

ECE 252 / CPS 220 Advanced Computer Architecture I

Fall 2007
Duke University

Prof. Daniel Sorin (sorin@ee.duke.edu)

based on slides developed by

Prof. Roth (Penn), Hill, Wood, Sohi, Smith,
Lipasti (Wisconsin), and Vijaykumar (Purdue)

Administrivia

- addresses, email, website, etc.
- list of topics
- expected background
- course requirements
- grading and academic misconduct

Instructors

professor: Dan Sorin (sorin@ee.duke.edu)

- research: fault-tolerant computer architecture, verification-aware microprocessor design, processors that tolerate CMOS process variability, plus other topics
- teaching: architecture (152, 252, 259), fault-tolerance (254)
- office: 209C Hudson Hall
- office hours: TBD

TA: Albert Meixner (albert@cs.duke.edu)

- PhD student finishing up dissertation on fault-tolerant computer architectures
- office: D311 LSRC
- office hours: TBD

Where to Get Answers

Consult course resources in this order:

- Course Website (<http://www.ee.duke.edu/~sorin/ece252>)
- TA: Albert Meixner (albert@cs.duke.edu)
- Professor Sorin

Email to TA and Professor must have subject that begins with
ECE252

- Otherwise I can't promise it won't end up in spam folder

What is This Course All About?

State-of-the-art computer hardware design

Topics

- Uniprocessor architecture (i.e., microprocessors)
- Memory architecture
- Multithreading
- Multicore processors

Fundamentals, current systems, and future systems

Will read from: classic papers, brand-new papers, textbook

Course Goals and Expectations

Course Goals

- + Understand how current processors work
- + Understand how to evaluate/compare processors
- + Learn how to use simulator to perform experiments
- + Learn research skills by performing term project
- + Learn how to critically read research papers

Course expectations:

- Will loosely follow textbook
- Major emphasis on cutting-edge issues
- Students will read a list of research papers
- Term project

What You Should Expect from Course

Things NOT to expect in this course:

- 100% of class = me lecturing to you
- Homework sets and exams where every question is either quantitative or has a single correct answer

Things to expect in this course:

- Active discussions/arguments about architecture ideas
- Essay questions
- Being asked to explain, discuss, defend, and argue
- Questions with multiple possible answers

What I Expect You to Know Already

Courses you should have taken already

- Basic architecture (ECE 152 / CPS 104 or equivalent)
- Programming (our simulator is in C)
- Basic OS (ECE 153 / CPS 110) — not critical, but helpful

Topics you should remember fondly - I will not cover these in any detail in this course

- Instruction sets, computer arithmetic, assembly programming, caches, memory, virtual memory, I/O

Course Components

Reading Materials

- *Computer Architecture: A Quantitative Approach* by Hennessy and Patterson, **4th Edition**
- **(optional)** *Modern Processor Design* by Shen and Lipasti
- Recent research papers (on course website)

Homework

- 4 to 6 homework assignments, performed in groups

Term Project

- Groups of 2 or 3 (or individual, if you prefer)

Exams

- Midterm and final exam

Term Project

This is a semester-long research project

- Do not expect to do whole thing in last week, because $E[\text{project grade}] < B$
- I will suggest a bunch of possible project ideas, but many students choose to pursue their own ideas
- Project proposals due Monday, October 15

You may “combine” this project with a project from another class, but you **MUST** consult with me first

You must absolutely, positively reference prior work

- Please ask me if you have ANY questions
- Not knowing != valid excuse

Grading

Grading breakdown

- Homework: 30%
- Midterm: 15%
- Project: 25%
- Final: 30%

Academic Integrity and Late Policy

Academic Misconduct

- **University policy will be followed strictly**
- **Zero tolerance for cheating and/or plagiarism**

Late policy

- Late homeworks (except for dean's excuses)
 - **late <1 day = 50% off**
 - **late >1 day = zero**
- No late term project will be accepted. Period.

