

# ECE565: Computer Architecture

Instructor: T. N. Vijaykumar

Fall 2009  
Purdue University

Slides include material from Profs. Mark Hill, David Wood, Guri Sohi, and Jim Smith at the University of Wisconsin-Madison

## ECE 565: Computer Architecture

---

Course schedule (approximate) on the course website

Computer Architecture: A Quantitative Approach

- Hennessy and Patterson, Fourth Edition

Related conference papers - both classic and cutting-edge

- ~1.5 papers a week - you should form study groups

Term project options:

- SimpleScalar simulator - will involve C programming

url: <http://shay.ecn.purdue.edu/~ee565>

## ECE 565: Computer Architecture

---

State-of-the-art processor design

Emphasis on quantitative performance

Advanced memory systems

I/O

Very basic multiprocessing

Blue sky

## ECE 565: Computer Architecture

---

Instructor information

- Prof. T. N. Vijaykumar
- EE 334B, vijay@ecn
- Office hours: T, W 4:30 or by appointment via email

I do research in

- computer architecture
- general-purpose high-performance microprocessors
- chip multiprocessors, low power, fault/defect tolerance
- hardware for networking, security

## Expected Background

---

ECE437 or equivalent

- Design simple uniprocessor
- Simple instruction sets
- Organization
- Datapath design
- Hardwired/microprogrammed control
- Simple pipelining
- Basic caches

C/C++ programming experience

## Related Courses

---

EE666 - multiprocessors

EE573 - compilers

EE559 - VLSI design

CS503 - OS

Math 532 - queueing theory (NOT linear algebra)

EE663 - advanced compilers

To be a good architect, you should know about compilers and OS!

## Student background

---

This class is a mix of people

- those that have had EE437 (i.e. simple pipelines, caches)
- those that have not even heard these terms

If you are in the second group:

- to ensure that your grade does not suffer
- read the book AHEAD of lectures, look at ece437 slides

I cannot cover all of ECE437 in this course

- but I will go through the key points of 437 (VERY quickly)

## Announcements

---

Undergraduate students in 565:

- If you are going to grad school, do 565 in grad school
  - you will have time and maturity for this material
  - av. grad student takes 2 courses
- you should do undergrad compilers and OS instead of 565
  - otherwise you will have an incomplete background

## Interactive Lectures

I like my classes to be highly interactive

Please raise your hand and **ask if something is not clear**

No question is stupid or trivial

No confusion is small or unimportant

Don't think everybody else gets it except you

If you didn't get it then probably some others did not either

By asking for clarification, you are helping everyone!

The course builds on itself so if something is not clear please ask **right away** else you will not get later chapters

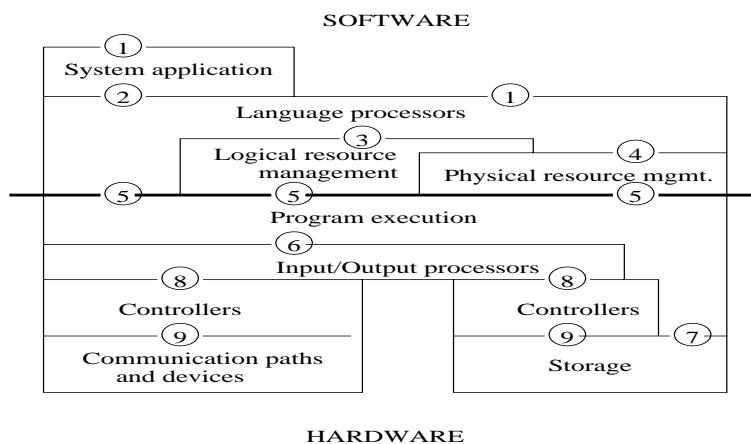
## What is Computer Architecture?

