

# CS/ECE 752: Advanced Computer Architecture I

## Lecture 1. Introduction

Professor Matthew D. Sinclair



Source: XKCD

Slide History/Attribution Diagram:

UW Madison  
Hill, Sohi,  
Smith, Wood

UPenn  
Amir Roth,  
Milo Martin

Various Universities  
Asanovic, Falsafi, Hoe, Lipasti,  
Shen, Smith, Vijaykumar

UW Madison  
Hill, Sohi, Wood,  
Sankaralingam, Sinclair

UCLA  
Nowatzki

# Announcements

- HW0 Due Friday
- Moore 1965 Review Due Friday
- Advanced Topics Voting Poll due next Tuesday (on Piazza)

# Welcome!

- About Me:
  - Assistant Prof. @Wisconsin since August 2018
  - Research focuses on accelerators/specialization/heterogeneous computers
- Course website:
  - <http://pages.cs.wisc.edu/~sinclair/courses/cs752/fall2024/>

# CS/ECE 752 Course Overview

# Why Study Computer Architecture?

- **Understand where computers are going**
  - Future capabilities drive the computing world
  - Forced to think 5+ years into the future
- **Exposure to high-level design**
  - Less about “design” than “what to design”
  - Engineering, science, art
  - Architects paint with broad strokes
  - The best architects understand all the levels
    - Devices, circuits, architecture, compilers, applications
- **Understand hardware for software tuning**
- **Real-world impact**
  - No computer architecture → no computers!
- **Get a job** (design or research)

G S S

# Some Course Goals

- **Exposure to “big ideas” in computer architecture**
  - Pipelining, parallelism, caching, locality, abstraction, etc.
- Exposure to examples of good (and some bad) engineering
- Understanding computer performance and metrics
  - Empirical evaluation
  - Understanding quantitative data and experiments
- “Research” exposure
  - Read research literature (i.e., papers)
  - Research-quality software
  - Course project
  - Cutting edge proposals

# Course Prerequisites

- Basic Computer Organization (e.g., CS/ECE 552)
  - Logic: gates, Boolean functions, latches, memories
  - Datapath: ALU, register file, muxes
  - Control: single-cycle control, micro-code
  - Caches & pipelining (will go into these in more detail here)
  - Some familiarity with assembly language
  - Hennessy & Patterson's "Computer Organization and Design"
  - Operating Systems (processes, threads, & virtual memory, e.g., CS 537)
- Significant programming experience
  - Why? assignments require writing code to simulate hardware
  - Not difficult if competent programmer; extremely difficult if not
- **This class will have a gentle ramp, so don't worry.**

# Course Components

- **Reviews:**
  - ~10 research papers from literature, including classic and modern works
  - You will write a short review, to be submitted on Canvas, for ~1 paper per week.
- **Homeworks:**
  - There will be ~7 homeworks during the semester
  - Goals:
    - Apply your knowledge to real-world architecture evaluation (e.g., gem5)
    - Practice problems for exams
- **Exams:**
  - Two in person midterm exams, non-cumulative.
  - No final exam.
- **Project:**
  - Option 1: Literature survey (higher expectations).
  - Option 2: Open ended project of your choice.



# Lectures

- During class: lectures and in-class exercises
  - In-class exercises: discussion of paper topics
- All lectures will be recorded and posted on Canvas (under “Kaltura Gallery” page)
  - Will be integrated with weekly modules too
  - If you are unwell, please consider watching recording
- All slides & annotations will be posted to Canvas

# Paper Readings/Reviews

- Expected to complete the assigned readings before class
  - Goal: actively participate in discussions
- Reviews on ~1 class day/week helps incentivize this.
- Reviews -- three short paragraphs: (max 3200 characters)
  - summarize the problem/goal/intended contributions
  - summarize the paper's methods and results
  - give your opinion of the paper (strengths, weaknesses)
- Scale:
  - 3: Excellent, 2: Satisfactory, 1: Unsatisfactory, 0: No submission
- Rules:
  - Submit on Canvas by 9AM the day of class
  - Welcome to discuss readings on Piazza or ask questions before class (I will monitor/participate)
  - 1 free dropped review

# Reviews (Cont.) – ChatGPT & Friends

- Tools like ChatGPT are tempting to “borrow” from for reviews
- Why not to do this
  - Learning how to review papers is an important part of grad school
  - These tools are not always accurate in their summaries
    - ... and often it is hard for students to identify what those inaccuracies are
  - I will compare a generated paper review to yours → misconduct
  - Reviews count for a small portion of your grade
  - Other components of your grade rely on your understanding of the papers (beyond surface level) → **reviewing them is in your best interest**

# Required Texts

- No required *traditional* textbook for this course.
- Required reading will include:
  - Computer Architecture Synthesis Lectures
  - Published papers (available on campus network through ACM and IEEE libraries).
  - Also posted on Canvas.
  - If you use VPN + Library EZProxy trick, also can download ACM/IEEE/etc. PDFs for free remotely
- Optional Textbooks:
  - John Shen and Mikko Lipasti, Modern Processor Design: Fundamentals of Superscalar Processors, McGraw-Hill, 2005.
  - John L. Hennessy and David A. Patterson, Computer Architecture: A Quantitative Approach Morgan Kaufmann Publishers, Sixth Edition.

# Homeworks

- ~7 Homeworks during the first 2/3rds of the course.
- Should be done individually
- Intention behind homeworks:
  - Practice problems for exams.
  - Teach basics about simulation.
  - Get experience in architecture analysis.
  - Get everyone familiar with a set of tools, so that you can cooperatively work together in the project later on
  - *Not to cover all principles discussed in class.*
- *HW0*: Introduction, posted already (due Friday)
- *HW1*: Do Parts 1 and 2 from “learning gem5” online course. <http://learning.gem5.org/book/index.html>
  - This assignment still being updated/modernized

# Exams

- Focusing on thinking more deeply about the questions, and put together well thought-out responses.
- Emphasizes reasoning/argumentation over memorization.
- Exam Content:
  - Will be similar to prior exams: mix of essays and problem solving.
  - Topic fair game: anything discussed in class or in readings.
  - Questions may be about a new aspect of a relevant subject.
- Exam Rules:
  - You **may** bring **1** 8.5"x11" double-sided, handwritten note sheet
  - **Taken individually**
- Advice:
  - Practice exams from previous years available online now
- Two exams (no final), testing  $\sim 1/3$  of material each
  - 10/11/24, 11/18/24

# Project

- In lieu of final exam, we will have a course project.
  - Start before the Exam2, but most of work can be done afterwards
  - Work in teams of 2 (preferably not 1 or 3).
- Why Project?
  - Give you a chance to put into practice some of the ideas
  - Give you freedom to work on something you like
  - Learn tools/approach that can be useful later
- What is a project? Options:
  1. Literature survey
  2. Open-ended: Propose a research idea and evaluate it using any means (okay to combine with ongoing/concurrent work)
- Deliverables: report (+ source code if applicable)
  - Report should be similar to research papers (but shorter)
  - Guidelines online
  - I will post sample project ideas soon

# Project Key Deadlines

- (May update slightly – tentative)
- All but final report, project talks due at 9 AM
- Preliminary Project Ideas – Monday 10/7
- Proposal – Friday 10/25
- Progress Report – Friday 11/15
- Lightning Talks – Wednesday 12/4
- Project Talks: 930 AM – 230 PM, TBA Room 12/9
  - Larger class this semester, may need to extend, have sessions
- Final Report – 230 PM, Saturday 12/14
  - **University has a firm deadline on this**
- 2 free late days across all phases
- Proposal, Progress Report, Final Report: additionally up to 24 hours late → 10% penalty



# Grading

- Grade Breakdown:
  - Reviews: 10%
  - Homeworks: 20%
  - Exams: 30% (15% each)
  - Project: 35%
  - Participation: 5%

# Logistics

- Canvas:
  - Turning things in and reporting grades
- Webpage:  
<https://pages.cs.wisc.edu/~sinclair/courses/cs752/fall2024>
  - Post homeworks, course schedule, project description, etc...
  - Will post presentation PDFs on course schedule, but probably not until just before or just after class
- Piazza:  
(<https://piazza.com/wisc/fall2024/fa24compsci752001/home>  
[e](#))
  - This link is also on the course webpage
  - Discussions & announcements
  - You should all be enrolled already on Piazza

# Contact Me

- Email:
  - [sinclair@cs.wisc.edu](mailto:sinclair@cs.wisc.edu)
  - Please put [CS752] in subject line
- Office Hours:
  - 6369 CS
  - Wednesdays 1-2 PM, Fridays 10-11 AM

# Extenuating Circumstances – Lectures

- In extenuating circumstances I have university travel I could not avoid
- There will be 2 recorded, posted “virtual” lectures for these
  - Next week Monday is one of them
  - Posted on Canvas
  - Also listed (“virtual”) on course schedule
- I will do my very best to avoid needing to do this

# Introduction

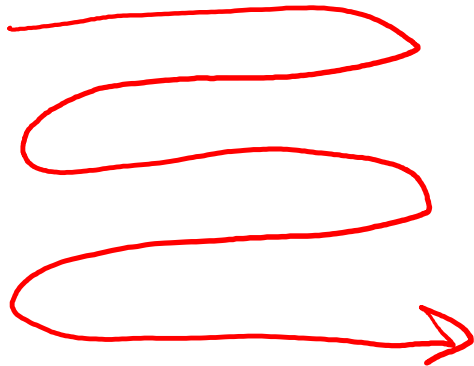
# Warmup 1

- Which of these is faster?

