EC792
High Performance Computing Architecture

Lecture 1-3
Overview of Computer Architecture

Topics, Overview

Readings: Chapter 1

M S Bhat

msbhat@nitk.edu.in

Overview

Syllabus

- What you (should) know
  - Basic Computer Architecture & Organization, Logic Design, any one HDL.
- What you will learn (Course Overview)
  - Processor performance measures
  - Instruction Set Design
  - Advanced Pipelining
  - Instruction level parallelism
  - Data & Thread level parallelism
  - Memory Hierarchy, Memory consistency
  - Multiprocessors – some recent advances
- Why you should learn this

Course Pragmatics

- Reference Book
  - *Computer Architecture – A Quantitative approach, 6th Ed.* John L. Hennessy and David A. Patterson, Morgan Kaufmann, .2019.
  - *Computer Architecture: A Quantitative Approach, 5th ed.,*John L. Hennessey and David A. Patterson, Morgan Kaufman, 2012

At the end of the course, the student would

- Be able to compare the performance of alternative design choices in a computer architecture.
- Acquire the ability to use a hierarchical (layered) approach to understand a complex computer system.
- Learn the principles of datapath and control design for scalar and pipelined computer systems.
- Implement a minimal pipelined RISC ISA.
- Understand memory hierarchy, memory consistency and the principles of virtual memory.
- Become aware of current design issues and concerns.

Course Outcomes

- CO1 – Understand the fundamental operations of a modern computer through its Instruction Set Architecture.
- CO2 – Analyze and design different components of ALU, Datapath and Control unit using HDL.
- CO3 – Understand and analyze data-level parallelism to improve performance in a modern computer.
- CO4 – Understand and analyze the use of various levels in the memory hierarchy of a modern computer.
- CO5 – Design a full system for a modern day ISA and demonstrate working programs on it.

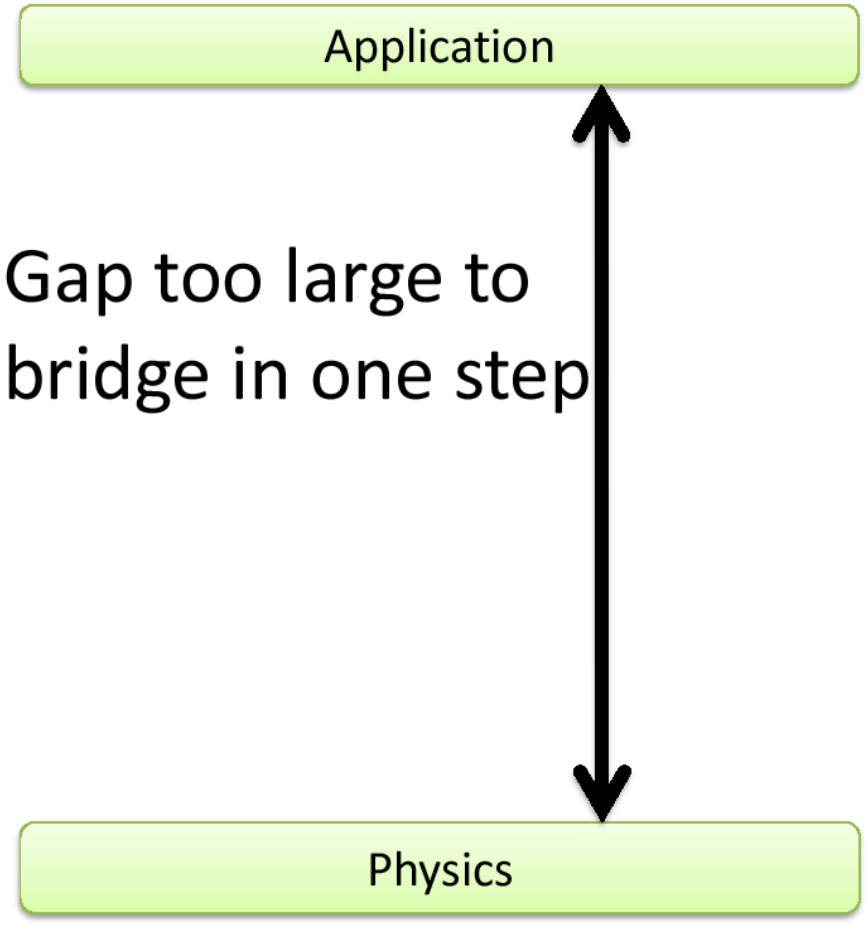
Course Coverage

The goal is to provide foundational knowledge of Computer Architecture with particular emphasis on improving processor performance.

Topics include:

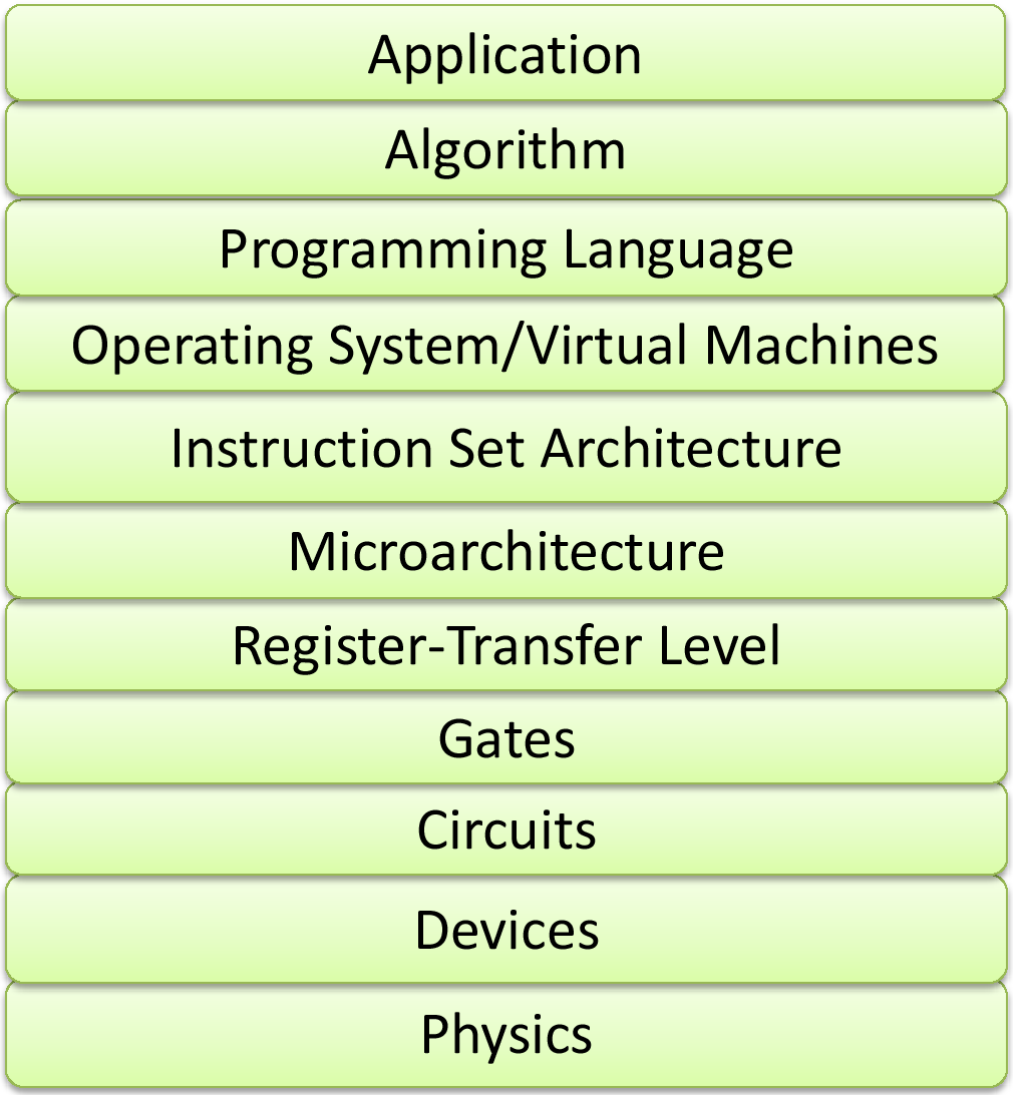
- Quantitative approach to computer architecture
- Review of the fundamentals of Pipelining, Instruction sets, and Memory Hierarchy
- Instruction Level parallelism
- Loop unrolling and static techniques
- Dynamic scheduling
- Limits on Instruction level Parallelism
- Multithreading, multiprocessing
- Memory consistency in Multi-processors.

What is Computer Architecture?

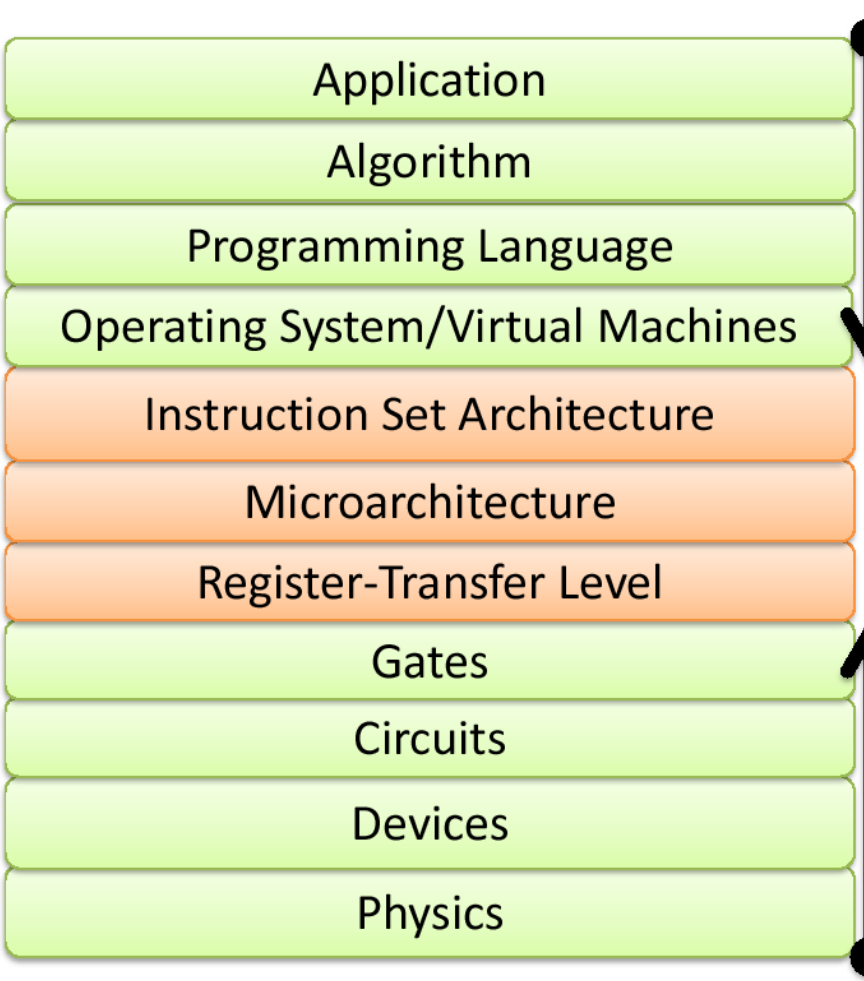


In its broadest definition, computer architecture is the design of the abstraction/implementation layers that allow us to execute information processing applications efficiently using manufacturing technologies

Abstraction in Modern Computing Systems



Computer Architecture is Constantly Changing



Application Requirements:

- Suggest how to improve architecture
- Provide revenue to fund development

Architecture provides feedback to guide application and technology research directions

Technology Constraints:

