

— Course Description —

Computer Architecture Mini-Course

Instructor: Prof. Milo Martin

Course Description

This three day mini-course is a broad overview of computer architecture, motivated by trends in semiconductor manufacturing, software evolution, and the emergence of parallelism at multiple levels of granularity. The first day discusses technology trends (including brief coverage of energy/power issues), instruction set architectures (for example, the differences between the x86 and ARM architectures), and memory hierarchy and caches. The second day focuses on core micro-architecture, including pipelining, instruction-level parallelism, superscalar execution, and dynamic (out-of-order) instruction scheduling. The third day touches upon data-level parallelism and overviews multicore chips.

The course is intended for software or hardware engineers with basic knowledge of computer organization (such as binary encoding of numbers, basic boolean logic, and familiarity with the concept of an assembly-level “instruction”). The material in this course is similar to what would be found in an advanced undergraduate or first-year graduate-level course on computer architecture. The course is well suited for: (1) software developers that desire more “under the hood” knowledge of how chips execute code and the performance implications thereof or (2) lower-level hardware/SoC or logic designers that seek understanding of state-of-the-art high-performance chip architectures.

The course will consist primarily of lectures, but it also includes three out-of-class reading assignments to be read before each day of class and discussed during the lectures.

Course Outline

Below is the the course outline for the three day course (starting 10am on the first day and ending at 5pm on the third day). The exact topics and order is tentative and subject to change.

Day 1: “Foundations & Memory Hierarchy”

- Introduction, motivation, & “What is Computer Architecture”
- Instruction set architectures
- Transistor technology trends and energy/power implications
- Memory hierarchy, caches, and virtual memory (two lectures)

Day 2: “Core Micro-Architecture”

- Pipelining
- Branch prediction
- Superscalar
- Hardware instruction scheduling (two lectures)

Day 3: “Multicore & Parallelism”

- Multicore, coherence, and consistency (two lectures)
- Data-level parallelism
- Wrapup

Instructor and Bio

Prof. Milo Martin

Dr. Milo Martin is an Associate Professor at University of Pennsylvania, a private Ivy-league university in Philadelphia, PA. His research focuses on making computers more responsive and easier to design and program. Specific projects include computational sprinting, hardware transactional memory, adaptive cache coherence protocols, memory consistency models, hardware-aware verification of concurrent software, and hardware-assisted memory-safe implementations of unsafe programming language. Dr. Martin has published over 40 papers which collectively have received over 2500 citations. Dr. Martin is a recipient of the NSF CAREER award and received a PhD from the University of Wisconsin-Madison.

Computer Architecture

Mini-Course

March 2013

Prof. Milo Martin

Day 1 of 3

[spacer]

Computer Architecture

Unit 1: Introduction

Slides developed by Milo Martin & Amir Roth at the University of Pennsylvania
with sources that included University of Wisconsin slides
by Mark Hill, Guri Sohi, Jim Smith, and David Wood

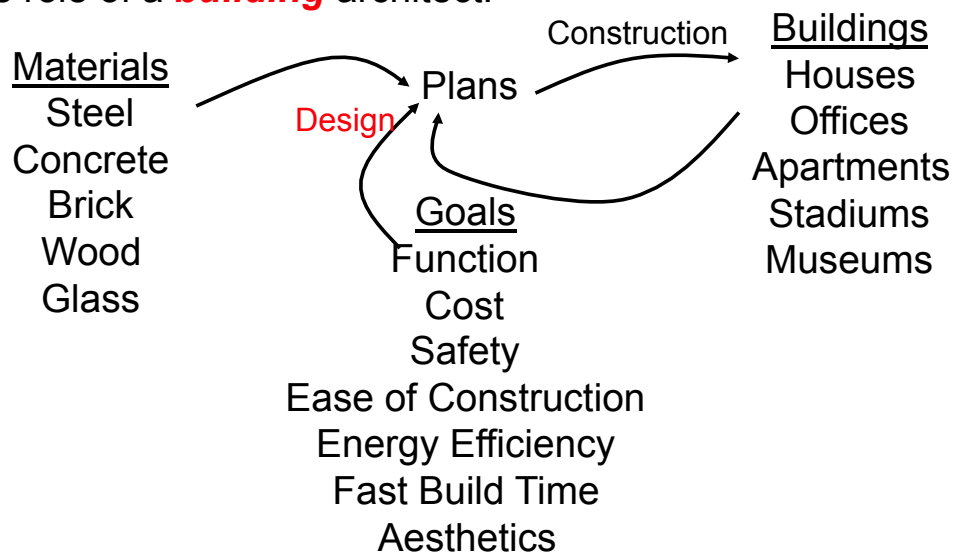
What is Computer Architecture?

What is Computer Architecture?

- “*Computer Architecture* is the science and art of selecting and interconnecting hardware components to create computers that meet functional, performance and cost goals.” - WWW Computer Architecture Page
- An analogy to architecture of buildings...

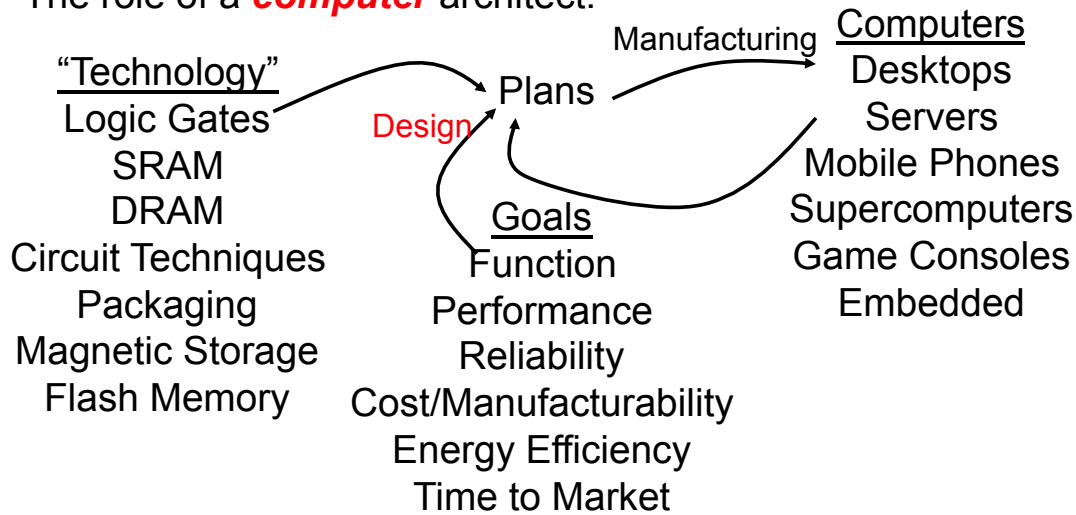
What is ~~Computer~~ Architecture?

The role of a **building** architect:



What is Computer Architecture?

The role of a **computer** architect:



Important differences: age (~60 years vs thousands), rate of change, automated mass production (magnifies design)

Computer Architecture Is Different...

- Age of discipline
 - 60 years (vs. five thousand years)
- Rate of change
 - All three factors (technology, applications, goals) are changing
 - Quickly
- Automated mass production
 - Design advances magnified over millions of chips
- Boot-strapping effect
 - Better computers help design next generation

Design Goals & Constraints

- **Functional**
 - Needs to be correct
 - And unlike software, difficult to update once deployed
 - What functions should it support (Turing completeness aside)
- **Reliable**
 - Does it *continue* to perform correctly?
 - Hard fault vs transient fault
 - Google story - memory errors and sun spots
 - Space satellites vs desktop vs server reliability
- **High performance**
 - "Fast" is only meaningful in the context of a set of important tasks
 - Not just "Gigahertz" – truck vs sports car analogy
 - Impossible goal: fastest possible design for all programs

Design Goals & Constraints

- **Low cost**
 - Per unit manufacturing cost (wafer cost)
 - Cost of making first chip after design (mask cost)
 - Design cost (huge design teams, why? Two reasons...)
 - (Dime/dollar joke)
- **Low power/energy**
 - Energy in (battery life, cost of electricity)
 - Energy out (cooling and related costs)
 - Cyclic problem, very much a problem today
- **Challenge: balancing the relative importance of these goals**
 - And the balance is constantly changing
 - No goal is absolutely important at expense of all others
 - Our focus: *performance*, only touch on cost, power, reliability

Shaping Force: Applications/Domains

- Another shaping force: **applications** (usage and context)
 - Applications and application domains have different requirements
 - Domain: group with similar character
 - Lead to different designs
- **Scientific**: weather prediction, genome sequencing
 - First computing application domain: naval ballistics firing tables
 - Need: large memory, heavy-duty floating point
 - Examples: CRAY T3E, IBM BlueGene
- **Commercial**: database/web serving, e-commerce, Google
 - Need: data movement, high memory + I/O bandwidth
 - Examples: Sun Enterprise Server, AMD Opteron, Intel Xeon

More Recent Applications/Domains

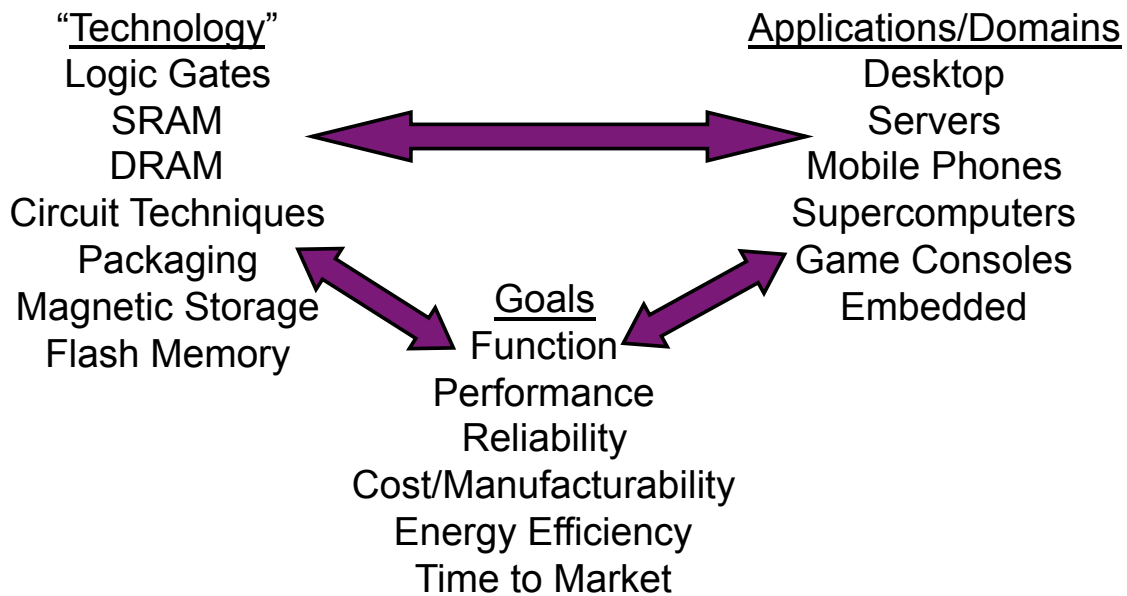
- **Desktop**: home office, multimedia, games
 - Need: integer, memory bandwidth, integrated graphics/network?
 - Examples: Intel Core 2, Core i7, AMD Athlon
- **Mobile**: laptops, mobile phones
 - Need: **low power**, integer performance, integrated wireless
 - Laptops: Intel Core 2 Mobile, Atom, AMD Turion
 - Smaller devices: ARM chips by Samsung, Qualcomm; Intel Atom
- **Embedded**: microcontrollers in automobiles, door knobs
 - Need: low power, **low cost**
 - Examples: ARM chips, dedicated digital signal processors (DSPs)
 - Over 6 billion ARM cores sold in 2010 (multiple per phone)
- **Deeply Embedded**: disposable “smart dust” sensors
 - Need: extremely low power, extremely low cost

Application Specific Designs

- This class is about **general-purpose CPUs**
 - Processor that can do anything, run a full OS, etc.
 - E.g., Intel Core i7, AMD Athlon, IBM Power, ARM, Intel Itanium
- In contrast to **application-specific chips**
 - Or **ASICs** (Application specific integrated circuits)
 - Also application-domain specific processors
 - Implement critical domain-specific functionality in hardware
 - Examples: video encoding, 3D graphics
 - General rules
 - Hardware is less flexible than software
 - + Hardware more effective (speed, power, cost) than software
 - + Domain specific more “parallel” than general purpose
 - But general mainstream processors becoming more parallel
- Trend: from specific to general (for a specific domain)

Technology Trends

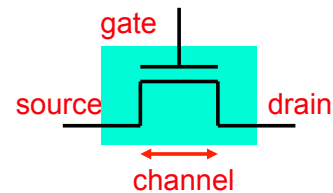
Constant Change: Technology

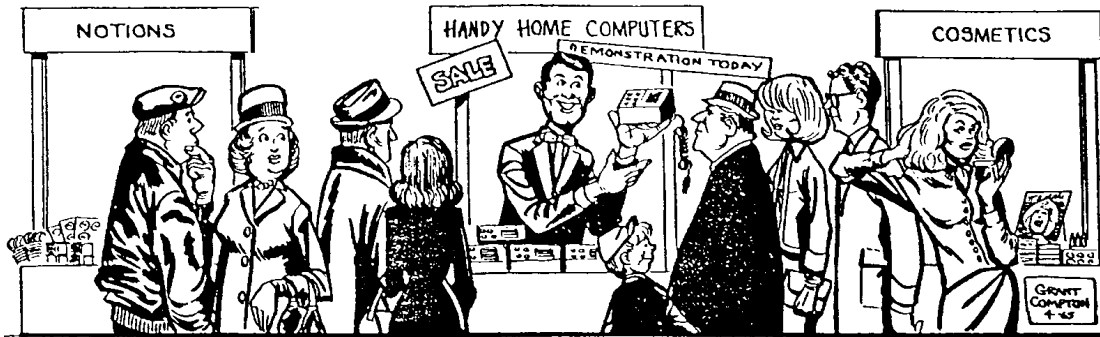


- Absolute improvement, **different rates of change**
- New application domains enabled by technology advances

“Technology”

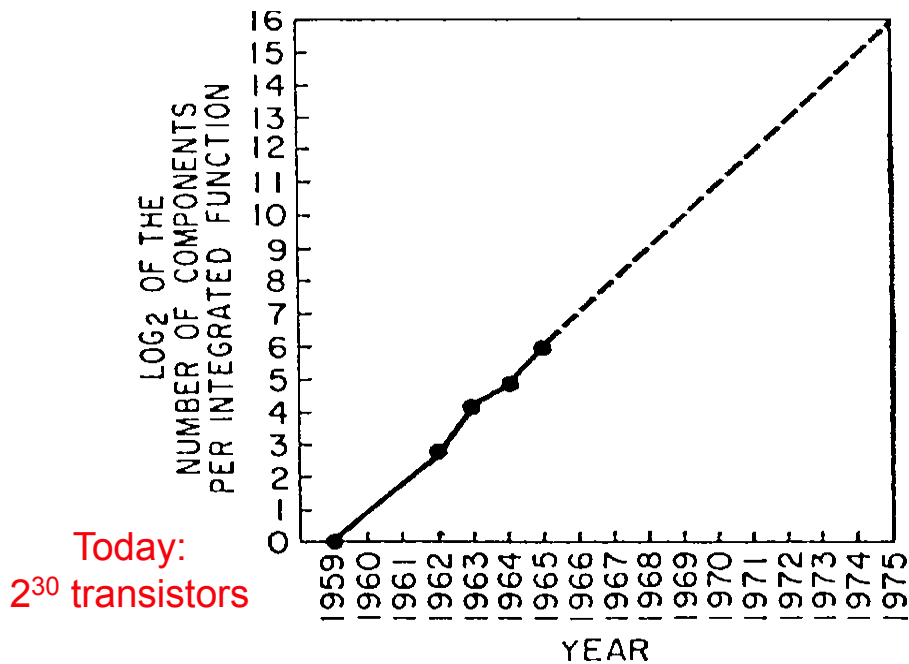
- Basic element
 - Solid-state **transistor** (i.e., electrical switch)
 - Building block of **integrated circuits (ICs)**
- What’s so great about ICs? Everything
 - + High performance, high reliability, low cost, low power
 - + Lever of mass production
- Several kinds of integrated circuit families
 - **SRAM/logic**: optimized for speed (used for processors)
 - **DRAM**: optimized for density, cost, power (used for memory)
 - **Flash**: optimized for density, cost (used for storage)
 - Increasing opportunities for integrating multiple technologies
- Non-transistor storage and inter-connection technologies
 - Disk, optical storage, ethernet, fiber optics, wireless





Funny or Not Funny?

Moore's Law - 1965



Technology Trends

- **Moore's Law**
 - Continued (up until now, at least) transistor miniaturization
- Some technology-based ramifications
 - Absolute improvements in density, speed, power, costs
 - SRAM/logic: density: ~30% (annual), speed: ~20%
 - DRAM: density: ~60%, speed: ~4%
 - Disk: density: ~60%, speed: ~10% (non-transistor)
 - Big improvements in flash memory and network bandwidth, too
- **Changing quickly and with respect to each other!!**
 - Example: density increases faster than speed
 - Trade-offs are constantly changing
 - **Re-evaluate/re-design for each technology generation**

Technology Change Drives Everything

- Computers get 10x faster, smaller, cheaper every 5-6 years!
 - A 10x quantitative change is qualitative change
 - Plane is 10x faster than car, and fundamentally different travel mode
- New applications become self-sustaining market segments
 - Recent examples: mobile phones, digital cameras, mp3 players, etc.
- Low-level improvements appear as discrete high-level jumps
 - Capabilities cross thresholds, enabling new applications and uses

Revolution I: The Microprocessor

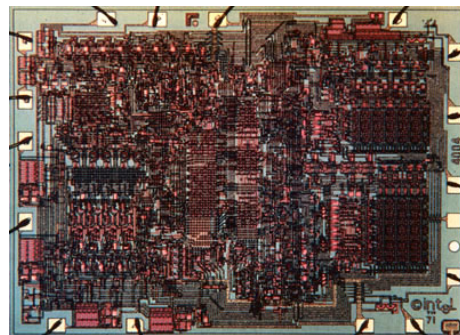
- **Microprocessor revolution**
 - One significant technology threshold was crossed in 1970s
 - Enough transistors (~25K) to put a 16-bit processor on one chip
 - Huge performance advantages: fewer slow chip-crossings
 - Even bigger cost advantages: one “stamped-out” component
- Microprocessors have allowed new market segments
 - Desktops, CD/DVD players, laptops, game consoles, set-top boxes, mobile phones, digital camera, mp3 players, GPS, automotive
- And replaced incumbents in existing segments
 - Microprocessor-based system replaced supercomputers, “mainframes”, “minicomputers”, etc.

First Microprocessor

- Intel 4004 (1971)
 - Application: calculators
 - Technology: 10000 nm

 - 2300 transistors
 - 13 mm²
 - 108 KHz
 - 12 Volts

 - 4-bit data
 - Single-cycle datapath



Pinnacle of Single-Core Microprocessors

- Intel Pentium4 (2003)
 - Application: desktop/server
 - Technology: 90nm (1/100x)
 - 55M transistors (20,000x)
 - 101 mm² (10x)
 - 3.4 GHz (10,000x)
 - 1.2 Volts (1/10x)
 - 32/64-bit data (16x)
 - 22-stage pipelined datapath
 - 3 instructions per cycle (superscalar)
 - Two levels of on-chip cache
 - data-parallel vector (SIMD) instructions, hyperthreading



Tracing the Microprocessor Revolution

- How were growing transistor counts used?
 - Initially to widen the datapath
 - 4004: 4 bits → Pentium4: 64 bits
 - ... and also to add more powerful instructions
 - To amortize overhead of fetch and decode
 - To simplify programming (which was done by hand then)

Revolution II: Implicit Parallelism

- Then to **extract implicit instruction-level parallelism**
 - Hardware provides parallel resources, figures out how to use them
 - Software is oblivious
- Initially using pipelining ...
 - Which also enabled increased clock frequency
- ... caches ...
 - Which became necessary as processor clock frequency increased
- ... and integrated floating-point
- Then deeper pipelines and branch speculation
- Then multiple instructions per cycle (superscalar)
- Then dynamic scheduling (out-of-order execution)

- We will talk about these things

Pinnacle of Single-Core Microprocessors

- Intel Pentium4 (2003)
 - Application: desktop/server
 - Technology: 90nm (1/100x)

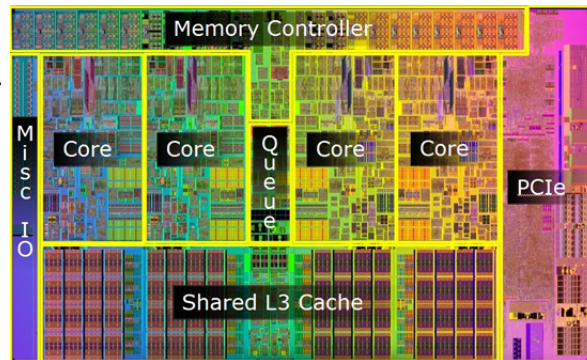
 - 55M transistors (20,000x)
 - 101 mm² (10x)
 - 3.4 GHz (10,000x)
 - 1.2 Volts (1/10x)

 - 32/64-bit data (16x)
 - 22-stage pipelined datapath
 - 3 instructions per cycle (superscalar)
 - Two levels of on-chip cache
 - data-parallel vector (SIMD) instructions, hyperthreading



Modern Multicore Processor

- Intel Core i7 (2009)
 - Application: desktop/server
 - Technology: 45nm (1/2x)



- 774M transistors (12x)
- 296 mm² (3x)
- 3.2 GHz to 3.6 Ghz (~1x)
- 0.7 to 1.4 Volts (~1x)
- 128-bit data (2x)
- 14-stage pipelined datapath (0.5x)
- 4 instructions per cycle (~1x)
- Three levels of on-chip cache
- data-parallel vector (SIMD) instructions, hyperthreading
- **Four-core multicore** (4x)

Revolution III: Explicit Parallelism

- Then to support **explicit data & thread level parallelism**
 - Hardware provides parallel resources, software specifies usage
 - Why? diminishing returns on instruction-level-parallelism
- First using (subword) vector instructions..., Intel's SSE
 - One instruction does four parallel multiplies
- ... and general support for multi-threaded programs
 - Coherent caches, hardware synchronization primitives
- Then using support for multiple concurrent threads on chip
 - First with single-core multi-threading, now with multi-core
- Graphics processing units (GPUs) are highly parallel
 - Converging with general-purpose processors (CPUs)?