Architecture here means Instruction Set Architecture (ISA)

The term architecture is used here to describe the attributes of a system as seen by the programmer, i.e., the conceptual structure and functional behavior as distinct from the organization of the dataflow and controls, the logic design, and the physical implementation.

- Gene Amdahl, IBM Journal of R&D, Apr 1964
- Instruction set should define WHAT the instructions do in a timing-INdependent manner WITHOUT including the details of HOW hardware implements the instructions

## Levels of Architecture



## Levels of Architecture

Myers, Advances in Computer Architecture, Wiley, 1981

Level 1: System architecture

- interface to outside world (e.g., languages, GUI)

Levels 2, 3, 4: Software architecture

- 2,3: programming language and 3,4: operating system

Level 5: Computer architecture

- interface between hardware and software

Levels 6, 8, 9: Physical I/O and Level 7: Memory architecture

## Architecture, Organization, Implementation

Computer architecture: HW/SW interface

- instruction set
- memory management and protection
- interrupts and traps
- floating-point standard (IEEE)

Organization: also called microarchitecture

- number/location of functional units
- pipeline/cache configuration

Implementation: low-level circuits

## Why Study Computer Architecture

Answer #1: Optimize cost/performance as technology changes

Technology	Annual improvement
Transistor count	25%
Transistor speed	20-25%
DRAM density	60%
DRAM speed	4%
Disk density	25%
Disk speed	4%

## Why Study Computer Architecture

What do these two intervals have in common?

- 1776-2006 (229 years)
- 2007-2009 (2 years)

Absolute speed improvements of computers comparable!!

- i.e., whatever speedup was achieved in 229 years
  - is achieved in 2 years
- If performance improves by 50%,  $1.5^2 = 2.25$

## Why Study Computer Architecture

Answer #2: innovation built-in to performance trends

Initially, transistor counts limited performance

- ~35% performance improvement per year

Now, larger transistor counts enable advanced microarchitectures

- >50% performance improvement per year
- the added growth due to implementation/organization

## Why Study Computer Architecture

---

Data from Borkar's Scaling paper (in the handout)

Microprocessor performance in the last decade

- Transistors 1.41x speedup every generation
- Microprocessor 2x speedup every generation

In ECE565, we will consider

- Features of state-of-the-art processors that enabled this
- AND that will allow this trend to continue

## Why Study Computer Architecture

---

Answer #3: User requirements change rapidly

Previously infeasible solutions become ubiquitous products!

- multimedia
- entertainment
- portable computing
- virtual reality
- probably you will be able to think up of something better

## Role of the Computer Architect

---

To define the interface

- what is above and what is below
- what does each interface look like

Decisions based on

- applications
- performance
- cost
- reliability
- power . . .

## Applications

---

Scientific/numerical - large memory, floating-point arithmetic

- weather prediction, molecular modeling, n-body problems

Commercial - integer arithmetic, high I/O

- inventory control, billing, payroll

Embedded - low power, low cost, interrupt driven,

- automobile engines, door knobs, microwave,

Home computing - high bandwidth data, graphics

- multimedia, games, entertainment

## Classes of Computers

### High performance

- massively parallel - Cray T3E
- servers - SGI Origin

### Balanced cost/performance

- workstations - Sun SPARCstations
- medium servers - SUN Wildfire
- high-end PCs - Pentium quads

### Low cost/power

- low-end PCs, laptops, personal digital assistants (PDAs)

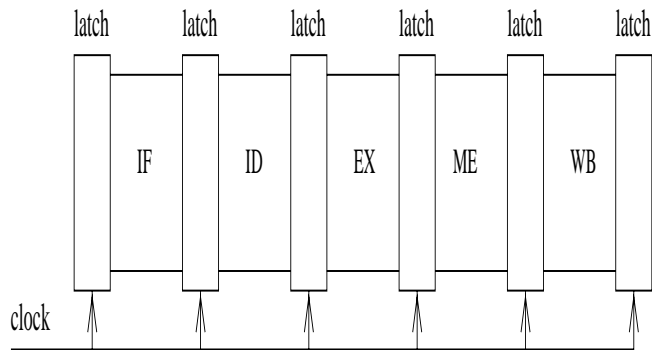
## Performance

### Two basic metrics

- latency - response or execution time
  - time from start to finish
- throughput - bandwidth
- rate of task completion ( = rate of task initiation)
  - =  $1/\text{time between task completions}$
- throughput =  $1/\text{latency}$ 
  - ONLY when there is no overlap
  - with overlap, throughput  $> 1/\text{av.response time}$