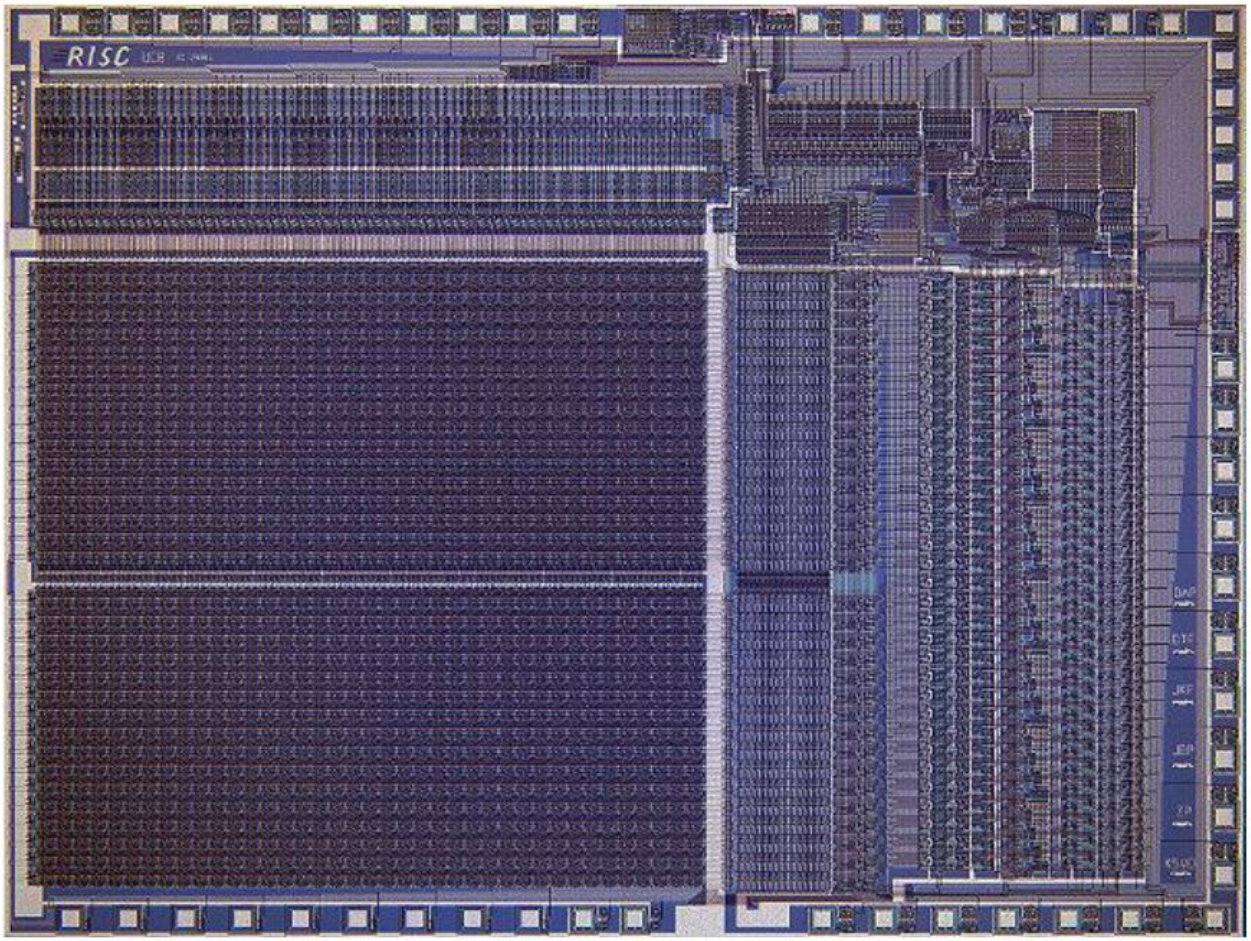
- Restrict what can be done efficiently
- New technologies make new arch possible

What is Computer Architecture?

- **Computer architecture** is the conceptual design and fundamental operational structure of a computer system. It is the technical drawings and functional description of all design requirements (especially speeds and interconnections)

- It deals with those aspects of the instruction set available to programmers, independent of the hardware on which the instruction set was implemented.

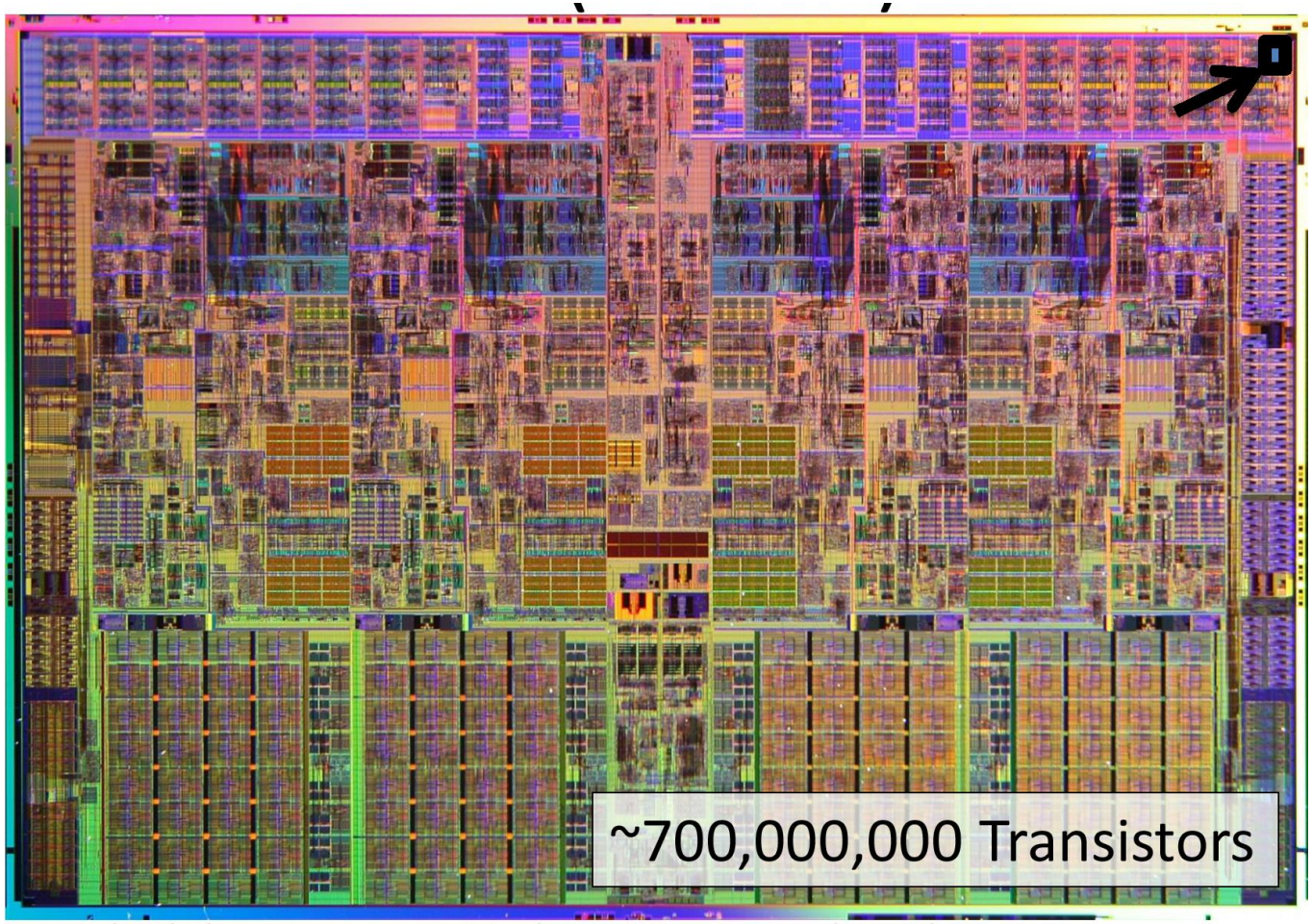
Computers Then



RISC 1 Processor
~50,000 Transistors

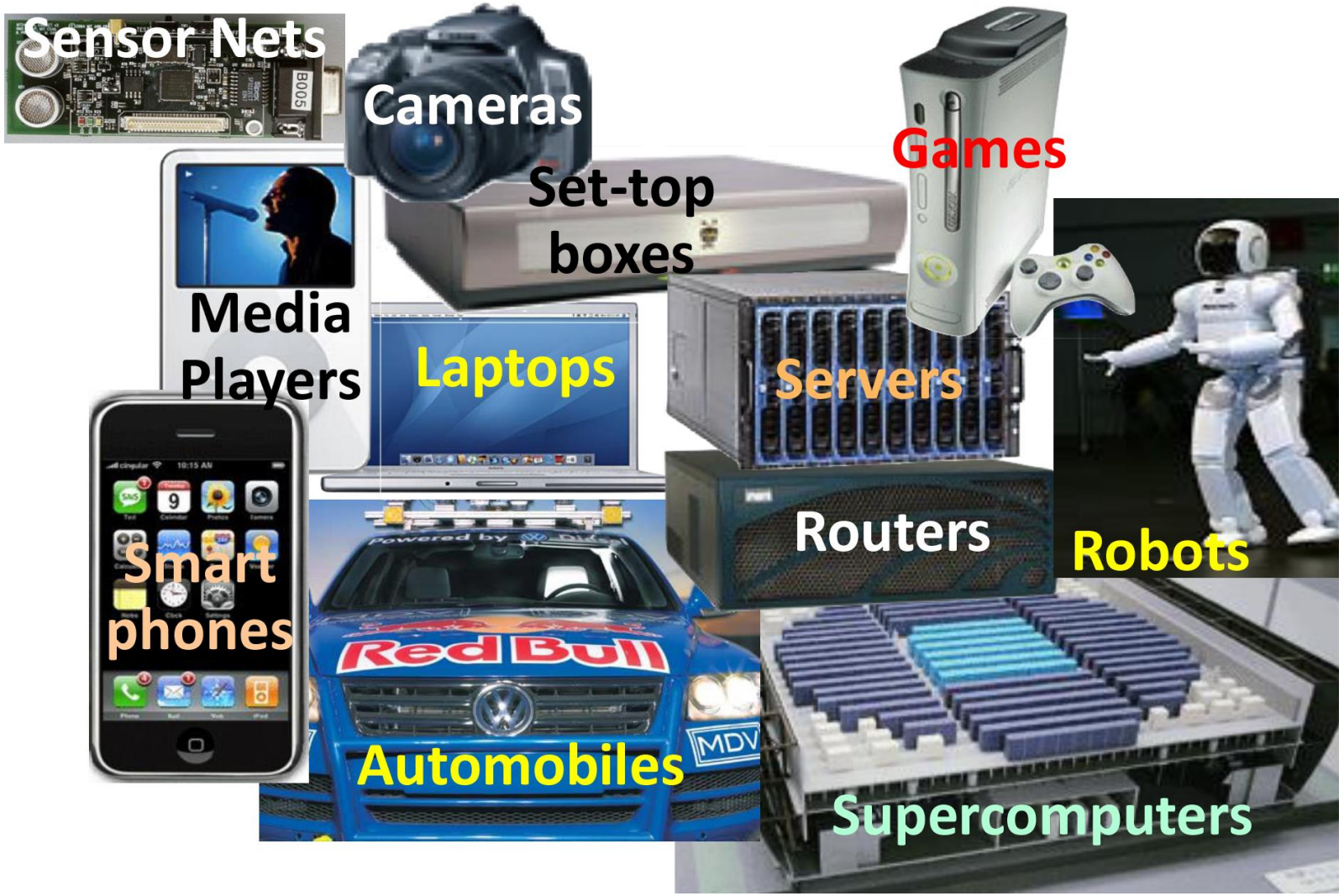
Photo of Berkeley RISC I, © University of California (Berkeley)

Computers - Recent past



Intel Nehalem Processor, Original Core i7, Image Credit Intel: http://download.intel.com/pressroom/kits/corei7/images/Nehalem_Die_Shot_3.jpg