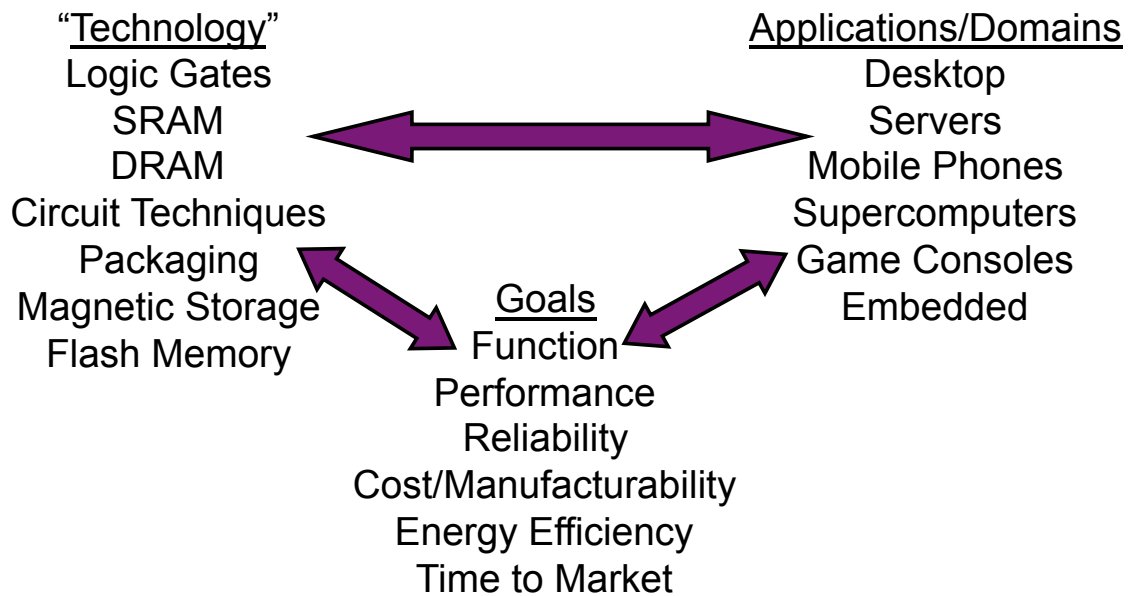
To ponder...

Is this decade's
"multicore revolution"
comparable to the original
"microprocessor revolution"?

Technology Disruptions

- Classic examples:
 - The transistor
 - Microprocessor
- More recent examples:
 - Multicore processors
 - Flash-based solid-state storage
- Near-term potentially disruptive technologies:
 - Phase-change memory (non-volatile memory)
 - Chip stacking (also called 3D die stacking)
- Disruptive "end-of-scaling"
 - "If something can't go on forever, it must stop eventually"
 - Can we continue to shrink transistors for ever?
 - Even if more transistors, not getting as energy efficient as fast

Recap: Constant Change



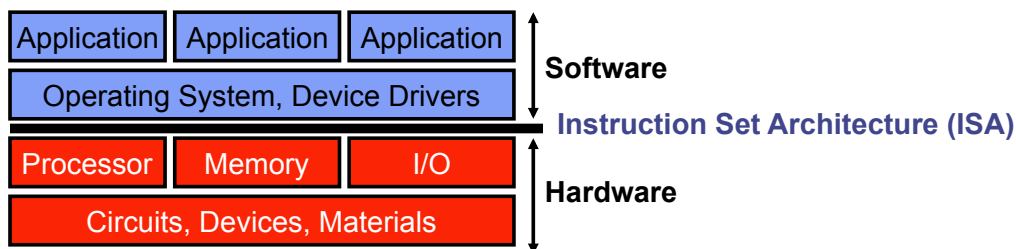
Managing This Mess

- Architect must consider all factors
 - Goals/constraints, applications, implementation technology
- Questions
 - How to deal with all of these inputs?
 - How to manage changes?
- Answers
 - Accrued institutional knowledge (stand on each other’s shoulders)
 - Experience, rules of thumb
 - Discipline: clearly defined end state, keep your eyes on the ball
 - **Abstraction and layering**

Pervasive Idea: Abstraction and Layering

- **Abstraction**: only way of dealing with complex systems
 - Divide world into objects, each with an...
 - **Interface**: knobs, behaviors, knobs → behaviors
 - **Implementation**: “black box” (ignorance+apathy)
 - Only specialists deal with implementation, rest of us with interface
 - Example: car, only mechanics know how implementation works
- **Layering**: abstraction discipline makes life even simpler
 - Divide objects in system into layers, layer n objects...
 - Implemented using interfaces of layer $n - 1$
 - Don't need to know interfaces of layer $n - 2$ (sometimes helps)
- **Inertia**: a dark side of layering
 - Layer interfaces become entrenched over time (“standards”)
 - Very difficult to change even if benefit is clear (example: Digital TV)
- **Opacity**: hard to reason about performance across layers

Abstraction, Layering, and Computers



- **Computer architecture**
 - Definition of **ISA** to facilitate implementation of software layers
- This course mostly on **computer micro-architecture**
 - Design of chip to implement **ISA**
- Touch on compilers & OS ($n + 1$), circuits ($n - 1$) as well

Course Overview

Why Study Computer Architecture?

- **Understand where computers are going**
 - Future capabilities drive the (computing) world
 - Real world-impact: no computer architecture → no computers!
- **Understand high-level design concepts**
 - The best architects understand all the levels
 - Devices, circuits, architecture, compiler, applications
- **Understand computer performance**
 - Writing well-tuned (fast) software requires knowledge of hardware
- **Write better software**
 - The best software designers also understand hardware
 - Need to understand hardware to write fast software
- **Design hardware**

Course Information

- Course goals
 - **See the “big ideas” in computer architecture**
 - Pipelining, parallelism, caching, locality, abstraction, etc.
 - Exposure to examples of good (and some bad) engineering
 - Understand how architectural ideas are evaluated (readings)
- Prerequisites
 - **General computer science/engineering background**
 - Basic computer organization concepts
 - digital logic, binary arithmetic
 - Familiarity with concept of “instruction” and assembly language
- Course level
 - Advanced undergraduate or first-year graduate-level course

Course Topics & Schedule (1 of 2)

- Day 1: Foundations & memory hierarchy
 - Introduction
 - Technology trends
 - ISAs (instruction set architectures)
 - Caches & memory hierarchy, virtual memory
- Day 2: Core micro-architecture & implicit parallelism
 - Pipelining
 - Branch prediction
 - Superscalar
 - Instruction scheduling – in hardware and software
- Day 3: Explicit parallelism
 - Multicore, synchronization, cache coherence, memory consistency
 - Data-level parallelism, vectors, and just a bit on GPUs
 - Xbox 360 & Wrapup

Course Topics & Schedule (2 of 2)

- A key course theme: **Parallelism**
 - Instruction-level: superscalar, dynamic scheduling, speculation
 - Data-level: vectors and GPUs
 - Thread-level: multicore, cache coherence, memory consistency
- Covered briefly
 - Virtual memory
 - GPUs
- Not covered
 - Off-chip memory technologies (DRAM)
 - Reliability
 - Input/output devices
 - Datacenter architectures & deeply embedded

Instructor Bio: Prof. Milo M. K. Martin

- Associate Professor
 - University of Pennsylvania (or "Penn"); private Ivy-league university in Philadelphia
 - PhD from the University of Wisconsin-Madison
- My research focuses on...
making computers more responsive and easier to design and program
- Specific projects include:
 - Computational sprinting
 - Hardware transactional memory & adaptive cache coherence
 - Memory consistency models
 - Hardware-aware verification of concurrent software
 - Hardware-assisted memory safety of unsafe programming language
- Published over 40 papers; have received over 2500 citations
- Recipient of the NSF CAREER award

Full Disclosure

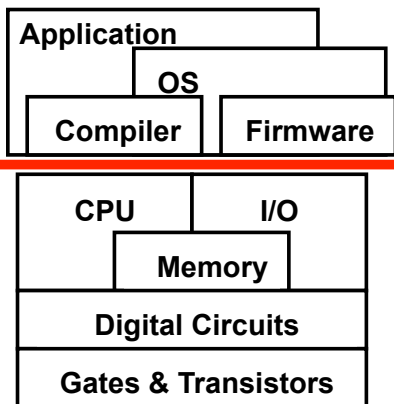
- Potential sources of bias or conflict of interest
- Most of my funding is governmental (your tax \$\$\$ at work)
 - National Science Foundation (NSF)
 - DARPA & ONR
- My non-governmental sources of research funding
 - NVIDIA (sub-contract of large DARPA project, now completed)
 - Intel
 - Sun/Oracle (hardware donation)
- Collaborators and colleagues
 - Intel, IBM, AMD, Oracle, Microsoft, Google, VMWare, ARM, etc.
 - (Just about every major computer company)
- Sabbatical at Google for 2013-2014

Computer Architecture

Unit 2: Instruction Set Architectures

Slides developed by Milo Martin & Amir Roth at the University of Pennsylvania
with sources that included University of Wisconsin slides
by Mark Hill, Guri Sohi, Jim Smith, and David Wood

Instruction Set Architecture (ISA)



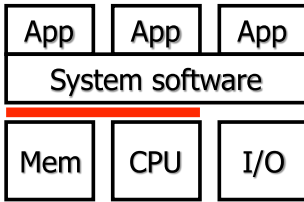
- What is an ISA?
 - A functional contract
- All ISAs similar in high-level ways
 - But many design choices in details
 - Two “philosophies”: CISC/RISC
 - Difference is blurring
- Good ISA...
 - Enables high-performance
 - At least doesn’t get in the way
- Compatibility is a powerful force
 - Tricks: binary translation, μ ISAs

Readings

- Suggested reading:
 - “The Evolution of RISC Technology at IBM”
by John Cocke and V. Markstein

Execution Model

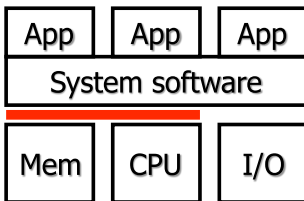
Program Compilation



```
int array[100], sum;
void array_sum() {
    for (int i=0; i<100;i++) {
        sum += array[i];
    }
}
```

- **Program** written in a “high-level” programming language
 - C, C++, Java, C#
 - Hierarchical, structured control: loops, functions, conditionals
 - Hierarchical, structured data: scalars, arrays, pointers, structures
- **Compiler**: translates program to **assembly**
 - Parsing and straight-forward translation
 - Compiler also optimizes
 - Compiler itself another application ... who compiled compiler?

Assembly & Machine Language

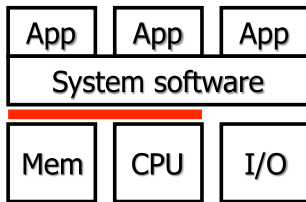


Machine code	Assembly code
x9A00	CONST R5, #0
x9200	CONST R1, array
xD320	HICONST R1, array
x9464	CONST R2, sum
xD520	HICONST R2, sum
x6640	LDR R3, R1, #0
x6880	LDR R4, R2, #0
x18C4	ADD R4, R3, R4
x7880	STR R4, R2, #0
x1261	ADD R1, R1, #1
x1BA1	ADD R5, R5, #1
x2B64	CMPI R5, #100
x03F8	BRn array_sum_loop

- **Assembly language**
 - Human-readable representation
- **Machine language**
 - Machine-readable representation
 - 1s and 0s (often displayed in “hex”)
- **Assembler**
 - Translates assembly to machine

Example is in “LC4” a toy **instruction set architecture, or ISA**

Example Assembly Language & ISA



- **MIPS**: example of real ISA
 - 32/64-bit operations
 - 32-bit insns
 - 64 registers
 - 32 integer, 32 floating point
 - ~100 different insns

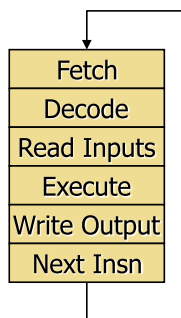
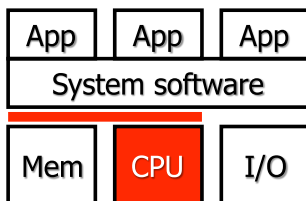
```

.data
array: .space 100
sum:   .word 0

.text
array_sum:
    li $5, 0
    la $1, array
    la $2, sum
array_sum_loop:
    lw $3, 0($1)
    lw $4, 0($2)
    add $4, $3, $4
    sw $4, 0($2)
    addi $1, $1, 1
    addi $5, $5, 1
    li $6, 100
    blt $5, $6, array_sum_loop
    
```

Example code is MIPS, but all ISAs are similar at some level

Instruction Execution Model



Instruction → Insn

- The computer is just finite state machine
 - **Registers** (few of them, but fast)
 - **Memory** (lots of memory, but slower)
 - **Program counter** (next insn to execute)
 - Called "instruction pointer" in x86
- A computer executes **instructions**
 - **Fetches** next instruction from memory
 - **Decodes** it (figure out what it does)
 - **Reads** its **inputs** (registers & memory)
 - **Executes** it (adds, multiply, etc.)
 - **Write** its **outputs** (registers & memory)
 - **Next insn** (adjust the program counter)
- **Program is just "data in memory"**
 - Makes computers programmable ("universal")

What is an ISA?

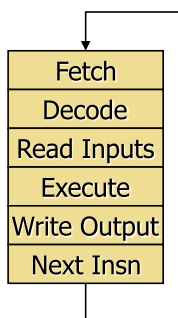
What Is An ISA?

- **ISA (instruction set architecture)**
 - A well-defined hardware/software interface
 - The **"contract"** between software and hardware
 - **Functional definition** of storage locations & operations
 - Storage locations: registers, memory
 - Operations: add, multiply, branch, load, store, etc
 - **Precise description** of how to invoke & access them
- Not in the "contract": non-functional aspects
 - How operations are implemented
 - Which operations are fast and which are slow and when
 - Which operations take more power and which take less
- Instructions
 - Bit-patterns hardware interprets as commands
 - Instruction → Insn (instruction is too long to write in slides)

A Language Analogy for ISAs

- Communication
 - Person-to-person → software-to-hardware
- Similar structure
 - Narrative → program
 - Sentence → insn
 - Verb → operation (add, multiply, load, branch)
 - Noun → data item (immediate, register value, memory value)
 - Adjective → addressing mode
- Many different languages, many different ISAs
 - Similar basic structure, details differ (sometimes greatly)
- Key differences between languages and ISAs
 - Languages evolve organically, many ambiguities, inconsistencies
 - ISAs are explicitly engineered and extended, unambiguous

The Sequential Model



- **Basic structure of all modern ISAs**
 - Often called VonNeuman, but in ENIAC before
- **Program order**: total order on dynamic insns
 - Order and **named storage** define computation
- Convenient feature: **program counter (PC)**
 - Insn itself stored in memory at location pointed to by PC
 - Next PC is next insn unless insn says otherwise
- Processor logically executes loop at left
- **Atomic**: insn finishes before next insn starts
 - Implementations can break this constraint physically
 - But must maintain illusion to preserve correctness

ISA Design Goals

What Makes a Good ISA?

- **Programmability**
 - Easy to express programs efficiently?
- **Performance/Implementability**
 - Easy to design high-performance implementations?
 - More recently
 - Easy to design low-power implementations?
 - Easy to design low-cost implementations?
- **Compatibility**
 - Easy to maintain as languages, programs, and technology evolve?
 - x86 (IA32) generations: 8086, 286, 386, 486, Pentium, PentiumII, PentiumIII, Pentium4, Core2, Core i7, ...

Programmability

- Easy to express programs efficiently?
 - For whom?
- Before 1980s: **human**
 - Compilers were terrible, most code was hand-assembled
 - Want high-level coarse-grain instructions
 - As similar to high-level language as possible
- After 1980s: **compiler**
 - Optimizing compilers generate much better code than you or I
 - Want low-level fine-grain instructions
 - Compiler can't tell if two high-level idioms match exactly or not
- This shift changed what is considered a "good" ISA...

Implementability

- Every ISA can be implemented
 - Not every ISA can be implemented efficiently
- Classic high-performance implementation techniques
 - Pipelining, parallel execution, out-of-order execution (more later)
- Certain ISA features make these difficult
 - Variable instruction lengths/formats: complicate decoding
 - Special-purpose registers: complicate compiler optimizations
 - Difficult to interrupt instructions: complicate many things
 - Example: memory copy instruction

Performance, Performance, Performance

- How long does it take for a program to execute?
 - Three factors
- 1. How many insn must execute to complete program?
 - **Instructions per program** during execution
 - "Dynamic insn count" (not number of "static" insns in program)
- 2. How quickly does the processor "cycle"?
 - **Clock frequency** (cycles per second) 1 gigahertz (Ghz)
 - or expressed as reciprocal, **Clock period** nanosecond (ns)
 - Worst-case delay through circuit for a particular design
- 3. How many *cycles* does each instruction take to execute?
 - **Cycles per Instruction** (CPI) or reciprocal, **Insn per Cycle** (IPC)

$$\text{Execution time} = (\text{instructions/program}) * (\text{seconds/cycle}) * (\text{cycles/instruction})$$

Maximizing Performance

$$\text{Execution time} = (\text{instructions/program}) * (\text{seconds/cycle}) * (\text{cycles/instruction})$$

$$(1 \text{ billion instructions}) * (1 \text{ ns per cycle}) * (1 \text{ cycle per insn}) = 1 \text{ second}$$

- Instructions per program:
 - Determined by program, compiler, instruction set architecture (ISA)
- Cycles per instruction: "CPI"
 - Typical range today: 2 to 0.5
 - Determined by program, compiler, ISA, micro-architecture
- Seconds per cycle: "clock period"
 - Typical range today: 2ns to 0.25ns
 - Reciprocal is frequency: 0.5 Ghz to 4 Ghz (1 Htz = 1 cycle per sec)
 - Determined by micro-architecture, technology parameters
- For minimum execution time, minimize each term
 - Difficult: **often pull against one another**

Example: Instruction Granularity

$$\text{Execution time} = (\text{instructions/program}) * (\text{seconds/cycle}) * (\text{cycles/instruction})$$

- **CISC** (Complex Instruction Set Computing) **ISAs**
 - Big heavyweight instructions (lots of work per instruction)
 - + Low "insns/program"
 - Higher "cycles/insn" and "seconds/cycle"
 - We have the technology to get around this problem
- **RISC** (Reduced Instruction Set Computer) **ISAs**
 - Minimalist approach to an ISA: simple insns only
 - + Low "cycles/insn" and "seconds/cycle"
 - Higher "insn/program", but hopefully not as much
 - Rely on compiler optimizations

Compiler Optimizations

- Primarily goal: reduce instruction count
 - Eliminate redundant computation, keep more things in registers
 - + Registers are faster, fewer loads/stores
 - An ISA can make this difficult by having too few registers
- But also...
 - Reduce branches and jumps (later)
 - Reduce cache misses (later)
 - Reduce dependences between nearby insns (later)
 - An ISA can make this difficult by having implicit dependences
- How effective are these?
 - + Can give 4X performance over unoptimized code
 - Collective wisdom of 40 years ("Proebsting's Law"): 4% per year
 - Funny but ... shouldn't leave 4X performance on the table

Compatibility

- In many domains, ISA must remain compatible
 - IBM's 360/370 (the *first* "ISA family")
 - Another example: Intel's x86 and Microsoft Windows
 - x86 one of the worst designed ISAs EVER, but survives
- **Backward compatibility**
 - New processors supporting old programs
 - Can't drop features (**caution in adding new ISA features**)
 - Or, update software/OS to emulate dropped features (slow)
- **Forward (upward) compatibility**
 - Old processors supporting new programs
 - Include a "CPU ID" so the software can test of features
 - Add ISA hints by overloading no-ops (example: x86's PAUSE)
 - New firmware/software on old processors to emulate new insn

Translation and Virtual ISAs

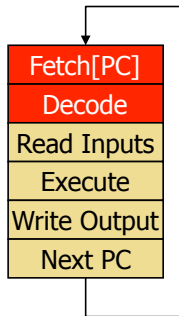
- New compatibility interface: ISA + translation software
 - **Binary-translation**: transform static image, run native
 - **Emulation**: unmodified image, interpret each dynamic insn
 - Typically optimized with just-in-time (JIT) compilation
 - Examples: FX!32 (x86 on Alpha), Rosetta (PowerPC on x86)
 - Performance overheads reasonable (many advances over the years)
- **Virtual ISAs**: designed for translation, not direct execution
 - Target for high-level compiler (one per language)
 - Source for low-level translator (one per ISA)
 - Goals: Portability (abstract hardware nastiness), flexibility over time
 - Examples: Java Bytecodes, C# CLR (Common Language Runtime)
NVIDIA's "PTX"

Ultimate Compatibility Trick

- Support old ISA by...
 - ...having a simple processor for that ISA somewhere in the system
 - How did PlayStation2 support PlayStation1 games?
 - Used PlayStation processor for I/O chip **& emulation**

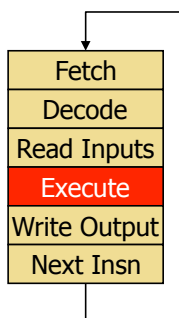
Aspects of ISAs

Length and Format



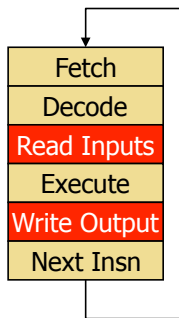
- **Length**
 - Fixed length
 - Most common is 32 bits
 - + Simple implementation (next PC often just PC+4)
 - Code density: 32 bits to increment a register by 1
 - Variable length
 - + Code density
 - x86 averages 3 bytes (ranges from 1 to 16)
 - Complex fetch (where does next instruction begin?)
 - Compromise: two lengths
 - E.g., MIPS16 or ARM's Thumb
- **Encoding**
 - A few simple encodings simplify decoder
 - x86 decoder one nasty piece of logic

Operations and Datatypes



- Datatypes
 - Software: attribute of data
 - Hardware: attribute of operation, data is just 0/1's
- All processors support
 - Integer arithmetic/logic (8/16/32/64-bit)
 - IEEE754 floating-point arithmetic (32/64-bit)
- More recently, most processors support
 - "Packed-integer" insns, e.g., MMX
 - "Packed-floating point" insns, e.g., SSE/SSE2/AVX
 - For "data parallelism", more about this later
- Other, infrequently supported, data types
 - Decimal, other fixed-point arithmetic

Where Does Data Live?



- **Registers**
 - “short term memory”
 - Faster than memory, quite handy
 - Named directly in instructions
- **Memory**
 - “longer term memory”
 - Accessed via “addressing modes”
 - Address to read or write calculated by instruction
- “Immediates”
 - Values spelled out as bits in instructions
 - Input only

How Many Registers?

- Registers faster than memory, have as many as possible?
 - **No**
- One reason registers are faster: there are **fewer of them**
 - Small is fast (hardware truism)
- Another: they are **directly addressed** (no address calc)
 - More registers, means more bits per register in instruction
 - Thus, fewer registers per instruction or larger instructions
- **Not everything can be put in registers**
 - Structures, arrays, anything pointed-to
 - Although compilers are getting better at putting more things in
- More registers means **more saving/restoring**
 - Across function calls, traps, and context switches
- Trend toward more registers:
 - 8 (x86) → 16 (x86-64), 16 (ARM v7) → 32 (ARM v8)

Memory Addressing

- **Addressing mode:** way of specifying address
 - Used in memory-memory or load/store instructions in register ISA
- Examples
 - **Displacement:** $R1 = \text{mem}[R2 + \text{immed}]$
 - **Index-base:** $R1 = \text{mem}[R2 + R3]$
 - **Memory-indirect:** $R1 = \text{mem}[\text{mem}[R2]]$
 - **Auto-increment:** $R1 = \text{mem}[R2], R2 = R2 + 1$
 - **Auto-indexing:** $R1 = \text{mem}[R2 + \text{immed}], R2 = R2 + \text{immed}$
 - **Scaled:** $R1 = \text{mem}[R2 + R3 * \text{immed1} + \text{immed2}]$
 - **PC-relative:** $R1 = \text{mem}[\text{PC} + \text{imm}]$
- What high-level program idioms are these used for?
- What implementation impact? What impact on insn count?