(C++, Java)

Version 1

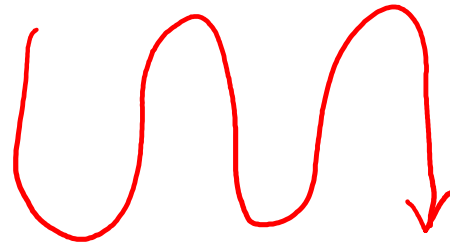
```
void copyij(int src[2048][2048],
            int dst[2048][2048])
{
    int i,j;
    for (i = 0; i < 2048; i++)
        for (j = 0; j < 2048; j++)
            dst[i][j] = src[i][j];
}
```



(Fortran, Matlab?)

Version 2

```
void copyji(int src[2048][2048],
            int dst[2048][2048])
{
    int i,j;
    for (j = 0; j < 2048; j++)
        for (i = 0; i < 2048; i++)
            dst[i][j] = src[i][j];
}
```



# Warmup 2

Sort Time vs. Branch Pred.

- Which of these is faster?

Version 1

```
for (unsigned c = 0; c < n; ++c)
    data[c] = std::rand() % 256;

//std::sort(data, data + n);

// BEGIN TIMER
for (int i = 0; i < 100000; ++i) {
    for (int c = 0; c < n; ++c) {
        if (data[c] >= 128)
            sum += data[c]*2;
    }
}
// END TIMER
```

Version 2



```
for (unsigned c = 0; c < n; ++c)
    data[c] = std::rand() % 256;

std::sort(data, data + n);

// BEGIN TIMER
for (int i = 0; i < 100000; ++i) {
    for (int c = 0; c < n; ++c) {
        if (data[c] >= 128)
            sum += data[c]*2;
    }
}
// END TIMER
```

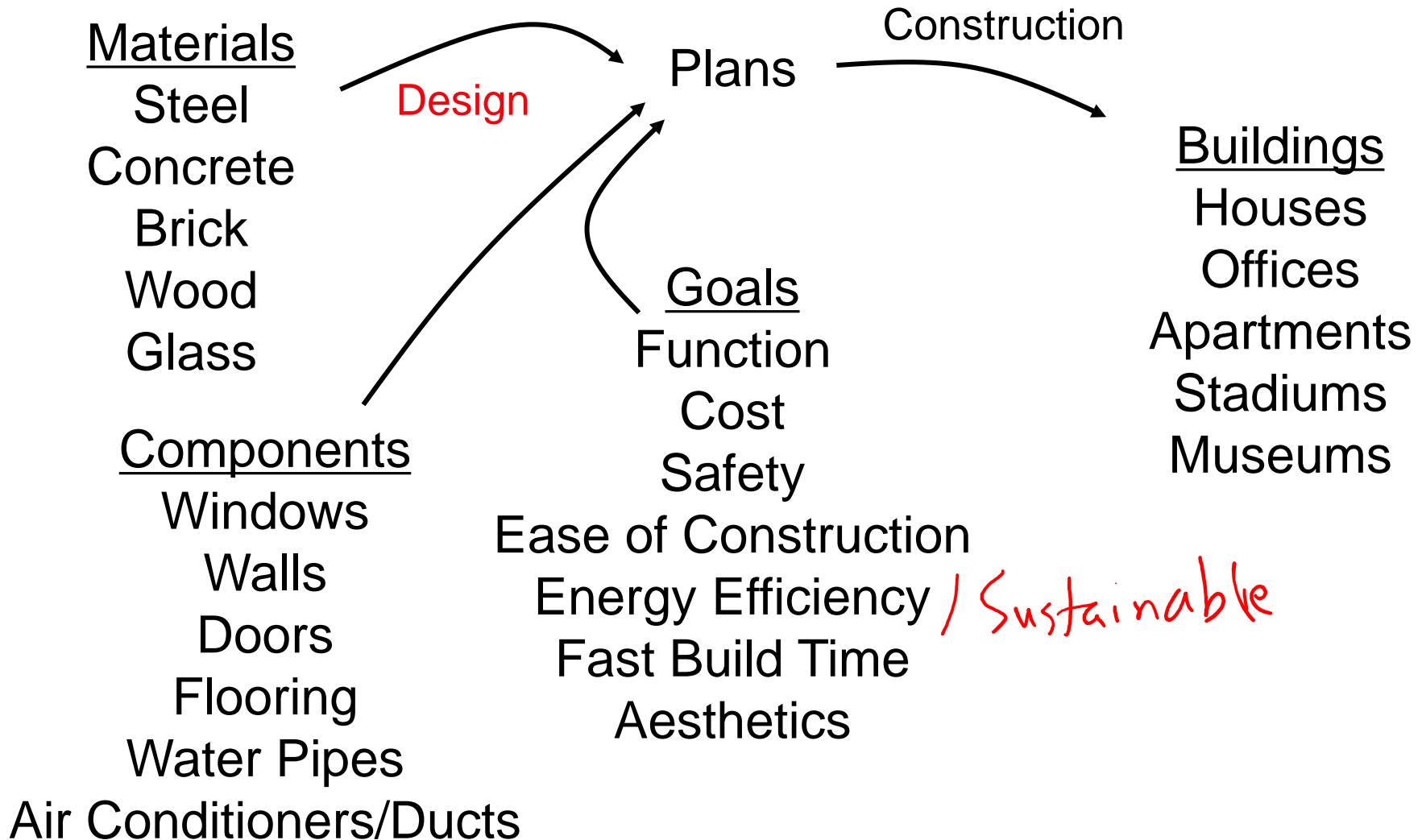
What is computer architecture?



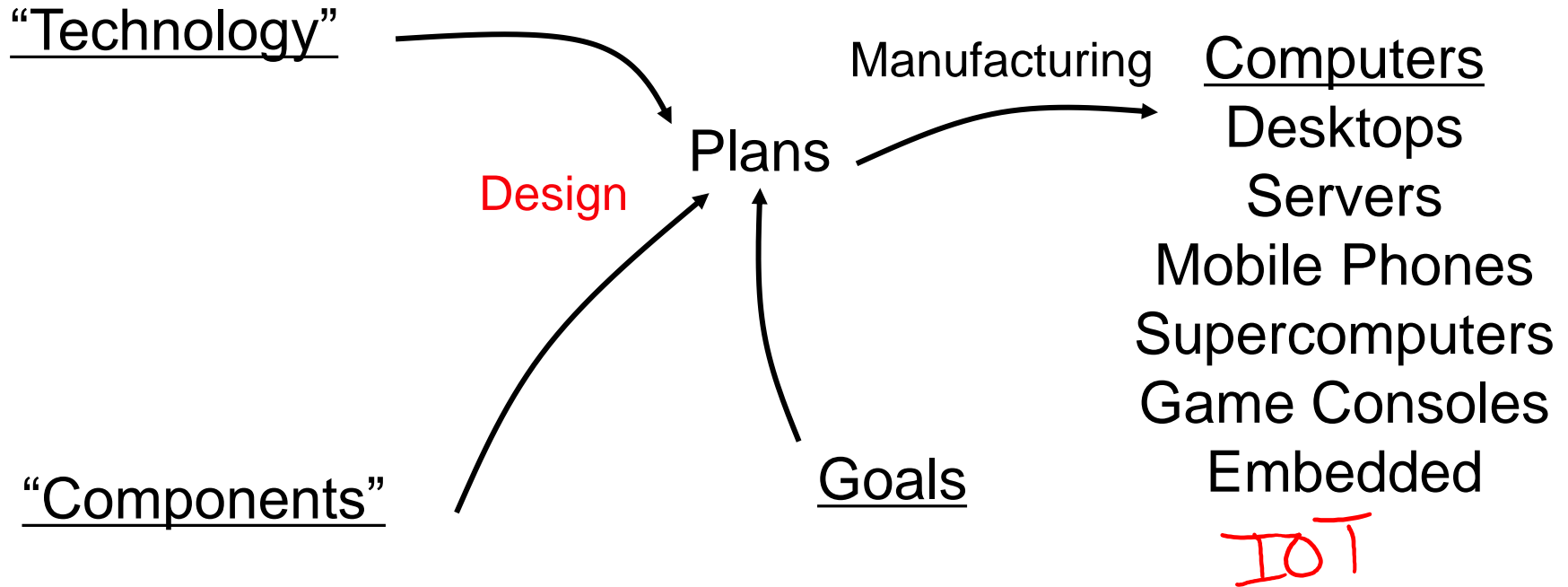
# Example Architectures



# Role of an ~~Computer~~ Architect?



# Role of a Computer Architect



/Sustainability

# Analogy Breakdown

- **Age of discipline**
  - 60 years (vs. five thousand years)
- **Fungibility**
  - No intrinsic value to a particular instance (or aesthetic value)
  - Don't care where my program lives
- **Durability**
  - Every two years you throw away your personal out-of-order multicore chip and buy a new one.
  - Compute devices will anyways become obsolete due to technology
- **Manufacturing Tradeoffs**
  - Nth+1 chip costs  $\sim \$0$
- **Boot-strapping**
  - Computers design Computers (especially with ML)

