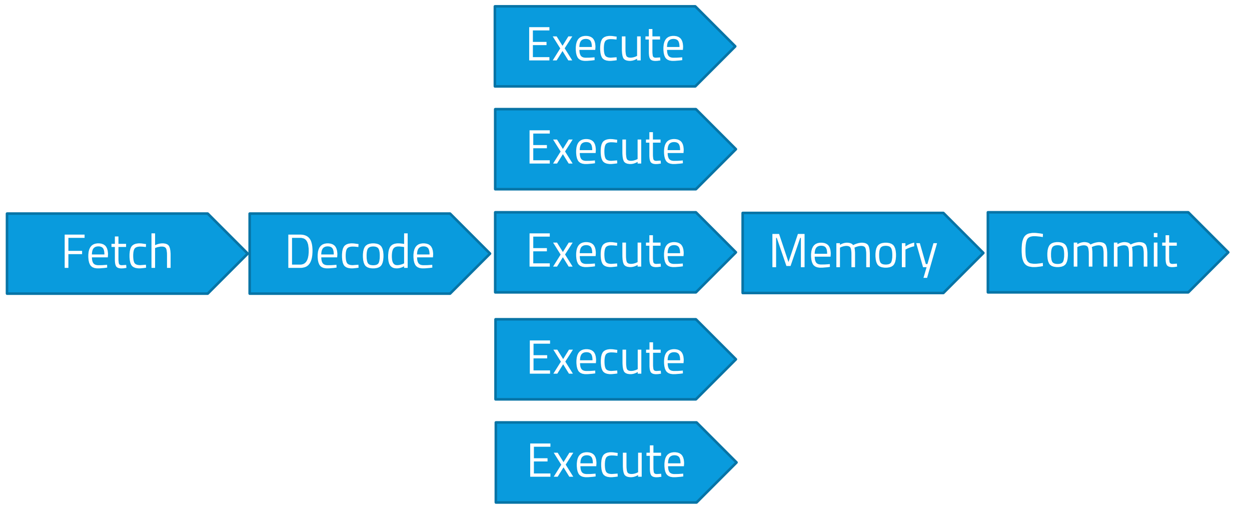
- How can we make the pipeline more efficient?

Improving Pipelines



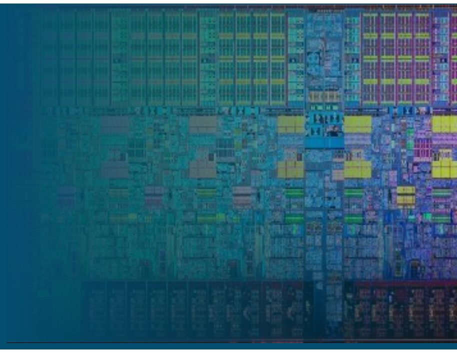
- How can we make the pipeline more efficient?
- **Branch Prediction**
  - Hint of conditional branch outcome ahead of time
- **Hazard Detection**
  - What if instruction n+1 needs result of instruction n?
  - Freeze the pipeline until result is ready
- **Forwarding Logic**
  - Reduce the cost of hazard detection

Improving Pipelines (1990s)