Addressing Modes Examples

- MIPS

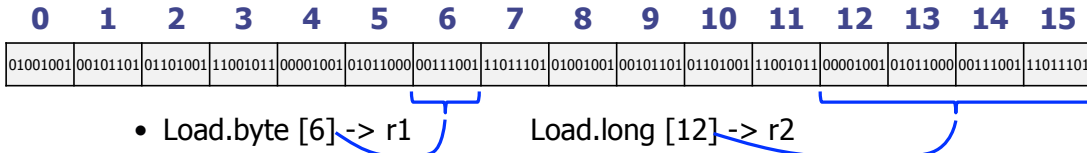
I-type	Op(6)	Rs(5)	Rt(5)	Immed(16)
--------	-------	-------	-------	-----------

 - **Displacement:** $R1 + \text{offset}$ (16-bit)
 - Why? Experiments on VAX (ISA with every mode) found:
 - 80% use small displacement (or displacement of zero)
 - Only 1% accesses use displacement of more than 16bits
- Other ISAs (SPARC, x86) have reg+reg mode, too
 - Impacts both implementation and insn count? (How?)
- x86 (MOV instructions)
 - **Absolute:** zero + offset (8/16/32-bit)
 - **Register indirect:** R1
 - **Displacement:** $R1 + \text{offset}$ (8/16/32-bit)
 - **Indexed:** $R1 + R2$
 - **Scaled:** $R1 + (R2 * \text{Scale}) + \text{offset}$ (8/16/32-bit) Scale = 1, 2, 4, 8

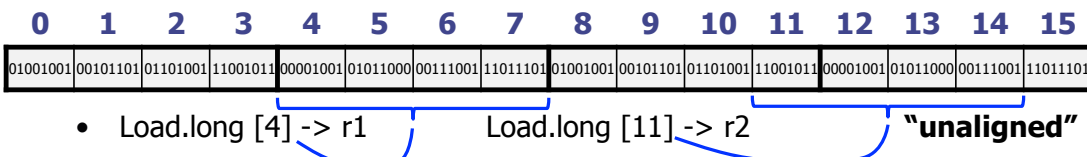
Access Granularity & Alignment

- **Byte addressability**

- An address points to a byte (8 bits) of data
- The ISA's minimum granularity to read or write memory
- ISAs also support wider load/stores
 - "Half" (2 bytes), "Longs" (4 bytes), "Quads" (8 bytes)



However, physical memory systems operate on **even larger chunks**



- **Access alignment:** if address % size is not 0, then it is "unaligned"
 - A single unaligned access may require multiple physical memory accesses

Handling Unaligned Accesses

- **Access alignment:** if address % size is not 0, then it is "unaligned"
 - A single unaligned access may require multiple physical memory accesses
- How do handle such unaligned accesses?
 1. Disallow (unaligned operations are considered illegal)
 - MIPS takes this route
 2. Support in hardware? (allow such operations)
 - x86 allows regular loads/stores to be unaligned
 - Unaligned access still slower, adds significant hardware complexity
 3. Trap to software routine? (allow, but hardware traps to software)
 - Simpler hardware, but high penalty when unaligned
 4. In software (compiler can use regular instructions when possibly unaligned)
 - Load, shift, load, shift, and (slow, needs help from compiler)
 5. MIPS? ISA support: unaligned access by compiler using two instructions
 - Faster than above, but still needs help from compiler

```
lw1 @XXXX10; lwr @XXXX10
```

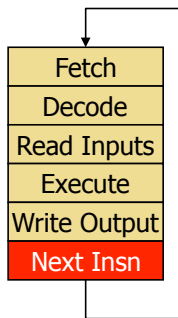

Operand Model: Register or Memory?

- “Load/store” architectures
 - Memory access instructions (loads and stores) are distinct
 - Separate addition, subtraction, divide, etc. operations
 - Examples: MIPS, ARM, SPARC, PowerPC
- Alternative: mixed operand model (x86, VAX)
 - Operand can be from register **or** memory
 - x86 example: `addl 100, 4(%eax)`
 - 1. Loads from memory location $[4 + \%eax]$
 - 2. Adds “100” to that value
 - 3. Stores to memory location $[4 + \%eax]$
 - Would requires three instructions in MIPS, for example.

How Much Memory? Address Size

- What does “64-bit” in a 64-bit ISA mean?
 - **Each program can address (i.e., use) 2^{64} bytes**
 - 64 is the **address size**
 - Alternative (wrong) definition: width of arithmetic operations
- Most critical, inescapable ISA design decision
 - Too small? Will limit the lifetime of ISA
 - May require nasty hacks to overcome (E.g., x86 segments)
- x86 evolution:
 - 4-bit (4004), 8-bit (8008), 16-bit (8086), 24-bit (80286),
 - 32-bit + protected memory (80386)
 - 64-bit (AMD’s Opteron & Intel’s Pentium4)
- All ISAs moving to 64 bits (if not already there)

Control Transfers



- Default next-PC is $PC + \text{sizeof}(\text{current insn})$
 - Branches and jumps can change that
- **Computing targets:** where to jump to
 - For all branches and jumps
 - **PC-relative:** for branches and jumps with function
 - **Absolute:** for function calls
 - **Register indirect:** for returns, switches & dynamic calls
- **Testing conditions:** whether to jump at all
 - **Implicit condition codes or "flags" (x86)**

```
cmp R1,10 // sets "negative" flag
branch-neg target
```
 - **Use registers & separate branch insns (MIPS)**

```
set-less-than R2,R1,10
branch-not-equal-zero R2,target
```

ISAs Also Include Support For...

- Function calling conventions
 - Which registers are saved across calls, how parameters are passed
- Operating systems & memory protection
 - Privileged mode
 - System call (TRAP)
 - Exceptions & interrupts
 - Interacting with I/O devices
- Multiprocessor support
 - "Atomic" operations for synchronization
- Data-level parallelism
 - Pack many values into a wide register
 - Intel's SSE2: four 32-bit float-point values into 128-bit register
 - Define parallel operations (four "adds" in one cycle)

The RISC vs. CISC Debate

RISC and CISC

- **RISC**: reduced-instruction set computer
 - Coined by Patterson in early 80's
 - RISC-I (Patterson), MIPS (Hennessy), IBM 801 (Cocke)
 - Examples: PowerPC, ARM, SPARC, Alpha, PA-RISC
- **CISC**: complex-instruction set computer
 - Term didn't exist before "RISC"
 - Examples: x86, VAX, Motorola 68000, etc.
- Philosophical war started in mid 1980's
 - RISC "won" the technology battles
 - CISC won the high-end commercial space (1990s to today)
 - Compatibility was a strong force
 - RISC winning the embedded computing space

CISCs and RISCs

- The CISCs: x86, VAX (**V**irtual **A**ddress **eX**tension to PDP-11)
 - Variable length instructions: 1-321 bytes!!!
 - 14 registers + PC + stack-pointer + condition codes
 - Data sizes: 8, 16, 32, 64, 128 bit, decimal, string
 - Memory-memory instructions for all data sizes
 - Special insns: `crc`, `insque`, `polyf`, and a cast of hundreds
 - x86: "Difficult to explain and impossible to love"
- The RISCs: MIPS, PA-RISC, SPARC, PowerPC, Alpha, ARM
 - 32-bit instructions
 - 32 integer registers, 32 floating point registers
 - ARM has 16 registers
 - Load/store architectures with few addressing modes
 - Why so many basically similar ISAs? Everyone wanted their own

Historical Development

- Pre 1980
 - Bad compilers (so assembly written by hand)
 - Complex, high-level ISAs (**easier to write assembly**)
 - Slow multi-chip micro-programmed implementations
 - Vicious feedback loop
- Around 1982
 - Moore's Law makes single-chip microprocessor possible...
 - **...but only for small, simple ISAs**
 - Performance advantage of this "integration" was compelling
- **RISC manifesto**: create ISAs that...
 - **Simplify single-chip implementation**
 - **Facilitate optimizing compilation**

The RISC Design Tenets

- **Single-cycle execution**
 - CISC: many multicycle operations
- **Hardwired (simple) control**
 - CISC: "microcode" for multi-cycle operations
- **Load/store architecture**
 - CISC: register-memory and memory-memory
- **Few memory addressing modes**
 - CISC: many modes
- **Fixed-length instruction format**
 - CISC: many formats and lengths
- **Reliance on compiler optimizations**
 - CISC: hand assemble to get good performance
- **Many registers** (compilers can use them effectively)
 - CISC: few registers

RISC vs CISC Performance Argument

- Performance equation:
 - $(\text{instructions/program}) * (\text{cycles/instruction}) * (\text{seconds/cycle})$
- **CISC** (Complex Instruction Set Computing)
 - Reduce "instructions/program" with "complex" instructions
 - But tends to increase "cycles/instruction" or clock period
 - Easy for assembly-level programmers, good code density
- **RISC** (Reduced Instruction Set Computing)
 - Improve "cycles/instruction" with many single-cycle instructions
 - Increases "instruction/program", but hopefully not as much
 - **Help from smart compiler**
 - Perhaps improve clock cycle time (seconds/cycle)
 - **via aggressive implementation allowed by simpler insn**

The Debate

- RISC argument
 - CISC is fundamentally handicapped
 - For a given technology, RISC implementation will be better (faster)
 - Current technology enables single-chip RISC
 - When it enables single-chip CISC, RISC will be pipelined
 - When it enables pipelined CISC, RISC will have caches
 - When it enables CISC with caches, RISC will have next thing...
- CISC rebuttal
 - CISC flaws not fundamental, can be fixed with more transistors
 - Moore's Law will narrow the RISC/CISC gap (true)
 - Good pipeline: RISC = 100K transistors, CISC = 300K
 - By 1995: 2M+ transistors had evened playing field
 - Software costs dominate, **compatibility** is paramount

Intel's x86 Trick: RISC Inside

- 1993: Intel wanted "out-of-order execution" in Pentium Pro
 - Hard to do with a coarse grain ISA like x86
- Solution? Translate x86 to RISC micro-ops (**μops**) in hardware

```
push $eax
becomes (we think, uops are proprietary)
store $eax, -4($esp)
addi $esp, $esp, -4
```

 - + Processor maintains **x86 ISA externally for compatibility**
 - + But executes **RISC μISA internally for implementability**
 - Given translator, x86 almost as easy to implement as RISC
 - Intel implemented "out-of-order" before any RISC company
 - "out-of-order" also helps x86 more (because ISA limits compiler)
 - Also used by other x86 implementations (AMD)
 - Different **μops** for different designs
 - **Not part of the ISA specification**, not publically disclosed

Potential Micro-op Scheme

- Most instructions are a **single** micro-op
 - Add, xor, compare, branch, etc.
 - Loads example: `mov -4(%rax), %ebx`
 - Stores example: `mov %ebx, -4(%rax)`
- Each memory access adds a micro-op
 - `"addl -4(%rax), %ebx"` is two micro-ops (load, add)
 - `"addl %ebx, -4(%rax)"` is three micro-ops (load, add, store)
- Function call (CALL) – 4 uops
 - Get program counter, store program counter to stack, adjust stack pointer, unconditional jump to function start
- Return from function (RET) – 3 uops
 - Adjust stack pointer, load return address from stack, jump register
- Again, just a basic idea, micro-ops are specific to each chip

Winner for Desktop PCs: CISC

- x86 was first mainstream 16-bit microprocessor by ~2 years
 - IBM put it into its PCs...
 - Rest is historical inertia, Moore's law, and "financial feedback"
 - x86 is most difficult ISA to implement and do it fast but...
 - Because Intel sells the most **non-embedded** processors...
 - It hires more and better engineers...
 - Which help it maintain competitive performance ...
 - **And given competitive performance, compatibility wins...**
 - So Intel sells the most **non-embedded** processors...
 - AMD as a competitor keeps pressure on x86 performance
- Moore's Law has helped Intel in a big way
 - Most engineering problems can be solved with more transistors

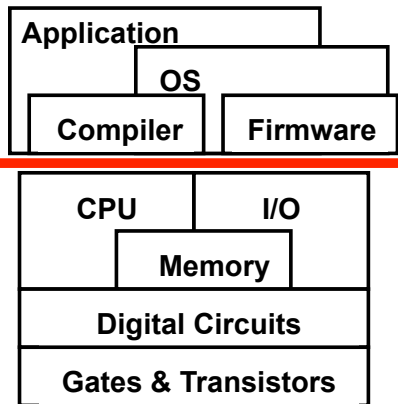
Winner for Embedded: RISC

- ARM (Acorn RISC Machine → Advanced RISC Machine)
 - First ARM chip in mid-1980s (from Acorn Computer Ltd).
 - 3 billion units sold in 2009 (>60% of all 32/64-bit CPUs)
 - Low-power and **embedded** devices (phones, for example)
 - Significance of embedded? ISA Compatibility less powerful force
- 32-bit RISC ISA
 - 16 registers, PC is one of them
 - Rich addressing modes, e.g., auto increment
 - Condition codes, each instruction can be conditional
- ARM does not sell chips; it licenses its ISA & core designs
- ARM chips from many vendors
 - Qualcomm, Freescale (was Motorola), Texas Instruments, STMicroelectronics, Samsung, Sharp, Philips, etc.

Redux: Are ISAs Important?

- Does “quality” of ISA actually matter?
 - Not for performance (mostly)
 - Mostly comes as a design complexity issue
 - Insn/program: everything is compiled, compilers are good
 - Cycles/insn and seconds/cycle: μ ISA, many other tricks
 - What about power efficiency? **Maybe**
 - ARMs are most power efficient today...
 - ...but Intel is moving x86 that way (e.g, Intel’s Atom)
 - **Open question: can x86 be as power efficient as ARM?**
- Does “nastiness” of ISA matter?
 - Mostly no, only compiler writers and hardware designers see it
- Even compatibility is not what it used to be
 - Software emulation
 - **Open question: will “ARM compatibility” be the next x86?**

Instruction Set Architecture (ISA)



- What is an ISA?
 - A functional contract
- All ISAs similar in high-level ways
 - But many design choices in details
 - Two “philosophies”: CISC/RISC
 - Difference is blurring
- Good ISA...
 - Enables high-performance
 - At least doesn't get in the way
- Compatibility is a powerful force
 - Tricks: binary translation, μ ISAs

[spacer]

Computer Architecture

Unit 3: Technology & Energy

Slides developed by Milo Martin & Amir Roth at the University of Pennsylvania
with sources that included University of Wisconsin slides
by Mark Hill, Guri Sohi, Jim Smith, and David Wood

This Unit: Technology & Energy

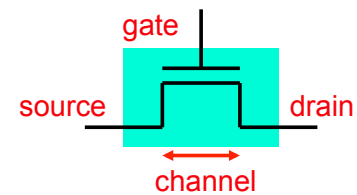
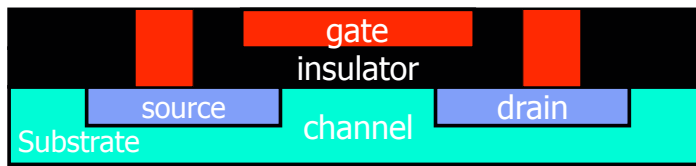
- Technology basis
 - Fabrication (manufacturing) & cost
 - Transistors & wires
 - Implications of transistor scaling (Moore's Law)
- Energy & power

Readings

- Assigned reading:
 - G. Moore, "Cramming More Components onto Integrated Circuits"

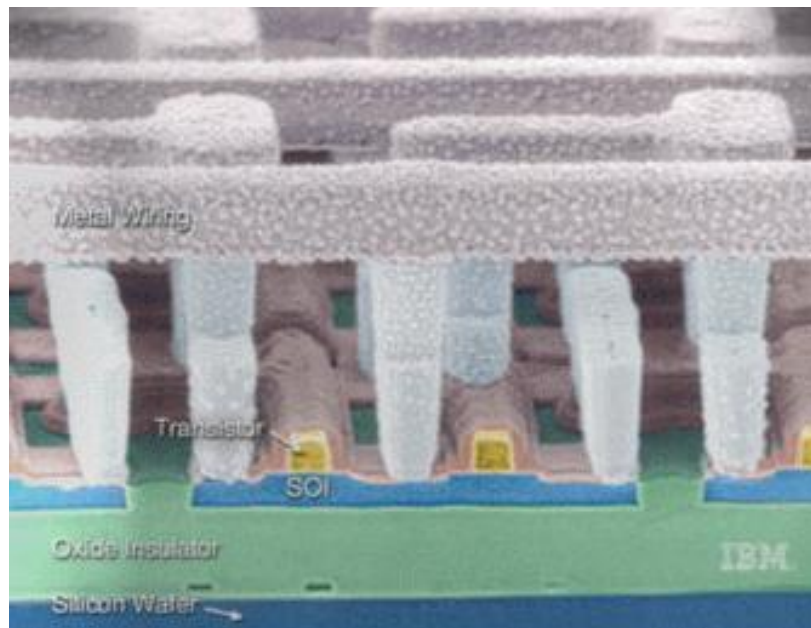
Technology & Fabrication

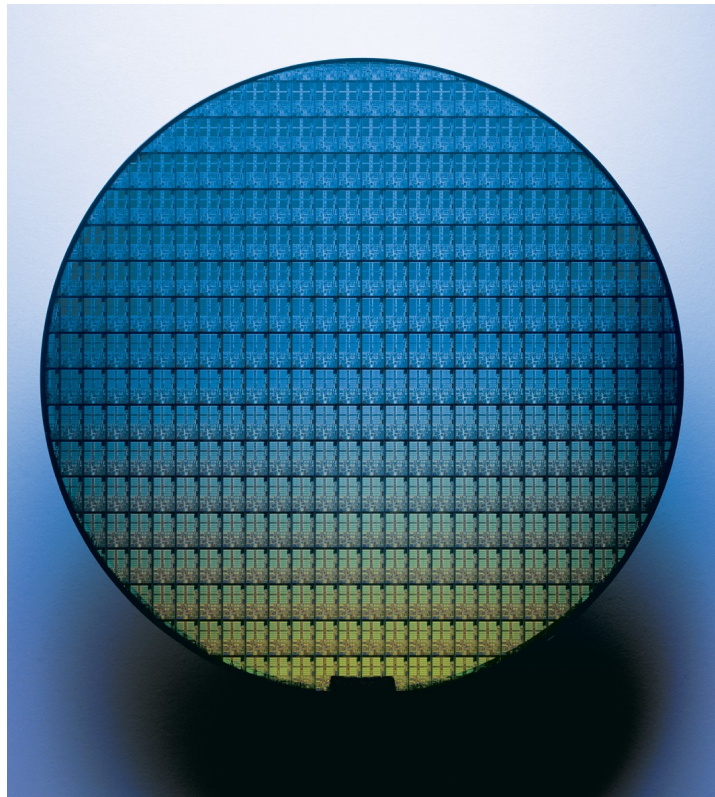
Semiconductor Technology



- Basic technology element: **MOSFET**
 - Solid-state component acts like electrical switch
 - **MOS**: metal-oxide-semiconductor
 - Conductor, insulator, semi-conductor
- **FET**: field-effect transistor
 - Channel conducts source→drain only when voltage applied to gate
- **Channel length**: characteristic parameter (short → fast)
 - Aka "feature size" or "technology"
 - Currently: 0.022 micron (μm), 22 nanometers (nm)
 - Continued miniaturization (scaling) known as "**Moore's Law**"
 - Won't last forever, physical limits approaching (or are they?)