# Design Goals / Constraints

- **Functional**

- Needs to be correct
  - And unlike software, difficult to update once deployed
- Security: Should provide guarantees to software

- **Reliable**

- Does it ***continue*** to perform correctly?
- Hard fault vs transient fault
- Space satellites vs desktop vs server reliability

- **High performance**

- Not just "Gigahertz" – truck vs sports car analogy

- **Generality**

- "Fast" is only meaningful in the context of a set of important tasks
- Impossible goal: fastest possible design for all programs

(Heterogeneous Computing)

# 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...)

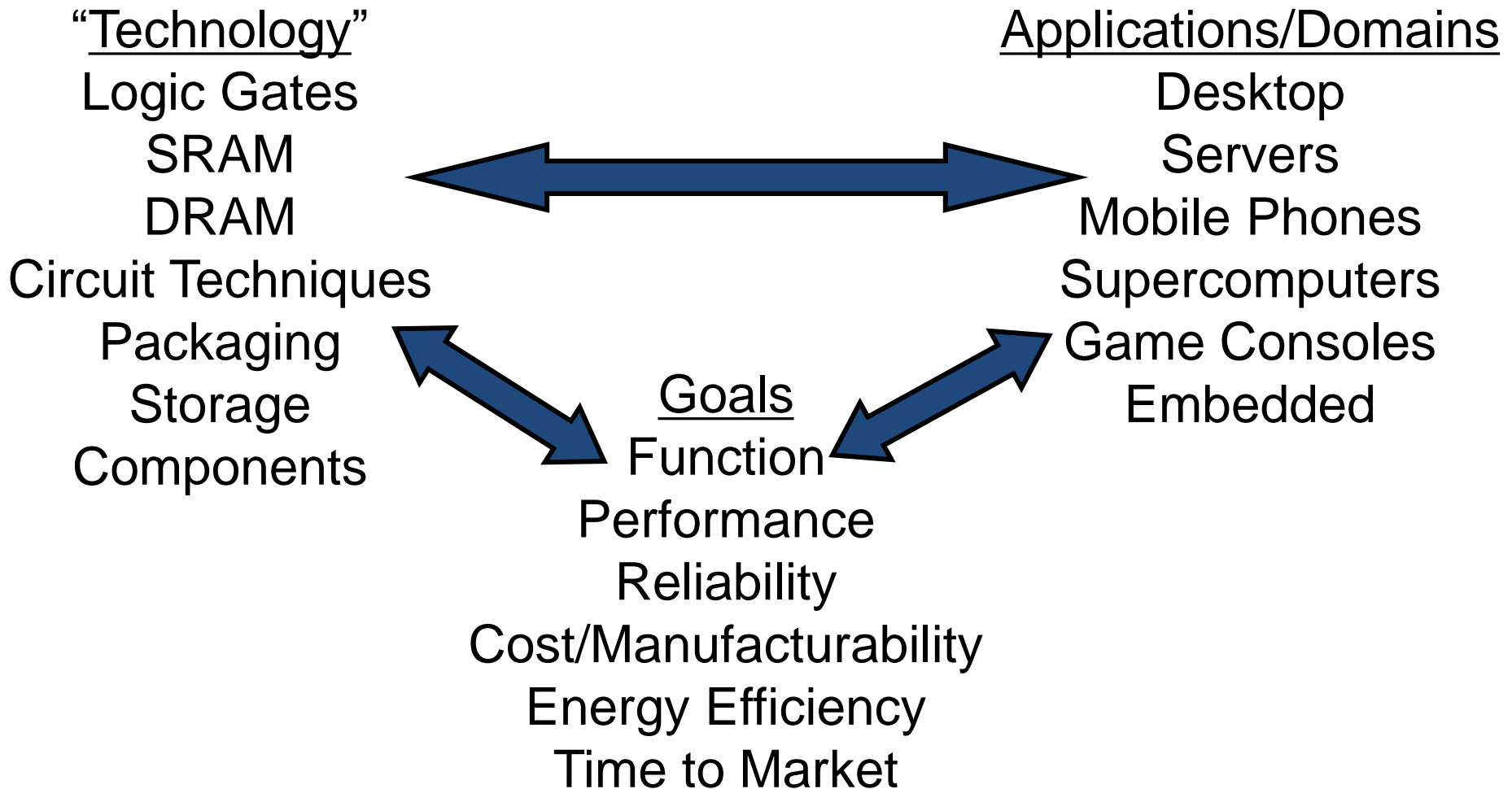
- **Low power/energy**

- 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

# Constant Change: Technology



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

# Rapid Change


Exciting: perhaps the fastest moving field ... ever

Processors vs. cars

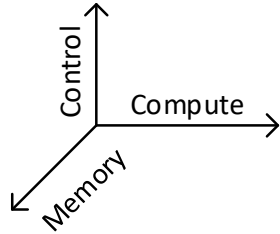
- 1985: processors = 1 MIPS, cars = 60 MPH
- 2000: processors = 500 MIPS, cars = 30,000 MPH?



# Layers of Abstraction

- Architects need to understand computers at many levels
    - Applications
    - Operating Systems
    - Compilers
    - Instruction Set Architecture
    - Microarchitecture
    - Circuits
    - Technology
  - Good architects are “Jacks of most trades”
- 

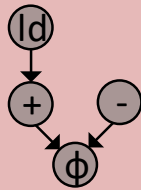
# Layers of Abstraction



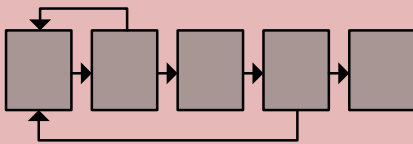
**Applications**



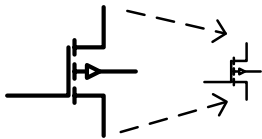
sub  
ld  
add  
br



**ISA: Hardware/  
Software Interface**



**Microarchitecture**



**Technology**

**Architects'  
Domain  
(Traditionally)**

# Instruction Set Architecture

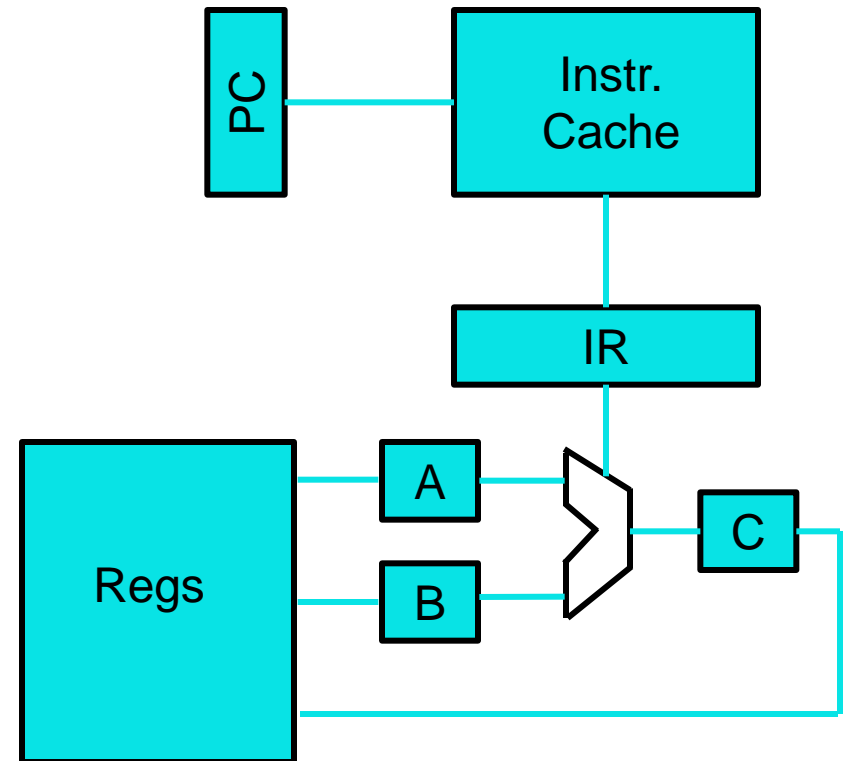
- Hardware/Software interface
  - Software impact
    - support OS functions
      - restartable instructions
      - memory relocation and protection
    - a good compiler target
      - simple
      - orthogonal
    - Dense
      - Improve memory performance
  - Hardware impact
    - admits efficient implementation
      - across generations
    - Allow/enable parallelism
      - no 'serial' bottlenecks
- Abstraction without interpretation