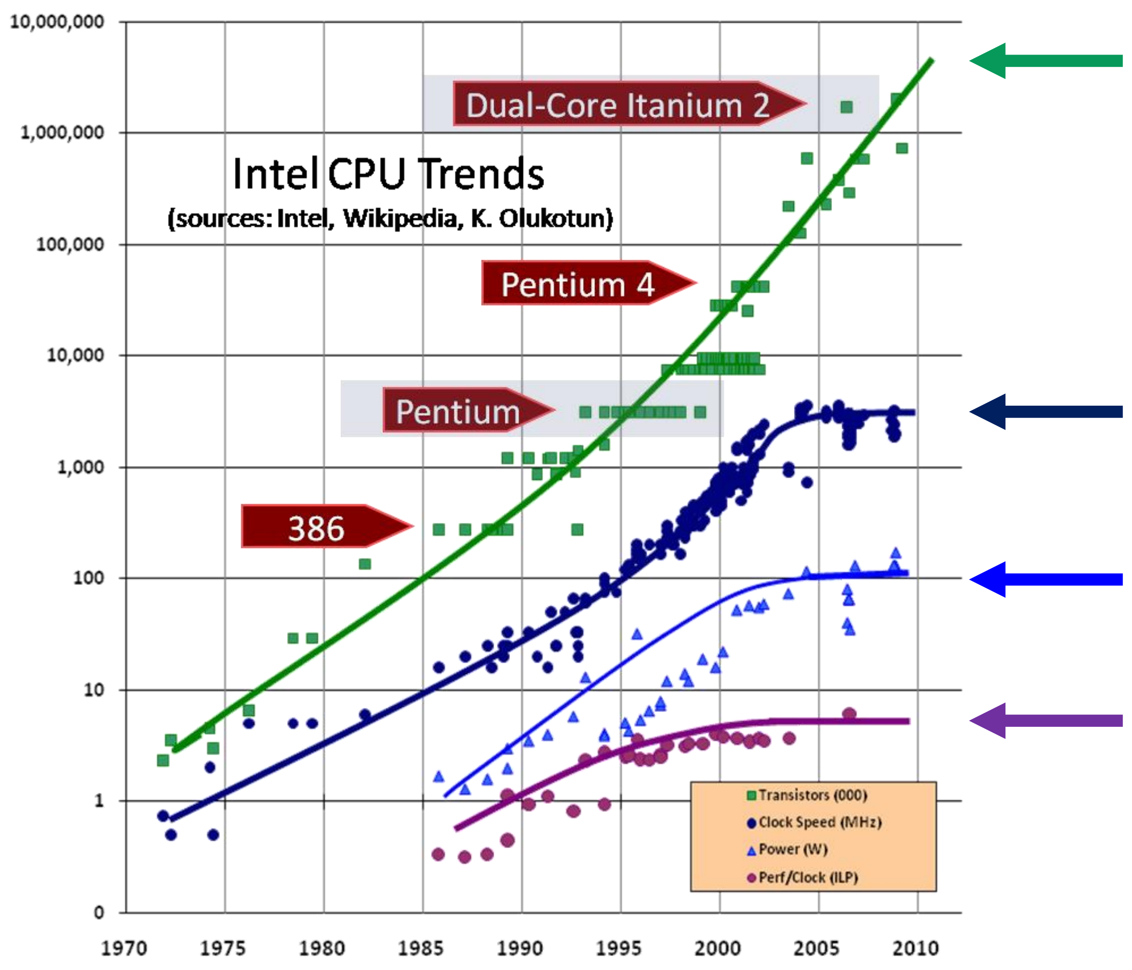
- Speed from
  - Duplicating functionality
  - Speeding up the clock
- Redundant units is instruction-level parallelism, not multiple cores!

The Clock Speed Limit



- We moved from single core to multicore for technological reasons (as we will see shortly)
- How to build the next computer, 1970 – 2000:
  do the same work, faster
  - Increase clock speed
  - Optimize execution (e.g., pipeline depth)
  - Enlarge and accelerate caches
- Free lunch is over for software folks
  - The software will not become faster with every new generation of processors