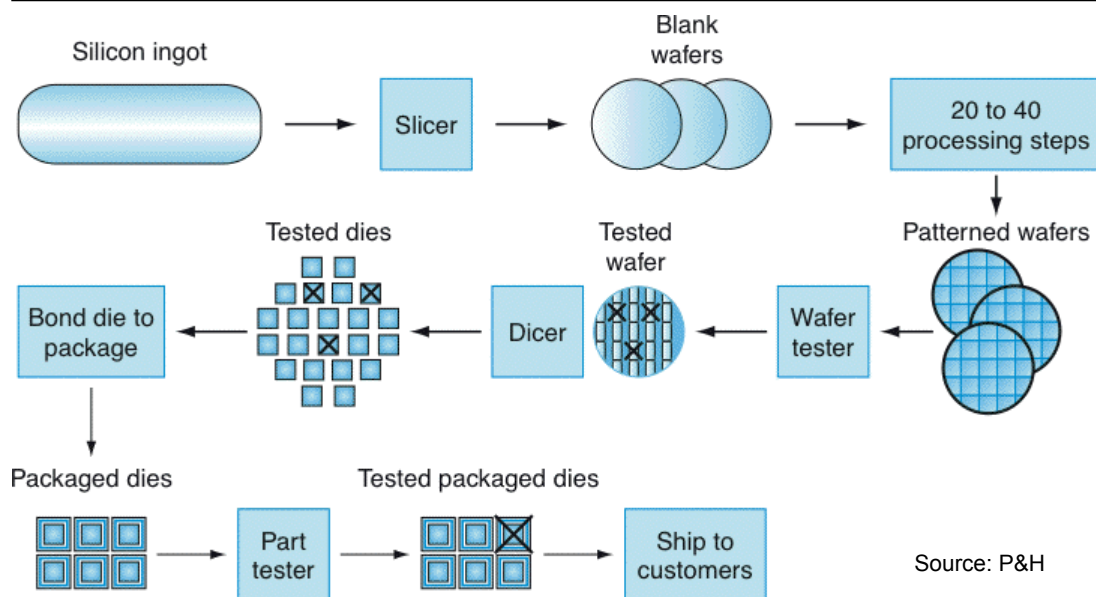
Transistors and Wires





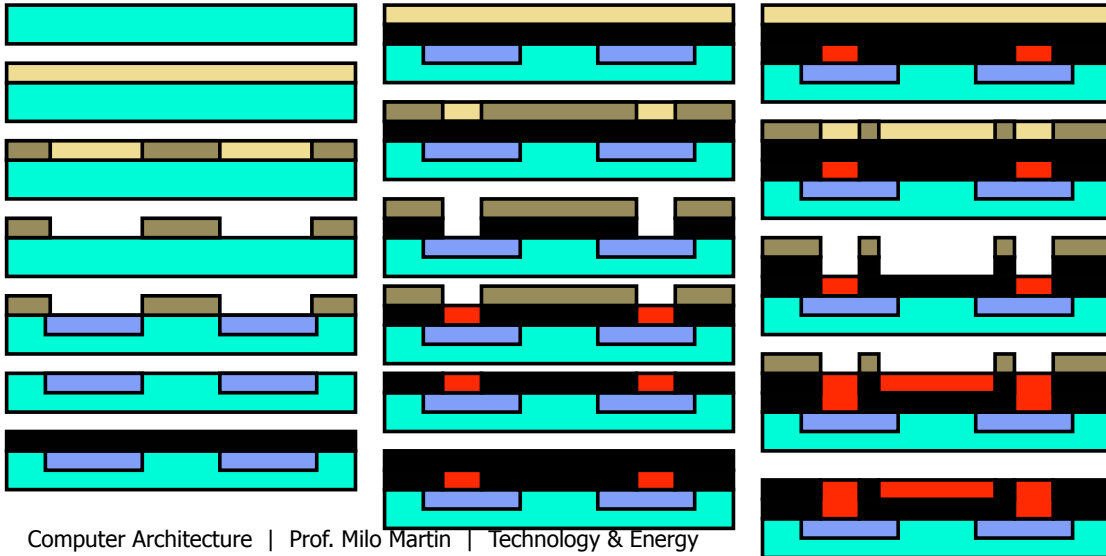
Intel Pentium M Wafer

Manufacturing Steps



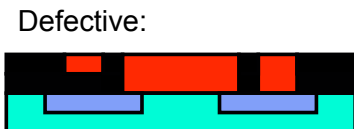
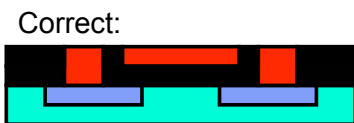
Manufacturing Steps

- Multi-step photo-/electro-chemical process
 - More steps, higher unit cost
- + Fixed cost mass production (\$1 million+ for "mask set")



Computer Architecture | Prof. Milo Martin | Technology & Energy

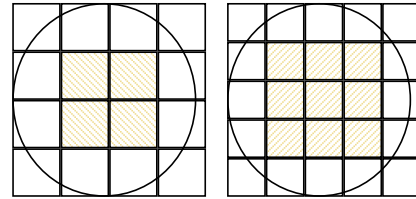
Manufacturing Defects



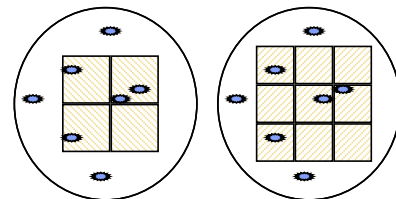
- Defects can arise
 - Under-/over-doping
 - Over-/under-dissolved insulator
 - Mask mis-alignment
 - Particle contaminants
- Try to minimize defects
 - Process margins
 - Design rules
 - Minimal transistor size, separation
- Or, tolerate defects
 - Redundant or "spare" memory cells
 - Can substantially improve yield

Cost Implications of Defects

- Chips built in multi-step chemical processes on **wafers**
 - Cost / wafer is constant, f(wafer size, number of steps)
- Chip (die) cost is related to **area**
 - Larger chips means fewer of them
- Cost is **more than** linear in area
 - Why? random defects
 - Larger chip, more chance of defect
 - Result: lower "yield" (fewer working chips)



- **Wafer yield:** % wafer that is chips
- **Die yield:** % chips that work
- Yield is increasingly non-binary - fast vs slow chips

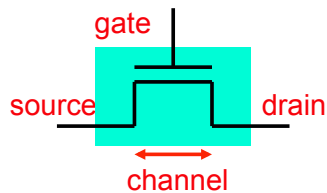


Manufacturing Cost

- **Chip cost vs system cost**
 - Cost of memory, storage, display, battery, etc.
- **Cost vs price**
 - Microprocessors not commodities; relationship complicated
 - Specialization, compatibility, different cost/performance/power
 - Economies of scale
- **Unit costs:** die manufacturing, testing, packaging, burn-in
 - Die cost based on area & defect rate (yield)
 - Package cost related to heat dissipation & number of pins
- **Fixed costs:** design & verification, fab cost
 - Amortized over "proliferations", e.g., Core i3, i5, i7 variants
 - Building new "fab" costs billions of dollars today
 - Both getting worse; trend toward "foundry" & "fabless" models

Technology Scaling Trends

Moore's Law: Technology Scaling



- **Moore's Law:** aka "technology scaling"
 - Continued miniaturization (esp. reduction in channel length)
 - + Improves switching speed, power/transistor, area(cost)/transistor
 - Reduces transistor reliability
 - Literally: DRAM density (transistors/area) doubles every 18 months
 - Public interpretation: performance doubles every 18 months
 - Not quite right, but helps performance in several ways...

Moore's Effect #1: Transistor Count

- Linear shrink in each dimension
 - 180nm, 130nm, 90nm, 65nm, 45nm, 32nm, 22nm, ...
 - Each generation is a 1.414 linear shrink
 - Shrink each dimension (2D)
 - Results in 2x more transistors (1.414×1.414) per area
- Generally reduces cost per transistor
- More transistors can increase performance
 - Job of a computer architect: use the ever-increasing number of transistors
 - Today, desktop/laptop processor chips have ~ 1 billion transistors

Moore's Effect #2: Delay

- **First-order: speed scales proportional to gate length**
 - Has provided much of the performance gains in the past
- Delay is proportional to: capacitance * resistance
- Scaling helps wire and gate delays in some ways...
 - + Transistors become shorter (Resistance \downarrow), narrower (Capacitance \downarrow)
 - + Wires become shorter (Length \downarrow \rightarrow Resistance \downarrow)
- Hurts in others...
 - Transistors become narrower (Resistance \uparrow)
 - Gate insulator thickness becomes smaller (Capacitance \uparrow)
 - Wires becomes thinner (Resistance \uparrow)
- What to do?
 - Take the good, use wire/transistor sizing to counter the bad
 - Exploit new materials: Aluminum \rightarrow Copper, metal gate, high-K

Moore's Effect #3: Cost

- Mixed impact on unit integrated circuit cost
 - + Either lower cost for same functionality...
 - + Or same cost for more functionality
 - Difficult to achieve high yields
- Increases startup cost
 - More expensive fabrication equipment
 - Takes longer to design, verify, and test chips
- Process variation across chip increasing
 - Some transistors slow, some fast
 - Increasingly active research area: dealing with this problem

Moore's Effect #4: Psychological

- **Moore's Curve**: common interpretation of Moore's Law
 - "CPU performance doubles every 18 months"
 - Self fulfilling prophecy: 2X every 18 months is $\sim 1\%$ per week
 - Q: Would you add a feature that improved performance 20% if it would delay the chip 8 months?
 - Processors under Moore's Curve (arrive too late) fail spectacularly
 - E.g., Intel's Itanium, Sun's Millennium

Moore's Law in the Future

- Won't last forever, approaching physical limits
 - "If something must eventually stop, it can't go on forever"
 - But betting against it has proved foolish in the past
 - Perhaps will "slow" rather than stop abruptly
- Transistor count will likely continue to scale
 - "Die stacking" is on the cusp of becoming main stream
 - Uses the third dimension to increase transistor count
- But transistor performance scaling?
 - Running into physical limits
 - Example: gate oxide is less than 10 silicon atoms thick!
 - Can't decrease it much further
 - Power is becoming the limiting factor

Power & Energy



Power/Energy Are Increasingly Important

- **Battery life** for mobile devices
 - Laptops, phones, cameras
- **Tolerable temperature** for devices without active cooling
 - Power means temperature, active cooling means **cost**
 - No room for a fan in a cell phone, no market for a hot cell phone
- **Electric bill** for compute/data centers
 - Pay for power twice: once in, once out (to cool)
- **Environmental concerns**
 - "Computers" account for growing fraction of energy consumption

Energy & Power

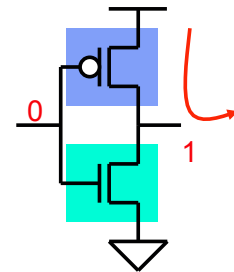
- **Energy**: measured in Joules or Watt-seconds
 - Total amount of energy stored/used
 - Battery life, electric bill, environmental impact
 - Instructions per Joule (car analogy: miles per gallon)
- **Power**: energy per unit time (measured in Watts)
 - Related to "performance" (which is also a "per unit time" metric)
 - Power impacts power supply and cooling requirements (cost)
 - Power-density (Watt/mm²): important related metric
 - Peak power vs average power
 - E.g., camera, power "spikes" when you actually take a picture
 - Joules per second (car analogy: gallons per hour)
- Two sources:
 - **Dynamic power**: active switching of transistors
 - **Static power**: leakage of transistors even while inactive

Technology Basis of Transistor Speed

- Physics 101: delay through an electrical component $\sim RC$
 - **Resistance (R)**  $\sim \text{length} / \text{cross-section area}$
 - Slows rate of charge flow
 - **Capacitance (C)**  $\sim \text{length} * \text{area} / \text{distance-to-other-plate}$
 - Stores charge
 - **Voltage (V)**
 - Electrical pressure
 - **Threshold Voltage (V_t)**
 - Voltage at which a transistor turns "on"
 - Property of transistor based on fabrication technology
 - **Switching time $\sim t_o (R * C) / (V - V_t)$**
- Components contribute to capacitance & resistance
 - Transistors
 - Wires (longer the wire, the more the capacitance & resistance)

Dynamic Power

- **Dynamic power (P_{dynamic}):** aka switching or active power
 - Energy to switch a gate (0 to 1, 1 to 0)
 - Each gate has capacitance (C)
 - Charge stored is $\propto C * V$
 - Energy to charge/discharge a capacitor is $\propto C * V^2$
 - Time to charge/discharge a capacitor is $\propto C / V$
 - Result: frequency $\sim 1 / C$
 - **$P_{\text{dynamic}} \approx N * C * V^2 * f * A$**
 - N: number of transistors
 - C: capacitance per transistor (size of transistors)
 - V: voltage (supply voltage for gate)
 - f: frequency (transistor switching freq. is \propto to clock freq.)
 - A: activity factor (not all transistors may switch this cycle)

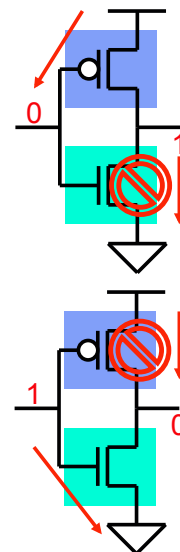


Reducing Dynamic Power

- Target each component: $P_{\text{dynamic}} \approx N * C * V^2 * f * A$
- **Reduce number of transistors (N)**
 - Use fewer transistors/gates
- **Reduce capacitance (C)**
 - Smaller transistors (Moore's law)
- **Reduce voltage (V)**
 - Quadratic reduction in energy consumption!
 - But also slows transistors (transistor speed is \sim to V)
- **Reduce frequency (f)**
 - Slower clock frequency (reduces power but not energy) Why?
- **Reduce activity (A)**
 - "Clock gating" disable clocks to unused parts of chip
 - Don't switch gates unnecessarily

Static Power

- **Static power (P_{static}):** aka idle or leakage power
 - Transistors don't turn off all the way
 - Transistors "leak"
 - Analogy: leaky valve
 - $P_{\text{static}} \approx N * V * e^{-V_t}$
 - N: number of transistors
 - V: voltage
 - **V_t (threshold voltage):** voltage at which transistor conducts (begins to switch)
- Switching speed vs leakage trade-off
- The lower the **V_t** :
 - Faster transistors (linear)
 - Transistor speed \propto to $V - V_t$
 - Leakier transistors (exponential)



Reducing Static Power

- Target each component: $P_{\text{static}} \approx N * V * e^{-Vt}$
- **Reduce number of transistors** (N)
 - Use fewer transistors/gates
- **Disable transistors** (also targets N)
 - "Power gating" disable power to unused parts (long latency to power up)
 - Power down units (or entire cores) not being used
- **Reduce voltage** (V)
 - Linear reduction in static energy consumption
 - But also slows transistors (transistor speed is \sim to V)
- **Dual V_t** – use a mixture of high and low V_t transistors
 - Use slow, low-leak transistors in SRAM arrays
 - Requires extra fabrication steps (cost)
- **Low-leakage transistors**
 - High-K/Metal-Gates in Intel's 45nm process, "tri-gate" in Intel's 22nm
- Reducing frequency can hurt energy efficiency due to leakage power

Continuation of Moore's Law

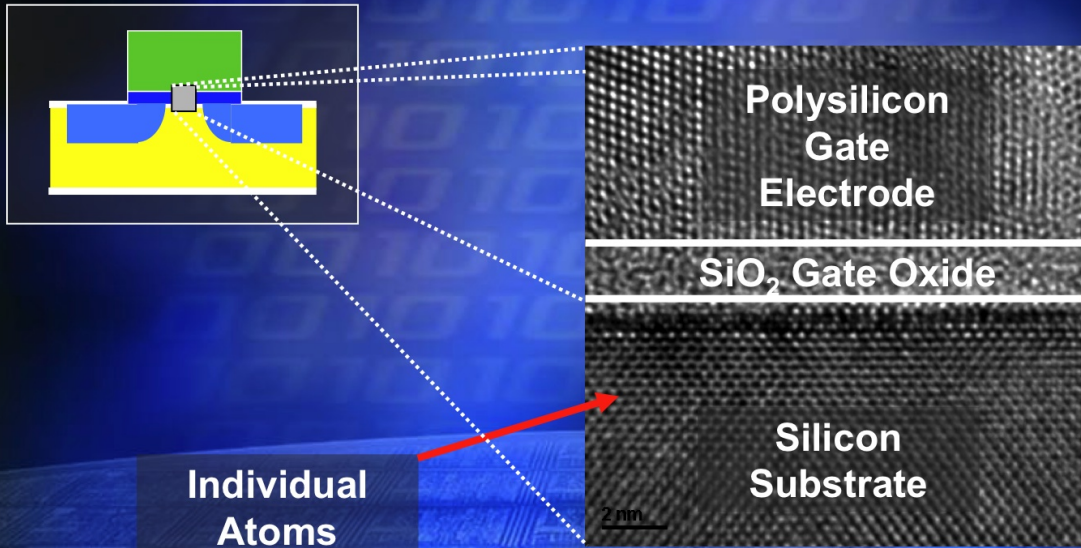
Process Name	P856	P858	Px60	P1262	P1264	P1266	P1268	P1270
1st Production	1997	1999	2001	2003	2005	2007	2009	2011
Process Generation	0.25 μ m	0.18 μ m	0.13 μ m	90 nm	65 nm	45 nm	32 nm	22 nm
Wafer Size (mm)	200	200	200/300	300	300	300	300	300
Inter-connect	Al	Al	Cu	Cu	Cu	Cu	Cu	?
Channel	Si	Si	Si	Strained Si	Strained Si	Strained Si	Strained Si	Strained Si
Gate dielectric	SiO ₂	SiO ₂	SiO ₂	SiO ₂	SiO ₂	High-k	High-k	High-k
Gate electrode	Poly-silicon	Poly-silicon	Poly-silicon	Poly-silicon	Poly-silicon	Metal	Metal	Metal

Introduction targeted at this time

Subject to change

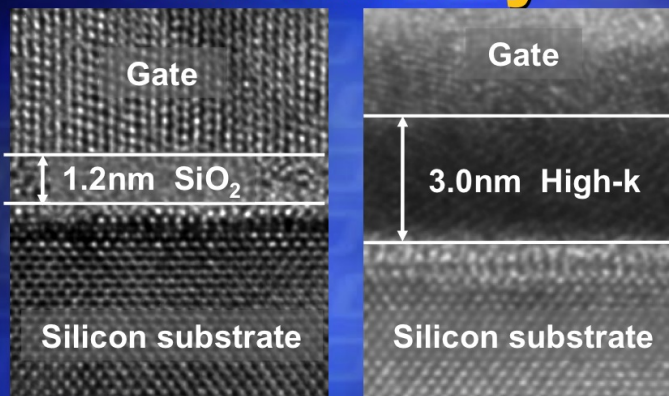
Intel found a solution for High-k and metal gate

Gate dielectric today is only a few molecular layers thick



intel.

High-k Dielectric reduces leakage substantially



Benefits compared to current process technologies

	High-k vs. SiO ₂	Benefit
Capacitance	60% greater	<i>Much faster transistors</i>
Gate dielectric leakage	> 100x reduction	<i>Far cooler</i>

intel.

Dynamic Voltage/Frequency Scaling

- **Dynamically trade-off power for performance**
 - Change the voltage and frequency at runtime
 - Under control of operating system
- Recall: $P_{\text{dynamic}} \approx N * C * V^2 * f * A$
 - Because frequency \propto to $V - V_t$...
 - $P_{\text{dynamic}} \propto$ to $V^2(V - V_t) \approx V^3$
- Reduce both voltage and frequency linearly
 - **Cubic decrease in dynamic power**
 - Linear decrease in performance (actually sub-linear)
 - Thus, only about quadratic in energy
 - Linear decrease in static power
 - Thus, static energy can become dominant
- Newer chips can adjust frequency on a per-core basis

Dynamic Voltage/Frequency Scaling

	Mobile PentiumIII "SpeedStep"	Transmeta 5400 "LongRun"	Intel X-Scale (StrongARM2)
f (MHz)	300–1000 (step=50)	200–700 (step=33)	50–800 (step=50)
V (V)	0.9–1.7 (step=0.1)	1.1–1.6V (cont)	0.7–1.65 (cont)
High-speed	3400MIPS @ 34W	1600MIPS @ 2W	800MIPS @ 0.9W
Low-power	1100MIPS @ 4.5W	300MIPS @ 0.25W	62MIPS @ 0.01W

- Dynamic voltage/frequency scaling
 - **Favors parallelism**
- Example: Intel Xscale
 - 1 GHz \rightarrow 200 MHz reduces energy used by 30x
 - But around 5x slower
 - 5 x 200 MHz in parallel, use **1/6th the energy**
 - Power is driving the trend toward multi-core

Moore's Effect on Power

- + Moore's Law reduces power/transistor...
 - Reduced sizes and surface areas reduce capacitance (C)
- ...but increases power density and total power
 - By increasing transistors/area and total transistors
 - Faster transistors → higher frequency → more power
 - Hotter transistors leak more (thermal runaway)
- What to do? Reduce voltage (V)
 - + Reduces dynamic power quadratically, static power linearly
 - Already happening: Intel 486 (5V) → Core2 (1.3V)
 - Trade-off: reducing V means either...
 - Keeping V_t the same and reducing frequency (f)
 - Lowering V_t and increasing leakage exponentially
 - Use techniques like high-K and dual- V_T
- The end of voltage scaling & "dark silicon"

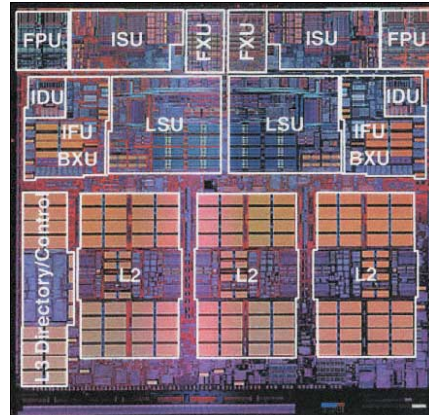
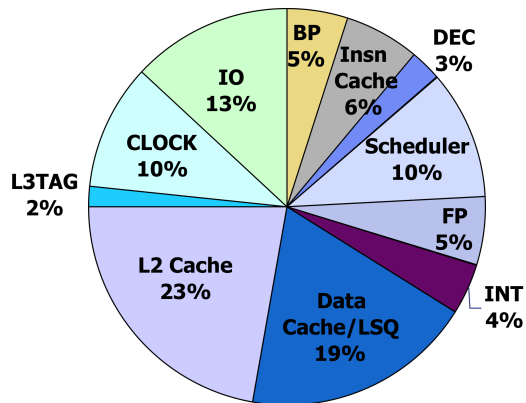
Trends in Power

	386	486	Pentium	Pentium II	Pentium4	Core2	Core i7
Year	1985	1989	1993	1998	2001	2006	2009
Technode (nm)	1500	800	350	180	130	65	45
Transistors (M)	0.3	1.2	3.1	5.5	42	291	731
Voltage (V)	5	5	3.3	2.9	1.7	1.3	1.2
Clock (MHz)	16	25	66	200	1500	3000	3300
Power (W)	1	5	16	35	80	75	130
Peak MIPS	6	25	132	600	4500	24000	52800
MIPS/W	6	5	8	17	56	320	406

- Supply voltage decreasing over time
 - But "voltage scaling" is perhaps reaching its limits
- Emphasis on power starting around 2000
 - Resulting in slower frequency increases
 - Also note number of cores increasing (2 in Core 2, 4 in Core i7)

Processor Power Breakdown

- Power breakdown for IBM POWER4
 - Two 4-way superscalar, 2-way multi-threaded cores, 1.5MB L2
 - Big power components are L2, data cache, scheduler, clock, I/O
 - Implications on “complicated” versus “simple” cores



Implications on Software

- Software-controlled dynamic voltage/frequency scaling
 - Example: video decoding
 - Too high a clock frequency – wasted energy (battery life)
 - Too low a clock frequency – quality of video suffers
 - “Race to sleep” versus “slow and steady” approaches
- Managing low-power modes
 - Don’t want to “wake up” the processor every millisecond
- Tuning software
 - Faster algorithms can be converted to lower-power algorithms
 - Via dynamic voltage/frequency scaling
- Exploiting parallelism & heterogeneous cores
 - NVIDIA Tegra 3: 5 cores (4 “normal” cores & 1 “low power” core)
- Specialized hardware accelerators

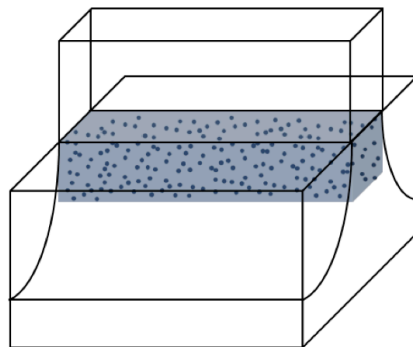
Recent Technology Update From Intel

Computer Architecture | Prof. Milo Martin | Technology & Energy

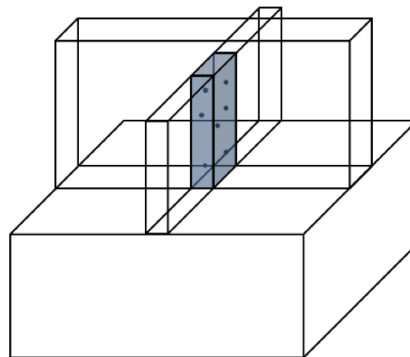
37

Reduced Channel Doping

Planar

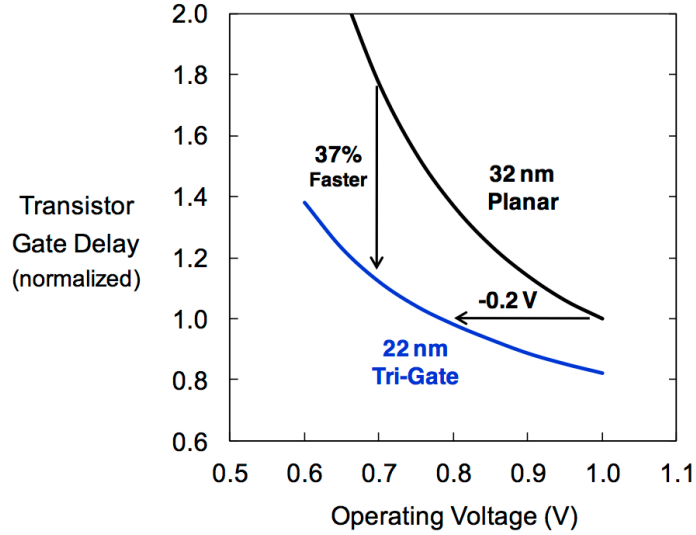


Tri-Gate



***Fully depleted Tri-Gate structure has reduced channel doping,
providing improved performance and reduced variability***