(ARM: exception)

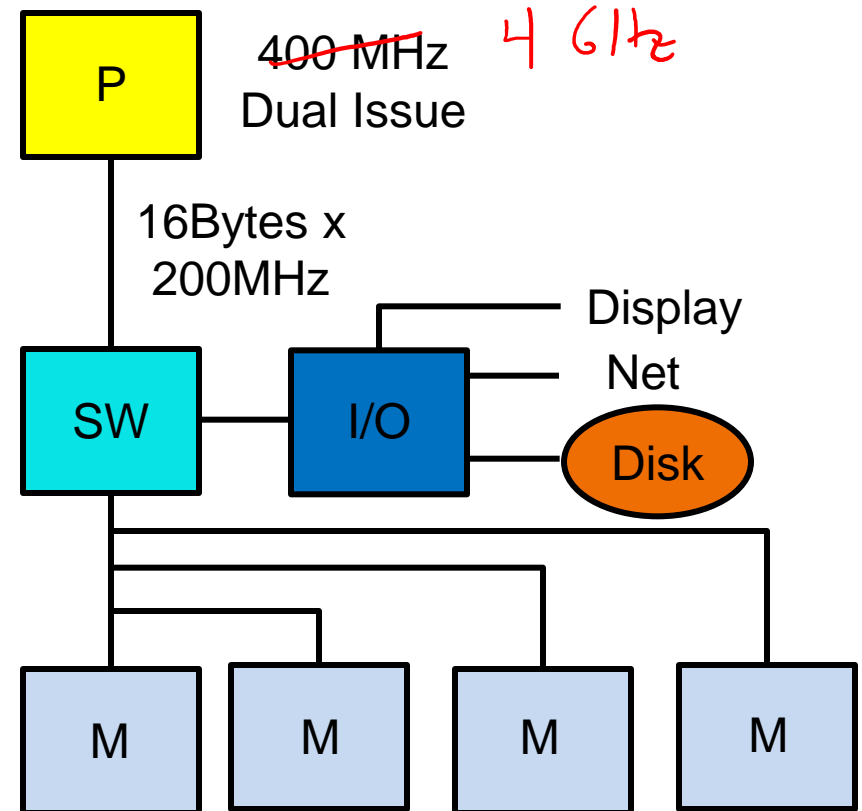
# Microarchitecture

- Emphasis is on overcoming sequential nature of programs
  - Deep pipelining
  - Multiple issue
  - Dynamic scheduling
  - Branch prediction/speculation
- Up-the-stack
  - Implement instruction set --
    - constrained by the ISA
  - Application behaviors make themselves apparent in microarchitecture
- Down-the-stack
  - Exploit circuit technology
  - Be aware of physical constraints (power, area, communication)
  - Register-transfer-level (RTL) design
- Iterative process
  - Generate proposed architecture
  - Estimate cost
  - Measure performance

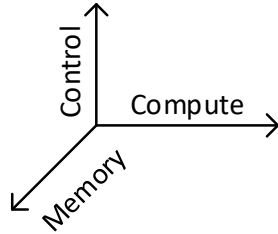


# System-Level Design

- Design at the level of processors, memories, and interconnect
- More important to application performance, cost, and power than CPU design
- Feeds and speeds
  - Constrained by IC pin count, module pin count, and signaling rates
- System balance
  - For a particular application
- Driven by
  - Performance/cost gains
  - Available components (cost/perf)
  - Technology constraints



# Layers of Abstraction



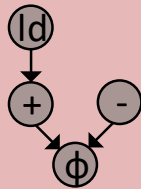
**Applications**

(ML)

**Major Driver  
Going forward?**

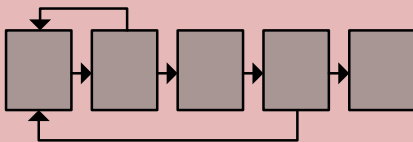


sub  
ld  
add  
br

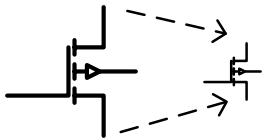


**ISA: Hardware/  
Software Interface**

Architects'  
Domain



**Microarchitecture**



**Technology**

**Major Driver  
for 50+ years**

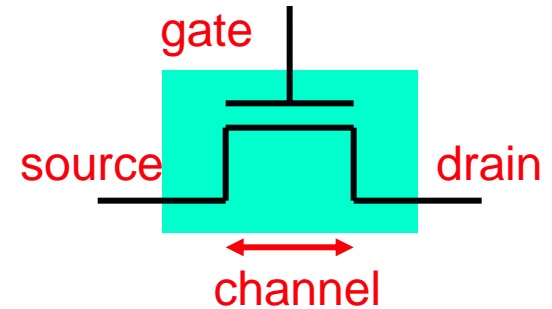
(Moore's Law)



# Technology as a Driver

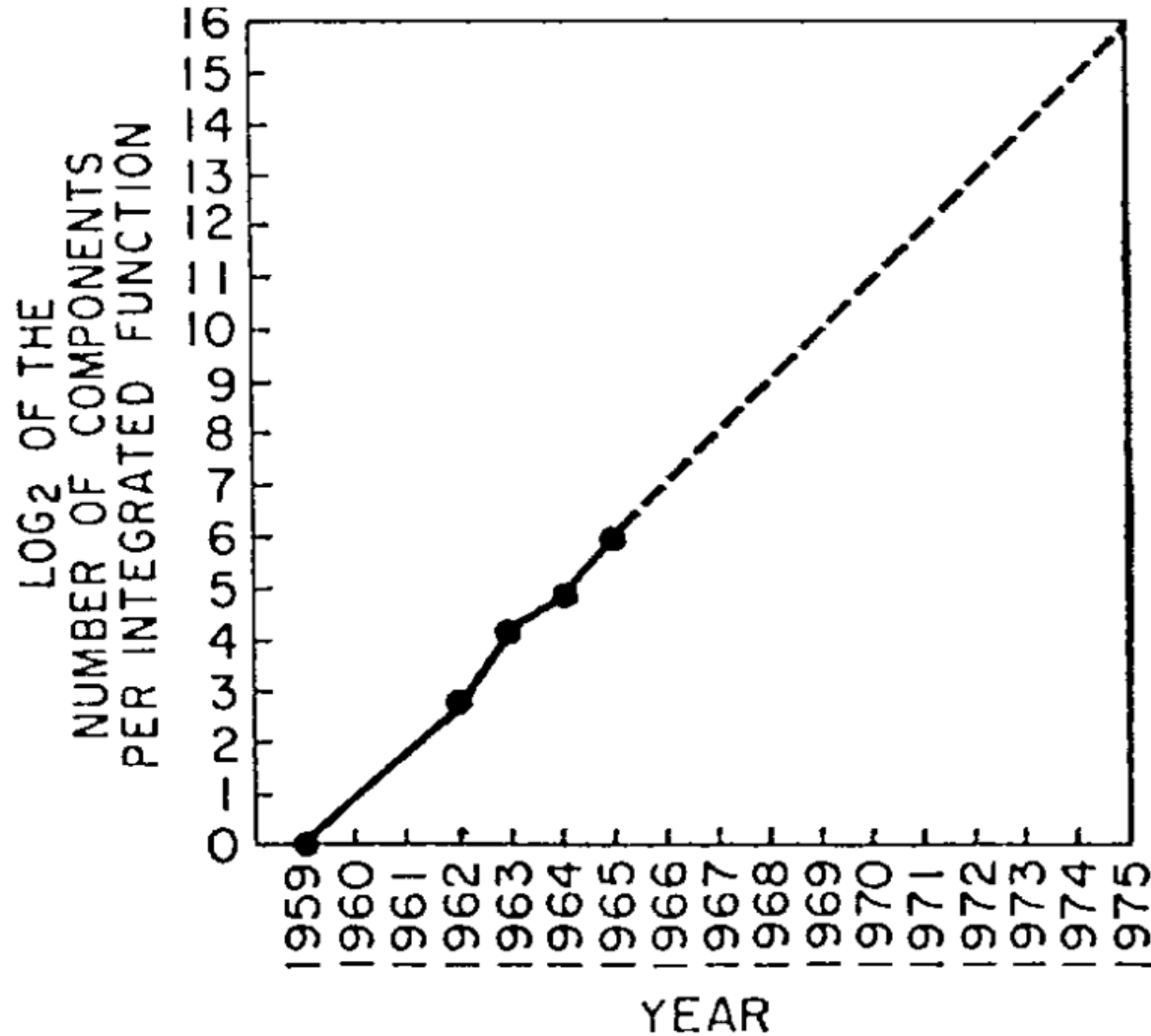
# “Technology”