CPU Trends, 1970 - 2010



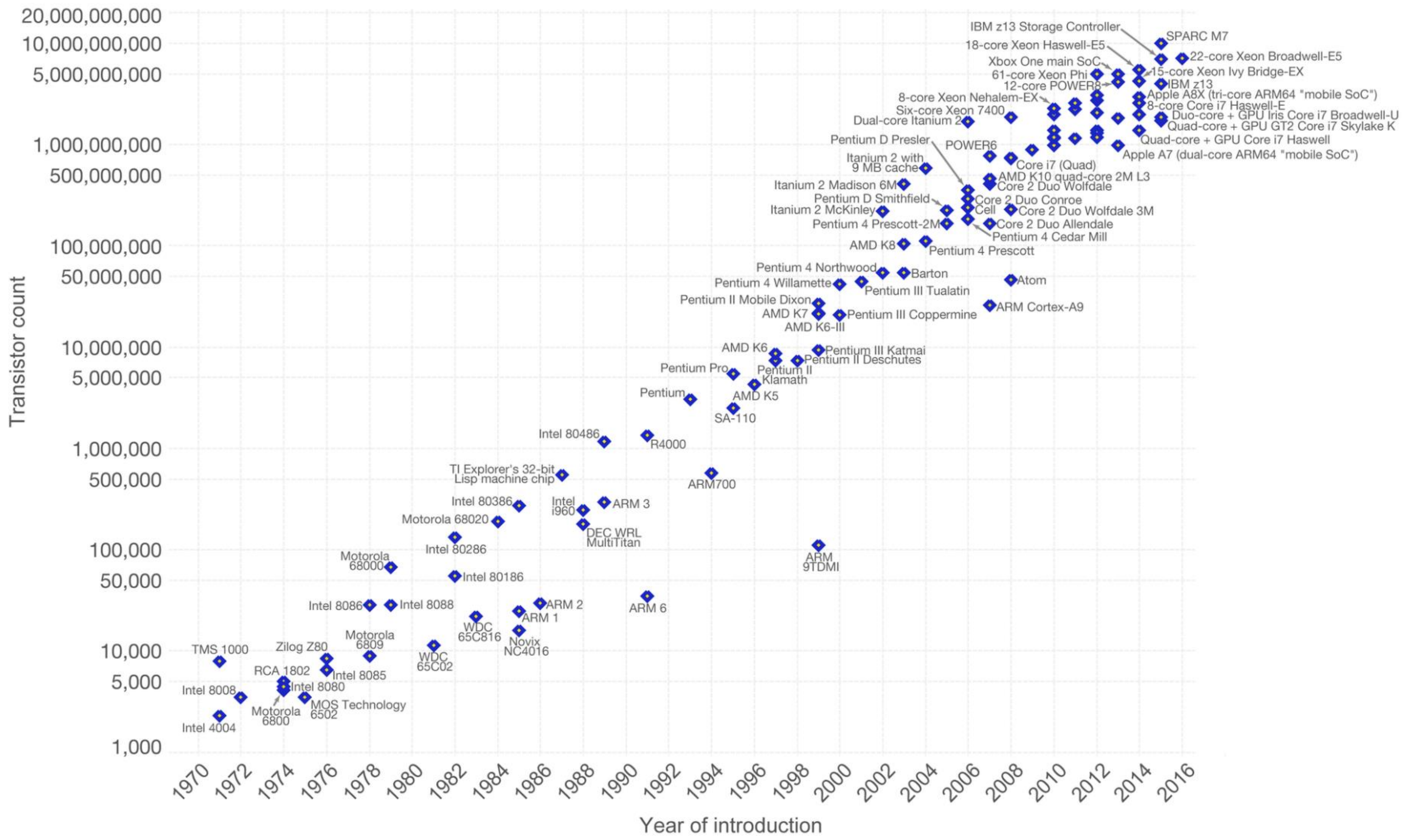
Number of transistors per CPU

Clock Speed

Power

Performance per clock cycle

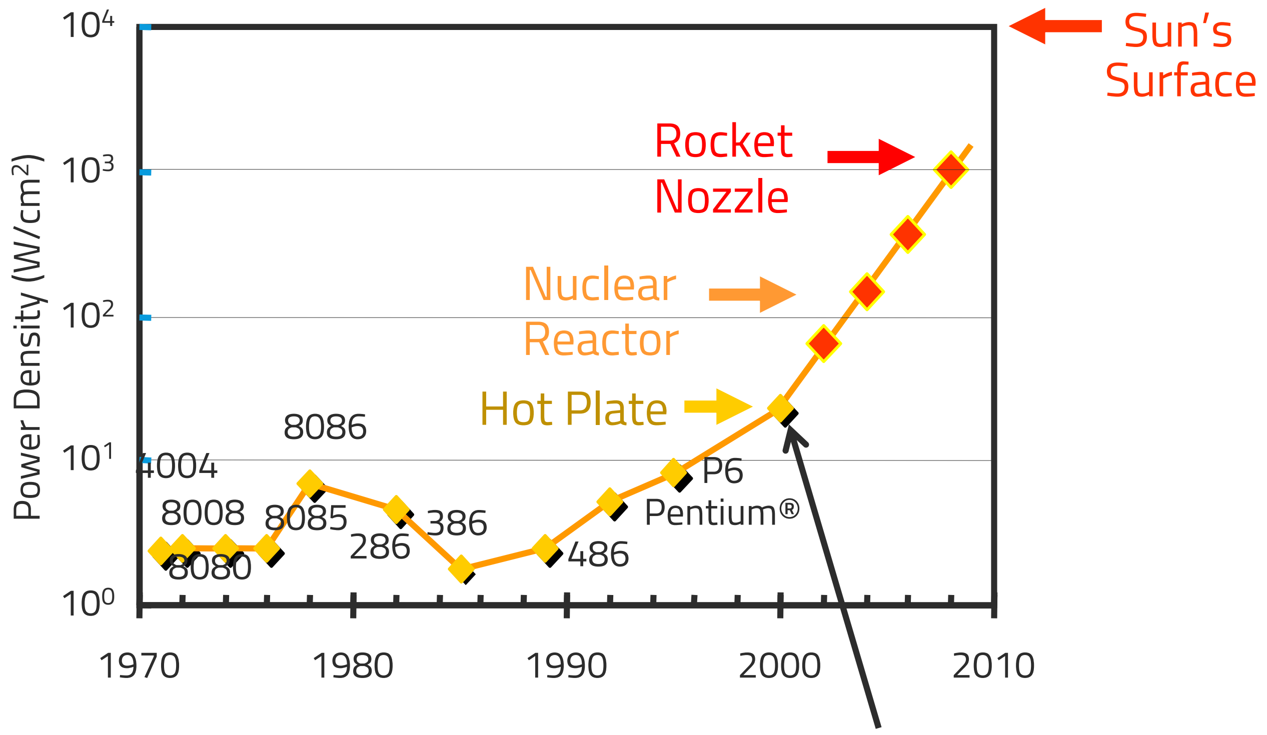
Moore’s Law



Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)
The data visualization is available at OurWorldinData.org. There you find more visualizations and research on this topic.

Power Density

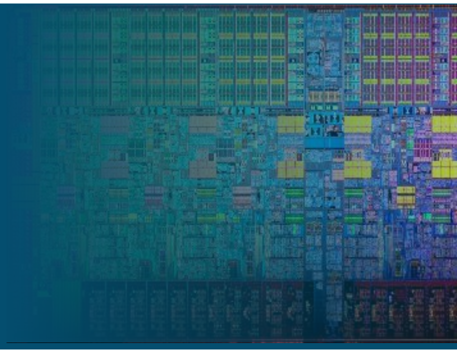
- **Moore's law is giving us more transistors than we can afford!**