Performance/Power Benefits

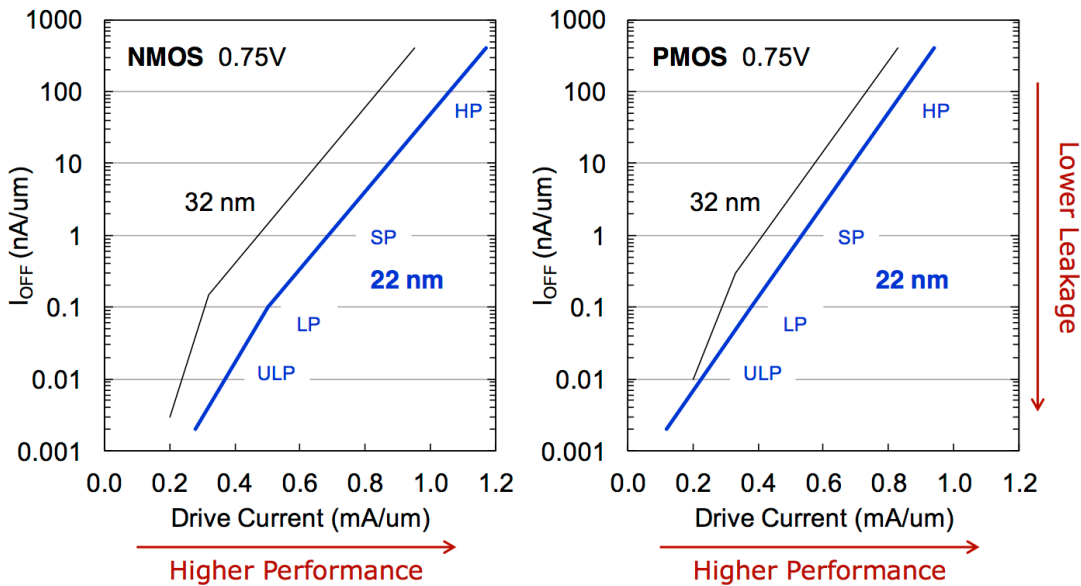


Tri-Gate provides 37% speed up at low voltage or 50% active power reduction at same performance

IDF2012
INTEL DEVELOPER FORUM

21

Transistor Performance vs. Leakage



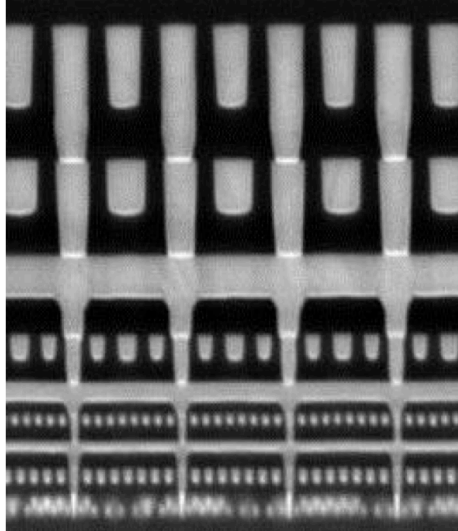
22 nm SoC technology offers a wide range of transistors

IDF2012
INTEL DEVELOPER FORUM

34

22 nm Interconnects

Layer	Pitch
TM	14 μm
M8	360 nm
M7	320 nm
M6	240 nm
M5	160 nm
M4	112 nm
M3	80 nm
M2	80 nm
M1	90 nm

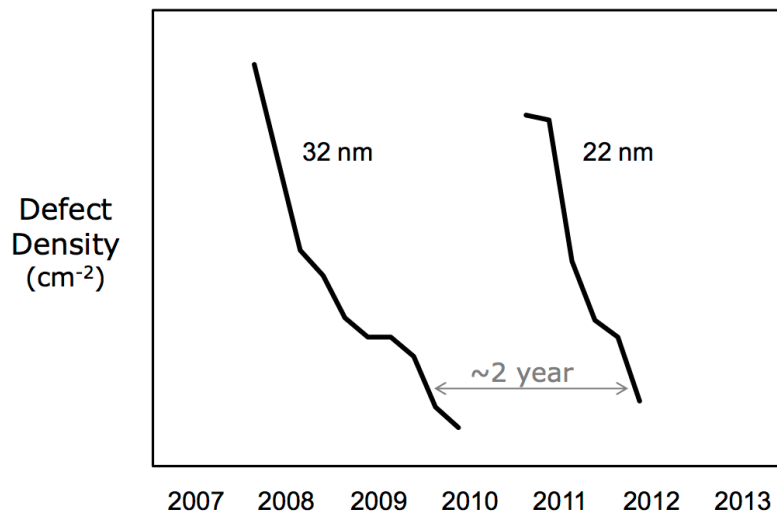


Minimum pitch scaled $\sim 0.7x$ from 32 nm for $\sim 2x$ transistor density improvement

IDF2012
INTEL DEVELOPER FORUM

24

22 nm Defect Density Trend

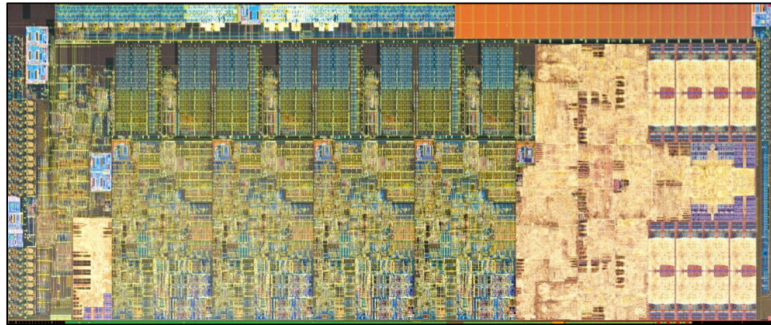


22 nm defect density now at low levels needed for volume manufacturing

IDF2012
INTEL DEVELOPER FORUM

28

3RD Generation Intel® Core™ Processor



22 nm Tri-Gate Technology
4 Cores + Integrated Graphics
1.4 Billion Transistors, 160 mm²

27

IDF2012
INTEL DEVELOPER FORUM

Summary

Technology Summary

- Has a first-order impact on computer architecture
 - Performance (transistor delay, wire delay)
 - Cost (die area & defects)
 - **Changing rapidly**
- Most significant trends for architects
 - More and more transistors
 - What to do with them? → integration → **parallelism**
 - Logic is improving faster than memory & cross-chip wires
 - “Memory wall” → caches, more integration
- Power and energy
 - Voltage vs frequency, parallelism, special-purpose hardware
- This unit: a quick overview, just scratching the surface

} Rest of course

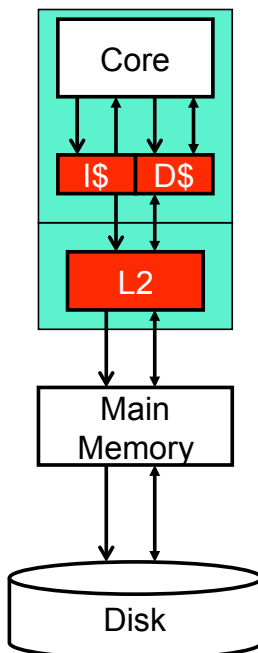
[spacer]

Computer Architecture

Unit 4: Caches

Slides developed by Milo Martin & Amir Roth at the University of Pennsylvania
with sources that included University of Wisconsin slides
by Mark Hill, Guri Sohi, Jim Smith, and David Wood

This Unit: Caches



- “Cache”: hardware managed
 - Hardware automatically retrieves missing data
 - Built from fast on-chip SRAM
 - In contrast to off-chip, DRAM “main memory”
- **Average access time** of a memory component
 - $latency_{avg} = latency_{hit} + (\%_{miss} * latency_{miss})$
 - Hard to get low $latency_{hit}$ and $\%_{miss}$ in one structure
→ memory hierarchy
- Cache ABCs (**associativity, block size, capacity**)
- **Performance optimizations**
 - Prefetching & data restructuring
- **Handling writes**
 - Write-back vs. write-through
- **Memory hierarchy**
 - Smaller, faster, expensive → bigger, slower, cheaper

Reading

- Assigned Reading
 - “Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers”,
by Norm Jouppi, ISCA 1990

Motivation

- Processor can compute only as fast as memory
 - A 3Ghz processor can execute an “add” operation in 0.33ns
 - Today’s “main memory” latency is more than 33ns
 - Naïve implementation:
 - loads/stores can be 100x slower than other operations
- Unobtainable goal:
 - Memory that operates at processor speeds
 - Memory as large as needed for all running programs
 - Memory that is cost effective
- Can’t achieve all of these goals at once
 - Example: latency of an SRAM is at least: $\sqrt{\text{number of bits}}$

Memories (SRAM & DRAM)

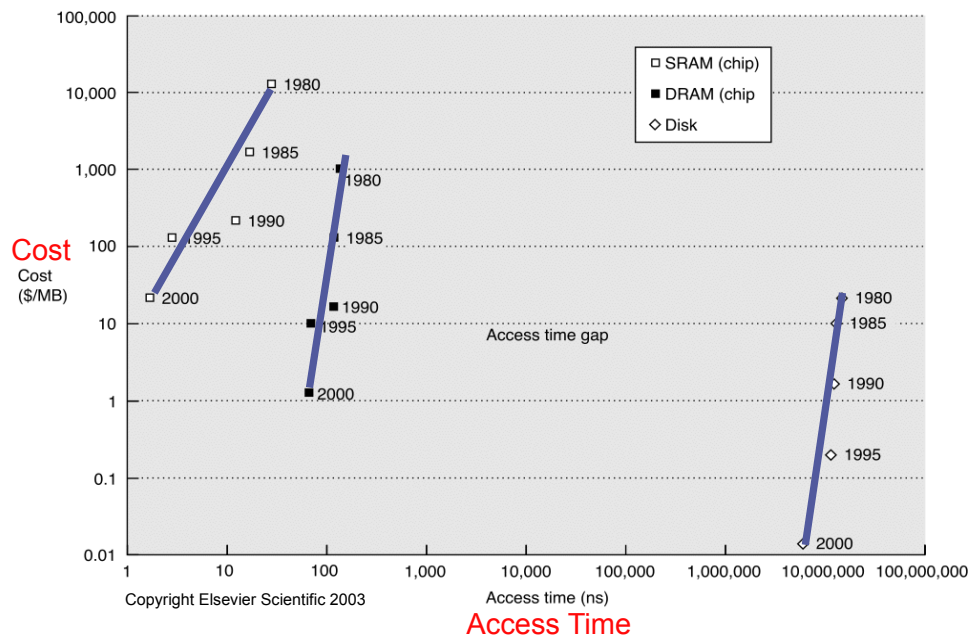
Types of Memory

- **Static RAM (SRAM)**
 - 6 or 8 transistors per bit
 - Two inverters (4 transistors) + transistors for reading/writing
 - Optimized for speed (first) and density (second)
 - Fast (sub-nanosecond latencies for small SRAM)
 - Speed roughly proportional to its area ($\sim \sqrt{\text{number of bits}}$)
 - Mixes well with standard processor logic
- **Dynamic RAM (DRAM)**
 - 1 transistor + 1 capacitor per bit
 - Optimized for density (in terms of cost per bit)
 - Slow ($>30\text{ns}$ internal access, $\sim 50\text{ns}$ pin-to-pin)
 - Different fabrication steps (does not mix well with logic)
- Nonvolatile storage: Magnetic disk, Flash RAM

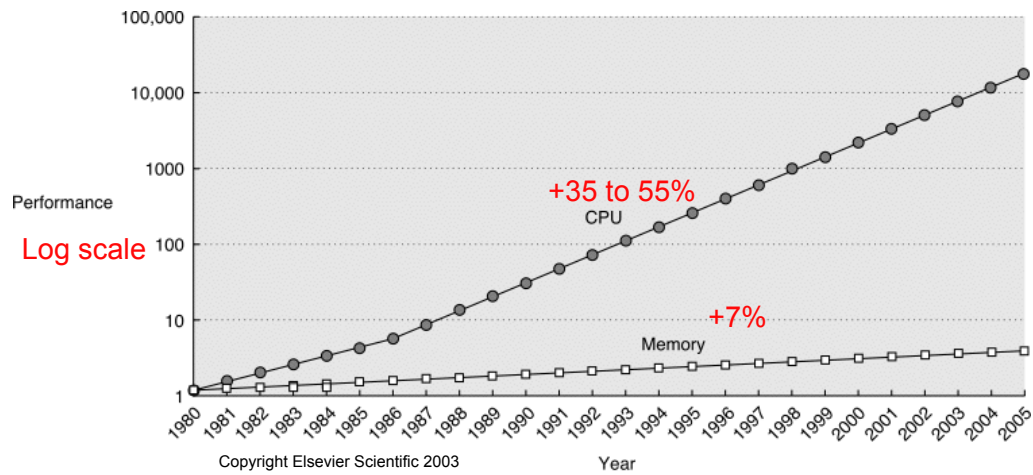
Memory & Storage Technologies

- **Cost** - what can \$200 buy (2009)?
 - SRAM: 16MB
 - DRAM: 4,000MB (4GB) – 250x cheaper than SRAM
 - Flash: 64,000MB (64GB) – 16x cheaper than DRAM
 - Disk: 2,000,000MB (2TB) – 32x vs. Flash (512x vs. DRAM)
- **Latency**
 - SRAM: <1 to 2ns (on chip)
 - DRAM: ~50ns – 100x or more slower than SRAM
 - Flash: 75,000ns (75 microseconds) – 1500x vs. DRAM
 - Disk: 10,000,000ns (10ms) – 133x vs Flash (200,000x vs DRAM)
- **Bandwidth**
 - SRAM: 300GB/sec (e.g., 12-port 8-byte register file @ 3Ghz)
 - DRAM: ~25GB/s
 - Flash: 0.25GB/s (250MB/s), 100x less than DRAM
 - Disk: 0.1 GB/s (100MB/s), 250x vs DRAM, **sequential** access only

Memory Technology Trends



The "Memory Wall"



- Processors get faster more quickly than memory (note log scale)
 - Processor speed improvement: 35% to 55%
 - Memory latency improvement: 7%

The Memory Hierarchy

Known From the Beginning

“Ideally, one would desire an infinitely large memory capacity such that any particular word would be immediately available ... We are forced to recognize the possibility of constructing a hierarchy of memories, each of which has a greater capacity than the preceding but which is less quickly accessible.”

Burks, Goldstine, VonNeumann

“Preliminary discussion of the logical design of an electronic computing instrument”

IAS memo 1946

Big Observation: Locality & Caching

- **Locality of memory references**
 - Empirical property of real-world programs, few exceptions
- **Temporal locality**
 - Recently referenced data is likely to be referenced again soon
 - **Reactive:** “cache” recently used data in small, fast memory
- **Spatial locality**
 - More likely to reference data near recently referenced data
 - **Proactive:** “cache” large chunks of data to include nearby data
- Both properties hold for data and instructions
- Cache: “Hashtable” of recently used blocks of data
 - In hardware, finite-sized, transparent to software

Spatial and Temporal Locality Example

- Which memory accesses demonstrate spatial locality?
- Which memory accesses demonstrate temporal locality?

```
int sum = 0;
int x[1000];

for(int c = 0; c < 1000; c++){
    sum += c;

    x[c] = 0;
}
```

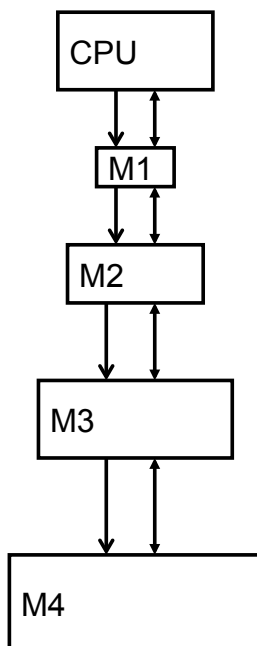
Library Analogy

- Consider books in a library
- Library has lots of books, but it is slow to access
 - Far away (time to walk to the library)
 - Big (time to walk within the library)
- How can you avoid these latencies?
 - Check out books, take them home with you
 - Put them on desk, on bookshelf, etc.
 - But desks & bookshelves have limited capacity
 - Keep recently used books around (**temporal locality**)
 - Grab books on related topic at the same time (**spatial locality**)
 - Guess what books you'll need in the future (prefetching)

Library Analogy Explained

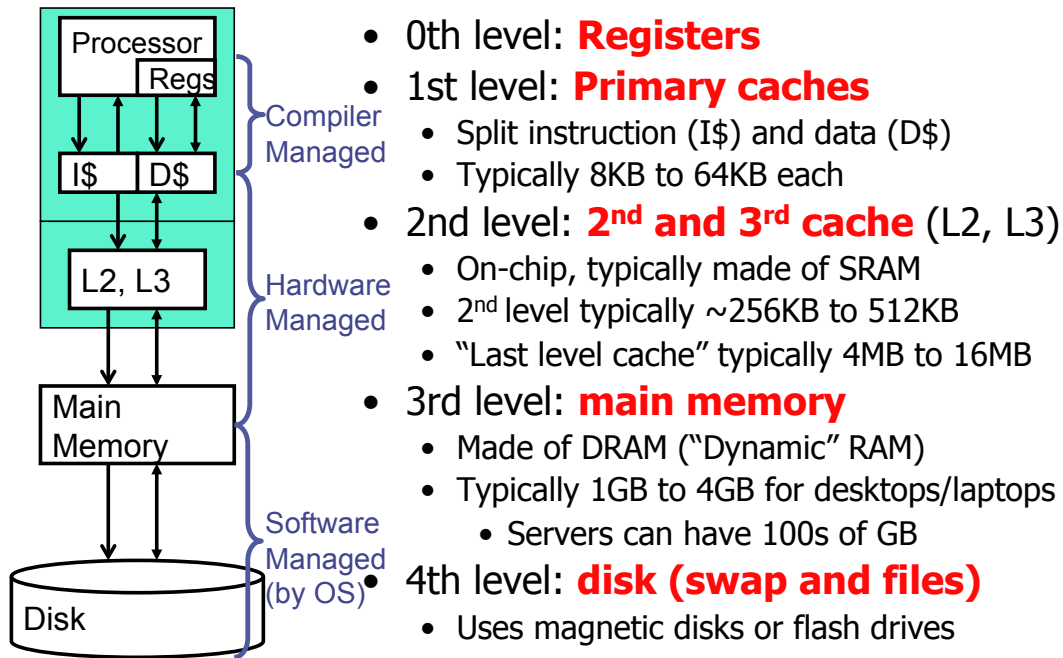
- Registers ↔ books on your desk
 - Actively being used, small capacity
- Caches ↔ bookshelves
 - Moderate capacity, pretty fast to access
- Main memory ↔ library
 - Big, holds almost all data, but slow
- Disk (virtual memory) ↔ inter-library loan
 - Very slow, but hopefully really uncommon

Exploiting Locality: Memory Hierarchy

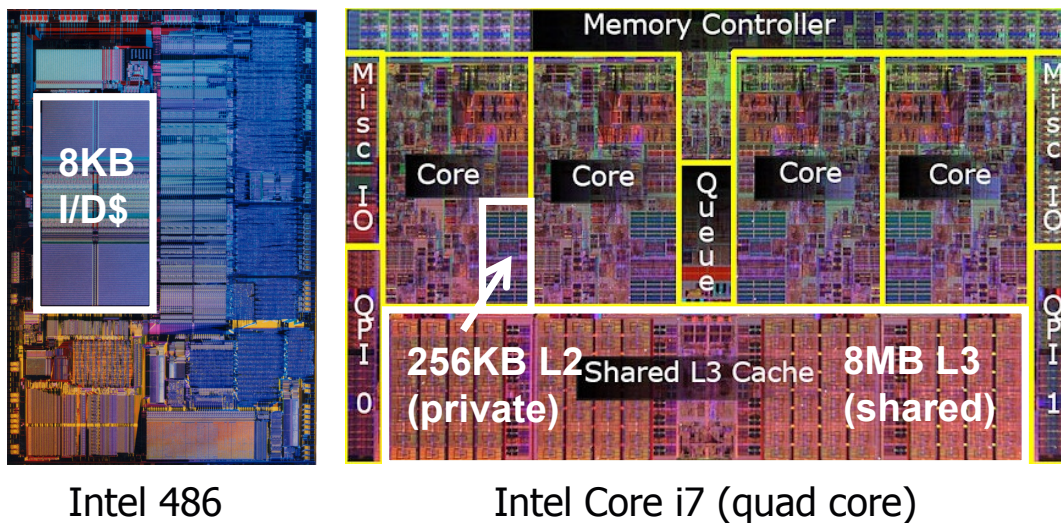


- Hierarchy of memory components
 - Upper components
 - Fast ↔ Small ↔ Expensive
 - Lower components
 - Slow ↔ Big ↔ Cheap
- Connected by “buses”
 - Which also have latency and bandwidth issues
- Most frequently accessed data in M1
 - M1 + next most frequently accessed in M2, etc.
 - Move data up-down hierarchy
- Optimize average access time
 - $latency_{avg} = latency_{hit} + (\%_{miss} * latency_{miss})$
 - Attack each component

Concrete Memory Hierarchy



Evolution of Cache Hierarchies



- Chips today are 30–70% cache by area

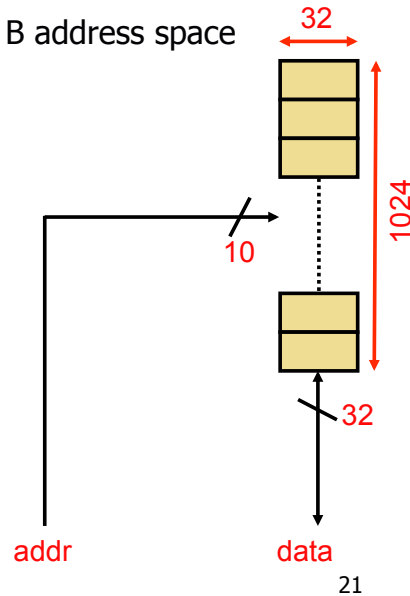
Caches

Analogy to a Software Hashtable

- What is a "hash table"?
 - What is it used for?
 - How does it work?
- Short answer:
 - Maps a "key" to a "value"
 - Constant time lookup/insert
 - Have a table of some size, say N , of "buckets"
 - Take a "key" value, apply a hash function to it
 - Insert and lookup a "key" at "hash(key) modulo N "
 - Need to store the "key" and "value" in each bucket
 - Need to check to make sure the "key" matches
 - Need to handle conflicts/overflows somehow (chaining, re-hashing)