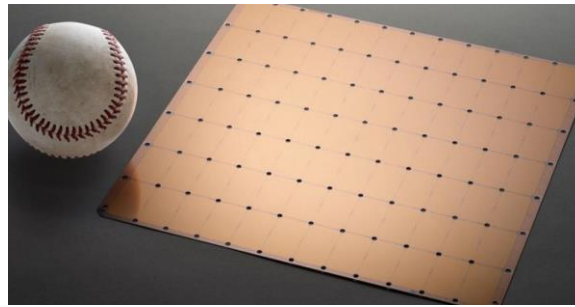
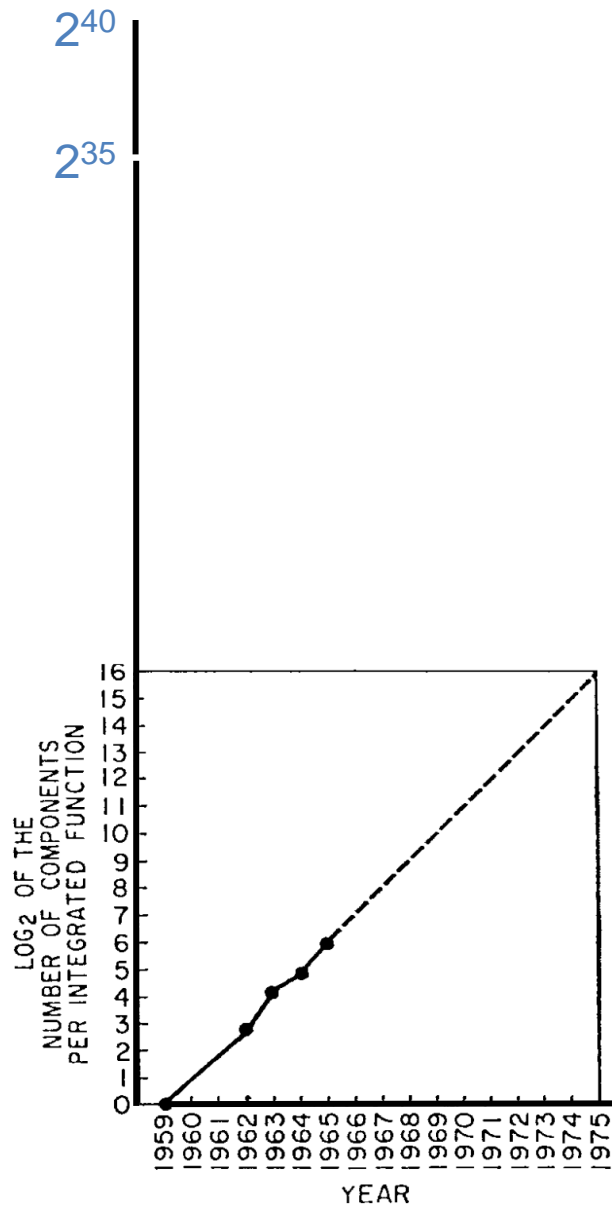
- Basic element
  - Solid-state **transistor** (i.e., electrical switch)
  - Building block of **integrated circuits (ICs)**
- What’s so great about ICs? Everything
  - + High performance, high reliability, low cost, low power
  - + Lever of mass production
- Several kinds of IC families
  - **SRAM/logic**: 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
    - Chiplets and Die Stacking
- Non-transistor storage and inter-connection technologies
  - Disk, ethernet, fiber optics, wireless





# Moore's Law -- 1965



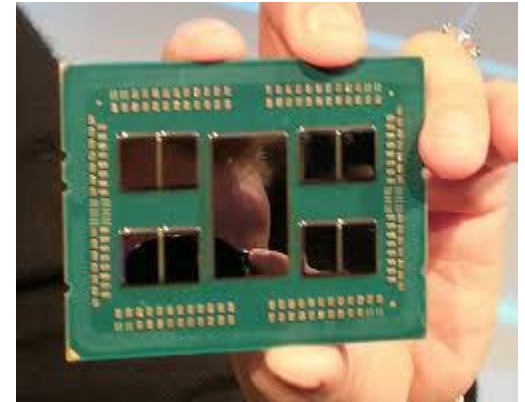


Cerebras

○

○

AMD EPYC  
ROME



2019

# 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 5-10 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
  - Examples: laptops, mobile phones, virtual/augmented reality, autonomous vehicles, 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 ( $\sim 25K$ ) 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, digital camera, mp3 players, GPS, mobile phones
- And replaced incumbents in existing segments
  - Microprocessor-based system replaced “mainframes”, “minicomputers”, etc.



# Revolution II: Implicit Parallelism

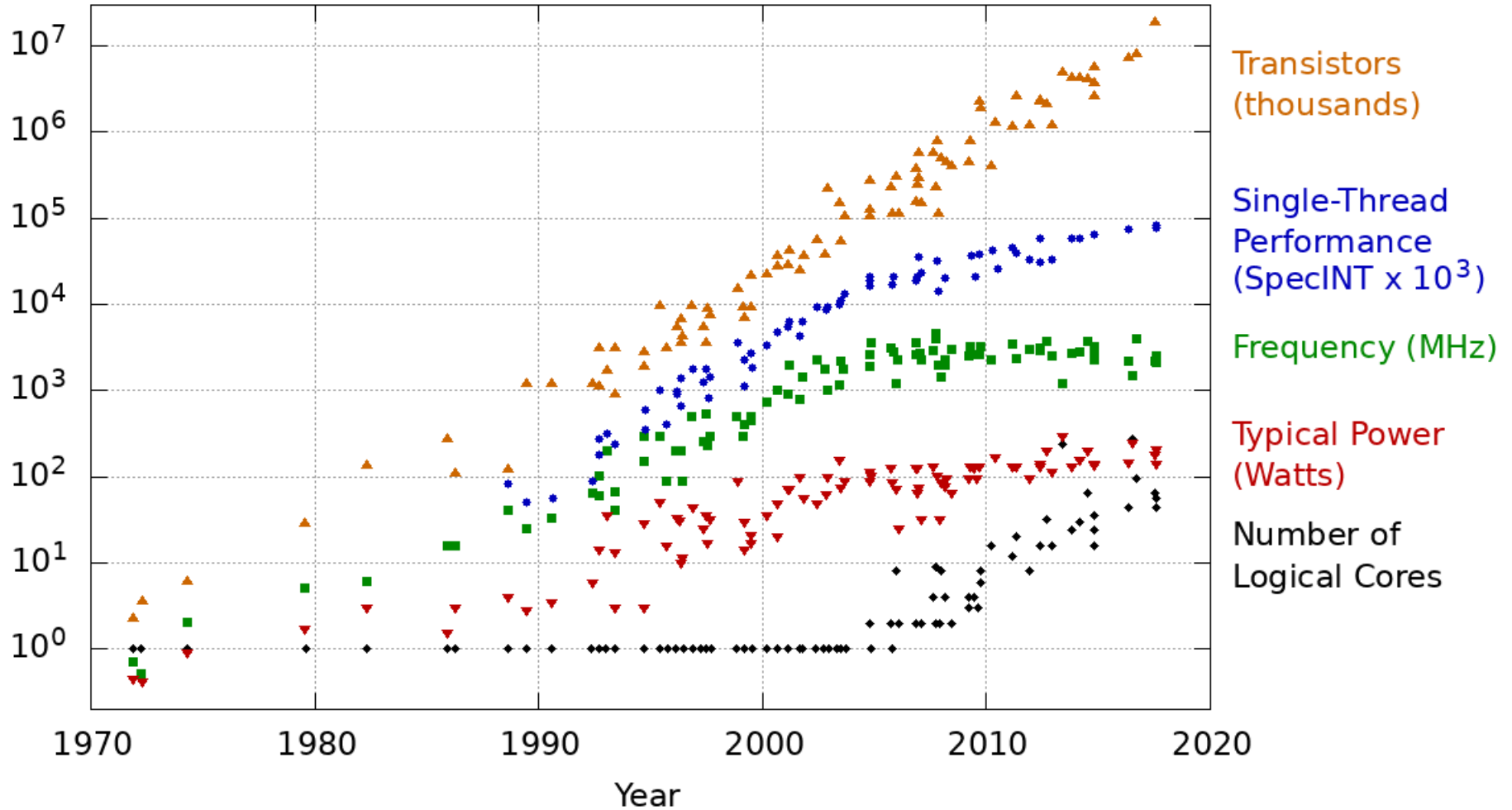
- Then to **extract implicit instruction-level parallelism**
  - Hardware provides parallel resources, figures out how to use them
  - Software is oblivious
- Initially using pipelining ...
  - Which also enabled increased clock frequency
- ... caches ...
  - Which became necessary as processor clock 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