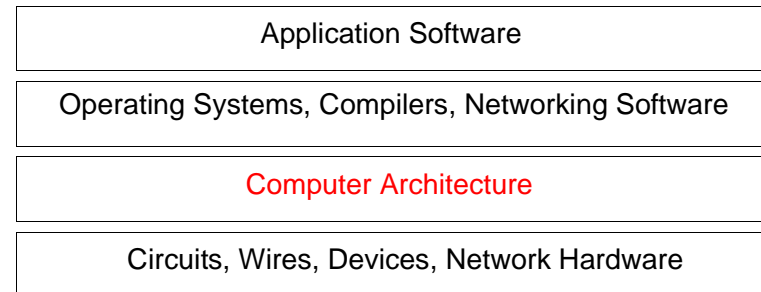
Now, moving on to computer architecture ...

What is Computer Architecture?

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

Architecture and Other Disciplines



Architecture interacts with many other fields

- Can't be studied in a vacuum

Levels of Computer Architecture

architecture

- functional appearance to software
 - opcodes, addressing modes, architected registers

microarchitecture (= focus of this course)

- logical structure that implements the architecture
 - pipelining, functional units, caches, physical registers

realization (circuits)

- physical structure that embodies the implementation
 - gates, cells, transistors, wires

The Role of the Microarchitect

architect: defines the hardware/software interface

microarchitect: defines the hardware implementation

- usually the same person as the architect

Two very important questions in this course:

What goals are we (microarchitects!) trying to achieve?

And what units do we use to measure our success?

Hint: how do you decide which computer to buy?

- desktop? laptop? mp3 player?
- is a Pentium4 box better/worse than an iMac?

Applications -> Requirements -> Designs

- **scientific**: weather prediction, molecular modeling
 - need: large memory, floating-point arithmetic
 - examples: CRAY XT4, IBM BlueGene/L
- **commercial**: inventory, payroll, web serving, e-commerce
 - need: integer arithmetic, high I/O
 - examples: SUN SPARCcenter, Enterprise, AlphaServer GS320
- **desktop**: multimedia, games, entertainment
 - need: high data bandwidth, graphics
 - examples: Intel CoreDuo, AMD Opteron Dual Core, IBM Power6
- **mobile**: laptops
 - need: low power (battery), good performance
 - examples: Intel Mobile Pentium 4, Transmeta TM5400
- **embedded**: cell phones, automobile engines, door knobs
 - need: low power (battery + heat), low cost
 - examples: Compaq/Intel StrongARM, X-Scale, Transmeta TM3200

Why Study Computer Architecture?

answer #1: requirements are always changing

aren't computers fast enough already?

- are they?
- fast enough to do everything we will EVER want?
 - AI, virtual reality, protein sequencing, ...
- "if you build it, they will come"

is speed the only goal?

- power: heat dissipation + battery life + utility bill
- cost
- reliability
- etc.

Why Study Computer Architecture?

answer #2: technology playing field is always changing

- annual technology improvements (approximate)
 - SRAM (logic): density +25%, speed +20%
 - DRAM (memory): density + 60%, speed: + 4%
 - disk (magnetic): density +25%, speed: + 4%
 - network interface: 10 Mb/s -> 100 Mb/s -> 1 Gb/s -> 10 GB/s -> ?
- parameters change *and* change relative to one another!
 - and that's not even including "exotic" nanotechnologies

designs change even if requirements fixed

... but requirements are not fixed

Examples of Changing Designs

example I: caches

- 1970: 10K transistors, DRAM faster than logic -> bad idea
- 1990: 1M transistors, logic faster than DRAM -> good idea
- will caches ever be a bad idea again?

example II: out-of-order execution

- 1985: 100K transistors + no precise interrupts -> bad idea
- 1995: 2M transistors + precise interrupts -> good idea
- 2005: 500M transistors + 4GHz clock -> bad idea?
- 2007: 1B transistors + multiple cores -> ???

semiconductor technology is an incredible driving force

Moore's Law

“Cramming More Components onto Integrated Circuits”