Hardware Cache Organization

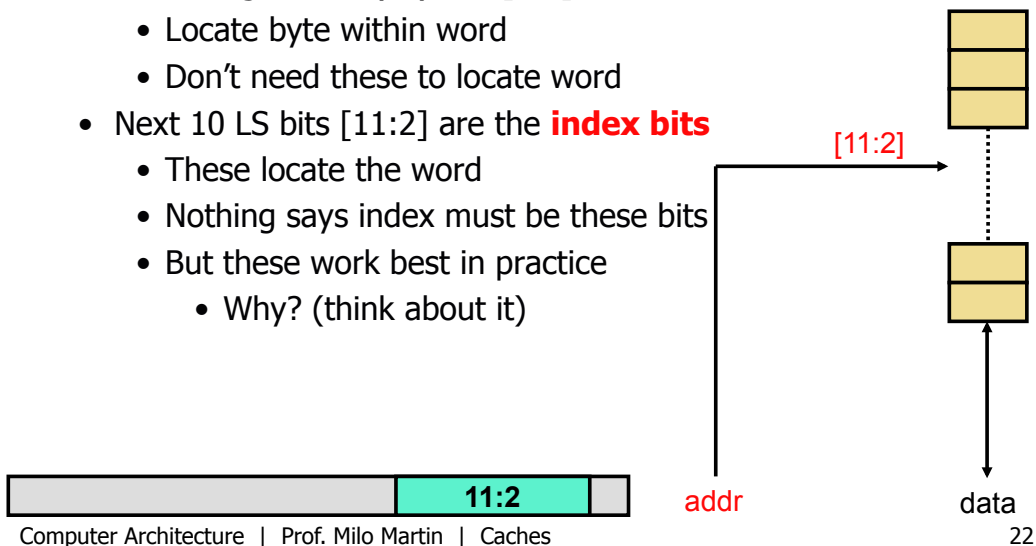
- **Cache is a hardware hashtable**
- The setup
 - 32-bit ISA → 4B words/addresses, 2^{32} B address space
- Logical cache organization
 - 4KB, organized as 1K 4B **blocks**
 - Each block can hold a 4-byte word
- Physical cache implementation
 - 1K (1024 bit) by 4B **SRAM**
 - Called **data array**
 - 10-bit address input
 - 32-bit data input/output



Computer Architecture | Prof. Milo Martin | Caches

Looking Up A Block

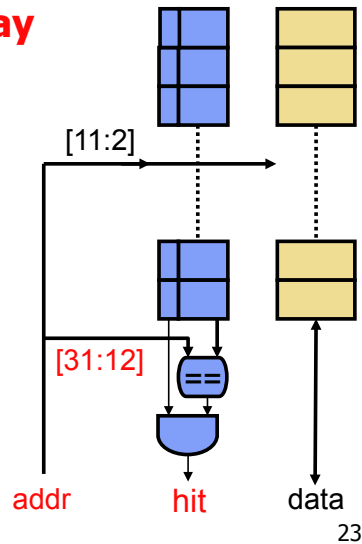
- Q: which 10 of the 32 address bits to use?
- A: bits [11:2]
 - 2 least significant (LS) bits [1:0] are the **offset bits**
 - Locate byte within word
 - Don't need these to locate word
 - Next 10 LS bits [11:2] are the **index bits**
 - These locate the word
 - Nothing says index must be these bits
 - But these work best in practice
 - Why? (think about it)



Computer Architecture | Prof. Milo Martin | Caches

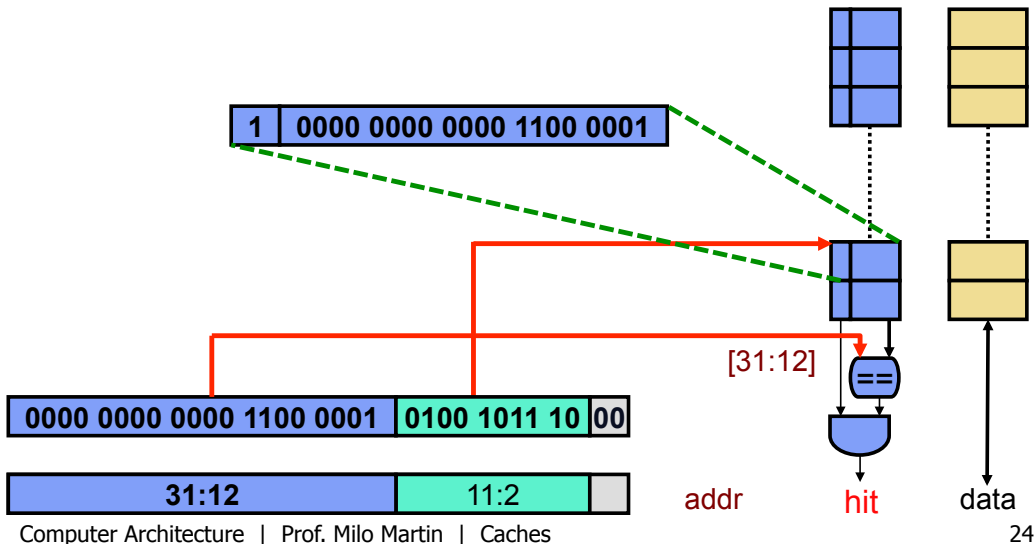
Knowing that You Found It

- Each cache row corresponds to 2^{20} blocks
 - How to know which if any is currently there?
 - Tag each cache word with remaining address bits [31:12]
- Build separate and parallel **tag array**
 - 1K by 21-bit SRAM
 - 20-bit (next slide) **tag** + 1 **valid bit**
- Lookup algorithm
 - Read tag indicated by index bits
 - If tag matches & valid bit set:
 - then: Hit → data is good
 - else: Miss → data is garbage, wait...



A Concrete Example

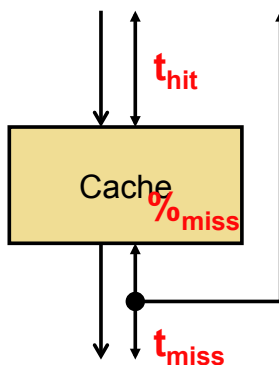
- Lookup address $x000C14B8$
 - Index = $\text{addr}[11:2] = (\text{addr} \gg 2) \& x7FF = x12E$
 - Tag = $\text{addr}[31:12] = (\text{addr} \gg 12) = x000C1$



Handling a Cache Miss

- What if requested data isn't in the cache?
 - How does it get in there?
- **Cache controller**: finite state machine
 - Remembers miss address
 - Accesses next level of memory
 - Waits for response
 - Writes data/tag into proper locations
- All of this happens on the **fill path**
- Sometimes called **backside**

Cache Performance Equation



- For a cache
 - **Access**: read or write to cache
 - **Hit**: desired data found in cache
 - **Miss**: desired data not found in cache
 - Must get from another component
 - No notion of "miss" in register file
 - **Fill**: action of placing data into cache
- $\%_{miss}$ (miss-rate): $\#misses / \#accesses$
- t_{hit} : time to read data from (write data to) cache
- t_{miss} : time to read data into cache

- Performance metric: average access time

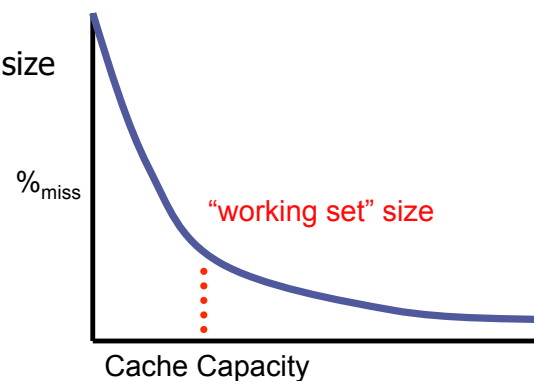
$$t_{avg} = t_{hit} + (\%_{miss} * t_{miss})$$

Measuring Cache Performance

- Ultimate metric is t_{avg}
 - Cache capacity and circuits roughly determines t_{hit}
 - Lower-level memory structures determine t_{miss}
 - Measure $\%_{miss}$
 - Hardware performance counters
 - Simulation

Capacity and Performance

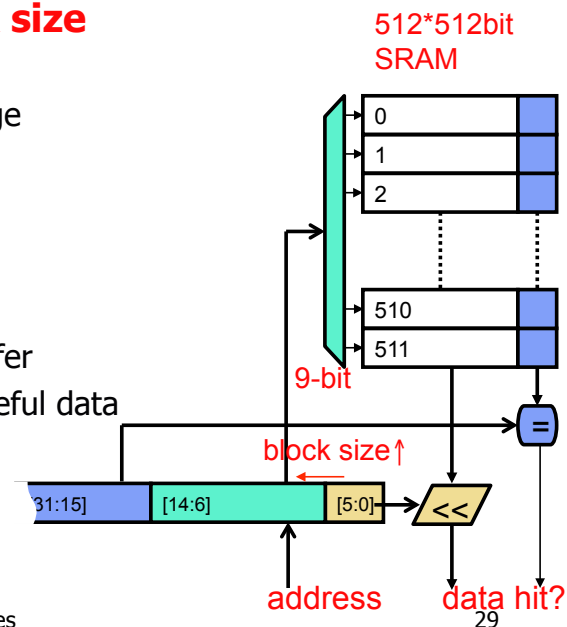
- Simplest way to reduce $\%_{miss}$: increase capacity
 - + Miss rate decreases monotonically
 - **“Working set”**: insns/data program is actively using
 - Diminishing returns
 - However t_{hit} increases
 - Latency grows with cache size
 - t_{avg} ?



- Given capacity, manipulate $\%_{miss}$ by changing **organization**

Block Size

- Given capacity, manipulate $\%_{\text{miss}}$ by changing organization
- One option: increase **block size**
 - Exploit **spatial locality**
 - Notice index/offset bits change
 - Tag remain the same
- Ramifications
 - + Reduce $\%_{\text{miss}}$ (up to a point)
 - + Reduce tag overhead (why?)
 - Potentially useless data transfer
 - Premature replacement of useful data
 - Fragmentation



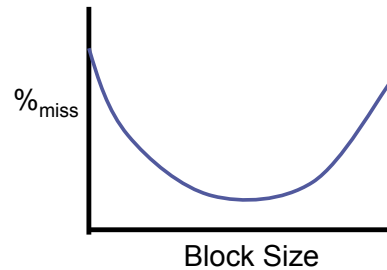
Computer Architecture | Prof. Milo Martin | Caches

Larger Blocks to Lower Tag Overhead

- Tag overhead of 32KB cache with 1024 32B frames
 - 32B frames \rightarrow 5-bit offset
 - 1024 frames \rightarrow 10-bit index
 - 32-bit address $-$ 5-bit offset $-$ 10-bit index = 17-bit tag
 - $(17\text{-bit tag} + 1\text{-bit valid}) * 1024 \text{ frames} = 18\text{Kb tags} = 2.2\text{KB tags}$
 - $\sim 6\%$ overhead
- Tag overhead of 32KB cache with 512 64B frames
 - 64B frames \rightarrow 6-bit offset
 - 512 frames \rightarrow 9-bit index
 - 32-bit address $-$ 6-bit offset $-$ 9-bit index = 17-bit tag
 - $(17\text{-bit tag} + 1\text{-bit valid}) * 512 \text{ frames} = 9\text{Kb tags} = 1.1\text{KB tags}$
 - + $\sim 3\%$ overhead

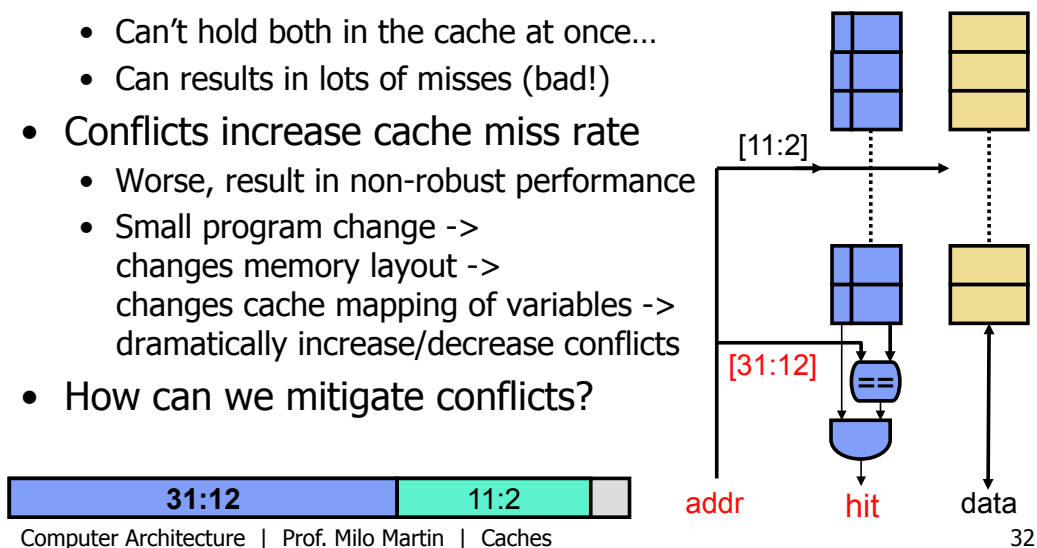
Effect of Block Size on Miss Rate

- Two effects on miss rate
 - + **Spatial prefetching (good)**
 - For blocks with adjacent addresses
 - Turns miss/miss into miss/hit pairs
 - **Interference (bad)**
 - For blocks with non-adjacent addresses (but in adjacent frames)
 - Turns hits into misses by disallowing simultaneous residence
 - Consider entire cache as one big block
- Both effects always present
 - Spatial “prefetching” dominates initially
 - Depends on size of the cache
 - Reasonable block sizes are 32B–128B
- But also increases traffic
 - More data moved, not all used



Cache Conflicts

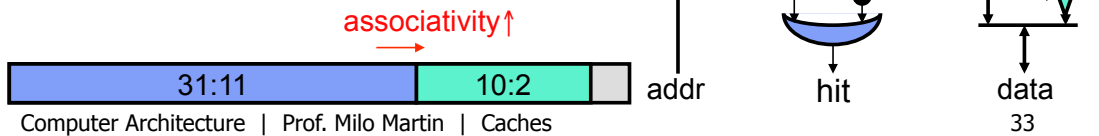
- Consider two frequently-accessed variables...
- What if their addresses have the same “index” bits?
 - Such addresses “conflict” in the cache
 - Can’t hold both in the cache at once...
 - Can result in lots of misses (bad!)
- Conflicts increase cache miss rate
 - Worse, result in non-robust performance
 - Small program change -> changes memory layout -> changes cache mapping of variables -> dramatically increase/decrease conflicts
- How can we mitigate conflicts?



Associativity

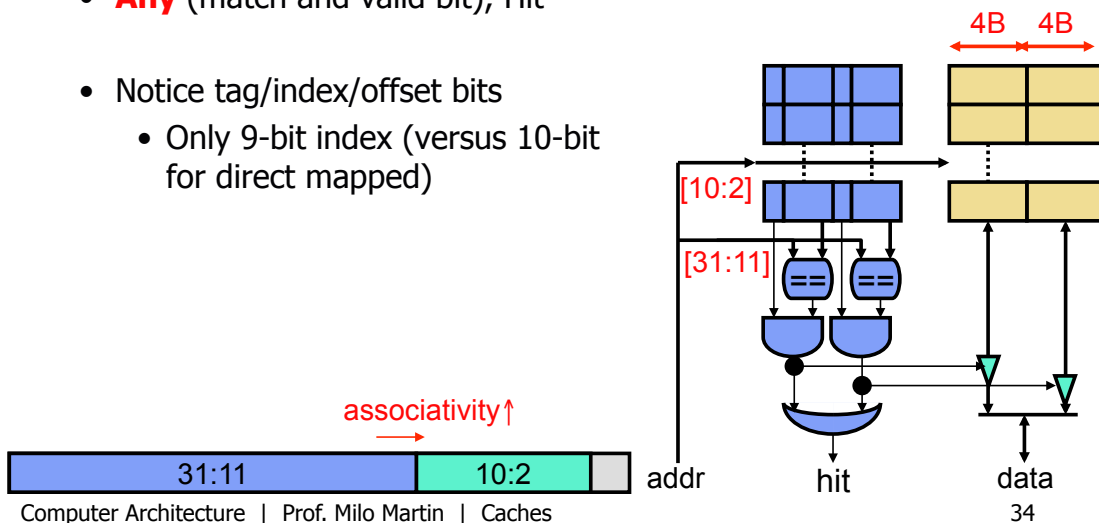
- **Set-associativity**
 - Block can reside in one of few frames
 - Frame groups called **sets**
 - Each frame in set called a **way**
 - This is **2-way set-associative (SA)**
 - 1-way → **direct-mapped (DM)**
 - 1-set → **fully-associative (FA)**

- + Reduces conflicts
- Increases latency_{hit}:
 - additional tag match & muxing



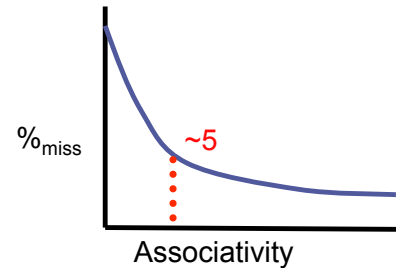
Associativity

- Lookup algorithm
 - Use index bits to find set
 - Read data/tags in all frames in parallel
 - **Any** (match and valid bit), Hit
- Notice tag/index/offset bits
 - Only 9-bit index (versus 10-bit for direct mapped)



Associativity and Performance

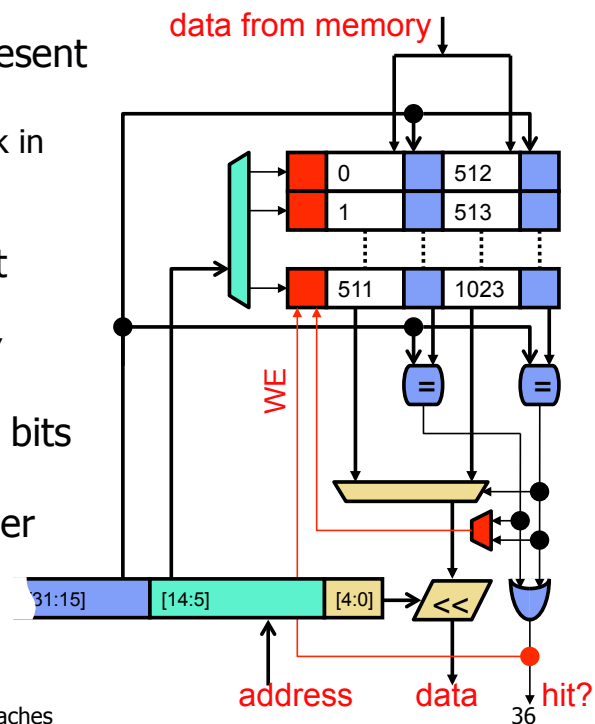
- Higher associative caches
 - + Have better (lower) $\%_{\text{miss}}$
 - Diminishing returns
 - However t_{hit} increases
 - The more associative, the slower
 - What about t_{avg} ?



- Block-size and number of sets should be powers of two
 - Makes indexing easier (just rip bits out of the address)
- 3-way set-associativity? No problem

Miss Handling & Replacement Policies

- Set-associative caches present a new design choice
 - On cache miss, which block in set to replace (kick out)?
- Add **LRU** field to each set
 - "Least recently used"
 - LRU data is encoded "way"
- Each access updates LRU bits
- Psudeo-LRU used for larger associativity caches



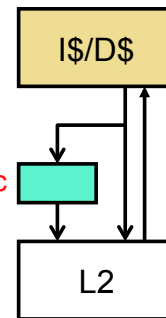
Replacement Policies

- Set-associative caches present a new design choice
 - On cache miss, which block in set to replace (kick out)?
- Some options
 - **Random**
 - **FIFO (first-in first-out)**
 - **LRU (least recently used)**
 - Fits with temporal locality, LRU = least likely to be used in future
 - **NMRU (not most recently used)**
 - An easier to implement approximation of LRU
 - Is LRU for 2-way set-associative caches
 - **Belady's**: replace block that will be used furthest in future
 - Unachievable optimum

Cache Optimizations

Prefetching

- Bring data into cache proactively/**speculatively**
 - If successful, reduces number of caches misses
- Key: anticipate upcoming miss addresses accurately
 - Can do in software or hardware
- Simple hardware prefetching: **next block prefetching**
 - Miss on address **X** → anticipate miss on **X+block-size**
 - + Works for insns: sequential execution
 - + Works for data: arrays
- Table-driven hardware prefetching
 - Use **predictor** to detect strides, common patterns
- Effectiveness determined by:
 - **Timeliness**: initiate prefetches sufficiently in advance
 - **Coverage**: prefetch for as many misses as possible
 - **Accuracy**: don't pollute with unnecessary data



Software Prefetching

- Use a special "prefetch" instruction
 - Tells the hardware to bring in data, doesn't actually read it
 - Just a hint
- Inserted by programmer or compiler
- Example

```
int tree_add(tree_t* t) {
    if (t == NULL) return 0;
    __builtin_prefetch(t->left);
    return t->val + tree_add(t->right) + tree_add(t->left);
}
```
- 20% performance improvement for large trees (>1M nodes)
 - But ~15% slowdown for small trees (<1K nodes)
- Multiple prefetches bring multiple blocks in parallel
 - More "memory-level" parallelism (MLP)

Software Restructuring: Data

- Capacity misses: poor spatial or temporal locality
 - Several code restructuring techniques to improve both
 - Loop blocking (break into cache-sized chunks), loop fusion
 - Compiler must know that restructuring preserves semantics
- **Loop interchange**: spatial locality
 - Example: row-major matrix: `x[i][j]` followed by `x[i][j+1]`
 - Poor code: `x[i][j]` followed by `x[i+1][j]`

```
for (j = 0; j<NCOLS; j++)
  for (i = 0; i<NROWS; i++)
    sum += X[i][j];
```
 - Better code

```
for (i = 0; i<NROWS; i++)
  for (j = 0; j<NCOLS; j++)
    sum += X[i][j];
```