# Not-so-recent Microprocessors

- Intel Pentium4 (2003)
  - Application: desktop/server
  - Technology: 90nm (1/100x)
  - 55M transistors (20,000x)
  - 101 mm<sup>2</sup> (10x)
  - 3.4 GHz (10,000x)
  - 1.2 Volts (1/10x)
  - 32/64-bit data (16x)
  - 22-stage pipelined datapath (22x)
  - 3 instructions per cycle (superscalar)
  - Two levels of on-chip cache
    - data-parallel vector (SIMD) instructions, hyperthreading
- Pinnacle of single-core microprocessors



## 42 Years of Microprocessor Trend Data



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten

New plot and data collected for 2010-2017 by K. Rupp

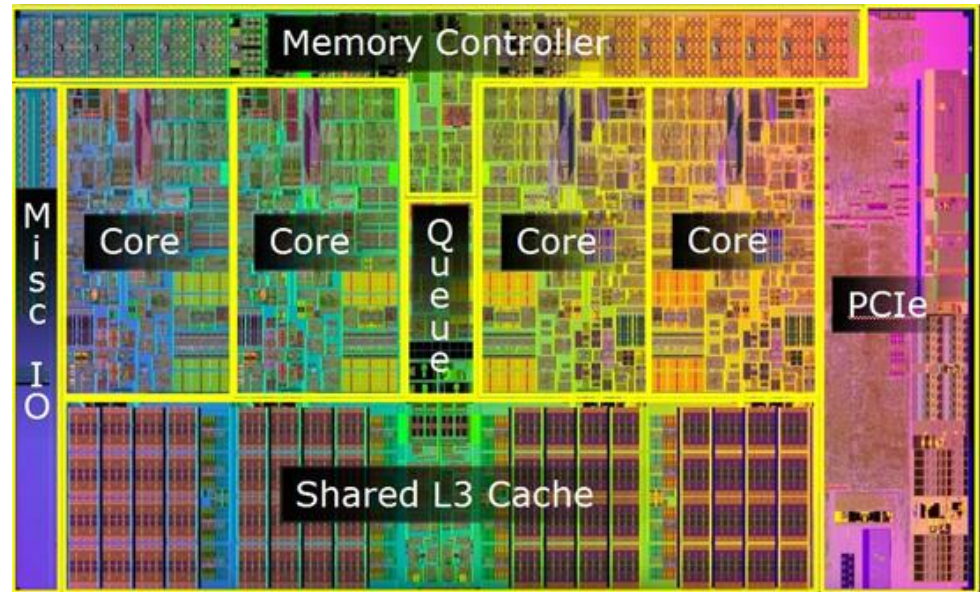


# Revolution III: Explicit Parallelism

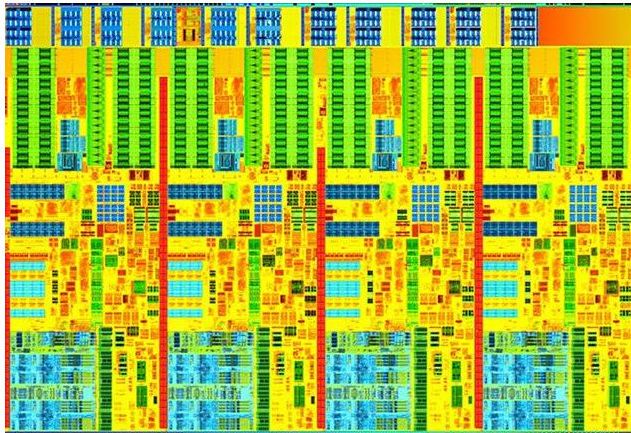
- Support **explicit data & thread level parallelism**
  - Hardware provides parallel resources, software specifies usage
  - Why? diminishing returns on instruction-level-parallelism
- First using (subword) vector instructions..., Intel's SSE
  - One instruction does four parallel multiplies
- ... and 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

# “Modern” Multicore Processor

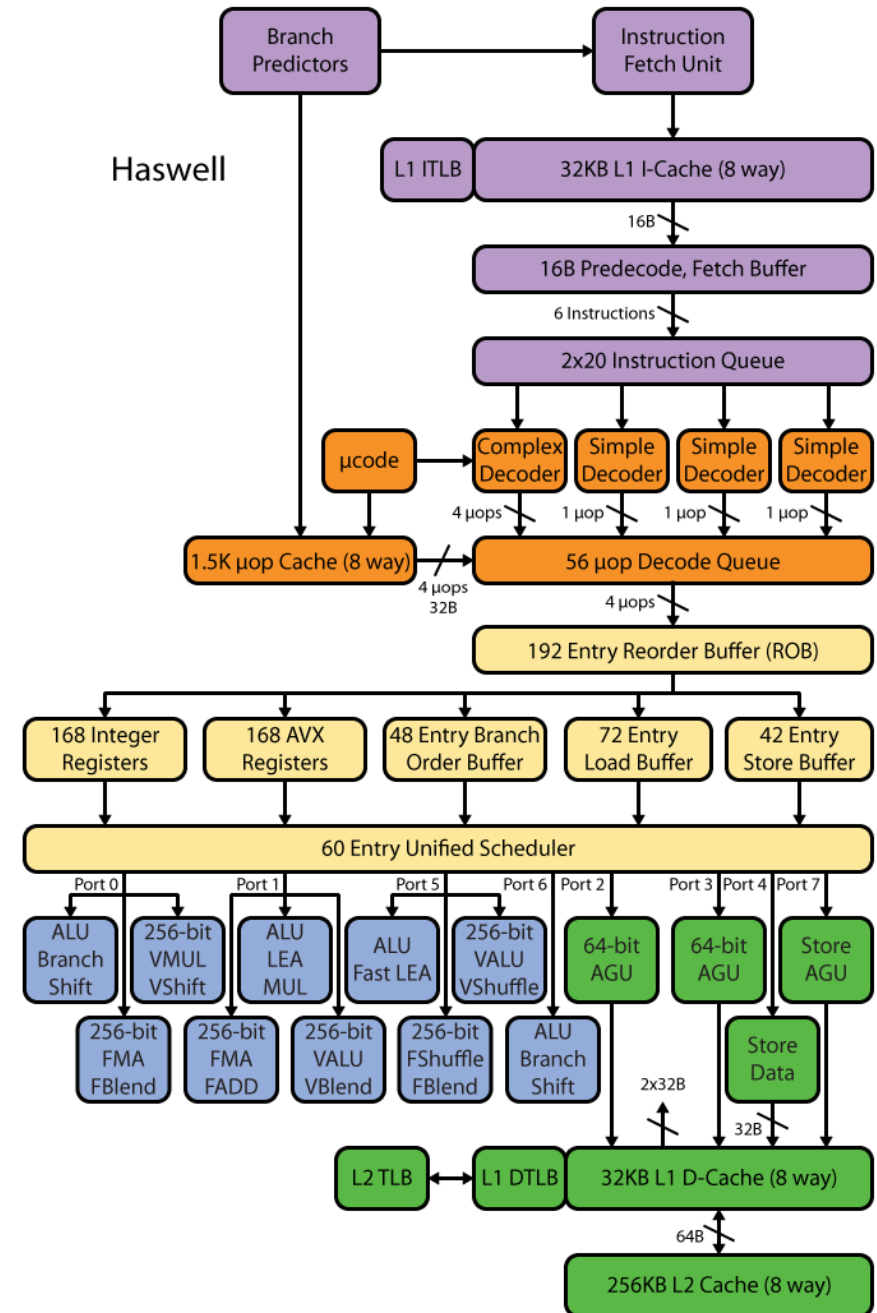
- Intel Core i7 (2009)
  - Application: desktop/server
  - Technology: 45nm (1/2x)
  - 774M transistors (12x)
  - 296 mm<sup>2</sup> (3x)
  - 3.2 GHz to 3.6 Ghz (~1x)
  - 0.7 to 1.4 Volts (~1x)
  - 128-bit data (2x)
  - 14-stage pipelined datapath (0.5x)
  - 4 instructions per cycle (~1x)
  - *Three levels of on-chip cache*
  - *data-parallel vector (SIMD) instructions, hyperthreading*
  - **Four-core multicore** (4x)