## Pipelining (EE437 and more in 565)



## Pipelining

### Up to 1 result per clock cycle

- e.g., lunch buffet

### But performance may be limited by

- unequal segments, latch overhead
- dependence

### Overlap or parallelism (pipelined or parallel) is KEY KEY KEY

- e.g., 10 memory accesses each 100 cycles
- sequential => 1000 cycles but parallel => 100 cycles (avg 10 cycles each) -- 10x faster! - POWER OF PARALLELISM



## Memory Hierarchy (EE437 material)

Principles of performance

- common case fast
- locality of data and instructions
  - temporal locality - access same data in near future
  - spatial locality - access nearby data

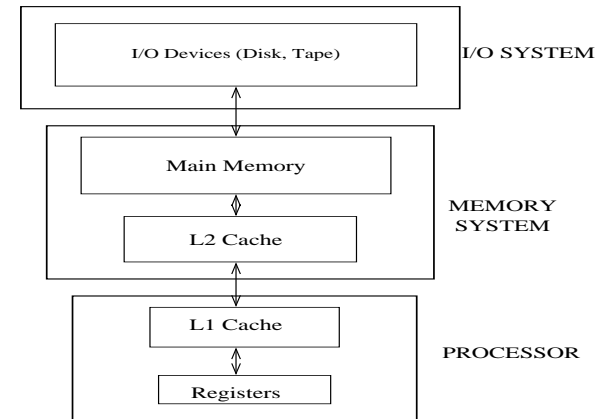
Implementation facts

- on-chip faster than off-chip
- SRAM faster than DRAM faster than disk

Keep recently referenced data close to the processor

## Memory Hierarchy

A hierarchy of successively larger, slower, and cheaper memories



## Memory Hierarchy

Type	Size	Latency	Bandwidth
Register	< 1 KB	0.5 ns	100 GB/s
L1 Cache	< 256 KB	2 ns	20 GB/s
L2 Cache	< 256 MB	20 ns	5 GB/s
Memory	< 64 GB	100 ns	2 GB/s
Disk	> 512 GB	10 ms	100 MB/s

the numbers are a little old but the trends are valid

## Performance and Cost

Performance metrics

- elapsed time vs CPU time

Iron law of performance

Benchmarks and benchmarking

Averaging

Amdahl's law

Balance and bursty behavior

Cost and price

## Performance Metrics

Time (latency)

- elapsed time vs processor time

Rate (bandwidth)

- performance = rate = work per time

Distinction is sometimes blurred

## MIPS: Million Instructions per Second

MIPS =  $\text{instruction count} / (\text{execution time} \times 10^6)$

$$= \text{\#instrs} / (\text{\#clock} * \text{cycle time} \times 10^6) = \text{clock rate} / (\text{CPI} \times 10^6)$$

The problem is in defining a uniform measure of work



- instruction sets are not equivalent
- different programs use different instruction mix
- instruction count is not a reliable indicator of work
  - some optimizations add instructions
  - instructions may have varying work

## Relative MIPS

Relative MIPS =  $\{\text{time}_{\text{reference}} / \text{time}_{\text{new}}\} \times \text{MIPS}_{\text{reference}}$

Relative MIPS (e.g., VAX MIPS) a little better than native MIPS

But very sensitive to reference machine



which compiler and OS?