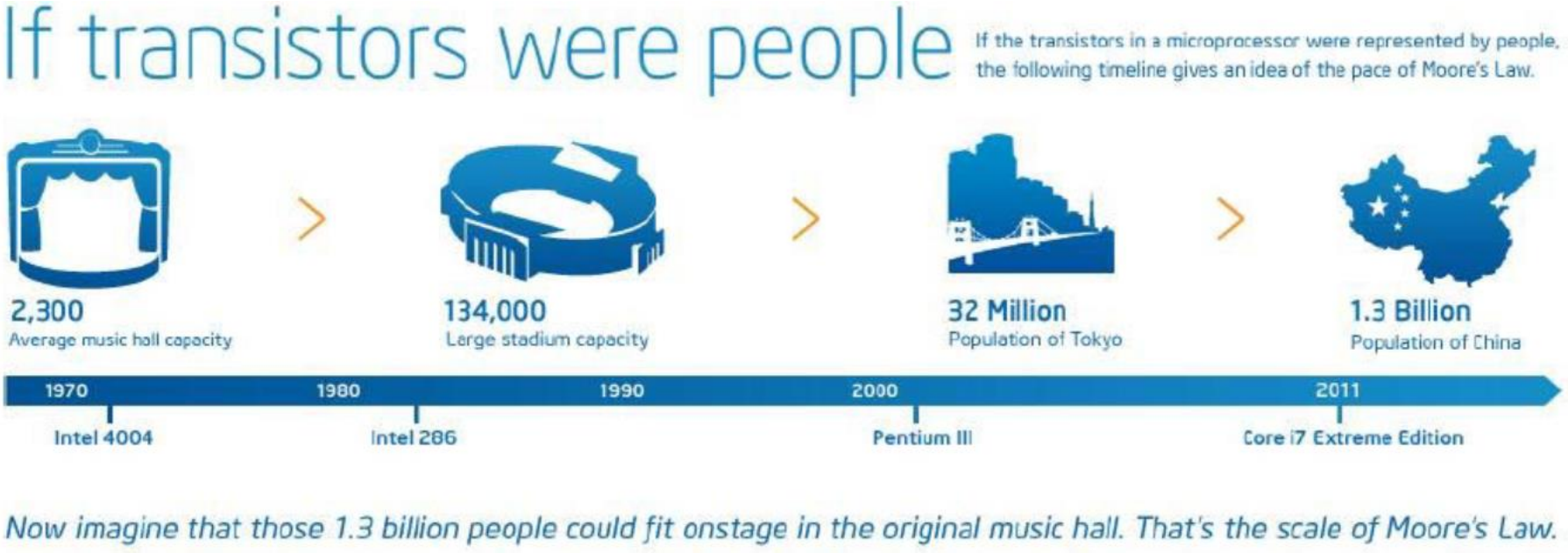
Computers Now



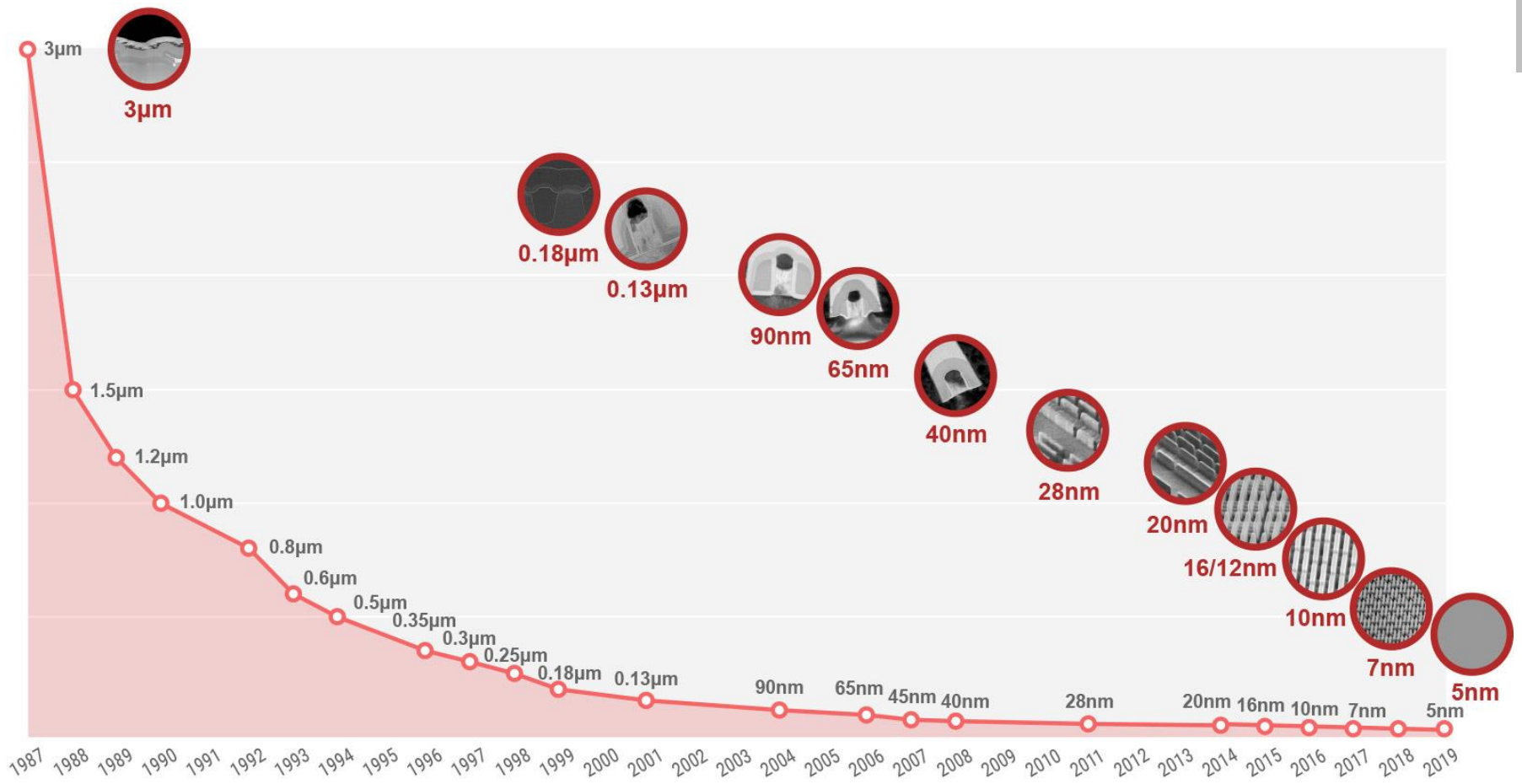
Moore's Law

Gordon Moore, one of the founders of Intel

- In 1965 he predicted the doubling of the number of transistors per chip every couple of years for the next ten years
- http://www.intel.com/research/silicon/mooreslaw.htm

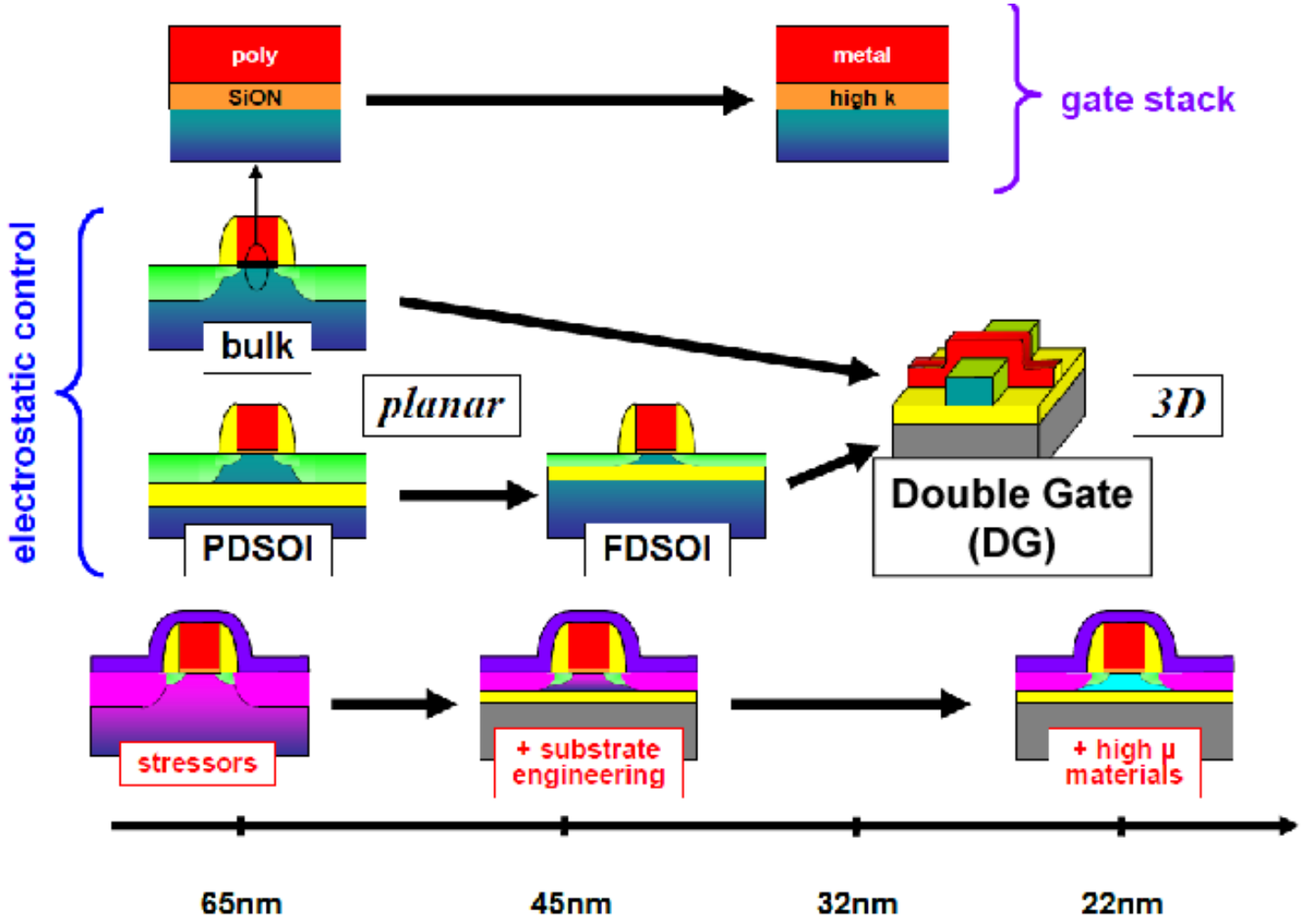


Technology Scaling



Data Source https://www.tsmc.com/english/dedicatedFoundry/technology/logic.htm

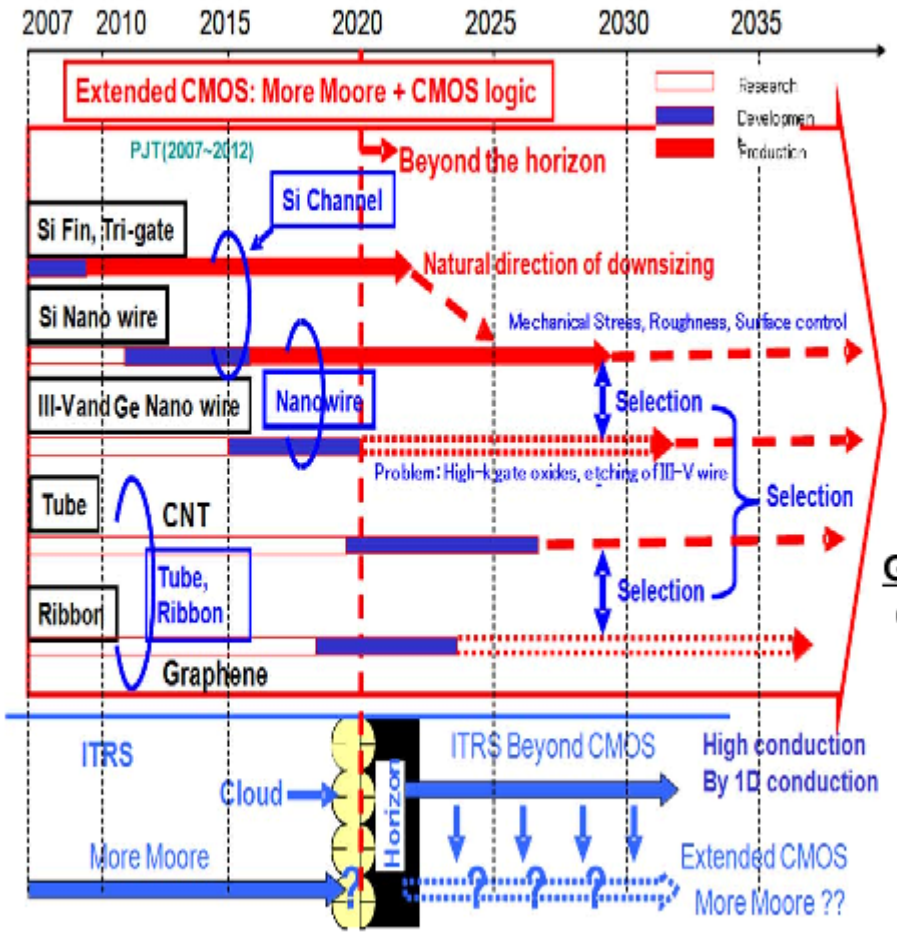
Technology update



Structure and technology innovation for MOSFETs

Data source : ITRS Roadmap 2013 http://www.itrs2.net/

Technology update



Current Issues

Si Nanowire

Control of wire surface property
Source Drain contact

Optimization of wire diameter

Compact I-V model

III-V & Ge Nanowire

High-k gate insulator
Wire formation technique

CNT:

Growth and integration of CNT
Width and Chirality control

Chirality determines conduction types: metal or semiconductor

Graphene:

Graphene formation technique
Suppression of off-current

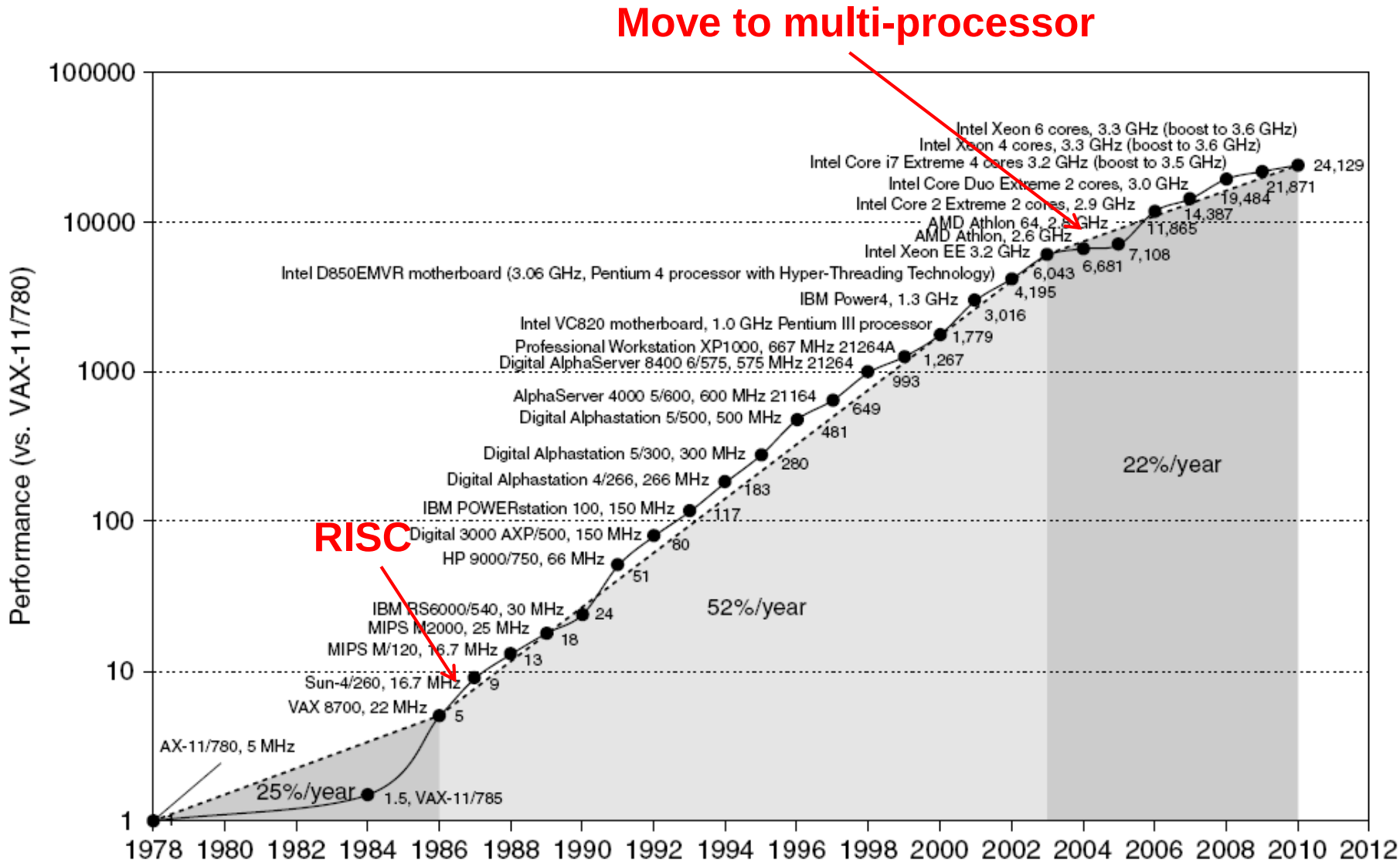
Very small bandgap or no bandgap (semi-metal)

Control of ribbon edge structure which affects bandgap

Long range roadmap for logic CMOS transistor research for 30 years.

Data source : ITRS Roadmap 2013 http://www.itrs2.net/

Sequential Processor Performance

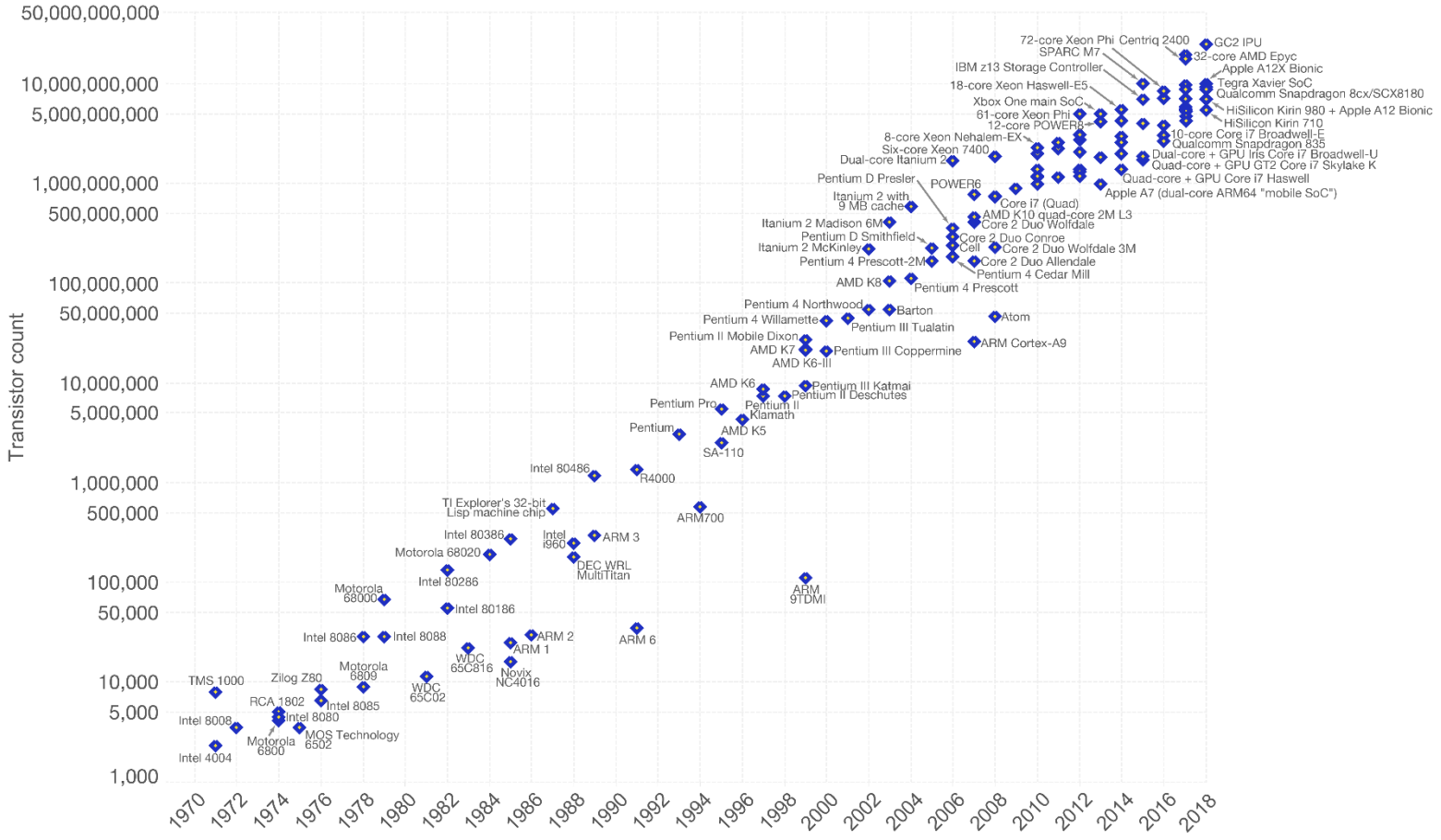


Moore's Law

Moore's Law – The number of transistors on integrated circuit chips (1971-2018)

Our World in Data

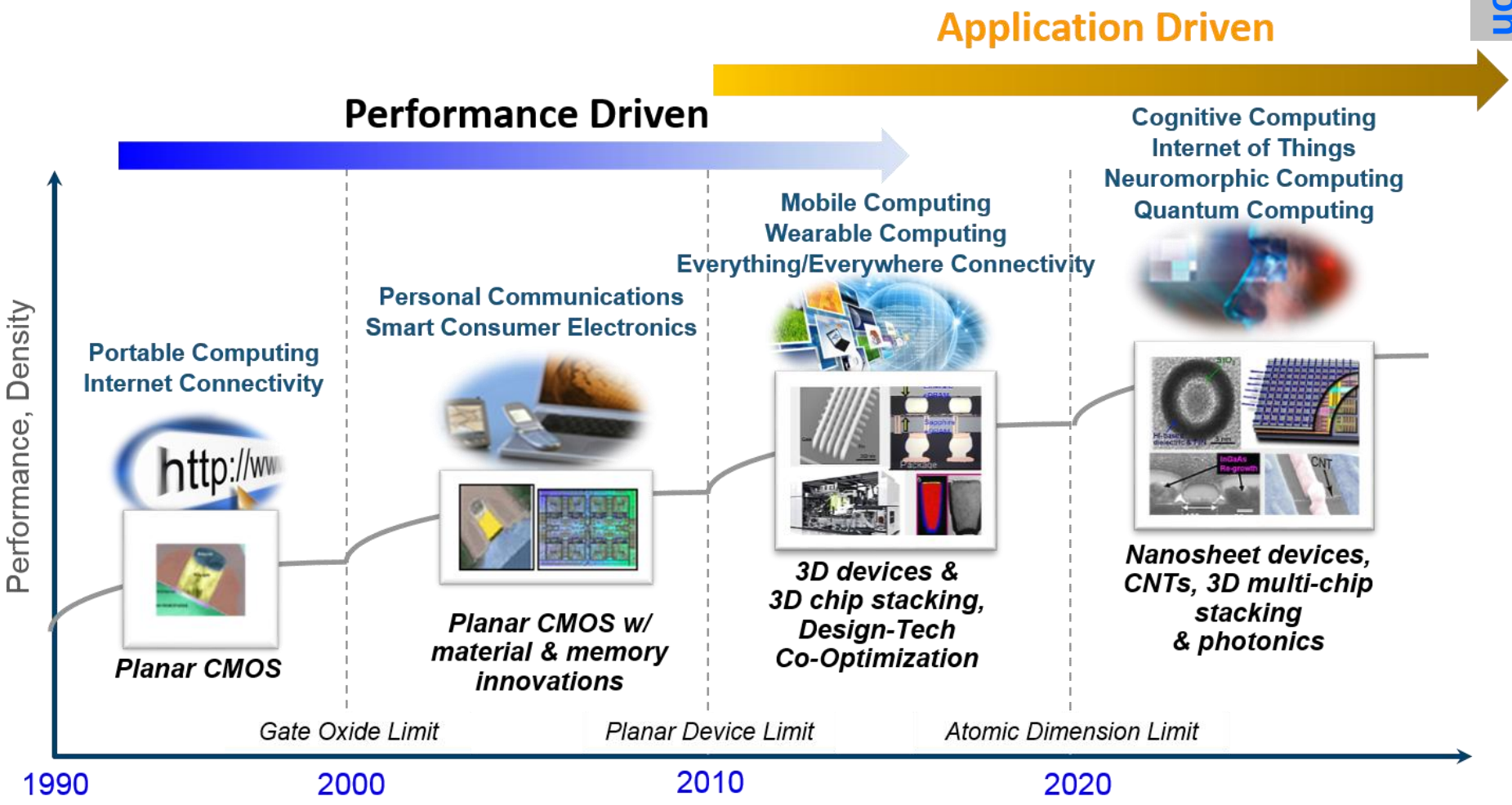
Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are linked to Moore's law.



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.