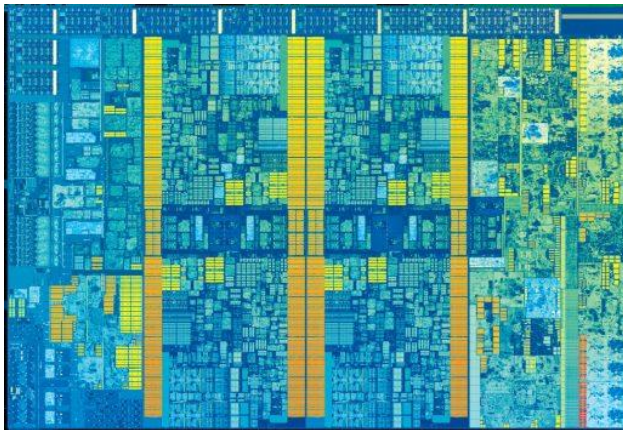
# General Purpose 10 Years Ago 2014



Intel Haswell

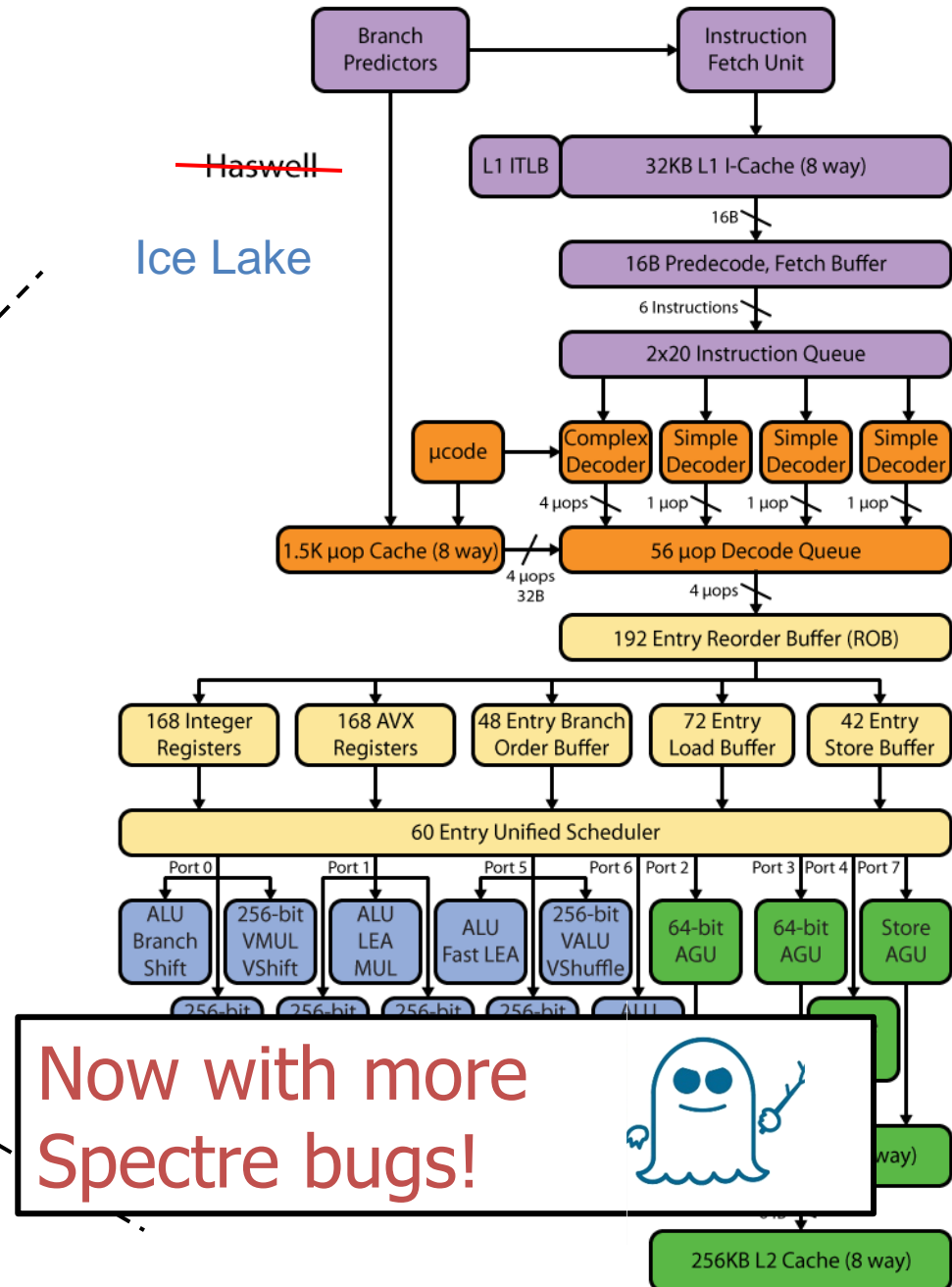


# General Purpose (2020 & Beyond)



Intel Ice Lake  
~20% Speedup

~~Haswell~~  
Ice Lake



# Revolution IV: Specialization

- Combine implicit/explicit parallelism with a focus on a particular domain
  - Scope can be very different:
    - GPGPUs are quite broad
    - TPUs are not
    - FPGAs in between
- Tradeoff the overheads of supporting “general purpose” workloads for efficiency on a smaller set of workloads.
- But why is this happening now?
  - Dark silicon – not all components of a chip can be kept active simultaneously

# Machine learning in Industry



**Google  
TPU (v4)**



**NVIDIA  
H100**



**Microsoft  
Brainwave**



**Cambricon  
MLU-100**



**GraphCore  
Colossus**

Startup	Funding (M)
GraphCore	300
Cambricon	200
Wave	200
SambaNova	150
Cerebras	112
Horizon Rob.	100 (for ml)
Habana	75
ThinCI	65
Groq	62
Mythic	55
ETA Compute	8
...	



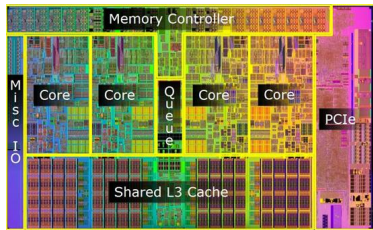
# Specialization Spectrum

General Purpose

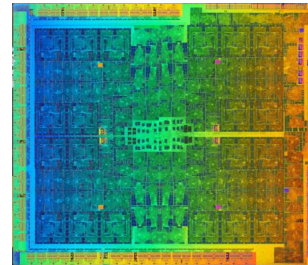
Application Specific



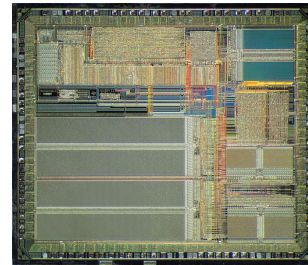
CPU  
("Ordinary" Apps)



Graphics Proc.  
Unit (GPU)



Digital Signal  
Proc. (DSP)



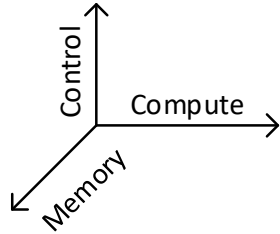
Filed Prog.  
Gate Array  
(FPGA)



Google TPU:  
(Deep Learning)



# Layers of Abstraction

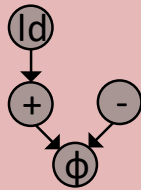


**Applications**

**Major Driver  
Going forward?**

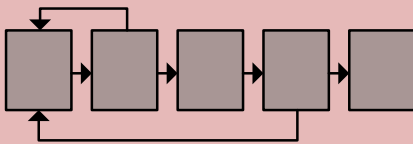


sub  
ld  
add  
br

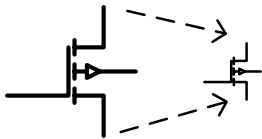


**ISA: Hardware/  
Software Interface**

**Architects'  
Domain**



**Microarchitecture**



**Technology**

**Major Driver  
for 50+ years**





# Applications as a Driver

# Applications Views

- Many ways to view distinction between application settings:
  - Deployment-centric view:
    - Where the machine is deployed affects the set of applications
  - Domain-centric view:
    - Set applications from a similar background
  - Property-centric view:
    - Set of applications with different properties