Bottomline: may be useful if same instruction set/compiler/OS

## MFLOPS: Million Float. Pt. Ops per Second

MFLOPS =  $\{\text{FP ops/program}\} \times \{\text{program/time}\} \times 10^{-6}$

Depends on

- program must be floating-point intensive
- ignores other instructions (e.g., loads and stores)
- in the extreme, some programs have no FP ops

Peak MFLOPS: manufacturer guarantees not to exceed!



## Normalized MFLOPS

Not all machines implement the same FP ops

- Cray does not implement divide
- Motorola has SQRT, SIN, and COS

Not all FP ops are same work

- adds usually faster than divide

Normalized FP: assign a canonical # FP ops to a HLL program

Normalized MFLOPS = {# canonical FP ops / time} x 10<sup>-6</sup>

## Iron Law



Time/program = instrs/program x cycles/instr x sec/cycle

sec/cycle (a.k.a. cycle time, clock time)

- mostly determined by technology and CPU organization

cycles/instr (a.k.a. CPI)

- mostly determined by ISA and CPU organization
- overlap among instructions makes this smaller

instr/program (a.k.a. instruction count)

- instrs executed (NOT static code)
- mostly determined by program, compiler, ISA

## Iron Law - Discussion

Execution time is a product of three factors but that does not mean we can optimize each factor in isolation without looking at the other factors (though it is tempting to do so because there are three separate factors)

- The three factors are interdependent
  - e.g., CPU organization affects both clock speed and CPI
  - ISA affects both CPI and instruction count
- Often, optimizations improve one factor but worsen other factor(s)
- we must consider the product of all three factors when thinking about and evaluating an optimization

## Controversial Example

Some have argued:

- CISC CPU time =  $P \times 8 \times T = 8PT$
- RISC CPU time =  $2P \times 2 \times T = 4PT$
- RISC CPU time = CISC CPU time/2



The truth is much more complex

## Performance Comparison



Machine A is n times faster than machine B iff

- $\text{perf}(A)/\text{perf}(B) = \text{time}(B)/\text{time}(A) = n$

Machine A is x% faster than machine B iff

- $\text{perf}(A)/\text{perf}(B) = \text{time}(B)/\text{time}(A) = 1 + x/100$

E.g., A 10s, B 15s

- $15/10 = 1.5 \Rightarrow$  A is 1.5 times faster than B
- $15/10 = 1 + 50/100 \Rightarrow$  A is 50% faster than B

## Simple Example

Old CPI =  $0.43 + 0.21 + 0.12 \times 2 + 0.24 \times 2 = 1.36$

New CPI =  $0.43 + 0.21 + 0.12 \times 1 + 0.24 \times 2 = 1.24$

Speedup = old time/new time

$$= \{P \times \text{old CPI} \times T\} / \{P \times \text{new CPI} \times 1.15 T\}$$

$$= 1.36 / \{1.24 \times 1.15\} = 0.95$$

Answer: Don't make the change

## Simple Example

Op	Frequency	Cycle count
ALU ops	43%	1
Loads	21%	1
Stores	12%	2
Branches	24%	2

Assume stores can execute in 1 cycle by slowing clock 15%

Should this be implemented?

## Type of Benchmarks

Real programs

- representative of real workload
- only accurate way to characterize performance
- requires considerable work

Kernels

- "representative" program fragments
- good for focussing on individual features not big picture

Mixes

- instruction frequency of occurrence; calculate

## Kernel Example

---

inner product

Do 3 L = 1, LP

Q = 0.0

DO 3 K = 1,N

Q = Q + Z(K)\*X(K)

## Mix Example

---

Gibson Mix - developed in 1950's at IBM

- |                 |     |            |     |
|-----------------|-----|------------|-----|
| • load/store    | 31% | branches   | 17% |
| • fixed add/sub | 6%  | compare    | 4%  |
| • float add/sub | 7%  | float mult | 4%  |
| • float div     | 2%  | fixed mul  | 1%  |
| • fixed div     | <1% | shifts     | 4%  |
| • logical       | 2%  |            |     |