Software Restructuring: Data

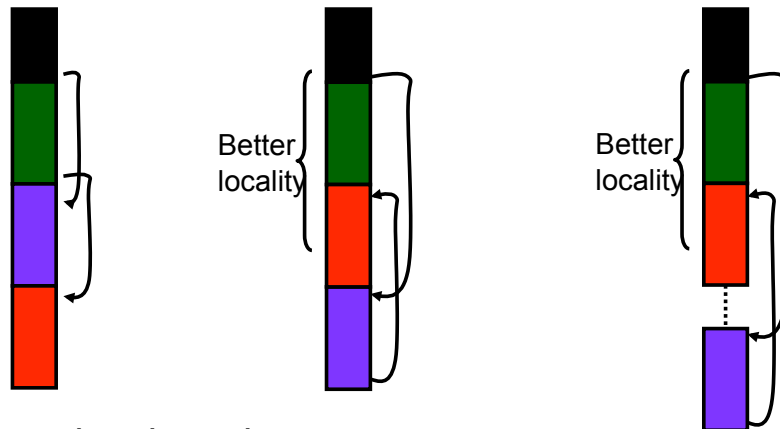
- **Loop "blocking" or "tiling"**: temporal locality
 - Consider following "image blur" code

```
for (i = 1; i<NROWS-1; i++)
  for (j = 1; j<NCOLS-1; j++)
    Y[i][j] = avg(X[i][j], X[i+1][j], X[i-1][j],
                  X[i][j+1], X[i][j-1]);
```
 - How many times is each block brought into the cache?
 - Answer: For a large image matrix, ~ 3 times
 - Why? exploits only spatial locality (not temporal locality)
- Can be reduced to ~ 1 time by exploiting temporal locality
 - Assume `NROWS-2` and `NCOLS-2` evenly divisible by `TILE_SIZE`

```
for (ii = 1; ii<NROWS-1; ii+=TILE_SIZE)
  for (jj = 1; jj<NCOLS-1; jj+=TILE_SIZE)
    for (i = ii; i<ii+TILE_SIZE; i++)
      for (j = jj; j<jj+TILE_SIZE; j++)
        Y[i][j] = avg(X[i][j], X[i+1][j], ...)
```

Software Restructuring: Code

- Compiler an layout code for temporal and spatial locality
 - If (a) { **code1;** } else { **code2;** } **code3;**
 - But, code2 case never happens (say, error condition)

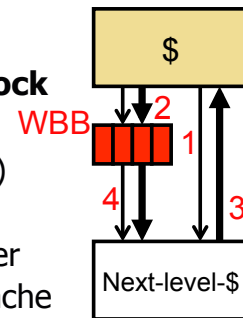


- Fewer taken branches, too

What About Stores? Handling Cache Writes

Handling Cache Writes

- When to propagate new value to (lower level) memory?
- **Option #1: Write-through**: immediately
 - On hit, update cache
 - Immediately send the write to the next level
- **Option #2: Write-back**: when block is replaced
 - Requires additional "dirty" bit per block
 - Replace **clean** block: **no extra traffic**
 - Replace **dirty** block: **extra "writeback" of block**
- + **Writeback-buffer (WBB)**:
 - Hide latency of writeback (keep off critical path)
 - Step#1: Send "fill" request to next-level
 - Step#2: While waiting, write dirty block to buffer
 - Step#3: When new blocks arrives, put it into cache
 - Step#4: Write buffer contents to next-level

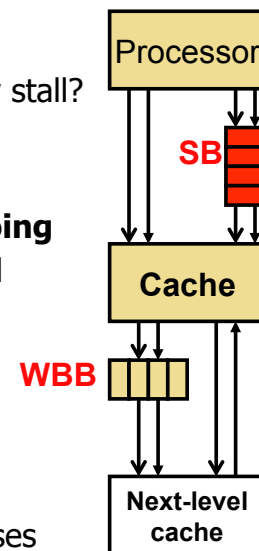


Write Propagation Comparison

- **Write-through**
 - Creates additional traffic
 - Consider repeated write hits
 - Next level must handle small writes (1, 2, 4, 8-bytes)
 - + No need for dirty bits in cache
 - + No need to handle "writeback" operations
 - Simplifies miss handling (no write-back buffer)
 - Sometimes used for L1 caches (for example, by IBM)
 - Usually **write-non-allocate**: on write miss, just write to next level
- **Write-back**
 - + Key advantage: uses less bandwidth
 - Reverse of other pros/cons above
 - Used by Intel, (AMD), and many ARM cores
 - Second-level and beyond are generally write-back caches
 - Usually **write-allocate**: on write miss, fill block from next level

Write Misses and Store Buffers

- Read miss?
 - Load can't go on without the data, it must stall
- Write miss?
 - Technically, no instruction is waiting for data, why stall?
- **Store buffer:** a small buffer
 - Stores put address/value to store buffer, **keep going**
 - Store buffer writes stores to D\$ in the background
 - Loads must search store buffer (in addition to D\$)
 - + Eliminates stalls on write misses (mostly)
 - Creates some problems (later)
- Store buffer vs. writeback-buffer
 - Store buffer: "in front" of D\$, for hiding store misses
 - Writeback buffer: "behind" D\$, for hiding writebacks

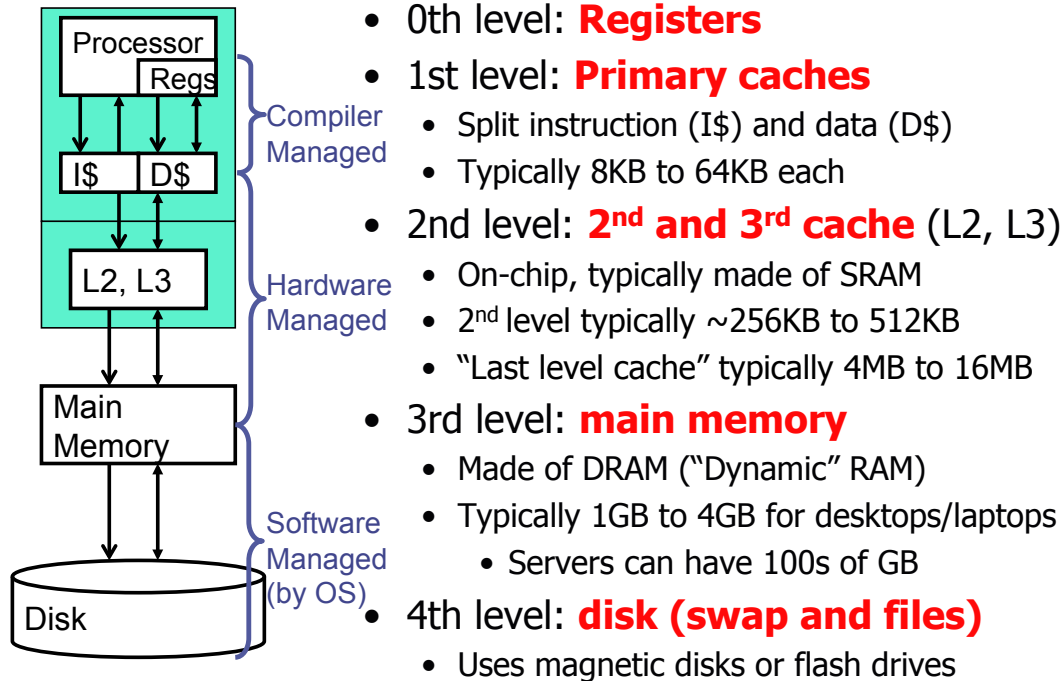


Computer Architecture | Prof. Milo Martin | Caches

47

Cache Hierarchies

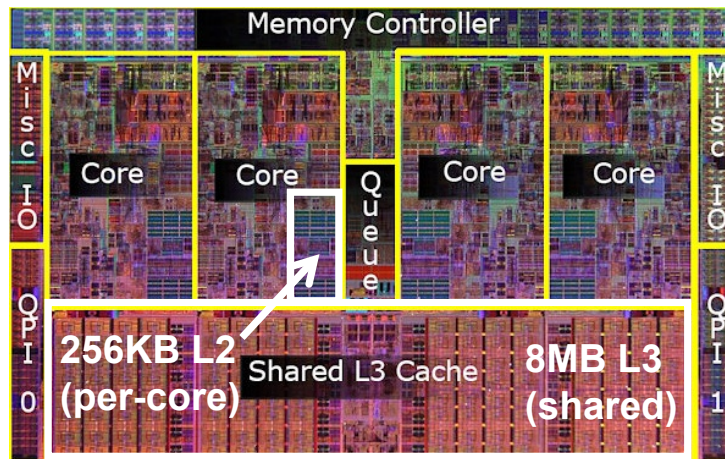
Concrete Memory Hierarchy



Designing a Cache Hierarchy

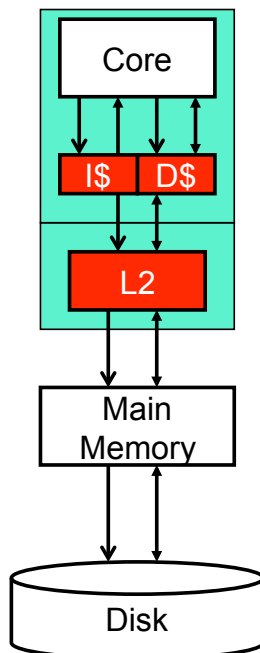
- For any memory component: t_{hit} vs. $\%_{miss}$ tradeoff
- Upper components (I\$, D\$) emphasize low t_{hit}
 - **Frequent access** → **t_{hit} important**
 - t_{miss} is not bad → $\%_{miss}$ less important
 - Lower capacity and lower associativity (to reduce t_{hit})
 - Small-medium block-size (to reduce conflicts)
 - **Split instruction & data cache to allow simultaneous access**
- Moving down (L2, L3) emphasis turns to $\%_{miss}$
 - **Infrequent access** → **t_{hit} less important**
 - t_{miss} is bad → $\%_{miss}$ important
 - High capacity, associativity, and block size (to reduce $\%_{miss}$)
 - **Unified insn & data caching to dynamic allocate capacity**

Example Cache Hierarchy: Core i7



- Each core:
 - 32KB insn & 32KB data, 8-way set-associative, 64-byte blocks
 - 256KB second-level cache, 8-way set-associative, 64-byte blocks
- 8MB shared cache, 16-way set-associative

Summary



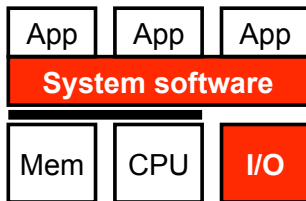
- “Cache”: hardware managed
 - Hardware automatically retrieves missing data
 - Built from fast on-chip SRAM
 - In contrast to off-chip, DRAM “main memory”
- **Average access time** of a memory component
 - $latency_{avg} = latency_{hit} + (\%_{miss} * latency_{miss})$
 - Hard to get low $latency_{hit}$ and $\%_{miss}$ in one structure → memory hierarchy
- Cache ABCs (**associativity, block size, capacity**)
- **Performance optimizations**
 - Prefetching & data restructuring
- **Handling writes**
 - Write-back vs. write-through
- **Memory hierarchy**
 - Smaller, faster, expensive → bigger, slower, cheaper

Computer Architecture

Unit 5: Virtual Memory

Slides developed by Milo Martin & Amir Roth at the University of Pennsylvania
with sources that included University of Wisconsin slides
by Mark Hill, Guri Sohi, Jim Smith, and David Wood

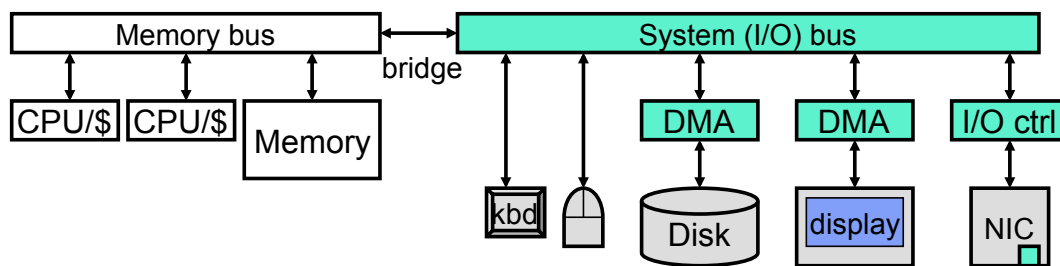
This Unit: Virtual Memory



- The operating system (OS)
 - A super-application
 - Hardware support for an OS
- Virtual memory
 - Page tables and address translation
 - TLBs and memory hierarchy issues

A Computer System: Hardware

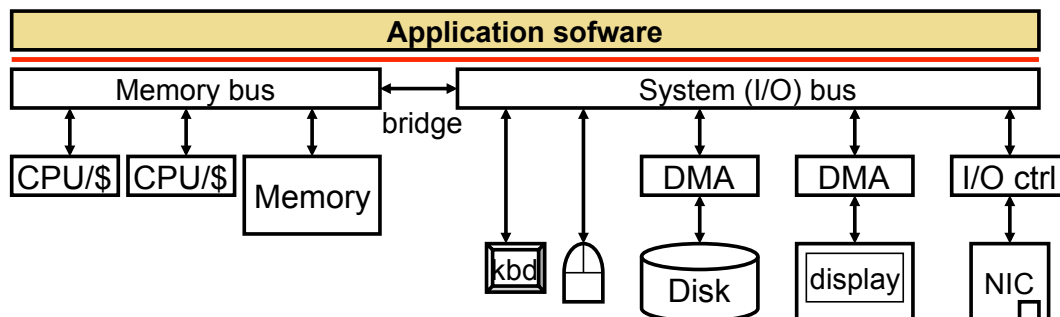
- CPUs and memories
 - Connected by memory bus
- **I/O peripherals**: storage, input, display, network, ...
 - With separate or built-in DMA
 - Connected by **system bus** (which is connected to memory bus)



Computer Architecture | Prof. Milo Martin | Virtual Memory

A Computer System: + App Software

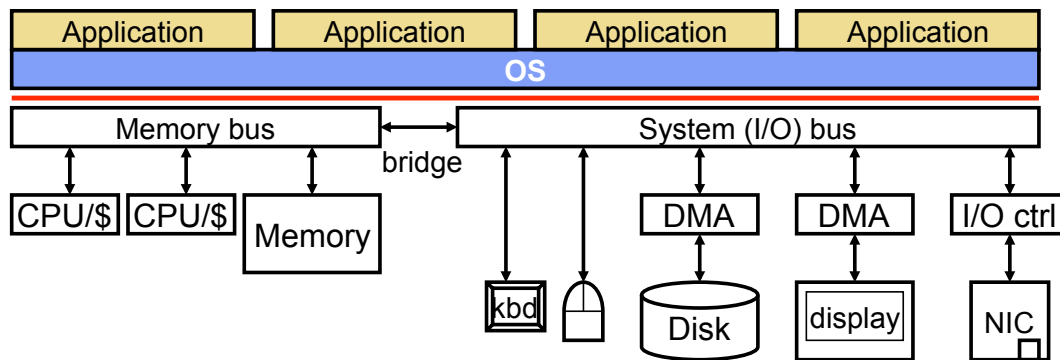
- **Application software**: computer must do something



Computer Architecture | Prof. Milo Martin | Virtual Memory

A Computer System: + OS

- **Operating System (OS):** virtualizes hardware for apps
 - **Abstraction:** provides **services** (e.g., threads, files, etc.)
 - + Simplifies app programming model, raw hardware is nasty
 - **Isolation:** gives each app illusion of private CPU, memory, I/O
 - + Simplifies app programming model
 - + Increases hardware resource utilization



Computer Architecture | Prof. Milo Martin | Virtual Memory

5

Operating System (OS) and User Apps

- Sane system development requires a split
 - Hardware itself facilitates/enforces this split
- **Operating System (OS):** a super-privileged process
 - Manages hardware resource allocation/revocation for all processes
 - Has direct access to resource allocation features
 - Aware of many nasty hardware details
 - Aware of other processes
 - Talks directly to input/output devices (device driver software)
- **User-level apps:** ignorance is bliss
 - Unaware of most nasty hardware details
 - Unaware of other apps (and OS)
 - Explicitly denied access to resource allocation features

Computer Architecture | Prof. Milo Martin | Virtual Memory

6

System Calls

- Controlled transfers to/from OS
- **System Call**: a user-level app “function call” to OS
 - Leave description of what you want done in registers
 - SYSCALL instruction (also called TRAP or INT)
 - Can’t allow user-level apps to invoke arbitrary OS code
 - Restricted set of legal OS addresses to jump to (**trap vector**)
 - Processor jumps to OS using trap vector
 - Sets privileged mode
 - OS performs operation
 - OS does a “return from system call”
 - Unsets privileged mode
- Used for I/O and other operating system services

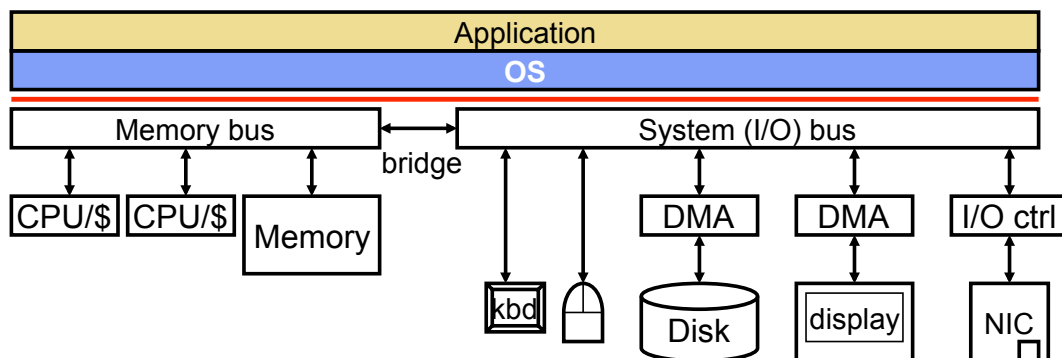
Input/Output (I/O)

- Applications use “system calls” to initiate I/O
- Only operating system (OS) talks directly to the I/O device
 - Send commands, query status, etc.
 - OS software uses special uncached load/store operations
 - Hardware sends these reads/writes across I/O bus to device
- Hardware also provides “Direct Memory Access (DMA)”
 - For big transfers, the I/O device accesses the memory directly
 - Example: DMA used to transfer an entire block to/from disk
- Interrupt-driven I/O
 - The I/O device tells the software its transfer is complete
 - Tells the hardware to raise an “interrupt” (door bell)
 - Processor jumps into the OS
 - Inefficient alternative: polling

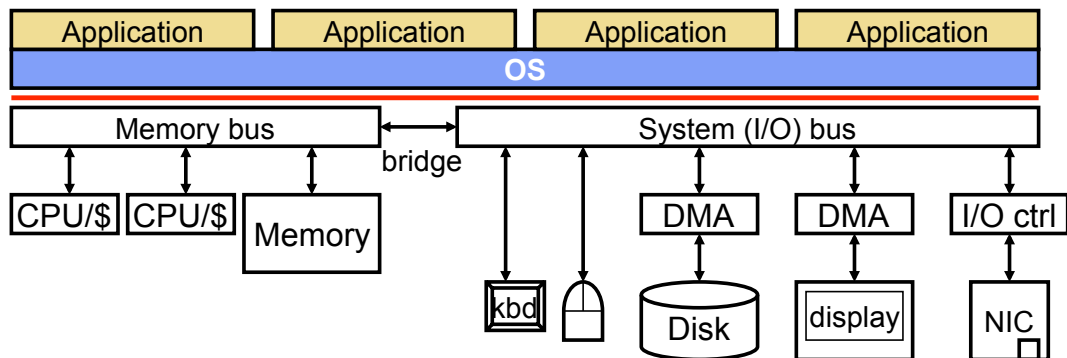
Interrupts

- **Exceptions**: synchronous, generated by running app
 - E.g., illegal insn, divide by zero, etc.
- **Interrupts**: asynchronous events generated externally
 - E.g., timer, I/O request/reply, etc.
- **“Interrupt” handling**: same mechanism for both
 - “Interrupts” are on-chip signals/bits
 - Either internal (e.g., timer, exceptions) or from I/O devices
 - Processor continuously monitors interrupt status, when one is high...
 - Hardware jumps to some preset address in OS code (interrupt vector)
 - Like an asynchronous, non-programmatic SYSCALL
- **Timer**: programmable on-chip interrupt
 - Initialize with some number of micro-seconds
 - Timer counts down and interrupts when reaches zero

A Computer System: + OS



A Computer System: + OS



Virtualizing Processors

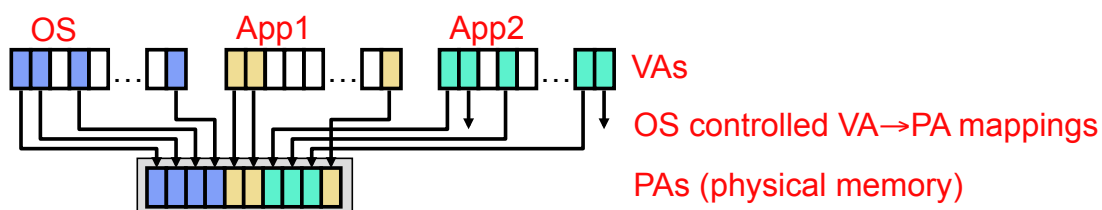
- How do multiple apps (and OS) share the processors?
 - **Goal:** applications think there are an infinite # of processors
- Solution: time-share the resource
 - Trigger a **context switch** at a regular interval (~1ms)
 - **Pre-emptive:** app doesn't yield CPU, OS forcibly takes it
 - + Stops greedy apps from starving others
 - **Architected state:** PC, registers
 - Save and restore them on context switches
 - Memory state?
 - **Non-architected state:** caches, predictor tables, etc.
 - Ignore or flush
- Operating system responsible to handle context switching
 - Hardware support is just a timer interrupt

Virtualizing Main Memory

- How do multiple apps (and the OS) share main memory?
 - **Goal: each application thinks it has infinite memory**
- One app may want more memory than is in the system
 - App's insn/data footprint may be larger than main memory
 - **Requires main memory to act like a cache**
 - With disk as next level in memory hierarchy (slow)
 - Write-back, write-allocate, large blocks or "pages"
 - No notion of "program not fitting" in registers or caches (why?)
- Solution:
 - Part #1: treat memory as a "cache"
 - Store the overflowed blocks in "swap" space on disk
 - Part #2: add a level of indirection (address translation)

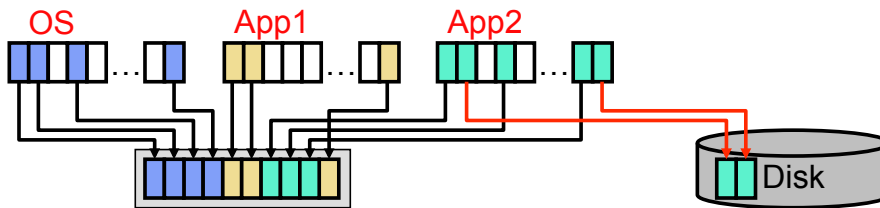
Virtual Memory (VM)

- **Virtual Memory (VM):**
 - Level of indirection
 - Application generated addresses are **virtual addresses (VAs)**
 - Each process *thinks* it has its own 2^N bytes of address space
 - Memory accessed using **physical addresses (PAs)**
 - VAs translated to PAs at some coarse granularity (page)
 - OS controls VA to PA mapping for itself and all other processes
 - Logically: translation performed before every insn fetch, load, store
 - Physically: hardware acceleration removes translation overhead



Virtual Memory (VM)