–G.E. Moore, Electronics, 1965

- observation: (DRAM) transistor density doubles annually
 - became known as “Moore’s Law”
 - wrong—density doubles every 18 months (had only 4 data points)
- corollaries
 - cost per transistor halves annually (18 months)
 - power per transistor decreases with scaling
 - speed increases with scaling
 - reliability starting to decrease with scaling

Moore's Law

“performance doubles every 18 months”

- common interpretation of Moore’s Law, not original intent
- wrong! “performance” used to double every ~2 years
- self-fulfilling prophecy (Moore’s Curve)
 - 2X every 2 years = ~3% increase per month
 - 3% per month used to judge performance features
 - if feature adds 9 months to schedule...
 - ...it should add at least 30% to performance ($1.03^9 = 1.30 \rightarrow 30\%$)
 - e.g., Intel Itanium: under Moore’s Curve in a big way

performance improvements have slowed down in past few years

- architects haven’t figured out how to use the extra transistors to improve performance of single core without melting the chip

Evolution of Single-Chip Processors

	1971–1980	1981–1990	1991–2000	2010
Transistor Count	10K–100K	100K–1M	1M–100M	5B?
Clock Frequency	0.2–2MHz	2–20MHz	20M–1GHz	4GHz?
IPC	< 0.1	0.1–0.9	0.9–2.0	2.0?
MIPS/MFLOPS	< 0.2	0.2–20	20–2,000	100,000?
Number of cores	1	1	1	64?

some perspective: 1971–2001 performance improved 35,000X!!!

- what if cars improved at this rate?
 - 1971: 60 MPH & 10 MPG, 2001: 2,100,000 MPH & 350,000 MPG
- but... what if cars crashed as often as computers did?

Performance

Much of the focus of this course is on improving performance

Topics:

- performance metrics
- CPU performance equation
- benchmarks and benchmarking
- reporting averages
- Amdahl’s Law
- Little’s Law
- concepts
 - balance
 - tradeoffs
 - bursty behavior (average and peak performance)

Readings

Hennessy & Patterson

- Chapter 1

R. P. Colwell et al. "Instruction Sets and Beyond: Computers, Complexity, and Controversy." IEEE Computer, 18(9), 1996.

Performance Metrics

latency: response time, execution time

- good metric for fixed amount of work (minimize time)

throughput: bandwidth, work per time, "performance"

- = $(1 / \text{latency})$ when there is NO OVERLAP
- > $(1 / \text{latency})$ when there is overlap
 - in real processors, there is always overlap (e.g., pipelining)
- good metric for fixed amount of time (maximize work)

comparing performance

- A is N times faster than B iff
 - $\text{perf}(A)/\text{perf}(B) = \text{time}(B)/\text{time}(A) = N$
- A is X% faster than B iff
 - $\text{perf}(A)/\text{perf}(B) = \text{time}(B)/\text{time}(A) = 1 + X/100$

Performance Metric I: MIPS

MIPS (millions of instructions per second)

- $(\text{instruction count} / \text{execution time in seconds}) \times 10^{-6}$
- instruction count is not a reliable indicator of work
 - Prob #1: some optimizations add instructions
 - #2: work per instruction varies (FP mult >> register move)
 - #3: instruction sets are not equal (3 Pentium instrs != 3 Alpha instrs)
- may vary inversely with actual performance
- not a good metric for multicore chips

Performance Metric II: MFLOPS

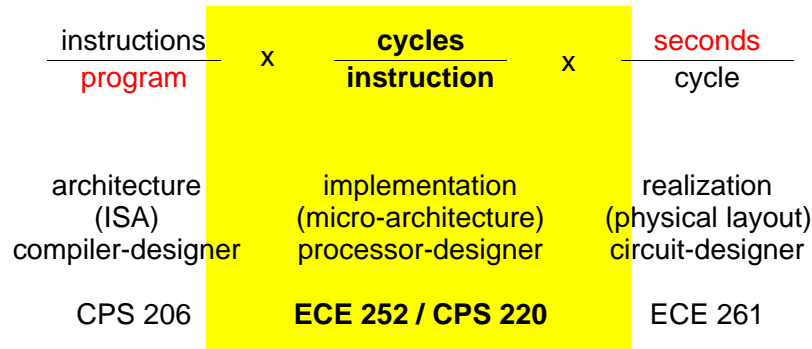
MFLOPS (millions of floating-point operations per second)