- Scaling clock speed (business as usual) will not work



This is what happened around 2002!

Source: Patrick Gelsinger, Intel

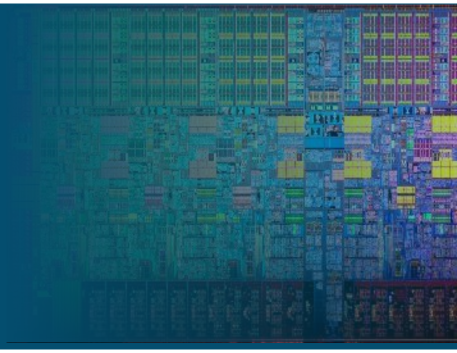
Moore’s Law Works Because of Dennard Scaling



“MOSFETs continue to function as voltage-controlled switches while all key figures of merit such as layout density, operating speed, and energy efficiency improve provided geometric dimensions, voltages, and doping concentrations are consistently scaled to maintain the same electric field.”

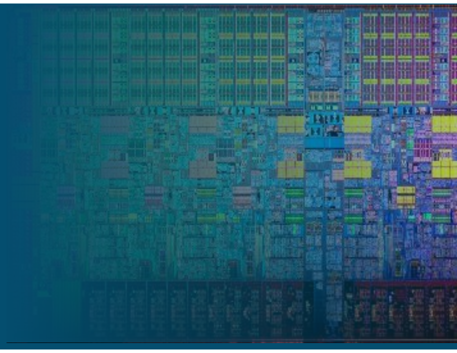
What??