Data Source https://en.wikipedia.org/wiki/Transistor_count

Technology Advancements



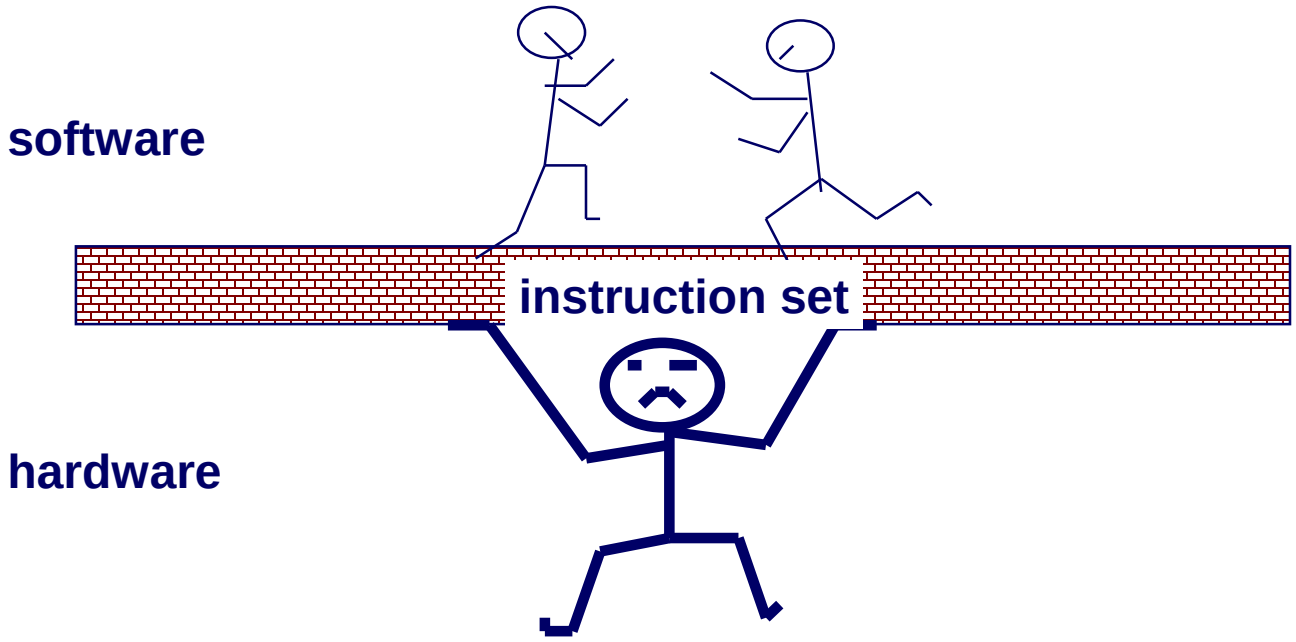
Data Source: https://researcher.watson.ibm.com/researcher/view_group.php?id=7859

Architecture vs. Microarchitecture

- "Architecture"/Instruction Set Architecture (ISA):
  - Programmer visible state (Memory & Register)
  - Operations (Instructions and how they work)
  - Execution Semantics (interrupts)
  - Input/Output
  - Data Types/Sizes
- Microarchitecture/Organization:
  - Tradeoffs on how to implement ISA for some metric (Speed, Energy, Cost)
  - Examples: Pipeline depth, number of pipelines, cache size, silicon area, peak power, execution ordering, bus widths, ALU widths

The Instruction Set: a Critical Interface

- ISA refers to the attributes of a [computing] system as seen by the programmer, i.e. the conceptual structure and functional behavior, as distinct from the organization of the data flows and controls of the logic design, and the physical implementation.
  - Amdahl and Brooks, 1964



Same Architecture
Different Microarchitecture

AMD Phenom X4

- X86 Instruction Set
- Quad Core
- 125W
- Decode 3 Instructions/Cycle/Core
- 64KB L1 I Cache, 64KB L1 D Cache
- 512KB L2 Cache
- Out-of-order
- 2.6GHz

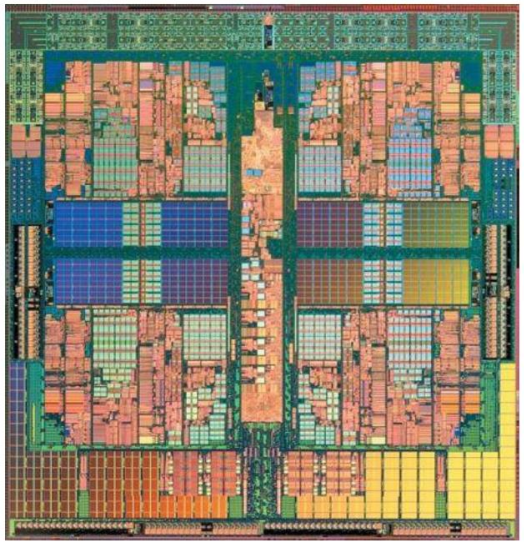


Image Credit: AMD

Intel Atom

- X86 Instruction Set
- Single Core
- 2W
- Decode 2 Instructions/Cycle/Core
- 32KB L1 I Cache, 24KB L1 D Cache
- 512KB L2 Cache
- In-order
- 1.6GHz

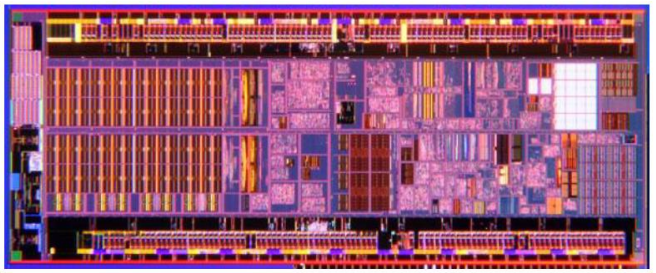


Image Credit: Intel

Different Architecture
Different Microarchitecture

AMD Phenom X4

- X86 Instruction Set
- Quad Core
- 125W
- Decode 3 Instructions/Cycle/Core
- 64KB L1 I Cache, 64KB L1 D Cache
- 512KB L2 Cache
- Out-of-order
- 2.6GHz

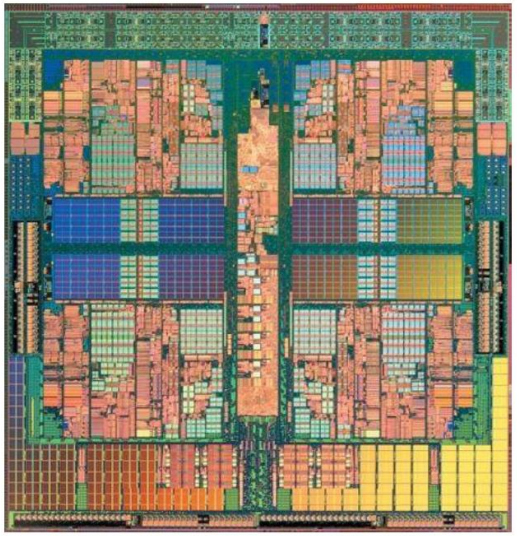


Image Credit: AMD

IBM POWER7

- Power Instruction Set
- Eight Core
- 200W
- Decode 6 Instructions/Cycle/Core
- 32KB L1 I Cache, 32KB L1 D Cache
- 256KB L2 Cache
- Out-of-order
- 4.25GHz

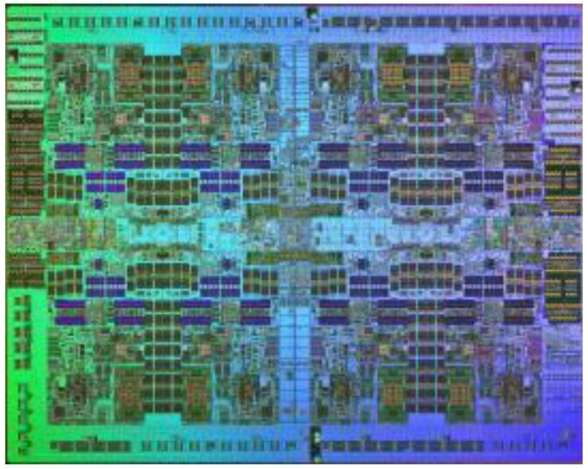


Image Credit: IBM

Crossroads: Conventional Wisdom in Comp. Arch

Old Conventional Wisdom: Power is free, Transistors expensive

New Conventional Wisdom: "Power wall" Power is expensive, Transistors are free
(Can put more on chip than can afford to turn on)

Old CW: Increase Instruction Level Parallelism via compilers, innovation (Out-of-order, speculation, VLIW, …)

New CW: "ILP wall" law of diminishing returns on more HW for ILP

Old CW: Multiplies are slow, Memory access is fast

New CW: "Memory wall" Memory slow, multiplies fast
(200 clock cycles to DRAM memory, 4 clocks for multiply)

Old CW: Uniprocessor performance 2X / 1.5 yrs

New CW: Power Wall + ILP Wall + Memory Wall = Brick Wall

⇒ Sea change in chip design: multiple "cores"
(2X processors per chip / ~ 2 years)

- More simpler processors are more power efficient

Transistors and Wires

Feature size

- Minimum size of transistor or wire in x or y dimension
- 10 microns in 1971 to .007 microns in 2017 and 0.005 micron in 2019
- Transistor performance scales linearly
  - Wire delay does not improve with feature size!
  - Long interconnect delay increases quadratically!
- Integration density scales quadratically

Current Trends in Architecture

Instruction-level parallelism (ILP):

- ILP is a measure of how many of the operations in a computer program can be performed simultaneously.
- The potential overlap among **instructions** is called **instruction level parallelism**.

Consider the following program:

1. e = a + b
2. f = c + d
3. m = e * f

Operation 3 depends on the results of operations 1 and 2, so it cannot be computed until both of them are completed. However, operations 1 and 2 do not depend on any other operation, so they can be computed simultaneously

- Single processor performance improvement ended in 2003
- Cannot continue to leverage Instruction-Level parallelism (ILP)

Current Trends in Architecture

Other models for performance:

- **Data-level parallelism (DLP) :SIMD (Image Proc., matrix operations)**
- **Thread-level parallelism (TLP)**
  - Thread - Separate process with instruction and data
    - Separate independent programs
    - Separate activities inside the same program etc.
    - Splitting program into independent tasks
    - Example: Parallel summation
    - Divide and conquer parallelism
    - Example: Parallel quicksort
- **Task-Level Parallelism (TLP)** :
  - The act of running multiple flows of execution of a single process simultaneously – e.g., Computing, memory access and IO simultaneously
- **Request-level parallelism (RLP)**

  These require explicit restructuring of the application

Parallelism

- Single instruction stream, single data stream (SISD)
- Single instruction stream, multiple data streams (SIMD)
  - Vector architectures
  - Multimedia extensions
  - Graphics processor units
- Multiple instruction streams, single data stream (MISD)
  - No commercial implementation
- Multiple instruction streams, multiple data streams (MIMD)
  - Tightly-coupled MIMD
  - Loosely-coupled MIMD

Main Memory

- DRAM – dynamic RAM – one transistor/capacitor per bit
- SRAM – static RAM – four to 6 (upto 10) transistors per bit
- DRAM density increases approx. 50% per year
- DRAM cycle time decreases slowly (DRAMs have destructive read-out, and data row must be rewritten after each read)
- DRAM must be refreshed every 2-8 ms
- Memory bandwidth improves about twice the rate that cycle time does due to improvements in signaling conventions and bus width

Trends in Technology

Integrated circuit technology

- Transistor density: 35%/year
- Die size: 10-20%/year
- Integration overall: 40-55%/year

DRAM capacity: 25-40%/year (slowing)

Flash capacity: 50-60%/year

- 15-20X cheaper/bit than DRAM

Magnetic disk technology: 40%/year

- 15-25X cheaper/bit than Flash
- 300-500X cheaper/bit than DRAM

Power and Energy

Problem: Power Dissipation

Peak Power, Avg. power

Thermal Design Power (TDP)

- Characterizes sustained power consumption
- Used as target for power supply and cooling system
- Lower than peak power, higher than average power consumption

Clock rate can be reduced dynamically to limit power consumption

Energy per task is often a better measurement

Summary: Power and Energy Equations

$$E = C_L V_{DD}^2 P_{0\rightarrow1} + t_{sc} V_{DD} I_{peak} P_{0\rightarrow1} + V_{DD} I_{leakage}$$

$$f_{0\rightarrow1} = P_{0\rightarrow1} * f_{clock}$$

$$P = C_L V_{DD}^2 f_{0\rightarrow1} + t_{sc} V_{DD} I_{peak} f_{0\rightarrow1} + V_{DD} I_{leakage}$$

Dynamic power (60-70% today and decreasing relatively)