- $(\text{FP ops} / \text{execution time}) \times 10^{-6}$
- like MIPS, but counts only FP operations
 - FP ops can't be optimized away (problem #1 from MIPS)
 - FP ops have longest latencies anyway (problem #2)
 - FP ops are the same across machines (problem #3)
- may have been valid in 1980 (most programs were FP)
 - most programs today are "integer" i.e., light on FP (#1)
 - load from memory takes longer than FP divide (#2)
 - Cray doesn't implement divide, Motorola has SQRT, SIN, COS (#3)

CPU Performance Equation

processor performance = **seconds / program**

- separate into three components (for single core)



CPU Performance Equation

instructions / program: *dynamic* instruction count

- mostly determined by program, compiler, ISA

cycles / instruction: CPI

- mostly determined by ISA and CPU/memory organization

seconds / cycle: cycle time, clock time, 1 / clock frequency

- mostly determined by technology and CPU organization

uses of CPU performance equation

- high-level performance comparisons
- back of the envelope calculations
- helping architects think about compilers and technology

CPU Performance Comparison

famous example: "RISC Wars" (RISC vs. CISC)

- assume
 - instructions / program: CISC = P, RISC = 2P
 - CPI: CISC = 8, RISC = 2
 - T = clock period for CISC and RISC (assume they are equal)
- CISC time = $P \times 8 \times T = 8PT$
- RISC time = $2P \times 2 \times T = 4PT$
- RISC time = CISC CPU time/2

the truth is much, much, much more complex

- actual data from IBM AS/400 (CISC -> RISC in 1995):
 - CISC time = $P \times 7 \times T = 7PT$
 - RISC time = $3.1P \times 3 \times T/3.1 = 3PT$ (+1 tech. gen.)

CPU Back-of-the-Envelope Calculation

base machine

- 43% ALU ops (1 cycle), 21% loads (1 cycle), 12% stores (2 cycles), 24% branches (2 cycles)
 - note: pretending latency is 1 because of pipelining

Q: should 1-cycle stores be implemented if it slows clock 15%?

- old CPI = $0.43 + 0.21 + (0.12 \times 2) + (0.24 \times 2) = 1.36$
- new CPI = $0.43 + 0.21 + 0.12 + (0.24 \times 2) = 1.24$
- speedup = $(P \times 1.36 \times T) / (P \times 1.24 \times 1.15T) = 0.95$

Answer: NO!

Actually Measuring Performance

how are execution-time & CPI *actually* measured?

- execution time: time (Unix cmd): wall-clock, CPU, system
- CPI = CPU time / (clock frequency * # instructions)
- more useful? **CPI breakdown** (compute, memory stall, etc.)
 - so we know what the performance problems are (what to fix)

measuring CPI breakdown

- hardware event counters (built into core)
 - calculate CPI using instruction frequencies/event costs
- cycle-level microarchitecture simulator (e.g., SimpleScalar)
 - + measure exactly what you want
 - model microarchitecture faithfully (at least parts of interest)
- method of choice for many architects (yours, too!)

Benchmarks and Benchmarking

“program” as unit of work

- millions of them, many different kinds, which to use?

benchmarks

- standard programs for measuring/comparing performance
 - + **represent programs people care about**
 - + **repeatable!!**
- benchmarking process
 - define workload
 - extract benchmarks from workload
 - execute benchmarks on candidate machines
 - project performance on new machine
 - run workload on new machine and compare
 - not close enough -> repeat

Let's Choose Some Benchmarks!

What benchmarks would you put in your benchmark suite?