“Smaller transistors -> less power per transistor -> power density stays constant”



Limits of Single-Core CPUs

- Effect of Moore's Law
  - ~1986 – 2002 → 50% performance increase per year
  - Since 2002 → ~20% performance increase per year
- ~2006: Dennard Scaling broke down
  - Current leakage
  - Thermal runaway
- Pipelines: wider than 6 units turned out to be very hard



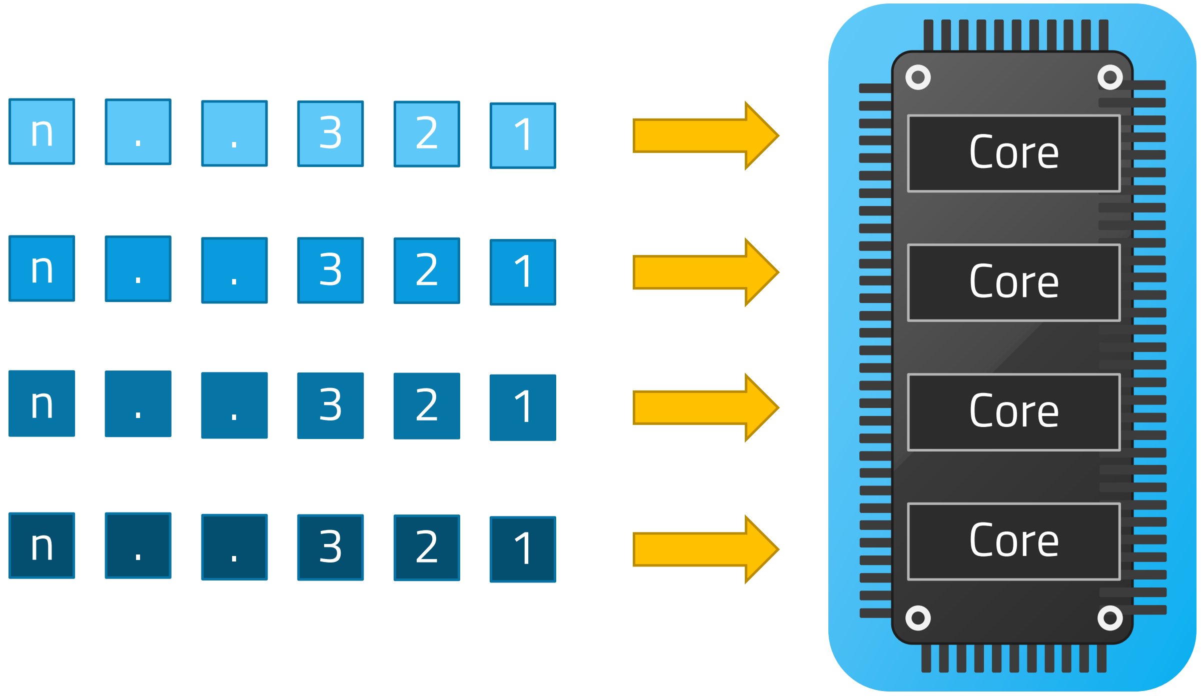
The Solution

- **Performance in the past achieved by:**
  - Increasing clock speed
  - Adding execution optimizations
  - Wider caches

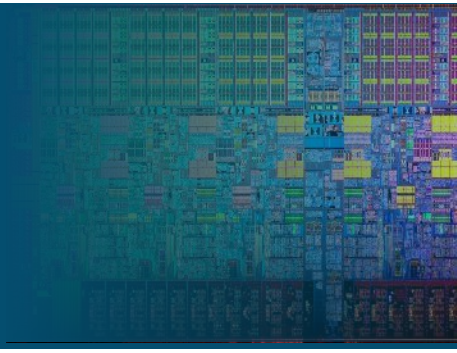
- **Performance currently achieved by:**
  - Hyperthreading
  - Multiple cores
  - Wider, deeper caches

The Solution

- Instead of designing and building faster microprocessors, put multiple processors on a single integrated circuit.

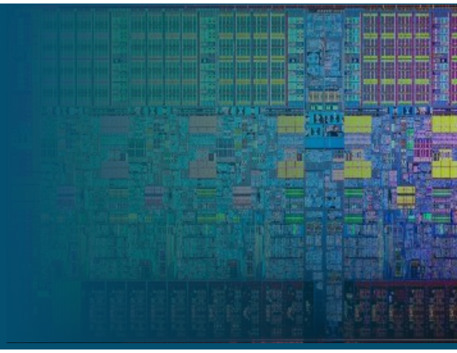


Why Do We Need More Performance?