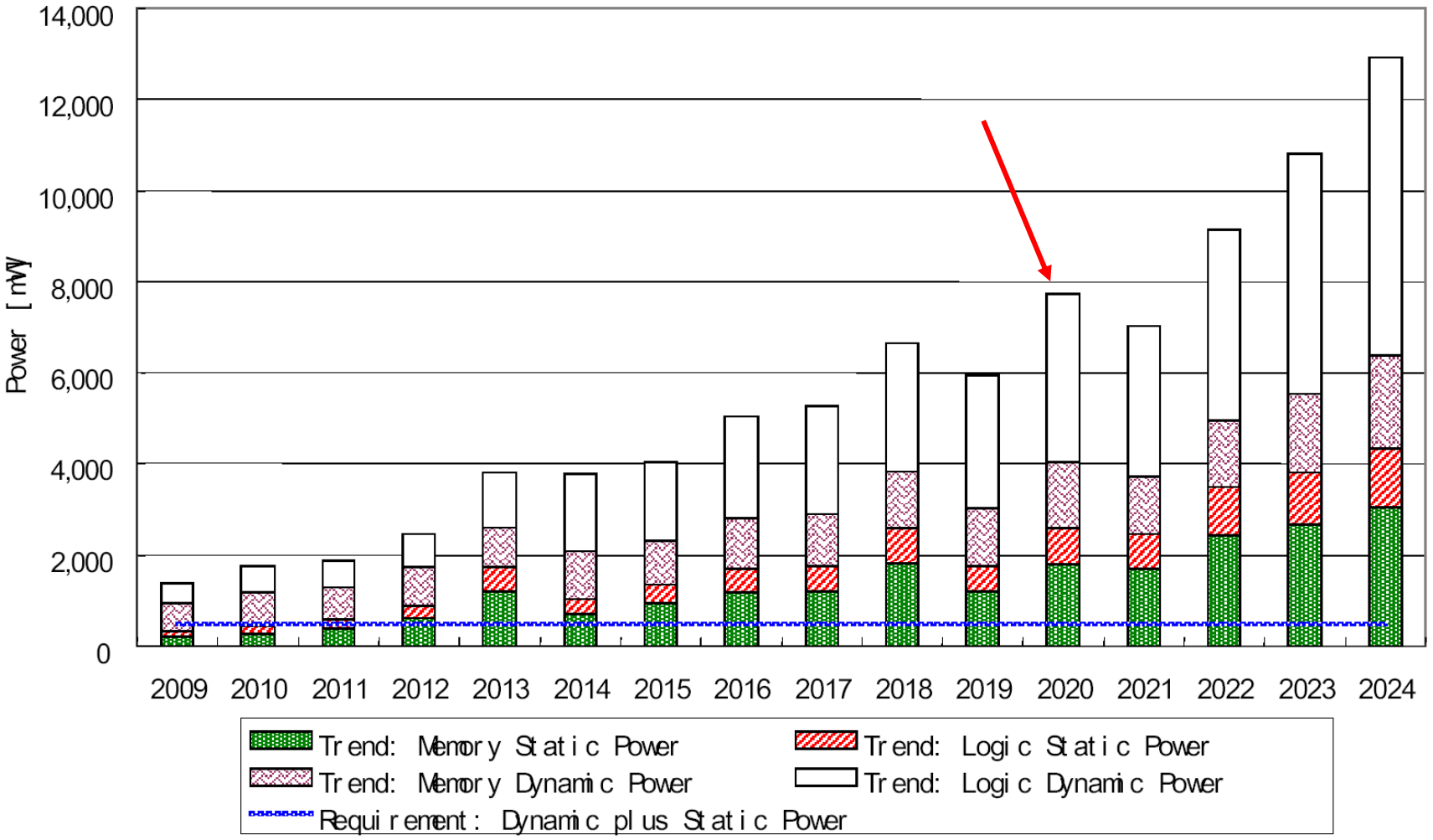
Short-circuit power (~5% today and decreasing absolutely)

Leakage power (30-40% today and **increasing relatively**)

- **Designers need to comprehend issues of memory and logic power, speed/power tradeoffs at the process (HiPerf vs. LowPower) level,**

Slide courtesy of Mary Jane Irwin, PSU

Static and Dynamic Power trends



Data source : ITRS Roadmap 2009

Dynamic Energy and Power

Dynamic energy

- Transistor switch from 0 -> 1 or 1 -> 0
- ½ x Capacitive load x Voltage^2

Dynamic power

- Capacitive load x Voltage^2 x Frequency switched

Reducing clock rate reduces power, not energy

Reducing Power

Techniques for reducing power:

- Dynamic Voltage-Frequency Scaling
- Low power state for DRAM, disks
- Overclocking, Clock gating, Turning off cores

Static Power

Static power consumption

- $\text{Current}_{\text{static}}$ x Voltage
- Scales with number of transistors
- To reduce: power gating

Cost of IC's

- **Cost of IC = (Cost of die + cost of testing die + cost of packaging and final test) / (Final test yield)**
- **Cost of die = Cost of wafer / (Dies per wafer * die yield)**
- **Dies per wafer is wafer area divided by die area, less dies along the edge**
- **≈ (wafer area) / (die area) - (wafer circumference) / (die diagonal)**

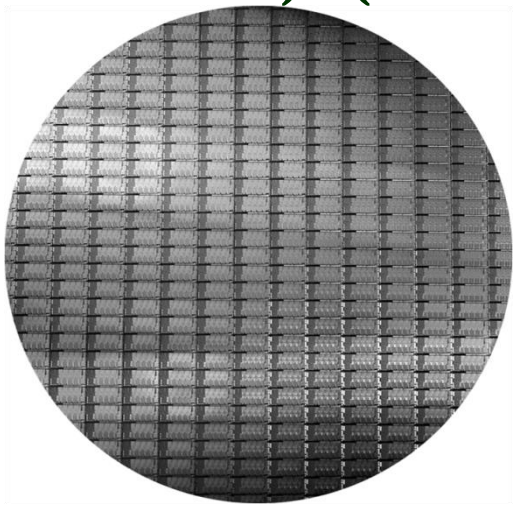


Figure : 300 mm (12 inch) wafer contains 280 full Sandy Bridge dies, each 20.7 by 10.5 mm in a 32 nm process. (Sandy Bridge is Intel's successor to Nehalem used in the Core i7.) At 216 mm^2, the formula for dies per wafer estimates 282. (Courtesy Intel.)

Performance Measures

Response time (latency) **: time between start and completion (Execution Time, elapsed time, wall-clock time etc.)**

Throughput (bandwidth) **: rate = work done per unit time**

Speedup **: B is n times faster than A**

- **exec_time_A/exec_time_B == rate_B/rate_A**

Other important measures

- **Power (impacts battery life, cooling, packaging)**
- **RAS (reliability, availability, and serviceability)**
- **Scalability (ability to scale up processors, memories, and I/O)**

Measuring Performance

Time is the measure of computer performance

Elapsed time = program execution + I/O + wait

-- important to user

CPU time = computing time on an unloaded system

-- this does not include memory access time.

-- important to architect

Bandwidth and Latency

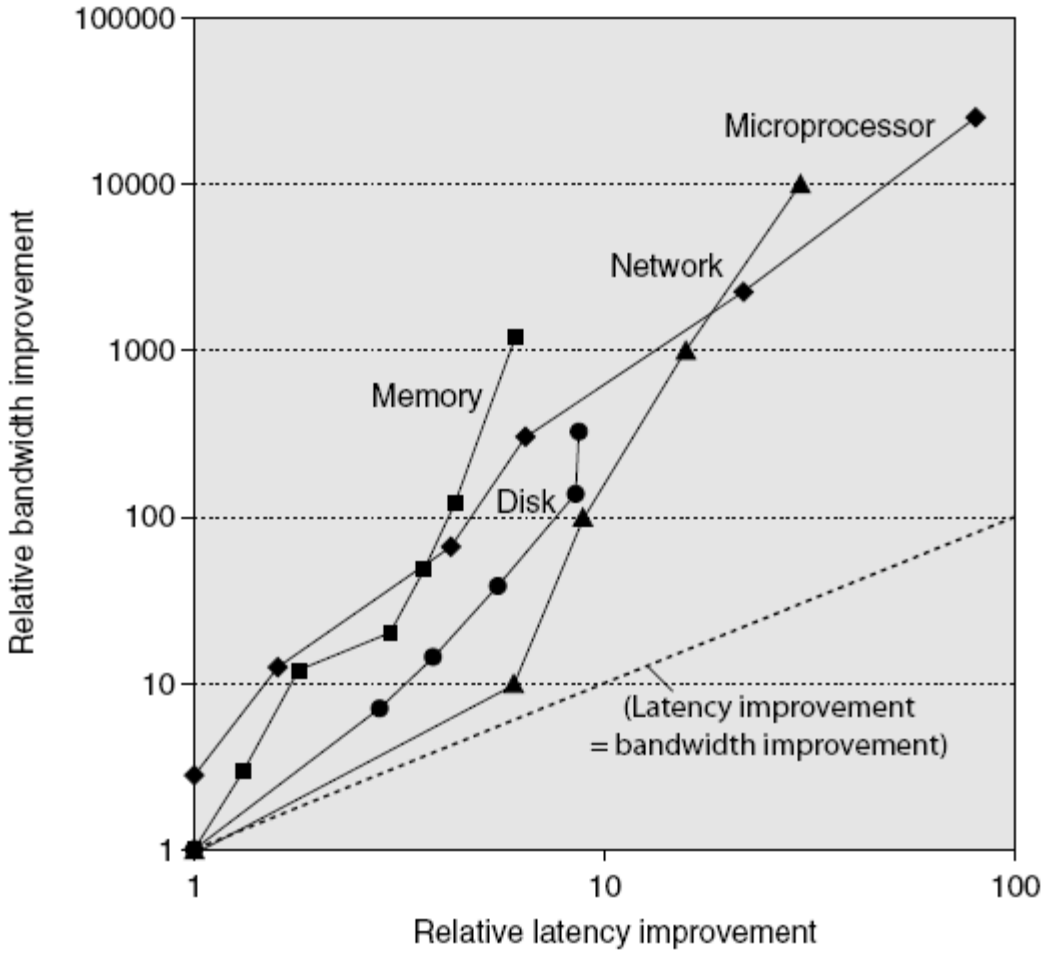
Bandwidth or throughput

- Total work done in a given time
- 10,000-25,000X improvement for processors
- 300-1200X improvement for memory and disks

Latency or response time

- Time between start and completion of an event
- 30-80X improvement for processors
- 6-8X improvement for memory and disks

Bandwidth and Latency



Log-log plot of bandwidth and latency milestones

Caveat: Gates vs. GM Joke contd.

- The oil, water temperature and alternator warning lights would be replaced by a single "general car default" warning light.
- The airbag system would say "Are you sure?" before going off.
- Occasionally for no reason whatsoever, your car would lock you out and refuse to let you in until you simultaneously lifted the door handle, turned the key, and grab hold of the radio antenna.
- Every time GM introduced a new model car buyers would have to learn to drive all over again because none of the controls would operate in the same manner as the old car.
- You'd press the "start" button to shut off the engine.

Real Performance

- **Benchmark suites (SPEC - Standard Performance Evaluation Corporation SPEC CPU2006 – 26 programs gives one SPEC rating, ESPRESSO….)**
- **Performance is the result of executing a workload on a configuration**
- **Workload = program + input**
- **Configuration = CPU + cache + memory + I/O + OS + compiler + optimizations**
- **compiler optimizations can make a huge difference**

Choosing Programs to evaluate Performance

- **Real programs – C compilers, TeX, CAD tools etc**
- **Kernels – Loops, LINPACK (e.g., Solving Ax=b using Gaussian elimination)**
- **Toy Benchmarks – solving Puzzles, quicksort etc.**
- **Synthetic Benchmarks – Whetstones, Dhrystones (floating point operations) etc.**

Measurement Tools

- Benchmarks, Traces, Mixes
- Hardware: Cost, delay, area, power estimation
- Simulation (many levels)
  - ISA, RT, Gate, Circuit
- Queuing Theory
- Rules of Thumb

Benchmark Suites

- Performance is measured with benchmark suites: a collection of programs that are likely relevant to the user
  - SPEC CPU 2006: cpu-oriented programs (for desktops)
  - SPECweb, TPC: throughput-oriented (for servers)
  - EEMBC: for embedded processors/workloads Whetstone (1976) -- designed to simulate arithmetic-intensive scientific programs.
  - Dhrystone (1984) -- designed to simulate systems programming applications. Structure, pointer, and string operations are based on observed frequencies, as well as types of operand access (global, local, parameter, and constant).
  - Whetstone – another synthetic benchmark
  - Linpack – FP benchmark (Ax=b)
  - PC Benchmarks – aimed at simulating real environments
    - Business Winstone – navigator + Office Apps