- Programs use **virtual addresses (VA)**
 - VA size (N) aka machine size (e.g., Core 2 Duo: 48-bit)
- Memory uses **physical addresses (PA)**
 - PA size (M) typically $M < N$, especially if $N=64$
 - 2^M is most physical memory machine supports
- VA \rightarrow PA at **page** granularity (VP \rightarrow PP)
 - Mapping need not preserve contiguity
 - VP need not be mapped to any PP
 - Unmapped VPs live on disk (swap) or nowhere (if not yet touched)



Computer Architecture | Prof. Milo Martin | Virtual Memory

15

VM is an Old Idea: Older than Caches

- Original motivation: **single-program compatibility**
 - IBM System 370: a family of computers with one software suite
 - + Same program could run on machines with different memory sizes
 - Prior, programmers explicitly accounted for memory size
- But also: **full-associativity + software replacement**
 - Memory t_{miss} is high: extremely important to reduce $\%_{miss}$

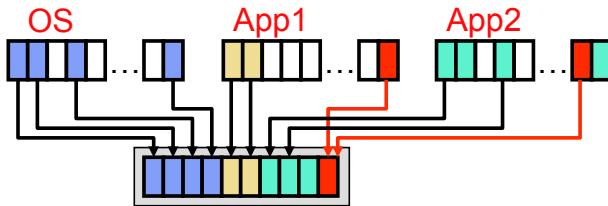
Parameter	I\$/D\$	L2	Main Memory
t_{hit}	2ns	10ns	30ns
t_{miss}	10ns	30ns	10ms (10M ns)
Capacity	8–64KB	128KB–2MB	64MB–64GB
Block size	16–32B	32–256B	4+ KB
Assoc./Repl.	1–4, LRU	4–16, LRU	Full, "working set"

Computer Architecture | Prof. Milo Martin | Virtual Memory

16

Uses of Virtual Memory

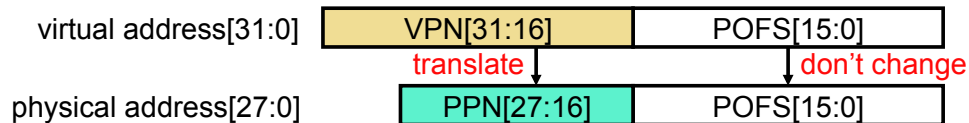
- More recently: **isolation** and **multi-programming**
 - Each app thinks it has 2^N B of memory, its stack starts 0xFFFFFFFF,...
 - Apps prevented from reading/writing each other's memory
 - Can't even address the other program's memory!
- **Protection**
 - Each page with a read/write/execute permission set by OS
 - Enforced by hardware
- **Inter-process communication.**
 - Map same physical pages into multiple virtual address spaces
 - Or share files via the UNIX `mmap()` call



Computer Architecture | Prof. Milo Martin | Virtual Memory

17

Address Translation



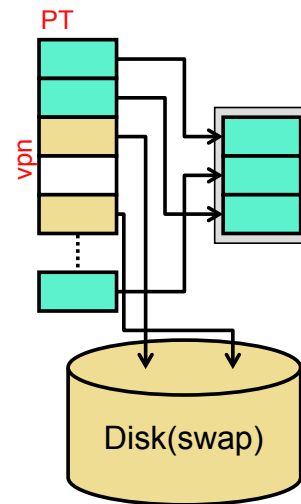
- VA→PA mapping called **address translation**
 - Split VA into **virtual page number (VPN)** & **page offset (POFS)**
 - Translate VPN into **physical page number (PPN)**
 - POFS is not translated
 - VA→PA = [VPN, POFS] → [PPN, POFS]
- Example above
 - 64KB pages → 16-bit POFS
 - 32-bit machine → 32-bit VA → 16-bit VPN
 - Maximum 256MB memory → 28-bit PA → 12-bit PPN

Computer Architecture | Prof. Milo Martin | Virtual Memory

18

Address Translation Mechanics I

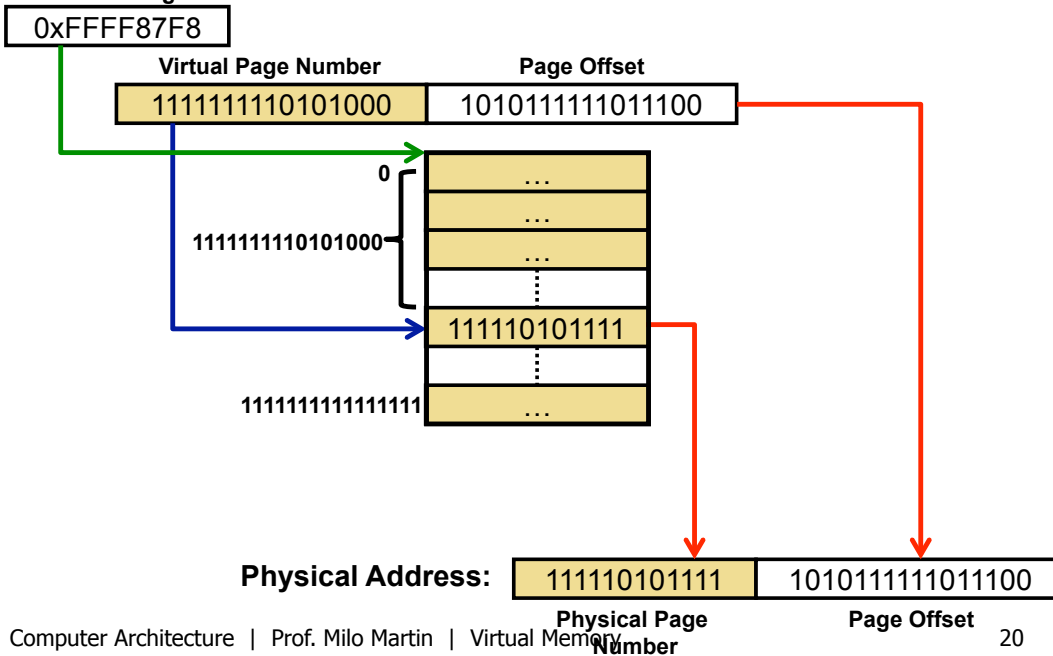
- How are addresses translated?
 - In software (for now) but with hardware acceleration (a little later)
- Each process allocated a **page table (PT)**
 - **Software data structure constructed by OS**
 - Maps VPs to PPs or to disk (swap) addresses
 - VP entries empty if page never referenced
 - Translation is table lookup



Page Table Example

Example: Memory access at address 0xFFA8AFBA

Address of Page Table Root



Page Table Size

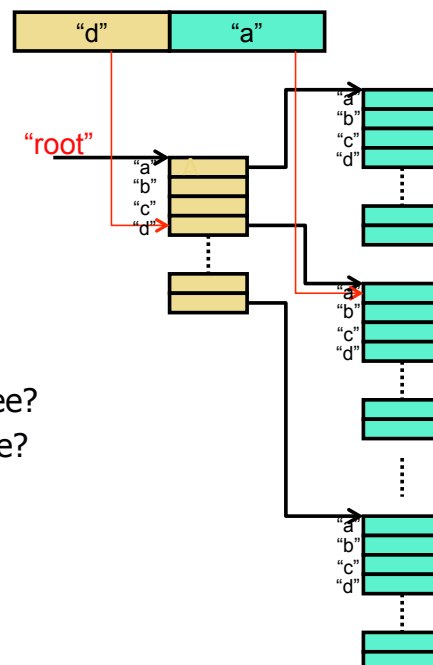
- How big is a page table on the following machine?
 - 32-bit machine
 - 4B page table entries (PTEs)
 - 4KB pages



- 32-bit machine → 32-bit VA → $2^{32} = 4\text{GB}$ virtual memory
 - 4GB virtual memory / 4KB page size → 1M VPs
 - 1M VPs * 4 Bytes per PTE → 4MB
- How big would the page table be with 64KB pages?
 - How big would it be for a 64-bit machine?
 - Page tables can get big
 - There are ways of making them smaller

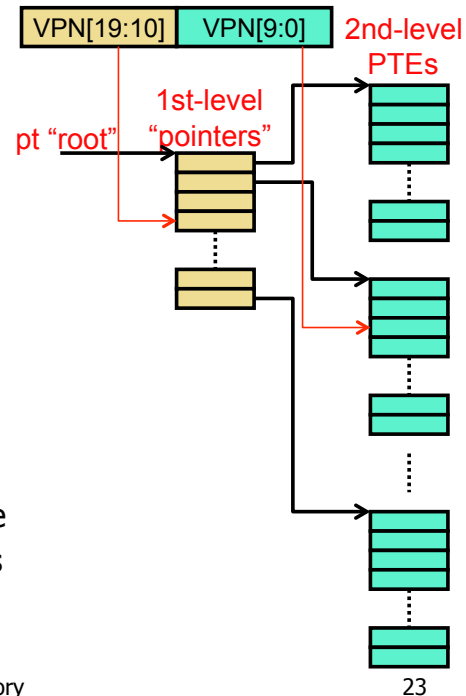
"Trie"

- What is a **"trie"** data structure
 - Also called a "prefix tree"
- What is it used for?
- What properties does it have?
 - How is it different from a binary tree?
 - How is it different than a hash table?



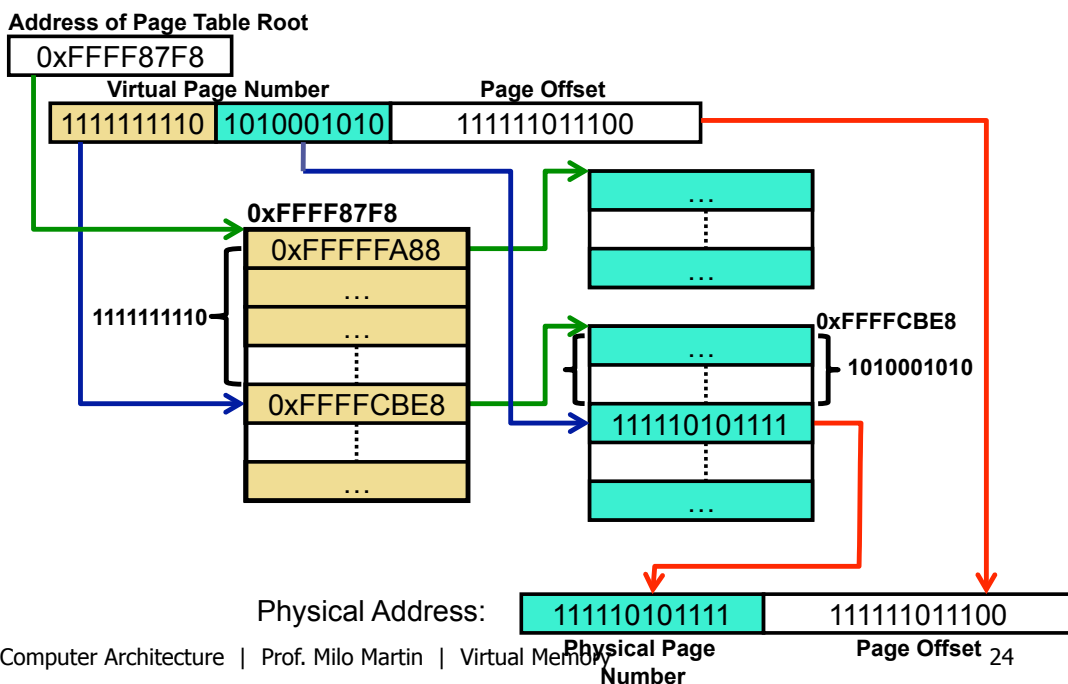
Multi-Level Page Table (PT)

- One way: **multi-level page tables**
 - Tree of page tables ("trie")
 - Lowest-level tables hold PTEs
 - Upper-level tables hold pointers to lower-level tables
 - Different parts of VPN used to index different levels
- 20-bit VPN
 - Upper 10 bits index 1st-level table
 - Lower 10 bits index 2nd-level table
 - In reality, often more than 2 levels



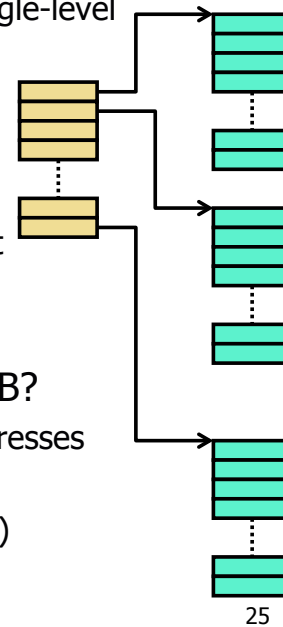
Multi-Level Address Translation

Example: Memory access at address 0xFFA8AFBA



Multi-Level Page Table (PT)

- Have we saved any space?
 - Isn't total size of 2nd level tables same as single-level table (i.e., 4MB)?
 - Yes, but...
- Large virtual address regions **unused**
 - Corresponding 2nd-level tables need not exist
 - Corresponding 1st-level pointers are *null*
- How large for contiguous layout of 256MB?
 - Each 2nd-level table maps 4MB of virtual addresses
 - One 1st-level + 64 2nd-level pages
 - 65 total pages = 260KB (much less than 4MB)



Computer Architecture | Prof. Milo Martin | Virtual Memory

Page-Level Protection

- **Page-level protection**
 - Piggy-back page-table mechanism
 - Map VPN to PPN + Read/Write/Execute permission bits
 - Attempt to execute data, to write read-only data?
 - Exception → OS terminates program
 - Useful (for OS itself actually)

VA Bits <47:39>	VA Bits <38:30>	VA Bits <29:21>	VA Bits <20:12>	VA Bits <11:0>
Level 1 table index	Level 2 table index	Level 3 table index	Level 4 table (page) index	Page offset address

- 4-level lookup, 4KB translation granule, 48-bit address
 - 9 address bits per level

VA Bits <41:29>	VA Bits <28:16>	VA Bits <15:0>
Level 1 table index	Level 2 table (page) index	Page offset address

- 2-level lookup, 64KB page/page table size, 42-bit address
 - 13 address bits per level
 - 3 levels for 48 bits of VA – top level table is a partial table

6 3	5 2	4 8	1 2	2	1	0
Upper attributes	SBZ	Address out	SBZ	Lower attributes and validity		

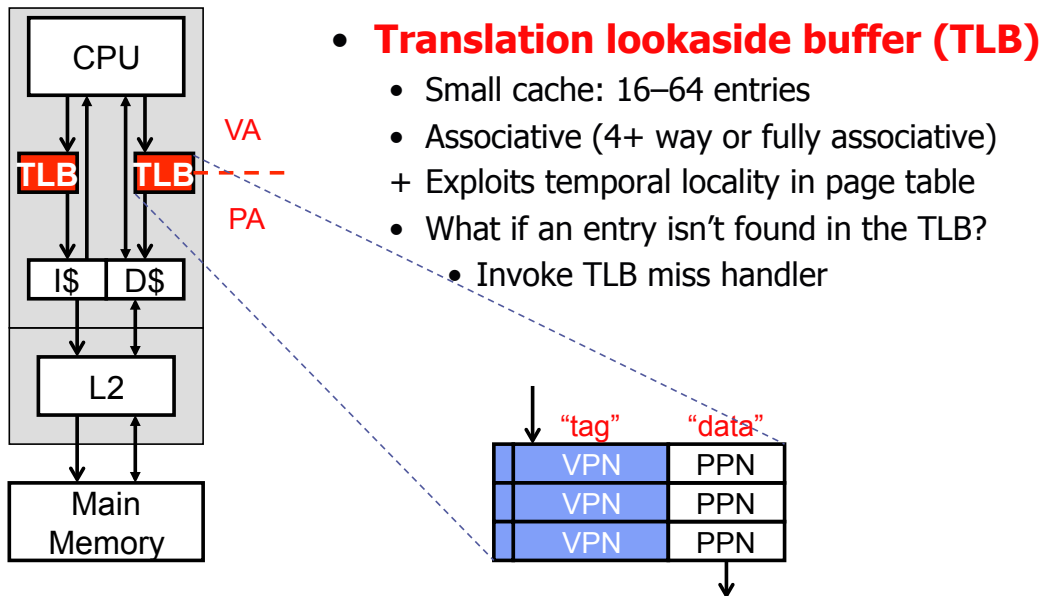
- 64-bit Translation table entry format

ARMv8 Technology Preview By *Richard Grisenthwaite Lead Architect and Fellow. ARM*
 Computer Architecture | Prof. Milo Martin | Virtual Memory 27

Address Translation Mechanics II

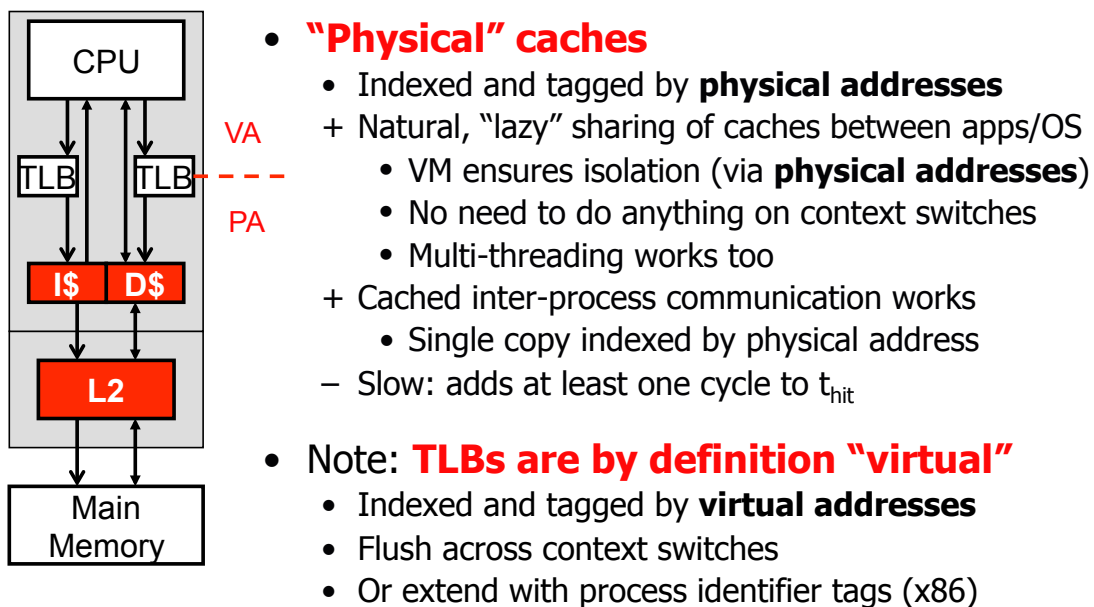
- Conceptually
 - Translate VA to PA before every cache access
 - Walk the page table before every load/store/insn-fetch
 - Would be terribly inefficient (even in hardware)
- In reality
 - **Translation Lookaside Buffer (TLB)**: cache translations
 - Only walk page table on TLB miss
- Hardware truisms
 - Functionality problem? Add indirection (e.g., VM)
 - Performance problem? Add cache (e.g., TLB)

Translation Lookaside Buffer



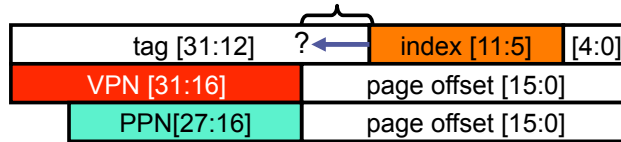
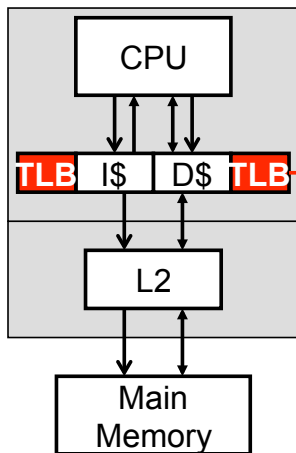
- **Translation lookaside buffer (TLB)**
 - Small cache: 16–64 entries
 - Associative (4+ way or fully associative)
 - + Exploits temporal locality in page table
 - What if an entry isn't found in the TLB?
 - Invoke TLB miss handler

Serial TLB & Cache Access



- **"Physical" caches**
 - Indexed and tagged by **physical addresses**
 - + Natural, "lazy" sharing of caches between apps/OS
 - VM ensures isolation (via **physical addresses**)
 - No need to do anything on context switches
 - Multi-threading works too
 - + Cached inter-process communication works
 - Single copy indexed by physical address
 - Slow: adds at least one cycle to t_{hit}
- Note: **TLBs are by definition "virtual"**
 - Indexed and tagged by **virtual addresses**
 - Flush across context switches
 - Or extend with process identifier tags (x86)

Parallel TLB & Cache Access



What about parallel access?

- Only if...
 - (cache size) / (associativity) ≤ page size**
 - Index bits same in virt. and physical addresses!
- Access TLB in parallel with cache
 - Cache access needs tag only at very end
 - + Fast: no additional t_{hit} cycles
 - + No context-switching/aliasing problems
 - Dominant organization used today
- Example: Core 2, 4KB pages, 32KB, 8-way SA L1 data cache
 - Implication: associativity allows bigger caches

TLB Organization

- **Like caches:** TLBs also have ABCs
 - Capacity
 - Associativity (At least 4-way associative, fully-associative common)
 - What does it mean for a TLB to have a block size of two?
 - Two consecutive VPs share a single tag
 - **Like caches:** there can be second-level TLBs
- Example: AMD Opteron
 - 32-entry fully-assoc. TLBs, 512-entry 4-way L2 TLB (insn & data)
 - 4KB pages, 48-bit virtual addresses, four-level page table
- **Rule of thumb:** TLB should "cover" size of on-chip caches
 - In other words: (#PTEs in TLB) * page size ≥ cache size
 - Why? Consider relative miss latency in each...

TLB Misses

- **TLB miss:** translation not in TLB, but in page table
 - Two ways to “fill” it, both relatively fast
- **Hardware-managed TLB:** e.g., x86, recent SPARC, ARM
 - Page table root in hardware register, hardware “walks” table
 - + Latency: saves cost of OS call (avoids pipeline flush)
 - Page table format is hard-coded
- **Software-managed TLB:** e.g., Alpha, MIPS
 - Short (~10 insn) OS routine walks page table, updates TLB
 - + Keeps page table format flexible
 - Latency: one or two memory accesses + OS call (pipeline flush)
- Trend is towards hardware TLB miss handler

Page Faults

- **Page fault:** PTE not in TLB or page table
 - → page not in memory
 - Or no valid mapping → segmentation fault
 - Starts out as a TLB miss, detected by OS/hardware handler
- **OS software routine:**
 - Choose a physical page to replace
 - **“Working set”:** refined LRU, tracks active page usage
 - If dirty, write to disk
 - Read missing page from disk
 - Takes so long (~10ms), OS schedules another task
 - Requires yet another data structure: **frame map**
 - Maps physical pages to <process, virtual page> pairs
 - Treat like a normal TLB miss from here

Summary

- OS virtualizes memory and I/O devices
- Virtual memory
 - “infinite” memory, isolation, protection, inter-process communication
 - Page tables
 - Translation buffers
 - Parallel vs serial access, interaction with caching
 - Page faults