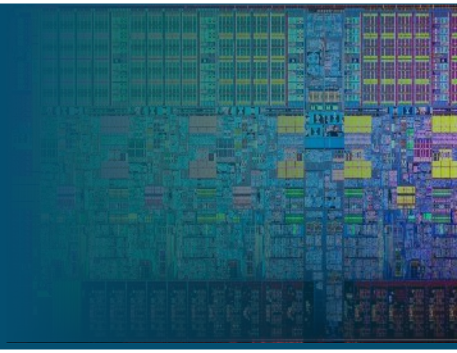
- More realistic games
- Decoding the human genome
- More accurate medical applications
- Applying “Deep Learning” to “Big Data”
- The list goes on and on …
- As our computational power increases → the number of problems we can seriously consider also increases.
- Dirty “secret”: new software needs more CPU power to do the same thing.

Programmers’ Burden



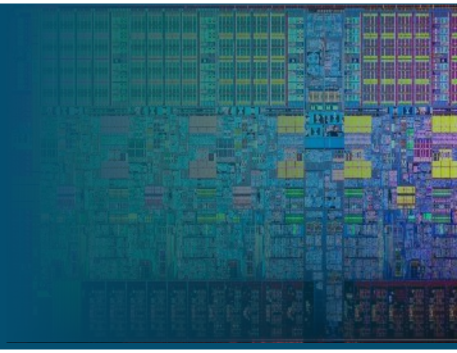
- Adding more processors doesn’t help much if programmers aren’t aware of them…
- … or don’t know how to use them.
- Serial programs don’t benefit from this approach (in most cases).

The Need for Parallel Programming



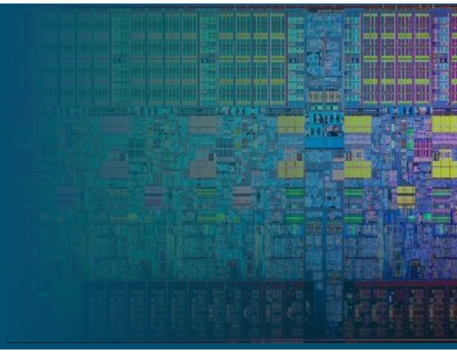
- Parallel computing: using multiple processors in parallel to solve problems more quickly than with a single processor
- Multicore machines aren't the only parallel machines:
  - A cluster computer that contains multiple PCs combined together with a high speed network
  - A shared memory multiprocessor (SMP) by connecting multiple processors to a single memory system
  - A Chip Multi-Processor (CMP) contains multiple processors (called cores) on a single chip

Cost and Challenges of Parallel Execution

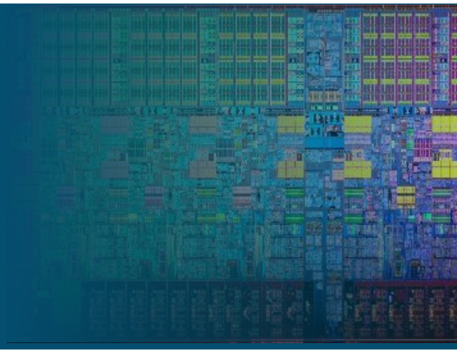


- Communication cost
- Synchronization cost
- Not all problems are amenable to parallelization
- Hard to think in parallel
- Hard to debug
- Even properly parallel programs don’t scale perfectly
  - “2 x 3GHz < 6 GHz”

Lecture 1 Outline



- Course Logistics and Syllabus
- History of the Multicore Processor
  - Evolution of the Modern CPU
  - Computer Architecture Interlude
  - The Performance Wall
  - Multicore Processors
- “Lab 0” Assignment



Lab 0 Assignment

- Assignment: Implement a list-based multimap
- See PDF on course webpage for details

- Meet or re-meet g++/gdb/git
- Implementation, testing, and submission

Due ~~Sunday, June 9, 2019~~ **TBD**