Benchmarks: Toys, Kernels, Synthetics

toy benchmarks: little programs that no one really runs

- e.g., fibonacci, 8 queens
- little value, what real programs do these represent?
- scary fact: used to prove the value of RISC in early 80's

kernels: important (frequently executed) pieces of real programs

- e.g., Livermore loops, Linpack (inner product)
- + good for focusing on individual features not big picture
- over-emphasize target feature (for better or worse)

synthetic benchmarks: programs made up for benchmarking

- e.g., Whetstone, Dhrystone
- toy kernels++, which programs do these represent?

Benchmarks: Real Programs

real programs

- + only accurate way to characterize performance
- requires considerable work (porting)

Standard Performance Evaluation Corporation (SPEC)

- <http://www.spec.org>
- collects, standardizes and distributes benchmark suites
- consortium made up of industry leaders
- SPEC CPU (CPU intensive benchmarks)
 - SPEC89, SPEC92, SPEC95, SPEC2000, SPEC2006
- other benchmark suites
 - SPECjvm, SPECmail, SPECweb, SPECComp

Other benchmark suite examples: TPC-C, TPC-H for databases

SPEC CPU2006

12 integer programs (C, C++)

- gcc (compiler), perl (interpreter), hmmer (markov chain)
- bzip2 (compress), go (AI), sjeng (AI)
- libquantum (physics), h264ref (video)
- omnetpp (simulation), astar (path finding algs)
- xalanc (XML processing), mcf (network optimization)

17 floating point programs (C, C++, Fortran)

- *fluid dynamics*: bwaves, leslie3d, ibm
- *quantum chemistry*: gamess, tonto
- *physics*: milc, zeusmp, cactusADM
- gromacs (biochem)
- namd (bio, molec dynamics), dealII (finite element analysis)
- soplex (linear programming), povray (ray tracing)
- calculix (mechanics), GemsFDTD (computational E&M)
- wrf (weather), sphinx3 (speech recognition)

Benchmarking Pitfalls

- benchmark properties mismatch with features studied
 - e.g., using SPEC for large cache studies
- careless scaling
 - using only first few million instructions (initialization phase)
 - reducing program data size
- choosing performance from wrong application space
 - e.g., in a realtime environment, choosing gcc
- using old benchmarks
 - “benchmark specials”: benchmark-specific optimizations

Benchmarks must be continuously maintained and updated!

Reporting Average Performance

averages: one of the things architects frequently get wrong

+ pay attention now and you won't get them wrong

important things about averages (i.e., means)

- **ideally proportional to execution time (ultimate metric)**
 - Arithmetic Mean (AM) for times
 - Harmonic Mean (HM) for rates (IPCs)
 - Geometric Mean (GM) for ratios (speedups)
- **there is no such thing as the average program**
- **use average when absolutely necessary**

What Does The Mean Mean?

arithmetic mean (AM): average execution times of N programs

- $\sum_{1..N}(time(i)) / N$

harmonic mean (HM): average IPCs of N programs

- arithmetic mean cannot be used for rates (e.g., IPCs)
 - 30 MPH for 1 mile + 90 MPH for 1 mile != avg. 60 MPH
- $N / \sum_{1..N}(1 / rate(i))$

geometric mean (GM): average speedups of N programs

- $N \sqrt[N]{\prod_{1..N}(speedup(i))}$

what if programs run at different frequencies within workload?

- “weighting”
- weighted AM = $(\sum_{1..N} w(i) * time(i)) / N$

Amdahl's Law

“Validity of the Single-Processor Approach to Achieving Large-Scale Computing Capabilities” –G. Amdahl, AFIPS, 1967

- let optimization speed up fraction f of program by factor s
 - $speedup = old / ((1-f) * old) + f/s * old) = 1 / (1 - f + f/s)$

- $f = 95\%$, $s = 1.1 \rightarrow 1 / [(1-0.95) + (0.95/1.1)] = 1.094$
- $f = 5\%$, $s = 10 \rightarrow 1 / [(1-0.05) + (0.05/10)] = 1.047$
- $f = 5\%$, $s = \infty \rightarrow 1 / [(1-0.05) + (0.05/\infty)] = 1.052$
- $f = 95\%$, $s = \infty \rightarrow 1 / [(1-0.95) + (0.95/\infty)] = 20$

make common case fast, but...