Generally speaking, these numbers are still valid today

## Other Benchmarks

---

Toy benchmarks

- e.g., fibonacci, prime number
- little value

Synthetic benchmarks

- programs intended to give specific mix
- ignore dependences
- may be ok for non-pipelined, non-cached, non-optimizing compilers

## Synthetic Benchmark Example

---

Dhrystone, Whetstone

X = 1.0

Y = 1.0

Z = 1.0

DO 88 I = 1, N8, 1

CALL P3(X,Y,Z)

## Synthetic Benchmark Example

---

SUBROUTINE P3(X,Y,Z)

- COMMON T, T2
- X1 = X
- Y1 = Y
- X1 = T \* (X1 - Y1)
- Y1 = T \* (X1 + Y1)
- Z = (X1 + Y1)/T2
- RETURN

## SPEC 2000 Integer Benchmark

---

- gzip, bzip      compression
- vpr              FPGA place & route
- gcc              compiler
- mcf              min cost flow network
- crafty            chess
- parser            natural language
- perl              perl language interpreter
- vortex            object-oriented database system
- eon                C++ , ray trace

## SPEC 2000 Integer Benchmark

---

- gap              computational group theory
- twolf             place & route

## SPEC2000 Floating Point Benchmark

---

- wupside      chromodynamics
- swim         shallow water model - finite differences
- mgrid        multigrid solver for 3d field
- applu        partial differential equations
- mesa         3D graphics
- galgel        fluid dynamics
- art            image recognition
- aspi         meteorology
- equake       wave propagation
- facerec      image processing

## SPEC2000 Floating Point Benchmark

- ammp computational chemistry
- lucas number theory
- fma3d finite elements
- sixtrack nuclear physics

## SPEC Benchmarks

SPEC benchmarks are useful only if they match your workload

SPEC is used by Intel, Compaq, IBM, etc for their chips

All research papers use SPEC

Throughout this course we will use SPEC

- we will run benchmarks on “cycle-accurate” simulators

## SPEC Benchmarking Process

steps:

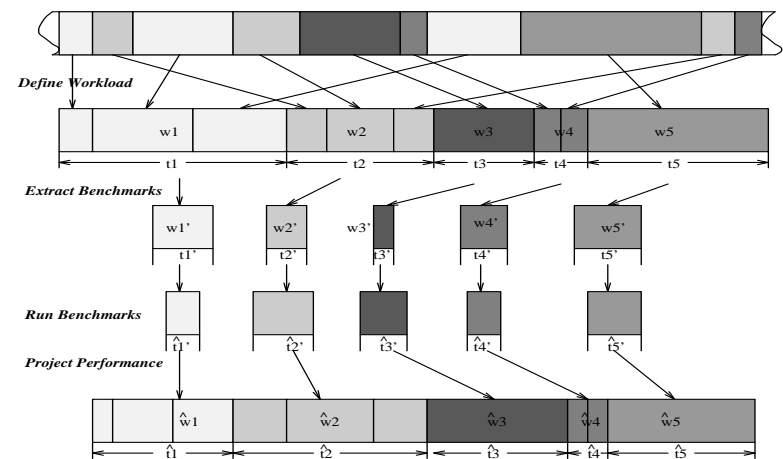
- for each benchmark  $i$ , look up  $T_{base, i}$
- foreach benchmark  $i$ , run target machine to get  $T_{new, i}$

- compute geometric mean: 
$$\sqrt[n]{\prod_{i=1}^n \frac{T_{base, i}}{T_{new, i}}}$$

But

- what workload does this match?
- how does geometric mean predict performance

## SPEC Benchmarking Process



## SPEC Benchmarking Process

Steps:

- extract benchmarks from applications
- choose performance metric
- execute benchmarks on candidate machines
- project performance in new machine

## Alternatives to Benchmarks

Analytical models

- queueing theory based models
- good for isolated components
- useful in back-of-the-envelope calculations
- not easily amenable to entire systems

Prototyping

- high credibility, good for small components
- cannot change parameters easily
- time consuming, expensive

## How to Average

arithmetic mean

Arithmetic mean of times:  $\left\{ \sum_1^n time(i) \right\} / n$  for n programs

Valid only if programs run equally often, so use “weight” factors

Weighted arithmetic mean:  $\left\{ \sum_1^n (weight(i) \times time(i)) \right\} / n$

## Other Averages

Example: what is the average speed?

- 30 mph for first 10 miles
- 90 mph for next 10 miles

Average speed =  $(30+90)/2$

Average speed =

- 
-

## Harmonic Mean

Harmonic mean of rates = 
$$\frac{1}{\left\{ \sum_{i=1}^n \frac{1}{rate(i)} \right\} / n}$$



Use HM if forced to start and end with rates

HM is a trick to do arithmetic mean of times

- but using rates and not times

## Dealing with Ratios

E.g.,

	Machine A	Machine B
Program 1	1	10
Program 2	1000	100

If we take ratios, with respect to Machine A (machine B)

	Machine A	Machine B
Program 1		
Program 2		

## Deaing with Ratios

averages different depending upon which is the reference

geometric mean of ratios = 
$$\sqrt[n]{\prod_{i=1}^n ratio(i)}$$



Use GM if forced to use ratios

Independent of reference machine (math property)

In the example, GM for machine A is 1, for machine B is also 1

- normalized with respect to either machine

## But .....

Geometric mean of ratios is not proportional to total time

AM in example says machine B is 9.1 times faster

GM says they are equal

If we took total execution time, A and B are equal only if

- program 1 is run 100 times more often than program 2

Generally, GM will mispredict for three or more machines



Use AM for time, HM for rates, and GM for ratios

## Pitfalls

Choosing performance from the wrong application space

- e.g., in a realtime environment, choosing MSWord

Choosing benchmarks from no application space

- e.g., synthetic workloads

Using toy benchmarks

- e.g., used to prove the value of RISC in early 80's

Mismatch of benchmark properties with scale of features studied

- e.g., using SPEC for large cache studies

## Pitfalls

Carelessly scaling benchmarks

- truncating benchmarks
- using only first few million instructions
- reducing program data size

Too many easy cases

- may not show value of a feature

Too few easy cases

- may exaggerate importance of a feature

## Amdahl's Law

Speedup = old time/new time = new rate/old rate

Let an optimization speed  $f$  fraction of time by a factor of  $s$

Spdup =  $[(1 - f) + f] \times oldtime / ([ (1 - f) \times oldtime ] + f / s \times oldtime)$



$$= 1 / (1 - f + f / s)$$

## Amdahl's Law: Common case important

Example 1:  $f = 95\%$   $s = 1.10$  - speedup common case

- speedup =  $1 / ((1 - 0.95) + (0.95 / 1.10)) = 1.094$

Example 2:  $f = 5\%$  and  $s = 10$

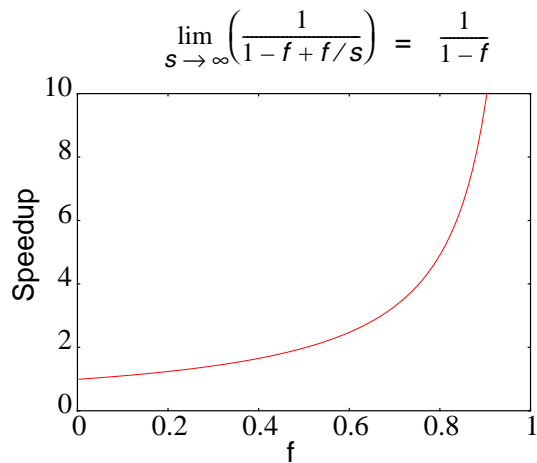
- speedup =  $1 / ((1 - 0.05) + (0.05 / 10)) = 1.047$