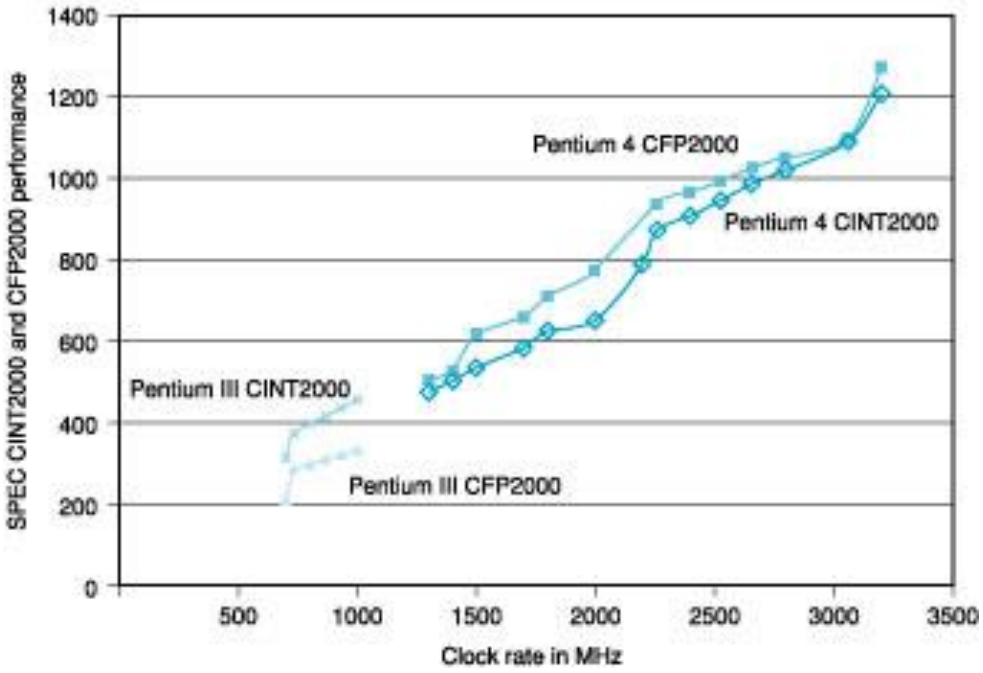
SPEC CPU Benchmarks www.spec.org

Integer benchmarks		FP benchmarks	
gzip	compression	wupwise	Quantum chromodynamics
vpr	FPGA place & route	swim	Shallow water model
gcc	GNU C compiler	mgrid	Multigrid solver in 3D fields
mcf	Combinatorial optimization	applu	Parabolic/elliptic pde
crafty	Chess program	mesa	3D graphics library
parser	Word processing program	galgel	Computational fluid dynamics
eon	Computer visualization	art	Image recognition (NN)
perlbmk	perl application	equake	Seismic wave propagation simulation
gap	Group theory interpreter	facerec	Facial image recognition
vortex	Object oriented database	ammp	Computational chemistry
bzip2	compression	lucas	Primality testing
twolf	Circuit place & route	fma3d	Crash simulation fem
		sixtrack	Nuclear physics accel
		apsi	Pollutant distribution

Programs in SPECCPU2000 benchmark suites.

Benchmark	Type	Source	Description
gzip	Integer	C	Compression using the Lempel-Ziv algorithm
vpr	Integer	C	FPGA circuit placement and routing
gcc	Integer	C	Consists of the GNU C compiler generating optimized machine code.
mcf	Integer	C	Combinatorial optimization of public transit scheduling.
crafty	Integer	C	Chess playing program.
parser	Integer	C	Syntactic English language parser
eon	Integer	C++	Graphics visualization using probabilistic ray tracing
perlmbk	Integer	C	Perl (an interpreted string processing language) with four input scripts
gap	Integer	C	A group theory application package
vortex	Integer	C	An object-oriented database system
bzip2	Integer	C	A block sorting compression algorithm.
twolf	Integer	C	Timberwolf: a simulated annealing algorithm for VLSI place and route
wupwise	FP	F77	Lattice gauge theory model of quantum chromodynamics.
swim	FP	F77	Solves shallow water equations using finite difference equations.
mgrid	FP	F77	Multigrid solver over 3-dimensional field.
apply	FP	F77	Parabolic and elliptic partial differential equation solver
mesa	FP	C	Three dimensional graphics library.
galgel	FP	F90	Computational fluid dynamics.
art	FP	C	Image recognition of a thermal image using neural networks
equake	FP	C	Simulation of seismic wave propagation.
facerec	FP	C	Face recognition using wavelets and graph matching.
ammp	FP	C	molecular dynamics simulation of a protein in water
lucas	FP	F90	Performs primality testing for Mersenne primes
fma3d	FP	F90	Finite element modeling of crash simulation
sixtrack	FP	F77	High energy physics accelerator design simulation.
apsi	FP	F77	A meteorological simulation of pollution distribution.

Example SPEC CPU Ratings



- Higher *Spec ratio* number indicates faster performance than a Sun Ultra 5_10 with a 300 MHz processor
- In this graph performance scales almost linearly with clock rate increase, **but this is often not the case.**

Comparing Performance

Total execution time (implies equal mix in workload)

- Just add up the times

Arithmetic average of execution time

- To get more accurate picture, compute the average of several runs of a program

Weighted execution time (weighted arithmetic mean)

- Program p1 makes up 25% of workload (estimated), P2 75% then use weighted average

Geometric mean consistent but cannot predict execution time

- N^{th} root of the product of execution time ratios

Example: Let's look at an example

The Bottom Line: Performance (and Cost)

Plane	New Delhi to Paris	Speed	Passengers	Throughput (in passenger mile per hour)
Boeing 747	6.5 hours	610 mph	470	286,700
Concorde	3 hours	1350 mph	132	178,200

- **Time to run the task (ExTime)**
  - **Execution time, response time, latency**
- **Tasks per day, hour, week, sec, ns … (Performance)**
  - **Throughput, bandwidth**

Defining (Speed) Performance

- Normally interested in reducing
  - Response time (aka execution time) – the time between the start and the completion of a task
    - Important to individual users
  - Thus, to maximize performance, need to minimize execution time

$$\text{performance}_X = 1 / \text{execution\_time}_X$$

If X is n times faster than Y, then

$$\frac{\text{performance}_X}{\text{performance}_Y} = \frac{\text{execution\_time}_Y}{\text{execution\_time}_X} = n$$

  - Throughput – the total amount of work done in a given time
    - Important to data center managers
  - Decreasing response time almost always improves throughput

Performance Factors

- Want to distinguish elapsed time and the time spent on our task
- CPU execution time (CPU time) : time the CPU spends working on a task
  - Does not include time waiting for I/O or running other programs

$$\text{CPU execution time for a program} = \text{\# CPU clock cycles for a program} \times \text{clock cycle time}$$

or

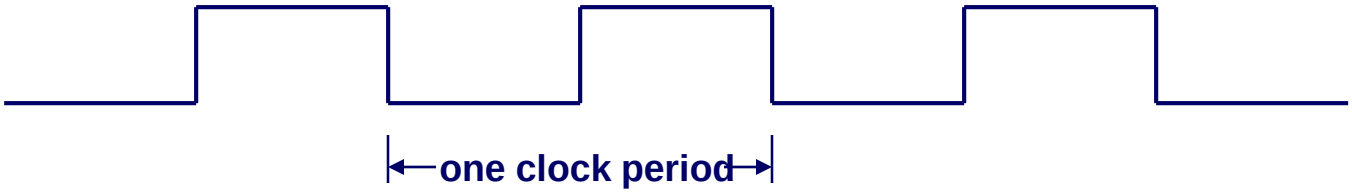
$$\text{CPU execution time for a program} = \frac{\text{\# CPU clock cycles for a program}}{\text{clock rate}}$$

- Can improve performance by reducing either the length of the clock cycle or the number of clock cycles required for a program

Review: Machine Clock Rate

- Clock rate (MHz, GHz) is inverse of clock cycle time (clock period)

CC = 1 / CR



10 nsec clock cycle => 100 MHz clock rate

1 nsec clock cycle => 1 GHz clock rate

500 psec clock cycle => 2 GHz clock rate

200 psec clock cycle => 5 GHz clock rate

Clock Cycles per Instruction

- Not all instructions take the same amount of time to execute
  - One way to think about execution time is that it equals the number of instructions executed multiplied by the average time per instruction

$$\text{\# CPU clock cycles for a program} = \text{\# Instructions for a program} \times \text{Average clock cycles per instruction}$$

- Clock cycles per instruction (CPI) – the average number of clock cycles each instruction takes to execute
  - A way to compare two different implementations of the same ISA

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

Effective CPI

- Computing the overall effective CPI is done by looking at the different types of instructions and their individual cycle counts and averaging

$$\text{Overall effective CPI} = \sum_{i=1}^{n} (CPI_i \times IC_i)$$

  - Where IC_i is the count (percentage) of the number of instructions of class i executed
  - CPI_i is the (average) number of clock cycles per instruction for that instruction class
  - n is the number of instruction classes

- The overall effective CPI varies by instruction mix – a measure of the dynamic frequency of instructions across one or many programs

THE Performance Equation

- Our basic performance equation is then

$$\text{CPU time} = \text{Instruction\_count} \times \text{CPI} \times \text{clock\_cycle}$$

or

$$\text{CPU time} = \frac{\text{Instruction\_count} \times \text{CPI}}{\text{clock\_rate}}$$

- These equations separate the three key factors that affect performance
  - Can measure the CPU execution time by running the program
  - The clock rate is usually given
  - Can measure overall instruction count by using profilers/ simulators without knowing all of the implementation details
  - CPI varies by instruction type and ISA implementation for which we must know the implementation details

Determinates of CPU Performance

CPU time = Instruction_count x CPI x clock_cycle

	Instruction_ count	CPI	clock_cycle
Algorithm	x	x	
Programming language	x	x	
Compiler	x	x	
ISA	x	x	x
Processor organization		x	x
Technology			x

A Simple Example

Op	Freq	CPI_i	Freq x CPI_i
ALU	50%	1	.5
Load	20%	5	1.0
Store	10%	3	.3
Branch	20%	2	.4
Overall effective CPI			$\sum =$ 2.2

A Simple Example

Op	Freq	CPI_i	Freq x CPI_i	Q1
ALU	50%	1	.5	.5
Load	20%	5	1.0	.4
Store	10%	3	.3	.3
Branch	20%	2	.4	.4
Overall effective CPI			$\sum =$ 2.2	1.6

- How much faster would the machine be if a better data cache reduced the average load time to 2 cycles?

A Simple Example

Op	Freq	CPI_i	Freq x CPI_i	Q1
ALU	50%	1	.5	.5
Load	20%	5	1.0	.4
Store	10%	3	.3	.3
Branch	20%	2	.4	.4
Overall effective CPI			$\sum =$ 2.2	1.6

- How much faster would the machine be if a better data cache reduced the average load time to 2 cycles?

CPU time new = 1.6 x IC x CC so 2.2/1.6 means 37.5% faster

(Notice that 2.2 x IC x CC / 1.6 x IC x CC = 2.2/1.6, since the IC and CC remain unchanged by adding a bigger cache)

A Simple Example

Op	Freq	CPI_i	Freq x CPI_i	Q2
ALU	50%	1	.5	.5
Load	20%	5	1.0	1.0
Store	10%	3	.3	.3
Branch	20%	2	.4	.2
Overall effective CPI			$\sum =$ 2.2	2.0

- How does this compare with using branch prediction to remove a cycle off the branch time?

A Simple Example

Op	Freq	CPI_i	Freq x CPI_i	Q2
ALU	50%	1	.5	.5
Load	20%	5	1.0	1.0
Store	10%	3	.3	.3
Branch	20%	2	.4	.2
Overall effective CPI			$\sum =$ 2.2	2.0

- How does this compare with using branch prediction to shave a cycle off the branch time?

CPU time new = 2.0 x IC x CC so 2.2/2.0 means 10% faster

A Simple Example

Op	Freq	CPI_i	Freq x CPI_i	Q3
ALU	50%	1	.5	.25
Load	20%	5	1.0	1.0
Store	10%	3	.3	.3
Branch	20%	2	.4	.4
Overall effective CPI			$\sum =$ 2.2	1.95

- What if two ALU instructions could be executed at once?

A Simple Example

Op	Freq	CPI_i	Freq x CPI_i	Q3
ALU	50%	1	.5	.25
Load	20%	5	1.0	1.0
Store	10%	3	.3	.3
Branch	20%	2	.4	.4
Overall effective CPI			$\sum$ = 2.2	1.95

- What if two ALU instructions could be executed at once?