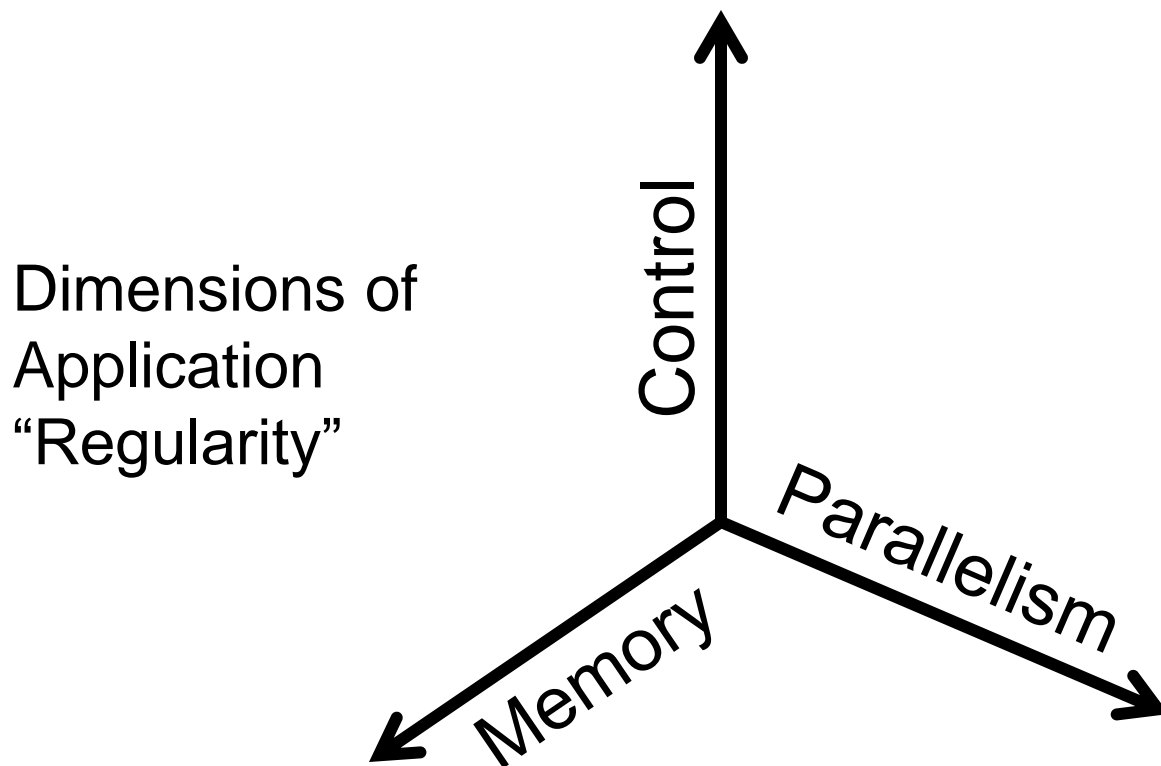
# Deployment Centric View

- **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**, heat, integer performance, integrated wireless
  - Laptops: Intel Core 2 Mobile, Atom, AMD Turion
  - Smaller devices: ARM chips by Samsung and others, Intel Atom
  - Over 1 billion ARM cores sold in 2006 (at least one per phone)
- **Embedded**: microcontrollers in automobiles, door knobs
  - Need: low power, **low cost**
  - Examples: ARM chips, dedicated digital signal processors (DSPs)
- **Deeply Embedded**: disposable “smart dust” sensors
  - Need: extremely low power, extremely low cost

# Domain-centric View

- Old Dichotomy:
- **Scientific**: weather prediction, genome sequencing
  - First computing application domain: naval ballistics firing table
  - Need: large memory, heavy-duty floating point
  - Examples: CRAY T3E, IBM BlueGene
- **Commercial**: database/web serving, e-commerce
  - Need: data movement, high memory + I/O bandwidth
  - Examples: Sun Enterprise Server, AMD Opteron, Intel Xeon, IBM Power 7
- **Recently – finer grain domains:**
  - Deep learning, Digital Signal Processing, Graphics, Genomics, Database Processing, Compression/Decompression

# Property-centric View



More regularity → Less dependences

Less dependences → Easier exploitation  
(h/w or s/w)

# Control Regularity

Increasing “Irregularity”

- No Control (or non critical)
- Data-Independent
- Data-Dependent, Predictable
- Data-Dependent, Unpredictable

(also, indirect branches)

```
for i
  ... = a[i]
```

```
for i
  if(i%2)
    ... = a[i]
```

```
for i
  if(age[i]>2)
    ... = a[i]
```

```
for i
  if(age[i]>22)
    ... = a[i]
```

# Memory Regularity

Increasing “Irregularity” 

- Data dependence

```
for i=0 to n  
  ... = a[i]
```

```
while(node)  
  ... = *node  
  node = node->next
```

- Alias freedom

```
for i=0 to n  
  ... = a[i]
```

```
for i=0 to n ... =  
  a[index[i]]++
```

- Locality

```
for i=0 to n  
  for j=0 to n  
    ... = a[j]
```

spatial & temporal

```
for i=0 to n  
  for j=0 to n  
    ... = a[i][j]
```

just spatial

```
for i=0 to n  
  for j=0 to n  
    ... = a[j][i]
```

neither

Ponder this: why does low-locality introduce dependences?

# Parallelism Regularity

- Types of Parallelism
  - Instruction-level Parallelism (ILP): Nearby instructions running together
  - Memory-level Parallelism (MLP): Same as ILP, but specifically cache misses.
  - Thread-level Parallelism (TLP): Independent threads (at least to some extent) running simultaneously.
  - Task-level Parallelism: Same as above, but implies dependences.
  - Data-level Parallelism (DLP): Do same thing to many pieces of data.
- Which is the most regular: (one perspective)
  - DLP: more regular
  - ILP: less regular
  - TLP: least regular
- Dimensions of Regularity
  - Granularity: Fine vs Coarse Grain
  - Data-dependence: Static vs Dynamic
- Complexity Ahead:
  - DLP implies ILP... (but not other way around)
  - DLP implies TLP ... (but not other way around)



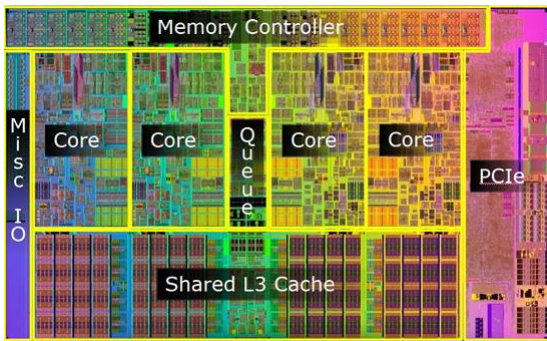
# Even lower-level properties...

- Instruction Locality
  - Temporal:
    - Do instructions repeat within some time?
    - Loopy vs function-call code
  - Spatial:
    - Branch Density vs Computation
- Datatype Regularity?
  - Integer vs Floating Point
  - Small vs Large Bitwidth
  - (Ratio of resources + conversion overheads)
- Probably many other important properties, depending on the context and architecture...

# A naïve property-based classification

## CPU

(“Ordinary” Apps)

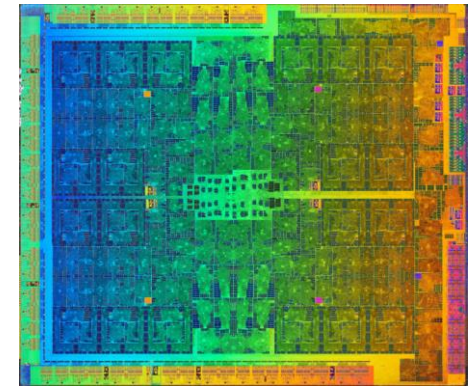


- + Irregular Control/Memory
- + Fine-grain Instruction Level Parallelism

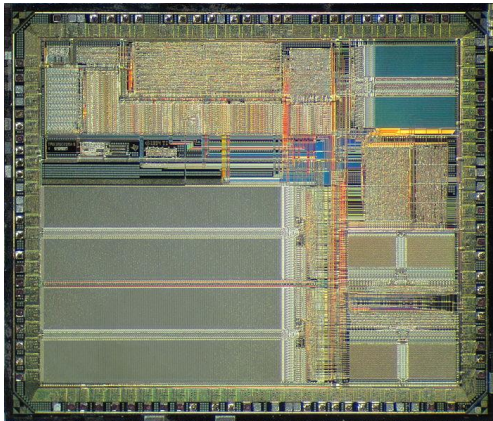
- + Thread-level and Data-level Parallelism
- + Medium Control/Memory

## GPU

(Graphics, Data-proc.)



## DSP (signal proc.)



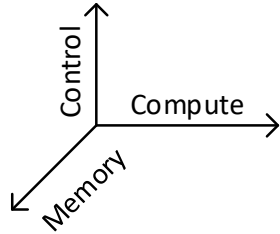
- + Fine-grain ILP
- + Highly-regular Memory

- + Extremely Regular Data Parallelism
- + Extreme Control/Memory Regularity

## Google TPU: (Deep Learning)

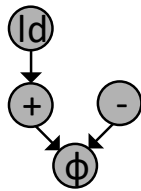


# Course Themes

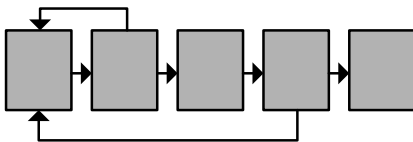


## Applications

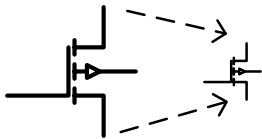
sub  
ld  
add  
br



## ISA: Hardware/ Software Interface



## Microarchitecture



## Technology

# Layers of Abstraction

- Levels:

- Applications
- Operating Systems
- Compilers
- Instruction Set Architecture
- Microarchitecture
- Circuits
- Technology



- Goal: Make our CPU better at machine learning?
- What can we do at different levels?