Example 3:  $f = 5\%$  and  $s \rightarrow \infty$

- speedup =  $1 / ((1 - 0.05) + (0.05 / \infty)) = 1.052$

These examples illustrate that common case should be sped up

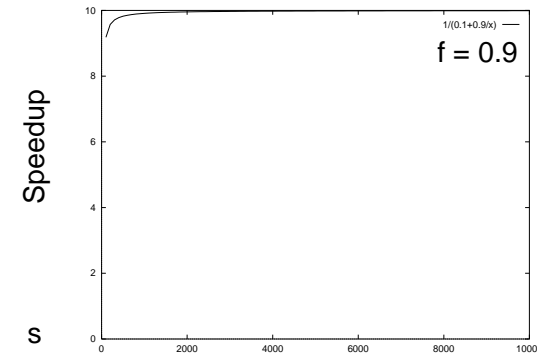
## Amdahl's Law: Make common case fast



## Amdahl's Law: don't ignore uncommon case

Amdahl was talking about a parallel processor with large speedup

At some point you have to pay attention to the serial part



## Balance

At a system level, bandwidths and capacities should be balanced

Each level capable of demanding/supplying bandwidths

Refer to memory hierarchy figure

## Balance: Example

Example:

- IPC = 1.5 (IPC is 1/CPI)
- 0.3 loads and stores
- 0.9 data cache hit rate
- 0.95 icache hit rate
- all cache misses require 32 bytes

So processor demand is:

- $1.5 * 0.3 * 0.10 * 32$  (data) +  $1.5 * 1.0 * 0.05 * 32$  (instrs)
- = 3.8 bytes/clock (memory supply rate to keep CPU busy)

## Balance

If the demand bandwidth = supply bandwidth of a resource

- then the computation is that resource-bound

E.g., if memory bandwidth = processor demand for program P

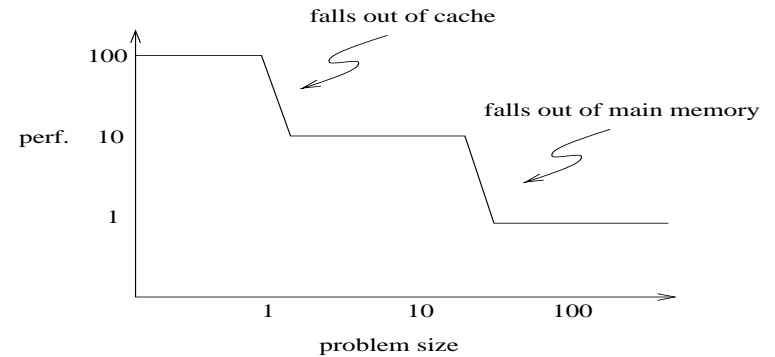
- then P is said to be memory-bound

We can similarly define CPU-bound, disk-bound or I/O bound

Goal: to be bound everywhere (why?)

## Memory Bandwidth

- copy:  $a[i] = b[i]$     scale:  $a[i] = q \cdot b[i]$
- sum:  $a[i] = b[i] + c[i]$     saxpy:  $a[i] = b[i] + q \cdot c[i]$



## Memory Bandwidth

Memory bandwidth of older systems (MB/s)

System	copy	scale	sum	saxpy
Cray C90	7000	7000	9400	9500
Alpha 150Mhz	98	90	68	90

## Memory Bandwidth

HP GS1280 Alpha21364 (1.2GHZ) server -  
SAXPY,copy, scale,sum

- memory bandwidth scales linearly with number of cores!
- 1 core: ~5.5 GB/s
- 64 cores: 350 GB/s