CPU time new = 1.95 x IC x CC so 2.2/1.95 means 12.8% faster

A Simple Example

Op	Freq	CPI_i	Freq x CPI_i	Q1	Q2	Q3
ALU	50%	1	.5	.5	.5	.25
Load	20%	5	1.0	.4	1.0	1.0
Store	10%	3	.3	.3	.3	.3
Branch	20%	2	.4	.4	.2	.4
Overall effective CPI			$\sum =$ 2.2	1.6	2.0	1.95

- How much faster would the machine be if a better data cache reduced the average load time to 2 cycles?
  CPU time new = 1.6 x IC x CC so 2.2/1.6 means 37.5% faster
- How does this compare with using branch prediction to remove a cycle off the branch time?
  CPU time new = 2.0 x IC x CC so 2.2/2.0 means 10% faster
- What if two ALU instructions could be executed at once?
  CPU time new = 1.95 x IC x CC so 2.2/1.95 means 12.8% faster

Comparing and Summarizing Performance

- How do we summarize the performance for benchmark set with a single number?
  - The average of execution times that is directly proportional to total execution time is the arithmetic mean (AM)

$$AM = 1/n \sum_{i=1}^{n} Time_i$$

  - Where $Time_i$ is the execution time for the i^{th} program of a total of n programs in the workload
  - A smaller mean indicates a smaller average execution time and thus improved performance

- Guiding principle in reporting performance measurements is reproducibility – list everything another experimenter would need to duplicate the experiment (version of the operating system, compiler settings, input set used, specific computer configuration (clock rate, cache sizes and speed, memory size and speed, etc.))

Comparing Performance

- Consider 3 programs from a benchmark set – how do we capture the behavior of 3 processors A, B & C for all 3 programs with a single number? Assume Sys-A to be a reference machine.

	Prgm1	Prgm2	Prgm3	SET	SWET	GM
Sys-A	10	8	25	43	3	12.6
Sys-B	12	9	20	41	3.1	12.9
Sys-C	8	8	30	46	3	12.43

  - Sum of execution times (SET) (Arithmetic Mean: AM)
  - Sum of weighted execution times (SWET) (AM) w.r.t. Sys-A
  - Geometric mean of execution times (GM)
    - Let E1,E2,E3 be the exec times of Sys-A for Prgm1, Prgm2, Prgm3

$$GM = \sqrt[3]{(E1 * E2 * E3)}$$

Sum of Weighted Exec Times – Example

- Let us fix a reference machine , X and run 4 programs A, B, C, D on it such that each program will run for 1 second (Total time = 4s.)
- The exact same workload (the four programs execute the same number of instructions that they did on machine X) is run on a new machine Y and the execution times for each program are 0.8, 1.1, 0.5, 2 (Total time = 4.4s.)
- With AM of normalized execution times, we can conclude that Y is 1.1 times slower than X – perhaps, not for all workloads, but atleast for this workload

GM Example

	Computer-A	Computer-B	Computer-C
P1	1 sec	10 secs	20 secs
P2	1000 secs	100 secs	20 secs
GM	31.62	31.62	20

Conclusion with GMs: (i) A=B
(ii) C is ~1.6 times faster

- For (i) to be true for AM too, P1 must occur 100 times for every occurrence of P2 i.e., Consider : P1x100+P2 . Now take GM for this scenario

- With the above assumption, (ii) is no longer true. Hence, GM can lead to inconsistencies

GM Example

	Computer-A	Computer-B	Computer-C
P1	1 sec	10 secs	20 secs
P2	1000 secs	100 secs	20 secs
GM	31.62	31.62	20

$$Spec_{B1} = \frac{Spec\ ratio_A}{Spec\ ratio_B} = \frac{Exec\ time_B}{Exec\ time_A} = \frac{10}{1} \qquad Spec_{C1} = \frac{Spec\ ratio_A}{Spec\ ratio_C} = \frac{Exec\ time_C}{Exec\ time_A} = \frac{20}{1}$$

$$Spec_{B2} = \frac{Spec\ ratio_A}{Spec\ ratio_B} = \frac{Exec\ time_B}{Exec\ time_A} = \frac{100}{1000} \qquad Spec_{C2} = \frac{Spec\ ratio_A}{Spec\ ratio_C} = \frac{Exec\ time_C}{Exec\ time_A} = \frac{20}{1000}$$

$$\sqrt{\frac{Spec_{B1} * Spec_{B2}}{Spec_{C1} * Spec_{C2}}} = \frac{GM_B}{GM_C}$$

Summarizing Performance

GM: Does not require a reference machine, but does not predict performance very well. So we multiplied execution times and determined that sys-C is 1.6x faster…but on what workload?

AM: Does predict performance for a specific workload, but that workload was determined by executing programs on a reference machine

Every year or so, the reference machine will have to be updated

CPU Performance Equation

- Clock cycle time = 1 / clock speed

- CPU time = Instruction_count x CPI x clock_cycle

- Influencing factors for each:
  - Instruction count: instruction set design and compiler
  - CPI: architecture and instruction set design
  - Clock cycle time: technology and pipeline

- CPI (cycles per instruction) or IPC (instructions per cycle) can not be accurately estimated analytically

An Alternative Perspective - I

- Each program is assumed to run for an equal number of cycles, so we're fair to each program

- The number of instructions executed per cycle is a measure of how well a program is doing on a system

- The appropriate summary measure is sum of IPCs or

$$\text{AM of IPCs} = \frac{1.2 \text{ instr}}{\text{cyc}} + \frac{1.8 \text{ instr}}{\text{cyc}} + \frac{0.5 \text{ instr}}{\text{cyc}} \left(= \frac{3.5 \text{ instr}}{3 \text{ cyc}} \right)$$

This measure implicitly assumes that 1 instr in prog-A has the same importance as 1 instr in prog-B

An Alternative Perspective - II

- Each program is assumed to run for an equal number of instructions, so we're fair to each program

- The number of cycles required per instruction is a measure of how well a program is doing on a system

- The appropriate summary measure is sum of CPIs or
  AM of CPIs = $\frac{0.8 \text{ cyc}}{\text{instr}} + \frac{0.6 \text{ cyc}}{\text{instr}} + \frac{2.0 \text{ cyc}}{\text{instr}} \left(= \frac{3.4 \text{ cyc}}{3 \text{ instr}} \right)$

- This measure implicitly assumes that 1 instr in prog-A has the same importance as 1 instr in prog-B

Problem 2

- My new laptop has an IPC that is 20% worse than my old laptop. It has a clock speed that is 30% higher than the old laptop. I'm running the same binaries on both machines. What speedup is my new laptop providing?

Exec time = cycle time * CPI * instrs

Perf = clock speed * IPC / instrs

Speedup = new perf / old perf

= new clock speed * new IPC / old clock speed * old IPC

= (new clock speed / old clock speed)*(new IPC/old IPC)

= 1.3 * 0.8 = 1.04

Speedup Vs. Percentage

- "Speedup" is a ratio = old exec time / new exec time
- "Improvement", "Increase", "Decrease" usually refer to percentage relative to the baseline = (new perf – old perf) / old perf
- A program ran in 100 seconds on my old laptop and in 70 seconds on my new laptop
  - What is the speedup? (100/70) = 1.42
  - What is the percentage increase in performance? $\frac{\frac{1}{70} - \frac{1}{100}}{\frac{1}{100}} = 42\%$
  - What is the reduction in execution time?

$$\frac{100 - 70}{100} = 30\%$$

Improve Performance

- **At Software level, change**
  - **algorithm**
  - **data structures**
  - **programming language**
  - **compiler**
  - **OS parameters**
- **Take advantage of parallelism:**
  - **At system level – Data Parallelism**
    - **Multiple processors, multiple disks : *Scalability***
  - **At processor level – Instruction Level Parallelism**
    - **Pipelining**
  - **At block level - e.g., Set associative cache, carry look ahead adder etc.**
- **Principle of locality: 10% code 90% of the time**
  - **locality of reference – Temporal and spatial**
- **Make common case fast**

Quantitative principle of computer design

- Make the common case fast
  - Improve the execution time of frequently occurring operations.
- Make design decisions that make frequently used operations fast, even if they make some infrequently used operations slow.
- Limits of improvement
  - Improvement is limited by how frequent the frequent case is!

Make the Common Case Fast!

Most pervasive principle in design

Common case

- Need to validate what is common or uncommon
- H/W that isn't used still costs you
- S/W done right that isn't used probably doesn't cost you

Its often the case that

- Common cases are simpler than uncommon ones
  - e.g. exceptions, overflow, interrupts, etc.
- Truly simple is often both cheap and fast.

Amdahl's Law

Quantification of the diminishing return principle

Defines speedup gained from a particular feature

$$\text{Speedup} = \frac{\text{Execution time without using the enhancement}}{\text{Execution time using the enhancement}}$$

Exec-time = 1/performance

Depends on 2 factors

- Fraction of original computation time that can take advantage of the enhancement
- Level of improvement of the enhancement

Amdahl's Law

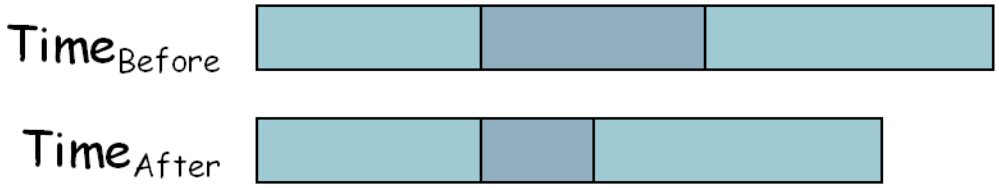
Speedup due to enhancement E:

$$\text{Speedup(E)} = \frac{\text{ExTime w/o E}}{\text{ExTime w/ E}} = \frac{\text{Performance w/ E}}{\text{Performance w/o E}}$$

Suppose that enhancement E accelerates a fraction F of the task by a factor S, and the remainder of the task is unaffected

Amdahl's Law

Speedup is due to enhancement(E):



Let F be the fraction where enhancement is applied => Also, called parallel fraction and (1-F) as the serial fraction

$$\textit{Execution time}_{after} = \text{ExTime}_{before} \times \left[(1-F) + \frac{F}{S}\right]$$

$$\textit{Speedup(E)} = \frac{\text{ExTime}_{before}}{\text{ExTime}_{after}} = \frac{1}{\left[(1-F) + \frac{F}{S}\right]}$$

S = $\text{Speedup}_{enhanced}$
F = $\text{Fraction}_{enhanced}$

Amdahl's Law

$$\text{ExTime}_{\text{after}} = \text{ExTime}_{\text{before}} \times \left[(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}} \right]$$

$$\text{Speedup}_{\text{overall}} = \frac{\text{ExTime}_{\text{before}}}{\text{ExTime}_{\text{after}}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

Simple Example

In an application:

- FP instructions account for 50% of instructions
- FPSQRT is 20%
- Other 30%

The following are possible (at the same cost)

- Improve FPSQRT by 40x
- improve all FP by 2x
- Improve other (non sqrt) FP by 8x

Which one should you do?

And the answer is…

$$\text{FPSQRT} = \frac{1}{(1-0.2) + \frac{0.2}{40}} = 1.242$$

$$\text{FP} = \frac{1}{(1-0.5) + \frac{0.5}{2}} = 1.333$$

$$\text{OTHER} = \frac{1}{(1-0.3) + \frac{0.3}{8}} = 1.356$$

Another Amdahl's Example

Suppose application is "almost all" parallel: 90%

- **What is the speedup using 10, 100, and 1000 processors?**

# of processors = P

Fraction $_{enhanced}$ = 0.9

$$\text{Speedup}_p = \frac{1}{0.1 + \frac{0.9}{P}}$$

$$\text{Speedup}_{10} = \frac{1}{0.1 + \frac{0.9}{10}} = \frac{1}{0.19} = 5.3$$

$$\text{Speedup}_{100} = \frac{1}{0.1 + \frac{0.9}{100}} = \frac{1}{0.109} = 9.1$$

$$\text{Speedup}_{1000} = 9.9$$

Example contd.

$$\text{Speedup}_{100} = \frac{1}{0.1 + \frac{0.9}{100}} = \frac{1}{0.109} = 9.1$$

What if you speed up the non-parallel part by a factor of 2?

$$\text{Speedup}_{100+2} = \frac{1}{\frac{0.1}{2} + \frac{0.9}{100}} = \frac{1}{0.05 + 0.009} = 16.95$$

Summary: Evaluating ISAs

- Design-time metrics:
  - Can it be implemented, in how long, at what cost?
  - Can it be programmed? Ease of compilation?
- Static Metrics:
  - How many bytes does the program occupy in memory?
- Dynamic Metrics:
  - How many instructions are executed? How many bytes does the processor fetch to execute the program?
  - How many clocks are required per instruction?
  - How "lean" a clock is practical?

Best Metric: Time to execute the program!

Depends on the instructions set, the processor organization, and compilation techniques.