...uncommon case eventually limits performance

GM Weirdness

what about averaging ratios (speedups)?

- HM / AM change depending on which machine is the base

	machine A	machine B	B/A	A/B
Program1	1	10	10	0.1
Program2	1000	100	0.1	10
		AM	$(10+.1)/2 = 5.05$ B is 5.05 times faster!	$(.1+10)/2 = 5.05$ A is 5.05 times faster!
		HM	$2/(1/10+1/.1) = 5.05$ B is 5.05 times faster!	$2/(1/.1+1/10) = 5.05$ A is 5.05 times faster!
		GM	$\sqrt{(10*.1)} = 1$	$\sqrt{(.1*10)} = 1$

- geometric mean of ratios is not proportional to total time!
 - if we take total execution time, B is 9.1 times faster
 - GM says they are equal

Little's Law

Key Relationship between latency and bandwidth:

Average number in system = arrival rate * mean holding time

Possibly the most useful equation I know

- Useful in design of computers, software, industrial processes, etc.

Example:

- How big of a wine cellar should we build?
- We drink (and buy) an average of 2 bottles per week
- On average, we want to age the wine for 5 years
- bottles in cellar = 2 bottles/week * 52 weeks/year * 5 years
 - = 520 bottles

System Balance

each system component produces & consumes data

- make sure data supply and demand is balanced
- $X \text{ demand} \geq X \text{ supply} \Rightarrow$ computation is “*X-bound*”
 - e.g., memory bound, CPU-bound, I/O-bound
- goal: be bound everywhere at once (why?)
- X can be bandwidth or latency
 - X is bandwidth \Rightarrow buy more bandwidth
 - X is latency \Rightarrow much tougher problem

Tradeoffs

“Bandwidth problems can be solved with money. Latency problems are harder, because the speed of light is fixed and you can’t bribe God” – David Clark

well...

- can convert some latency problems to bandwidth problems
- solve those with money
- the famous “bandwidth/latency tradeoff”

- architecture is the art of making tradeoffs

Bursty Behavior

Q: to sustain 2 IPC... how many instructions should processor be able to

- fetch per cycle?
- execute per cycle?
- complete per cycle?

A: NOT 2 (more than 2)

- dependences will cause stalls (under-utilization)
- if desired performance is X, peak performance must be $> X$

programs don’t always obey “average” behavior

- can’t design processor only to handle average behavior

Performance in the Real World

A paper comparing performance of RISC vs. CISC and trying to show that RISC is not obviously better

- “Instruction Sets and Beyond: Computers, Complexity, and Controversy” by Colwell et al., IEEE Computer 1986.

Roadmap for Rest of Semester

Primary topics for rest of course

- Pipelined processors
- Multiple-issue (superscalar), in-order processors
- Hardware managed out-of-order instruction execution
- Static (compiler) instruction scheduling, VLIW, EPIC
- Advanced cache/memory issues
- Multithreaded processors
- Intro to multicore chips and multi-chip multiprocessors

Advanced topics

- Power-efficiency, fault tolerance, security, virtual machines, grid processors, nanocomputing

Topics NOT covered in this course

These topics have been well-covered in your prior courses

- Digital logic design
- Computer arithmetic
- Instruction sets
- Cache/memory basics, including virtual memory
- I/O (disks, etc.)

If you're uncomfortable with these topics, please see me

Other Courses I'd Recommend

Topics related to this course (non-exhaustive list!)

- Advanced comp. architecture II (ECE 259/CPS 221)
- VLSI design (ECE 261)
- Fault tolerant computing (ECE 254/CPS 225)
- Performance/reliability analysis (ECE 255/257)
- Advanced digital system design (ECE 251)
- Operating systems (CPS 210)
- Compilers (offered at UNC or NC State)