AMD Opteron (desktop) - Block read:

- fits in L1: 20GB/s
- fits in L2: 10GB/s and off-chip (memory): 3 GB/s
- modern GPUs can achieve 2-3x more bandwidth

---

## Balance (again)

---

Storage capacity and bandwidth requirements

- e.g., large cache => higher hit rate => lower demand
- Or large memory => less paging => lower I/O demand

Amdahl's rule:

- 1 MIPS  $\Leftrightarrow$  1 MB memory  $\Leftrightarrow$  1 Mbits/s I/O
- if corrected to 1 Mbytes/s of I/O, the rule is still good!

---

## Bursty Behavior

---

To get 2 IPC how many instructions should you -

- fetch per cycle?
- issue per cycle?
- complete per cycle?
- the answer is NOT 2

Instructions are not like sand where peaks and valleys can be leveled

---

## Bursty Behavior

---

E.g.,

- $a = b + c$
- $d = e * f$

Dependencies will cause pipeline stalls (or bubbles or wait times)

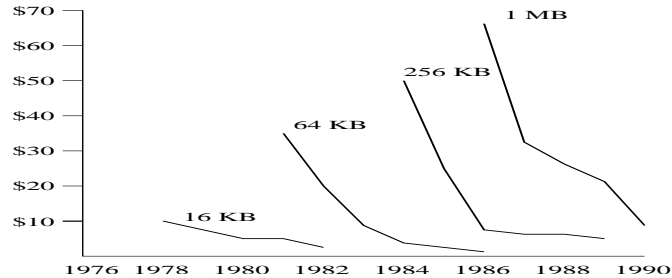
So sometimes pipeline will be full and at other only partially full

=> higher PEAK is needed for desired AVERAGE performance

## Cost

Cost is very important to most real designs

- cost changes over time
- learning curve lowers manufacturing costs
- technology improvements lower costs e.g., DRAM



## IC Cost

$$\text{cost (IC)} = \frac{\text{cost}(die) + \text{cost}(\text{testing}) + \text{cost}(\text{packaging})}{\text{yield}}$$

$$\text{cost (die)} = \frac{\text{cost}(wafer)}{(\text{die}/wafer) \times \text{yield}(die)}$$

$$\text{yield (die)} = \text{yield}(wafer) \times \left\{ 1 + \frac{\text{defects}/\text{cm}^2 \times \text{area}}{\alpha} \right\}^{-\alpha}$$

often  $\alpha$  is 0.30

$$\text{cost (die)} = f(\text{die area}^4)$$

## Cost Breakdown

Component cost

- microprocessor, SRAM, DRAM + disk
- power supplies, packaging

Direct costs

- manufacturing (labor, scrap) + warranty

Indirect costs

- R&D + marketing
- administrative
- profits + taxes

## Price

Only loosely related to cost!

- start with component cost
- add 25-40% for direct cost
- add 45-65% gross margin
- = average selling price
- add 60-75% to correct discounts and allow dealer profits
- = list price

Note

- component cost - 15-30%, R&D - 8-15% of list price

## BGvN46: Concepts

---

Classic paper

- most observations are still true
- most historians credit Eckert and Mauchly for this idea

“It is evident that the machine must be capable of storing in some manner not only the data but also the instructions which govern the actual machine.”

“Conceptually we have discussed above two different forms of memory: storage of numbers and storage of orders the memory organ can be used to store both numbers and orders.”

## BGvN46: Arithmetic

---

Binary arithmetic

Two's complement

Iterative carry

Iterative multiply (carry-save adders) (see EE437 notes)

Rounding vs. jamming (see EE437 notes)

Non-restoring division (see EE437 notes)

No floating-point. Why?

## BGvN46: Control

---

40-bit data

20-bit instructions

- 8-bit opcode
- 12-bit addresses

Basic instructions

- conditional and unconditional branches
- data transfer
- ALU and shift
- store into orders - why?